Computational Logic

The (ISO-)Prolog Programming Language
(ISO-)Prolog

- A practical logic language based on the logic programming paradigm.
- Main differences with “pure” logic programming:
  - more control on the execution flow,
  - depth-first search rule, left-to-right control rule,
  - some pre-defined predicates are not declarative (generally for efficiency),
  - higher-order and meta-logical capabilities,
  - no occur check in unification; but often regular (i.e., infinite) trees supported.
- Advantages:
  - it can be compiled into fast and efficient code,
  - more expressive power,
  - industry standard (ISO-Prolog),
  - mature implementations with modules, graphical environments, interfaces, ...
- Drawbacks: *incompleteness* (due to depth-first search rule),
  possible *unsoundness* (if no occur check and regular trees not supported).

Programming interface (writing and running programs)

- Not specified in the language standard.
- Specific to the particular system implementing the language.
- Covers issues such as:
  - User interaction (top-level, GUI, etc.).
  - Interpreter(s).
  - Compiler(s).
  - Debugger(s).
  - *(Module system.)*
- Different Prolog systems offer different facilities for these purposes.
The ISO Standard (Overview)

- Arithmetic
- Type checking and state checking
- Structure inspection
- Term comparison
- Input/Output
- Meta-calls and aggregation predicates
- Dynamic program modification
- Control structures (cut, true, fail, ...)
- Exception handling

Additionally (not in standard):
- Definite Clause Grammars (DCGs): parsing
Built-in Arithmetic

- Practicality: interface to the underlying CPU arithmetic capabilities.
- These arithmetic operations are not as general as their logical counterparts.
- Interface: evaluator of arithmetic terms.
- The type of arithmetic terms:
  - a number is an arithmetic term,
  - if \( f \) is an \( n \)-ary arithmetic functor and \( X_1, ..., X_n \) are arithmetic terms then \( f(X_1, ..., X_n) \) is an arithmetic term.
- Arithmetic functors: \(+\), \(-\), \(*\), \(/\) (float quotient), \(/\!/\) (integer quotient), \(\text{mod}\), and more.
- Examples:
  - \((3*X+Y)/Z\), correct if \( \text{when evaluated} \) \( X, Y \) and \( Z \) are arithmetic terms, otherwise it will raise an error.
  - \( a+3*X \) raises an error (because \( a \) is not an arithmetic term).
Built-in Arithmetic (Contd.)

- Built-in arithmetic predicates:
  - the usual \(<\), \(\geq\), \(\leq\), \(\leq\) (arithmetic equal), \(\neq\) (arithmetic not equal), ...
    Both arguments are evaluated and their results are compared
  - Z is X
    X (which must be an arithmetic term) is evaluated and result is unified with Z.

- Examples: let X and Y be bound to 3 and 4, respectively, and Z be a free variable:
  - Y \(<\) X+1, X is Y+1, X =:= Y. fail (the system will backtrack).
  - Y \(<\) a+1, X is Z+1, X =:= f(a). error (abort).
Arithmetic Programs

- \texttt{plus(X,Y,Z) :- Z is X + Y}
  - Only works in one direction (\(X\) and \(Y\) bound to arithmetic terms).
  - Meta-logical tests (see later) allow using it in both directions.
  - We have lost the recursive structure of the numbers.
  - But we have won (a lot) in performance!

- Factorial:

  Using Peano arithmetic:
  
  \begin{align*}
  \texttt{factorial(0,s(0)).} \\
  \texttt{factorial(s(N),F):-} \\
  \hspace{1em} & \texttt{factorial(N,F1),} \\
  \hspace{2em} & \texttt{times(s(N),F1,F).}
  \end{align*}

  Using Prolog arithmetic:
  
  \begin{align*}
  \texttt{factorial(0,1).} \\
  \texttt{factorial(N,F):-} \\
  \hspace{1em} & \texttt{N > 0,} \\
  \hspace{2em} & \texttt{N1 is N-1,} \\
  \hspace{3em} & \texttt{factorial(N1,F1),} \\
  \hspace{4em} & \texttt{F is F1*N.}
  \end{align*}

- Wrong goal order can raise an error (e.g., moving last call to \texttt{is/2} before call to \texttt{factorial}).
Type Checking Predicates

- Unary relations which *check* the type of a term:
  - integer(X)
  - float(X)
  - number(X)
  - atom(X)  (nonvariable term of arity 0 other than a number)
  - atomic(X)  atom or number
  - ...

- They behave as if defined by a (possibly infinite) table of facts (in part, see below).
- They either succeed or fail, but do not produce an error.
- Thus, they cannot be used to *generate* (e.g., if argument is a variable, they fail instead of instantiating it to possible values).
- This behaviour is outside first order logic because it allows checking the instantiation state of a variable.
Example: implementing a better behavior for `plus/3`:

```
plus(X,Y,Z):- number(X), number(Y), Z is X + Y.
plus(X,Y,Z):- number(X), number(Z), Y is Z - X.
plus(X,Y,Z):- number(Y), number(Z), X is Z - Y.
```

Then:

```
?- plus(3,Y,5).
Y = 2 ?
```

Still, it cannot be used to partition a number into two others:

```
?- plus(X,Y,5).
no
```

(in fact, this should raise an error, rather than simply failing).
Structure Inspection

- **functor**(X, F, A):
  - ◦ X is a compound term \( f(X_1, \ldots, X_n) \rightarrow F = f \quad A = n \)
  - ◦ F is the atom \( f \) and A is the integer \( n \rightarrow X = f(X_1, \ldots, X_n) \)
  - ◦ Error if \( X \), and either \( F \) or \( A \) are variables
  - ◦ Fails if the unification fails, \( A \) is not an integer, or \( F \) is not an atom

Examples:
- ◦ functors(t(b,a),F,A) \( \rightarrow F = t \), A=2.
- ◦ functor(Term,f,3) \( \rightarrow \) Term = \( f(_{-},_{-},_{-}) \).
- ◦ functor(Vector,v,100) \( \rightarrow Vector = v(_{-}, \ldots, _{-}) \).

(Note: in some systems functors arity is limited to 256)
Structure Inspection (Contd.)

- \text{arg}(N, X, \text{Arg}):
  - $N$ integer, $X$ compound term $\rightarrow \text{Arg}$ unified with $n$-th argument of $X$.
  - Allows accessing a structure argument in constant time and in a compact way.
  - Error if $N$ is not an integer, or if $X$ is a free variable.
  - Fails if the unification fails.

Examples:

?- \_T=date(9,February,1947), \text{arg}(3,\_T,X).
X = 1947
?- \_T=date(9,February,1947), \_T=date(_,_,X).
X = 1947
?- \text{functor}(\text{Array},\text{array},5),
\hspace{1em} \text{arg}(1,\text{Array},\text{black}),
\hspace{1em} \text{arg}(5,\text{Array},\text{white}).
\hspace{1em} \text{Array} = \text{array}(\text{black},_,_,_,\text{white}).

- What does \- \text{arg}(2, [a,b,c,d], X). return?
Example of Structure Inspection

- Define `subterm(Sub,Term)` (Term will always be a compound term):

```
subterm(Term,Term).
subterm(Sub,Term):-
    functor(Term,F,N),
    subterm(N,Sub,Term).

subterm(N,Sub,Term):-
    arg(N,Term,Arg), % also checks N > 0 (arg/1 fails otherwise!)
    subterm(Sub,Arg).
subterm(N,Sub,Term):-
    N>1,
    N1 is N-1,
    subterm(N1,Sub,Term).
```
Example of Structure Access

- Define `add_arrays(A1,A2,A3)`:
  
  ```prolog
  add_arrays(A1,A2,A3):- % Same N imposes equal length:
    functor(A1,array,N), functor(A2,array,N), functor(A3,array,N),
    add_elements(N,A1,A2,A3).

  add_elements(0,_A1,_A2,_A3).
  add_elements(I,A1,A2,A3):-
    I>0, arg(I,A1,X1), arg(I,A2,X2), arg(I,A3,X3),
    X3 is X1 + X2, I1 is I - 1,
    add_elements(I1,A1,A2,A3).
  ```

- Alternative, using lists instead of structures:
  
  ```prolog
  add_arrays_lists([],[],[]).
  add_arrays_lists([X|Xs],[Y|Ys],[Z|Zs]):-
    Z is X + Y,
    add_arrays_lists(Xs,Ys,Zs).
  ```

- In the latter case, where do we check that the three lists are of equal length?
Higher-Order Structure Inspection

- $T = \ldots L$ (known as “univ”)
  - $L$ is the decomposition of a term $T$ into a list comprising its principal functor followed by its arguments.
    
    ?- date(9,february,1947) =\ldots L.
    L = [date,9,february,1947].
    ?- _F = ’+’, X =\ldots [_F,a,b].
    X = a + b.

- Allows implementing higher-order primitives (see later).
  - Example: Extending derivative
    
    derivative(sin(X),X,cos(X)).
    derivative(cos(X),X,-sin(X)).
    derivative(FG_X, X, DF_G * DG_X):-
    FG_X =\ldots [_G_X],
    derivative(FG_X, G_X, DF_G), derivative(G_X, X, DG_X).

- But do not use unless strictly necessary: expensive in time and memory.
Conversion Between Strings and Atoms (New Atom Creation)

- Classical primitive: `name(A,S)`
  - A is the atom/number whose name is the list of ASCII characters S

```prolog
?- name(hello,S).
A = hello
?- name(A,"hello").
A = hello
```

- Ambiguity when converting strings which represent numbers.
  Example: `?- name('1',X), name(Y,X).`

- In the ISO standard fixed by dividing into two:
  * `atom_codes(Atom,String)`
  * `number_codes(Number,String)`
Meta-Logical Predicates

- var(X): succeed iff X is a free variable.
  
  \[- var(X), X = f(a). \text{ % Succeeds} \]
  \[- X = f(a), \text{ var}(X). \text{ % Fails} \]

- nonvar(X): succeed iff X is not a free variable.
  
  \[- X = f(Y), \text{ nonvar}(X). \text{ % Succeeds} \]

- ground(X): succeed iff X is fully instantiated.
  
  \[- X = f(Y), \text{ ground}(X). \text{ % Fails} \]

- Outside the scope of first order logic.

- Uses:
  
  ◦ control goal order,
  ◦ restore some flexibility to programs using certain builtins.
• Example:

```prolog
length(Xs,N):-
    var(Xs), integer(N), length_num(N,Xs).
length(Xs,N):-
    nonvar(Xs), length_list(Xs,N).

length_num(0,[]).
length_num(N,[__|Xs]):-
    N > 0, N1 is N - 1, length_num(N1,Xs).

length_list([],0).
length_list([X|Xs],N):-
    length_list(Xs,N1), N is N1 + 1.
```

• But note that it is not really needed: the normal definition of length is actually reversible! (although less efficient than length_num(N,L) when L is a variable).
Comparing Non-ground Terms

- Many applications need comparisons between non–ground/non–numeric terms.
- Identity tests:
  - $X == Y$ (identical)
  - $X \not== Y$ (not identical)

```{verbatim}
?- f(X) == f(X). %Succeeds
?- f(X) == f(Y). %Fails
```

- Term ordering:
  - $X @> Y$, $X @>= Y$, $X @< Y$, $X @=<= Y$ (alphabetic/lexicographic order)

```{verbatim}
?- f(a) @> f(b). %Fails
?- f(b) @> f(a). %Succeeds
?- f(X) @> f(Y). %Implementation dependent!
```
Comparing Non-ground Terms (Contd.)

- Reconsider `subterm/2` with non-ground terms

```
subterm(Sub,Term):- Sub == Term.
subterm(Sub,Term):- nonvar(Term),
                     functor(Term,F,N),
                     subterm(N,Sub,Term).
```

where `subterm/3` is identical to the previous definition

- Insert an item into an ordered list:

```
insert([], Item, [Item]).
insert([H|T], Item, [H|T]):- H == Item.
insert([H|T], Item, [Item, H|T]):- H @> Item.
insert([H|T], Item, [H|NewT]) :- H @< Item, insert(T, Item, NewT).
```

- Compare with the same program with the second clause defined as

```
insert([H|T], Item, [Item|T]) :- H = Item.
```
**Input/Output**

- A minimal set of input-output predicates ("DEC-10 Prolog I/O"):

<table>
<thead>
<tr>
<th>Class</th>
<th>Predicate</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O stream control</td>
<td>see(File)</td>
<td>File becomes the current input stream.</td>
</tr>
<tr>
<td></td>
<td>seeing(File)</td>
<td>The current input stream is File.</td>
</tr>
<tr>
<td></td>
<td>seen</td>
<td>Close the current input stream.</td>
</tr>
<tr>
<td></td>
<td>tell(File)</td>
<td>File becomes the current output stream.</td>
</tr>
<tr>
<td></td>
<td>telling(File)</td>
<td>The current output stream is File.</td>
</tr>
<tr>
<td></td>
<td>told</td>
<td>Close the current output stream.</td>
</tr>
<tr>
<td>Term I/O</td>
<td>write(X)</td>
<td>Write the term X on the current output stream.</td>
</tr>
<tr>
<td></td>
<td>nl</td>
<td>Start a new line on the current output stream.</td>
</tr>
<tr>
<td></td>
<td>read(X)</td>
<td>Read a term (finished by a full stop) from the current input stream and unify it with X.</td>
</tr>
<tr>
<td>Character I/O</td>
<td>put_code(N)</td>
<td>Write the ASCII character code N. N can be a string of length one.</td>
</tr>
<tr>
<td></td>
<td>get_code(N)</td>
<td>Read the next character code and unify its ASCII code with N.</td>
</tr>
</tbody>
</table>
Other stream-based input-output predicates:

<table>
<thead>
<tr>
<th>Class</th>
<th>Predicate</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O stream control</td>
<td>open(File,M,S)</td>
<td>Open ‘File’ with mode M and return in S the stream associated with the file. M may be read, write or append.</td>
</tr>
<tr>
<td></td>
<td>close(Stream)</td>
<td>Close the stream ‘Stream’.</td>
</tr>
<tr>
<td>Term I/O</td>
<td>write(S,X)</td>
<td>Write the term X on stream S.</td>
</tr>
<tr>
<td></td>
<td>nl(S)</td>
<td>Start a new line on stream S.</td>
</tr>
<tr>
<td></td>
<td>read(S,X)</td>
<td>Read a term (finished by a full stop) from the stream S and unify it with X.</td>
</tr>
<tr>
<td>Character I/O</td>
<td>put_code(S,N)</td>
<td>Write the ASCII character code N on stream S.</td>
</tr>
<tr>
<td></td>
<td>get_code(S,N)</td>
<td>Read from stream S the next character code and unify its ASCII code with N.</td>
</tr>
</tbody>
</table>
Example:

```
write_list_to_file(L,F) :-
telling(OldOutput), % Grab current output stream.
tell(F), write_list(L), told, % Write into F, close.
tell(OldOutput). % Reset previous output stream.
```

```
write_list([]).
write_list([X|Xs]) :- write(X), nl, write_list(Xs).
```

- More powerful and format-based input-output predicates are available (see, e.g., `format/2` and `format/3` – Prolog system manuals).
- All these input-output predicates are “side-effects”!
Meta–calls and Implementing Higher Order

- The meta-call `call(X)` converts a term `X` into a goal and calls it.
- When called, `X` must be instantiated to a term, otherwise an error is reported.
- Used for meta-programming, specially interpreters and shells. Also for defining negation (as we will see) and implementing higher order.

**Example:**

```
q(a).
p(X) :- call(X).
?- p(q(Y)).
Y = a
```

**Example:**

```
q(a,b).
apply(F,Args) :- G =.. [F|Args], call(G).
?- apply(q,[Y,Z]).
Y = a
Z = b
```
Meta–calls – Aggregation Predicates

• Other meta–calls are, e.g., `findall/3`, `bagof/3`, and `setof/3`.

• `findall(Term, Goal, ListResults)`: `ListResults` is the set of all instances of `Term` such that `Goal` is satisfied
  - If there are no instances of `Term` `ListResults` is `[]`
  - For termination, the number of solutions should be finite (and enumerable in finite time).

```prolog

?- findall(X, likes(X,Y), S).
S = [bill,dick,tom,tom,harry,jan] ?

?- findall(X, likes(X,water), S).
S = [] ?
```

Meta–calls – Aggregation Predicates (Contd.)

- \texttt{setof(Term, Goal, ListResults)}: ListResults is the ordered set (no duplicates) of all instances of Term such that Goal is satisfied
  - If there are no instances of Term the predicate fails
  - The set should be finite (and enumerable in finite time)
  - If there are un-instantiated variables in Goal which do not also appear in Term then a call to this built-in predicate may backtrack, generating alternative values for ListResults corresponding to different instantiations of the free variables of Goal
  - Variables in Goal will not be treated as free if they are explicitly bound within Goal by an existential quantifier as in \( Y^\ldots \)
  (then, they behave as in \texttt{findall/3})

- \texttt{bagof/3} same, but returns list unsorted and with duplicates (in backtracking order)
Meta-calls – Aggregation Predicates: Examples

?- setof(X, likes(X,Y), S).
S = [dick,harry,tom],
Y = beer ? ;
S = [bill,jan,tom],
Y = cider ? ;
no

likes(bill, cider).
likes(dick, beer).
likes(harry, beer).
likes(jan, cider).
likes(tom, beer).
likes(tom, cider).

?- setof((Y,S), setof(X, likes(X,Y), S), SS).
SS = [(beer,[dick,harry,tom]),
      (cider,[bill,jan,tom])] ? ;
no

?- setof(X, Y^(likes(X,Y)), S).
S = [bill,dick,harry,jan,tom] ? ;
no
Meta-calls – Negation as Failure

- Uses the meta-call facilities, the cut and a system predicate `fail` that fails when executed (similar to calling `a=b`).

  ```prolog
  not(Goal) :- call(Goal), !, fail.
  not(Goal).
  ```

- Available as the (prefix) predicate `\+/1`:

  ```prolog
  \+ member(c, [a,k,l])
  ```

- It will never instantiate variables.

- Termination of `not(Goal)` depends on termination of `Goal`. `not(Goal)` will terminate if a success node for `Goal` is found before an infinite branch.

- It is very useful but dangerous:

  ```prolog
  unmarried_student(X) :- not(married(X)), student(X).
  student(joe).
  married(john).
  ```

- Works properly for ground goals (programmer’s responsibility to ensure this).
Cut-Fail

- Cut-fail combinations allow forcing the failure of a predicate — somehow specifying a negative answer (useful but very dangerous!).

- Example – testing groundness: fail as soon as a free variable is found.

```prolog
ground(Term):- var(Term), !, fail.
ground(Term):-
    nonvar(Term),
    functor(Term,F,N),
    ground(N,Term).

ground(0,T). % All subterms traversed
ground(N,T):-
    N>0,
    arg(N,T,Arg),
    ground(Arg),
    N1 is N-1,
    ground(N1,T).
```
Repeat Loops

- `repeat` always succeeds: it has infinite answers.
- Used to implement loops: make use of backtracking to iterate by failing repeatedly.
- Example – reading loop:

```prolog
read_loop :-
  repeat,
  read(X),
  process(X),
  X == end_of_file, !.

process(end_of_file):- !.
process(X):- ... <deterministic computation> ...
```
Dynamic Program Modification (I)

- `assert/1`, `retract/1`, `abolish/1`, ...
- Very powerful: allows run-time modification of programs. Can also be used to simulate global variables.
- Sometimes this is very useful, but very often a mistake:
  - Code hard to read, hard to understand, hard to debug.
  - Typically, slow.
- Program modification has to be used scarcely, carefully, locally.
- Still, assertion and retraction can be logically justified in some cases:
  - Assertion of clauses which logically follow from the program. (*lemmas*)
  - Retraction of clauses which are logically redundant.
- Other typically non-harmful use: simple global switches.
- Behavior/requirements may differ between Prolog implementations. Typically, the predicate must be declared: `:- dynamic`. 
Dynamic Program Modification (II)

• Example program:

\[
\text{relate_numbers}(X, Y) :- \text{assert(related}(X, Y)).
\]
\[
\text{unrelate_numbers}(X, Y) :- \text{retract(related}(X, Y)).
\]

• Example query:

\[
?- \text{related}(1, 2).
\]
\{EXISTENCE ERROR: ...\}

?- \text{relate_numbers}(1, 2).

yes

?- \text{related}(1, 2).

yes

?- \text{unrelate_numbers}(1, 2).

yes

?- \text{related}(1, 2).

no

• Rules can be asserted dynamically as well.
Dynamic Program Modification (III)

- Example program:
  
  fib(0, 0).
  fib(1, 1).
  fib(N, F):-
      N > 1,
      N1 is N - 1,
      N2 is N1 - 1,
      fib(N1, F1),
      fib(N2, F2),
      F is F1 + F2.

  lfib(N, F):-
      lemma_fib(N, F), !.
  lfib(N, F):-
      N > 1,
      N1 is N - 1,
      N2 is N1 - 1,
      lfib(N1, F1),
      lfib(N2, F2),
      F is F1 + F2,
      assert(lemma_fib(N, F)).
  :- dynamic lemma_fib/2.
  lemma_fib(0, 0). lemma_fib(1, 1).

- Compare fib(24, N) versus lfib(24, N)
Meta-Interpreters

- **clause**(head, body):
  - Reads a clause head :- body from the program.
  - For facts body is true.

- To use clause/2 a predicate must be declared dynamic.

- Simple (“vanilla”) meta-interpreter:
  
  solve(true).
  solve((A,B)) :- solve(A), solve(B).
  solve(A) :- clause(A,B), solve(B).

- This code can be enhanced to do many tasks: tracing, debugging, explanations in expert systems, implementing other computation rules, ...

- Issues / interactions with module system.
Parsing (using append and traditional lists)

```prolog
%% ?- myphrase([t,h,e,’ ’,p,l,a,n,e,’ ’,f,l,i,e,s]).

myphrase(X) :-
    append(A,T1,X), article(A), append(SP,T2,T1), spaces(SP),
    append(N,T3,T2), noun(N), append(SPN,V,T3), spaces(SPN), verb(V).

article([a]).
article([t,h,e]).

spaces([' ']).
spaces([' ' | Y]) :- spaces(Y).
noun([c,a,r]).
noun([p,l,a,n,e]).

verb([f,l,i,e,s]).
verb([d,r,i,v,e,s]).
```
Parsing (using standard clauses and difference lists)

%% ?- myphrase([t,h,e,’ ’,p,l,a,n,e,’ ’,f,l,i,e,s],[]).

myphrase(X,CV) :-
    article(X,CA), spaces(CA,CS1), noun(CS1,CN),
    spaces(CN,CS2), verb(CS2,CV).

article([t,h,e|X],X).
article([a|X],X).

spaces([’ ’ | X],X).
spaces([’ ’ | Y],X) :- spaces(Y,X).

noun([p,l,a,n,e | X],X).
noun([c,a,r | X],X).

verb([f,l,i,e,s | X],X).
verb([d,r,i,v,e,s | X],X).
Parsing (same, using some string syntax)

%%% ?- myphrase("the plane flies",[]).

myphrase(X,CV) :-
    article(X,CA), spaces(CA,CS1), noun(CS1,CN),
    spaces(CN,CS2), verb(CS2,CV).

article( "the" || X, X).
article( "a" || X, X).

spaces( " " || X, X).
spaces( " " || Y, X) :- spaces(Y, X).

noun( "plane" || X, X).
noun( "car" || X, X).

verb( "flies" || X, X).
verb( "drives" || X, X).
Parsing (same, using additional syntax: DCGs)

- Add syntactic transformation to avoid writing all the auxiliary variables. The result is called**Definite Clause Grammars** (“DCGs”).

```
%% ?- myphrase("the plane flies",[]).
%% or, use ‘‘phrase/2’’ builtin:
%% ?- phrase(myphrase,"the plane flies").

:- use_package(dcg).

myphrase --> article, spaces, noun, spaces, verb.

article --> "the".
article --> "a".
noun --> "plane".
noun --> "car".
spaces --> " ".
spaces --> " ", spaces.
verb --> "flies".
verb --> "drives".
```
Parsing + actions (calling Prolog in DCGs)

- Other actions can be interspersed with the grammar. Raw Prolog can be called (between "{
  ...
}")

```prolog
%% ?- myphrase(NChars,"the plane flies",[]).
%% ?- phrase(myphrase(N),"the plane flies").
:- use_package(dcg).

myphrase(N) --> article(AC), spaces(S1), noun(NC), spaces(S2), verb(VC), \{ N is AC + S1 + NC + S2 + VC\}.

article(3) --> "the".
article(1) --> "a".
noun(5) --> "plane".
noun(3) --> "car".
verb(5) --> "flies".
verb(6) --> "drives".
```

---

38
Creating Executables

- Most systems have methods for creating 'executables':
  - Saved states (save/1, save_program/2, etc.).
  - Stand-alone compilers (e.g., ciaoC).
  - Scripts (e.g., prolog-shell).
  - "Run-time" systems.
  - etc.
Other issues in Prolog (see “The Art of Prolog” and Bibliography)

- Exception handling.
- Extending the syntax beyond operators: term expansions/macros.
- Delay declarations/concurrency.
- Operating system interface (and sockets, etc.).
- Foreign language (e.g., C) interfaces.
- Many other built-ins...
- ...


Some Typical Libraries in Prolog Systems

- Most systems have a good set of libraries.
- Worth checking before re-implementing existing functionality!
- Some examples:

<table>
<thead>
<tr>
<th>Arrays</th>
<th>Assoc</th>
<th>Attributes</th>
<th>Heaps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lists</td>
<td>Term Utilities</td>
<td>Ordset</td>
<td>Queues</td>
</tr>
<tr>
<td>Random</td>
<td>System Utilities</td>
<td>Tree</td>
<td>UGraphs</td>
</tr>
<tr>
<td>WGraphs</td>
<td>Sockets</td>
<td>Linda/Distribution</td>
<td>Persistent DB</td>
</tr>
<tr>
<td>CLPB</td>
<td>CLPQR</td>
<td>CLPFD</td>
<td>Objects</td>
</tr>
<tr>
<td>GCLA</td>
<td>TclTk</td>
<td>Tracing</td>
<td>Chars I/O</td>
</tr>
<tr>
<td>Runtime Utilities</td>
<td>Timeout</td>
<td>Xrefs</td>
<td>WWW</td>
</tr>
<tr>
<td>Java Interface</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Some Additional Libraries and Extensions (Ciao)

Other systems may offer additional extensions. Some examples from Ciao:

- Other execution rules:
  - Breadth-first execution
  - Iterative-deepening execution
  - Fuzzy Prolog, MYCIN rules, ...
  - Andorra (“determinate-first”) execution

- Interfaces to other languages and systems:
  - C, Java, ... interfaces
  - Persistent predicates and SQL database interface
  - Web/HTML/XML/CGI programming (PiLLoW) / HTTP connectivity
  - Interface to VRML (ProVRML)
  - Tcl/Tk interface
  - daVinci interface
  - Calling emacs from Prolog, etc.
Some Additional Libraries and Extensions (Ciao, Contd.)

- Numerous libraries as well as syntactic and semantic extensions:
  - Terms with named arguments - records/feature terms
  - Multiple argument indexing
  - Functional notation
  - Higher-order
  - The script interpreter
  - Active modules (high-level distributed execution)
  - Concurrency/multithreading
  - Object oriented programming
  - ...

Some Additional Libraries and Extensions (Ciao, Contd.)

- Constraint programming (CLP)
  - rationals, reals, finite domains, ...

- Assertions:
  - Regular types
  - Modes
  - Properties which are native to analyzers
  - Run-time checking of assertions

- Advanced programming support:
  - Compile-time type, mode, and property inference and checking, ... (CiaoPP).
  - Automatic documentation (LPdoc).
  - ...