Testing

Introduction

A lot of the testing I did whilst designing and creating the initial parser/ interpreter. These less formal tests are not included in this document.

This document details the more formal testing I did, commencing with unit testing each of the functions and then testing as many cases as possible within the integrated design to make sure that these functions worked together.

Overall I tried to make the unit testing as comprehensive as possible in order to put less pressure on the integrated testing, and instead focussed the integrated tests mainly on edge cases, special cases and any circumstances which might confuse the parser or interpreter.

Section 1: Unit Testing

Overview: I began with unit testing everything using the CUnit testing harness. I embedded the unit tests within the body of the parser/ interpreter so that as each function was called it simultaneously had a unit test ran on it. This meant that any test file used would have to call all functions but this is fairly easy to achieve. The reason I did it this way is due to having had this technique suggested to us in Software Engineering.

Rather than run through every single unit test I did I will do a detailed run through of one and then briefly outline all of the others and include their test output tables. The code to compile the unit tests will be attached with the submission as well as a set of unit test .txt files detailing all of the outputs of each of the individual tests.

Note that the code will need CUnit to compile which is not available on the lab computers, however do look over the code to see how they were done.

Example Unit Test (Function: *Do Loop*):

The unit test used as an example is *`do_loop()`* (included in *`parser_functions.c`*). The reason I have picked this to demonstrate with is as it encompasses several different techniques used for the unit testing and serves as a good example.

The function is used when it is registered that “DO” has been entered as a command and the program needs to parse and interpret the results according to the grammar. The unit test is designed to run through all of the different possible cases that could occur when the parser registers the DO command and check that the results are cohesive with what we would expect.

```
//////////////////////////                Testing

static int first_pass = 0;

if(first_pass == 0){

    set_up_test("Do Loop Suite", "Test of do_loop()", test_do_loop);

    ++first_pass;

}
//////////////////////////
```

DO Loop test being set up.

A new suite in CUnit is set up in order to accommodate the results of the unit testing. The reason that they are all set up separately this way is as it makes it easier to see what has been tested in each function and to observe the related print messages. The first pass clause is as we only want to run a unit test the first time a function is used, it would be unnecessary to test it every time the function is called.

```
void test_do_loop(void)
{
   int i, j, k;
   program prog, *prog_point;
   do_loop_test_indicators indicator_struct;

   prog_point = &prog;

   for(i = 0; i <= 1; ++i){
      for(j = 0; j <= 1; ++j){
         for(k = 0; k <= 1; ++k){

            initialise_do_loop_indicator_components(&indicator_struct, i, j, k);

            initialise_do_loop_components(prog_point, indicator_struct);

            do_loop_case_checker(prog_point, &indicator_struct);
         }
      }
   }

   printf("\n");
}
```

Inside test_do_loop

The DO loop testing suite is then run in CUnit to check that the contents of the *do_loop()* function do what we would expect of them under any situation.

The loop shown is a way of running through all of the different possible ways of a DO loop being correctly or incorrectly read by the parser. The *prog_point* variable at this point serves as what would be the program in the integrated design. It is initialised with all different possibilities for the DO loop before being passed into the *do_loop()* function to check its reaction.

```
void initialise_do_loop_indicator_components(do_loop_test_indicators *indicator_struct, on_off FROM_ind, on_off TO_ind, on_off bracket_ind)
{
   indicator_struct -> FROM_correct = FROM_ind;
   indicator_struct -> TO_correct = TO_ind;
   indicator_struct -> bracket_correct = bracket_ind;

   indicator_struct -> FROM_wrong = 1 - FROM_ind;
   indicator_struct -> TO_wrong = 1 - TO_ind;
   indicator_struct -> bracket_wrong = 1 - bracket_ind;
}

void initialise_do_loop_components(program *prog_point, do_loop_test_indicators indicator_struct)
{
   printf("\n");

   randomly_assign_point(prog_point);

   initialise_single_do_component(indicator_struct.FROM_correct, "FROM", prog_point, "\nCurrent string successfully set to FROM.\n");

   check_program_incremented(prog_point);

   initialise_single_do_component(indicator_struct.TO_correct, "TO", prog_point, "\nCurrent string successfully set to TO.\n");

   check_program_incremented(prog_point);

   initialise_single_do_component(indicator_struct.bracket_correct, "{", prog_point, "\nCurrent string successfully set to {.\n");

   prog_point -> current_word -= NUM_INSTRUCTIONS_DEFINED;

}

void initialise_single_do_component(on_off indicator, char *word, program *prog_point, char *assign_message)
{
   if(indicator == on){
      assign_word_to_point(word, prog_point, assign_message);
   }
   else{
      assign_word_to_point("!", prog_point, "\nCurrent string successfully set to !.\n");
   }
}
```

Initialising the different components of a DO loop.

In this particular example the i, j and k components act as indicators as to whether or not the loop should have the FROM, TO and { terms in the DO loop correctly initialised and are used to run through all possible combinations. These are initialised accordingly at a randomly set point in our pseudo-program to imitate a genuine user input and are then sent to the *do_loop()* function contained in our parser/ interpreter.

```
void do_loop_case_checker(program *prog_point, do_loop_test_indicators *indicator_struct)
{
   int start_of_loop, end_of_loop;

   if( same_string(prog_point -> prog_line[prog_point -> current_word], "FROM") ){

      do_loop_section_check(indicator_struct -> FROM_correct, "\nCorrectly forwarded through FROM part of do_loop\n",
                            "\nIncorrectly forwarded through FROM part of do_loop\n", prog_point);

      start_of_loop = rand() % MAX_LOOP_SIZE;

      if( same_string(prog_point -> prog_line[prog_point -> current_word], "TO") ){

         do_loop_section_check(indicator_struct -> TO_correct, "\nCorrectly forwarded through TO part of do_loop\n",
                               "\nIncorrectly forwarded through TO part of do_loop\n", prog_point);

            do{
               end_of_loop = rand() % MAX_LOOP_SIZE;
            }while(end_of_loop < start_of_loop);

            if( same_string( prog_point -> prog_line[prog_point -> current_word], "{" ) ){

               do_loop_section_check(indicator_struct -> bracket_correct, "\nCorrectly forwarded through { part of do_loop\n",
                                     "\nIncorrectly forwarded through { part of do_loop\n", prog_point);

               looping_component_check(start_of_loop, end_of_loop);

            }
            else{

               do_loop_section_check(indicator_struct -> bracket_wrong, "\nIncorrect { statement caught.\n",
                                     "\nIncorrect { statement not caught.\n", prog_point);

            }

      }
      else{

         do_loop_section_check(indicator_struct -> TO_wrong, "\nIncorrect TO statement caught.\n",
                               "\nIncorrect TO statement not caught.\n", prog_point);

      }

   }
   else{

      do_loop_section_check(indicator_struct -> FROM_wrong, "\nIncorrect FROM statement caught.\n",
                            "\nIncorrect FROM statement not caught.\n", prog_point);

   }
}
```

Asserts after all lines to test the program is behaving correctly.

Above we have the unit test version of the *do_loop()* function shown earlier. They are identical except that after every line the unit test version contains a function in order to assert that the program is in the state we would expect it to be. The *do_loop_section_check()* functions above are simply wrappers for the assert statements in order for messages to be printed to the screen detailing which tests were undergone.

After the tests have completed the results are available via the terminal. I ran one script unit testing every single function and directed it to a unit test .txt file which you can flick through to see the unit test results, or you can compile and run the code with the makefile provided.

An example output would be:

```
Current place in instructions appropriately randomly assigned.
Current string successfully set to FROM.
Word counter successfully incremented by one.
Current string successfully set to TO.
Word counter successfully incremented by one.
Current string successfully set to {.
Correctly forwarded through FROM part of do_loop
Word counter successfully incremented by one.
Correctly forwarded through TO part of do_loop
Word counter successfully incremented by one.
Correctly forwarded through { part of do_loop
Word counter successfully incremented by one.
DO loop initiated.
Currently on iteration 1/6 of the DO loop.
Currently on iteration 2/6 of the DO loop.
Currently on iteration 3/6 of the DO loop.
Currently on iteration 4/6 of the DO loop.
Currently on iteration 5/6 of the DO loop.
Currently on iteration 6/6 of the DO loop.
DO loop finished.
```

Example Unit Test Result

Here we see that the correct parts of the loop are all assigned to the program and are then successfully read by the function itself. A pseudo DO loop is entered at the end to simulate what would happen in a genuine program and a finish statement is printed.

Finally after all of the different combinations have been tested a table summarising the results of the unit test is put to screen.

```
passed

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts     82     82     82      0      n/a

Elapsed time =    0.001 seconds
```

Example output table from a unit test.

We see that in every single case that, as a standalone unit, the *do_Loop()* function works as we would expect it to. All of the unit tests work along similar lines to this:

- **Initialisation Stage:** The components of the unit are initialised in different combinations. Sometimes this does not appear exhaustive in the code but in reality previous functions that have taken place before the unit under consideration will have prevented certain combinations of inputs taking place. If this occurs it will be explained in the code.

- **Function Stage:** The parameters representing the different possible inputs for the function are entered and the function is run. The code will for the most part be identical to the integrated versions, however sometimes it will be modified slightly to allow for a standalone property.

  A lot of the functions at this point boil down to flow control, making sure the program or data is passed along to the correct next function and so a lot of the code disappears or simplifies down.

Final Point about Unit Testing:

- In the larger scale of testing I wanted to exhaustively test as much as I could in the unit tests as it would be harder to do so within the integrated testing. The integrated testing is based around how the separate modules interact and how what is assigned or actioned on at one place translates into another module. The unit testing is more focussed around special cases and possible places the interpreter/ parser may trip up.

Function by Function Unit Tests

Below is a quick summary of the functions unit tested, a description of the function, unit test and the results. For a more verbose break down such as that seen in the do_loop() example look in the unit tests .txt files.

Function: *clear_screen()*
Description: Simulates the screen having cleared successfully and unsuccessfully.

Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      2      2      2      0      n/a

Elapsed time =    0.000 seconds
```

Function: *initialise_program()*
Description: Asserts that all elements of the program have been successfully initialised.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts    102    102    102      0      n/a

Elapsed time =    0.000 seconds
```

Function: *initialise_player()*
Description: Asserts all elements of the player structure have been successfully initialised.
Output Testing Table:

```
Run Summary:    Type   Total     Ran  Passed Failed Inactive
              suites       1       1     n/a      0        0
               tests       1       1       1      0        0
             asserts 1000540 1000540 1000540      0      n/a

Elapsed time =    0.005 seconds
```

Function: *open_program_file()*
Description: Simulates opening existent and non-existent program files.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts     37     37     37      0      n/a

Elapsed time =    0.000 seconds
```

Function: *parse_text_and_interpret()*
Description: Simulates a valid and invalid start of a program file.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      4      4      4      0      n/a

Elapsed time =    0.000 seconds
```

Function: *same_string()*
Description: Asserts that the same string function correctly acknowledges identical strings.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      2      2      2      0      n/a

Elapsed time =    0.000 seconds
```

Function: *instrctlist()*
Description: Simulates being inside the instrctlist part of the grammar.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      6      6      6      0      n/a
```

Function: *instruction()*
Description: Looks for all possible instructions that could be entered and checks them.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      6      6      6      0      n/a

Elapsed time =    0.000 seconds
```

Function: *fd()*
Description: Simulates valid and invalid entries for function for when parser looks for FD.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      6      6      6      0      n/a

Elapsed time =    0.000 seconds
```

Function: *lt_rt()*
Description: Simulates valid and invalid entries for function for when parser looks for LT/RT.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts     12     12     12      0      n/a

Elapsed time =    0.000 seconds
```

Function: *do_check()*
Description: Simulates valid and invalid entries for function for when parser looks for DO.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      6      6      6      0      n/a

Elapsed time =    0.000 seconds
```

Function: *do_loop()*
Description: Examines behaviour of parser when looking for DO function components.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts     82     82     82      0      n/a

Elapsed time =    0.000 seconds
```

Function: *set_check()*
Description: Simulates valid and invalid entries for function for when parser looks for SET.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      6      6      6      0      n/a

Elapsed time =    0.000 seconds
```

Function: *set_loop()*
Description: Tests what occurs in the program once we look to set a variable.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      6      6      6      0      n/a

Elapsed time =    0.000 seconds
```

Function: *initialise_pol_stack()*
Description: Initialises and asserts all the components of the polish stack.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts    101    101    101      0      n/a

Elapsed time =    0.000 seconds
```

Function: *polish()*
Description: Goes through all the possible inputs to the polish part of the grammar.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      7      7      7      0      n/a

Elapsed time =    0.000 seconds
```

Function: *op()*
Description: Tests the detection of operations.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts     19     19     19      0      n/a
```

Function: *varnum()*
Description: Looks and checks for variables and numbers in a command.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts     12     12     12      0      n/a

Elapsed time =    0.000 seconds
```

Function: *var()*
Description: Simulates identifying a variable in a command.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      6      6      6      0      n/a

Elapsed time =    0.000 seconds
```

Function: *fd_interpret()*
Description: Assumes FD registers and simulates the interpretation of this.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      3      3      3      0      n/a

Elapsed time =    0.000 seconds
```

Function: *check_variable()*
Description: Tests to confirm unassigned and assigned variables are registered.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      2      2      2      0      n/a
```

Function: *draw_angled_line()*
Description: Sets up the initial components of drawing lines for a variety of cases.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts     75     75     75      0      n/a
```

Function: *draw_next_square()*
Description: Assigns next drawing square for a selection of possible cases.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts     19     19     19      0      n/a

Elapsed time =    0.000 seconds
```

Function: *drawing_process()*
Description: Actually colours the squares given a range of test inputs.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      9      9      9      0      n/a

Elapsed time =    0.000 seconds
```

Function: *lt_rt_interpret()*
Description: Simulates the interpreting of a command given LT or RT detected.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts     12     12     12      0      n/a

Elapsed time =    0.000 seconds
```

Function: *adjust_angle()*
Description: Simulates adjusting the angle for a variety of inputs.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      3      3      3      0      n/a

Elapsed time =    0.000 seconds
```

Function: *op_interpret()*
Description: Tests how different operands are dealt with within the RPN functions.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts     26     26     26      0      n/a

Elapsed time =    0.000 seconds
```

Function: *polish_varnum_interpret()*
Description: Tests how variables or numbers are dealt with within the RPN functions.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      3      3      3      0      n/a

Elapsed time =    0.000 seconds
```

Function: *push_to_stack()*
Description: Short function to test variables are pushed to the stack correctly.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      1      1      1      0      n/a

Elapsed time =    0.000 seconds
```

Function: *interpret_set_variable()*
Description: Simulates SET being detected and how the function deals with that.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts     26     26     26      0      n/a

Elapsed time =    0.000 seconds
```

Function: *check_and_assign_variable()*
Description: Used in SET to assign the end variable to the result.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      5      5      5      0      n/a

Elapsed time =    0.000 seconds
```

Function: *loop_subject_var_interpret()*
Description: Test that in the DO loop a variety of different subject variables can be detected.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      4      4      4      0      n/a

Elapsed time =    0.000 seconds
```

Function: *loop_start_var_interpret()*
Description: Test that in the DO loop a variety of different start values can be detected.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      6      6      6      0      n/a

Elapsed time =    0.000 seconds
```

Function: *loop_end_var_interpret()*
Description: Test that in the DO loop a variety of different end values can be detected.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      6      6      6      0      n/a

Elapsed time =    0.000 seconds
```

Function: *check_loop_finish()*
Description: Simulates the function for checking if the DO loop is finished after each iteration.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      2      2      2      0      n/a

Elapsed time =    0.000 seconds
```

Note on Graphics Test: I also ran graphics tests to make sure that everything that should have been drawn to screen was put where it should have been. These have been commented out of the unit test code as it takes a while to complete. It works by highlighting the grid pixel by pixel and then printing that to screen, checking that where should be highlighted has been.

Section 2: Integrated Testing

Overview: The way I approached the integrated testing section was to annotate and print comments at the points where the integrated program was taking an action of any sort in order to check that this matched what I imagined was going to happen.

As the unit testing attempted to be so thorough it felt less crucial to exhaustively try every possible movement that could be made within the integrated design. Instead I focussed on making sure the basics were functioning correctly, the designs given in the instruction sheet functioned as expected and that any ‘unusual’ or ‘edge’ cases of functions were handled by the parser and interpreter.

The outputs of the first few programs with print statements integrated are available as the specified .txt files for each of the tests. In order to save space in this documentation I have just given a brief overview of the test and the results for each one I ran. The full breakdown is available in the .txt files.

I go through the first few tests in a lot of detail to give a feel for how I carried out each of them and then after a few only discuss each test briefly and don’t include the outputted .txt files with my submission. However you can compile them yourself with the annotated version of the parser to view what they do if need be.

Although I did look for test harnesses to incorporate, CUnit was unsuitable as the test functions cannot have inputs. I did use ValGrind to check for memory leaks and gdb for debugging, but only contain the results of the former here. I instead replaced the test harnesses with print statements. Due to the nature of the testing I thought this might be the most productive way of doing it.

Test 1: Basic Test

Test that it can deal with:

- Moving forwards.
- Turning by any amount, specifically those on the borders of the special cases.

Basic Test Output Image

Text File (integration_one_test_program_file.txt) :

```
{
        FD 30
        LT 90
        FD 30
        RT 90
        FD 10
        LT 180
        FD 10
        RT 180
        FD 20
        LT 270
        FD 20
        RT 270
        FD 40
        LT 360
        FD 40
        RT 360
        FD 50
        LT -40
        FD 50
        RT -40
}
```

Notes: This in general showed that the parser worked well and correctly interpreted the basic results. However there were problems when the angle went below or above 360 degrees, this was later rectified by putting in a modulus expression so that the angle was always kept within these boundaries.

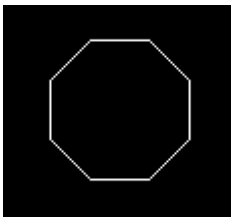
A negative forward movement also caused problems with the interpreter and so a function to check all movement forwards was positive and so did not cause conflict with the trigonometry functions was added.

Output from Annotated File: *integration_one_test.txt*

Test 2: Loop to Draw Octagon:

This was taken from the instructions sheet and tests the basic components of loops.

Image:



Loop to Draw Octagon Output Image

Text File (*integration_two_test_program_file.txt)*:

```
{
	DO A FROM 1 TO 8 {
		FD 30
		LT 45
	}
}
```

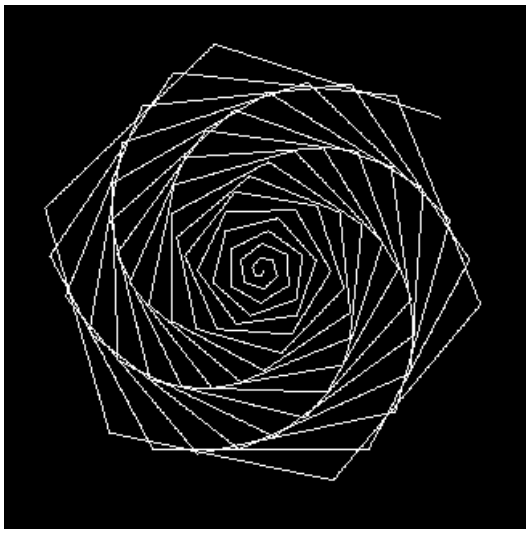
Notes: This worked very well, it cycled through all of the components of the loops the correct amount of times and exited when it needed to. The angle was correctly added to each time and the output was what was expected.

Output from Annotated File: *integration_two_test.txt*

Test 3: Using Variables:

Again this was taken from the instructions to make sure that setting variables worked as desired.

Image:



Using Variables Output

Text File (*integration_three_test_program_file.txt)*:

```
{
	DO A FROM 1 TO 100 {
		SET C := A 1.5 * ;
		FD C
		RT 62
	}
}
```

Notes: Again, this seemed to work as it should. The expected results were pushed to the Polish stack at the correct times, the loop was exited after the correct number of iterations and the angle was consistent throughout. In addition to this the value of C was updated regularly and as it should have been.

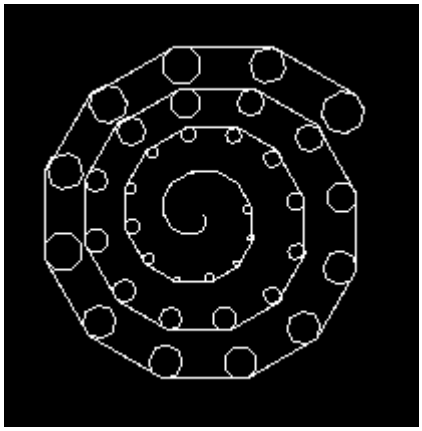
From the above two tests I assumed basic loops to be working fine as well assigning variables.

Output from Annotated File: *integration_three_test.txt*

Test 4: Nested Loops

This is the final example I have taken from the lecture notes and the last one I will go through in great detail with the image, description of the .txt file etc. From here onwards I will just give a quick description of the test and where to find the annotated output file and test file.

Image:



Output from Nested Loop

Text File (*integration_four_test_program_file.txt)*:

```
{
	DO A FROM 1 TO 50 {
		FD A
		RT 30
		DO B FROM 1 TO 8 {
			SET C := A 5 / ;
			FD C
			RT 45
		}
	}
}
```

Notes: The function dropped in and out of the loops as expected which was promising and printed the output we thought that it would. In conjunction with the previous test this gave me confidence in the basic nested loops working as we might expect them to and that variables were also being registered correctly.

Output from Annotated File: *integration_four_test.txt*

Test 5: Maximum number of commands exceeded *(commands_exceeded.txt).*

Notes: Correctly caught, error message printed.

Test 6: Wrong number of variables at command line *(no relevant .txt file).*

Notes: Correctly caught for both too many and too few arguments

Test 6: Invalid filename at command line *(no relevant .txt file).*

Notes: Correctly caught.

Test 7: Adding on angles over 360° *(adding_angles_over_360.txt)*.

Notes: This actually caught out that if you RT by more than 360 degrees then you have a negative angle which will still be negative if you add on 360 again. Therefore there is the loop to continuously add on 360 until you get back into positive territory.

Test 8: Loops defined within each other with same variable *(loop_same_variable_inside.txt).*

Notes: Originally thought this might fail but works as the next stage of the loop is determined relative to the current iteration and not to the previous value of the variable.

Test 9: Operations being defined first on the Polish stack *(operator_defined_first.txt).*

Notes: Correctly caught with error message printed.

Test 10: Too many/ few operations on the Polish stack *(too_many_operators.txt/ too_few_operators.txt).*

Notes: Too few operations gets passed through to look for an instruction which fails and prints an error message. Too many treated identically.

Test 11: Loop starts on a value that is not one *(loop_not_begin_at_one.txt)*.

Notes: Works as expected.

Test 12: Incorrect braces *(DO_no_end_brace.txt, DO_no_opening_bracket.txt, no_starting_brace.txt, no_ending_brace.txt).*

Notes: DO Loop: Start and end caught, start prints it is looking for starting bracket, end prints looking for instruction. Program Braces: Start and end caught, start prints looking for start bracket, end prints looking for instruction.

Test 13: Maximum number of embedded loops exceeded *(max_embedded_loops.txt).*

Notes: This was easiest to test by changing the MAX_EMBEDDED_LOOPS down rather than writing out 100 loops inside each other. Prints an error if you exceed the maximum number of embedded loops otherwise continues as normal, although obviously will be very slow if you embed a lot.

Test 14: Design strays outside of defined area *(leave_defined_area.txt).*

Notes: Successfully caught and error message printed.

Test 15: Assigning variables to themselves.

Notes: Here uncovered a deeper problem in how the polish stack deals with multiple operations on the same line in the wrong way, instead of assigning the last variable in the stack it assigns the second last one. Fixed this.

Aside from that works.

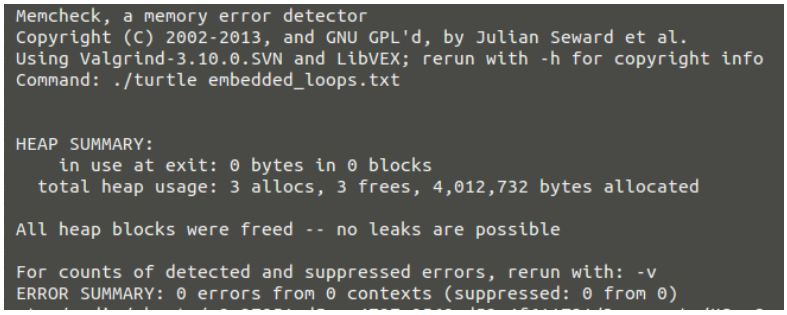
Test 16: Unrecognised variable *(unrecognised_variable.txt)*.

Notes: Will give the warning corresponding to the next command it is looking for.

Test 17: Valgrind.

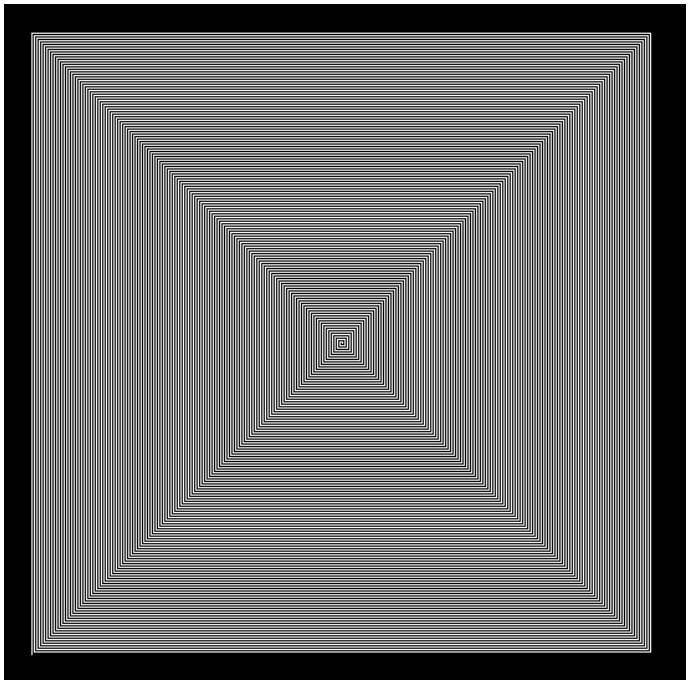
Notes: Only memory lost was in the SDL functions.

Originally ValGrind pointed to some invalid read/ write signals but I fixed this by mallocing space for pointers rather than assigning the variables in the stack.

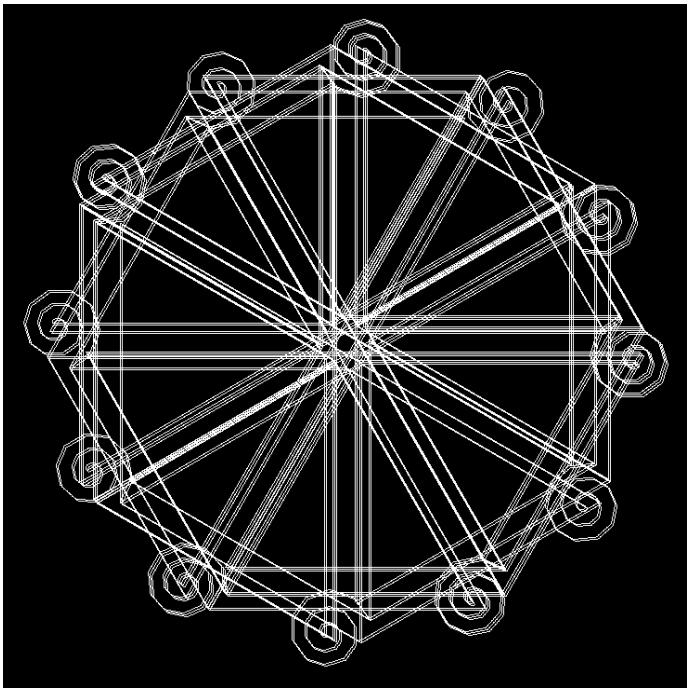


ValGrind output for basic parser/interpreter.

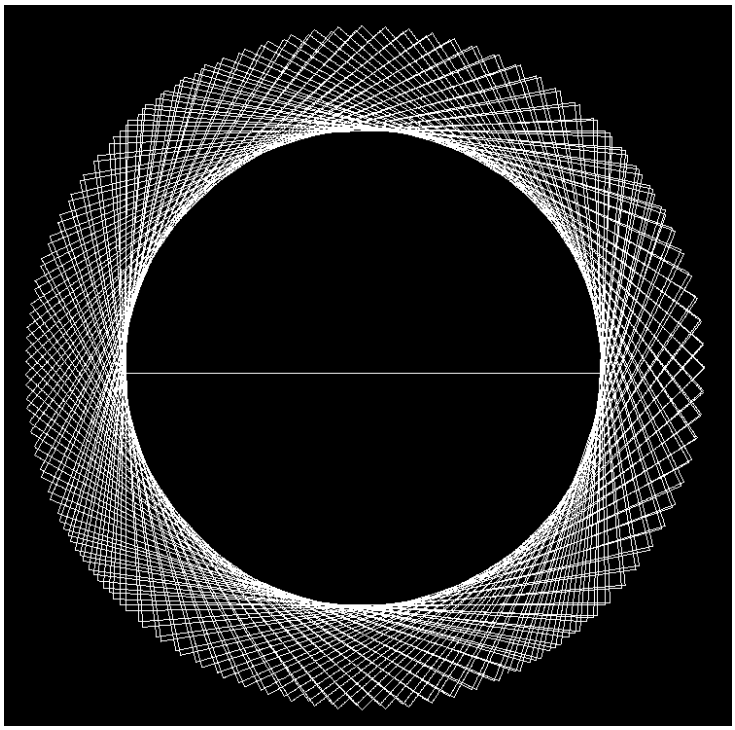
Miscellaneous Shapes: I included a series of shapes I drew using my basic parser and interpreter along with the name of the .txt file used to draw them.



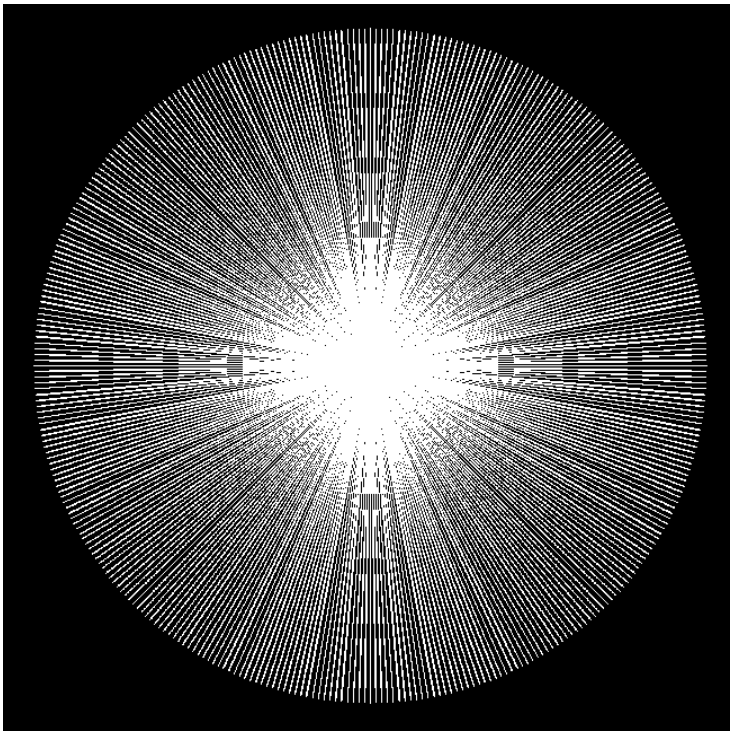
Tight Spiral Output (spiral.txt)



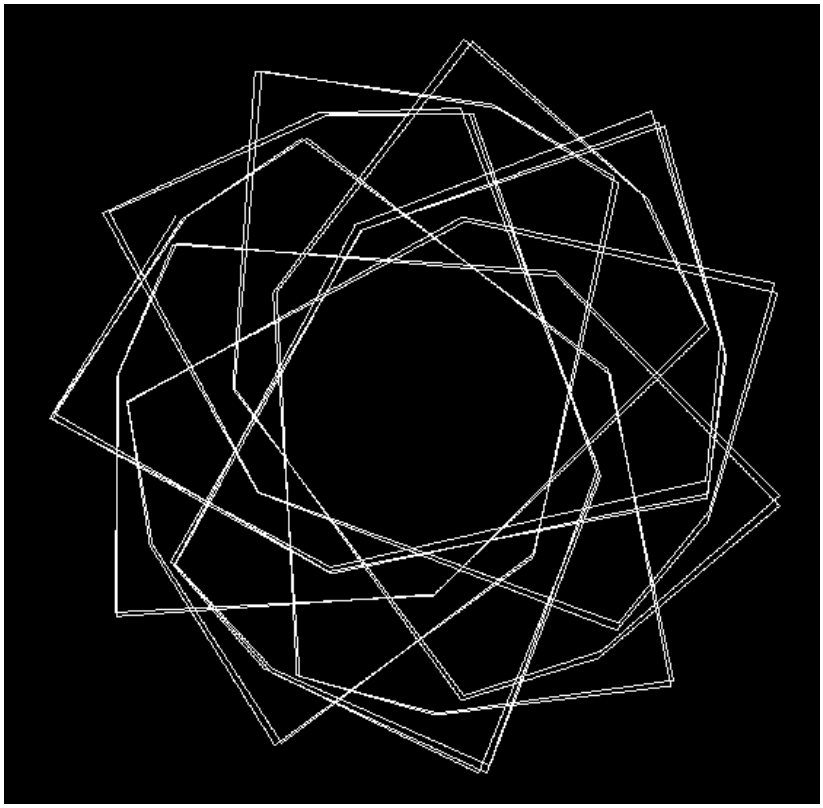
More embedded loops (embedded_loops.txt)



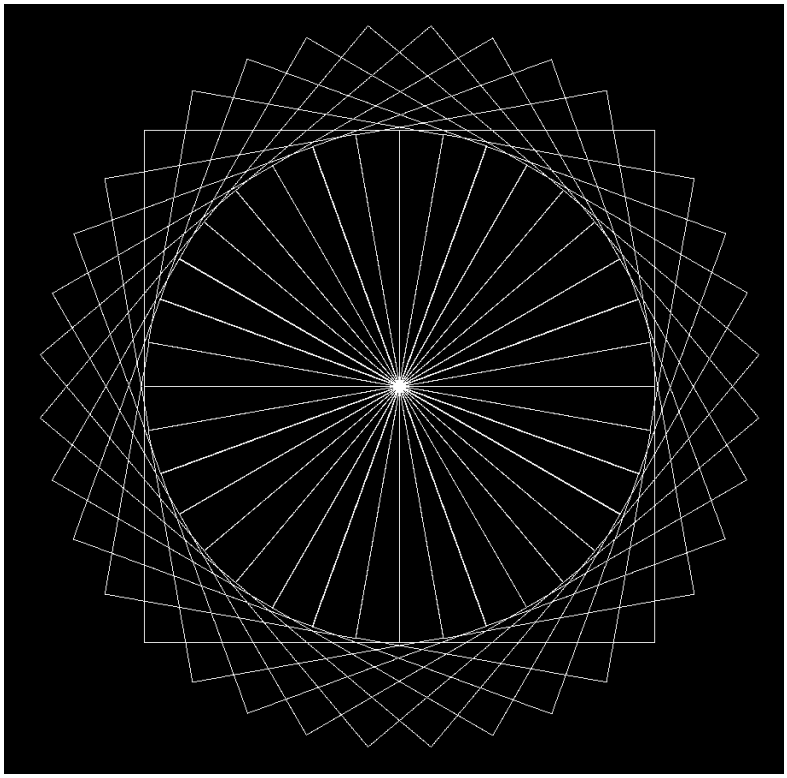
Pseudo Torus (pseudo_torus.txt)



Multiple Embedded Loops (multi_embedded_loops.txt)



Cog (cog.txt)



Circle Square (circle_square.txt)

Section 3: Regression Testing

Every single one of the above tests was used in a regressive framework. Once one had been tested, if anything had been fixed then they were all re-tested and made sure to be still working.

Section 4: Extension Testing

The extension testing was done in the exact same way as above except with the additional tests for the new functions. A very few functions were not unit tested as the unit testing mainly handled flow control and in that respect they were very similar or were just wrappers for collections of other functions.

A lot of the functions that were created during the extension relied entirely on reading in files. Their testing was not done through CUnit but by giving them different files to read in and observing the outcome. Refer to the later testing sections for details.

Similar to the testing for the non-extension I will print out the results tables for each of the unit tests and then move on to discuss the integration testing.

Function by Function Unit Tests

Function: *check_initialisation()*
Description: Used to make sure that all mallocs at the start of the program have been assigned, simulates all possibilities.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      6      6      6      0      n/a

Elapsed time =    0.000 seconds
```

Function: *read_in_pw_array()*
Description: Simulates reading in the relevant file for decoding the password.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      3      3      3      0      n/a

Elapsed time =    0.000 seconds
```

Function: *map_arrays()*
Description: Used to map one password file to another according to the user's code.

Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts   1000   1000   1000      0      n/a

Elapsed time =    0.000 seconds
```

Function: *compare_arrays()*
Description: Used to compare the encoded final array and decoded array according to the user.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      2      2      2      0      n/a

Elapsed time =    0.000 seconds
```

Function: *examine_fourth_argument()*
Description: This is used to check the last argument has correctly been entered as ‘COMBINE’
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      3      3      3      0      n/a

Elapsed time =    0.000 seconds
```

Function: *initialise_whitespace_prog()*
Description: Runs through the initialisation of all the WhiteSpace program elements.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts    442    442    442      0      n/a

Elapsed time =    0.000 seconds
```

Function: *scan_in_whitespace_components_verbatim()*
Description: Used for scanning WhiteSpace elements directly into the program.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts     10     10     10      0      n/a

Elapsed time =    0.000 seconds
```

Function: *write_combination_to_file()*
Description: Used to simulate writing combined files of different sizes together.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts     79     79     79      0      n/a

Elapsed time =    0.000 seconds
```

Function: *write_prog_to_whitespace_file()*
Description: Function used to write programs from regular text into WhiteSpace.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts   8200   8200   8200      0      n/a

Elapsed time =    0.148 seconds
```

Function: *free_all_print_boards()*
Description: Used to free all boards at the end of the program.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      5      5      5      0      n/a

Elapsed time =    0.000 seconds
```

Function: *check_animation_preferences()*
Description: This is so the user can set their own animation speed depending on the capabilities of their CPU.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      4      4      4      0      n/a

Elapsed time =    0.000 seconds
```

Function: *open_whitespace_program_file()*
Description: Used for opening a file that should contain some WhiteSpace code.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      2      2      2      0      n/a

Elapsed time =    0.000 seconds
```

Function: *strip_non_whitespace_components()*
Description: Simulates opening files for reading/ writing WhiteSpace from/to.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      3      3      3      0      n/a

Elapsed time =    0.000 seconds
```

Function: *scan_in_whitespace_components()*
Description: Used for taking WhiteSpace from a file.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts     14     14     14      0      n/a

Elapsed time =    0.000 seconds
```

Function: *initialise_decoder_array()*
Description: Used to copy in the decoder array for writing WhiteSpace files out into a structure.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts     41     41     41      0      n/a

Elapsed time =    0.000 seconds
```

Function: *write_file_from_whitespace()*
Description: Opens files for writing to regular from WhiteSpace.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      2      2      2      0      n/a

Elapsed time =    0.000 seconds
```

Function: *write_whitespace_prog_to_file()*
Description: Writes a program that has been translated into WhiteSpace into a file.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts    169    169    169      0      n/a

Elapsed time =    0.000 seconds
```

Function: *assign_board_to_list()*
Description: Assigns a board into a linked list to be printed.
Output Testing Table:

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      5      5      5      0      n/a

Elapsed time =    0.000 seconds
```

Function: *copy_board_into_list()*
Description: Copies the current user board into the board in the linked list.
Output Testing Table:

```
Run Summary:    Type   Total     Ran  Passed Failed Inactive
              suites       1       1     n/a      0        0
               tests       1       1       1      0        0
             asserts 1000000 1000000 1000000      0      n/a

Elapsed time =    0.005 seconds
```

Comment on Extension Unit Testing: This did not unearth any huge errors, but did identify a couple of problems such as not closing a file in the *strip_non_whitespace_components()* function and not having limits on the integers you could enter when changing your password.

Extension Integration Testing

Having tested all the functions it was sensible to test, it was time to move onto the integration testing for the extension. Again the methodology was the same: focus the unit tests on exhaustively testing the program and focus integration testing on edge and unusual cases. I drew up the following list of integration tests:

Test 1: Translate to WhiteSpace and back to regular.

Notes: To carry out these tests yourself do:

- ./turtle *filename* TRANSLATE to get a WhiteSpace file that can be run with the command ./turtle reg_to_ws_translation.txt SECRET.
- ./turtle ws_to_reg_translation.txt is the translation from WhiteSpace back to regular language.

File	Run as WhiteSpace Result.	Translation Result
circle_square.txt	Works	Works
cog.txt	Works	Works
multi_embedded_loops.txt	Works	Works
pseudo_torus.txt	Works	Works
embedded_loops.txt	Works	Works
spiral.txt	Works	Works
integration_four_test_program_file.txt	Works	Works
integration_three_test_program_file.txt	Works	Works
integration_two_test_program_file.txt	Works	Works
integration_one_test_program_file.txt	Works	Works

Test 2: Enter non-WhiteSpace file as WhiteSpace and vise versa.

Notes: Errors according to the appropriate message of the language you are trying to interpret in.

Test 3: Check password is registering as correct amount.

Notes: Checked with a variety of values and printed the value registered by the program, matches every time.

Test 4: Enter invalid amounts for a new password.

Notes: Checked with a variety of values, errors every time. Discussed more in a further section.

Test 5: Check command line arguments are registered correctly.

Notes:

Command	Correctly Input	Incorrectly Input
None	Registered.	-
TRANSLATE	Registered.	Registered.
COMBINE	Registered.	Registered.
SECRET	Registered.	Registered.

Test 6: Combine all files in all different combinations of WhiteSpace and regular and check working.

Notes:

File 1 (to be in WhiteSpace).	File 2 (to be in Regular).	WhiteSpace interpretation.	Regular Interpretation.
circle_square.txt	cog.txt	Works	Works
	multi_embedded_loops.txt	Works *	Works
	pseudo_torus.txt	Works	Works
	embedded_loops.txt	Works	Works
	spiral.txt	Works **	Works
	integration_four_test_program_file.txt	Works	Works

	integration_three_test_program_file.txt	Works	Works
	integration_two_test_program_file.txt	Works	Works
	integration_one_test_program_file.txt	Works	Works
cog.txt	multi_embedded_loops.txt	Works	Works
	pseudo_torus.txt	Works	Works
	embedded_loops.txt	Works	Works
	spiral.txt	Works	Works
	integration_four_test_program_file.txt	Works	Works
	integration_three_test_program_file.txt	Works	Works
	integration_two_test_program_file.txt	Works	Works
	integration_one_test_program_file.txt	Works	Works
multi_embedded_loops.txt	pseudo_torus.txt	Works	Works
	embedded_loops.txt	Works	Works
	spiral.txt	Works	Works
	integration_four_test_program_file.txt	Works	Works
	integration_three_test_program_file.txt	Works	Works
	integration_two_test_program_file.txt	Works	Works
	integration_one_test_program_file.txt	Works	Works
pseudo_torus.txt	embedded_loops.txt	Works	Works
	spiral.txt	Works	Works

	integration_four_test_program_file.txt	Works	Works
	integration_three_test_program_file.txt	Works	Works
	integration_two_test_program_file.txt	Works	Works
	integration_one_test_program_file.txt	Works	Works
embedded_loops.txt	spiral.txt	Works	Works
	integration_four_test_program_file.txt	Works	Works
	integration_three_test_program_file.txt	Works	Works
	integration_two_test_program_file.txt	Works	Works
	integration_one_test_program_file.txt	Works	Works
spiral.txt	integration_four_test_program_file.txt	Works	Works
	integration_three_test_program_file.txt	Works	Works
	integration_two_test_program_file.txt	Works	Works
	integration_one_test_program_file.txt	Works	Works
integration_four_test_program_file.txt	integration_three_test_program_file.txt	Works	Works
	integration_two_test_program_file.txt	Works	Works
	integration_one_test_program_file.txt	Works	Works
integration_three_test_program_file.txt	integration_two_test_program_file.txt	Works	Works
	integration_one_test_program_file.txt	Works	Works

integration_two_test_program_file.txt	integration_one_test_program_file.txt	Works	Works

*Changed the code here so that instead of printing out new lines between every part of the regular code it spaced them regularly.

**Here noticed that wasn't closing one of the files in the code combination functions.

Test 7: Translate all files into WhiteSpace and check.

Notes: Covered in test 1.

Test 8: Enter over maximum number of WhiteSpace tokens (*max_whitespace_tokens.txt)*.

Notes: Printed correct error.

Test 9: Enter nonsense code in WhiteSpace (*nonsense_whitespace.txt)*.

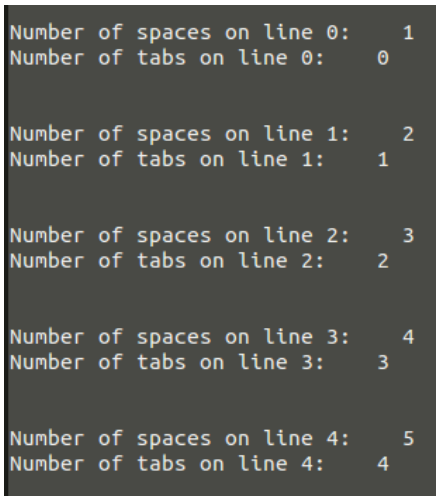
Notes: Printed correct error.

Test 10: Combine nonsense WhiteSpace code with regular code (*use nonsense with COMBINE feature)*.

Notes: Regular file component of Combined_File.txt works properly. The WhiteSpace version prints the appropriate error.

Test 11: Check that what has been written to WhiteSpace file is being registered by program (*test_tabs_spaces_registered.txt)*.

Notes: In this file the number of space and tabs on a line is the same as the line number itself. So line one has one space, one tab, line two has two spaces, two tabs etc. The program then registers these correctly. (Note that the number of tabs displayed in this message is one less than the number of tabs entered. This is as the first tab on the line is used as an indicator as to whether the line contains a number or not).



Tabs and spaces being correctly registered by program.

Test 12: Check that the movement of the password files makes sense.

Notes: Covered more extensively in the password testing section below.

Test 13: Try entering animation speed out of bounds.

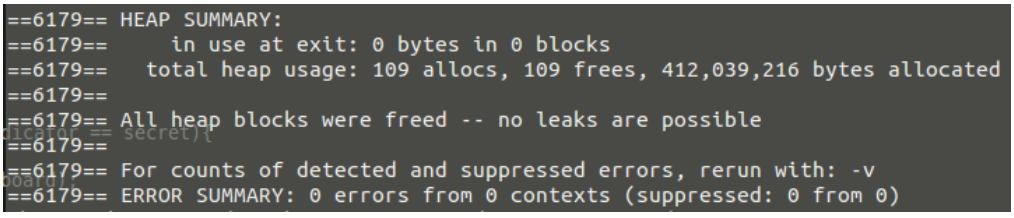
Notes: Both numbers that are too high and too low are registered and the user is prompted to try again.

Test 14: Lowest animation speeds entered (check CPU)

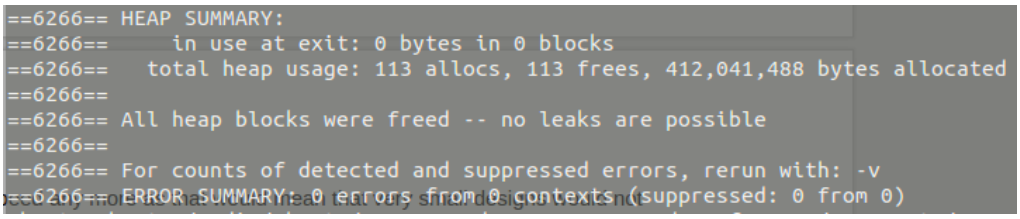
Notes: Breaks my laptop on the multiple embedded loops one at 500, but is OK on the lab computers. The program doesn't like adding so many different things to a linked list so rapidly, especially as the boards are so large. However, it did not make sense to limit the speed any more as that would mean that very small designs would not animate at all. Therefore I compromised and printed a warning.

Test 13: ValGrind

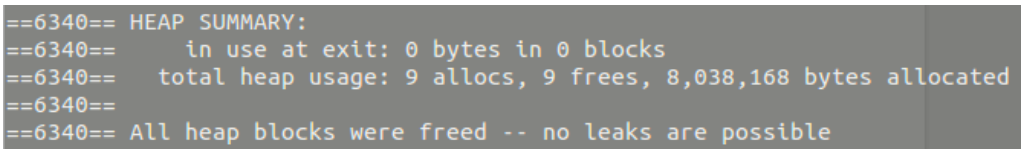
Notes: Ran clean on a regular run, decoding a WhiteSpace file run, and a translation run:



Regular run ValGrind Output

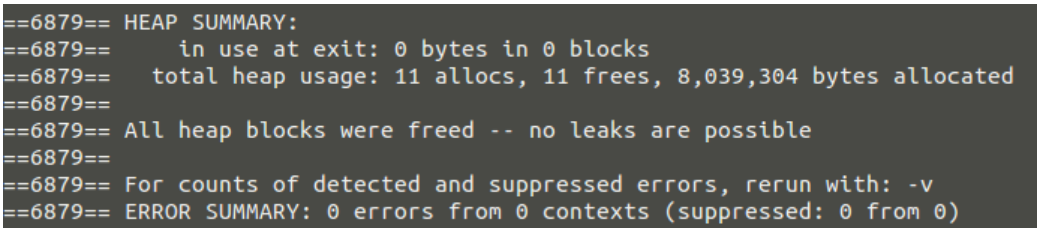


SECRET run ValGrind Output



TRANSLATE run ValGrind Output

However on a combination run there was an error about accessing an uninitialized area of memory. This was as I had tried to access the line after the last line of the WhiteSpace file and regular files which had not had memory allocated to it and therefore did not exist. This was then fixed.



Clean COMBINE ValGrind OutPut

Extension Password Testing

At this point I was fairly confident that everything worked. I had thoroughly checked the parser/ interpreter and also the WhiteSpace extension. All that I needed to check now was the password extension. To do this I wrote a standalone program to place the password functions into to test all possible combinations of attempted passwords and their results.

The way this works is that we use the password functions to set a password (starting with 1, then going up to 1000). We then use the password entry functions with inputs starting at 1 going up to 1000 to try and crack it. If the two passwords don't match and it we error and quit. If they do match we print this to a file which is included in submission and can be checked.