Semantic and Declarative Technologies

Péter Szeredi

szeredi@cs.bme.hu
Aquincum Institute of Technology

Budapest University of Technology and Economics
Department of Computer Science and Information Theory

2012 Spring Semester

Part I: Declarative Programming – the Prolog language

- Prolog – PROgramming in LOGic (PROgrammation en LOGique)
  - The program = statements in (simplified) first-order pred. calculus
  - The execution of a program is a (very simple) reasoning process pattern-based procedure invocations and backtracking
- Dual semantics: declarative and procedural
  - WHAT rather than HOW
    (focus on the logic first, but then think over Prolog execution, too).
- A single predicate – multiple functions
  - An example: concatenating two lists – pay one, get many free!

```
    % Concatenate two lists
| ?- append([1,2], [3,4], L).          ⟹   L = [1,2,3,4] ? ; no
    % Check if one list is a prefix of another
| ?- append([1,2], _, [1,2,3,4,5]).    ⟹   yes
    % Split a list into two
| ?- append(L1, L2, [1,2]).           ⟹   L1 = [],    L2 = [1,2] ? ;
                                           L1 = [1],   L2 = [2] ? ;
                                           L1 = [1,2], L2 = [] ? ; no
```

- The CLP($\mathcal{X}$) schema

  Prolog or some other prog. language, e.g. C++ + "strong" reasoning capabilities on a restricted domain $\mathcal{X}$ involving specific constraint (relation) and function symbols.

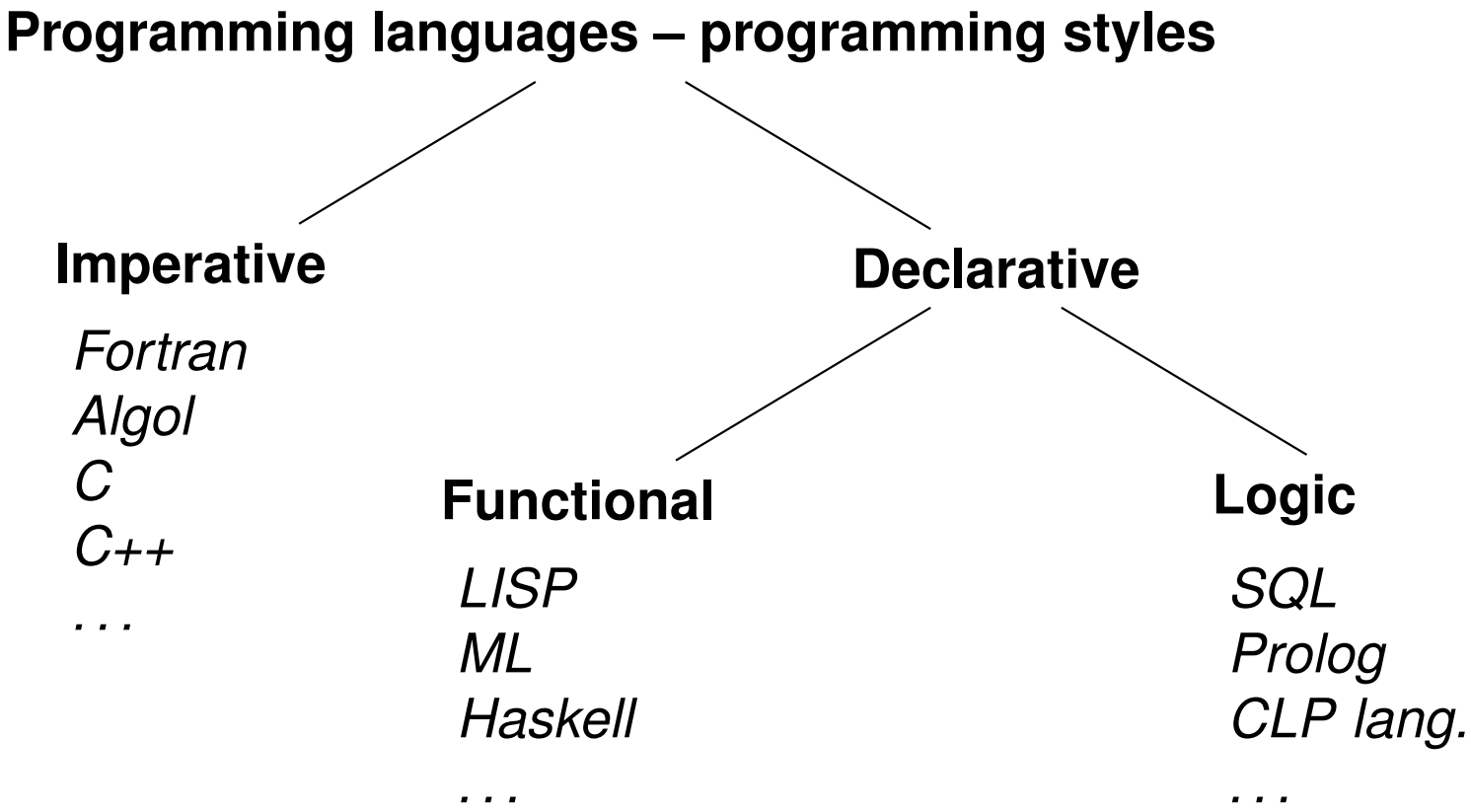
- Examples for the domain $\mathcal{X}$:
  - $\mathcal{X} =$ Q or R (rational or real numbers)
    constraints: linear equalities and inequalities
    reasoning techniques: Gauss elimination and the simplex method
  - $\mathcal{X} =$ FD (Finite Domains, e.g. of integers)
    constraints: various arithmetic, logic, and combinatorial relationships
    reasoning techniques: those developed for Constraint Satisfaction Problems (CSPs)

Part I

Declarative Programming with Prolog

Contents

A Classification of some programming languages



Imperative and declarative programming styles

- A sample imperative program
  - Imperative style: use commands
  - Variable: a mutable location
  - example in C:

```
int pow0(int a, int n) {    // pow0(a,n) = a^n, n is an integer, n≥0
  int p = 1;                // Set variable p to 1.
  while (n > 0) {           // Repeat while n>0 :
    n = n-1;                //    Decrease n by 1.
    p = p*a;    }           //    Multiply p by a.
  return p;           }     // Return the value of p.
```

- The same task solved declaratively
  - Declarative style: state truths
  - Variable: as in math, a single, yet unknown value

```
int powr(int a, int n) {    // powr(a,n) = a^n, n is an integer, n≥0
  if (n > 0)                // If n > 0
    return a*powr(a,n-1);   //   then a^n = a*a^(n-1)
  else return 1;       }    //   else a^n = 1
```

- This kind of recursion is expensive, requires non-constant memory

Declarative programming languages — motto

- WHAT rather than HOW: The program describes the *task to be solved* (WHAT to solve), rather than the *exact steps of the solution process* (HOW to solve).
- In practice, both aspects have to be taken care of – dual semantics:
  - Declarative semantics — What (kind of task) does the program solve;
  - Procedural semantics — How does the program solve it.

A Short and Slightly Biased Overview of early Prolog/LP history

1960s	Early theorem proving programs
1970-72	The theoretical basis of logic programming (R A Kowalski)
1972	The first Prolog interpreter (A Colmerauer)
1975	The second (?) Prolog interpreter (P Szeredi)[1]
1977	The first Prolog compiler (D H D Warren)
1977–85	Several experimental Prolog applications in Hungary
1981	The Japanese 5th Generation Project chooses Logic Programming as the basis for intelligent parallel computing
1982	The Hungarian MProlog is one of the first commercial Prolog implementations
1983	A new compiler model, the WAM abstract machine (D H D Warren)
1986	The beginning of the Prolog standardization
1987–	New logic programming languages (CLP, Mercury, Gödel etc.)
1987–. . .	Parallel Prolog implementations
	.....

[1] `http://dtai.cs.kuleuven.be/projects/ALP/newsletter/nov04/nav/articles/szeredi/szeredi.html`

Information about Logic Programming – LP systems

- Some free logic programming systems:
  - SWI Prolog – (semantic) Web interfaces (Univ. of Amsterdam) `http://www.swi-prolog.org/`
  - GNU Prolog – constraints (INRIA) `http://www.gprolog.org/`
  - XSB – a LP and Deductive Database system with tabling (Univ. Stony Brook) `http://xsb.sourceforge.net/`
  - The ECLiPSe Constraint Programming System (CISCO) `http://eclipseclp.org/`
  - YAP – Yet another Prolog (Univ. Porto) `http://yap.sourceforge.net/`
  - Mercury – types, modes, efficiency (Univ. Melbourne) `http://www.mercury.csse.unimelb.edu.au/index.html`
- Some commercial logic programming systems:
  - Visual Prolog (Windows integration) `http://www.visual-prolog.com/`
  - Quintus Prolog `http://www.sics.se/quintus`
  - SICStus Prolog, used in this course `http://www.sics.se/sicstus`

Information about Logic Programming – Resources

- The WWW Virtual Library, Logic Programming:
  `http://www.fmi.uni-sofia.bg/fmi/logic/skordev/ln/lp/logic-prog.html`
- CMU Prolog Repository:
  `http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/`
  - Main page: `...0.html`
  - Prolog Resource Guide: `...faq/prg_1.faq`, `...faq/prg_2.faq`
- The Association for Logic Programming `http://www.cs.nmsu.edu/ALP/`
  - Conference series: Int. Conference on Logic Programming (ICLP)
    - 28th ICLP will be held in Budapest, 4-8 September 2012
      `http://www.cs.bme.hu/iclp2012/`
  - Journal: Theory and Practice of Logic Programming (Cambridge Univ. Press) `http://journals.cambridge.org/tlp`

Information on Prolog

- In this course we use the SICStus Prolog system, version 4.2
  `http://www.sics.se/sicstus`
- SICStus documentation:
  `http://www.sics.se/sicstus/docs/latest4/html/sicstus.html`
- Some textbooks on Prolog
  - Programming in PROLOG: Using the ISO Standard, C.S. Mellish, W.F. Clocksin, Springer-Verlag Berlin, Paperback – July 2003
  - Logic, Programming and Prolog, 2nd Ed., by Ulf Nilsson and Jan Maluszynski, Previously published by John Wiley & Sons Ltd. (1995) Downloadable as a pdf file from
    `http://www.ida.liu.se/~ulfni/lpp`
  - Prolog Programming for Artificial Intelligence, 3rd Ed., Ivan Bratko, Longman, Paperback – March 2000
  - The Art of PROLOG: Advanced Programming Techniques, Leon Sterling, Ehud Shapiro, The MIT Press, Paperback – April 1994

Contents

The “happy” example

- We describe a world in which certain facts hold, and there are rules for being happy:

```
% happy: I'm happy.
happy :-                          % I'm happy if
    workday, good_lecture.        % it's a workday and I'm at a good lecture.
happy :-
    hot, swimming.                % I'm happy if it's hot and I'm swimming.
happy :-
    my_partner_is_happy.          % I'm happy if my partner is happy.
```

```
% workday: It's a workday.
workday.
```

```
% swimming: I'm swimming.
swimming.
```

```
% my_partner_is_happy: My partner is happy.
my_partner_is_happy.
```

- To express that some statements do not hold, we use the built-in predicate `false`:

```
% good_lecture: I'm at a good lecture.
good_lecture :- false.
```

```
% hot: It's hot.
hot :- false.
```

- Alternatively, this *directive* makes Prolog return false if an undefined predicate is called:

```
:- initialization set_prolog_flag(unknown, fail).
```

The structure of Prolog programs

- A Prolog program is a sequence of clauses. A clause can be a
  - a ⟨ fact ⟩ of the form "⟨ head ⟩.", e.g. `my_partner_is_happy.`
  - a ⟨ rule ⟩ of the form "⟨ head ⟩ `:-` ⟨ body ⟩.", e.g. `happy :- my_partner_is_happy.`
- The meaning of the clauses:
  - A ⟨ fact ⟩ is unconditionally true.
  - A ⟨ rule ⟩ states that its ⟨ head ⟩ is true if its ⟨ body ⟩ is true. Read "`:-`" as "if".
- A ⟨ body ⟩ is a sequence of one or more ⟨ goal ⟩s, separated by commas.
- A ⟨ body ⟩ is true if all its ⟨ goal ⟩s are true. Read "," as "and".
- The ⟨ goal ⟩ and ⟨ head ⟩ are both Prolog ⟨ term ⟩s, initially just names. Names are alphanumeric sequences starting with a lower case letter – these are called *atoms* in Prolog. Alphanumeric characters include letters, digits and the underline.
- The clauses with the same (or similar, see later) head form a predicate.
- Precede each predicate by a head comment describing its meaning.
- A comment lasts from a "%" until the end of line, or from "/*" to "*/".

Executing Prolog programs

- To execute the "happy" program, load it into the Prolog system and issue a ⟨ query ⟩:

```
| ?- happy.
yes
```

- A ⟨ query ⟩ is a ⟨ body ⟩, i.e. a sequence of goals. It may
  - succeed, i.e. return true (cf. the above `yes` answer) – in which case it may also bind some variables, possibly in several alternative ways, see later; or
  - fail, i.e. return false (displaying `no`) – in which case it will not bind any variables.

Executing the “happy” example

```
% happy: I'm happy.
happy :-
     workday, good_lecture.
happy :-
    hot, swimming.
happy :-
    my_partner_is_happy.

workday.

swimming.

my_partner_is_happy.
```

```
| ?- trace, happy.
% The debugger will first creep -- showing everything
        1      1 Call: happy ?
        2      2 Call: workday ?
        2      2 Exit: workday ?
        3      2 Call: good_lecture ?
        3      2 Fail: good_lecture ?
        4      2 Call: hot ?
        4      2 Fail: hot ?
        5      2 Call: my_partner_is_happy ?
        5      2 Exit: my_partner_is_happy ?
        1      1 Exit: happy ?
yes
% trace
| ?-
```

Summary of "propositional" Prolog

- So far we discussed a sublanguage of Prolog where predicates have no arguments (cf. propositional logic).
- A Prolog program is a collection of predicates (also called procedures), a predicate is a list of clauses.
- A clause can be a fact "`H.`", or a rule "`H :- B.`". Both have a head `H` (consequence), and a rule has a body `B` (precondition). A fact can be viewed as a rule with an empty body, or as a rule of the form "`H :- true.`" (where `true` is a built-in predicate which always succeeds).
- The declarative meaning of a clause "`H :- B.`" is the implication: `H` follows from `B`.
- The procedural meaning of a clause "`H :- B.`" is the following: in order to find out that `H` is true, try to find out if `B` is true.
- When executing a procedure, its clauses are considered in the order they are written.
  - If a body is found which executes successfully, then the procedure exits with success, otherwise it fails
  - A body conjunction is executed from left to right until a conjunct fails.

Contents

Adding arguments to heads and goals

- So far we used just names for clause heads and body goals.
- We now extend this to allow names followed by arguments, e.g. the term `happy(P, D)` may have the meaning: Person `P` is happy on day `D`.
- A *callable* Prolog term may thus be an atom (an alphanumeric sequence starting with a lower case letter), or an atom followed by a parenthesised, comma-separated list of arguments.
- An argument is a Prolog term, for now it may be an atom, a number, or a variable (an alphanumeric sequence starting with an upper case letter or an underline)
- An example:

```
% happy(P, D): Person P is happy on day D.
happy(P, D) :-          % For all P and D: person P is happy on day D if
    hot(D),             % it's hot on day D and
    swimming(P, D).     % person P is swimming on day D.
```

- The head comment of the predicate takes the form of an English sentence describing the relationship between the predicate arguments (and all arguments have to be present).
- All variables in a clause are universally quantified, so the scope of a variable is a single clause.

Selecting clauses – unification

- Clause selection uses bi-directional pattern matching, called *unification*.
- Prolog terms A and B can be unified if there is a (possibly empty) substitution σ of variables by Prolog terms, which makes the terms identical: $A\sigma = B\sigma$, e.g.
  - `sw(john,mon)` and `sw(john,mon)` unify, $\sigma = \{\}$
  - `sw(john,When)` and `sw(john,mon)` unify, $\sigma = \{$`When`$\leftarrow$`mon`$\}$
  - `sw(john,When)` and `sw(Who,mon)` unify, $\sigma = \{$`When`$\leftarrow$`mon, Who`$\leftarrow$`john`$\}$
  - `happy(john,When)` and `happy(P,D)` unify, $\sigma = \{$`P`$\leftarrow$`john, D`$\leftarrow$`When`$\}$ (a variable can be substituted by another variable)
  - `happy(john,When)` and `happy(P,D)` also unify using $\sigma = \{$`P`$\leftarrow$`john, D`$\leftarrow$`mon, When`$\leftarrow$`mon`$\}$.
- As the last two examples show, two terms can be unified by several substitutions.
- Prolog uses the *most general unifier* (mgu) substitution. This is a substitution from which all other unifying substitutions can be obtained by specialisation, i.e. applying another substitution.
- The substitutions in all but the last example are mgu's.
- It can be shown that apart from variable renaming, mgu is unique.

Predicates

- Two Prolog terms unify only if they have the same name and the same number of arguments (also called arity, cf. the term “n-ary relation”).
- If a term has the name *f* and the arity *n*, then the expression *f*/*n* is referred to as the functor of the given term.
- Refine the notion of predicate: Two clauses belong to the same predicate if their heads have the same functor (the functor of the predicate). The predicate below has the functor `swimming/2`.

```
swimming(kate, tue).
swimming(claire, thu).
swimming(claire, wed).

| ?- swimming(john, mon).
yes
| ?- swimming(john, tue).
no
| ?- swimming(claire, When).
When = thu ? ;
When = wed ? ;
When = mon ? ;
no
```

```
|  swimming(claire, mon).
|  swimming(john, mon).
|  swimming(john, sat).

|  | ?- swimming(Who, mon).
|  Who = claire ? ;
|  Who = john ? ;
|  no
|  | ?- swimming(Who, When).
|  Who = kate, When = tue ? ;
|  Who = claire, When = thu ? ;
|  Who = claire, When = wed ? ;
|  (...)
```

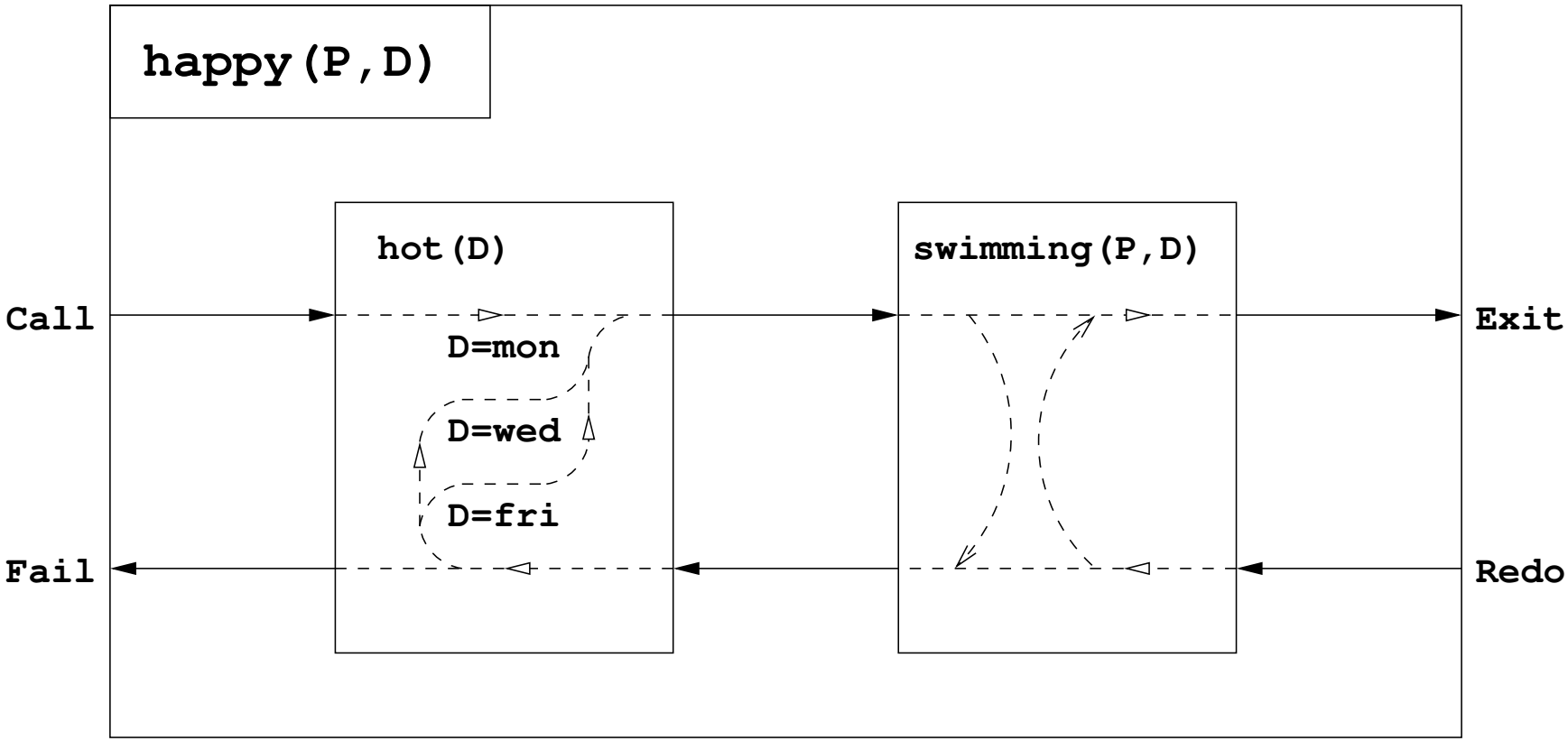
The procedure-box of the happy example

```
happy(P, D) :-
        hot(D),
        swimming(P, D).

hot(mon).
hot(wed).
hot(fri).

swimming(kate, tue).
swimming(claire, thu).
swimming(claire, wed).

swimming(claire, mon).
swimming(john, mon).
swimming(john, sat).
```



The ports of the procedure-box

- A procedure has four ports:
  - `Call`: The procedure is entered, the arguments are supplied.
  - `Exit`: The procedure is successfully completed. Variable substitutions representing the solution may be returned.
  - `Fail`: The procedure completes with failure.
  - `Redo`: The procedure is re-entered, for requesting another solution. The last port passed through must have been an `Exit` port.
- A Prolog procedure = an object (in OO sense) with the two methods: `Call` and `Redo`, each returning a Boolean value (`true` →`Exit` and `false` →`Fail`)
- A procedure may throw an exception – this is often considered a port.
- The question mark "?" at the beginning of the debugger output line for the `Exit` box means that the Prolog engine thinks there may be further solutions within the given procedure invocation.
- No question mark in an `Exit` line means that the Prolog engine knows that there are no more solutions within the given procedure invocation. The `Redo` port will not be entered in this case.

Tracing “happy”

```
% happy(P, D): Person P is happy on D.
happy(P, D) :-
        hot(D),
        swimming(P, D).

% hot(D): it's hot on D.
hot(mon).
hot(wed).
hot(fri).

% swimming(P, D): P is swimming on D.
swimming(kate, tue).
swimming(claire, thu).
swimming(claire, wed).
swimming(claire, mon).
swimming(john, mon).
swimming(john, sat).

| ?- happy(P, W).
P = claire, W = mon ? ;
P = john,   W = mon ? ;
P = claire, W = wed ? ;
no
```

```
| ?- trace, happy(P, W).
        1      1 Call: happy(_P,_W) ?
        2      2 Call: hot(_W) ?
?       2      2 Exit: hot(mon) ?
        3      2 Call: swimming(_P,mon) ?
?       3      2 Exit: swimming(claire,mon) ?
?       1      1 Exit: happy(claire,mon) ?
P = claire, W = mon ? ;
        1      1 Redo: happy(claire,mon) ?
        3      2 Redo: swimming(claire,mon) ?
?       3      2 Exit: swimming(john,mon) ?
?       1      1 Exit: happy(john,mon) ?
P = john, W = mon ? ;
        1      1 Redo: happy(john,mon) ?
        3      2 Redo: swimming(john,mon) ?
        3      2 Fail: swimming(_P,mon) ?
        2      2 Redo: hot(mon) ?
?       2      2 Exit: hot(wed) ?
        4      2 Call: swimming(_P,wed) ?
?       4      2 Exit: swimming(claire,wed) ?
?       1      1 Exit: happy(claire,wed) ?
P = claire, W = wed ? ;
(...)
```

Bi-directional nature of unification

```
% happy(P, D): Person P is happy on day D.
happy(P, D) :-
        hot(D),
        swimming(P, D).

% hot(D): it's hot on day D.
hot(_).                 % Every day is hot.

% swimming(P, D): Person P is swimming on day D.
swimming(kate, mon).
swimming(claire, _X).  % Claire is swimming every day.
swimming(john, wed).
```

```
| ?- happy(P, W).
P = kate, W = mon ? ;
P = claire ? ;
P = john, W = wed ? ;
no
| ?- happy(claire, W).
true ? ;
no
```

```
| ?- trace, happy(claire, W).
        1      1 Call: happy(claire,_520) ?
        2      2 Call: hot(_520) ?
        2      2 Exit: hot(_520) ?
        3      2 Call: swimming(claire,_520) ?
        3      2 Exit: swimming(claire,_520) ?
        1      1 Exit: happy(claire,_520) ?
true ? ;
no
```

The procedure box of a predicate with multiple clauses

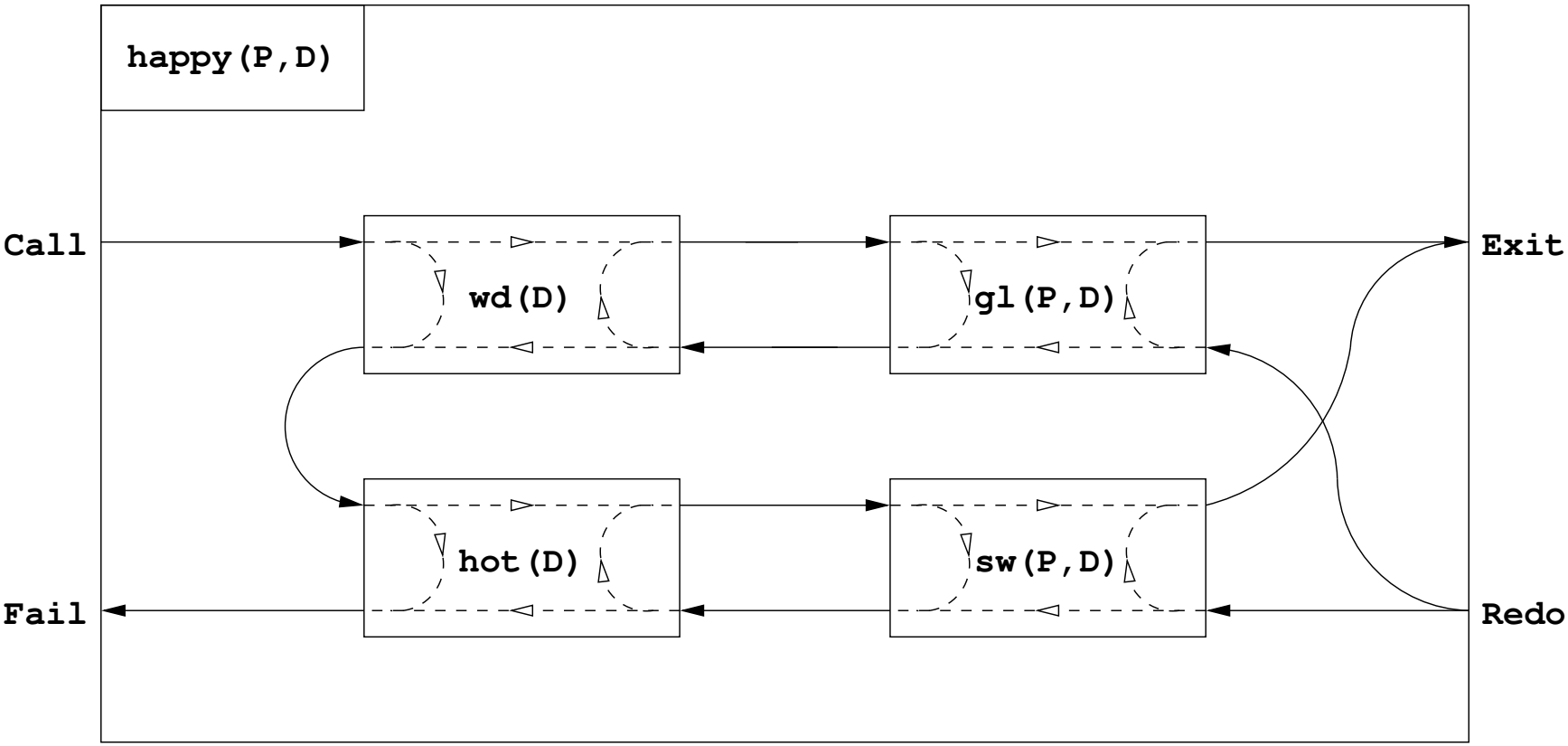
```
happy(P, D) :-
    wd(D), gl(P, D).
happy(P, D) :-
    hot(D), sw(P, D).
```

```
wd(mon).
wd(tue).
wd(wed).
wd(thu).
wd(fri).
```

```
gl(peter, mon).
gl(claire, sat).
gl(john, wed).

sw(graham, sat).
sw(john, wed).
```

```
hot(_).
```

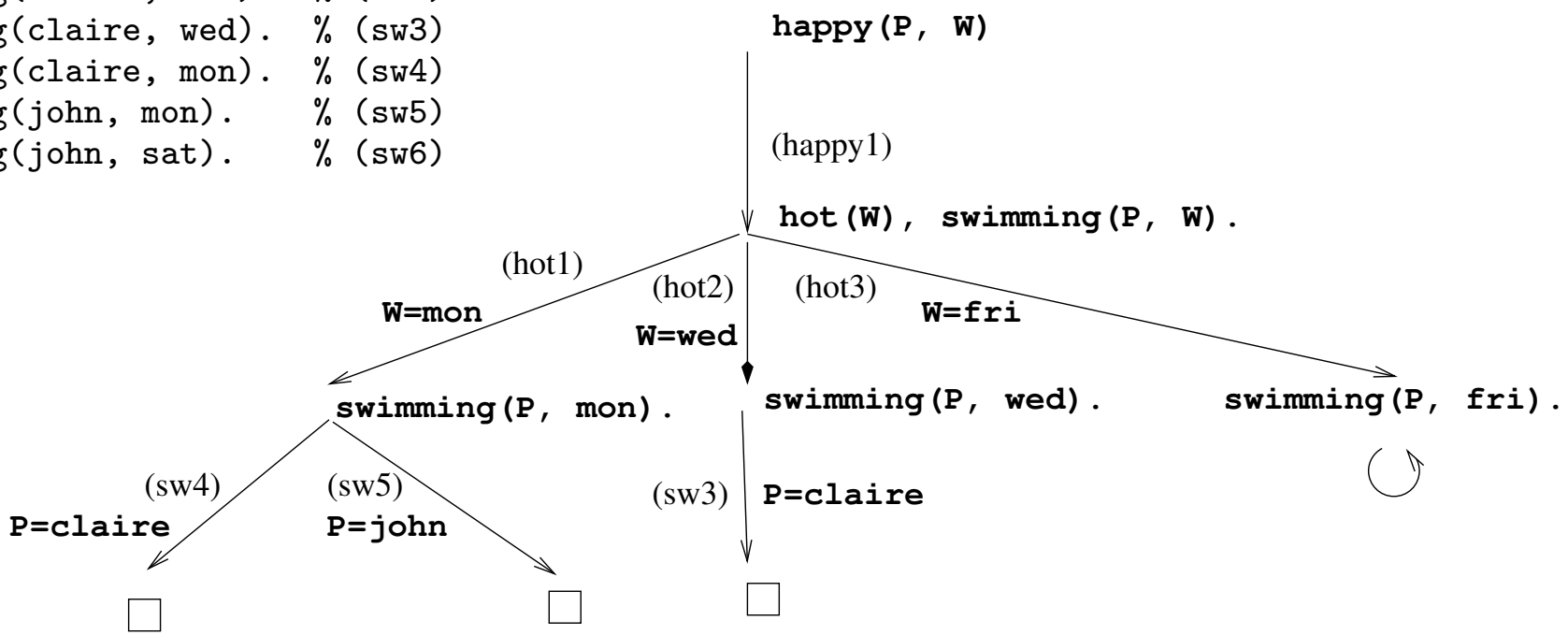


Another Procedural Model for Prolog: Goal Reduction

- Executing the original "happy" example with the goal reduction model

```
happy(P, D) :-              % (happy1)    |    hot(mon).                % (hot1)
        hot(D),                            |    hot(wed).                % (hot2)
        swimming(P, D).                    |    hot(fri).                % (hot3)

swimming(kate, tue).    % (sw1)
swimming(claire, thu).  % (sw2)
swimming(claire, wed).  % (sw3)
swimming(claire, mon).  % (sw4)
swimming(john, mon).    % (sw5)
swimming(john, sat).    % (sw6)
```



The Goal-Reduction Model for Prolog Execution

- The main idea of the goal-reduction model
  - The execution state: a query, i.e. a sequence of goals
  - The execution consists of two kinds of steps:
    - reduction step: query + a clause $\rightarrow$ a new query ($Q + Cl_i \rightarrow NQ$)
    - backtracking step (when a failure, i.e. a dead end is reached): continue at the most recent choice point
  - A choice point stores $\langle Q, i \rangle$ i.e. the query Q **before** the reduction and the counter i of the matched clause
    - Created at each reduction step except when Cl_i is the last
    - Backtracking: go to the query of the **most recent** choice point and try **further** matching clauses, i.e. start with clause Cl_{i+1}
- Prolog search tree: a graphical representation of execution using the reduction model
  - The nodes of the tree are traversed using depth-first search
  - The Prolog execution engine has to store the choice points on the path from the root to the current node of the search tree – this the choice point stack.

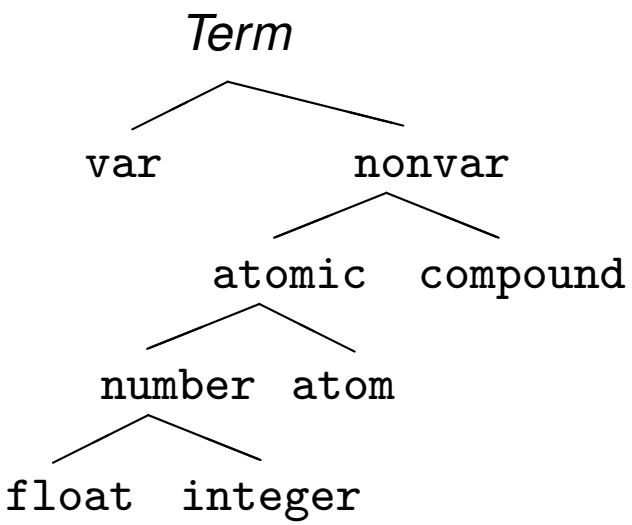
The reduction model – more details

- Reduction step: reduce a query Q to a new query NQ using a program clause Cl_i:
  - **Copy** clause Cl_i, i.e. systematically replace all variables by new ones, giving H :- B
  - Split query Q into a first goal Q_0 and a residual query RQ
  - **Unify** the goal Q_0 and the head H resulting in a substitution σ
    - If the goal and head are not unifiable then the reduction step fails
  - Return the new query $NQ = (B, RQ)\sigma$ (i.e. append the clause body and the residual query and apply σ to it)
- To execute a query Q, reduce with each applicable clause, top-to-bottom
- If the reduction of a query Q with a clause Cl_i is successful and results in a query NQ
  - create a choice point which stores $\langle Q, i \rangle$, unless Cl_i is the last clause
  - If NQ is empty, exit with success, otherwise execute NQ
- If no successful reduction is found, backtracking occurs:
  - if the most recent choice point contains $\langle Q, i \rangle$ then go back to query Q and continue trying to reduce it using the clauses after Cl_i.
  - if there are no choice points, exit the whole execution with failure

Contents

The notion of Prolog term revisited

- Prolog is a dynamically typed language
- 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

- Variables, e.g. `X`, `Parent`, `X2`, `_var`, `_`, `_123`
  - A variable is initially uninstantiated, it can be instantiated once during unification to an arbitrary Prolog term (including another variable)
- Constants
  - numbers: integer or float, eg. `1`, `-2.3`, `3.0e10`
  - atoms, i.e. symbolic constants, e.g. `'Peter'`, `has_son`, `+`, `<<`, `[]`

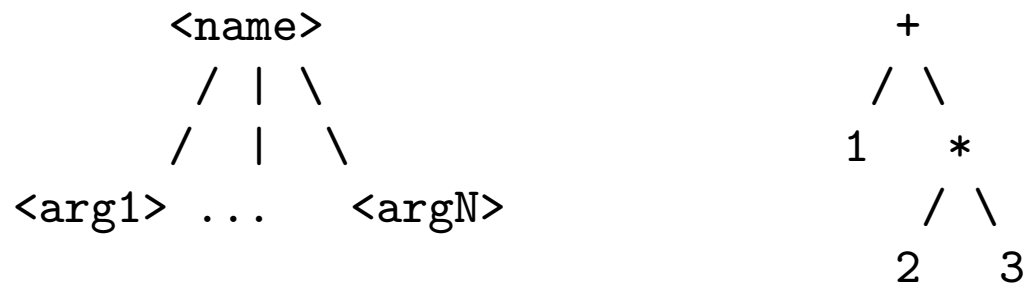
Composite data structures in Prolog

Compound terms in Prolog (sometimes called records or a structures)

- base (canonical) form: ⟨ name ⟩`(`⟨ arg_1 ⟩`, ...)`
  - the ⟨ name ⟩ is an atom,
  - the ⟨ arg_i ⟩ arguments are arbitrary Prolog terms
  - examples: `leaf(1)`, `person(william,smith,2003,1,22)`, `<(X,Y)`, `is(X, +(Y,1))`
- non-canonical form, syntactically sweetened.
  - Operators, e.g. `X is Y+1` ≡ `is(X, +(Y,1))`
  - List notation, e.g. `[1,2]` ≡ `.(1,.(2,[]))`
  - The BIP `write_canonical(X)` writes out `X` in canonical form

    `| ?- write_canonical(1+2*3).` ⟹ `+(1,*(2,3))`

    `| ?- write_canonical([1,2]).` ⟹ `'.'(1,'.'(2,[]))`
- Compound terms are often represented as trees



Our first compound data structure – binary tree

- A binary tree data structure can be defined as being
  - either a node (`node`) which contains two subtrees (`left`,`right`);
  - or a leaf (`leaf`) which contains an integer
- Define 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;
};
```

```
% 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).

% Data type description,
% borrowed from Mercury.
% Appears as a comment.
% :- type tree --->
%       node(tree, tree)
%     | leaf(int).
```

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 node, add the sums of the two subtrees
  - if the tree is a leaf, return the integer in the leaf

```
% 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.
```

- `X is Expr` is a built-in predicate:
  evaluates arithmetic expression `Expr`, and unifies the result with `X`.
- I/O mode notation: `+`: input (bound), `-`: output (unbound var.), `?`: arbitrary.

Sum of Binary Trees

- A Prolog sample run:

```
% sicstus
SICStus 4.2.1 (x86-linux-glibc2.5): (...)
| ?- consult(tree).
% consulting /home/szeredi/examples/tree.pl...
% consulted /home/szeredi/examples/tree.pl in module user, (...)
yes
| ?- tree_sum(node(leaf(5),
                   node(leaf(3), leaf(2))), 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!

Tree summation working both ways (ADVANCED)

- If we use an "addition table" instead of the built-in addition, and are careful about infinite recursion, then we get a bi-directional `tree_sum/2` predicate:

```
% tree_sum(?Tree, ?Sum): The sum of the leaves of Tree is Sum.
tree_sum(leaf(Value), Value).
tree_sum(node(Left, Right), S) :-
        plus(S1, S2, S), tree_sum(Left, S1), tree_sum(Right, S2).

plus(1, 1, 2).  plus(1, 2, 3).  plus(1, 3, 4).
plus(2, 1, 3).  plus(2, 2, 4).  plus(3, 1, 4).

| ?- tree_sum(T, 3).
T = leaf(3) ? ;
T = node(leaf(1),leaf(2)) ? ;
T = node(leaf(1),node(leaf(1),leaf(1))) ? ;
T = node(leaf(2),leaf(1)) ? ;
T = node(node(leaf(1),leaf(1)),leaf(1)) ? ;
no
| ?- tree_sum(node(leaf(2),leaf(1)), S).
S = 3 ? ; no
```

Contents

Lists in Prolog

- The Prolog list
  - The empty list is the `[]` atom. A non-empty list is a compound `'.'(Head,Tail)` where
    - `Head` is the first element of the list, while
    - `Tail` is the list composed of the remaining elements.
  - Lists can be written in simplified form ("syntactic sugar").
  - The implementation of lists is optimised: it is more space- and time-efficient than for other compound structures.

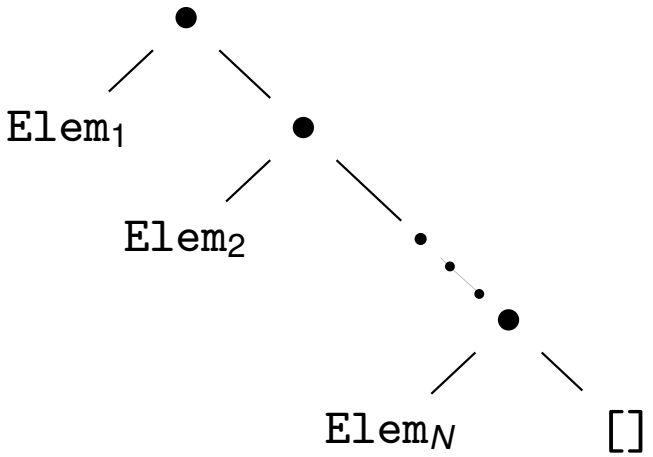
```
% list_of_numbers(L): L is a list of numbers.
list_of_numbers(.(E,L)) :-
    number(E), list_of_numbers(L).
list_of_numbers([]).
```

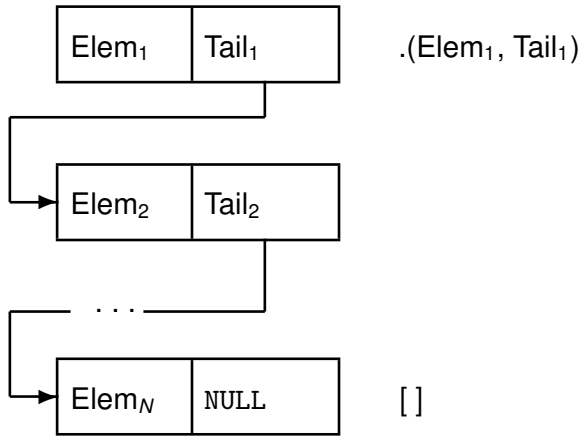
```
| ?- listing(list_of_numbers).
list_of_numbers([A|B]) :-
    number(A), list_of_numbers(B).
list_of_numbers([]).
```

```
| ?- list_of_numbers([1,2]).    %  [1,2] == .(1,.(2,[])) == [1|[2|[]]]
     yes
| ?- list_of_numbers([1,a,f(2)]).
     no
```

Various ways for writing lists

- Options for writing a list of N elements:
  - canonical form : $.(\texttt{Elem}_1, .(\texttt{Elem}_2, \ldots, .(\texttt{Elem}_N, [])\ldots))$
  - an equivalent list notation: $[\texttt{Elem}_1, \texttt{Elem}_2, \ldots, \texttt{Elem}_N]$
- The tree structure of lists and their implementation





The list notation – more syntactic sugar

- `[Head|Tail]` $\equiv$ `.(Head, Tail)`
- As a generalisation of the above:
  $[E_1, E_2, \ldots, E_N|Tail] \equiv [E_1|[E_2|\ldots, E_N|Tail]\ldots]$
- When the tail is `[]`: $[E_1, E_2, \ldots, E_N] \equiv [E_1, E_2, \ldots, E_N|[]]$
- Examples:

```
| ?- [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 = [1|_], L = [_,2|_].           ⇒   L = [1,2|_A] ? % open ended
| ?- L = .(1,[2,3|[]]).                ⇒   L = [1,2,3] ?
| ?- L = [1,2|.(3,[])].                ⇒   L = [1,2,3] ?
| ?- [X|[3-Y/X|Y]]= .(A, [A-B,6]).     ⇒   A=3, B=[6]/3, X=3, Y=[6] ?
```

Concatenating lists – append/3

- `append(L1, L2, L3)` is meant to express the following relation:
  - the concatenation of lists `L1` and `L2` is `L3`; or, in other words:
  - list `L3` consists of the elements of `L1` followed by those of `L2`.
- We will use the $\oplus$ sign to denote list concatenation
- The predicate `append/3` is built-in in SICStus Prolog 4, but it could[2] be defined in Prolog as:

```
% append(L1, L2, L3): L3 = L1⊕L2,
% i.e. the concatenation of L1 and L2 is L3.
append([], L, L).                % The conc. of [] and L is L.
append([X|L1], L2, [X|L3]) :- % The conc. of [X|L1] and L2 is [X|L3] if
    append(L1, L2, L3).          % the conc. of L1 and L2 is L3.
```

- Try reading the meaning of the second clause backwards:
  If the concatenation of `L1` and `L2` is `L3`
  then the concatenation of `[X|L1]` and `L2` is `[X|L3]`.

[2] Built-in predicates cannot be redefined, so one would have to use another name, e.g. app.

Predicate append/3 – procedural reading

```
% append(L1, L2, L3): The concatenation of L1 and L2 is L3.
append([], L, L).
append([X|L1], L2, [X|L3]) :-
        append(L1, L2, L3).
```

- Assume all three arguments are input (given lists), i.e. the I/O mode is `append(+,+,+)`. To execute `append(A, B, C)`, i.e. to check that `A` $\oplus$ `B` = `C`:
  - If `A` is an empty list (`A = []`) and `B = C` then return success.
  - If `A` is a non-empty list (`A = [X|L1]`), and `C` has the same head as `A` (`C = [X|L3]`), then proceed to check if (`A`'s tail $\oplus$ `B`) = (`C`'s tail), i.e. execute `append(L1, B, L3)`.
  - Otherwise fail.
- Assume mode `append(+,+,-)` – the third arg. is output (uninst. variable). To construct `A` $\oplus$ `B` in `C`, i.e. to execute `append(A, B, C)`:
  - If `A` is an empty list (`A = []`), then set `C = B` and return success.
  - If `A` is a non-empty list (`A = [X|L1]`), then set `C`'s head to that of `A` (`C = [X|L3]`), then construct (`A`'s tail $\oplus$ `B`), making this the tail of `C`, i.e. execute `append(L1, B, L3)`.
- `append(+L,_,_)` completes in $\sim n$ reduction steps when `L` has length n

Open ended lists, and their use in append

- A list is *open ended* iff it is an unbound variable, or its tail is open ended
- A list is *closed* or *proper* iff sooner or later an `[]` appears as the tail
- For example, `[X,1,Y]` is a proper list, `[X,1|Y]` is open ended.

```
append([], L, L).                                  (cl1)
append([X|L1], L2, [X|L3]) :-  append(L1, L2, L3).  (cl2)
```

```
| ?- append([1,2], [4,5], C).                      (1)
```

- Head of clause (cl1) does not unify. Unify goal (1) with the head of (cl2):
  - unify `[1,2]=[1| [2] ]` with `[X|L1]` $\longrightarrow$ `X = 1, L1 = [2]`;
  - unify `[4,5]` with `L2` $\longrightarrow$ `L2 = [4,5]`;
  - unify `C` with `[X|L3] = [1|L3]` $\longrightarrow$ `C = [1|L3]`
- Proceed to execute `append([2], [4,5], L3).` (2)
- Head of clause (cl1) does not unify. Unify goal (2) with the head of (cl2):
  `[2]=[2|[]] = [X'|L1']` $\longrightarrow$ `X' = 2, L1' = []; L2' = [4,5]; L3 = [2|L3']`.
- Proceed to execute `append([], [4,5], L3').` (3)
- Unify (3) with (cl1): `L3' = [4,5]`. Thus `C = [1|L3], L3 = [2|L3'], L3' = [4,5]`, hence `C = [1,2,4,5]`.

Tail recursion optimisation

- Tail recursion optimisation (TRO), or more generally last call optimisation (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 `append/3`:

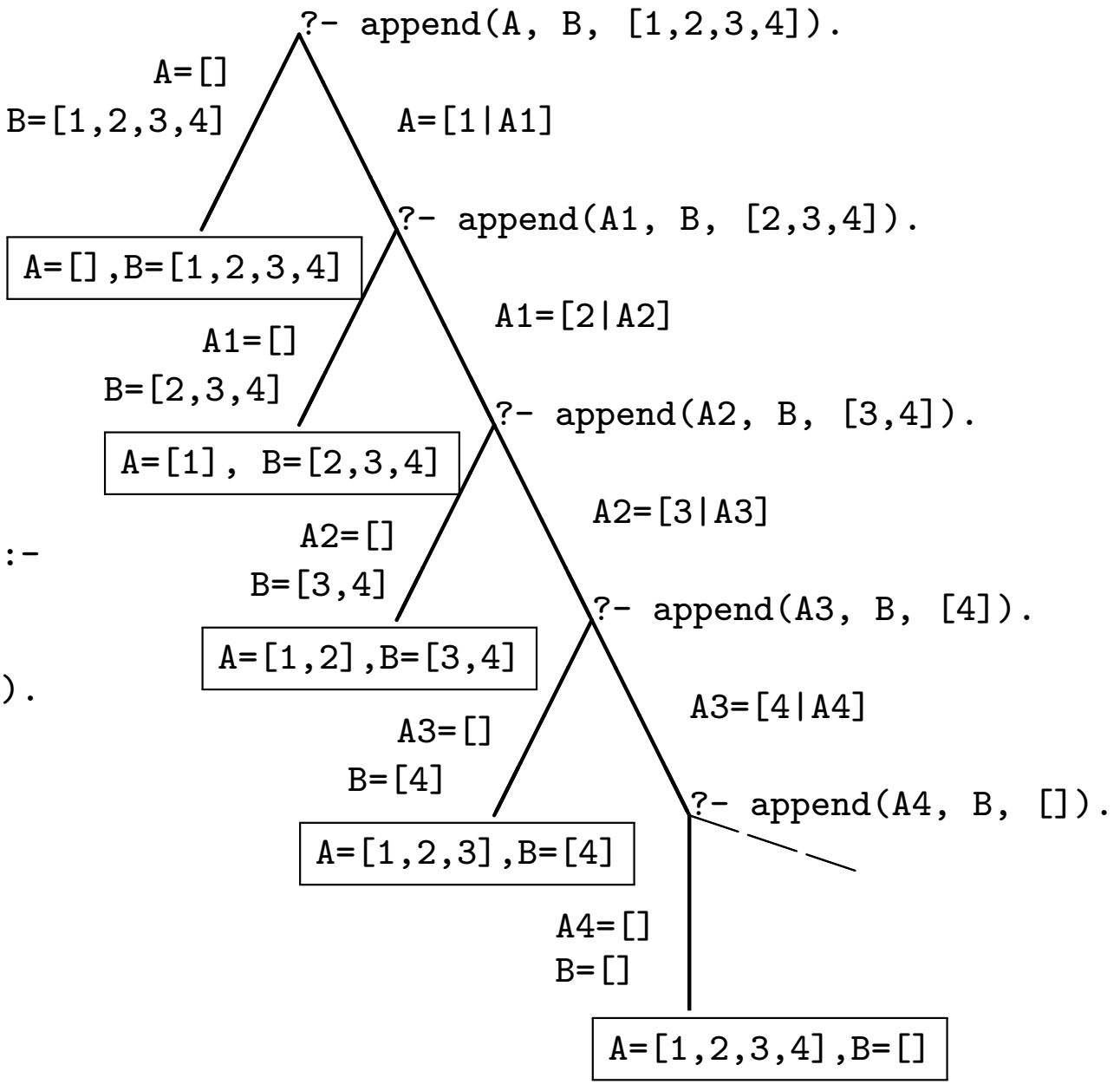
```
% append(L1, L2, L3): The concatenation of L1 and L2 is L3.
append([], L, L).
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
```

- This feature relies on open ended lists:
  - It is possible to build a list node *before* building its tail
  - Imperatively, this corresponds to passing to append a pointer which points 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/3

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

| ?- append(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
```



append/3 – what other I/O modes are possible?

- Discussed so far: check – `append(+,+,+)`, concatenate – `append(+,+,-)`, split – `append(-,-,+)`
- Variations on splitting: "subtract" front – `append(+,-,+)`, subtract back – `append(-,+,+)`

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

- Append a yet unknown, or open ended, list – `append(+,-,-)`
  No problem, creates an open ended list!

```
| ?- append([a,b], Tail, L).          ⟹  L = [a,b|Tail] ? ; no
| ?- append([a,b], [c|T], L).         ⟹  L = [a,b,c|T] ? ; no
```

- The search space becomes infinite, if the first **and** the third arguments are **both** open ended – modes `append(-,+,-)` and `append(-,-,-)`

```
| ?- append([1|X], [a,b], Y).         ⟹
                  X = [], Y = [1,a,b] ? ;
                  X = [_A], Y = [1,_A,a,b] ? ;
                  X = [_A,_B], Y = [1,_A,_B,a,b] ? ;        ad infinitum
```

- The recursive depth of executing `append`$(L_1, L_2, L_3) \leq min(len(L_1), len(L_3))$

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(R, L): List R is the reverse of list L.
reverse(R, L) :-  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

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

Variations on `append` 1. — Appending three lists (ADVANCED)

- Recall: `append/3` has **finite** search space, if its 1st **or** 3rd arg. is proper. `append(L,_,_)` completes in $\sim n$ reduction steps when `L` has length n
- Let us define `append(L1,L2,L3,L123)`: `L1` $\oplus$ `L2` $\oplus$ `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 – creates infinite choice points
- An efficient version, suitable for splitting a given list to three parts:

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

  - `L3` can be open ended or proper, it does not matter
  - Recall that the first `append/3` call produces an open ended list:

```
| ?- append([1,2], L23, L).      ⟹      L = [1,2|L23]
```

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

- Elements occuring in pairs

```
% in_pair(+List, ?E, ?I):  E is an element of List equal to its
% right neighbour, occurring at position I (numbered from 1).
in_pair(L, E, I) :-
        append(Before, [E,E|_], L),
        length(Before, I0)[3], I is I0+1.
```

```
| ?- in_pair([1,8,8,3,4,4], E, I). ⟹     E = 8, I = 2 ? ;
                                   ⟹     E = 4, I = 5 ? ; no
```

- Stuttering sublists

```
% stutter(L, D): D is a nonempty sublist of L immediately
% followed by an identical sublist.
stutter(L, D, I) :-
        append(_Before, Tail, L),
        D = [_|_],
        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, I).
        ⟹          D = [2] ? ; D = [2,2,1] ? ; D = [2] ? ; no
```

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

Finding list elements – BIP `member/2`

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

- Mode `member(+,+)` – checking membership

```
| ?- member(2, [2,1,2]).          ⟹   yes                    BUT
| ?- member(2, [2,1,2]), X=X.     ⟹   true ? ; true ? ; 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 proper.

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 proper.

Permutation of lists (ADVANCED)

- `permutation(List, Perm)`: the permutation of `List` is list `Perm`

```
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]).                    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

Prolog implementation – some milestones

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

Term representation (ADVANCED)

- Prolog compound term storage – the two main approaches

	structure-sharing	structure-copying
Memory requirement	O(number of variables)	O(compound size)
Building time	constant	O(compound size)
Argument access time	higher	lower

- Building a compound: unify a free var. and a compound in program text
- **Important:** the unification of a free variable with a compound already stored in a variable has constant cost! (Except for occurs check.)
- Example: the time and space cost of `replicate(`n`, L)` is $O(n)$ with structure-sharing, $O(n^2)$ with structure-copying.

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

replicate(0, []).
replicate(N, L) :-
        N > 0, prefix_n(L0, L), N1 is N-1, replicate(N1, L0).
```

- Still, in practice, structure-copying proved to be more efficient.

WAM: Storage of Prolog terms (LBT – low bit tagging scheme)

- Prolog object

global/local stack *global stack only*

- Unbound variable:

own addr	REF

- Reference to other variable:

addr of var	REF

- Atom (symb. constant):

atom table index	A	CON

- Integer:

integer value	I	CON

- List:

addr	LIST

addr:

head term
tail term

- Compound:

addr	STRU

addr:

functor table index
argument term
...

- The SICStus 3 release used high bit tagging (*upper* 4 bits), thus the size of the stacks was limited to 256 MBs. SICStus 4 uses the LBT scheme.

WAM: Further details (ADVANCED)

- Handling of variables
  - Unification of two variables: the *younger* variable (the one creater later) becomes a reference to the *older*
  - **Dereferencing**: following and resolving a chain of references until an unbound variable or a non-REF object is reached
  - Uninstantiated variable $\equiv$ self reference $\Rightarrow$ easier dereferencing
- Backtracking
  - **Conditional variable**: uninstantiated variable, older than the newest choicepoint
  - When substituting a conditional variable, its address is stored on the trail
  - At backtrack, variable substitutions are undone using the trail, then the trail is truncated
- For further details on the WAM see the wiki page:
  `http://en.wikipedia.org/wiki/Warren_abstract_machine`
- Reference [2] is Hassan Aït-Kaci's downloadable tutorial: "Warren's Abstract Machine: A Tutorial Reconstruction"

Contents

The Unification Algorithm

- The unification algorithm takes (canonical) terms A and B as input.
- It returns the most general unifier of A and B, $\sigma = mgu(A, B)$, or failure.
- In practice, the substitution σ has to be applied to the query at hand.
- The (practical) unification algorithm:
  1. If A and B are identical variables or constants, then return success.
  2. Else, if A is a variable, then substitute $A \leftarrow B$ and return success.
  3. Else, if B is a variable, then substitute $B \leftarrow A$ and return success.
     (Steps 2 and 3 can be executed in arbitrary order, i.e. when both A and B are variables, one of them is substituted by the other)
  4. Else, if A and B are compounds with the same name and arity, and their arguments are $A_1,\ldots,A_N$ and $B_1,\ldots,B_N$ resp., then for $i = 1,\ldots,N$ do

     Perform (recursively) the unification alg. for A_i and B_i;
     If the recursive invocation fails, return failure;

     If the above loop completes, return success.
  5. In all other cases return failure (A and B are not unifiable)

The Occurs Check in unification (ADVANCED)

- Can one unify `X` and `f(Y,g(X))`?
  - Mathematically: *no*, `X` cannot be bound to a compound containing `X`
  - Theoretically, step 2 (and 3) of the unification alg. should include an "occurs check": before binding $A \leftarrow B$ check that no A occurs in B,
  - The (costly) check is almost always useless $\Longrightarrow$ not used by default.
- No occurs check $\Longrightarrow$ so-called cyclic terms may be created, e.g.

  ```
  | ?- X = s(1,X).   ⟹   X = s(1,s(1,s(1,s(1,s(...))))) ? ; no
  ```
- Unification with occurs check is available as a standard BIP:

  ```
  | ?- unify_with_occurs_check(X, s(1,X)).   ⟹   no
  ```
- Some Prologs (eg. SICStus) support the unification and other operations on cyclic terms

  ```
  | ?- X = s(X), Y = s(s(Y)), X = Y.          ⟹
          X = s(s(s(s(s(...))))), Y = s(s(s(s(s(...))))) ?
  ```

  (Other Prologs may go to infinite loop on this example.)

The Unification Algorithm – mathematical formulation

Preliminaries

- A substitution is a function σ which maps variables to arbitrary Prolog terms. $\mathtt{X}\sigma$ denotes σ applied to variable `X`
- Example: $\sigma = \{\mathtt{X} \leftarrow \mathtt{a},\ \mathtt{Y} \leftarrow \mathtt{s(b,B)},\ \mathtt{Z} \leftarrow \mathtt{C}\}$, $Dom(\sigma) = \{\mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$, e.g. $\mathtt{X}\sigma = \mathtt{a}$
- The substitution function can be naturally extended:
  - $T\sigma$: σ applied to an *arbitrary* term T: all occurrences in T of variables in $Dom(\sigma)$ are *simultaneously* substituted according to *sigma*
  - Example: $\mathtt{f(g(Z,h),A,Y)}\sigma = \mathtt{f(g(C,h),A,s(b,B))}$
- Composition of substitutions:
  $\sigma \otimes \theta$ is a substitution obtained by first performing σ and then θ
  - Subst. $\sigma \otimes \theta$ maps variables $x \in Dom(\sigma)$ to $(x\sigma)\theta$, while variables $y \in Dom(\theta) \backslash Dom(\sigma)$ to $y\theta$ ($Dom(\sigma \otimes \theta) = Dom(\sigma) \bigcup Dom(\theta)$):

    $$\sigma \otimes \theta = \{x \leftarrow (x\sigma)\theta \mid x \in Dom(\sigma)\} \bigcup \{y \leftarrow y\theta \mid y \in Dom(\theta) \backslash Dom(\sigma)\}$$

  - For example, $\theta = \{\mathtt{X} \leftarrow \mathtt{b},\ \mathtt{B} \leftarrow \mathtt{d}\}$
    $\sigma \otimes \theta = \{\mathtt{X} \leftarrow \mathtt{a},\ \mathtt{Y} \leftarrow \mathtt{s(b,d)},\ \mathtt{Z} \leftarrow \mathtt{C},\ \mathtt{B} \leftarrow \mathtt{d}\}$

The Unification Algorithm – mathematical formulation

- The unification algorithm takes (canonical) terms A and B as input.
- It returns the most general unifier of A and B, $\sigma = mgu(A, B)$, or failure.
  1. If A and B are identical variables or constants, then return $\sigma = \{\}$ (empty substitution).
  2. Else, if A is a variable, then return $\sigma = \{A \leftarrow B\}$
  3. Else, if B is a variable, then return $\sigma = \{B \leftarrow A\}$ (the order of steps 2 and 3 is arbitrary, they may involve an occurs check)
  4. Else, if A and B are compounds with the same name and arity, and their arguments are $A_1,\ldots,A_N$ and $B_1,\ldots,B_N$ resp., then initialise $\sigma = \{\}$ and for $i = 1,\ldots,N$ do

     Perform (recursively) the unification alg. for $A_i\sigma$ and $B_i\sigma$; If the recursive invocation fails, return failure, otherwise set $\sigma = \sigma \otimes mgu(A_i, B_i)$

     If the above loop completes, return σ
  5. In all other cases return failure (A and B are not unifiable)

The goal reduction execution algorithm

The definition of reduction step

- Reduce a query Q to a new query NQ using a program clause Cl_i:
  - Split query Q into a first goal Q_0 and a residual query RQ
  - **Copy** clause Cl_i, i.e. introduce new variables, and split the copy to a head H and body B
  - **Unify** the goal Q_0 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 query $NQ = (B, RQ)\sigma$
      (i.e. apply σ to both the body and the residual query)
- reduce a query Q to a new query NQ by executing a built-in goal (when the first goal is a built-in procedure call):
  - Split query Q into a built-in goal Q_0 and a residual query RQ
  - **Execute** the BIP Q_0
    - If the BIP fails then exit the reduction step with failure
    - If the BIP succeeds with a substitution σ then return the new query $NQ = RQ\sigma$

The Goal Reduction Algorithm for Prolog execution

The algorithm uses a variable `QU`, storing a query, a variable `I` which is a clause counter; and a stack consisting of pairs of the form `<QU,I>`

1. *(Initialisation:)* The stack is initalised to empty, `QU := query`
2. *(BIP:)* If the first call of `QU` is built-in then perform a reduction step,
   a. If it fails $\Rightarrow$ step 6.
   b. If it is succeeds, `QU :=` the result of reduction step, $\Rightarrow$ step 5.
3. *(Non built-in procedure – initialise a clause counter)* `I := 1`.
4. *(Reduction step:)* Select the list of clauses applicable to the first call of `QU`.[4] Assume the list has `N` elements.
   a. If `I > N` $\Rightarrow$ step 6.
   b. perform a reduction step between the `I`th clause of the list and `QU`.
   c. If this fails, then `I := I+1`, $\Rightarrow$ step 4 a.
   d. If `I < N` (non-last clause), then push `<QU,I>` on the stack.
   e. `QU :=` the query returned by the reduction step
5. *(Success:)* If `QU` is nonempty $\Rightarrow$ step 2, otherwise exit with success.
6. *(Failure:)* If the stack is empty, then exit with failure.
7. *(Backtrack:)* Pop `<QU,I>` from the stack, `I := I+1`, and $\Rightarrow$ step 4.

[4] If there is no indexing, then this list will contain all clauses of the predicate. With indexing this will be an appropriate subset of all clauses.

Indexing

- What is indexing?
  - quick 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 `C` (atom or number);
  - `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, the compiler builds the list of matching clauses.
  - At run-time: the Prolog engine selects the relevant clause list using the call argument, if it is instantiated. By using *hashing*, this selection is done in practically constant time.
  - **Important:** if the selected list contains a single clause, *no choice point* is created.

An example of indexing

- A sample program to illustrate indexing.

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

```
q(1).
q(2).
```

- For the `p(A, B)` call, the compiler builds a switch selecting the list of applicable clauses:
  - if `A` is a variable: `(1) (2) (3) (4) (5)`
  - if `A = 0`: `(1) (2)`
  - if the main functor of `A` is `s/1`: `(2) (3) (4)`
  - if `A = 9`: `(2) (5)`
  - in all other cases: `(2)`
- Example calls:
  - `p(1, Y)` does not create a choice point.
  - `p(s(1), Y)` creates a choice point, but removes it and exits without leaving a choice point.
  - `p(s(0), Y)` exits leaving a choice point.

Indexing – further details

- If there are clauses where the first arguments have the same functor, an auxiliary predicate can help. E.g., by transforming `p/2` to `q/2` one avoids choice points for goals of the form `q(`*Ground*[5]`, Y)`

```
p(0, a).
p(s(0), b).
p(s(1), c).
p(9, z).
```

```
q(0, a).
q(s(X), Y) :-
    q_aux(X, Y).
q(9, z).
```

```
q_aux(0, b).
q_aux(1, c).
```

- Indexing does not deal with arithmetic comparisons.
  - E.g., `N = 0` and `N > 0` are not recognised as mutually exclusive.
- Indexing and lists
  - Putting the (input) list in the first argument makes indexing work.
  - Indexing distinguishes between `[]` and `[...|...]` (resp. functors: `'[]'/0` and `'.'/2`).
  - For proper lists, the order of the two clauses is not relevant
  - By putting the clause for `[]` first, one avoids an infinite loop when the predicate is called with an open ended list (but an infinite choice may still remain).

[5] A term is called ground if it contains no variable.

Indexing list handling predicates: examples (ADVANCED)

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

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

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

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

- The variant `last2/2` uses a helper predicate, creates no choice points:

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

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

- A variant of `member/2` with no choice point left at the last element:

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

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

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 ⟩ . . .	
⟨ predicate ⟩	::=	⟨ clause ⟩ . . .	{with the same functor}
⟨ clause ⟩	::=	⟨ fact ⟩.␣ |	
		⟨ rule ⟩.␣	{clause functor = head functor}
⟨ fact ⟩	::=	⟨ head ⟩	
⟨ rule ⟩	::=	⟨ head ⟩:-⟨ body ⟩	
⟨ body ⟩	::=	⟨ goal ⟩, . . .	
⟨ goal ⟩	::=	⟨ term ⟩	
⟨ head ⟩	::=	⟨ term ⟩	

Prolog terms

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}
		⟨ comp. term ⟩ |	{⟨ comp. name ⟩/⟨ arity ⟩}
		. . . syntax extensions . . .	{lists, operators}
⟨ constant ⟩	::=	⟨ atom ⟩ |	{symbolic constant}
		⟨ number ⟩	
⟨ number ⟩	::=	⟨ integer ⟩ |	
		⟨ float ⟩	
⟨ comp. term ⟩	::=	⟨ comp. name ⟩ (⟨ argument ⟩, . . .)	
⟨ comp. name ⟩	::=	⟨ atom ⟩	
⟨ argument ⟩	::=	⟨ term ⟩	

Lexical elements

Examples

```
% variable:        Fact FACT _fact X2 _2 _
% atom:            fact ≡ 'fact' 'István' [] ; ',' += ** \= ≡ '\\='
% number:          0 -123 10.0 -12.1e8
% not an atom:     !=, Istvan
% 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} | \ ⟨ escape sequence ⟩

⟨ 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 the string `/*` 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 the clauses of a predicate continuously, with no empty lines between
  - Precede each predicate by an empty line and a 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

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 ⟩)	{parenthesised 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 integer between 1–1200 – smaller priorities bind tighter
  - `Type` determines the placement of the operator and the associativity: infix: `yfx`, `xfy`, `xfx`; prefix: `fy`, `fx`; postfix: `yf`, `xf`
  - `Op` or $\texttt{Op}_i$: 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 parenthesised 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	+

Adding parentheses to expressions with operators – general rules

- Let term `T = X` $\mathtt{op}_1$ `Y` $\mathtt{op}_2$ `Z`, where $\mathtt{op}_1$ and $\mathtt{op}_2$ have priority n_1 and n_2:
  - if $n_1 > n_2$ then `T` $\equiv$ `X` $\mathtt{op}_1$ `(Y` $\mathtt{op}_2$ `Z)`;
  - if $n_1 < n_2$ then `T` $\equiv$ `(X` $\mathtt{op}_1$ `Y)` $\mathtt{op}_2$ `Z`;
  - if $n_1 = n_2$ and $\mathtt{op}_1$ is right-assoc (`xfy`), then `T` $\equiv$ `X` $\mathtt{op}_1$ `(Y` $\mathtt{op}_2$ `Z)`; otherwise,
  - if $n_1 = n_2$ and $\mathtt{op}_2$ is left-assoc. (`yfx`), then `T` $\equiv$ `(X` $\mathtt{op}_1$ `Y)` $\mathtt{op}_2$ `Z`;
  - otherwise `T` contains a syntax error
- An interesting example: `:- op(500, xfy, +^).`

  ```
  | ?- :- write((1 +^ 2) + 3), nl.   ⇒ (1+^2)+3
  | ?- :- write(1 +^ (2 + 3)), nl.   ⇒ 1+^2+3
  ```

  thus: in such a conflict, the associativity of the first operator “wins”.
- Base rule: an operator of priority n accepts an (unparenthesised) term T
  - on the `x` side, if the priority of the outermost operator of $T \leq n - 1$
  - on the `y` side, if the priority of the outermost operator of $T \leq n$
- This rule is also applicable to prefix and postfix operators
- Explicit parentheses overrule all the above

Operators – additional comments

- The twofold role of the “comma”
  - it separates the arguments of a compound term
  - an `xfy` op. of priority 1000, e.g.: `(p:-a,b,c)`$\equiv$`:-(p,','(a,','(b,c)))`
- Disambiguation: if the outermost operator of a compound argument has priority $\geq$ 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
- Can an atom be an operator with multiple types simultaneously?
  - not for types from the same class, e.g. `xfy` and `xfx`
  - but otherwise, yes, e.g. built-in ops `+` and `-` have types `yfx` and `fy`
- To make parsing simpler, the Prolog standard stipulates that
  - an operator used as an operand has to be parenthesised: `Cmp = (>)`
  - an operator cannot be declared both infix and postfix
- These restrictions are not compulsory in many Prolog systems.

Use of operators

- What are operators good for?
  - to allow usual arithmetic expressions, such as in `X is (Y+3) mod 4`
  - symbolic processing of expressions (such as symbolic derivation)
  - for writing the clauses themselves (`:-` and `','` are both operators)
    - clauses can be passed as arguments to meta-predicates:

      ```
      asserta( (p(X):-q(X),r(X)) )
      ```
  - to make clause heads and procedure calls more readable:

    ```
    :- op(800, xfx, in).
    X in L :-  member(X, L).
    ```
  - to make data structures more readable:

    ```
    :- op(100, xfx, [.]).
    acid(sulphur, h.2-s-o.4).
    ```
- Why are operators bad?
  - The pool of operators is a single global resource. This can cause problems in a larger project.

Arithmetic in Prolog

- Operators make it possible to write arithmetic expressions the usual way, as we do in mathematics or in other programming languages.
- BIP `is` expects an arithmetic expression on its right side (argument 2), evaluates it, and unifies the result with the argument on the left side.
- BIPs `=:=` (`<`, `>`, `=<`, `>=`, `=\=`) expect arithmetic expressions on both sides, evaluate these, and succeed if the first value is *arithmetically* equal to (less than, greater than, . . . , not equal to) the second value.
- Examples:

```
| ?- X = 1+2, write(X), write_canonical(X), Y is X.
⇒           1+2       +(1,2)    ⟹ X = 1+2, Y = 3 ? ; no
| ?- X = 4, Y is X/2, Y =:= 2.    ⟹ X = 4, Y = 2.0 ? ; no
| ?- X = 4, Y is X/2, Y = 2.      ⟹ no
```

- Terms composed of arithmetic operators (+,-,. . .) are **compound terms**. Only the arithmetic BIPs evaluate these as arithmetic expressions!
- Prolog terms are symbolic by default, arithmetic evaluation is the “exception”.

Classical symbolic processing: 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
```

An example: the substitution value of a polynomial

- Polynomial: a Prolog term built from numbers and the atom ‘x’, using the operators ‘+’ and ‘*’ .
- The task: calculate the value of a polynomial for a given value of x .

```
% value_of(Poly, X, V): V is the value of the polynomial Poly,
% for the substitution x=X
value_of(x, X, X).
value_of(Poly, _, V) :-  number(Poly), V = Poly.
value_of(P1+P2, X, V) :-
        value_of(P1, X, V1),
        value_of(P2, X, V2),
        V is V1+V2.
value_of(P1*P2, X, V) :-
        value_of(P1, X, V1),
        value_of(P2, X, V2),
        V is V1*V2.
```

```
| ?- value_of((x+1)*x+x+2*(x+x+3), 2, V).   ⟹   V = 22 ?
```

Contents