Tom Russell – tr16879

PARSER/INTERPRETER ASSIGNMENT - TESTING REPORT

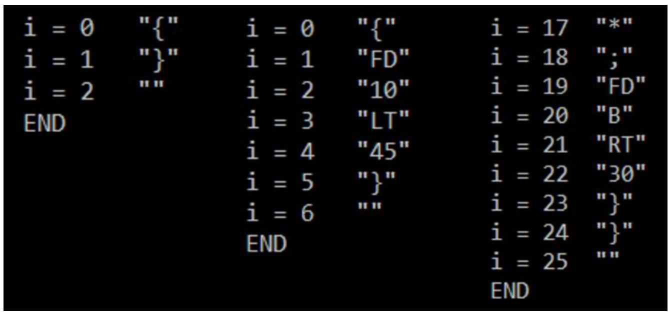
TABLE OF CONTENTS

SECTION 1: THE PARSER

BUGS/QUESTIONS RAISED WHILE WRITING THE PARSER:

- The parser has been written to accept variables and numbers in both decimal and/or negative formats.
- Although all negative operations can be performed more simply with positive ones (such as a right turn instead of a negative left turn), allowing negative numbers gives more options to the user when writing movement files.
- This is also preferable to the alternatives of defaulting negative numbers to zero, which could be misleading, or exiting the program when a negative number is used.

- The program has been written to run with any size of input file. I decided against using a dynamically resizing list or linked-list strategy, opting instead to have a first pass over the file to determine the number of tokens before creating the token array. I chose this method since the array will not be modified in any way after the file has been loaded, so there is no need to resize the list during parsing.
- The array always has an empty string token at the end, so that the parser can check if the last token has been reached. This prevents an out of bounds array access if the final token is not correct.
- Several example files of different length were used to test the array sizing functionality:


```
i = 0   "{"     i = 0   "{"      i = 17  "*"
i = 1   "}"     i = 1   "FD"     i = 18  ";"
i = 2   ""      i = 2   "10"     i = 19  "FD"
END             i = 3   "LT"     i = 20  "B"
                i = 4   "45"     i = 21  "RT"
                i = 5   "}"      i = 22  "30"
                i = 6   ""       i = 23  "}"
                END              i = 24  "}"
                                 i = 25  ""
                                 END
```

- In order to make the upgrade from parser to interpreter easier, several functions have been structured in an unusual way. Using the ProgramOP function as an example, instead of checking if a token in a polish expression is any valid operator, it checks the 4 operators separately. Since the interpreter will need to apply a different operator based on the token, this will prevent code from having to be significantly modified for the interpreter.

- Since the formal grammar defines the end of the file to be the closing brace (‘}’) from the initial instruction list, it was decided that if any extra tokens are present at the end of the file then a parsing error should be triggered.

- For the parser it was decided that polish expressions and do loops only need to follow the formal grammar. Semantics issues such as infinite do loops and invalid polish expressions are covered in the interpreter section.

BLACK-BOX TESTING ON THE PARSER:

At this point the program should parse all valid input files. Print statements have been added to several of the instruction functions (can be turned off by setting TESTING = 0 in the header file), so that when FD, LT, RT, DO, SET, VAR & NUM are used the program output will indicate if the program has parsed correctly.

Various black-box tests were made based on the formal grammar, aimed to either fit to the grammar or cause a parsing error. To verify the output, we check if parsing fails on the specific token we expect it to. Where a test gives us an incorrect result we attempt a fix, detailed in the section below the table, then reattempt the test.

Test File Description	**Grammar Tested**	**Expected Output**	**Test Result**
`Empty text file`	`<MAIN>`	`Error: token 1`	`Pass`
`Incorrect starting token, instruction (not '{')`	`<MAIN>`	`Error: token 1`	`Pass`
`Incorrect starting token, number (not '{')`	`<MAIN>`	`Error: token 1`	`Pass`
`Valid empty program { }`	`<INSTRCTLST>`	`Parse Success`	`Pass`
`Correct starting token only`	`<INSTRCTLST>`	`Error: token 2`	`Pass`
`Invalid instruction token`	`<INSTRUCTION>`	`Error: token 2`	`Pass`
`Invalid instruction (FD XX)`	`<FD><VARNUM>`	`Error: token 3`	`Fail`(1)
`Invalid instruction (FD XX) (att. 2)`	`<FD><VARNUM>`	`Error: token 3`	`Pass`
`Valid instruction with number (FD 30)`	`<FD><VARNUM>`	`Parse Success`	`Fail`(2)
`Valid instruction with number (FD 30) (att. 2)`	`<FD><VARNUM>`	`Parse Success`	`Pass`
`Valid instruction statement with Variable (FD A)`	`<FD><VARNUM>`	`Parse Success`	`Pass`
`Invalid instruction statement (LT AB)`	`<LT><VARNUM>`	`Error: token 3`	`Pass`
`Valid instruction with decimal number (LT 30.5)`	`<LT><VARNUM>`	`Parse Success`	`Pass`
`Valid instruction statement with Variable (LT Z)`	`<LT><VARNUM>`	`Parse Success`	`Pass`
`Invalid instruction statement (RT -A)`	`<RT><VARNUM>`	`Error: token 3`	`Pass`
`Valid instruction with negative number (RT -150)`	`<RT><VARNUM>`	`Parse Success`	`Pass`
`Valid instruction statement with Variable (RT F)`	`<RT><VARNUM>`	`Parse Success`	`Pass`
`Invalid Number (--120)`	`<VARNUM>`	`Error: token 3`	`Pass`
`Invalid Number (-40.0-)`	`<VARNUM>`	`Error: token 3`	`Pass`
`Invalid Number (10.9.)`	`<VARNUM>`	`Error: token 3`	`Pass`
`Invalid Number (.5.1)`	`<VARNUM>`	`Error: token 3`	`Pass`
`Valid Number (10.0)`	`<VARNUM>`	`Parse Success`	`Pass`
`Valid Number (-1.0)`	`<VARNUM>`	`Parse Success`	`Pass`
`Valid Number (-50)`	`<VARNUM>`	`Parse Success`	`Pass`
`Valid DO statement (DO A FROM 1 TO 8)`	`<DO>`	`Parse Success`	`Fail`(3)
`Valid DO statement (DO A FROM 1 TO 8) (att. 2)`	`<DO>`	`Parse Success`	`Pass`
`Valid DO statement (DO A FROM -100.5 TO 100.2)`	`<DO>`	`Parse Success`	`Pass`
`Valid DO statement (DO A FROM 1.2 TO 1.2)`	`<DO>`	`Parse Success`	`Pass`
`Invalid DO statement (DO C 1 TO 8)`	`<DO>`	`Error: token 4`	`Pass`
`Invalid DO statement (DO 4 FROM 1 TO 8)`	`<DO>`	`Error: token 3`	`Fail`(4)
`Invalid DO statement (DO 4 FROM 1 TO 8) (att. 2)`	`<DO>`	`Error: token 3`	`Pass`
`Invalid DO statement (DO C FROM TO 8)`	`<DO>`	`Error: token 5`	`Pass`
`Invalid DO statement (DO C FROM 1 8)`	`<DO>`	`Error: token 6`	`Pass`
`Nested DO statement (DO within DO)`	`<DO>`	`Parse Success`	`Pass`
`Nested SET within DO statement`	`<DO><SET>`	`Parse Success`	`Pass`
`Valid SET statement (SET A := 5 ;)`	`<SET><POLISH>`	`Parse Success`	`Pass`
`Invalid SET statement (SET 4 := 5 ;)`	`<SET><POLISH>`	`Error: token 3`	`Pass`
`Invalid SET statement (missing :=)`	`<SET><POLISH>`	`Error: token 4`	`Pass`
`Invalid SET statement (missing semi-colon)`	`<SET><POLISH>`	`Error: token 6`	`Pass`
`Invalid SET statement (SET Z = 12.5 ;)`	`<SET><POLISH>`	`Error: token 4`	`Pass`

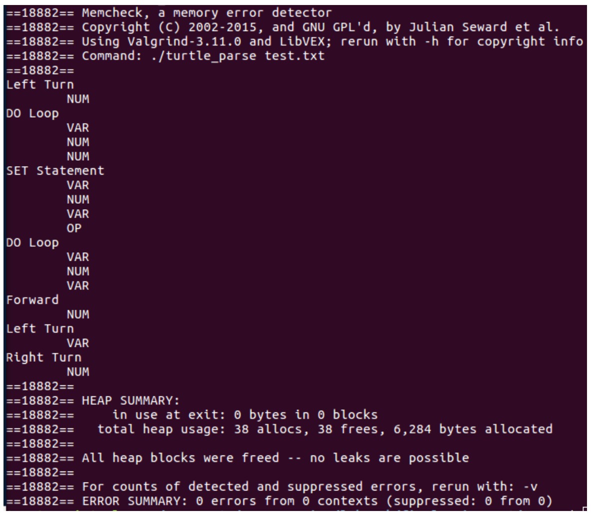
Valid SET statement with mixed vars & operators 1	<SET><POLISH>	Parse Success	Pass
Valid SET statement with mixed vars & operators 2	<SET><POLISH>	Parse Success	Pass
Valid SET statement with mixed vars & operators 3	<SET><POLISH>	Parse Success	Pass
Tokens remaining after MAIN closing brace 1	<MAIN>	Error: token 7	Pass
Tokens remaining after MAIN closing brace 2	<MAIN>	Error: token 15	Pass

BUG FIXES ON THE PARSER:

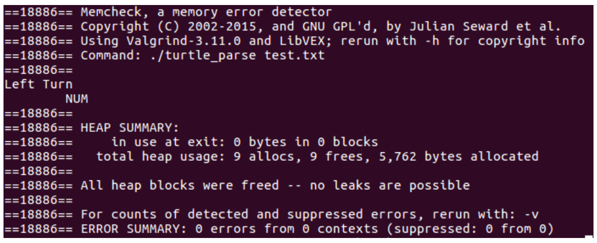
1. Parser gives us no error when given an invalid VARNUM token.
    - If statement was written incorrectly when checking if a number is valid. Instead of allowing decimal points it was allowing all characters.
2. Parser fails when trying to read in a number token.
    - Simple syntax mistake; '<' used instead of '>'
3. Parser fails when testing the DO instruction
    - Error occurs when parsing the '{' token as part of the DO statement
        - '{' token is parsed twice by accident
    - The current token counter is incremented after the start location of the do loop is copied
    - When the do loop begins, it starts on the '{' token instead of the first instruction in the DO loop
    - Switched order of two lines to fix, so the current token increments before storing the value.
4. Parser fails during DO loop when a number is given instead of a variable.
    - Variable function had been modified to return a -1 instead of a '0' character when the token is not a variable. DO function had not been updated to reflect this.

TESTING THE PARSER FOR MEMORY LEAKS:

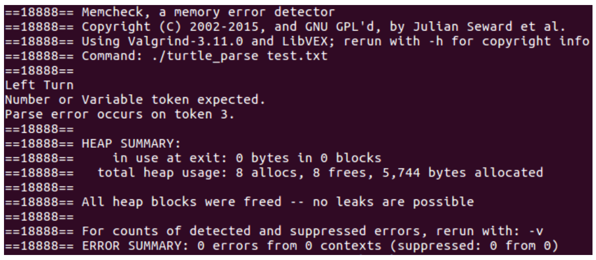
Valgrind was used to check for memory leaks in the parser. A short valid input, long valid input and an invalid input that will fail to parse were use as tests. All 3 tests reported no memory leaks.



```
==18882== Memcheck, a memory error detector
==18882== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==18882== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==18882== Command: ./turtle_parse test.txt
==18882==
Left Turn
        NUM
DO Loop
        VAR
        NUM
        NUM
SET Statement
        VAR
        NUM
        VAR
        OP
DO Loop
        VAR
        NUM
        VAR
Forward
        NUM
Left Turn
        VAR
Right Turn
        NUM
==18882==
==18882== HEAP SUMMARY:
==18882==     in use at exit: 0 bytes in 0 blocks
==18882==   total heap usage: 38 allocs, 38 frees, 6,284 bytes allocated
==18882==
==18882== All heap blocks were freed -- no leaks are possible
==18882==
==18882== For counts of detected and suppressed errors, rerun with: -v
==18882== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```



```
==18886== Memcheck, a memory error detector
==18886== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==18886== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==18886== Command: ./turtle_parse test.txt
==18886==
Left Turn
        NUM
==18886==
==18886== HEAP SUMMARY:
==18886==     in use at exit: 0 bytes in 0 blocks
==18886==   total heap usage: 9 allocs, 9 frees, 5,762 bytes allocated
==18886==
==18886== All heap blocks were freed -- no leaks are possible
==18886==
==18886== For counts of detected and suppressed errors, rerun with: -v
==18886== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```



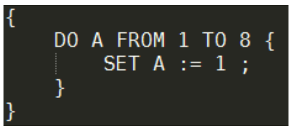
```
==18888== Memcheck, a memory error detector
==18888== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==18888== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==18888== Command: ./turtle_parse test.txt
==18888==
Left Turn
Number or Variable token expected.
Parse error occurs on token 3.
==18888==
==18888== HEAP SUMMARY:
==18888==     in use at exit: 0 bytes in 0 blocks
==18888==   total heap usage: 8 allocs, 8 frees, 5,744 bytes allocated
==18888==
==18888== All heap blocks were freed -- no leaks are possible
==18888==
==18888== For counts of detected and suppressed errors, rerun with: -v
==18888== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

SECTION 2: THE INTERPRETER

BUGS/QUESTIONS RAISED WHILE WRITING THE INTERPRETER:

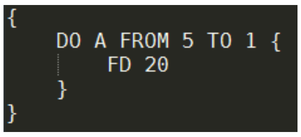
- The formal grammar for <POLISH> suggests that two odd cases should parse correctly:
    - A single token semicolon token ';'.
    - Unbalanced polish expressions. Computable polish expressions must have N variables and N-1 operators.
- I have chosen to disallow these two cases, since this makes it clear to the user there is an error in their input, rather than taking some other action such as defaulting to 0 which could mislead the user.

- Two semantics issues could also arise with the <DO> formal grammar:
    - Users could modify the variable used for the do loop from within the do loop, causing an infinite loop and the program to freeze for the user. Example on the right.
    - To avoid this the parser will keep track of variables currently in use in looping and attempting to modify an in-use variable will cause a parsing error.



```
{
    DO A FROM 1 TO 8 {
        SET A := 1 ;
    }
}
```

-
    - The second semantics problem is if the user creates a DO loop that will never run. Example on the right.
    - I have chosen to cause a parse error in these cases rather than allowing this to parse but running the loop 0 times, since the latter could be misleading and cause difficult in tracking down bugs for the user.



```
{
    DO A FROM 5 TO 1 {
        FD 20
    }
}
```

- Since the math.h functions require angles in radian form, it was decided that the angle of the 'turtle' will be stored in radian form.
    - In order to conform to the SDL standard of x increasing to the right and y increasing downwards:
        - The 0 rad angle will also point downwards, with the π rad angle pointing upwards.
    - The angle will increase in an anti-clockwise direction, so the angle to the right will be $\frac{\pi}{2}$, with $\frac{3\pi}{2}$ to the left.
    - The angle will be kept within the range $0 \leq h < 2\pi$, leaving this range will cause it to wrap around.

- Initially the 'turtle' coordinate were stored as integers to match the SDL coordinates system. This causes significant errors when drawing in the SDL window, since the coordinates will be rounded down, especially for small movements (e.g. forward 1 pixel). Calculating and storing the coordinates as doubles, then typecasting the coordinates to integers when drawing in the SDL window solves this problem.

- While testing the interpreter minor errors were occurring with the SDL, with straight lines being drawn at a 1 pixel angle, despite coordinates appearing correct.
- This was due to inaccuracies with floating points, where numbers would often drift fractionally below the correct value, such as 9.99999 instead of 10.00000. When this is typecasted to an int, it is rounded down to 9 instead of rounding up to 10.
- To solve this a rounding function was written to round the number the correct way when calculating SDL coordinates.

REGRESSION TESTING THE INTERPRETER:

All previous tests done on the formal grammar for the parser were repeated, to ensure all parsing functionality had not been modified.

BLACK-BOX TESTING ON THE INTERPRETER:

At this point the program should correctly parse all files, calculate coordinates based on forward and left/right turn instructions, and draw a line on the screen for each forward instruction.

The black-box testing will focus on the specific functionality that was added for the interpreter. This includes valid polish expressions, variables in use being unmodifiable, DO loops that never run & changes in coordinates and angle values.

To help determine if the interpreter is acting correctly, print statements have been added to for FD, LT & RT to print out coordinates and angle value changes (can be turned off by setting TESTING = 0 in the header file). Coordinates are represented in the SDL format, with (0,0) being the top left corner, y increasing downwards and x increasing to the right. Some tests were also repeated several times with different input files, with the number of repeats indicated in square brackets.

Test File Description [repeats]	**Functionality Tested**	**Expected Output**	**Test Result**
Valid Polish expression	Polish	Correct Value	Fail(1)
Valid Polish expression (att. 2) [10]	Polish	Correct Value	Pass
Invalid Polish expression (too few operators) [3]	Polish	Parser Error	Pass
Invalid Polish (incorrect operator placement) [3]	Polish	Parser Error	Pass
Invalid Polish expression (too many operators) [3]	Polish	Parser Error	Pass
Set a Var to a value, check Var is correct value [5]	Setting Vars	Correct Value	Pass
DO within DO, using same variable	Vars in use	Parser Error	Pass
Multi-nested DO, using same variable as outermost DO [3]	Vars in use	Parser Error	Pass
Do within DO, using different variable	Vars in use	Parse Success	Pass
SET within DO, using same variable	Vars in use	Parser Error	Pass
Multi-nested SET, using same variable as outermost DO [3]	Vars in use	Parser Error	Pass
SET within DO, using different variable	Vars in use	Parse Success	Pass
DO with invalid range (start > end) [3]	Do loops	Parser Error	Pass
DO with valid start range (start <= end) [3]	Do loops	Parse Success	Pass
Turn angle outside $0 < 2\pi$ range	Angle Range	Angle loops to stay in range	Fail(2)
Turn angle outside $0 < 2\pi$ range	Angle Range	Angle loops to stay in range	Fail(3)
Turn angle outside $0 < 2\pi$ range [5]	Angle Range	Angle loops to stay in range	Pass
Turn angle set amounts, multiple times (LT) [5]	Angle Changes	Angle changes expected amount	Pass
Turn angle set amounts, multiple times (RT) [5]	Angle Changes	Angle changes expected amount	Pass
Change angle many times (LT x 1,000,000)	Angle Changes	Angle changes expected amount	Pass
Change angle many times (RT x 1,000,000)	Angle Changes	Angle changes expected amount	Pass

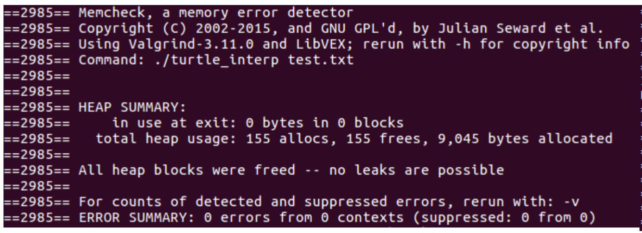
Up/Down/Left/Right Coordinate Changes [10]	Coord Changes	Coords change expected amount	Pass
U/D/L/R Coordinate Changes many times (FD x 1000) [10]	Coord Changes	Coords change expected amount	Pass
Angled Coordinate Changes [10]	Coord Changes	Coords change expected amount	Pass

BUG FIXES ON THE INTERPRETER:

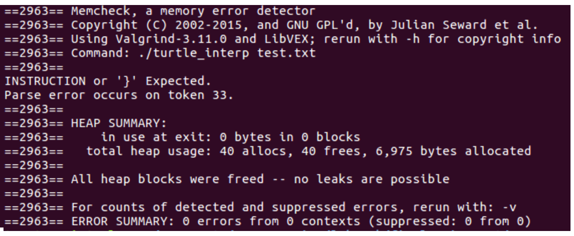
1. Parser gives us an incorrect output for the given polish expression.
    - Values were taken off the stack and assigned in the wrong order. When given the expression ‘5 2 /’ we need to evaluate ‘5 / 2’ and not ‘2 / 5’.
    - Value assignment names (val1 &val2) were switched to fix this.
2. Angle stays within the correct range, but LT and RT instructions both turn in the same direction.
    - When modifications were made to these functions, the RT no longer changed the input value to a negative number (LT should be positive, RT negative).
    - Minus symbol was added to fix this.
3. Angle occasionally prints out as being 360 degrees, which is not within the valid range.
    - It was concluded that this was due to inaccuracies with using floating point values. The angle should be set to 360, then wrap back round to 0. However, the number is actually a tiny fraction below 360 (which can only be seen when printing with > 10 decimal places) hence it does not loop back round to 0.
    - These miniscule inaccuracies in angle will have no visible effect, since a change of a fraction of a pixel while calculating coordinates will not affect the SDL drawing.

TESTING THE INTERPRETER FOR MEMORY LEAKS:

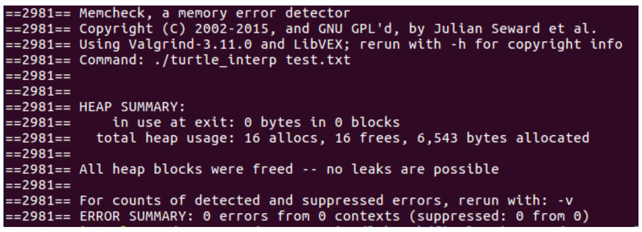
Unfortunately the use of SDL causes significant memory leaks. To test the underlying interpreter code for leaks, all SDL features were disabled before being run through valgrind. A short valid input, long valid input and an invalid input were used for testing.



```
==2985== Memcheck, a memory error detector
==2985== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==2985== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==2985== Command: ./turtle_interp test.txt
==2985==
==2985==
==2985== HEAP SUMMARY:
==2985==     in use at exit: 0 bytes in 0 blocks
==2985==   total heap usage: 155 allocs, 155 frees, 9,045 bytes allocated
==2985==
==2985== All heap blocks were freed -- no leaks are possible
==2985==
==2985== For counts of detected and suppressed errors, rerun with: -v
==2985== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```



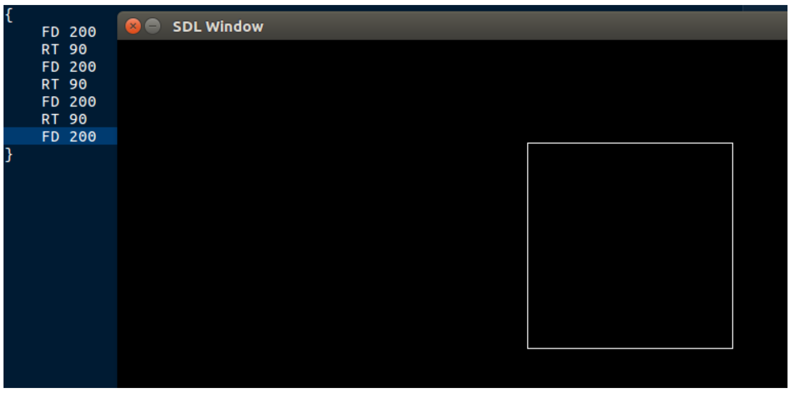
```
==2963== Memcheck, a memory error detector
==2963== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==2963== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==2963== Command: ./turtle_interp test.txt
==2963==
INSTRUCTION or '}' Expected.
Parse error occurs on token 33.
==2963==
==2963== HEAP SUMMARY:
==2963==     in use at exit: 0 bytes in 0 blocks
==2963==   total heap usage: 40 allocs, 40 frees, 6,975 bytes allocated
==2963==
==2963== All heap blocks were freed -- no leaks are possible
==2963==
==2963== For counts of detected and suppressed errors, rerun with: -v
==2963== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```



```
==2981== Memcheck, a memory error detector
==2981== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==2981== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==2981== Command: ./turtle_interp test.txt
==2981==
==2981==
==2981== HEAP SUMMARY:
==2981==     in use at exit: 0 bytes in 0 blocks
==2981==   total heap usage: 16 allocs, 16 frees, 6,543 bytes allocated
==2981==
==2981== All heap blocks were freed -- no leaks are possible
==2981==
==2981== For counts of detected and suppressed errors, rerun with: -v
==2981== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

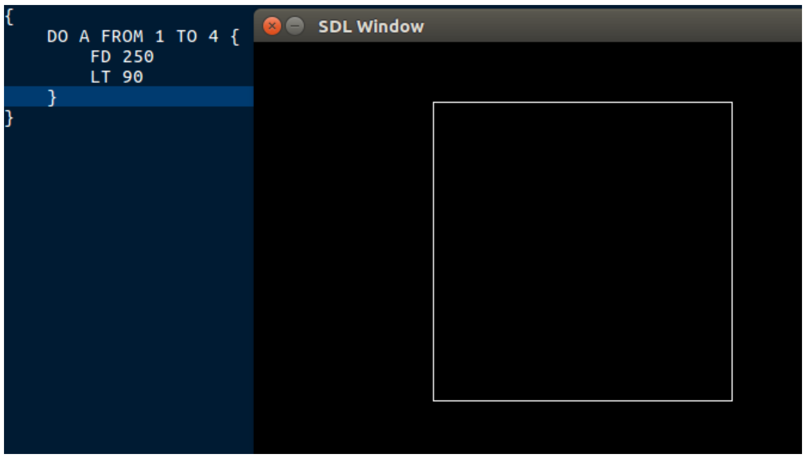
SECTION 3 - EXAMPLE PROGRAM OUTPUTS:

This section will show example outputs for the interpreter. Input files have been selected to show all the key functionality of the program.



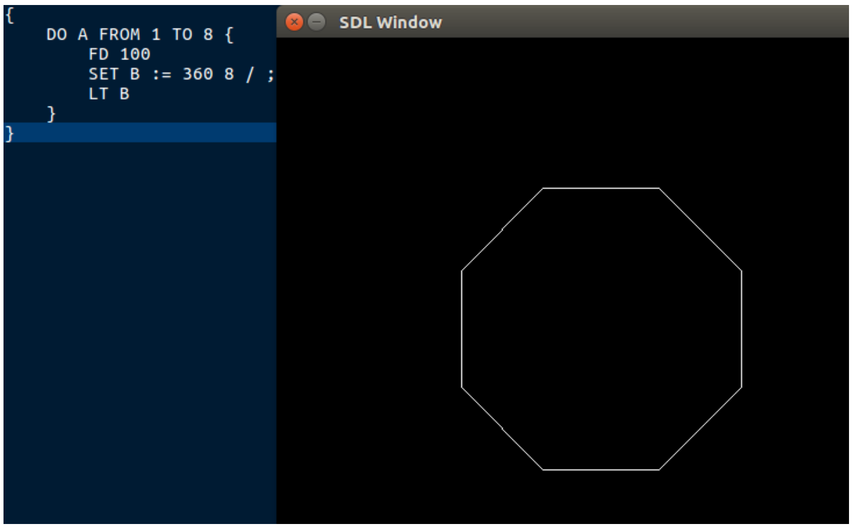
```
{
    FD 200
    RT 90
    FD 200
    RT 90
    FD 200
    RT 90
    FD 200
}
```

Figure 1 - Simple square using FD & RT



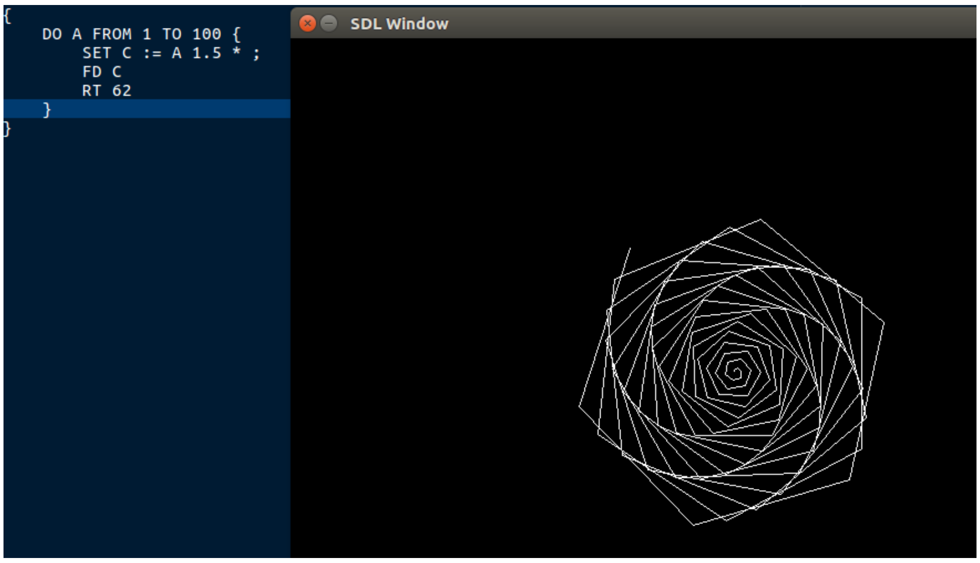
```
{
    DO A FROM 1 TO 4 {
        FD 250
        LT 90
    }
}
```

Figure 2 - Simple square using Loops



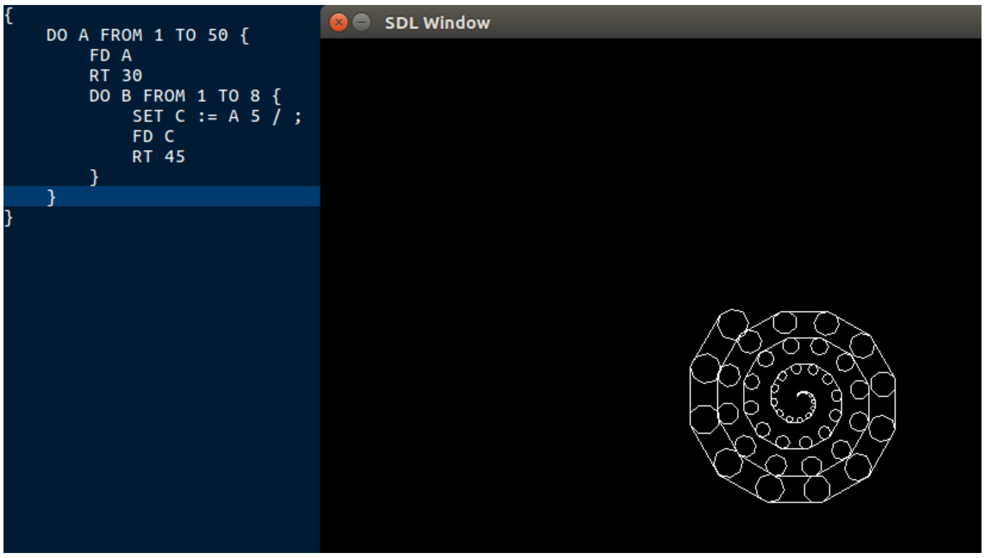
```
{
    DO A FROM 1 TO 8 {
        FD 100
        SET B := 360 8 / ;
        LT B
    }
}
```

Figure 3 - Octagon using SET & DO



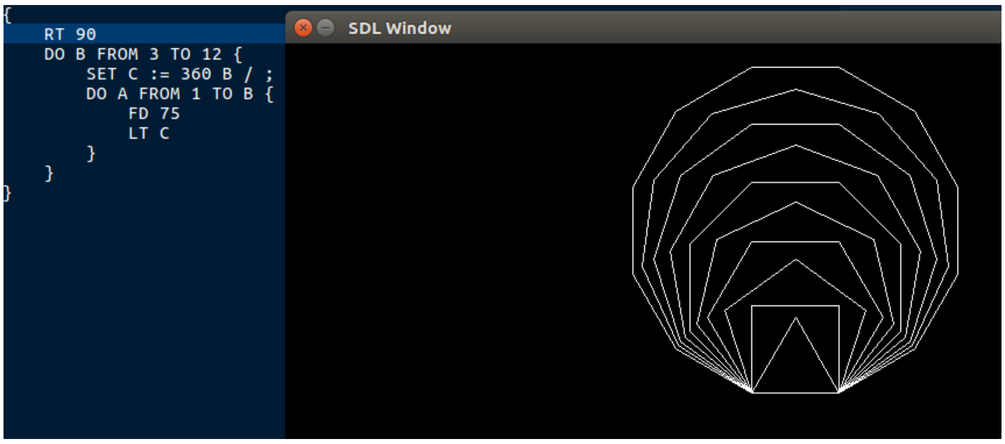
```
{
    DO A FROM 1 TO 100 {
        SET C := A 1.5 * ;
        FD C
        RT 62
    }
}
```

Figure 3 - Complex DO Loop Example



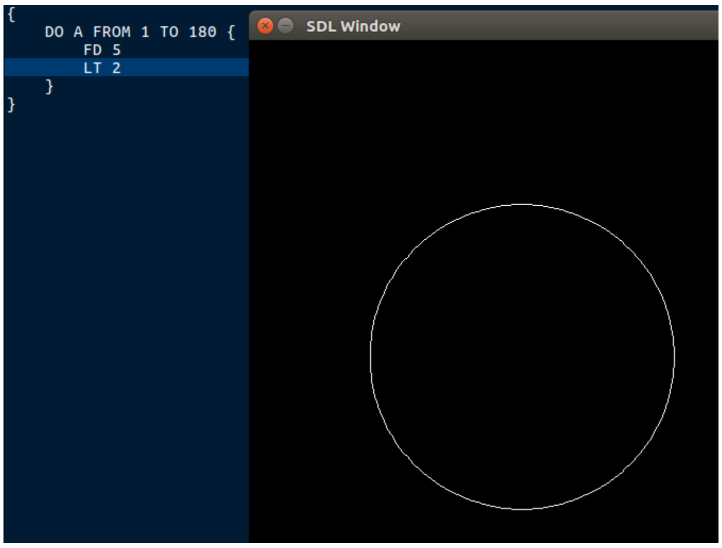
```
{
    DO A FROM 1 TO 50 {
        FD A
        RT 30
        DO B FROM 1 TO 8 {
            SET C := A 5 / ;
            FD C
            RT 45
        }
    }
}
```

Figure 4 - Nested DO Loop Example 1



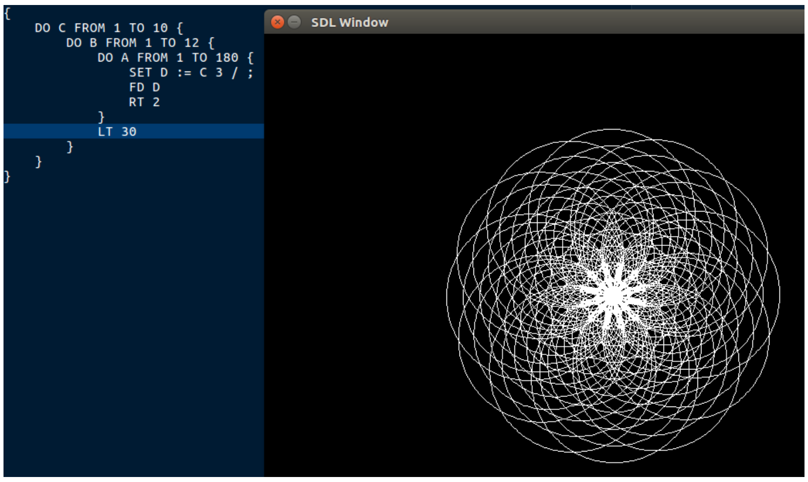
```
{
    RT 90
    DO B FROM 3 TO 12 {
        SET C := 360 B / ;
        DO A FROM 1 TO B {
            FD 75
            LT C
        }
    }
}
```

Figure 5 - Nested DO Loop Example 2



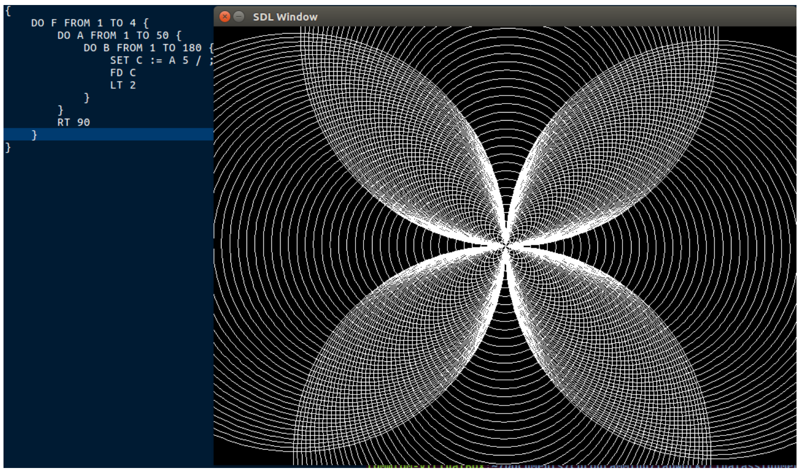
```
{
    DO A FROM 1 TO 180 {
        FD 5
        LT 2
    }
}
```

Figure 6 - Drawing a Circle



```
{
    DO C FROM 1 TO 10 {
        DO B FROM 1 TO 12 {
            DO A FROM 1 TO 180 {
                SET D := C 3 / ;
                FD D
                RT 2
            }
            LT 30
        }
    }
}
```

Figure 7 - Lots of Circles



```
{
    DO F FROM 1 TO 4 {
        DO A FROM 1 TO 50 {
            DO B FROM 1 TO 180 {
                SET C := A 5 / ;
                FD C
                LT 2
            }
        }
        RT 90
    }
}
```

Figure 8 - Even more circles