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

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 complier will not raise these incidents as errors. As compared to the named languages the complier 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-build 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 complier 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 complier 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 in 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 unattended 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*.

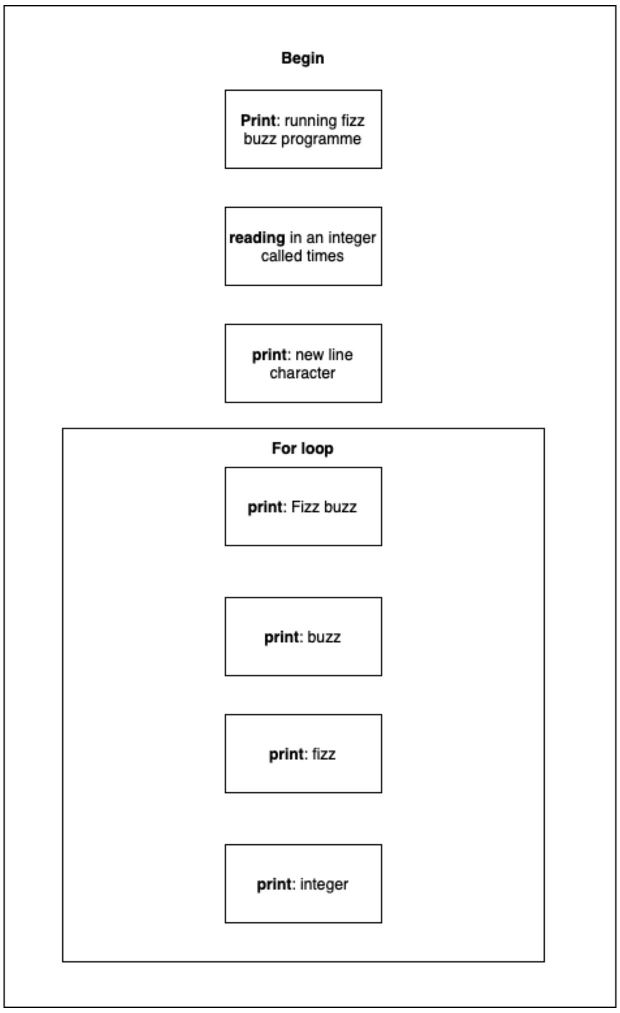
I found Fortran a lot harder to debug as the complier doesn't display helpful messages. During this programming assignment I spent hours trying to figure out why my programme was not working, although the logical structure of the programme was correct. To only find out what it was due to that I had not initialised a variable properly. Therefore, Fortran is not a language I would personally use for large scale projects as the complier is not helpful. Additionally, due to one small mistake breaking a programme Fortran is not *reliable*.

Additionally, the complier 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 how would the programme know that they're going to be accessing the correct variable or constant in the programme.

4. ALGOL 68

4.1. ALGOL 68: Fizz-buzz Activation Record. The programmed fizz buzz algorithm doesn't make use of actual functions, although it makes use of in-built functions such as the print statement, and the read statement. Therefore, the display can be illustrated as shown in figure 1, and the accompanying static and dynamic chaining is demonstrated in figure 2. The construction of the display is based on the submitted ALGOL 68 implementation of fizz-buzz.

4.2. ALGOL 68: reflection. Writing the fizz buzz programme with ALGOL 68 was more pleasant than FORTRAN as ALGOL 68 has started to represent languages which I am more accustomed too.


Begin
Print: running fizz buzz programme
reading in an integer called times
print: new line character
For loop
print: Fizz buzz
print: buzz
print: fizz
print: integer
Figure 1: The display of the Fizz Buzz algorithm

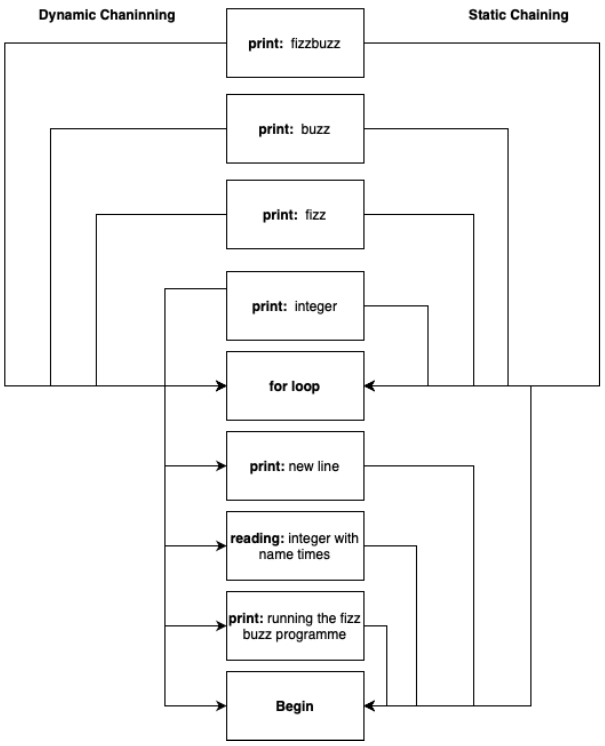


Figure 2: Static and dynamic chaining of the Fizz Buzz programme

ALGOL 68 strongly reminds me of the bash scripting language as the constructs are going to end with a word instead of terminating delimiter Therefore, looking at ALGOL 68 code is a lot easier, as all the constructs are clearly structured resulting in compliance with the *structured programming principle*. Additionally, since the language is highly structured it's gong to be easier to read hence, complying with the *readability principle*. To me it almost seems like modern day programming language structure was derived from ALGOL 68.

ALGOL 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 ALGOL 68 you're not going to get the side effects which you may get in FORTRAN as the variables are going to be protected by scope resulting in ALGOL 68 adhering to the *information hiding principle*.

5. ADA

5.1. ADA: Comparison With C Bubble sort. A difference between C and ADA Bubble sort is in the manner which they treat functions. In C the construction of function which will return nothing or something is going to have the same declaration, ADA separates the declaration of functions which are going to return something and nothing. With functions returning nothing being declared with the *procedure* key word, and functions which are going to be returning something declared as *sub-routines*. Therefore, in the bubble sort implementation, the swap, bubble-sort and display functions are going to be declared as functions in C, and as procedures in ADA.

Another 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 an 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. Therefore, in the bubble sort algorithm where it checks if the leading number is going to be less than the trailing number, ADA's if statement will terminate with a semi-colon, and C will terminate with a parenthesis.

A similarity between C and ADA implementation is going to be found in the positioning and definition of functions. In C this phenomenon is going to be referred to as forward declaration whereby, the function header has to be declared before the main body of code, in ADA the header and the function has to be declared and implemented before the main body of code otherwise, ADA and C complier will fail the programme. Therefore the swap, bubble-sort, and display methods have to be declared in C's implementation of the bubble sort, and in ADA the function has to be declared and implemented before the main method. Furthermore, this idea will also extend to variable deceleration positioning, in C-89 implementations of bubble sort, as variables will have to be declared before any commands are, and in ADA variables and types will have to be declared before the actual main body code, respective procedure and sub-routine functions.

Another similarity between the ADA and C implementation of bubble-sort 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 in-front of the current element. In both algorithms the elements to be sorted array is going to be dereferenced, and then put swapped into the required position as seeing by the swap function.

Furthermore, C and ADA are going to be both strongly type languages meaning, that a variable or a data structure must be declared with a type before they're going to be used. Therefore in the bubble sort implementation all variable names, imports, and exports are going to require a type.

5.2. ADA: reflection. ADA has a unique manner of declaring it's variables which I think is a little bit more readable as it follows the natural manner which we speak in. In ADA the variable name will be declared first proceeded by the variable type, in other languages such as Java an C the variable type will be declared first, and then the variable name will be declared second. In the human language it's more logical to say "A F-22 raptor is a plane", and not "a plane is a F-22 raptor" in this sentence clearly the type is going to be a plane, and the name is F-22 raptor. The logical most logical of constructing a sentence is to say the name followed by the type as demonstrated in the example above which, is in the manner which ADA declares its variables. Therefore, due to the manner of ADA's variable declaration it will make the language more readable.

6. Yacc and Lex

The first step in implementing symbol table is to choose the appropriate data structure to store the symbols based on requirements. Given that compliers are instruments which are used multiple times, and are going to be called multiply times it's important that the chosen data structure will have a fast access time, an ideal access time will be

$$O(1)$$

, and a more appropriate access time will be

$$O(nlogn)$$

. Therefore, the more appropriate data structures to be chosen will a form of a binary search tree, and or a hash table.

After chosen the appropriate data structure, the next thing to be decide is in the manner which you're going to be accessing and setting your data in your data structure, and how it's going to interface with the YACC file.

Then after, the symbol table can be implemented. Lex is going to server as the tokeniser hence, it's going to declare what each part of a programme is going to be, and Yacc is going to be where you grammar is going to be constructed hence, it's going to given meaning to a programme. The combination of Yacc and Lex will allow to create a parse tree. A parse tree is going to be a tree which is going to represent the hierarchical syntactical structure of your programme which is all the instructions ran in your programme and the root is going to be the actual programme, and the branches are going to be the individual instructions in the programme. Therefore, this parse tree is going to form your symbol table where as it has all the variable and the accompanying type. Lastly, the parse tree has to be stored in memory through the chosen data structure in step one.

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 complier is going to be constructed. Furthermore, doing this practical gave on appreciation of some of the fundamental concepts taught in early computing units.

A 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 the yystype warning, and as soon as I ran the programme through Yacc it produced segmentation faults. Since, I thought the variable type yystype was just a warning, I thought it was going to be harmless. This cardinal error lead into many hours of debugging, and playing around in Valgrind. I would have wished the Yacc complier would have treated the data type mismatch as an actually error as other languages such as java would have. Therefore, since the complier will allow access of invalid memory addresses without any other precautions Yacc will violate the *defense in depth* principle.

Yacc and Lex code are both going to be segmented into sections, whereby the first section is going to be the imports for the programme, then it's going to be the definitions, the next section in the lex file is going to be the tokenising rules and the corresponding return types, 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 more *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 **??** 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 3: 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 4.For, this reason it made writing the Small-talk programme more difucult 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 4: 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.