

The Daily PL - 7/25/2023

Let's get the party started!

I Do Declare

In this module we are going to focus on logic (also known as declarative) programming languages. Users of a declarative programming language *declare* the outcome they wish to achieve and let the compiler do the work of achieving it. This is in marked contrast to users of an imperative programming language who have to tell the compiler not only the outcome they wish to achieve but also *how* to achieve it. A declarative programming language does not have program control flow constructs, per se. Instead, a declarative programming language gives the programmer the power to control execution by means of recursion (again?? I know, sorry!) and *backtracking*. Backtracking is a concept that we will return to later. A declarative program is all about defining facts (either as axioms or as ways to build new facts from axioms) and asking questions about those facts. From the programming language's perspective, those facts have no inherent meaning. We, the programmers, have to impugn meaning on to the facts. Finally, declarative programming languages do have variables, but they are nothing like the variables that we know and love in imperative programming languages.

As we have worked through the different programming paradigms, we have discussed the theoretical underpinning of each. For imperative programming languages, the theoretical model is the Turing Machine. For the functional programming languages, the theoretical model is the Lambda Calculus. The declarative programming paradigm has a theoretical underpinning, too: first-order predicate calculus. We will talk more about that in class soon!

In the Beginning

Unlike imperative, object-oriented and functional programming languages, there is really *only one* extant declarative/logic programming language: Prolog. **Prolog was developed by Alain Colmerauer, Phillipe Roussel, and Robert Kowalski in order to do research in artificial intelligence and natural language processing** (<https://uc.instructure.com/courses/1610326/files/167463789?wrap=1>). Its official birthday is 1972.



Prolog programs are made up of exactly three components:

1. Facts
2. Rules
3. Queries

The syntax of Prolog defines the rules for writing facts, rules and queries. Syntactic elements in Prolog belong to one of three categories:

1. Atoms: The most fundamental unit of a Prolog program. They are simply symbols. *Usually* they are simply sequences of characters that begin with a lowercase letter. However, atoms can contain spaces (in which case they are enclosed in 's) and they can start with uppercase letters (in which case they are wrapped with 's).
2. Variables: Like atoms, but variables always start with uppercase letters.
3. Functors: Like atoms, but functors define relationships/facts.

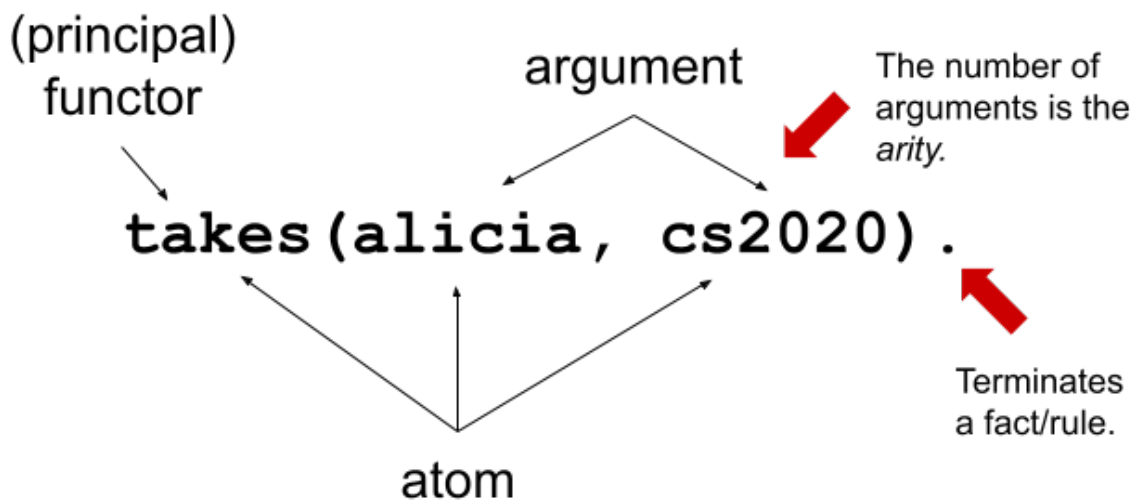
If Prolog is a logic programming language, there must be some means for specifying logical operations. There is! In the context of specifying a rule, the *and* operation is written using a `&`. In the context of specifying a rule, the *or* operation is written using a `;`. Obviously!

The best way to learn Prolog is to start writing some programs. We'll come back to the theory later!

Just The Facts

At its most basic, a Prolog program is a set of facts:

```
takes(jane, cs4999).
takes(alicia, cs2020).
takes(alice, cs4000).
takes(mary, cs1021).
takes(bob, cs1021).
takes(kristi, cs4000).
takes(sam, cs1021).
takes(will, cs2080).
takes(alicia, cs3050).
```



In relation to the first-order predicate logic that Prolog models, *takes* is a logical predicate. We'll refer to them as facts or predicates, depending on what's convenient. Formally, the predicates and facts are written as `<principle functor>/<arity>`. Two (or more) facts/predicates with the same functor but different arities are not the same. For instance, *takes/1* and *takes/2* are completely different.

Let's *read* one of these facts in English:

```
takes(jane, cs4999).
```

could be read as "Jane takes CS4999.". As programmers, we know what that means: the student named Jane is enrolled in the class CS4999. However, Prolog does not share our sense of meaning! Prolog simply thinks that we are defining one element of the *takes* relationship where `jane` is *somehow* related to `cs4999`. As a Prolog programmer, we could just have easily have written

```
tennis_shoes(jane, cs4999).
tennis_shoes(alicia, cs2020).
tennis_shoes(alice, cs4000).
tennis_shoes(mary, cs1021).
tennis_shoes(bob, cs1021).
tennis_shoes(kristi, cs4000).
tennis_shoes(sam, cs1021).
tennis_shoes(will, cs2080).
tennis_shoes(alicia, cs3050).
```

and gotten the same result! But, we programmers want to define something that is meaningful, so we choose to use atoms that reflect their semantic meaning. With nothing more than the facts that we have defined above, we can write *queries*. In order to interact with queries in real time, we can use the Prolog REPL. Once we have started the Prolog REPL, we will see a prompt like this:

```
?-
```

The world awaits ...

To load a Prolog file in to the REPL, we will use the consult predicate:

```
?- consult('intro.pl').  
true.
```

The Prolog facts, rules and queries in the `intro.pl` file are now available to us. Assume that `intro.pl` contains the `takes` facts from above. Let's make some queries:

```
?- takes(bob, cs1021).  
true.  
  
?- takes(will, cs2080).  
true.  
  
?- takes(ali, cs4999).  
false.
```

These are simple yes/no queries and Prolog obliges us with terse answers. But, even with the simple facts shown above, Prolog can be used to make some powerful inferences. Prolog can tell us the names of all the people it knows who are taking a particular class:

```
?- takes(Students, cs1021).  
Students = mary ;  
Students = bob ;  
Students = sam.
```

Wow! Here Prolog is telling us that there are three different values of the `Students` variable that will make the query true: `mary`, `bob` and `sam`. In the lingo, Prolog is *unifying* `Students` with the values that will make our query true. Let's go the other way around:

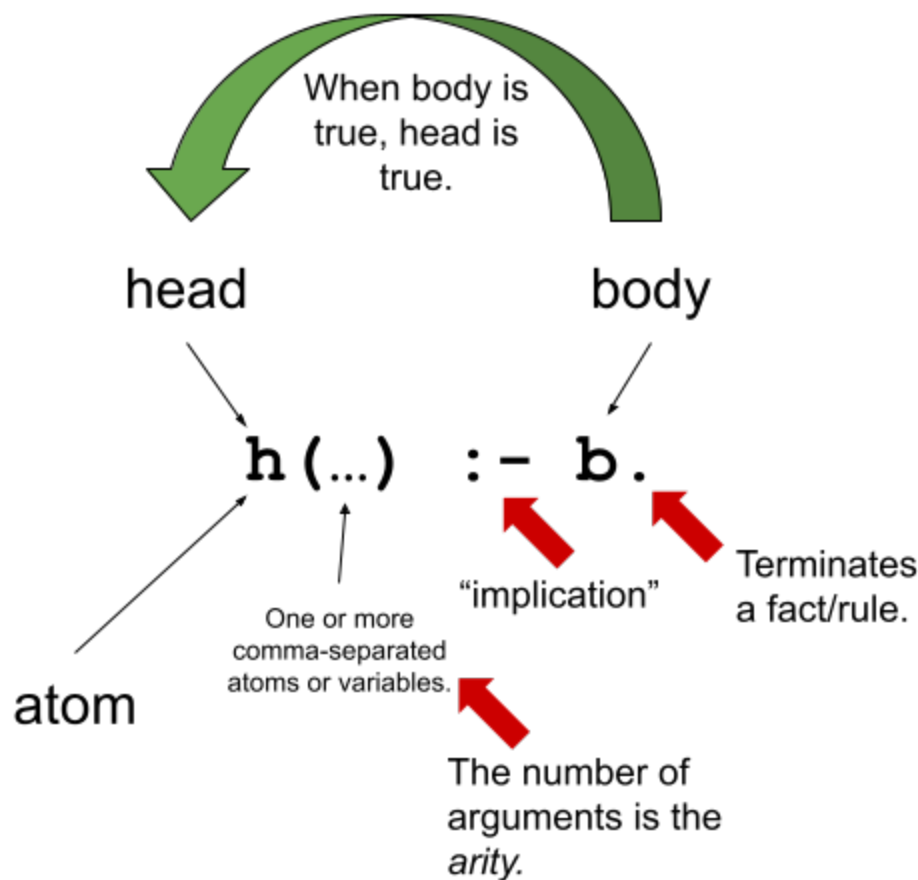
```
?- takes(alicia, Classes).  
Classes = cs2020 ;  
Classes = cs3050.
```

Here Prolog is telling us that there are two different classes that Alicia is taking. Nice.

That's great, but pretty limited: it's kind of terrible if we had to write out each fact explicitly! The good news is that we don't have to do that! We can use a Prolog *rule* to define facts based on the existence of other facts. Let's define a rule which will tell us the students who are CS majors. To be a CS major, you must be taking (exactly) two classes:

```
cs_major(X) :- takes(X, Y), takes(X, Z), Y @< Z.
```

That's lots to take in at first glance. Start by looking at the general format of a rule:



Okay, so now back to our particular rule that defines what it means to be a CS Major. (For the purposes of this discussion, assume that the `@<` operator is "not equal"). Building on what we know (e.g., `,` is *and*, `:-` is implication, etc), we can read the rule like: "X is a CS Major if X takes class Y and X takes class Z and class Y and Z are not the same class." Pretty succinct definition.

To make the next few examples a little more meaningful, let's update our list of facts before moving on:

```
takes(jane, cs4999).
takes(alicia, cs2020).
takes(alice, cs4000).
takes(mary, cs1021).
takes(bob, cs1021).
takes(bob, cs8000).
takes(kristi, cs4000).
takes(sam, cs1021).
takes(will, cs2080).
takes(alicia, cs3050).
```

With that, let's find out if our rule works as intended!

```
?- cs_major(alicia).
true ;
false.
```

Wow! Pretty cool! Prolog used the rule that we wrote, combined it with the facts that it knows, and inferred that Alicia is a CS major! (For now, disregard the second False -- we'll come back to that!). Like we could use Prolog to generate the list of classes that a particular student is taking, can we ask Prolog to generate a list of all the CS majors that it knows?

```
?- cs_major(X).  
X = alicia ;  
X = bob ;  
false.
```

Boom!

The retreat to move forward.

Backtracking

Understanding the concepts of *backtracking* and *choice points* is incredibly important for understanding how Prolog determines the meaning of our programs.

There is a formal definition of *choice points* and *backtracking* from the Prolog glossary:

backtracking: Search process used by Prolog. If a predicate offers multiple clauses to solve a goal, they are tried one-by-one until one succeeds. If a subsequent part of the proof is not satisfied with the resulting variable binding, it may ask for an alternative solution, causing Prolog to reject the previously chosen clause and try the next one.

There are lots of undefined words in that definition! Let's dig a little deeper.

A predicate is like a boolean function. A predicate takes one or more arguments and yields true/false. As we said at the outset of our discussion about declarative programming languages, the *reason* that a predicate may yield true or false depends on the semantics imposed upon it from the outside. A predicate is a term borrowed from first-order logic, a topic that we will return to later.

Remember our *takes* example? *takes* is a predicate! *takes* has two arguments and returns a boolean.

In Prolog, rules and facts are written to define predicates. A rule defines the conditions under which a predicate is true using a body -- a list of other predicates, logical conjunctives, implications, etc. A fact is a rule without a body and unconditionally defines that a certain relationship is true for a predicate.

```
related(will, bill).
related(ali, bill).
related(bill, william).

related(X, Y) :- related(X, Z), related(Z, Y).
```

In the example above, *related* is a predicate defined by facts and rules. The facts and rules above are the *clauses* of the predicate.

choicepoint: A choice point represents a choice in the search for a solution. Choice points are created if multiple clauses match a query or using disjunction (`;/2`). On backtracking, the execution state of the most recent choice point is restored and search continues with the next alternative (i.e., next clause or second branch of `;/2`).

That's a mouthful! I think that the best way to understand this is to look at backtracking in action and see where choice points exist.

Give and Take

Remember the *takes* predicate that we defined previously:

```
takes(jane, cs4999).
takes(alicia, cs2020).
takes(alice, cs4000).
takes(mary, cs1021).
takes(bob, cs1021).
takes(bob, cs8000).
takes(kristi, cs4000).
takes(sam, cs1021).
takes(will, cs2080).
takes(alicia, cs3050).
```

We subsequently defined a *cs_major* predicate:

```
cs_major(X) :- takes(X, Y), takes(X, Z), Y @< Z.
```

The *cs_major* predicate says that a student who takes two CS classes is a CS major. Let's walk through how Prolog would respond to the following query:

```
cs_major(X).
```

To start, Prolog realizes that in order to satisfy our query, it has to *at least* satisfy the query

```
takes(X, Y).
```

So, Prolog starts there. In order to satisfy that query, it searches its knowledge base (its list of facts/rules that define predicates) from top to bottom. X and Y are variables and the first appropriate fact that it finds is

```
takes(jane, cs4999).
```

So, it *unifies* X with jane and Y with cs4999. Having satisfied *that* query, Prolog realizes that it must also satisfy the query:

```
takes(X, Z).
```

However, Prolog has already provisionally unified X with jane. So, Prolog really attempts to satisfy the query:

```
takes(jane, Z).
```

Great. For Prolog, attempting to satisfy this query is *completely* distinct from its attempt to satisfy the query takes(X, Y) which means that Prolog starts searching its knowledge base anew (again, from top to bottom!). The first appropriate fact that it finds that satisfies this query is

```
takes(jane, cs4999).
```

So, it unifies Z with cs4999. Having satisfied *that* query too, Prolog moves on to the third query:

```
Y @< Z.
```

Unfortunately, because Z and Y are both unified to cs4999, Prolog *fails* to satisfy that query. In other words, Prolog realizes that its provisional unification of X with jane, Y with cs4999 and Z with cs4999 is not a way to satisfy our original query (cs_major(X)).

Does Prolog just give up? Absolutely not! It's persistent. It backtracks! To where?

Well, according to the definition above, it backtracks to the most-recent choicepoint! In this case, the most recent choicepoint was its provisional unification of Z with cs4999. So, Prolog forgets about that attempt, and restarts the search of its knowledge base.

Where does it restart that search, though? This is important: It restarts its search *where it left off*. In other words, it starts at the first fact at takes(jane, cs4999). Because there are no other facts about classes that Jane takes, Prolog fails *again*, this time attempting to satisfy the query takes(jane, Z).

I ask again, does Prolog just give up? No, it backtracks again! This time it backtracks to its most-recent choicepoint. Now, that most recently choicepoint was its provisional unification of X with jane. Prolog forgets that attempt, and restarts the search of its knowledge base! Again, because this is the continuation of a previous search, Prolog begins where it left off in its top-to-bottom search of its knowledge base. The next fact that it see is

```
takes(alicia, cs2020)
```


So, Prolog provisionally unifies X with $alicia$ and Y with $cs2020$. Having satisfied *that* query (for a second time!), Prolog realizes that it must also satisfy the query:

```
takes(X, Z).
```

However, Prolog has provisionally unified X with $alicia$. So, Prolog really attempts to satisfy the query:

```
takes(alicia, Z).
```

Great. For Prolog, attempting to satisfy this query is *completely* distinct from its attempt to satisfy the query $takes(X, Y)$ *and* its previous attempt to satisfy the query $takes(jane, Z)$. Therefore, Prolog starts searching its knowledge base anew (again, from top to bottom!). The first appropriate fact that it finds that satisfies this query is

```
takes(alicia, cs2020).
```

So, it unifies Z with $cs2020$. Having satisfied *that* query too, Prolog moves on to the third query:

```
Y @< Z.
```

Unfortunately, because Z and Y are both unified to $cs2020$, Prolog *fails* to satisfy that query. In other words, Prolog realizes that its provisional unification of X with $alicia$, Y with $cs2020$ and Z with $cs2020$ is not a way to satisfy our original query ($cs_major(X)$). Again, Prolog does not give up and it backtracks to its most recent choicepoint. The good news is that Prolog can satisfy the query

```
takes(alicia, Z)
```

a second way by unifying Z with $cs3050$. Prolog proceeds to the final part of the rule

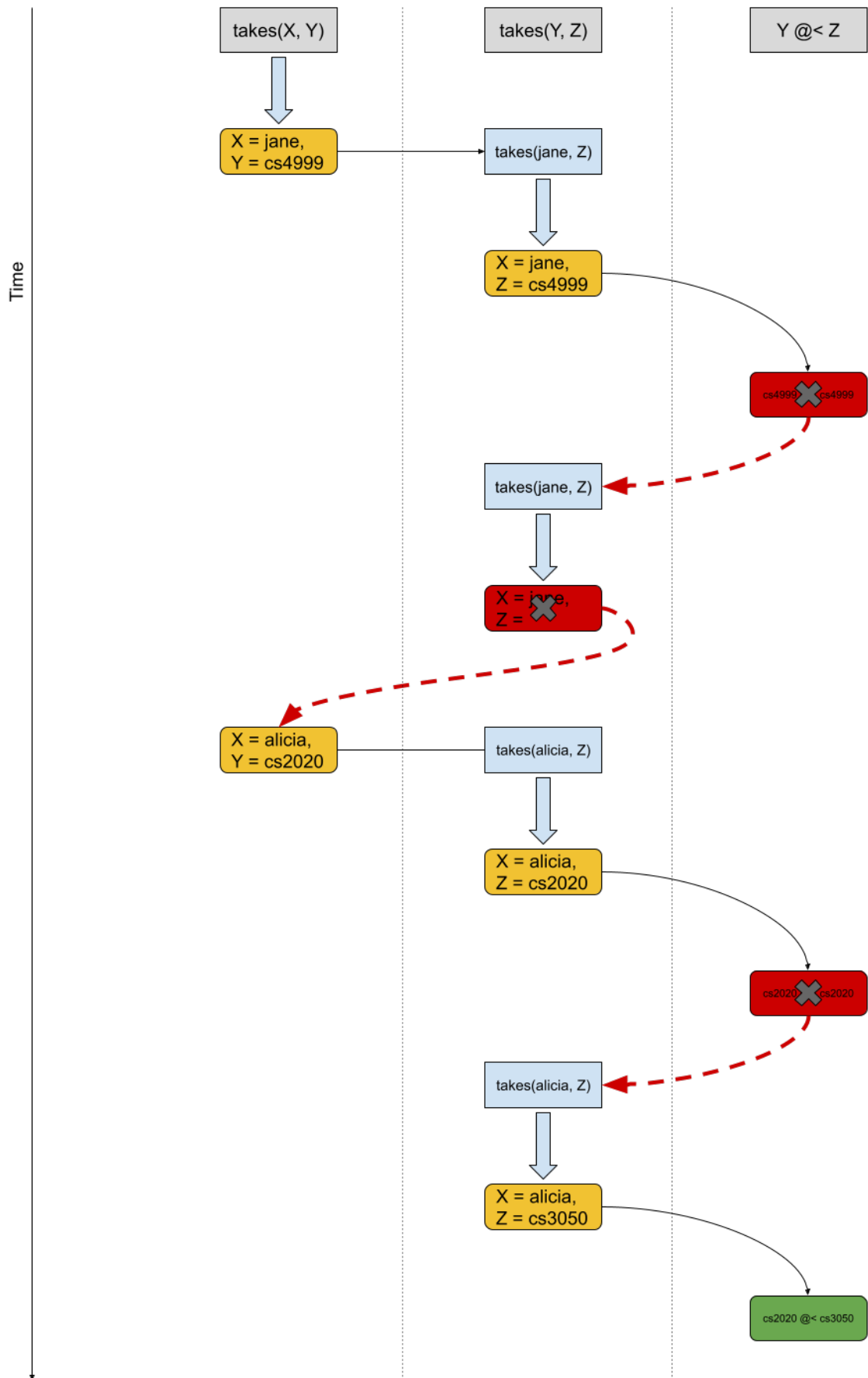
```
X @< Y
```

which can be satisfied this time because $cs3050$ and $cs2020$ are different!


Victory!

Prolog was able to satisfy the original query when it unified X with $alicia$, Y with $cs2020$ and Z with $cs3050$.

Below is a visualization of the description given above:




A Prolog user at the REPL (or a Prolog program using this rule) could ask for *all* the ways that this query is satisfied. And, if the user/program does, then Prolog will backtrack as if it did not find a satisfactory unification for Z, Y or X (in that order!).

In the true spirit of a [picture being worth a thousand words](https://en.wikipedia.org/wiki/A_picture_is_worth_a_thousand_words)  (https://en.wikipedia.org/wiki/A_picture_is_worth_a_thousand_words), think of this Daily PL section as a graphic novel.

Going Over Backtracking Again (see what I did there?)

Let's dive deeper into the discussion of backtracking and its utility. In particular, we discussed the following Prolog program for *generating* all the integers.

```
generate_integer(0).  
generate_integer(X) :- generate_integer(Y), X is Y + 1.
```

This is an incredibly *succinct* way to declare what it means to be an integer. This generator function is attributable to [Programming in Prolog by Mellish and Clocksin](https://link.springer.com/book/10.1007/978-3-642-55481-0)  (<https://link.springer.com/book/10.1007/978-3-642-55481-0>). In other words, we know that it's a reasonable definition. The best way to *learn* Prolog, I think, is to play with it! So, let's see what this can do!

```
?- generate_integer(5).  
true ;
```

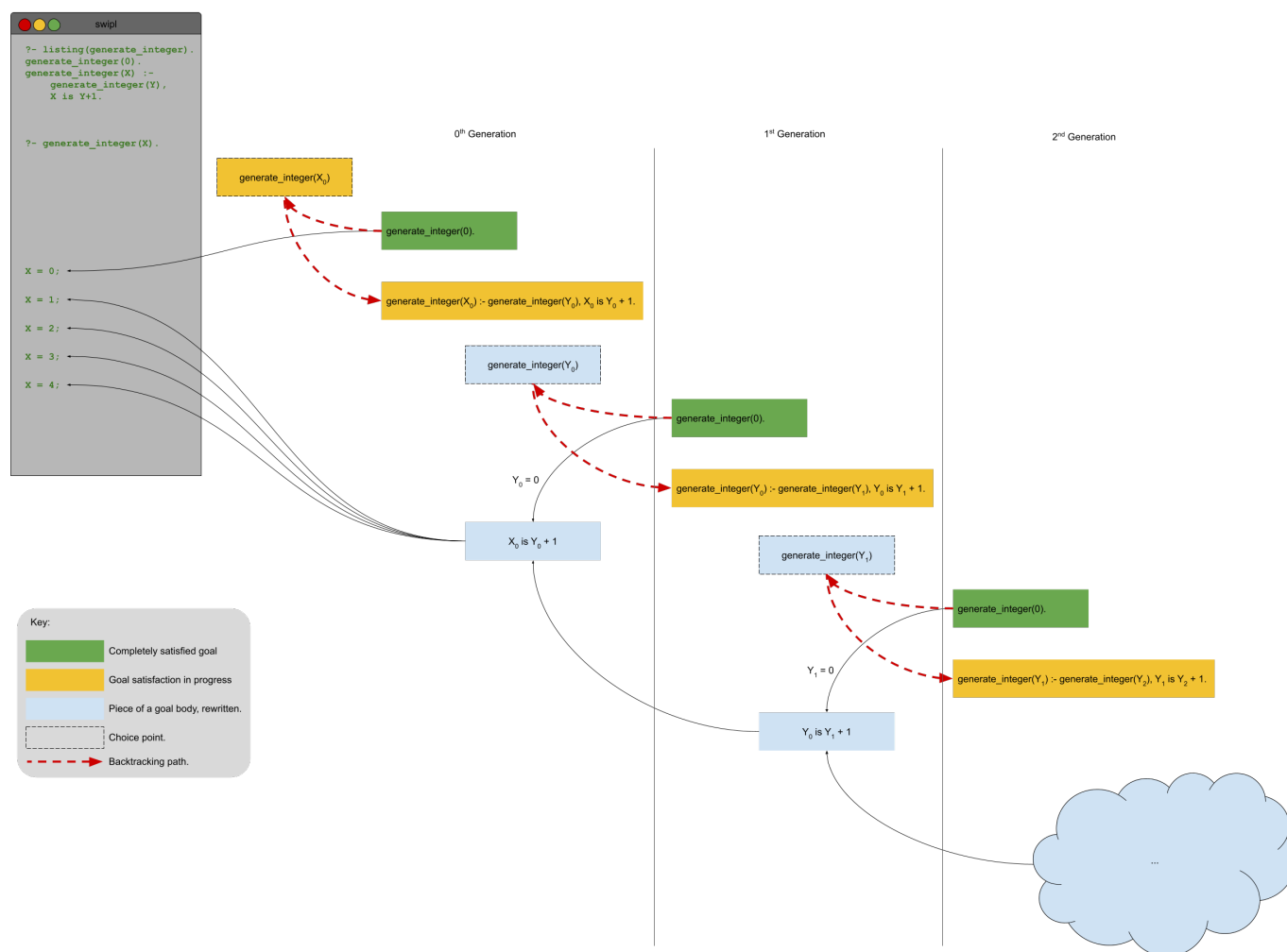
In other words, it can be used to determine whether a given number is an integer. Awesome.

```
?- generate_integer(X).  
X = 0 ;  
X = 1 ;  
X = 2 ;  
X = 3 ;  
X = 4 ;  
X = 5 ;  
X = 6
```

Woah, look at that ... *generate_integer/1* can do double duty and generate all the integers, too. Pretty cool!

The generation of the numbers using this definition is possible thanks to the power of backtracking.

The (im)possibility of using words to describe the backtracking involved in *generate_integer/1*, led me to create the following diagram that attempts to illustrate the process. I encourage you to look at the diagram, run *generate_integer/1* in swipl with trace enabled and ask any questions that you have! Again, this is *not* a simple concept, but once you understand what is going on, Prolog will begin to make more sense!



It may be necessary to use a high-resolution version of the diagram if you are curious about the details. Such a version is available in SVG format [here \(https://uc.instructure.com/courses/1610326/files/167463775?wrap=1\)](https://uc.instructure.com/courses/1610326/files/167463775?wrap=1).

Chasing Our Tails

Yes, I can hear you! I know, `generate_integer/1` is *not* tail recursive. We learned that tail recursion is a useful optimization in functional programming languages (and even imperative programming languages). Does that mean that it's an important optimization in Prolog?

To find out, I timed how long it took Prolog to answer the query

```
?- generate_integer(50000).
```

The answer? On my desktop computer, it took 1 minute and 48 seconds.

If we want something for comparison, we'll have to come up with a tail-recursive version of `generate_integer/1`. Let's call it `generate_integer_tr/1` (creative, I know), and define it like:

```
generate_integer_tr(X) :- next_integer(0,X).

next_integer(J, J).
next_integer(J, L) :- K is J + 1, next_integer(K, L).
```

The fact `next_integer(J, J)` is a "trick" to define a base case for our definition in the same way that `generate_integer(0)` was a base case in the definition of `generate_integer/1`. To get a sense for the need for `next_integer(J, J)` think about what happens when the first query of `next_integer(0,X)` is performed in order to satisfy the query `generate_integer_tr(50000)`. In this case, the `next_integer(J, J)` fact matches (convince yourself why! Hint: there are no restrictions on J). As a result, `J` unifies with the 0, and the `X` unifies with the `J`. That's great, but `(X =) 0` does not equal 50000. So, Prolog does what?

In unison: BACKTRACKS.

The choice point is Prolog's selection of `next_integer(J, J)`, so Prolog tries again at the next possible fact/rule: `next_integer(J, L) :- K is J + 1, next_integer(K, L)`. `J` is unified with 0, `K` is unified with 1 (`J + 1`) and Prolog must now satisfy a new goal: `next_integer(1, L)`. Because *this* query for `next_integer/1` is completely different than the one it is currently attempting to satisfy, Prolog starts the search anew at the top of the list of facts. The first fact to match? `next_integer(J, J)`. Therefore, `J` unifies with 1, `L` unifies with `J` (making `L` 1), and `X` (from the initial query) unifies with `L`. Unfortunately, the result again does not satisfy our query. But, Prolog persists and backtracks but *only as far as the second attempt to satisfy next_integer (using next_integer(J, J))*. In much the same way that `generate_integer/1` worked, Prolog continues to progress, then backtrack, then progress, then backtrack ... while solving the `generate_integer_tr(50000)` query.

The difference between the two functions is that in `next_integer/2`, the recursive act is the last thing that is done. In other words, `generate_integer_tr/1` is tail recursive.

Does this impact performance? *Absolutely!* On my desktop. Prolog can answer the query `generate_integer_tr(50000)` in 0.029 seconds. Yeow!

The Daily PL - 7/27/2023

One amazing function in logic programming epitomizes the beauty of the whole system: *append/3*. *append/3* can be used to implement many other different rules, including a rule that will generate all the permutations of a list!

Pin the Tail on the List

The goal (pun intended) of *append* is to determine whether two lists, X and Y, are the same as a third list, Z, when appended together. We could use the *append* query like this:

```
?- append([1,2,3], [a,b,c], [1,2,3,a,b,c]).  
true.
```

or

```
?- append([1,2,3], [a,b], [1,2,3,a,b,c]).  
false.
```

What we will find is that *append/3* has a hidden superpower besides its ability to simply answer yes/no queries like the ones above.

The definition of *append/3* will follow the pattern of other recursively defined rules that we have seen so far. Let's start with the base case. The result of appending an empty list with some list Y, is just list Y. Let's write that down:

```
append([], Y, Y).
```

And now for the recursive case: appending list X to list Y yields some list Z where Z is the first element of the list X following by the result of appending the *tail* of X with Y. The natural language version of the definition is complicated but I think that the Prolog definition makes it more clear:

```
append([H|T], Y, [H|AppendedList]) :- append(T, Y, AppendedList).
```

Let's see how Prolog attempts to answer the query `append([1,2], [a,b], [1,2,a,b])`.

```
append([1,2], [a,b], [1,2,a,b]).
```

```
H = 1, T = [2], Y = [a,b], AppendedList = [2,a,b]
```

```
append([2], [a,b], [2,a,b]).
```

```
H = 2, T = [], Y = [a,b], AppendedList = [a,b]
```

```
append([], [a,b], [a,b]).
```

And now let's look at it's operation for the query `append([1,2], [a,b], [1,a,b]).`

```
append([1,2], [a,b], [1,a,b]).
```

```
H = 1, T = [2], Y = [a,b], AppendedList = [a,b]
```

```
append([2], [a,b], [a,b]).
```

These elements do not match! Therefore, Prolog cannot apply either rule defining *append/3* and the query fails.

It's also natural to look at *append/3* as a tool to "assign" a variable to be the result of appending two lists:

```
?- append([1,2,3], [a,b,c], Z).
Z = [1, 2, 3, a, b, c].
```

Here we are asking Prolog to assign variable *Z* to be a list that holds the appended contents of the lists in the first two arguments.

Look at Append From a Different Angle

We've seen *append/3* in action so far in a *procedural* way -- answering whether two lists are equal to one another and "assigning" a variable to a list that holds the contents of another two lists appended to one another. But earlier I said that *append/3* has some magical powers.


If we look at *append/3* from a different angle, the *declarative* angle, we can see how it can be used to *generate* all the different combinations of two lists that, when appended together, yield a third list! For example,

```
?- append(X, Y, [1,2,3]).  
X = [],  
Y = [1, 2, 3] ;  
X = [1],  
Y = [2, 3] ;  
X = [1, 2],  
Y = [3] ;  
X = [1, 2, 3],  
Y = [] ;
```

Wow. *That's* pretty cool! Prolog is telling us that it can figure out three different combinations of lists that, when appended together, will equal the list [1,2,3]. I mean, if that doesn't make your blood boil, I don't know what will.

The bottom line is that you can look at predicates in Prolog declaratively or procedurally. Depending upon your vantage point, predicates can be used to accomplish different ends.

Let's Ride the Thoroughbred

The power of *append/3* to be used declaratively and procedurally makes it useful in so many different ways. When I started learning Prolog, the resource I was using ([Learn Prolog Now](http://www.let.rug.nl/bos/lpn/) ) spent an inordinate amount of time discussing *append/3* and its utility. It took me a long time to really understand the author's point. A long time.

Prefix and Suffix

Let's take a quick look at how to define a rule *prefix/2*. *prefix/2* takes two arguments -- a possible prefix, *PP*, and a list, *L* -- and determines whether *PP* is a prefix of *L*. We've gotten so used to writing recursive definitions, it seems obvious that we would define *prefix/2* using that pattern. In the base case, an empty list is a prefix of any list:

```
prefix([], _).
```


(Remember that `_` is "I don't care."). With that out of the way, we can say that PP is a prefix of L if

1. the head element of PP is the same as the head element of L, and
2. the tail of PP is a prefix of the tail of L:

```
prefix([H|Xs], [H|Ys]) :- prefix(Xs, Ys).
```

Fantastic. That's a pretty easy definition and it works in all the ways that we would expect:

```
?- prefix([1,2], [1,2,3]).
true.

?- prefix([1,2], [2,3]).
false.

?- prefix(X, [1,2,3]).
X = [] ;
X = [1] ;
X = [1, 2] ;
X = [1, 2, 3] ;
```

But, what if there was a way that we could write the definition of *prefix*/2 more succinctly!

Remember, programmers are lazy -- the fewer keystrokes the better!

Think about this alternate definition of prefix: PP is a prefix of L, when there is a (possibly empty) list, W (short for "whatever"), such that PP appended with W is equal to L. Did you see the magic word? Appended! We have *append*/3, so let's use it:

```
prefix(PP, L) :- append(PP, _, L).
```

(Note that we have replaced W from a natural-language definition with `_` because we don't care about it's value!)

We could go through the same process of defining *suffix*/2 recursively, or we could cut to the chase and define it in terms of *append*/3. Let's save ourselves some time: SS is a suffix of L, when there is a (possibly empty) list, W (short for "whatever"), such that W appended with SS is equal to L. Let's codify that:

```
suffix(SS, L) :- append(_, SS, L).
```

But, does it work?

```
?- suffix_append([3,4], [1,2,3,4]).
true.

?- suffix_append([3,5], [1,2,3,4]).
false.

?- suffix_append(X, [1,2,3,4]).
X = [1, 2, 3, 4] ;
X = [2, 3, 4] ;
```

```
X = [3, 4] ;
X = [4] ;
X = [] ;
```

Permutations

We're all friends here, aren't we? Good. I have no problem admitting that I am terrible at thinking about permutations of a set. I have tried and tried and tried to understand the section in Volume 4 of **Knuth's TAOCP** [↗\(https://www-cs-faculty.stanford.edu/~knuth/taocp.html\)](https://www-cs-faculty.stanford.edu/~knuth/taocp.html) about **generating permutations** [↗\(https://praeclarum.org/2006/04/14/knuths-solution-to-the-permutation-problem.html\)](https://praeclarum.org/2006/04/14/knuths-solution-to-the-permutation-problem.html) but it's just too hard for me. Instead, I just play around with them until I grok it. To help me play, I want Prolog's help. I want Prolog to generate for me all the permutations of a given list. We will call this *permute/2*. Procedurally, *permute/2* will say whether its second argument is a permutation of its first argument. Declaratively, *permute/2* will generate a list of permutations of elements in the list in its first argument. Let's work back from the end: We'll see how it should work before actually defining it:

```
?- permutation([1,2,3], [3,1,2]).
true .

?- permutation([1,2,3], L).
L = [1, 2, 3] ;
L = [1, 3, 2] ;
L = [2, 1, 3] ;
L = [2, 3, 1] ;
L = [3, 1, 2] ;
L = [3, 2, 1] ;
false.
```

Cool. If I run *permute/2* enough times I might start to understand them!

Now that we know the results of invoking *permute/2*, how are we going to define it? Again, let's take the easy way out and see the definition and then walk through it piece-by-piece in order to understand its meaning:

```
permute([], []).
permute(List, [X|Xs]) :- append(W, [X|U], List), append(W, U, ListWithoutX), permute(ListWithoutX, Xs).
```

Well, the first rule is simple -- the permutation of an empty list is just the empty list!

The second rule, well, not so much! There are three conditions that must be satisfied for a list defined as `[X|Xs]` to be a permutation of List.

Think of the first condition in a declarative sense: "Prolog, make me two lists that, when appended together, equal List. Name the first one W. And, while you're at it, make sure that the first element in the second list is X and call the tail of the second list U. Thanks."

Think of the second condition in a procedural sense: "Prolog, append W and U to create a new list

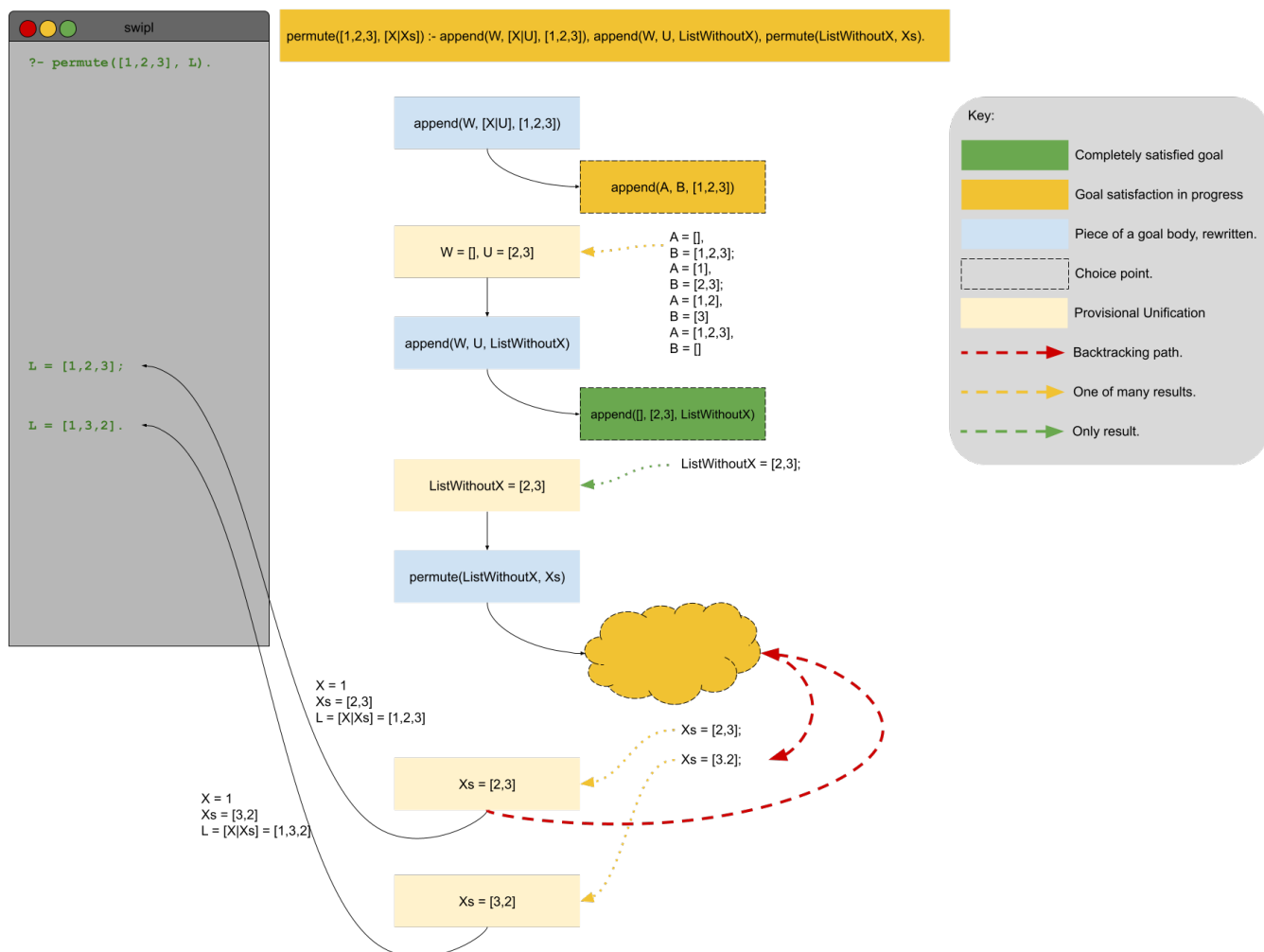
named ListWithoutX." The name ListWithoutX is pretty evocative because, well, it is a list that contains every element of List but X.

Finally, think of the third condition in a declarative sense: "I want Xs to be all the possible permutations of ListWithoutX -- Prolog, make it so!"

Let's try to put all this together into a succinct natural-language definition: A list whose first element is X and whose tail is Xs is a permutation of List if:

1. X is one of the elements of List, and
2. Xs is a permutation of the list List without element X.

Below is a visualization of the process Prolog takes to generate the first two permutations of [1,2,3]:



If *append/3* is a horse that we can ride to accomplish many different tasks, then Prolog is like a wild stallion that tends to run in a direction of its choosing. We can use *cuts* to tame our mustang and make it go where we want it to go!

The Possibilities Are Endless

Let's start the discussion by writing a *merge/3* predicate. The first two arguments are sorted lists. The final argument should unify to the in-order version of the first two arguments merged. Before starting to write some Prolog, let's think about how we could do this.

Let's deal with the base cases first: When either of the first two lists are empty, the merged list is the non-empty list. We'll write that like

```
merge(Xs, [], Xs).
merge([], Ys, Ys).
```

And now, for the recursive cases: We will call the first argument Left, the second argument Right, and the third argument Sorted. The first element of Left can be called HeadLeft and the rest of Left can be called TailLeft. The first element of Right can be called HeadRight and the rest of Right can be called TailRight. In order to merge, there are three cases to consider:

1. HeadLeft is less than HeadRight
2. HeadLeft is equal to HeadRight
3. HeadRight is less than HeadLeft

For case (1), the head of the merged result is HeadLeft and the tail of the merged result is the result of merging TailLeft with Right. For case (2), the head of the merged result is HeadLeft and the tail of the merged result is the result of merging TailLeft with TailRight. For case (3), the head of the merged result is HeadRight and the tail of the merged result is the result of merging Left with TailRight.

It's far easier to write this in Prolog than English:

```
merge([X|Xs], [Y|Ys], [X|Zs]) :- X<Y, merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [X|Zs]) :- X==Y, merge(Xs, Ys, Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :- Y<X, merge([X|Xs], Ys, Zs).
```

(Note: `==` is the "equal to" boolean operator in Prolog. See <https://www.swi-prolog.org/pldoc/man?predicate=%3D%3A%3D/2> for more information.

For *merge/3*, let's write the base cases after our recursive cases. With that switcheroo, we have the following complete definition of *merge/3*:

```
merge([X|Xs], [Y|Ys], [X|Zs]) :- X<Y, merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [X|Zs]) :- X==Y, merge(Xs, Ys, Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :- Y<X, merge([X|Xs], Ys, Zs).
merge(Xs, [], Xs).
merge([], Ys, Ys).
```

Let's follow Prolog as it attempts to use our predicate to answer the query

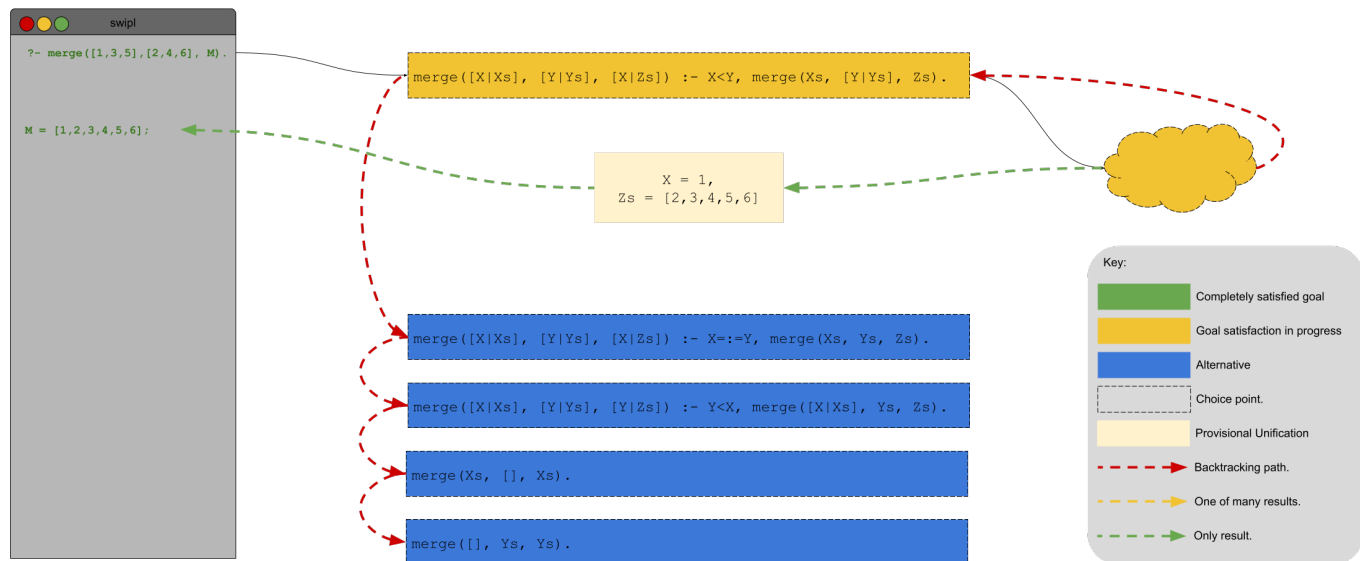
```
merge([1, 3, 5], [2,4,6], M).
```

As we know, Prolog will search top to bottom when looking for ways to unify and the first rule that Prolog sees is applicable:

```
merge([X|Xs], [Y|Ys], [X|Zs]) :- X<Y, merge(Xs, [Y|Ys], Zs).
```

Why? Because Prolog sees **X** as 1 and **Y** as 2 and $1 < 2$. Therefore, Prolog will complete its unification for this query by replacing it with another query:

```
merge([3,5], [2,4,6], Zs).
```



Once Prolog has completed that query, the response comes back:


```
M = [1,2,3,4,5,6]
```

Unfortunately, that's not the whole story. While Prolog is off attempting to satisfy the *subquery* `merge([3,5], [2,4,6], Zs).`, it believes that there are still several other viable alternatives for satisfying our original query:

```
merge([X|Xs], [Y|Ys], [X|Zs]) :- X:=Y, merge(Xs, Ys, Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :- Y<X, merge([X|Xs], Ys, Zs).
merge(Xs, [], Xs).
merge([], Ys, Ys).
```

The result is that Prolog will have to use memory to remember those alternatives. As the lists that we ask Prolog to merge get longer and longer, that memory will have an impact on the system's performance. However, we know that those other alternatives will never match and keeping them around is a waste. How do we know that? Well, if $X < Y$ then it cannot be equal to Y and it certainly cannot be greater than Y . Moreover, the lists in the first two arguments cannot be empty. Overall, each of the possible rules for *merge/3* are mutually exclusive. You can only choose one.

If there were a way to tell Prolog that those other possibilities are impossible after it encounters a matching rule that would save Prolog from having to keep them around. The good news is that there is!

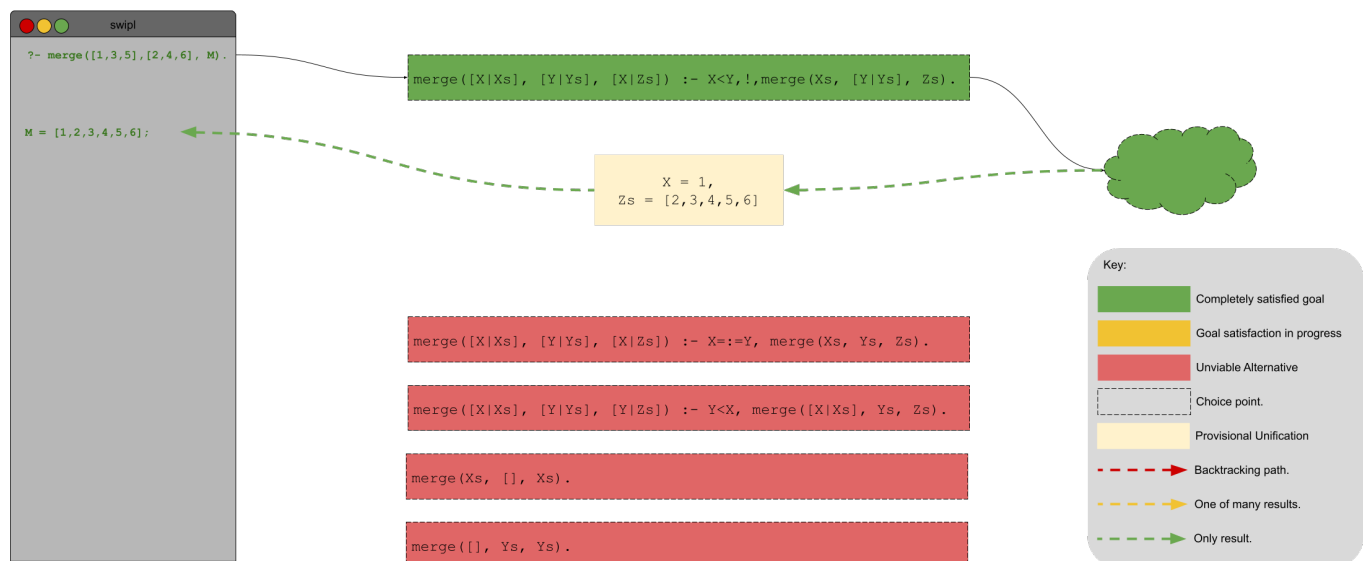
We can use the **cut**  <https://www.swi-prolog.org/pldoc/man?predicate=!/0> operator to tell Prolog that once it has "descended" down a particular path, there is no sense backtracking beyond that point to look for alternate solutions. The technical definition of a cut is

Discard all choice points created since entering the predicate in which the cut appears. In other words, commit to the clause in which the cut appears and discard choice points that have been created by goals to the left of the cut in the current clause.

Let's rewrite our *merge/3* predicate to take advantage of cuts and save some overhead:

```
merge([X|Xs], [Y|Ys], [X|Zs]) :- X<Y, !, merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [X|Zs]) :- X>=Y, !, merge(Xs, Ys, Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :- Y<X, !, merge([X|Xs], Ys, Zs).
merge(Xs, [], Xs) :- !.
merge([], Ys, Ys) :- !.
```

Returning to the definition of cut, in the first rule we are telling Prolog (through our use of the cut) to disregard all choice points created to the left of the **!**. In particular, we are telling Prolog to forget about the choice it made that **X < Y**. The result is that Prolog is no longer under the impression that there are other rules that are applicable. Visually, the situation resembles



Dr. Cutyll and Mr. Unify

Cuts are not always so beneficial. In fact, their use in Prolog is somewhat controversial. A cut necessarily limits Prolog's ability to backtrack. If the Prolog programmer uses a cut in a rule that is meant to be used *declaratively* (in order to generate values) and *procedurally*, then the cut may

change the results.

There are two types of cuts. A *green cut* is a cut that does not change the meaning of a predicate. The cut that we added in the merge/3 predicate above is a green cut. A *red cut* is, technically speaking, a cut that is not a green cut. I know that's a satisfying definition. Sorry. The implication, however, is that a *red cut* is a cut that changes the meaning of predicate.

Red Alert

In the previous section, we discussed two different types of cuts the red and the green. A green cut is one that does not alter the behavior of a Prolog program. A red cut does alter the behavior of a Prolog program. The implication was that red cuts are bad and green cuts are good. But, is this always the case?

To frame our discussion, let's look at a Prolog program that performs a Bubble Sort on a list of numbers. The predicate, *bsort/2*, is defined like this:

```
bsort(Unsorted, Sorted):-  
    append(Left, [A, B | Right], Unsorted),  
    B<A,  
    append(Left, [B, A | Right], MoreSorted),  
    bsort(MoreSorted, Sorted).  
bsort(Sorted, Sorted).
```

The first rule handles the recursive case (when there is at least one list item out of order) and the second rule handles the base case (when the list is sorted). According to this definition, how does Prolog handle the query

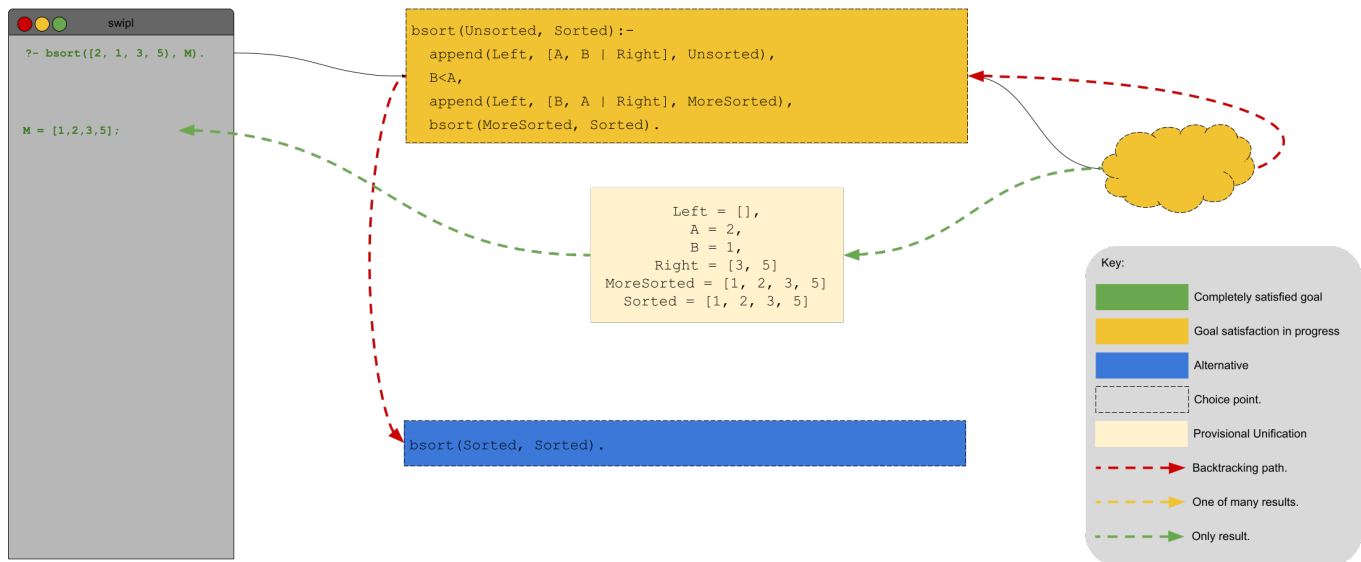
```
bsort([2,1,3,5], M).
```

Prolog (as we are well aware) searches its rule top to bottom. The first rule that it sees is the recursive case and Prolog immediately opts for it. Opting for this case is not without consequences -- a choice point is created. Why? Because Prolog made the choice to attempt to satisfy the query according to the first rule and not the (just as applicable) second rule! Let's call this choice point A.

Next, Prolog attempts to unify the subquery `append(Left, [A, B | Right], Unsorted)`. The first (of many) unifications that *append/3* generates is `Left = [], A = 2, B = 1, Right = [3, 5]`. By continuing with this particular unification, Prolog creates another choice point. For what reason? Well, Prolog knows that *append/3* could generate other potential unifications and it will need to check those to see if they, too, satisfy the original query! Let's call this choice point B. Next, Prolog checks whether `A` is less than `B` -- it is. Prolog continues in a like manner to satisfy the final two subgoals of the rule.

When Prolog does satisfy those, it has generated a sorted version of `[2,1,3,5]`. It reports that

result to the querier. Great! There's only one problem. There are still choice points that Prolog wants to explore. In particular, there are choice points A and B. In this case, we can forget about choice point B because there are no other unifications of *append/3* that meet the criteria for *A < B* in *Unsorted*. In other words, visually the situation looks like this:



If the querier is using *bsort/2* declaratively, this scenario is a problem: Prolog will backtrack to choice point A and then consider the base case rule for *bsort/2*. In other words, Prolog will also generate

```
[2,1,3,5]
```

as a unification! This is clearly *not right*. What's more, the Prolog programmer could query for

```
bsort([3,2,1,4,5], Sorted).
```

in which choice point B will have consequences. Run this query for yourself (using trace), examine the results, and make sure you understand why the results are generated.

So, what are we to do? **cut/0** <https://www.swi-prolog.org/pldoc/man?predicate=!/0> to the rescue! Logically speaking, once our *bsort/3* rule finds a single element out of order and we adjust that element, the recursive call to itself will handle ordering any other unordered elements. In other words, we can forget any earlier choice points! This is exactly what the cut is intended for!

Let's rewrite *bsort/3* using cuts and see if our problem is sorted (see what I did there?):

```
bsort(Unsorted, Sorted):-
    append(Left, [A, B | Right], Unsorted),
    B < A, !,
    append(Left, [B, A | Right], MoreSorted),
    bsort(MoreSorted, Sorted).
bsort(Sorted, Sorted).
```


And now let's see how our earlier troublesome queries perform:

```
?- bsort([2,1,3,5], Sorted).  
Sorted = [1, 2, 3, 5].  
  
?- bsort([3,2,1,4,5], Sorted).  
Sorted = [1, 2, 3, 4, 5].
```

Amazing!

Here's the rub: The version of our Prolog program with the cut gave different results than the version without. Is this cut a green or a red cut? That's right, it's a red cut. I guess there are such things as good red cuts after all!

The Fundamentals

As we have said in the past, there is a theoretical underpinning for each of the programming paradigms that we have studied this semester. Logic programming is no different. The theory behind logic programming is *first-order predicate logic*. Predicate logic is an extension of *propositional logic*. Propositional logic is based on the evaluation of propositions.

A *proposition* is simply any statement that can be true or false. For example,

- Will is a programmer.
- Alicia is a programmer.
- Sam is a programmer.

Those statements can be true or false. In propositional logic we can build more complicated propositions from existing propositions using logical connectives like *and*, *or*, *not*, and *implication* (*if ... then*). Each of these has an associated truth table to determine the truth of two combined propositions.

Look closely at the example propositions above and you will notice an underlying theme: they all do with someone (let's call them x) being a programmer. In propositional logic, our ability to reason using that underlying theme is impossible. We can only work with the truth of the statement as a whole.

If we add to propositional logic *variables*, *constants*, and *quantifiers* then we get predicate logic and we are able to reason more subtly. Although you might argue that propositional logic has variables, everyone can agree that they are limited -- they can only have two values, true and false. In first-order predicate logic, variables can have domains other than just {T, F}. That's already a huge step!

Quantifiers "define" variables and "constrain" their possible values. There are two quantifiers in first-order predicate logic -- the *universal* and the *existential*. The universal is the "for all" (aka

"every") quantifier. We can use the universal quantifier to write statements (in logic) like "Every person is a Bearcats fan." Symbolically, we write the universal quantifier with the \forall . We can write our obvious statement from above in logic like

$$\forall x(\text{person}(x) \implies \text{fan}(x, \text{bearcats}))$$

Now we are talking! As for the existential quantifier, it allows us to write statements (in logic) like "There is some person on earth that speaks Russian." Symbolically, we write the existential quantifier with the \exists . We can write the statement about our Russian-speaking person as

$$\exists x(\text{person}(x) \wedge \text{speaks}(x, \text{russian}))$$

How are quantifiers embedded in Prolog? Variables in Prolog queries are existentially qualified -- "Does there exist ...?" Variables in Prolog rules are universally quantified -- "For all"

In first-order predicate logic, there are such things as Boolean-valued functions. This is familiar territory for us programmers. A *Boolean-valued function* is a function that has 0 or more parameters and returns true or false.

With these definitions, we can now define *predicates*: A predicate is a proposition in which some Boolean variables are replaced by Boolean-valued functions and quantified expressions. Let's look at an example.

$$p \implies q$$

is a proposition where p and q are boolean variables. Replace p with the Boolean-valued function *is_teacher*(x) and q with the quantified expression $\exists y(\text{student}(x) \wedge \text{teaches}(x, y))$ and we have the predicate

$$\text{is_teacher}(x) \implies \exists y(\text{student}(x) \wedge \text{teaches}(x, y))$$

There is only one remaining question: Why is it called *first-order* predicate logic and not, say, *higher-order* predicate logic? "First-order" here indicates that the predicates in this logic cannot manipulate or reason about predicates themselves. Does this sound familiar? Imperative languages can define functions but they cannot reason about functions themselves while functional languages can!