Assignment 1

Jordan Dube | CMSC 403 | 1/23/2022

1. [9 pts] Define the von Neumann’s architecture. Include both how a von Neumann machine is wired and the process used to execute programs.

**The Von Neumann’s Architecture is a model proposed by von Neumann in a First Draft that described the model as using a stored-program concept. This means that the computer has a set of hard-wired instructions that determine the basic operations that the computer can perform. With a program stored in memory, the instructions are carried out in a sequential order. Regarding memory, data values and programs are stored together, while the CPU carries the instructions out in a separate location from this memory. This means that instructions need to be transmitted (fetched) from memory to the CPU, executed, and then the results need to be transmitted back to memory to be stored. This process is referred to as the Fetch-Decode-Execute cycle, and it is how a computer using the von Neumann Architecture performs series of operations and programs. The memory is separate from the CPU, and the CPU contains both the Arithmetic Logic Unit and the Control Unit. Data and instructions are carried back and forth between the CPU and memory over wires that collectively make up the bus.**

1. [9 pts] Define the von Neumann Bottleneck and explain how it is related to the Processor-Memory performance gap.

**Since the memory and CPU components of the computer are separated, the bus is the only form of communication has between the two sections. Given the Fetch-Decode-Execute cycle that must be performed to execute programs, the number of fetch operations that can be performed is limited by the bus. This limited throughput between the CPU and memory is called the von Neumann Bottleneck, and it is inherent to the von Neumann Architecture. Due to Moore’s Law which states that the number of transistors on a circuit doubles about every two years, the performance of computer processors increases exponentially. Memory, on the other hand, does not have a corresponding Moore’s Law, as the performance of memory increases almost linearly over time, as opposed to the exponential growth exhibited by processors. This is known as the Processor-Memory performance gap, and it only increases as time goes on. This causes computers to execute instructions extremely fast, yet the CPU must wait for another instruction to arrive on the bus. So even as the processors get more powerful, the von Neumann bottleneck causes the execution cycle to stall due to the memory retrieval.**

1. [9 pts] List the computational paradigms discussed *in class*.

* **Imperative Languages** are classified by a sequential execution of instructions, usage of variables to refer to memory locations, and assignment to change the value of variables.
* **Functional languages** treat programs as functions; they treat functions as data, which limits mutable data.
* **Logical Languages** are based on formal symbolic logic.
* **Scripting Languages** are focused on gluing components together from surrounding context.
* **Object Oriented Languages** allow the writing of reusable code. Using concepts of real-world objects allows programmers to encapsulate code blocks.

1. [9 pts] How are compilers and interpreters similar? How are they different?

**Compilers and Interpreters both translate high-level code into machine code that is ready to be executed by the CPU. They are different in their timing: Compilers translate the whole source code before runtime, while Interpreters translate sequentially as the program proceeds. Interpreters are easier to implement than Compilers because Compilers are more complicated. Compiles have many different stages, including lexical, syntax, and semantics analysis in addition to machine code generation. Compilers have a slower translation cycle, but a fast execution time; on the other hand, Interpreters are much slower on the execution time because the program must buffer while the Interpreter translates the next line into machine code. While both translators have pros and cons, many programming languages have evolved to incorporate positive aspects of both models, such as Java’s JIT system. These combination systems are called hybrid implementations.**

1. [9 pts] What two language design criteria focus areas would you prioritize when designing a programming language? Why?

**I would focus on efficiency and regularity. To make a successful language, I would choose ease of use as my main goal. To make my code easier to use than other languages, I would emphasize regularity. By focusing on this area, I could consequently increase the programmer efficiency of my language. The more programmer efficient my language is, the more likely my language would be to adopt, and therefore, my language would be more successful. Having good uniformity would create an easy-to-understand environment for the user and limit exceptions and special cases. If my program is orthogonal, context shouldn’t impact uses of constructs, which would then increase efficiency. All these aspects would increase programmer efficiency and would make my language preferrable to those that do not possess the same focus areas. Security is not one of my tops areas because while it would be beneficial, it is not as important as efficiency due to the programmer can debug their code and find errors, but if programmer efficiency is low, then debugging becomes a major chore. To increase efficiency, I would implement features such as static data typing and concise syntax ({}).**

1. [9 pts] Discuss two or more examples of efficiency in the Java programming language. Specify if your example is computational efficiency or programmer efficiency.
2. **Java has concise syntax. With the use of {} to denote the beginning and end of code blocks such as classes, methods, and loops, Java has an advantage compared with other classes such as PASCAL which uses BEGIN and END. Not a big difference, but it is easy to identify which language between the two is more efficient. This is an example of programmer efficiency since it makes it easier on the programmer to type when the computer can use any other string or symbol to denote beginnings and ends.**
3. **Java has static data typing. Since Java can check the data types at compile time, it does not need to check them at runtime. This means that the CPU can execute operations without worrying about incompatible data types causing an error. This is a computationally efficient practice because it saves time at runtime by doing the computation before running. Programmers don’t necessarily need to check the data types because the JVM will do it for them, and they can alter the data types only if an error is thrown.**
4. [9 pts] Discuss two or more examples of inefficiency in the Java programming language. Specify if your example is computational efficiency or programmer efficiency.
5. **Java does not have pointers. This is a computational inefficiency because Java still has implicit pointers that are assigned with object memory allocation. This may improve programmer efficiency, but it still places more weight on the computations undertaken by the CPU. This more complicated runtime is a result of Java having this computational inefficiency.**
6. **Java focuses a lot more on security, and that requires Java to use up more memory than native languages like C and CPP. This is a detriment to both computational and programmer efficiency because a programmer must take time to specify things like private, static, global, and protected modifiers, and the subsequent debugging of such features when things go wrong, as well as requiring more memory to store the verbose headers and type checking that the compiler must perform. Another example of Java being verbose is displaying text with System.out.println; it’s a lot longer than “print.”**
7. [9 pts] Discuss two or more examples of regularity in the Java programming language.
8. **Java has universal operators. == applies to all data types, so this is an example of regularity, while C can not directly compare structures with ==. This is an example of generality because there are no special cases for operators in Java. The + is used for adding and concatenation with Strings, and the \* is used for only multiplication. If data types are different, then Java has set rules for the returned type, such as adding a float and an int returns a float.**
9. **Java methods can return all data types. Java can return the primitive data types: int, Boolean, long, float, etc., as well as reference data types: classes, ArrayList, String, etc. Java can also return void. This is an example of orthogonality as the return keyword can be used universally, regardless of context or interactions between constructs.**
10. [9 pts] Discuss two or more examples of irregularity in the Java programming language.
11. **Java can only store multiple primitive types in arrays. This is a lack of orthogonality because the array in Java is restricted by the context of the types that are being stored in the array. An array can only hold one type of element. If the programmer wants to use the ArrayList and corresponding API, they must only store objects.**
12. **Java uses two different methods of assignment. This is a lack of orthogonality because Java handles assignment between constructs differently based on if the element is of a primitive or reference type. Primitive types are handled during assignment by simply copying the value stored in the primitive type variable, while objects being assigned the value of another object simply get assigned the reference to the object within memory. This means that the two different object variables refer to a single location in memory.**
13. [10 pts] Should a language require the declaration of variables? Languages such as Lisp and Python allow variables names to be used without declarations, while C, Java, and Ada require all variables to be declared. Discuss the requirement that variables should be declared from the point of view of readability, writability, efficiency, and security.

**Yes, a language should require the declaration of variables. It improves readability in that it sacrifices conciseness for clarity. It is a simple matter for a fellow programmer to follow along the execution path of a program if the variables are declared before usage. It is also easier to create a symbol table for your program and keep track of the values of the variables used. It increases writability by telling the compiler how to interpret future uses of the variable and sets the stage for static type checking to increase efficiency. It is a simple matter to say whether a variable is a String or a different type of variable without sacrificing too much programmer time, and it saves on computational time because it limits the amount of information that the compiler must guess or use a junk value for. One word can prevent the program from crashing due to incompatible type operations. Readability and writability are improved due to clarity and correctness. Anyone can easily tell what a variable is and where it is applicable if they know the type as well as a name for the variable, two important parts of a variable declaration. Efficiency is improved because it saves the programmer time with debugging and guessing, and it saves the computer time by helping the compiler check the types before runtime. It improves security by checking types before running so that the programmer can quickly debug and maintain programs that have declared variables. It makes programs reliable by helping programmers correct problems that arise from incorrect operations with certain variable types. An incorrect variable type is easily discovered and fixed, while undeclared variables can be found anywhere and are harder to detect the exact source of these errors.**

1. [9 pts] List the two types of programming language abstractions and the three levels of programming language abstractions
2. **Data Abstractions:** simplify the behaviors and attributes of data (numbers).
3. **Control Abstractions:** simplify properties of the transfer of control (loops, conditionals).

* **Basic Abstractions:** collect localized machine information.
* **Structured Abstractions:** collect intermediate information about program structure.
* **Unit Abstractions:** collect large-scale information of a program.