Programming Languages Assignment Report

Index page:

Fortran

Reflection:

Fortran violates the Labelling Programming Principle as all loop structures and goto statements use arbitrary number statement labels that hold no meaning other than being a position to go to and the numerical value of the statement value is to be made unique by the programmer makes it less writable as a language since the programmer needs to keep track of what numbers the programmer has used. The statement labels also make it less readable as people need to look at the statement labels to know when the loop structure ends. The uses of goto in Fortran violates the Structured Programming principle as there is no single entry and exit for loops simulated using gotos.

It is also less readable in Fortran as there is 3 different types of gotos that does different things which violates Syntactic Consistency since they look the same but don't do very similar operations. Fortran only allows up to 3 dimensional arrays which violates the Zero One Infinity principle. Fortran is violating the Regularity Principle by having implicit declaration of variables that start with the letter i, j, k, l, m or n as integer variables and all other variables are implicitly declared as reals if type of variable is not defined. Fortran also violates Defense in Depth since it doesn't have no type checking or exception handling.

Fortran is only a bit readable if the program were to be complex with the amount of gotos needed. It is not very writable because of its limited primitives and implicit type casting and the same goes to its reliability as a language since it only has gotos for loop so it will be hard to maintain the program in the future.

Fortran violating nearly all programming principles is kind of understandable since it was the first language that allowed floating point computation in hardware which lead to newer and better languages to be made.

Weekly question:

Fortran feels very different when compared to previous languages I have programmed in (C and Java) since Fortran's syntax is stricter with there being rules for how Fortran must be formatted with what is allowed in which column. I 100% understand now why Mark didn't allow gotos in OOPD after being forced by Fortran to use statement labels and gotos to simulate loops which look horrible to understand if the program was lengthy whereas Java and C has defined looping structures to use. The one thing I definitely didn't like was that the statement labels are just arbitrary numbers so it looks like it will be horrible to find where the line of code that has the numerical value in the statement label section for a goto statement has a statement label number to know the flow of the program

Code:

```
      program fizzbuzz

      integer currentNum, maxNum

      currentNum = 1
   60 maxNum = 100
   10 if (mod(currentNum, 15) .EQ. 0) then
        write (*,*) 'FizzBuzz'
   13 elseif (mod(currentNum, 3) .EQ. 0) then
        write (*,*) 'Fizz'
        elseif (mod(currentNum, 5) .EQ. 0) then
        write (*,*) 'Buzz'
      else
        write (*,*) currentNum
      endif

      if (currentNum .LT. maxNum) then
        currentNum = currentNum + 1
        GOTO 10
      endif

      stop
      end
```

Output: (Ran with the command “f77 fizzbuzz,f && ./a.out”)

```
           1
           2
 Fizz
           4
 Buzz
 Fizz
           7
           8
 Fizz
 Buzz
          11
 Fizz
          13
          14
 FizzBuzz
          16
          17
 Fizz
          19
 Buzz
 Fizz
          22
          23
 Fizz
 Buzz
          26
```

```
Fizz
          28
          29
FizzBuzz
          31
          32
Fizz
          34
Buzz
Fizz
          37
          38
Fizz
Buzz
          41
Fizz
          43
          44
FizzBuzz
          46
          47
Fizz
          49
Buzz
Fizz
          52
          53
Fizz
Buzz
          56
Fizz
          58
          59
FizzBuzz
          61
          62
Fizz
          64
Buzz
Fizz
          67
          68
Fizz
Buzz
          71
Fizz
          73
          74
FizzBuzz
          76
          77
Fizz
          79
Buzz
Fizz
          82
          83
Fizz
```

```
Buzz
          86
Fizz
          88
          89
FizzBuzz
          91
          92
Fizz
          94
Buzz
Fizz
          97
          98
Fizz
Buzz
```

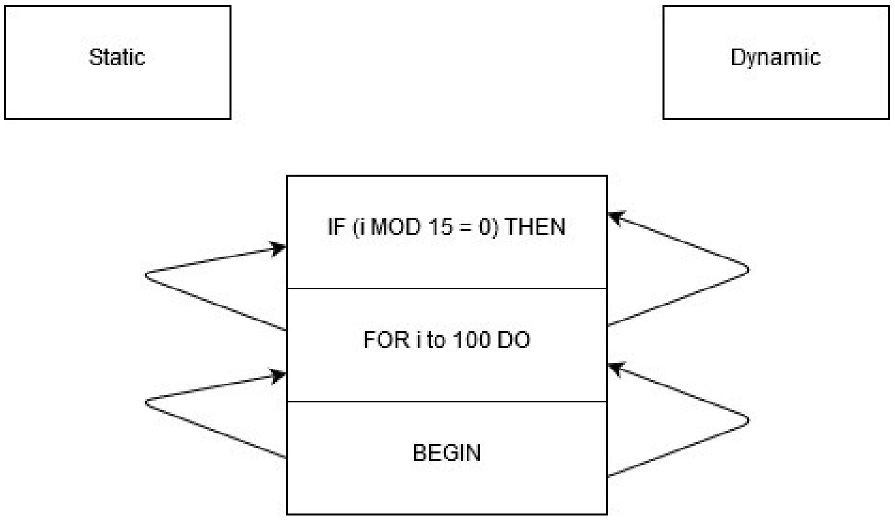
Algol

Reflection:

Algol complies with structured programming as it has built in control structures and don't need gotos to simulate loops. The use of "=" instead of Fortran's ".EQ." makes Algol more readable than Fortran. Algol is machine independent and has reserved keywords which make it better than Fortran in terms of writability since you can't accidentally use IF as a variable. Algol is really regular since it aimed to be regular hence obeying the regularity principle with examples like every control structure must have their respective "end" keyword to specify the end of a control structure like "OD" for "DO.

I think Algol is a lot more readable, writable and reliable than Fortran the language it tried to succeed and being machine independent. It complies with more programming principle than Fortran such as zero-one-infinity principle as you can have any number of dimensions for an array. However, the biggest downfall of Algol was having no built-in IO and users need to write their own IO for Algol to have IO and no double data type.

Weekly question:



Code:

```
BEGIN
      FOR i TO 100
      DO
            IF (i MOD 15 = 0) THEN
                  print(("FizzBuzz", new line))
            ELIF (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
      OD
END
```

Output: (Ran with the command: "a68g fizzbuzz.a68"

```
          +1
          +2
Fizz
          +4
Buzz
Fizz
          +7
          +8
Fizz
Buzz
         +11
Fizz
         +13
         +14
FizzBuzz
         +16
         +17
Fizz
         +19
Buzz
Fizz
         +22
         +23
Fizz
Buzz
         +26
Fizz
         +28
         +29
FizzBuzz
         +31
         +32
Fizz
         +34
Buzz
Fizz
         +37
         +38
Fizz
Buzz
         +41
Fizz
         +43
         +44
FizzBuzz
         +46
         +47
Fizz
         +49
Buzz
Fizz
         +52
         +53
Fizz
Buzz
         +56
Fizz
         +58
```

```
        +59
FizzBuzz
        +61
        +62
Fizz
        +64
Buzz
Fizz
        +67
        +68
Fizz
Buzz
        +71
Fizz
        +73
        +74
FizzBuzz
        +76
        +77
Fizz
        +79
Buzz
Fizz
        +82
        +83
Fizz
Buzz
        +86
Fizz
        +88
        +89
FizzBuzz
        +91
        +92
Fizz
        +94
Buzz
Fizz
        +97
        +98
Fizz
Buzz
```

Ada

Reflection:

Ada compiles with the Information Hiding principle by hiding the code of a module being used since the person using the module only knows what the module does and how to call it and what to pass into it with but not necessarily know how or what is being done by the module to achieve its functionality. Information hiding is also done using packages since packages define what within a package can be seen by the outside world.

Ada is orthogonal by enforcing the keyword “loop” the loop structure and doesn’t have any other keywords that would specify a loop structure. Ada complies to regularity principle as it requires a semicolon at the end of every statement without any exceptions.

Ada’s syntax is really readable and quite writable since it is orthogonal, regular, complies with structured programming principle and has information hiding which is good. This means Ada is also quite reliable since it is easy to maintain Ada code and you can easily understand what the code is doing in Ada and easily modify the code.

Weekly question:

Ada’s implementation of bubblesort is very similar to an implementation of bubblesort in C. They are quite similar since C is based on Ada’s syntax aside from C using curly brackets for begin and end of a control structure like an if statement or loop while Ada uses the keyword “end <control structure>”. Small details like how syntax for assignment variable and declaration of variables are also different since in C you declare the datatype then the name of the variable instead of the other way around that is in in Ada. C’s loop structure fairly resembles the loop structures in Ada where you either loop until a condition is met with the “while” keyword and loop for a fixed number of times with the “for” keyword. Ada has built-in Boolean data type whereas in C you need to use an int as a Boolean.

Code:

```
with Ada.Text_IO;

procedure BubbleSort is
      package IO renames Ada.Text_IO;
arr : array(0..10) of Integer;
swapped : boolean;
n : Integer;
temp : Integer;
begin
      arr(0) := 11;
      arr(1) := 10;
      arr(2) := 9;
      arr(3) := 8;
      arr(4) := 7;
      arr(5) := 6;
      arr(6) := 5;
      arr(7) := 4;
      arr(8) := 3;
      arr(9) := 2;
      arr(10) := 1;
      n := 11;
      swapped := True;
      while swapped loop
            swapped := False;
            for i in Integer range 1..(n - 1) loop
                  if arr(i-1) > arr(i) then
                        temp := arr(i);
                        arr(i) := arr(i-1);
                        arr(i-1) := temp;
                        swapped := True;
                  end if;
            end loop;
            n := n - 1;
      end loop;


    for i in Integer range 0..10 loop
            IO.Put_Line(Integer'Image(arr(i)));
    end loop;

end BubbleSort;
```

Output: (Ran with the command “gnatmake bubblesort && ./bubblesort”

```
 1
 2
 3
 4
 5
 6
 7
 8
 9
 10
 11
```

Yacc and Lex

Reflection:

Yacc complies with the zero-one-infinity principle because it only allows 1 empty clause for the context free grammar part. This makes Yacc more readable as you don't need to keep track of multiple symbols that could be empty and make break the context free grammar you are trying to achieve. Yacc and Lex violates Syntactic Consistency principle as it uses "%%" as the delimiter between the declaration part, rules part and the program part and uses "%{" and "}%" for the start and end of some C code declaration, also uses % in the syntax for defining the tokens and the type of return values for the tokens. This makes Yacc and Lex less readable and maintainable as you need to keep track of "%%" and "%{ }%" that have different purposes.

Weekly question:

I would use a hash table to implement a symbol table and a hash function to get the hash value of a symbol and linear hashing for any collision. With using a hash function, it will be easy to find any symbol by using the hash function and go to the location of the hash value in the hash table. Each entry of the hash table will be either a struct or object depending on the compiler but each entry will have all of the information of the symbol in that entry.

Code for Yacc:

```
%{
#include <stdio.h>
#include <string.h>
int count = 0;
int array[100000];

void yyerror(const char *str)
{
      fprintf(stderr,"error: %s\n",str);
}
%}
%union{
      int value;
}
%type <value> num
%token <value> NUMBER
%token COMMA OSQUAREBRAC ESQUAREBRAC MINUS

%%
list:
      OSQUAREBRAC numbers ESQUAREBRAC

numbers:
        numbers COMMA num { array[count++] = $3; }
      | num { array[count++] = $1; }
      |

num:
      NUMBER { $$ = $1; }
      | MINUS NUMBER { $$ = $2 * (-1); }

%%

main()
{
      while(1)
      {
            count = 0;
            yyparse();

            int i,n,newn,temp;
            n = count;
            do
            {
                  newn = 0;
                  for(i = 1; i <= n-1; i++)
                  {
                        if(array[i-1] > array[i])
                        {
                              temp = array[i];
                              array[i] = array[i-1];
                              array[i-1] = temp;
                              newn = i;
                        }
                  }
                  n = newn;
            }while(n >= 1);

            printf("Sorted:[");
            for(i = 0; i < count; i++)
            {
                  if(i == count - 1)
                  {
                        printf("%d", array[i]);
                  }
                  else
                  {
                        printf("%d,", array[i]);
                  }
            }
            printf("]\n");
      }
      return 0;
}
```

Code for Lex:

```
%{
#include "y.tab.h"
#include <stdlib.h>
%}
%%
[0-9]+                  { yylval.value = atoi(yytext); return NUMBER;}
"["             { return OSQUAREBRAC; }
"]"                     { return ESQUAREBRAC; }
","                     { return COMMA; }
"-"                     { return MINUS; }
\n                      { return 0;}
"exit"                  { exit(0); }
[ \t]+                  { /* ignore whitespace */; }
%%
#ifndef yywrap
      yywrap() { return 1; }
#endif
```

Output: (Ran with the command: “yacc -d sorter.y && lex sorter.l && gcc -o sorter y.tab.c lex.yy.c -lfl && ./sorter”)

```
[2,0,19,-2,-20,200]
Sorted:[-20,-2,0,2,19,200]
[3,1,10,200,-10,-20]
Sorted:[-20,-10,1,3,10,200]
Exit
```

Scripting

Reflection:

Bash is orthogonal as you pass flags to a method you call in Bash if you want to add extra information for that method call and Bash complies the Labelling programming principle as you can pass flags in any order and the name of the flags are meaningful to what they do. Bash has distinct methods to perform certain operations and the use of flags makes it clear what you are passing to the command called so it is really readable and writable and reliable.

Ruby complies with Regularity principle as it follows its rule of being an OO language where any method outside its standard library is called using the notation "classname.methodname" which signifies object orientation and indicates that you are calling a method of a class and not any default built-in method of standard Ruby. This makes Ruby very readable in terms of knowing which method is being called as you know the class that the method is from.

Perl uses very similar syntax to C++ that is a language that can do object orientation whereas Perl is an object oriented language. Perl is not orthogonal as it uses the "\&" to pass a reference to a method and "&&" is used as the AND operator so the character "&" is not used for a specific purpose. Perl is very writable with a wide variety of methods to do something but it is hard to maintain and hard to read since there is so many ways to do one thing and most people would likely only know one way or two and not all of the ways to do it.

Weekly question:

Perl was the hardest to write this program because I was already familiar with Bash from UCP and Bash is a shell language so it has simpler file related methods so Bash was the easiest for me, and it was easy to find a method that works and debug for me in Ruby. For Perl, I tried a few methods and I couldn't really understand why they didn't work and the error messages weren't helping before finding the File::Find method that works for me and was easy to understand how it works I guess.

Code for Bash:
```
#!/bin/bash
find / -name \*.conf
```

Code for Perl:
```
#!/usr/bin/perl
use File::Find;
find(\&wanted, "/");
sub wanted {
      if($File::Find::name =~ /\.conf$/)
      {
            print "$File::Find::name\n";
      }
}
```

Code for Ruby:
```
#!/usr/bin/env ruby
puts Dir.glob('/**/*.conf')
```

Output for Bash (truncated to 30 results):
```
/var/lib/NetworkManager/dhclient-eno1.conf
/var/lib/NetworkManager/NetworkManager-intern.conf
/var/lib/ucf/cache/:etc:idmapd.conf
/var/lib/ucf/cache/:etc:rsyslog.d:50-default.conf
/var/lib/ghc/package.conf.d/syb-0.7.conf
/var/lib/ghc/package.conf.d/parsec-3.1.11.conf
/var/lib/ghc/package.conf.d/network-2.6.3.2.conf
/var/lib/ghc/package.conf.d/case-insensitive-1.2.0.10.conf
/var/lib/ghc/package.conf.d/ghc-boot-th-8.0.2.conf
/var/lib/ghc/package.conf.d/ghci-8.0.2.conf
/var/lib/ghc/package.conf.d/regex-compat-0.95.1.conf
/var/lib/ghc/package.conf.d/directory-1.3.0.0.conf
/var/lib/ghc/package.conf.d/attoparsec-0.13.1.0.conf
/var/lib/ghc/package.conf.d/parallel-3.2.1.1.conf
/var/lib/ghc/package.conf.d/Cabal-1.24.2.0.conf
/var/lib/ghc/package.conf.d/ghc-8.0.2.conf
/var/lib/ghc/package.conf.d/hashable-1.2.6.1.conf
/var/lib/ghc/package.conf.d/async-2.1.1.1.conf
/var/lib/ghc/package.conf.d/regex-posix-0.95.2.conf
/var/lib/ghc/package.conf.d/unix-2.7.2.1.conf
/var/lib/ghc/package.conf.d/OpenGLRaw-3.2.7.0.conf
/var/lib/ghc/package.conf.d/deepseq-1.4.2.0.conf
/var/lib/ghc/package.conf.d/terminfo-0.4.0.2.conf
/var/lib/ghc/package.conf.d/containers-0.5.7.1.conf
/var/lib/ghc/package.conf.d/array-0.5.1.1.conf
/var/lib/ghc/package.conf.d/bytestring-0.10.8.1.conf
/var/lib/ghc/package.conf.d/transformers-0.5.2.0.conf
/var/lib/ghc/package.conf.d/haskell-src-1.0.2.0.conf
/var/lib/ghc/package.conf.d/split-0.2.3.2.conf
/var/lib/ghc/package.conf.d/StateVar-1.1.0.4.conf
```

Output for Perl (truncated to 30 results):
```
/var/lib/NetworkManager/dhclient-eno1.conf
/var/lib/NetworkManager/NetworkManager-intern.conf
/var/lib/ucf/cache/:etc:idmapd.conf
/var/lib/ucf/cache/:etc:rsyslog.d:50-default.conf
/var/lib/ghc/package.conf.d/syb-0.7.conf
/var/lib/ghc/package.conf.d/parsec-3.1.11.conf
/var/lib/ghc/package.conf.d/network-2.6.3.2.conf
/var/lib/ghc/package.conf.d/case-insensitive-1.2.0.10.conf
/var/lib/ghc/package.conf.d/ghc-boot-th-8.0.2.conf
/var/lib/ghc/package.conf.d/ghci-8.0.2.conf
/var/lib/ghc/package.conf.d/regex-compat-0.95.1.conf
/var/lib/ghc/package.conf.d/directory-1.3.0.0.conf
/var/lib/ghc/package.conf.d/attoparsec-0.13.1.0.conf
/var/lib/ghc/package.conf.d/parallel-3.2.1.1.conf
/var/lib/ghc/package.conf.d/Cabal-1.24.2.0.conf
/var/lib/ghc/package.conf.d/ghc-8.0.2.conf
/var/lib/ghc/package.conf.d/hashable-1.2.6.1.conf
/var/lib/ghc/package.conf.d/async-2.1.1.1.conf
/var/lib/ghc/package.conf.d/regex-posix-0.95.2.conf
/var/lib/ghc/package.conf.d/unix-2.7.2.1.conf
/var/lib/ghc/package.conf.d/OpenGLRaw-3.2.7.0.conf
/var/lib/ghc/package.conf.d/deepseq-1.4.2.0.conf
/var/lib/ghc/package.conf.d/terminfo-0.4.0.2.conf
/var/lib/ghc/package.conf.d/containers-0.5.7.1.conf
/var/lib/ghc/package.conf.d/array-0.5.1.1.conf
/var/lib/ghc/package.conf.d/bytestring-0.10.8.1.conf
/var/lib/ghc/package.conf.d/transformers-0.5.2.0.conf
```

```
/var/lib/ghc/package.conf.d/haskell-src-1.0.2.0.conf
/var/lib/ghc/package.conf.d/split-0.2.3.2.conf
/var/lib/ghc/package.conf.d/StateVar-1.1.0.4.conf
```

Output for Ruby (truncated to 30 results):

```
/var/lib/NetworkManager/dhclient-eno1.conf
/var/lib/NetworkManager/NetworkManager-intern.conf
/var/lib/ucf/cache/:etc:idmapd.conf
/var/lib/ucf/cache/:etc:rsyslog.d:50-default.conf
/var/lib/ghc/package.conf.d/syb-0.7.conf
/var/lib/ghc/package.conf.d/parsec-3.1.11.conf
/var/lib/ghc/package.conf.d/network-2.6.3.2.conf
/var/lib/ghc/package.conf.d/case-insensitive-1.2.0.10.conf
/var/lib/ghc/package.conf.d/ghc-boot-th-8.0.2.conf
/var/lib/ghc/package.conf.d/ghci-8.0.2.conf
/var/lib/ghc/package.conf.d/regex-compat-0.95.1.conf
/var/lib/ghc/package.conf.d/directory-1.3.0.0.conf
/var/lib/ghc/package.conf.d/attoparsec-0.13.1.0.conf
/var/lib/ghc/package.conf.d/parallel-3.2.1.1.conf
/var/lib/ghc/package.conf.d/Cabal-1.24.2.0.conf
/var/lib/ghc/package.conf.d/ghc-8.0.2.conf
/var/lib/ghc/package.conf.d/hashable-1.2.6.1.conf
/var/lib/ghc/package.conf.d/async-2.1.1.1.conf
/var/lib/ghc/package.conf.d/regex-posix-0.95.2.conf
/var/lib/ghc/package.conf.d/unix-2.7.2.1.conf
/var/lib/ghc/package.conf.d/OpenGLRaw-3.2.7.0.conf
/var/lib/ghc/package.conf.d/deepseq-1.4.2.0.conf
/var/lib/ghc/package.conf.d/terminfo-0.4.0.2.conf
/var/lib/ghc/package.conf.d/containers-0.5.7.1.conf
/var/lib/ghc/package.conf.d/array-0.5.1.1.conf
/var/lib/ghc/package.conf.d/bytestring-0.10.8.1.conf
/var/lib/ghc/package.conf.d/transformers-0.5.2.0.conf
/var/lib/ghc/package.conf.d/haskell-src-1.0.2.0.conf
/var/lib/ghc/package.conf.d/split-0.2.3.2.conf
/var/lib/ghc/package.conf.d/StateVar-1.1.0.4.conf
```

SmallTalk

Reflection:

Smalltalk complies with Regularity Principle as every block structure requires open square brackets for the start and close square brackets for the end of a block structure. This improves readability of Smalltalk's code as long as the programmer follows a fixed style guide and doesn't freely indents the square brackets or put them to a new line when it is not needed which will reduce readability. Smalltalk violates Syntactic Consistency principle as it uses ":" for variable assignment and also used after reserved keywords like "to:" so this makes Smalltalk less readable and writable as you need to know if you need the ":" character because the word you just typed is a reserved keyword or because you wanted variable assignment. This will make Smalltalk less reliable as it will be harder to maintain code since it is not very readable or writable when compared to other languages like Python.

Weekly question:

Smalltalk is more readable and writable than Fortran as Smalltalk complies with Structure Programming principle as Smalltalk has loop structure whereas Fortran uses goto for looping, so it is more readable and easier to write since it is easier to remember the flow of your program in Smalltalk. Algol is as writable as Smalltalk in terms of Fizzbuzz implementation as both have loop structure and explicit end of if statements. Algol is more readable than Smalltalk as Algol has the condition of the if statement within the if statement so it is easier to see what will be executed when it is true as the code to run when the condition is true will be directly below the if statement unlike Smalltalk that defines the condition then has the explicit if true or false statements afterwards which may make it hard to read as the order of "ifTrue" and "ifFalse" don't have a specific order where the "ifFalse" can be above the "ifTrue" statement which may make it hard to find what code will run if the condition is true.

Code:

```
#!/usr/bin/env gst

1 to: 100 do: [:count|
      (count \\ 15 == 0)
      ifFalse: [
            (count \\ 3 == 0)
            ifTrue: [ 'Fizz' printNl. ]
            ifFalse: [
                  (count \\ 5 == 0)
                  ifTrue: [ 'Buzz' printNl. ]
                  ifFalse: [ count printNl.
                  ]
            ]
      ]
      ifTrue: [ 'FizzBuzz' printNl. ]

]
```

Output: (Ran with the command: “./fizzbuzz.st”)

```
1
2
'Fizz'
4
'Buzz'
'Fizz'
7
8
'Fizz'
'Buzz'
11
'Fizz'
13
14
'FizzBuzz'
16
17
'Fizz'
19
'Buzz'
'Fizz'
22
23
'Fizz'
'Buzz'
26
'Fizz'
28
29
'FizzBuzz'
31
32
'Fizz'
34
'Buzz'
'Fizz'
37
38
'Fizz'
'Buzz'
41
'Fizz'
43
44
'FizzBuzz'
46
47
'Fizz'
49
'Buzz'
'Fizz'
52
53
'Fizz'
'Buzz'
56
'Fizz'
```

```
58
59
'FizzBuzz'
61
62
'Fizz'
64
'Buzz'
'Fizz'
67
68
'Fizz'
'Buzz'
71
'Fizz'
73
74
'FizzBuzz'
76
77
'Fizz'
79
'Buzz'
'Fizz'
82
83
'Fizz'
'Buzz'
86
'Fizz'
88
89
'FizzBuzz'
91
92
'Fizz'
94
'Buzz'
'Fizz'
97
98
'Fizz'
'Buzz'
```

C++

Reflection:

C++ doesn't feel as weird as the other OO languages. It is a bit intuitive like java's syntax but with C's style. You just declare the classfields like a struct and define the methods the class will have so its just a struct that has methods tied to it. C++ violates the Syntactic Consistency principle with its use of "<<" for bitvise operation and for passing data to output streams like standard output. C++ is just like what in the lecture slide says where C++ is just C that can do object orientation but its not a Object Oriented language. Although C++ violates the Syntactic Consistency principle in some ways, it does add more unique keywords like "new" for mallocing an object and "delete" for deallocating the object. This makes C++ more readable and writable and reliable as it is obvious when you are creating a new object. "<<" for writing to output streams is a bit readable as it visually kind of looks like the data is passed to the output stream with where the "<<" are pointing at. Using "<<" also makes it obvious when you are writing to a stream and not calling another method.

Weekly question:

Objects in C++ works like pointers that point to structs and have access to certain methods that other objects and programs won't have access to as it is exclusive to that object and its subclasses. This means C++ is just C that has more added functionality which is object orientation and the use of object orientation is done using pointers that have knowledge of classfields and methods of the class so everything is done through memory access. This means C++ can have repeated and multiple inheritance since everything is done through memory management so there are no rules like in Java where it being an OO language forces certain rules on objects like single inheritance and multiple parent classes using interfaces.

Code:

book.hpp
```
#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();
};
```

book.cpp
```
#include <string>
#include "book.hpp"
```

```
Book::Book()
{
	this->bookID = 1;
	this->bookName = "Book";
	this->ISBN = "ISBN";
}
int Book::GetBookID()
{
	return bookID;
}
std::string Book::GetBookName()
{
	return bookName;
}
std::string Book::GetISBN()
{
	return ISBN;
}
void Book::SetBookID(int bookID)
{
	this->bookID = bookID;
}
void Book::SetBookName(std::string bookName)
{
	this->bookName = bookName;
}
void Book::SetBookISBN(std::string ISBN)
{
	this->ISBN = ISBN;
}
Book::~Book()
{
}
```

Main.cpp

```
#include <string>
#include <cstdlib>
#include <iostream>
#include "book.hpp"
void quickSort(Book bookArray[], int lo, int hi);
int partition(Book bookArray[], int lo, int hi);
int main()
{
	Book bookArray[10];
	std::cout << "[";
	for(int i = 10; i > 0; i--)
	{
		bookArray[i-1].SetBookID(i);
		if(i == 1)
		{
			std::cout << bookArray[i-1].GetBookID() << "]" <<
std::endl;
		}
		else
		{
			std::cout << bookArray[i-1].GetBookID() << " ";
		}
	}
```

```
      quickSort(bookArray, 0, 9);
      std::cout << "[";
      for(int i = 0; i < 10; i++)
      {
            if(i == 9)
            {
                  std::cout << bookArray[i].GetBookID() << "]" <<
std::endl;
            }
            else
            {
                  std::cout << bookArray[i].GetBookID() << " ";
            }
      }
      return 0;
}
void quickSort(Book bookArray[], int lo, int hi)
{
      if(lo < hi)
      {
            int p = partition(bookArray, lo, hi);
            quickSort(bookArray, lo, p - 1);
            quickSort(bookArray, p + 1, hi);
      }
}
int partition(Book bookArray[], int lo, int hi)
{
      Book pivot = bookArray[hi];
      Book temp;
      int i = lo;
      for(int j = lo; j < hi; j++)
      {
            if(bookArray[j].GetBookID() < pivot.GetBookID())
            {
                  temp = bookArray[i];
                  bookArray[i] = bookArray[j];
                  bookArray[j] = temp;
                  i++;
            }
      }
      temp = bookArray[i];
      bookArray[i] = bookArray[hi];
      bookArray[hi] = temp;
      return i;
}
```

Output: (Ran with the command: “make && ./Program”

```
[10 9 8 7 6 5 4 3 2 1]
[1 2 3 4 5 6 7 8 9 10]
```

Prolog

Reflection:

I got my prolog program working with a “main(1001).” at the top of my program after having issues with halting the recursive calls when it reaches 1000 and has printed out the result because when I tried to capture the output of my infinite program, it generates about 10MB of text per second.
The hard part about Prolog is that you need to list down rules that will halt the recursive calls when you want it to. The order of the statements does matter so that would increase the complexity of more complex algorithm unlike fizzbuzz. Prolog being only consisting of rules, facts and queries complies with the Simplicity principle as there can only be 3 different type of statements. This makes Prolog less writable as you need to reinvent the wheel a lot to do more complex algorithms. Prolog violates Information Hiding principle as every rule and fact will be looked through when trying to evaluate a query.

Weekly question:

After understanding the syntax of Prolog, I would say writing a FizzBuzz program in Prolog will be easier than the all of the other ones I have done since all you need is rules and facts that hold true when a number is divisible by 15, 3, 5 or not divisible by 3 and 5, and a rule that halts the recursive call when I have printed the result of number 1000. This makes Prolog really writable as you just need to break down an algorithm you need to write into logical statements and that is all, you don’t need to worry about scoping. You don’t need to worry about what variables you need for the entire program or what data type you need to declare them to be and only need to worry about what variable is needed per statement so the problem is simplified.

Code:

```
main(1001).
main(N):- fizzbuzz(N), N1 is N + 1, main(N1).
fizzbuzz(N) :- 0 is mod(N, 15), write('FizzBuzz'), nl.
fizzbuzz(N) :- 0 is mod(N, 3) , write('Fizz'), nl.
fizzbuzz(N) :- 0 is mod(N, 5), write('Buzz'), nl.
fizzbuzz(N) :- write(N), nl.
:- initialization(main(1)).
```

Output: (truncated to the first 30 lines of output): (Ran with the command: “gplc fizzbuzz.pro && ./fizzbuzz”)

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
```

```
FizzBuzz
16
17
Fizz
19
Buzz
Fizz
22
23
Fizz
Buzz
26
Fizz
28
29
FizzBuzz
```

Scheme

Reflection:

Scheme obeys the simplicity principle as there are only a few keywords and parenthesis for the syntax of Scheme, but this means we have to reinvent the wheel a lot to make the more complex algorithms and functions. This makes Scheme not really readable or writable as you need to write complex functions to do what other languages can do built in whereas procedural languages are at least much more readable but Scheme being a functional language guarantees no side-effects so it is really reliable in parallel computing and not very reliable in terms of maintaining the code since it is very unreadable with the amount of parenthesis needed to make it work and 1 pair less of parenthesis will make the program not work as intended so debugging is hard to do. Scheme violates the Preservation of Information Principle as the operation to perform on lists is stated before the list such as "(+ 1 2)". This way of representing information is completely different to how we perceive the math formula for addition, so this makes Prolog less readable.

Weekly question:

Scheme's File IO complies with Regularity principle because files are treated as ports when opened for reading or writing and then the file ports are treated like every other ports without exception and you need to use the port read and write methods to read from or write to the file ports and there is no port method that specifically doesn't work on file ports.

Code:

```
(define (check-sorted lst)
    (cond
        ((null? lst) #t)
        ((null? (cdr lst)) #t)
        ((or (< (car lst) (cadr lst)) (= (car lst) (cadr lst))) (check-
sorted (cdr lst)))
        (else #f))
)
(define (forward-sort lst)
    (cond
        ((null? lst) lst)
        ((null? (cdr lst)) lst)
        ((< (car lst) (cadr lst)) (cons (car lst) (forward-sort (cdr
lst))))
        (else
            (cons (cadr lst) (forward-sort (cons (car lst) (cddr
lst))))
           )
      )
)
(define (backward-sort lst)
    (cond
        ((null? lst) lst)
        ((null? (cdr lst)) lst)
        ((> (car lst) (cadr lst)) (cons (car lst) (backward-sort (cdr
lst))))
        (else
            (cons (cadr lst) (backward-sort (cons (car lst) (cddr
lst))))
           )
      )
)
(define (cocktail-sort lst)
    (cond
        ((check-sorted lst) lst)
        (else
            (cocktail-sort (reverse (backward-sort (reverse (forward-
sort lst)))))
        )
    )
)
(cocktail-sort '(10 9 8 7 6 4 2 1)))
```

Output: (Ran with the command: “scheme < cocktail-shaker-sort.scm” and only extracted out the sorted output since the line with the list to sort is at the bottom of the code above)

```
Value 13: (1 2 4 6 7 8 9 10)
```