


The Daily PL (Issue 1)

Introduction

Welcome to the 1st edition of the *Daily PL* for Summer 2023 volume of CS3003. It's an honor to have such a high readership. We have almost as many daily subscribers as the *Wall Street Journal* or the *Cincinnati Enquirer* (there is, sadly, almost truth to that facetious statement about the latter). Throughout the semester I, **Will Kent**  (<https://en.wikipedia.org/wiki/Superman>), will write (sometimes not so) short recaps of what we discuss in class. These materials are for you to use to learn and are absolutely a resource that you can rely on during the exams and when you do your homework. I hope that you enjoy!

Programming Languages

In today's class, we spent a significant amount of time writing a solid definition of *programming language*. After lots of great contributions from you, we settled on the following definition:

Programming language: A system for communicating *computational* ideas between *and among* people and computing machines.

Notice the two *italicized* words in the definition. They are written that way for a reason -- they are important. First, *among*: It is important to realize that there are two audiences for the code that we write. It is painfully obvious (especially when the compiler is mad at us!) that the computer is one of the intended audiences. What or who is the other? Just as painful in certain cases but less obvious: other humans (i.e., developers) are the other audience.

Bob Martin, the person who developed the Agile Method, believes that most developers spend ten times more, uhm, time reading code than writing it:

“Indeed, the ratio of time spent reading versus writing is well over 10 to 1. We are constantly reading old code as part of the effort to write new code. ...[Therefore,] making it easy to read makes it easier to write.”

What then, are we to make of *computational*? There is an important limitation on what we can ask the computer to do on our behalf. Although I can use natural language to ask my friend to do any task that I can conceive (and if your friends are like mine, they will promptly refuse to obey!). On the other hand, a famous theoretical result credited to Dr. Alan Turing (yes, *that* Alan Turing) holds that computers are only capable of computing a small subset of all the things that we humans can calculate. Wow!

Before wrapping up our first class, we discussed the difference between high- and low-level programming languages. The distinction comes down to whether or not the language is dependent (low-level programming languages) or independent (high-level programming languages) on the underlying machine. In this class we are going to spend all of our time thinking about high-level programming languages. In fact, some argue that the logic language that we will discuss (Prolog) is a *very-high-level* programming language.

References

- [1] R. C. Martin, *Clean code: A handbook of agile software craftsmanship*. Philadelphia, PA: Prentice Hall, 2008.

The Daily PL - 5/16/2023

Recap

programming language: A system for communicating computational ideas *among* people and computing machines.

low-level programming language: Programming languages that are closer to the hardware and provide a high level of control over computer resources such as memory, registers, and other hardware components.

high-level programming language: Programming languages designed to be easily understood and used by humans (A programming language that is independent of any particular machine architecture.)

Fundamental components of every programming language

1. syntax: The rules for constructing structurally *valid* (note: *valid*, not *correct*) computer programs in a particular language.
2. names: A means of identifying "entities" in a programming language.
3. types: A type denotes the kinds of values that a program can manipulate. (A more specific definition will come later in the course).
4. semantics: The effect of each statement's execution on the program's operation.

Practice

For your favorite programming language, attempt to explain the most meaningful parts of its syntax. Think about what are valid identifiers in the language, how statements are separated, whether blocks of code are put inside braces, etc.

Next, ask yourself how your favorite language handles types. Are variables in the language given types explicitly or implicitly? If the language is compiled, are types assigned to variables before the code is compiled or as the program executes? These are issues that we will discuss in detail in Module 2.

Finally, think about how statements in your favorite programming language affect the program's execution. Does the language have loops? if-then statements? function composition? goto?

The Value of Studying Programming Languages

Every new language that you learn gives you new ways to think about and solve problems. There is a parallel here with natural languages. Certain written/spoken languages have words for

concepts that others do not. Linguists have said that people can only conceive ideas for which there are words.

The same is true in programming languages. In certain languages there may be constructs ("words") that give you a power to solve problems and write algorithms in new, interesting ways. Having studied (broadly) programming languages, you will know these constructs and be able to deploy them appropriately to solve a given problem.

With a broad and complete knowledge of programming languages, you are equipped with the power to choose the right tool for the job. When all you have is a hammer, everything looks like a nail.

Knowing about myriad programming languages makes you an increasingly flexible programmer. In fact, the more you know about the concepts of programming languages (and how those concepts are embedded in the PLs that you know) the easier it is to learn new languages.

Finally, studying PLs will give you the ability to use the languages you know in better ways. Consider Python code that performs the task of opening a file and reading its contents. You want the code to be robust and worry that the file the user specifies is not available, so you use a `try` `... except` block:

```
from typing import TextIO

def readIO(readable: TextIO) -> str:

    """ A function that reads something from a file named readable """

    pass

if __name__=="__main__":

    result = ""

    try:

        testing_txt = open("testing.txt", "r")

        result = readIO(testing_txt)

    except IOError as ioe:

        print(f"Could not open testing.txt: {ioe}")

        raise ioe
```

Well, that's great, but it is a little verbose. As PL pros, you would look at the situation and realize

that there must be a better way. Indeed there is:

```
from typing import TextIO

def readIO(readable: TextIO) -> str:

    """ A function that reads something from a file named readable """

    pass

if __name__=="__main__":

    result = ""

    with open("testing.txt", "r") as testing_txt:

        testing_txt = open("testing.txt", "r")

        result = readIO(testing_txt)
```

Not only will you learn how to better *use* the languages you know on the *syntactic* level, you will also get a sense for how to leverage their *semantics* and take advantage of the language's implementation. This "awareness" can give you insight into the "right way" to do something in a particular language. For example, if you know that recursion and looping are equally performant and computationally powerful, you can choose to use the combining form (a term that I misappropriated from a paper by John Backus [that you will read later in the semester] and use [incorrectly] to describe patterns or forms in programming languages that allow sequences of statements to be combined. e.g. `for`, `while`, `if ... else`) that improves the readability of your code. However, if you know that iteration is faster (and speed is important for your application) then you will choose *that* method for invoking statements repeatedly.

Programming Domains

We do not write programs in a vacuum -- we write them to solve real-world problems.

The problems that we are attempting to solve lend themselves to solutions in particular programming languages with certain characteristics. Some of those real-world problems are meta (no, not *that* meta (<https://about.facebook.com/>)) because they are all about helping others solve their real-world problems. The solutions to these problems are known as systems programs and include operating systems, utilities, compilers, interpreters, drivers, servers, etc. There are a number of good languages for writing these applications: C, C++, Rust, Python, Go, etc. Languages used to write systems software need to be fast, safe and allow access to the underlying hardware, to name a few of the requirements.

However, most of programs that are written are designed/written to solve *actual* real-world

problems:

- scientific calculations: these applications need to be fast (parallel?) and mathematically precise (work with numbers of many kinds). Scientific applications were the earliest programming domain and inspired the first high-level programming language, Fortran.
- artificial intelligence: AI applications manipulate symbols (in particular, lists of symbols) as opposed to numbers. This application requirement gave rise to a special type of language designed especially for manipulating lists, Lisp (List Processor).
- world wide web: WWW applications must embed code in data (HTML). Because of how WWW applications advance so quickly, it is important that languages for writing these applications support rapid iteration. Common languages for writing web applications are PERL, Python, JavaScript, Ruby, Go, etc.
- business: business applications need to produce reports, process character-based data, describe and store numbers with specific precision (aka, decimals). COBOL has traditionally been the lingua franca of business application developers, although new business applications are being written in other languages these days (Java, the .Net languages).
- machine learning: machine learning applications require sophisticated math and algorithms and most developers do not want to rewrite these when good alternatives are available. For this reason, a language with a good ecosystem of existing libraries makes an ideal candidate for writing ML programs (Python).
- game development: So-called AAA games must be fast enough to generate lifelike graphics and immersive scenes in near-real time. For this reason, games are often written in a language that is expressive but generates code that is optimized, C++.

This list is non-exhaustive, obviously!

Language Evaluation Criteria

There are four (common) criteria for evaluating a programming language.

Readability is a metric for describing how easy/hard it is to comprehend the meaning of a computer program written in a particular language, *without running it*. The most thorough study to date about the daily activities of computer programmers shows that

on average program comprehension takes up ~58 percent of [our] time [1]. 

(<https://ieeexplore-ieee-org.uc.idm.oclc.org/document/7997917>)

If we spend *that* much time doing something, it must be important.

In a sense, *writeability*, the second evaluation criteria, is the complement to readability. Writeability is how/easy or hard it is to create programs for a particular domain in a particular programming language. Take note that writeability is a way to analyze a particular language in the *context of a particular domain*. If you evaluate the writeability of the same programming language, say

JavaScript, in the context of writing web applications you will score the language differently than you would if you evaluated its writeability in the context of writing climate-modeling software.

Finally, the third criteria that we discussed in class was *reliability*. Because the software that we write will continue to be used long after we have finished writing it (if we are lucky enough to write software that people use!), it is important that it works correctly even long after we stop paying attention to it. What's more, these days it is vital that our software is correct because we are beginning to rely on it to protect our safety! Specifically, the reliability of a programming language is the extent to which a program written in that language performs according to its specifications under all conditions.

As a bonus, it is possible to consider the *cost* of a programming language. To evaluate the cost of a programming language we will have to broaden our perspective a bit. We will have to include the cost of training programmers to use that language, the cost of each execution of the program (think about it, executing a program is not free), the cost of maintaining and updating the program if it has poor reliability or does not include all the features that users desire.

A Language's Mannerisms (Check out that sophisticated vocabulary!)

Several characteristics of programming languages can be used to score it according to the criteria above. A language's *overall simplicity* is based upon the number of basic concepts that it has.

Feature multiplicity (**having more than one way to accomplish the same thing**) ➡

(https://en.wikipedia.org/wiki/There%27s_more_than_one_way_to_do_it) decreases a language's overall simplicity. *Operator overloading* (when operators perform different computation depending upon the context (e.g., the type of its operands)) also decreases the simplicity of a language.

Before you get carried away thinking that a language should pursue simplicity at all costs, think about the most simplistic language you can imagine and then decide if you want to write

industrial-strength programs in assembler! ➡ (<http://www.chrissawyergames.com/faq3.htm>) A language's overall simplicity plays a role in its readability (positive, for the most part), writeability (also positive) and reliability (again, positive).

The presence/absence of type information for variables in a programming language can also play a role in how readable/writeable/reliable a program written in that language is. Types are a way to limit/detect invalid operations on, or invalid values of, variables. *NB*: We are going to learn a more precise definition of types in a future lecture -- stay tuned -- but this definition will do for now. That said, type information does not prevent all mistakes. Nor are types always a good thing. Having to write the type of every variable can cause finger fatigue and make it harder to *write* code. Types positively impact a language's readability and reliability but (can) negatively impact a language's writeability.

The next characteristic is a little tricky. *Orthogonality* is all about how regularly (ie, normally,

without exception) operations in a programming language interact with one another. (Alternate definition: The mutual independence of a programming language's primitive operations.) The definition is a little verbose, so let's conjure up some examples. It would seem like a good idea for a language to support return values from functions that can be of *any* type the language supports. So, if you had a language that allowed you to have variables that are integers, if it were orthogonal you would be able to return integers from functions. The same for doubles. And even composite types like arrays, if the language supported them! In C, what can you `return` from a function? An `int`? Check. A `float`? `double` (see what I did there?) check. Well, what about an array? Bzzzz. Nope. What we just experienced is an example of a piece of C's syntax that is non-orthogonal. What you can probably surmise from this example is an important corollary to the amount of orthogonality in a programming language: As the orthogonality of a programming language decreases, the number of *exceptions to the rule* increases. *Exceptions to the rule* look like what we just saw. In a highly orthogonal language, the rule would be as succinct as:

You can return a variable of any type from a function.

Too bad we have to make it clunkier when talking about C by adding caveats (the *exceptions*):

... except functions and arrays.

Before you think that these things don't actually matter, let me reassure that they exist in real life:

6.5.2.2 Function calls

Constraints


- 1 The expression that denotes the called function¹⁰²⁾ shall have type pointer to function returning **void** or returning a complete object type other than an array type.

Be careful, though. There is a Dr. Jekyll lurking: The more orthogonal a language, the slower it is: The compiler/interpreter must be able to compute based on every single possible combination of language constructs. If those combinations are restricted, the compiler can make optimizations and assumptions that will speed up program execution. The orthogonality of a language plays a part in determining how readable, writeable and reliable a program written in a given language is.

The syntax of a language also plays a crucial role in determining its readability, writability, and reliability. For instance, Python employs indentations to define the structure and hierarchy of code blocks, rather than relying on braces or explicit keywords.

In contrast, languages like Java, C, and C++ depend on braces or semicolons to establish the structure and hierarchy of code blocks. In Java, improper indentation doesn't impact code execution as computers disregard indentation and whitespace, focusing solely on the code's syntax. The compiler parses the code based on the placement of braces ({}) and semicolons (;) to discern the structure of code blocks and statements.

Improper indentation can render code difficult to read and less reliable since it increases the likelihood of errors and makes it easy to lose track of brace placement. However, it may make the code more writable, as there is no need to be concerned about indentation. Conversely, proper indentation enhances code readability and reliability.

[1] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan and S. Li, "Measuring Program Comprehension: A Large-Scale Field Study with Professionals," in IEEE Transactions on Software Engineering, vol. 44, no. 10, pp. 951-976, 1 Oct. 2018, doi: 10.1109/TSE.2017.2734091.  (https://ieeexplore-ieee-org.uc.idm.oclc.org/document/7997917)

The Daily PL - 5/18/2023

Implementations of programming languages

In our last class, we delved into the distinction between high-level and low-level programming languages. While computers can directly understand low-level languages, they require assistance in comprehending high-level languages. This assistance comes in the form of another computer program that translates the high-level code into machine language. There are three main types of language implementation: compilers, interpreters, and hybrid implementations.

Compilers

Compilers are responsible for translating programs from a source language to machine language. This process involves several phases, including lexical analysis, syntax analysis, intermediate code generation, semantic analysis, optimization, and code generation. Lexical analysis removes white space and comments, converting the source code into lexical units. Syntax analysis then uses these units to generate parse trees, which represent the syntactic structure of the program. The intermediate code generator produces a program in a different language that lies between the source program and the final machine language output. The semantic analyzer, integrated with the intermediate code generator, checks for errors that are difficult to detect during syntax analysis, such as type errors. Optimization is an optional part of compilation and aims to improve program size and speed. The code generator translates the optimized intermediate code into an equivalent machine language program.

Throughout the compilation process, a symbol table serves as a database containing type and attribute information for user-defined names in the program. While the machine language generated by a compiler can be executed directly on the hardware, it is often necessary to link the program with other code, such as system programs for input and output. This linking process is accomplished by a linker, which locates and connects the required programs from the operating system. Additionally, user programs may need to be linked to previously compiled programs in libraries.

Compiled languages offer the advantage of fast program execution once the translation process is complete.

Most production implementations of languages, such as C, COBOL, C++, and Ada, are by compilers.

Interpreters

Interpreters are software programs that execute code directly without prior compilation. They operate by reading the source code line by line and immediately executing the instructions encountered. Unlike compilers, which translate the entire program into machine code beforehand, interpreters interpret and execute each line or statement as they encounter it. This approach offers the advantage of easy implementation of source-level debugging operations since all run-time error messages can refer to source-level units. Interpreted languages are also more portable as they rely on the interpreter to execute the code.

However, there are notable drawbacks to interpretation. Execution in interpreted systems is typically 10 to 100 times slower compared to compiled systems. This slowness primarily stems from decoding the high-level language statements, which are more complex than machine language instructions. Moreover, regardless of how many times a statement is executed, it must be decoded every time. Another disadvantage of pure interpretation is that it often requires more space as the symbol table and the source program need to be present during interpretation.

Popular interpreted languages include Python and JavaScript.

Hybrid implementations

A hybrid implementation represents a compromise between compilers and interpreters. In this approach, the source language is translated into an intermediate language, which is then interpreted rather than being directly translated to machine code. Hybrid implementations offer certain advantages over pure interpreters. They are typically faster than pure interpreters since the decoding of the intermediate code only happens once, rather than repeatedly for each execution. This results in improved execution speed compared to pure interpretation. Furthermore, hybrid implementations tend to use less space as they do not require the presence of the entire source program during interpretation. An example of a language implemented with a hybrid system is Perl. Perl programs are partially compiled to detect errors and simplify interpretation.

In the early stages of Java's development, it employed a hybrid implementation. The intermediate form of Java, known as bytecode, enables portability across machines that have a bytecode interpreter and an associated runtime system, collectively referred to as the Java Virtual Machine (JVM). However, it's worth noting that there are now systems available that translate Java bytecode into machine code for even faster execution. This hybrid approach allows for both portability and performance enhancements in Java programs.

Practice

For your favorite programming language, explore how it is implemented and how it handles both compile-time and runtime errors. Additionally, try to delve into the reasons behind the chosen error reporting mechanisms.

Language Design

The programming language evaluation criteria provides a framework for language design.

However, these criteria can often contradict each other, presenting challenges in finding the right balance. One such conflict arises between readability and reliability. Designing a highly readable language may come at the cost of reduced reliability, and vice versa.

Another conflicting pair of criteria is reliability and the cost of execution. For instance, in Java, all references to array elements are checked to ensure they fall within legal ranges, enhancing program reliability. However, this rigorous checking adds overhead to the execution of Java programs, making them slower compared to semantically equivalent programs written in C, which does not require index range checking. In this case, the designers of Java prioritized reliability over execution efficiency.

Memory management is another aspect with trade-offs. Languages like Java or Python employ automatic memory management, providing convenience and reliability by automatically reclaiming memory. However, this convenience comes at the cost of increased runtime overhead. On the other hand, languages like C or C++ require manual memory management, offering more control and potentially faster execution at the expense of additional developer responsibility.

Type safety is yet another criterion that influences language design. Dynamically typed languages like JavaScript or Python offer flexibility and productivity as they can handle varying data types at runtime. However, the absence of strict compile-time type checking can result in potential runtime errors. In contrast, statically typed languages such as C++ or Java provide stronger type safety through compile-time checks, ensuring greater reliability but potentially increasing development time.

Conflicting criteria can be observed in the design of specific languages as well. APL, for example, includes a rich set of operators for array operations, enabling concise and expressive code. However, this expressivity often sacrifices readability, making APL programs difficult to understand for anyone other than the programmer. The trade-off between writability and readability is evident here.

These conflicts among language design criteria highlight the complex nature of language development, where compromises and trade-offs are necessary. Designers must carefully consider various factors and prioritize certain criteria over others based on the intended goals and trade-offs they are willing to make.

Programming Paradigms

1. A *paradigm* is a pattern or model. A *programming paradigm* is a pattern of problem-solving thought that underlies a particular genre of programs and languages. Traditionally, computer

scientists have classified programming languages (according to their *syntax*, *names and types*, and *semantics*) into one of four categories: imperative (aka, procedural), object oriented, functional and logic (alias declarative). More and more, however, computer scientists believe that it is a fools errand to try to classify modern programming languages because these languages take the best of all four paradigms. More on that later.

Imperative/Procedural

Imperative languages were the first programming languages to be designed. They came about a time when hardware cost much more than programmer's time. In other words, the goal of these programming languages was to make it possible to harness as much power from the computer as possible. Therefore, the programming languages' constructs for describing algorithms were just a thin wrapper over the underlying hardware architecture. Remember that the hardware, at the time and today, was built using the von Neumann model. You can see this by mapping the central semantic elements of an imperative programming language to corresponding elements of the hardware:

1. The assignment statement/expression is akin to the pipeline between the processor and the memory;
2. Variables are simply thin abstractions over the memory; and
3. Sequences of instructions (interrupted with conditionals, etc) are a close approximation of the Fetch, Decode, Execute cycle that the CPU repeats ad infinitum.

The assignment statement/expression is tightly related to another hallmark of programs written in imperative/procedural programs: They have *state*: The contents of the memory at a particular point during execution. An assignment statement/expression changes the state of the program. Think about how state makes it difficult (if not impossible) to completely enclose (for lack of a better word) functionality in one self-contained unit.

Imperative languages use conditionals, sequences, loops and functions as their *combining forms*. A combining form is just a phrase that describes how more than one of a language's basic semantic elements can be grouped together with others to build code that performs an action that cannot be completed by one of those single elements.

As a token gift to the programmer to make their life easier, imperative/procedural programs offer *process abstraction* in the form of functions/procedures.

The most famous purely imperative/procedural programming languages are Fortran, Cobol, C and Pascal.

Object Oriented

Once hardware costs began to drop, the dominant cost of writing software shifted from the hardware to the software and, by extension, to the cost of employing developers to write that

software. Programmers began thinking about ways to make programmers more efficient and began to rely on abstractions around data. The focus on *data abstraction* as a means of organizing software projects culminated with the development of object-oriented programming languages. At their core, object-oriented programming languages are defined by their reliance on *encapsulation* of data together with the code that operates on that data. Each entity (*object*) in an object-oriented programming language, each object is responsible for maintaining its internal data and modifying it according to the user's requests.

The data contained by an object is hidden (*data hiding*). Users of that object can access to that data and/or manipulate that data by sending the objects *messages*. *Message passing* is the way that a user accesses (views) an object's internal data or instructs an object to manipulate its internal data. Data hiding gives the object implementer the ability to change the way that the data is represented internally (for expansion, performance or, really, any reason) without having to change the code of software that interacts with it. As long as the object continues to respond to the messages the same way that it always has, then it can represent its data internally however it chooses. Cool!

Objects in an object-oriented programming language can be arranged to form a hierarchy. The descendants of an object are related (somehow) to their parents. Descendants can, in turn, be the parents of another object. Descendant objects *inherit* the functionality of objects from which they descend! Those objects can simply reuse the functionality of their parents *or* modify (even if slightly) the functionality. This technique provides a powerful method for reusing code -- another way to help the programmer improve their efficiency.

But, if every object can (re)implement the methods of its parents, then the language needs a mechanism to determine which version of a given function to call at a given time. For example, if my function expects to be invoked with a Car object, valid arguments may be a Tesla object, a Ford object, or a Rivian object. Which implementation of code that responds to a start message should be invoked? That decision is made at runtime in a process known as *dynamic binding*. Dynamic binding is also known as *runtime polymorphism*.

Object-oriented programming languages are not a complete break from imperative/procedural programming languages. They also have loops, conditionals, sequences and functions.

Well-known object-oriented programming languages include C++, Java, Smalltalk, and Ruby.

The Daily PL - 5/23/2023

Programming Paradigms (continued)

Functional

Now for something entirely different! (First, let's assume that we are only talking about *pure* functional programming languages -- we will talk about the distinction between pure and impure functional programming languages in more depth when we come to that module.)

In programs written in functional programming languages, there is no state! Are you kidding? No, it's amazing.

Moreover, functional programming language do *not* contain loop combining forms -- they contain only recursion to accomplish repeated execution of a block of statements!

The most important component of a functional programming language is, well, functions! In functional programming languages, unlike imperative and object-oriented programming languages, a function produces the same output when given the same input *no matter when it is executed* -- a consequence of the program having no state!

All kinds of great benefits accrue when you can make the assumption that a program has no state. Most importantly, it makes it possible to easily reason about the parts of a program that can execute in parallel!

The final neat thing about functional programming languages is that, like imperative and object-oriented programming languages can do with `int`, `double`, `string` (etc) types, functions are *first-class entities* which means that they can be passed as parameters to other functions, returned as the output of functions, etc. Wait until you see how this power can be put to good use!

The theoretical underpinning of the functional programming paradigm is the *lambda calculus*.

Examples of (again, pure) functional programming languages include Haskell, Miranda, and Elm. Impure functional programming languages (though still occupying a niche like all functional programming languages) are more popular. Examples include: Lisp, Scheme, Racket, Erlang, Clojure, and Scala.

Despite your effort to avoid functional programming, I *bet* that your favorite language includes patterns/syntax that support writing programs in a functional "style". In fact, JavaScript was originally developed as a way to give programmers the ability to write programs for the web browser in a functional way (other examples include Python, C++, Java and Rust).

Logic/Declarative

If you thought that functional programming languages were weird, wild stuff, just wait until you see logic/declarative programming languages.

Though by far the most different paradigm from imperative/procedural, the concept behind logic programs can be stated succinctly: Whereas programs written in imperative, object-oriented and functional programming languages describe *how* to compute a result, programs written in logic programming languages describe the characteristics necessary for a correct solution.


In other words, logic programmers specify the *what* and not the *how*. Want a program that sorts a list, just describe the requirements of a list that is in sorted order (e.g., `for all indexes i and j of an array a where i < j, a[i] < a[j]`) and let the "compiler" figure out how to make it happen!

It's awesome!

The theoretical underpinning of the logic programming paradigm is the *predicate calculus*.

"They" say that SQL is a logic/declarative programming language. I've never understood their rationale, but that's a story for a different day. In this class, we will learn Prolog, (arguably) *the* logic programming language.

The Influence of the von Neumann Architecture on PLs

This computing model has had more influence on the development of PLs than we can imagine. It was proposed by the legendary mathematician **John von Neumann**  (<https://lithub.com/the-real-life-and-times-of-the-scientist-who-inspired-dr-strangelove/>).

There are two hardware components in von Neumann model (the processor [CPU] and the memory) and they are connected by a pipe.

1. The CPU pipes data and instructions (see below) to/from the memory (fetch).
2. The CPU reads that data to determine the action to take (decode).
3. The CPU performs that operation (execute).

Because there is only one path between the CPU and the memory, the speed of the pipe that connects the two is a bottleneck on the processor's efficiency.


The von Neumann model was novel when it was first proposed because of the way that it stores instructions and data together in the same memory. It is different than the Harvard Architecture, a competing architectural proposal at the time, where programs and data are stored in different memory. Interestingly enough, given the presence of a separate instruction and data cache on the CPU, today's von Neumann-based machines take a sip from the Arnold Palmer of Harvard Architecture.

In the von Neumann model, every byte (usually!) of data is accessible according to its address and sequential instructions are placed nearby in memory. For instance, in


```
for (int i = 0; i<100; i++) {  
    statement1;  
    statement2;  
    statement3;  
}
```

`statement1`, `statement2` and `statement3` are all stored one after the other in memory.

We referenced how slow it must be for the CPU to have to ask for each instruction (and each byte of data) individually! So. Slow. And then, once those meticulously collected bytes of data have been transformed by the diligently decoded instructions, the new version must be transmitted (again, one byte at a time) back to the memory! That is terrible. To fix this problem, CPU designers made the bottleneck and CPU more aggressive. They assumed (in hindsight, it was clearly a good assumption) that nearby data and instructions were likely to be used in the near future. In other words, an instruction fetched a byte of data at address, say, `0xf5`, it was likely to need the bytes at `0xf6`, `0xf7`, `0xf8`, etc, with high probability. So, designers added a feature to the CPU where, instead of just getting the byte of memory requested by the CPU, it would get the requested byte plus several of its neighbors.

And this optimization started a virtuous (?) cycle of designers making the **prefetching**  (<https://safari.ethz.ch/architecture/fall2020/lib/exe/fetch.php?media=onur-comparch-fall2020-lecture18-prefetching-afterlecture.pdf>) more aggressive and the programmers pushing it to its limit.

As a result, implementing repeated instructions with loops is faster than implementing repeated loops with recursion. The instructions in the body of the loop are, after all, back to back to back to (you get the idea) in memory, aren't they?