Programming Languages Assignment  
Report

# Preface

Program source code is included in the appendices at the end of the document.

Program output is not included as part of the testing procedure descriptions, for brevity (also because I could have falsified them anyway). In several cases, the testing setup/data is included within the program, and the results may be verified by running the programs.

# Fortran 77

## Program Testing Procedure

Run program, observe console output. Every 3rd line should be "Fizz", every 5th line "Buzz", every 15th line "Fizz buzz". The other lines should contain the line number.

Also make sure output formatting is correct – Fortran is finnicky with print formatting.

## Weekly Question

The most striking difference between Fortran and languages I have used before is the column-based syntax. This is due to Fortran’s origins with punch-cards; ancient technology obviously not relevant to any modern language. Most of the languages I use structure syntax with whitespace, braces, semicolons, etc. (Additionally, modern compilers’/interpreters’ greater capabilities compared to Fortran’s likely enable their handling of more flexible syntax.)  
Further exhibiting Fortran’s primitiveness are the PROGRAM, STOP, and END statements, which inform the compiler where the program starts and ends. In any modern language you would take it for granted that the compiler/interpreter could figure this information out for itself.

However, for the simple program we had to code, Fortran is not that different to something like C89, albeit more primitive. IF/ELSEIF/ELSE is the same as in modern procedural languages. There is a DO loop, which is close enough to the modern “for” loop. The algebraic and boolean notation is largely the same as we use now (besides some differences in operator naming). The one function call to MOD() that I needed to use worked as I expected. Variables are declared with a type before use, the same as C.  
I suspect that Fortran’s difference would become even more apparent if I had to write a larger program, for example using subprograms.

## Reflection

Fortran’s violation of the labelling principle negatively affects the program’s readability, writability and reliability. Firstly, each line must obey a strict column-based format, with various columns indicating different things depending on their contents, none of which is labelled (you must know what column means what). Secondly, lines are labelled with numbers, not meaningful names (again, you must remember which number line does what). Additionally, the use of numerical line labels confuses statements such as DO: for example, it is not immediately clear what numbers mean what in the following statement: DO 10 i=1, 100  
Having to remember these arbitrary orders and labels reduces reliability; the programmer is likely to make a mistake and (knowing Fortran, silently) break the code.

Fortran violates the syntactic consistency principle, reducing readability. For example, the function call MOD(a, b) includes the parameters within the parentheses, but the WRITE statement, which looks like a function call, places the value to write outside the parentheses. Setting a variable is done with =, but equality checking is done with .EQ..

Fortran violates many other language principles, for example:

* Defence in depth: implicit variable declarations, ignored whitespace, existence of GOTO
* Information hiding: everything is global
* Security: existence of GOTO
* Structured program: existence of GOTO
* Zero-one-infinity: array dimensions restriction

But fortunately, they do not affect my program much or at all, due to the simplicity of the task. Note that while, for example, I used IF/ELSEIF/ELSE instead of GOTO, this does not mean Fortran in general adheres to the structured program principle.

# Algol 68

## Program Testing Procedure

Run program, observe console output. Every 3rd line should be "Fizz", every 5th line "Buzz", every 15th line "Fizz buzz". The other lines should contain the line number.

Also make sure the output formatting is correct – Algol, like Fortran, defaults to using interesting formatting.

## Weekly Question

## Reflection

Algol’s adherence to the regularity principle with respect to program structure improves readability. Since every part of the program consists of a hierarchical block structure that relates directly to scope, it is very easy to identify name binding.  
A corollary of this regularity is that the language also adheres to the structured program principle (at least for this small program, that does not need forward declarations). It is very easy to follow the control flow of the program (aided by the block nesting), increasing readability.

Algol’s disallowance of implicit variable declaration and usage of reserved keywords adheres to the defence in depth principle, improving reliability. While I was figuring out the syntax of the language, I made various keyword and name related mistakes, all of which were immediately detected by the compiler – once the program compiled successfully, I was confident the program executed as desired.

Some aspects of Algol’s syntax somewhat violate the syntactic consistency principle, reducing readability and writability (at least for a beginner).  
For example, the modulo operator is %\* - with the \* operator performing multiplication, and the % operator performing integer division. This makes little sense, as each of these is distinct operations.  
Equality checking is performed with the = operator, while assignment is performed with :=. Again, these are distinct operations sharing very similar syntax, and are likely to trip up those used to = for assignment, for example from Fortran or C.

Algol’s I/O formatting violates the simplicity and regularity principles, reducing readability and writability. The formatting behaviour of print() changes depending on the type of the argument – numbers are by default aligned, but strings are not. Furthermore, it is, in my opinion, too difficult to change these default behaviours. For example, in order to print a number n without any formatting, one can use print(whole(n, 0)). But in order to also include a trailing newline, one must use print((whole(n, 0), newline)) (note the extra parentheses), or printf(($%dl$, n)) (using the “format-text” syntax, which is quite complex and unintuitive).

# Ada

## Program Testing Procedure

Test bubble sort procedure with differently sized/ordered arrays and observe output:

|  |  |  |
| --- | --- | --- |
| **Test name** | **Test array** | **Expected output** |
| Already sorted | 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 | 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 |
| Reverse sorted | 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 | 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 |
| Random order | 7, 6, 3, 5, 9, 4, 8, 10, 1, 2 | 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 |
| Single element | 42 | 42 |
| Empty array | <nothing> | <nothing> |

Make sure console output formatting is correct (we still have not gotten to languages that default to printing numbers without alignment).

## Weekly Question

The general structure and control flow of the Ada and C versions are the same (or at least can easily be made the same). They both utilise nested control structures.  
The looping mechanisms differ slightly, however. For the outer loop, C will probably use a do…while, while Ada has loop and exit when. For the inner loop, C will probably use a for loop with an explicitly managed loop counter, while Ada has a slightly more expressive/abstract for…in that loops automatically over a range.

The overall syntax of the C version is somewhat more streamlined than Ada. C uses braces rather than Ada’s more verbose block keywords such as begin, end, loop, and then. Ada uses := for assignment, one more character than C’s =. Ada also has some constructs which may not strictly be necessary, for example using the range keyword before specifying start..end, and loop following for. It may be because I am far more used to C’s syntax, but I believe the extra verbosity in the Ada code makes it “noisier” and more difficult to read compared to the C version. However, this added verbosity likely improves reliability, a design goal for Ada.

Ada has a built-in array type, making the handling of arrays slightly easier, more abstract, and more reliable than the C version. Ada’s arrays can store their size, while C arrays are simply pointers with no additional information. Due to this, the C version, if defined as a procedure, must accept another parameter specifying the size of the array. Additionally, the C version is more prone to bugs because of the programmer providing the wrong array size. The Ada version has no such problem, as the size is handled by the language itself.

Finally, (modern) C allows variable definitions at any point in a procedure, whereas Ada requires them to be before the procedure body. Requiring the definitions to be before the body reduces readability, writability, and reliability, as it is possible that variables’ scope must be inflated beyond where they are used. For example, the Tmp variable in the Ada version is in scope for the entire procedure, even though it is only used within the inner if block. In C, this variable could be constrained to where it is used.

## Reflection

Ada’s compliance with the abstraction principle with respect to arrays and loops improves readability, writability, and reliability. Array types store the range of indices which are valid as the Range member, which can be used with the for…in… loop to automatically loop over the indices. Such functionality is very easy to use since there is no need to manually manage the loop variable. It is similarly easy to read, as the functionality is intuitive and concise. Finally, since the loop variable an exit condition is handled by the compiler, it is less prone to bugs because of human error.

Although not demonstrated in this program, Ada’s modules comply strongly with the information hiding principle, increasing the readability and reliability of code. Modules enable implementation details to be “hidden” from the rest of the code, with only a specific interface exposed. This information hiding is enforced by the compiler, making it very difficult (or impossible) to circumvent.

However, Ada’s exception handling (also not demonstrated in this program) violates the information hiding principle, since it is not possible to catch a “generic” exception. As a result, users of modules are required to know the exact exception type – a module implementation detail – when handling exceptions.

# Yacc and Lex

## Program Testing Procedure

Run program, try various inputs, and observe parsed and sorted list data. Focus on parsing behaviour; I assume std::sort() works correctly. Make sure all valid syntax is accepted, but also that all invalid syntax is rejected.

|  |  |  |
| --- | --- | --- |
| **Test name** | **Test input** | **Expected behaviour** |
| Empty list | [] | Parsed: []  Sorted: [] |
| 1 element | [42] | Parsed: [42]  Sorted: [42] |
| 2 elements | [42, 7] | Parsed: [42, 7]  Sorted: [7, 42] |
| Many elements | [5, 3, 8, 3, 2, 9] | Parsed: [5, 3, 8, 3, 2, 9]  Sorted: [2, 3, 3, 5, 8, 9] |
| Negative numbers | [-1, -4, -2, -2] | Parsed: [-1, -4, -2, -2]  Sorted: [-4, -2, -2, -1] |
| Long numbers | [1234567, 86432, -10000, 999999] | Parsed: [1234567, 86432, -10000, 999999]  Sorted: [-10000, 86432, 999999, 1234567] |
| Various whitespace | [1,2, -1, -4,6,3 , 8 ] | Parsed: [1, 2, -1, -4, 6, 3, 8]  Sorted: [-4, -1, 1, 2, 3, 6, 8] |
| Leading and trailing whitespace | [87, 53, 26, 3, 5, 3] | Parsed: [87, 53, 26, 3, 5, 3]  Sorted: [3, 3, 5, 26, 53, 87] |
| Unmatched brace | [1, 2, 3, 4 | Syntax error |
| Extra comma 1 | [1, 2, 3, 4,] | Syntax error |
| Extra comma 2 | [,] | Syntax error |
| Non-numeric values | [a, foo, bar] | Syntax error |
| Just whitespace |  | Syntax error |
| Multiple lists | [1, 2, 3] [3, 2, 1] | Syntax error |
| Leading garbage | foo[1, 2, 3] | Syntax error |
| Trailing garbage | [1, 2, 3]foo | Syntax error |
| Malformed numbers | [1, 2, 43e] | Syntax error |

## Weekly Question

I would implement a symbol table with a hash table, mapping from names to AST nodes. To allow nested scopes, multiple hash tables could be used, stored in order of nesting, in list. To retrieve a name, the list is traversed in reverse (going up the tree of scopes) from the scope of the name usage, until a match is found.  
If it is important that symbols may be used only after declaration, then name resolution may be done as soon as possible (e.g. one-pass compilers such as C), or, the ordering of name declarations is stored in the symbol table as well.

## Reflection

Lex and Bison’s file structure violates the structured program principle, reducing readability and reliability. The grammar definitions are declarative in nature, with not much notion of control flow (that is controlled by the Lex and Yacc internals). However, the flow of control is important to consider since the rule actions manipulate shared global state. It can be difficult to discern the ordering of execution and how different parts of the code interact, leading to poor readability and increased probability of bugs. I had trouble making syntax error handling work as I desired, since it was difficult to determine what state the program was in when an error occurs.

The nature of Lex and Bison as code generators / preprocessors violates the syntactic consistency principle, reducing readability. The files consist of effectively three syntaxes, all intermixed: C or C++ code (with the ability to also use the C preprocessor!), grammar definition (including a regex-like syntax), and the supporting Lex and Yacc file structure and setup. Naturally, there are many syntax conflicts between them. In addition, C, and especially C++, by themselves are not particularly syntactically consistent. The result is that the semantics of any given portion of the program is highly context-dependent and potentially difficult to understand.

# Scripting Languages

## Program Testing Procedure

Run programs and observe output. Each matched file should be output on a separate line. Make sure only files ending in “.conf” are matched (not directories). Unfortunately, cannot test that all such files are found, as the whole filesystem is too big to search in a reasonable amount of time using scripting languages.

## Weekly Question

Perl was the hardest to write program in. This is because I had the least experience with and knowledge of Perl, in addition to Perl being quite tricky to learn/decipher.

In Bash, I effectively already knew how to do the task, so a brief search online and review of the man pages allowed me to write it quite quickly. Finding files is a typical task for a shell language like Bash, so there is a well-established, well-documented, well-designed, single way to do it (find). The only Bash syntax that was required to navigate was invoking a program with arguments, which is trivial.

I had seen snippets of Ruby previously, so the general syntax was not particularly new to me. After an online search for how to find files, I quickly found documentation for a library function that recursively iterates over directories and files (Find.find()). The documentation included a code example, which, thanks to Ruby’s simple, structured syntax, was very readable and easy to figure out. Finally, I knew from previous reading of the provided Ruby guide (and knowing that Ruby is a sane language) that the string class probably has some sort of “does this string end with x?” (String.end\_with?()) function, which I used to check if the file name ends with “.conf”. (I figured that using regexes would be a harder way to perform the task, since Ruby likely does not have any extremely convenient regex functionality like Perl.)

For Perl, however, I had no idea at all what to do – I did not know the syntax nor built-in/library functionality. Searching online for how to find files in Perl yielded several different ways to perform the task, none of which seemed to be particularly simple or intuitive. (I even found two third party libraries, to perform such a simple task!) Additionally, reading the documentation did not help me much, because Perl’s quirky syntax and behaviour made it difficult to understand anything. Eventually, I decided on the “simplest” option (File::Find::find()), which unfortunately could not do the filename matching itself. I had to implement for myself a check to see if the file name ends with “.conf”. I did not bother trying to find functionality that could check if a string ends with another string like in Ruby, because I figured that Perl, having so many features, likely has built-in regex matching so concise (once you learnt the syntax) to use that the advice would be to “just use a regex”. It turns out that I was correct, and I therefore had to examine the provided Perl manual to find out how to perform regex matching. I expected there would be some sort of obscure, built-in syntax or operator for it, and I was correct; there was no way I would have figured it out on my own.

## Reflection

Ruby’s adherence to the regularity and simplicity principles via focus on object orientation increases readability. Most functionality in the program is achieved via object/class method calls, which are largely intuitive and self-documenting. The result is that the code is very readable and easy to understand (even if you did not know Ruby, you could probably figure out what the program does easily).

Perl’s violation of the syntactic consistency principle decreases readability of the program. Many aspects of the syntax are effectively random characters, whereby it is very difficult to decipher what is happening. For example:

* Acquiring a pointer to a function: \&
* The function parameter list: $\_
* The regex match operator: =~
* The regex syntax: /pattern/ (regex by itself is already a mess of special characters)

# Smalltalk

## Program Testing Procedure

Run program an observe output. Every 3rd line should be "Fizz", every 5th line "Buzz", every 15th line "Fizz buzz". The other lines should contain the line number.

## Weekly Question

The readability and writability of the Smalltalk version are better than the Fortran version, but worse than the Algol version. Fortran’s syntax is rather lacking in regularity and has unintuitive elements such as column-based syntax and output formatting, making the language difficult to learn, write, and understand. Smalltalk, while not having the most obvious nor intuitive syntax, is at least regular.  
Algol is a quite structured, regular procedural language, lending itself to the simple task of fizz buzz. Anyone familiar with basic procedural programming concepts such as loops and if statements would surely be able to write and understand the program. On the other hand, Smalltalk’s strong focus on object orientation is not particularly well suited to fizz buzz. A simple if-elseif-else chain must be built via object orientation abstractions, resulting in clumsy and less readable syntax (although there is probably a more advanced, cleaner way to do it that I did not think of). Smalltalk’s syntax is also not intuitive, meaning one will likely struggle to understand how the program works without being familiar with the language.

## Reflection

Smalltalk’s adherence to the regularity and orthogonality principles improves readability and writability. Smalltalk is based on the idea of sending “messages” to objects, from which functionality is achieved. The messages may have arguments, including code blocks. Despite this concept and the associated syntax not being particularly intuitive, it exhibits very high regularity and orthogonality, which does aid the readability of the program.

Although not particularly demonstrated in the program, Smalltalk’s implementation of classes and objects complies with the information hiding principle, improving code reliability. Smalltalk achieves this by allowing specification of public and private class members, with disallowance of public fields. All interaction with an object’s data must be explicitly defined within the class.

# C++

## Program Testing Procedure

Test sorting procedure with various arrays of books and observe output. Make sure the books are sorted by name, lexicographically, ascending. Please see the code for the test data, as there is too much to be shown here.

## Weekly Question

In terms of syntax and capabilities, C++ and Java objects are similar. Both enable objects via classes.  
However, the way object allocation, lifetime, and handling are done differs significantly. In Java, all object types (i.e. non-primitives) are allocated on the heap, and variables hold references to them – all handling of objects is done “by reference”. The lifetime of objects is managed by the garbage collector. In C++, objects are allocated on the stack unless one explicitly performs a heap allocation. Variables directly represent the storage and lifetime of their objects – handling of objects is done “by value”. The lifetime of an object is bound to the scope of the variable to which it is assigned. C++ also makes no distinction between primitive types and class/object types in terms of allocation and lifetime.  
In C++, objects more so represent allocated storage than pure OOP objects. This is highlighted by C++11’s introduction of “move semantics”, whereby an object’s state may be “moved” from its allocated storage to enable efficient memory management. In Java, an object conceptually encapsulates state and memory allocation.

## Reflection

C++’s const functionality adheres to the preservation of information, defence in depth, and security principles, increasing the reliability of the program. Variables may be marked as “const” to state that modification of them is not permitted/intended, which is enforced by the compiler. This enables compile-time detection of unintended mutation of program state.

C++’s multi-mode arithmetic and numerical conversions violates the defence in depth and security principles, decreasing the reliability of the program. When subtracting integer types, unsigned operands are not converted to signed types, resulting in the possibility of underflow. In the quicksort procedures, I have to explicitly use a signed type (std::ptrdiff\_t) for array indices, as some calculations result in negative numbers (this is not obvious from reading the code, however). Usage of unsigned types, which are more natural for array indices, would have resulted in silent underflow bugs.

C++’s templates adheres to the abstraction principle, increasing the readability, writability, and expressivity of the program. Templates enable a construct to be parameterised on a type or value, allowing, for example, a single data structure or algorithm to be used for multiple data types. The program makes use of std::vector, a dynamically-sized array type, and std::swap(), a function for swapping object values, both of which are templated constructs. These constructs can simply be instantiated for the Book class, enabling reuse of existing, well-tested functionality.

# Prolog

## Program Testing Procedure

Execute fizzbuzz(1000) and observe output. Every 15th line should be “FizzBuzz”, every 3rd line should be “Fizz”, every 5th line should be “Buzz”, and every other line should contain the line number.

## Weekly Question

Fizz buzz was more difficult to implement in Prolog than the other languages. Fizz buzz is naturally conceptualised in a procedural manner: **loop** over numbers, **if** divisible by 3/5/15, **print**, etc. However, in Prolog, logic is the main functionality provided by the language, and the presence of any procedural/imperative elements is almost coincidental. Enacting a specific control flow (as is required for fizz buzz) in Prolog requires manipulating the logic such that the inference steps involved produce the desired control flow, which is obviously more difficult than specifying the control flow directly.

## Reflection

Prolog’s focus on logical programming violates the structured programming principle, reducing the readability, writability, and reliability of the program. The control flow of the program is decided by the inference rules, largely hidden from the programmer. As a result, the relationship between the rules written and the orders in which they are executed is difficult to determine. Complicating the issue are constructs such as the cut.

The interpreted and symbolic nature of Prolog decreases the reliability of code. Symbols/names are not resolved until runtime, meaning any errors involving invalid or nonexistent symbols will not be detected until they are used at runtime (possibly not detected at all if inference happens to not require them). Additionally, the usage of some symbols is restricted in some contexts (for example, arithmetic/numerical operations require the symbols to be “instantiated”), errors relating to which also cannot be detected until runtime.

# Scheme

## Program Testing Procedure

Run the shaker sort function with various inputs and observe the output. In all cases, the resulting list should be sorted in ascending order.

|  |  |  |
| --- | --- | --- |
| **Test name** | **Test input** | **Expected output** |
| Empty list | () | () |
| 1 element | (42) | (42) |
| 2 elements, in order | (1 2) | (1 2) |
| 2 elements, reverse order | (2 1) | (1 2) |
| Many elements, in order | (1 2 3 4 5 6 7 8 9 10) | (1 2 3 4 5 6 7 8 9 10) |
| Many elements, reverse order | (10 9 8 7 6 5 4 3 2 1) | (1 2 3 4 5 6 7 8 9 10) |
| Many elements, random order | (5 2 3 7 1 7 10 4 9 8) | (1 2 3 4 5 6 7 8 9 10) |

## Weekly Question

Scheme performs I/O via “ports”, which can be thought of as streams or handles. The port must be closed, so their lifetime is significant. This breaks regularity, as the lifetime of values/objects in functional languages is usually irrelevant (the concept of lifetime is not really needed). After being closed, ports can no longer be used, which also somewhat breaks regularity.

## Reflection

Scheme’s use of lists and atoms for both data and computations violates the syntactic consistency principle, reducing the readability of the program. It can be ambiguous whether any given piece of code is intended to be data or a computation.

Scheme’s use of lists for everything also violates the labelling principle. Computational expressions depend on the order of items in the list to be correct, with no alternatives available (no keyword arguments, for example).

Scheme’s ability to treat data and code the same allows first-class functions, which complies with the abstraction principle. The single\_pass function can be reused for both the forward and reverse sorting passes since the comparison function can be passed as a function argument.

# Appendix 1 – Fortran 77

c Fortran 77 fizz buzz implementation for numbers up to 100.

program fizzbuzz

integer i

c For loop from 1 to 100, inclusive

do 10 i = 1, 100

if (mod(i, 15) .EQ. 0) then

write(\*, '(A)') 'Fizz buzz'

elseif (mod(i, 3) .EQ. 0) then

write(\*, '(A)') 'Fizz'

elseif (mod(i, 5) .EQ. 0) then

write(\*, '(A)') 'Buzz'

else

write(\*, '(I0)') i

endif

10 continue

stop

end

# Appendix 2 – Algol 68

BEGIN

FOR n TO 100

DO

IF n %\* 15 = 0 THEN

print(("Fizz buzz", newline))

ELIF n %\* 3 = 0 THEN

print(("Fizz", newline))

ELIF n %\* 5 = 0 THEN

print(("Buzz", newline))

ELSE

print((whole(n, 0), newline))

FI

OD

END

# Appendix 3 – Ada

with Ada.Text\_IO;

with Ada.Integer\_Text\_IO;

procedure Bubble\_Sort is

type Int\_Array is array (Positive range <>) of Integer;

procedure Do\_Bubble\_Sort(A: in out Int\_Array) is

Did\_Swap: Boolean;

Tmp: Positive;

begin

loop

Did\_Swap := false;

for I in Integer range (A'First + 1) .. A'Last loop

if A(I) < A(I - 1) then

Tmp := A(I - 1);

A(I - 1) := A(I);

A(I) := Tmp;

Did\_Swap := true;

end if;

end loop;

exit when not Did\_Swap;

end loop;

end Do\_Bubble\_Sort;

A: Int\_Array (1..10) := (7, 6, 3, 5, 9, 4, 8, 10, 1, 2);

-- A: Int\_Array (1..10) := (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

-- A: Int\_Array (1..10) := (10, 9, 8, 7, 6, 5, 4, 3, 2, 1);

-- A: Int\_Array (1..1) := (others => 42);

-- A: Int\_Array (1..0);

begin

Do\_Bubble\_Sort(A);

for I in A'Range loop

Ada.Integer\_Text\_IO.Put(A(I), 0);

if I /= A'Last then

Ada.Text\_IO.Put(", ");

end if;

end loop;

Ada.Text\_IO.New\_Line;

end Bubble\_Sort;

# Appendix 4 – Yacc and Lex

## Lex

%{

#include "list.tab.h"

#include <cstdlib>

%}

%option noyywrap

%%

\[ return OPEN\_BRACE;

\] return CLOSE\_BRACE;

, return COMMA;

-?[0-9]+ { yylval.ll\_val = std::strtoll(yytext, NULL, 10); return NUMBER; }

[ \t]+ ;

\n return NEWLINE;

. return \*yytext;

%%

## Yacc

%{

#include <algorithm>

#include <cstddef>

#include <iostream>

#include <vector>

std::vector<long long> parsed\_list;

int yylex();

void prompt();

void process\_list();

void print\_list(std::vector<long long> const& list);

void yyerror(char const\* message);

%}

%union {

long long ll\_val;

}

%token OPEN\_BRACE CLOSE\_BRACE COMMA NEWLINE

%token <ll\_val> NUMBER

%%

line: list NEWLINE { YYACCEPT; } ;

line: error NEWLINE { YYABORT; } ;

list: empty\_list ;

list: nonempty\_list ;

empty\_list: OPEN\_BRACE CLOSE\_BRACE ;

nonempty\_list: OPEN\_BRACE number\_list CLOSE\_BRACE ;

number\_list: NUMBER { parsed\_list.push\_back($1); };

number\_list: number\_list COMMA NUMBER { parsed\_list.push\_back($3); };

%%

int main() {

while (true) {

prompt();

if (yyparse() == 0) {

process\_list();

}

parsed\_list.clear();

}

}

void prompt() {

std::cout << "> ";

}

void process\_list() {

std::cout << "Parsed list: ";

print\_list(parsed\_list);

std::cout << std::endl;

std::sort(parsed\_list.begin(), parsed\_list.end());

std::cout << "Sorted list: ";

print\_list(parsed\_list);

std::cout << std::endl;

}

void print\_list(std::vector<long long> const& list) {

std::cout << '[';

for (std::size\_t i = 0; i < list.size(); ++i) {

std::cout << list.at(i);

if (i + 1 != list.size()) {

std::cout << ", ";

}

}

std::cout << ']';

}

void yyerror(char const\* message) {

std::cout << "Syntax error" << std::endl;

}

# Appendix 5

## Bash

#!/bin/bash

# This isn't strictly bash, but I don't think anyone ever is going to do

# such a search in bash and NOT use find.

find / -type f -name "\*.conf"

## Perl

# Import the file iteration function

use File::Find;

sub callback {

# If filename matches regex .\*\.conf$

if ($\_ =~ /.\*\.conf$/) {

# Print filename with newline

print "$\_\n";

}

}

# Iterate over all files in system, call "callback" for each one

find(\&callback, "/");

## Ruby

require "find"

Find.find("/") do |path|

if FileTest.file?(path)

if path.end\_with?(".conf")

puts path

end

end

end

# Appendix 6 – Smalltalk

1 to: 100 do: [:x |

(x \\ 15 == 0) ifTrue: ['FizzBuzz' displayNl]

ifFalse: [

(x \\ 3 == 0) ifTrue: ['Fizz' displayNl]

ifFalse: [

(x \\ 5 == 0) ifTrue: ['Buzz' displayNl]

ifFalse: [x displayNl]

]

]

]

# Appendix 7 – C++

#include <cstddef>

#include <iostream>

#include <random>

#include <string>

#include <utility>

#include <vector>

class Book {

public:

Book() : bookID(0), bookName(), ISBN() {}

Book(int id, std::string name, std::string isbn) : bookID(id), bookName(name), ISBN(isbn) {}

~Book() = default;

int GetBookID() const { return bookID; }

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

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

void SetBookID(int value) { bookID = value; }

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

void SetBookISNB(std::string value) { ISBN = value; }

private:

int bookID;

std::string bookName;

std::string ISBN;

};

bool operator<(Book const& lhs, Book const& rhs) {

// Assume we're sorting by book name.

return lhs.GetBookName() < rhs.GetBookName();

}

namespace impl {

std::size\_t partitionBooks(std::vector<Book>& books, std::ptrdiff\_t left, std::ptrdiff\_t right, std::ptrdiff\_t pivot) {

Book const pivotVal = books.at(pivot);

std::swap(books.at(pivot), books.at(right));

std::ptrdiff\_t cur = left;

for (std::ptrdiff\_t i = left; i < right; ++i) {

if (books.at(i) < pivotVal) {

std::swap(books.at(i), books.at(cur));

++cur;

}

}

books.at(right) = books.at(cur);

books.at(cur) = pivotVal;

return cur;

}

std::ptrdiff\_t selectPivot(std::vector<Book>& books, std::ptrdiff\_t left, std::ptrdiff\_t right) {

static std::default\_random\_engine randEng{std::random\_device{}()};

return (randEng() % (right + 1 - left)) + left;

}

void sortBooks(std::vector<Book>& books, std::ptrdiff\_t left, std::ptrdiff\_t right) {

if (left < right) {

std::ptrdiff\_t const pivot = selectPivot(books, left, right);

std::ptrdiff\_t const newPivot = partitionBooks(books, left, right, pivot);

sortBooks(books, left, newPivot - 1);

sortBooks(books, newPivot + 1, right);

}

}

}

// Sorts `books` in place using quicksort.

void sortBooks(std::vector<Book>& books) {

// Using std::ptrdiff\_t for things because it's a bit easier and safer than using unsigned.

// Just assume we won't have > PTRDIFF\_MAX elements in the vector.

impl::sortBooks(books, 0, static\_cast<std::ptrdiff\_t>(books.size()) - 1);

}

void doTest(std::string testName, std::vector<Book> books) {

std::cout << testName << std::endl;

sortBooks(books);

for (Book const& book : books) {

std::cout << " " << book.GetBookName() << std::endl;

}

std::cout << std::endl << std::endl;

}

int main() {

std::vector<Book> books1{

{1, "BarQux", "98693457"},

{2, "Effective Modern C++", "235096343"},

{0, "Foo", "0123456789"},

{3, "Harry Potter", "576098234"},

{4, "The Lord of the Rings", "004563896"}

};

doTest("In order", books1);

std::vector<Book> books2{

{4, "The Lord of the Rings", "004563896"},

{3, "Harry Potter", "576098234"},

{0, "Foo", "0123456789"},

{2, "Effective Modern C++", "235096343"},

{1, "BarQux", "98693457"}

};

doTest("Reverse order", books2);

std::vector<Book> books3{

{0, "Foo", "0123456789"},

{1, "BarQux", "98693457"},

{2, "Effective Modern C++", "235096343"},

{3, "Harry Potter", "576098234"},

{4, "The Lord of the Rings", "004563896"}

};

doTest("Random order", books3);

std::vector<Book> books4;

doTest("Empty array", books4);

}

# Appendix 8 – Prolog

fizz(X) :- (X mod 3) =:= 0.

buzz(X) :- (X mod 5) =:= 0.

fizzbuzz\_print(X) :- fizz(X), buzz(X), write('FizzBuzz'), nl, !.

fizzbuzz\_print(X) :- fizz(X), write('Fizz'), nl, !.

fizzbuzz\_print(X) :- buzz(X), write('Buzz'), nl, !.

fizzbuzz\_print(X) :- write(X), nl, !.

fizzbuzz(0).

fizzbuzz(N) :- N2 is N - 1, fizzbuzz(N2), fizzbuzz\_print(N), !.

# Appendix 9 – Scheme

(define (single\_pass l compare)

(if (null? (cdr l))

l

(let ((first (car l)) (second (cadr l)))

(if (compare first second)

(cons first (single\_pass (cdr l) compare))

(cons second (single\_pass (cons first (cddr l)) compare))

)

)

)

)

(define (forward\_pass l)

(single\_pass l <=)

)

(define (reverse\_pass l)

(reverse (single\_pass (reverse l) >))

)

(define (shaker\_sort\_impl l counter)

(if (null? counter)

l

(shaker\_sort\_impl (reverse\_pass (forward\_pass l)) (cdr counter))

)

)

(define (shaker\_sort l)

(shaker\_sort\_impl l l)

)

(shaker\_sort `())

(shaker\_sort `(42))

(shaker\_sort `(1 2))

(shaker\_sort `(2 1))

(shaker\_sort `(1 2 3 4 5 6 7 8 9 10))

(shaker\_sort `(10 9 8 7 6 5 4 3 2 1))

(shaker\_sort `(5 2 3 7 1 6 10 4 9 8))