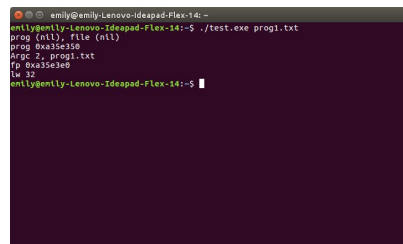


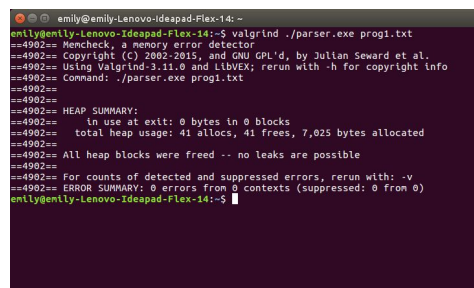
Testing Report

I used a combination of testing strategies to ensure my code is bulletproof. While I did some overarching, high level analysis using black box testing and tools such as GCov and Valgrind, the most fruitful testing was done during the build process on individual functions/lines. My testing report will walk through my interpreter program outlining how each part has been tested (as the interpreter is simply an extension of the parser, I will not discuss them separately).

As the main function does little but call other functions to progress through the program, the only thing I tested here was that all pointers/integers had been initialised correctly. I also tested that my “openfile” function was behaving correctly, by checking that it identifying the correct file to open and the file pointer “fp” pointed to this location (as opposed to NULL). I did this by printing the appropriate values as seen below.

A terminal window with a dark purple background. The prompt is 'emily@emily-Lenovo-IdeaPad-Flex-14:~'. The user enters 'prog (nil), file (nil)' and 'prog mainloop'. The output shows 'arg 2: prog1.txt' and 'fp: 0x00000000'. The prompt returns to 'emily@emily-Lenovo-IdeaPad-Flex-14:~\$'.

Beyond main, I primarily used CUnit for unit testing (file included in submission). This tested that my “readfile” was reading in the correct strings and placing them correctly in the array. I also used Valgrind to ensure that no memory leaks were occurring when resizing my word array, and GCov to ensure my enlarge_array function was being called the correct number of times.

A terminal window with a dark purple background. The prompt is 'emily@emily-Lenovo-IdeaPad-Flex-14:~'. The user enters 'valgrind ./parser.exe prog1.txt'. The output shows Valgrind version 4.0.2, copyright information, and a command line. It then shows a 'HEAP SUMMARY' section with details on memory usage: 'in use at exit: 0 bytes in 0 blocks', 'total heap usage: 41 allocs, 41 frees, 7,025 bytes allocated'. It concludes with 'All heap blocks were freed -- no leaks are possible' and 'ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)'. The prompt returns to 'emily@emily-Lenovo-IdeaPad-Flex-14:~\$'.

While CUnit was valuable in testing some functions, I used it fairly limitedly throughout the build process as I often needed to know not only that an assertion had failed, but why it had failed and what the actual values were. This was often much more easily discovered through white box testing individual functions. For example, the total degrees, left_turn and right_turn functions used to calculate the angle of movement caused many problems as I needed the numbers to wrap around at 360 degrees. My initial right_turn function was as follows:

```

void right_turn(Prog *ptr, float val)
{
    if ((val+ptr->degrees)>CIRCLE){
        ptr->degrees = (val+ptr->degrees)-CIRCLE;
    }
    else {
        ptr->degrees = ptr->degrees+(val);
    }
}

```

However, after examining the outputs and putting in large degree turns such as RT 400, I realised this was not wrapping around and giving me an ever increasing value for ptr->degrees. Below I altered the function and tested the outputs appropriately:

```

int right_turn(Prog *ptr, float val)
{
    if ((val+ptr->degrees)>CIRCLE){
        ptr->degrees = (val+ptr->degrees)-CIRCLE;
        while (ptr->degrees>CIRCLE){
            ptr->degrees-=CIRCLE;
        }
        return 1;
    }
    else {
        ptr->degrees = ptr->degrees+(val);
        return 1;
    }
}

```

GCov (reports in submission folder) alerted me to the bug in this function as I realised that the “else” statement was never being executed.

My functions related to the do loop instruction were perhaps the hardest to debug and test as they operate recursively. My initial looping function was as follows:

```

void loop_thru(char c, int start, int finish, Prog *ptr, DrawPic *n)
{
    for (i=start;i<finish+1;i++){
        while (strcmp(ptr->wds[ptr->cw], "}")!=0){
            instruction(ptr, n);
            ptr->cw=ptr->cw+1;
        }
        b=ptr->cw;
    }
}

```

```

        n->vars[(c-'A')]=i;
        ptr->cw=a;
    }
    ptr->cw=b;
}

```

Not only did I realise this would not work if there were no closing brackets (an invalid program) but also that it would not work if I had for example:

```

DO A FROM 1 TO 10 {
    SET A := 2 B * ; etc.

```

My for loop was going to loop through 10 times regardless of the value of A. My modified function has been changed to a while loop to accommodate this:

```

while ((s*ptr->vars[(c-'A')])<finish+s){
    instrclist(ptr, sw);
    end=ptr->cw; /*Track end of doloop*/
    ptr->vars[(c-'A')]+=s; /*Update variable*/
    ptr->cw=begin; /*Jump to top of doloop*/
}

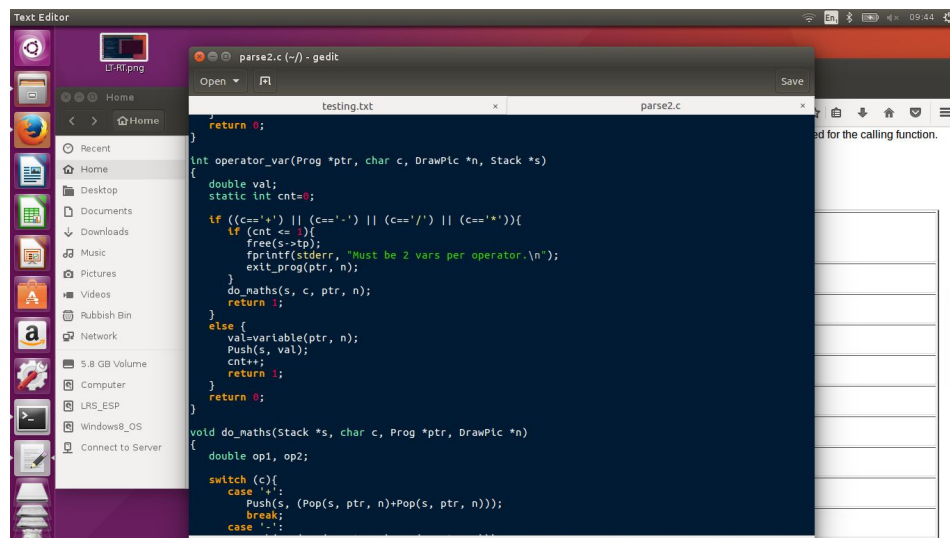
```

At first I had two functions allowing the do loop to increase and decrease (eg. Do A from 1 to 10, and Do a from 10 to 1). While this worked effectively when testing, it was clunky and the functions were close to identical. I decided to add the variable 's' to allow for increasing and decreasing loops . If it is increasing s=1, if it is decreasing s=-1. This integer is then used in the initial condition for my while loop, however it does not change the value of the variable in ptr->vars.

A crucial part to ensuring that my do loops were operating correctly was ensuring that my variables were being checked and assigned. I chose not to use sscanf as this would not pick up on certain errors such as "1.5*" (simply converted to 1.5). The only way to accurately check whether the variable/number was valid was to check each individual character. CUnit was also able to test this by checking my variable values at the end of the program. I further tested this through my black box testing at the end where I through many different possible options at the function to make sure it threw an error correctly. This picked up on the fact that my initial function was accepting variables such as "-4.5.6" and "-.". I therefore introduced a count of the number '.' to ensure this would not be accepted, and a further function (valid_number) to confirm that there was at least one number in the string (so that ".", "-", and "-." would not be accepted). I was also able to print my ptr->vars array to check the values were assigned correctly.

The SET instruction was also best tested using white and black box testing as it operates recursively with Reverse Polish Notation. For my Parser, it was very simple to test that each

string was valid and I did not need to test the validity of the Reverse Polish expression. For the interpreter, my initial operator_var function was as follows:



```
int operator_var(Prog *ptr, char c, DrawPlc *n, Stack *s)
{
    double val;
    static int cnt=0;

    if ((c=='+' || c=='-' || c=='/' || c=='*')){
        if (cnt <= 1){
            free(s->tp);
            fprintf(stderr, "Must be 2 vars per operator.\n");
            exit_prog(ptr, n);
        }
        do_maths(s, c, ptr, n);
        return 1;
    }
    else {
        val=variable(ptr, n);
        Push(s, val);
        cnt++;
        return 1;
    }
    return 0;
}

void do_maths(Stack *s, char c, Prog *ptr, DrawPlc *n)
{
    double op1, op2;

    switch (c){
        case '+':
            Push(s, (Pop(s, ptr, n)+Pop(s, ptr, n)));
            break;
        case '-':
            Push(s, (Pop(s, ptr, n)-Pop(s, ptr, n)));
            break;
        case '/':
            Push(s, (Pop(s, ptr, n)/Pop(s, ptr, n)));
            break;
        case '*':
            Push(s, (Pop(s, ptr, n)*Pop(s, ptr, n)));
            break;
    }
}
```

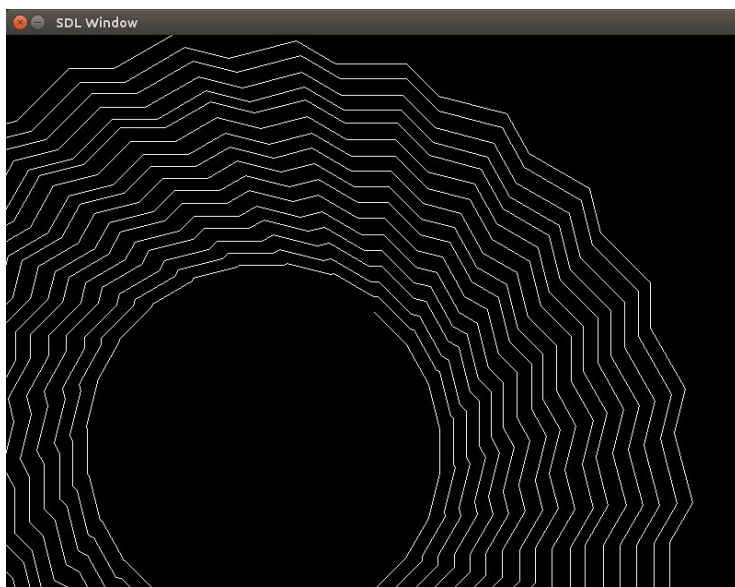
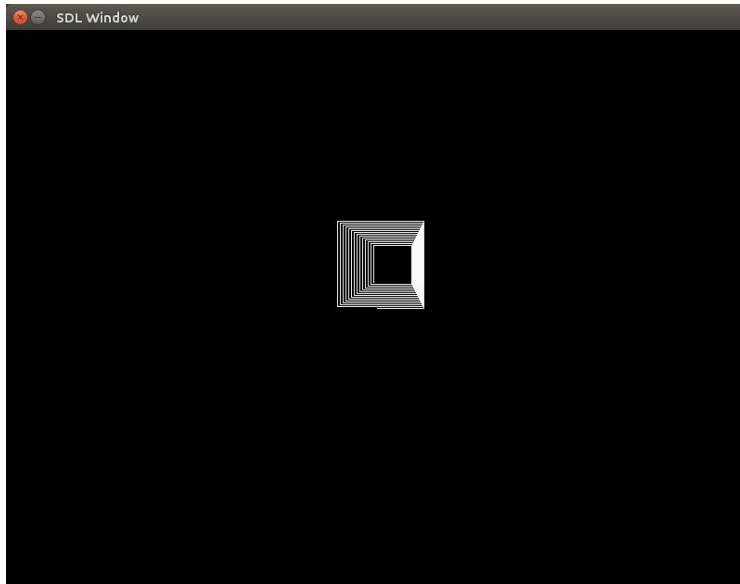
With this method I was trying to ensure that there were enough operators per operand. However this did not work for a number of black box tests, for example “SET A := 5 * ;”. It also did not catch a string such as “*5”. I have now adjusted my operator_var function so that it checks the string length of operators and my “Pop” function will error if there are insufficient operands. I have also added a function, “check_stack” which ensures that “SET A := 5 6 ;” will not be interpreted as it is incorrect polish notation. The polish notation was also very easy to test through black box testing. This involved throwing all sorts of expressions, variables, and operators to the program and ensuring that they failed correctly.

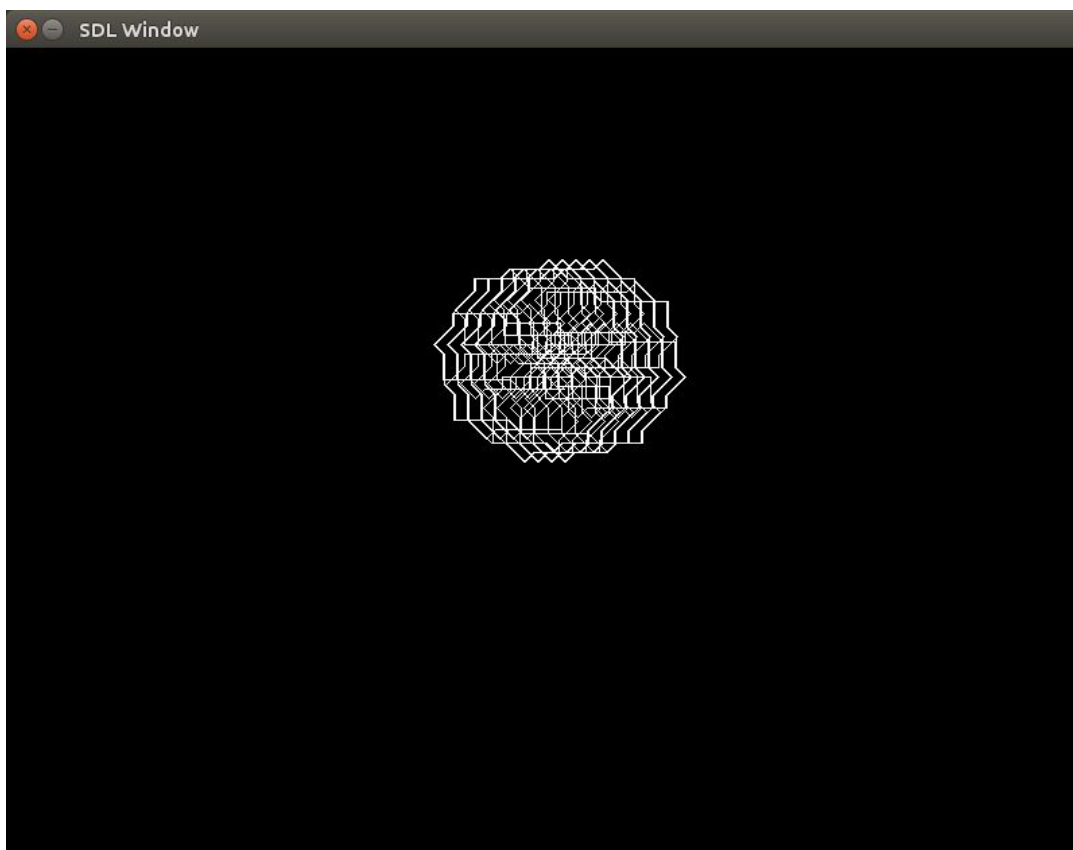
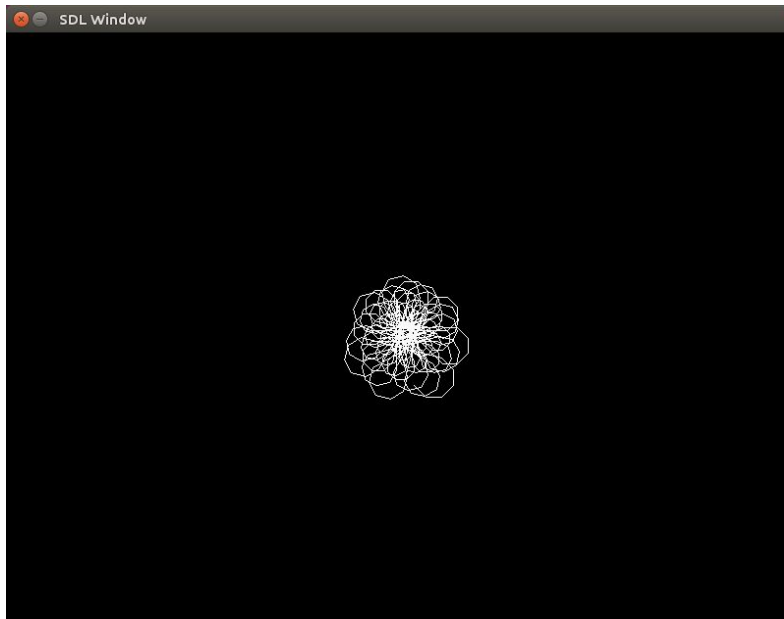
While dividing by zero is not allowed in maths, I made the decision that my program would not throw an error in this situation and have tested that it will not seg fault. It produces a wacky SDL output but is not invalid coding and therefore should be allowed by both the parser and interpreter.

My exit functions, exit_prog, freewords and freestack were tested continuously throughout the process of testing other functions. I do not call exit_prog before ptr->wds[0] or after ptr->wds[lw-1] as this would cause a segmentation fault when printing the strings before and after the current word. I used Valgrind to ensure memory was freed appropriately.

With regards to black box testing, I have included a sample of some test programs I ran to ensure my parser/interpreter failed correctly.

Sample Outputs





Output 1:

```
{  
  FD 50  
  DO A FROM 100 TO 1 {  
    SET B := 5 A - ;  
    LT 90  
    FD B  
  }  
}
```

Output 2:

```
{  
  DO A FROM 1 TO 500 {  
    SET C := A 5 / ;  
    FD C  
    RT 45  
    FD 50  
    LT 30  
  }  
}
```

Output 4:

```
{  
  DO A FROM 1 TO 50 {  
    SET C := 5 ;  
    FD C  
    RT 45  
    DO B FROM 1 TO 10 {  
      LT 90  
      FD 10  
      RT 45  
      FD 20  
    }  
  }  
}
```

Output 3:

```
{
DO A FROM 20 TO 100 {
    FD A
    RT 30
DO B FROM 1 TO 1 {
    SET C := A 5 / ;
    FD C
    RT 45
DO B FROM 1 TO 1 {
    SET C := A 5 / ;
    FD C
    RT 45
DO B FROM 1 TO 1 {
    SET C := A 5 / ;
    FD C
    RT 45
DO B FROM 1 TO 1 {
    SET C := A 5 / ;
    FD C
    RT 45
}
}
}
}
}
}
```