Prolog by Example

- A Prolog program is composed of predicates defined by facts and rules, which are special cases of definite (or Horn) clauses, i.e. clauses with at most one positive literal (the head of the clause).

- A predicate may have several alternative definitions (both facts and rules).

- A fact has no negative literals and expresses positive knowledge that is definite (no disjunctions). Example: The knowledge that John is a child of Ann and Alex, and that Ann is a child of Bob, is expressed by two facts, namely

  \[
  \text{child_of(john, ann).} \\
  \text{child_of(john, alex).} \\
  \text{child_of(ann, bob).}
  \]

- A rule has one or more negative literals (the body of the clause), and is used to infer predicates from other predicates. Example: A grand child is the child of the child, i.e.

  \[
  \text{grand_child_of(X,Y) :-} \\
  \text{child_of(X,Z),} \\
  \text{child_of(Z,Y).}
  \]

- The name rule is due to the fact that a definite clause, \( \text{H} \leftarrow \neg B_1 \lor \neg B_2 \ldots \lor \neg B_k \) can be written in the form of an if expression \( \text{H} \leftarrow B_1 \land B_2 \land \ldots \land B_k \), which is the syntax adopted in Prolog.
Prolog by Example

child_of(john, ann).
child_of(john, alex).
child_of(ann, bob).


- All variables (starting with capital letters) are universally quantified. The rule can be read “For any X, Y and Z, if X is a child of Z and Z is a child of Y, then X is a grand child of Y”.

- Rules can be rephrased such that variables appearing only in the body of a rule are existentially quantified. In this case, “X is a grand child of Y, if there is some Z, of which X is a child, and is a child of Y”.

- A program may be regarded as a knowledge base, combining explicit knowledge of facts as well as implicit knowledge by means of rules. This knowledge may be queried by negative rules (no head). Notice that due to the negation, variables are read existentially.

- Examples:
  - Are there any parents of Alex (any X of which John is a parent)?
    ?- child_of(X,alex).
  - Are there any A and B such that A is a grand parent of B (or B is a grand child of A)?
    ?- grand_child_of(A,B).
Prolog by Example

child_of(john, ann).
child_of(john, alex).
child_of(bob, john).

- A Prolog engine is able to answer the queries (by implementing resolution on Horn clauses), finding all solutions and presenting them, one at a time.

?- child_of(X, alex).
   X = john; no.

?- grand_child_of(A, B).
   A = bob, B = ann ;
   A = bob, B = alex; no.

- In practice, to program in Prolog, it is important to understand the options that Prolog engines assume, in applying resolution.

- Informally, to search for all solutions, Prolog uses the query as an initial resolvent, and organises all the other resolvents in an execution tree.

- Solutions, if any, are the success leafs of the tree (the other leafs are failed).
child_of(john, ann).
child_of(john, alex).
child_of(bob, john).

- For example, query ?- grand_child_of(A,B), expands into the execution tree

- Notice that
  - Instances of the answers are obtained by composing the substitutions in a path to a successful leaf
  - The tree is searched depth-first, left-to-right.
- Let us assume some elements from the larger family tree.

- The basic relationship that must be recorded is parent-child (the one kept in the ID cards and databases), modeled by the binary predicate `child_of/2`.

```
child_of(john,doug).
child_of(john,carol).
child_of(mary,carol).
child_of(tess,mary).
child_of(fred,tess).
child_of(peter,john).
child_of(bob,john).
child_of(alex,john).
child_of(peter,ann).
child_of(bob,ann).
child_of(alex,ann).
child_of(alice, peter).
child_of(alice,vicky).
```
Other useful information can be introduced, such as gender (although usually implicit in the name – but still this info is explicit in ID cards and passports). This is expressed by the unary predicate `is_male/1` and `is_female/1`.

- \( \text{is\_male}(\text{doug}). \) 
- \( \text{is\_male}(\text{john}). \) 
- \( \text{is\_male}(\text{fred}). \) 
- \( \text{is\_male}(\text{peter}). \) 
- \( \text{is\_male}(\text{bob}). \) 
- \( \text{is\_male}(\text{alex}). \) 
- \( \text{is\_female}(\text{mary}). \) 
- \( \text{is\_female}(\text{carol}). \) 
- \( \text{is\_female}(\text{tess}). \) 
- \( \text{is\_female}(\text{ann}). \) 
- \( \text{is\_female}(\text{alice}). \) 
- \( \text{is\_female}(\text{vicky}). \)

- Once the basic relationships are established among the elements of the family, other family relationships and predicates. For example, the relationship `parent_of` is the opposite of `child_of`:

\[
\text{parent\_of}(X, Y) :- \text{child\_of}(Y, X).
\]

- The predicate `mother` is assigned to any female that is the parent of someone.

\[
\text{mother\_of}(X, Y) :- \text{parent\_of}(X, Y), \text{is\_female}(X).
\]

- A mother is the mother of someone.

\[
\text{is\_mother}(X) :- \text{mother\_of}(X, \_).
\]
- For any mother with many children, the predicate `is_mother/1` may be deduced several times, once for each child. This leads to repeated answers to query

```
?- is_mother(X).
X = carol ;
X = carol ;
X = mary ;
X = tess ;
X = ann ;
X = ann ;
X = ann ;
X = vicky ;
no.
```

```
is_mother(X)  
father_of(X,Y), is_female(X).
child_of(Y,X), is_female(X).

is_female(doug)  
X/john
Y/doug

is_female(carol)  
X/john
Y/carol
```
- Other indirect relationships may be derived from their definition, namely
  - `grand_child_of/2`
  - `sibling_of/2, brother_of/2, sister_of/2`
  - `uncle_of/2, aunt_of/2,`
  - `great_uncle_of, great_aunt_of/2,`
  - `cousin_of, second_cousin_of/2, third_cousin_of/2, ...`

- Although most definitions are direct, some care must be taken. For example, siblings are children of a common parent

  ```prolog
  sibling_of(X,Y):-
  child_of(X,Z),
  child_of(Y,Z).
  ```

- This definition suffers from a quite common mistake: to assume that different variables should stand for different values. In the definition above a person is a sibling of him/herself!

- Here is the correct definition, where the difference (\( \neq \)) is made explicitly:

  ```prolog
  sibling_of(X,Y):-
  child_of(X,Z),
  child_of(Y,Z).
  X \neq Y.
  ```
child_of(john,doug).
child_of(john,carol).
child_of(mary,carol).
child_of(tess,mary).
child_of(fred,tess).
child_of(peter,john).
child_of(bob,john).
child_of(alex,john).
child_of(peter,ann).
child_of(bob,ann).
child_of(alex,ann).
child_of(alice,peter).
child_of(alice,vicky).

sibling_of(X,Y):-
    child_of(X,Z),
    child_of(Y,Z),
    X \= Y.

cd

child_of(john,doug), john \= Y

Y/john

john \= john

\x

\x

\x

\x

\x
Care must also be taken to avoid definitions that, correct as they might be, eventually become circular, leading to non termination.

For example, the symmetric property of the sibling_of relationship could, in principle, be explicitly specified, by the additional rule

\[
\text{sibling}_{of}(X,Y):= \\
\text{sibling}_{of}(Y,X).
\]

But this definition would lead, when \(X\) and \(Y\) are non-ground, to an infinite tree:

\[
\text{sibling}_{of}(X,Y):=
\text{child}_{of}(X,Z), \\
\text{child}_{of}(Y,Z), \\
X \neq Y.
\]

\[
\text{sibling}_{of}(X,Y):=
\text{sibling}_{of}(Y,X).
\]

- For example, the simetric property of the sibling_of relationship could, in principle, be explicitly specified, by the additional rule

\[
\text{sibling}_{of}(X,Y):=
\text{child}_{of}(X,Z), \\
\text{child}_{of}(Y,Z), \\
X \neq Y.
\]

\[
\text{sibling}_{of}(Y,X):
\text{sibling}_{of}(X,Y).
\]

- But this definition would lead, when \(X\) and \(Y\) are non-ground, to an infinite tree:
Logic has been used to formalise arithmetics, and a number of interesting (mostly theoretical) results where obtained from this effort (for example, the famous Gödel incompleteness theorem).

The “standard” formalisation is due to Peano (end of 19th century), that proposed a simple set of axioms, for this purpose.

To begin with, integers are defined inductively: an integer is either 0 or the successor of another integer.

The basic algebraic operations can also be formalised, again inductively. For example the addition of two integers:

- Base clause: The sum of 0 and some integer M, is that integer.
- Inductive clause: The sum of the successor of an integer M and another integer N is the successor of the sum of the integers M and N.

Similar definitions can be used to define the operations of multiplication and potentiation.

Logic programming can thus be used to directly implement arithmetics (although in a very inefficient way).
The definition of integers is straightforward. Below the predicate int/1 is used to assign the property integer, and the functor s/1 to denote the successor of an integer.

\[
\text{int}(0). \quad \% 0 \text{ is an integer} \\
\text{int}(\text{s}(M)) :- \text{int}(M). \quad \% \text{the successor of an integer is integer.}
\]

- The addition follows directly the definition.

\[
\text{sum}(0, M, M). \quad \% \text{The sum of 0 and any integer } M \text{ is } M \\
\text{sum}(\text{s}(N), M, \text{s}(K)) :- \text{sum}(N, M, K), \quad \% \text{The sum the successor of } N \text{ and } M \text{ is the successor of the sum of } N \text{ and } M.
\]

- The product, as a repeated sum, is modelled by means of addition.

\[
\text{prod}(0, M, 0). \quad \% \text{The product of 0 with any integer } M \text{ is 0.} \\
\text{prod}(\text{s}(N), M, P) :- \text{prod}(N, M, K), \quad \% \text{The product of the successor of } N \text{ and } M \text{ is the sum of } M \text{ with the product of } M \text{ and } N. \\
\text{sum}(K, M, P). \quad \% \text{i.e. } (N+1)*M = N*M + M
\]

- And so is the power, as a repeated multiplication (the exponent is the first argument).

\[
\text{pow}(0, 0, \text{s}(0)). \quad \% 0 \text{ raised to } 0 \text{ is 0.} \\
\text{pow}(0, \text{s}(_), \text{s}(0)). \quad \% \text{Integer } M (>0) \text{ raised to } 0 \text{ is 1 (i.e. } S(0)). \\
\text{pow}(\text{s}(N), M, P) :- \text{pow}(N, M, K), \quad \% \text{Integer } M \text{ raised to the successor of } N \\
\text{prod}(K, M, P). \quad \% \text{i.e. } M^{(N+1)} = M^N * M
\]
- It is interesting to note the functioning of Prolog, with these predicates.

- First, the predicate integer can be used either to check whether an argument is an integer, either to generate integers.

| ?- int(s(s(0))).
| yes
| ?- int(X).
  | X = 0 ? ;   % X=0
  | X = s(0) ? ; % X=1
  | X = s(s(0)) ? ; % X=2
  | X = s(s(s(0))) ? % X=3
| ?- int(s(s(X))).
  | X = 0 ? ;    %
  | X = s(0) ? ;

- Notice in the last case, that the arguments of predicate int that are reported in the answers are usually denoted by 2, 3, ...
- The predicate sum performs as expected. Given two numbers to be summed, the predicate returns their sum. Let us sum 3 with 2

\[
\text{?- sum}(s(s(0)),s(s(s(0))),X).
\]

\[
X = s(s(s(s(0)))) ? \; \text{no}
\]

- The corresponding execution tree can be displayed

The result X is then obtained by composition of substitutions

\[
X / s(K1)
\]

but \( K1 / s(K2) \)

so \( X / s(K2) \)

but \( K2 / s(s(0)) \)

so \( X / s(s(s(s(0)))) \)

i.e. \( X = 5 \)
Logic programming is flexible. Predicate sum also defines subtraction! Given two numbers to be subtracted, the predicate also returns their difference. Let us subtract 2 from 5.

```
?- sum(s(s(0)), s(s(s(0))), X).
X = s(s(s(s(s(0))))) ; no
```

```
sum(0, M, M).
sum(s(N), M, s(K)):-
    sum(N, M, K).
```

```
sum(s(s(0)), X, s(s(s(s(s(0))))))
```

```
sum(s(0), X, s(s(s(0))))
```

```
sum(0, X, s(s(s(s(s(0))))))
```

```
X / s(s(s(0)))))
```

Diagram showing the process of subtracting 2 from 5 using the sum predicate.
The flexibility of logic programming allows that still the same definition enables to find all operands whose sum is 5.

| ?- sum(Y,S, sum(s(s((0))))).
Y = s(s(s(s(0))))) , X = 0?
Y = s(s(s(s(0))))) , X = s(0)?
Y = s(s(s(0))) , X = s(s(0))?; Y = s(s(0)) , X = s(s(s(0)))?; Y = s(0) , X = s(s(s(s(0)))))?; Y = 0 , X = s(s(s(s(s(0)))))?;
no

sum(0,M,M).
sum(s(N),M,s(K)):=
sum(N,M,K).
The other predicates, for product and potentiation, behave similarly to the predicate for sum. However, if no solutions exist there might be termination problems, if the recursion is performed in the “wrong” argument. For example, there are no problems with \(2 \times X = 3\).

```prolog
prod(0,M,0).
prod(s(N),M,P):-
    prod(N,M,K),
    sum(K,M,P).
sum(0,M,M).
sum(s(N),M,s(K)):-
    sum(N,M,K).

// Example statements
prod(s(s(s(0))),X,s(s(s(0)))).
sum(0,X,K1), sum(X,K1,s(s(0))).
prod(0,X,K2), sum(X,K2,K1), sum(X,K1,s(s(0))).
sum(0,X,K1), sum(X,K1,s(s(0))).
```
- However, the same is not true for “\( X \times 2 = 3 \)”, where the unknown argument is the one used in the recursion.

- It is left as an exercise to check what types of calls do not terminate with the above definitions.
Although important from a theoretical viewpoint, Peano arithmetics is not very useful in practice due to the complexity of both the representation of numbers and the basic operations.

For example, the usual representation of an integer $N$ in the usual base 2 or 10 requires $O(\lg_2 N)$ or $O(\lg_{10} N)$ bits. In contrast, Peano representation requires $O(N)$ symbols (more precisely $N$ s´es + $2N$ parenthesis + 1 zero, which is significantly larger.

As to the sum operation, say $N$ plus $N$, the usual method is proportional to the number of bits / digits i.e. it has complexity $O(\lg N)$.

In contrast, peano sum requires $N$ calls to the sum definition, i.e. It has complexity $O(N)$

```
sum(s(s(s(0))),X,P)
sum(s(s(0)),X,P)
sum(s(0),X,P)
sum(0,X,P)
```

```prolog
sum(M,0,M).
sum(s(N),M,s(K)):- sum(N,M,K).
```
- If Peano arithmetics is not very useful in practice, the same kind of definitions are widely applicable, namely in structures like lists.

- Assuming the Peano representation of integers, the following definition of predicate p_length/2 highlights the correspondence between 0 and the empty list as well as between the s and the list functors

```
p_length([],0).
p_length([_|T],s(N)):-
    p_length(T,N).
```

- Of course, if the usual arithmetics is used, the definition has to be adapted, such that the notion of successor is replaced by the usual “+1” operation

```
a_length([],0).
a_length([_|T], M):-
    a_length(T,N),
    M is N+1.
```

- There are however disadvantages in this formulation, namely non-termination. This is illustrated by the execution trees of both predicates to obtain lists of size 2.
There are however disadvantages in this formulation, namely non-termination. This is illustrated by the execution trees of both predicates to obtain lists of size 2.

- In the peano modelling,
  - the goal is of course \( ?- \text{p_length}(L, s(s(0))) \).
  - and the execution tree is as follows

```
P_length([],0).
P_length([T],s(N)):-
  p_length(T,s(0)).
```

Composing substitutions, the value of \( L \) is obtained

- \( T2 = [] \)
- \( T1 = [\_|T2] = [\_] \)
- \( L = [\_|T1] = [\_, \_] \)
- With the modelling with standard arithmeticals, the corresponding tree, for query
  \( \text{?- a_length(L,2)} \), is more complex

- The correct answer is obtained as before, in the left part of the tree

\[
\begin{align*}
a_{\text{length}}([],0).
a_{\text{length}}([\_|T],M):-
a_{\text{length}}(T,N),
M \text{ is } N+1.
\end{align*}
\]

\[
\begin{array}{rcl}
| & | & |
\hline
L/ [\_|T1], M/2 & a_{\text{length}}(L,2) & |
\hline
| & | & |
\hline
a_{\text{length}}(T1,N1), 2 \text{ is } N1+1 & & |
\hline
T1/[] , N1/0 & 2 \text{ is } 0+1 & |
\hline
X & T2/[] , N2/0 & |
\hline
N1 \text{ is } 0+1, 2 \text{ is } N1+1 & & |
\hline
N1/1 & 2 \text{ is } 1+1 & |
\hline
a_{\text{length}}(T2,N2), N1 \text{ is } N2+1, 2 \text{ is } N1+1 & & |
\hline
T1/[]|T2], M2/N1 & & |
\hline
a_{\text{length}}(T3,N3), N2 \text{ is } N3+1, ... & & |
\hline
T2/[]|T3], M3/N2 & & |
\hline
\end{array}
\]

- However, there is now a right part of the tree...
However, the right part of the execution tree is infinite!

```
a_length([],0).
a_length([_|T],M):-
    a_length(T,N),
    M is N+1.
```

```
N3 is 0+1,..., 2 is N1+1

a_length(T3,N3), N2 is N3+1, ...

T4/ [__T3], M3 / N2
```
A very common operation between lists is their concatenation. Predicate cat/3 below is true, when the list in the third argument is the concatenation of the lists in the first and second arguments.

As usual, the definition below relies on the inductive structure of the first list, and considers the cases where it is empty (the base clause) or not (the induction clause).

```prolog
cat([],L,L).
cat([H|T],L,[H|R]):-
cat(T,L,R).
```

It is interesting to notice in the execution tree, that concatenating a list with $N_1$ elements with a list with $N_2$ elements, requires

- $N_1$ calls to the recursive clause, for lists with sizes $N_1$, $N_1-1$, $N_1-2$, ..., 2, 1.
- $N_1$ trivial failed calls to the base clause, for these lists (in fact, implementations of prolog indexing on the first argument do not make these calls).
- 1 successful call with an empty list as first argument.

The concatenation of a list requires thus $N_1+1$ (non-trivial) calls, and presents thus a complexity of $O(N)$, where $N$ is the size of the first list argument.
- From an algorithmic view point, this complexity is the one that can be expected for lists, maintained as simply linked lists, with a pointer to its head (first element).

- This is in fact the case with Prolog, where lists are maintained in this form. However, a different form of representation, difference lists, allow the tail append to be a simple operation, as if lists were implemented with a pointer to their tail.

- In any case, the declarativity of this definition allows the use of this cat/3 predicate either
  - to concatenate two lists into a third one, or
  - to remove a sublist (prefix or postfix) from a list.

```prolog
?- cat([1,2],[3,4,5],L).
   L = [1,2,3,4,5]; no.
?- cat(L,[3,4,5],[1,2,3,4,5]).
   L = [1,2]; no.
?- cat([1,2],L,[1,2,3,4,5]).
   L = [3,4,5]; no.
?- cat(L1,L2,[1,2,3]).
   L1 = [], L2 = [1,2,3];
   L1 = [1], L2 = [2,3];
   L1 = [1,2], L2 = [3];
   L1 = [1,2,3], L2 = []; no.
```

```
cat([],L,L).
cat([H|T],L,[H|R]):-
   cat(T,L,R).
```
List Processing in Prolog: Sublists

- The declarativity of logic programming, makes it possible to reuse the definition of concatenation, to define other relations between a list and its (non-empty) sublists.

- **Prefix**: List P is a (non-empty) prefix of list L if there is some other list X such that L is the concatenation of P and X (in Prolog, X may be declared as an anonymous list).

  
  \[
  \text{prefix}([H|T],L):- \\
  \text{cat}([H|T],_,L).
  \]

- **Posfix**: List P is a (non-empty) postfix of list L if there is some other list X such that L is the concatenation of X and P (in Prolog, X may be declared as an anonymous list).

  
  \[
  \text{posfix_of}([H|T],L):- \\
  \text{cat}(_,[H|T],L).
  \]

- **Sublist**: The sublist relation can be defined upon the prefix and posfix relations: list S is a (non-empty) sublist of L if there is a prefix P of list L, of which S is a posfix.

  
  \[
  \text{sublist}(S,L):- \\
  \text{prefix}(P,L), \\
  \text{posfix}(S,P).
  \]
- In addition to relationships between lists, it is often important to consider relationships between lists and their members.

- The most useful predicate, `member/2`, relates a list with any of its members

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

- Another important predicate relates two lists, L1 and L2, with an element X. Given the declarativeness of logic programming this predicate can either be interpreted as
  - Inserting X into L1 to obtain L2
  - Removing X from L2 to obtain L1

- The definition is similar to that above, considering the cases where the insertion is done in the beginning or not of list L1.

```
insert(X,L,[X|L]).
insert(X,[H|T], [H|R]):-
    insert(X,T,R).
```
- Other interesting predicates relate a list with any of its permutations. As usual the definition considers two cases
  - The (only) permutation of an empty list is an empty list
  - A permutation of a non-empty list is a permutation of its tail, to which the head is inserted.

```prolog
perm([],[]).
perm([H|T],L):-
    perm(T,P),
    insert(H,P,L).
```

- A similar definition is used to relate a list with its reverse.
  - The (only) reversion of an empty list is an empty list
  - A reversion of a non-empty list is a reversion of its tail, to which the head is inserted at the end (this append at the end can be obtained by concatenation of the list with a list composed of the element to be appended)

```prolog
reverse([],[]).
reverse([H|T], L):-
    reverse(T, R),
    cat(R,[H], L).
```

- Notice the non termination problems when the first argument is a variable. Why?
List Processing in Prolog: Permutation and Reversion

- The reversion of a list is not very efficient. Analysing the execution tree (left as an exercise), the reversion of a list of size N requires
  - One (non-trivial) call to revert a list (of length N-1)
  - One call to “append” an element to the end of this list, which requires N “non-trivial calls” to predicate `cat/3`.

- Hence, the complexity of this “algorithm” measured by the number of non-trivial “calls”, is
  - 1 call to reverse and a call to `cat` for a list of length N (i.e. N+1 calls to `cat`)
  - 1 call to reverse and a call to `cat` for a list of length N-1 (i.e. (N-1)+1 calls to `cat`)
  - ...
  - 1 call to reverse and a call to `cat` for a list of length 1 (i.e. 1+1 calls to `cat`)
  - 0 call to reverse and 1 call to `cat` for a list of length 0 (i.e. 0+1 calls to `cat`)

- The number of calls is thus
  - N (non-trivial) calls to predicate `reverse`
  - \([N+1]+[(N-1)+1]+...+2+1=(N+1)(N+2)/2\) (non-trivial) calls to predicate `cat`

- The complexity of the algorithm is then \(O(N^2)\) which is quite inefficient, since it is quite easy to get an algorithm to perform reversion of a list in \(O(N)\) steps.
As seen before, concatenation of lists could be made more efficient if a pointer to the end of the list is maintained. This is the basic idea of the coding of lists as difference lists.

A difference list is a structure L-T where L and T are lists and T is the tail of L.

This structure represents a list whose elements are those belonging to L but not to T, (hence the name and the functor used). For example, the difference list \([1,2,3,4,5,6]-[5,6]\) encodes list \([1,2,3,4]\).

The interesting case for processing is the case where T is a variable. In this case, appending to the end of the list requires processing T.

For example, assume list \([1,2,3,4]\), encoded as \([1,2,3,4|T]-T\). Appending 5 to this list, corresponds to

- Obtaining a new tail \(T = [5|S]\)
- Subtracting L by S.

This is the declarative meaning of predicate \(td\_app/3\)

\[
\text{td_app}(X,L-T,L-S):= T = [X|S].
\]

which can be simplified with head unification

\[
\text{td_app}(X,L-[X|S],L-S).
\]
The same technique can be used to concatenate two difference lists, $L_1-T_1$ and $L_2-T_2$, into a third one $L_3-T_3$.

To exemplify, let us concatenate lists $[1,2,3]$ to $[4,5]$ resulting into $[1,2,3,4,5]$, all encoded as difference lists:

$$[1,2,3|T_1] - T_1 + [4,5|T_2]-T_2 = [1,2,3,4,5|T_3] – T_3$$

By analysing this example, it is clear that:

- We can make $L_1 = L_3$ if the substitution $T_1 = [4,5|T_3]$ is imposed.
- Since $L_2 = [4,5|T_2]$, the above substitution is imposed when $T_2 = T_3$ and $T_1 = L_2$.

Hence predicate `cat_diff/3`, describes the concatenation of difference lists:

```
cat_diff(L1-T1,L2-T2,L3-T3):-
    L3 = L1,
    T3 = T2,
    L2 = T1
```

By using head unification the concatenation becomes quite “intuitive”:

```
cat_diff(L1-T1,T1-T2,L1-T2).
```
- The concatenation of difference lists is thus performed through simple unification, instead of requiring the inefficient recursion of the usual lists.

- This result can thus be used to reverse difference lists, by adapting the previous definition of reverse/2 (for standard lists) to the new encoding as difference lists (the special encoding of the empty difference list is explained in the next slide)

```
rev_diff(L-T,L-T):- L == T.
rev_diff([H|L1]-T1, L2-T2):-
    rev_diff(L1-T1, Lx-Tx),
    cat_diff(Lx-Tx,[H|T]-T,L2-T2).
```

- The definition can be “improved” by head unification. Since the call to cat_diff/3 performs substitutions \( Lx = L2 \), \( Tx = [H|T] \) and \( T = T2 \), such substitutions can be made directly, avoiding the explicit call to cat_diff/3

```
rev_diff(L-T,L-T):- L == T.
rev_diff([H|L1]-T1, L2-T2):-
    rev_diff(L1-T1, T1-[H|T]).
```
In principle, the encoding of an empty difference list is L-L. As such the reversion of an empty difference list could be made with a clause such as

\[
\text{rev\_diff}(\text{L-L}, \text{L-L}) \iff \text{rev\_diff}(\text{L-T}, \text{L-T}) : - \text{L} = \text{T}
\]

Hence, when this clause is tested to reverse a non-empty list, say \(\text{rev\_diff}([1,2|Z]-Z, X-Y)\), the call should fail, since \([1,2|Z]-Z\) does not encode an empty list (it encodes of course list \([1,2]\)) and should not unify with L-L.

More precisely, it is not possible to unify \(L = [1,2|Z]\) and \(L = Z\), because that would require \(Z = [1,2|Z]\), i.e. to substitute a variable, \(Z\), by a term, \([1,2|Z]\), where the variable occurs.

This “occurs-check” failure does not occur in most Prolog systems that do not implement it correctly (ending in an infinite loop \(X = f(X) = f(f(X)) = f(f(f(X))) = ...\)).

For this reason, instead of checking if the two variable L-T are the same through unification (i.e. with predicate “=/2”), the clause tests this case with predicate “==/2”, that only succeeds if the terms are already the same at the time of call.

\[
\text{rev\_diff}(\text{L-T}, \text{L-T}) : - \text{L} == \text{T}
\]
The declarativity of Prolog allows simple programs to solve apparently complex problems.

The Langford(M,N) problem aims at finding a list of M*N elements, such that the integers 1 to M each appear N times, in such positions that between two occurrences of any J (in the range 1..M) are separated by J different elements.

The specification of the Langford(4,2) is quite obvious as when the sublist/2 predicate is used

```
langford42(L):-
    L = [__,__,__,__,__,],
    sublist([1,__,1], L),
    sublist([2,__,__,2], L),
    sublist([3,__,__,__,3], L),
    sublist([4,__,__,__,__,4], L).
```

Of course, such simple specification is only possible due to the flexibility of logic programming, namely the lack of fixed input-output arguments.
List Processing in Prolog: Zebra

A more challenging problem is the classic zebra puzzle:

1. There are five houses, each of a different color and inhabited by men of different nationalities, with different pets, drinks, and cigarettes.
2. The Englishman lives in the red house.
3. The Spaniard owns the dog.
4. Coffee is drunk in the green house.
5. The Ukrainian drinks tea.
6. The green house is immediately to the right of the ivory house.
7. The Old Gold smoker owns snails.
8. Kools are smoked in the yellow house.
9. Milk is drunk in the middle house.
10. The Norwegian lives in the first house on the left.
11. The man who smokes Chesterfields lives in the house next to the man with the fox.
12. Kools are smoked in the house next to the house where the horse is kept.
13. The Lucky Strike smoker drinks orange juice.
15. The Norwegian lives next to the blue house.

16. NOW, who drinks water? And who owns the zebra?
List Processing in Prolog: Zebra

- Again, this problem is solved with a very simple program, but requires slightly more involved data structures. Here is a possible solution
  - A solution is a list of 5 entities, each composed of a house, a nationality, a pet, a drink and a cigarette brand.
  - These different components can be associated in a single term, h(H,N,P,D,C).
  - In list L, an element A that is before another element B, encodes the fact that house $H_A$ is to the left of house $H_B$.

- Once this is done, the program has the following structure, where the constraints are still to be specified

```
zebra(L):-
  L = [h(H1,N1,P1,D1,C1),
      h(H2,N2,P2,D2,C2),
      h(H3,N3,P3,D3,C3),
      h(H4,N4,P4,D4,C4),
      h(H5,N5,P5,D5,C5)],
  ... constraints...
```
Most constraints associate two entities in the same house. For example

2. The Englishman lives in the red house.

This type of constraint (common to constraints 2, 3, 4, 5, 7, 8, 13, 14 and the implicit constraints in the query) is simply modelled by imposing that a term with the specified association is a member of list L

```
member(h(red,english,_,_,_),L),  %2
```

Constraints 9 and 10 state facts about the house in the middle and the leftmost house, which are respectively the third and the first member of list L.

9. Milk is drunk in the middle house.

10. The Norwegian lives in the first house on the left.

```
L = [_,_,h(_,_,_.,milk,_),_.,],  % 9
L = [h(_,norwegian,_,_,_)| _],  % 10
```
Another constraint refers to a house left to another

6. The green house is immediately to the right of the ivory house.

This type of constraint is simply modelled with the sublist relation

```
sublist([h(ivory,_,_,_,_),h(green,_,_,_,_)],L),  % 6
```

Other constraints refer to houses next to eachother, but not specifying which is to the left and which is to the right. Hence, the predicate next/3 may be defined specifying both alternatives

```
next(A,B,L):- sublist([A,B],L).
next(A,B,L):- sublist([B,A],L).
```

Now, constraints 11, 12 and 15 may be expressed declaratively. For example.

15. The Norwegian lives next to the blue house.

```
next(h(_,norwegian,_,_,_),h(blue,_,_,_,_),L),  % 15
```
Here is the full program

```prolog
zebra(L) :-
    L = [h(H1,N1,P1,D1,C1),
         h(H2,N2,P2,D2,C2),
         h(H3,N3,P3,D3,C3),
         h(H4,N4,P4,D4,C4),
         h(H5,N5,P5,D5,C5)],
    member(h(red,english,_,_,_),L), % 2
    member(h(_,spanish,dog,_,_),L), % 3
    member(h(green,_,_,coffee,_),L), % 4
    member(h(_,ukrainian,_,tea,_),L), % 5
    sublist([h(ivory,_,_,_,_),h(green,_,_,_,_)],L), % 6
    member(h(_,snailIs,_,oId gold),L), % 7
    member(h(yellow,_,_,_,kool),L), % 8
    L = [_,_,h(_,_,_,mIlk,_),_], % 9
    L = [h(_,norwegian,_,_)] % 10,
    next(h(_,_,_,_,chesterfields),h(_,_,fox,_,_),L), % 11
    next(h(_,_,_,_,kool),h(_,_,horse,_,_),L), % 12
    member(h(_,_,orange,lucky strike),L), % 13
    member(h(_,japanese,_,_,parliaments),L), % 14
    next(h(_,norwegian,_,_),h(blue,_,_,_),L), % 15
    member(h(_,_,zebra,_,_),L), % Q1
    member(h(_,_,_,water,_),L). % Q2
```