I don't expect anyone to understand all of this at this time. But I thought it important to articulate why Prolog is an interesting language.

How can we position Prolog among programming languages?

Prolog developed from a series of experimental languages by several groups, but generally there's agreement that the first real Prolog was written in 1972 by Alain Colmeraur.

Every computer language has a '**paradigm**' - a model of what computation **'is'** that underlies how we describe what the computer should do.

- For an **imperative** language like Java, computation is a series of instructions for the computer to follow.
- For a **functional** language like Haskell, computation is a function to be evaluated.
- For a **logic** language like Prolog, computation is searching for proofs of propositions in a problem space.
- For a **flow** language like node-red, computation is the flow of data

So what are Prolog's characteristics, and why do they make Prolog worth existing?

Prolog is a **declarative language**. We can describe some facts and rules about our world, then ask Prolog to solve the problem **without ever describing the algorithm** to use.

Prolog has **no true control structures**. No while loops, for loops, or if statements. The body of Prolog clauses (a bit like functions) is just a list of goals. So we can actually **inspect our code and reason about it**. Compare a language like Clojure, where 'parsing' the body of a fn would be quite difficult in practice, though available at runtime. Modifying and inspecting the body of Prolog predicates is somethign people actually do.

Prolog is **homoiconic**. Data is expressed as terms defined in a knowledgebase. Code is also expressed as terms defined in a knowledgebase. So code and data are 'the same thing'. We can even say there is '**no code**' - only entries in the knowledgebase.

Prolog does **proof search** by a method called Selective Linear Definite resolution. In imperative programs, much of our code is to **distinguish between groups of things (an array) and a single thing** (an element of the array). In Prolog, we often **need not worry about this distinctio**n. I sometimes say Prolog data **'carries it's control structure with it**'.

Prolog predicate **arguments bind by unifying**. This means that the same argument can either be used to pass, or return a value, a property called **multimodality**. So, for example, the Prolog predicate append/3, which appends two lists to make a third, can also be used to check if two lists are parts of a third, to get the prefix, the suffix, to find all partitions of a list, and so on. **Libraries are dramatically smaller**. A newly launched instance of SWI-Prolog loads just 2231 predicates on my machine.

Prolog is dual paradigm. If the logic paradigm isn't working for you, you can write pretty much normal imperative code in Prolog. When C++ came out, many C programmers were said to be 'Writing C in C++'. Many beginning Prolog programmers tend to write Java in Prolog for a while.

Prolog, unlike most languages, is usually taught as an academic subject. Naturally the logical side of Prolog is emphasized, and this gives students the impression that one can't or shouldn't think about Prolog's execution order. Of course that's silly - if you're planning a series of moves for a robot, the order you do them in is important!

Prolog is **syntactically plastic**. You can insert arbitrary Prolog code to be run at program load time which will **arbitrarily transform the program**. Prolog uses a very minimal set of hard definitions - there are no keywords, there are only a few fixed syntactic constructions. Many DSLs have been written in Prolog which used only operator definitions, for example. Beyond this, quasiquotes allow translating any sort of foreign code on the fly.

Prolog does not evaluate it's arguments. In virtually every other language, foo(2+1) calls foo with 3. Prolog calls foo with '+'(2,1). This makes **every argument of every predicate effectively a DSL**.

Prolog is semantically plastic. Because you can arbitrarily transform code at load time, and because code is just entries in the knowledgebase, you can alter the semantics of Prolog.

Libraries in SWI-Prolog allow constraint programming (I don't know this variable's value, but I know it's between 0 and 5), probabilistic programming (this variable is Gaussian distributed with a mean of 45 and sigma 15), object oriented programming (SWI-Prolog's built-in IDE is built atop an OO language inside SWI-Prolog), production systems, and more.

If Prolog were a tool, it would be a big scary buzz saw. It has few guards, and in foolish hands Prolog code can become unmaintainable. There is no Prolog equivalent of obfuscated C, since one could easily write Prolog code that no human could read.

People who work around tools soon learn that it's the dull tool that is dangerous, not the sharp one. The dull tool must be forced through the work, hammered on, and when it slips it moves fast. Prolog is a very sharp tool.

Prolog is **typeless**. Any sort of data can be bound to any variable. There have been various responses to this, ranging from 'fixing' it (see Logtalk, for example), to treating type systems as a 'bolt on' (more successful with dynamic typing so far).

However, Prolog is not as anarchic as it appears at first glance. **Unification** of arguments can be interpreted as saying that all dispatch is typed. If a predicate starts executing and realizes it has the wrong type, it can make the execution disappear by **failing**.

Prolog is probably headed for a world where static type systems are provided separately as 'bolt on' libraries (see semantic plasticity), compete for popularity and find ecological niches. The dynamic typing system **mavis** is already widely used.

Prolog encourages writing for cases. Prolog code is in disjunctive normal form -in this case, this rule applies, in this other case, this other rule applies.

Since the syntax encourages this way of writing, edge cases often 'discover themselves'. I often fined myself saying 'oh yes, need to cover that case' when my code makes it obvious.

There are several realistic choices for a Prolog implementation. SWI-Prolog is the 'default', the most widely used, and if you talk with Prolog programmers they'll assume you're talking SWI unless you say otherwisel GProlog has a nice set of constraint libraries, and is sometimes faster, but is poorly maintained. Some commercial users need a commercially supported Prolog, and for that there's SICSTUS. Logtalk and Mercury are alternative logic languages with small user bases.

In this course we'll focus on SWI-Prolog, obviously. I'd encourage deferring an exploration of other Prologs until you're confident in the language.

SWI-Prolog is 'Real World'. That libraries are full featured and performant. The central language is fast and reliable. All of the facilities you'd expect in a 'real' language are there. There are several very large scale Prolog installations deployed - for example, most aircraft gate assignments are done in Prolog. 40% of New Zealand's stock trades are in Prolog. Many 'quant' trading systems are in Prolog. Prolog is used to do NLP tasks for national security.

HIstorically, Prolog implementations have focused on filling the needs of academic research. This has had the good effect that Prolog is blessed with an amazing set of libraries to do esoteric things, the legacy of decades of PhD theses written in and about Prolog. The bad side was that until the past 10 years or so Prolog implementations often lacked library support for pedestrian things like UDP and software engineering support for large projects.

This changed when Jan Wielemaker, and before him Anjo Anjewierden, picked up maintainence of the codebase in 2005. Jan is a professor at VNU in Amsterdam, and VNU has been incredibly supportive of Jan working on SWI-Prolog as his professional research commitment.

The SWI-Prolog codebase's oldest portions date back to the early 80's, but the codebase has been maintained as a standalone implementation since 1986, when a new implementation that could call C and have C in turn call Prolog recursively was needed. When we say 'mature' we're not kidding.

In 2014, SWI-Prolog made several fundamental changes to the language, breaking compatibility, to reflect changes in the computing environment (notably strings) and advances in software practice since the 1970's. We'll cover those changes in the course.

SWI-Prolog is 'batteries included'.

I've personally try very hard to only work where I can primarily program in Prolog, since discovering the language. Given it's small user base, that's been hard.

Why? It's not that I particularly love logic languages, or Prolog's syntax, or any of the features described above, though I do appreciate them. It's that installing SWI-Prolog pretty much means you have finished messing with your SWI-Prolog system. **Time not spent doing sysadmin chores** means time spend programming business logic.

The default install comes with an IDE, a web framework, ODBC connection, etc. etc. - If you need something not in the install, it's probably coming from a pack - SWI-Prolog's answer to Ruby gems, and just as easy to use. There's really nothing else to set up to start developing in SWI-Prolog.

This 'batteries included' philosophy is a sharp contrast to the common phenomenon in functional languages "**I love Foo#, but hate <*name of virtually required ecosystem piece*>**". No Cabal, Leiningen, etc. Things that might be extra pieces in another language - compiler, autodoc tool, IDE, etc etc ad nauseum, tend to just come in the default install. Currently the only 'pretty much have to use' is Falco Nogatz's swivm tool, which isn't really that necessary, and which I expect swivm to be rolled into SWI-Prolog at some point.

SWI-Prolog code compiles to a low level machine called ZIP. ZIP code is never stored on disk. It is possible to create a binary executable. The executable bundles a ZIP interpreter and minimized set of libraries, so the user need not install SWI-Prolog. But the usual way to run SWI-Prolog programs is to just load from source files at startup.

The plasticity and expressiveness of Prolog means libraries tend to look after themselves. SWI-Prolog is all about reducing ceremony. So the user usually just needs to say "I want this". The famous 'import antigravity' XKCD cartoon about Python applies even more to Prolog.

Prolog is not a 'dead' language. The SWI-Prolog site is, you can see (at swi-prolog.org) a pretty bare-bones site, designed for rapid download. Yet our servers serve around 5GB/month, and we need a CDN and multiple servers for the site (which is, incidentally, a wiki entirely written in SWI-Prolog).

Usage appears to be growing. I've tried twice before to organize a class like this, and got lukewarm response. Now I'm rather taken aback - I've got 250+ students!

That's where Prolog lies, and SWI-Prolog.