

Programming Languages Assignment

Student name: *Tawana Kwaramba: 19476700*

Course: *Programming Languages - COMP2007* – Lecturer: *Associate Lecturer: Arlen Brower*
Due date: *October 25th, 2021*



Curtin University

Contents

1	Introduction	1
2	Programme Testing and Programme execution	1
3	Fortran	1
3.1	Fortran: Discussion	1
3.2	Fortran: reflection	2
4	algo68	3
4.1	algo 68: fizzbuzz activation record	3
4.2	algo 68: reflectoin	3
5	ada	3
5.1	ada: comparison with other bubble sorts	3
5.2	ada: reflection	4
6	yacc and lex	4
6.1	reflection	4
7	scripting languages	5
7.1	scripting languages: reflection	5
7.1.1	bash:	5
7.1.2	ruby	5
7.1.3	perl	5
8	Small-talk	6
8.1	Small-talk: Discussion	6
8.2	Small-talk: Reflection	6
9	C++	6
9.1	C++:Discussion	6

9.2 C++: Reflection	6
10 Prolog	7
10.1 Prolog: Discussion	7
10.2 Prolog: Reflection	7
11 Scheme	7
11.1 Scheme: Discussion	7
11.2 Scheme: Reflection	7

List of Figures

1	Demonstration of Perl’s design strucutre	6
2	Small talk if-else-if statments	6

List of Tables

1. Introduction

2. Programme Testing and Programme execution

Each programme folder in this assignment is going to have its own independent **README.md**, and a **run.sh** file. Therefore, all the marker has to do is to execute the run.sh file by typing `./run.sh` and that will demonstrate the functionality of my programme. It should be noted that practical four doesn't contain a run.sh file instructions outlining on how the scripts should be executed has being left in README.md.

3. Fortran

3.1. Fortran: Discussion. Fortran was based on the programming paradigm of the punch card system which imposed specific rules on Fortran. Which included the following: the first column of each row is going to be reserved for the comment character which is either a c or an asterisk ("c" or "*"); column one to five of each row is going to be reserved for statements or labels; column six is going to be reserved for the continuation of a command from the previous line; commands will terminate on column seventy-two; and column seventy-three to eighty are going to be reserved for sequence numbers. As a consequence, this made Fortran more intellectually involving compared to previously written languages of Java, C and Python.

Programming in Fortran I had to be more conscious on the current column number which is typically not a behaviour I do while programming in the named languages. In the named languages I would typically choose to keep each line below 80 characters to make it easier for others to read the my code. Therefore, due to this imposed rule Fortran's *writability* is not like the named languages.

Fortran's variables is only limited to one-to-six characters long limiting the expressivity of variables in Fortran. In some cases six characters is not enough to fully explain the purpose of variable hence, compromises in variable naming will have to be made. Compared to Java, C and Python the programmer can fully express the purpose of variable as they is not limitation in variable name length. Therefore, as demonstrated Fortran is less expressive in variable naming as compared to Java, C and Python. Additionally, due to the limitation of the variable length Fortran as well is going to break the *zero-one-infinity* programming principle.

Fortran doesn't support reserved words causing the change of behaviour of functions, and resulting in a less *reliable* programming experience. Programming in Fortran requires double checking variable names to ensure that they were not overriding in-built functions as the compiler will not raise these incidents as errors. As compared to the named languages the compiler will raise the incidents as errors, and the programmer typically would not have to concern themselves with the naming of variables. Therefore, due to this Fortran will require the programmer to involve themselves with behaviours which they're not accustomed too. Additionally, since Fortran will allow the re-definition of in-built functions meaning that in a programming project an intern can override a in-built function to do something else and they would not know as the compiler will not flag it i.e. the do-while loop can be over ride to add one instead of looping. Hence, for this reason programming in Fortran in less *reliable* as compared to the named languages.

Additionally, Fortran is less *reliable* than Java, C and Python due the compilation process. Fortran doesn't look for uninitialised variables hence, Fortran will allow you to compile and run a

programme even if the variable hasn't be declared and not assigned to anything. Additionally, during execution the uninitialised variable will be a random memory address thus, any operations in the programme will be done to that specific memory address which in some cases can lead into unintended actions resulting, in Fortran breaking the *security* and *defense in depth* programming principle. As compared to the named languages the compiler will raise this incident as an *uninitialised variable error* therefore, stopping the user from accessing memory which they should not be accessing. Therefore, as demonstrated Fortran is less *reliable* and less *secure* than the names languages.

Similarities of Fortran as compared to the named programming languages is that the first character of each variable has to start with a letter and can't start with a number, Fortran will require you to declare the types of your variables same as C, and Java. Furthermore, Fortran will require the programme fields to be the same name as the current file name which is a similar idea which is seen in java whereby the file name has to be the same as the class name, and Fortran will require terminating statements for each command same as Java, and C.

3.2. Fortran: reflection. Fortran doesn't have scope as consequence it's difficult to plan and write a full programme. As a result of no scope variables can be accessed from anywhere causing unintended side effects as they is not protection from scope. Therefore, this is going to make it harder to debug larger programmes, as the programmer would have to have full knowledge of the programme instead of knowledge of the current scope. Making Fortran difficult to structure code in the appropriate hierarchies hence, violating the *structured programming principle*. Additionally, Fortran not allowing the hiding the implementation of abstract data structures, and implementations violating the *information hiding principle*.

as discussed in the previous section the fortran compiler is not helpful as it doesn't warn the programmer of common programming mistakes. therefore, one small mistake made by the programme can result of the built programme not running as intended. the compiler in modern languages is more of a helping guide in writing correct and reliable code therefore, since fortran compiler doesn't warn the programmer about these mistakes this makes the programmes experience less reliable violating the *reliability principle*.

in relation to fortran violating the *reliability principle*, the compiler will see all commands and variables as upper cases hence, making the language case insensitive. further adding on onto the unreliability of the fortran programming language.

as discussed previously fortran variable names will have to be between one to six characters long. the principle of *zero one infinity* outlines that the only valid length of anything is going to be either one, zero or infinity. as can be seen with the variable length names fortran clearly violates the *zero one infinity* principle.

Additionally, in Fortran if you had forgotten to declare a variable the fortran compiler would not warn you of your mistake but, instead will print out a random memory address. as compared to the name programming languages, the compiler would warn you of your mistake immediately and the line number which the mistake had occurred. therefore in fortran you would have to spend more time debugging the code at hand.

4. algo68

4.1. algo 68: fizzbuzz activation record. don't forget a picture of it here dawg

4.2. algo 68: reflectoin. writting the fizz buzz programme with algo 68 was more plesant than fortran. algo 68 is starting to represent the langauges which we're more accostumed too. alog 68 strongly reminds me of the bash scripting language as the constructs are going to end with a word instead of terminating delimiter.

algo 68 is going to be a language which is going to be scoped hence, the variables which are in a *begin* and *end* block are only going to be found to be accessed within that block, and variables which are going to be defined in any given construct are going to be only accessed within those constructs. therefore in algo 68 you're not going to get the side effects whichy you may get in fortran as the variables are going to be protected by scope therefore, algo 68 adhering to the *information hiding principle*.

additionally, due to the introduction of scope, and the introduction of clear termination of constructs, it's a lot easier to look at an algo 68 programme and get a good idea on what the intededed purpose of the programme is going to be hence, adhering to the *readability principle*. since, the algo-68 is going to be highly structured this also adheres to the *structured programming principle*. as a consquence to these observation, wrting a programme in algo-68 was a lot easier than writing a programme in fortran.

5. ada

5.1. ada: comparison with other buble sorts. a difference between the two lagnauges is going to be the way which they treat fucntions. in c the construction of function which will return nothing or something is going to have the same prototype, ada will actually will clearly segement these two classes of functions. functions which are going to return nothing are going to be referred to as *procedures*, and functions which are going to return something are going to be referred to as *sub-routines*.

a difference between c and ada is going to be the choice of terminating characters for commands. in c the end of a construct such as a "if and else" statement will need to be terminated with a semi-colon, and in c the end of a "if and else" statement is going to be terminated with a right parenthesis.

a similarity between c and ada implementation is going to be found in the positioning of the functions in the file. in c this phenomenon is going to be referred to as forward declartion whereby, the functions (procedures) have to be declared before the main body of code otherwise both the ada and c compilier will complain that it doesn't kow what the following funciton is going to be. this idea will also extend to variable decelartoin positioning as well, in c-89 variables will have to be decaled first before any commands are written, and in ada variables and types will have to be declared before the *begin* key word.

another similarity point between the ada and c implementation is going to be the use of pointer manipulation in order to swap any element which is going to be greater than the element which is going to be infront of it as both algorithms will derefrence that memory location to get what is going to be stored their.

c and ada are going to be both strongly type languages. meaning that that a variable or a data structure must be declared with a type before they're going to be used. this idea will extend to function imports as well, the type of the imports must be declared before use

come back to this, i don't feel like writing it more

furthermore, ada and c will both require you to import packages in order to be able to do simple commands such as print a statement to the screen, and to use data structures such as arrays.

5.2. ada: reflection. come back to this as well

6. yacc and lex

6.1. reflection. i really enjoyed the yacc and lex practical, although it was very time consuming and painful to do. overall, it was a good experience to see how the basis of a programming language are going to be constructed. furthermore, doing this practical gave an appreciation of some of the fundamental concepts taught in my computer science degrees.

a thing point of frustration while writing the programme is that the *yytext* variable was going to be a type of *yystype* which is going to be yacc's own datatype for defining a string. therefore, during compilation i didn't think much about it but, as soon as i ran the programme through yacc it produced segmentation faults. since, i thought the variable type *yystype* was just a warning, and it was going to be harmless i overlooked the warning and that led into many hours of trying to discover on why my programme was producing a segmentation fault. i would have wished the yacc compiler would have treated the data type mismatch as an actual error as other languages such as java would have. therefore, since the compiler will allow access of invalid memory addresses this going to make the yacc language less *reliable* to write in.

yacc and lex code are both going to be segmented into sections, whereby the first section is going to be the imports into the programme, and then after that it's going to be either the yacc or lex definitions. the next section in the lex file is going to be the tokenising rules and the corresponding code it is going to send to yacc, in yacc the next section is going to be the grammar of the language, and finally the last section is going to be the c functions which are going to be associated with each file. due to the structured nature of the programming language it made it easier to find area of interest i.e. if you want to change the manner which lex tokenize strings you can jump to the middle of the file. therefore, for this reason yacc and lex are going to adhere to the *structured programming principle* and as a consequence making the language readable.

a good thing with yacc is that it was really easy to pick up as it was heavily coupled to the c language. therefore, there was not that much to learn in the yacc language except on how it would process its language grammars.

7. scripting languages

7.1. scripting languages: reflection.

7.1.1. bash:. although, i have used bash since starting my computer science degree, and i use it in my daily navigation of the terminal environment i have always find it hard to write. the reason why i think that bash scripting is hard to write is because it doesn't follow the common conventions which i am accustomed to with c, java, and python. for example, if you want to access a variable it's not just enough to use its name like how you would in java, c and python, you will have to have the variable name with a \$ then you can use the desired variable. furthermore, if you want to declare a variable as something you will have to be cautious of your spacing in

bash you can't have spaces before or after the assignment of your variable, in languages such as java, c, and python they don't really care on how much spaces you will have. these just some of bash conventions which are different from the languages which i am accustomed too hence, while scripting in bash i will have to be thinking of a lot more things than i would be typically doing while programming hence for this reason bash has low *writability*.

in conjunction to the point discussed above, bash will have different methods to be able to access specific commands. you can access a command by just typing the name of the command, and with some of commands you will have to access them using the back-tick (`) which adds another layer of thought while scripting. the question then becomes "am i accessing this command the right way", which is typically not a question i would ask myself while writing other languages. therefore, in this regards bash will lack *regularity*, and will add another layer of difficulty for *writability*.

7.1.2. ruby. out of the three scripting languages done, ruby was the easiest one to write in because it's a close representation to the python scripting language, and java script to some extent. therefore, given that i have had a vast experience in python in relation to industry projects, teaching, and university assignment the *writability* of ruby was far better than bash, and perl.

furthermore, due to the close representation of ruby to other popular scripting languages this makes ruby very easy to learn, and to understand what is being conveyed code. additionally, the syntax of ruby is very simplistic, and the structure of the language closely represents what the programme's aim is going to be. therefore, in this manner ruby also adheres to the *simplicity* programming principle.

7.1.3. perl. in relation to the three scripting languages i would place perl as a middle child in relation to *writability*. perl borrows syntactic constructs, and conventions from both the bash scripting language, and modern popular scripting languages such as python and java script. for example perl relies heavily on anonymous functions and calling functions within a function as demonstrated in figure ?? which is a design pattern which is heavily used in javascript, and a little bit in java. additionally, perl only had fewer deviations from the conventional structure of modern day programming for example with perl, it doesn't really matter on how many blanks you have after an assignment which follows normal programming conventions. although, like bash if you want to access a variable, the variable has to be preceded by a dollar-sign ("\$"). therefore, perl borrows ideas from modern languages which i am accustomed too and borrow ideas from unix based scripting languages therefore, placing perl as a middle child in relation to *writability*.

```
#files_wanted is a reference to the address of a function
find(\&files_wanted, $searchPath)
sub files_wanted{
    #code for your function
}
```

Figure 1: Demonstration of Perl's design structure

8. Small-talk

8.1. Small-talk: Discussion.

8.2. Small-talk: Reflection. Constructing the conditional statements was the hardest part of the small talk practical this is because small talk doesn't natively have if-else-if statements, and only has if-else statements. To simulate the if-else-if statement nature of the programme, Small-talk will require you to nest an if-else statement inside the else clause of the parent if-else statement as demonstrated in figure 2. For this reason it made writing the Small-talk programme more difficult as I had to keep a conscious note on the location and the number of terminating square brackets ("]") hence, in this regards Small-talk will have low *writability*. Additionally, for this reason, it's difficult to see the purpose of the if-else structure from first glance as compared to the native if-else-if statements found in C, Java and Python. To be able to understand the structure it would require the programme to actually carefully read the programme and in this regards Small-talk is going to have low *readability*.

```
<True condition> ifTrue: [ <statement> ] ifFalse:
[ <child if-else-statement> ]
```

Figure 2: Small talk if-else-if statements

9. C++

9.1. C++: Discussion.

9.2. C++: Reflection. Out of the covered languages in the Programming languages assignment, this was probably the most intuitive, easiest, and most well rounded programming languages as compared to the ones covered in the unit. This is due to that C++ is very similar to programming languages which I have experience namely Java and C. Therefore, writing the C++ programme was a lot easier as it follows most of the programming principles which I am accustomed too. C++ is *reliable* as the compiler is very helpful in its error messages and will tell you exactly what is wrong in your code. Additionally, the C++ is almost the same as Java and C syntax hence, it was easier for me to read the C++ programmes and to know the function of a programme therefore, in this regards C++ is *syntactically* consistent with other languages; C++ is *regular* as all groups of conditions and constructs are going to be the same throughout the language; and C++ has the same structure as Java, and C therefore it's *structurally* consistent with those languages and also the structure of the language represents its function and purpose therefore adhering to the *structured programming* principle.

C++ is a language which encourages the use of streams, and most of its constructs is written to be used in conjunction with streams. However, C++ doesn't enforce the use of streams as it will allow you to do operations in the manner which you're accustomed too hence, making the barrier in learning C++ a lot smaller, as other languages will force you in doing things their way. Therefore, for that reason C++ is a lot easier to learn and to pick up if you have prior experience to Java, and C.

10. Prolog

10.1. Prolog: Discussion. Throughout my experience of programming I have only dealt with imperative languages hence, languages which you tell what it should do at each every single step.

Prolog is a logical language whereby its paradigm is that the programmer is going to specify the world's rules and the world's facts, and the language is going to make conclusions based on those rules and facts. Prolog is going to be based on the idea of a decision tree where it will use forward chaining and backwards chaining to form conclusions. For this reason writing the Prolog

10.2. Prolog: Reflection.

11. Scheme

11.1. Scheme: Discussion.

11.2. Scheme: Reflection.