Jim Crowell
CS-354: Programming Languages
Kennington, MW: 1030-1145

P16:3 Why are there so many programming languages?

Languages are idiomatic, and computer languages are no exception. There are nearly as many reasons for the existence of langauges as there are languages themselves; some excuses are more understandable/worthy than others. For instance, a new language may be created to perform some new purpose. I would argue this is probably the most common reason behind language genesis. Whether that's because it's a new type of problem that a new language must be written to solve (as was the case with Lisp, I think), or because the march of technology means we can do new things with it (like the first object-oriented languages coming about once we had the understanding and technological capacity to conceive/run them). Sometimes, though, it's as simple as cosmetics. Some people just want to do things a certain way and can't bring themselves to adapt to a prescribed method, and so design their own languages to mimic one they already use, but that performs in a way that's more natural for the designer. Such is my understanding, at any rate. I think in many ways, Java was meant as a successor to C++; to include everything C++ was capable of doing, but better, as well as more.

P16:4: What makes a programming language successful?

At its most basic level, a PL's success is based off of whether or not it can do what the designer needs it to do at the time, and generally, languages will be ranked above each other based on whichever one is most capable of doing the task the most easily, in the shortest amount of time. These are ALL relative rubrics, though. C++ is, objectively, not a particularly great language to start learning how to program with, if you're a complete newbie. It IS, however, monstrously powerful and versatile, and one of the most popular languages in use today. This is just one example of a language that has remained successful, keeping up with the top languages worldwide, despite the fact that it has some steep downsides (stiff barrier to entry being just one of them; memory management is a whole other thing). This illustrates that there is no one unifying metric that if a language can beat all other languages at, it's the "best" language. Success is relative.

P16:6: What distinguishes declarative langauges from imperative ones?

The primary difference between the two is how prescriptive the process is. Imperative programming is a step-by-step series of orders you tell the computer to accomplish a goal. Do this, then this, then this, etc. Declarative programming appears to more give the computer a goal, but not prescribe the manner in which the computer achieves it. The way I've seen it explained to me is that Imperative languages are used when the designer is primarily focused on HOW the machine gets an answer, while Declarative languages are the purview of designers who are more focused on WHAT answer the computer returns.

P25:11: Explain the distinction between interpretation and compilation. What are the comparative advantages and disadvantages of both?

The biggest immediate difference between them is one of scope. A compiler is an intermediary between programmer and user; the programmer writes a function in a compiled language, then compiles it, and that compiled product is the actual program that the user will interact with, giving input to and receiving output from. Once the compiler has done its job of compiling the source code (translating it into assembly language and then often machine language), the compiler's job is DONE, and it is no longer a part of the equation. An interpreter, by contrast, has more of a real-time involvement. The interpreter is there as long as the program is running, and is actually the "brains" of the operation, while the program is running. An interpreter will read statements in the interpreted language's code, line by line, and execute them, real-time.

The advantage of compiled language use is primarily related to run time/performance. Since the process of actually turning the programming language into code executable by the machine is already done, that's one less thing that the computer has to do while it's running. If you consider interpretation to be a form of "simultaneous compilation", you can see how the machine is effectively doing two jobs at once, and therefore takes longer to get the same amount of work done (each task being done at half capacity).

Conversely, the primary advantage of an interpreted language is the fact that its real-time status means the designer could run his program for the purposes of debugging before finishing his development of it. That is to say, since an interpreted language is executed more or less line by line, as input comes in from a prospective user, it's much easier for a designer to run through and see where potential errors crop up, and fix them in the source code.

P25:12: Is Java compiled or interpreted? Or Both? How do you know?

Java is both. It is compiled because one writes a .java file and then runs the "javac" command at the prompt in order to translate the .java file (source code) into a .class file (executable file/actual program). The "interpreter" part comes in with the fact that running the "java" command to RUN the program spawns a virtual machine (JVM) to actually take care of executing the program itself, which is an aspect of interpreted languages.

P36:24 Describe the form in which the scanner is passed from the scanner to the parser; from the parser to the semantic analyzer; from the semantic analyzer to the intermediate code generator.

I am not completely sure I understand the question, but I will try:

So during compilation, the program exists as a series of characters in a file for the compiler to read. The first thing it does is scan the file, breaking up the lines into keywords, symbols, numerals, etc. This is the scanner tokenizing the file (tokens being the smallest meaningful unit of a program), which is part of "lexical analysis". This makes things easier for the parser.

The parser receives these tokens from the scanner, and organizes them into a tree (a BTree, if I'm reading this right. How timely). This is a way of subdividing each piece into its related, higher-order organization, and so is how statements, functions, expressions, et al are organized. The parser also pulls double-duty in making sure that the tokens are grammatically accurate, which is necessary for the method by which the program organizes and recognizes commands/statements, which is called "context -free grammar".

The semantic analyzer receives this tree from the parser (I think), and its job is to make a table of symbols and meanings (for instance, the statement "int x = 12" would cause the character "x" to be assigned the meaning of "integer variable" in the symbol table). This is the way the computer can be made to recognize the identifiers that the designer intended. The semantic analyzer is also responsible for recognizing more strict rules that the scanner and parser mostly ignore/pass off (declaration of identifiers prior to use, identifiers follow the proper context constraints, etc). Through this process, the semantic analyzer will take the parse tree it is given and turn it into an abstract syntax tree. The SA will evidently also add annotation to nodes with info like pointers and the like. This is, generally, the form that the intermediate code generator will pick up.

The code generator is responsible for constructing actual “target code”, which can be assembly or machine language, depending on what language is being discussed. The CG will walk the symbol table and give variables locations in memory, as well as walk the syntax tree and attending to the references, operators, etc. By the time this process is done, you have either pure machine code, which a computer can read/act on, or assembly code, which is (if memory serves) only one level “higher” than machine code.

p38:1.1: 1.1 Errors in a computer program can be classified according to when they are detected and, if they are detected at compile time, what part of the compiler detects them. Using your favorite imperative language, give an example of each of the following.

We'll be using Java here, since it's the only language I have any real experience with that qualifies.

(a) A lexical error, detected by the scanner

A lexical error refers to using an identifier that's not recognized by the language as a proper token. It's not "in the lexicon", you might say. An example would be:

```
wark x = 12
```

"Wark" is not a word in Java, and I've placed it there as if it were a type. This would be lexically incorrect.

(b) A syntax error, detected by the parser

A syntax error is (to me, as a linguist) the simplest. Think, "the right words; the wrong order", usually. Something you can do in Java is split a command between two lines, but if you don't have the right grammatical markers attached to it, your compiler will go screwy. For example:

```
System.out.println("This is a string to print out that I've started on this line
and finished on this line");
```

That right there will cause a syntax error. You CAN do this, but you would need to enclose both lines in quotation marks, and have a concatenation operator between them. I.e.:

```
System.out.println("This is a string to print out that I've started on this line"
+ "and finished on this line");
```

(c) A static semantic error, detected by semantic analysis

Static semantic has to do with rules that are always in play, and immutable. Therefore, an example of a static semantic error would be calling a variable before it's been declared, because one static semantic rule for Java is that variables must be declared before they are called. So if I have

```
y = x + 12;
```

Before I've declared x or y or both, then we have a static semantic error.

(d) A dynamic semantic error, detected by code generated by the compiler

If I understand this correctly, a dynamic semantic error is largely dependent on context; it's made up of things that are perfectly valid to do, ordinarily, but the exact circumstances they're happening in the code after generation are breaking the program. I think. If so, then an example of a DSE would be a divide by 0 error. Ordinarily division is fine, and ordinarily 0 is fine, but if you have them in the wrong order/association with each other:

```
int x = 0;
int y = 12/x;
```

Or what have you, the program will most assuredly crash, and I believe that qualifies as a dynamic semantic error.

(e) An error that the compiler can neither catch nor easily generate code to catch (this should be a violation of the language definition, not just a program bug)

Okay, this is tricky. Some research I've done on the side suggests that this kind of error is "undefined behavior" errors. So, basically, asking the language to do something that it's never been programmed to do; no designer has ever made a process that recognizes and crunches that command. An example would be something like

```
String drillString = "This is most certainly a drill";
drillString[0] = 'S';
```

The idea there is like, "a string is an array of chars, so let me change the char at index 0 to 'S'". This is not something that Java is capable of doing, but I do not believe the compiler will catch that. This is the trickiest one to think of and I confess to being somewhat shaky on my answer, but this is what I found looking around the web; the book didn't really supply a solution that I recognized and was confident in.