# Programming Languages Assignment

**Nicholas Klvana-Hooper (19782944) – Updated 4/12/2020**

## Generic Comments

Before I showcase the weekly programs and reflection questions there are some general comments about how I tested the fizzbuzz programs. I wrote a bash script for every one of these programs that would compile (if needed) and then run the code, would then diff (Linux command to compare two files) the piped output of the fizzbuzz program with an expected result file.

Each of the bash scripts are located under the language, however I only provided a basic output of fizzbuzz with all of the output of the program just to make it more generic. The actual files that the outputs where compared with the bash script were tailor made through the python program, this is just due to different languages having different syntax for how they output strings and integers – therefore had to be adjusted per language.

I have placed the python script below (taken from the fizzbuzz program in Smalltalk):

```
1.  for num in range(101):
2.      if num % 3 == 0 and num % 5 == 0:
3.          print("\'fizzbuzz\'")
4.      elif num % 5 == 0:
5.          print("\'buzz\'")
6.      elif num % 3 == 0:
7.          print("\'fizz\'")
8.      else:
9.          print(str(num))
```

A sample file for the test case outputted by this script for Smalltalk is below:

```
1.  1
2.  2
3.  'fizz'
4.  4
5.  'buzz'
6.  'fizz'
7.  7
8.  8
9.  'fizz'
10. 'buzz'
11. 11
12. 'fizz'
13. 13
14. 14
15. 'fizzbuzz'
16. 16
17. 17
18. 'fizz'
19. 19
20. 'buzz'
```

# Table Of Contents

# Fortran

## Program

```
1.          program fizzbuzz
2.
3.          integer i
4.
5.  c Loops through from 1-100
6.          do 10 i = 1, 100
7.  c Fizzbuzz occurs every multiple of 15
8.              if(mod(i, 15) .EQ. 0) then
9.                  write (*,*) 'fizzbuzz'
10.             else
11. c Otherwise fizz every multiple of 3 and buzz every one of 5
12.                 if (mod(i, 3) .EQ. 0) then
13.                     write (*,*) 'fizz'
14.                 elseif (mod(i, 5) .EQ. 0) then
15.                     write (*,*) 'buzz'
16.                 else
17. c If not a multiple of 3,5,15 then print number
18.                     write (*,*) i
19.                 endif
20.             endif
21. 10      continue
22.
23.         stop
24.         end
```

## Output of Program vs Expected

Output of program:

```
1.            1
2.            2
3.  fizz
4.            4
5.  buzz
6.  fizz
7.            7
8.            8
9.  fizz
10. buzz
11.          11
12. fizz
13.          13
14.          14
15. fizzbuzz
16.          16
17.          17
18. fizz
19.          19
20. buzz
```

```
81. fizz
82.          82
83.          83
84. fizz
85. buzz
86.          86
87. fizz
```

Expected output:

```
1.  1
2.  2
3.  fizz
4.  4
5.  buzz
6.  fizz
7.  7
8.  8
9.  fizz
10. buzz
11. 11
12. fizz
13. 13
14. 14
15. fizzbuzz
16. 16
17. 17
18. fizz
19. 19
20. buzz
```

```
81. fizz
82. 82
83. 83
84. fizz
85. buzz
86. 86
87. fizz
```

```
88.          88
89.          89
90. fizzbuzz
91.          91
92.          92
93. fizz
94.          94
95. buzz
96. fizz
97.          97
98.          98
99. fizz
100.  buzz
```

```
88. 88
89. 89
90. fizzbuzz
91. 91
92. 92
93. fizz
94. 94
95. buzz
96. fizz
97. 97
98. 98
99. fizz
100.  buzz
```

## Testing and Comments

The output for this file matched what it should've been with a few differences. As seen Fortran outputs the strings in a different column which is just how its implemented in the language. Other than that, the program outputs as expected. One thing to note it the funny indexing on 100 with buzz. This is due to the syntax highlighting program I used with 100 being one character longer so this can be ignored.

In order to test that the fizzbuzz program worked in Fortran, the following script was used in conjunction with an output file created by a similar python script to the one at the beginning of the document with syntax adjustments made.

```
1.  #!/bin/bash
2.
3.  f77 fizzbuzz.f
4.  ./a.out | diff fizzbuzz_ans.txt -
5.
6.  echo "  Test done: if there is other output, the test failed"
7.
8.  rm a.out
```

## Reflection

When looking though the Fortran manual given in the practical I noticed there was a do-while loop structure which I ended up using. If I didn't, I may have had to use a "while loop" which in Fortran is just a goto structure, which has many violations of the programming principles such as syntactic consistency, structured programming and defence in depth. This results in Fortran becoming less readable, writeable and reliable due to the syntax being less consistent as well as not having a simple structured flow. Fortran is also less reliable due to defence in depth being violated by not including exception handling and no type checking.

Instead, the do-while loop does not require the use of a goto, allowing me to dodge those issues. This structure actually complies with the principle of preservation of information because it shows what variable is being used for indexing inside the loop.

Using the do-while loop instead of the goto statement makes the program much more readable as it isn't as hard to follow the flow of the program. This in turn also helps writability as you don't have to think how your program is going to be structured and how it jumps around making it much easier to solve the fizzbuzz program. Because of using this do-

while loop it also made Fortran feel more reliable because I wasn't forced to write the fizzbuzz program in a strange way that didn't seem right to me (more readable and writeable).

The program did have issues such as syntactic consistency with modular which used a method style using brackets while other operations like equals, addition, etc. used their maths operator symbol (+, =). This makes the language feel harder to read and understand what is happening and harder to write and remember which style you have to use for a math operation.

## Question

Fortran has some familiar aspects to it such as the if statement structure and the general layout of a program, however there are a lot of differences such as the modular function and equality syntax and usage. The output for the fizzbuzz program with different columns for integers and strings is also a foreign concept compared to languages such as C and Java that all appear in the same column.

Even within the file, having different columns for comments, the code itself, and also for the continue statement to jump back to made it more difficult to read and understand than the typical C-like programs I have been used to programming in.

# Algol

## Program

```
# Loop from 1 to 100 #
FOR i TO 100 DO
    # if divisible by 15, print fizzbuzz otherwise go into else #
    IF i MOD 15 = 0
        THEN print(("fizzbuzz", new line))
    ELSE
        IF i MOD 3 = 0
            THEN print(("fizz", new line))
        ELIF i MOD 5 = 0
            THEN print(("buzz", new line))
        ELSE
            print((i, new line))
        FI
    FI
OD
```

## Output of Program vs Expected

Output of program:

```
         +1
         +2
fizz
         +4
buzz
fizz
         +7
         +8
fizz
buzz
        +11
fizz
        +13
        +14
fizzbuzz
        +16
        +17
fizz
        +19
buzz
```

```
fizz
        +82
        +83
fizz
buzz
        +86
fizz
        +88
        +89
fizzbuzz
        +91
        +92
fizz
        +94
buzz
fizz
```

Expected output:

```
1
2
fizz
4
buzz
fizz
7
8
fizz
buzz
11
fizz
13
14
fizzbuzz
16
17
fizz
19
buzz
```

```
fizz
82
83
fizz
buzz
86
fizz
88
89
fizzbuzz
91
92
fizz
94
buzz
fizz
```

```
97.        +97
98.        +98
99. fizz
100.  buzz
```

```
97. 97
98. 98
99. fizz
100.  buzz
```

## Testing

The output for this file matched what it should've been with a few differences. As seen Algol outputs the strings in a different column, as well as the integers have a positive sign in front of them, which is just how its implemented in the language and can be ignored. Other than that, the program outputs as expected. One thing to note it the funny indexing on 100 with buzz. This is due to the syntax highlighting program I used with 100 being one character longer so this can be ignored.

In order to test that the fizzbuzz program worked in Algol, the following script was used in conjunction with an output file created by a similar python script to the one at the beginning of the document with syntax adjustments made.

```
1.  #!/bin/bash
2.
3.  a68g fizzbuzz.a68 | diff fizzbuzz_ans.txt -
4.
5.  echo "  Test done: if there is other output, the test failed"
```

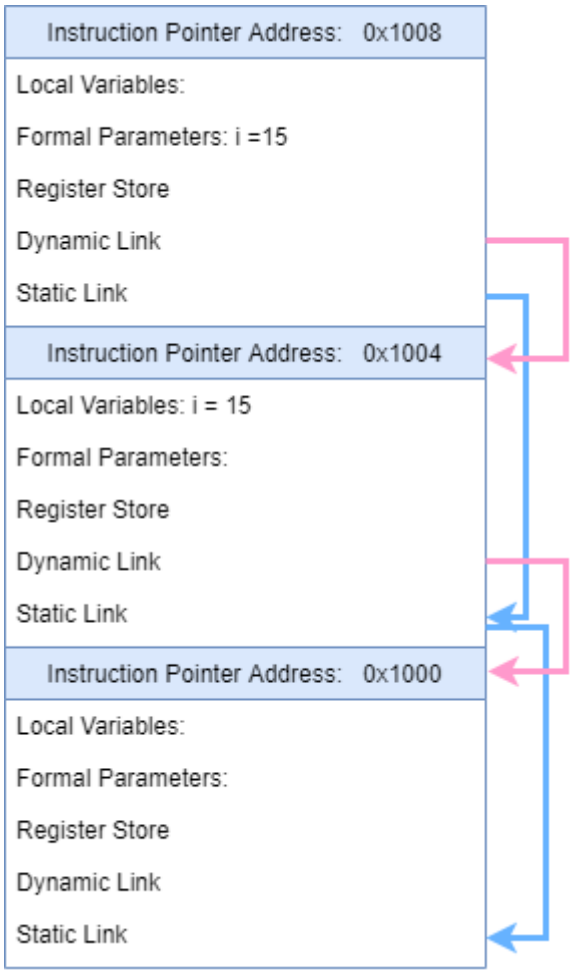
## Reflection

For the fizzbuzz program that was written in this language, there weren't too many issues that I came across with the programming principles. The parts that I did encounter were fairly syntactically consistent with "fi" being the end clause in an if statement. This was a little strange compared with "od" which was used in for loops, which violated syntactic consistency affecting both readability and especially writability of Algol

The overall use of these backwards versions of the function allows for a more readable program as it is easier to see where each section begins and ends, following the flow of the code. It is also quite writeable in this sense as its fairly simple where to put the endings of these statements. Because of the increased readability and writability Algol seems to be quite a decently reliable language, as most things such as loops feel more familiar than Fortran and straightforward (even though this conflicts with the overall consensus that Algol is a very orthogonal but not a very simple language), though it does lack exception handling which does make it less reliable as a language and violates the defence in depth principle.

It is also noted that Algol has a for loop included within the language, compared to Fortran which does not have one, only a while and do-while loop. This is more expressive and is therefore easier to write than a counting while loop. This is a big trade-off because in one hand having the for loop increases the number of types loops in Algol, also increasing the orthogonality of the language which in turn increases the readability of Algol . On the other side, this increases the complexity of the language and therefore decreases readability by making it harder to understand what loop is being used. The differing syntax between the loops makes it less writeable as it is harder to remember how to write the different loops.

## Question




I'm assuming that this is what the stack would look like **immediately** after printing fizzbuzz, and not iterating onto the 16$^{th}$ iteration in the for loop. The pointer addresses are generalisations in this example, obviously the activation records would be bigger, and would not associate in this order most likely.

# Ada

## Program [1]

```
1.  with Ada.Text_IO; use Ada.Text_IO;
2.
3.  procedure bubblesort is
4.      -- rename so less has to be written everytime
5.      package IO renames Ada.Text_IO;
6.      -- array to be sorted
7.      A: array(0 .. 4) of Natural := (5, 4, 3, 1, 2);
8.      temp : integer:= 0;
9.  begin
10.     --outer sort
11.     For1 : for I in reverse A'Range loop
12.         -- inner sort
13.         For2 : for J in A'First ..I loop
14.             if A(I) < A(J) then
15.                 temp := A(J);
16.                 A(J) := A(I);
17.                 A(I) := temp;
18.             end if;
19.         end loop For2;
20.     end loop For1;
21.
22.     ForPrint : for K in A'Range loop
23.         IO.Put_Line(Integer'Image(A(K)));
24.     end loop ForPrint;
25. end bubblesort;
```

## Output of Program vs Expected

For input list (1, 2, 3, 4, 5)

Output of program:

```
1.  1
2.  2
3.  3
4.  4
5.  5
```

Expected output:

```
1.  1
2.  2
3.  3
4.  4
5.  5
```

For input list (5, 4, 3, 2, 1)

Output of program:

```
1.  1
2.  2
3.  3
4.  4
5.  5
```

Expected output:

```
1.  1
2.  2
3.  3
4.  4
5.  5
```

For input list (3, 2, 5, 4, 1)

Output of program:

```
1.  1
2.  2
3.  3
4.  4
5.  5
```

Expected output:

```
1.  1
2.  2
3.  3
4.  4
5.  5
```

## Testing and Comments

The actual list that gets sorted is hardcoded within the program itself (see line 5 of the Ada program to see the list). In order to test this I manually changed that list into the 3 cases that are included above. These test cases included an already sorted list, a reverse list and a randomly generated one. When the list was changed, the following script was used to compile and test them against a file that just included the expected result (found above) in a file which the output was piped and compared to.

```
1.  #!/bin/bash
2.
3.  gnatmake bubblesort.adb
4.
5.  ./bubblesort | diff bubblesort_ans.txt -
6.
7.  echo "  Test done: if no other output, test passed"
8.
9.  rm bubblesort
10. rm bubblesort.ali
11. rm bubblesort.o
```

## Reflection

Ada is incredibly regular; for instance all the loops have the same structure and are all called loop. Another example is that if statements and also the loop structures have an associated end word such as “end if” or “end loop”, making it really easy to remember.

This regular syntax that is consistent over everything I encountered in this program made Ada very readable as it was very easy to see where structures exist and to follow the flow of the program. Ada is also very writeable because of this, its very easy to remember how to structure everything especially the loops. Reliability is also increased in this sense because it doesn’t encourage the user to do something that doesn’t make sense, and is more likely to write code that is correct.

Another comment is that Ada is not the most simple language. The declaration of the procedure and ways of making the loops and such makes Ada less readable and writeable as its harder to understand what is happening and arbitrary code.

Although I did not include it within my code, Ada included exception handling which makes it such a robust and safe language. The inclusion of exception handling in a language and program makes it comply with the defence in depth principle in which an error should be caught by the next line of defence if it gets through one. Supporting this principle makes Ada more reliable because of this.

## Question

One of the most notable differences between the Ada and C implementations of the bubble sort program is the fact that you have to pass the array length to the sorting algorithm for looping in C, this is not the case in Ada where you can just get the range of the array. Another notable difference is the second loop in Ada is structured slightly differently because for loops can’t have arithmetic telling the loop when to stop. So instead, the first

loop counts down from length-1 to 0, instead of counting forward like most C versions. The second loop however still counts forward from 0 to the value inside the first for loop. This allows it so skip the last values at the end of the array which have already been sorted.

There are a lot of major syntax differences between the two languages. For instance, instead of the []  brackets with arrays in C we use the () ones in Ada. Or instead of the closing brackets ( } ) for loops and if statements in C, its “end loop”/”end if” in Ada.

# Yacc & Lex

Note: There are a yacc and lex file, a linkedlist c and h file, and also a makefile which

### Program (sort.l – flex file)

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

%%
[1234567890]+   yylval=atoi(yytext); return NUMBER;
,               ;
\[              ;
\]              return CBRACE;
\n              ;
[ \t]+          ;
%%
```

### Program (sort.y – bison file) [2]

```
%{
#include <stdio.h>
#include <string.h>
#include "LinkedList.h"

typedef struct{
    int num;
} numEntry;

extern void yyerror(const char *str);
extern int yywrap();
extern int main();
extern int yyparse(void);
extern int yylex(void);
extern void printList(LListNode* node);
extern void swap(LListNode *a, LListNode *b);

LinkedList * list;

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

int yywrap()
{
    return 1;
}

int main()
{
    list = NULL;
    list = createLinkedList();

    yyparse();
    freeList(&list);
}

void printList(LListNode* node)
```

```
{
    numEntry *curr;
    curr = node->data;

    if(curr == list->head->data)
    {
        printf("[%d", curr->num);
    }
    else
    {
        printf(", %d", curr->num);
    }

    if(curr == list->tail->data)
    {
        printf("]");
    }
}

void bubbleSort(LinkedList* list)
{
    int swapped, i;
    LListNode *tmp;
    LListNode *lptr = NULL;
    numEntry *currE, *nextE;

    if(list->head == NULL)
    {
    }
    else
    {
        do
        {
            swapped = 0;
            tmp = list->head;

            while(tmp->next != lptr)
            {
                currE = tmp->data;
                nextE = tmp->next->data;
                if(currE->num > nextE->num)
                {
                    swap(tmp, tmp->next);
                    swapped = 1;
                }
                tmp = tmp->next;
            }
            lptr = tmp;
        }
        while(swapped);
    }
}

void swap(LListNode *a, LListNode *b)
{
    LListNode *temp = a->data;
    a->data = b->data;
    b->data = temp;
}

      %}

      %token NUMBER COMMA OBRACE CBRACE

      %%
      commands:
```

```
            | commands command
            ;

        command:
            number
            |
            close
            ;

        number:
            NUMBER
            {
                numEntry *numEnt, *head;
                numEnt = (numEntry*)malloc(sizeof(numEntry));
                numEnt->num = $1;

                insertStart(list, numEnt);
                bubbleSort(list);
            }
            ;
        close:
            CBRACE
            {
                printLinkedList(list, &printList);
                printf("\n");
            }
            ;
        %%
```

## Program (makefile – make file)

```
#Makefile

CC = gcc
EXEC = SortList
CFLAGS = -Wall -ansi -Werror -pedantic
OBJ = lex.yy.c sort.tab.c LinkedList.o

$(EXEC) : $(OBJ)
    $(CC) $(OBJ) -o $(EXEC)

lex.yy.c : sort.l sort.tab.h
    flex sort.l

sort.tab.c : sort.y LinkedList.h
    bison -d sort.y

LinkedList.c : LinkedList.c LinkedList.h
    $(CC) -c LinkedList.c $(CFLAGS)

clean:
    rm -f $(EXEC) $(OBJ)
```

## Program (linkedlist.c – linked list c file) [3]

```
/*
 * File: LinkedList.c
 * File Created: Tuesday, 1st October 2019
 * Author: Nicholas Klvana-Hooper
 * -----
 * Last Modified: Monday, 21st October 2019
 * Modified By: Nicholas Klvana-Hooper
 * -----
```

```
 * Purpose: A data stucture that can contain data and can expand and shrink
 * Reference: Functions provided are based on Lecture 7 (UCP)
              Also is a modified version of prac7 LinkedList.c made by Nicholas Klvana-Hooper (me)
              Accessed on the 21st October 2019
 */

#include <stdio.h>
#include <stdlib.h>
#include "LinkedList.h"

/*
 * SUBMODULE: createLinkedList
 * IMPORT:
 * EXPORT: list(LinkedList*)
 * ASSERTION: Create a new linked list with default values
 */
LinkedList* createLinkedList()
{
    LinkedList* list;

    /* Creates the linked list pointer */
    list = (LinkedList*)malloc(sizeof(LinkedList));
    /* Sets defaults for the linked list */
    (*list).head = NULL;
    (*list).tail = NULL;
    list->count = 0;

    return list;
}

/*
 * SUBMODULE: insertStart
 * IMPORT: list(LinkedList*), data(void*)
 * EXPORT:
 * ASSERTION: Inserts a new node at the front of the list
 */
void insertStart(LinkedList* list, void* data)
{
    /* Creates a newnode with data given */
    LListNode* newNode;
    newNode = (LListNode*)malloc(sizeof(LListNode));
    (*newNode).data = data;

    /* If List is empty head and tail should be made to be the new ndoe */
    if(list->count == 0)
    {
        (*list).head = newNode;
        (*list).tail = newNode;
        (*(*list).head).next = NULL;
    }
    /* Otherwise put new node at the beginning of the list */
    else
    {
        (*newNode).next = (*list).head;
        (*list).head = newNode;
    }
    list->count += 1;
}

/*
 * SUBMODULE: removeStart
 * IMPORT: list(LinkedList*)
 * EXPORT: node(void*)
 * ASSERTION: Removes the node at the front and returns it
 */
```

```
74. void* removeStart(LinkedList* list)
75. {
76.     LListNode* toRemove;
77.     void* value;
78. 
79.     /* Checks if list is empty, and thus cannot remove */
80.     if(list->count == 0)
81.     {
82.         printf(ERRCLR "Can't remove from empty list" CLR);
83.     }
84.     /* Otherwise there is at least a node to delete */
85.     else
86.     {
87.         /* Gets data for export */
88.         toRemove = (*list).head;
89.         value = toRemove->data;
90. 
91.         /* If list is 1 then removing should set list to NULL */
92.         if(list->count ==1)
93.         {
94.             list->head = NULL;
95.             list->tail = 0;
96.         }
97.         /* Otherwise set next to the value after the front */
98.         else
99.         {
100.                     list->head = list->head->next;
101.                 }
102. 
103.                 /* Cleanup node and decrease count of nodes */
104.                 free(toRemove);
105.                 list->count -= 1;
106.             }
107. 
108.             return value;
109.         }
110. 
111.         /*
112.          * SUBMODULE: insertLast
113.          * IMPORT: list(LinkedList*), data(void*)
114.          * EXPORT: void
115.          * ASSERTION: Inserts a new node at the end of the list
116.          */
117.         void insertLast(LinkedList* list, void* data)
118.         {
119.             /* Creates new node with given data */
120.             LListNode* newNode;
121.             newNode = (LListNode*)malloc(sizeof(LListNode));
122.             (*newNode).data = data;
123. 
124.             /* If list is empty then newNode is tail and head */
125.             if(list->count == 0)
126.             {
127.                 (*list).head = newNode;
128.                 (*list).tail = newNode;
129.                 (*(*list).tail).next = NULL;
130.             }
131.             /* Otherwise insert node at the end of the list */
132.             else
133.             {
134.                 newNode->next = NULL;
135.                 list->tail->next = newNode;
136.                 (*list).tail = newNode;
137.             }
138.             /* Increase number of nodes in list */
139.             list->count += 1;
```

```
140.        }
141.
142.        /*
143.         * SUBMODULE: removeLast
144.         * IMPORT: list(LinkedList*)
145.         * EXPORT: data(void*)
146.         * ASSERTION: Removes node at end of list and returns it
147.         */
148.        void* removeLast(LinkedList* list)
149.        {
150.            /* Uses a single linked, double ended so therefore need a prev node */
151.            LListNode* curr;
152.            LListNode* prev;
153.            void* value;
154.
155.            curr = (*list).head;
156.
157.            /* Checks if list is empty and thus can't remove */
158.            if(list->count == 0)
159.            {
160.                printf(ERRCLR "Can't remove from empty list" CLR);
161.            }
162.            /* Otherwise can remove a node */
163.            else
164.            {
165.                /* If there is one value get it and set list to NULL */
166.                if(list->count == 1)
167.                {
168.                    curr = list->head;
169.                    list->head = NULL;
170.                    list->tail = NULL;
171.                }
172.                /* Otherwise remove and set tail to point to second last node */
173.                else
174.                {
175.                    /* Find second last node */
176.                    while ((*curr).next != NULL)
177.                    {
178.                        prev = curr;
179.                        curr = (*curr).next;
180.                    }
181.
182.                    /* Set tail to second last ndoe and unlink the last one */
183.                    (*list).tail = prev;
184.                    (*(*list).tail).next = NULL;
185.                }
186.
187.                /* Get ready to return data and free up last node */
188.                value = curr->data;
189.                free(curr);
190.                list->count -= 1;
191.            }
192.            return value;
193.        }
194.
195.        /*
196.         * SUBMODULE: writeLinkedList
197.         * IMPORT: list(LinkedList*), file(FILE*), void (*funcPointer)(LListNode*, F
    ILE*)
198.         * EXPORT: void
199.         * ASSERTION: Traverses list and gives data to writing function for struct
200.         */
201.        void writeLinkedList(LinkedList* list, FILE* file, void (*funcPointer)(LList
    Node*, FILE*))
202.        {
203.            LListNode* curr;
```

```
204.            curr = list->head;
205.
206.            /* Traverse list */
207.            while(curr != NULL)
208.            {
209.                /* Calls function given to write the data in the node */
210.                (*funcPointer)(curr, file);
211.                /* Now get the next node to print */
212.                curr = curr->next;
213.            }
214.
215.        }
216.
217.        /*
218.         * SUBMODULE: printLinkedList
219.         * IMPORT: list(LinkedList*), void (*funcPointer)(LListNode*, FILE*)
220.         * EXPORT: void
221.         * ASSERTION: Traverses list and gives data to print function for struct
222.         */
223.        void printLinkedList(LinkedList* list, void (*funcPointer)(LListNode*))
224.        {
225.            LListNode* curr;
226.            curr = list->head;
227.
228.            /* Traverse list */
229.            while(curr != NULL)
230.            {
231.                /* Calls function to print the data in the node */
232.                (*funcPointer)(curr);
233.                /* Now get the next node to print */
234.                curr = curr->next;
235.            }
236.
237.        }
238.
239.        /*
240.         * SUBMODULE: freeLinkedList
241.         * IMPORT: list(LinkedList*), file(FILE*), void (*funcPointer)(LListNode*, F
    ILE*)
242.         * EXPORT: void
243.         * ASSERTION: Calls function to free nodes and then the list pointer
244.         */
245.        void freeLinkedList(LinkedList* list, void (*funcPointer)(LListNode*))
246.        {
247.            /* Call for the nodes to be freed */
248.            freeNode((*list).head, funcPointer);
249.            /* Free the list pointer */
250.            free(list);
251.        }
252.
253.        /*
254.         * SUBMODULE: freeLinkedList
255.         * IMPORT: node(LListNode*), file(FILE*), void (*funcPointer)(LListNode*, FI
    LE*)
256.         * EXPORT: void
257.         * ASSERTION: Recursive function freeing every node using function provided
    for struct
258.         */
259.        void freeNode(LListNode *node, void (*funcPointer)(LListNode*))
260.        {
261.            /* Traverse nodes */
262.            if(node != NULL)
263.            {
264.                /* Recursive call to reach all nodes */
265.                freeNode(node->next, funcPointer);
266.
```

```
267.                /* Calls function provided to free the node data */
268.                (*funcPointer)(node);
269.                /* Free the node itself */
270.                free(node);
271.            }
272.        }
273.
274.        void freeList(LinkedList** list)
275.        {
276.            freeLinkedList(*list, &freeNodes);
277.        }
278.
279.        void freeNodes(LListNode* node)
280.        {
281.            free(node->data);
282.        }
```

## Program (linkedlist.h – linked list h file) [3]

```
1.  /*
2.   * File: LinkedList.h
3.   * File Created: Tuesday, 1st October 2019
4.   * Author: Nicholas Klvana-Hooper
5.   * -----
6.   * Last Modified: Thursday, 17th October 2019
7.   * Modified By: Nicholas Klvana-Hooper
8.   * -----
9.   * Purpose: Includes method definitions for LinkedList.c
10.  * Reference: Functions provided are based on Lecture 7 (UCP)
11.                Also is a modified version of prac7 LinkedList.c made by Nicholas Klv
    ana-Hooper (me)
12.  */
13.
14. #ifndef LINKEDLIST_H
15. #define LINKEDLIST_H
16.
17. /* Normal Text colour definition */
18. #define CLR "\033[0m"
19. /* Red text definitions - for error printing */
20. #define ERRCLR "\033[1;31m"
21.
22. /* LinkedListNode Struct defintion */
23. typedef struct LListNode{
24.     void* data;
25.     struct LListNode* next;
26. } LListNode;
27.
28. /* LinkedList Struct defintion */
29. typedef struct{
30.     LListNode* head;
31.     LListNode* tail;
32.     int count;
33. } LinkedList;
34.
35. LinkedList* createLinkedList();
36. void insertStart(LinkedList* list, void* entry);
37. void* removeStart(LinkedList* list);
38. void insertLast(LinkedList* list, void* entry);
39. void* removeLast(LinkedList* list);
40. void printLinkedList(LinkedList* list, void (*funcPointer)(LListNode*));
41. void writeLinkedList(LinkedList* list, FILE* file, void (*funcPointer)(LListNode*,
    FILE*));
42. void freeLinkedList(LinkedList* list, void (*funcPointer)(LListNode*));
43. void freeNode(LListNode *node, void (*funcPointer)(LListNode*));
```

```
44. void freeList(LinkedList** list);
45. void freeNodes(LListNode* node);
46.
47. #endif
```

## Output of Program vs Expected

For input list (1, 2, 3, 4, 5)

Output of program:

```
1.  [1, 2, 3, 4, 5]
```

Expected output:

```
1.  [1, 2, 3, 4, 5]
```

For input list (5, 4, 3, 2, 1)

Output of program:

```
1.  [1, 2, 3, 4, 5]
```

Expected output:

```
1.  [1, 2, 3, 4, 5]
```

For input list (3, 2, 5, 4, 1)

Output of program:

```
1.  [1, 2, 3, 4, 5]
```

Expected output:

```
1.  [1, 2, 3, 4, 5]
```

## Testing and Comments

In order to test that the yacc and lex programs work, I have tested them together using the following bash script. On line five is a string that is echoed into the input of the program. The above test cases were completed by manually changing that string to the test cases and running the script again to see if the program outputted correctly. The answer file was just a text file that contained the expected output which is also shown above.

```
1.  #!/bin/bash
2.
3.  make
4.
5.  echo "[5,4,2,3,1]" | ./SortList | diff sort_ans.txt -
6.  echo "  Test Done: if no print above this, test worked"
7.
8.  make clean
```

## Reflection

Yacc and Lex were interesting “languages” to program in, they are both quite similar with a big integration of the C language inside of the yacc language. A common issue with both of these language is regularity in regards to the use of the % symbol. Both languages have the use of the “%{“ and “%}” for putting the header code within it, as well as “%%” for the beginning and end of the body and then also “%token”. In each example the “%” is used for a different purpose but all look like similar things.

The differing usages of the “%” symbol in different contexts proves to be a syntactic consistency problem, in which they all do different things whilst having similar syntax instead of differing syntax. This makes these two languages harder to read and write as its harder to understand what is happening in program, and when it is required to use the different symbols. This also decreases the reliability of the program as a bi-product of the lesser readability and writability as the outcome of a program is harder to predict.

## Question

A symbol table is quite important in compilers, it contains what variable types mean and their associated tasks. In yacc its basically a list, with a recursive one-symbol look ahead. This would mean it is quite slow if the symbol happened to be at the end of the document. Most languages should anyways, have the more frequent and important symbols at the front of the list which would speedup the compiling time as it takes less time to find the more common symbols.

However there are lots of ways you could store the symbol table to make it even faster. A hash table for instance would increase the lookup for a symbol from O(n) in using a list to a potential O(1). There are some problems that would have to be overcome with hash tables though, such as the hashing algorithm and length of the table and type of hash table. However implementing this instead of a list would severely increase the speed of symbol look ups during compiling.

# Scripting Languages (Bash, Perl, Ruby)

## Bash Script

```
#!/bin/bash

find / -type f -ipath *.conf 2>/dev/null
```

## Perl Script

```
#!/usr/bin/env perl

use Modern::Perl;
use autodie;


use File::Find;
find ( \&wanted, "/" );

# this function is called on every file that the find comes up with
sub wanted {
    # this prints the file name if the file name matches the regex (.conf)
    print "$File::Find::name\n" if "$File::Find::name" =~ /\.conf$/;
}
```

## Ruby Script

```
require 'find'

Find.find("/") do |path|
    if File.basename(path).end_with?('.conf')
        puts path
    end
end
```

## Output of Program vs Expected

### For Bash Script

Output of program:

```
/snap/snapd/9721/etc/apt/apt.conf.d/20snapd.conf
/snap/snapd/9721/etc/ld.so.conf
/snap/snapd/9721/etc/ld.so.conf.d/libc.conf
/snap/snapd/9721/etc/ld.so.conf.d/x86_64-linux-gnu.conf
/snap/snapd/9721/usr/lib/environment.d/990-snapd.conf
/snap/snapd/9721/usr/share/dbus-1/session.d/snapd.session-services.conf
/snap/snapd/9721/usr/share/dbus-1/system.d/snapd.system-services.conf
/snap/core18/1932/etc/adduser.conf
/snap/core18/1932/etc/apparmor/parser.conf
/snap/core18/1932/etc/apparmor/subdomain.conf
/snap/core18/1932/etc/ca-certificates.conf
/snap/core18/1932/etc/dbus-1/system.d/wpa_supplicant.conf
/snap/core18/1932/etc/deluser.conf
/snap/core18/1932/etc/depmod.d/ubuntu.conf
/snap/core18/1932/etc/dhcp/dhclient.conf
/snap/core18/1932/etc/host.conf
/snap/core18/1932/etc/ld.so.conf
/snap/core18/1932/etc/ld.so.conf.d/i386-linux-gnu.conf
```

```
19. /snap/core18/1932/etc/ld.so.conf.d/libc.conf
20. /snap/core18/1932/etc/ld.so.conf.d/x86_64-linux-gnu.conf
```

Not 21st line… jumps to end of output and then last

```
21. /etc/security/pwquality.conf
22. /etc/security/sepermit.conf
23. /etc/security/pam_env.conf
24. /etc/security/time.conf
25. /etc/security/group.conf
26. /etc/logrotate.conf
27. /etc/selinux/semanage.conf
28. /etc/libaudit.conf
29. /etc/popularity-contest.conf
30. /etc/sysctl.conf
31. /home/nicholas/.config/Trolltech.conf
32. /home/nicholas/.config/xsettingsd/xsettingsd.conf
33. /var/lib/ucf/cache/:etc:rsyslog.d:50-default.conf
34. /var/lib/ucf/cache/:etc:samba:smb.conf
35. /run/tmpfiles.d/static-nodes.conf
36. /run/NetworkManager/resolv.conf
37. /run/NetworkManager/no-stub-resolv.conf
38. /run/NetworkManager/conf.d/10-globally-managed-devices.conf
39. /run/systemd/resolve/stub-resolv.conf
40. /run/systemd/resolve/resolv.conf
```

## For Perl Script

Output of program:

```
1.  /snap/snapd/9721/etc/ld.so.conf
2.  /snap/snapd/9721/etc/apt/apt.conf.d/20snapd.conf
3.  /snap/snapd/9721/etc/ld.so.conf.d/libc.conf
4.  /snap/snapd/9721/etc/ld.so.conf.d/x86_64-linux-gnu.conf
5.  /snap/snapd/9721/usr/lib/environment.d/990-snapd.conf
6.  /snap/snapd/9721/usr/share/dbus-1/session.d/snapd.session-services.conf
7.  /snap/snapd/9721/usr/share/dbus-1/system.d/snapd.system-services.conf
8.  /snap/core18/1932/etc/adduser.conf
9.  /snap/core18/1932/etc/ca-certificates.conf
10. /snap/core18/1932/etc/deluser.conf
11. /snap/core18/1932/etc/host.conf
12. /snap/core18/1932/etc/ld.so.conf
13. /snap/core18/1932/etc/libaudit.conf
14. /snap/core18/1932/etc/nsswitch.conf
15. /snap/core18/1932/etc/resolv.conf
16. /snap/core18/1932/etc/ucf.conf
17. /snap/core18/1932/etc/apparmor/parser.conf
18. /snap/core18/1932/etc/apparmor/subdomain.conf
19. /snap/core18/1932/etc/dbus-1/system.d/wpa_supplicant.conf
20. /snap/core18/1932/etc/depmod.d/ubuntu.conf
```

Not 21st line… jumps to end of output and then last

```
21. /etc/fonts/conf.d/64-02-tlwg-norasi.conf
22. /etc/fonts/conf.d/60-generic.conf
23. /etc/ldap/ldap.conf
24. /etc/modules-load.d/cups-filters.conf
25. /etc/modules-load.d/modules.conf
26. /etc/bluetooth/network.conf
27. /etc/bluetooth/input.conf
28. /etc/bluetooth/main.conf
29. /etc/security/limits.conf
30. /etc/security/access.conf
31. /etc/security/namespace.conf
```

```
32. /etc/security/capability.conf
33. /etc/security/pwquality.conf
34. /etc/security/sepermit.conf
35. /etc/security/pam_env.conf
36. /etc/security/time.conf
37. /etc/security/group.conf
38. /etc/selinux/semanage.conf
39. /home/nicholas/.config/Trolltech.conf
40. /home/nicholas/.config/xsettingsd/xsettingsd.conf
```

## For Ruby Script

Output of program:

```
1.  /etc/GNUstep/GNUstep.conf
2.  /etc/NetworkManager/NetworkManager.conf
3.  /etc/NetworkManager/conf.d/default-wifi-powersave-on.conf
4.  /etc/PackageKit/PackageKit.conf
5.  /etc/PackageKit/Vendor.conf
6.  /etc/UPower/UPower.conf
7.  /etc/adduser.conf
8.  /etc/alsa/conf.d/10-samplerate.conf
9.  /etc/alsa/conf.d/10-speexrate.conf
10. /etc/alsa/conf.d/50-arcam-av-ctl.conf
11. /etc/alsa/conf.d/50-jack.conf
12. /etc/alsa/conf.d/50-oss.conf
13. /etc/alsa/conf.d/50-pulseaudio.conf
14. /etc/alsa/conf.d/60-upmix.conf
15. /etc/alsa/conf.d/60-vdownmix.conf
16. /etc/alsa/conf.d/98-usb-stream.conf
17. /etc/alsa/conf.d/99-pulse.conf
18. /etc/apache2/conf-available/javascript-common.conf
19. /etc/apparmor/parser.conf
20. /etc/apport/crashdb.conf
```

Not 21st line… jumps to end of output and then last

```
21. /usr/share/pulseaudio/alsa-mixer/profile-sets/native-instruments-traktor-
    audio2.conf
22. /usr/share/pulseaudio/alsa-mixer/profile-sets/native-instruments-traktor-
    audio6.conf
23. /usr/share/pulseaudio/alsa-mixer/profile-sets/native-instruments-traktorkontrol-
    s4.conf
24. /usr/share/pulseaudio/alsa-mixer/profile-sets/sb-omni-surround-5.1.conf
25. /usr/share/pulseaudio/alsa-mixer/profile-sets/steelseries-arctis-common-usb-
    audio.conf
26. /usr/share/pulseaudio/alsa-mixer/profile-sets/usb-gaming-headset.conf
27. /usr/share/qtchooser/qt4-x86_64-linux-gnu.conf
28. /usr/share/qtchooser/qt5-x86_64-linux-gnu.conf
29. /usr/share/rsyslog/50-default.conf
30. /usr/share/samba/smb.conf
31. /usr/share/sddm/themes/breeze/theme.conf
32. /usr/share/ufw/ufw.conf
33. /usr/share/upstart/sessions/session-migration.conf
34. /usr/src/linux-headers-5.4.0-51-generic/include/config/auto.conf
35. /usr/src/linux-headers-5.4.0-51-generic/include/config/tristate.conf
36. /usr/src/linux-headers-5.4.0-52-generic/include/config/auto.conf
37. /usr/src/linux-headers-5.4.0-52-generic/include/config/tristate.conf
38. /usr/src/virtualbox-guest-6.1.10/dkms.conf
39. /var/lib/ucf/cache/:etc:rsyslog.d:50-default.conf
40. /var/lib/ucf/cache/:etc:samba:smb.conf
```

## Testing and Comments

The testing of these three different scripts was different than other cases in this assignment as there is not a concrete expected answer. Instead of doing this I had a bash script as shown below, which ran each of the scripts and then piped the output of the scripts to a grep which printed out every line that didn't end in ".conf". If there was anything outputted by this bash script, the programs failed.

```
1.  #!/bin/bash
2.
3.  ./bashS | grep -ve ".conf$"
4.  ./perlS 2>/dev/null | grep -ve ".conf$"
5.  ruby rubyS | grep -ve ".conf$"
```

## Reflection

Writing the bash script was very easy as the language is very orthogonal with there being lots of commands to use that are very good at what they need to do and nothing else. The commands are expected to be "piped" or used in conjunction with other commands to create more complex programs. This made writing very easy as the find command does what it needed to without having to rely on much. It also makes reading very easy in this case as its only one line and has very basic syntax. Because of this its very reliable, its very clear what it does so it can be assumed to work.

There are some trade-offs with orthogonality, in some cases it can be detriment to simplicity of the language and make it less readable/writeable/reliable. In this case however, because it was short I had no problems.

Working with the Ruby script it is noticeable that everything feels Object-like and as such, a lot of the functions and methods run derive from type of data dealt with it in the case of the filename and seeing if the string ends with ".conf". This can be obvious from looking at the put command which places something on the output, as if it was interacting with an object/data structure itself.

Because of this abstraction Ruby is very readable in the sense that as things don't have to be recoded and there is additional code re-use. It is also much more writeable as there is less to write and also all the syntax is made in a way that looks as if it is working with objects, making it easier to remember the syntax and how to write the different commands. The increase in readability and writability increases reliability as less code means there is less of a chance for error.

Perl was not too fun to program in, the implementation was very different to the other two scripting languages and did not adhere to the simplicity principle at all. There was so many ways you could do anything you wanted, which made it very confusing which was the best way to make it happen.

Even then the approach I took has a lot of weird and complex syntax and structure which makes it hard to read, and also was very hard to write in the first place, figuring out what

was going on. In turn this affected reliability of the language as its very hard to tell if what you have works until you try it.

## Question

Perl was the hardest one to program in. There were so many ways to figure out how to write this program and even when I decided on using the find with the submodule called “wanted” (found in the docs) it was so different to the other languages and far more complex syntax wise, but also just had a different structure than I was used to.

Also with this Perl implementation I had to use regex’s which I did not have to deal with in Bash and Ruby who either had a command that searched (grep) or Ruby which had a command to look a the end of the string.

# Smalltalk

## Program

```
1.  1 to: 100 do: [:x|
2.      “if mod of 15 is 0”
3.      (x \\ 15 == 0)
4.          ifTrue: [ 'fizz buzz' printNl ]
5.          ifFalse: [
6.              “if mod of 3 is 0”
7.              (x \\ 3 == 0)
8.                  ifTrue: [ 'fizz' printNl ]
9.                  ifFalse: [
10.                     “if mod of 5 is 0”
11.                     (x \\ 5 == 0)
12.                         ifTrue: [ 'buzz' printNl ]
13.                         “otherwise print the number”
14.                         ifFalse: [ x printNl ]
15.                 ]
16.         ]
17.     ]
```

## Output of Program vs Expected

Output of program:

```
1.  1
2.  2
3.  'fizz'
4.  4
5.  'buzz'
6.  'fizz'
7.  7
8.  8
9.  'fizz'
10. 'buzz'
11. 11
12. 'fizz'
13. 13
14. 14
15. 'fizz buzz'
16. 16
17. 17
18. 'fizz'
19. 19
20. 'buzz'
```

```
81. 'fizz'
82. 82
83. 83
84. 'fizz'
85. 'buzz'
86. 86
87. 'fizz'
88. 88
89. 89
90. 'fizz buzz'
91. 91
92. 92
93. 'fizz'
94. 94
```

Expected output:

```
1.  1
2.  2
3.  fizz
4.  4
5.  buzz
6.  fizz
7.  7
8.  8
9.  fizz
10. buzz
11. 11
12. fizz
13. 13
14. 14
15. fizzbuzz
16. 16
17. 17
18. fizz
19. 19
20. buzz
```

```
81. fizz
82. 82
83. 83
84. fizz
85. buzz
86. 86
87. fizz
88. 88
89. 89
90. fizzbuzz
91. 91
92. 92
93. fizz
94. 94
```

```
95. 'buzz'
96. 'fizz'
97. 97
98. 98
99. 'fizz'
100.  'buzz'
```

```
95. buzz
96. fizz
97. 97
98. 98
99. fizz
100.  buzz
```

## Testing and Comments

The output for this file matched what it should've been with a few differences. As seen Smalltalk outputs the strings with '' around them, which is just how its implemented in the language and can be ignored. Other than that, the program outputs as expected. One thing to note it the funny indexing on 100 with buzz. This is due to the syntax highlighting program I used with 100 being one character longer so this can be ignored.

In order to test that the fizzbuzz program worked in Smalltalk, the following script was used in conjunction with an output file created by a similar python script to the one at the beginning of the document with syntax adjustments made.

```
1.  #!/bin/bash
2.
3.  gst fizzbuzz.st | diff fizzbuzz_ans.txt -
4.
5.  echo "  Test done: if no output, test passed"
```

## Reflection

Smalltalk is a fairly pure OO language and as such was an interesting language to try programming in. The OO nature of being very abstract did not help in this particular application with the fizzbuzz program. Although in some cases this dedication to the abstraction principle may help, it made this program very hard to read. If statements did not have a syntax for specifying what statement is happening, instead just having the evaluation itself creating a Boolean which is then operated on whether it's true or false.

Because of this more abstract and OO driven structure, the program is very hard to read and know what is happening. It is also very hard to write and solve problems when there is no else-if statement, instead having to write another nested if statement to get the same job done. Because of this lack of readability and writability, reliability is decreased for Smalltalk in this application as errors are more likely going to occur because of complexity that occurs.

## Question

Ironically, despite being a newer language the Smalltalk implementation of fizzbuzz was much worse in terms of readability and writability. If statements work very strangely in the sense there is no indication an evaluation is occurring like Fortran or Algol, this makes it very hard to understand what is happening and makes it harder to both read and write. There is also no else-if statement in Smalltalk which means you have to implement that yourself, which goes against abstraction and makes it harder to see what is happening, affecting again both the readability and writability of the program.

# C++

## Program (Cpp file) [4]

```cpp
#include <iostream>
#include <bits/stdc++.h>

#include "booksort.h"

int main() {
    Book books[5];

    books[0].SetBookID(3);
    books[0].SetBookName("Harry Potter");
    books[1].SetBookID(6);
    books[1].SetBookName("Percy Jackson");
    books[2].SetBookID(2);
    books[2].SetBookName("The Hobbit");
    books[3].SetBookID(5);
    books[3].SetBookName("Lord of the Rings");
    books[4].SetBookID(1);
    books[4].SetBookName("Othello");

    int IDs[sizeof(books)/sizeof(Book)];
    Book output[sizeof(books)/sizeof(Book)];


    //store indexes in an array in order
    for (int i = 0; i < sizeof(books)/sizeof(Book); i++)
    {
        IDs[i] = books[i].GetBookID();
    }

    quickSort(IDs, sizeof(IDs)/sizeof(int));

    //rebuild the array of books from sorted indexes
    for (int i = 0; i < sizeof(books)/sizeof(Book); i++)
    {
        for (int j = 0; j < sizeof(IDs)/sizeof(int); j++)
        {
            if(books[i].GetBookID() == IDs[j]) {
                output[j] = books[i];
            }
        }
    }

    for(int i=0; i < sizeof(output)/sizeof(Book); i++) {
        std::cout << output[i].GetBookID() << " ";
        std::cout << output[i].GetBookName();
        std::cout << std::endl;
    }

    //delete[] IDs;
    return 0;
}

int* quickSort(int* A, int length) {
    A = quickSortRecurse(A, 0, length - 1);
    return A;
}

//taken from lecture notes for DSA (see references)
int* quickSortRecurse(int* A, int leftIdx, int rightIdx) {
```

```
60.     int pivotIdx, newPivotIdx;
61.
62.     if(rightIdx > leftIdx) {
63.         pivotIdx = leftIdx;
64.         newPivotIdx = doPartitioning(A, leftIdx, rightIdx, pivotIdx);
65.
66.         quickSortRecurse(A, leftIdx, newPivotIdx -1);
67.         quickSortRecurse(A, newPivotIdx + 1, rightIdx);
68.     }
69.
70.     return A;
71. }
72.
73. //taken from lecture notes for DSA (see references)
74. int doPartitioning(int* A, int leftIdx, int rightIdx, int pivotIdx) {
75.     int newPivotIdx, pivotVal, currIdx, temp;
76.
77.     pivotVal = A[pivotIdx];
78.     A[pivotIdx] = A[rightIdx];
79.     A[rightIdx] = pivotVal;
80.
81.     currIdx = leftIdx;
82.     for(int ii = leftIdx; ii <= rightIdx - 1; ii++) {
83.         if(A[ii] < pivotVal) {
84.             temp = A[ii];
85.             A[ii] = A[currIdx];
86.             A[currIdx] = temp;
87.             currIdx++;
88.         }
89.     }
90.     newPivotIdx = currIdx;
91.     A[rightIdx] = A[newPivotIdx];
92.     A[newPivotIdx] = pivotVal;
93.
94.     return newPivotIdx;
95. }
96.
97. //Book constructor
98. Book::Book() {
99.     bookID = 0;
100.         bookName = "Othello";
101.         ISBN = "12345678";
102.     }
103.
104.     void Book::SetBookID(int iID) {
105.         bookID = iID;
106.     }
107.
108.     void Book::SetBookName(std::string iBookName) {
109.         bookName = iBookName;
110.     }
111.
112.     int Book::GetBookID() {
113.         return bookID;
114.     }
115.
116.     std::string Book::GetBookName() {
117.         return bookName;
118.     }
119.
120.     //Destructor
121.     Book::~Book() {
122.     }
```

## Program (Header file)

```
1.  #pragma once
2.
3.  class Book
4.  {
5.  private:
6.      int bookID;
7.      std::string bookName;
8.      std::string ISBN;
9.  public:
10.     int GetBookID();
11.     std::string GetBookName();
12.     std::string GetISBN();
13.     void SetBookID(int);
14.     void SetBookName(std::string);
15.     void SetBookISBN(std::string);
16.     Book();
17.     ~Book();
18. };
19.
20. int* quickSort(int* A, int length);
21. int* quickSortRecurse(int* A, int leftIdx, int rightIdx);
```

## Output of Program vs Expected

For input list (3, 6, 2, 5, 1)

Output of program:

```
1.  1 Othello
2.  2 The Hobbit
3.  3 Harry Potter
4.  5 Lord of the Rings
5.  6 Percy Jackson
```

Expected output:

```
1.  1 Othello
2.  2 The Hobbit
3.  3 Harry Potter
4.  5 Lord of the Rings
5.  6 Percy Jackson
```

For input list (1, 2, 3, 5, 6)

Output of program:

```
6.  1 Othello
7.  2 The Hobbit
8.  3 Harry Potter
9.  5 Lord of the Rings
10. 6 Percy Jackson
```

Expected output:

```
1.  1 Othello
2.  2 The Hobbit
3.  3 Harry Potter
4.  5 Lord of the Rings
5.  6 Percy Jackson
```

For input list (6, 5, 3, 2, 1)

Output of program:

```
11. 1 Othello
12. 2 The Hobbit
13. 3 Harry Potter
14. 5 Lord of the Rings
15. 6 Percy Jackson
```

Expected output:

```
6.  1 Othello
7.  2 The Hobbit
8.  3 Harry Potter
9.  5 Lord of the Rings
10. 6 Percy Jackson
```

## Testing and comments

Testing this program is a bit more complex than the other languages in this assignment. The actual books along with their values are hard coded inside of the c++ file itself. You can see this above from lines 9 to 18. In order to test the program with the books in different orders to be sorted, the books were assigned different indexes inside of the array which is then

sorted. The three test cases were inputted independently within the program and then run using the following script. The output of the file was compared with a file that just contained the expected output (shown above in the test cases). If nothing was outputted, the program worked as expected

```
1.  #!/bin/bash
2.
3.  g++ booksort.cpp
4.  ./a.out | diff ans.txt -
5.  echo "  Test Finished: if no output above, test passed"
6.  rm a.out
```

## Reflection

C++ was a nice language to program in, it was very similar to C so it felt familiar. The inclusion of objects in the language which was ostentatiously used within this particular sorting program. The usage of objects supports the abstraction principle as you can share objects for some code re-use instead of having to write code out multiple times. This reduces the clutter in the code, making it easier read and because of having to write less, also makes it more writeable. Because of a shorter and easier to read program, the reliability of the program is also increased.

Within this particular program however, I had to use pointers due to passing the array around. Pointers have a wide range of issues, in this particular context this breaks the preservation of information principle due to the array becoming a pointer when passed into a function. This unnecessarily increases the complexity of the program, making it harder to read due to the user not seeing it as an array but a pointer in the next function. This makes it less writeable as now pointer arithmetic is used on it instead of array style syntaxes. Pointers make a program less reliable due to side effects that occur to pointers doing unrestricted operations within memory.

## Question

The way in which objects work in c++ is quite similar with Java. It didn't feel totally dissimilar with its use of constructors, accessors and mutators. I'd say because of this objects would work in a similar way behind the scenes.

Some differences do exist between the two, for instance c++ has the de-constructor class because of memory management and tie in with C. Also the implementation of the class and methods did not exist in their own file like we're used to in Java. These differences would affect how objects work within c++, almost making it seem like c++ isn't such as much of an object orientated language when compared to a language like Java or small talk.

# Prolog

## Program

```
% base case is 1 for recursion
fizzbuzz(1) :- !, write('1'), nl.
fizzbuzz(N) :-
    (   % This is an if statement. With else at the end with N >0
        0 is mod(N,15), N1 is N-1, fizzbuzz(N1), write('fizzbuzz'), nl;
        0 is mod(N,3), N1 is N-1, fizzbuzz(N1), write('fizz'), nl;
        0 is mod(N,5), N1 is N-1, fizzbuzz(N1), write('buzz'), nl;
        N > 0, N1 is N-1, fizzbuzz(N1), write(N), nl
    ).
```

## Output of Program vs Expected

Output of program:

```
1
2
fizz
4
buzz
fizz
7
8
fizz
buzz
11
fizz
13
14
fizzbuzz
16
17
fizz
19
buzz
```

```
fizz
982
983
fizz
buzz
986
fizz
988
989
fizzbuzz
991
992
fizz
994
buzz
fizz
997
998
fizz
buzz
```

Expected output:

```
1
2
fizz
4
buzz
fizz
7
8
fizz
buzz
11
fizz
13
14
fizzbuzz
16
17
fizz
19
buzz
```

```
fizz
982
983
fizz
buzz
986
fizz
988
989
fizzbuzz
991
992
fizz
994
buzz
fizz
997
998
fizz
buzz
```

## Testing and Comments

The output of the program matched with what was expected. One thing to note it the funny indexing on 100 with buzz. This is due to the syntax highlighting program I used with 100 being one character longer so this can be ignored.

In order to test that the fizzbuzz program worked in Prolog a script was made as followed. This is different than the other fizzbuzz programs due to not being able to echo a string inside of the input of the program without it having strange results. In order to test it, instead this would open the prolog command line of the program where “fizzbuzz(1000).” Was entered which prompted the result to be displayed. In this case to test the first 20 and last 20 lines, “fizzbuzz(20).” was also entered as I couldn’t scroll up to the top of the output to see if the first 20 lines were correct when I executed the full 1000 lines of fizzbuzz.

```
1.  #!/bin/bash
2.
3.  gplc fizzbuzz.pl
4.  ./fizzbuzz
5.
6.  rm fizzbuzz
```

## Reflection

Prolog was a hard one to program in; despite my program being quite sure it was difficult to understand the syntax in order to write it and even read. One of the issues I found was the differing syntax for some of the maths operators, checking if the modular of an integer is 0 had a very different syntax to checking if an integer is bigger than 0. This irregular and inconsistent syntax is much harder to read as they look very different. It’s also hard to remember the different specialised rules when writing the functions. This effects reliability negatively as its more difficult to understand the program and can lead to issues with misunderstanding the output.

Another issue with the syntax is the complexity of conditional statements syntax. There is no actual “if”, “elseif” or “else” statements, despite the fact that is what the lines are doing in order after the brackets. This is incredibly hard to understand when reading the program with no context.

## Question

Writing a fizzbuzz program in prolog was not easy, especially when compared to the other languages that were explored in this unit. The major reason this was more difficult is that fizzbuzz is a rather procedural concept, whether prolog is not procedural but instead, logic based.

The lack of loops in Prolog, mainly because of loops being quite procedural, meant that the iteration through 1 to 1000 had to be done recursively. This once again made the

programming of fizzbuzz harder in program than Fortran and such which could handle iterative loops due to them being procedural.

# Scheme

## Program

```
1.  ; Function that checks if something is an atom or not
2.  (define (atom? x)
3.      (and (not (null? x))
4.           (not (pair? x))))
5.
6.  ;Taken from lecture notes, finds length of list
7.  (define (length list)
8.      (cond (( null? list) 0 )
9.          (( atom? list) 1 )
10.         ( else
11.         ( + 1 ( length (cdr list))))))
12.
13. ; Outer function for sort. Sorts list, makes the list shorter by one on the left
    and passed this back into the function to be sorted.
14. (define (sort list)
15.     (if (EQ? (length list) 1)
16.         list
17.         (cons (car (sort_inner list)) (sort (cdr (sort_inner list))))
18.     )
19. )
20.
21. ; inner function for sort. Does a bubble sort and then reverses the list, performs
    opposite which does the bubble sort the opposite way, and then flips the list
    again
22. (define (sort_inner list)
23.     (reverse (opposite (reverse
24.         (if (EQ? (length list) 2)
25.             (if (< (car list) (cadr list))
26.                 list
27.                 (cons (cadr list) (cons (car list) `() ))
28.             )
29.             (if (< (car list) (cadr list))
30.                 (cons (car list) (sort_inner (cdr list)))
31.                 (cons (cadr list) (sort_inner (cons (car list) (cddr list))))
32.             )
33.         ))
34.     ))
35. )
36.
37. ; same as inner function sort except has a > instead of a <
38. (define (opposite list2)
39.     (if (EQ? (length list2) 2)
40.             (if (> (car list2) (cadr list2))
41.                 list2
42.                 (cons (cadr list2) (cons (car list2) `() ))
43.             )
44.             (if (> (car list2) (cadr list2))
45.                 (cons (car list2) (opposite (cdr list2)))
46.                 (cons (cadr list2) (opposite (cons (car list2) (cddr list2))))
47.             )
48.     )
49. )
```

## Output of Program vs Expected

For input list (1,2,3,4,5)

Output of program:

```
1. (1 2 3 4 5)
```

Expected output:

```
1. (1 2 3 4 5)
```

For input list (5,4,3,2,1)

Output of program:

```
1. (1 2 3 4 5)
```

Expected output:

```
1. (1 2 3 4 5)
```

For input list (3,2,5,4,1)

Output of program:

```
1. (1 2 3 4 5)
```

Expected output:

```
1. (1 2 3 4 5)
```

## Testing

The cocktail shaker sort was tested using the following script that would output the program three times with different lists provided for different test cases. The output of this should be compared with the expected output as done above.

```
1. #!/bin/bash
2.
3. echo "(sort \`(5, 4, 3, 2, 1))" | mit-scheme --load cocktailshaker.scm
4. echo "(sort \`(1, 2, 3, 4, 5))" | mit-scheme --load cocktailshaker.scm
5. echo "(sort \`(3, 2, 5, 4, 1))" | mit-scheme --load cocktailshaker.scm
```

## Reflection

Working in Scheme once you get over the recursion and funny syntax is very refreshing. It is incredible regular and consistent with syntax in what I dealt with in my program. All of the brackets were the regular () brackets, as well as everything being declared in the same way with as functions such as the math operations, if statements and just calling other methods/functions.

This very regular design made the program very easy to write in predominantly, there was no specialised syntax that had to be remembered per command making it very natural to write the sort in. It's quite readable as everything is consistent making it easy to follow the flow of the program and command structure. This in turn makes the language quite reliable.

There is a problem with the regular and simple design in the sense that because everything uses the () brackets, it can be very hard to see where the end of statements and functions are like the if statements and such. This makes is less readable and can make the program less reliable as you don't know what to expect from the output.

## Question

With scheme there are two data types you can use. Atoms and lists. However, when dealing with IO there are additional data types that pop up. For instance there are objects for input

streams and also characters when pulling from the input stream. This is not very with a programming language that indicates only having two data types.

The one that most breaks the idea of scheme is ports, such as input and output ports which are similar to pointers. Scheme does not deal with memory because of it being a functional language, so it's highly irregular for scheme to break that by using pointers for file IO.

Another issue with the regularity with file IO in scheme is that the declarations are less functional and seem to be more like variable declarations which is against the general design of what you see with the rest of Scheme.

# References


[1] Valorie Maxville, Class Lecture, Topic: "Intro and Sorting (Lecture One)", COMP1002, Curtin University – Bentley, Perth, "School of Elec Eng, Comp and Math Sci (EECMS)", 30 July 2019

[2] *"Bubble Sort – GeeksForGeeks"*, Sep. 9 2020. Accessed on: Nov. 11 2020. [Online]. Available: https://www.geeksforgeeks.org/bubble-sort/

[3] Nicholas Klvana-Hooper, Class Assignment, Topic: "Assignment S2 2019", COMP1000, Curtin University – Bentley, Perth, "School of Elec Eng, Comp and Math Sci (EECMS)", 21 October 2019

[4] Valorie Maxville, Class Lecture, Topic: "Advanced Sorting (Lecture 08)", COMP1002, Curtin University – Bentley, Perth, "School of Elec Eng, Comp and Math Sci (EECMS)", 06 October 2019