CS 352 Prolog Supplement

Prolog = Programming in Logic.

References

W F Clocksin & C S Mellish, *Programming in Prolog*. Fourth Edition. New York: Springer-Verlag, 1994.

http://www.intranet.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html

---

Prolog on the Intranet

Check the manual page …

```
intranet (2) % man pl
Reformatting page.  Wait... done

Misc. Reference Manual Pages                                    pl(1L)

NAME
     pl - SWI-Prolog 3.1.0

SYNOPSIS
     pl [-help|-v|-arch
     pl [options]
     pl [options] [-o output] -c file ...
     pl [options] [-o output] -b initfile ...

DESCRIPTION
     SWI-Prolog is an implementation of Prolog in  the  Edinburgh
     tradition.   It  is  based  on  a restricted form of the WAM
     (RISC-WAM?).  It has a good collection  of  built-in  predi-
     cates, a large set of which it shares with C-Prolog, Quintus
     Prolog and SICStus Prolog.  It has  a  fairly  good  perfor-
     mance,  with  a  fast  compiler.  It includes a Quintus-like
     module system, a library autoload facility, a  garbage  col-
     lector,  on-line help, a trasparent and fast C interface (in
     both directions), and a profiler.
```

Entering prolog…

```
intranet (1) % pl
Welcome to SWI-Prolog (Version 3.1.0)
Copyright (c) 1993-1998 University of Amsterdam.  All rights reserved.

For help, use ?- help(Topic). or ?- apropos(Word).

?-
```

Exiting prolog…

```
?- halt.
```

The "?-" means that Prolog is ready to answer a question.

```
?- 3=3.
yes
?- 3=2.
no
?- a=a.
yes
?- a=b.
no
```

We can also tell facts to prolog:

factset1

```
?- consult(user).
happy(john).
happy(mary).
sad(bill).
sad(barbara).
sad(john).
likes(john,mary).
likes(bill,barbara).
likes(bill,mary).
likes(mary,flowers).
likes(john,corvettes).
likes(john,john).
^d                          this is the end-of-file character
```

Or, we could've put these into a file, say FOO.BAR,

and then said

```
?- consult('foo.bar').
```

or

```
?- ['foo.bar'].
```

Now we can ask questions about the facts.

```
?- happy(john).
yes
?- sad(john).
yes
?- happy(bill).
no
```

this "no" means "I can't prove it"

```
?- likes(bill,mary).
yes
?- likes(bill,john).
no

?- lonely(bill).
[WARNING: Undefined predicate: `lonely/1']

No
?-
```

We can list all facts which use a given predicate.

```
?- listing(happy).

happy(john).
happy(mary).

Yes
?- listing(sad).

sad(bill).
sad(barbara).
sad(john).

Yes
?- listing(likes).

likes(john, mary).
likes(bill, barbara).
likes(bill, mary).
```

```
likes(mary, flowers).
likes(john, corvettes).
likes(john, john).

Yes


?- help(listing).
listing(+Pred)
    List  specified predicates  (when an  atom is  given all  predicates
    with  this name will  be listed).   The listing  is produced on  the
    basis  of the  internal representation,  thus loosing user's  layout
    and variable name information.  See also portray_c
listing
    List all predicates of the database using listi

Yes
```

---

help

```
?- help(help).
help
help(+What)
    Show specified part of the manual.  What is one of:

          <Name>/<Arity> give help on specified predicate
          <Name>         give  help on  named  predicate with  any
                         arity or  C interface function  with that
                         name
          <Section>      display  specified  section.      section
                         numbers are dash-separated  numbers:  2-3
                         refers  to  section 2.3  of  the  manual.

                         Section   numbers  are   obtained   using
                         apropos/1.

    Examples

       ?- help(assert).     give help on predicate assert
       ?- help(3-4).        display section 3.4 of the manual
       ?- help('PL_retry'). give    help   on    interface   function


Yes
?-
```

```
?- listing.

happy(john).
happy(mary).

sad(bill).
sad(barbara).
sad(john).

%   NOTE: system definition has been overruled for help/1

help(A) :-
        feature(gui, true), !,
        call(prolog_help(A)).
help(A) :-
        give_help(A).

'$user_query'(13, []) :-
        listing.

likes(john, mary).
likes(bill, barbara).
likes(bill, mary).
likes(mary, flowers).
likes(john, corvettes).
likes(john, john).

Yes
?-
```

We can also use **variables**, which start with a capital letter:

A **term** is a **constant** or a **variable**.

```
?- happy(X).
X=john ;
X=mary ;
no
```

the semicolon tells prolog that we want another answer

```
?- likes(john,X).
X=mary ;
X=corvettes ;
X=john ;
no
```

```
?- likes(X,mary).
X=john ;
X=bill ;
no

?- likes(X,Y).
X=john, Y=mary ;
X=bill, Y=barbara ;
...

?- likes(X,X).
X=john ;
no

?- likes(john,mary),likes(mary,flowers).
yes
```

the ‘,’ is pronounced **and**

```
?- likes(john,X),likes(X,flowers).
X=mary

?- likes(john,X),likes(bill,X).
X=mary

?- likes(X,Y),likes(Y,X).
X=john
Y=john ;
no

?- likes(bill,X),sad(X).
X=barbara

?- likes(bill,X),happy(X).
X=mary

?- happy(X),sad(Y).
X=john, Y=bill
X=john, Y=barbara
X=john, Y=john
X=mary, Y=bill
X=mary, Y=barbara
X=mary, Y=john
```

```
?- happy(X),happy(Y).

X = john
Y = john ;

X = john
Y = mary ;

X = mary
Y = john ;

X = mary
Y = mary ;

No


?- happy(X),sad(X).
X=john
```

RULES

```
/* factset2 */
happy(john).
happy(mary).
sad(bill).
sad(barbara).
sad(john).
likes(john,mary).
likes(bill,barbara).
likes(bill,mary).
likes(mary,flowers).
likes(john,corvettes).
likes(john,john).
wise(sam).
happy(X) :- wise(X).
likes(sam,X) :- likes(john,X).
fool(X) :- likes(X,X).


?- happy(X).
X = john ;
X = mary ;
X = sam ;
```

```
?- likes(john,X).
X = mary ;
X = corvettes ;
X = john ;

?- likes(sam,X).
X = mary ;
X = corvettes ;
X = john ;

?- fool(X).
X = john ;
```

Some family relationships — **`royalfamily`**:

```
mother(charles,elizabeth).
father(charles,phillip).
mother(andrew,elizabeth).
father(andrew,phillip).
mother(edward,elizabeth).
father(edward,phillip).
mother(william,diana).
father(william,charles).
mother(harry,diana).
father(harry,charles).
mother(elizabeth,elizabeth_tqm).
father(elizabeth,george).

grandfather(X,Y) :- father(X,Z),father(Z,Y).
grandfather(X,Y) :- mother(X,Z),father(Z,Y).

?- grandfather(william,phillip).
yes

?- grandfather(william,X).
X=phillip

?- grandfather(X,phillip).
X=william;
X=harry;

?- grandfather(X,Y).
X=charles,Y=george;
X=andrew,Y=george;
X=edward,Y=george;
X=william,Y=phillip;
X=harry, Y=phillip;
```

Anonymous variable

```
?- grandfather(andrew,_).
yes

?- likes(_,mary).
yes
```

Structures

```
person(name(george,bush),president).
person(name(laura,bush),first_lady).

?- person(X,Y).
X=name(george,bush), Y=president ;
X=name(laura,bush), Y=first_lady;
no

?- person(name(X,Y),Z).
X=george, Y=bush, Z=president ;
X=laura, Y=bush, Z=first_lady ;
no

?- person(name(X,bush),Y).
X=george, Y=president ;
X=laura, Y=first_lady ;
no
```

Structures are useful for things like:

```
(john,fitzgerald,kennedy)
(cher)
```

Equality

```
?- f(a,b)=f(a,X).
X = b ;
No


?- f(a,X)=f(Y,c(d,e)).
X = c(d, e)
```

```
Y = a ;
No
```

Arithmetic

```
=    \=   <    >    =<   >=
```

Monarch Example (reigns)

from Clocksin & Mellish, Fourth Edition, page 31

```
reigns(victoria,1837,1901).
reigns(edward7,1901,1910).
reigns(george5,1910,1936).
reigns(edward8,1936,1936).
reigns(george6,1936,1952).
reigns(elizabeth2,1952,2004).

monarch(Person,Year) :-
     reigns(Person,Low,High),
     Year >= Low,
     Year =< High.
```

```
?- monarch(X,1900).
X = victoria ;
No


?- monarch(X,1901).
X = victoria ;
X = edward7 ;
No


?- monarch(X,1936).
X = george5 ;
X = edward8 ;
X = george6 ;
No
```

Density Example (density)

from Clocksin & Mellish, Fourth Edition, pages 32-33

```
pop(usa,203).
pop(india,548).
pop(china,800).
pop(brazil,108).

area(usa,3).
area(india,1).
area(china,4).
area(brazil,3).

density(Country,Density) :-
     pop(Country,P),
     area(Country,A),
     Density is P/A.
```

```
?- density(brazil,X).
X = 36 ;
No


?- density(X,200).
X = china ;
No
```

LISTS

```
[]
[a]
[a,b]
[a,a]
[a,[b]]
[a,[],b]


?- [a,b,c]=[a,X,c].
X = b ;
No
```

```
?- [a,b,b]=[X,Y,Y].
X = a
Y = b ;
No


?- [X,is,a,fool]=[john,is,a,fool].
X = john ;
No
```

HEAD & TAIL

`[X|Y]` matches a list with head X and tail Y

In Prolog, we extract **car** and **cdr** by pattern matching,
not by explicit function calls:

```
| ?- [a,b,c]=[X|Y].
X = a,
Y = [b,c] ;
no


| ?- [a,b,c]=[a|Y].
Y = [b,c] ;
no


?- [a,b,c]=[X,Y|Z].
X = a
Y = b
Z = [c] ;
No


?- [a,b]=[X,Y|Z].
X = a
Y = b
Z = [] ;
No
```

LENGTH OF A LIST (length.pl)

```
len([],0).
len([X|Y],N) :- len(Y,M), N is M+1.


?- [length].
[WARNING: (/dfs/user/bisoroka/length.pl:2)
        Singleton variables: X]
length compiled, 0.00 sec, 1,080 bytes.


?- len([],N).
N = 0 ;
No


?- len([a,b,c],N).
N = 3 ;
No

?- len([a,b,c],5).
No

?- len(X,3).
X = [_G200, _G203, _G206]
```

these are gensyms

TRACE & SPY

```
?- trace(len).
        len/2: call redo exit fail


?- len([a,b,c],N).
T Call:  (  8) len([a, b, c], _G163)
T Call:  (  9) len([b, c], _L142)
T Call:  ( 10) len([c], _L155)
T Call:  ( 11) len([], _L168)
T Exit:  ( 11) len([], 0)
T Exit:  ( 10) len([c], 1)
T Exit:  (  9) len([b, c], 2)
T Exit:  (  8) len([a, b, c], 3)
N = 3
```

```
?- spy(len).
Spy point on len/2

?- len([a,b,c],N).
*  Call:  (  8) len([a, b, c], _G163) ? creep
*  Call:  (  9) len([b, c], _L142) ? creep
*  Call:  ( 10) len([c], _L155) ? creep
*  Call:  ( 11) len([], _L168) ? creep
*  Exit:  ( 11) len([], 0) ? creep
 ^ Call:  ( 11) _L155 is 0+1 ? creep
 ^ Exit:  ( 11) 1 is 0+1 ? creep
*  Exit:  ( 10) len([c], 1) ? creep
 ^ Call:  ( 10) _L142 is 1+1 ? creep
 ^ Exit:  ( 10) 2 is 1+1 ? creep
*  Exit:  (  9) len([b, c], 2) ? creep
 ^ Call:  (  9) _G163 is 2+1 ? creep
 ^ Exit:  (  9) 3 is 2+1 ? creep
*  Exit:  (  8) len([a, b, c], 3) ? creep
N = 3 ;
No

?- nospy(len).
Spy point removed from len/2

?- len([a,b,c],N).
N = 3 ;
No
```

MEMBER

```
member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).


| ?- member(a,[]).
no
| ?- member(a,[b,c,a]).
yes
| ?- member(a,[b,c,[a]]).
no
| ?- member([a],[b,c,[a]]).
yes
```

TRACING

```
| ?- trace.
yes
[trace]
| ?- member(a,[b,c,a]).
   (1) 0 Call (dynamic): member(a,[b,c,a]) ?
   (2) 1 Call (dynamic): member(a,[c,a]) ?
   (3) 2 Call (dynamic): member(a,[a]) ?
   (3) 2 Exit (dynamic): member(a,[a]) ?
   (2) 1 Exit (dynamic): member(a,[c,a]) ?
   (1) 0 Exit (dynamic): member(a,[b,c,a]) ?
yes
[trace]
| ?- notrace.
yes


| ?- member(X,[a,b,c]).
X = a ;
X = b ;
X = c ;
no


| ?- member(a,X).
X = [a|_8322] ;
X = [_8321,a|_8340] ;
X = [_8321,_8339,a|_8358] ;
X = [_8321,_8339,_8357,a|_8376] ;
X = [_8321,_8339,_8357,_8375,a|_8394] ;
X = [_8321,_8339,_8357,_8375,_8393,a|_8412] ;
```

```
?- statistics.
0.25 seconds cpu time for 27,806 inferences
2,337 atoms, 1,077 functors, 1,378 predicates, 22 modules,
32,495 VM-codes

                       Limit    Allocated       In use
Heap         : 2,135,416,832    1,213,224      453,380 Bytes
Local  stack :     2,048,000        8,192          596 Bytes
Global stack :     4,096,000       16,384          784 Bytes
Trail  stack :     4,096,000        8,192          240 Bytes

Yes
?-
```

NATURAL NUMBERS (natural.pl)

```
nat(0).
nat(X) :- nat(Y), X is Y+1.
```

```
?- nat(0).
Yes
```

```
?- nat(3).
Yes
```

```
?- nat(100).
Yes
```

```
?- nat(-1).
```
infinite loop!

```
?- nat(X).
X = 0 ;
X = 1 ;
X = 2 ;
X = 3
```
…

```
?- nat(X),X>10.
X = 11 ;
X = 12 ;
X = 13 ;
X = 14
```
…

```
?- nat(X),X<5.
X = 0 ;
X = 1 ;
X = 2 ;
X = 3 ;
X = 4 ;
```
…infinite loop…

Generating integer sequences

```
?- nat(X),Y is X+2.
X = 0, Y = 2 ;
X = 1, Y = 3 ;
X = 2, Y = 4 ;
X = 3, Y = 5 ;
…

?- nat(X),Y is 2-X.
X = 0, Y = 2 ;
X = 1, Y = 1 ;
X = 2, Y = 0 ;
X = 3, Y = -1 ;
X = 4, Y = -2 ;
…

?- nat(X),Y is 2-X, Y>=0.
X = 0, Y = 2 ;
X = 1, Y = 1 ;
X = 2, Y = 0 ;
```

infinite loop

We can generate integer sequences with specified properties:

```
even(X) :- nat(Y), X is 2*Y.
odd(X) :- nat(Y), X is 2*Y+1.


?- even(X).
X = 0 ;
X = 2 ;
X = 4 ;
```

```
X = 6 ;
```

…

This doesn't work well as a predicate:

`?- even(10).` → yes

`?- even(15).` → *infinite loop*

A SENTENCE TRANSFORMER

Suppose we want to transform sentences:

```
[you,are,a,computer] --> [i,[am,not],a,computer]
[do,you,speak,french] --> [no,i,speak,german]
```

```
/* from Clocksin & Mellish, Fourth Edition, page 53 */

change(you,i).
change(are,[am,not]).
change(french,german).
change(do,no).
change(X,X).

alter([],[]).
alter([Hi|Ti],[Ho|To]) :- change(Hi,Ho),alter(Ti,To).

\* where Hi = Head In      Ti = Tail In
         Ho = Head Out     To = Tail Out
*\
```

EXERCISE

Write the Prolog code which determines if a list consists entirely of atoms.

aa = all atoms

```
aa([]).
```

```
aa(H|T) :- atom(H), aa(T).


aa([a,b,c]) → yes
aa([a,2,c]) → no
```

EXERCISE

```
     /* Clocksin & Mellish, Third Edition, page 144 */

     nextto(X,Y,[X,Y|T]).
     nextto(X,Y,[_|T]) :- nextto(X,Y,T).
```

Try:

```
     nextto(a,b,[b,a,b,y]).
     nextto(a,b,X).
```

FACTORIAL

```
fac(0,1).
fac(N,Out) :-
        M is N-1,
        fac(M,Y),
        Out is N*Y.


| ?- fac(5,N).
** (1) 0 Call (dynamic): fac(5,_8346) ? l
** (4) 1 Call (dynamic): fac(4,_8476) ? l
** (7) 2 Call (dynamic): fac(3,_8531) ? l
** (10) 3 Call (dynamic): fac(2,_8586) ? l
** (13) 4 Call (dynamic): fac(1,_8641) ? l
** (16) 5 Call (dynamic): fac(0,_8696) ? l
** (16) 5 Done (dynamic): fac(0,1) ? l
** (13) 4 Done (dynamic): fac(1,1) ? l
** (10) 3 Done (dynamic): fac(2,2) ? l
** (7) 2 Done (dynamic): fac(3,6) ? l
** (4) 1 Done (dynamic): fac(4,24) ? l
** (1) 0 Done (dynamic): fac(5,120) ? l
N = 120
```

But, notice the possible infinite loop if we try backtracking …

```
?- fac(0,X).

X = 1 ;
[WARNING: Out of local stack]
   Exception: (38911) fac(-38903, _L505843) ? abort

Execution Aborted
```

APPEND

```
/* our own append */
/* Clocksin & Mellish, 4th ed., page 59 */

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


| ?- app([a],[b],X).
X = [a,b]


| ?- app(X,[c,d],[a,b,c,d]).
X = [a,b]


| ?- app(X,Y,[a,b,c]).
X=[],Y=[a,b,c];
X=[a],Y=[b,c];
X=[a,b],Y=[c];
X=[a,b,c],Y=[];
no
```

Let's restrict the answers to those which contain `c` —

```
?- app(X,Y,[a,b,c,d]),member(c,Y).
X = [],      Y = [a, b, c, d] ;
X = [a],     Y = [b, c, d] ;
X = [a, b],  Y = [c, d] ;
No
```

Derivation Trees & Choices

See Section 3.1 of Fisher's Prolog website.

NOT

```
?- atom(a).
yes
?- not(atom(a)).
no
```

OR goals — these use semicolon

```
f(a).
h(X) :- f(X);g(X).        /* an OR goal */
g(b).
```

Note that

```
h(X) :- f(X);g(X).
```

is equivalent to

```
h(X) :- f(X).
h(X) :- g(X).
```

FACTORIAL with CUT

Remember our original factorial? It permitted infinite loops.

```
fac(0,1).
fac(N,Out) :-
        M is N-1,
        fac(M,Y),
        Out is N*Y.
```

We want to prevent Prolog from using the second rule in computing fac(0,X).

```
/* factorial with cut */

     fac(0,1) :- !.
     fac(N,Out) :-
             M is N-1,
             fac(M,Y),
             Out is N*Y.



     ?- fac(0,X).
     X = 1 ;
     No
```

We're prevented from using rule 2 if rule 1 fails.

Another way to avoid the problem is this:

```
     fac(0,1).
     fac(N,Out) :-
             N > 0,
             M is N-1,
             fac(M,Y),
             Out is N*Y.
```

Another Example of the CUT

```
f(X) :- a(X),b(X).
f(X) :- c(X).

g(X) :- a(X),!,b(X).
g(X) :- c(X).

a(z).
c(z).


| ?- f(z).
yes
| ?- g(z).
no
```

because we're forbidden to backtrack beyond the cut!

981130 Generating pairs of numbers *with* replacement

```
/* pair.pl */

pair(L) :- L=[X,Y],
           digit(X),
           digit(Y).

digit(0).
digit(1).
digit(2).


?- pair(L).
L = [0,0] ;
L = [0,1] ;
L = [0,2] ;
L = [1,0] ;
L = [1,1] ;
L = [1,2] ;
L = [2,0] ;
L = [2,1] ;
L = [2,2] ;
no
```

920825

Generating pairs of numbers *without* replacement

```
/* pair2.pl */

pair(L) :- L = [X,Y],
           free_digit(X,L),
           free_digit(Y,L).

free_digit(Dig,UsedList) :- var(Dig),
                            digit(X),
                            not_in(X,UsedList),
                            Dig = X.

free_digit(Dig,_) :- nonvar(Dig).

digit(0).
digit(1).
digit(2).

not_in(_,[]).
```

```
not_in(D,[H|T]) :- var(H), not_in(D,T).
not_in(D,[H|T]) :- nonvar(H), D \= H, not_in(D,T).

?- pair(L).
L = [0,1] ;
L = [0,2] ;
L = [1,0] ;
L = [1,2] ;
L = [2,0] ;
L = [2,1] ;
no
```

A CRYPTARITHMETIC PROBLEM:

```
      S E N D
    + M O R E
    ---------
    M O N E Y
```

CRYPTARITHMETIC — 1

```
/* ca1.pl */

/* an early version of cryptarithmetic --
   allows the same digit to be used more than once */

ca(L) :- write('     S  E  N  D  M  O  R  Y'),
         L = [S,E,N,D,M,O,R,Y],
         col( 0,D,E,Y,C1),
         col(C1,N,R,E,C2),
         col(C2,E,O,N,C3),
         col(C3,S,M,O,C4),
         col(C4,0,0,M, 0).

col(Ci,X,Y,Z,Co) :- carry(Ci),
                    digit(X),digit(Y),digit(Z),
                    carry(Co),
                    sum(Ci,X,Y,Z),
                    carry(Ci,X,Y,Co).

sum(Ci,X,Y,Z) :- Z is (Ci+X+Y) mod 10.

carry(Ci,X,Y,Z) :- Z is (Ci+X+Y) // 10.
```

```
digit(0).
digit(1).
digit(2).
digit(3).
digit(4).
digit(5).
digit(6).
digit(7).
digit(8).
digit(9).

carry(0).
carry(1).
```

CRYPTARITHMETIC — 1

```
?- ca([0,0,0,0,0,0,0,0]). → Yes

?- ca([1,1,1,1,1,1,1,1]). → No


?- ca(L).
     S  E  N  D  M  O  R  Y
L = [0, 0, 0, 0, 0, 0, 0, 0] ;
L = [9, 0, 0, 0, 1, 0, 0, 0] ;
L = [0, 0, 1, 0, 0, 0, 9, 0] ;
L = [9, 0, 1, 0, 1, 0, 9, 0] ;
L = [1, 0, 2, 0, 0, 1, 8, 0] ;
L = [2, 0, 3, 0, 0, 2, 7, 0] ;
L = [3, 0, 4, 0, 0, 3, 6, 0] ;
L = [4, 0, 5, 0, 0, 4, 5, 0] ;
L = [5, 0, 6, 0, 0, 5, 4, 0] ;
L = [6, 0, 7, 0, 0, 6, 3, 0] ;
L = [7, 0, 8, 0, 0, 7, 2, 0] ;
L = [8, 0, 9, 0, 0, 8, 1, 0] ;
L = [8, 1, 0, 0, 0, 9, 1, 1] ;
L = [0, 1, 1, 0, 0, 0, 0, 1] ;
L = [9, 1, 1, 0, 1, 0, 0, 1] ;
```

```
L = [0, 1, 2, 0, 0, 0, 9, 1] ;
L = [9, 1, 2, 0, 1, 0, 9, 1] ;
L = [1, 1, 3, 0, 0, 1, 8, 1] ;
L = [2, 1, 4, 0, 0, 2, 7, 1] ;
...
```

CRYPTARITHMETIC — 2

```
/* ca2.pl
   Cryptarithmetic:
   forbids the same digit being used more than once */

ca(L) :- write('     S  E  N  D  M  O  R  Y'),
         L = [S,E,N,D,M,O,R,Y],
         col( 0,D,E,Y,C1,L),
         col(C1,N,R,E,C2,L),
         col(C2,E,O,N,C3,L),
         col(C3,S,M,O,C4,L),
         col(C4,0,0,M, 0,L).

col(Ci,X,Y,Z,Co,L) :- carry(Ci),
                      free_digit(X,L),
                      free_digit(Y,L),
                      free_digit(Z,L),
                      carry(Co),
                      sum(Ci,X,Y,Z),
                      carry(Ci,X,Y,Co).

sum(Ci,X,Y,Z) :- Z is (Ci+X+Y) mod 10.

carry(Ci,X,Y,Z) :- Z is (Ci+X+Y) // 10.

digit(0).
digit(1).
digit(2).
digit(3).
digit(4).
digit(5).
digit(6).
digit(7).
digit(8).
digit(9).

carry(0).
```

```
carry(1).

free_digit(Dig,UsedList) :- var(Dig),
                            digit(X),
                            not_in(X,UsedList),
                            Dig = X.
free_digit(Dig,_) :- nonvar(Dig).

not_in(_,[]).
not_in(D,[H|T]) :- var(H), not_in(D,T).
not_in(D,[H|T]) :- nonvar(H), D \= H, not_in(D,T).
```

CRYPTARITHMETIC — 2

```
?- ca(L).
     S  E  N  D  M  O  R  Y
L = [7, 5, 3, 1, 0, 8, 2, 6] ;
L = [5, 7, 3, 1, 0, 6, 4, 8] ;
L = [3, 8, 2, 1, 0, 4, 6, 9] ;
L = [6, 8, 5, 1, 0, 7, 3, 9] ;
L = [8, 4, 3, 2, 0, 9, 1, 6] ;
L = [8, 5, 4, 2, 0, 9, 1, 7] ;
L = [3, 7, 1, 2, 0, 4, 6, 9] ;
L = [5, 7, 3, 2, 0, 6, 4, 9] ;
L = [7, 6, 4, 3, 0, 8, 2, 9] ;
L = [6, 8, 5, 3, 0, 7, 2, 1] ;
L = [8, 3, 2, 4, 0, 9, 1, 7] ;
L = [6, 5, 2, 4, 0, 7, 3, 9] ;
L = [7, 5, 3, 4, 0, 8, 2, 9] ;
L = [6, 4, 1, 5, 0, 7, 3, 9] ;
L = [7, 3, 1, 6, 0, 8, 2, 9] ;
L = [9, 5, 6, 7, 1, 0, 8, 2] ;
…
```

Note:

```
?- ca([0,0,0,0,0,0,0,0]).
```

```
     S  E  N  D  M  O  R  Y
Yes
```

CRYPTARITHMETIC — 2

If we require that M=1, then there's only one solution:

```
?- ca([S,E,N,D,M,O,R,Y]),M=1.
     S  E  N  D  M  O  R  Y
S = 9
E = 5
N = 6
D = 7
M = 1
O = 0
R = 8
Y = 2 ;
No
```

---continue

Sentence Transformer -- without CUT

```
/* from Clocksin & Mellish, 4th ed., page 55 */
change(you,i).
change(are,[am,not]).
change(french,german).
change(do,no).
change(X,X).
alter([],[]).
alter([H|T],[X|Y]) :- change(H,X),alter(T,Y).


| ?- alter([you,are,my,friend],S).
S = [i,[am,not],my,friend] ;
S = [i,are,my,friend] ;
S = [you,[am,not],my,friend] ;
S = [you,are,my,friend] ;
no
```

Sentence Transformer *with* CUT

```
change(you,i) :- !.
change(are,[am,not]) :- !.
change(french,german) :- !.
change(do,no) :- !.
change(X,X).
alter([],[]).
alter([H|T],[X|Y]) :- change(H,X),alter(T,Y).


| ?- alter([you,are,my,friend],S).
S = [i,[am,not],my,friend] ;
no
```

DFSA definitions

acceptor — yes/no — of a “language”

draw diagram!

state machine

alphabet

initial state

final state

DFSA code

```
parse(L) :- start(S), trans(S,L).


trans(X,[A|B]) :-
      delta(X,A,Y),   /*  X ---A---> Y */
      write(X),
      write('  '),
      write([A|B]),
      nl,
```

```
      trans(Y,B).

trans(X,[]) :-
      final(X),
      write(X),
      write('  '),
      write([]), nl.
```

DFSA encoding

This DFSA recognizes (a,b)*ab(a,b)*.
From John Fisher tutorial.

```
delta(0,a,1).
delta(0,b,0).
delta(1,a,1).
delta(1,b,2).
delta(2,a,2).
delta(2,b,2).
start(0).
final(2).
```

DFSA examples

```
?- parse([a]).
0  [a]
No


?- parse([a,b]).
0  [a, b]
1  [b]
2  []
Yes
```

```
?- parse([b,a]).
0  [b, a]
0  [a]
No


?- parse([]).
No


?- parse([b,b,a,a,b,a,b]).
0  [b, b, a, a, b, a, b]
0  [b, a, a, b, a, b]
0  [a, a, b, a, b]
1  [a, b, a, b]
1  [b, a, b]
2  [a, b]
2  [b]
2  []
Yes
```

Searching a Maze

[Clocksin & Mellish, 4th Edition, pages 136ff.]

Show figure of rooms & doors.

Suppose we want to find a room with a telephone in it.

Here's the representation of the rooms & doors:

```
d(a,b).
d(b,e).
d(b,c).
d(d,e).
d(c,d).
d(e,f).
d(g,e).
```

Also,

```
hasphone(g).
```

Searching a Maze 2

If doors were one-way, we'd write:

```
go(X,X,T).
go(X,Y,T) :- d(X,Z), not(member(Z,T)), go(Z,Y,[Z|T]).
```

But, doors go both ways.

```
go(X,X,T).
go(X,Y,T) :- d(X,Z), not(member(Z,T)), go(Z,Y,[Z|T]).
go(X,Y,T) :- d(Z,X), not(member(Z,T)), go(Z,Y,[Z|T]).
```

or

```
go(X,X,T).
go(X,Y,T) :- ( d(X,Z) ; d(Z,X) ),
             not(member(Z,T)),
             go(Z,Y,[Z|T]).
```

or

```
go(X,X,T).
go(X,Y,T) :- connected(X,Z),
             not(member(Z,T)),
             go(Z,Y,[Z|T]).
connected(X,Y) :- d(X,Y) ; d(Y,X).
```

Searching a Maze 3

What about the telephone?

```
?- go(a,X,[]), hasphone(X).

?- hasphone(X), go(a,X,[]).
```

```
/* after Fisher page 15 */
/* simulating a DFSA */

:- dynamic trans/2, delta/3.
accept(L) :- start(S), trans(S,L).

trans(X,[A|B]) :-
        delta(X,A,Y),
        trans(Y,B).
trans(X,[]) :-
        final(X).

delta(0,a,1).
delta(0,b,0).
delta(1,a,1).
delta(1,b,2).
delta(2,a,2).
delta(2,b,2).

start(0).
final(2).
```

```
/* John Fisher prolog  tutorial 2.8 Change for a dollar  */


change([H,Q,D,N,P]) :-
    member(H,[0,1,2]),                  /* half-dollars */
    member(Q,[0,1,2,3,4]),              /* quarters     */
    member(D,[0,1,2,3,4,5,6,7,8,9,10]) , /* dimes       */
    member(N,[0,1,2,3,4,5,6,7,8,9,10,   /* nickels      */
              11,12,13,14,15,16,17,18,19,20]),
    S is 50*H + 25*Q +10*D + 5*N,
    S =<100,
    P is 100-S.
```

```
?- change([H,Q,D,N,P]).

H = 0
Q = 0
D = 0
N = 0
P = 100 ;
```

```
H = 0
Q = 0
D = 0
N = 1
P = 95 ;

H = 0
Q = 0
D = 0
N = 2
P = 90 ;

H = 0
Q = 0
D = 0
N = 3
P = 85 ;

H = 0
Q = 0
D = 0
N = 4
P = 80 ;

H = 0
Q = 0
D = 0
N = 5
P = 75 ;

H = 0
Q = 0
D = 0
N = 6
P = 70 ;
```

```
?- change([H,Q,D,N,P]),P=0.

H = 0
Q = 0
D = 0
N = 20
P = 0 ;

H = 0
Q = 0
D = 1
N = 18
P = 0 ;
```

```
H = 0
Q = 0
D = 2
N = 16
P = 0 ;
```

```
?- change([H,Q,D,N,P]),Q=4.

H = 0
Q = 4
D = 0
N = 0
P = 0 ;

No


?- change([H,Q,D,N,P]),Q=3.

H = 0
Q = 3
D = 0
N = 0
P = 25 ;

H = 0
Q = 3
D = 0
N = 1
P = 20 ;

H = 0
Q = 3
D = 0
N = 2
P = 15 ;

H = 0
Q = 3
D = 0
N = 3
P = 10 ;

H = 0
Q = 3
D = 0
N = 4
P = 5
```

981115

```
?- [factset1].
factset1 compiled, 0.00 sec, 1,472 bytes.
Yes


?- source_file(F).
F = '/usr/local/lib/pl-3.1.0/library/help.pl' ;
F = '/usr/local/lib/pl-3.1.0/library/helpidx.pl' ;
F = '/dfs/user/bisoroka/factset1.pl' ;
No
```

981115 — online help

help
Equivalent to help(help/1).

help(+What)
Show specified part of the manual. What is one of:
<Name>/<Arity>
give help on specified predicate
<Name>
give help on named predicate with any arity or
C interface function with that name
<Section>
display specified section. section numbers are
dash-separated numbers: 2-3 refers to section
2.3 of the manual. Section numbers are
obtained using apropos/1.

Examples
?- help(assert).
give help on predicate assert
?- help(3-4).
display section 3.4 of the manual
?- help('PL_retry').
give help on interface function PL_retry()

apropos(+Pattern)
Display all predicates, functions and sections that have Pattern in their name or summary

description. Lowercase letters in Pattern also match a corresponding uppercase letter. Example:

```
?- apropos(file).
```

Display predicates, functions and sections that have `file' (or `File', etc.) in their summary description.

explain(+ToExplain)

Give an explanation on the given `object'. The argument may be any Prolog data object. If the argument is an atom, a term of the form Name/Arity or a term of the form Module:Name/Arity, explain will try to explain the predicate as well as possible references to it.

explain(+ToExplain, -Explanation)

Unify Explanation with an explanation for ToExplain. Backtracking yields further explanations.

GRAPH TRAVERSAL

```
/* Mueller & Page 230 */
edge(1,2).
edge(2,3).
edge(1,4).
edge(3,4).
edge(4,1).
path(A,B,[A,B]) :- edge(A,B).
path(A,B,[A|P]) :- edge(A,X),path(X,B,P).
```

```
| ?- path(1,4,P).
P = [1,4] ;
P = [1,2,3,4] ;
P = [1,2,3,4,1,4] ;
P = [1,2,3,4,1,2,3,4] ;
P = [1,2,3,4,1,2,3,4,1,4] ;
P = [1,2,3,4,1,2,3,4,1,2,3,4] ;
P = [1,2,3,4,1,2,3,4,1,2,3,4,1,4] ;
P = [1,2,3,4,1,2,3,4,1,2,3,4,1,2,3,4] ;
P = [1,2,3,4,1,2,3,4,1,2,3,4,1,2,3,4,1,4] ;
P = [1,2,3,4,1,2,3,4,1,2,3,4,1,2,3,4,1,2,3,4]
```

Show & discuss my **prolog.ini**:

```
$ type prolog.ini

bs :- vms(dcl('eve foo')), reconsult('foo').
```

```
vms :- vms(dcl('')).

user :- consult(user).

?- load_files(library(not)).

?- load_files(library(strings)).

ed(File) :- concat_atom(['eve ',File],C),
        vms(dcl(C)),
        reconsult(File).

ed :- vms(dcl('eve')).
```

Halina's Graph Problem

file **hp1**

```
| ?- reachable(a,c,X).
X = [a,e,f,c] ;
X = [a,b,f,c] ;
X = [a,b,c] ;
no

| ?- reachable(a,h,X).
no

| ?- reachable(a,d,X).
X = [a,e,d] ;
no

| ?- reachable(a,b,X).
X = [a,b] ;
no

| ?- reachable(a,a,X).
X = [a] ;
no
```

Searching a Directed Graph

Clocksin & Mellish, 4th edition, page 158.

Show graph & Prolog clauses.

```
a(g,h).
a(g,d).
a(e,d).
a(h,f).
a(e,f).
a(a,e).
a(a,b).
a(b,f).
a(b,c).
a(f,c).


go(X,X).
go(X,Y) :- a(X,Z), go(Z,Y).
```

This can get into infinite loops — *e.g.* if we add `a(d,a)`.

Searching a Directed Graph 2 — avoiding cycles

```
go(X,X,T).
go(X,Y,T) :- a(X,Z), not(member(Z,T)), go(Z,Y,[Z|T]).
```

This is *depth-first* search.

```
:- dynamic min/2.
min([X],X).
min([X,Y|T],M) :- X<Y, min([X|T],M).
min([X,Y|T],M) :- min([Y|T],M).
```

```
/* Clocksin & Mellish page 141 */

rev([],[]).
rev([H|T],L) :- rev(T,Z) , append(Z,[H],L).



/* Clocksin & Mellish page 141 */

:- dynamic revzap/3.
rev2(L1,L2) :- revzap(L1,[],L2).
```

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

/* from Clocksin & Mellish page 165 */

```
:- dynamic permute/2.
permute([],[]).
permute(L,[H|T]) :-
        append(V,[H|U],L),
        append(V,U,W),
        permute(W,T).
```

How does this work?

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

BAGOF

```
parent(jan,bet).
parent(jan,cat).
parent(joe,ann).
parent(joe,cat).

| ?- bagof(X,parent(jan,X),B).
X = _8237, B = [bet,cat]

| ?- bagof(X,parent(Who,X),B).
X = _8237, Who = jan, B = [bet,cat] ;
X = _8237, Who = joe, B = [ann,cat]
```

PERMUTATIONS

Fisher:

```
:- dynamic sprinkle/3, perm/2.
perm([X|Y],Z) :- perm(Y,W), sprinkle(X,W,Z).
perm([],[]).

sprinkle(X,Y,[X|Y]).
sprinkle(X,[Y|R1],[Y|R2]) :- sprinkle(X,R1,R2).
```

From somebody or other:

```
:- dynamic permute/2.
permute([],[]).
permute(L,[H|T]) :-
        append(V,[H|U],L),
        append(V,U,W),
        permute(W,T).
```
