Prince Kannah
CS - 354: Programming Languages
HW1

P16:3

There are a number of varying reasons why there are so many programming languages. One of the main big one is that one size does not fit all. What can be done effortlessly in one language can become a labor of love in another. Java for example is not meant to be a functional language so trying to code functionally will take a lot more work (and probably third-party libraries). Likewise trying to approach languages like Erlang or Elixir with a OOP design pattern will likely lead to headaches. As technology has evolved and our computational needs become ever more complex new programming languages have sprung up to try and meet those needs.

P16:4

The success of a PL can be dependent on a number of factors. Those included in *Programming Language Pragmatics* are ease of use, standardization, open source, and having a very efficient compiler. Ease of use and implementation are key. If a language is needlessly complicated to implement and even more so to learn developers will look for alternatives. The success of Python can be contributed to its ease of learning, make it a great first time language for novice. The language is also open source with countless libraries and derivatives. Standardization can also help lead to the success of a language as it makes it easier to move one's' code around which means greater adaptability as open try it on new systems and extend the language in the process. Often time one bragging right one language has over the other is how "fast" it is or in other words how good the compiler is. In *Programming Language Pragmatics* Scott states "...some languages (e.g., Common Lisp) have been successful in part because they have compilers and supporting tools that do an unusually good job of helping the programmer manage very large objects".

P16:6

They key distinction between declarative languages and imperative languages is that declarative languages place emphasis on what the computer should do and imperative languages place their emphasis on how the computer should do it. It should be noted that the line between the two can become blurred easily with many cross-pollination of features. The declarative category include functional languages (Lisp, Haskell), dataflow (Id, Val) and logical languages (Prolog, SQL). A rule based language like Prolog for example is more concerned with coming up with the desired result within using a set of rules. The emphasis is placed on the what and not so much on the how. Within the imperative category there are subfamilies that include von Neumann, OOP, and scripting languages.

P25:11

Compilation is akin to translation. The process of compilation takes the human readable source code and translate it to machine code much as a translator translate from one language to another. Optional optimization is performed to reduce the footprint of the program as well as make it faster. Unlike like interpretation, compilation is usually a one time thing. Compiled programs are (or can be) usually faster than interpreted one as the program is not being compiled on the fly and decisions made at compile time are decisions that do not need to be made during run time. One big advantage compiled languages have over interpreted ones, at least in industry, is that source code is private. The end user only has the compiled version. Pure interpretation is almost the opposite of compilation. Whereas compilation is a one-time thing interpretation happens at runtime. This makes interpreted languages slower because because each statement has to be decoded every time. One big advantage interpreted languages have is providing very good debugging information because "all run-time error messages can refer to source-level units. For example, if an array index is found to be out of range, the error message can easily indicate the source line and the name of the array" (Sebesta, 49). Interpreted languages' source code are usually public.

<u>P25:12</u>
Java on the surface appears to be compilation only. However, on upon closer examination one sees that it's a hybrid system of compilation and interpretation known as Just-in-time (JIT) compilation. Java source code is compiled to bytecode (the .class file). The Java Virtual Machine (JVM) then interprets this byte code to execute the program.

<u>P36:24</u>
Going from source code to intermediate code is a long process. The first few phases is to try to figure out the meaning of the source code. First the scanner reads in the stream of characters that is our program. The scanner performs a lexical analysis removing white-spaces, comments and other miscellaneous information. The stream of characters are stored as token. Each step in compiling is there to make the next step easier so lexical analysis to make things easier to parse by the parser. The parser then takes the tokens and create a parse tree based on the language's grammar. At this point any token that does not adhere to the language rules will cause the parser to throw an error. The last leg of compilation is figuring out the *meaning* of a program. Semantic analysis is performed to make sure that non-void actually do return something or functions are called with the correct number of arguments and so on. An inorder tree traversal is performed on the annotated parse tree to execute the program.

<u>P38:1.1</u>

A. A lexical java error would be $int\ @x\ = 5;$. @ is a reserved symbol within java.
B. A syntax error in java would be $int\, x\ =\ 5$. Notice the missing semicolon
C. An example of a static semantic error in java is using operators that don't apply. Using the equality ($==$) to compare two objects.
D. A dynamic semantic error would be going out of bounds of an array. This leads to the IndexOutofBoundsException in java.