Question 1.

A vast number of coding languages make use of various data types, and some of the most common types relate to numbers. **Despite this similarity, not all languages handle those numeric data types the same way, and rather frequently they handle them with very different strategies.** C, JavaScript, and Haskell all make use of numeric data types such as int, double, float, etc., but they each handle them slightly different depending on how the language designer set them up.

In the world of C, there are quite a few numeric datatypes and most of them have a normal (int) and an unsigned version. Those types include (with unsigned being omitted) short int, int, long int, long long int, float, double, and long double ("Data types in C," 2021). In order to use any of these datatypes, they have to be explicitly stated that a variable is one: int counter. If that statement is omitted, the program will throw an error. This is in stark contrast to how variable data types are handled in JavaScript and Haskell.

In Haskell, a number can be defined just as an int would be in C, but the language was made to be used without the declaration as well (Karthiq, 2017). Simply putting "4 + 4" will be interpreted as an integer addition, and Haskell will provide a result of 8 (Karthiq, 2017). Another similarity that Haskell has with C are the data types int, float, and double, but from there the similarities end because Haskell has no other numeric datatypes (Karthiq, 2017). Int, float, and double all three are the same number of bytes (2, 4, 8) in both C and Haskell as well, and both int types have the same range of numbers. Unlike C however, Haskell has another datatype: integer. Unlike all of the numeric datatypes in C, an integer in Haskell is not bound by any length constraints and therefore can be any number that the user wants (Karthiq, 2017). So, in the worlds of Haskell and C, there are a few similarities when it comes to the numeric datatypes as well as several drastic differences; but when compared with JavaScript, C and Haskell look like they could be almost perfectly interchangeable.

In the world of JavaScript, the designer seems to have taken the lazy approach to the numeric datatypes and opted for only a single type known as: Number ("JavaScript data types," n.d.). As with C, in JavaScript the variable has to be declared with a preface, such as int in C, but unlike in C this can be done simply by writing "let x = 10" and it will

know that x is a number. This is identical to how Haskell determines datatypes for a number, with a touch of C requiring the *let* preface. That's about the point that the similarities between JavaScript, and C/Haskell end. In JavaScript, since there is only the single datatype of number, any value can be put into a variable and it wont have a problem with or without decimals, and even scientific notation ("JavaScript data types," n.d.). Another unique behavior in JavaScript is adding an int and a string. Writing something like "5+5+'A'" will result in 10A, while "A+5+5" will result in A55, which is not how the other two would handle this at all. In the prior, Java handles the 5's as ints, and then adds that as a string to A while the latter treats all of the statement as strings. Any datatype can be stored into a given variable, even if it started as a number, and the new datatype that ended up being stored was a string. In C this would require redeclaring the variable with the new datatype, but in Java it can simply be done by assigning it the new value thanks to types being dynamic ("JavaScript data types," n.d.). So, while Haskell and C share several datatypes and even some behaviors/stats on those types, JavaScript very different and allows for a much lazier approach to datatypes in general.

Question 2.

In the world of computer programming, the number one limiting factor for a designer has to be themselves. A designer that is rigid and incapable or unwilling to change overtime will undoubtedly get left behind and eventually find themselves without a job. **A great way to combat this stagnation of a designer is to study multiple programming languages, even if they may never be used.** As with any realm of study, broadening one's mind with potentially useless studying of various topics, can provide them with a new way of looking at a problem that someone with less "useless" knowledge might never see.

An entire section in the text book *Concepts of programming languages* focused on the reasons behind studying multiple languages, and some great points are brought up. As mentioned before, studying additional languages can broaden a programmer's horizons and provide them with new ways of expressing an idea they had (Sebesta, 2012, p.2). If a programmer has a very limited knowledge of languages, they may end up trying to do something that the given language was never designed to handle and therefore, will

have an extremely hard time doing it, if not finding out it can't be done at all (Sebesta, 2012, p.2). One language can't do everything, so knowing several will give a programmer the ability to pick between a set of languages that will handle the task at hand better than others ("Whystudyprogramminglanguages," n.d.).

In addition to the benefits of knowing several languages, knowing the currently used language better can help a programmer in many areas. Having a solid understanding of the syntax, grammar, and structures available in the currently used language will allow a programmer to use it more efficiently ("Whystudyprogramminglanguages," n.d.). Potentially even simulating a behavior from another language in the current one which doesn't have that behavior natively ("Whystudyprogramminglanguages," n.d.). Knowing the limiting factors of a language is just as useful, if not more useful, as knowing it abilities. Again, another aspect to being able to choosing the correct language for the job at hand.

Another boon to learning more than just a single language, is that it will become easier to learn more languages for each one a programmer learns (Sebesta, 2012, p.3). With how fast the tech world is evolving, and the fact that the evolution is only getting faster, understanding the ins and outs of a single language aren't as useful as being flexible to new languages. As more languages are learned, the focus on specific behaviors of a language begins to fade, and a programmer will start looking more to the concepts or constructs that are shared by multiple languages such as types, loops, and inheritance ("Whystudyprogramminglanguages," n.d.). All of this flexibility can be gained by looking into more than just a single language, paradigm, or grammar. If the programmer knows about the structure of grammars, then when the time comes that no current language exists that can solve the problem, they will have the knowledge required to write a language that does solve the problem.

Question 3.

Declarative languages can be very useful and are used by a large number of big companies for various tasks. One reason for choosing a declarative language such as Prolog or Haskell is that it can be quite simple and fast to produce. **Though despite the fact that Haskell and Prolog are both declarative languages, they have several major**

differences that show the range of declarative languages. In general, a declarative language is one that focuses on the desired result rather than the steps to get there like in an imperative language ("Declarative programming: When “what” is more important than “how”," 2020). This allows the language to be much more concise, a single line can do something ten or more would be needed for in other languages, but it also makes it harder to read at times ("Declarative programming: When “what” is more important than “how”," 2020). The main difference between Haskell and Prolog is the subset of declarative programming that they fall into: function and logical respectively.

The Haskell language uses expressions in the form of lambda calculus to express what is to be solved, but doesn’t show the steps of doing so ("What is Haskell, and who should use it?," 2020). The functions used in Haskell to determine the steps are hidden behind the code a user writes, following the definition of a declarative language to a T. In addition to that, every function in Haskell is a mathematical function, and without them the program would have no clue how to run the code written by the user ("What is Haskell, and who should use it?," 2020).

Prolog on the other hand falls into the logic subset of declarative languages. Logic in Prolog is expressed by using relations that are determined by the user in the knowledge base ("Prolog | An introduction," 2019). There’s nothing in the programmer’s code that tells the computer how to work, the code written simply states how pieces are related to one another, and then through user queries in the command prompt the results are provided ("Prolog | An introduction," 2019). Since Prolog works based off of this knowledge base, which is a set of facts or rules about the pieces of the code, it follows the definition of a declarative language in the fact that it has no code pertaining to the steps taken by the program.

Prolog and Haskell both fall into the same category of languages, declarative languages, but they aren’t totally alike. As mentioned before, Haskell falls into the functional language subset, and Prolog into the logic subset. When the code written for both is considered, they do end up looking rather identical in their structure. Both languages use function names and variable names in the same manner mostly. In Haskell, the programmer has to tell the program what a method is going to return by typing a double colon (::) and then giving it the type it receives, follow by “->” and the type it will

return. In Prolog however, the input and output variables are put into parenthesis just like passing in variables to a function in C. After the name of the method, the lines that follow are preceded by a ":-". So, in terms of how a new, programmer defined method is structured, both languages follow the same pattern with different symbols. From there, they both follow the idea that the desired output is what is coded instead of the steps to get there. In Haskell, this is achieved by using lambda calculus such as [x | x <-[100..x]], and in Prolog it's just simple lines of codes relating to a relationship such as "parent(X, Z)". Both languages share an identical structure to develop code, but go about it in very different ways. Prolog is much more readable to the average person thanks to the lacking of lambda calculus.

References

Data types in C. (2021, June 28). GeeksforGeeks. https://www.geeksforgeeks.org/data-types-in-c/

Declarative programming: When "what" is more important than "how". (2020, February 24). IONOS Digitalguide. https://www.ionos.com/digitalguide/websites/web-development/declarative-programming/

JavaScript data types. (n.d.). W3Schools Online Web Tutorials. https://www.w3schools.com/js/js_datatypes.asp

Karthiq. (2017, November 14). *Data types in Haskell*. CherCherTech. https://chercher.tech/haskell/datatypes

Prolog | An introduction. (2019, October 15). GeeksforGeeks. https://www.geeksforgeeks.org/prolog-an-introduction/

Sebesta, R. W. (2012). *Concepts of programming languages*. Addison-Wesley Longman.

What is Haskell, and who should use it? (2020, August 21). 47 Degrees. https://www.47deg.com/blog/what-is-haskell

Whystudyprogramminglanguages. (n.d.). https://cs.lmu.edu/~ray/notes/whystudyprogramminglanguages/