CURTIN UNIVERSITY
FACULTY OF SCIENCE AND ENGINEERING: SCHOOL OF ELECTRICAL ENGINEERING, COMPUTING AND MATH SCIENCE

Programming Languages Assignment

Student name: *Tawana Kwaramba: 19476700*

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

Contents

List of Figures

List of Tables

1. Introduction

you will need to write this one last, as you will need to know what you're talking about each of the sections which you're coding for

2. Programme Testing and Programme execution

Each programme folder in this assignment is going to have its own independent **README.md**, and **run.sh** files. Therefore, all the marker has to do is to execute the run.sh file by typing *./run.sh*. Although, practical four doesn't contain a run.sh file but, instructions in the read me file has being left in the read me file on how to execute the individual scripts.

3. Fortran

3.1. Fortran: Discussion. Fortran was a first generation programming language which was mainly based on the punch card system which was used at the times **site me**. Fortran was a language intended for mathematical operations which can be used in the field of engineering and applied sciences. The name Fortran embodies this purpose as Fortran stands for *Formula translator*.

Due to Fortran being based on the punch card system this imposed specific rules on the programming language.Including the following: the first column of each row is going to be reserved for the comment character which is either "c" or "*"; column one to five of each row are 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; each of your commands given a row much terminate on column seventy-two; and column seventy-three to eighty are going to be reserved for sequence numbers, to just name few of the rules which are going to emposed onto Fortran. As a consquence of some of these rules, this made Fortran more intellectually involving compared to previously written languages of the plethora of rules to be memorised this made it more intellectually involving as compared to previoulsy written languages of Java, C, and Python.

3.1.1. Fortran: Comparison with written languages. While writting the fizzbuzz programme in Fortran I noticed I was thinking more than I typically would be while programming, and fortran has a lot of similarities to the named programming languages.

While programming in Fortran I had be consistently worrying about the current column numbber which I was currently in, this is not a behaviour I typically do while writing C, Java,

and Python programs. Typically writting these languages I don't really care which line number or column in, I would typically choose to keep each line below 80 characters to make it easier for others to read the my code. Although, in Fortran this is going to be a hard imposed rule.

Furthermore, Fortran variables can be only limited to one to six characters long therefore, while I was programming I had to always double check my variable names to ensure that they're going to be below six characters hence, limiting my expressivity in the purpose of variable. As compared to the name programming languages, I didn't have to think about the length of my variable names I typically would make them a length which is going to be sufficient in discribing their purpose hence, in the named programming languages I was able to express myself more.

Additionally, due to the lack of reserved words in the Fortran language, I had to always double check all my Fortran variables didn't have any other intended purposes as the Fortran complier will not tell you if a variable was reserved or not, and it will just allow you to override that variable. As compared to the named programmed languages I would typically just express the problem in the language, and rely on the compilier to warn me if I have tried to override a reserved word. Therefore, while programming in Fortran I had to rely more on myself, and in other languages I can rely more on the complier.

Lastly, in Fortran if you had forgetten to declare a variable the fortran complier would not warn you of your mistake but, instead will print our a random memory address. As compared to the name programming languages, the complier would warn you of your mistake immeditely and the line number which the mistake had occured. Therefore in Fortran you would have to spend more time debugging the code at hand.

Similarities of Fotran 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 field to be the same name as the current file name which is a similar idea which is seen in Java wherby the file name has to be the same as the class name, and Fortran will require terminating statements for each command as seing in languages such as Java, and C.

3.2. Fortran: Reflection. Fortran doesn't really have any scope as consquence, you were planning to build a full operational programme all the variables can be accesssed from anywhere therefore, unattended side affects may occur in the excution of the programme as the variables can be modified by anything as they're not protected by scope. Therefore, this is going to make it harder to debugg the current programme, as the programmer would have to have full knowledge of the programme as a hole instead of knowledged of the current scopes. As a consqunce, this is going to make Fortran diffucult to strucuture code in the appropriate heirachies hence, violating the *structured programming principle*. Additionally, since they is not really any scope in Fortran this is going to make it hard to hide the implmentations of any subroutines and fucntions therefore

violating the *information hiding principle*.

As discussed in the previous section the Fortran complier is not helpful as it doesn't warn the programmer of common programming mistakes. Therefore, one small mistak made by the programme can result of the built programme not running as intetended. The complier in modern languages is more of a helping guide in writing correct and relabile code therefore, since Fortran complier doesn't warn the programmer about these mistakes this makes the programmes exprience less reliable violiting the *reliability principle*.

In relation to Fortran violating the *reliability principle*, the complier will see all commands and variables as upper cases hence, making the language case insensitive. Further adding on onto the unrealiblity 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 ifinity* outlines thtat the only valid length of anything is going to be either one, zero or ifinity. As can be seeing with the variable length names Fortran clearly violates the *zero one ifinity* principle.

talk about how it violates the readibility principle

3.3. Fortran: Testing and compiling. you will actually need to go and do this at uni

3.3.1. Results from running programme. you will actually need to go and do this at uni

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 delimeter.

Algo 68 is going to be a language which is going to be scopped 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 therfore, 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 refferred 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 withe 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 refferred 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 satement to the screen, and to use data strucutres 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. Overal, it was a good experience to see how the basis of a programming langauge are going to be constructed. Furthermore, doing this practical gave on appreciation of some of the fundemental concepts taught in my computer science degrees.

A thing point of frustration while writign 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, durind compilitation 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 over looked the warning and that led into many hours of trying to discover on why my programme was producing a segementation fault. I would have wished the yacc compilier would have treated the data type mistmatch as an actually error as other languages such as Java would have. Therefore, since the compilier 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 segemented 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 consquence making the lnanguage 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, they 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 becuase it doesn't follow the common conventions which I am accostumed to with C, Java, and python.For example, if you want to access a variable it's not just enought 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 somethign 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 accoustmed too hence, while scripting in bash I will have would have to be thinking of a lot more things than I would be typically doing while programming hence fort his reason bash has low *writability*.

In conjuction 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 *regulairty*, and will add another layer of diffuculty 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 represention to the Python scripting language, and java script to some extent. Therefore, given that I have had a vast exprienc; in Python in relation to industry projects, teaching, and univeristy assignment the *wrtiability* 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 simplisitic, and the strucutre of the language closly 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 langauges I would place Perl as a middle child in relation to *writiability*. Perl borrows syntatic constructs, and conventions from both the bash scripting language, and modern popular scripting languages such as Python and java script. For example

Perl relays heavily on annoymous functions and calling functions within a function as demonstrated in figure 1 which is a design patern which is heavily used in Javascript, and a little bit in Java. Additionally, Perl only had fewer deviations from the conventional strucutre 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 preceeded by a dollar-sign ("$"). Therefore, Perl borrows ideas from modern languages which I am accoustmed 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(\&filesWanted, $searchPath)
sub filesWanted{
        #code for your function
    }
```

Figure 1: Demonstration of Perl's design strucutre

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 statemnets. To simulate the if-else-if statement nature of the programme, Small-talk will require you to nest an if-else staement inside the else clause of the parent if-else statement as demonsstraed in figure 2.For, this reason it made writing the Small-talk programme more diffucult as I had to keep a consicious note on the location and the number of terminating square brackers ("]") hence, in this regards Small-talk will have low *writability*. Additionally, for this reason, it's diffucult to see the purpose of the if-else structure from firs 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 required the programme to actually carefully read the programme and in this regards Small-talk is going to have low *readability*.

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

Figure 2: Small talk if-else-if statments

9. C++

9.1. C++:Discussion.

9.2. C++: Reflection. Out of the covered languages in the Programmning languages assignment, this was probably the most intutive, easiest, and most well rounded programming languages as compared to the ones coverred in the unit. This is due to that C++ is very similar to programming languges which I have experience namely Java and C. Therefore, writing the C++ programme was a lot easier as it follows most of the programing principles which I am accoustmed too. C++ is *reliable* as the complier is very helpful in it's error messages and wil 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 read the C++ programmes and to know the funcion of a porgramme therefore, in this regards C++ is *syntacically* 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 *structually* consistent with those languages and also the strucutre of the language represents it's function and purpose therefor 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 conjuction with streams. However, C++ doesn't enforce the use of streams as it iwll allow you to do operations in the manner which you're accoustemed too hence, making the barrier in learning C++ a lot smaller, as other languages will force you in doing thing their way. Therefore, for that reason C++ is a lot easier to learn and to pick up if you have prior exprience 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 is should do at each every single step. Prolog is a logical language whereby it's 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 Prologa

10.2. Prolog: Reflection.

11. Scheme

11.1. Scheme: Discussion.

11.2. Scheme: Reflection.