2020

Programming Languages Assignment Report

ALEX MCLEOD

STUDENT NUM: 19493178

Contents

Task 1 – Fortran

Code:

FizzBuzz.f:

```
  program FizzBuzz
c real values that determine whether there is a fizz or buzz
c when they equal 0 it means they occur, otherwise they do not
     real fizzCheck, buzzCheck

c this is a while loop that goes from 1 to 100
c   10 if (counter .LE. 100) then
     do 10 i = 1, 100
c compute the fizz and buzz check variables
c negate the integer division from the real division to detemine if
c there is a remainder, < 0 there is remainder, = 0 there isnt
        fizzCheck = dble(i)/3.0 - dble(i/3)
        buzzCheck = dble(i)/5.0 - dble(i/5)

c if both fizz and buzz check equal 0 then current number is a
c multiple of both 3 and 5, hence print FizzBuzz
        if((fizzCheck .EQ. 0.0) .AND. (buzzCheck .EQ. 0.0)) then
           write (*,*) 'FizzBuzz'
c if just fizz check is equal to 0 then print Fizz
        elseif (fizzCheck .EQ. 0.0) then
           write (*,*) 'Fizz'
c if just buzz check is equal to 0 then print Buzz
        elseif (buzzCheck .EQ. 0.0) then
           write (*,*) 'Buzz'
        else
c if both dont equal 0 then number isnt a multiple of either so print
c just print the number
           write (*,*) i
        endif

c increment the counter to go to the next number
c        counter = counter + 1
```

```
c go back to the start of the loop
c        goto 10
c     endif
  10 continue


     stop
     end
```

Testing

Test 1:

Input: No input into program as by default it is ran up to 100 numbers and you don't choose how far it runs up to. Hence should be the same input and output each run of the program.

Output: a number, Fizz, Buzz or FizzBuzz is displayed.

Evaluation: I evaluated the output by going through each line of output and figuring out myself whether each lines number is divisible by 3, 5, 3 and 5 or neither. When it was divisible by 3 I checked whether it displayed just Fizz, when it was divisible by 5 I made sure it displayed 5, when it was divisible by 3 and 5 I made sure it displayed FizzBuzz and lastly when it wasn't divisible by either number I made sure it displayed just the number itself.

Weekly Question:

How does Fortran compare to languages you have written in previously?

It is different to any language I have previously written in, particularly in its writability. It has very strict writing conventions where each column has a specific use, so you must make sure you write the correct code in the correct column. e.g. column 1 for starting a comment with c, 1-5 statement labels, 6 continuation of the previous line and 7 – 72 is for actual statements. Unlike many more modern languages that ive written in such as Java where there are standards as to how you should write code you dont have to follow it, such as having indenting in your code. Though enforcing these strict writing standards does make Fortran code more readable as all code no matter the writer must format there code in this specific way. It is also harder to write as it doesnt follow common conventions of many modern languages I am use to using for its relational operators and other operators. For its relational operators instead of using symbols such as <, <=, it uses letters such .LT. For < and .LE. For <=. It also lacks many common control structures that many languages I have used have such as while loops and it lacks operators such as the % modulus operator. Hence the lack of these make the language less writable.

Reflection:

One of the most apparent principles that f77 goes against is the simplicity principle. This was apparent when trying to figure out if a number was divisible by 3 or 5. F77 does not provide the modulus operator that returns the remainder when a number is divided by another number. The inclusion of this operator would have made completing this task much easier. Instead I had to perform more complex computations. This hence makes F77 much less writable as getting the final result took much longer compared to if I used any other language. These calculations also make the

code less readable because if another programmer was reading my code, they would be forced to figure out how the modulus operation is being calculated and what exactly is happening.

In terms of readability f77 is made easier to read through the use of indentation where each column of a file has a specific role. For example, column 1 is for comments, 1-5 is for statements labels, 6 for the continuation of the previous line, 7 – 72 for statements, etc. This hence follows the structured programming principle as the strict positioning rules make my program more readable for a user as they can easily identify different parts of the code. On the other hand, it did make my program harder to write as causing an error in the code was a lot easier due to the strict rules. It forced me to count the number of columns from the left each time I wanted to write something different such as labels and statements. F77 follows the structured programming principle with its inclusion of control structures such as the do-loop and if statements that make the program much easier to write and read. The do-loop that was included in particular made it easier to write as instead of having to use an if statement with a goto and counter variable I could easily just write a do statement that goes from 1 to 100. This also makes the program more readable as the user will easily associate a do statement to a do loop which is a common control structure in most programming languages. Making it easier for them to understand what is going on when compared to if an if statement and goto was used. Another aspect that shows that F77 follows the structured programming principle is the declaration of each variable that f77 forces you to have stated at the top of the program before any variables are initialised. This allows anyone reading the code to identify what data type each variable is easily.

When it comes to output in F77 the syntax used is strange and unlike any other language causing it to go against the regularity principle. Firstly, the word write is used when creating output for the user unlike many other languages that use the word print. The word print being universal word amongst most languages that most programmers will instantly understand as being output to the user. Hence the use of the word write could make the code less readable as the programmer has to figure out what write does and what it writes to. Secondly after the write statement you then need to include the symbols (*,*) before you specify the output you want to give. These two asterisks can be replaced by code to specify output options. This weird use of symbols could cause anyone reading the code to wonder what it means and becoming instantly confused as to what it does. It hence goes against the simplicity principle as it makes writing code more complex as each time you want to do basic output you have to remember to include these symbols unlike most languages where you can just print(<output>). This strange syntax is also seen with the comparison operators equal, greater than, less than etc. In this case instead of using the common ==, <, > symbols f77 uses a full stop followed by capital letters representing a particular comparison and then followed by another full stop. E.g. .EQ. is used to check if data is equal. Similar to the syntax used for output this goes against the regularity principle and affects readability as programmers who have used other languages will be use to the common symbols used and this syntax could make them very confused. This is also makes writing as a programmer harder as each time you want to do a comparison you have to look up the correct syntax. In my case I had to keep looking up which letters I needed to use for each comparison as they are all very different. The combination of the symbols: <, >, =, ! is much more logical and is used in most programming languages, the fact that it isn't used here makes reading and writing much harder. This strange and unnatural syntax also makes the languages itself more unreliable as the reading and writing that would have to occur for the maintenance of the code would be much harder. Programmers tasked with fixing a bug in the code would have a hard time modifying it with this unnatural syntax.'

When it comes to reliability, I also noticed that type checking does not occur all the time hence going against the regularity principle. F77 allows you to declare an integer and then assign a real value to it, then if you were to output it the decimal point and everything after it would be truncated. Though it does not let you assign a character value to a declared integer or real variable. Hence during my writing of my program, I had to make sure the appropriate data types were being used to avoid unexpected output. This in turn makes the language more unreliable as there is a higher chance something could go wrong.

Task 2 – Algol

Code:

FizzBuzz.a68:

```
BEGIN
  INT n := 100;


  # loop that goes from 0 to 100 #
  FOR i TO n DO


    IF ((i MOD 3) = 0 AND (i MOD 5) = 0) THEN
      print (("FizzBuzz", new line))
    ELIF ((i MOD 3) = 0) THEN
      printf($"Fizz"l$)
    ELIF ((i MOD 5) = 0) THEN
      printf($"Buzz"l$)
    ELSE
      # prints out current integer #
      # the procedure whole helps format the output. parameter 1 is the integer to #
      # output and the second parameter is the width of the output 0 meaning it    #
      # outputs in the first column. #
      print((whole(i, 0), new line))
    FI
  OD
END
```

Testing

Test 1:

Input: For this program there is no input. You just run the program and it performs the FizzBuzz algorithm up to 100

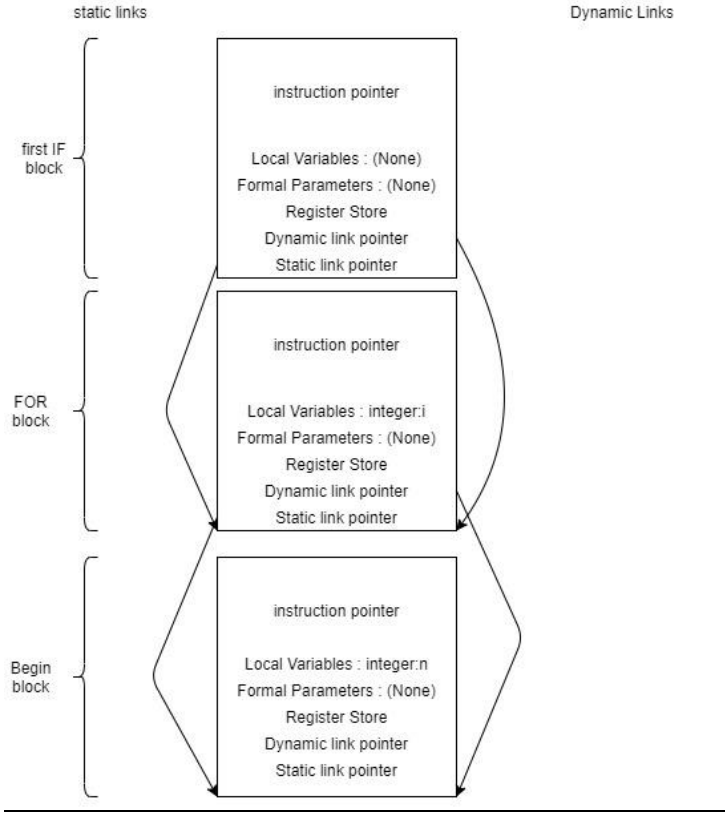
Output: After running the program it displays the numbers 1 through to 100 with each number on a new line. The numbers that are multiples of 3 are replaced with Fizz, the numbers that are multiples of 5 are replaced with buzz and numbers that are multiples of 3 and 5 are replaced with FizzBuzz.

Evaluation:

I evaluated my program by going through each number that is displayed up to 100. I check that each number which is divisible by 3 was replaced by the word fizz, each number divisible by 5 was replaced by Buzz and lastly that any number divisible by both 3 and 5 was replaced by FizzBuzz. Going through each one I was able to conclude that the program successfully worked with each line displaying the correct output.

Weekly Question:

Assume your program pauses when it prints out fizzbuzz. Draw the stack with all activation records, and both static and dynamic chains, at this point.



Reflection:

Whilst programming in algol68 I first noticed that it complies with the structured programming principle in many ways, making it easy to trace when looking at the source code itself. This is made apparent through the control structures that must be in capital letters making them easy to identify and allows the user to easily see where different blocks start and where they end, this also follows the regularity principle as this capitalization of control structures is followed throughout the language without exceptions. As well as this the actual block structure of the language follows the structured programming principle, making it easy to distinguish parts of the code, particularly where different control structures start end and the code associated with them. This structuring of the language hence makes the language very readable compared to older languages as the capitalization of keywords and block structure allows new programmers viewing the code to understand what the control structures are and where they are. The capitalization of all control structure keywords made

writing the code harder as missing one capital letter could cause the whole program to not work. With many languages not having this requirement, this was easy to do. The readability of the code caused by its structure makes the language more reliable wit

Whilst writing the code I also noticed that algol68 goes against the syntactic consistency principle. This being where similar looking things should do similar things and different things should look different. This is seen through the ‘:=’ and ‘=’ operators for assignment and comparison. Both these symbols are very similar with completely different uses. The ‘:=’ operator is used for assigning a value to a variable and the ‘=’ operator is a comparison operator that is used to determine whether two values are the same. This made the language less writable as it was easy to mistakenly use the comparison operator instead of the assignment operator when assigning values to variables. Many languages I am familiar with use the ‘=’ operator for assignment making the mistake even more common. This hence caused many errors to occur during compilation. This may also affect the readability of the code with programmers confusing the two operations.

I also found that the program goes against the regularity principle with the use of the semicolon being very inconsistent. It is used to separate different statements of code but at the end of a block you do not need to include it. This irregular use of the semicolon caused algol68 to be much harder to write in, with many errors occurring at compile time. It took quite some time to figure out why these errors were occurring, as with many programming languages I am use to you include a semicolon at the end of every statement and there aren’t statements where you don’t need to include it.

Another way in which I found the language follows the regularity principle is through the type checking that occurs with all types. Unlike Fortran-77 that I previously used if you were to accidently to assign a real value to a integer variable the error is caught at compile time. This hence Increasing the reliability of the program as any weird outcomes caused by no type checking are avoided with all variables being assigned their correct values.

Task 3 – Ada

My implementation of bubble sort allows a user to enter 5 numbers which is then sorted and the output to the user.

Code:

sort_package.adb:

```
with Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;


-- purpose: package holding sorting algorithm
package body Sort_Package is


   -- function that takes in an integer array and sorts it in ascending order
   -- using bubble sort
   procedure bubbleSort (Arr : in out Int_Array) is
   Swapped : Boolean;
   begin
      -- until loop that ends when no swaps occur
      Until_Loop2:
      loop
         Swapped := false;
         -- compare each value that is next to each other in the array
         -- if the right value is high then the left then swap them
         -- otherwise continue to the next to values
         -- only stop going through the array when no stops have occured
         -- on one pass
         for I in Length range 1 .. 4 loop
            if(Arr(I-1) > Arr(I)) then
               swap(Arr(I), Arr(I - 1));
               Swapped := true;
            end if;
         end loop;
         exit Until_Loop2 when (Swapped = false);
      end loop Until_Loop2;
   end BubbleSort;
```

```
   procedure swap (value1: in out Integer; value2: in out Integer) is
   Temp : Integer;
   begin
      Temp := value1;
      value1 := value2;
      value2 := Temp;
   end swap;
end Sort_Package;
```

sort_package.ads:

```
with Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;

-- packagae containing the sorting algorithm
package Sort_Package is

  -- types used for array that is sorted
  type Length is range 0 .. 4;
  type Int_Array is array (Length) of Integer;

  -- bubble sorter function
  procedure bubbleSort (Arr : in out Int_Array);

  -- swap procedure
  procedure swap (value1 : in out Integer; value2 : in out Integer);

end Sort_Package;
```

usebubblesort.adb:

```
with sort_package;
with Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;

-- types taken from sort_package to declare and array and
-- loop through its length
use type sort_package.Int_Array;
use type sort_package.Length;
```

```
-- purpose: procedure for allowing IO where user can enter numbers and they are displayed
--         in increasing order
procedure useBubbleSort is
   -- rename the TEXT_IO package to IO for ease of user
   package IO renames Ada.Text_IO;

   -- declare the integer array for storing values
   Arr : sort_package.Int_Array := (0, 0, 0, 0, 0);
   -- indicates whether an error is present
   error : Boolean;

   begin
      IO.Put_Line("Enter 5 numbers to sort");

      -- loop from 0 to 4 to get the integer from user input for the array
      for I in sort_package.Length range 0 .. 4 loop
      Until_Loop1 :
         -- while loop, that keep loop when input is incorrect
         loop
                  begin
                           error := false; -- false when no error
                     IO.Put_Line("Enter integer: ");
               Arr(I) := Integer'Value (IO.Get_Line);
            exception --catch exceptions with user input
               when Constraint_Error =>
                     IO.new_Line (1);
                     IO.Put_Line ("Error: input must be an integer data type");
                     IO.Put_Line ("       and the input length must be from -999999999 to 9999999999");
                           IO.Put_Line ("You must re-enter the number");
               error := true;
            end;
                  exit Until_Loop1 when error = false; -- only exit the loop when there hasnt been an error
               end loop Until_Loop1;
      end loop;
```

```
    -- input the array into the bubble sort function to sort the array in ascending order
    Sort_package.bubbleSort (Arr);


    -- output the resulting array
    IO.New_Line(1);
    IO.Put_Line("Numbers in ascending order:");
    for I in sort_package.Length loop
       IO.Put_Line(INTEGER'Image (Arr(I)));
    end loop;
 end useBubbleSort;
```

Testing

Test 1:

Input: user enters 5 integer values (values = 5, 1, 7, 2, 3)

Output: outputs 1, 2, 3, 5, 7 each number on a new line.

Evaluation: Bubble sort should sort the values in increasing order. Hence viewing the output and going through each number we can see they are in order and the output is correct.

Test 2:

Input: entering a real value first (value = 23.12)

Output: outputs the message “Error: input must be an integer data type and the input length must be from -999999999 to 9999999999” then new line, “You must re-enter the number”, and then a new line “Enter Integer:” where the user can re-enter another value.

Evaluation: My bubble sort only deals with integer values, so any real values that is entered should throw an exception. In this case the exception is caught and the user is told what is wrong and is then able to enter another value instead. Hence this is the correct output when a real is value is input.

Test 3:

Input: entering a string value first (value = “hello”)

Output: outputs the message “Error: input must be an integer data type and the input length must be from -999999999 to 9999999999” then new line, “You must re-enter the number”, and then a new line “Enter Integer:” where the user can re-enter another value.

Evaluation: My bubble sort only deals with integer values, so any real values that is entered should throw an exception. In this case the exception is caught and the user is told what is wrong and is then able to enter another value instead. Hence this is the correct output when a real is value is input.

Test 4:

Input: entering a large integer greater then 9999999999 (value = 99999999999)

Output: outputs the message “Error: input must be an integer data type and the input length must be from -999999999 to 9999999999” then new line, “You must re-enter the number”, and then a new line “Enter Integer:” where the user can re-enter another value.

Evaluation: My bubble sort algorithm only deals with integer values so anything that’s not between -999999999 and 9999999999 will throw an error message which is caught. Hence we can see that the test is successful.

Test 5:

Input: entering a small integer less then -999999999 (value = -9999999999)

Output: outputs the message “Error: input must be an integer data type and the input length must be from -999999999 to 9999999999” then new line, “You must re-enter the number”, and then a new line “Enter Integer:” where the user can re-enter another value.

Evaluation: My bubble sort algorithm only deals with integer values so anything that’s not between -999999999 and 9999999999 will throw an error message which is caught. Hence we can see that the test is successful

Test 6:

Input: entering nothing and pressing enter

Output: The same error message is displayed. outputs the message “Error: input must be an integer data type and the input length must be from -999999999 to 9999999999” then new line, “You must re-enter the number”, and then a new line “Enter Integer:” where the user can re-enter another value

Evaluation: When the user enters nothing I have chosen to throw an error message so they are forced to enter 5 integers. Hence this test is successful.

Test 7:

Input: entering two duplicate values 2 and 2 (input: 3, 6, 2, 1, 2,)

Output: 1, 2, 2, 3, 6

Evaluation: When two duplicate values are entered they should just be places next to each other. Hence the test is successful.

Weekly Question:

Compare your implementation of bubble sort with an implementation in C (you may have written one, if not, there are plenty on the internet you can look at). What similarities and differences are there?

C Algorithm from GeekForGeeks (https://www.geeksforgeeks.org/bubble-sort/)

One of the main difference from my ada implementation is the use of pointer variables in the C algorithm. This is seen when swapping two variables that the reference of the two variables is

imported into the swap function. They are then dereferenced and swapped. This is how it returns both new values to the array in their swapped order. In my ada program when calling my swap function instead of using reference to swap the two value instead in ada we use pass by value-result to swap the two integers. This is done by using the in out keywords for the parameters passed in. Then instead of using references new copies are made, changes are made and the resulting value are copied back into the variables that were passed in. By doing using this method it made passing parameters a lot easier to comprehend as there is no referencing and dereferencing of pointers that has to occur. There is also less risk of accidently breaking the program. Using combinations of in out shows how ada uses the orthogonality principle to combine keywords for different uses. This makes defining parameters and controlling them easier as we know exactly what they are doing either going in, out or in and out unlike c. Hence Ada is shown to be a lot more readable and writable in this case. In terms of similarities both programs split up the different algorithms such as IO, bubble sort and sorting into separate procedures and function adding to the modularity of both implementations. This makes them both easy to read as they are split up into their separate tasks. The only difference is that my program uses a separate package for my bubble sort while this c implementation is all in one file. By adding a separate packages this provides abstraction where users of the package don't need to know the inner workings of package and can just call it to produce a bubble sort given an array. It also provides more modularity as they are completely separated tasks in different files.

Reflection:

From my program I found that Ada follows the defence in depth principle by allowing for exception handling to occur. In this way it allows you to raise and catch exceptions. In my case I didn't raise an exception but in the case that a user enters a incorrect input a Constraint_Error is thrown which I then would catch In the useBubbleSort procedure. This deals with instances when user enters the wrong data type, doesn't enter anything or enters an integer that it too large. This system makes catching all the different errors a lot simpler rand easier than having to manually check the input for all these cases. It hence adds to the reliability of my program as there is much smaller chance that something goes wrong. Ada also follows the abstraction principle as I was able to create a separate package used for sorting the data. By creating this separate package allows for data abstraction where users of the package don't need to know how the sorting works but only that it sorts the data in ascending order. In this way from my main program that gets the input all I have to do is call the bubble sort function and import my array. This abstraction makes the program a lot more readable as everything is separated into their own files and procedures. When reading one thing you do not need to understand the inner workings of everything that is called. When writing it also makes thing a lot easier as you can create different procedures separately and call them in other procedures when needed without having to worry about how they work. As previously discussed Ada follow the orthogonality principle when it comes to parameter passing. It uses the in and out keywords in separately or in combination to describe how a parameter can be controlled. This provides reliability as it less likely for the parameters you import into a function to be used incorrectly and in turn create errors. By import using 'in' we can restrict use to write only, by using 'out' we can restrict to read only and then by using 'in' and 'out' we can give read and write permission. In my case I allowed for in and out functionality as the values need to read before changes are then made. This parameter passing technique also follows the simplicity principle as there is only smaller number of ways of combining in and out and the concept is very easy to follow in any situation you need to use them. This is in comparison to reference passing techniques using pointers where you have to ensure that you correctly reference and deference the variables you pass in and out of a function.

Task 4 – Bison and Flex

Code:

Sorter.l

```
%{
#include <stdio.h>
#include "y.tab.h"
%}

%%
\[          return OPENBRACKET;
\]          return CLOSEBRACKET;
\,          return COMMA;
[1-9]+              yylval.num = atoi(yytext); return NUMBER;
\n           /*ignore end of line*/;
[ \t]+              /*ignore whitespace*/;
%%
```

Sorter.y

```
%{

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <stdbool.h>
void yyerror (char * str);
int yylex();
int length();
void printList();
void bubbleSort();
void insertFirst(int key, int data);

//purpose: node for linked list
struct node {
```

```
    int integerVal;
    int countNum;
    struct node *next;
};

//purpose: pointers to the head of the linked list and the current node
struct node *head = NULL;
struct node *current = NULL;
int count = 0;



%}



%union {int num;}
%token OPENBRACKET
%token CLOSEBRACKET
%token COMMA
%token <num> NUMBER
%type <num> integer



%%
list:
            OPENBRACKET integer_list CLOSEBRACKET
      {
            bubbleSort();
            printList();
            return 0;
      }
      ;

integer_list:
             integer
             | integer COMMA integer_list
```

```
	;
integer:
		NUMBER
	{
			printf("number: %d\n", $1);
			$$ = $1;
		insertFirst(count, $1);
		count = count + 1;

	}
	;

%%

void yyerror(char * str)
{
	fprintf(stderr, "error: %s\n", str);
}

int yywrap()
{
	return 1;
}

//purpose: inserting a new element to the linked list
void insertFirst(int newCountNum, int newIntegerVal)
{
   struct node *link = (struct node*) malloc(sizeof(struct node));

   link->countNum = newCountNum;
   link->integerVal = newIntegerVal;

   link->next = head;

   head = link;
```

```
}

//purpose: gets the length of the linked list
int length() {
  int length = 0;
  struct node *current;

  for(current = head; current != NULL; current = current->next) {
    length++;
  }

  return length;
}

//prints outs everything in the list as well as free the allocated memory
void printList() {
  struct node *ptr = head;
  printf("\n[ ");

  //start from the beginning
  while(ptr != NULL) {
    printf("%d ", ptr->integerVal);
    ptr = ptr->next;
  }

  //free allocated memory of each node of the list
  ptr = head;
  while(ptr != NULL) {
    struct node *nd = ptr;
    ptr = ptr->next;
    free(nd);
  }

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

```
//purpose: sorts the numbers within the linked list using bubble sort
void bubbleSort()
{
  int i, j, k, tempNum, tempInteger;
  struct node *current;
  struct node *next;

  int n = length();

   for (i = 0; i < n; i++)
   {
      //start from the beginning of the list
      current = head;
      next = head->next;
      //as the list sorts reduce by one how many numbers must be sorted
      for (j = 0; j < n-i-1; j++)
      {
         if (current->integerVal > next->integerVal)
         {
            //swap the integer value of the two nodes
            tempInteger = current->integerVal;
            current->integerVal = next->integerVal;
            next->integerVal = tempInteger;

            //swap the count id of each node
            tempNum = current->countNum;
            current->countNum = next->countNum;
            next->countNum = tempNum;
         }
         current = current->next;
         next = next->next;
      }

   }
```

```
}
int main()
{
    return yyparse();
}
```

Testing

Test 1: set of number in unsorted order

Input: [3, 1, 5, 2]

Output: Value: [1 2 3 5)

Evaluation: As we can see the test was successful as the input numbers were output in sorted order starting with 1 and ending with 5.

Test 2: test empty list input

Input: []

Output: “error: syntax error”

Evaluation: As we can see the test was successful as the program responds correctly to an empty input stating that there was an error.

Test 3: test one number in the list

Input: [1]

Output: [1]

Evaluation: As we can see the test was successful as the program just outputs the single number in the list without trying to sort it. If there is only one number, then it is already sorted.

Test 4: numbers in descending order

Input: [5, 4, 3, 2, 1]

Output: [1 2 3 4 5]

Evaluation: As we can see the test was successful as the algorithm successfully reverses the output so it is in ascending order.

Test 5: list containing the same number twice

Input: [5, 2, 2, 1, 9, 3]

Output: [1 2 2 3 5 9]

Evaluation: As we can see the test was successful as the algorithm successfully outputs the list in sorted order placing the two repeated numbers next to each other.

Test 6: numbers in already sorted order

Input: [1, 2, 3, 4, 5]

Output: [1 2 3 4 5]

Evaluation: As we can see the test was successful as the algorithm successfully outputs the same input list as it is already sorted.

Test 7: incorrectly formatted input without commas

Input: [3 1 5 2]

Output: "error: syntax error"

Evaluation: As we can see the test was successful as the program outputs an error message, terminating the program when an incorrect input is entered

Test 8: incorrectly formatted input without square brackets

Input: 3, 1, 5, 2

Output: "error: syntax error"

Evaluation: As we can see the test was successful as the program outputs an error message, terminating the program when an incorrect input is entered.

Test 9: incorrect types used. Character entered into the list instead of a number

Input: [a, 1, 5, 2]

Output: "error: syntax error"

Evaluation: As we can see the test was successful as the program outputs an error message, terminating the program when an incorrect type is entered into the list.

Test 10: incorrect format with two commas used instead of 1

Input: [3,, 1, 5, 2]

Output: "error: syntax error"

Evaluation: As we can see the test was successful as the program outputs an error message, terminating the program when an incorrect format is entered.

Weekly Question:

If you were building a compiler, how do you think you would implement a symbol table?

I would start by using lex to create rules to define the different types for matching for example keywords such as double and int. I would also use lex to create rules to identify different types of scope, for example it will match keywords such as extern, global, for, etc. It will identify other common symbols that will be used to identify the type and scope of variables such as bracket, curly brackets etc. I will also identity any random assortment of character that doesn't match any other rule to be a name(could be a variable name or function name). Then using Yacc I will combine the different tokens identified by Lex into more rules. I will also use an array in Yacc to store a 2D array of the symbol information. For example if I were to find the global token followed by the double token and the name token I will take these tokens and store them in the 2-D dimensional array. The

first column in the array being for the name of the symbol which in this case would be the variable name, then the second column would hold the type, which in this case would be the type of the variable which is double and then the 3rd column would hold the scope which in this case would be the global keyword. Obviously creating a compiler to implement a symbol table would be lot more complex due to the larger number of possibilities that can occur and need to be considered.

Reflection:

I believe that both bison and flex follow the structured programming principle. This being because they use the '%%' to split up the different parts of code. For flex Before the first %% is the definitions, after the first %% is where matching rules occurs and after the second %% the subroutines are. By having this structure it improves the readability of both bison and flex as you can instantly tell where particular sections start and where they end. It also adds to the writability as deciding where to write different parts of code is easy due to this separation. Lastly this structure also adds to the reliability of the code as it is less likely for a programmer to accidently write something in the wrong section and cause an error to occur.

Bison and Flex uses C to perform subroutines. In my case I used C to perform sorting of the numbers, creation of a linked list to store the numbers and for printing out the numbers to console. Using C I found that it breaks the Simplicity principle this is mainly due to its use of pointers that can makes things highly complex and can lead to many problems such as memory leaks and dangling pointers if not implemented properly. In my case I used a number of pointers particular within my link list to point to the next node in the list. This high complexity of pointers makes c less writable in the sense that it can be hard to write code that works properly when using pointers. It is very easy for someone who is inexperienced with pointer to become confused as to why there code doesn't work. In my case I have not used c for a while so understanding how to use pointers to create a linked list was a but challenging. It can also lead to poor readability of code particularly when there is a large number of pointers as we must keep track of what they are pointing at. Further more pointers can lead to lessen the reliability of code as programmers are forced to manage their own memory leading to problems such as memory leaks when someone doesn't free allocated memory or dangling pointers when a pointer points at null can occur.

Task 5 – Bash, Perl, Ruby

Code:

Bash:

```
# move to the first directory of the system
cd ~/../..
# -name allows you to specify a name
# -f type specified that the search is for a file
find -name "*.conf" -type f
```

Perl

```
#!usr/bin/perl

use File::Find;

# use the find module to cycle through all files and directories
# imports a code reference checkFile to do verification on each file
# parameter two being the starting directory
find(\&checkFile, "/");

# subroutine that imports a file and outputs a file if its a .conf
sub checkFile
{
   # get the imported filename and assign to $file
   my $file = $_;

   # use a regex to see if $file matches the regex expression
   # pattern to match is contained within /     /
   # \. specifies the . character
   # anchor $ matches the end of the string
   # hence end of the string must match .conf
   if($file =~ /\.conf$/)
   {
            # if we have a match print the filename, followed by a new line
            #period concatenates the file and its path to the new line character
      print $File::Find::name."\n";
```

```
    }
    #print $File::Find::name."\n" if -f $file;

    #print $File::Find::name."\n" if ($file =~ /\.conf$/i );

}
```

Ruby:

```
#!/usr/bin/ruby

require 'find'

# initialise a new empty array
pdf_file_paths = []

# use the find module to go through each path in the system
# loop through each path starting at the first directory of the computer
Find.find('/') do |path|
  # if the file at the particular path has .conf at the end of it
  # add to the list of paths. This is done by checking if it matches
  # the regex defined between /      /
  pdf_file_paths.append(path) if path =~ /.*\.conf$/
end

Find.find('/home') do |path|
  # if the file at the particular path has .conf at the end of it
  # add to the list of paths. This is done by checking if it matches
  # the regex defined between /      /
  pdf_file_paths.append(path) if path =~ /.*\.conf$/
end

# intialise a counter to to cycle through each path in the list
$counter = 0
# while there are still more paths in the list keeping printing them out
while $counter < pdf_file_paths.length do
  # output the path at the particular counter index in the array
```

```
 print pdf_file_paths[$counter] + "\n"
 # increment the counter
 $counter +=  1
end
```

Testing

Test 1: placing a .conf file in the home directory and running each script program

Input: no input just involves running each script

Output

… (other .conf files)

/home/alex/myfile.conf (ruby and perl)

./home/alex/myfile.conf (Bash)

Evaluation: Hence the test is successful as it is able to locate and output the .conf file that I placed in the home directory

Test 2: testing whether each script can locate a .conf file within a directory. Hence a file named random.conf is placed within a directory "random" which is in the home directory

Input: no input just involves running each script

Output

… (other .conf files)

/home/alex/random/random.conf (ruby and perl)

./home/alex/random/random.conf (Bash)

Evaluation: Hence the test is successful as it can locate and output the .conf file that I placed in the random directory

Test3: testing whether the scripts successfully don't detect a file containing an extension with a conf substring within it. E.g. random.confer. I placed a file named random.confer within the home directory.

Input: no input just involves running each script

Output

… (other .conf files)

Evaluation: I tested to check whether the random.confer file was detected by piping the output to a file then using ctrl-f to search for random.confer. It was not found, hence the test was successful.

Weekly Question:

Which of these languages was hardest to write this program in, and why?

In my opinion I found that Perl was the hardest to write the program in. This was mainly because when trying to research the different available modules in perl that could help in solving this problem there was so many sources of information that figuring out the most optimal way that I would understand was hard. I ended up using the module find which took me a while to understand as it was unlike the find modules in ruby or bash. This module did not simply take in one input and give a return instead it took a subroutine as input along with the path of the starting directory. I input a subroutine that I created called checkFile. Now each time the find module found a file it would call the checkfile subroutine to see if it was a .conf file, printing the files path if it was. This was unlike bash was extremely easy to write with just two lines of code needed. It was also easier in Ruby to solve the problem as its find function was a lot more logical and straight forward compared to perl as you didn't have to input a subroutine into the function. Instead Ruby's Find.find function simply takes a path as input and then performs a top-down traversal of all the file paths beneath it returning each one which allows you to check if they are a .conf file. Therefore perl was a lot harder to write due to the amount of time it took me to figure out what to use and the amount of time it took me to figure out how its inbuild find function works.

Reflection:

From what I wrote I found that bash follows the simplicity principle. This was made apparent due to the small amount of code that was need to perform what might be a somewhat complex task. All I needed was two lines of code. One that would to the root directory and another that uses the find function provided by bash to begin traversing down the different directories to find the .conf files and print them. This simplicity hence shows how bash is very writable as it only takes a small amount of time to solve a problem. I believe what I have written is easy to read, but for more complex problems this simplicity could affect how easy the bash code is to read with really complex things being done in a few lines, causing people who are reading the code to wonder how it works.

Whilst writing my Ruby program I found that it follows the structured programming principle. This is mainly due to the block structure that it follows. This is seen through the do loop that starts with 'do' and ends with 'end' clearly showing where the block starts and ends. Also the while loop start with 'while' and ends with 'end'. This block structure clearly separates the different scopes and increases the readability of the language. It Allows the user to easily track how the program will execute and which variables can be used by different scopes. It also increases readability by forcing the user to indent the code and separate the different sections so they can clearly be understood. I believe that this in turn also increases the writability of the code as whilst someone is writing their code it is easy to keep track of the different scopes and use the variables correctly to complete their task. It also provides reliability as a program that is well structured and easy to read leads to less errors occurring when programming. Whilst writing my program I also found that Ruby breaks the simplicity principle. This was made apparent when writing my for loop as I found two ways of writing it which would give the same result. By using the for keyword. For example for i in 0..3 this would go through 0 to 3 placing each value in i. Or by just using the do keyword. For example (0..3).each do |i| which would also go through 0 to 3 place each value in i. The fact that there are multiple ways of doing one thing makes Ruby less readable. This is because someone may get confused when reading someone else's Ruby code as their way of doing a for loop may be different leading to confusion. They will be forced to learn these two different ways making the task of reading the code harder. This also lessens the reliability of ruby code as code that is harder to read is harder to maintain when errors occur.

When writing my Perl program I definitely found that Perl doesn't follow the simplicity principle. This being due to the many possible ways that tasks can be completed. For example after writing my

code for checking if a file is .conf I found that I could reduce the last four lines of code into one statement. E.g. *print $File::Find::name."\n" if ($file =~ /\.conf$/i);.* Hence Perl also allows you include if conditional statements within your print statements. This being different to my code where I checked the file to see if it's a .conf before printing. This ability to do particular tasks in different ways causes the code to become more writable as there are so many ways of doing things. In another sense it makes it less writable because for someone who wants to write the most optimal code with so many options it can be hard to choose what way to complete a task. It also makes the code less readable as one person's way of doing things may be completely different to someone else's. With the many ways Perl allows you to complete task it can make it almost impossible to be able to read every person's Perl code without doing research on its capabilities and including commenting.

Whilst writing my program I also found that Ruby and Perl do not follow defence in depth as they both have implicit variable declarations rather than explicit variable declarations. Because of there is no explicit variable declaration a layer of defence is removed as variables can be accidently given the wrong data type during assignment leading to unexpected results in the program and possible errors. This hences reduce the reliability of these two languages as the possibility of errors occurring or something going wrong with the program when it comes to variable assignment increases. Not having explicit declarations of variables also makes these languages less readable as it is not clear what type each variable is leading to possible confusion.

Task 6 – Smalltalk

Code:

```
count := 1.0. "start count from 1"

[count <= 100] whileTrue:[ "keep looping until all 100 elements are printed"

  (count \\ 3.0 = 0.0)

    ifTrue: [

        (count \\ 5.0 = 0.0)

          ifTrue: [ 'fizzbuzz' printNl ] "if divisible by 3 and 5 then output FizzBuzz"

          ifFalse: [ 'fizz' printNl ]. "if divisible only by 3 print fizz"

        ]

    ifFalse: [

         (count \\ 5.0 = 0.0)

           ifTrue: [ 'buzz' printNl ] "if divisible by 5 print buzz"

           ifFalse: [ count printNl ]. "if not divisible by 3 or 5 print the number"

    ].

  count := count + 1.0. "increment to get next value"

  ].
```

Testing

Test 1:

Input: No input into program as by default it is ran up to 100 numbers and you don't choose how far it runs up to. Hence should be the same input and output each run of the program.

Output: : Going from 1 to 100 each number is printed on a new line. If the number is divisible by 3 'fizz' replaces the number, if the number is divisible by 5 'buzz' replaces the number and lastly if the number is divisible by 3 and 5 'FizzBuzz' replaces the number.

Console output:

1.0

2.0

'fizz'

4.0

'buzz'

'fizz'

7.0

8.0

'fizz'

‘buzz’

11.0

‘fizz’

13.0

14.0

‘fizzbuzz’

Etc…

Evaluation: I evaluated the output by going through each line of output checking whether 100 items are printed. Making sure that numbers divisible by 3, 5 or both are replaced with the correct words. 3 being fizz, 5 being buzz and 3 and 5 being fizzbuzz.

Weekly Question:

How does your implementation of fizz buzz here compare with the implementations you wrote in Fortran and Algol in terms of readability and writability?

In comparison to Fortran I found that smalltalk much more writable. This was mainly because of the \\ operator which performs modulus division. Allowing us to determine whether a number is divisible by 3 or 5. Returning 0 if the number is divisible and another number it is not. This is similar to Algol which I found was also more writable than Fortran as has the MOD operator allowing modulus division to be performed. Fortran’s F77 does not have this operator which forced me to computer more complex calculations making it less writable and harder to get to the final result.

Something which made smalltalk less writable compared to Fortran and Algol was the fact that it didn’t have the if-else control structure. You can only have one condition and choose what to do when that it true and when that is false. You cannot have an else-if to define multiple conditions. This hence forced me to have nested conditionals to do more checks when something is true or false. This made things less writable because as a programmer I am use to having an if-else control structure to perform simple control like this. This also made the program less readable as other programmers are also use to having an if-else control and not having this makes figuring out the different cases more complex as they aren’t written one after the other but instead nested within each other. Fortran’s F77 and Algol a68 on the other hand does have this making it more writable and readable as a programmer can easily write out the different cases and read them one after the other. Similarly, all three languages do have loop control structures which makes looping through the numbers 1 to 100 easy improving their writability. It also improves readability as the programmer can easily identify this control structure and understand what it is doing.

Reflection:

To begin with I found that GNU Smalltalk follows the syntactic consistency principle this is evident through the difference between the symbols used for the assignment operator ‘:=’ and the equality operator ‘=’. This ensures that any assignment can be easily identified and aren’t confused with an equality operator. This hence adds to the readability of the language as different symbols are easily distinguishable. In my code I can easily see when I am assigning a value to the count variable and when im checking its equality with other valuers. It also adds to the writability of the language as its

harder to accidently do an equality instead of an assignment and vice versa as the symbols used are so different. This hence adds to the reliability of the language as it is less likely that errors will occur in code when it comes to assignment and equality checking. Something being easier to write results in less errors occurs.

I believe that it follows the simplicity principle as it does not have some complex control structures. This was made apparent when writing my fizzbuzz code as there was no if-else statement available to use within SmallTalk. This made writing conditional statements harder as I could not have multiple conditions following each other. Instead I had to use Smalltalk's ifTrue and ifFalse and then nest more ifTrue's and ifFalses's within them to have multiple conditional statements. As previously discussed in the weekly question this made Smalltalk less writable as I wasn't able to simply use an if-else structure like most procedural languages but instead had to use nested conditions. This also makes the it break the structured programming principle as it becomes harder to track the different paths that the program can take during execution. In my case it isn't that hard as there are only a few possible scenarios but when there is a large number of possibilities having a lot of nested conditional statement can make tracking the execution from the source code hard. This will hence reduce the readability of the language. This can also make the program less reliable when there are a lot of nested conditional statements. Keeping track of these will be hard resulting in unwanted results or errors occurring.

Whilst writing my program I found that GNU SmallTalk breaks the regularity principle. This was made apparent when terminating statements using the period (“.”). At the end of the program I found that the last statement can either end with a period or not. I also found that when writing an ifTrue and ifFalse conditional statement I had to include a period after the ifFalse block statement but not after the ifTrue block statement. This made writing the FizzBuzz a lot harder due to the irregular use of the period symbol. Unlike in other languages where the semicolon is used after every statement to state when it ends smalltalk is very unpredictable. I had a lot of errors occurring in my code where periods were placed in the wrong spot, figuring out where exactly these periods were supposed to go was difficult. This makes Smalltalk also less readable as someone might read someone code who uses a semicolon for their last statement and someone else who doesn't leaving them confused. This hence also makes the language less reliable as it results in more errors occurring in code where a programmer has accidently placed the period symbol in the wrong place in code.

Task 7 – C++

Code that sorts Book objects based on id number

Code:

Book.cpp:

```
#include <string>
#include <cstdlib>
#include <iostream>

#include "Book.h"

Book::Book()
{
   bookID = 0;
   bookName = "default book";
   ISBN =  "0";
}

int Book::GetBookID()
{
   return bookID;
}

std::string Book::GetBookName()
{
   return bookName;
}

std::string Book::GetISBN()
{
   return ISBN;
}

void Book::SetBookID(int newID)
{
   bookID = newID;
```

```
}

void Book::SetBookName(std::string newName)
{
    bookName = newName;
}

void Book::SetBookISBN(std::string newISBN)
{
    ISBN = newISBN;
}

Book::~Book()
{

}
```

Book.h:

```
#pragma once

class Book
{
private:
    int bookID;
    std::string bookName;
    std::string ISBN;
public:
    int GetBookID();
    std::string GetBookName();
    std::string GetISBN();

    void SetBookID(int);
    void SetBookName(std::string);
    void SetBookISBN(std::string);

    Book();
```

```
    ~Book();
};
```

Program.cpp:

```
#include <iostream>
#include <regex>
#include <fstream>
#include <stdio.h>
#include <sstream>
#include "Book.h"

// purpose: swaps two elements
void swap(Book* a, Book* b)
{
    Book t = *a;
    *a = *b;
    *b = t;
}

// reference: C++ Program for QuickSort. (2014, January 7). GeeksforGeeks. https://www.geeksforgeeks.org/cpp-program-for-quicksort/
/* purpose: This function makes the right most element at the index 'lastIndex' the pivot
 * It then sorts the elements so that everything less then the pivot is on its
 * left and everything greater then the pivot is on the right                */
int partition (Book arr[], int firstIndex, int lastIndex)
{
    Book pivot = arr[lastIndex];//set the pivot
    int i = (firstIndex - 1);  // Index of smaller element

    for (int j = firstIndex; j <= lastIndex - 1; j++)
    {
        // If current element j is smaller than or equal ot pivot perform a swap with
            // i++ element
        if (arr[j].GetBookID() <= pivot.GetBookID())
        {
            i++;    // increment index of smaller element
```

```
            swap(&arr[i], &arr[j]);
        }
    }
    //swap pivot so everything on the left is less then it and everything on the right is greater
    swap(&arr[i + 1], &arr[lastIndex]);
    return (i + 1);
}

// reference: C++ Program for QuickSort. (2014, January 7). GeeksforGeeks. https://www.geeksforgeeks.org/cpp-program-for-quicksort/
/* purpose: The function that implements QuickSort
   arr[]: array of books to be sorted,
   firstIndex: Starting index,
   lastIndex: Ending index */
void quickSort(Book arr[], int firstIndex, int lastIndex)
{
    //once all elements have been sorted stop recursing
    if (firstIndex < lastIndex)
    {
        // sets the index partioning two sections
        int pi = partition(arr, firstIndex, lastIndex);

        // sorts elements before partition
        quickSort(arr, firstIndex, pi - 1);
                // sorts elements after the partition
        quickSort(arr, pi + 1, lastIndex);
    }
}

//purpose: prints out book information, from an array of Book objects
void printElements(Book arr[], int length)
{
    int i;
    std::cout << "Books sorted using quicksort based on ID:\n";
    //for each book prints out its information, starting fromt he first index of the array
```

```
    for (i = 0; i < length; i++)
    {
        std::cout << "book ID: " + std::to_string(arr[i].GetBookID()) << std::endl;
        std::cout << "book Name: " + arr[i].GetBookName() << std::endl;
        std::cout << "book ISBN: " + arr[i].GetISBN() << std::endl;
        std::cout << "\n";
    }
}

//purpose: this is where the program starts and the file is input by the user
int main()
{
    Book * arr;
    int numBooks;
    std::string line;
    std::string filename;
    bool error;

    error = false;
    //do while used so that if a user enters an incorrect file they can try again
    do
    {
            //try block used to throw any errors that occur when trying to open or read the file
        try
        {
            error = false;
                std::cout << "Type the name of the text file containing book information: ";
                std::cin >> filename; //user input filename is assigned to string
            numBooks = 0;
            std::ifstream myfile (filename);

                //regex used to check the format of the input file is correct
            std::regex formatChecker("^[0-9]+,[A-Za-z0-9\\s]+,[0-9]+$");

                //try opening the file is it exists
```

```
   if (myfile.is_open())
   {
      //retrieve each line
      while(getline(myfile, line))
      {
         numBooks += 1;//count the number of books in the file
                        if(!std::regex_match(line, formatChecker))//of incorrectly formatted throw
                                                                  //error message
         {
                                 myfile.close();
            throw "Error: input file is incorrectly formatted";
                  }
      }


                     //if no books are found throw an error message
      if(numBooks == 0)
               {
                  throw "Error: the file is empty";
      }
      myfile.close();
   }
   else//if file could not be opened then throw and error message
   {
      throw "Error: file could not be opened";
   }
}
catch(const char* errorMsg)//catch any error messages that occur and display them to user
{
   std::cerr << errorMsg << std::endl;
   error = true;//set error to true so that the loop is repeated
}
        catch(...)//catch any other errors that might occur
{
            std::cerr << "Error: something when wrong when opening the file" << std::endl;
         }
```

```
}
while(error == true);



arr = new Book [numBooks];
std::ifstream myfile2 (filename);
//open the file again to extract book information
if (myfile2.is_open())
{
   int i = 0;
   while(getline(myfile2, line))//go line by line to get book information
   {
      std::stringstream s_stream(line);


      std::string substr;
      getline(s_stream, substr, ','); //get first string delimited by comma
      arr[i].SetBookID(std::stoi(substr));


      getline(s_stream, substr, ',');//get next string after first comma
      arr[i].SetBookName(substr);


      getline(s_stream, substr, ',');//get next string after second comma
      arr[i].SetBookISBN(substr);


      i += 1;


   }
   myfile2.close();
}


quickSort(arr, 0, numBooks - 1);//sort the array of books using quicksort
printElements(arr, numBooks);//output the sorted books to console


delete[] arr;
return 0;
```

}

Testing:

For my program the information of each book is contained within a text file called 'BookInformation', When the program is run the file name must be entered. The user can also create their own file if they want containing their own book information but it must be formatted in the correct way.

Test 1: sorting books from file "BookInfo.txt"

Input: file name "BookInfo.txt" contains correctly formatted book information. Each line first containing the book ID (integer), then the book name (String), then the ISBN (integer). Each of these is separated by a comma.

Output: program successfully outputs the information of each book, sorted in ascending order based on the book ID.

Evaluation: I evaluated this test by going through each output book ensuring that the book ID's going up in ascending order. I also made sure that every book is displayed.

Test 2: empty file is entered

Input: file name "Empty.txt" is input which contains no text.

Output: program successfully outputs the error message, "Error: the file is empty".

Evaluation: I evaluated this test by ensuring that the correct error output was displayed for this particular error.

Test 3: file containing books is entered but the ID's contain letters

Input: file name "IDTest.txt" contains book information but the ID's contain letters

Output: program successfully outputs the error message, "Error: input file is incorrectly formatted".

Evaluation: I evaluated this test by ensuring that the correct error output was displayed for this particular error.

Test 4: file containing books is entered but the ISBN's contain letters

Input: file name "ISBNTest.txt" contains book information but the ID's contain letters.

Output: program successfully outputs the error message, "Error: input file is incorrectly formatted".

Evaluation: I evaluated this test by ensuring that the correct error output was displayed for this particular error.

Test 5: missing field test

Input: file name "MissingFieldTest.txt" contains book information but there are missing fields for some books.

Output: program successfully outputs the error message, "Error: input file is incorrectly formatted".

Evaluation: I evaluated this test by ensuring that the correct output was displayed for this particular error.

Test 6: the name of a non-existent file is entered

Input: file name “random.txt” is entered which does not exist.

Output: program successfully outputs the error message, “Error: file could not be opened”.

Evaluation: I evaluated this test by ensuring that the correct output was displayed for this particular error.

Weekly Question:

Compare how you have implemented objects in C++ with how you implement them in Java. What does this tell you about how objects work in C++?

I think that the implementation of objects in C++ is very similar to how I would implement an object in java. This is because when writing my Book class I had methods that I would usually include when creating a Java class. Such as the constructor methods, getters and setter which I would also use when writing a Java class. This is hence shows how objects work in somewhat the same way in C++ as Java as they both allow you to construct and manipulate the objects in the same way. One main difference would be the inclusion of a destructor method when writing my C++ class which is invoked when the object is deleted. This mainly is used to deallocate any dynamically allocated memory in the class. This is unlike Java where garbage collection is automatic and there is no use of a destructor. Hence it shows how C++ objects and C++ in general when compared to java allows you to have more control of memory management as you are the one who allocates and deallocates the memory.

Another main difference I found when implementing objects in C++ was that when creating my C++ class I had to split the class up into a .h file for the classes definitions and another .cpp file containing the code for the class. This shows how unlike Java C++ classes are thought to be data structs with pointers to a shared group of behaviours. Hence you must declare this struct and then separately implement the behaviours. This is different to Java where you have all the code for you class in one file containing both the definitions and the behaviours following theses definitions. The behaviours and definitions are not separated.

Reflection:

Whilst writing my program I found that C++ follows the structured programming principle due to its block structure with each method being separated by curly braces. This separates the scope of each block of code from each other, where the static structure of the program allows you to easily understand the dynamic structure of the program and how it will carry out upon execution. This in turn makes the program readable as you can easily see where the different sections of code are and the scope in which different variables relate to. It makes the code more writable as the programmer can easily keep track of the different scopes and write their program without getting confused with what relates to what.

I found that C++ does break the simplicity principle. This is mainly due to large number of things a programmer can do with C++. Making C++ extremely powerful but hard to master and understand everything about. One of example of this is seen with my program is how C++ combines both

modular programming and object oriented programming. It allows you to create modules contain algorithms which you can use throughout your program whilst also allowing you to design your own classes and create your own objects which can work with these modules. In my case I creates modules to perform the quicksort of my own Book objects. The large number of things that C++ allows you to do can make it writable in the sense that a user can do so many things if they have knowledge of C++. Someone who is new to coding or writing in C++ may have issues writing in C++ as there are so many concepts you must understand. The large number of possible things a programmer can do in C++ makes it less readable because even if you have knowledge of C++ when trying to read someone's code they may use things you have never seen before or do something in a completely different way to what you may have done. Hence reading someone's code may take a longer amount of time.

Whilst writing my C++ program I found that C++ it also follows the regularity principle as it requires you to always declare what type each variable is. This in turn makes C++ strongly typed as it checks that what you have declared your variables as matches what has been assigned to it. This in turn helps with the reliability of the program as C++ will always checks that variables have the correct type and it ensure that errors will occur when you accidently assign the wrong type to a variable. This makes unexpected problems due to the wrong type being assigned to a variable to not occur. Whilst writing my program I also found that C++ follows the defence in depth principle as it has exception handling. I used this mainly to check that the input of the books from file were correctly formatted. In the case that they weren't I threw an error and dealt with it by asking the user to re-enter the name of a different file. Like Java C++ has the try block to specify where the error might occur and the catch block which catches any exceptions that might be thrown in the try block. I then used the throw keyword to throw exception within the try block when the file was formatted incorrectly. This hence also provides reliability to C++ as exceptions can be handled and it stops the program from unexpectedly crashing if something were to go wrong. Instead we can deal with these exceptions appropriately, providing a clear message to the user as to what might have gone wrong.

Task 8 – Prolog

Program that prints out FizzBuzz from 0 to 1000

Code:

```
/*purpose: rule that checks whether X is divisble by 3
 *       if true then Fizz it output to console      */
fizzbuzzCheck(X) :-
   Y is X//3, X =:= 3 * Y, Z is X//5, \+ (X =:= 5 * Z),
   write('Fizz'), nl.

/*purpose: rule that checks whether X is divisble by 5
 *       if true then Buzz is output to the console  */
fizzbuzzCheck(X) :-
   Y is X//5, X =:= 5 * Y, Z is X//3, \+ (X =:= 3 * Z),
   write('Buzz'), nl.

/*purpose: rule that checks whether X is divisible by 3 and 5
 *       if true then FizzBuzz is printed to the console    */
fizzbuzzCheck(X) :-
   Y is X//3, X =:= 3 * Y, Z is X//5, X =:= 5 * Z,
   write('FizzBuzz'), nl.

/*purpose: rule that checks whether X isn't divisible by 3 or 5
 *       if true then the current number is just printed     */
fizzbuzzCheck(X) :-
   Y is X//3, \+ (X =:= 3 * Y), Z is X//5, \+ (X =:= 5 * Z),
   format('~d ~n', [X]).

/*purpose: this rule checks the current number for fizz and buzz, it then
 *       increments it and if this number is less then or equal to 1000
 *       increment is called again on the new incremented number. Otherwise
 *       it does not recall increment.                        */
increment(X) :-
   fizzbuzzCheck(X),
   succ(X, Y),
   Y =< 1000,
```

```
    increment(Y).


/*purpose: this is the starting rule that begins the incrementation at 1*/
startFizzBuzz :-
    increment(1).
```

Testing

Test 1:

Input: No input into program as by default it is ran up to 1000 numbers and you don't choose how far it runs up to. Hence should be the same input and output each run of the program.

Output: : Going from 1 to 1000 each number is printed on a new line. If the number is divisible by 3 'fizz' replaces the number, if the number is divisible by 5 'buzz' replaces the number and lastly if the number is divisible by 3 and 5 'FizzBuzz' replaces the number.

Evaluation: I evaluated the output by going through each line of output checking whether 1000 items are printed. Making sure that numbers divisible by 3, 5 or both are replaced with the correct words. 3 being fizz, 5 being buzz and 3 and 5 being fizzbuzz.

Weekly Question:

Was fizz buzz easier or harder to implement in Prolog? Why or why not?

In my opinion fizz buzz was a lot harder to implement in Prolog when compared to other languages such as Fortran, Algol, or any other procedural language. This is because the majority of languages used in todays world are procedural, so I have only learnt procedural languages and haven't ever used a logic-based language. This made writing code in this language harder due to the logic that must be used to solve the problems. It does not have traditional control structures such as a for loop or if statement like the procedural languages I am use to. Using these control structures makes writing fizz buzz very easy as you can simply have one loop that goes from 1 to 1000 and within that loop have multiple if-else statements that check each number for their divisibility so that the appropriate output is printed. Instead using Prolog I had to recursively call an increment rule inorder to increment up to 1000 then use multiple rules to check the divisibility of each number. These rules must check for different things so that none of them can be true at the same time. E.g. for a number like 15 it is divisible by 3 and 5. So we do not want 'Fizz', 'Buzz' and 'FizzBuzz' outputted. So we have 4 rules one that checks if something is just divisible by 3 and not 5, another that check whether something is just divisible by 5 and not 3, another that check if something is divisible by 3 and 5, and lastly one that checks if something is not divisible by 3 and 5. This ensure that none of these rules overlap and that there is only one output to the console for each number. In a procedural language doing this checking is much easier with if-else statements. Using an if-else we can first check whether something is divisible by both 3 and 5 and if they are skip all the other checks unlike prolog where each rule is checked for each number. If it is not divisible by 3 and 5 then an elseif statements can be used to check if they are just divisible by 3 or 5. The last else statement is also very useful in procedural languages as we can just say if it doesn't match any of the previous if statements do something else. Which in our case is to just print the number. Unlike in prolog where we need a separate rule to check whether it is not divisible by 3 or 5. Therefore prolog in my opinion is a lot

harder to write fizz buzz in due to its lack of control structures that we are use to in most procedural languages and the different logic that is used.

Reflection:

While writing my fizz buzz program I found that prolog does not follow the structured programming principle. This is mainly because it does not have any control structures which allow us to perform checks using if statements sequentially. This made writing the fizzbuzz program a lot harder than any other language as we can not use if-else statements when doing checks but instead have to do implicit checks using rule statements. This also makes reading my code a lot harder for others as checks are separated into different rules which all have the same name. This makes it hard to understand what each rule does without any comments that explain what they do. This is unlike procedural languages with if-else statements where the checks take place one after the other and the user can easily understand what is going on. Most programmers in particular are familiar with if-else statements as most procedural languages that are popular use them.

While writing my program I found that Prolog follows the simplicity principle. This is because it only has two types of statements rules and facts and nothing else. This made writing my program a bit easier as there's only a few concepts to be understand in order to start writing the program. I only had to do a bit of research to know the different concepts and combine them for my application. My program was made up of a combination of these rules. I believe it also follows the syntactic consistency principle as it separates the symbol used for the unification of two terms using the '=' from the symbol used to check for the equality of two symbols which is '=:='. This makes the code a lot more readable as the user can easily tell when a unification or a comparison is taking place as the symbols are so different.

Whilst writing my program I also found that Prolog follows the defence in depth principle, this is because it has default exceptions and allows you to handle these exceptions. Though I didn't have to handle these exceptions in my program as it is a very simple program, the existence of these exception types made dealing with problems in my code easy to solve. For example it has the type_error exception with allows me to easily realise problems with using the wrong type when writing a query. This in turn increases the reliability of the program allowing the user to easily identify and handle the different exceptions that might occur in their program. This reduces the likelihood of bugs occurring in the final code as most exceptions that are possible can be caught.

Task 9 – Scheme

A program that uses a cocktail shaker sorting algorithm to sort a list of numbers

Code:

```
; Author : Alex McLeod

; Purpose : performs cocktail sort on a list of items, algorithm alternates between cocktail-sort-right and cocktail-sort-left, performing bubble sort from

;         left to right and then right to left.



;purpose: swaps the first two elements of a list

(define swap (lambda (x)

                (reverse (list (car x) (cadr x)))))


;purpose: swaps first two elements of list, removes the first element from list and returns the rest of the list

(define next (lambda (x)

                (cdr (append (swap x) (cddr x)))))


;purpose: swaps first two elements of input list, saves first element of list by adding to list y and returns list y

(define save (lambda (x y)

                (append y (list (car (swap x))))))


;purpose: performs bubble-sort from right to left of list, first occurence occurs after cocktail-sort-right

;x : current list of element to perform bubble sort, first item is continually removed as functions recurses

;y : used to save the list of elements that have already been traversed, one item is added to list as function recurses

;s : used to hold the number of swaps that have occurred during one recurse from the left to right

;l : used to hold the length of the initial input list

(define cocktail-sort-left (lambda (x y s l)

                (if (= 1 l) ; if there is one item left in the list we have finished right-left traversal, so check num swaps

                  (if (= 0 s) ; if num swaps is 0 list has been sorted so output otherwise bubble-sort from left to right

                     (reverse (append y x)) ; need to output the reverse list as input was reversed upon entering cocktail-sort-left

                     (cocktail-sort-right (reverse (append y x)) '() 0 (length (append y x)))) ; perform bubblesort left to right

                  (if (< (car x) (cadr x)) ; if first item is greater then second item perform swap otherwise move to the right

                                 (if (= 2 (length x)) ; if only two items left in list x just swap dont need to append back

                     (cocktail-sort-left (cdr (swap x)) (save x y) (+ s 1) (- l 1))

                     (cocktail-sort-left (next x) (save x y) (+ s 1) (- l 1)))
```

```
                (cocktail-sort-left (cdr x) (append y (list (car x))) s (- l 1)))))) ; move to the right of list by removing first item


;purpose: performs bubble-sort from left to right of list
;x : current list of element to perform bubble sort, first item is continually removed as functions recurses
;y : used to save the list of elements that have already been traversed, one item is added to list as function recurses
;s : used to hold the number of swaps that have occurred during one recurse from the left to right
;l : used to hold the length of the initial input list
(define cocktail-sort-right (lambda (x y s l)
                        (if (= 1 l) ; if there is one item left in the list we have finished left-right traversal, so check num swaps
                          (if (= 0 s) ; if num swaps is 0 list has been sorted so output otherwise bubble-sort from right to left
                            (append y x) ; append last item to the end of the saved list and return
                            (cocktail-sort-left (reverse (append y x)) '() 0 (length (append y x)))) ; perform bubblesort right to left
                          (if (> (car x) (cadr x)) ; if first item is greater then second item perform swap otherwise move to the right
                            (if (= 2 (length x)) ; if only two items left in list x just swap dont need to append back
                                (cocktail-sort-right (cdr (swap x)) (save x y) (+ s 1) (- l 1))

                              (cocktail-sort-right (next x) (save x y) (+ s 1) (- l 1)))

                            (cocktail-sort-right (cdr x) (append y (list (car x))) s (- l 1)))))) ; move to the right of list by removing first item




(define cocktail-sort (lambda (x)
                        (if (>= (length x) 1)
                          (cocktail-sort-right x '() 0 (length x))
                          (begin
                            (newline (current-output-port))
                            (display "input is empty" (current-output-port))
                            (newline (current-output-port)))))) ; function that starts algorithm by calling cocktail sort

;input tests
;test set of numbers
(cocktail-sort '(3 1 5 2))
```

```
;test empty input
(cocktail-sort '())

;test one number
(cocktail-sort '(1))

;numbers in descending order
(cocktail-sort '(5 4 3 2 1))

;list containing the same number twice
(cocktail-sort '(5 2 2 1 9 3))

;test already sorted list
(cocktail-sort '(1 2 3 4 5))
```

Testing

Test 1: set of number in unsorted order

Input: ‘(3 1 5 2)

Output: Value: (1 2 3 5)

Evaluation: As we can see the test was successful as the input numbers were output in sorted order starting with 1 and ending with 5.

Test 2: test empty list input

Input: ‘()

Output: “input is empty”

Evaluation: As we can see the test was successful as the program responds correctly to an empty input stating the message I set which was that the user has given an empty input.

Test 3: test one number in the list

Input: ‘(1)

Output: (1)

Evaluation: As we can see the test was successful as the program just outputs the single number in the list without trying to sort it. If there is only one number then it is already sorted.

Test 4: numbers in descending order

Input: ‘(5 4 3 2 1)

Output: (1 2 3 4 5)

Evaluation: As we can see the test was successful as the algorithm successfully reverses the output so it is in ascending order.

Test 5: list containing the same number twice

Input: '(5 2 2 1 9 3)

Output: (1 2 2 3 5 9)

Evaluation: As we can see the test was successful as the algorithm successfully outputs the list in sorted order placing the two repeated numbers next to each other.

Test 6: numbers in already sorted order

Input: '(1 2 3 4 5)

Output: (1 2 3 4 5)

Evaluation: As we can see the test was successful as the algorithm successfully outputs the same input list as it is already sorted.

Investigate how scheme performs file IO. Does this break regularity? Why or why not?

Scheme performs file IO using ports. Ports being objects that can either be input ports that deliver data from file or output ports that accept data and add to a file. These ports act as pointers to the stream of characters in a file allowing a user to extract data from a file or place objects into the stream of characters. The *open-input-port* function will take a filename as input and return a port to that file. This port can then be input into the *read-char* function which will return the next character from the input port or the port can be input in to the *read* function which will return the next object from the input port. The *open-output-file* can take as input a filename and then will return a output port. This output port can then be input into the function *display* along with an object containing strings or characters. These strings or characters will be printed to the file. The write-char function can also be used to write a single character to a file. It takes as input a character and an output port to the file. I believe that file IO does break regularity as it introduces the port object which is unlike any other object. It is used just for input and output in scheme. Scheme only permitting two type of objects atoms and lists, having the port object breaks this regularity. Also, being able to open a port to a file from anywhere within a scheme file may also break regularity by causing there to now be side affects between functions when there is usually few side effects in scheme as a functional language. Now it will matter the order in which functions are called because if two functions are manipulating a file then changing that order can cause unexpected side effects. For example you may want to use two functions to print text to file in a specific order, then you must call one before the other. Hence we will not always have the same result from two functions if they are completed in a different order and both interact with the same file.

Reflection:

Whilst writing my cocktail sort I found that scheme follows the simplicity principle. This is mainly due to it only having one type of construct the function call. This means that everything you do in scheme involves calling a function. E.g. in order to do basic arithmetic it involves entering number as argument to the addition function (e.g. + 1 2) to do basic conditional control it involves entering arguments into the if function. This makes the language writable in that there are few concepts to understand to begin writing a program. It also makes the program a lot more readable as there are

few things that a programmer must understand to analyse someone's code. In terms of simplicity there is no inbuilt control structures except for the if and else statements. This means that writability is easier in the sense that there are few control structures to know about before you start programming. But it does make writing harder especially if you are familiar with procedural programming because control structures such as while loops and for loops are commonly used to perform iteration and not having these may make it harder to perform these operations, forcing you to create your own functions. In my case performing the cocktail sort was much harder in scheme as I am used to using for loops to iterate over my data and not having this was unusual. Instead I had to use recursion to cycle through the data, which may be easy to implement if you are familiar with recursion. In terms of readability not having these inbuilt control structures may make understanding what is going on in the code hard for someone unfamiliar with recursion as it doesn't simply iterate like most for loops.

I would also consider Scheme very orthogonal due to the small amount of core primitive functions that you can combine to create more complex functions. A prime example of this is the CAR and CDR pointers which can be combine to perform complex operations on a list. CAR returning the first atom/list in a list and CDR returning the rest of the list. We can combine these two operations in several ways such as CADDR to obtain the third atom/list in a list. In my case throughout my program I used these two operations to obtains elements from the list. This orthogonality makes scheme very writable as you only need to know how these two operations work In order to perform complex operations by combining them. It allows you to do a number of different things just by using these two simple operations. In terms of readability it requires someone reading the code to have to only know how these two operations work to understand the code, making it readable in this sense. But it can make the code more unreadable if a programmer uses these operations in very complex ways such CADDADDDAR, understanding what this it doing on a particular list may take some time to understand.

I also found that it follows the abstraction principle as through lambda expressions and the define function you can easily define your own function and factor out any reoccurring patterns. This allows you to easily split up your code making it much more readable to other programmers as they can see what each individual expression does. I created a number of my own lambda expressions to split up my code. And stop the repetition of large amounts of code. Particularly to swap two elements, to go to the next element, to append elements, etc. This makes coding much more writable as you can easily split up code and test them separately before combining. Can also make it easier to keep track of what you have written as you don't just have one large section of code.

I found that the scheme does not follow the structured programming. With their being no variable assignment all functions occur within each other, so the the return of one function can be the input of another. This results in several parentheses particularly at the end of a function which make it hard to keep track of where functions start and end. This makes it hard to write in scheme when your functions are very complex as there are so many parathesis you must keep track of. It also makes the language less readable as a programmer may get confused when they are trying to figure out the input for a particular function.

In terms of reliability scheme follows the defence in depth principle by having exception handling. Though I didn't have to use it in my program whilst researching scheme I did come across it. You are to raise errors and handling them. This hence makes the program more reliable by adding a level of safety to your program so that if something unexpected goes wrong with your program it can be dealt with, stopping it from crashing.

References:

"Linked List Program in C - Tutorialspoint", *Tutorialspoint.com*, 2020. [Online]. Available: https://www.tutorialspoint.com/data_structures_algorithms/linked_list_program_in_c.htm. [Accessed: 06- Nov- 2020].

"QuickSort - GeeksforGeeks", *GeeksforGeeks*, 2020. [Online]. Available: https://www.geeksforgeeks.org/quick-sort/. [Accessed: 06- Nov- 2020].

"Bubble Sort - GeeksforGeeks", *GeeksforGeeks*, 2020. [Online]. Available: https://www.geeksforgeeks.org/bubble-sort/. [Accessed: 06- Nov- 2020].

"Fizz buzz", *En.wikipedia.org*, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Fizz_buzz. [Accessed: 06- Nov- 2020].

D. Banas, *Youtube.com*, 2020. [Online]. Available: https://www.youtube.com/watch?v=SykxWpFwMGs&t=2499s. [Accessed: 06- Nov- 2020].

"syntax in Smalltalk", *Rigaux.org*, 2020. [Online]. Available: http://rigaux.org/language-study/syntax-across-languages-per-language/Smalltalk.html. [Accessed: 06- Nov- 2020].