PART II

SML to Prolog transition: similarities

SML

```
fun append ([], ys) = ys
  | append (x::xs, ys) =
                x::append (xs, ys)
```

Prolog

```
append([], L, L).
append([X|L1], L2, [X|L3]) :-
        append(L1, L2, L3).
```

“Prologized” SML

```
fun append([], L) = L
  | append(X::L1, L2) =
    let val L3 = append(L1, L2)
    in  X::L3 end
```

“SML-ized” Prolog

```
append([], L, Res) :- Res = L.
append([X|L1], L2, Res) :-
        append(L1, L2, L3),
        Res = [X|L3].
```

function	predicate
clause	clause
variable: a single known value	variable: a single, perhaps unkown value
pattern: interpreted in compilation time	pattern: first-class data structure
simple patterns: `X::X::Xs` not allowed	complex patterns, e.g., `[X,X|Xs]`
one-way pattern matching	two-way pattern matching
unambiguous clause selection	multiple choice clause selection
a single result	nondeterminism (multiple results)
one-way usage	multi-way usage (e.g., composing and decomposing via `append`)
data constructor function	compound structure (record)
embedded function calls	conjuction (using auxiliary variables)

SML to Prolog transition: further examples

SML	**Prolog**
`fun append xs ys = foldr op:: ys xs`	`/* Higher order predicates are seldom used in Prolog */`

`fun fact 0 = 1`
`  | fact n = n * fact (n-1)`
	`fact(0, 1).`
	`fact(N, F) :-`
	`    N>0, N1 is N-1,`
	`    fact(N1, F1), F is N*F1.`

typed language higher order function recursion exception	untyped language recursion, rarely higher order predicates backtracking loop (e.g., common elements of two lists) failure, exception

Summary: Semantics of Prolog programs

- Meaning of a Prolog program = what answers (substitutions) we get by running a goal:
  - Procedural semantics — the explained execution and unification algorithm.
  - Declarative semantics:
    - program: set of logic statements (clauses, i.e., implications);
    - result of a goal: a substitution for which the goal is a **consequence** of the program.
- The procedural semantics gives only results which are also results of the declarative semantics! (If the predicates are “true”, an invalid result is not possible, “only” the infinite loop is `:-( )`

Reminder: The Prolog execution mechanism (in a nutshell)

- (Start): If the goal is empty $\longrightarrow$ success.
- (Proceed): Look for **first** clause head which is unifiable with the first subgoal (by making a fresh copy of each clause, iterating though the clauses from top to bottom).
- If found:
  - If there are further potential matches, then create a choicepoint: push current execution state (goal + number of clause matched) onto the stack.
  - Perform the substitutions of the unification both on the goal and the clause body.
  - Replace the first subgoal with the clause body, then go to (Proceed).
- If there is no matching clause head, *pop* the last choicepoint, restore the stored sate, go to (Proceed) and look for the next matching clause head.

Reminder: The Prolog unifi cation algorithm (in a nutshell)

- Determining the **most generic unifier**:
  - Identical variables and constants can be unified without substitutions.
  - Variable can be unified with everything else, using trivial substitution (without inclusion check)
  - Two compound structures are unifiable if their functors are identical and their arguments are unifiable respectively after performing the substitutions resulting from the unification of the preceding arguments. The composite of all the substitutions of the argument unifications is the most generic unifier.
  - In all other cases the two terms are not unifiable, the unification fails.

Part II: Prolog programming techniques

- Part I was:
  - about the basics of the Prolog language;
  - focusing on logicly clean features.
- Part II will introduce:
  - built-in predicates; and
  - programming techniques

  which help to:
  - write efficient Prolog applications;
  - by applying tools beyond the limits of clean logic.

Prolog programming techniques – Contents

- Narrowing the search space
- Control predicates
- Determinism and indexing
- About Prolog implementations
- Tail recursion and accumulators
- Algorithms in Prolog
- Collecting and enumerating solutions
- Solution-collecting predicates
- Meta-logic predicates
- Modularity
- Higher order predicates
- Dynamic database handling
- Lexical analysis
- “Traditional” built-in predicates

NARROWING THE SEARCH SPACE

Prolog language features for narrowing the search space

- Tools
  - the **cut** built-in predicate: `!` (from the early Prolog environments)
  - conditional disjunctive structure (later extension): `( if -> then ; else )`
- Conditional structure — procedural semantics (reminder)
  Execution of the `(if->then;else),cont` goal:
  - Execute the `if` subgoal (in a separate "environment").
  - If the `if` subgoal succeeds, then reduce the goal to `then,cont`, using the substitutions coming from the **first** solution of the `if` subgoal. **Further solutions of the `if` goal are ignored.**
  - If the `if` subgoal fails, then reduce the goal to `else,cont`.
- Conditional structure — alternative procedural semantics:
  - The conditional structure is considered as a special kind of disjunction:

```
(   if, {cut}, then
;   else
)
```

  - Meaning of **{cut}**: remove the choicepoints created by the `if` subgoal, and inhibit the selection of `else` as an alternative.

Conditional structure: choicepoints in the condition

- So far we have mostly seen deterministic conditions.
- Example: `first_pos_element(L, P)`: `P` is the first positive element of the `L` list.
  - First solution, using recursion (the *engineer*)

    ```
    first_pos_element([FP|_], FP) :- FP > 0.
    first_pos_element([X|L], FP) :- X =< 0, first_pos_element(L, FP).
    ```

  - Second solution, backtracking search (the *mathematician*)

    ```
    first_pos_element(L, FP) :-
            append(Nk, [FP|_], L), FP > 0, \+ has_pos_element(Nk).

    has_pos_element(L)  :- member(P, L), P > 0.
    ```

  - Third solution, conditional structure (the quick prototyping *Prolog hacker*)

    ```
    first_pos_element(L, FP) :-
            (   member(FP, L), FP > 0 -> true
            ;   fail                                % this line can be omitted
            ).
    ```

- Note: the third solution relies on the order in which `member/2` enumerates the elements of a list!

The cut predicate

- Execution of the cut built-in predicate (called `!`): disallows the selection of any further clauses and deletes all choicepoints created by preceding goals in the body of the current clause.
- Examples for using the cut:
  - The engineer:

    ```
    first_pos_element([FP|_], FP) :- FP > 0, !.
    first_pos_element([X|L], FP) :- X =< 0, first_pos_element(L, FP).
    ```
  - The Prolog hacker:

    ```
    first_pos_element(L, FP) :-  member(FP, L), FP > 0, !.
    ```
- Why to cut branches in the search tree?
  - because *we* know that there are no solutions there, but the Prolog implementation does not — *green cut*, semantically “innocent”.
    - (For example, most Prolog implementations do not know that $X > 0$ and $X \leq 0$ are mutually exclusive conditions, see indexing.)
  - we purposefully throw away solutions — *red cut*, changes the meaning of the program
    - (Many red cuts are created from green cuts when the “unnecessary” conditions are removed — e.g., `X =< 0` in the second clause above.)

Examples for using the cut

```
% fact(+N, ?F): N! = F.
fact(0, 1) :- !.                                  % green cut
fact(N, F) :- N > 0, N1 is N-1, fact(N1, F1), F is N*F1.

% last(+L, ?E): Last element of L is E. (in library lists)
last([E], E) :- !.                                % green cut
last([_|L], E) :- last(L, E).

% positives(+L, -P): P contains all the positive elements of L.
positives([], []).
positives([E|Es], [E|Ps]) :-
       E > 0, !,                                  % red cut
       positives(Es, Ps).
positives([_E|Es], Ps) :-
       /* \+ _E > 0, */ positives(Es, Ps).
```

Note: the above examples are not perfect, more efficient and more generic equivalents will be presented later on!

The defi nition of the cut

- Auxiliary concept
  - The **parent** of a goal G is the goal which was unified with the head of the clause containing G.
  - E.g., the parent of the cut in `last([E], E) :- !.` could be the goal `last([7], X)`.
  - The `g` (ancestors) debugger command prints all the ancestors of the current goal (the parent, the parent of the parent, etc.)
- Execution of the cut:
  - always succeeds: removes all choicepoints in the search tree, from the current execution state up to the state of the parent goal (inclusive).
- The cut removes two kinds of choicepoints:

```
r(X):- s(X), !.    % the choicepoints in s(X) - commit to the first solution
                   % of all goals preceding the cut
r(X):- t(X).       % the choicepoints in r(X) - commit to the clause containing the cut
```

- Depicting the cut in the 4-port box model: the **redo** port of the cut is connected straight to the **fail** port of the parent box.

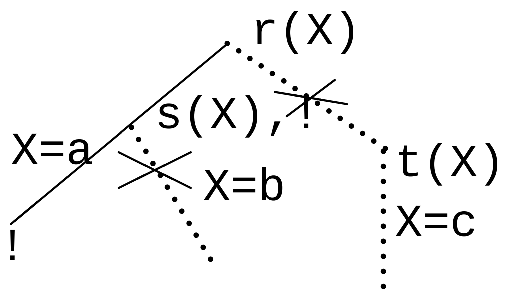
Choicepoints removed by the cut

```
% example without cut
q(X):- s(X).
q(X):- t(X).

% same example with cut
r(X):- s(X), !.
r(X):- t(X).

s(a).       s(b).       t(c).

% executing the example without cut
:- q(X), write(X), fail.
                   --->        abc
% executing the example with cut
:- r(X), write(X), fail.
                   --->        a
```



Implementing the disjunctive conditional using the cut

- The disjunctive conditional – similarly to the disjunction in general – can be replaced with an auxiliary predicate:

```
p :-                                 p :-
    ...                                  ...
    (   cond1 -> then1                   aux(...)
    ;   cond2 -> then2                   ... .
    ;   ...                   ⟹
    ;   else                         aux(...) :- cond1, !, then1.
    )                                aux(...) :- cond2, !, then2.
    ... .                            ...
                                     aux(...) :- else.
```

- The `else` part can be omitted, then the equivalent clause is omitted as well.
- In SICStus mode, the `cond`*`I`* parts cannot contain a cut. In ISO mode they can, but their range of effect is limited to the conditional goal.
- The `then`*`I`* parts can contain cuts. The range of effect of these cuts is the complete `p` predicate, as opposed to the cut generated from the `->` arrow. (The implementation uses a special *long range* cut in this case.)
- Cut inside conditional structures is rarely needed and is discouraged.

Examples for using conditional disjunctive structures

```
% fact(+N, ?F): N! = F.
fact(N, F) :-
        (   N = 0 -> F = 1
        ;   N > 0, N1 is N-1, fact(N1, F1), F is N*F1
        ).

% last(+L, ?E): the last element of the non-empty list L is E.
last([E|L], Last) :-
        (   L = [] -> Last = E
        ;   last(L, Last)
        ).

% positives(+L, ?Ps): Ps consists of the positive elements of L.
positives([], []).
positives([E|Es], Ps) :-
        (   E > 0 -> Ps = [E|Ps0]
        ;   Ps = Ps0
        ),
        positives(Es, Ps0).
```

First use case – commitment to a clause

- Commitment to a clause usually means a simple conditional structure.

```
parent :- condition, !, then.
parent :- else.
```

- The cut makes the execution of the inverse of `condition` unecessary. The logicly clean but inefficient form:

```
parent :- condition, then.
parent :- \+ condition, else.
```

The above to froms are equivalent only if `condition` does not create a choicepoint.

- Analogy: if `a, b` and `c` are logic variables (e.g., in SML) then
  `if a then b else c` $\equiv \mathtt{a} \wedge \mathtt{b} \vee \neg \mathtt{a} \wedge \mathtt{c}$

- It is recommended to write the unnecessary negated condition in comments:

```
parent :- condition, !, then.
parent :- /* \+ condition, */ else.
```

Conditional structures

Conditional structure —example

```
% abs(X, A): A is the absolute value of X.
abs(X, A)    :- X < 0, !, A is -X.
abs(X, X) /* :- X >= 0 */.
```

Generic form

```
p :- cond1, !, then1.
p :- cond2, !, then2.
...
p :- else.
```

Disjunctive conditional structure

```
abs(X, A) :-
        (   X < 0 -> A is -X
        ;   A = X
        ).
```

Generic structure

```
p :-
        (   cond1 -> then1
        ;   cond2 -> then2
        ;   ...
        ;   else
        ).
```

Conditional structures and head unifi cation

- Warning: unification of the head of a clause is always part of the condition!

```
% head unification before cut:   % unification made explicit:
abs(X, X) :- X >= 0, !.          abs(X, A) :- A = X, X >= 0, !.
abs(X, A) :- A is -X.            abs(X, A) :- A is -X.
```

- Head unification can cause problems, if the predicate is used for verification:

```
| ?- abs(10, -10).  --->  yes
```

- The solution is the **golden rule of cutting**:
  - Always perform unification of output parameters *after* the cut!

    ```
    abs(X, A) :- X >= 0, !, A = X.
    abs(X, A) :- A is -X.
    ```

  - This rule results in not only more generally usable but also more efficient code: the output argument is substituted only when its value is already known
  - **Output parameters** — when cut is involved, usually there is no two-way usage

Rule following variants of the earlier examples

```
% fact(+N, ?F): N! = F.
fact(0, F) :- !, F = 1.
fact(N, F) :- N > 0, N1 is N-1, fact(N1, F1), F is N*F1.

% last(+L, ?E): the last element of the non-empty list L is E.
last([E], Last) :- !, Last = E.
last([_|L], E)  :- last(L, E).

% positives(+L, ?Ps): Ps consists of the positive elements of L.
positives([], []).
positives([E|Es], Ps) :-
        E > 0, !, Ps = [E|Ps0], positives(Es, Ps0).
positives([_E|Es], Ps) :-
        /* \+ _E > 0, */ positives(Es, Ps).
```

Note: conditions in the disjuncitve form are inherently explicit, there is no hidden head unification, therefore it is strongly recommended to use disjunctive conditional structures instead of cuts wherever possible.

Example suite: `max(X, Y, Z)`: the maximum of `X` and `Y` is `Z`.

- 1st variant, clean Prolog. Slow (pre-substitution, two comparisons), creates choicepoint.

```
max(X, Y, X) :- X >= Y.
max(X, Y, Y) :- Y > X.
```

- 2nd variant, green cut. Slow (pre-substitution, two comparisons), does not create choicepoint.

```
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y) :- Y > X.
```

- 3rd variant, red cut. Faster (pre-substitution, one comparison), does not create choicepoint, but not good for verification, e.g., `| ?- max(10, 1, 1)` succeeds.

```
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y).
```

- 4th variant, red cut. Correct, fast (one comparison, no pre-substitution) and does not create a choicepoint.

```
max(X, Y, Z) :- X >= Y, !, Z = X.
max(X, Y, Y) /* :- Y > X */.
```

Second use case —commitment to the fi rst solution

- When is the cut committing to the first solution used?
  - decidable question not resulting in substitutions;
  - problem-specific optimization;
  - utilization of predicates creating an inifinite search space.
- Decidable question: predicate call with fully substituted arguments

```
% exists_long_enough_route(+N, +A, +B, +Min):
% There exists a route of N steps between A and B,
% which is at least Min km long.
exists_long_enough_route(N, A, B, Min) :-
        route(N, A, B, Len), Len >= Min, !.
```

- It is usually meaningless to return/expect multiple answers for a decidable question.

Problem specifi c optimization

- The problem: find the length of a “plateau” at the beginning of a list (a plateau is a continuous sublist consisting of identical elements).

```
% The first element of list L is repeated H times
% at the start of the list.
prefixlength(L, H) :-
        L = [E|_], append(Es, Tail, L),
        \+ Tail = [E|_], !,             % red cut
        /* identical(Es, E), */
        length(Es, H).
/*
% identical(Es, E): All elements of Es are E.
identical([], _).
identical([E|Es], E) :-
        identical(Es, E).
*/
| ?- prefixlength([1,1,1,2,3,5], H).
H = 3 ? ; no
```

Taming the infi nite choicepoint: `memberchk` (`lists` library)

- Definition of `memberchk/2`:

```
% memberchk(X, L): first solution of the query "X is an element of list

% 1st variant                    % 2nd equivalent variant
memberchk(X, L):-                memberchk(X, [X|_]) :- !.
        member(X, L), !.         memberchk(X, [_|L]) :-
                                         memberchk(X, L).
```

- Use of `memberchk/2`
  - Decidable questions (fails immediately when forced to backtrack):

```
| ?- memberchk(1, [1,2,3,4,5,6,7,8,9]).
```

  - Put an element into an open ended list:

```
| ?- memberchk(1,L), memberchk(2,L), memberchk(1,L).
          L = [1,2|_A] ?
```

Handling open ended lists with `memberchk`: a dictionary

```
dict(D):-
        read(H-E), !,
        % The read(X) built-in predicate unifies X
        % with a term read from the standard input
        memberchk(H-E,D),
        write(H-E), nl,
        dict(D).
dict(_).
```

A run:

```
| ?- dict(D).
|: alma-apple.            |: alma-X.
alma-apple                alma-apple
|: korte-pear.            |: X-pear.
korte-pear                korte-pear
                          |: stop.    % not unifiable with H-E

                          D = [alma-apple,korte-pear|_A] ?
```

CONTROL PREDICATES

Control predicates, the `call/1` built-in predicate

- Control predicate: a built-in predicate effecting the Prolog execution mechanism (e.g., cut, if-then-else).
- Most of the control predicates are **higher order** predicates, i.e., one or more of their arguments are taken as calls to other predicates. (Often called as **meta predicates**.)
- The most commonly used meta predicate is `call/1`.
  - Call pattern: `call(+Goal)`.
  - `Goal` is a "callable term" (cf. `callable/1`), i.e., a structure or a name constant.
  - Meaning (declarative semantics): `Goal` is true.
  - Effect (procedural semantics): transforms `Goal` into a procedure call and invokes it.
- In the body of clauses it is allowed to use a variable (e.g., `X`) as a goal, this is transformed into a `call(X)` goal.

```
| twice(Goal) :- call(Goal), Goal.

| ?- twice(write(do)), nl.        --->  dodo
| ?- listing(twice).              --->  twice(A) :-
                                                call(user:A), call(user:A).
```

Control structures as predicates

- Control structures can also appear in the argument of `call/1`, because these themselves are present in the Prolog system as predicates:
  - `(’,’)/2` conjunction.
  - `(;)/2`: disjuction.
  - `(->)/2`: if-then.
  - `(;)/2`: if-then-else.
- The control structures inside a `call/1` argument are executed just as if they were interpreted (loaded with `consult/1`, as opposed to `compile/1`).
- Examples:

```
| ?- _Goal = (twice(write(do)), write(’ ’)), twice(_Goal), nl.
dodo dodo
| ?- twice((member(X, [a,b,c,d]), write(X), fail ; nl)).
abcd
abcd
```

Example for `call/1`: meta predicate measuring run-time

```
% Prints the time required to find the first solution or
% prove the failure of Goal, prefixed by Txt
time(Txt, Goal) :-
        statistics(runtime, [T0,_]),  % T0 is the CPU time in ms  passed since
                                      % the toplevel query was issued (w/o gc)
        (   call(Goal) -> Res = true
        ;   Res = false
        ),
        statistics(runtime, [T1,_]), T is T1-T0,
        format('~w run time = ~3d sec\n', [Txt,T]),
                     % ~w format seq: write using write/1
                     % ~3d format seq: write integer I as I/1000 to 3 digits
        Res = true.
```

`call/1` is relatively expensive: reversing a list of 1414 elements with `nrev` (approx. 1 million calls to `append`) with and without wrapping all calls to `append` in a `call/1`:

	with `call`	without `call`	slowdown
compiled	0.140 sec	1.680 sec	12.00 sec
interpreted	1.710 sec	3.520 sec	2.06 sec

Further built-in control predicates

- `\+ Goal`: `Goal` is “not provable”. Definition:

```
\+ X :- call(X), !, fail.
\+ _X.
```

- `once(Goal)`: `Goal` is true, and we only request its first solution. Definition:

```
once(X) :- call(X), !.
```

- `true`: constant true, always succeeds, `fail`: constant false, always fails.
- `repeat`: succeeds infinitely many times (creates an inifinite choicepoint). Definition:

```
repeat.
repeat :- repeat.
```

- `repeat/0` is used mostly to repeat a predicate with side effect. The infinite choicepoint can be neutralized by using a cut.
- Example (simple calculator):

```
bc :- repeat, read(Expr),
           (    Expr = end_of_file -> true
           ;    Res is Expr, write(Expr = Res), nl, fail
           ),
      !.
```

Example: defi nining a higher order relation

- Define implication (`P` ⇒ `Q`) by using negation:

```
% Q is true for all solutions of P.
forall(P, Q) :-
     \+ (P, \+Q). % syntactic reminder:
                  % space is required after the first \+ !

| ?- _L = [1,2,3],
     % All elements of _L are positive:
     forall(member(X, _L), X > 0).
true ?
| ?- _L = [1,-2,3], forall(member(X, _L), X > 0).
no
| ?- _L = [1,2,3],
     % _L is strictly monotonically increasing:
     forall(append(_,[A,B|_], _L), A < B).
true ?
```

- `forall/2` can only be used for decidable questions.

DETERMINISM AND INDEXING

Determinism

- A procedure *call* is **deterministic**, if it can succeed at most once.
- The successful execution of a procedure call is **deterministically run** if:
  - it did not leave any choicepoints in the search tree belonging to the call, i.e.:
    - it ran without creating any choicepoints (may depend on the Prolog implementation!);
    - or it created choicepoints but these were removed (either exhausted or cut) afterwards.
- In the execution trace of SICStus Prolog, a **?** signifies a **non**deterministic run:

```
p(1, a).  | | ?- p(1, X).                        % det. call
p(2, b).  |              1 1 Exit: p(1,a)   %      det. run
p(3, b).  | | ?- p(Y, a).                        % det. call,
          |            ? 1 1 Exit: p(1,a)   %      nondet. run
          | | ?- p(Y, b), Y > 2.                 % nondet. call
          |            ? 1 1 Exit: p(2,b)   %      nondet. run
          |              1 1 Exit: p(3,b)   %      det. run
```

Deterministic run

- What is the advantage of a deterministic run?
  - the runtime is shorter;
  - the memory footprint is smaller;
  - various optimization schemes (e.g., tail recursion) can be applied.
- How can the compiler tell that there is no need to create a choicepoint?
  - indexing;
  - cuts and conditional structures.
- With either of the following declarations the call `p(`*`Nonvar`*`, Y)` *does not* create a choicepoint:

```
p(1, a).
p(2, b).
```

```
p(1, Y) :- !,
    Y = a.
p(_, b).
```

```
p(X, Y) :-
    (   X =:= 1 -> Y = a
    ;   Y = b
    ).
```

Indexing —reminder

- What is indexing?
  - quick selection of clauses matching a particular call
  - using compile-time grouping of the clauses of the predicate.
- Most Prolog systems, including SICStus, performs *first argument indexing*.
- The target of indexing is the principal functor of the first argument:
  - for `C` number and name constants, it is `C/0`;
  - for compound terms with name `R` and arity `N` it is `R/N`;
  - for variables, it is undefined.
- Implementing indexing:
  - At compile-time, calculate the list of matching clauses for all possible functors.
  - At run-time, in practically constant time pick the relevant set.
  - **Important:** if the chosen set contains a single clause, *do not create a choicepoint!*

Indexing example

```
p(0, a).              /* (1) */          q(1).
p(X, t) :- q(X).      /* (2) */          q(2).
p(s(0), b).           /* (3) */
p(s(1), c).           /* (4) */
p(9, z).              /* (5) */
```

- Clauses matching the `p(A, B)` call:

`{(1) (2) (3) (4) (5)}`	if `A` is a variable;
`{(1) (2)}`	if `A = 0`;
`{(2) (3) (4)}`	if main functor of `A` is `s/1`;
`{(2) (5)}`	if `A = 9`;
`{(2)}`	in all other cases.

- Call examples:
  - `p(1, Y)` does not create a choicepoint.
  - `p(s(1), Y)` creates a choicepoint, but runs deterministically.
  - `p(s(0), Y)` runs nondeterministically.

Compounds and variables in the fi rst argument

- Compounds with identical functors in the first argument:
  - If the “details” of the first argument (a compound) are required to make a choice between the clauses, it is worth it to introduce an auxiliary predicate.
  - E.g., `p/2` and `q/2` are equivalent, but `q(`*`Nonvar`*`, Y)` *runs deterministically!*

```
p(0, a).            | q(0, a).             | q_aux(0, b).
p(s(0), b).         | q(s(X), Y) :-        | q_aux(1, c).
p(s(1), c).         |      q_aux(X, Y).    |
p(9, z).            | q(9, z).             |
```

- Replacing unification in the head with `=/2` (cf. SML layered patterns)
  - Indexing takes a call to `=/2` as the first subgoal of the body into account:
    In case of `p(X, ...)  :- X = Expr, ...` indexes based on the functor of `Expr`.
  - Example: inverse of length of a list, 0 if list is empty:

```
ilen([], 0).
ilen(L, RH) :- L = [_|_], length(L, H), RH is 1/H.
% ilen([X|L], RH) :-  length([X|L], H), RH is 1/H.
        % less efficient, because reconstructs [X|L].
% ilen(L, RH) :-  L \= [], length(L, H), RH is 1/H.
        % less efficient, because leaves choicepoint for L=[].
```

Indexing —further details

- Indexing and arithmetics
  - Indexing does not recognize arithmetical comparisons.
  - E.g., the `N = 0` and `N > 0` conditions are not “exclusive”.
  - The following `fact/2` procedure runs nondeterministically:

```
fact(0, 1).
fact(N, F) :- N > 0, N1 is N-1, fact(N1, F1), F is N*F1.
```

- Indexing and lists
  - It is often needed to distinguish between empty and non-empty lists.
  - The input list argument should be put in the first argument.
  - Indexing distinguishes between `[]` and `[...|...]` (resp. functors: `'[]'/0` and `'.'/2`).
  - The order of the two clauses is not relevant (assuming the predicate is called with a closed list) — but write the terminating clause first anyway.

Indexing list handling predicates: examples

- `append/3` runs deterministically if the first argument is a closed list.

```
append([], L, L).
append([X|L1], L2, [X|L3]) :-  append(L1, L2, L3).
```

- The trivial implementation of `last/2` runs nondeterministically:

```
% last(L, E): The last element of L is E.
last([E], E).
last([_|L], E) :- last(L, E).
```

- Introduce an auxiliary predicate, `last2/2` runs deterministically:

```
last2([X|L], E) :- last2(L, X, E).

% last2(L, X, E): The last element of [X|L] is E.
last2([], E, E).
last2([X|L], _, E) :- last2(L, X, E).
```

- `member` (from the `lists` library) enumerating the final element without leaving a choicepoint:

```
member(E, [H|T]) :-     member_(T, H, E).

% member_(L, X, E): E is an element of [X|L].
member_(_, E, E).
member_([H|T], _, E) :- member_(T, H, E).
```

Interaction of indexing and the cut

- How to interpret the cut during the construction of the index tables?
- E.g., `p(1, A)` does not create a choicepoint, but `q(1, A)` does!

```
p(1, Y) :- !, Y = 2.     % (1)   |   q(1, 2) :- !.           % (1)
p(X, X).                 % (2)   |   q(X, X).                % (2)
Arg1=1 → (1), Arg1≠1 → (2)       |   Arg1=1 → {(1),(2)}, Arg1≠1 → (2)
```

- The compiler considers the cut in indexing, if it is guaranteed that the cut is reached given a particular principal functor. In detail:
  - the first argument is a variable, constant, or compound term with only variable arguments;
  - all further arguments are variables;
  - all variable occurences in the head are independent;
  - first subgoal of the body is the cut (or follows a `=/2` replacing unification in the head).
- If these hold, the clauses below the cut are omitted from the entry belonging to the given functor.
- Example: `p(X, D, E) :- X = s(A, B, C), !, ....  p(X, Y, Z) :- ....`
- This is another justification for the golden rule of cutting:

$\boxed{\textbf{Output arguments should always be unified after the cut only!}}$

Effi ciency of the cut and indexing

- A Fibonacci-like sequence: $f_1 = 1; \quad f_2 = 2; \quad f_n = f_{\lfloor 3n/4 \rfloor} + f_{\lfloor 2n/3 \rfloor}, \quad n > 2$

```
% deterministic
fib(1, 1).
fib(2, 2).
fib(N, F) :-
        N > 2,
        N2 is N*3//4,
        N3 is N*2//3,
        fib(N2, F2),
        fib(N3, F3),
        F is F2+F3.
```

```
% runs deterministically
fibc(1, 1) :- !.
fibc(2, 2) :- !.
fibc(N, F) :-
        N > 2,
        N2 is N*3//4,
        N3 is N*2//3,
        fibc(N2, F2),
        fibc(N3, F3),
        F is F2+F3.
```

```
% no choicepoints created
fibci(1, F) :- !, F = 1.
fibci(2, F) :- !, F = 2.
fibci(N, F) :-
        N > 2,
        N2 is N*3//4,
        N3 is N*2//3,
        fibci(N2, F2),
        fibci(N3, F3),
        F is F2+F3.
```

- Run times for $N = 2000$

	`fib`	`fibc`	`fibci`
run time	990 msec	890 msec	830 msec
time to fail	440 msec	30 msec	0 msec
total	1430 msec	920 msec	830 msec
trail-stack size	4.1Mbyte	2.0 Mbyte	256 byte

- For `fibc`, time to fail is non-zero the system builds and processes a trail-stack, which stores undo information for variable substitutions.

Choicepointlessness with disjunctive conditional structures

- Choicepoints are usually created when executing a conditional structure.
- In **SICStus Prolog**, the `( cond -> then ; else )` structure is executed without creating a choicepoint, if `cond` contains only:
  - arithmetical comparison operators (e.g., `<, =<, =:=`), and/or
  - type checking predicates (e.g., `atom, number`), and/or
  - generic comparison operators (see later, e.g., `@<, @=<, ==`).
- Analogously, no choicepoint is created from a clause like `head :- cond, !, then.`, if all arguments of `head` are distinct variables, and `cond` is just like above.
- For example, no choicepoints are created with either of the following definitions:

```
vector_type(X, Y, Type) :-
    (   X =:= 0, Y =:= 0
        % X = 0, Y = 0 would not work
    ->  Type = null
    ;   Type = not_null
    ).
```

```
vector_type(X, Y, Type) :-
    X =:= 0, Y =:= 0, !,
    Type = null.
vector_type(_X, _Y, not_null).
```

PROLOG IMPLEMENTATION DETAILS

Prolog implementation milestones

- 1973: Marseille Prolog (A. Colmerauer et al.)
  - interpreter in Fortran language
  - term representation: structure-sharing
  - stack structure: single stack (freed upon backtracking)
- 1977: DEC-10 Prolog (D. H. D. Warren)
  - compiler in Prolog and assembly (+ interpreter in Prolog)
  - term representation: structure-sharing
  - stack structure: three stacks (all freed upon backtracking)
    - global stack: global variables (inside terms), garbage collected
    - local stack: procedures, choicepoints, variables, freed upon deterministic run
    - trail: variable substitutions, freed after cut
- 1983: WAM — Warren Abstract Machine (D. H. D. Warren)
  - abstract machine for executing Prolog programs
  - term representation: structure-copying
  - three stacks as in DEC-10 Prolog, global stack stores the structures
  - Most modern Prolog systems are WAM based (SICStus, SWI, GNU Prolog, . . .)

Term representation

- Comparison of the two possible term representation strategies:

	structure-sharing	structure-copying
storage size	prop. to number of variables	prop. to structure size
building cost	constant	prop. to structure size
decomposition cost	more	less

- Structure **building**: unification of a variable and a structure in the **program text**
- **Important:** the unification ofa free variable with a structure appearing as the value of another variable always has constant cost!
- Example:

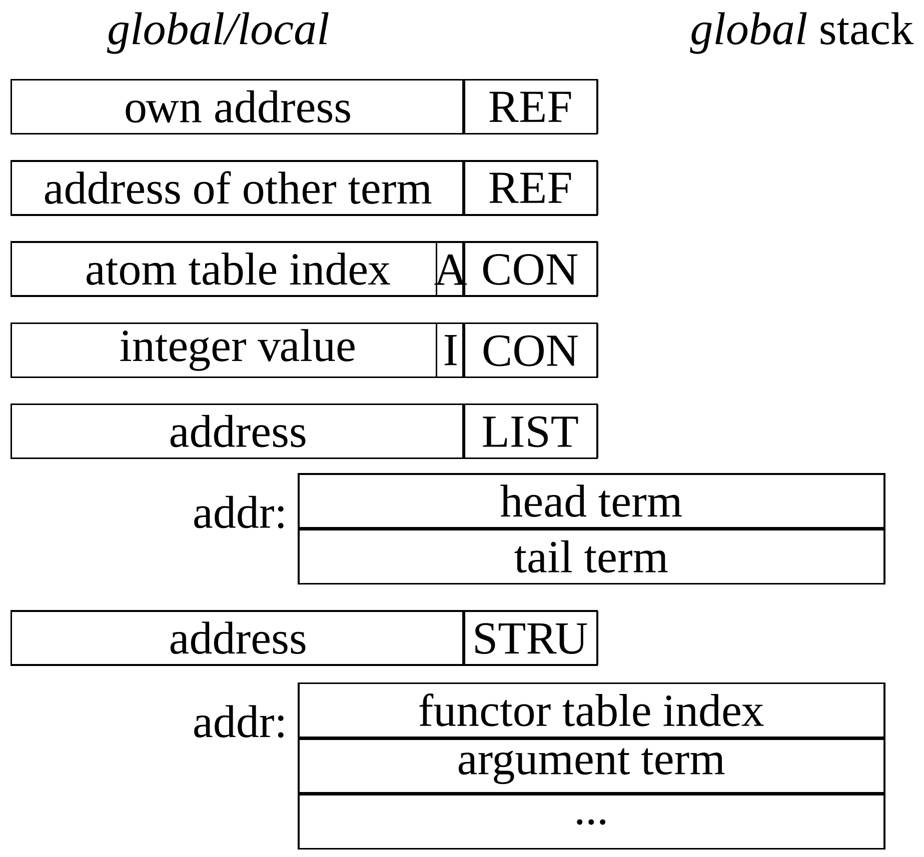
```
lengthen(L, [1,2,3,...,n|L]).

replicate(0, L) :- !, L = [].
replicate(N, L) :-
        lengthen(L0, L), N1 is N-1, replicate(N1, L0).
```

- The cost of `replicate(`n`, L)` is $O(n)$ with structure-sharing, $0(n^2)$ with structure-copying.
- Still, in practice, structure-copying proved to be more efficient.

WAM: Storage of Prolog terms

- The recommended term representation in WAM is LBT (Low Bit Tagging scheme)
  - Uninstantiated variable:
  - Reference to other variable/term:
  - Name constant
  - Integer
  - List
  - Structure



- The SICStus 3.x system stores the tags on the *upper* 4 bits, thus the size of the stacks is limited to 256 MByte. (SICStus 4 will already use LBT.)

WAM: Further details

- Handling of variables
  - Unification of two variables: the *younger* becomes a reference to the *older*
  - **Dereferencing**: follow and resolve a chain of references
  - Uninstantiated variable $\equiv$ self reference $\Rightarrow$ easier dereferencing
- Backtracking
  - **Conditional variable**: uninstantiated variable, older than the newest choicepoint
  - When substituting a conditional variable, its address is stored on the trail
  - At backtrack, variable substitutions are undone using the trail, then the trail is truncated
- SICStus programs can be compiled into WAM code: (`File.pl` $\Rightarrow$ `File.wam`):

```
| ?- prolog:fshell_files(File, wam, []).
```

- Introduction to WAM (tutorial): `http://www.vanx.org/archive/wam/wam.html`

TAIL RECURSION AND ACCUMULATORS

Tail recursion, optimization

- In general, recursion is expensive both in time and in space.
- We talk about tail recursion, when
  - the recursive call is the last call in the clause body, or the last call of a branch of a disjunction in the last position, etc.; and
  - at the time of the recursive call, there is no choicepoint in the predicate (the goals preceding the recursive call ran deterministically, no disjunction was left open).
- Tail recursive optimizaton: **before** the last call is executed, the memory allocated by the predicate is released (becomes candidate for garbage collection).
- This optimization is performed not only for recursive calls but for all **last** calls in general — the accurate name: *last call optimization*.
- Thus, tail recursion does not increase memory usage, the recursive predicate can run infinitely long — like loops in imperative languages. Example:

```
loop(State) :- next(State, State1), !, loop(State1).
loop(_State).
```

Bringing predicates to a tail recursive form —summing a list

- “Natural”, not tail recursive definition of the sum of a list:

```
% sum(+L, ?S): The sum of the elements of list L is S (S = 0+Ln+Ln−1+...+L1).
sum([], 0).
sum([X|L], S):- sum(L,S0), S is S0+X.
```

- First tail recursive variant, for checking only:

```
% sum1(+L, +S): The sum of the elements of list L is S (S-L1-L2-...-Ln = 0).
sum1([], 0).
sum1([X|L], S) :- S0 is S-X /* instead of S is S0+X */, sum1(L, S0).
```

- Second tail recursive variant, only prints the result on the console:

```
% sum2(+L): Prints the sum of the elements of list L (0+L1+L2+...+Ln).
sum2(L):- sum2(L, 0).

% sum2(+L, +S0): Prints the sum of the elements of L plus S0.
sum2([], S) :-    write(S), nl.
sum2([X|L], S0):- S1 is S0+X, sum2(L, S1).
```

- To be able to return the sum as a result, we need a further output argument.

Tail recursive sum of a list —using an accumulator pair

- Third variant: full-fledged tail recursive list summing:

```
% sum3(+L, ?S): The sum of the elements of list L is S.
sum3(L, S):- sum3(L, 0, S).

% sum3(+L, +S0, ?S): S is the sum of the elements of L plus S0. (≡ ∑Li = S − S0)
sum3([], S, S).
sum3([X|L], S0, S):-
        S1 is S0+X, sum3(L, S1, S).
```

- The tail recursive `sum3` is more than **3 times faster** than the not tail recursive `sum`!
- The **accumulator** is the declarative equivalent of the imperative (i.e., mutable) variable.
  - `S0` and `S` form an *accumuator pair* in the `sum3(L, S0, S)` predicate.
  - The two members of the accumulator pair represent the value of a mutable quantity at distinct time instances:
    - `S0` is the value of the sum at the **call** of `sum3/3`: the initial value of the imperative variable;
    - `S` is the value of the sum **after the completion** of `sum3/3`: the final value of the imperative variable.

Using accumulators

- In general, multiple consecutive changes can be written with accumulators:

```
p(..., A0, A) :-
        q0(..., A0, A1), ...,
        q1(..., A1, A2), ...,
        qn(..., An, A).
```

- Bringing the second clause of `sum3/3` to this form:

```
sum3([X|L], S0, S):- plus(X, S0, S1), sum3(L, S1, S).

plus(X, S0, S) :- S is S0+X.
```

- Naming convention for accumulator variables: initial value: *`Var`*`0`; intermediate values: *`Var`*`1`, $\ldots$, *`Var`*n; end value: *`Var`*.
- A Prolog accumulator pair is conceptually the same as the couple of the accumulator argument and the function result in SML.

Using accumulators —contd.

- Sum of three lists

```
% sum_3_lists(+L, +LL, +LLL, +S0, ?S): The sum of list sums of the
% lists L, LL, LLL is S-S0
sum_3_lists(L, LL, LLL, S0, S) :-
        sum3(L, S0, S1), sum3(LL, S1, S2), sum3(LLL, S2, S).
```

Forward note: the DCG (Definite Clause Grammar) form of the above rule:

```
sum_3_lists(L, LL, LLL) --> sum3(L), sum3(LL), sum3(LLL).
```

- Multiple accumulation — sum and square sum of lists

```
% sum12(+L, +S0, ?S, +Q0, ?Q): S − S0 = ∑Li, Q − Q0 = ∑Li ∗ Li
sum12([], S, S, Q, Q).
sum12([X|L], S0, S, Q0, Q):-
        S1 is S0+X, Q1 is Q0+X*X, sum12(L, S1, S, Q1, Q).
```

- Grouping multiple accumulators

```
% sum12(+L, +S0/Q0, ?S/Q): S − S0 = ∑Li, Q − Q0 = ∑Li ∗ Li
sum12([], SQ, SQ).
sum12([X|L], S0/Q0, SQ):-
        S1 is S0+X, Q1 is Q0+X*X, sum12(L, S1/Q1, SQ).
```

Difference lists

- `revapp` as an accumulating predicate

```
% revapp(Xs, L0, L): By prepending the reverse of Xs to L0 we get L.
% In other words: the reverse of Xs is L-L0.
revapp([], L, L).
revapp([X|Xs], L0, L) :-
        L1 = [X|L0], revapp(Xs, L1, L).
```

- The `L-L0` notation (*difference list*): denotes the list which we get by omitting `L0` from the end of `L`. (Assuming that `L0` is a suffix of `L`.)
- For example, difference lists representing `[1,2,3]`:
  - `[1,2,3,4]-[4]`, `[1,2,3,a,b]-[a,b]`, `[1,2,3]-[]`, ...
  - In the most generic (open ended) difference list the “tail” is a variable: `[1,2,3|L]-L`
- It is possible to append a (difference) list to an open ended difference list in constant time:

```
% app_dl(DL1, DL2, DL3): the concatenation of DL1 and DL2 difflists is DL3.
app_dl(L-L0, L0-L1, L-L1).

| ?- app_dl([1,2,3|L0]-L0, [4,5|L1]-L1, DL).
       ⟹ DL = [1,2,3,4,5|L1]-L1, L0 = [4,5|L1]
```

- The open ended difflist is for “one time use” only: following append, it is no more open ended!

Difference lists (cont.)

- Example: linear time reverse, `nrev` style, with difflists:

```
% nrev(L, DR): The reverse of L is the DR difference list.
nrev_dl([], L-L).                    % L-L ≡ empty difflist
nrev_dl([X|L], DR) :-
    nrev_dl(L, DR0),
    app_dl(DR0, [X|T]-T, DR).   % [X|T]-T ≡ one long difflist

% app_dl(DL1, DL2, DL3): the concatenation of DL1 and DL2 difflists is DL3.
app_dl(L-L0, L0-L1, L-L1).

% The reverse of L is R.
rev(L, R) :-
        nrev_dl(L, R-[]).
```

- It is advantageous to swap the two calls in the body of `nrev_dl/2` (tail recursion!)
- By `nrev_dl(L, R-R0)` $\Longrightarrow$ `rev2(L, R0, R)` and avoiding `app_dl` from the above `nrev_dl/2` we get `rev2/3`, which is identical to `revapp/3`!
- This minor change makes the program approx. **3 times faster** $\Rightarrow$ it is worthwile to use accumulators instead of difference lists!
- From now on, we will only use difference lists for documentation, in head comments.

append as an accumulating procedure

- Let us write a `prefix_suffix(Prefix, L, Suffix)` predicate!

```
% prefix_suffix(Prefix, L, suffix): The prefix of list L is Prefix, the
% remaining list is Suffix.
prefix_suffix([], L, L).
prefix_suffix([X|Xs], L0, L) :-
        L0 = [X|L1],
        prefix_suffix(Xs, L1, L).
```

- The accumulating step: `L0 = [X|L1]`, **omitting** an element from the start of the list.
- By swapping the 2nd and 3rd arguments the `prefix_suffix` is transformed into `append`!
- Thus we can consider `append` an accumulating procedure (even though the 2nd and 3rd arguments are swapped when compared to the customary accumulator pairs):

```
% append(Xs, L, L0): we get L by omitting the elements of Xs from the start
% of L0.  In other words: XS = L0-L.
append([], L, L).
append([X|Xs], L, L0) :-
        L0 = [X|L1], append(Xs, L, L1).
```

- The accumulating step: the `L0` variable is instantiated with a list, the tail of which is `L1`, the accumulated value: the variable in which we expect the result of the concatenation.

An example: sequences of the form $a^n b^n$

- First solution, $3n$ steps

```
% anbn(N, L): L consists of N a-s
% followed by N b-s.
anbn(N, L) :-
        an(N, a, AN),
        an(N, b, BN),
        append(AN, BN, L).

% an(N, A, L): L is a list containing
% the A element N times
an(0, _A, L) :- !, L = [].
an(N, A, [A|L]) :-
        N > 0,
        N1 is N-1,
        an(N1, A, L).
```

- Second solution, $2n$ steps

```
anbn(N, L) :-
        an(N, b, [], BN),
        an(N, a, BN, L).

% an(N, A, L0, L): L-L0 is a list
% containing A N times
an(0, _A, L0, L) :- !, L = L0.
an(N, A, L0, [A|L]) :-
        N > 0,
        N1 is N-1,
        an(N1, A, L0, L).
```

An example: sequences of the form $a^n b^n$ (cont.)

- Third solution, n steps

```
anbn(N, L) :-
        anbn(N, [], L).

% anbn(N, L0, L): The L-L0 list contains N a-s followed by N b-s.
anbn(0, L0, L) :- !, L = L0.
anbn(N, L0, [a|L]) :-
        N > 0,
        N1 is N-1,
        anbn(N1, [b|L0], L).
```

- Non tail recursive variant of the second clause

```
anbn(N, L0, L) :-
        N > 0, N1 is N-1,
        L1 = [b|L0],       % 1. step: prepend b to L0 => L1
        anbn(N1, L1, L2), % 2. step: prepend a^N1 b^N1 to L1 => L2
        L = [a|L2].       % 3. step: prepend a to L2 => L
```

$a^n b^n$ alakú sorozatok —solutions in other languages

- SML solution

```
local
  fun ab(0, L) = L
    | ab(N, L0) = #"a"::ab(N-1, #"b"::L0)
in fun anbn N = ab(N, [])
end
```

- C++ solution

```
link *anbn(unsigned n) {
  link *l = 0, *b = 0;    // prepend b-s to this
  link **a = &l;          // put a-s in this
  for (; n > 0; --n) {
    *a = new link('a');   // from upfront...
    a = &(*a)->next;      // ...towards the back
    b = new link('b', b); // from back to front
  }
  *a = b; return l;
}
```

COLLECTING AND ENUMERATING SOLUTIONS

Search in Prolog —enumeration or collection?

- Search problem: find values satisfying certain conditions.
- In Prolog, such a problem can be solved with two different approaches:
  - collection — collect *all* solutions, e.g., into a list;
  - enumeration — enumerate solutions via backtracking: get one solution at a time, but return *all* before failure.
- Simple example: find the even members of a list:

```
% Collection:
% even_members(L, Es): Es is the
% list of even members of L.
even_members([], []).
even_members([X|L], Es) :-
    X mod 2 =\= 0, !,
    even_members(L, Es).
even_members([E|L], [E|Es]) :-
    even_members(L, Es).
```

```
% Enumeration:
% even_member(L, E): E is an even
% member of the L list.
even_member([X|L], E) :-
    X mod 2 =:= 0, E = X.
even_member([_X|L], E) :-
    % _X either odd or even,
    % continue the enumeration:
    even_member(L, E).

% simpler solution:
even_member2(L, E) :-
    member(E, L), E mod 2 =:= 0.
```

What is common in enumeration and collection?

- Look for the common parts in `even_members` and `even_member` predicates!
- Both steps over the odd members and locates the first even member of the list:

```
% next_even(L0, E, L) :- The first even member of L0 is E, rest is L.
next_even([X|L0], E, L) :-
    X mod 2 =\= 0, !, next_even(L0, E, L).
next_even([E|L], E, L).
```

- The two procedures reusing `next_even`:

```
% even_members(L, Es): Es is the
% list of even members of L.
even_members(L0, Es) :-
    next_even(L0, E, L1), !,
    Es = [E|Es1],
    even_members(L1, Es1).
even_members(_, []).
```

```
% even_member(L, E): E is an even
% member of the L list.
even_member(L0, E) :-
    next_even(L0, E0, L1),
    (   E = E0
    ;   even_member(L1, E)
    ).
```

Comparison of the collecting and enumerating schemas

- Based on the procedures collecting resp. enumerating even numbers, find a generic schema for the two procedure types!
- In the general case, the search can have one or more `Param` parameters. For example, we can look for list members divisible by `Param`.
- The common building block: `next(V0, Param, S, V1)`: The first solution of the search space characterized by `V0` is `S`, and the remaining search space is `V1`, given parameter `Param`.

The collecting schema:

```
% L is the list of solutions with
% parameter Param in search space V0.
solutions(V0, Param, L) :-
  next(V0, Param, S, V1), !,
  L = [S|L1],
  solutions(V1, Param, L1).
solutions(_, _, []).
```

The enumerating schema:

```
% S is a solution with parameter Param
% in the search space V0.
solution(V0, Param, S) :-
  next(V0, Param, S0, V1),
  (   S = S0
  ;   solution(V1, Param, S)
  ).
```

A more complex example: enumerating plateaus

- In a list we call a *plateau*:
    - a sublist consisting of at least two, solely identical elements;
    - which is the longest among these (not extendable in either direction).
- The task: enumerate the plateaus and their starting positions in a list.
- `plateau(L, I, H)`: There is a `H` long plateau in `L` starting in position `I`.
- Quick-and-dirty prototype (à la Prolog hacker):

```
plateau0(L, I, H) :-
    Body = [E,E|_],
    append(Prefix, Body, L),
    \+ last(Prefix, E),
    length(Prefix, I0), I is I0+1,
    prefixlength(Body, H).
    % see definition of
    % prefixlength/2 earlier
```

```
plateau1(L, I, H) :-
    Body = [E,E|_],
    append(Prefix, Body, L),
    \+ last(Prefix, E),
    length(Prefix, I0), I is I0+1,
    % prefixlength/2 expanded:
    (   append(Es, Tail, Body),
        \+ Tail = [E|_] ->
        length(Es, H)
    ).
```

Enumerating plateaus —2nd, effi cient solution

- Let us use the solution enumerating schema: `solution(`*`V0, Param, S`*`)`!
  - *`V0`*: »`L, P`«, the list to traverse and the index of its first element;
  - *`Param`*: empty;
  - *`S`*: »`I, H`«, the solution plateau’s start position and length.

```
% There is an H long plateau at index I in list L.
plateau(L, I, H) :-
        plateau(L, 1, I, H).

% There is an H long plateau in position I in the list L
% which is indexed from P0.
plateau(L0, P0, I, H) :-
        % details of first plateau: I0 and H0,
        % remaining list after plateaus: L1:
        first_plateau(L0, P0, I0, H0, L1),
        (   I = I0, H = H0
        ;   P1 is I0+H0,    % start index of L1 = P1 is the same as
                            % previous solution’s length plus start index
            plateau(L1, P1, I, H)
        ).
```

Enumerating plateaus —2nd, effi cient solution (cont.)

- Finding the first solution:

```
% first_plateau(+L0, +P0, -I, -H, -L): In the L0 list indexed from P0 the
% first plateau occurs at I, its length is H, the remaining list is L.
first_plateau([E,E|L1], P0, I, H, L) :-
        !, I = P0, identical(L1, E, 2, H, L).
first_plateau([_|L1], P0, I, H, L) :-
        P1 is P0+1,
        first_plateau(L1, P1, I, H, L).

% identical(+L0, +E, +H0, -H, -L): By omitting the most elements from the
% start of L0 which are identical to E, the remaining list is L, the number
% of omitted elements is H-H0.
identical([X|L0], E, H0, H, L) :-
        E = X, !,
        H1 is H0+1,
        identical(L0, E, H1, H, L).
identical(L, _, H, H, L).
```

BUILT-IN PREDICATES FOR COLLECTING SOLUTIONS

Connection between collection and enumeration

- We have seen how to write a collecting *and* an enumerating predicate using a shared core predicate.
- Now examine how we can write an enumerating procedure reusing a collecting procedure and vice versa:
  - enumeration from collection: using the `member/2` built-in predicate, e.g.:

    ```
    even_member(L, E) :-
        even_members(L, Es), member(E, Es).
    ```

    Naturally this is inefficient!
  - collection from enumeration: using the built-in solution collecting procedures, e.g.:

    ```
    even_members(L, Es) :-
        findall(E, even_member(L, E), Es).
        % Es is the list of all E solutions
        % of the goal even_member(L, E).
    ```

The `findall(?Acc, :Goal, ?List)` built-in procedure

- The execution of the procedure (procedural semantics):
  - `Goal` is interpreted as a procedure call, and called (the `:` annotation means a meta (i.e., procedure) argument);
  - for all solutions, a *copy* of `Acc` is created (if it contains variables, they are replaced by fresh ones);
  - All `Acc` values are collected into a list, and this list is unified with `List` at the end.
- Examples for using `findall/3`:

```
| ?- findall(X, (member(X, [1,7,8,3,2,4]), X>3), L).
              ⟹   L = [7,8,4] ? ; no
| ?- findall(X-Y, (between(1, 3, X), between(1, X, Y)), L).
              ⟹   L = [1-1,2-1,2-2,3-1,3-2,3-3] ? ; no
```

- The meaning of the procedure (declarative semantics):
  `List` $=$ { `Acc` copy | $(\exists \mathtt{X} \ldots \mathtt{Z})$ `Goal` is true }
  where `X`, $\ldots$, `Z` are the free variables in the `findall` call (i.e., variables uninstantiated at the time of the call, which occur in `Goal`, but not in `Acc`).

The `bagof(?Acc, :Goal, ?List)` built-in procedure

- The execution of the procedure (procedural semantics):
  - `Goal` is interpreted as a procedure call, and called;
  - collects its solutions (instantiations of `Acc` and the free variables);
  - all instantiations of the free variables are *enumerated*, and for each, the corresponding values of `Acc` are given in `List`.
- Examples of using `bagof/3`:

```
graph([a-b,a-c,b-c,c-d,b-d]).

| ?- graph(_G), findall(B, member(A-B, _G), EndP).
                                    ⟹ EndP = [b,c,c,d,d] ? ; no
| ?- graph(_G), bagof(B, member(A-B, _G), EndP).
                                    ⟹ A = a, EndP = [b,c] ? ;
                                       A = b, EndP = [c,d] ? ;
                                       A = c, EndP = [d] ? ; no
```

- The meaning of the procedure (declarative semantics):
  `List` $=$ {`Acc` | `Goal` is true }, `List` $\neq$ `[]`.

The `bagof` solution collecting procedure (cont.)

- Explicit quantification
  - `bagof(Acc, V1 ^ ...^ Vn ^ Goal, List)` considers the variables `V1, ...,` `Vn` existentially quantified, i.e., they are not enumerated.
  - meaning: `List` $=$ { `Acc` | $(\exists$`V1`$,\ldots,$`Vn`$)$`Goal` is true } $\neq$ `[]`.

```
| ?- graph(_G), bagof(B, A^member(A-B, _G), EndP).
```

$\Longrightarrow$ `EndP = [b,c,c,d,d] ? ; no`

- Embedded collections
  - in case of free variables, `bagof` can be nondeterministic, thus it can be embedded:

```
% The degree list of the directed graph G is DL:
% DL = { A-N | N = |{ V | A-V ∈ G }|}
degrees(G, DL) :-
        bagof(A-N, Vs^(bagof(V, member(A-V, G), Vs),
                       length(Vs, N)                 ), DL).
```

(in the second comment line: $A-N \mid N = |\{\, V \mid A-V \in$ `G` $\}|$; "**directed**" is bold)

```
| ?- graph(_G), degrees(_G, DL).
```

$\Longrightarrow$ `DL = [a-2,b-2,c-1] ? ; no`

The `bagof` solution collecting procedure (cont.)

- Calculating the degree list more efficiently
  - it is advantageous to avoid control structures in the meta arguments
  - by adding an auxiliary predicate, the quantor becomes unnecessary:

```
% The degree of vertex A in G directed graph is N, N > 0.
degree(A, G, N) :-
        bagof(V, member(A-V, G), Vs), length(Vs, N).

% The degree list of G directed graph is DL:
degrees(G, DL) :-    bagof(A-N, degree(A, G, N), DL).
```

- Examples for minor differences between `bagof/3` and `findall/3`:

```
| ?- findall(X, (between(1, 5, X), X<0), L).   ⟹ L = [] ? ; no
| ?-   bagof(X, (between(1, 5, X), X<0), L).   ⟹ no
| ?- findall(S, member(S, [f(X,X),g(X,Y)]), L).
                                ⟹ L = [f(_A,_A),g(_B,_C)] ? ; no
| ?-   bagof(S, member(S, [f(X,X),g(X,Y)]), L).
                                ⟹ L = [f(X,X),g(X,Y)] ? ; no
```

- `bagof/3` is logicly cleaner than `findall/3`, but it is more time consuming!

The `setof(?Acc, :Goal, ?List)` built-in procedure

- The execution of the procedure:
  - same as: `bagof(Acc, Goal, L0), sort(L0, List)`,
  - here `sort/2` is a universal sorting procedure (see later), which
  - sorts the result list (and also filters repetitions).
- Example for using `setof/3`:

```
graph([a-b,a-c,b-c,c-d,b-d]).

% A vertex of Graph is V.
vertex(V, Graph) :- member(A-B, Graph), ( V = A ; V = B).

% The set of vertices of G is Vs.
graph_vertices(G, Vs) :- setof(V, vertex(V, G), Vs).

| ?- graph(_G), graph_vertices(_G, Vs). ⟹ Vs = [a,b,c,d] ? ; no
```

META-LOGIC PROCEDURES

Construction and deconstruction of structures: the *univ* procedure

- Call patterns for the *univ* procedure:
  - `+Expr =..  ?List`
  - `-Expr =..  +List`
- The meaning of the procedure: true, if
  - `Expr = `$Fun(A_1, \ldots, A_n)$ and `List = `$[Fun, A_1, \ldots A_n]$, where Fun is a name constant and $A_1, \ldots A_n$ are arbitrary terms; or
  - `Expr = `C and `List = `$[C]$, where C is a constant.
- Examples

```
| ?- el(a,b,10) =.. L.        ⟹  L = [el,a,b,10]
| ?- Expr =.. [el,a,b,10].    ⟹  Expr = el(a,b,10)
| ?- apple =.. L.              ⟹  L = [apple]
| ?- Expr =.. [1234].         ⟹  Expr = 1234
| ?- Expr =.. L.              ⟹  error
| ?- f(a,g(10,20)) =.. L.     ⟹  L = [f,a,g(10,20)]
| ?- Expr =.. [/,X,2+X].      ⟹  Expr = X/(2+X)
| ?- [a,b,c] =.. L.           ⟹  L = ['.',a,[b,c]]
```

Construction and deconstruction of structures: the `functor` procedure

- `functor/3`: determine functor of an expression, create expression of given functor
  - Call patterns: `functor(-Expr, +Name, +Argcnt)`
    `functor(+Expr, ?Name, ?Argcnt)`
  - Meaning: true, if `Expr` is an expression with functor `Name`/`Argcnt`.
    - If `Expr` is output, it is unified with the most generic expression of the given functor (distinct variables in its arguments).
- Examples:

```
| ?- functor(el(a,b,1), F, N).        ⟹   F = el, N = 3
| ?- functor(E, el, 3).               ⟹   E = el(_A,_B,_C)
| ?- functor(apple, F, N).            ⟹   F = apple, N = 0
| ?- functor(Expr, 122, 0).           ⟹   Expr = 122
| ?- functor(Expr, el, N).            ⟹   error
| ?- functor(Expr, 122, 1).           ⟹   error
| ?- functor([1,2,3], F, N).          ⟹   F = '.', N = 2
| ?- functor(Expr, ., 2).             ⟹   Expr = [_A|_B]
```

Construction and deconstruction of structures: the `arg` procedure

- `arg/3`: the given argument of an expression.
  - Call pattern: `arg(+Index, +Expr, ?Arg)`
  - Meaning: The `Index`th argument of the `Expr` structure is `Arg`.
  - Execution: `Arg` is **unified** with the specified argument.
  - Thus, the `arg/3` procedure can also be used for instantiating a variable argument of the structure (see 2nd example below).
- Examples:

```
| ?- arg(3, el(a, b, 23), Arg).    ⟹   Arg = 23
| ?- K=el(_,_,_), arg(1, K, a),
     arg(2, K, b), arg(3, K, 23). ⟹   K = el(a,b,23)
| ?- arg(1, [1,2,3], A).           ⟹   A = 1
| ?- arg(2, [1,2,3], B).           ⟹   B = [2,3]
```

- *univ* can be implemented using `functor` and `arg`, and vice versa, for example:

```
Expr =.. [F,A1,A2]   ⟺     functor(Expr, F, 2),
                           arg(1, Expr, A1), arg(2, Expr, A2)
```

Using *univ*: merging repetitive patterns

- The task: in a symbolic arithmetical expression, substitute all evaluable subexpressions with their values.
- 1st solution, without *univ*:

```
% The X X symbolic expression can be simplified as SX.
simpl0(X, SX) :-
    atomic(X), !, SX = X.
simpl0(U+V, SExp) :-
    simpl0(U, SU), simpl0(V, SV),
    calculate(SU+SV, SU, SV, SExp).
simpl0(U*V, SExp) :-
    simpl0(U, SU), simpl0(V, SV),
    calculate(SU*SV, SU, SV, SExp).
%...
% SUV built from SU and SV parts can be simplified as SExp.
calculate(SUV, SU, SV, SExp) :-
    number(SU), number(SV), !, SExp is SUV.
calculate(SUV, _, _, SUV).

| ?- deriv((x+y)*(2+x), x, D), simpl0(D, ED).
     ⟹ D = (1+0)*(2+x)+(x+y)*(0+1), ED = 1*(2+x)+(x+y)*1 ? ; no
```

Using *univ*: merging repetitive patterns (cont.)

- Expression simplification, 2nd solution, using *univ*

```
simpl(X, SX) :-
    atomic(X), !, SX = X.
simpl(Exp, SExp) :-
    Exp =.. [Op,U,V],     % Exp = Op(U,V)
    simpl(U, SU), simpl(V, SV),
    SUV =.. [Op,SU,SV],   % SUV = Op(SU,SV)
    calculate(SUV, SU, SV, SExp).
```

- Expression simplification, for arbitrary *ground* expressions:

```
simpl1(Exp, SExp) :-
    Exp =.. [O|ArgL], simpl1_list(ArgL, EArgL), SExp0 =.. [O|EArgL],
    % catch(:Goal,?Exc,:EGoal): if Goal throws an exception, runs EGoal:
    catch(SExp is SExp0, _, SExp = SExp0).

simpl1_list([], []).
simpl1_list([K|Kk], [E|Ek]) :-
    simpl1(K, E), simpl1_list(Kk, Ek).

| ?- simpl1(f(1+2+a, exp(3,2), a+1+2), E). ⟹ E = f(3+a,9.0,a+1+2)
```

Using *univ*: searching subexpressions

- The task: find all numbers in an arbitrary expression, and enumerate them along with their *selectors*!
- The selector of a subexpression is the list of the indices of the arguments in which it occurs, from the outside towards the inside.
- The $[i_1, i_2, \ldots, i_k]$ list selects from an `Expr` expression the i_1st argument's i_2nd argument's $\ldots i_k$th argument.
- For example, in `a*b+f(1,2,3)/c` the selector of `b` is `[1,2]`, the selector of `3` is `[2,1,3]`.

```
% expr_num(?Expr, ?N, ?Sel): N is a number in Expr with selector Sel.
expr_num(X, N, Sel) :-
        number(X), !, N = X, Sel = [].
expr_num(X, N, [I|Sel]) :-
        compound(X), % important to exclude variables!
        functor(X, _F, ArgNo), between(1, ArgNo, I), arg(I, X, X1),
        expr_num(X1, N, Sel).

| ?- expr_num(f(1,[b,2]), N, S).
====> S = [1], N = 1 ? ;
      S = [2,2,1], N = 2 ? ; no
```