Name: Milo Craun
PID: miloc

1. The book describes 4 criteria for language evaluation. List those 4 criteria. Choose one that you feel is the most important and give at least 3 reasons for why.

ANSWER:

The four criteria for language evaluation are readability, writeability, reliability, and cost. I think that readability is the most important of the 4 because it allows programs to be better maintained, makes debugging programs much easier, and helps people new to programming understand what programs are doing.

Readable programs are much easier to maintain. If the language is readable, anyone can look at code and easily see what it is doing and how it works. The structure of the programming language intentionally makes it easy to understand the code.

Having a readable programming language also makes it easier to debug programs. When the language is readable, it makes it easy to see what the code is supposed to do and then to use that to see where something is going wrong. There are not any weird or tricky syntactical constructions that are powerful, but hard to get right.

Finally, readable programming languages help new programmers learn programming. They can focus on the core ideas of breaking problems down into pieces, problem solving, and understanding basic programming constructions like variables and loops instead of trying to manage a complex and tricky programming language. They can also read other people's code and understand it which is vital to becoming a better programmer.

Readability is the most import of the four listed language criteria because it enables programs to be more easily understood. This fact applies both to the original author while debugging, new authors while trying to expand the code, and brand new programmers using the code to learn programming.

2. The book describes 4 language categories, sometimes called paradigms. List the 4 paradigms. Choose which paradigm you believe is the most influential on programming in general and give at least 3 supporting arguments for why.

ANSWER:

The four main language paradigms described in the book are imperative, functional, logic, and object-oriented. I think imperative is the most influential on programming because it is the closest to the hardware, a very early paradigm, and is a very commonly taught first paradigm.

The most popular computer architecture today is the von Neumann architecture. This architecture works by reading in instructions, decoding them, and then executing them. This architecture maps directly onto an imperative paradigm. We tell the computer what instruction to do, and it executes that instruction. Because the hardware maps so easily to this programming paradigm, it makes sense for pieces of software that interact with the hardware would follow it. That means systems software would be written in it, and systems software is very influential on how people think about and interact with the computer.

Imperative programming is also one of the earliest programming paradigms. Assembly languages are imperative. Highly influential programming languages like FORTRAN and the ALGOL family are imperative programming languages. The contributions these programming languages made are tremendously influential.

Imperative languages are also very commonly the first paradigm students are introduced to. Students typically either start with a imperative language, like C, or an object-oriented language, like Java. Students also may start with a language like Python. C is an almost textbook example of a purely imperative language. Many of the features used in Java are also imperative, but they are situated in an objected-oriented structure. Inside objects, code is written in a highly imperative style. Python is in a similar boat here with support for object oriented practices, but also support for a highly imperative style. Experiences with languages like these set the stage for a programmers conception of programming. Hence why people seem to have a hard time using purely functional or logic programming languages.

Imperative languages highly match hardware and have a long history that extends throughout many modern languages and other programming paradigms. Because of these things they are also usually the first languages that new programmers use. This then forms a lot of programmers understanding of how computing works. And then the cycle continues.

3. Select 1 language from the list below. Describe the purpose of the language, any major contributions to programming languages overall they added, and any subsequent language that branched from them.

FORTRAN, ALGOL, LISP, Smalltalk, SIMULA

ANSWER:

FORTRAN is any early imperative programming language that was created specifically for numerical computations. FORTRAN is also credited as being the first widely accepted compiled high level programming language. The major contributions of FORTRAN were a simpler English based syntax and the highly efficient optimizing compiler. This meant that now programmers could write efficient programs in a high level language instead of having to manually write machine code. This was huge because it allowed for people now intimately familiar with computing hardware to write programs that were fast to execute. The language also included special data types for scientific computation, such as a complex number data type.

Because of the success of the FORTRAN compiler, we can say that all compiled languages can stem from FORTRAN. In terms of direct ancestors, the ALGOL family was a big descendant. Following this path eventually gets us to C. Additionally BASIC is also a descendant of FORTRAN.

4. Imagine you are going to create a new programming language. Describe the purpose of the language, the major paradigm or category the language belongs to, which language, if any, that it is a descendant of, the execution style, e.g. compiled, interpreted, etc.

ANSWER:

I would create a programming language that uses a functional style to generate audio. It would take influence from projects like SuperCollider, Max, and Pure Data. Instead of being a visual programming language, we would imagine soundscapes as nested application and composition of various sound functions. It would have both interpreter and compiler support. The interpreted version would have different characteristics to improve real time performance and to allow on the fly editing. This is to support live performances using the programming language. The compiler would produce higher quality audio and have additional features for changing how the end result sounds. It would also produce an executable that produces the audio, as well as an actual audio file of the result.

Creative programming is fun and interesting, and the idea of applying functional concepts to that domain seems like it would be a weird (read as interesting) combination of ideas. I also think that a lot of the ideas make sense. An audio effect is essentially a function. Similarly an oscillator is also a function. New possibilities could be supported by combining these ideas.