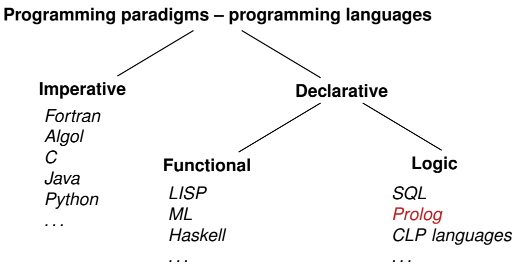
Part II

Declarative Programming with Prolog

Contents

Prolog in the family of programming languages



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
`has_p(b, c).`	`% b has a parent c.`	`+has_p(b, c).`
`has_p(b, d).`	`% b has a parent d.`	`+has_p(b, d).`
`has_p(d, e).`	`% d has a parent e.`	`+has_p(d, e).`
	`% for all GC, GP, P holds`	
`has_gp(GC, GP) :-`	`% GC has grandparent GP if`	`+has_gp(*GC, *GP)`
`has_p(GC, P),`	`% GC has parent P and`	`-has_p(*GC,*P)`
`has_p(P, GP).`	`% P has parent GP.`	`-has_p(*P,*GP).`

FOL: $\forall GC, GP.\ (\texttt{has\_gp}(GC, GP) \leftarrow \exists P.(\texttt{has\_p}(GC, P) \land \texttt{has\_p}(P, GP)))$

- 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, *cannonical* 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.          ⟹   X = 1-2, Z = (1-2)*(1-2)
| ?- U = X/Y, c(X,b)=c(a,Y).    ⟹   U = a/b, X = a, Y = b
| ?- 1-2*3 = X*Y.               ⟹   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.         ⟹   X = 2, Y = 6 ?
| ?- X = 2, 7 is X*X+2.         ⟹   no
| ?- X = 6, 7-1 is X.           ⟹   no
| ?- X is f(1,2).               ⟹   '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.          ⟹   X = 6
| ?- X = 6, X*X =:= (X+3)*(X-2). ⟹  X = 6
| ?- X = 6, X+3 =:= 2*(X-2).    ⟹   no
| ?- X = 6, X+3 =:= 2*(Y-2).    ⟹   'Instantiation Error'
```

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

An example: cryptarithmetic puzzle

- Consider this cryptarithmetic 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:

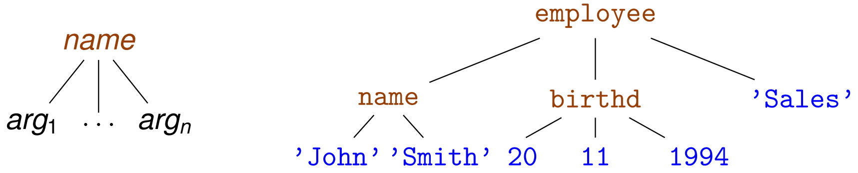
```
ad_day(AD, DAY) :-
    between(1, 9, A), between(1, 9, D), between(0, 9, Y),
    A =\= D, A =\= Y, D =\= Y,
    DAY is D*100+A*10+Y, AD is A*10+D,
    AD * AD =:= DAY.
```

- 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 `_` as a var 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*$(arg_1, \ldots, arg_n)$, where
  - *name* is an atom, arg_i 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 can be assigned (unified with) a non-variable value only once:
  `| ?- X = 2.` $\Longrightarrow$ `no`
- However, two variables may be unified and then assigned a (common) value:
  `| ?- X = Y, X = 2.` $\Longrightarrow$ `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).          ==>     Y = c ? ; Y = d ? ; no
```

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

```
| ?- Emp = employee(Name,Birth,Dept), Dept='Sales',
        Name=name(First,Last), First = 'John'.
   ==>  Emp = employee(name('John',Last),Birth,'Sales') ?
```


The logic variable (cont'd)

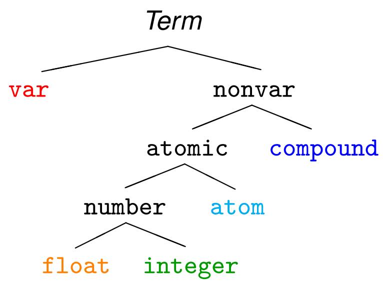
- A variable may also appear several times in a compound, e.g. `name(X,X)` is a Prolog term, which will match the first argument of the employee/3 record, iff the person's first and last names are the same:

```
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)



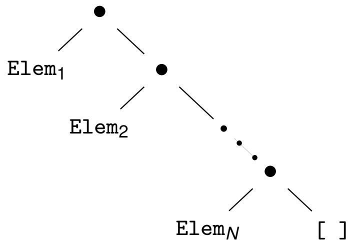
`var(X)`	X is a variable
`nonvar(X)`	X is not a variable
`atomic(X)`	X is a constant (atom or number)
`compound(X)`	X is a compound
`number(X)`	X is a number
`atom(X)`	X is an atom
`float(X)`	X is a floating point number
`integer(X)`	X is an integer

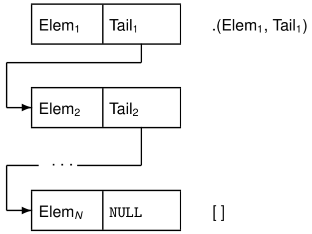
- The five coloured BIPs correspond to the five basic term types.
- Two further type-checking BIPs:
  - `simple(X)`: X is not compound, i.e. it is a variable or a constant.
  - `ground(X)`: X is a constant or a compound with no (uninstantiated) variables in it.

Another syntactic “sweetener” – list notation

- A Prolog list `[a,b,...]` 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: $[E_1,E_2,\ldots,E_n|Tail] \equiv [E_1|[E_2|\ldots,[E_n|Tail]\ldots]]$
  $[E_1,E_2,\ldots,E_n] \equiv [E_1,E_2,\ldots,E_n|[]]$

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([_]).
| ?- head0(L), singleton(L). ⇒ L = [0]      % L is a proper list
| ?- head0(L1). ⇒ L1 = [0|_A]              % L1 is an open ended list
```

- A Prolog term is called an *open ended* 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|Y]` 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 examples

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

  `% boring(+L, ?A): List L is A-boring.`
- Transform the following statements in English to Prolog clauses
  - List `L` is `A`-boring, if `L` has a single element `A`.
  - List `L` is `A`-boring, if `L`'s head equals `A` and `L`'s tail is `A`-boring.
- Given a list of numbers, calculate the sum of the list elements.

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

Another recursive data structure – binary tree

- A binary tree data structure can be defined as being
  - either a leaf (`leaf`) which contains an integer
  - 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 Node, Leaf;
struct tree {
  enum treetype type;
  union {
    struct { struct tree *left;
             struct tree *right;
           } node;
    struct { int value;
           } leaf;
  } 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(V)) :- integer(V).
is_tree(node(Left,Right)) :-
    is_tree(Left),
    is_tree(Right).
```

Recall: `integer(V)` 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), 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

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 implied by (1) and   (g2)
(g2)  driving_fast, raining     % (g2) is implied by (2) and   (g3)
(g3)  in_a_hurry,   raining     % (g3) is implied by (3) and   (g4)
(g4)  raining                   % (g4) is implied by (4) and   (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)
```

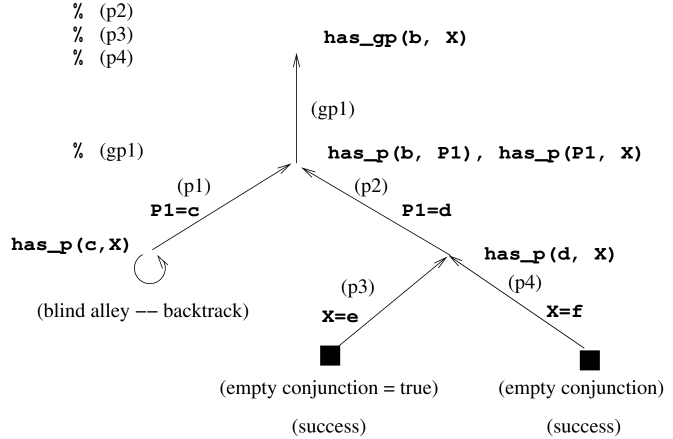
- To show that `get_fined` holds, resolution does an indirect proof
- Assume `get_fined` does not hold, deduce false (contradiction) using clauses (1)–(4)

```
(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 $G' \rightarrow G$
- E.g. reducing G = `has_gp(b, X)` using (gp1) gives
  G' = `has_p(b, P1), has_p(P1, X)`

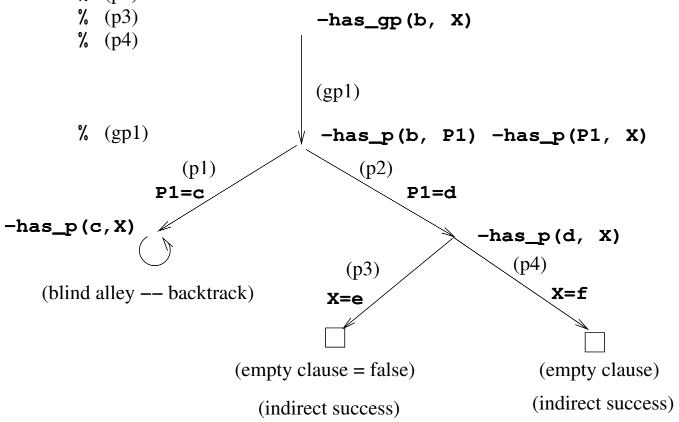
```
has_p(b, c).            % (p1)
has_p(b, d).            % (p2)
has_p(d, e).            % (p3)
has_p(d, f).            % (p4)

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, i.e. a *disjunction* of neg. literals *NG* and using a clause *C* deduces new neg. goal *NG′*, so that $NG \to NG'$
- E.g. resolving NG = `-has_gp(b, X)` using (gp1) gives
  NG' = `-has_p(b, P1) -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 $G = A, B, \ldots$ and a clause (A `:-` $D, \ldots$)
  $G' = B, \ldots, D, \ldots$ is obtained as the new goal

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

- Resolution:
  - to prove `A` $\wedge$ `B` $\wedge \ldots$, we negate it obtaining $\neg G_0$ = `-A -B ...`
  - resolution step : clause Cl = (`+A -D ...`) resolved with $\neg G_0$ produces $\neg G_1$ = `-D ...-B ...`
    $\neg G_n \wedge Cl \to \neg G_{n+1}$ (resolution)
  - success of indirect proof: reaching an empty clause $\square \equiv$ false
- Goal reduction:
  - to prove `A` $\wedge$ `B` $\wedge \ldots$, we start with G_0 = `A, B, ...`
  - reduction step : using Cl = (`A :- D, ...`) one can reduce G_0 to G_1 = `D, ..., B, ...`
    $G_{n+1} \wedge Cl \to G_n$ (reduction)
  - success of the reduction proof: reaching an empty goal $\blacksquare \equiv$ 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 97: $G =$ `has_gp(b, X)` using (gp1) $\Rightarrow G' =$ `has_p(b, P1),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 $G' = G_R\sigma$

E.g., homework P1: $G =$ `R1 is 2-1, list_length([a], R1)` $\Rightarrow G' =$ `list_length([a], 1)`

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).`

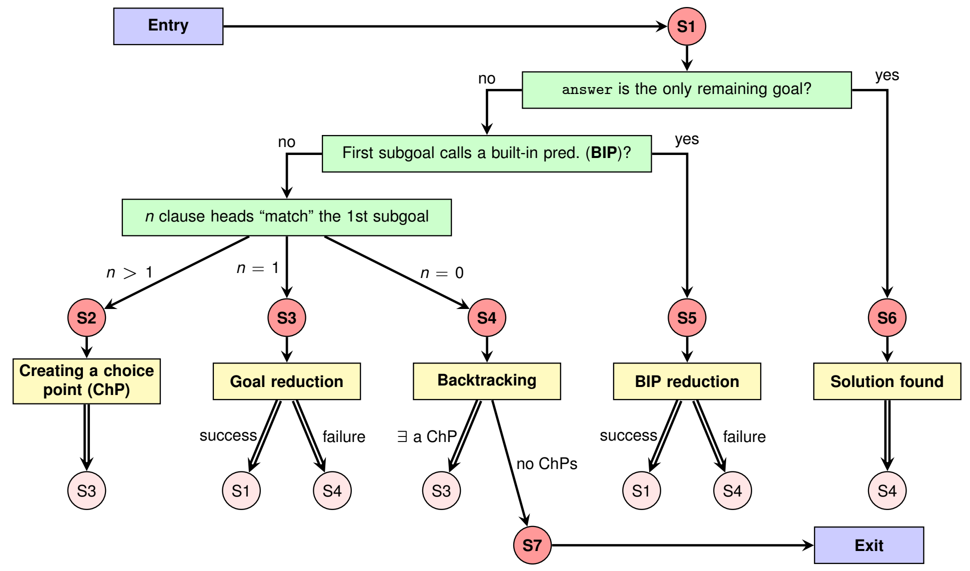
The main data structures used in the model

- 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	
CHP2	[p3,p4]	(4)	`hasP(d,Y),answer(b-Y).`
CHP1	[p2,p3,p4]	(2)	`hasP(X,P),hasP(P,Y),answer(X-Y).`

- 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.)

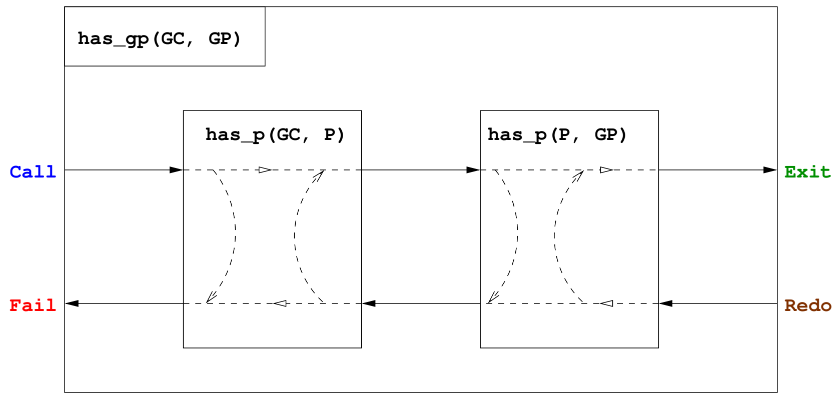
Remarks on the flowchart

- 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 ⇒ **S6**;
  - when the first subgoal of `Goal` calls a BIP ⇒ **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 ⇒ **S2**, **S3** or **S4**.
- **S2** creates a new `ChPSt` entry, and ⇒ **S3** (to reduce with the first clause).
- **S3** performs the reduction. If that fails ⇒ **S4**, otherwise ⇒ **S1**.
- **S4** retrieves the next clause from the top `ChPSt` entry, if any (⇒ **S3**), otherwise execution ends (⇒ **S7**).
- In **S5**, similarly to **S3**, if the BIP succeeds ⇒ **S1**, otherwise ⇒ **S4**.
- In **S6**, the solution is displayed and further solutions are sought (⇒ **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

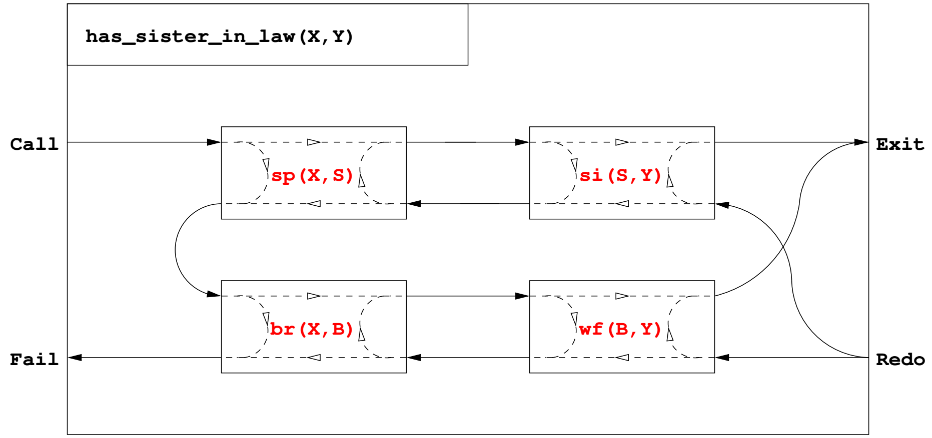
```
| ?- consult(gp3).
% consulting gp3.pl...
% consulted gp3.pl ...
yes
| ?- listing.
has_gp(Ch, G) :-
        has_p(Ch, P),
        has_p(P, G).

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

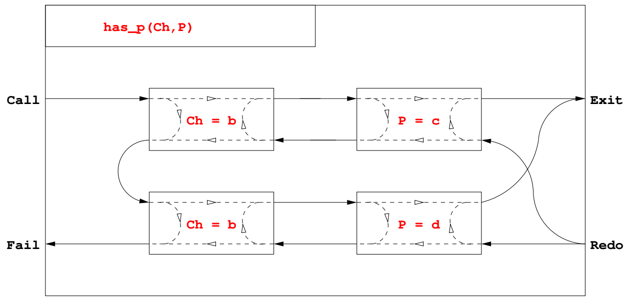
- In general in a multi-clause predicate the clauses have different heads
- A database of facts is a typical example:

```
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

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

⟨program⟩	::=	⟨predicate⟩ . . .	{i.e. a sequence of predicates}
⟨predicate⟩	::=	⟨clause⟩ . . .	{with the same functor}
⟨clause⟩	::=	⟨fact⟩.␣ |	
		⟨rule⟩.␣	
⟨fact⟩	::=	⟨head⟩	
⟨rule⟩	::=	⟨head⟩:-⟨body⟩	{clause functor = head functor}
⟨body⟩	::=	⟨goal⟩, . . .	{i.e. a seq. of goals sep. by commas}
⟨head⟩	::=	⟨callable term⟩	{atom or compound}
⟨goal⟩	::=	⟨callable term⟩	{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

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 :- q, (r ; s).`
  Distributive expansion inefficient, as it calls `q` twice:

```
p :- q, r.
p :- 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 – `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) :-                    pow1(A, E, P) :-
    (   E > 0, E1 is E-1,   ⟹          (   E > 0 -> E1 is E-1,
        pow(A, E1, P1),                     pow(A, E1, P1),
        P is A*P1                           P is A*P1
    ;   E = 0, P = 1                    ;   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`.

Disjunction – defining “childless”

- 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 fail (unless no parents).
- 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 (it does not look for further solutions of `G`)
- Since a failed goal produces no bindings, “`\+ G`” will never bind a variable.
- Read “`\+`” as “not provable”, cf. $\nvdash$ tilted slightly to the left.

Negation by failure – siblings and cousins

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

- Define predicates `has_sibling/2` and `has_cousin/2`:

```
has_sibling(A, B) :-
        has_parent(A, P), has_parent(B, P), \+ A = B.    % ≡ A \= B
has_cousin(A, B) :-
        has_grandparent(A, GP), has_grandparent(B, GP),
        \+ has_sibling(A, B), A \= B.
```

- There are some pitfalls in negation-by-failure, to be discussed later
- Most pitfalls can be avoided by using, in negation, either
  - ground goals, i.e. goals containing no unbound variables at the time of invocation, as e.g. in `has_sibling` and `has_cousin`; or
  - goals containg void (i.e. single occurrence) variables only, as in `childless`

The relationship of if-then-else and negation

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

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

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

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

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

- As semicolon binds to the right, there is no need to (and please don't) use nested parentheses for multiple if-then-else branches:

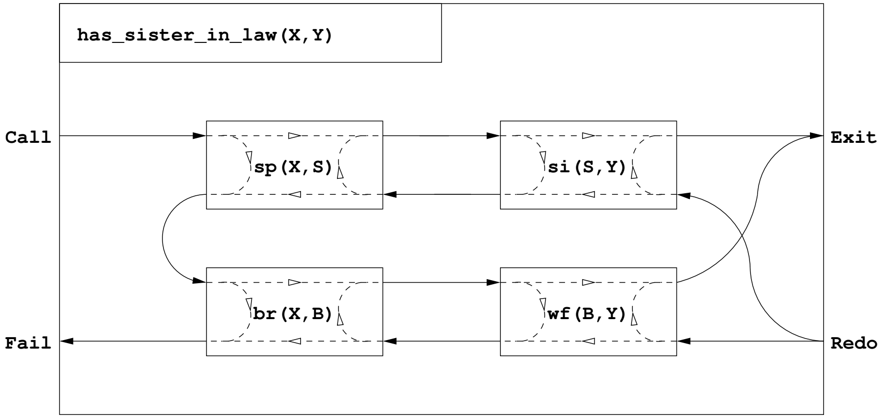
```
(   cond1 -> then1                 (   cond1 -> then1
;   (   cond2 -> then2             ;   cond2 -> then2
    ;   (   (...)   )      ⟹       ;   (...)
    )
;   else                           ;   else
)                                  )
```

The procedure-box of disjunctions

A disjunction can be transformed into a multi-clause predicate

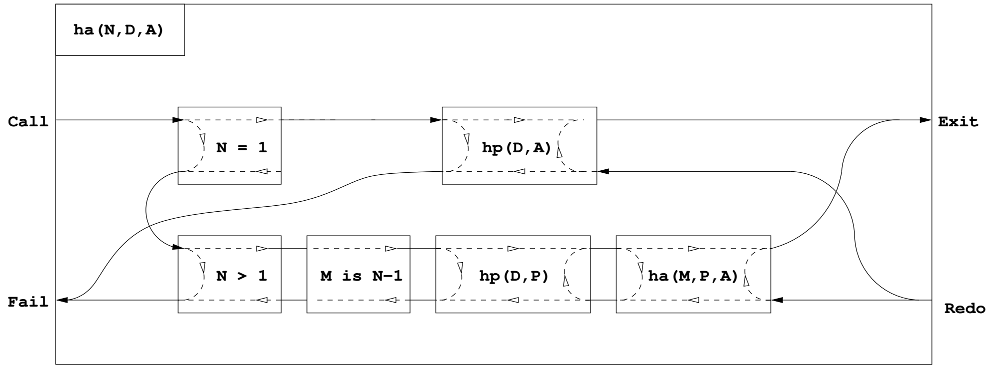
```
has_sister_in_law(X, Y) :-
    (   has_spouse(X, S), has_sister(S, Y)
    ;
        has_brother(X, B), has_wife(B, Y)
    ).
```

```
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)
        ).
```

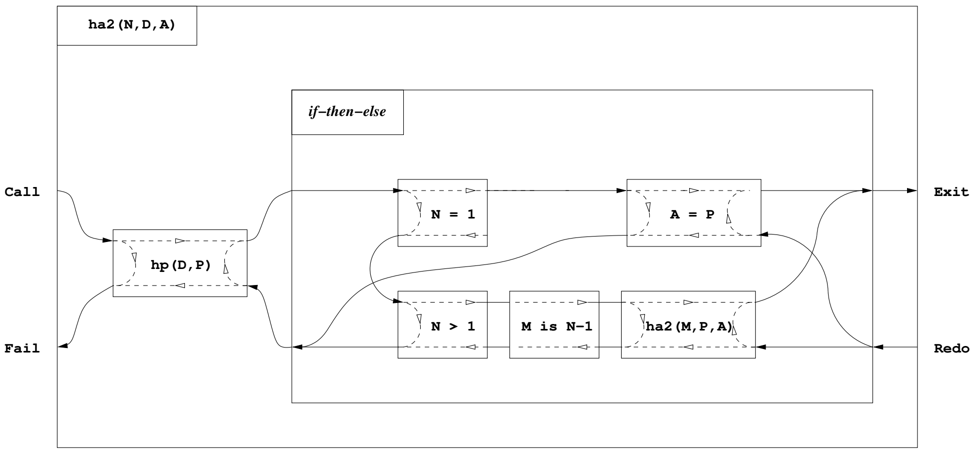


- Failure of the "then" part leads to failure of the whole if-then-else construct

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

Introducing operators

- Example: `S is -S1+S2` is equivalent to: `is(S, +(-(S1),S2))`
- Syntax of terms using operators

  ⟨comp. term⟩ ::=
  ⟨comp. name⟩ (⟨argument⟩, …) {so far we had this}
  | ⟨argument⟩ ⟨operator name⟩ ⟨argument⟩ {infix term}
  | ⟨operator name⟩ ⟨argument⟩ {prefix term}
  | ⟨argument⟩ ⟨operator name⟩ {postfix term}
  | (⟨term⟩) {parenthesized term}

  ⟨operator name⟩ ::= ⟨comp. name⟩ {if declared as an operator}
- The built-in predicate for defining operators:
  `op(Priority, Type, Op)` or `op(Priority, Type, [Op1,Op2,...])`:
  - `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` ≡ `f(X, Y)`
	`fy`	`fx`	prefix	`f X` ≡ `f(X)`
`yf`		`xf`	postfix	`X f` ≡ `f(X)`

Parentheses implied by operator priorities and associativities

- `a/b+c*d` ≡ `(a/b)+(c*d)` as the priority of `/` and `*` (400) is less than the priority of `+` (500) smaller priority = **stronger** binding
- `a-b-c` ≡ `(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` ≡ `a^(b^c)` as `^` has type `xfy`, therefore it is right-associative
- `a=b=c` ⟹ 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` ≡ `((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 discontiguous 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)≡:-(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).    ⇒   ! write_canonical/3 does not exist
```

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

Functions and operators allowed in arithmetic expression

- Standard Prolog functions allowed in arithmetic expressions (represented by compounds and the atom `pi`, as listed below):

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)

    `| ?- 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, 0).          ==>   no
```


Contents

Concatenating lists

- Let `L1` ⊕ `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` ⊕ `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 ⊕ (concatenation) function:

```
L1 ⊕ L2 =  if L1 == NULL return L2
           else L3 = tail(L1) ⊕ 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])` ⇝ 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

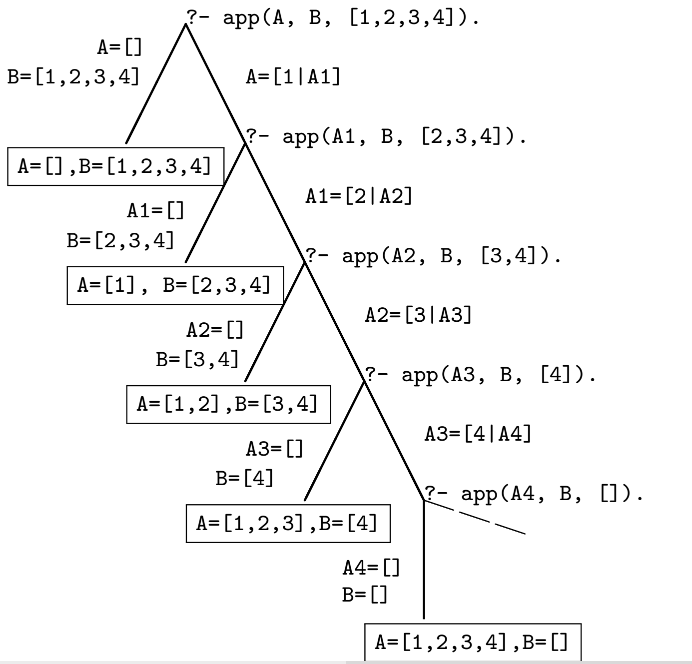
- Tail recursion optimization (TRO), or more generally last call optimization (LCO) is applicable if
  - the goal in question is the last to be executed in a clause body, and
  - there are no choice points in the given clause body.
- LCO is applicable to the recursive call of `app/3`:

```
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).              ⟹  L = [a,b|Tail] ? ; no
| ?- append([a,b], [c|T], L).             ⟹  L = [a,b,c|T] ?  ; no
| ?- append([a,b], [c|T], [_,_,d,_]).     ⟹  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]). ⟹  L1=[], L2=[1,2] ? ; L1=[1], L2=[2] ? ;
                              L1=[1,2], L2=[] ? ; no
| ?- append([1,2], L, [1,2,3,4,5]). ⟹  L = [3,4,5] ? ; no
| ?- append(L1,[4|L2],[1,2,3,4,5]). ⟹  L1 = [1,2,3],L2 = [5] ? ; no
| ?- append(L1,[4,2],[1,2,3,4,5]). ⟹  no
```

- The search may be infinite, if **both** the 1st **and** the 3rd arg. is open ended

```
| ?- append([1|L1], [a,b], L3).       ⟹
              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]).            ⟹  no
```


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

```
  :- mode append(+, +, +).   % checking if L1 ⊕ L2 = L3 holds
| ?- append([1,2], [3,4], [1,2,3,4]).     ⟹  yes

  :- mode append(+. +, -).   % appending L1 and L2 to obtain L3
| ?- append([1,2], [3,4], L3).            ⟹  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).         ⟹  L2 = [3,4] ? ; no

  :- mode append(+. -, -).   % prepending L1 to an open ended L2 to obtain L3
| ?- append([1,2], [3|L2], L3).           ⟹  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).         ⟹  L1 = [1,2] ? ; no

  :- mode append(-. -, +).   % splitting L3 to L1 and L2 in all possible ways
| ?- append(L1, L2, [1]).  ⟹  L1=[],L2=[1] ? ; L1=[1],L2=[] ? ; no

  :- mode append(-. +, -). (see prev. slide) and :- mode append(-. -, -).
| ?- append(L1, L2, L3). ⟹  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]
```


Searching for patterns in lists using `append/3` (ADVANCED)

- Elements occurring in pairs

```
% in_pair(+List, ?E, ?I):  E is an element of List equal to its
% right neighbour, occurring at position I (indexed from 0).
in_pair(L, E, I) :-
       append(Before, [E,E|_], L),
       length(Before, I).
```

BIP `length(?List, ?Len)`: `List` is a list of length `Len`

```
| ?- in_pair([1,8,8,3,4,4], E, I). ⟹     E = 8, I = 1 ? ;
                                   ⟹     E = 4, I = 4 ? ; no
```

- Stuttering sublists

```
% stutter(L, D): D \= [] concatenated with itself is a sublist of L.
stutter(L, D) :-
        append(_Before, Tail, L),   % same as: suffix(L, Tail),
        D = [_|_],                  % D is nonempty
        append(D, D, _, Tail).  % Using append/4 from prev. slide
%/*OR*/ append(D, End, Tail), append(D, _, End).
| ?- stutter([2,2,1,2,2,1], D).
        ⟹         D = [2] ? ; D = [2,2,1] ? ; D = [2] ? ; no
```

Appending a list of lists

- Library `lists` contains a predicate `append/2`
  see e.g. `https://www.swi-prolog.org/pldoc/doc/_SWI_/library/lists.pl`
  `% 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):
  1. either each element of `LL` is a closed list
  2. or `L` is a closed list
- Examples:

```
| ?- append([[1,2],[3,4,5],[6]], L).  ⟹  L = [1,2,3,4,5,6] ? ; no
| ?- append([L1,L2,L3], [1,2]).
      ⟹                 L1 = [],    L2 = [],    L3 = [1,2] ? ;
                         L1 = [],    L2 = [1],   L3 = [2] ? ;
                         L1 = [],    L2 = [1,2], L3 = [] ? ;
                         L1 = [1],   L2 = [],    L3 = [2] ? ;
                         L1 = [1],   L2 = [2],   L3 = [] ? ;
                         L1 = [1,2], L2 = [],    L3 = [] ? ; no
```

- Implementation of `stutter` from prev. slide, using `append/2`:

```
stutter(L, D) :- append([_,D,D,_], L).
```


The BIP `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(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.

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).                          revapp([], L, L).
app([X|L1], L2, [X|L3]) :-              revapp([X|L1], L2, L3) :-
    app(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 revapp(list L1, list L2)
{ list L3, *lp = &L3;                       { list l = L2;
  for (list p=L1; p; p=p->next)               for (list p=L1; p; p=p->next)
  { list newl = new link(p->elem);            { list newl = new link(p->elem);
    *lp = newl; lp = &newl->next;               newl->next = l; l = newl;
  }                                           }
  *lp = L2;  return L3;                       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

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

```
permutation([], []).
permutation(List, [First|Perm]) :-
        select(First, List, Rest),
        permutation(Rest, Perm).
```

- Possible uses:

```
| ?- permutation([1,2], L).                   mode (+,-)
        L = [1,2] ? ; L = [2,1] ? ; no
| ?- permutation([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
| ?- permutation(L, [1,2]).      Can it be used in mode (-,+)?
        L = [1,2] ? ; infinite loop
```

- If the first argument in permutation/2 is unbound, then the search space of the select call is infinite!
- The variant of permutation/2 in library lists works for both modes.

Contents

Higher order programming: using predicates as arguments

- Higher order predicates take predicates/goals as arguments
- Example: extracting all nonzero elements of a number list

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

- Generalize to a pred. 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:

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


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
- The workings of the BIP `call(PG, X)` where `PG` is a partial goal:
  - if `PG` is an atom ⇒ it calls `PG(X)`
  - if `PG` is a compound `Pred(`A_1`,...,`A_n`)` ⇒ it calls `Pred(`A_1`,...,`A_n`,X)`
- Predicate `include(Pred, L, FL)` is in `library(lists)`

```
| ?- use_module(library(lists)).
| ?- L=[1,2,a,X,b,0,3+4],
     include(number, L, Nums).  % Nums = { x ∈ L | number(x) }
Nums = [1,2,0] ? ; no

| ?- L=[0,2,0,3,-1,0],
     include(\=(0), L, NZs).    % NZs =  { x ∈ L | \=(0,x) }
NZs = [2,3,-1] ?
```


Calling predicates with additional arguments

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

```
even(X) :- X mod 2 =:= 0.
| ?- include0([1,3,2,5,4,0], even, FL).   ⟹     FL = [2,4,0] ; no.
```


An important higher order predicate: `maplist/3`

- `maplist(:PG, ?L, ?ML)`[2]: List `ML` contains elements `Y` obtained by calling `PG(X,Y)` for each `X` element of list `L`, where `PG` is a partial goal to be expanded with two arguments

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

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

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

[2] annotation "`:`" marks a meta argument, i.e. a term to be interpreted as a (partial) goal

Another important higher order predicate: `scanlist/4` or `foldl/4`

- These are the same predicates, SICStus: `scanlist/4`, SWI: `foldl/4`.
- Example:

```
plus(A, Sum0, Sum) :-      Sum is Sum0+A.
sumlst(L, Sum) :-         scanlist(plus, L, 0, Sum).
| ?- sumlst([1,3,5], Sum).    ⟹   Sum = 9 ? ; no
```

- `scanlist(:PG, ?L, ?Init, ?Final)`:
  - `PG` represents a two-argument function: `call(PG, Elem, Acc0, Acc)` calculates the function on `Elem` and `Acc0` arguments and returns the function value in `Acc`.
  - `scanlist` applies this function repeatedly, on all elements of list `L`, left-to-right, where `Init` is the initial and `Final` is the final value of the accumulator.
- For example: `scanlist(plus, [X,Y,Z], 0, Sum)` is converted to:

```
plus(0, X, S1), plus(S1, Y, S2), plus(S2, Z, Sum)
```

- `scanlist` is also available in 5, 6 and 7 argument variants.
- `maplist` is also available in 2, 4 and 5 argument variants.

All solution built-in predicates

- All solution BIPs are higher order predicates analogous to list comprehensions in Haskell, Python, etc.
- Examples:

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

Predicate `between(+N, +M, ?X)` enumerates the integers `N`, `N+1`, ..., `M` in `X`. In SICStus, it is defined in `library(between)`

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

Approximate meaning: `L` is a list of `Templ` terms for all solutions 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 `Templ` (copy ⟹ replace vars in `Templ` by new ones) (note: copying requires time proportional to the size of `Templ`)
  - continue with failure (to enumerate further solutions)
- When there are no more solutions (`Goal` fails)
  - collect the stored `Templ` values into a list, unify it with `L`.

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


All solutions: the BIP `bagof(?Templ, :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 `Templ` (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 = $\{E \mid \exists R. (R \text{ employs } E)\}$
- `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, L = [b,c] ? ;
        ⟹   R = b, L = [c,d] ? ; no
```

- Use operator `^` to achieve existential quantification in `bagof`:

```
| ?- bagof(E, R^emp(R, E), Es). %  Collect E-s for which ∃ R ...
           ⟹   Es = [b,c,c,d] ? ; 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)`
- Example for using `setof/3`:

```
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).  ⟹  Vs = [a,b,c,d] ? ; no
```


Meta-predicates: the *univ* predicate

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

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


Building and decomposing compounds: functor/3 (ADVANCED)

- `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 (ADVANCED)

- `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)`

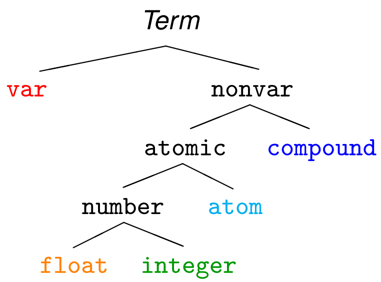
Error handling in Prolog (ADVANCED)

- A BIP for catching exceptions (errors): `catch(:Goal, ?ETerm, :EGoal)`:
- Annotation "`:`" 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
  - If an exception is raised then an exception term `E` is produced
    - 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.
- Examples

```
| ?- 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
```


Contents

Principles of Prolog term ordering $\prec$



Different kinds ordered left-to-right:

`var` $\prec$ `float` $\prec$ `integer` $\prec$
$\prec$ `atom` $\prec$ `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: $\texttt{name}_a(a_1, \ldots, a_n) \prec \texttt{name}_b(b_1, \ldots, b_m)$ iff
  1. $n < m$, e.g. `p(x,s(u,v,w))` $\prec$ `a(b,c,d)`, or
  2. $n = m$, and $\texttt{name}_a \prec \texttt{name}_b$ (lexicographically), e.g. `a(x,y)` $\prec$ `p(b,c)`, or
  3. $n = m$, $\texttt{name}_a = \texttt{name}_b$, 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
`Term1 == Term2`	`Term1` $\not\prec$ `Term2` $\wedge$ `Term2` $\not\prec$ `Term1`
`Term1 \== Term2`	`Term1` $\prec$ `Term2` $\vee$ `Term2` $\prec$ `Term1`
`Term1 @< Term2`	`Term1` $\prec$ `Term2`
`Term1 @=< Term2`	`Term2` $\not\prec$ `Term1`
`Term1 @> Term2`	`Term2` $\prec$ `Term1`
`Term1 @>= Term2`	`Term1` $\not\prec$ `Term2`

- The comparison predicates are not purely logical:
  `| ?- X @< 3, X = 4.` $\Longrightarrow$ `X = 4`
  `| ?- X = 4, X @< 3.` $\Longrightarrow$ `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).` $\Longrightarrow$ **yes**
- BIP `sort(L, S)` sorts (using `@<`) a list `L` of arbitrary Prolog terms, removing duplicates (w.r.t. `==`). The result is a strictly increasing list `S`.

```
| ?- 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

- U `=` V: U unifies with V
  No errors. May bind vars.

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

- U `==` V: U is identical to V, i.e. U=V succeeds with no bindings
  No errors, no bindings.

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

- 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.

```
| ?- X =:= 1+2.    ⟹  error
| ?- 1+2 =:= X.    ⟹  error
| ?- 2+1 =:= 1+2.  ⟹  yes
| ?- 3.0 =:= 1+2.  ⟹  yes
```

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

```
| ?- X is 1+2.     ⟹  X = 3
| ?- 3.0 is 1+2.   ⟹  no
| ?- 1+2 is X.     ⟹  error
| ?- 3 is 1+2.     ⟹  yes
| ?- 1+2 is 1+2.   ⟹  no
```


Nonequality-like Prolog predicates – a summary

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

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

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

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

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

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

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


(Non)equality-like Prolog predicates – examples

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

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

Contents

Causes of inefficiency – preview

- Unnecessary choice points (ChPs)
  Recursive definitions often leave choice points behind on exit, e.g.:
  - ```
    fact0(N, F) :-       % fact0(+N, ?F): F = N!.
        ( N = 0, F = 1   % Replace , by -> to avoid choicepoints
        ; N > 0, N1 is N-1, fact0(N1, F1), F is N*F1
        ).
    ```
    Remedy: use if-then-else (above) or cut (see later)
  - ```
    % 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 apply tail recursion, using accumulators

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

```
fact1(0, F) :- !, F = 1.                                        (1)
fact1(N, F) :- N > 0, N1 is N-1, fact1(N1, F1), F is N*F1.      (2)

% is_a_parent(+P): check if a given P is a parent.
is_a_parent(P) :- has_parent(_, P), !.
```

- 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 first solution:
    Remove all choice points created within the goals preceding the cut.
  - ***Commit to clause***:
    Remove the choice of any further clauses in the current predicate.
- Definition: if `q :- ..., p, ....` then
  the parent goal of `p` is the goal matching the clause head `q`
- In the box model: the parent goal is the goal invoking the surrounding box
- Effects of cut in the goal reduction model: 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 $\Longrightarrow$ Fail port of parent.
- The behavior of `(1)–(2)` is identical to the if-then-else on previous slide
- In fact, SICStus transforms this if-then-else to the pred. `(1)–(2)` above

Avoid leaving unnecessary choice points

- Add green cuts (those cutting off branches doomed to fail)
  ```
  % last1(L, E): The last element of L is E.
  last1([E], E)   :- !.
  last1([_|L], E) :- last1(L, E).
  ```
- Use if-then-else, rather than disjunction or multiple clauses
  ```
  fact1(N, F) :-     ( N = 0 -> F = 1
                     ; N > 0, N1 is N-1, fact1(N1, F1), F is N*F1
                     ).
  last2([E|L], X) :- ( L = [] -> X = E
                     ; last2(L, X)
                     ).
  ```
- Rely on indexing – applicable when the first arg. is input, and the outermost functor of the first head arg is different in each clause, e.g.
  ```
  tree_sum(leaf(Value), Value).          1st head arg functor: leaf/1
  tree_sum(node(Left, Right), S) :-      1st head arg functor: node/2
          tree_sum(Left, S1), tree_sum(Right, S2), S is S1+S2.
  ```


Avoiding the creation of choice points in if-then-else

- 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 `fact1` and `last2` with no ChPs created:

```
fact2(N, F) :-      ( N =:= 0 -> F = 1     % used to be N = 0
                    ; N > 0, N1 is N-1, fact2(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 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 for selecting the list of applicable clauses:

(VAR)	if `A` is a variable:	(1) (2) (3) (4) (5)
(0/0)	if `A = 0`:	(1) (2)
(s/1)	if the main functor of `A` 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), 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:
  - `C/0`, 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:
  - At compile-time: for each main functor which occurs in the first argument, the compiler collects the list of matching clauses.
  - At run-time: the Prolog engine selects the relevant clause list using the call argument, if instantiated. This is practically a constant time operation, as its implementation normally uses *hashing*.
  - **Important:** If a single clause is selected, *no choice point* is created. If a choice point *is* created, it is removed when the last branch is entered.

Indexing list handling predicates: examples

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

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

- The trivial implementation of `last/2` leaves a choice point behind.

```
% last0(L, E): The last element of L is E.
last0([E], E).
last0([_|L], E) :- last0(L, E).
```

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

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

% last(L, X, E): The last element of [X|L] is E.
last([], E, E).
last([X|L], _, E) :- last(L, X, 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 clause when the recursive call is reached
- *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.
```

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

```
% sum(+List, +Sum0, -Sum):
% (Σ List) + Sum0 = Sum, i.e.  Σ 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`:

```
% sum(+List, -Sum): the sum of the elements of List is Sum.
sum(List, Sum) :-  sum(List, 0, Sum).
```

- A higher order implementation using `scanlist`:

```
plus(X, Sum0, Sum1) :-     Sum1 is Sum0+X.
sum(L, Sum) :-       scanlist(plus, L, 0, Sum).
```


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 N*F1
                   ).
| ?- fact0(4, F). ⟹  F = 24 ∼  (4*(3*(2*(1*1))))
```

- 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 N*F0, 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 – `revapp/3`

- Recap predicate `revapp/3`:

```
% revapp(L, R0, R): The reverse of L prepended to  R0 gives R.
revapp([], R0, R) :-
    R = R0.
revapp([X|L], R0, R) :-
    R1 = [X|R0],
    revapp(L, R1, R).
```


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 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).
```

- Advantages:
  - One of the two recursive calls is tail-recursive.
  - There is no need to append the intermediate lists!

Accumulators for implementing imperative (mutable) variables

- Let $L = [x_1, \ldots,]$ 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, VC0, VC): L has VC-VC0
% left-visible elements which are > MV.
viscnt([], _, VC0, VC) :- VC = VC0.
viscnt(L0, MV0, VC0, VC) :-      % (1)
    L0 = [H|L1],
    (   H > MV0
    ->  VC1 is VC0+1, MV1 = H
    ;   VC1 = VC0, MV1 = MV0     % (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 head[3] *(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)*

[3] References of the form *(n)* point to the previous slide.

Do-loops for writing simple, tail recursive iterations (ADVANCED)

- Example: increment by 1 each element of list `L` to obtain list `IL`:

```
| ?- L = [1,2,3], ( foreach(X, L), foreach(Y, IL) do  Y is X+1 ).
IL = [2,3,4] ? ; no
```

The loop goal with two `foreach` iterators is replaced by `helper_1(L,IL)`:

```
                                      helper_1([], []) :- !.
   (  foreach(X, L),                  helper_1([X|L],
      foreach(Y, IL)           ⇒               [Y|IL]) :-
   do  Y is X+1                           Y is X+1,
   )                                      helper_1(L, IL).
```

- Vars `X` and `Y` are local; should not occur elsewhere in the query/body :-(
- To increment by an arbitrary number `N`, an iterator `param(...)` is needed:

```
                                      helper_2([], [], _) :- !.
   (  foreach(X, L),                  helper_2([X|L],
      foreach(Y, IL), param(N) ⇒               [Y|IL], N) :-
   do Y is X+N                             Y is X+N,
   )                                      helper_2(L, IL, N).
```


Do-loops, examples of further iterators (ADVANCED)

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

Translation:

```
                                    helper_3(I0, I0, []) :- !.
  (  for(I,N0,N), foreach(X,L)      helper_3(I, S, [X|T]) :-
  do X = I                     ⇒        I1 is I+1, X = I,
  ).                                    helper_3(I1, S, T).
```

and the do-loop is replaced by:

```
  Frst is N0, Stp is max(Frst,N+1), helper_3(Frst, Stp, L)
```

(This loop can be simplified to: `( for(I,1,N), foreach(I,List) do true )`)

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

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

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