

Part II

Declarative Programming with Prolog

- 1 Introduction to Logic
- 2 Declarative Programming with Prolog**
- 3 Declarative Programming with Constraints
- 4 The Semantic Web

Contents

2 Declarative Programming with Prolog

- Prolog – first steps
- Prolog execution models
- The syntax of the (unsweetened) Prolog language
- Further control constructs
- Operators and special terms
- Working with lists
- Term ordering
- Higher order predicates
- All solutions predicates
- Efficient programming in Prolog
- Building and decomposing terms
- Executable specifications
- Block declarations
- Further reading

Prolog in the family of programming languages

Programming paradigms – programming languages

Imperative

Fortran
Algol
C
Java
Python
...

Declarative

Functional

LISP
ML
Haskell
...

Logic

SQL
Prolog
Constraint Prog.
...

Prolog

- Birth date: 1972, designed by Alain Colmerauer, Robert Kowalski
- First public implementation (Marseille Prolog):
1973, interpreter in Fortran, A. Colmerauer, Ph. Roussel
- Second implementation (Hungarian Prolog):
1975, interpreter in CDL, Péter Szeredi

<http://dtai.cs.kuleuven.be/projects/ALP/newsletter/nov04/nav/articles/szeredi/szeredi.html>

- First compiler (Edinburgh Prolog, DEC-10 Prolog):
1977, David H. D. Warren (current syntax introduced)
- Wiki: <https://en.wikipedia.org/wiki/Prolog>

Prolog – PROgramming in LOGic: standard (Edinburgh) syntax

Standard syntax	English	Marseille syntax
<code>has_p(b, c).</code>	<code>% b has a parent c.</code>	<code>+has_p(b, c).</code>
<code>has_p(b, d).</code>	<code>% b has a parent d.</code>	<code>+has_p(b, d).</code>
<code>has_p(d, e).</code>	<code>% d has a parent e.</code>	<code>+has_p(d, e).</code>
<code>has_p(d, f).</code>	<code>% d has a parent f.</code>	<code>+has_p(d, f).</code>
	<code>% for all GC, GP, P holds</code>	
<code>has_gp(GC, GP) :-</code>	<code>% GC has grandparent GP if</code>	<code>+has_gp(*GC, *GP)</code>
<code>has_p(GC, P),</code>	<code>% GC has parent P and</code>	<code>-has_p(*GC,*P)</code>
<code>has_p(P, GP).</code>	<code>% P has parent GP.</code>	<code>-has_p(*P,*GP).</code>

FOL: $\forall GC, GP. (has_gp(GC, GP) \leftarrow \exists P. (has_p(GC, P) \wedge has_p(P, GP)))$

- Program execution is SLD resolution, which can also be viewed as pattern-based procedure invocation with backtracking
- Dual semantics: **declarative** and **procedural**
 - Slogan: **WHAT** rather than **HOW**
(focus on the **logic** first, but then think over Prolog **execution**, too).

Prolog clauses and predicates - some terminology

- A Prolog program is a sequence of *clauses*
- A clause represents a statement, it can be
 - a *fact*, of the form '*head*.' , e.g. `has_parent(a,b).`
 - a *rule*, of the form '*head* :- *body*.' ,
 e.g. `has_gp(GC, GP) :- has_p(GC, P), has_p(P, GP).` (*)
- Read ':-' as 'if', ',' as 'and'
- A *fact* can be viewed as having an empty body, or the body `true`
- A *body* is comma-separated list of *goals*, also named *calls*
- A *head* as well as a *goal* has the form *name*(*argument*,...), or just *name*
- A functor of a *head* or a *goal* (or a term, in general) is *F/N*, where *F* is the name of the term and *N* is the number of args (also called *arity*).
 Example: the functor of the head of (*) is `has_gp/2`
- The functor of a clause is the functor of its head.
- The collection of clauses with the same functor is called a *predicate* or *procedure*
- Clauses of a predicate should be contiguous (you get a warning, if not)

And what happened to the *function* symbols of FOL?

- Recall: In FOL, atomic predicates have arguments that are terms, built from variables using **function symbols**, e.g. $lseq(plus(X, 2), times(Y, Z))$
- In maths this is normally written in *infix operator* notation as $X + 2 \leq Y \cdot Z$
- In Prolog, graphic characters (and sequences of such) can be used for both relation and function names: $=<(+(X, 2), *(Y, Z))$ (1)
- As a “syntactic sweetener”, Prolog supports operator notation in user interaction, i.e. (1) is normally input and displayed as $x+2 =< Y*Z$. However, (1) is the internal, *canonical* format
- The built-in predicate (BIP) `write/1` displays its arg. using operators, while `write_canonical/1` shows the canonical form

?- write(1 - 2 =< 3*4).	⇒	1-2=<3*4
?- write_canonical(1 - 2 =< 3*4).	⇒	=<(-(1,2),*(3,4))
- Notice that the predicate arguments are not evaluated, function names act as **data constructors** (e.g. the op. `-` is used **not** only for subtraction)
- Prolog is a symbolic language, e.g. symbolic derivation is easy
- However, doing arithmetic requires special built-in predicates

Prolog built-in predicates (BIPs) for unification and arithmetic

- Unification. $X = Y$: unifies X and Y . Examples:

| ?- $X = 1-2$, $Z = X*X$. $\implies X = 1-2$, $Z = (1-2)*(1-2)$
 | ?- $U = X/Y$, $c(X,b)=c(a,Y)$. $\implies U = a/b$, $X = a$, $Y = b$
 | ?- $1-2*3 = X*Y$. \implies no (unification unsuccessful)

- Arithmetic evaluation. X is A : A is evaluated, the result is unified with X . A must be a **ground** arithmetic expression (**ground**: no free vars inside)

| ?- $X = 2$, Y is $X*X+2$. $\implies X = 2$, $Y = 6$?
 | ?- $X = 2$, 7 is $X*X+2$. \implies no
 | ?- $X = 6$, $7-1$ is X . \implies no
 | ?- X is $f(1,2)$. \implies 'Type Error'

- Arithmetic comparison. $A == B$: A and B are evaluated to numbers. Succeeds iff the two numbers are equal.

(Both A and B have to be ground arithmetic expressions.)

| ?- $X = 6$, $7-1 == X$. $\implies X = 6$
 | ?- $X = 6$, $X*X == (X+3)*(X-2)$. $\implies X = 6$
 | ?- $X = 6$, $X+3 == 2*(X-2)$. \implies no
 | ?- $X = 6$, $X+3 == 2*(Y-2)$. \implies 'Instantiation Error'

Further BIPs: $A < B$, $A > B$, $A <= B$ (\leq), $A >= B$ (\geq), $A \neq B$ (\neq),

An example: cryptarithmic puzzle

- Consider this cryptarithmic puzzle: $AD*AD = DAY$.
Here each letter stands for a *different* digit, initial digits cannot be zeros.
Find values for the digits A, D, Y, so that the equation holds.
- We'll use a library predicate `between/3` from library `between`.

```
% between(+N, +M, ?X): X is an integer such that N <= X <= M,
%                               Enumerates all such X values.
```
- I/O mode notation for pred. arguments (used **only** in comments):
`+`: input (bound), `-`: output (unbound var.), `?`: arbitrary.
- To load a library: (in SICStus) include the line below in your program:

```
:- use_module(library(between)).
```

 In SWI Prolog the predicate is loaded automatically.
- The Prolog predicate for solving the $AD*AD = DAY$ puzzle:

```
ad_day(AD, DAY) :-
    between(1, 9, A), between(1, 9, D), between(0, 9, Y),
    A \= D, A \= Y, D \= Y,
    DAY is D*100+A*10+Y, AD is A*10+D,
    AD * AD == DAY.
```
- Solve this puzzle yourself: `GO+TO=OUT`

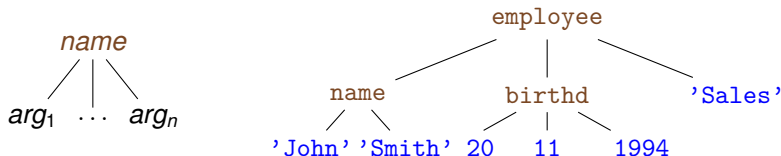
Data structures in Prolog

Prolog is a dynamically typed language, i.e. vars can take arbitrary values. Prolog data structures correspond to **FOL terms**. A Prolog term can be:

- **var** (**variable**), e.g. `X`, `Sum`, `_a`, `_`; the last two are *void* (don't care) vars (If a var occurs **once** in a clause, prefix it with `_`, or get a **WARNING!!!** Multiple occurrences of a single `_` symbol denote different vars.)
- **constant** (**0 argument function symbol**):
 - **number** (**integer** or **float**), e.g. `3`, `-5`, `3.1415`
 - **atom** (symbolic constant, cf. enum type), e.g. `a`, `susan`, `=<`, `'John'`
- **compound**, also called **record**, **structure** (**n -arg. function symbol**, $n > 0$)

A compound takes the form: `name(arg1, ..., argn)`, where

- `name` is an atom, `argi` are arbitrary Prolog terms
- e.g. `employee(name('John', 'Smith'), birthd(20, 11, 1994), 'Sales')`
- Compounds can be viewed as trees



Variables in Prolog: the logic variable

- A variable cannot be assigned (unified with) two distinct ground values:

| ?- $X = 1$, $X = 2$. \implies no

- Two variables may be unified and then assigned a (common) value:

| ?- $X = Y$, $X = 2$. $\implies X = 2$, $Y = 2$?

- The above apply to a single branch of execution. If we backtrack over a branch on which the variable was assigned, the assignment is undone, and on a new branch another assignment can be made:

has_p(b, c). has_p(b, d). has_p(d, e).

| ?- has_p(b, Y). $\implies Y = c$? ; $Y = d$? ; no

- A logic variable is a “first class citizen” data structure, it can appear inside compound terms:

| ?- $Emp = \text{employee}(\text{Name}, \text{Birth}, \text{Dept})$, $\text{Dept} = \text{'Sales'}$,

$\text{Name} = \text{name}(\text{First}, \text{Last})$, $\text{First} = \text{'John'}$.

$\implies Emp = \text{employee}(\text{name}(\text{'John'}, \text{Last}), \text{Birth}, \text{'Sales'})$?

- The `Emp` data structure represents an arbitrary employee with given name `John` who works in the `Sales` department

The logic variable (cont'd)

- A variable may also appear several times in a compound, e.g. `name(X,X)` is a Prolog term, which will match the first argument of the `employee/3` record, iff the person's first and last names are the same:

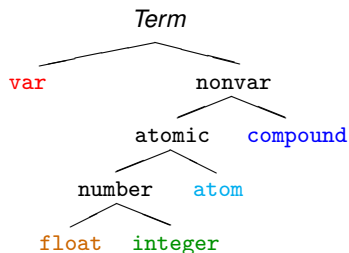
```
employee(1, employee(name('John', 'John'), birthd(2000,12,21), 'Sales')).
employee(2, employee(name('Ann', 'Kovach'), birthd(1988,8,18 ), 'HR')).
employee(3, employee(name('Peter', 'Peter'), birthd(1970,2,12 ), 'HR')).
```

```
| ?- employee(Num, Emp), Emp = employee(name(_X,_X),_,_).
Num = 1, Emp = employee(name('John', 'John'), birthd(2000,12,21), 'Sales') ? ;
Num = 3, Emp = employee(name('Peter', 'Peter'), birthd(1970,2,12), 'HR') ? ; no
```

- If a variable name starts with an underline, e.g. `_X`, its value is not displayed by the interactive Prolog shell (often called the *top level*)

Classification of Prolog terms

- The taxonomy of Prolog terms – corresponding built-in predicates (BIPs)



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

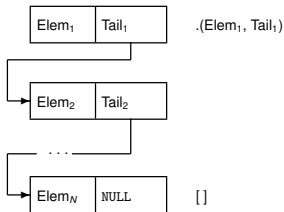
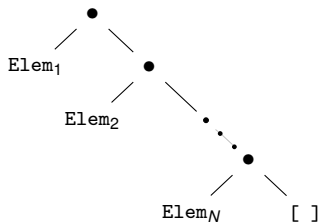
- The five coloured BIPs correspond to the five basic term types.
- Two further type-checking BIPs:
 - `simple(X)`: X is not compound, i.e. it is a variable or a constant.
 - `ground(X)`: X is a constant or a compound with no (uninstantiated) variables in it.

Another syntactic “sweetener” – list notation

- A Prolog **list** $[a, b, \dots]$ represents a sequence of terms (cf. linked list)

```
| ?- L = [a,b,c], write_canonical(L).
```

```
'. '(a, '. '(b, '. '(c, [])))
```



(Since version 7, SWI Prolog uses `'[]'`, instead of `'.' :-((((.`)

- The **head** of a list is its first element, e.g. L 's head: a
the **tail** is the list of all but the first element, e.g. L 's tail: $[b, c]$
- One often needs to split a list to its head and tail: $List = .(Head, Tail).$
The “square bracketed” counterpart: $List = [Head|Tail]$
- Further sweeteners: $[E_1, E_2, \dots, E_n | Tail] \equiv [E_1 | [E_2 | \dots, [E_n | Tail] \dots]]$
 $[E_1, E_2, \dots, E_n] \equiv [E_1, E_2, \dots, E_n | []]$

Open ended and proper lists

- Example:

```
% head0(L): L's first element is 0.
```

```
head0(L) :- L = [0|_]. % '_' is a void, don't care variable
```

```
% singleton(L): L has a single element.
```

```
singleton([_]).
```

```
| ?- singleton(L1). => L1 = [_A] % L1 = [_A|[]] is a proper list
```

```
| ?- head0(L2). => L2 = [0|_A] % L2 is an open ended list
```

- A Prolog term is called an *open ended* (or *partial*) list iff
 - either it is an unbound variable,
 - or it is a nonempty list structure (i.e. of the form `[_|_]`) and its tail is *open ended*,
 i.e. if sooner or later an unbound variable appears as the tail.
- A list is *closed* or *proper* iff sooner or later an `[]` appears as the tail
- Further examples: `[x,1,y]` is a proper list, `[x,1|z]` is open ended.

Working with lists – some practice

(Each occurrence of a void variable (`_`) denotes a different variable.)

?- [1,2] = [X Y].	⇒	X = 1, Y = [2] ?
?- [1,2] = [X,Y].	⇒	X = 1, Y = 2 ?
?- [1,2,3] = [X Y].	⇒	X = 1, Y = [2,3] ?
?- [1,2,3] = [X,Y].	⇒	no
?- [1,2,3,4] = [X,Y Z].	⇒	X = 1, Y = 2, Z = [3,4] ?
?- L = [a,b], L = [_ ,X _].	⇒	..., X = b ? % X = 2nd elem
?- L = [a,b], L = [_ ,X,_ _].	⇒	no ? % length >= 3, X = 2nd elem
?- L = [1 _], L = [_ ,2 _].	⇒	L = [1,2 _A] ? % open ended list

Programming with lists – simple example

- Recall: I/O mode notation for pred. arguments (**only** in comments):
+: input (bound), -: output (unbound var.), ?: arbitrary.
- Write a predicate that checks if all elements in a list are the same. Let's call such a list A-boring, where A is the element appearing repeatedly.
- Remember, you can read ':-' as 'if', ',' as 'and'

```
% boring(+L, ?A): List L is A-boring.  
boring([], _)    % [] is A-boring for every A.  
boring(L, A) :- % List L is A-boring, if  
    L=[A|L1],    % L's head equals A and  
    boring(L1, A). % L's tail is A-boring.
```


Programming with lists – further examples

- Given a list of numbers, calculate the sum of the list elements.
- Remember, you can do arithmetic calculations with 'is'

```
% sum(+L, ?Sum): L sums to Sum. (L is a list of numbers.)
sum([], 0).           % [] sums to 0.
sum([H|T], Sum) :-  % A list with head H and tail T sums to Sum if
    sum(T, Sum0),   % T sums to Sum0 and
    Sum is Sum0+H. % Sum is the value of Sum0+H.
```

- Given two arbitrary lists, check that they are of equal length.

```
% same_length(?L1, ?L2): Lists L1 and L2 are of equal length.
same_length([], []). % [] has the same length as []
same_length(L1, L2) :- % L1 and L2 are of equal length if
    L1 = [_|T1],      % the tail of L1 is T1 and
    L2 = [_|T2],      % the tail of L2 is T2 and
    same_length(T1, T2). % the T1 and the T2 are of equal length.
```

Another recursive data structure – binary tree

- A binary tree data structure can be defined as being
 - either a leaf (`leaf`) which contains an integer (`value`)
 - or a node (`node`) which contains two subtrees (`left`, `right`)
- Defining binary tree structures in C and Prolog:

```
% Declaration of a C structure
enum treetype Leaf, Node;
struct tree {
    enum treetype type;
    union {
        struct { int value;
                } leaf;

        struct { struct tree *left;
                struct tree *right;
                } node;
    } u;
};
```

```
% No need to define types in Prolog
% A type-checking predicate can be
% written, if this check is needed:
```

```
% is_tree(T): T is a binary tree
is_tree(leaf(Value)) :-
    integer(Value).
is_tree(node(Left,Right)) :-
    is_tree(Left),
    is_tree(Right).
```

Recall: `integer(Value)` is a BIP which succeeds if and only if `v` is an integer.

Calculating the sum of numbers in the leaves of a binary tree

- Calculating the sum of the leaves of a binary tree:
 - if the tree is a leaf, return the integer in the leaf
 - if the tree is a node, add the sums of the two subtrees

```
% C function (declarative)
int tree_sum(struct tree *tree) {
  switch(tree->type) {
    case Leaf:
      return tree->u.leaf.value;
    case Node:
      return
        tree_sum(tree->u.node.left) +
        tree_sum(tree->u.node.right);
  }
}
```

```
% Prolog procedure
% tree_sum(+T, ?S):
% The sum of the leaves
% of tree T is S.
tree_sum(leaf(Value), S) :-
    S = Value.
tree_sum(node(Left,Right), S) :-
    tree_sum(Left, S1),
    tree_sum(Right, S2),
    S is S1+S2.
```

Sum of Binary Trees – a sample run

```

% sicstus
SICStus 4.3.5 (...)
| ?- consult(tree).      % alternatively: compile(tree). or [tree].
% consulting /home/szeredi/examples/tree.pl...
% consulted /home/szeredi/examples/tree.pl in module user, (...)
| ?- tree_sum(node(leaf(5),
                    node(leaf(3), leaf(2))), Sum).

Sum = 10 ? ; no
| ?- tree_sum(leaf(10), 10).
yes
| ?- tree_sum(leaf(10), Sum).
Sum = 10 ? ; no
| ?- tree_sum(Tree, 10).
Tree = leaf(10) ? ;
! Instantiation error in argument 2 of is/2
! goal: 10 is _73+_74
| ?- halt.

```

The cause of the error: the built-in arithmetic is one-way: the goal `10 is S1+S2` causes an error!

Contents

2 Declarative Programming with Prolog

- Prolog – first steps
- **Prolog execution models**
- The syntax of the (unsweetened) Prolog language
- Further control constructs
- Operators and special terms
- Working with lists
- Term ordering
- Higher order predicates
- All solutions predicates
- Efficient programming in Prolog
- Building and decomposing terms
- Executable specifications
- Block declarations
- Further reading

Two Prolog execution models

- The **Goal Reduction** model
 - a reformulation of the resolution proof technique
 - good for visualizing the search tree
- The **Procedure Box** model
 - reflects actual implementation better
 - used by the Prolog trace mechanism

Goal reduction vs. resolution – a propositional example

```

get_fined :-    driving_fast, raining.           (1)
driving_fast :- in_a_hurry.                     (2)
...
in_a_hurry.   (3)
raining.      (4)

```

- To show that the goal `get_fined` holds, goal reduction repeatedly *reduces* it to other goals using clauses (1)-(4)
- When an empty goal (`true`) is obtained the goal gets proved.

```

(g1)  get_fined                % (g1) is reduced, using (1), to    (g2)
(g2)  driving_fast, raining    % (g2) is reduced, using (2), to    (g3)
(g3)  in_a_hurry,  raining     % (g3) is reduced, using (3), to    (g4)
(g4)  raining                % (g4) is reduced, using (4), to    (g5)
(g5)  ■ (empty goal) ≡ true

```

Goal reduction vs. resolution (cnt'd)

```

+get_fined      -driving_fast -raining.      (1)
+driving_fast   -in_a_hurry                  (2)
...
+in_a_hurry.    (3)
+raining.       (4)

```

- To show that `get_fined` holds, resolution does an indirect proof
- Assume `get_fined` does not hold, deduce false (contradiction) using clauses (1)–(4)

```

(g1) -get_fined          % (g1) and          (1) implies (g2)
(g2) -driving_fast -raining % (g2) and      (2) implies (g3)
(g3) -in_a_hurry  -raining % (g3) and      (3) implies (g4)
(g4) -raining      % (g4) and              (4) implies (g5)
(g5) □ (empty clause) ≡ false

```


The Goal Reduction model – the grandparent example

- Goal reduction takes a goal, i.e. a *conjunction* of subgoals G and using a clause C reduces it to goal G' ,

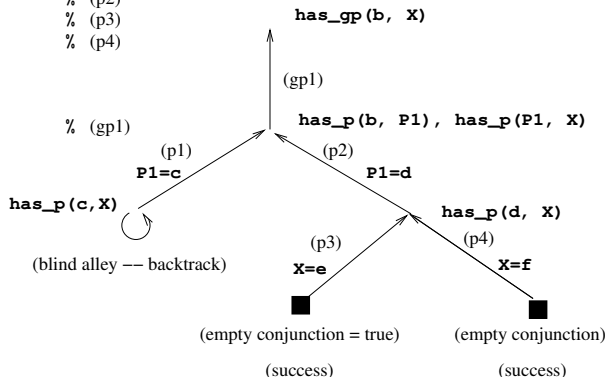
so that $G' \rightarrow G$
using (gp1) gives

- E.g. reducing $G = \text{has_gp}(b, X)$
 $G' = \text{has_p}(b, P1), \text{has_p}(P1, X)$

```
has_p(b, c).           % (p1)
has_p(b, d).           % (p2)
has_p(d, e).           % (p3)
has_p(d, f).           % (p4)
```

```
has_gp(GC, GP) :-
    has_p(GC, P),
    has_p(P, GP).      % (gp1)
```

```
| ?- has_gp(b, X).
```



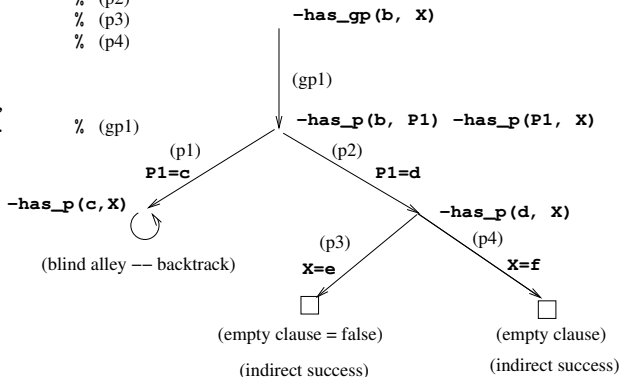
Resolution – same example

- Resolution takes a negated goal NG (which is a *disjunction* of neg. literals) and using a clause C deduces new negated goal NG' , so that $NG \rightarrow NG'$ using (gp1) gives
- E.g. resolving $NG = \text{-has_gp}(b, X)$
 $NG' = \text{-has_p}(b, P1) \text{-has_p}(P1, X)$

```
+has_p(b, c).           % (p1)
+has_p(b, d).           % (p2)
+has_p(d, e).           % (p3)
+has_p(d, f).           % (p4)
```

```
+has_gp(GC, GP)
  -has_p(GC, P),
  -has_p(P, GP).       % (gp1)
```

```
-has_gp(b, X).
```



The Goal Reduction model (ADVANCED)

Goal reduction: a goal is viewed as a conjunction of subgoals

- Given a goal $G = A, B, \dots$ and a clause $(A :- D, \dots)$
 $G' = B, \dots, D, \dots$ is obtained as the new goal

Goal reduction is the same as resolution, but viewed as backwards reasoning

- Resolution:
 - to prove $A \wedge B \wedge \dots$, we negate it obtaining $\neg G_0 = \neg A \neg B \dots$
 - resolution step : clause $Cl = (+A \neg D \dots)$ resolved with $\neg G_0$
 produces $\neg G_1 = \neg D \dots \neg B \dots$

$$\neg G_n \wedge Cl \rightarrow \neg G_{n+1}$$
 (resolution)
 - success of indirect proof: reaching an empty clause $\square \equiv \text{false}$
- Goal reduction:
 - to prove $A \wedge B \wedge \dots$, we start with $G_0 = A, B, \dots$
 - reduction step : using $Cl = (A :- D, \dots)$ one can reduce G_0 to
 $G_1 = D, \dots, B, \dots$

$$G_{n+1} \wedge Cl \rightarrow G_n$$
 (reduction)
 - success of the reduction proof: reaching an empty goal $\blacksquare \equiv \text{true}$
- the (resolution) and (reduction) reasoning rules are equivalent!

The definition of a goal reduction step

Reduce a goal G to a new goal G' using a program clause Cl_i :

- Split goal G into the **first** subgoal G_F and the residual goal G_R
- **Copy** clause Cl_i , i.e. rename all variables to new ones, and split the copy to a head H and body B
- **Unify** the goal G_F and the head H
 - If the unification fails, exit the reduction step with failure
 - If the unification succeeds with a substitution σ , return the new goal $G' = (B, G_R)\sigma$ (i.e. apply σ to both the body and the residual goal)

E.g., slide 109: $G = \text{has_gp}(b, X) \text{ using } (\text{gp1}) \Rightarrow G' = \text{has_p}(b, P1), \text{has_p}(P1, X)$

Reduce a goal G to a new goal G' by executing a built-in predicate (BIP)

- Split goal G into the first, BIP subgoal G_F and the residual goal G_R
- **Execute** the BIP G_F
 - If the BIP fails then exit the reduction step with failure
 - If the BIP succeeds with a substitution σ then return the new goal $G' = G_R\sigma$

The goal reduction model of Prolog execution – outline

- This model describes how Prolog builds and traverses a search tree
- A web app for practicing the model: <https://ait.plwin.dev/P1-1>
- The inputs:
 - a Prolog program (a sequence of clauses), e.g. the `has_gp` program
 - a goal, e.g. `:- has_gp(b, GP).`
extended with a special goal, carrying the solution: `answer(Sol):`
`:- has_gp(b, GP), answer(GP). % Who are the grandparents of a?`
`:- has_gp(Ch, GP), answer(Ch-GP). % Which are the child-gparent pairs?`
- When only an `answer` goal remains, a solution is obtained
- Possible outcomes of executing a Prolog goal:
 - Exception (error), e.g. `:- Y = apple, X is Y+1.`
(This is not discussed further here)
 - Failure (no solutions), e.g. `:- has_p(c, P), answer(P).`
 - Success (1 or more solutions), e.g. `:- has_p(d, P), answer(P).`

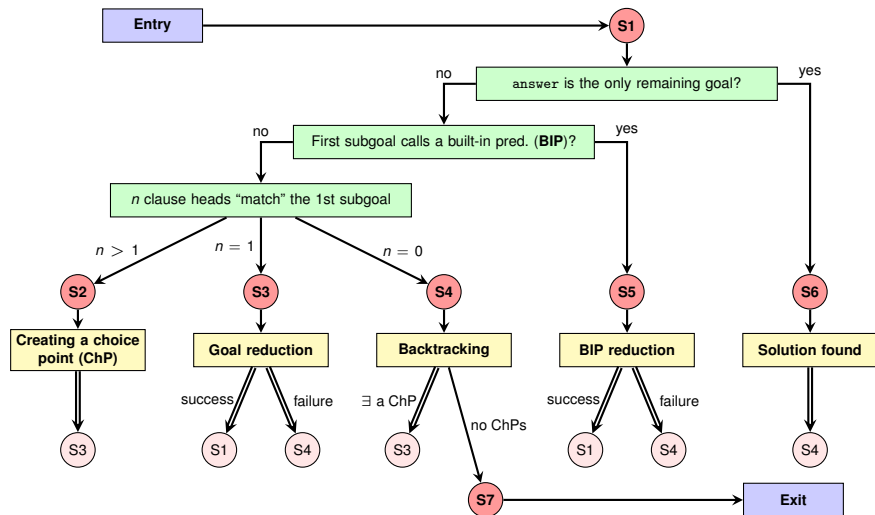
The main data structures used in the model

- There are only two (imperative, mutable) variables in this model:
 - Goal**: the current goal sequence, **ChPSt** the stack of choice points (ChPs)
- If, in a reduction step, two or more clause heads unify (match) the first subgoal, a new **ChPSt** entry is made, storing:
 - the list of clauses with possibly matching heads
 - the current goal sequence (i.e. **Goal**)

ChPoint name	Clause list	Goal
<i>CHP2</i>	[p3,p4]	(4) <code>hasP(d,Y), answer(b-Y).</code>
<i>CHP1</i>	[p2,p3,p4]	(2) <code>hasP(X,P), hasP(P,Y), answer(X-Y).</code>

- At a failure, the top entry of the **ChPSt** is examined:
 - the goal stored there becomes the current **Goal**,
 - the first element of the list of clauses is removed, the second is remembered the as the “**current clause**”,
 - if the list of clauses is now a singleton, the top entry is removed,
 - finally the **Goal** is reduced, using the **current clause**.
- If, at a failure, **ChPSt** is empty, execution ends.

The flowchart of the Prolog goal reduction model



(Double arrows indicate a jump to the step in the pink circle, i.e. execution continues at the given red circle.)

Remarks on the flowchart

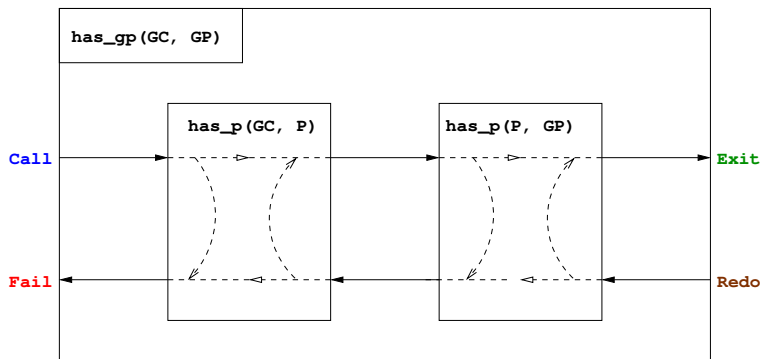
- There are seven different execution steps: **S1–S7**, where **S1** is the initial (but also an intermediate) step, and **S7** represents the final state.
- The main task of **S1** is to branch to one of **S2–S6**:
 - when `Goal` contains an `answer` goal only \Rightarrow **S6**;
 - when the first subgoal of `Goal` calls a BIP \Rightarrow **S5**;
 - otherwise the first subgoal calls a user predicate. Here a set of clauses is selected which *contains* all clauses whose heads match the first subgoal (this may be a *superset* of the matching ones).
Based on the number of clauses \Rightarrow **S2**, **S3** or **S4**.
- **S2** creates a new `ChPSt` entry, and \Rightarrow **S3** (to reduce with the first clause).
- **S3** performs the reduction. If that fails \Rightarrow **S4**, otherwise \Rightarrow **S1**.
- **S4** retrieves the next clause from the top `ChPSt` entry, if any (\Rightarrow **S3**), otherwise execution ends (\Rightarrow **S7**).
- In **S5**, similarly to **S3**, if the BIP succeeds \Rightarrow **S1**, otherwise \Rightarrow **S4**.
- In **S6**, the solution is displayed and further solutions are sought (\Rightarrow **S4**).

The Procedure Box execution model – example

- The **procedure box** execution model of `has_gp`

```
has_gp(GC, GP) :- has_p(GC, P), has_p(P, GP).
```

```
has_p(b, c).
has_p(b, d).
has_p(d, e).
has_p(d, f).
```



Prolog tracing, based on the four port box model

```
| ?- consult(gp3).
% consulting gp3.pl...
% consulted gp3.pl ...
yes
| ?- listing.
has_gp(Ch, G) :-
    has_p(Ch, P),
    has_p(P, G).

has_p(b, c).
has_p(b, d).
has_p(d, e).
has_p(d, f).

yes
| ?- trace.
% The debugger will ...
yes
```

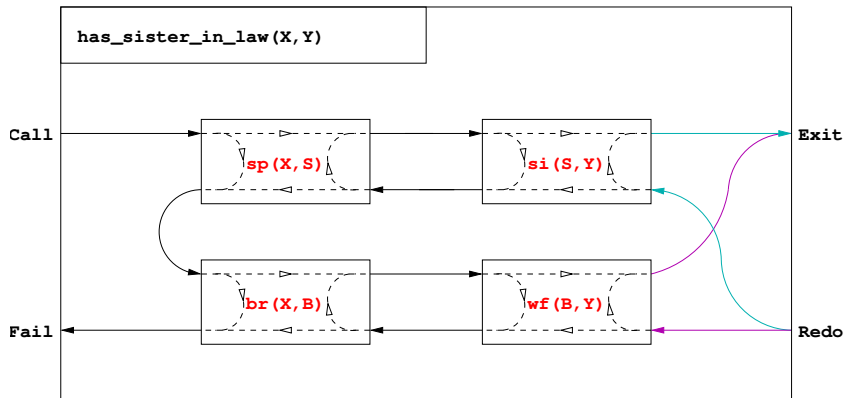
```
| ?- has_gp(Ch, f).
Det? BoxId Depth Port Goal
      1      1 Call: has_gp(Ch,f) ?
      2      2 Call: has_p(Ch,P) ?
?      2      2 Exit: has_p(b,c) ?
      3      2 Call: has_p(c,f) ?
      3      2 Fail: has_p(c,f) ?
      2      2 Redo: has_p(b,c) ?
?      2      2 Exit: has_p(b,d) ?
      4      2 Call: has_p(d,f) ?
      4      2 Exit: has_p(d,f) ?
                               No choice left in box 4, box removed (no ?)
?      1      1 Exit: has_gp(b,f) ?
Ch = b ? ;
      1      1 Redo: has_gp(b,f) ?
      2      2 Redo: has_p(b,d) ?
?      2      2 Exit: has_p(d,e) ?
      5      2 Call: has_p(e,f) ?
      5      2 Fail: has_p(e,f) ?
      2      2 Redo: has_p(d,e) ?
      2      2 Exit: has_p(d,f) ?
                               No choice left in box 2, box removed (no ?)
      6      2 Call: has_p(f,f) ?
      6      2 Fail: has_p(f,f) ?
      1      1 Fail: has_gp(Ch,f) ?

no
| ?-
```

The procedure-box of multi-clause predicates

‘Sister in law’ can be one’s spouse’s sister; or one’s brother’s wife:

```
has_sister_in_law(X, Y) :-
    has_spouse(X, S), has_sister(S, Y).
has_sister_in_law(X, Y) :-
    has_brother(X, B), has_wife(B, Y).
```



The procedure-box of a “database” predicate of facts

- In general in a multi-clause predicate the clauses have different heads
- A database of facts is a typical example:

```
has_p(b, c).
```

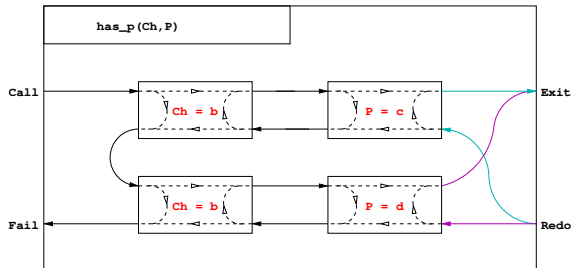
```
has_p(b, d).
```

- These clauses can be massaged to have the same head:

```
has_p(Ch, P) :- Ch = b, P = c.
```

```
has_p(Ch, P) :- Ch = b, P = d.
```

- Consequently, the procedure-box of this predicate is this:



Contents

2 Declarative Programming with Prolog

- Prolog – first steps
- Prolog execution models
- **The syntax of the (unsweetened) Prolog language**
- Further control constructs
- Operators and special terms
- Working with lists
- Term ordering
- Higher order predicates
- All solutions predicates
- Efficient programming in Prolog
- Building and decomposing terms
- Executable specifications
- Block declarations
- Further reading

Summary – syntax of Prolog predicates, clauses

Example

```
% A predicate with two clauses, the functor is: tree_sum/2
tree_sum(leaf(Val), Val).           % clause 1, fact
tree_sum(node(Left,Right), S) :- % head \
    tree_sum(Left, S1),           % goal \ |
    tree_sum(Right, S2),         % goal | body | clause 2, rule
    S is S1+S2.                  % goal / /
```

Syntax

$\langle \text{program} \rangle ::=$	$\langle \text{predicate} \rangle \dots$	{i.e. a sequence of predicates}
$\langle \text{predicate} \rangle ::=$	$\langle \text{clause} \rangle \dots$	{with the same functor}
$\langle \text{clause} \rangle ::=$	$\langle \text{fact} \rangle . \sqcup $ $\langle \text{rule} \rangle . \sqcup$	
$\langle \text{fact} \rangle ::=$	$\langle \text{head} \rangle$	
$\langle \text{rule} \rangle ::=$	$\langle \text{head} \rangle :- \langle \text{body} \rangle$	{clause functor = head functor}
$\langle \text{body} \rangle ::=$	$\langle \text{goal} \rangle, \dots$	{i.e. a seq. of goals sep. by commas}
$\langle \text{head} \rangle ::=$	$\langle \text{callable term} \rangle$	{atom or compound}
$\langle \text{goal} \rangle ::=$	$\langle \text{callable term} \rangle$	{or a variable, if instantiated to a callable}

Prolog terms (canonical form)

Example – a clause head as a term

```
% tree_sum(node(Left,Right), S)      % compound term, has the
% ----- - - - - - - - - - - - - - % functor tree_sum/2
%      |           |           |
% compound name \           argument, variable
%              \ - argument, compound term
```

Syntax

```
< term > ::= < variable > | {has no functor}
           < constant > |   {< constant >/0}
           < compound term > | {< comp. name >/< # of args >}
           ... extensions ... {lists, operators}

< constant > ::= < atom > | {symbolic constant}
              < number >

< number > ::= < integer > | < float >

< compound term > ::= < comp. name > ( < argument >, ... )
< comp. name > ::= < atom >
< argument > ::= < term >
< callable term > ::= < atom > | < compound term >
```

Lexical elements

Examples

```
% variable:      Fact FACT _fact X2 _2 _
% atom:         fact ≡ 'fact' 'István' [] ; ',,' += ** \= ≡ '\\='
% number:      0 -123 10.0 -12.1e8
% not an atom:  !=, István
% not a number: 1e8 1.e2
```

Syntax

```
< variable > ::= < capital letter > < alphanum > ... |
               _ < alphanum > ...
< atom > ::= ' < quoted char > ... ' |
            < lower case letter > < alphanum > ... |
            < sticky char > ... | ! | ; | [] | {}
< integer > ::= {signed or unsigned sequence of digits}
< float > ::= {a sequence of digits with a compulsory decimal point
              in between, with an optional exponent}
< quoted char > ::= {any non ' and non \ character} | \ < escaped char >
< alphanum > ::= < lower case letter > | < upper case letter > | < digit > | _
< sticky char > ::= + | - | * | / | \ | $ | ^ | < | > | = | ' | ~ | : | . | ? | @ | # | &
```


Comments and layout in Prolog

- Comments
 - From a % character till the end of line
 - From /* till the next */
- Layout (spaces, newlines, tabs, comments) can be used freely, except:
 - No layout allowed between the name of a compound and the “(”
 - If a prefix operator (see later) is followed by “(”, these have to be separated by layout
 - Clause terminator (._␣): a stand-alone full stop (i.e., one not preceded by a sticky char), followed by layout
- The recommended formatting of Prolog programs:
 - Write clauses of a predicate continuously, no empty lines between
 - Precede each pred. by an empty line and a spec (head comment)

```
% predicate_name(A1, ..., An): A declarative sentence (statement)  
% describing the relationship between terms A1, ..., An
```
 - Write the head of the clause at the beginning of a line, and prefix each goal in the body with an indentation of a few (8 recommended) spaces.

Contents

2 Declarative Programming with Prolog

- Prolog – first steps
- Prolog execution models
- The syntax of the (unsweetened) Prolog language
- **Further control constructs**
- Operators and special terms
- Working with lists
- Term ordering
- Higher order predicates
- All solutions predicates
- Efficient programming in Prolog
- Building and decomposing terms
- Executable specifications
- Block declarations
- Further reading

Disjunctions

- Disjunctions (i.e. subgoals separated by “or”) can appear as goals
- A disjunction is denoted by semicolon (“;”)
- Enclose the **whole** disjunction in parentheses, align chars (, ; and)

```
has_sister_in_law(X, Y) :-
    (   has_spouse(X, S), has_sister(S, Y)
    ;   has_brother(X, B), has_wife(B, Y)
    ).
```

- The above predicate is equivalent to:

```
has_sister_in_law(X, Y) :- has_spouse(X, S), has_sister(S, Y).
has_sister_in_law(X, Y) :- has_brother(X, B), has_wife(B, Y).
```

- A disjunction is itself a valid goal, it can appear in a conjunction:

```
has_ancestor(X, A) :-
    has_parent(X, P), (   A = P
                        ;   has_ancestor(P, A)
    ).
```

Can you make an equivalent variant which does not use “;”?

Disjunctions, continued

- An example with multiple disjunctions:

```
% first_1(L): the first nonzero element of L is 1.
first_1([A,B,C]) :-
    (   A = 1
    ;   A = 0,
        (   B = 1
        ;   B = 0, C = 1
        )
    ).
```

- Note: the $v=Term$ goals can no longer be got rid of in disjunctions
- Comma binds more tightly than semicolon, e.g.

```
p :- ( q, r ; s ) ≡ p :- ((q, r) ; s).
```

Please, never enclose disjuncts (goals on the sides of ;) in parentheses!

- You can have more than two-way “or”s:

```
p :- ( a ; b ; c ; ... ) which is the same as
p :- ( a ; (b ; (c ; ...)) )
```

- Please, do not use the unnecessary parentheses (colored red)!

Expanding disjunctions to helper predicates

- Example: $p \text{ :- } q, (r ; s).$

$$p \text{ :- } q, r.$$

Distributive expansion inefficient, as it calls q twice:

$$p \text{ :- } q, s.$$

- For an efficient solution introduce a helper predicate. Example:

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

- Collect variables that occur both inside and outside the disj. – Y, Z .
- Define a helper predicate – $\text{aux}(Y, Z)$ – with these vars as args, transform each disjunct to a separate clause of the helper predicate:

```
aux(Y, Z) :- q(Y, U), r(U, Z).
```

```
aux(Y, Z) :- s(Y, Z).
```

```
aux(Y, Z) :- t(Y), w(Z).
```

- Replace the disjunction with a call of the helper predicate:

```
t(X, Z) :- p(X, Y), aux(Y, Z), v(X, Z).
```

The if-then-else construct

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

`% pow(A, E, P): P is A to the power E.`

`pow(A, E, P) :-`

`(E > 0, E1 is E-1, \implies`
`pow(A, E1, P1),`
`P is A*P1`

`; E = 0, P = 1`

`).`

`pow1(A, E, P) :-`

`(E > 0 -> E1 is E-1,`
`pow(A, E1, P1),`
`P is A*P1`

`; E = 0, P = 1`

`).`

- `pow1` is about 25% faster than `pow` and requires much less memory
- The atom `->` is a standard operator
- 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`.

Defining “childless” using if-then-else

- Given the `has_parent/2` predicate, define the notion of a `childless` person
- If we can find a child of a GIVEN person, then `childless` should fail, otherwise it should succeed.

```
% childless(+Person): A given Person has no children
childless(Person) :-      (   has_parent(_, Person) -> fail
                           ;   true
                           ).
```

- What happens if you call `childless(P)`, where `P` is an unbound var? Will it enumerate childless people in `P`? No, it will simply fail.
- The above if-then-else can be simplified to:


```
childless(Person) :- \+ has_parent(_, Person).
```
- “`\+`” is called Negation by Failure, “`\+ G`” runs by executing `G`:
 - if `G` fails “`\+ G`” succeeds.
 - if `G` succeeds “`\+ G`” fails (ignoring further solutions of `G`, if any)
- Since a failed goal produces no bindings, “`\+ G`” will never bind a variable.
- Read “`\+`” as “not provable”, cf. \neg tilted slightly to the left.

Open and closed world assumption

`has_parent(a, b).` `has_parent(a, c).` `has_parent(c, d).` (1)-(3)

- Does (1)-(3) imply that `a` is childless: $\varphi = \forall x. \neg \text{has_parent}(x, a)$?
- No. Although `has_parent(c, a)` cannot be proven, φ does not hold!
- But in the world of databases we do conclude that `a` is childless. . .
- Databases use the Closed World Assumption (CWA): anything that cannot be proven is considered false.
- Mathematical logic uses the Open World Assumption (OWA)
 - A statement S follows from a set of statements P (premises), if S holds in any world (interpretation) that satisfies P .
 - thus φ is not a logical consequence of (1)-(3)
- Classical logic (OWA) is monotonic:
the more you know, the more you can deduce
- Negation by failure (CWA) is non-monotonic:
add the fact "`has_parent(e, a).`" to (1)-(3) and `\+ has_parent(_, a)` will fail.

Checking inequality – siblings and cousins

```
has_p('Charles', 'Elizabeth'). has_p('Andrew', 'Elizabeth').
has_p('William', 'Charles').   has_p('Beatrice', 'Andrew').
has_p('Harry', 'Charles').     has_p('Eugenie', 'Andrew').
```

- Recall homework L4, define predicate `has_sibling/2`, first attempt:

```
has_sibling0(A, B) :- \+ A = B, has_p(A, P), has_p(B, P).
```

- `has_sibling0` does **not** work properly, e.g. this goal fails:

```
| ?- has_sibling0('Charles', X).
```

because `\+ 'Charles' = X` fails (as `'Charles' = X` succeeds)

- Negated goals should be instantiated as much as possible, therefore always place them at the end of the body:

```
has_sibling(A, B) :- has_p(A, P), has_p(B, P), \+ A = B.
```

- Define `has_cousin/2` (using `has_gp/2`, the “has grandparent” predicate)

```
has_cousin(A, B) :-
    has_gp(A, GP), has_gp(B, GP), \+ has_sibling(A, B), A \= B.
```

- Note that the BIP `A \= B` is equivalent to `\+ A = B`

The relationship of if-then-else and negation

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

$$\backslash+ p \quad \equiv \quad \begin{array}{l} (\quad p \rightarrow \text{false} \\ \quad ; \quad \text{true} \\) \end{array}$$

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

$$\begin{array}{l} (\quad \text{cond} \rightarrow \text{then} \\ \quad ; \quad \text{else} \\) \end{array} \quad \Longrightarrow \quad \begin{array}{l} (\quad \text{cond}, \text{then} \\ \quad ; \quad \backslash+ \text{cond}, \text{else} \\) \end{array}$$

These are equivalent only if `cond` succeeds at most once.

The if-then-else is more efficient (no choice point left).

- As semicolon is associative, there is no need to use nested parentheses (...) if multiple if-then-else branches are present (and please don't):

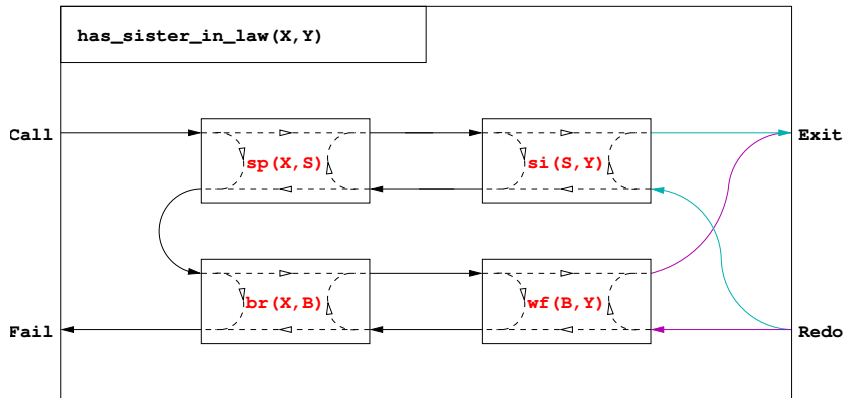
$$\begin{array}{l} (\quad \text{cond1} \rightarrow \text{then1} \\ \quad ; \quad (\quad \text{cond2} \rightarrow \text{then2} \\ \quad \quad ; \quad (\quad \dots \quad) \\ \quad) \\ \quad ; \quad \text{else} \\) \end{array} \quad \Longrightarrow \quad \begin{array}{l} (\quad \text{cond1} \rightarrow \text{then1} \\ \quad ; \quad \text{cond2} \rightarrow \text{then2} \\ \quad ; \quad (\quad \dots \quad) \\ \quad ; \quad \text{else} \\) \end{array}$$

The procedure-box of disjunctions

A disjunction can be transformed into a multi-clause predicate

```
has_sister_in_law(X, Y) :-
  ( has_spouse(X, S), has_sister(S, Y)
  ;
  has_brother(X, B), has_wife(B, Y)
  ).
```

```
has_sister_in_law(X, Y) :-
  has_spouse(X, S), has_sister(S, Y).
has_sister_in_law(X, Y) :-
  has_brother(X, B), has_wife(B, Y).
```

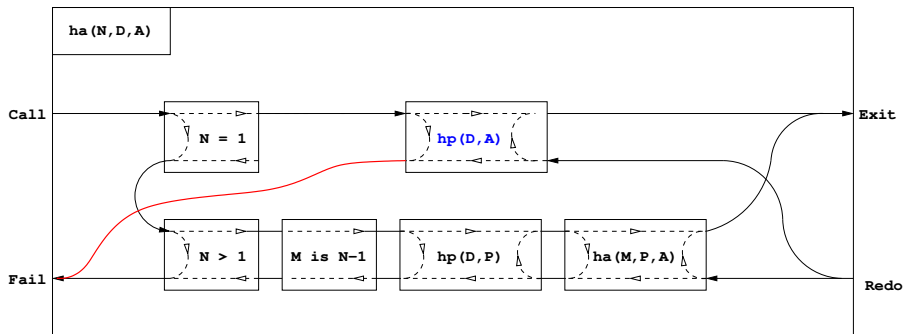


The procedure box for if-then-else

```

% ha(+N, ?D, ?A): D has A as their Nth generation ancestor (N>0 int)
% The 1st,      2nd,      3rd generation ancestors are
%   parents, grandparents, great-grandparents etc.
ha(N, D, A) :-
    (   N = 1 -> hp(D, A)                % hp(D, A): D has a parent A
    ;   N > 1, M is N-1, hp(D, P), ha(M, P, A)
    ).

```

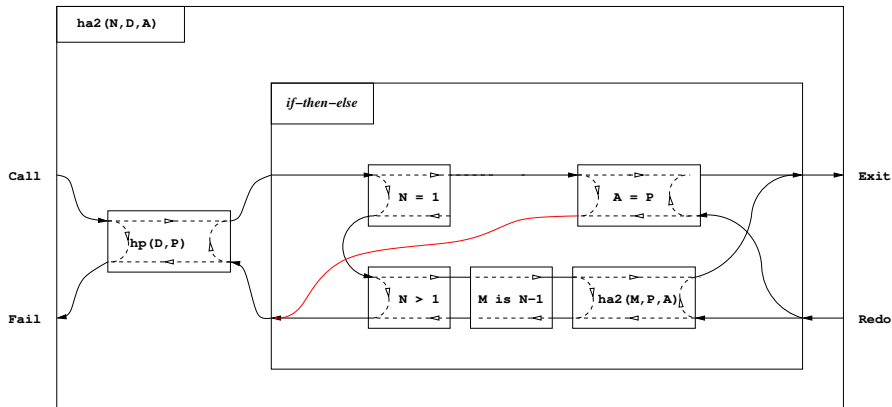


- Failure of the “then” part leads to failure of the whole if-then-else construct

The if-then-else box, continued

- When an if-then-else occurs in a conjunction, or there are multiple clauses, then it requires a separate box

```
ha2(N, D, A) :- hp(D, P), (
    N = 1 -> A = P
    ; N > 1, M is N-1, ha2(M, P, A)
).
```



Contents

2 Declarative Programming with Prolog

- Prolog – first steps
- Prolog execution models
- The syntax of the (unsweetened) Prolog language
- Further control constructs
- **Operators and special terms**
- Working with lists
- Term ordering
- Higher order predicates
- All solutions predicates
- Efficient programming in Prolog
- Building and decomposing terms
- Executable specifications
- Block declarations
- Further reading

Introducing operators

- Example: s is $-s_1+s_2$ is equivalent to: $is(s, +(-(s_1), s_2))$
- Syntax of terms using operators

$\langle \text{comp. term} \rangle ::=$	
$\langle \text{comp. name} \rangle (\langle \text{argument} \rangle, \dots)$	{so far we had this}
$\langle \text{argument} \rangle \langle \text{operator name} \rangle \langle \text{argument} \rangle$	{infix term}
$\langle \text{operator name} \rangle \langle \text{argument} \rangle$	{prefix term}
$\langle \text{argument} \rangle \langle \text{operator name} \rangle$	{postfix term}
$(\langle \text{term} \rangle)$	{parenthesized term}
$\langle \text{operator name} \rangle ::= \langle \text{comp. name} \rangle$	{if declared as an operator}

- The built-in predicate for defining operators:

$op(\text{Priority}, \text{Type}, Op)$ OR $op(\text{Priority}, \text{Type}, [Op_1, Op_2, \dots])$:

- Priority: an int. between 1 and 1200 – smaller priorities bind tighter
 - Type determines the **placement** of the operator and the associativity:
 - infix**: yfx, xfy, xfx ; **prefix**: fy, fx ; **postfix**: yf, xf ($f - op, x, y - args$)
 - Op or Op_i : an arbitrary atom
- The call of the BIP $op/3$ is normally placed in a **directive**, executed immediately when the program file is loaded, e.g.:

```
:- op(800, xfx, [has_tree_sum]).      leaf(V) has_tree_sum V.
```

Characteristics of operators

Operator properties implied by the operator type

Type			Class	Interpretation
left-assoc.	right-assoc.	non-assoc.		
yfx	xfy	xfx	infix	$X \ f \ Y \equiv f(X, Y)$
	fy	fx	prefix	$f \ X \equiv f(X)$
yf		xf	postfix	$X \ f \equiv f(X)$

Parentheses implied by operator priorities and associativities

- $a/b+c*d \equiv (a/b)+(c*d)$ as the priority of $/$ and $*$ (400) is less than the priority of $+$ (500) smaller priority = **stronger** binding
- $a-b-c \equiv (a-b)-c$ as operator $-$ has type yfx , thus it is left-associative, i.e. it binds to the left, the leftmost operator is parenthesized first (the position of y wrt. f shows the direction of associativity)
- $a^b^c \equiv a^(b^c)$ as $^$ has type xfy , therefore it is right-associative
- $a=b=c \implies$ syntax error, as $=$ has type xfx , it is non-associative
- the above also applies to different operators of same type and priority:
 $a+b-c+d \equiv ((a+b)-c)+d$

Standard built-in operators

Standard operators

```

1200  xfx  :- -->
1200   fx  :- ?-
1100   xfy ;
1050   xfy ->
1000   xfy ', '
  900   fy  \+
  700   xfx = \= =..
        < =< =:= =\=
        > >= is
        == \==
        @< @=< @> @>=

500   yfx  + - /\ \\/
400   yfx  * / // rem
        mod << >>

200   xfx  **
200   xfy  ^
200   fy   - \

```

Further built-in operators of SICStus Prolog

```

1150   fx  mode public dynamic
        volatile discontinuous
        initialization multifile
        meta_predicate block

1100   xfy  do
  900   fy  spy nospy
  550   xfy :
  500   yfx \
  200   fy  +

```

Operators – additional comments

- The “comma” is heavily overloaded:
 - it separates the arguments of a compound term
 - it separates list elements
 - it is an xfy op. of priority 1000, e.g.:
 $(p:-a,b,c) \equiv :- (p, ', '(a, ', '(b,c)))$
- Ambiguities arise, e.g. is $p(a,b,c) \stackrel{?}{\equiv} p((a,b,c))$?
- Disambiguation: if the outermost operator of a compound argument has priority ≥ 1000 , then it should be enclosed in parentheses

```
| ?- write_canonical((a,b,c)). => ', '(a, ', '(b,c))
```

```
| ?- write_canonical(a,b,c). => Error: ! write_canonical/3 does not exist
```

```
| ?- write_canonical((hgp(A,B):-hp(A,C),hp(C,B))).
```

```
=> :- (hgp(A,B), ', ', '(hp(A,C),hp(C,B)))
```

- Note: an unquoted comma (,) is an operator, but **not** a valid atom

Functions and operators allowed in arithmetic expressions

- The Prolog standard prescribes that the following functions can be used in arithmetic expressions:

plain arithmetic:

+X, -X, X+Y, X-Y, X*Y, X/Y,
X//Y (int. division, truncates towards 0),
X div Y (int. division, truncates towards $-\infty$),
X rem Y (remainder wrt. //),
X mod Y (remainder wrt. div),
X**Y, X^Y (both denote exponentiation)

conversions:

float_integer_part(X), float_fractional_part(X), float(X),
round(X), truncate(X), floor(X), ceiling(X)

bit-wise ops:

X/\Y, X\Y, xor(X,Y), \ X (negation), X<<Y, X>>Y (shifts)

other:

abs(X), sign(X), min(X,Y), max(X,Y),
sin(X), cos(X), tan(X), asin(X), acos(X), atan(X),
atan2(X,Y), sqrt(X), log(X), exp(X), pi

Uses of operators

- What are operators good for?
 - to allow usual arithmetic expressions, such as in `X is (Y+3) mod 4`
 - processing of symbolic expressions (such as symbolic derivation)
 - for writing the clauses themselves
(`:-`, `'`, `,`, `;` ... are all standard operators)
 - clauses can be passed as arguments to meta-predicates:
`asserta((p(X):-q(X),r(X)))`
 - to make Prolog data structures look like natural language sentences (controlled English), e.g. Smullyan's island of knights and knaves (knights always tell the truth, knaves always lie):
We meet natives A and B, A says: one of us is a knave.
`| ?- solve_puzzle(A says A is a knave or B is a knave).`
 - to make data structures more readable:
`acid(sulphur, h*2-s-o*4).`

Classical symbolic computation: symbolic derivation

- Write a Prolog predicate which calculates the derivative of a formula built from numbers and the atom `x` using some arithmetic operators.

% deriv(Formula, D): D is the derivative of Formula with respect to x.

```
deriv(x, 1).
```

```
deriv(C, 0) :-                number(C).
```

```
deriv(U+V, DU+DV) :-         deriv(U, DU), deriv(V, DV).
```

```
deriv(U-V, DU-DV) :-         deriv(U, DU), deriv(V, DV).
```

```
deriv(U*V, DU*V + U*DV) :-   deriv(U, DU), deriv(V, DV).
```

```
| ?- deriv(x*x+x, D).        =>    D = 1*x+x*1+1 ? ; no
```

```
| ?- deriv((x+1)*(x+1), D).  =>    D = (1+0)*(x+1)+(x+1)*(1+0) ? ; no
```

```
| ?- deriv(I, 1*x+x*1+1).    =>    I = x*x+x ? ; no
```

```
| ?- deriv(I, 2*x+1).        =>    no
```

```
| ?- deriv(I, 0).            =>    no
```

Contents

2 Declarative Programming with Prolog

- Prolog – first steps
- Prolog execution models
- The syntax of the (unsweetened) Prolog language
- Further control constructs
- Operators and special terms
- **Working with lists**
- Term ordering
- Higher order predicates
- All solutions predicates
- Efficient programming in Prolog
- Building and decomposing terms
- Executable specifications
- Block declarations
- Further reading

Concatenating lists

- Let $L1 \oplus L2$ denote the concatenation of $L1$ and $L2$, i.e. a list consisting of the elements of $L1$ followed by those of $L2$.
- Building $L1 \oplus L2$ in an imperative language
(A list is either a `NULL` pointer or a pointer to a head-tail structure):
 - Scan $L1$ until you reach a tail which is `NULL`
 - Overwrite the `NULL` pointer with $L2$
- If you still need the original $L1$, you have to copy it, replacing its final `NULL` with $L2$. A recursive definition of the \oplus (concatenation) function:

```
L1  $\oplus$  L2 =  if L1 == NULL return L2
              else L3 = tail(L1)  $\oplus$  L2
              return a new list structure whose head is head(L1)
                    and whose tail is L3
```

- Transform the above recursive definition to Prolog:

```
% app0(A, B, C): the conc(atenation) of A and B is C
app0([], L2, L2).           % The conc. of [] and L2 is L2.
app0([X|L1], L2, L) :-    % The conc. of [X|L1] and L2 is L if
    app0(L1, L2, L3),     % the conc. of L1 and L2 is L3 and
    L = [X|L3].           % L's head is X and L's tail is L3.
```

Efficient and multi-purpose concatenation

- Drawbacks of the `app0/3` predicate:
 - Uses “real” recursion (needs stack space proportional to length of `L1`)
 - Cannot split lists, e.g. `app0(L1, [3], [1, 3])` \rightsquigarrow infinite loop
- Apply a generic optimization: eliminate variable assignments
 - Remove goal `var = T`, and replace occurrences of variable `var` by `T`

Not applicable in the presence of disjunctions or if-then-else

- Apply this optimization to the second clause of `app0/3`:

```
app0([X|L1], L2, L) :- app0(L1, L2, L3), L = [X|L3].
```

- The resulting code (renamed to `app`, also available as the BIP `append/3`)

```
% app(A, B, C): The conc. of A and B is C, i.e. C = A⊕B
```

```
app([], L2, L2). % The conc. of [] and L2 is L2.
```

```
app([X|L1], L2, [X|L3]) :- % The conc. of [X|L1] and L2 is [X|L3] if  

  app(L1, L2, L3). % the conc. of L1 and L2 is L3.
```

- This uses constant stack space and can be used for multiple purposes, thanks to Prolog allowing **open ended** lists

Tail recursion optimization

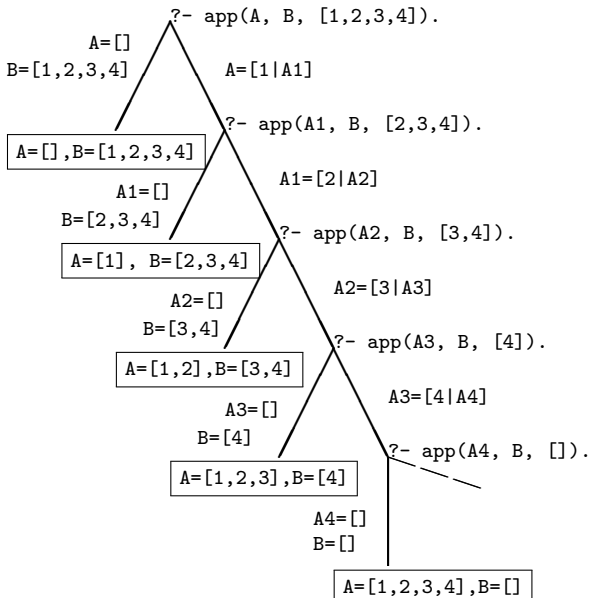
- Tail recursion optimization (TRO), or more generally last call optimization (LCO) is applicable if
 - the goal in question is the last to be executed in a clause body, and
 - no choice points exist in the given predicate.
- LCO is applicable to the recursive call of `app/3`:

```
app([], L, L).  
app([X|L1], L2, [X|L3]) :- app(L1, L2, L3).
```
- This feature relies on open ended lists:
 - It is possible to build a list node *before* building its tail
 - This corresponds to passing to `append` a pointer to the location where the resulting list should be stored.
- Open ended lists are possible because unbound variables are *first class* objects, i.e. unbound variables are allowed inside data structures. (This type of variable is often called the logic variable).

Splitting lists using append

```
% app(L1, L2, L3):
% L1 ⊕ L2 = L3.
app([], L, L).
app([X|L1], L2, [X|L3]) :-
    app(L1, L2, L3).
```

```
| ?- app(A, B, [1,2,3,4]).
A = [], B = [1,2,3,4] ? ;
A = [1], B = [2,3,4] ? ;
A = [1,2], B = [3,4] ? ;
A = [1,2,3], B = [4] ? ;
A = [1,2,3,4], B = [] ? ;
no
```



How does the “openness” of arguments affect `append(L1,L2,L3)` ?

- L2 is never decomposed (“looked inside”) by `append`, whether it is open ended, does not affect execution
- If L1 is closed, `append` produces at most one answer
 - | `?- append([a,b], Tail, L).` \implies `L = [a,b|Tail] ? ; no`
 - | `?- append([a,b], [c|T], L).` \implies `L = [a,b,c|T] ? ; no`
 - | `?- append([a,b], [c|T], [_,_ ,d,_]).` \implies `no`
- If L3 is closed (of length n), `append` produces at most $n + 1$ solutions, where L1 and L2 are closed lists (also see previous slide):
 - | `?- append(L1,L2,[1,2]).` \implies `L1=[], L2=[1,2] ? ; L1=[1], L2=[2] ? ; L1=[1,2], L2=[] ? ; no`
 - | `?- append([1,2], L, [1,2,3,4,5]).` \implies `L = [3,4,5] ? ; no`
 - | `?- append(L1,[4|L2],[1,2,3,4,5]).` \implies `L1 = [1,2,3], L2 = [5] ? ; no`
 - | `?- append(L1,[4,2],[1,2,3,4,5]).` \implies `no`
- The search may be **infinite**: if **both** the 1st **and** the 3rd arg. is open ended
 - | `?- append([1|L1], [a,b], L3).` \implies
 - `L1 = [], L3 = [1,a,b] ? ;`
 - `L1 = [_A], L3 = [1,_A,a,b] ? ;`
 - `L1 = [_A,_B], L3 = [1,_A,_B,a,b] ? ;` **ad infinitum** :-((((
 - | `?- append([1|L1], L2 , [2|L3]).` \implies `no`

Eight ways of using `append(L1,L2,L3)` (safe or unsafe)

- `:- mode append(+, +, +). % checking if $L1 \oplus L2 = L3$ holds`
`| ?- append([1,2], [3,4], [1,2,3,4]). \implies yes`
- `:- mode append(+, +, -). % appending L1 and L2 to obtain L3`
`| ?- append([1,2], [3,4], L3). \implies L3 = [1,2,3,4] ? ; no`
- `:- mode append(+, -, +). % checking if L1 is a prefix of L3, obtaining L2`
`| ?- append([1,2], L2, [1,2,3,4]). \implies L2 = [3,4] ? ; no`
- `:- mode append(+, -, -). % prepending L1 to an open ended L2 to obtain L3`
`| ?- append([1,2], [3|L2], L3). \implies L3 = [1,2,3|L2] ? ; no`
- `:- mode append(-, +, +). % checking if L2 is a suffix of L3 to obtain L1`
`| ?- append(L1, [3,4], [1,2,3,4]). \implies L1 = [1,2] ? ; no`
- `:- mode append(-, -, +). % splitting L3 to L1 and L2 in all possible ways`
`| ?- append(L1, L2, [1]). \implies L1=[],L2=[1] ? ; L1=[1],L2=[] ? ; no`
- `:- mode append(-, +, -). (see prev. slide) and :- mode append(-, -, -).`
`| ?- append(L1, L2, L3). \implies L1=[], L3=L2 ? ; L1=[A], L3=[A|L2] ? ;
L1=[A,B], L3=[A,B|L2] ? ...`

Variation on append — appending three lists

- Recall: `append/3` has **finite** search space, if its 1st **or** 3rd arg. is closed. `append(L,_,_)` completes in $\leq n + 1$ reduction steps when `L` has length n
- Let us define `append(L1,L2,L3,L123): L1 ⊕ L2 ⊕ L3 = L123`. First attempt:
`append(L1, L2, L3, L123) :-`
`append(L1, L2, L12), append(L12, L3, L123).`
 - Inefficient: `append([1,...,100],[1,2,3],[1], L)` – 203 and not 103 steps...
 - Not suitable for splitting lists – may create an infinite choice point
- An efficient version, suitable for splitting a given list to three parts:

```
% L1 ⊕ L2 ⊕ L3 = L123,
% where either both L1 and L2 are closed, or L123 is closed.
append(L1, L2, L3, L123) :-
    append(L1, L23, L123), append(L2, L3, L23).
```

- `L3` can be open ended or closed, it does not matter
- Note that in the first `append/3` call either `L1` or `L123` is closed.
If `L1` is closed, the first `append/3` produces an open ended list:

```
| ?- append([1,2], L23, L123).    ⇒    L123 = [1,2|L23]
```

The BIP $\text{length}/2$ – length of a list

- `length(?List, ?N)`: list `List` is of length `N`

```
| ?- length([4,3,1], Len).           Len = 3 ? ;
                                     no
| ?- length(List, 3).               List = [_A,_B,_C] ? ;
                                     no
| ?- length([[4,1,3],[2,8,7]], Len). Len = 2 ? ;
                                     no

| ?- length(L, N).                  L = [], N = 0 ? ;
                                     L = [_A], N = 1 ? ;
                                     L = [_A,_B], N = 2 ? ;
                                     L = [_A,_B,_C], N = 3 ? ...
```

- `length/2` has an infinite search space if the first argument is an open ended list and the second is a variable.

Appending a list of lists

- Library `lists` contains a predicate `append/2`

see e.g. <https://www.swi-prolog.org/search?for=append%2F2>

```
% append(LL, L): LL is a closed list of lists.
```

```
%                L is the concatenation of the elements of LL.
```

- Conditions for safe use (finite search space):

- Each element of `LL` is a closed list

```
| ?- append([[1,2],[3],[4,5]], L).  => L = [1,2,3,4,5] ? ; no
```

- `L` is a closed list

```
| ?- append([L1,L2,L3], [1,2]), L1 \= [],
```

```
    =>                L1 = [1],    L2 = [],    L3 = [2] ? ;
```

```
                L1 = [1],    L2 = [2],    L3 = [] ? ;
```

```
                L1 = [1,2], L2 = [],    L3 = [] ? ; no
```

- Finding a sublist matching a given pattern:

```
| ?- Pattern = [_A,_,_A], append([_Pref,Pattern,_],[1,2,3,2,1,2]),
```

```
    length(_Pref, Index).                % obtain the index of the Pattern
```

```
Pattern = [2,3,2], Index = 1 ? ;        % Index is zero-based
```

```
Pattern = [2,1,2], Index = 3 ? ; no
```

Finding list elements – BIP member/2

```
% member(E, L): E is an element of list L
member(Elem, [Elem|_]).                member1(Elem, [Head|Tail]) :-
member(Elem, [_|Tail]) :-              (   Elem = Head
    member(Elem, Tail).                  ;   member1(Elem, Tail)
                                        ).
```

- Mode member(+,+) – checking membership

```
| ?- member(2, [2,1,2]). => yes BUT
| ?- member(2, [2,1,2]), R=yes. => R = yes ? ; R = yes ? ; no
```

- Mode member(-,+) – enumerating list elements:

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

- Finding common elements of lists – with both above modes:

```
| ?- member(X, [1,2,3]),
    member(X, [5,4,3,2,3]). => X = 2 ? ; X = 3 ? ; X = 3 ? ; no
```

- Mode member(+,-) – making a term an element of a list (infinite choice):

```
| ?- member(1, L). => L = [1|_A] ? ; L = [_A,1|_B] ? ;
                    L = [_A,_B,1|_C] ? ; ...
```

- The search space of member/2 is **finite**, if the 2nd argument is closed.

Reversing lists

- Naive solution (quadratic in the length of the list)

```
% nrev(L, R): List R is the reverse of list L.
nrev([], []).
nrev([X|L], R) :-
    nrev(L, RL),
    append(RL, [X], R).
```

- A solution which is linear in the length of the list

```
% reverse(L, R): List R is the reverse of list L.
reverse(L, R) :- revapp(L, [], R).
```

```
% revapp(L1, L2, R): The reverse of L1 prepended to L2 gives R.
revapp([], R, R).
revapp([X|L1], L2, R) :-
    revapp(L1, [X|L2], R).
```

- In SICStus 4 `append/3` is a BIP, `reverse/2` is in library `lists`
- To load the library place this directive in your program file:


```
:- use_module(library(lists)).
```

append and revapp — building lists forth and back (ADVANCED)

• Prolog

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

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

• C++

```
struct link    { link *next;
                char elem;
                link(char e): elem(e) {}    };
typedef link *list;
```

```
list app(list L1, list L2)
{ list L3, *lp = &L3;
  for (list p=L1; p; p=p->next)
  { list newl = new link(p->elem);
    *lp = newl; lp = &newl->next;
  }
  *lp = L2; return L3;
}
```

```
list revapp(list L1, list L2)
{ list l = L2;
  for (list p=L1; p; p=p->next)
  { list newl = new link(p->elem);
    newl->next = l; l = newl;
  }
  return l;
}
```

Generalization of member: select/3 – defined in library lists

```
% select(E, List, Rest): Removing E from List results in list Rest.
select(E, [E|Rest], Rest).      % The head is removed, the tail remains.
select(E, [X|Tail], [X|Rest]):- % The head remains,
    select(E, Tail, Rest).      % the element is removed from the Tail.
```

Possible uses:

```
| ?- select(1, [2,1,3,1], L).      % Remove a given element
    L = [2,3,1] ? ; L = [2,1,3] ? ; no
| ?- select(X, [1,2,3], L).      % Remove an arbitrary element
    L=[2,3], X=1 ? ; L=[1,3], X=2 ? ; L=[1,2], X=3 ? ; no
| ?- select(3, L, [1,2]).      % Insert a given element!
    L = [3,1,2] ? ; L = [1,3,2] ? ; L = [1,2,3] ? ; no
| ?- select(3, [2|L], [1,2,7,3,2,1,8,9,4]).
    no                               % Can one remove 3 from [2|L]
                                       % to obtain [1,...]?
| ?- select(1, [X,2,X,3], L).
    L = [2,1,3], X = 1 ? ; L = [1,2,3], X = 1 ? ; no
```

- The search space of select/3 is **finite**, if the 2nd **or** the 3rd arg. is closed.

Permutation of lists – two solutions (ADVANCED)

`perm(+List, ?Perm)`: The list `Perm` is a permutation of `List`

```
perm0([], []).
perm0(L, [H|P]) :-
    select(H, L, R),           % Select H from L as the head of the output, R remaining.
    perm0(R, P).              % Permute R to become P, the tail of the output list.

| ?- perm0([a,b,c], L).
L = [a,b,c] ? ; L = [a,c,b] ? ; L = [b,a,c] ? ;
L = [b,c,a] ? ; L = [c,a,b] ? ; L = [c,b,a] ? ; no

perm1([], []).
perm1([H|T], P) :-
    perm1(T, P1),             % Permute T, the tail of the input list, obtaining P1.
    select(H, P, P1).         % Insert H, the head of the input list, into an arbitrary
    % mode:+ - +              % position within P1 to obtain the output list, P.

| ?- perm1([a,b,c], L).
L = [a,b,c] ? ; L = [b,a,c] ? ; L = [b,c,a] ? ;
L = [a,c,b] ? ; L = [c,a,b] ? ; L = [c,b,a] ? ; no
```

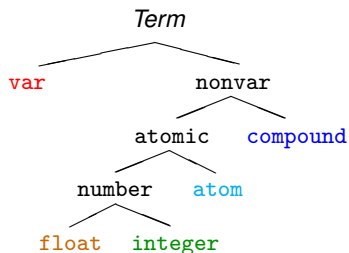
- `perm` is symmetric, so the two predicates have the same meaning (WHAT)
- But the second variant is much faster!

Contents

2 Declarative Programming with Prolog

- Prolog – first steps
- Prolog execution models
- The syntax of the (unsweetened) Prolog language
- Further control constructs
- Operators and special terms
- Working with lists
- **Term ordering**
- Higher order predicates
- All solutions predicates
- Efficient programming in Prolog
- Building and decomposing terms
- Executable specifications
- Block declarations
- Further reading

Principles of Prolog term ordering \prec



Different kinds ordered left-to-right:

$\text{var} \prec \text{float} \prec \text{integer} \prec$
 $\prec \text{atom} \prec \text{compound}$

- Ordering of variables: system dependent
- Ordering of floats and integers: usual ($x \prec y \Leftrightarrow x < y$)
- Ordering of atoms: lexicographical ($abc \prec abcd$, $abcv \prec abcz$)
- Compound terms: $\text{name}_a(a_1, \dots, a_n) \prec \text{name}_b(b_1, \dots, b_m)$ iff
 - $n < m$, e.g. $p(x, s(u, v, w)) \prec a(b, c, d)$, or
 - $n = m$, and $\text{name}_a \prec \text{name}_b$ (lexicographically), e.g. $a(x, y) \prec p(b, c)$, or
 - $n = m$, $\text{name}_a = \text{name}_b$, and for the first i where $a_i \neq b_i$, $a_i \prec b_i$, e.g. $r(1, u+v, 3, x) \prec r(1, u+v, 5, a)$

Built-in predicates for comparing Prolog terms

- Comparing two Prolog terms:

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

- The comparison predicates are not purely logical:

| ?- $X @< 3, X = 4. \implies X = 4$

| ?- $X = 4, X @< 3. \implies \text{no}$

as they rely on the **current instantiation** of their arguments

- Comparison uses, of course, the canonical representation:

| ?- $[1, 2, 3, 4] @< s(1,2,3). \implies \text{yes}$

- BIP $\text{sort}(L, S)$ sorts (using $@<$) a list L of arbitrary Prolog terms, removing duplicates (w.r.t. $==$). Thus the result is a strictly increasing list S .

| ?- $\text{sort}([1, 2.0, s(a,b), s(a,c), s, X, s(Y), t(a), s(a), 1, X], L).$

$L = [X, 2.0, 1, s, s(Y), s(a), t(a), s(a,b), s(a,c)] ?$

Equality-like Prolog predicates – a summary

Recall: a Prolog term is *ground* if it contains no unbound variables

- | | |
|--|--|
| <ul style="list-style-type: none"> • $U = V$: U unifies with V
No errors. May bind vars. | <ul style="list-style-type: none"> ?- $X = 1+2.$ \implies $X = 1+2$?- $3 = 1+2.$ \implies no |
| <ul style="list-style-type: none"> • $U == V$: U is identical to V, i.e. $U=V$ succeeds with no bindings
No errors, no bindings. | <ul style="list-style-type: none"> ?- $X == 1+2.$ \implies no ?- $3 == 1+2.$ \implies no ?- $+(X,Y)==X+Y$ \implies yes |
| <ul style="list-style-type: none"> • $U ::= V$: The value of U is arithmetically equal to that of V.
No bindings. Error if U or V is not a (ground) arithmetic expression. | <ul style="list-style-type: none"> ?- $X ::= 1+2.$ \implies error ?- $1+2 ::= X.$ \implies error ?- $2+1 ::= 1+2.$ \implies yes ?- $3.0 ::= 1+2.$ \implies yes |
| <ul style="list-style-type: none"> • U is V: U is unified with the value of V.
Error if V is not a (ground) arithmetic expression. | <ul style="list-style-type: none"> ?- X is $1+2.$ \implies $X = 3$?- 3.0 is $1+2.$ \implies no ?- $1+2$ is $X.$ \implies error ?- 3 is $1+2.$ \implies yes ?- $1+2$ is $1+2.$ \implies no |

Nonequality-like Prolog predicates – a summary

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

- $U \neq V$: U does not unify with V .
No errors.

?- $X \neq 1+2$.	\implies	no
?- $X \neq 1+2, X = 1$.	\implies	no
?- $X = 1, X \neq 1+2$.	\implies	yes
?- $+(1,2) \neq 1+2$.	\implies	no

- $U \neq= V$: U is not identical to V .
No errors.

?- $X \neq= 1+2$.	\implies	yes
?- $X \neq= 1+2, X=1+2$.	\implies	yes
?- $3 \neq= 1+2$.	\implies	yes
?- $+(1,2) \neq= 1+2$	\implies	no

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

?- $X = \neq 1+2$.	\implies	error
?- $1+2 = \neq X$.	\implies	error
?- $2+1 = \neq 1+2$.	\implies	no
?- $2.0 = \neq 1+1$.	\implies	no

(Non)equality-like Prolog predicates – examples

		Unification		Identical terms		Arithmetic		
U	V	$U = V$	$U \backslash = V$	$U == V$	$U \backslash == V$	$U ::= V$	$U =\backslash = V$	$U \text{ is } V$
1	2	<i>no</i>	yes	<i>no</i>	yes	<i>no</i>	yes	<i>no</i>
a	b	<i>no</i>	yes	<i>no</i>	yes	error	error	error
1+2	+(1,2)	yes	<i>no</i>	yes	<i>no</i>	yes	<i>no</i>	<i>no</i>
1+2	2+1	<i>no</i>	yes	<i>no</i>	yes	yes	<i>no</i>	<i>no</i>
1+2	3	<i>no</i>	yes	<i>no</i>	yes	yes	<i>no</i>	<i>no</i>
3	1+2	<i>no</i>	yes	<i>no</i>	yes	yes	<i>no</i>	yes
X	1+2	X=1+2	<i>no</i>	<i>no</i>	yes	error	error	X=3
X	Y	X=Y	<i>no</i>	<i>no</i>	yes	error	error	error
X	X	yes	<i>no</i>	yes	<i>no</i>	error	error	error

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

Contents

2 Declarative Programming with Prolog

- Prolog – first steps
- Prolog execution models
- The syntax of the (unsweetened) Prolog language
- Further control constructs
- Operators and special terms
- Working with lists
- Term ordering
- **Higher order predicates**
- All solutions predicates
- Efficient programming in Prolog
- Building and decomposing terms
- Executable specifications
- Block declarations
- Further reading

Higher order programming: using predicates as arguments

- Example: collect all nonzero elements of a list

```
% nonzero_elems(Xs, Ys): Ys is a list of all nonzero elements of Xs
nonzero_elems([], []).
nonzero_elems([X|Xs], Ys) :-
    ( 0 \= X -> Ys = [X|Ys1]
    ;   Ys = Ys1
    ),
    nonzero_elems(Xs, Ys1).
```

- Generalize to a predicate where the **condition** is given as an argument

```
% include(Pred, Xs, Ys): Ys = list of elems of Xs that satisfy Pred
include(_Pred, [], []).
include(Pred, [X|Xs], Ys) :-
    ( call(Pred, X) -> Ys = [X|Ys1]
    ;   Ys = Ys1
    ),
    include(Pred, Xs, Ys1).
```

- Specialize `include` for collecting nonzero elements:

```
nonz(X) :- 0 \= X.
nonzero_elems(L, L1) :- include(nonz, L, L1).
```

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*
- A **partial goal** is a goal with the last few arguments missing
 - e.g., a predicate name is a partial goal
(hence variable name `Pred` is often used for partial goals)
- The BIP `call(PG, X)`, where `PG` is a partial goal, adds `X` as the last argument to `PG` and executes this new goal:
 - if `PG` is an atom \Rightarrow it calls `PG(X)`, e.g. `call(number, X) \equiv number(X)`
 - if `PG` is a compound `Pred(A1, ..., An)` \Rightarrow it calls `Pred(A1, ..., An, X)`, e.g. `call(\=(0), X) \equiv \=(0,X) \equiv 0 \= X`
- Predicate `include(Pred, L, FL)` is in `library(lists)`

```
| ?- L=[1,2,a,X,b,0,3+4],
      include(number, L, Nums).    % Nums = { X  $\in$  L | number(X) }
Nums = [1,2,0] ? ; no
| ?- L=[0,2,0,3,-1,0],
      include(\=(0), L, NZs).     % NZs = { X  $\in$  L | \=(0,X) }
NZs = [2,3,-1] ?
```

Calling predicates with additional arguments

- Recall: a **callable term** is a compound or atom.
- There is a group of built-in predicates `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, A1, ..., An)`: Adds `A1, ..., An` 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.

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

```
| ?- include(even, [1,3,2,9,6,4,0], FL).
           ⇒      FL = [2,6,4,0] ; no
```

```
divisible_by(N, X) :- X mod N == 0.
```

```
| ?- include(divisible_by(3), [1,3,2,9,6,4,0], FL).
           ⇒      FL = [3,9,6,0] ; no
```

- In descriptions we often abbreviate `call(PG, A1, ..., An)` to `PG(A1, ..., An)`

An important higher order predicate: `maplist/3`

- `maplist(:PG, ?L, ?ML)`: for each X element of L and the **corresponding** Y element of ML , `call(PG, X, Y)` holds, where PG is a partial goal requiring two additional arguments
- Annotation “:” (as in `:PG` above) marks a **meta** argument, i.e. a term to be interpreted as a goal or a partial goal

```
maplist(_PG, [], []).
maplist(PG, [X|Xs], [Y|Ys]) :-
    call(PG, X, Y),
    maplist(PG, Xs, Ys).
```

```
| ?- maplist(reverse, [[1,2],[3,4]], LL). => LL = [[2,1],[4,3]] ? ; no
```

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

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

```
| ?- maplist(square, [1,2,3,4], L). => L = [1,4,9,16] ? ; no
```

```
| ?- maplist(mult(2), [1,2,3,4], L). => L = [2,4,6,8] ? ; no
```

```
| ?- maplist(mult(-5), [1,2,3], L). => L = [-5,-10,-15] ? ; no
```

Variants of `maplist`

In SICStus, `maplist` can also be used with 2 and 4 arguments

- `maplist(:Pred, +Xs)` is true if for each x element of Xs , $\text{Pred}(x)$ holds.
- Example: check if a condition holds for all elements of a list

```
all_positive(Xs) :-          % all elements of Xs are positive
    maplist(<(0), Xs).      %  $\forall X \in Xs, <(0, X), \text{i.e. } 0 < X \text{ holds}$ 
```

- `maplist(:Pred, ?Xs, ?Ys, ?Zs)` is true when Xs , Ys , and Zs are lists of equal length, and $\text{Pred}(X, Y, Z)$ is true for corresponding elements x of Xs , y of Ys , and z of Zs . At least one of Xs , Ys , Zs has to be a closed list.
- Example: add two vectors

```
add_vectors(VA, VB, VC) :-
    maplist(plus, VA, VB, VC).          plus(A, B, C) :- C is A+B.
| ?- add_vectors([10,20,30], [3,2,1], V).  $\implies V = [13,22,31]$  ? ; no
```

- The implementation of `maplist/4` (easy to generalize :-):

```
maplist(_PG, [], [], []).
maplist(PG, [X|Xs], [Y|Ys], [Z|Zs]) :-
    call(PG, X, Y, Z), maplist(PG, Xs, Ys, Zs).
```


Another important higher order predicate: `scanlist` (SWI: `fold1`)

- Example:


```
plus(A, S0, S) :- S is S0+A.
| ?- scanlist(plus, [1,3,5], 0, Sum).    => Sum = 9 ? ; no
      % 0+1+3+5 = 9
```

This executes as: `plus(0, 1, S1)`, `plus(S1, 3, S2)`, `plus(S2, 5, Sum)`.

- In general: `scanlist(acc, [E1,E2,...,En], S0, Sn)` is expanded as:


```
acc(S0, E1, S1), acc(S1, E2, S2), ..., acc(Sn-1, En, Sn)
```
- `scanlist(:PG, ?L, ?Init, ?Final)`:
 - `PG` represents the above accumulating predicate `acc`
 - `scanlist` applies the `acc` predicate repeatedly, on all elements of list `L`, left-to-right, where `Init` = `S0` and `Final` = `Sn`.
- For processing two lists (of the same length), use `scanlist/5`, e.g.


```
prodsum(A, B, PS0, PS) :- PS is PS0 + A*B.
scalar_product(As, Bs, SP) :- scanlist(prodsum, As, Bs, 0, SP).
| ?- scalar_product([1,0,2], [3,4,5], SP).    => SP = 13 ? ; no
```
- In SICStus, there is also a `scanlist/6` predicate, for processing 3 lists

Contents

2 Declarative Programming with Prolog

- Prolog – first steps
- Prolog execution models
- The syntax of the (unsweetened) Prolog language
- Further control constructs
- Operators and special terms
- Working with lists
- Term ordering
- Higher order predicates
- **All solutions predicates**
- Efficient programming in Prolog
- Building and decomposing terms
- Executable specifications
- Block declarations
- Further reading

All solutions built-in predicates – introduction

- All solution BIPs are higher order predicates analogous to list comprehensions in Haskell, Python, etc.
- There are three such predicates: `findall/3` (the simplest), `bagof/3` and `setof/3`; having the same arguments, but somewhat different behavior
- Examples for `findall/3`:

```
| ?- findall(X, (member(X, [1,7,8,3,2,4]), X > 3), L).
```

```
% {X | X ∈ {1,7,8,3,2,4}, X > 3} = L
```

```
⇒ L = [7,8,4] ? ; no
```

```
| ?- findall(X, (member(X, [1,7,8,3,2,4]), X > 8), L).
```

```
% {X | X ∈ {1,7,8,3,2,4}, X > 8} = L
```

```
⇒ L = [] ? ; no
```

```
| ?- findall(X-Y, (between(1, 3, X), between(1, X, Y)), L).
```

```
% {X-Y | 1 ≤ X ≤ 3, 1 ≤ Y ≤ X} = L
```

```
⇒ L = [1-1,2-1,2-2,3-1,3-2,3-3] ? ; no
```

Recall: `between(+N, +M, ?X)` enumerates in `X` the integers `N, N+1, ..., M`.
In SICStus, it requires loading `library(between)`.

Finding all solutions: the BIP `findall(?Temp1, :Goal, ?L)`

Approximate meaning: `L` is a list of `Temp1` terms for each solution of `Goal`

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 `Temp1` (`copy` \implies replace vars in `Temp1` by new ones)
Note that copying requires time proportional to the size of `Temp1`
 - continue with failure (to enumerate further solutions)
- When there are no more solutions (`Goal` fails)
 - collect the stored `Temp1` values into a list, unify it with `L`.
- When a solution contains (possibly multiple instances of) a variable (e.g. `A`), then each of these will be replaced by a single new variable (e.g. `_A`):

```
| ?- findall(T, member(T, [A-A,B-B,A]), L).
```

```
     $\implies$  L= [_A-_A,_B-_B,_C] ? ; no
```

All solutions: the BIP bagof(?Temp1, :Goal, ?L)

- Exactly the same arguments as in findall/3.

bagof/3 is the same as findall/3, except when there are unbound variables in Goal which do not occur in Temp1 (so called **free** variables)

```
% emp(Er, Ee): employer Er employs employee Ee.
```

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

```
| ?- findall(E, emp(R, E), Es). % Es ≡ the list of all employees
```

```
⇒ Es = [b,c,c,d] ? ; no i.e. Es = {E | ∃ R. (R employs E)}
```

- bagof does not treat free vars as existentially quantified. Instead it **enumerates** all possible values for the free vars (all employers) and for each such choice it builds a separate list of solutions:

```
| ?- bagof(E, emp(R,E), Es). % Es ≡ list of Es employed by any possible R.
```

```
⇒ R = a, Es = [b,c] ? ;
```

```
⇒ R = b, Es = [c,d] ? ; no
```

- Use operator \wedge to achieve existential quantification in bagof:

```
| ?- bagof(E, R^emp(R, E), Es). % Collect Es for which ∃R. emp(R, E)
```

```
⇒ Es = [b,c,c,d] ? ; no
```

- bagof preserves variables (but it is slower than findall :-()):

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

All solutions: the BIP setof/3

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

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

```
% Graph has a node V.
```

```
has_node(Graph, V) :- member(A-B, Graph), ( V = A ; V = B ).
```

```
% The set of nodes of G is Vs.
```

```
graph_nodes(G, Vs) :- setof(V, has_node(G, V), Vs).
```

```
| ?- graph(_G), graph_nodes(_G, Vs).  $\implies$  Vs = [a,b,c,d] ? ; no
```

Contents

2 Declarative Programming with Prolog

- Prolog – first steps
- Prolog execution models
- The syntax of the (unsweetened) Prolog language
- Further control constructs
- Operators and special terms
- Working with lists
- Term ordering
- Higher order predicates
- All solutions predicates
- **Efficient programming in Prolog**
- Building and decomposing terms
- Executable specifications
- Block declarations
- Further reading

Causes of inefficiency – preview

- **Unnecessary choice points** (ChPs) waste both time and space
Recursive definitions often leave choice points behind on exit, e.g.:
 - `% fact0(+N, ?F): F = N!.`
`fact0(0, 1).`
`fact0(N, F) :- N > 0, N1 is N-1, fact0(N1, F1), F is N*F1.`
 - **Remedy:** use **if-then-else** or the **cut** BIP (coming soon)
 - `% last0(L, E): The last element of L is E.`
`last0([E], E).`
`last0([_|L], E) :- last0(L, E).`
 - **Remedy:** rewrite to make use of **indexing** (or cut, or if-then-else)
- **General recursion**, as opposed to tail recursion
As an example, see the `fact0/2` predicate above
Remedy: re-formulate to a **tail recursive** form, using **accumulators**

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`.
- Execution: the cut always succeeds with these two side effects:
 - **Restrict to the first solution of a goal:**
Remove all choice points created within the goal(s) preceding the `!`.

```
% is_a_parent(+P): check if a given P is a parent.
is_a_parent(P) :- has_parent(_, P), !.
```
 - **Commit to the clause containing the cut:**
Remove the choice of any further clauses in the current predicate.

```
fact1(0, F) :- !, F = 1. % Assign output vars only after the cut,
                        % both for correctness and efficiency
fact1(N, F) :- N > 0, N1 is N-1, fact1(N1, F1), F is N*F1.
```
- Definition: if `q :- ..., p, ...` then the **parent goal** of `p` is the goal matching the clause head `q`
- Effects of cut in the search tree: removes all choice points up to and including the node labelled with the **parent goal of the cut**.
- In the procedure box model: Fail port of cut \implies Fail port of parent goal

How does “cut” prune the search tree – an example

```

a(X, Y) :- b(X), c(X, Y).
a(X, Y) :- d(X, Y).

c(s(X), Y) :- Y is X+10.
c(s(X), Y) :- Y is X+20.

a_cut(X, Y) :- b(X), !, c(X, Y).
a_cut(X, Y) :- d(X, Y).

test(Pred, X, Res) :-
    findall(X-Y, call(Pred, X, Y), Res).

```

Sample runs:

```

| ?- test(a, s(_), Res). => Res = [s(1)-11,s(1)-21,s(2)-12,
                                   s(2)-22,s(3)-30] ?
| ?- test(a, t(_), Res). => Res = [t(4)-40] ?
| ?- test(a_cut, s(_), Res). => Res = [s(1)-11,s(1)-21] ?
| ?- test(a_cut, s(3), Res). => Res = [s(3)-30] ?
| ?- test(a_cut, t(_), Res). => Res = [t(4)-40] ?

```

Avoid leaving unnecessary choice points

- Add a cut if you know that remaining branches are doomed to fail. (These are so called **green** cuts, which do not remove solutions.)
- Example of a green cut:

```
% last1(L, E): The last element of L is E.
```

```
last1([E], E) :- !.
```

```
last1(_|L, E) :- last1(L, E).
```

In the absence of the cut, the goal `last1([1], X)` will return the answer `X = 1`, and leave a choice point. When this choice point is explored `last1([], X)` will be called which will always fail.

- Instead of a cut, one can use if-then-else:

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

```
fact2(N, F) :- ( N == 0 -> F = 1
               ; N > 0, N1 is N-1, fact2(N1, F1), F is N*F1
               ).
```

Avoid leaving unnecessary choice points – indexing

- Recall a simple example predicate, summing a binary tree:

```
% tree_sum(+Tree, ?Sum):
% Sum is the sum of integers in the leaves of Tree.
tree_sum(leaf(Value), Value).           1st head arg's functor: leaf/1
tree_sum(node(Left, Right), S) :-      1st head arg's functor: node/2
    tree_sum(Left, S1), tree_sum(Right, S2), S is S1+S2.
```

- Indexing groups the clauses of a predicate based on the outermost functor of (usually) the first argument.
- The compiler generates code (using hashing) to select the subset of clauses that corresponds to this outermost functor.
- If the subset contains a single clause, no choicepoint is created. (This is the case in the above example.)

SICStus specific: avoid choice points in if-then-else (ADVANCED)

- Consider an if-then-else goal of the form: `(cond -> then ; else)`.
- Before `cond`, a ChP is normally created (removed at `->` or before `else`).
- In **SICStus Prolog** no choice points are created, if `cond` only contains:
 - arithmetical comparisons (e.g., `<`, `=<`, `==`); and/or
 - built-in predicates checking the term type (e.g., `atom`, `number`); and/or
 - general comparison operators (e.g., `@<`, `@=<`, `==`).
- Analogously, no ChPs are made for `head :- cond, !, then.,` if all arguments of `head` are distinct variables, and `cond` is just like above.
- Further improved variants of `fact2` and `last2` with no ChPs created:

```
fact3(N, F) :-      ( N == 0 -> F = 1      % used to be N = 0
                    ; N > 0, N1 is N-1, fact(N1, F1), F is N*F1
                    ).
```

```
last3([E|L], X) :- ( L == [] -> X = E      % used to be L = []
                    ; last3(L, X)
                    ).
```

Indexing – an introductory example

- A sample (meaningless) program to illustrate indexing.

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) */		

- For the call `p(A, B)`, the **compiler** produces a **case statement**-like construct, to determine the list of applicable clauses:

(VAR)	if A is a variable:	(1) (2) (3) (4) (5)
(0/0)	if A = 0 (A's main functor is 0/0):	(1) (2)
(s/1)	if A's main functor is s/1:	(2) (3) (4)
(9/0)	if A = 9:	(2) (5)
(OTHER)	in all other cases:	(2)

- Example calls (do they create and leave a choice point?)
 - `p(1, Y)` takes branch (OTHER), does not create a choice point.
 - `p(s(1), Y)` takes branch (s/1), creates a choice point, but removes it and exits without leaving a choice point.
 - `p(s(0), Y)` takes branch (s/1), and exits leaving a choice point.

Indexing

- Indexing improves the efficiency of Prolog execution by
 - speeding up the selection of clauses matching a particular call;
 - using a **compile-time** grouping of the clauses of the predicate.
- Most Prolog systems, including SICStus, use only the main (i.e. outermost) functor of the *first* argument for indexing, which is
 - C/O, if the argument is a constant (atom or number) C;
 - R/N, if the argument is a compound with name R and arity N;
 - undefined, if the argument is a variable.

Implementing indexing

- Compile-time: collect the set of (outermost) functors of nonvar terms occurring as first args, build the **case statement** (see prev. slide)
- Run-time: select the relevant clause list using the first arg. of the call. This is practically a constant time operation, as it uses *hashing*.
 - If the clause list is a singleton, *no choice point* is created.
 - Otherwise a choice point *is* created, which will be removed before entering the **last** branch.

Getting the most out of indexing

- Get deep indexing through helper predicates (rewrite $p/2$ to $q/2$):

$$\begin{array}{l}
 p(0, a). \\
 p(s(0), b). \\
 p(s(1), c). \\
 p(9, z).
 \end{array}
 \implies
 \left|
 \begin{array}{l}
 q(0, a). \\
 q(s(X), Y) :- \\
 \quad q_aux(X, Y). \\
 q(9, z).
 \end{array}
 \right|
 \left|
 \begin{array}{l}
 q_aux(0, b). \\
 q_aux(1, c).
 \end{array}
 \right|$$

Pred. $q(X, Y)$ will not create choice points if x is ground.

- Indexing does not deal with arithmetic comparisons
 - E.g., $N = 0$ and $N > 0$ are not recognized as mutually exclusive.
- Indexing and lists
 - Putting the (input) list in the first argument makes indexing work.
 - Indexing distinguishes between $[]$ and $[\dots | \dots]$ (resp. functors: $'[]'$ / 0 and $'.'$ / 2).
 - For proper lists, the order of the two clauses is not relevant
 - For use with open ended lists: put the clause for $[]$ first, to avoid an infinite loop (an infinite choice may still remain)

Indexing list handling predicates

- Predicate `app/3` creates no choice points if the first argument is a proper list:

```
% app(L1, L2, L3): L1 ⊕ L2 = L3.                                % 1st arg funct:
app([], L, L).                                               % []/0
app([X|L1], L2, [X|L3]) :-                                   % . /2
    app(L1, L2, L3).
```

- The same is true for `revapp/3`:

```
% revapp(L1, L2, L3):
% appending the reverse of L1 and L2 gives L3
revapp([], L, L).                                           % []/0
revapp([X|L1], L2, L3) :-                                   % . /2
    revapp(L1, [X|L2], L3).
```

Indexing list handling predicates, cont'd

- Getting the last element of a list: `last0/2` leaves a choice point.

```
% last0(L, E): The last element of L is E.
```

```
last0([H], H). % . /2
```

```
last0(_|T, E) :- last0(T, E). % . /2
```

- The variant `last4/2` uses a helper predicate, creates no choice points:

```
last4([H|T], E) :- last4(T, H, E). (*)
```

```
% last4(T, H, E): The last element of [H|T] is E.
```

```
last4([], E, E). % []/0
```

```
last4([H|T], _, E) :- last4(T, H, E). % . /2
```

- `member0/2` (as defined earlier) always leaves a choice point.

```
% member0(E, L): E is an element of L.
```

```
member0(E, [E|_T]). % VAR
```

```
member0(E, [_H|T]) :- member0(E, T). % VAR
```

- Write the head comment and the clauses of `member1/3`, so that `member1/2` leaves no choice point when the last element of a (proper) list is returned.

```
member1(E, [H|T]) :- member1(T, H, E). % cf. (*)
```

```
% member1(T, H, E): ...
```

Tail recursion

- In general, recursion is expensive both in terms of time and space.
- The special case of **tail recursion** can be compiled to a loop. Conditions:
 - ① the recursive call is the last to be executed in the clause body, i.e.:
 - it is textually the last subgoal in the body; or
 - the last subgoal is a disjunction/if-then-else, and the recursive call is the last in one of the branches
 - ② no ChPs left in the predicate when the recursive call is reached

- Example

```
% all_pos(+L): all elements of number list L are positive.
all_pos([]).
all_pos([X|L]) :-
    X > 0, all_pos(L).
```

- *Tail recursion optimization, TRO*: the memory allocated by the clause is freed **before** the last call is executed.
- This optimization is performed not only for recursive calls but for the **last** calls in general (*last call optimization, LCO*).

Making a predicate tail recursive – accumulators

- Example: the sum of a list of numbers. The left recursive variant:

```
% sum0(+List, -Sum): the sum of the elements of List is Sum.
sum0([], 0).
sum0([X|L], Sum) :- sum0(L, Sum0), Sum is Sum0+X.
```

Note that $\text{sum0}([a_1, \dots, a_n], S) \implies S = 0 + a_n + \dots + a_1$ (right to left)

- For TRO, define a helper pred, with an arg. storing the “sum so far”:

```
% sum(+List, +Sum0, -Sum):
% ( $\sum$  List) + Sum0 = Sum, i.e.  $\sum$  List = Sum - Sum0.
sum([], Sum, Sum).
sum([X|L], Sum0, Sum) :-
    Sum1 is Sum0+X,    % Increment the ‘sum so far’
    sum(L, Sum1, Sum). % recurse with the tail and the new sum so far
```

- Arguments `Sum0` and `Sum` form an **accumulator pair**: `Sum0` is an intermediate while `Sum` is the final value of the accumulator.

The initial value is supplied when defining `sum/2`:

```
% sumlist(+List, ?Sum):  $\sum$  List = Sum. Available from library(lists).
sumlist(List, Sum) :- sum(List, 0, Sum).
```

Note that $\text{sumlist}([a_1, \dots, a_n], S) \implies S = 0 + a_1 + \dots + a_n$ (left to right)

Accumulators – making factorial tail-recursive

- Two arguments of a pred. forming an **accumulator** pair: the declarative equivalent of the imperative variable (i.e. a variable with a mutable state)
- The two parts: the state of the mutable quantity at pred. entry and exit.
- Example: making factorial tail-recursive. The mid-recursive version:

```
% fact0(N, F): F = N!.
```

```
fact0(N, F) :-      (   N == 0 -> F = 1
                    ;   N > 0, N1 is N-1, fact0(N1, F1), F is F1*N
                    ).
```

```
| ?- fact0(4, F). => F = 24 ~ 1*1*2*3*4
```

- Helper predicate: `fact(N, F0, F)`, `F0` is the product accumulated so far.

```
% fact(N, F0, F): F = F0*N!.
```

```
fact(N, F0, F) :-  (   N == 0 -> F = F0
                    ;   N > 0, F1 is F0*N, N1 is N-1, fact(N1, F1, F)
                    ).
```

```
fact(N, F) :-
    fact(N, 1, F).
```

```
| ?- fact(4, F). => F = 24 ~ 1*4*3*2*1
```

Accumulating lists – higher order approaches (ADVANCED)

- Recap predicate `revapp/3`:

% revapp(L, R0, R): The reverse of L prepended to R0 gives R.

```
revapp0([], R0, R) :- R = R0.
```

```
revapp0([X|L], R0, R) :- R1 = [X|R0], revapp0(L, R1, R).
```

- Introduce the list construction predicate `cons/3`

% L1 is a list constructed from the head X and tail L0.

```
cons(X, L0, L1) :- L1 = [X|L0].
```

```
revapp1([], R0, R) :- R = R0.
```

```
revapp1([X|L], R0, R) :- cons(X, R0, R1), revapp1(L, R1, R).
```

- A higher order (HO) solution (in SWI use `foldl` instead of `scanlist`):

```
revapp2(L, R0, R) :- scanlist(cons, L, R0, R).
```

- Summing a list, HO solution (*% sum2(L, Sum): list L sums to Sum.*)

```
plus(X, S0, S1) :- S1 is S0+X.
```

```
sum2(L, Sum) :- scanlist(plus, L, 0, Sum).
```

- (ADV²) Appending lists, HO sol. (*% app(L1, L2, L): L1 ⊕ L2 = L.*)

% decomp(X, C, B): List C can be decomposed to head X and tail B

```
decomp(X, C, B) :- C = [X|B].
```

```
app(A, B, C) :- scanlist(decomp, A, C, B).
```

Accumulating lists – avoiding append

- Example: calculate the list of leaf values of a tree. Without accumulators:

```
% tree_list0(+T, ?L): L is the list of the leaf values of tree T.
tree_list0(leaf(Value), [Value]).
tree_list0(node(Left, Right), L) :-
    tree_list0(Left, L1), tree_list0(Right, L2), append(L1, L2, L).
```

- Building the list of tree leaves using accumulators:

```
tree_list(Tree, L) :-
    tree_list(Tree, [], L). % Initialize the list accumulator to []
% tree_list(+Tree, +L0, L): The list of the
% leaf values of Tree prepended to L0 is L.
tree_list(leaf(Value), L0, L) :- L = [Value|L0].
tree_list(node(Left, Right), L0, L) :-
    tree_list(Right, L0, L1), tree_list(Left, L1, L).
| ?- tree_list(node(node(leaf(a),leaf(b)),leaf(c)), L).  $\implies$  L = [a,b,c]? ; no
```

- Note that one of the two recursive calls is tail-recursive.
- Also, there is no need to append the intermediate lists!

Accumulators for implementing imperative (mutable) variables

- Let $L = [x_1, \dots,]$ be a number list. x_i is *left-visible* in L , iff $\forall j < i. (x_j < x_i)$
- Determine the count of left-visible elements in a list of **positive** integers:

Imperative, C-like algorithm

```
int viscnt(list L) {
    int MV = 0; // max visible
    int VC = 0; // visible cnt

loop:
    if (empty(L)) return VC;

    { int H = hd(L), L = tl(L);
      if (H > MV)
          { VC += 1; MV = H; }
          // else VC,MV unchanged
    }
    goto loop;
}
```

Prolog code

```
% List L has VC left-visible elements.
viscnt(L, VC) :- viscnt(L,
                        0,
                        0, VC).

% viscnt(L, MV, VCO, VC): L has VC-VCO
% left-visible elements which are > MV.
viscnt([], _, VCO, VC) :- VC = VCO.
viscnt(L0, MVO, VCO, VC) :- % (1)
    L0 = [H|L1],
    ( H > MVO
    -> VC1 is VCO+1, MV1 = H
    ; VC1 = VCO, MV1 = MVO % (2)
    ),
    viscnt(L1, MV1, VC1, VC). % (3)
```


Mapping a C loop to a Prolog predicate

- Each C variable initialized before the loop and used in it becomes an input argument of the Prolog predicate
- Each C variable assigned to in the loop and used afterwards becomes an output argument of the Prolog predicate
- Each **occurrence** of a C variable is mapped to a Prolog variable, whenever the variable is assigned, a new Prolog variable is needed, e.g. `MV` is mapped to `MV0`, `MV1`, ... :
 - The initial values (`L0`, `MV0`, ...) are the args of the clause head² (1)
 - If a branch of if-then(-else) changes a variable, while others don't, then the Prolog code of latter branches has to state that the new Prolog variable is equal to the old one, (2)
 - At the end of the loop the Prolog predicate is called with arguments corresponding to the current values of the C variables, (3)

²References of the form (n) point to the previous slide.

Contents

2 Declarative Programming with Prolog

- Prolog – first steps
- Prolog execution models
- The syntax of the (unsweetened) Prolog language
- Further control constructs
- Operators and special terms
- Working with lists
- Term ordering
- Higher order predicates
- All solutions predicates
- Efficient programming in Prolog
- **Building and decomposing terms**
- Executable specifications
- Block declarations
- Further reading

Building and decomposing compounds: the *univ* predicate

- BIP =.. /2 (pronounce *univ*) is a standard op. (`xfx`, 700; just as =, ...)
- Term =.. List holds if
 - Term = $Fun(A_1, \dots, A_n)$ and List = $[Fun, A_1, \dots, A_n]$, where Fun is an atom and A_1, \dots, A_n are arbitrary terms; or
 - Term = C and List = $[C]$, where C is a constant.
(Constants are viewed as compounds with 0 arguments.)
- Whenever you would like to use a var. as a compound name, use *univ*:
 $X = F(A_1, \dots, A_n)$ causes **syntax error**, use $X =.. [F, A_1, \dots, A_n]$ instead
- Call patterns for *univ*:
 - `+Term =.. ?List` decomposes Term
 - `-Term =.. +List` constructs Term
- Examples

<code>?- edge(a,b,10) =.. L.</code>	\implies	<code>L = [edge,a,b,10]</code>
<code>?- Term =.. [edge,a,b,10].</code>	\implies	<code>Term = edge(a,b,10)</code>
<code>?- apple =.. L.</code>	\implies	<code>L = [apple]</code>
<code>?- Term =.. [1234].</code>	\implies	<code>Term = 1234</code>
<code>?- Term =.. L.</code>	\implies	error
<code>?- f(a,g(10,20)) =.. L.</code>	\implies	<code>L = [f,a,g(10,20)]</code>
<code>?- Term =.. [/ ,X,2+X].</code>	\implies	<code>Term = X/(2+X)</code>

An interesting Prolog task

- A job interview question: construct an arithmetic expression containing integers 1, 3, 4, 6 each exactly once, using the four basic arithmetic operators +, -, *, /, 0 or more times, so that the expression evaluates to 24
- Let's write a Prolog program for solving this task:

```
:- use_module(library(lists), [permutation/2]).
```

```
% arith_expr(+L, +OpL, +Val, -Expr) :
```

```
% Expr is an arithmetic expression containing only operators present
```

```
% in the list OpL (operators may be used 0 or more times) and
```

```
% integers given in list L (each integer has to appear exactly once),
```

```
% so that the value of the expression is Val.
```

```
arith_expr(L, OpL, Val, Expr) :-
```

```
    permutation(L, PL), % permute the list of integers into PL
```

```
    leaves_ops_expr(PL, OpL, Expr), % build Expr with PL as the leaves-list
```

```
    catch(Expr == Val, _, fail). % check if Expr evaluates to Val, fail
```

```
% if there is a division-by-0 error.
```

An interesting Prolog task, cont'd

```

% leaves_ops_expr(+L, +OpL, ?Expr): Expr is an arithmetic expression
% which uses operators from OpL (0 or more times each) whose leaves,
% read left-to-right, form the list L.
leaves_ops_expr(L, _OpL, Expr) :-
    L = [Expr].          % If L is a singleton, Expr is the only element
leaves_ops_expr(L, OpL, Expr) :-
    append(L1, L2, L),          % Split L to nonempty L1 and L2,
    L1 \= [], L2 \= [],
    leaves_ops_expr(L1, OpL, E1), % generate E1 from L1 (using OpL),
    leaves_ops_expr(L2, OpL, E2), % generate E2 from L2 (using OpL),
    member(Op, OpL),           % choose an operator Op from OpL,
    Expr =.. [Op,E1,E2].       % build the expression 'E1 Op E2'

| ?- solve(66).
(3*4-1)*6
(4*3-1)*6
6*(3*4-1)
6*(4*3-1)
yes

```

A motivating symbolic processing example

- Polynomial: built from the atom 'x' and numbers 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, for x=X
```

```
value_of0(x, X, V) :- V = X.
```

```
value_of0(N, _, V) :-
```

```
    number(N), V = N.
```

```
value_of0(P1+P2, X, V) :-
```

```
    value_of0(P1, X, V1),
```

```
    value_of0(P2, X, V2),
```

```
    V is V1+V2.
```

```
value_of0(Poly, X, V) :-
```

```
    Poly = *(P1,P2),
```

```
    value_of0(P1, X, V1),
```

```
    value_of0(P2, X, V2),
```

```
    PolyV = *(V1,V2),
```

```
    V is PolyV.
```

```
value_of(x, X, V) :- !, V = X.
```

```
value_of(N, _, V) :-
```

```
    number(N), !, V = N.
```

```
value_of(Poly, X, V) :-
```

```
    Poly =.. [Func,P1,P2],
```

```
    value_of(P1, X, V1),
```

```
    value_of(P2, X, V2),
```

```
    PolyV =.. [Func,V1,V2],
```

```
    V is PolyV.
```

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

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

Building and decomposing compounds: 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)`, where $N \geq 0$ holds
 - Execution: The N th argument of `Compound` is **unified** with `A`.
If `Compound` has less than N arguments, or $N = 0$, `arg/3` fails
 - Arguments are **unified** – `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)
```


Finding arbitrary subterms using `arg/3` and `functor/3`

- Given a term T_0 with a (not necessarily proper) subterm T_n at depth n , the position of T_n within T_0 is described by a *selector* $[I_1, \dots, I_n]$ ($n \geq 0$):
`select_subterm(T0, [I1, ..., In], Tn) :-`
`arg(I1, T0, T1), arg(I2, T1, T2), ..., arg(In, Tn-1, Tn).`
- E.g. within term `a*b+f(1,2,3)/c`, `[1]` selects `a*b`, `[1,2]` selects `b`, `[2,1,3]` selects `3`, `[]` selects the whole term
- Given a term, enumerate all subterms and their *selectors*.

```
% subterm(?T, ?Sub, ?Sel): Sub is subterm in T at position Sel.
subterm(X, X, []).
```

```
subterm(X, Sub, [I|Sel]) :-
    compound(X),                % it is important that X is not a var.
    functor(X, _, Arity),       % because functor would raise an error
    between(1, Arity, I),
    arg(I, X, Y), subterm(Y, Sub, Sel).
```

```
| ?- subterm(f(1,[b]), T, S). => T = f(1,[b]), S = [] ? ;
                              => T = 1,      S = [1] ? ;
                              => T = [b],    S = [2] ? ;
                              => T = b,      S = [2,1] ? ;
                              => T = [],     S = [2,2] ? ; 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`

- Examples:

```
| ?- atom_codes(ab, Cs).           ==> Cs = [97,98]
| ?- atom_codes(ab, [0'a|L]).     ==> L = [98]
| ?- Cs="bc", atom_codes(Atom, Cs). ==> Cs = [98,99], Atom = bc3
| ?- atom_codes(Atom, [0'a|L]).   ==> error
```

³A string "abc..." is treated as a list of character codes of a, b, ...

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`
- Examples:

```
| ?- number_codes(12, Cs).           ⇒ Cs = [49,50]
| ?- 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 :-)
```

Contents

2 Declarative Programming with Prolog

- Prolog – first steps
- Prolog execution models
- The syntax of the (unsweetened) Prolog language
- Further control constructs
- Operators and special terms
- Working with lists
- Term ordering
- Higher order predicates
- All solutions predicates
- Efficient programming in Prolog
- Building and decomposing terms
- **Executable specifications**
- Block declarations
- Further reading

Executable specifications – what are they?

- An executable specification is a piece of **non-recursive** Prolog code which is in a one-to-one correspondence with its **specification**
- Example 1: Finding a contiguous sublist with a given sum

```
% sublist_sum(+L, +Sum, ?SubL): SubL is a sublist of L summing to Sum.
| ?- sublist_sum([1,2,3], 3, SL).  $\implies$  SL = [1,2] ? ; SL = [3] ? ; no
:- use_module(library(lists)). % To import sublist/2, append/2
sublist_sum(L, Sum, SubL) :-
    append( _, SubL, _), % SubL is a sublist of L
    sublist(SubL, Sum). %  $\sum$  SubL = Sum
```

- Example 2: Finding elements occurring in pairs

```
% paired(+List, ?E, ?I): E is an element of List equal to its
% right neighbour, occurring at (zero-based) index I.
| ?- paired([a,b,b,c,d,d], E, I).  $\implies$  E = b, I = 1 ? ;
 $\implies$  E = d, I = 4 ? ; no

paired(L, E, I) :-
    append(Pref, [E,E|_], L), % L starts with a sublist Pref,
 $\implies$  % followed by two elements equal to E
    length(Pref, I). % The length of Pref is I
```

Contents

2 Declarative Programming with Prolog

- Prolog – first steps
- Prolog execution models
- The syntax of the (unsweetened) Prolog language
- Further control constructs
- Operators and special terms
- Working with lists
- Term ordering
- Higher order predicates
- All solutions predicates
- Efficient programming in Prolog
- Building and decomposing terms
- Executable specifications
- **Block declarations**
- Further reading

Prolog extensions: coroutines (Prolog II)

- Wikipedia: Coroutines are computer program components that allow execution to be suspended and resumed, generalizing subroutines for cooperative multitasking. Coroutines are well-suited for implementing familiar program components such as cooperative tasks, exceptions, event loops, iterators, infinite lists and pipes.
- A typical example of coroutines, the Hamming problem: Generate, in increasing order, the sequence of all positive integers divisible by no primes other than 2, 3, 5.
- We implement a simplified version: the only divisors allowed are 2 and 3, using predicates `times/3` and `merge/3` in dataflow programming style

- For this we add the block declaration

```
:- block times(-, ?, ?).
```

Meaning: suspend pred. `times` if the first arg. is an unbound variable

- Also, suspend pred. `merge` if the first **or** second arg is unbound

```
:- block merge(-, ?, ?), merge(?, -, ?).
```

Helper predicates for the Hamming problem

- Multiply each element of a list by a number:

```
% times(As, M, Bs): List Bs is obtained from number list As by
% multiplying each list element by M.
:- block times(-, ?, ?).      % blocks if the 1st arg is a variable.
times([A|X], M, Bs) :-
    B is M*A, Bs = [B|Cs], times(X, M, Cs).
times([], _, []).
```

- Merge two sorted lists into a single sorted list

```
% merge(As, Bs, Cs): Sorted list Cs is obtained by
% collating sorted lists As and Bs, removing duplicates
:- block merge(-, ?, ?), merge(?, -, ?).
merge([A|As], [B|Bs], Cs) :-
    ( A < B -> Cs = [A|Ds], merge(As, [B|Bs], Ds)
    ; A > B -> Cs = [B|Ds], merge([A|As], Bs, Ds)
    ; Cs = [A|Ds], merge(As, Bs, Ds)
    ).
merge([], Bs, Bs).
merge(As, [], As).
```


Solving the Hamming problem via coroutinging

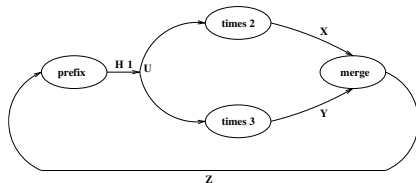
% U is the list of the first N (2,3)-Hamming numbers

```
hamming(N, U) :-
```

```
  U = [1|_], times(U, 2, X), times(U, 3, Y), merge(X, Y, Z),
```

```
  prefix_length([1|Z], U, N).           % A predicate from library(lists)
```

% prefix_length(L, P, N): L has a prefix P of length N



Contents

2 Declarative Programming with Prolog

- Prolog – first steps
- Prolog execution models
- The syntax of the (unsweetened) Prolog language
- Further control constructs
- Operators and special terms
- Working with lists
- Term ordering
- Higher order predicates
- All solutions predicates
- Efficient programming in Prolog
- Building and decomposing terms
- Executable specifications
- Block declarations
- Further reading

Additional slides

Subsequent slides were not presented in the class, these are included as further reading and for reference purposes

Error handling in Prolog

- A BIP for catching exceptions (errors): `catch(:Goal, ?ETerm, :EGoal):`
- Recall: “:” marks a **meta** argument, i.e. a term which is a goal
- BIP `catch/3` runs `Goal`
 - If no exception is raised (no error occurs) during the execution of `Goal`, `catch` ignores the remaining arguments
 - When an exception occurs, an exception term `E` is produced, which contains the details of the exception
 - If `E` unifies with the 2nd argument of `catch`, `ETerm`, it runs `EGoal`
 - Otherwise `catch` propagates the exception further outwards, giving a chance to surrounding `catch` goals
 - If the user code does not “catch” the exception, it is caught by the top level, displaying the error term in a readable form.

```
| ?- X is Y+1.
! Instantiation error in argument 2 of (is)/2
! goal:  _177 is _183+1
| ?- catch(X is Y+1, E, true).
E = error(instantiation_error,instantiation_error(_A is _B+1,2)) ? ; no
| ?- catch(X is Y+1, _, fail).
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).
p :- write(ga). (2)
```

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

```
| ?- q1.  =>  gogo
| ?- q2.  =>  gaga
| ?- r.   =>  gogo
```

`:-` (counter-intuitive)

- 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>), ... .
```

- $\langle \text{modespec}_i \rangle$ 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` is declared a meta predicate in `module1_m`)

```
:- module(module3, [quadruple/1,r/0]).
```

```
:- use_module(module1_m). % the loaded form:
```

```
r :- double(p).  $\implies$  r :- double(module3:p).4
```

```
:- meta_predicate quadruple(:).
```

```
quadruple(X) :- double(X), double(X).  $\implies$  unchanged4
```

⁴The imported goal `double` gets a prefix `module1:`, not shown here, to save space.