Prolog

How does that work?

How can Prolog solve a request?

```
prolog> append([], Ys, Ys).
prolog> append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
prolog> :d
#prolog> ?append([1], Y, [1,2,3]).
query List(append([1],Y,[1, 2, 3]))
trying append([],Ys1,Ys1) :- .
unify [1] with [] = None
unify append([1],Y,[1, 2, 3]) with append([],Ys1,Ys1) = None
```

Unification of 2 clauses :

- Attempt to find an assignation that makes the 2 clauses true.

How can Prolog solve a request?

```
trying append([X2|Xs3],Ys4,[X2|Zs5]) :- append(Xs3,Ys4,Zs5).
unify X2 with 1 = Some(List(X2 = 1))
unify 1 with X2 = Some(List(X2 = 1))
unify Xs3 with [] = Some(List(Xs3 = [], X2 = 1))
unify [] with Xs3 = Some(List(Xs3 = [], X2 = 1))
unify [1] with [X2|Xs3] = Some(List(Xs3 = [], X2 = 1))
unify Y with Ys4 = Some(List(Y = Ys4, Xs3 = [], X2 = 1))
unify 0 with 0 = Some(List(Y = Ys4, Xs3 = [], X2 = 1))
unify 1 with 1 = Some(List(Y = Ys4, Xs3 = [], X2 = 1))
unify X2 with 1 = Some(List(Y = Ys4, Xs3 = [], X2 = 1))
unify 1 with X2 = Some(List(Y = Ys4, Xs3 = [], X2 = 1))
unify Zs5 with [2, 3] = Some(List(Zs5 = [2, 3], Y = Ys4, Xs3 = [], X2 = 1))
unify [2, 3] with Zs5 = Some(List(Zs5 = [2, 3], Y = Ys4, Xs3 = [], X2 = 1))
unify [1, 2, 3] with [X2|Zs5] = Some(List(Zs5 = [2, 3], Y = Ys4, Xs3 = [],
X2 = 1))
unify append([1],Y,[1, 2, 3]) with append([X2|Xs3],Ys4,[X2|Zs5]) =
Some(List(Zs5 = [2, 3], Y = Ys4, Xs3 = [], X2 = 1))
```

How can Prolog solve a request?

```
trying append([],Ys6,Ys6) :- .
unify [] with [] = Some(List(Zs5 = [2, 3], Y = Ys4, Xs3 = [], X2 = 1))
unify Xs3 with [] = Some(List(Zs5 = [2, 3], Y = Ys4, Xs3 = [], X2 = 1))
unify Ys4 with Ys6 = Some(List(Ys4 = Ys6, Zs5 = [2, 3], Y = Ys4, Xs3 = [],
X2 = 1))
unify Ys6 with [2, 3] = Some(List(Ys6 = [2, 3], Ys4 = Ys6, Zs5 = [2, 3], Y
= Ys4, Xs3 = [], X2 = 1))
unify [2, 3] with Ys6 = Some(List(Ys6 = [2, 3], Ys4 = Ys6, Zs5 = [2, 3], Y
= Ys4, Xs3 = [], X2 = 1))
unify Zs5 with Ys6 = Some(List(Ys6 = [2, 3], Ys4 = Ys6, Zs5 = [2, 3], Y =
Ys4, Xs3 = [], X2 = 1))
```

How can Prolog solve a request?

```
solved List() = Stream(List(Ys6 = [2, 3], Ys4 = Ys6, Zs5 = [2, 3], Y = Ys4,
Xs3 = [], X2 = 1), ?)
solved List() = Stream(List(Ys6 = [2, 3], Ys4 = Ys6, Zs5 = [2, 3], Y = Ys4,
Xs3 = [], X2 = 1), ?)
solved List(append(Xs3,Ys4,Zs5)) = Stream(List(Ys6 = [2, 3], Ys4 = Ys6, Zs5
= [2, 3], Y = Ys4, Xs3 = [], X2 = 1), ?)
solved List() = Stream(List(Ys6 = [2, 3], Ys4 = Ys6, Zs5 = [2, 3], Y = Ys4,
Xs3 = [], X2 = 1), ?)
solved List(append([1],Y,[1, 2, 3])) = Stream(List(Ys6 = [2, 3], Ys4 = Ys6,
Zs5 = [2, 3], Y = Ys4, Xs3 = [], X2 = 1), ?)
List(Y = [2, 3])
```

Principle

Attempt to match the request with the predicates, in the order they were defined.

- Match means: if the left hand side of the predicate can be unified with the request, then try now to recursively match the right hand side of the unified predicate.
- Recursively find new matches, till either
  - One of the match is a fact (empty right hand side) => success.
  - No match can be found => failure.

Backtracking

- When attempting to find a match for the right hand side of a predicate, the interpreter might fail.
  - It needs still to try out the remaining predicates (in order) to see if one of them is a match.
- The fact of failing makes the process go back in the list of known unifications for variables.
  - This is called backtracking.

Shortcomings of Logic Programming

- The clauses determine the « intelligence » of the system.
- The order of the clauses determine the order of searching a solution and thus the performance of requests.
  - It makes it hard or impossible to have rules that are efficient for all useful requests.
- The only possible way to express the « intelligence » of the system is through clauses.
  - Not very well suited for lots of problems.