Backtracking

If we reach a point where a goal can't be matched, or the body of a rule can't be matched, we *backtrack* to the last (most recent) spot where a choice of matching a particular fact or rule was made. We then try to match a different fact or rule. If this can't be done, we go back to the next previous place where a choice was made and try a different match there. We try alternatives until we are able to solve all the goals in our query or until all possible choices have been tried and found to fail. If this happens, we answer "no" the query can't be solved.

As we try to match facts and rules we try them in their order of definition.

Example

Let's trace how

```
| ?- grandMotherOf(tom,GM).
```

is solved.

Recall that

```
grandMotherOf(X,GM) :-
  motherOf(X,M),
     motherOf(M,GM).

grandMotherOf(X,GM) :-
  fatherOf(X,F),
    motherOf(F,GM).
fatherOf(tom,dick).
fatherOf(dick,harry).
fatherOf(jane,harry).
motherOf(tom,judy).
motherOf(dick,mary).
motherOf(jane,mary).
```


We try the first `grandMotherOf` rule first.

This forces `X = tom`. We have to solve

```
 motherOf(tom,M),
      motherOf(M,GM).
```

We now try to solve

`motherOf(tom,M)`

This forces `M = judy`.

We then try to solve

`motherOf(judy,GM)`

None of the `motherOf` rules match this goal, so we backtrack. No other `motherOf` rule can solve

`motherOf(tom,M)`

so we backtrack again and try the second `grandMotherOf` rule:


```
grandMotherOf(X,GM) :-
  fatherOf(X,F),
    motherOf(F,GM).
```

This matches, forcing `X = tom`.

We have to solve

`fatherOf(tom,F), motherOf(F,GM).`

We can match the first goal with

`fatherOf(tom,dick).`

This forces `F = dick`.

We then must solve

`motherOf(dick,GM)`

which can be matched by

`motherOf(dick,mary).`

We have matched all our goals, so we know the query is true, with `GM = mary`.

List Processing in Prolog

Prolog has a notation similar to "cons cells" of Lisp and Scheme.

The "." functor (predicate name) acts like cons.

Hence `.(a,b)` in Prolog is essentially the same as `(a . b)` in Scheme.

Lists in Prolog are formed much the same way as in Scheme and ML:

`[]` is the empty list

`[1,2,3]` is an abbreviation for
`.(1, .(2, .(3,[])))`

just as

`(1,2,3)` in Scheme is an abbreviation for
`(cons 1 (cons 2 (cons 3 () )))`

The notation `[H|T]` represents a list with `H` matching the head of the list and `T` matching the rest of the list.

Thus `[1,2,3]` ≡ `[1| [2,3]]` ≡ `[1,2| [3]]` ≡ `[1,2,3| [] ]`

As in ML, "_" (underscore) can be used as a wildcard or "don't care" symbol in matches.

Given the fact

```
p([1,2,3,4]).
```

The query

```
| ?- p([X|Y]).
```

answers

```
X = 1,
Y = [2,3,4]
```


The query

```
p([_,_,X|Y]).
```

answers

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


List Operations in Prolog

List operations are defined using rules and facts. The definitions are similar to those used in Scheme or ML, but they are *non-procedural*.

That is you don't given an execution order. Instead, you give recursive rules and non-recursive "base cases" that characterize the operation you are defining.

Consider `append`:

```
append([],L,L).
append([H|T1],L2,[H,T3]) :-
   append(T1,L2,T3).
```

The first fact says that an empty list (argument 1) appended to any list `L` (argument 2) gives `L` (argument 3) as its answer.

The rule in line 2 says that if you take a list that begins with **H** and has **T1** as the rest of the list and append it to a list **L** then the resulting appended list will begin with **H**.

Moreover, the rest of the resulting list, **T3**, is the result of appending **T1** (the rest of the first list) with **L2** (the second input list).

The query

```
| ?- append([1],[2,3],[1,2,3]).
```

answers

```
Yes
```

because with **H=1**, **T1=[]**, **L2 =[2,3]** and **T3=[2,3]** it must be the case that **append([],[2,3],[2,3])** is true and fact (1) says that this is so.

Inverting Inputs and Outputs

In Prolog the division between "inputs" and "outputs" is intentionally vague. We can exploit this. It is often possible to "invert" a query and ask what *inputs* would compute a given output. Few other languages allow this level of flexibility.

Consider the query

```
append([1],X,[1,2,3]).
```

This asks Prolog to find a list **X** such that if we append **[1]** to **X** we will get **[1,2,3]**.

Prolog answers

```
X = [2,3]
```


How does it choose this answer?

First Prolog tries to match the query against fact (1) or rule (2).

Fact (1) doesn't match (the first arguments differ) so we match rule (2).

This gives us **H=1**, **T1=[]**, **L2=X** and **T3 = [2,3]**.

We next have to solve the body of rule (2) which is

```
append([],L2,[2,3]).
```

Fact (1) matches this, and tells us that **L2=[2,3]=X**, and that's our answer!

The Member Relation

A common predicate when manipulating lists is a membership test—is a given value a member of a list?

An "obvious" definition is a recursive one similar to what we might program in Scheme or ML:

```
member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).
```

This definition states that the first argument, **X**, is a member of the second argument (a list) if **X** matches the head of the list *or* if **X** is (recursively) a member of the rest of the list.

Note that we don't have to "tell" Prolog that **x** *can't* be a member of an empty list—if we don't tell Prolog that something is true, it automatically assumes that it must be false.

Thus saying nothing about membership in an empty list is the same as saying that membership in an empty list is impossible.

Since inputs and outputs in a relation are blurred, we can use **member** in an unexpected way—to iterate through a list of values.

If we want to know if any member of a list **L** satisfies a predicate **p**, we can simply write:

```
member(X,L),p(X).
```

There is no explicit iteration or searching. We simply ask Prolog to find an **x** such that **member(X,L)** is true (**x** is in **L**) *and* **p(X)** is true. Backtracking will find the "right" value for **x** (if any such **x** exists).

This is sometimes called the "guess and verify" technique.

Thus we can query

```
member(X,[3,-3,0,10,-10]),
  (X > 0).
```

This asks for an **x** in the list **[3,-3,0,10,-10]** which is greater than **0**.

Prolog answers


```
X = 3 ;
X = 10 ;
```

Note too that our "obvious" definition of member is not the only one possible.

An alternative definition (which is far less obvious) is

```
member(X,L) :-
  append(_,[X|_],L).
```

This definition says **x** is a member of **L** if I can take some list (whose value I don't care about) and append it to a list that begins with **x** (and which ends with values I don't care about) and get a list equal to **L**.

Said more clearly, **x** is a member of **L** if **x** is anywhere in the "middle" of **L**.

Prolog solves a query involving **member** by partitioning the list **L** in *all possible ways*, and checking to see if **x** ever is the head of the second list. Thus for **member(X,[1,2,3])**, it tries the partition **[]** and **[1,2,3]** (exposing **1** as a possible **x**), then **[1]** and **[2,3]** (exposing **2**) and finally **[1,2]** and **[3]** (exposing **3**).

Sorting Algorithms

Sorting algorithms are good examples of Prolog's definitional capabilities. In a Prolog definition the "logic" of a sorting algorithm is apparent, stripped of the cumbersome details of data structures and control structures that dominate algorithms in other programming languages.

Consider the simplest possible sort imaginable, which we'll call the "naive sort."

At the simplest level a sorting of a list **L** requires just two things:

- The sorting is a permutation (a reordering) of the values in **L**.
- The values are "in order" (ascending or descending).

We can implement this concept of a sort directly in Prolog. We
(a) permute an input list
(b) check if it is in sorted order
(c) repeat (a) & (b) until a sorting is found.

Permutations

Let's first look at how permutations are defined in Prolog. In most languages generating permutations is non-trivial—you need data structures to store the permutations you are generating and control structures to visit all permutations in some order.

In Prolog, permutations are defined quite concisely, though with a bit of subtlety:

perm(X,Y) will be true if list **Y** is a permutation of list **X**.

Only two definitions are needed:

```
perm([],[]).
perm(L,[H|T]) :-
 append(V,[H|U],L),
 append(V,U,W), perm(W,T).
```


The first definition,

```
perm([],[]).
```

is trivial. An empty list may only be permuted into another empty list.

The second definition is rather more complex:

```
perm(L,[H|T]) :-
append(V,[H|U],L),
 append(V,U,W), perm(W,T).
```

This rule says a list **L** may be permuted in to a list that begins with **H** and ends with list **T** if:

(1) **L** may be partitioned into two lists, **V** and **[H|U]**. (That is, **H** is somewhere in the "middle" of **L**).

(2) Lists **V** and **U** (all of **L** except **H**) may be appended into list **W**.

(3) List **W** may be permuted into **T**.

Let's see **perm** in action:

```
| ?- perm([1,2,3],X).
X = [1,2,3] ;
X = [1,3,2] ;
X = [2,1,3] ;
X = [2,3,1] ;
X = [3,1,2] ;
X = [3,2,1] ;
no
```

We'll trace how the first few answers are computed. Note though that *all* permutations are generated, and with no apparent data structures or control structures.

We start with **L=[1,2,3]** and **X=[H|T]**.

We first solve **append(V, [H|U],L)**, which simplifies to **append(V, [H|U], [1,2,3])**.

One solution to this goal is

V = [], H = 1, U = [2,3]

We next solve **append(V,U,W)** which simplifies to **append([], [2,3],W)**.

The only solution for this is **W=[2,3]**.

Finally, we solve **perm(W,T)**, which simplifies to **perm([2,3],T)**.

One solution to this is **T=[2,3]**.

This gives us our first solution: **[H|T]=[1,2,3]**.

To get our next solution we *backtrack*. Where is the most recent place we made a choice of how to solve a goal?

It was at **perm([2,3],T)**. We chose **T=[2,3]**, but **T=[3,2]** is another solution. Using this solution, we get out next answer **[H|T]=[1,3,2]**.

Let's try one more. We backtrack again. No more solutions are possible for **perm([2,3],T)**, so we backtrack to an earlier choice point.

At **append(V, [H|U], [1,2,3])** another solution is

V=[1], H = 2, U = [3]

Using this binding, we solve **append(V,U,W)** which simplifies to **append([1], [3],W)**. The solution to this must be **W=[1,3]**.

We then solve **perm(W,T)** which simplifies to **perm([1,3],T)**. One solution to this is **T=[1,3]**. This makes our third solution for **[H|T] = [2,1,3]**.

You can check out the other bindings that lead to the last three solutions.

A Permutation Sort

Now that we know how to generate permutations, the definition of a permutation sort is almost trivial.

We define an **inOrder** relation that characterizes our notion of when a list is properly sorted:

```
inOrder([]).
inOrder([_]).
inOrder([A,B|T]) :-
   A =< B, inOrder([B|T]).
```

These definitions state that a null list, and a list with only one element are always in sorted order. Longer lists are in order if the first two elements are in proper order. **(A=<B)** checks this and then the rest of the list, excluding the first element, is checked.

Now our naive permutation sort is only one line long:

```
naiveSort(L1,L2) :-
  perm(L1,L2), inOrder(L2).
```

And the definition works too!

```
| ?-
naiveSort([1,2,3],[3,2,1]).
no
?- naiveSort([3,2,1],L).
L = [1,2,3] ;
no
| ?-
naiveSort([7,3,88,2,1,6,77,
 -23,5],L).
L = [-23,1,2,3,5,6,7,77,88]
```


Though this sort works, it is hopelessly inefficient—it repeatedly "shuffles" the input until it happens to find an ordering that is sorted. The process is largely undirected. We don't "aim" toward a correct ordering, but just search until we get lucky.