Disjunctions – “happy” example

- Disjunctions (i.e. subgoals separated by “or”) can appear as goals
- A disjunction is denoted by semicolon (“;”), an `xfy` op. of priority 1100
- Comma (priority 1000) has tighter binding, e.g. `q, r; s` ≡ `(q , r) ; s`
- Enclose each disjunction in parentheses, align the characters `( ; )`

```
happy :-                        % I'm happy if
    (   workday,                % it's a workday and
        good_lecture            % I'm at a good lecture;
    ;   hot, swimming           % or it's hot and I'm swimming.
    ).
```

- Disjunctions are just “syntactic sugar”, they can be easily eliminated:

```
t(X, Z) :-                       t(X, Z) :- p(X, Y), aux(Y, Z), v(X, Z).
     p(X,Y),
     (   q(Y,U), r(U,Z)          aux(Y, Z) :- q(Y,U), r(U,Z).
     ;   s(Y, Z)           ⟹     aux(Y, Z) :- s(Y, Z).
     ;   t(Y), w(Z)              aux(Y, Z) :- t(Y), w(Z).
     ),
     v(X, Z).
```

The trace of “happy” with a disjunction

```
% happy: I'm happy.
happy :- % I'm happy if
    (   workday,
        % it's a workday and
        good_lecture
        % I'm at a good lecture;
    ;   hot,
        % or it's hot and
        swimming
        % I'm swimming.
    ).

workday.

swimming.

hot.

good_lecture :- fail.
```

```
| ?- happy.
   1      1 Call: happy ?
   2      2 Call: workday ?
   2      2 Exit: workday ?
   3      2 Call: good_lecture ?
   3      2 Fail: good_lecture ?
   4      2 Call: hot ?
   4      2 Exit: hot ?
   5      2 Call: swimming ?
   5      2 Exit: swimming ?
   1      1 Exit: happy ?
              yes
```

Negation by failure

- As a modification of the previous variant of “happy” consider a person who, on workdays, is happy only if attending a good lecture.
- Thus the condition “It isn’t a workday” has to appear in the 2nd disjunct.
- This can be achieved using the “\+” construct, called negation by failure:

```
happy  :-                % I'm happy if
    (   workday,         % it's a workday and
        good_lecture     % I'm attending a good lecture;
    ;   \+ workday,      % or it isn't a workday and
        hot, swimming    % it's hot and I'm swimming.
    ).
```

- The goal “`\+ G`” is executed by first executing `G`. If this fails “`\+ G`” succeeds, otherwise it fails.
- Read “`\+`” as “not provable”, cf. $\nvdash$ tilted slightly to the left.
- Negation by failure has its limitations, to be discussed soon.

Executing the variant of “happy” with negation

- As the “\+” construct is not shown in the trace, an auxiliary predicate `not_a_workday` is introduced, to make the effect of “\+” visible:

```
happy :-
    (   workday, good_lecture
    ;   not_a_workday, hot, swimming
    ).
                            | ?- happy.
workday.                            1      1 Call: happy ?
                                    2      2 Call: workday ?
swimming.                           2      2 Exit: workday ?
                                    3      2 Call: good_lecture ?
hot.                                3      2 Fail: good_lecture ?
                                    4      2 Call: not_a_workday ?
% It isn't a workday.               5      3 Call: workday ?
not_a_workday :-                    5      3 Exit: workday ?
    \+ workday.                     4      2 Fail: not_a_workday ?
                                    1      1 Fail: happy ?
                            no
```

- Note that predicate `workday` is called twice (calls number 2 and 5).

The if-then-else construct

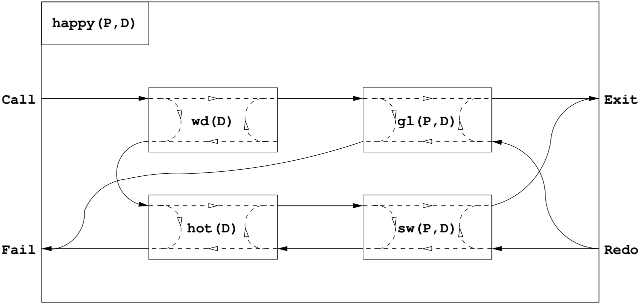
- When the two branches of a disjunction exclude each other, use the if-then-else construct `( condition -> then-branch ; else-branch )`:

```
happy :-                                    happy :-
    (   workday,                                (   workday ->
        good_lecture                                good_lecture
    ;   \+ workday,          ==>                ;
        hot, swimming                               hot, swimming
    ).                                          ).
```

- The atom `->` is a standard operator, of type `xfy` and priority 1050
- The construct `( Cond -> Then ; Else )` is executed by first executing `Cond`. If this succeeds, `Then` is executed, otherwise `Else` is executed. **Important**: Only the first solution of `Cond` is used for executing `Then`. The remaining solutions are discarded!
- Note that `( Cond -> Then ; Else )` looks like a disjunction, but it is not
- The else-branch can be omitted, it defaults to `false`.

The procedure box for if-then-else

```
happy(P, D) :-
    (   wd(D) -> gl(P, D)
    ;   hot(D), sw(P, D)
    ).
```



- The 2nd etc. solutions of `wd` are not produced (cf. the dangling Redo port of `wd`).
- With uninstantiated vars in the condition, *if-then-else* may not work as expected.

The if-then-else construct (contd.)

- If-then-else can be transformed to a disjunction with a negation:

```
(   cond -> then                  (   cond, then
;   else                 ⟹        ;   \+ cond, else
)                                 )
```

  These are equivalent only if `cond` succeeds at most once.
  The if-then-else is more efficient (no choice point left).

- Negation can be fully defined using if-then-else

```
                                  (   p -> false
\+ p                     ≡        ;   true
                                  )
```

- The semicolon binds to the right, preferably avoid nested parentheses when making multiple if-then-else branches:

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

Pitfalls of Negation by Failure – declarative reading

- Given some facts find an employer who is not an employee.

```
% emp(Employer, Employee): Employer employs Employee.
emp(a, b).                                                      (f1)
emp(a, c).                                                      (f2)
emp(d, a).                                                      (f3)

| ?- emp(E, X), \+ emp(Y, E).  ⟹  E =  d ? ; no                 (q1)
```

- The meaning of query `(q1)`: $(\exists \mathtt{X}.\mathtt{emp}(\mathtt{E},\mathtt{X})) \land (\neg\exists \mathtt{Y}.\mathtt{emp}(\mathtt{Y},\mathtt{E}))$
- What happens when the two calls are switched?

`| ?- \+ emp(Y, E), emp(E, X).` ⟹ `no` { irresp. of the `emp/2` facts} (q2)

- Prolog first calls G = `emp(Y, E)`. Since both arguments are unbound, this succeeds if there is at least one `emp/2` fact. ⟹ `\+`G fails.
- Thus the meaning of query `(q2)`: $(\neg\exists \mathtt{Y},\mathtt{E}.\mathtt{emp}(\mathtt{Y},\mathtt{E})) \land (\exists \mathtt{X}.\mathtt{emp}(\mathtt{E},\mathtt{X}))$
- The meaning of `\+`G depends on which variables of G are unbound!
- In general: the meaning of `\+` G: $\neg\exists X_1,\ldots X_n G$, where X_i are the unbound variables in G at the time of invocation.

Pitfalls of Negation by Failure – open and closed worlds

- Mathematical logic uses the open world assumption (OWA)
  - A statement *S* follows from a set of statements *SS* (premises), if *S* holds in any world (interpretation) that satisfies *SS*.
  - ¬ `emp(_, d)` is not a logical consequence of the facts `(f1)-(f3)`.
  - (But, one can still deduce ¬ `emp(_, d)` using a rule: Those receiving unemployment benefit are not employed by anyone)
- Negation in database queries (and `\+` in Prolog) uses closed world assumption (CWA)
  - a single world is considered in which the given facts, and only these are true
  - when something cannot be proved, it is considered false
- Classical logic with OWA is monotonic:
  - the more you know, the more you can deduce
- Negation by failure (CWA) is non-monotonic:
  - Add the fact "`emp(b, d).`" to `(f1)-(f3)` and query `(q1)` will fail

Pitfalls of if-then-else

- Can a predicate involving if-then-else be used in multiple I/O modes?

```
happy(P, D) :-  (   workday(D) -> good_lecture(P, D)
                ;   hot(D), swimming(P, D)
                ).
```

- It can be used in mode `happy(?, +)`, but not in mode `happy(?, -)`
- Reason: `workday(-D)` will bind `D` to the first workday only! Workarounds:

```
happy1(P, D) :- day(D),                 % ≡ member(D, [mon,...,sun])
                happy(P, D).

happy2(P, D) :- (  workday(D), good_lecture(P, D)
                ;  weekend_day(D),  % ≡ member(D, [sat,sun])
                   hot(D), swimming(P, D)
                ).
```

- Don’t use unbound vars in IF conditions unless for expressing “there is”:

```
employer(E) :-                        % E is an employer if
    (   emp(E, _X) -> true   ).  % there is an _X such that emp(E, _X)
```

- For facts `(f1)-(f3)` `employer(a)` succeeds once, while `emp(a,_)` twice!
- The control BIP `once/1` does exactly this: `employer(E) :- once(emp(E,_)).`

The cut – the BIP underlying if-then-else and negation

- The cut, denoted by `!` is a BIP with no arguments, i.e. its functor is `!/0`.

```
happy(P, D) :-          workday(D), !, good_lecture(P, D).          (1)
happy(P, D) :-          hot(D), swimming(P, D).                     (2)
```

- Execution: succeeds unconditionally, but has the following side effects:
  - Restrict to first solution:
    Remove all choice points created within the goals preceding the cut.
  - ***Commit to clause***:
    Remove the choice of any further clauses in the current predicate.
- Definition: if `q :- ..., p, ....` then
  the parent goal of `p` is the goal matching the clause head `q`
- Effects of cut in the goal reduction model: removes all choice points up to and including the node labelled with the query parent goal of the cut, . . .
- In the procedure box model: Fail port of cut ⟹ Fail port of parent.
- The behaviour is identical to the following if-then-else:

```
happy(P, D) :-            (    workday(D) -> good_lecture(P, D)
                          ;    hot(D), swimming(P, D)
                          ).
```

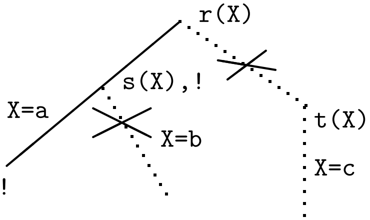
- In fact, SICStus transforms this to the predicate `(1)`–`(2)` above

Choicepoints removed by the cut – pruning the search tree

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

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

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



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

Variants of cut

- An example: `firstp(+L, -FP)`: `FP` is the first positive element in list `L`
  - The computer scientist's solution:

```
firstp_green([X|_], X)  :- X > 0, !.                         (1)
firstp_green([X|L], FP) :- X =< 0, firstp_green(L, FP).      (2)
```

  - The Prolog hacker's (relies on the enumeration order of `member/2`):

```
firstp_hacker(L, FP) :-  member(FP, L), FP > 0, !.           (3)
```

- The green cut: *we* know that there are no solutions, but the Prolog implementation does not – semantically "harmless".
  - `X > 0` $\equiv \neg$ `X =< 0`, but Prolog does not "know" this
  - When a green cut is removed the set of solutions stays the same, but the program may become less (space and time)-efficient
- The red cut: we throw away solutions on purpose.
  - In `(3)` we throw away all solutions but the first
  - Also, a green cut may become red when an "unnecessary" condition, such as `X =< 0`, is removed
  - When a red cut is removed, the set of solutions changes.

The dangers of using the cut – the base rule

- Consider `fp0` with the "unnecessary" `X=<0` , and `fp1` without it:

```
fp0([X|_], X) :- X > 0,  !.         fp1([X|_], X) :- X > 0, !.      (1)
fp0([X|L], Y) :- X=<0, fp0(L, Y).  fp1([X|L], Y) :- fp1(L, Y).     (2)
```

- In mode `(+,-)`, `fp0` and `fp1` behave the same way. But:

```
| ?- Z = 2, fp0([1,2], Z).    ⟹  no
| ?- Z = 2, fp1([1,2], Z).    ⟹  yes  % (1)  does not match,
                                       % (2)  ⟹  fp1([2],2)
| ?- fp1([1,2], Z), Z = 2.    ⟹  no   % fp1  is not steadfast
```

- Definition: `p(+,?)` is steadfast iff "`p(foo, X), X = bar`" is equivalent to "`X = bar, p(foo, X)`"
- Rewrite `(1)` to notice that output arg. unification is part of the condition:

```
fp1([X|_], Y)  :- Y = X, X > 0, !.                              (1*)
```

- The base rule of cut: **unify output arguments after the cut!**
- Steadfast version, which observes the base rule and is faster:

```
                                            fp3([X|L], Y) :-
fp2([X|_], Y) :- X > 0, !, Y = X.               (   X > 0 -> Y = X
fp2([_|L], Y) :- /*X=<0*/, fp2(L, Y).           ;   /*X=<0*/, fp3(L, Y) )
                                                )
```

- If in doubt, use if-then-else instead of cut.

Contents

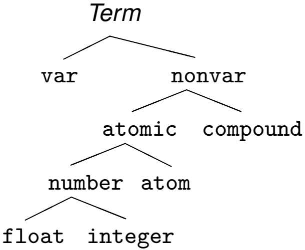
1 Declarative Programming with Prolog
- Declarative and imperative programming
- Propositional Prolog
- Prolog with Simple Data Structures
- Compound Data Structures in Prolog
- Lists
- Prolog implementation – a brief overview
- Prolog execution – definitions
- Prolog syntax
- Syntactic sugar: operators
- Further control constructs
- BIPs 1 – meta-preds, all solutions, dynamic preds
- BIPs 2 – higher order programming, loops, modules
- Efficient programming in Prolog

Built-in predicates – batch 1

- Meta-predicates
  - term classification: `var(X)`, `number(X)`, ...
  - composition and decomposition of compound terms: a compound ⇔ name + arguments
  - composition and decomposition of atoms and numbers: an atom or a number ⇔ list of characters
  - universal term comparison: comparing arbitrary Prolog terms
- All-solutions predicates: finding all solutions of a goal
- Dynamic predicates: adding and removing program clauses from within a running Prolog program

Classification of terms

- Classification BIPs ⇔ nodes of the Prolog term hierarchy (recap)



`var(X)`	`X` is a variable
`nonvar(X)`	`X` is not a variable
`atomic(X)`	`X` is a constant (atom or number)
`compound(X)`	`X` is a compound
`number(X)`	`X` is a number
`atom(X)`	`X` is an atom
`float(X)`	`X` is a floating point number
`integer(X)`	`X` is an integer

- Some further SICStus-specific (non-standard) classification predicates:
  - `simple(X)`: `X` is a non-compound term (i.e., constant or variable);
  - `ground(X)`: `X` is ground, i.e. contains no unbound variables
- All the above BIPs test the current state of the argument
  - E.g. `number(X)` checks that `X` is currently a number, rather than imposing a constraint that `X` has to be a number.

Building and decomposing compounds: the *univ* predicate

- BIP `=.. /2` (pronounce *univ*) is a standard op. (`xfx`, 700; just as =, …)
- `Term =.. List` holds if
  - `Term =` $Fun(A_1, \ldots, A_n)$ and `List =` $[Fun, A_1, \ldots\ \ A_n]$,
    where Fun is an atom and $A_1, \ldots\ \ A_n$ are arbitrary terms; or
  - `Term =` C and `List =` $[C]$, where C is a constant.
    (Constants are viewed as compounds with 0 arguments.)
- `X = F(A1, ..., An)` ⟹ syntax error, use `X =.. [F,A1,...,An]` instead
- Call patterns for *univ*:
  - `+Term =.. ?List` – decomposing `Term`
  - `-Term =.. +List` – constructing `Term`
- Examples

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

Building and decomposing compound structures: `functor/3`

- `functor(Term, Name, Arity)`:
  `Term` has the name `Name` and arity `Arity`, i.e.
  `Term` has the functor `Name/Arity`.

  (A constant `C` is considered to have the name `C` and arity 0.)
  - Call patterns:

    `functor(+Term, ?Name, ?Arity)` – decompose `Term`

    `functor(-Term, +Name, +Arity)` – construct a most general `Term` (*)
  - If `Term` is output (*), it is unified with the most general term with the given name and arity (with distinct new variables as arguments)
- Examples:

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

Building and decomposing compounds: `arg/3`

- `arg(N, Compound, A)`: the `N`th argument of `Compound` is `A`
  - Call pattern: `arg(+N, +Compound, ?A)`
  - Execution: The `N`th argument of `Compound` is **unified** with `A`. If `Compound` has less than `N` arguments, or `N = 0`, `arg/3` fails
  - Thus `arg/3` can also be used for instantiating a variable argument of the structure (as in the second example below).
- Examples:

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

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

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

Using *univ* for simplifying an earlier example

- Polynomials: built from numbers and the atom ‘`x`’, using ops ‘+’ and ‘$*$’
- Calculate the value of a polynomial for a given substitution of `x`

```
% value_of(Poly, X, V): Poly has the value V, if x=X
value_of(x, X, V) :-              value_of1(x, X, V) :-
    V = X.                            V = X.
value_of(Poly, _, V) :-           value_of1(Poly, _, V) :-
    number(Poly), V = Poly.           number(Poly), V = Poly.
value_of(P1+P2, X, V) :-
    value_of(P1, X, V1),
    value_of(P2, X, V2),
    V is V1+V2.
value_of(Poly, X, V) :-           value_of1(Poly, X, V) :-
    Poly = P1*P2,                     Poly =.. [Func,P1,P2],
    value_of(P1, X, V1),              value_of1(P1, X, V1),
    value_of(P2, X, V2),              value_of1(P2, X, V2),
    VPoly = V1*V2,                    VPoly =.. [Func,V1,V2],
    V is VPoly.                       V is VPoly.
```

- Predicate `value_of1` works for all binary functions supported by `is/2`.

```
| ?- value_of1(exp(100,min(x,1/x)), 2, V).     ⟶     V = 10.0 ? ; no
```

Using *univ* for finding subexpressions (ADVANCED)

- Given a term $\texttt{T}_0$ with a (not necessarily proper) subterm $\texttt{T}_n$ at depth n, the position of $\texttt{T}_n$ within $\texttt{T}_0$ is described by a *selector* $[\texttt{I}_1,\ldots,\texttt{I}_n]$ ($n \geq 0$):

$$\texttt{select\_subterm}(\texttt{T}_0,\ [\texttt{I}_1,\ldots,\texttt{I}_n],\ \texttt{T}_n)\ \texttt{:-}$$
$$\texttt{arg}(\texttt{I}_1,\ \texttt{T}_0,\ \texttt{T}_1),\ \texttt{arg}(\texttt{I}_2,\ \texttt{T}_1,\ \texttt{T}_2),\ \ldots,\ \texttt{arg}(\texttt{I}_n,\ \texttt{T}_{n-1},\ \texttt{T}_n).$$

- E.g. within term `a*b+f(1,2,3)/c`, `[1,2]` selects `b`, `[2,1,3]` selects `3`.
- Given a term, enumerate number subterms and their *selectors*.

```
% number_subterm(?Term, ?N, ?Sel):
% N is a number which occurs as a subterm in Term at position Sel.
number_subterm(X, N, Sel) :-
    number(X), !, N = X, Sel = [].
number_subterm(X, N, [I|Sel]) :-
    compound(X),      % it is important to exclude variables!
    X =.. [_|L],
    nth1(I, L, Y), % The Ith element of list L is Y.
                   % If L is proper, finitely enumerates I and Y.
                   % Defined in library(lists).
    number_subterm(Y, N, Sel).
```

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

Decomposing and building atoms

- `atom_codes(Atom, Cs)`: `Cs` is the list of character codes comprising `Atom`.
  - Call patterns: `atom_codes(+Atom, ?Cs)`
    `atom_codes(-Atom, +Cs)`
  - Execution:
    - If `Cs` is a proper list of character codes then `Atom` is unified with the atom composed of the given characters
    - Otherwise `Atom` has to be an atom, and `Cs` is unified with the list of character codes comprising `Atom`
- `atom_chars(Atom, Chs)`: `Chs` is the list of characters (single character atoms) comprising `Atom`.
- Examples:

```
| ?- atom_codes(ab, Cs).              ⟹   Cs = [97,98]
| ?- atom_chars(ab, Cs).              ⟹   Cs = [a,b]
| ?- atom_codes(ab, [0'a|L]).         ⟹   L = [98]
| ?- Cs="bc", atom_codes(Atom, Cs).   ⟹   Cs = [98,99], Atom = bc
| ?- atom_codes(Atom, [0'a|L]).       ⟹   error
```

Decomposing and building numbers

- `number_codes(Number, Cs)`: `Cs` is the list of character codes of `Number`.
  - Call patterns: `number_codes(+Number, ?Cs)`
    `number_codes(-Number, +Cs)`
  - Execution:
    - If `Cs` is a proper list of character codes which is a number according to Prolog syntax, then `Number` is unified with the number composed of the given characters
    - Otherwise `Number` has to be a number, and `Cs` is unified with the list of character codes comprising `Number`
- `number_chars(Number, Chs)`: `Chs` is the list of characters comprising `Number`.
- Examples:

```
| ?- number_codes(12, Cs).          ⟹   Cs = [49,50]
| ?- number_chars(12, Cs).          ⟹   Cs = ['1','2']
| ?- number_codes(0123, [0'1|L]).   ⟹   L = [50,51]
| ?- number_codes(N, " - 12.0e1").  ⟹   N = -120.0
| ?- number_codes(N, "12e1").       ⟹   error (no decimal point)
| ?- number_codes(120.0, "12e1").   ⟹   no (The first arg. is given :-)
```

Ordering all Prolog terms

- Each Prolog term belongs to one of the five classes: `var`, `float`, `integer`, `atom`, `compound` (cf. the leaves of the Prolog term hierarchy, page 105)
- The relation "precedes" $X \prec Y$ is defined as follows:
  1. If X and Y belong to different classes, then their class determines the order, as listed above (e.g. all floats $\prec$ all integers); otherwise
  2. If X and Y are variables, then their order is system-dependent (normally variables are ordered according to their memory address)
  3. If X and Y are numbers, then $X \prec Y \Leftrightarrow X < Y$
  4. If X and Y are atoms, then $X \prec Y \Leftrightarrow$ either X is a proper prefix of Y, or $X_i < Y_i$ where i is the index of the first different char, (A_i is the code of the ith char of A)
  5. If both X and Y are compounds:
     1. If their arities differ, $X \prec Y \Leftrightarrow X$'s arity $<$ Y's arity
     2. Otherwise (same arity), if their names differ, $X \prec Y \Leftrightarrow N_X \prec N_Y$ (N_A is the name of the compound A)
     3. Otherwise (same name and arity): $X \prec Y \Leftrightarrow X_i \prec Y_i$ where i is the index of the first non-identical argument, (A_i is the ith argument of the compound A)

Built-in predicates for comparing Prolog terms

- Comparing two Prolog terms:

Goal	holds if
`Term1 == Term2`	$\mathtt{Term1} \not\prec \mathtt{Term2} \wedge \mathtt{Term2} \not\prec \mathtt{Term1}$
`Term1 \== Term2`	$\mathtt{Term1} \prec \mathtt{Term2} \vee \mathtt{Term2} \prec \mathtt{Term1}$
`Term1 @< Term2`	$\mathtt{Term1} \prec \mathtt{Term2}$
`Term1 @=< Term2`	$\mathtt{Term2} \not\prec \mathtt{Term1}$
`Term1 @> Term2`	$\mathtt{Term2} \prec \mathtt{Term1}$
`Term1 @>= Term2`	$\mathtt{Term1} \not\prec \mathtt{Term2}$

- The comparison predicates are not pure:

```
| ?- X @< 3, X = 4. ⟹  X = 4
| ?- X = 4, X @< 3. ⟹  no
```

- Comparison uses, of course, the canonical representation:

```
| ?- [1, 2, 3, 4] @< s(1,2,3).   ⟹
```
yes (rule 5.1)

Equality-like Prolog predicates – a summary

- `U = V`: *U* unifies with *V*
  No errors.

```
| ?- X = 1+2.      ⟹  X = 1+2
| ?- 3 = 1+2.      ⟹  no
```

- `U == V`: *U* is identical to *V*.
  No errors, no bindings.

```
| ?- X == 1+2.     ⟹  no
| ?- 3 == 1+2.     ⟹  no
| ?- +(1,2)==1+2   ⟹  yes
```

- `U =:= V`: The value of *U* is equal to that of *V*.
  No bindings. Error if *U* or *V* is not a (ground) arithmetic expression.

```
| ?- X =:= 1+2.    ⟹  error
| ?- 1+2 =:= X.    ⟹  error
| ?- 2+1 =:= 1+2.⟹  yes
| ?- 2.0 =:= 1+1.⟹  yes
```

- `U is V`: *U* is unified with the value of *V*.
  Error if *V* is not a (ground) arithmetic expression.

```
| ?- 2.0 is 1+1.   ⟹  no
| ?- X is 1+2.     ⟹  X = 3
| ?- 1+2 is X.     ⟹  error
| ?- 3 is 1+2.     ⟹  yes
| ?- 1+2 is 1+2.   ⟹  no
```

- (`U =.. V`: The “decomposition” of term *U* is the list *V*).

```
| ?- 1+2 =.. X.    ⟹  X = [+,1,2]
| ?- X =.. [f,1].⟹  X = f(1)
```

Nonequality-like Prolog predicates – a summary

- Nonequality-like Prolog predicates **never** bind variables.

- *U* `\=` *V*: *U* does not unify with *V*.
  No errors.

```
| ?- X \= 1+2.         ⟹  no
| ?- +(1,2) \= 1+2.    ⟹  no
```

- *U* `\==` *V*: *U* is not identical to *V*.
  No errors.

```
| ?- X \== 1+2.        ⟹  yes
| ?- 3 \== 1+2.        ⟹  yes
| ?- +(1,2)\==1+2      ⟹  no
```

- *U* `=\=` *V*: The values of the arithmetic expressions *U* and *V* are different.
  Error if *U* or *V* is not a (ground) arithmetic expression.

```
| ?- X =\= 1+2.        ⟹  error
| ?- 1+2 =\= X.        ⟹  error
| ?- 2+1 =\= 1+2.      ⟹  no
| ?- 2.0 =\= 1+1.      ⟹  no
```

(Non)equality-like Prolog predicates – examples

		Unification		*Identical terms*		*Arithmetic*		
U	*V*	`U = V`	`U \= V`	`U == V`	`U \== V`	`U =:= V`	`U =\= V`	`U is V`
`1`	`2`	*no*	`yes`	*no*	`yes`	*no*	`yes`	*no*
`a`	`b`	*no*	`yes`	*no*	`yes`	error	error	error
`1+2`	`+(1,2)`	`yes`	*no*	`yes`	*no*	`yes`	*no*	*no*
`1+2`	`2+1`	*no*	`yes`	*no*	`yes`	`yes`	*no*	*no*
`1+2`	`3`	*no*	`yes`	*no*	`yes`	`yes`	*no*	*no*
`3`	`1+2`	*no*	`yes`	*no*	`yes`	`yes`	*no*	`yes`
`X`	`1+2`	`X=1+2`	*no*	*no*	`yes`	error	error	`X=3`
`X`	`Y`	`X=Y`	*no*	*no*	`yes`	error	error	error
`X`	`X`	`yes`	*no*	`yes`	*no*	error	error	error

Legend: `yes` – success; *no* – failure.

Finding multiple solutions: enumeration vs. collection

- Search problem: find values satisfying certain conditions.
- Two approaches to solving search problems in Prolog:
  - collect solutions – e.g., return a list of all solutions;
  - enumerate solutions – return one solution at a time, enumerate all solutions via backtracking
- A simple example: find the even members of a list:

Collect solutions:

```
% 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).
```

Enumerate solutions:

```
% even_member(_L, E): E is an even
% member of the list L.
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).

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

Collecting and enumerating solutions

- Given a “collecting” predicate, write an “enumerating” one:
  - Use the `member/2` built-in predicate, e.g.:

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

    This is less efficient than directly implementing `even_member/2`.
- Given an “enumerating” predicate, write a “collecting” one:
  - Not possible with the tools shown so far
  - A new kind of BIP, an “all-solutions” predicate is needed, e.g.

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

  - All-solutions predicates often help in making the code very compact (but the result may be less efficient than the code written directly)

    ```
    even_members(L, Es) :-
        findall( E, (member(E, Es), E mod 2 =:= 0), Es).
    %          { E | member(E, Es), E mod 2 =:= 0} =Es)
    ```

The built-in predicate `findall(?Templ, :Goal, ?L)`

Approximate meaning: `L` is a list of `Templ` terms for all solutions of `Goal`[6]

Examples[7]

```
| ?- 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 execution of the BIP `findall/3` (procedural semantics);

- Interpret term `Goal` as a goal, and call it
- For each solution of `Goal`:
  - store a *copy* of `Templ` (copy ⟹ replace vars in `Templ` by new ones)
  - continue with failure (to enumerate further solutions)
- When there are no more solutions (`Goal` fails)
  - collect the stored `Templ` values into a list, unify it with `L`.

```
| ?- findall(T, member(T, [A-A,B-B,A]), L). ⟹ L= [_A-_A,_B-_B,_C] ? ; no
```

---

[6] annotation “`:`” marks a meta argument, i.e. a term to be interpreted as a goal

[7] Predicate `between(+N, +M, ?X)` enumerates in `X` the integers `N`, `N+1`, . . . , `M`. Defined in `library(between)`.

The built-in predicate `findall` – further details

- Example: collect employees

```
% emp(R, E): employer R employs employee E.
emp(a,b).   emp(a,c).   emp(b,c).   emp(c,d).   emp(b,d).
```

```
| ?- findall(E, emp(R, E), Employees).                         (1)
             ⟹  Employees = [b,c,c,d,d] ? ; no
```

i.e. `Employees` = $\{E \mid \exists\ R.\ (R \ \text{employs}\ \ E)\}$

```
| ?- R = a, findall(E, emp(R, E), Employees).                  (2)
             ⟹  Employees = [b,c] ? ; no
```

i.e. `Employees` = $\{E \mid \ (R \ \text{employs}\ \ E)\}$

```
| ?- findall(E, emp(R, E), Employees), R = a.                  (3)
             ⟹  Employees = [b,c,c,d,d] ? ; no      % findall is not pure
```

- The declarative meaning of `findall(?Templ, :Goal, ?List)`:
  `List` $=$ { a copy of `Templ` | $(\exists$`X`$\ldots$`Z`$)$ `Goal` is true }
  where `X`, ..., `Z` are the free variables in the `findall` call.
- A variable is *free* in a `findall(Templ, Goal, List)` call, if it occurs in `Goal` but not in `Templ`. E.g. `R` is free in the `findall` goals `(1)` and `(3)`, but not in `(2)`.

An example illustrating BIP `bagof/3`

```
emp(a,b).   emp(a,c).   emp(b,c).   emp(c,d).   emp(b,d).
```

```
| ?- bagof(E, emp(R, E), L). % L ≡ list of E's employed by given R.
           ⟹  R = a, L = [b,c] ? ;
           ⟹  R = b, L = [c,d] ? ;
           ⟹  R = c, L = [d] ? ; no
```

Execution details

- Collect the list of free variables: `FreeVars = [R]`, `Templ = E`,
- For each solution store a copy of `FreeVars` and `Templ`

FreeVars	**Templ**
[a]	b
[a]	c
[b]	c
[c]	d
[b]	d

- Collect the distinct `FreeVars` instances: `[a], [b], [c]`
- Enumerate these instances: `FreeVars=[R]= [a];   [b];   [c]`
- For each `FreeVars` collect `Templ` values: `Employees=   [b,c]; [c,d]; [d]`

The BIP `bagof(?Templ, :Goal, ?L)` – semantics

The execution of the BIP (procedural semantics):

- Collect the `FreeVars` list of free variables in the `bagof` goal
- Interpret term `Goal` as a goal, and call it; for each solution of `Goal`
  - store a normalised copy of the pair ⟨ `FreeVars`, `Templ` ⟩
    - normalisation: rename any vars in `FreeVars` to $X_1, \ldots, X_n, \ldots$ (in the order of the first occurrences of the vars)
  - continue with failure (so as to enumerate further solutions)
- When there are no more solutions (i.e. `Goal` fails)
  - fail, if there are no stored copies; otherwise
  - collect the `FreeVars` instances distinct wrt. `==`
  - enumerate in `FreeVars` the distinct instances (in some order)
  - for a given `FreeVars` instance collect the list of corresponding `Templ` values, and unify it with `L`.

The meaning of the BIP (declarative semantics):

- $\mathtt{L} = \{\mathtt{Templ} \mid \mathtt{Goal}\ \text{is true}\ \}$, $\mathtt{L} \neq \mathtt{[]}$.

An example illustrating that `bagof/3` is the “inverse” of `member/2`

`| ?- bagof(T, member(T, [A-A,B-B,A]), L).` ⟹ `L=[A-A,B-B,A] ? ; no`

Execution details

- Collect the list of free variables: `FreeVars = [A,B]`, `Templ = T`,
- For each solution store a normalised copy of `FreeVars` and `Templ`

norm. `FreeVars`	`Templ`
$[X_1,X_2]$	X_1-X_1
$[X_1,X_2]$	X_2-X_2
$[X_1,X_2]$	X_1

- The normalised `FreeVars` instances are all identical
- “Enumerate” the only `FreeVars` instance:
  `FreeVars = [A,B] =` $[X_1,X_2]$, i.e. X_1 `= A`, X_2 `= B`
- For the single `FreeVars` collect the `Templ` values:
  `L =` $[X_1-X_1,X_2-X_2,X_1]$ `= [A-A,B-B,A]`

The built-in predicate `bagof` – explicit quantification

- Explicit existential quantification can be added to a `bagof` call:

```
| ?- bagof(E, R^emp(R, E), L).
     % L≡list of E's for which
     % there exists an R, such that emp(R, E)
                         ⟹ L = [b,c,c,d,d] ? ; no
```

- In general explicit quantification takes the following form:

  `bagof(Templ,` $\mathtt{V_1\hat{}\ldots\hat{}V_n\hat{}}$ **`Goal, List)`**

  - variables $\mathtt{V}_1, \ldots, \mathtt{V}_n$ are existentially quantified,
  - i.e., not considered free any more.

- The declarative semantics of the above goal:

  $\mathtt{List} = \{\ \mathtt{Templ} \mid (\exists \mathtt{V1}, \ldots, \mathtt{Vn})\mathtt{Goal}$ is true $\} \neq \mathtt{[]}$.

Nesting `bagof/3`

- If a `bagof` call has free variables then it can be nondeterministic
- Thus it may make sense to nest `bagof` calls within each other

```
% Employer R has C employees.
employee_count(R, C) :-
        bagof(E, emp(R, E), Es), length(Es, C).

% The employee-counts list RCL is the list of R-C pairs, where
% R is an employer and C is the number of its employees
employee_counts(RCL) :-
        bagof(R-C, employee_count(R, C), RCL).

| ?- employee_counts(RCL).
                ⇒ RCL = [a-2,b-2,c-1] ? ; no
```

- The helper predicate `employee_count` can be eliminated:

```
employee_counts2(RCL) :-
    bagof(R-C, Es^(bagof(E, emp(R, E), Es),
                   length(Es, C)           ), RCL).
```

- Note the need for the explicit quantification
- Also note that the latter predicate is slower, as control structures in meta-arguments are interpreted and not compiled

The built-in predicate `bagof` – further details

- Further minor differences between `bagof/3` and `findall/3`:

```
| ?- findall(X, emp(d, X), L).   ⟹  L = [] ? ; no
| ?-   bagof(X, emp(d, X), L).   ⟹  no
```

- Summary: `bagof/3` is cleaner than `findall/3`, but it is less efficient.

The built-in predicate `setof`

- `setof(?Templ, :Goal, ?List)`
- The execution of the procedure:
    - same as: `bagof(Templ, Goal, L0), sort(L0, List),`
    - here `sort(+L, ?SL)` is a built-in predicate which sorts `L` and removes duplicates (wrt. ==) and unifies the result with `SL`
- 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
```

Dynamic predicates

- Dynamic predicates are Prolog predicates, with the following properties
  - The predicate can be modified during runtime by adding (asserting) and removing (retracting) clauses
  - There can be 0 or more clauses of the predicate in the program text
  - The predicate is interpreted (slower execution)
- A dynamic predicate can be created
  - by placing a directive in the program: `:- dynamic(Predicate/Arity).` (preceding any clauses of the predicate in the program text); or
  - by using a database modification BIP[8]
- Built-in predicates for database modification
  - Add a clause: `asserta/1, assertz/1`
  - Remove a clause (can be nondeterministic): `retract/1`
  - Retrieve a clause (can be nondeterministic): `clause/2`
- Adding or removing clauses is permanent, this is not undone at backtracking.

---

[8] The set of program clauses is often called the Prolog database.

Adding a clause: `asserta/1`, `assertz/1`

- `asserta(:Clause)`[9]
  - the term `Clause` is interpreted as a clause, it has to be sufficiently instantiated for its functor `P/N` to be to determined
  - If pred. `P/N` exists, it has to be dynamic, if not, it is made dynamic
  - a copy of `Clause` is added to pred. `P/N` as the first clause

  By copying we mean systematically replacing variables with new ones.
- `assertz(:Clause)`
  - Same as `asserta`, but `Clause` is added as the last clause
- Most Prolog systems support the non-standard BIP `assert/1`, which adds a clause in an arbitrary position in the predicate (mostly ≡ `assertz/1`)
- Examples:

```
| ?- assertz((p(1,X):-q(X))), asserta(p(2,0)),      p(2, 0).
     assertz((p(2,Z):-r(Z))), listing(p).      ⟹   p(1, A) :-   q(A).
                                                    p(2, A) :-   r(A).

| ?- assertz(s(X,X)), s(U,V), U == V, X \== U.  ⟹   V = U ? ; no
```

---

[9] Recall that the `:` character indicates that the argument is a meta-argument.

Removing a clause: `retract/1`

- `retract(:Clause)` where `Clause` viewed as a clause is sufficiently instantiated so that its functor `P/N` can be determined:
  - looks up a clause of pred. `P/N` which unifies with `Clause`;
  - if found (and unified), removes the clause from the program;
  - on backtracking keeps looking up and removing further clauses
- Example (continued from the previous slide):

```
| ?- listing(p), retract((p(2,X):-B)),
     assertz((s(3,X):-B)), listing(p), listing(s), fail. ⟹  no
```

- The output

```
p(2, 0).             | p(1, A) :-           | p(1, A) :-
p(1, A) :-           |           q(A).      |         q(A).
        q(A).        | p(2, A) :-           |
p(2, A) :-           |           r(A).      | s(3, 0).
        r(A).        |                      | s(3, A) :-
                     | s(3, 0).             |         r(A).
```

An example – a simplified `findall`

- Predicate `findall1/3` implements the BIP `findall/3`, except for not supporting nested invocations

```
:- dynamic(solution/1).

% findall1(T, Goal, L):
% L is the list of copies of T, for each solution of Goal
findall1(T, Goal, _L) :-
    call(Goal),
    asserta(solution(T)), %  solutions stored in reverse order!
    fail.
findall1(_Templ, _Goal, L) :-
    solution_list([], L).

% solution_list(L0, L): L  = rev(list of retracted solutions) ⊕ L0
solution_list(L0, L) :-
    retract(solution(S)), !,
    solution_list([S|L0], L).
solution_list(L, L).

| ?- findall1(Y, (member(X, [1,2,3]),Y is X*X), SL). ⟹  SL = [1,4,9]
```

Retrieving a clause: `clause/2`

- `clause(:Head, ?Body)` where `Head` is instantiated sufficiently so that its functor `P/N` can be determined
  - looks up a clause of pred. `P/N` which unifies with `(Head :- Body)`[10]
  - if found exits with success (having performed the unification);
  - on backtracking keeps looking up further clauses
- Example (continued from previous slides)

```
:- listing(p), clause(p(2, 0), Body).
```

```
p(2, 0).                        ⟹  Body = true ? ;
p(1, A) :-                      ⟹  Body = r(0) ? ;
        q(A).                   ⟹  no
p(2, A) :-
        r(A).
```

---

[10] For facts. `Body = true` is assumed.

An example with the BIP `clause`: wallpaper tracing

An interpreter for tracing pure Prolog programs, with no BIPs.

```
% interp(G, D): Interprets and traces goal G with an indentation D.
interp(true, _) :- !.
interp((G1, G2), D) :- !,
    interp(G1, D), interp(G2, D).
interp(G, D) :-
    (   trace(G, D, call)
    ;   trace(G, D, fail), fail % shows the fail port, keeps backtracking
    ),
    D2 is D+2,
    clause(G, B), interp(B, D2),
    (   trace(G, D, exit)
    ;   trace(G, D, redo), fail % shows the redo port, keeps backtracking
    ).

% Traces goal G at port Port with indentation D.
trace(G, D, Port) :-
    /* Writing out D spaces:*/ format('~|~t~*+', [D]),
    write(Port), write(': '), write(G), nl.
```

A sample run of the wallpaper trace interpreter

```
:- dynamic app/3,app/4. % (*)

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

app(L1, L2, L3, L123) :-
    app(L1, L23, L123),
    app(L2, L3, L23).
```

- Assuming that above text is stored in file, say, `app34.pl`, line `(*)` becomes unnecessary if the file is loaded by

```
| ?- load_files(app34,
         compilation_mode(
                 assert_all)).
```

```
| ?- interp(app(_,[b,c],L,[c,b,c,b]), 0).
  call: app(_203,[b,c],_253,[c,b,c,b])
    call: app(_203,_666,[c,b,c,b])
    exit: app([],[c,b,c,b],[c,b,c,b])
    call: app([b,c],_253,[c,b,c,b])
    fail: app([b,c],_253,[c,b,c,b])
    redo: app([],[c,b,c,b],[c,b,c,b])
      call: app(_873,_666,[b,c,b])
      exit: app([],[b,c,b],[b,c,b])
    exit: app([c],[b,c,b],[c,b,c,b])
    call: app([b,c],_253,[b,c,b])
      call: app([c],_253,[c,b])
        call: app([],_253,[b])
        exit: app([],[b],[b])
      exit: app([c],[b],[c,b])
    exit: app([b,c],[b],[b,c,b])
  exit: app([c],[b,c],[b],[c,b,c,b])
L = [b] ?
```

Contents

1. Declarative Programming with Prolog
    - Declarative and imperative programming
    - Propositional Prolog
    - Prolog with Simple Data Structures
    - Compound Data Structures in Prolog
    - Lists
    - Prolog implementation – a brief overview
    - Prolog execution – definitions
    - Prolog syntax
    - Syntactic sugar: operators
    - Further control constructs
    - BIPs 1 – meta-preds, all solutions, dynamic preds
    - BIPs 2 – higher order programming, loops, modules
    - Efficient programming in Prolog

Higher order predicates

- A higher order predicate (or meta-predicate) is a predicate with an argument which is interpreted as a goal, or a *partial goal*
  - e.g., `findall/3` is a meta-predicate, as its second argument is a goal
- A partial goal is a goal with some (usually the last *n*) arguments missing
  - e.g., a predicate name is a partial goal
- Example: `filter(L, Pred, FL)`: List `FL` contains those elements of `L` which satisfy `Pred`, where `Pred` is the name of a unary predicate

```
filter0(L, Pred, FL) :-
      Goal =.. [Pred,X], findall(X, (member(X,L), Goal), FL).

even(X) :- X mod 2 =:= 0.

| ?- filter0([1,3,2,5,4,0], even, FL).    ⟹     FL = [2,4,0] ; no.
```

- A less compact, but more efficient variant:

```
filter1([], _Pred, []).
filter1([X|L], Pred, FL) :-
    Goal =.. [Pred,X],
    (   call(Goal) -> FL = [X|FL1], filter1(L, Pred, FL1)
    ;   filter1(L, Pred, FL)
    ).
```

Calling predicates with additional arguments

- Definition: a callable term is a compound or atom.
- Built-in predicate group `call/N`
  - `call(Goal)`: invokes `Goal`, where `Goal` is a callable term
  - `call(PG, A)`: Adds `A` as the last argument to `PG`, and invokes it.
  - `call(PG, A, B)`: Adds `A` and `B` as the last two args to `PG`, invokes it.
  - `call(PG,` A_1`, ...,` A_n`)`: Adds $A_1, \ldots, A_n$ as the last n arguments to `PG`, and invokes the goal so obtained.
- `PG` is a partial goal, to be extended with additional arguments before calling. It has to be a callable term.
- Implementing `filter` using `call/2`

```
filter([], _PG, []).
filter([X|L], PG, FL) :-     (   call(PG, X) -> FL = [X|FL1]
                             ;   FL = FL1
                             ), filter(L, PG, FL1).

less(N, X) :- X < N.

| ?- filter([2,3,4,5,1,7], less(3), FL). ⟹  FL = [2,1] ? ; no
| ?- filter([2,3,4,5,1,7], =<(4), FL).   ⟹  FL = [4,5,7] ? ; no
```

Another useful higher order predicate: `map/3`

- `map(L, PG, ML)`: List `ML` contains elements `Y` obtained by calling `PG(X,Y)` for each `X` element of list `L`, where `PG` is a partial goal to be expanded with two arguments
- Variants:

```
map0(L, PG, ML) :-                                   % PG has to be an atom
     Goal =.. [PG,X,Y],   findall(Y, (member(X,L), Goal), ML).

map1(L, PG, ML) :-                               % PG can be a callable term
     findall(Y, (member(X,L), call(PG, X, Y)), ML).

map([], _, []).
map([X|L], PG, [Y|ML]) :-                        % PG can be a callable term
     call(PG, X, Y),
     map(L, PG, ML).

square(X, Y) :- Y is X*X.

mult(N, X, NX) :- NX is N*X.

| ?- map0([1,2,3,4], square, L).  ⟹  L = [1,4,9,16] ? ; no
| ?- map1([1,2,3,4], mult(2), L). ⟹  L = [2,4,6,8] ? ; no
| ?- map([1,2,3,4], mult(-5), L). ⟹  L = [-5,-10,-15,-20] ? ; no
```

Do-loops

- The main advantage of higher order predicates is that one can avoid writing auxiliary predicates.
- Another, even more efficient approach is to use do-loops.
  - Implementing `map(L, square, ML)` using a do-loop:
    `(  foreach(X, L), foreach(Y, ML) do  Y is X*X  )`
  - Implementing `map(L, mult(N), ML)` using a do-loop:
    `(  foreach(X, L), foreach(Y, ML), param(N)  do   Y is N*X  )`
- Examples of further iterators:

```
| ?- (  for(I,1,5), foreach(I,List) do true  ).
                          ⟹  List = [1,2,3,4,5] ? ; no

| ?- (  foreach(X,[1,2,3]), fromto(0,In,Out,Sum) do Out is In+X  ).
                          ⟹  Sum = 6 ? ; no

| ?- (  foreach(X,[a,b,c,d]), count(I,1,N), foreach(I-X,Pairs) do true  ).
                          ⟹  N = 4, Pairs = [1-a,2-b,3-c,4-d] ? ; no

| ?- (  foreacharg(A,f(a,b,c,d,e),I), foreach(I-A,List) do true  ).
                          ⟹  List = [1-a,2-b,3-c,4-d,5-e] ? ; no
```

Principles of the SICStus Prolog module system

- Each module should be placed in a separate file
- A module directive should be placed at the beginning of the file:
  `:- module( ModuleName, [ExportedFunc1, ExportedFunc2, ...]).`
- *ExportedFunc$_i$* – the functor (*Name/Arity*) of an exported predicate
- Example

```
:- module(drawing_lines, [draw/2]).      % line 1 of file draw.pl
```

- Built-in predicates for loading module files:
  - `use_module(FileName)`
  - `use_module(FileName, [ImportedFunc1,ImportedFunc2,...])`

    ImportedFunc$_i$ – the functor of an imported predicate

    FileName – an atom (with the default file extension `.pl`); or a special compound, such as `library(LibraryName)`
- Examples:

```
:- use_module(draw).                       % load the above module
:- use_module(library(lists), [last/2]). % only import last/2
```

- Goals can be module qualified: *Mod*:*Goal* runs *Goal* in module *Mod*
- Modules do not hide the non-exported predicates, these can be called from outside if the module qualified form is used

Meta predicates and modules

- Predicate arguments in imported predicates may cause problems:

File `module1.pl`:

```
:- module(module1, [double/1]).

% (1)

double(X) :-
        X, X.



p :- write(go).
```

File `module2.pl`:

```
:- module(module2, [q1/0,q2/0,r/0]).
:- use_module(module1).

q1 :- double(module1:p).

q2 :- double(module2:p).

r :-  double(p).                (2)

p :- write(ga).
```

- Load file `module2.pl`, e,g, by `| ?- [module2].`, and run some goals:

```
| ?- q1.   ⟹   gogo
| ?- q2.   ⟹   gaga
| ?- r.    ⟹   gogo
```

`:-(` counterintuitive

- Solution: Tell Prolog that `double` has a meta-arg. by adding at `(1)` this:

`:- meta_predicate double(:).`

This causes `(2)` to be replaced by '`r :- double(module2:p).`' at load time, making predicates `r` and `q2` identical.

Meta predicate declarations, module name expansion

- Syntax of meta predicate declarations
  `:- meta_predicate ⟨pred. name⟩(⟨modespec₁⟩, ..., ⟨modespecₙ⟩), ....`
  - ⟨`modespec`$_i$⟩ can be '`:`', '`+`', '`-`', or '`?`'.
  - Mode spec '`:`' indicates that the given argument is a meta-argument
- In all subsequent invocations of the given predicate the given arg. is replaced by its *module name expanded* form, at load time
  - Other mode specs just document modes of non-meta arguments.
- The module name expanded form of a term *`Term`* is:
  - *`Term`* itself, if *`Term`* is of the form *`M`*`:`*`X`* or it is a variable which occurs in the clause head in a meta argument position; otherwise
  - *`SMod`*`:`*`Term`*, where *`SMod`* is the current source module (`user` by default)
- Example, ctd. (`double` in `module1_m` is declared a meta predicate)

```
:- module(module3, [quadruple/1,r/0]).
:- use_module(module1_m).                         % the loaded form:
r :- double(p).                              ⟹    r :- double(module3:p).[11]

:- meta_predicate quadruple(:).
quadruple(X) :- double(X), double(X).  ⟹    unchanged[11]
```

---

[11] The imported goal `double` gets a prefix "`module1:`", not shown here, to save space.