Computational Logic

The (ISO-)Prolog Programming Language

(ISO-)Prolog

- A practical programming language based on the logic programming paradigm.
- Main differences with "pure" logic programming:
  - ◇ *depth-first* search rule (also, left-to-right control rule),
  - ◇ more control of the execution flow,
  - ◇ many useful pre-defined predicates (some not declarative, for efficiency),
  - ◇ higher-order and meta-logical capabilities, ...
- Advantages:
  - ◇ it can be *compiled* into fast and efficient code (including native code),
  - ◇ more expressive power,
  - ◇ industry standard (ISO-Prolog),
  - ◇ mature implementations with modules, graphical environments, interfaces, ...
- Drawbacks of "classical" systems (and how addressed by modern systems):
  - ◇ Depth-first search rule is efficient but can lead to *incompleteness*
    $\rightarrow$ alternative search strategies (e.g., Ciao's `bfall`, tabling, id, etc.).
  - ◇ No *occur check* in unification (which led to *unsoundness* in older systems)
    $\rightarrow$ support regular (i.e., infinite) trees: `X = f(X)` (already *constraint*-LP).

Programming Interface (Writing and Running Programs)

- **Not** specified in the ISO-Prolog language standard.

→ Is left to each particular system implementing the standard.

- This typically includes 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.

- **See the part on Developing Programs with a Logic Programming System for more details for the particular system used in the course (Ciao).**

The ISO Standard (Overview)

- Syntax (incl. operators) and operational semantics
- Arithmetic
- Checking basic types and state
- Structure inspection and term comparison
- Input/Output
- Pruning operators (cut)
- Meta-calls, higher-order, aggregation predicates
- Negation as failure, cut-fail
- Dynamic program modification
- Meta-interpreters
- Incomplete data structures
- Exception handling

Additionally (not in standard):

- Definite Clause Grammars (DCGs): parsing

Prolog Syntax and Terminology

- Variables, constants, functors, etc. as before.

  Note: in Prolog terminology constants are also referred to as "atoms."

- Some new data types and syntax:

  ◇ Numbers: `0`, `999`, `-77`, `5.23`, `0.23e-5`, `0.23E-5`.

    Infinite precision integers supported by many current systems (e.g., Ciao).

  ◇ Strings (of "codes"): `"Prolog" = [80,114,111,108,111,103]`
    (list of ASCII –also, e.g., utf8– character codes).

    * *Important note:* if we set `?- set_prolog_flag(write_strings, on).`
      lists character codes will be printed as strings: `" "`, very useful!

  ◇ Comments:

    * Using `%`: rest of line is a comment.
    * Using `/* ... */`: everything in between is a comment.

Prolog Syntax — Defining Operators

- Certain functors and predicate symbols are *predefined* as infix, prefix, or postfix *operators*, aside from the standard term notation, and *new ones can be added*.
- Very useful to make programs/data files more readable! → Language extensions.
- Stated using *operator declarations*:

  `:- op(` $< precedence >$`,` $< type >$`,` $< operator(s) >$`).`

  ◇ $< precedence >$: is an integer from 1 to 1200.

  General rule: the operator with the *highest* precedence number is the principal functor.

  E.g., if '+' has *higher* precedence number than '/', then

  `a+b/c` ≡ `a+(b/c)` ≡ `+(a,/(b,c))`.

  Alternatively, we can always use parenthesis: `/(+(a,b),c)` ≡ `(a+b)/c`

  (Note that in some other languages the ordering of precedence values is the opposite.)

  ◇ $< type >$:

  - infix: `xfx` (not associative), `xfy` (right associative), `yfx` (left associative).
  - prefix: `fx` (non-associative), `fy` (associative).
  - postfix: `xf` (non-associative), `yf` (associative).

  ◇ $< operator(s) >$: can be a single atom or a list of atoms.

Prolog Syntax — Operators (Contd.)

- Examples: run example ⟼

Standard Notation	Operator Notation
`+(a,/(b,c))`	`a+b/c`
`is(X, mod(34, 7))`	`X is 34 mod 7`
`<(+(3,4),8)`	`3+4 < 8`
`=(X,f(Y))`	`X = f(Y)`
`-(3)`	`-3`
`spy(/(foo,3))`	`spy foo/3`
`:-(p(X),q(Y))`	`p(X) :- q(Y)`
`:-(p(X),',' (q(Y),r(Z)))`	`p(X) :- q(Y),r(Z)`

- Note that, with this syntax convention, Prolog clauses *are also Prolog terms!*
- Parenthesis must always be used for operators with higher priority than 1000 (i.e., the priority of ','):

```
..., assert( (p :-q)), ...
```

- Operators are by default *local to modules* (explained later).

Prolog Syntax — Operators (Contd.)

- Typical standard operators:

```
:- op( 1200, xfx, [ :-, --> ]).
:- op( 1200,  fx, [ :-, ?- ]).
:- op( 1150,  fx, [ mode, public, dynamic,
                    multifile, block, meta_predicate,
                    parallel, sequential ]).
:- op( 1100, xfy, [ ; ]).
:- op( 1050, xfy, [ -> ]).
:- op( 1000, xfy, [ ',' ]).
:- op(  900,  fy, [ \+, spy, nospy ]).
:- op(  700, xfx, [ =, is, =.., ==, \==, @<, @>, @=<, @>=,
                                 =:=, =\=, <, >, =<, >= ]).
:- op(  550, xfy, [ : ]).
:- op(  500, yfx, [ +, -, #, /\, \/ ]).
:- op(  500,  fx, [ +, - ]).
:- op(  400, yfx, [ *, /, //, <<, >> ]).
:- op(  300, xfx, [ mod ]).
:- op(  200, xfy, [ ^ ]).
```

The Execution Mechanism of (classical) Prolog

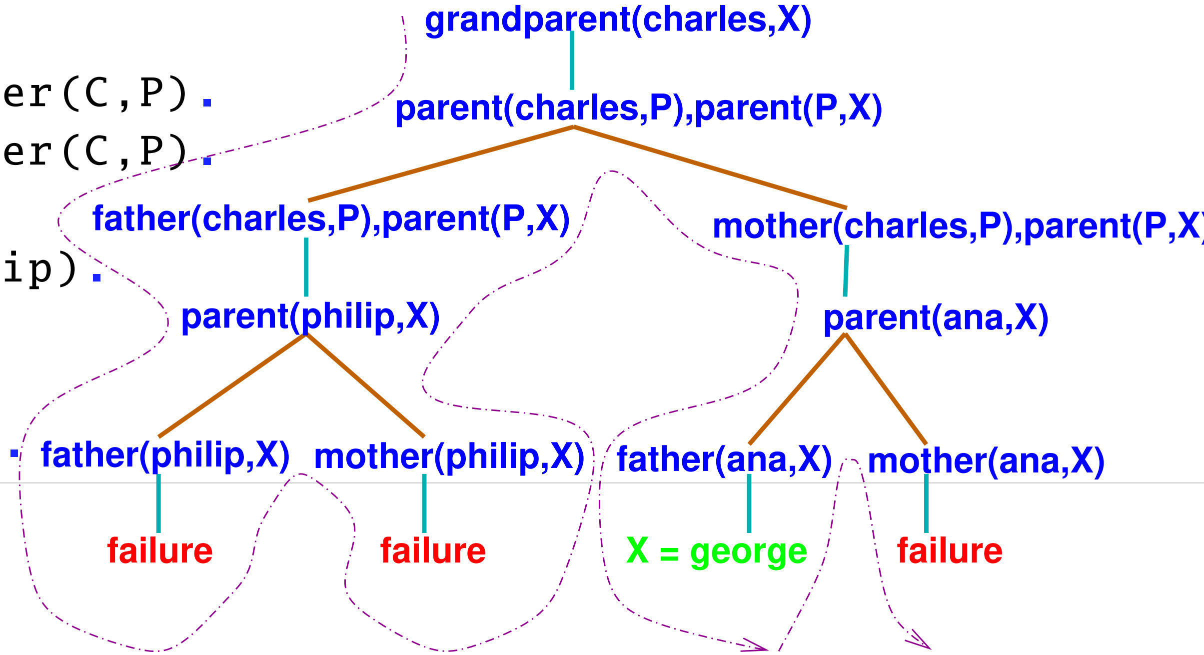
- Always execute calls in the body of clauses *left-to-right*.
- When entering a procedure, if several clauses unify (a *choice point*), take the *first unifying clause* (i.e., the leftmost unexplored branch).
- On failure, *backtrack* to the *next unexplored clause* of the *last choice point*.

```
grandparent(C,G) :- parent(C,P), parent(P,G).

parent(C,P) :- father(C,P).
parent(C,P) :- mother(C,P).

father(charles,philip).
father(ana,george).

mother(charles,ana).
```



- Check how Prolog explores this tree by running the **debugger**!

run example ⟼

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* (`arithexpr/1` – in next slide):
  - 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), `mod`, ... (see later).

  Examples:

  - `(3*X+Y)/Z`, correct if *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.) – The arithexpr type

```
arithexpr :=
     | num
     | + arithexpr                 | - arithexpr
     | ++ arithexpr                | -- arithexpr
     | aritexpr + arithexpr        | aritexpr - arithexpr
     | aritexpr * arithexpr        | aritexpr // arithexpr
     | aritexpr / arithexpr        | abs(arithexpr)
     | sign(arithexpr)             | float_integer_part(arithexpr)
     | float(arithexpr)            | float_fractional_part(arithexpr)
     | aritexpr ** arithexpr       | exp(arithexpr)
     | log(arithexpr)              | sqrt(arithexpr)
     | sin(arithexpr)              | cos(arithexpr)
     | atan(arithexpr)             | [arithexpr]
     |...
```

Other arithmetic operators that can appear in arithmetic expressions (see manuals):

- `rem, mod, gcd, >>, <<, /\, \/, \\, #, ...`
- `integer, truncate, floor, round, ceiling, ...`

Built-in Arithmetic (Contd.)

- Built-in arithmetic predicates:
  - ◇ `Z is X`
    `X` (which must be an arithmetic term) is *evaluated* and result is unified with `Z`.
  - ◇ the usual `<`, `>`, `=<`, `>=`, `=:=` (arithmetic equal), `=\=` (arithmetic not equal), ...
    Both arguments are evaluated (as in `is/2`) and their results are compared.
- *Examples:* run example ⟼
  - ◇ `X is 3+3//2.` — `X = 4`
  - ◇ `Z = 3//2, X is 3+Z.` — `X = 4`
- *Examples of failure and errors:*
  - ◇ `X=3, Y=4, Y<X+1.` — fail (the system will backtrack).
  - ◇ `X=3, Y=4, X is Y+1.` — fail (the system will backtrack).
  - ◇ `X=3, Y=4, X =:=Y.` — fail (the system will backtrack).
  - ◇ `Y=4, Y<a+1.` — throws error (the system will abort).
  - ◇ `X is Z+1.` — throws error (the system will abort).
  - ◇ `X=3, X =:=f(a).` — throws error (the system will abort).

Arithmetic Programs

- `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:

run example ⟼

Using Peano arithmetic:

```
factorial(0,s(0)).
factorial(s(N),F) :-
    factorial(N,F1),
    times(s(N),F1,F).
```

Using ISO-Prolog `is/2`, `>/2`:

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

- Wrong goal order can raise an error (e.g., moving last call to `is/2` before call to factorial).

Dynamic Checking of Basic Types

- 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.

Type Checking Predicates (Contd.)

- *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.
```

Now:

run example ⟼

```
?- 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, should raise an error, rather than simply failing (will see solutions later).

Type Checking Predicates (Contd.)

- *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.
```

  Now: run example ⟼

```
?- 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, should raise an error, rather than simply failing (will see solutions later).

→ **Real solution:** the addition of *constraints* to the language (CLP) –see later!

Structure Inspection

- `functor(X, F, A)`:
  - ◇ `X` is a term `f(X1,...,Xn)` $\rightarrow$ `F=f`   `A=`n
  - ◇ `F` is the atom `f` and `A` is the integer n $\rightarrow$ `X = f(X1,..,Xn)`
  - ◇ Some errors: if `X`, and either `F` or `A` are variables;
    or `A` is not an integer or less than 0; etc.
  - ◇ Fails if unification of an output with given value fails.

Examples:

run example $\longmapsto$

- ◇ `functor(t(b,a),F,A)` $\rightarrow$ `F=t, A=2`.
- ◇ `functor(Term,f,3)` $\rightarrow$ `Term =f(_,_,_)`.
- ◇ `functor(Vector,v,100)` $\rightarrow$ `Vector =v(_, ... ,_)`.

(Note: in some systems functor arity is limited to 256 by default;
there are libraries that allow unbounded arrays.)

Structure Inspection (Contd.)

- `arg(N, X, Arg)`:
  - ◇ `N` integer, `X` compound term → `Arg` unified with `N`-th argument of `X`.
  - ◇ Allows accessing a structure argument in constant time and in a compact way.
  - ◇ Error if `N` not integer, `N<0`, `X` not a compound, or `X` a free variable.
  - ◇ Fails for `N=0`, or `N>=`Arity, or if unification fails.

Examples:

```
?- _T=date(9,'February',1947), arg(3,_T,X).
X = 1947
?- _T=date(9,'February',1947), _T=date(_,_,X).
X = 1947
?- functor(Array,array,5),
      arg(1,Array,black),
      arg(5,Array,white).
Array = array(black,_,_,_,white).
```

- What does `?- arg(2,[a,b,c,d],X).` return? Explain why!

Example of Structure Inspection: Arrays

- Define `add_arrays(A,B,C)`: run example ⟼

```
add_arrays(A,B,C) :-    % Same N below imposes equal length:
     functor(A,array,N), functor(B,array,N), functor(C,array,N),
     add_elements(N,A,B,C).

add_elements(0,_,_,_).
add_elements(I,A,B,C) :-
     I>0, arg(I,A,AI), arg(I,B,BI), arg(I,C,CI),
     CI is AI + BI, I1 is I - 1,
     add_elements(I1,A,B,C).
```

- Alternative, using lists instead of structures:

```
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?

Example of Structure Inspection: Arrays (Contd.)

- Defining some syntactic sugar for `arg/3`:

run example ⟼

```
:- use_package(fsyntax). % Use functional notation.

:- op(250,xfx,@).    % Define @ as an infix operator
:- fun_eval '@'/2.   % Call @ when it appears as a term (no need for ~)

T@N := A :- arg(N,T,A).   % Define @ as simply calling arg/3
% Same as:   @(T,N) := A :- arg(N,T,A).
% and        @(T,N,A) :- arg(N,T,A).

:- fun_eval arith(true). % Evaluate arithmetic expressions (I-1)
```

- Now we can write `add_elements/4` as:

```
add_elements(0,_,_,_).
add_elements(I,A,B,C) :-
     I>0,
     C@I is A@I + B@I,
     add_elements(I-1,A,B,C).
```

Example of Structure Inspection: Subterms of a Term

- Define `subterm(Sub,Term)`:

```
subterm(Term,Term).   % a) A term is always a subterm of itself
subterm(Sub,Term):-   % b) The arguments are also subterms:
    nonvar(Term),     %    Check Term not a free variable
    functor(Term,_F,N),%   N is the number of arguments of Term
    n_to_one(N, J),   %    J is a natural between N and 1
    arg(J,Term,Arg),  %    Arg is the J-th argument of Term
    subterm(Sub,Arg).%    Sub are the subterms of Arg

n_to_one(N, N) :- N > 0.
n_to_one(N, X) :- N > 1, N1 is N-1, n_to_one(N1, X).
```

- Some queries:

run example ⟼

```
?- subterm( f(a) , g(b,f(a)) ).
?- subterm( f(b) , g(b,f(a)) ).
?- subterm( g(b,f(a)) , g(b,f(a)) ).
?- subterm( X , g(b,f(a)) ).
?- subterm( f(X) , g(b,f(a)) ).
?- subterm( X , g(X,f(a)) ).
?- subterm( f(X) , g(b,f(X)) ).
```

'Higher-Order' Structure Inspection

- `T =..L` (read as "univ") run example ⟼
  - ◇ `L` is the decomposition of a term `T` into a list comprising its principal functor followed by its arguments.

```
?- date(9,february,1947) =.. L.
L = [date,9,february,1947].
?- _F = '+', X =.. [_F,a,b].
X = a + b.
```

  - ◇ Allows *implementing* higher-order primitives (see later).

    Example – extending derivative: run example ⟼

```
deriv(sin(X),X,cos(X)).
deriv(cos(X),X,-sin(X)).
deriv(FG_X, X, DF_G * DG_X) :-
    FG_X =.. [_, G_X],
    deriv(FG_X, G_X, DF_G), deriv(G_X, X, DG_X).
```

```
?- deriv(sin(cos(x)),x,D).
D = cos(cos(x))* -sin(x) ?
```

  - ◇ But use *only when strictly necessary*: expensive in time and memory, HO.

Conversion Between Strings and Atoms (New Atom Creation)

- Classical primitive: `name(A,S)` run example ⟼
  - ◇ `A` is the atom or number whose name is the list of ASCII characters `S`

```
?- name(hello,S).
S = [104,101,108,108,111]
?- name(A,[104,101,108,108,111]).
A = hello
?- name(A,"hello").
A = hello
?- set_prolog_flag(write_strings,on).
?- name(hello,S).
S = "hello"
```

  - ◇ Ambiguity when converting strings which represent numbers.
    E.g.: `?- X='1', atom(X), name(X,S), name(Y,S), number(Y).` succeeds!
    (Note that we start with the *atom* `'1'` and we end up with the *number* `1`.)
  - ◇ 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).% Succeeds
?- X = f(a), var(X).% Fails
```

- `nonvar(X)` : succeed iff `X` is not a free variable.

```
?- X = f(Y), nonvar(X).%Succeeds
```

- `ground(X)` : succeed iff `X` is fully instantiated.

```
?- X = f(Y), ground(X).%Fails
```

- Outside the scope of first order logic.
- Uses:
  - ◇ control goal order,
  - ◇ restore some flexibility to programs using certain builtins.

Meta-Logical Predicates (Contd.) – choosing implementations

- Example: list length run example ⟼

```
len([],0).
len([_|T],N) :- len(T,TN), N is TN+1.
```

- Choosing between two implementations based on calling mode.
  I.e., implementing *reversibility* ”by hand.”

```
mylength(L,N) :- var(L), integer(N), create_list(N,L).
mylength(L,N) :- nonvar(L), compute_length(L,N).

create_list(0,[]).
create_list(N,[_|T]) :- N > 0, NT is N-1, create_list(NT,T).

compute_length([],0).
compute_length([_|T],N) :- compute_length(T,TN), N is TN+1.
```

- Not strictly needed: the normal definition of length is actually reversible!
- But note that `len/2` when called with `L` a variable and `N` a number is less efficient than `create_list/2` (since it is tail-recursive!).

Meta-Logical Predicates (Contd.) – choosing implementations

- Example (Contd.): Choosing between implementations based on calling mode.

  With more efficient version of `compute_length/2`
  (using an "accummulating parameter" – see slides on Prolog efficiency):

```
mylength(L,N) :- var(L), integer(N), create_list(N,L).
mylength(L,N) :- nonvar(L), compute_length(L,N).

create_list(0,[]).
create_list(N,[_|T]) :- N > 0, NT is N-1, create_list(NT,T).

compute_length(L,N) :- compute_length_(L,0,N).

compute_length_([],N,N).
compute_length_([_|T],A,N) :- NA is A+1, compute_length_(T,NA,N).
```

run example ⟼

Comparing Non-ground Terms

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

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

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

```
?- 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: run example ⟼

```
subterm_ng(Sub,Term) :- % a) A term is a subterm of another
    Sub == Term.        %    if they are identical.
subterm_ng(Sub,Term) :- % b) The arguments are also subterms:
    nonvar(Term),       %    Check Term not a free variable
    functor(Term,_F,N), %    N is number of arguments of Term
    n_to_one(N, J),     %    J is a natural between N and 1
    arg(J,Term,Arg),    %    Arg is the J-th argument of Term
    subterm_ng(Sub,Arg).%    Sub are the subterms of Arg
```

where `n_to_one/2` is identical to the previous definition.

- Insert an item into an ordered list: run example ⟼

```
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"): run example ⟼

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

Input/Output (Contd.)

- Other stream-based input-output predicates:

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

Input/Output (Contd.)

- Example:

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

Pruning Operator: Cut

- The "cut" (`!/0`) is a predicate which, *when executed*, commits Prolog to all the choices made since the current call to the predicate in which the cut is executed.
- Thus, when it is executed, a cut *prunes*:
  - ◇ all clauses below the clause in which the executed cut appears, and
  - ◇ all alternative solutions to the goals in the clause to the left of the executed cut.

  It does not affect the search in the goals to the right of the cut.

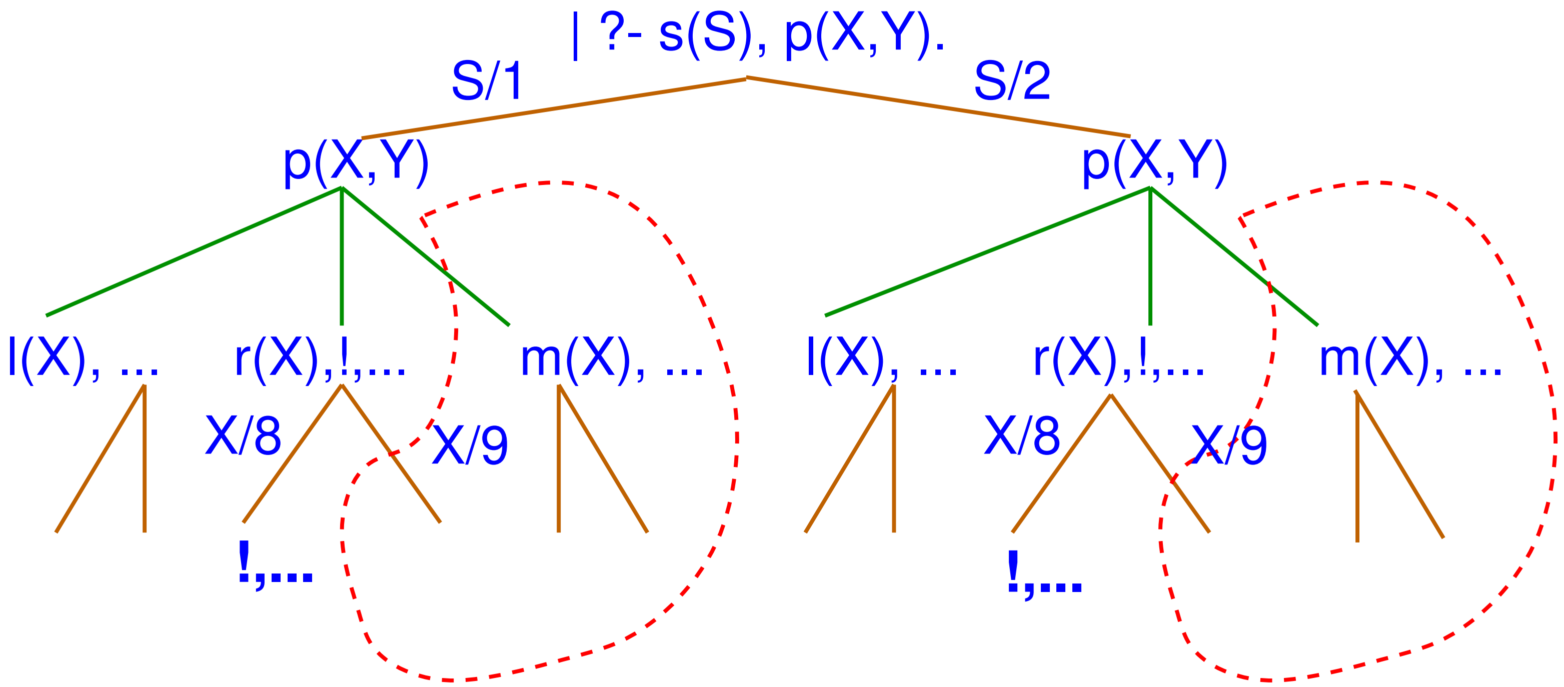
  Example: run example ⟼

```
s(1).          p(X,Y):- l(X), ...           r(8).
s(2).          p(X,Y):- r(X), !, ...        r(9).
               p(X,Y):- m(X), ...
```

  with query `?- s(A),p(X,Y).`
  If execution reaches the cut (`!`):
  - ◇ The second alternative of `r/1` is not considered.
  - ◇ The third clause of `p/2` is not considered.

Pruning Operator: Cut (Contd.)

```
s(1).          p(X,Y):- l(X), ...          r(8).
s(2).          p(X,Y):- r(X), !, ...       r(9).
               p(X,Y):- m(X), ...
```



Types of Cut: White and Green Cuts

run example ⟼

- *White* cuts: do not discard solutions.

```
max(X,Y,X):- X > Y, !.
max(X,Y,Y):- X =< Y.
```

They affect neither completeness nor correctness – use them freely.
(In many cases the system “introduces” them automatically.)

- *Green* cuts: discard correct solutions which are not needed.

```
address(X,Add):- home_address(X,Add), !.
address(X,Add):- business_address(X,Add).
```

```
membercheck(X,[X|Xs]):- !.
membercheck(X,[Y|Xs]):- membercheck(X,Xs).
```

They affect completeness but not correctness.
Necessary in many situations (but beware!).

Types of Cut: Red Cuts

run example ⟼

- *Red* cuts: discard solutions which are not correct according to the intended meaning.
  - ◇ Example:

```
max( X,Y,X):- X > Y,!.
max(_X,Y,Y).
```

wrong answers to, e.g., `?- max(5, 2, 2).` ⤳ `yes`

  - ◇ Example:

```
days_in_year( X,366):- leap_year(X),!.
days_in_year(_X,365).
```

```
leap_year(X):- number(X), 0 is X mod 4.
```

wrong answers to, e.g., `?- days_in_year(4,365).` ⤳ `yes`

`?- days_in_year(a, D).` ⤳ `D=365`

Red cuts affect completeness and one can no longer rely on the declarative meaning of the program to reason about correctness – avoid when possible.

Types of Cut: Red Cuts (Contd.)

- Useful tips regarding red cuts:
  - Can be improved by delaying output bindings to after the cut:

```
max( X,Y,M) :- X > Y, !, M = X.
max(_X,Y,Y).
```

    Now also *correct* answer to `?- max(5, 2, 2).` ⇝ `no`

  - Using *if-then-else* (syntactic sugar over cut) can often be a better solution:

```
max(X,Y,M) :- ( X>Y -> M=X ; M=Y).
```

    Also *correct* answer to `?- max(5, 2, 2).` ⇝ `no`

Types of Cut: Red Cuts (Contd.)

- Useful tips regarding red cuts (leap year example): run example ⟼
  - The best solution: making the cut white by defining the other condition.

```
days_in_year_good(Y,D) :- leap_year(Y),  !, D = 366.
days_in_year_good(Y,D) :- standard_year(Y), D = 365.
standard_year(Y) :- number(Y), R is Y mod 4, R \= 0.
```

  - Delaying output bindings to after the cut:

```
days_in_year_delay_output(Y,D) :- leap_year(Y), !, D = 366.
days_in_year_delay_output(_Y,365).
```

    Note that `?- days_in_year_delay_output(Y,366).` still produces a wrong answer! This is because we are probably thinking of a 'mode': we provide the year and ask for the number of days.

  - Improvement: we check the 'mode':

```
days_in_year_moded(Y,_D) :- var(Y), !,
     write('ERROR: year unbound.\n'), abort.
days_in_year_moded(Y, D) :-  leap_year(Y), !, D = 366.
days_in_year_moded(_Y,365).
```

  - Even better: do it with an assertion!

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, such as interpreters and shells.
  Also for defining negation (as we will see) and *implementing* higher order.
- Example:

```
q(a,1).      q(b,2).      p(X) :- call(X).
```

```
?- G=q(a,Y), p(G).  ⇝  Y = 1
```

- Example (implementing higher-order): run example ⟼

```
apply(F,Args) :- G =.. [F|Args], call(G).
```

```
?- P = q, apply(P,[a,Y]).  ⇝  Y = 1
```

- Example (maptuples/2):

```
maptuples(_P,[]).
maptuples(P,[Tuple|RTuples]) :-
    apply(P,Tuple),
    maptuples(P,RTuples).
```

```
?- maptuples(q,[[a,X],[b,Y]]).  ⇝  X=1, Y=2
```

Further Higher Order Facilities

- There is also a family of predicates `call/1`, `call/2`, ..., `call/n`.
  - ◇ The first argument can be a term including some of the arguments; the rest of the arguments of the call should be in the additional arguments of `call/n`.
- Examples: run example ⟼

```
t(a,1).   t(b,2).
?- P = t, call(P,a,Y).  ⤳   Y = 1
?- P = t(a), call(P,Y). ⤳   Y = 1
```

```
?- P = append, call(P,[1,2],[3.4],Y). ⤳   Y = [1,2,3,4]
?- P = append(X,Y), call(P, [1,2,3,4]).
⤳  X=[], Y=[1,2,3,4]; X=[1], Y=[2,3,4]; ...
```

- Also, higher-order library:
- Examples:

```
?- use_module(library(hiordlib)).
?- maplist(plus2, [1,2,3], [4,5,6], R). ⤳ R  = [5,7,9]
?- maplist(plus2, [1,2,3], L2, [5,7,9]). ⤳ L2 = [4,5,6]
?- maplist(append, [ [a],    X,      X ],
                   [ [c,d], [e,f], Y ],
                   [ X,      Y,      Z] ).
⤳ X = [a,c,d], Y = [a,c,d,e,f], Z = [a,c,d,a,c,d,e,f]
```

Further Higher Order Facilities (Ciao)

- In Ciao the `hiord` package allows writing `G(Y,Z)` and provides much additional functionality, such as *anonymous predicates*. See also the `hiordlib` library.
- Examples:

run example ⟼

```
t(a,1).
t(a,2).
```

```
?- use_package(hiord). % (Loads hiord package in top level)
?- P = t, P(X,Y).    ⇝    X=a, Y=1;    X=b, Y=2
?- P = append, P([1,2],[3.4],Y).    ⇝    Y=[1,2,3,4]
?- P = append, P(X,Y,[1,2,3,4]).
   ⇝   X=[], Y=[1,2,3,4]; X=[1], Y=[2,3,4]; ...
```

Anonymous predicates (read _ as lambda):

```
?- P = { _(X,Y) :- Y is X+1 }, P(3,R).
   ⇝   R=4
```

Meta-calls – Aggregation Predicates

run example ⟼

- 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).

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

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

Meta-calls – Aggregation Predicates (Contd.)

- `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^...` (then, they behave as in `findall/3`)
- `bagof/3`:
  same, but returns list unsorted and with duplicates (in backtracking order).

Meta-calls – Aggregation Predicates: Examples

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

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

?- 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`). run example ⟼

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

- Available as the (prefix) predicate `\+/1`: `\+ member(c, [a,k,l])`
- It will never instantiate variables.
  - Using `\+` twice useful to *test* without binding variables. E.g., `\+ \+ X = 1`, checks if `X` is bound (or can be bound) to 1, without binding `X` if is free.
- 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:

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

`?- unmarried_student(X).` → `no`

- Does work correctly for **ground goals** (programmer's responsibility to ensure it).

Meta-calls – Negation as Failure – Ensuring Correctness

- We can check that negation is called with a ground term:

```
not(G) :-
    ground(G), !,
    \+ G.
not(G) :-
    write('ERROR: Non-ground goal in negation: '), write(G), nl,
    abort.
```

- Or using assertions:

```
:- pred not(G) : ground(G).
not(G) :- \+ G.
```

I.e., we declare that `G` must be ground when called (`:` field).
This will be checked, e.g., dynamically if we turn on run-time checking:

```
:- use_package([rtchecks]).
```

- Another example of meta-call + cut: getting only one solution:

```
once(G) :- call(G), !.
```

E.g., `?- once(member(X,[1,2,3])).`

Cut-Fail

- Cut-fail combinations allow forcing the failure of a predicate — somehow specifying a negative answer (useful but dangerous!).
- Example: testing groundness; fail as soon as a free variable is found.

run example ⟼

```
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).
```

Dynamic Program Modification (I)

- `assert/1`, `retract/1`, `abolish/1`, ...
- Very powerful: allow 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, new predicates (not already defined in the program) can always be asserted; else they must be declared `:- dynamic`.

Dynamic Program Modification (II)

- Example program:

run example ⟼

```
:- dynamic related/2.
related(3,4).

relate_numbers(X, Y):- assert(related(X, Y)).
unrelate_numbers(X, Y):- retract(related(X, Y)).
```

- Sample queries:

```
?- related(3, 4).             % -> yes
?- related(1, 2).             % -> no
?- relate_numbers(1, 2).      % -> yes
?- related(1, 2).             % -> yes
?- unrelate_numbers(1, 2).    % -> yes
?- related(1, 2).             % -> no
?- abolish(related/2).        % -> yes
?- related(1, 2).             % -> error (does not exist)
```

- *Rules* can be asserted dynamically as well.

Dynamic Program Modification (III)

- Example: computing the Fibonacci sequence run example $\longmapsto$

  $fib(0) = 0, fib(1) = 1, fib(n) = fib(n-1) + fib(n-2)$ i.e., $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$

1a) Direct definition:

```
fib(0, 0).
fib(1, 1).
fib(N, F):-
    N>1,
    N1 is N-1, fib(N1, F1),
    N2 is N-2, fib(N2, F2),
    F is F1 + F2.
```

1b) Direct definition in functional syntax:

```
fib(0) := 0.
fib(1) := 1.
fib(N) := fib(N-1)+fib(N-2)
        :- N>1.
```

2) Version saving 'lemmas' (things already proved):

```
% 'memoing' table, preloaded w/N=0, N=1:
:- dynamic lemma_fib/2.
lemma_fib(0, 0).
lemma_fib(1, 1).

% First check if Nth element in table:
lfib(N, F):-  lemma_fib(N, F), !.
% Else compute it and record the result:
lfib(N, F):-
    N > 1, N1 is N - 1, lfib(N1, F1),
           N2 is N - 2, lfib(N2, F2),
    F is F1 + F2,
    assert(lemma_fib(N, F)).
```

- Compare `?- fib(31,N)` versus `?- lfib(31,N)`.
  (adjust the number depending on CPU speed).

"Those who cannot remember the past are condemned to repeat it"

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: run example ⟼

```
solve(true).
solve((A,B)) :-  solve(A), solve(B).
solve(A) :-  clause(A,B), solve(B).
```

Some sample queries:

```
?- solve(lappend([1,2],[3,4],L)).
?- solve(lappend(X,Y,[1,2,3,4])).
```

- This code also implements backtracking! Note that `clause/2` introduces choice-points since `A` can unify with several clause heads.
- Interactions with module system: remember that clauses must be dynamic (and use the `dynamic_clauses` package).

Meta-Interpreters: extending the basic meta-interpreter

- The basic meta-interpreter code:

```
solve(true).
solve((A,B)) :-  solve(A), solve(B).
solve(A) :-  clause(A,B), solve(B).
```

  can be easily extended to do many tasks: tracing, debugging, explanations in expert systems, implementing other computation rules, ...

- E.g., an interpreter that counts the number of (forward) steps:

```
csolve( true,   0).
csolve( (A,B), N) :- csolve(A,NA), csolve(B,NB), N is NA+NB.
csolve( A,      N) :- clause(A,B),  csolve(B,N1), N is N1+1.

?- csolve(lappend([1,2],[3,4],L),N).
```

Incomplete Data Structures - Example: Difference Lists

- `[1,2,3]`: a closed list; cannot easily add at the end. run example ⟼
- Objective: have direct access to list tail to append in constant time.
- A pseudo-type:

```
dlist(X-X) :- var(X).
dlist([_|DL]-Y) :- dlist(DL-Y).
```

  (Note: just for "minimal" difference lists, and not declarative because of `==/2`)
- Allows us to *keep a pointer to the end of the list* → *append in constant time*:

```
append_dl(B1-E1,B2-E2,B3-E3) :- B3=B1, E3=E2, B2=E1.
```

  Or, more compactly:

```
append_dl(X-Y,Y-Z,X-Z).
```

  And, actually, no call to `append_dl` is normally necessary!

... But can only be done once (see later).

One can also build difference (open ended) trees, dictionaries, queues, etc., by leaving variables at the ends (e.g., at the leaves for trees).

Playing with difference lists

- Create two difference lists (`L1` and `L2`) and append them (`L2=X`) "by hand":

```
?-  L1 = [1,2,3|X], L2 = [4,5|Y], L2=X.
L1 = [1,2,3,4,5|Y],
L2 = [4,5|Y],
X = [4,5|Y] ?
yes
```

`L1` contains the resulting difference list `[1,2,3,4,5|Y]`.

- Given: `append_dl(B1-E1,E1-E2,B1-E2)`

```
?- append_dl([1,2,3|X]-X,[4,5|Y]-Y,L).
L = [1,2,3,4,5|Y]-Y, X = [4,5|Y] ?
```

`L` has the resulting (appended) difference list.
But note that we have modified the first list: we cannot append to it again.

```
?- append_dl(L-X,[4,5|Y]-Y,[1,2,3,4,5|Z]-Z).
L = [1,2,3,4,5|Y], X = [4,5|Y], Z = Y ?
```

Quick-sort using standard lists and append

```
qsort([],[]).
qsort([X|L],S) :-            % Take first element of list in X
   partition(L,X,LS,LB),  % LS elements of L < X, LB els >= X
   qsort(LS,LSS),           % LSS is LS sorted
   qsort(LB,LBS),           % LBS is LB sorted
   append(LSS,[X|LBS],S). % We append the small ones sorted to
                            % the big ones sorted, w/X in front
partition([],_P,[],[]).
partition([E|R],P,[E|Smalls],Bigs) :- % Take first element E
    E < P,                  % If E < P add to list of smaller ones
    partition(R,P,Smalls,Bigs).
partition([E|R],P,Smalls,[E|Bigs]) :-
    E >= P,                 % If E >= P add to list of larger ones
    partition(R,P,Smalls,Bigs).
```

run example ⟼

Quick-sort using standard lists and append

```
?- qsort([5,2,1,3,7,6], SL).

qsort([],[]).
qsort([X|L],S) :-                % X=5, L=[2,1,3,7,6]
   partition(L,X,LS,LB),         % LS=[2,1,3], LB=[7,6]
   qsort(LS,LSS),                % LSS=[1,2,3]
   qsort(LB,LBS),                % LBS=[6,7]
   append(LSS,[X|LBS],S).        % call: append([1,2,3],[5,6,7],S)
                                 % S=[1,2,3,5,6,7]
partition([],_P,[],[]).
partition([E|R],P,[E|Smalls],Bigs) :-
    E < P,
    partition(R,P,Smalls,Bigs).
partition([E|R],P,Smalls,[E|Bigs]) :-
    E >= P,
    partition(R,P,Smalls,Bigs).
```

Quick-sort using difference Lists (no append!)

- First list `L` is a normal list, second is built as a difference list (`SL-SLE`).
- Version 1: using `-/2` functor and explicit unifications.

run example ⟼

```
                              % ?- qsort_dl([5,2,1,3,7,6], SL).

qsort_dl(L,SL) :-             % L   = [5,2,1,3,7,6]
    qsort_dl_(L,SL-SLE),      % SL  = [1,2,3,5,6,7|SLE]
    SLE = [].                 % SL  = [1,2,3,5,6,7]

qsort_dl_([],SLE-SLE).
qsort_dl_([X|L],SL-SLE) :-   % X = 5, L = [2,1,3,7,6]
    partition(L,X,S,B),       % S = [2,1,3], B = [7,6]
    qsort_dl_(S,SS-SSE),      % SS  = [1,2,3|SSE]
    qsort_dl_(B,BS-BSE),      % BS  = [6,7|BSE]
    SSE = [X|BS],             % SSE = [5,6,7|BSE]
    SL = SS,                  % SL  = [1,2,3,5,6,7|BSE]
    SLE = BSE.                % SL  = [1,2,3,5,6,7|SLE]
% Partition is the same as before.
```

Quick-sort using difference Lists (no append!)

- Version 2: still using `-/2` functor,
- but unifications made in-place (no calls to `=/2`).

```
qsort_dl(L,SL) :-
    qsort_dl_(L,SL-[]).

qsort_dl_([],SLE-SLE).
qsort_dl_([X|L],SL-SLE) :-
    partition(L,X,S,B),
    qsort_dl_(S,SL-[X|BS]),
    qsort_dl_(B,BS-SLE).

% Partition is the same as before.
```

Quick-sort using difference Lists (no append!)

- Version 3: using an extra argument (instead of the `-/2` functor),
- and in-place unifications (no calls to `=/2`).

```
qsort_dl(L,SL) :-
    qsort_dl_(L,SL,[]).

qsort_dl_([],SLE,SLE).
qsort_dl_([X|L],SL,SLE) :-
    partition(L,X,S,B),
    qsort_dl_(S,SL,[X|BS]),
    qsort_dl_(B,BS,SLE).

% Partition is the same as before.
```

Parsing (using append and traditional lists)

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

run example ⟼

```
myphrase(P) :-
    % A (article) must be at beginning of P, T1 is the rest:
    append(A,T1,X),   article(A),
    append(S1,T2,T1), spaces(S1), % Spaces must follow
    append(N,T3,T2),  noun(N),    % A noun must follow
    append(S2,V,T3),  spaces(S2), % Spaces must follow
                      verb(V).    % Must end with a verb

article([a]).                spaces([' ']).
article([t,h,e]).            spaces([' ' | Y]) :- spaces(Y).

noun([c,a,r]).               verb([f,l,i,e,s]).
noun([p,l,a,n,e]).           verb([d,r,i,v,e,s]).
```

Parsing (using difference lists)

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

run example ⟼

```
myphrase(P,C) :-
% Article is beginning of phrase, rest is beginning of spaces
    article(P,CA), spaces(CA,CS1),
    noun(CS1,CN),  spaces(CN,CS2),
    verb(CS2,C).




article([a      | T],T).   spaces([' ' | T],T).
article([t,h,e  | T],T).   spaces([' ' | Y],T) :- spaces(Y,T).

noun([c,a,r     | T],T).   verb([f,l,i,e,s | T],T).
noun([p,l,a,n,e | T],T).   verb([d,r,i,v,e,s | T],T).
```

Parsing (same, using string syntax)

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

run example ⟼

```
myphrase(X,C) :-
    % This clause does not change
    article(X,CA), spaces(CA,CS1),
    noun(CS1,CN),  spaces(CN,CS2),
    verb(CS2,C).


% Here we use strings "..."
article("a"     || T, T).   spaces(" " || T, T).
article("the"   || T, T).   spaces(" " || Y, T) :- spaces(Y, T).

noun( "car"     || T, T).   verb( "flies" || T, T).
noun( "plane"   || T, T).   verb( "drives"|| T, T).
```

Parsing (same, using string syntax)

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

run example ⟼

```
myphrase(X,C) :-
    % This clause does not change
    article(X,CA), spaces(CA,CS1),
    noun(CS1,CN),  spaces(CN,CS2),
    verb(CS2,C).



% Here we use strings "..."
article("a"     || T, T).  spaces(" " || T, T).
article("the"   || T, T).  spaces(" " || Y, T) :- spaces(Y, T).

noun( "car"     || T, T).  verb( "flies" || T, T).
noun( "plane"   || T, T).  verb( "drives"|| T, T).
```

- Reminder: strings `"..."` are lists of characters.
  I.e., `X = "the"` means `X = [0't,0'h,0'e]`, i.e., `X = [116,104,101]`
- `"..."|| T` is for strings like `[a,b,c|T]` for lists, i.e., `T` is the tail of the list.

Parsing (same, using additional syntax: DCGs)

- Add syntactic transformation to avoid writing all the auxiliary variables.
  The result is called a **Definite Clause Grammar** ("DCG"). run example ⟼

```
:- use_package(dcg).

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




article --> "a".                spaces --> " ".
article --> "the".              spaces --> " ", spaces.

noun --> "car".                 verb --> "flies".
noun --> "plane".               verb --> "drives".
```

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

or, use the "phrase/2" builtin:

```
?- phrase(myphrase,"the plane flies").
```

Parsing (same, using additional syntax: DCGs)

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

```
:- use_package(dcg).

myphrase --> article, spaces, noun, spaces, verb.
% Translates to: myphrase(X,C) :- article(X,CA),
% spaces(CA,CS1), noun(CS1,CN), spaces(CN,CS2), verb(CS2,C).

%        ↙translates to: article( "a" || T, T).
article --> "a".                    spaces --> " ".
article --> "the".                  spaces --> " ", spaces.

noun --> "car".                     verb --> "flies".
noun --> "plane".                   verb --> "drives".
```

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

or, use the “phrase/2” builtin:

```
?- phrase(myphrase,"the plane flies").
```

Parsing + actions (calling plain Prolog from DCGs)

- Other actions can be interspersed with the grammar.
  Plain Prolog can be called between "{ ... }"
- Additional arguments can be added; they go before the implicit ones.

Example: we add an additional argument to count number of characters parsed:

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

```
?- phrase(myphrase(N),"the plane flies").
```

run example ⟼

```
:- use_package(dcg).

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

article(1) --> "a".          spaces(1) --> " ".
article(3) --> "the".        spaces(N) --> " ", spaces(N1),
                                               {N is N1+1}.
noun(3) --> "car".           verb(5) --> "flies".
noun(5) --> "plane".         verb(6) --> "drives".
```

Other issues in Prolog (see "The Art of Prolog" and Bibliography)

- Repeat loops ("failure-driven" loops):

```
main(_) :- repeat, read(X), process(X).

process(end).
process(X) :- display(X), nl, fail.
```

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

Arrays	Assoc	Attributes	Heaps
Lists	Term Utilities	Ordset	Queues
Random	System Utilities	Tree	UGraphs
WGraphs	Sockets	Linda/Distribution	Persistent DB
CLPB	CLPQR	CLPFD	Objects
GCLA	TclTk	Tracing	Chars I/O
Runtime Utilities	Timeout	Xrefs	WWW
Java Interface	...	...	...

Some Additional Libraries and Extensions (Ciao)

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

- Other execution rules:
  - ◇ Breadth-first, Iterative-deepening, Random, ...
  - ◇ Tabling
  - ◇ CASP (negation with multiple models)
  - ◇ Andorra (“determinate-first”) execution, fuzzy Prolog, ...
- Interfaces to other languages and systems:
  - ◇ Interfaces to C, Java, JavaScript, Python, LLVM, ...
  - ◇ SQL database interface and persistent predicates
  - ◇ Web/HTML/XML/CGI programming (PiLLoW) / HTTP connectivity / JSON / compilation to JavaScript ...
  - ◇ Interfaces to solvers (PPL, Mathematica, MiniSAT, Z3, Yikes, ...)
  - ◇ Graphviz, daVinci interfaces
  - ◇ Interfaces to Electron, wxWidgets, Tcl/Tk, VRML (ProVRML), ...
  - ◇ Calling Emacs from Prolog, etc.

Some Additional Libraries and Extensions (Ciao, Contd.)

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

Some Additional Libraries and Extensions (Ciao, Contd.)

- Constraint programming (CLP)
  - rationals, reals, finite domains, ...
  - CHR (constraint handling rules), GeCode, ...
- Assertions:
  - Regular types, Modes, Determinacy, etc.
  - Other properties
  - Run-time checking of assertions
  - Assertion-based unit tests and automatic test case generation
  - Compile-time property inference and assertion checking (CiaoPP).
- Additional programming support:
  - Automatic documentation (LPdoc).
  - Partial evaluation, optimization, parallelization (CiaoPP).
  - ...