**PROLOG – EXEMPLE**

**REFERENCES:**

[http://www.sics.se/sicstus.html](http://www.sics.se/sicstus.html) | Sicstus PROLOG
[http://cs.wwc.edu/~cs_dept/KU/PR/Prolog.html](http://cs.wwc.edu/~cs_dept/KU/PR/Prolog.html) (Computer Science Department, Walla Walla College)

SICStus Prolog -[http://www.sics.se/sicstus/](http://www.sics.se/sicstus/)
SWI Prolog -[http://www.swi-prolog.org](http://www.swi-prolog.org)
Prolog.NET -[http://prolog.hodroj.net](http://prolog.hodroj.net)
Visual Prolog -[http://www.visual-prolog.com](http://www.visual-prolog.com)

**Prolog** was invented in the early seventies at the University of Marseille. Prolog stands for PROgramming in LOGic. It is a logic language that is particularly used by programs that use non-numeric objects.

For this reason it is a frequently used language in Artificial Intelligence where manipulation of symbols is a common task. Prolog differs from the most common programmings languages because it is declarative language.

Traditional programming languages are said to be procedural. This means that the programmer specify how to solve a problem. In declarative languages the programmers only give the problem and the language find himself how to solve the problem.

---

**J. A. Robinson**: A program is a theory (in some logic) and computation is deduction from the theory.

**N. Wirth**: Program = Data structure + Algorithm

**R. Kowalski**: Algorithm = Logic + Control
"Chai May Lim" <green_apple_seed@hotmail.com> wrote:
    How do I write a predicate that generates all the subsets of a list, Eg:  generatesubsets([1,2],X)?

Q. Why do I think this is a homework problem?
A. Because if you have a list of N elements, it has 2**N subsets, with an average of N/2 elements each, for a total space cost of 2**N * N/2 list cells.

Q. Suppose you have 1GB of memory, and a list cell costs 8 bits, how big can N be before you run out of memory?
A. N = 20 will fit. For N=21, you run out of memory.

Q. So that means that this predicate has no practical use?
A. It's certainly hard to imagine any practical use for it.

I have these:

```
generatesubsets([], []).
generatesubsets([Head|Tail], [Head|More]) :- generatesubsets(Tail, More).
generatesubsets([_|Tail], Subset) :- generatesubsets(Tail, Subset).
```

Pleasedon'trunyourwordstogetheritishardtoreadthemwhenyoudothat.
And don't forget to put spaces after top level commas and top-most argument commas.

```
subset([], []).
subset([X|Xs], [X|Ys]) :- subset(Xs, Ys).
subset([_|Xs], Ys) :- subset(Xs, Ys).
```
The output I want is in this form:

\[ X = [[1,2],[1],[2],[1]] \]

Method 1:

\[ \text{findall}(\text{Subset}, \text{subset}(Xs,\text{Subset}), \text{All\_Subsets}) \]

Method 2:

Program it directly.

\[ \text{all\_subsets}([], \text{[[ ]]}) \]
\[ \text{all\_subsets}(\text{X|Xs}, \text{Subsets}) :- \]
\[ \text{all\_subsets}(\text{Xs}, \text{Subsets1}), \]
\[ \text{subset}(\text{X|Xs}, \text{Subsets}, \text{Subsets1}) \]
\[ \text{library\_directory}('C:\Program Files\sicstus3\library') \]
\[ \text{file\_search\_path}(\text{library, A}) :- \]
\[ \text{library\_directory}(\text{A}) \]
This is a lot more space-efficient than the previous version. It takes only $2^{(N+1)} + (N-1)$ list cells. Even so, I can only fit $N=17$ into my copy of SWI Prolog with default settings for memory area sizes.

Can you please explain how can I do it? I know this is considered an easy question, but I'm new to prolog.

The question does not, in fact, have anything to do with Prolog. It is basic combinatorial programming, and the code would be practically identical in Scheme, ML, Haskell, and not that much different in C.

typedef struct Node *list;
struct Node {
    list next;
    int item;
};

list cons(int item, list next) {
    list r = malloc(sizeof *r);
    if (r == 0) abort();
    r->item = item, r->next = next;
    return r;
}

typedef struct NNode *llist;
struct NNode {
    llist next;
    list item;
};

llist ccons(list item, llist next) {
    llist r = malloc(sizeof *r);
    if (r == 0) abort();
    r->item = item, r->next = next;
    return r;
}

llist all_subsets(list xs) {
    llist r, t;
    if (xs == 0) {
        r = ccons(xs, 0);
    } else {
        r = all_subsets(xs->next);
        for (t = r; t != 0; t = t->next)
            r = ccons(cons(xs->item, t->item), r);
    }
    return r;
/* prolog tutorial 2.7 Prolog lists */

/* built-in SWI-Prolog */
member(X,[X|R]).
member(X,[Y|R]) :- member(X,R).
*/
takeout(X,[X|R],R).
takeout(X,[F|R],[F|S]) :- takeout(X,R,S).

putin(X,L,R) :- takeout(X,R,L).

/* built-in SWI-Prolog */
append([X|Y],Z,[X|W]) :- append(Y,Z,W).
append([],X,X).
*/

/* built-in SWI-Prolog */
reverse([X|Y],Z,W) :- reverse(Y,[X|Z],W).
reverse([],X,X).

reverse(A,R) :- reverse(A,[],R).
*/
perm([X|Y],Z) :- perm(Y,W), takeout(X,Z,W).
perm,[],[]).

subset([X|R],S) :- member(X,S), subset(R,S).
subset([],_).

union([X|Y],Z,W) :- member(X,Z), union(Y,Z,W).
union([X|Y],Z,[X|W]) :- \+ member(X,Z), union(Y,Z,W).
union([],Z,Z).

intersection([X|Y],M,[X|Z]) :- member(X,M), intersection(Y,M,Z).
intersection([X|Y],M,Z) :- \+ member(X,M), intersection(Y,M,Z).
intersection([],M,[]).

mergesort([],[]). /* covers special case */
mergesort([A],[A]).
mergesort([A,B|R],S) :-
    split([A,B|R],L1,L2),
    mergesort(L1,S1),
    mergesort(L2,S2),
    merge(S1,S2,S).

split([],[],[]).
split([A],[A],[]).
split([A,B|R],[A|Ra],[B|Rb]) :- split(Ra,Rb).

http://bach.istc.kobe-u.ac.jp/llp/current/examples/prolog/
merge(A, [], A).
merge([], B, B).
merge([A|Ra], [B|Rb], [A|M]) :- A =< B, merge(Ra, [B|Rb], M).
merge([A|Ra], [B|Rb], [B|M]) :- A > B, merge([A|Ra], Rb, M).

sequence_append((X, R), S, (X, T)) :-
    !,
    sequence_append(R, S, T).
sequence_append((X), S, (X, S)).

**Examples**

Here follow some program examples written in Prolog.

[edit] **QuickSort**

- partition(Xs, Pivot, Smalls, Bigs) partitions the list Xs into elements smaller than Pivot and elements equal to or larger than Pivot
- quicksort(Xs, Ss) relates a list to its sorted version

```prolog
partition([], _, [], []).
partition([X|Xs], Pivot, Smalls, Bigs) :-
    ( X @< Pivot ->
        Smalls = [X|Rest],
        partition(Xs, Pivot, Rest, Bigs)
    ;   Bigs = [X|Rest],
        partition(Xs, Pivot, Smalls, Rest)
    ).

quicksort([], []).
quicksort([X|Xs], Ascending) :-
    partition(Xs, X, Smaller0, Bigger0),
    quicksort(Smaller0, Smaller),
    quicksort(Bigger0, Bigger),
    append(Smaller, [X|Bigger], Ascending).
```

In systems supporting DCGs (as is common), one can write quicksort more concisely using a grammar rule:

```prolog
quicksort([]) --> [].
quicksort([X|Xs]) -->
    { partition(Xs, X, Smaller, Bigger) },
    quicksort(Smaller),
    [X],
    quicksort(Bigger).
```
[edit] Towers of Hanoi

This example simulates the Towers of Hanoi problem of moving discs from a left pole to a right pole.

```prolog
hanoi(N) :- move(N, left, right, centre).
move(0, _, _, _) :- !.
move(N, A, B, C) :-
    M is N-1,
    move(N, A, C, B),
    format("move a disc from the ~w pole to the ~w pole\n", [A,B]),
    move(M, C, B, A).
```

[edit] Computer Algebra

This example demonstrates the power and ease-of-use of symbolic manipulation in Prolog.

```prolog
/* Derivation Definition */
d(X,X,1) :- !. /\* d x dx = 1 */
d(C,_,0) :- atomic(C). /* d c dx = 0 */
d(-U,X,-A) :- d(U,X,A). /* d -u dx = - d u dx */
d(U+V,X,A+B) :- d(U,X,A), d(V,X,B). /* d u+v dx = d u dx + d v dx */
d(U-V,X,A-B) :- d(U,X,A), d(V,X,B). /* d u-v dx = d u dx - d v dx */
d(C*U,X,C*A) :- atomic(C), C \= X, d(U,X,A), !. /* d c*u dx = c*d u dx */
d(U*V,X,B*U+A*V) :- d(U,X,A), d(V,X,B). /* d u*v dx = u*d v dx + v*d u dx */
d(U/V,X,A) :- d(U*V^(-1),X,A). /* d u/v dx = d u dx / (u*v) */
d(C^U,X,C*U^(C-1)*W) :- atomic(C), C \= X, d(U,X,W). /* d u^c dx = c*u^(c-1)*u^dx */
d(log(U),X,A*U^(-1)) :- d(U,X,A). /* d ln(u) dx = u^(-1) * d u dx */
/* Integral Definition */
i(0,X,0) :- !. /* Int 0 dx = 0 */
i(X,X,(X^2)/2) :- !. /* Int X dx = (X^2)/2 */
i(C,X,C*X) :- atomic(C). /* Int c x dx = c*x */
i(-U,X,-A) :- i(U,X,A). /* Int -U dx = - Int U dx */
i(U+V,X,A+B) :- i(U,X,A), i(V,X,B). /* Int U+V dx = Int U dx + Int V dx */
i(U-V,X,A-B) :- i(U,X,A), i(V,X,B). /* Int U-V dx = Int U dx - Int V dx */
i(C*U,X,C*A) :- atomic(C), C \= X, i(U,X,A), !. /* Int cU dx = c (Int U dx) */
```
\[ i(X^C, X, (X^{C+1})/(C+1)) :- \text{atomic}(C), !. \] /* Int \( x^c \) \( dx = x^{c+1}/(c+1) \) */

\[ i(U, V, U*V-A) :- d(V, U, A), !. \] /* Int \( u \) \( dv = u*v - \text{Int} \ v \text{ } du \) */

/* Simplification Rules */

\[ s(+, X, 0, X). \] /* \( x + 0 = x \) */

\[ s(+, 0, X, X). \] /* \( 0 + x = x \) */

\[ s(+, X, Y, X+Y). \] /* \( x + y = x + y \) */

\[ s(+, X, Y, Z) :- \text{integer}(X), \text{integer}(Y), Z \text{ is } X+Y. \] /* \( x + y = z \) */

<- Calculate */

\[ s(*, \_, 0, 0). \] /* anything * 0 = 0 */

\[ s(*, 0, \_, 0). \] /* 0 * anything = 0 */

\[ s(*, 1, X, X). \] /* 1 * x = x */

\[ s(*, X, 1, X). \] /* x * 1 = x */

\[ s(*, X, Y, X*Y). \] /* x * y = x * y */

\[ s(*, X*Y, W, X*Z) :- \text{integer}(Y), \text{integer}(W), Z \text{ is } Y*W. \]

\[ s(*, X, Y, Z) :- \text{integer}(X), \text{integer}(Y), Z \text{ is } X*Y. \] /* \( x * y = z \) */

<- Calculate */

/* Simplification Definition */

simp(E, E) :- \text{atomic}(E), !.

simp(E, F) :- E =.. \[ \text{Op}, \text{La}, \text{Ra} \], simp(La, X), simp(Ra, Y), s(Op, X, Y, F).

[turing machine]

**Turing completeness** of Prolog can be shown by using it to simulate a Turing machine:

```prolog

turing(Tape0, Tape) :-
    perform(q0, [], Ls, Tape0, Rs),
    reverse(Ls, Ls1),
    append(Ls1, Rs, Tape).

perform(qf, Ls, Ls, Rs, Rs) :- !.

perform(Q0, Ls0, Ls, Rs0, Rs) :-
    ( Rs0 == [] -> Sym = b, RsRest = [] ; Rs0 = [Sym|RsRest] ),
    once(rule(Q0, Sym, Q1, NewSym, Action)),
    action(Action, Ls0, Ls1, [NewSym|RsRest], Rs1),
    perform(Q1, Ls1, Ls, Rs1, Rs).

action(left, Ls0, Ls, Rs0, Rs) :-
    ( Ls0 == [] -> Ls = [], Rs = [b|Rs0] ; Ls0 = [L|Ls], Rs = [L|Rs0] )
).

action(stay, Ls, Ls, Rs, Rs).
```

**edit**
The predicate turing/2 defines a relation between initial tape contents and the contents after a given Turing machine, starting in state q0 and looking at the first cell specified in the input (or blank if no input is specified), has performed its actions and moved to its final state, qf. The atom "b" is used to denote blank tape cells. The tape cells not specified in the input and not shown in the output are to be considered as containing blank symbols. A simple example Turing machine is specified by the facts:

\[
\text{rule(q0, 1, q0, 1, right).} \\
\text{rule(q0, b, qf, 1, stay).}
\]

This machine performs incrementation by one of a number in unary encoding: It loops over any number of "1" cells and appends an additional "1" at the end. Example query and result:

\[
?\text{- turing([1,1,1], Ts).} \\
Ts = [1, 1, 1, 1]
\]

**[edit] Meta-circular evaluator**

Prolog is a **homoiconic** language and provides many facilities for reflexion. Its implicit execution strategy make it possible to write a concise meta-circular evaluator for pure Prolog code:

\[
\text{mi_circ(true).} \\
\text{mi_circ((A,B)) :- mi_circ(A), mi_circ(B).} \\
\text{mi_circ(clause(A,B)) :- clause(A, B).} \\
\text{mi_circ(A \= B) :- A \= B.} \\
\text{mi_circ(G) :-} \\
\text{~~ G \= true,} \\
\text{~~ G \= (_,_),} \\
\text{~~ G \= (_\=_),} \\
\text{~~ G \= clause(_,_),} \\
\text{~~ clause(G, Body),} \\
\text{~~ mi_circ(Body).}
\]

A simple example predicate:

\[
\text{natnum(0).} \\
\text{natnum(s(X)) :- natnum(X).}
\]

The result of the interpreter interpreting itself on its execution of the query natnum(X):

\[
?\text{- mi_circ(mi_circ(natnum(X))).} \\
X = 0 ; \\
X = s(0) ; \\
X = s(s(0)) a \\
\text{Yes}
\]
Since Prolog code can be treated as data (represented using Prolog terms) that is easily read using built-in mechanisms (like read/1), it is easy to write interpreters that augment Prolog with domain-specific features.

>Data Structures

- All data types are inherently abstract.
  - A data type is defined by giving an algebraic description of the properties of its operators.
    - E.g., natural numbers could be defined by:
      - \[\text{sum}( \text{succ}(X), Y, \text{succ}(Z)) :- \text{sum}(X, Y, Z).\]
      - \[\text{sum}(0, X, X).\]
      - \[?- \text{sum}( \text{succ}(\text{succ}(0)), \text{succ}(\text{succ}(\text{succ}(0))), \text{Answer}.\]
      - \[\text{Answer} = \text{succ}(\text{succ}(\text{succ}(\text{succ}(\text{succ}(0))))).\]

Due to the inefficiency of this approach, most (all?) versions of Prolog provide basic arithmetic predicates, functions, and operators.

- Prolog provides no data constructors.
  - Prolog can operate directly with a logical description of the properties of a data structure.
  - Compound terms can represent data structures.
    - E.g., lists may be represented by the expressions that construct them:
      - \[\text{list}(\text{nil}).\]
      - \[\text{list}(\text{cons}(X, L)) :- \text{list}(L).\]
      - \[\text{null}(\text{nil}).\]
      - \[\text{car}(\text{cons}(X, L), X).\]
      - \[\text{cdr}(\text{cons}(X, L), L).\]
      - \[?- \text{car}(\text{cons}(\text{cons}(a, \text{cons}(b, \text{nil}))),\]
      - \[\text{cons}(\text{c, cons}(\text{d, nil}))), L).\]
      - \[L = \text{cons}(a, \text{cons}(b, \text{nil})).\]

- Prolog provides a built-in operator and some syntactic simplifications for representing lists.
  - The \textit{dot functor} or dot operator represents \text{cons}.
    - E.g.,
      - \[?- \text{car}((a.b.nil).(c.d.nil), X)\]
      - \[X = a.b.nil\]

If your version of Prolog does not support \texttt{.} (dot) as an operator, you can either use it as a functor (in which case the list \texttt{a.b.nil} would be written as \texttt{(a, (b, nil))}) or define it as an operator by entering \texttt{op(51, xfy, '.')}.

- [] represents \textit{nil}.
  - E.g.,
    - \[?- \text{car}((a.b[])(.c.d[]), X)\]
Prolog does not support `nil` as an alias for `[]` -- we introduced its use in the list-related predicates above in order to show the relationship between Prolog lists and LISP lists.

Like LISP, Prolog allows a simpler list notation. E.g.,

```
?- car([[a, b], c, d], X)
X = [a, b]
```

This is *syntactic sugar* -- lists are still compound terms.

To simplify splitting a list, Prolog uses `{ X | Y }` to represent a list with head `X` and tail `Y`. E.g.,

```
member(Element, [Element | _]).
member(Element, [_ | Tail]) :- member(Element, Tail).
?- member(a, [a, b, c]).
yes
?- member(a, [b, a, c]).
yes
?- member(d, [c, b, a]).
no
```

`[]` does not match either `{Element | _}` or `{ _ | Tail }`.

- Strings are represented internally as lists.
  - A string is another form of list notation. E.g., "abc" is treated internally as a list of ASCII character codes:
    ```
    [97, 98, 99]
    ```
- Example of a user-defined data type: `set`.
  - `set` membership -- same as list membership (see above)
  - `subset`
    ```
    subset([A | X], Y) :- member(A, Y), subset(X, Y).
    subset([], Y). % The empty set is a subset of every set.
    ```
  - `intersection`
    ```
    % Assumes lists contain no duplicate elements.
    intersection([], X, []).
    intersection([X | R], Y, [X | Z]) :-
      member(X, Y), !, intersection(R, Y, Z).
    intersection([X | R], Y, Z) :- intersection(R, Y, Z).
    ```
  - `union`
    ```
    union([], X, X).
    union([X | R], Y, Z) :-
      member(X, Y), !,
    ```
This lecture covers sort algorithms. Notice the natural and short representation of sort algorithms in Prolog.

### naive sort

Naive sort is not very efficient algorithm. It generates all permutations and then it tests if the permutation is a sorted list.

```prolog
naive_sort(List,Sorted):=-perm(List,Sorted),is_sorted(Sorted).

is_sorted([]).
is_sorted([_]).
is_sorted([X,Y|T]):=-X=<Y,is_sorted([Y|T]).
```

Naive sort uses the **generate and test** approach to solving problems which is usually utilized in case when everything else failed. However, sort is not such case.

### insert sort

Insert sort is a traditional sort algorithm. Prolog implementation of insert sort is based on idea of **accumulator**.

```prolog
insert_sort(List,Sorted):-i_sort(List,[],Sorted).
i_sort([],Acc,Acc).
i_sort([H|T],Acc,Sorted):-insert(H,Acc,NAcc),i_sort(T,NAcc,Sorted).

insert(X,[Y|T],[Y|NT]):-X=<Y.
insert(X,[Y|T],[X|T]):=-X>Y.
insert(X,[],[X]).
```

### bubble sort

Bubble sort is another traditional sort algorithm which is not very effective. Again, we use accumulator to implement bubble sort.

```prolog
bubble_sort(List,Sorted):-b_sort(List,[],Sorted).
b_sort([],Acc,Acc).
b_sort([H|T],Acc,Sorted):-
bubble(H,T,NT,Max),b_sort(NT,[Max|Acc],Sorted).

bubble(X,[],[X]).
bubble(X,[Y|T],[Y|NT],Max):=-X>Y,bubble(X,T,NT,Max).
bubble(X,[Y|T],[X|NT],Max):=-X=<Y,bubble(Y,T,NT,Max).
```
**merge sort**

Merge sort is usually used to sort large files but its idea can be utilized to every list. If properly implemented it could be a very efficient algorithm.

```
merge_sort([],[]).  % empty list is already sorted
merge_sort([X],[X]). % single element list is already sorted
merge_sort(List,Sorted):-
    List=[_,_,_],divide(List,L1,L2), % list with at least two
    elements is divided into two parts
    merge_sort(L1,Sorted1),merge_sort(L2,Sorted2), % then each part
    is sorted
    merge(Sorted1,Sorted2,Sorted). % and sorted
parts are merged
merge([],L,L).
merge(L,[],L):-L=[].
merge([X|T1],[Y|T2],[X|T]):=X=<Y,merge(T1,[Y|T2],T).
merge([X|T1],[Y|T2],[Y|T]):=X>Y,merge([X|T1],T2,T).
```

We can use **distribution into even and odd elements** of list

```
divide(L,L1,L2):-even_odd(L,L1,L2).
```

or traditional distribution into **first and second half** (other distributions are also possible)

```
divide(L,L1,L2):-halve(L,L1,L2).
```

**quick sort**

Quick sort is one of the fastest sort algorithms. However, its power is often overvalued. The efficiency of quick sort is sensitive to choice of pivot which is used to distribute list into two "halves".

```
quick_sort([],[]).
quick_sort([H|T],Sorted):-
pivoting(H,T,L1,L2),quick_sort(L1,Sorted1),quick_sort(L2,Sorted2),
    append(Sorted1,[H|Sorted2]).
pivoting(H,[],[],[]).
pivoting(H,[X|T],[X|L],G):-X=<H,pivoting(H,T,L,G).
pivoting(H,[X|T],L,[X|G]):-X>H,pivoting(H,T,L,G).
```

Similarly to merge sort, quick sort exploits the **divide and conquer** method of solving problems.

The above implementation of quick sort using append is not very effective. We can write better program using accumulator.

```
quick_sort2(List,Sorted):-q_sort(List,[],Sorted).
q_sort([],Acc,Acc).
q_sort([H|T],Acc,Sorted):-
pivoting(H,T,L1,L2),
    q_sort(L1,Acc,Sorted1),q_sort(L2,[H|Sorted1],Sorted)
```
Some type of sort can be found

% generated: 16 November 1989
% option(s): SOURCE_TRANSFORM_1
% qsort
% David H. D. Warren
% quicksort a list of 50 integers
qsort :- qsort([27, 74, 17, 33, 94, 18, 46, 83, 65, 2,
32, 53, 28, 85, 99, 47, 28, 82, 6, 11,
55, 29, 39, 81, 90, 37, 10, 0, 66, 51,
7, 21, 85, 27, 31, 63, 75, 4, 95, 99,
11, 28, 61, 74, 18, 92, 40, 53, 59, 8], _, []).
qsort([X|L], R, R0) :-
    partition(L, X, L1, L2),
    qsort(L2, R1, R0),
    qsort(L1, R, [X|R1]).
qsort([], R, R).
partition([X|L], Y, [X|L1], L2) :-
    X =< Y, !,
    partition(L, Y, L1, L2).
partition([X|L], Y, L1, [X|L2]) :-
    partition(L, Y, L1, L2).
partition([], _, [], []).

% generated: 10 November 1989
% option(s):
% (queens) queens_8
% from Sterling and Shapiro, "The Art of Prolog," page 211.
% solve the 8 queens problem
queens_8 :- queens(8, _), !.
queens_8 :-
    This program solves the N queens problem: place N pieces on an N
    by N rectangular board so that no two pieces are on the same line
    - horizontal, vertical, or diagonal. (N queens so placed on an N
    by N chessboard are unable to attack each other in a single move
    under the rules of chess.) The strategy is incremental generate-
    and-test.
    A solution is specified by a permutation of the list of numbers 1 to
    N. The first element of the list is the row number for the queen in
    the first column, the second element is the row number for the queen
    in the second column, et cetera. This scheme implicitly
    incorporates
    the observation that any solution of the problem has exactly one
    queen
    in each column.
The program distinguishes symmetric solutions. For example,

?- queens(4, Qs).
produces

Qs = [3,1,4,2] ;
Qs = [2,4,1,3]

queens(N,Qs) :-
        range(1,N,Ns),
        queens(Ns,[],Qs).

queens([],Qs,Qs).
queens(UnplacedQs,SafeQs,Qs) :-
        select(UnplacedQs,UnplacedQs1,Q),
        not_attack(SafeQs,Q),
        queens(UnplacedQs1,[Q|SafeQs],Qs).

not_attack(Xs,X) :-
        not_attack(Xs,X,1).

not_attack([],_1) :- !.
not_attack(Ds,X,N) :-
        X =\= Y+N, X =\= Y-N,
        N1 is N+1,
        not_attack(Ys,X,N1).

select([X|Xs],Xs,X).
select([Y|Ys],[Y|Zs],X) :- select(Ys,Zs,X).

range(N,N,[N]) :- !.
range(M,N,[M|Ns]) :-
        M < N,
        M1 is M+1,
        range(M1,N,Ns).

Where does the zebra live?
Puzzle solution written by Claude Sammut.

main :-
        houses(Houses),
        member(house(red, english, _, _, _), Houses),
        member(house(_, spanish, dog, _, _), Houses),
        member(house(green, _, _, coffee, _), Houses),
        member(house(_, ukrainian, _, tea, _), Houses),
        right_of(house(green,_,_,_), house(ivory,_,_,_), Houses),
        member(house(_, snails, _, winstons), Houses),
        member(house(yellow,_,_,_ kools), Houses),
        Houses = [_, _, house(_, _, milk, _), _],
        Houses = [house(_, norwegian, _, _)]|_],
        next_to(house(_,_,_,_ chesterfields), house(_,fox,_,_), Houses),
        next_to(house(_,_,_ kools), house(_,horse,_,_), Houses),
        member(house(_, _, orange_juice, lucky_strikes), Houses),
member(house(_, japanese, _, _, parliaments), Houses),
next_to(house(_, norwegian, _, _, _), house(blue, _, _, _), Houses),
member(house(_, _, zebra, _, _), Houses),
member(house(_, _, _, water, _), Houses),
print_houses(Houses).

houses([%
house(_, _, _, _, _),
house(_, _, _, _, _),
house(_, _, _, _, _),
house(_, _, _, _, _),
house(_, _, _, _, _)
%]).

right_of(A, B, [B, A | _]).
right_of(A, B, [__ | Y]) :- right_of(A, B, Y).

next_to(A, B, [A, B | _]).
next_to(A, B, [B, A | _]).
next_to(A, B, [__ | Y]) :- next_to(A, B, Y).

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

print_houses([A|B]) :- !,
    write(A), nl,
    print_houses(B).
print_houses([]).

% generated: 17 November 1989
% option(s): SOURCE_TRANSFORM_1
% %
% query
% %
% David H. D. Warren
% %
% query population and area database to find coun-
% tries of approximately equal population density

query :- query(_), fail.
query.

query([[C1,D1,C2,D2]]) :-
    density(C1,D1),
    density(C2,D2),
    D1 > D2,
    T1 is 20*D1,
    T2 is 21*D2,
    T1 < T2.

density(C,D) :-
    pop(C,P),
    area(C,A),
    D is (P*100)//A.
% populations in 10000's
pop(china, 8250).
pop(india, 5863).
pop(ussr, 2521).
pop(usa, 2119).
pop(indonesia, 1276).
pop(japan, 1097).
pop(brazil, 1042).
pop(bangladesh, 750).
pop(pakistan, 682).
pop(w_germany, 620).
pop(nigeria, 613).
pop(mexico, 581).
pop(uk, 559).
pop(italy, 554).
pop(france, 525).
pop(phillippines, 415).
pop(thailand, 410).
pop(turkey, 383).
pop(egypt, 364).
pop(spain, 352).
pop(poland, 337).
pop(s_korea, 335).
pop(iran, 320).
pop(ethiopia, 272).
pop(argentina, 251).

% areas in 1000's of square miles
area(china, 3380).
area(india, 1139).
area(ussr, 8708).
area(usa, 3609).
area(indonesia, 570).
area(japan, 148).
area(brazil, 3288).
area(bangladesh, 55).
area(pakistan, 311).
area(w_germany, 96).
area(nigeria, 373).
area(mexico, 764).
area(uk, 86).
area(italy, 116).
area(france, 213).
area(phillippines, 90).
area(thailand, 200).
area(turkey, 296).
area(egypt, 386).
area(spain, 190).
area(poland, 121).
area(s_korea, 37).
area(iran, 628).
area(ethiopia, 350).
area(argentina, 1080).

--------
% generated: 7 March 1990
% option(s):
%  (deriv) times10
%  David H. D. Warren
%  symbolic derivative of (((((((x*x)*x)*x)*x)*x)*x)*x)*x)*x

\text{times10} :- d((((((((x*x)*x)*x)*x)*x)*x)*x)*x)*x,x,_).

\text{d}(U+V,X,DU+DV) :- !,
\text{d}(U,X,DU),
\text{d}(V,X,DV).
\text{d}(U-V,X,DU-DV) :- !,
\text{d}(U,X,DU),
\text{d}(V,X,DV).
\text{d}(U*V,X,DU*V+U*DV) :- !,
\text{d}(U,X,DU),
\text{d}(V,X,DV).
\text{d}(U/V,X,(DU*V-U*DV)/(^2(V))) :- !,
\text{d}(U,X,DU),
\text{d}(V,X,DV).
\text{d}(^N(U,N),X,DU*N*(^N-1(U,N))) :- !,
\text{integer}(N),
\text{N1 is N-1},
\text{d}(U,X,DU).
\text{d}(-U,X,-DU) :- !,
\text{d}(U,X,DU).
\text{d}(\text{exp}(U),X,\text{exp}(U)*DU) :- !,
\text{d}(U,X,DU).
\text{d}(\text{log}(U),X,DU/U) :- !,
\text{d}(U,X,DU).
\text{d}(X,X,1) :- !.
\text{d}(_,_,0).

% generated: 7 March 1990
% option(s):
%  (deriv) divide10
%  David H. D. Warren
%  symbolic derivative of (((((((x/x)/x)/x)/x)/x)/x)/x)/x)/x

\text{divide10} :- d((((((((x/x)/x)/x)/x)/x)/x)/x)/x)/x,x,_).

\text{d}(U+V,X,DU+DV) :- !,
\text{d}(U,X,DU),
\text{d}(V,X,DV).
\text{d}(U-V,X,DU-DV) :- !,
\text{d}(U,X,DU),
\text{d}(V,X,DV).
\text{d}(U*V,X,DU*V+U*DV) :- !,
\text{d}(U,X,DU),
\text{d}(V,X,DV).
\text{d}(U/V,X,(DU*V-U*DV)/(^2(V))) :- !,
% Find the unique answer to:
% OEE
%   EE
%  ---
%  EOEE
%  EOE
%  ----
%  OOEE
% where E=even, O=odd.
% This program generalizes easily to any such problem.
% Written by Peter Van Roy

main :-
    odd(A), even(B), even(C),
    even(E),
    mult([C,B,A], E, [I,H,G,F|X]),
    lefteven(F), odd(G), even(H), even(I), zero(X),
    lefteven(D),
    mult([C,B,A], D, [L,K,J|Y]),
    lefteven(J), odd(K), even(L), zero(Y),
    sum([I,H,G,F], [0,L,K,J], [P,O,N,M|Z]),
    odd(M), odd(N), even(O), even(P), zero(Z),
    write(' '), write(A), write(B), write(C), nl,
    write('  '), write(D), write(E), nl,
    write(F), write(G), write(H), write(I), nl,
    write(J), write(K), write(L), nl,
    write(M), write(N), write(O), write(P), nl.

% Addition of two numbers
sum(AL, BL, CL) :- sum(AL, BL, 0, CL).

sum([A|AL], [B|BL], Carry, [C|CL]) :- !,
    X is (A+B+Carry),
    C is X mod 10,
    NewCarry is X // 10,
    sum(AL, BL, NewCarry, CL).
sum([], BL, 0, BL) :- !.
sum(AL, [], 0, AL) :- !.
sum([], [B|BL], Carry, [C|CL]) :- !,
    X is B+Carry,
    NewCarry is X // 10,
    C is X mod 10,
    sum([], BL, NewCarry, CL).
sum([A|AL], [], Carry, [C|CL]) :- !,
    X is A+Carry,
    NewCarry is X // 10,
    C is X mod 10,
    sum([], AL, NewCarry, CL).
sum([], [], Carry, [Carry]).

% Multiplication
mult(AL, D, BL) :- mult(AL, D, 0, BL).

mult([A|AL], D, Carry, [B|BL] ) :-
    X is A*D+Carry,
    B is X mod 10,
NewCarry is X // 10,
mult(\(AL, D, \text{NewCarry, BL}\).
mult([], _, Carry, [C, Cend]) :-
  C is Carry mod 10,
  Cend is Carry // 10.

text:

zero([]).
zero([0|L]) :- zero(L).

odd(1).
odd(3).
odd(5).
odd(7).
odd(9).

even(0).
even(2).
even(4).
even(6).
even(8).

text:

lefteven(2).
lefteven(4).
lefteven(6).
lefteven(8).

TIC-TAC-TOE
/*
 * Filename:  tictac.pro
 */

* Programmer:  Br. David Carlson
*
* Date:  September 23, 1989
* Modified:  March 15, 1991 to include new strategy.
* November 11, 1993 to block a known way to beat the program.
* November 13, 1993 to block another way to beat the program.
* December 7, 1999 to run under yap.
*
This is an intelligent tic-tac-toe program. The computer chooses its moves based on certain rules which embody human intuitions about how to play the game.
*
A board is represented by a structure of the form:
* For example, board(x,o,x,_,o,_,o,_,_)) represents the board:
*   x | o | x
*        --------
*      | o |
*        --------
*        o |   
*/
/*
 *
go :- nl, write('Playing tic-tac-toe.'), nl, explainMoves, !, games, !, nl, nl.

explainMoves :- nl, write('You choose a move by giving the number of the'), nl, write('square to which you wish to move:'), nl, nl, write('  1 | 2 | 3'), nl, printline, write('  4 | 5 | 6'), nl, printline, write('  7 | 8 | 9'), nl.
/* show square numbers */

printline :- write(' -----------'), nl.
games :- game, !, optionalGames.

optionalGames :- nl, write('Care for another game (y/n)? '), inputResponse(Response), !, process(Response).

inputResponse(Response) :- read(Answer), okResponse(Answer, Response), !.
/* error check and put in standard form */
inputResponse(Response) :- write('Not a valid response -- type y or n: '), inputResponse(Response).   /* reinput */
/* The standard response forms are yes and no: */
okResponse(y, yes).   okResponse('Y', yes).   okResponse(yes, yes).
okResponse('YES', yes).   okResponse('Yes', yes).
okResponse(n, no).   okResponse('N', no).   okResponse(no, no).
okResponse('NO', no).   okResponse('No', no).

process(no) :- !.   /* end the program */
process(yes) :- games.   /* continue play */
game :- nl, write('Do you wish to be x or o? '), inputPlayer(Player), nl, write('Do you want to go first (y/n)? '), inputResponse(PlayerMoves), !,
play(Player, PlayerMoves, 9, board(_,_,_,_,_,_,_,_,_)).   /* 3rd arg of play is # of open squares, originally 9 */

printBoard(board(V1, V2, V3, V4, V5, V6, V7, V8, V9)) :-
write(' '), writeVal(V1), write('|'), writeVal(V2), write('|'), writeVal(V3), nl, printline,
write(' '), writeVal(V4), write('|'), writeVal(V5), write('|'), writeVal(V6), nl, printline,
write(' '), writeVal(V7), write('|'), writeVal(V8), write('|'), writeVal(V9), nl, nl.

writeVal(V) :- var(V), !, write('   ').
writeVal(x) :- write(' x ').
writeVal(o) :- write(' o ').
inputPlayer(Player) :- read(Player), x_or_o(Player). /* error check */
inputPlayer(Player) :- write('Not a valid response -- type x or o: '),
inputPlayer(Player). /* reinput Player */

x_or_o(x). x_or_o(o).

play(Player, PlayerMoves, NumOpen, Board) :-
oneMove(Player, PlayerMoves, Board), !,
    negate(PlayerMoves, NewPlayerMoves),
    Open is NumOpen - 1,
    continuePlay(Player, NewPlayerMoves, Open, Board).

oneMove(Player, yes, Board) :- !, inputMove(Board, Location),
    makeMove(m(Location, Player), Board). /* human player moves */
oneMove(Player, no, Board) :- opposite(Player, Computer),
generateMove(Computer, Board, Location),
    write('My move: '), write(Location), nl,
    makeMove(m(Location, Computer), Board). /* computer moves */

negate(yes, no). negate(no, yes).

opposite(x, o). opposite(o, x).

inputMove(Board, Location) :- nl, write('Your move: '), read(Location),
    location(Location), argb(Location, Board, Val), var(Val). /* must be open */
inputMove(Board, Location) :- nl, write('Not a valid location.'),
    inputMove(Board, Location). /* reinput location */

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

argb(1, board(Val,_,_,_,_,_,_,_,_,_), Val) :- !.
argb(2, board(_,Val,_,_,_,_,_,_,_,_), Val) :- !.
argb(3, board(_,_,Val,_,_,_,_,_,_,_), Val) :- !.
argb(4, board(_,_,_,Val,_,_,_,_,_,_), Val) :- !.
argb(5, board(_,_,_,_,Val,_,_,_,_,_), Val) :- !.
argb(6, board(_,_,_,_,_,Val,_,_,_,_), Val) :- !.
argb(7, board(_,_,_,_,_,_,Val,_,_), Val) :- !.
argb(8, board(_,_,_,_,_,_,_,Val,_) , Val) :- !.
argb(9, board(_,_,_,_,_,_,_,_,Val), Val).

makeMove(m(Location, Val), Board) :- argb(Location, Board, Val),
    printBoard(Board).

position(1, 1, 2). position(2, 1, 6). position(3, 1, 10).
position(4, 3, 2). position(5, 3, 6). position(6, 3, 10).
position(7, 5, 2). position(8, 5, 6). position(9, 5, 10).

generateMove(Computer, Board, Location) :- slice(Pos1, Pos2, Pos3),
    argb(Pos1, Board, Val1), argb(Pos2, Board, Val2), argb(Pos3, Board, Val3),
    winningMove(Computer, Pos1, Pos2, Pos3, Val1, Val2, Val3, Location), !.
    /* First check if computer can win with this move */
generateMove(_, Board, Location) :- slice(Pos1, Pos2, Pos3),
argb(Pos1, Board, Val1), argb(Pos2, Board, Val2), argb(Pos3, Board, Val3),
  forcedMove(Pos1, Pos2, Pos3, Val1, Val2, Val3, Location), !.
/* Then check if computer is forced to move */
generateMove(Computer, Board, Pos3) :- diagonal(Pos1, Pos2, Pos3),
  argb(Pos1, Board, Val1), argb(Pos2, Board, Val2), argb(Pos3, Board, Val3),
  neighbors(Pos3, NbrA, NbrB),
  argb(NbrA, Board, ValA), argb(NbrB, Board, ValB),
  specialMove(Computer, Val1, Val2, Val3, ValA, ValB), !.
/* Computer tries to get 2 in a row along a diagonal with 3rd square * 
  open and with enemy occupied cells as neighbors of location to which * 
  to move, which is a corner. */
genenerateMove(Computer, Board, Pos3) :- diagonal(Pos1, Pos2, Pos3),
  argb(Pos1, Board, Val1), argb(Pos2, Board, Val2), argb(Pos3, Board, Val3),
  neighbors(Pos3, NbrA, NbrB),
  argb(NbrA, Board, ValA), argb(NbrB, Board, ValB),
  special2Move(Computer, Val1, Val2, Val3, ValA, ValB), !.
/* Computer tries to get 2 in a row along a diagonal with 3rd square * 
  open, location to move to is a corner, and neighbors of this * 
  corner are such that 1 is open and 1 is held by the opponent. */
genenerateMove(Computer, Board, Location) :- slice(Pos1, Pos2, Pos3),
  argb(Pos1, Board, Val1), argb(Pos2, Board, Val2), argb(Pos3, Board, Val3),
  goodMove(Computer, Pos1, Pos3, Val1, Val2, Val3, Location), !.
/* Computer tries to get 2 in a row with 3rd square open */
genenerateMove(_, Board, 5) :- argb(5, Board, Val), var(Val), !.
/* Computer takes the center square, if open */
genenerateMove(_, Board, Location) :- openCorner(Board, Location), !.
/* Computer prefers a corner position */
genenerateMove(_, Board, Location) :- location(Location),
  argb(Location, Board, Val), var(Val), !. /* take any open square */
slice(1,2,3). slice(4,5,6). slice(7,8,9). slice(1,4,7).
slice(2,5,8). slice(3,6,9). slice(9,5,1). slice(7,5,3).
diagonal(1,5,9). diagonal(9,5,1). /* list both orders */
diagonal(3,5,7). diagonal(7,5,3).
neighbors(1,2,4). neighbors(3,2,6).
neighbors(7,4,8). neighbors(9,6,8).
/* A winning move is found if computer has 2 in a row, 3rd square open */
winningMove(Computer, Pos1,_,_, Val1, Val2, Val3, Pos1) :- var(Val1),
  nonvar(Val2), Val2 == Computer, nonvar(Val3), Val3 == Computer, !.
winningMove(Computer,_, Pos2,_, Val1, Val2, Val3, Pos2) :- var(Val2),
  nonvar(Val1), Val1 == Computer, nonvar(Val3), Val3 == Computer, !.
winningMove(Computer,_,_, Pos3, Val1, Val2, Val3, Pos3) :- var(Val3),
  nonvar(Val1), Val1 == Computer, nonvar(Val2), Val2 == Computer, !.
/* Computer is forced to move if Player has 2 in a row, 3rd square open */
forcedMove(_, _, Pos3, Val1, Val2, Val3, Pos3) :- var(Val3),
    nonvar(Val1), nonvar(Val2), Val1 == Val2, !.
forcedMove(_, Pos2, _, Val1, Val2, Val3, Pos2) :- var(Val2),
    nonvar(Val1), nonvar(Val3), Val1 == Val3, !.
forcedMove(Pos1, _, _, Val1, Val2, Val3, Pos1) :- var(Val1),
    nonvar(Val2), nonvar(Val3), Val2 == Val3, !.

/* This special move was added 11/13/93 to block a known way to beat
the program. The suggested move is to Pos3. */
specialMove(Computer, Val1, Val2, Val3, ValA, ValB) :-
    var(Val2), var(Val3), nonvar(Val1), Val1 == Computer,
    nonvar(ValA), ValA \== Computer, nonvar(ValB), ValB \== Computer, !.
specialMove(Computer, Val1, Val2, Val3, ValA, ValB) :-
    var(Val1), var(Val3), nonvar(Val2), Val2 == Computer,
    nonvar(ValA), ValA \== Computer, nonvar(ValB), ValB \== Computer, !.

/* This special move was also added 11/13/93 to block a known way to
beat the program. The suggested move is to Pos3. */
special2Move(Computer, Val1, Val2, Val3, ValA, ValB) :-
    var(Val2), var(Val3), nonvar(Val1), Val1 == Computer,
    nonvar(ValA), ValA \== Computer, var(ValB), !.
special2Move(Computer, Val1, Val2, Val3, ValA, ValB) :-
    var(Val1), var(Val3), nonvar(Val2), Val2 == Computer,
    var(ValA), nonvar(ValB), ValB \== Computer, !.
special2Move(Computer, Val1, Val2, Val3, ValA, ValB) :-
    var(Val2), var(Val3), nonvar(Val1), Val1 == Computer,
    var(ValA), nonvar(ValB), ValB \== Computer, !.
special2Move(Computer, Val1, Val2, Val3, ValA, ValB) :-
    var(Val1), var(Val3), nonvar(Val2), Val2 == Computer,
    var(ValA), ValA \== Computer, var(ValB), !.

goodMove(Computer, _, Pos3, Val1, Val2, Val3, Pos3) :- var(Val2),
    var(Val3), nonvar(Val1), Val1 == Computer, !.
goodMove(Computer, Pos1, _, Val1, Val2, Val3, Pos1) :- var(Val1),
    var(Val3), nonvar(Val2), Val2 == Computer, !.
goodMove(Computer, Pos1, _, Val1, Val2, Val3, Pos1) :- var(Val1),
    var(Val2), nonvar(Val3), Val3 == Computer, !.

openCorner(Board, Location) :- corner(Location),
    argb(Location, Board, Val), var(Val).
openCorner(Board, Location) :-
corner(1).   corner(3).   corner(7).   corner(9).

corner(1).   corner(3).   corner(7).   corner(9).

/* A good move is for the computer to get 2 in a row, 3rd square open */
goodMove(Computer, _, Pos3, Val1, Val2, Val3, Pos3) :- var(Val2),
    var(Val3), nonvar(Val1), Val1 == Computer, !.
goodMove(Computer, Pos1, _, Val1, Val2, Val3, Pos1) :- var(Val1),
    var(Val3), nonvar(Val2), Val2 == Computer, !.
goodMove(Computer, Pos1, _, Val1, Val2, Val3, Pos1) :- var(Val1),
    var(Val2), nonvar(Val3), Val3 == Computer, !.

openCorner(Board, Location) :-
    corner(Location), argb(Location, Board, Val), var(Val).
oc = 0.

corner(1).   corner(3).   corner(7).   corner(9).

continuePlay(Player, Pos2, Pos3) :- won(Who, Board), 
    !, winnerOut(Who, Player).
continuePlay(Player, PlayerMoves, NumOpen, Board) :- NumOpen < 3,
    getMover(Player, PlayerMoves, Mover),
    openSquares(Board, Positions),
    willBeDraw(Mover, Positions, Board), !,
    write('The game is a draw.'), nl.
/* If NumOpen is 1 or 2 we check for a draw */
continuePlay(Player, PlayerMoves, NumOpen, Board) :-
    NumOpen == 1, 
    willBeDraw(Mover, Positions, Board), !,
    continuePlay(Player, PlayerMoves, NumOpen, Board) :-
    NumOpen == 2, 
    willBeDraw(Mover, Positions, Board), !,
    continuePlay(Player, PlayerMoves, NumOpen, Board) :-
    NumOpen == 3, 
    willBeDraw(Mover, Positions, Board), !,
    continuePlay(Player, PlayerMoves, NumOpen, Board) :-
    NumOpen == 4, 
    willBeDraw(Mover, Positions, Board), !,
    continuePlay(Player, PlayerMoves, NumOpen, Board) :-
    NumOpen == 5, 
    willBeDraw(Mover, Positions, Board), !,
play(Player, PlayerMoves, NumOpen, Board).
   /* Continue play if no win or draw has been found */

won(Who, Board) :- slice(Pos1, Pos2, Pos3), argb(Pos1, Board, Val1),
   nonvar(Val1), Who = Val1, /* Who is potential winner */
   argb(Pos2, Board, Val2), nonvar(Val2), Val1 == Val2,
   argb(Pos3, Board, Val3), nonvar(Val3), Val1 == Val3.

winnerOut(Player, Player) :- !, write('You won. Congratulations!'), nl.
winnerOut(_,_) :- write('I won!'), nl.

/* getMover gets letter (x or o) of player ready to move */
getMover(Player, yes, Player) :- !.
getMover(Player, no, Computer) :- opposite(Player, Computer).

openSquares(Board, Positions) :-
   findall(Pos, openPos(Board, Pos), Positions).
   /* Positions becomes list of open positions in Board */

openPos(Board, Pos) :- location(Pos), argb(Pos, Board, Val), var(Val).

willBeDraw(Mover, Positions, Board) :-
   not(couldBeWin(Mover, Positions, Board)).

couldBeWin(Mover, [Pos1, Pos2], Board) :-
   test2(Mover, Pos1, Pos2, Board), !.
   /* Test if the possible last 2 moves could lead to a win */
couldBeWin(Mover, [Pos1, Pos2], Board) :- !,
   test2(Mover, Pos2, Pos1, Board).

couldBeWin(Mover, [Pos], Board) :- test1(Mover, Pos, Board).
   /* Test if last possible move gives a draw */

test2(Mover, Pos1, Pos2, Board) :- argb(Pos1, Board, Mover),
   opposite(Mover, Next), argb(Pos2, Board, Next), win(Board).
   /* Test if move to Pos1 and next move to Pos2 leads to winner */
test1(Mover, Pos, Board) :- argb(Pos, Board, Mover), win(Board).
   /* Test if move to Pos leads to winner */

win(Board) :- won(x, Board), !.
win(Board) :- won(o, Board).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%
% Copyright (C) 1990 Regents of the University of California.
% All rights reserved. This program may be freely used and modified for
% non-commercial purposes provided this copyright notice is kept unchanged.
% Written by Peter Van Roy as a part of the Aquarius project.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%
% Benchmark based on part of Aquarius Prolog compiler
% Compiling unification into abstract machine code.
main :- main(X), write(X), nl.

main(Size) :- u(X, [1,Y], [X], Code), size(Code, 0, Size).

% Unify variable X with term T and write the result:
% u(X, T, In, Code) :- unify(X, T, In, __, Code, []).

% Unify the variable X with the term T, given that
% In = set of variables initialized before the unification.
% Returns the intermediate code for the unification and
% Out = set of variables initialized after the unification.
% unify(X, T, In, Out) --> {in(X, In)}, !, uninit(X, T, In, Out).
% unify(X, T, In, Out) --> {in(X, In)}, !, init(X, T, In, Out, nonlast, _).

%**** Unify assumes X has not yet been initialized:
uninit(X, T, In, Out) --> {compound(T)}, !, [move(Tag^h, X)],
   {termtag(T, Tag)}, unify_block(nonlast, T, __, In, Mid, __),
   {incl(X, Mid, Out)}.

uninit(X, T, In, Out) --> {atomic(T)}, !, [move(tatm^T, X)], {incl(X, In, Out)}.
uninit(X, T, In, Out) --> {var(T)}, !, unify_var(X, T, In, Out).

%**** Init assumes X has already been initialized:
init(X, T, In, Out, Last, LLbls) --> {nonvar(T)}, !,
   {termtag(T, Tag)}, [deref(X), switch(Tag,X,[trail(X) | Write],Read,fail)],
   {unify_writemode(X, T, In, Last, LLbls, Write, [])},
   {unify_readmode(X, T, In, Out, Last, LLbls, Read, [])}.

init(X, T, In, Out, _, _) --> {var(T)}, !, unify_var(X, T, In, Out).

%**** Unifying two variables together:
unify_var(X, Y, In, In) --> {in(X, In), in(Y, In)}, !,
   [move(X,Y,fail)].
unify_var(X, Y, In, Out) --> {in(X, In), \+in(Y, In)}, !, [move(X,Y)],
   {incl(Y, In, Out)}.
unify_var(X, Y, In, Out) --> \+in(X, In), in(Y, In), !, [move(X,Y)],
   {incl(X, In, Out)}.
unify_var(X, Y, In, Out) --> \+in(X, In), \+in(Y, In), !,
   [move(tvar^h,X), move(tvar^h,Y), add(1,h), move(Y,[h-1])],
   {incl(X, In, Mid), incl(Y, Mid, Out)}.

%**** Unify_readmode assumes X is a dereferenced nonvariable
% at run-time and T is a nonvariable at compile-time.
unify_readmode(X, T, In, Out, LLbls) --> {structure(T)}, !,
   [equal([X],tatm^F/P/N,fail)],
   {functor(T, F, N)}, unify_args(1, N, T, In, Out, 0, X, LLbls).
unify_readmode(X, T, In, Out, LLbls) --> {cons(T)}, !,
   unify_args(1, 2, T, In, Out, -1, X, LLbls).
unify_readmode(X, T, In, In, _) --> {atomic(T)}, !,
   [equal(X,tatm^T,fail)].

unify_args(I, N, __, In, __, __) --> {I>N}, !.
unify_args(I, N, T, In, Out, D, X, [__ | LLbls]) --> {I=N}, !,
   unify_arg(I, T, In, Out, D, X, last, LLbls).
unify_args(I, N, T, In, Out, D, X, LLbls) --> {I<N}, !,
unify_arg(I, T, In, Mid, D, X, nonlast, _),
   (I1 is I+1), unify_args(I1, N, T, Mid, Out, D, X, LLbls).

unify_arg(I, T, In, Out, D, X, Last, LLbls) --> [move([X+ID],Y)],
   (ID is I+D, incl(Y, In, Mid), arg(I, T, A)),
   init(Y, A, Mid, Out, Last, LLbls).

%**** Unify_writemode assumes X is a dereferenced unbound
% variable at run-time and T is a nonvariable at compile-time.
unify_writemode(X, T, In, Last, LLbls) --> {compound(T)}, !,
   [move(Tag^h,[X])],
   {termtag(T, Tag)}, unify_block(Last, T, _ In, _, LLbls).
unify_writemode(X, T, _, _, _, _) --> {atomic(T)}, !,
   [move(tatm^T,[X])].

%**** Generate a minimal sequence of moves to create T on the heap:
block(T, Inf, Outf, In, Out, LLbls) --> {structure(T)}, !,
   [move(tatm^(F/N),[h+Inf])],
   {functor(T, F, N), Midf is Inf+N+1, S is Inf+1},
   make_slots(1, N, T, S, Offsets, In, Mid),
   block_args(1, N, T, Midf, Outf, Offsets, Mid, Out, LLbls).
block(T, Inf, Outf, In, Out, LLbls) --> {cons(T)}, !,
   {Midf is Inf+2},
   make_slots(1, 2, T, Inf, Offsets, In, Mid),
   block_args(1, 2, T, Midf, Outf, Offsets, Mid, Out, LLbls).
block(T, Inf, Inf, In, In, []) --> {atomic(T)}, !.
block(T, Inf, Inf, In, In, []) --> {atomic(T)}, !.

block_args(I, N, _, Inf, Inf, [], In, In, []) --> {I>N}, !.
block_args(I, N, T, Inf, Outf, [Inf], In, Out, [Lbl | LLbls]) --> {I=N}, !,
   [label(Lbl)],
   {arg(I, T, A)}, block(A, Inf, Outf, In, Out, LLbls).
block_args(I, N, T, Inf, Outf, [Inf | Offsets], In, Out, LLbls) --> {I<N}, !,
   {arg(I, T, A)}, block(A, Inf, Midf, In, Mid, _), (I is I+1),
   block_args(I, N, T, Midf, Outf, Offsets, Mid, Out, LLbls).

make_slots(I, N, S, [Off | Offsets], In, Out) --> {I<=N}, !,
   {arg(I, T, A)}, init_var(A, S, In),
   {incl(A, In, Mid), make_word(A, Off, Word)}, [move(Word,[h+S])],
   {S1 is S+1, I1 is I+1},
   make_slots(I1, N, T, S1, Offsets, Mid, Out).

% Initialize first-time variables in write mode:
init_var(V, I, In) --> {var(V), \+in(V, In)), !, [move(tvar^(h+I),V)].
init_var(V, _, In) --> {var(V), in(V, In)), !.
init_var(V, _, _) --> {nonvar(V)}, !.


make_word(C, Off, Tag^(h+Off)) :- compound(C), !, termtag(C, Tag).
make_word(V, _, V)           :- var(V), !.
make_word(A, _, tatm^A)      :- atomic(A), !.

% Calculate the size of T on the heap:
size(T) --> {structure(T)}, !, {functor(T, _, N)}, add(1), add(N),
           size_args(1, N, T).
size(T) --> {cons(T)}, !, add(2), size_args(1, 2, T).
size(T) --> {atomic(T)}, !.
size(T) --> {var(T)}, !.

size_args(I, N, _) -->  {I>N}, !.
size_args(I, N, T) --> {I=<N}, !, {arg(I, T, A)}, size(A), {I1 is I+1},
                      size_args(I1, N, T).

%**** Utility routines:
add(I, X, Y) :- Y is X+I.

in(A, [B|S]) :-
    compare(Order, A, B),
    in_2(Order, A, S).

in_2(=, _, _).
in_2(>, A, S) :- in(A, S).

incl(A, S1, S) :- incl_2(S1, A, S).

incl_2([], A, [A]).
incl_2([B|S1], A, S) :-
    compare(Order, A, B),
    incl_3(Order, A, B, S1, S).

incl_3(<, A, B, S1, [A,B|S1]).
incl_3(=, _, B, S1, [A,B|S1]).
incl_3(>, A, B, S1, [B|S]) :- incl_2(S1, A, S).

compound(X) :- nonvar(X), \+atomic(X).
cons(X) :- nonvar(X), X=[_|_].
structure(X) :- compound(X), \+X=[_|_].

termtag(T, tstr) :- structure(T).
termtag(T, tlst) :- cons(T).
termtag(T, tatm) :- atomic(T).
termtag(T, tvar) :- var(T).

//********************************************************************************

Prolog Example
Pentomino Puzzle
(posed by Henry E. Duden, 1907 and Solomon W. Golomb, 1953)
written by
Mutsunori Banbara (banbara@cs32.scitec.kobe-u.ac.jp)
Nara National College of Technology
main :-
    write('Pentomino Puzzle '), nl,
    write('Please select a board size.
    3 : 3 * 20'), nl,
    write('  4 : 4 * 15'), nl,
    write('  5 : 5 * 12'), nl,
    write('  6 : 6 * 10'), nl,
    write('  8 : 8 * 8 (with 4 blanks in the center)'), nl,
    write('Please select a board size in (3..8).'), nl,
    write('Number = '), nl,
    read_int(X), nl,
    read_yn('All solutions (y/n)? ', All), nl,
    read_yn('Output (y/n)? ', Output), nl,
    statistics(runtime, _), nl,
    % statistics(cputime, T1), nl,
    solve_pent(X, all(All), output(Output)), nl,
    % statistics(cputime, T2), nl,
    % T is (T2 - T1)*1000, nl,
    statistics(runtime, [_, T]), nl,
    write('CPU time = '), write(T), write(' msec'), nl.

read_yn(Message, YN) :-
    write(Message), nl,
    read_line([Char|_]), nl,
    (Char == "y" -> YN = yes; YN = no).

solve_pent(X, All(All), output(Output)) :-
    solve_pentomino(X, Board),
    (Output == yes -> show_result(Board), nl; true),
    All == no, !.

solve_pent(_, _, _).

solve_pentomino(X, Board) :-
    pent_board(X, Y, Board),
    Col Is Y+2,
    set_x_pentomino(X, Y, Col, Board, 20, Z),
    solve_pent(X, Y, Col, Board),
    remove_symmetry(Z0, Z).

remove_symmetry(Z0, Z) :-
    nonvar(Z0), nonvar(Z), !,
Z0 @=< Z.
remove_symmetry(_, _).

solve_pent(X, Y, Col, Board) :-
    get_search_list(List, 1, 1, X, Y, Col, Board),
    Pts = [\'F\', \'I\', \'L\', \'N\', \'P\', \'T\', \'U\', \'V\', \'W\', \'Y\', \'Z\'],
    solve_pent0(Pts, List, Col, Board).

solve_pent0([], _, _, _) :- !.
solve_pent0(Pts, [(N, E)|Ls], Col, Board) :-
    var(E),
    !,
    not_one_space(N, Col, Board),
    pent_select(P, Pts, Pts1),
    place_pent(P, N, Col, Board),
    solve_pent0(Pts1, Ls, Col, Board).

not_one_space(N, Col, B) :-
    (   N1 is N+1, arg(N1, B, X1), var(X1)
    ;   N2 is N+Col, arg(N2, B, X2), var(X2)
    ),
    !.

get_search_list([], I, J, X, Y, _, _) :-
    J =:= Y+1, I =:= X-1,
    !.
get_search_list(L, I, J, X, Y, Col, B) :-
    I =:= 0, J =< X,
    !,
    get_search_list(L, J, 1, X, Y, Col, B).
get_search_list(L, I, J, X, Y, Col, B) :-
    I =:= 0, J > X,
    !,
    J1 is J-X+1,
    get_search_list(L, X, J1, X, Y, Col, B).
get_search_list(L, I, J, X, Y, Col, B) :-
    J =:= Y+1,
    !,
    J1 is I+2+Y-X,
    get_search_list(L, X, J1, X, Y, Col, B).
get_search_list([\(\(N,Z\)\)|Ls], I, J, X, Y, Col, B) :-
    N is Col*I+J+1,
    arg(N, B, Z),
    var(Z),
    !,
    I1 is I-1,
    J1 is J+1,
    get_search_list(Ls, I1, J1, X, Y, Col, B).
get_search_list(L, I, J, X, Y, Col, B) :-
    I1 is I-1,
    J1 is J+1,
    get_search_list(L, I1, J1, X, Y, Col, B).

/**************************************************************************
Remove Symmetry by using X-pentomino
******************************************************************
set_x_pentomino(X, _, _, Board, Z0, Z) :-
  X =:= 3, !,
  (place_pent('X', 2, 1, 22, Board),
   look_up_board(1, 3, 22, Board, Z0),
   look_up_board(3, 3, 22, Board, Z);
   place_pent('X', 2, 2, 22, Board),
   look_up_board(1, 4, 22, Board, Z0),
   look_up_board(3, 4, 22, Board, Z);
   place_pent('X', 2, 3, 22, Board),
   look_up_board(1, 5, 22, Board, Z0),
   look_up_board(3, 5, 22, Board, Z);
   place_pent('X', 2, 4, 22, Board),
   look_up_board(1, 6, 22, Board, Z0),
   look_up_board(3, 6, 22, Board, Z);
   place_pent('X', 2, 5, 22, Board),
   look_up_board(1, 7, 22, Board, Z0),
   look_up_board(3, 7, 22, Board, Z);
   place_pent('X', 2, 6, 22, Board),
   look_up_board(1, 8, 22, Board, Z0),
   look_up_board(3, 8, 22, Board, Z);
   place_pent('X', 2, 7, 22, Board),
   look_up_board(1, 9, 22, Board, Z0),
   look_up_board(3, 9, 22, Board, Z);
   place_pent('X', 2, 8, 22, Board),
   look_up_board(1, 10, 22, Board, Z0),
   look_up_board(3, 10, 22, Board, Z);
   place_pent('X', 2, 9, 22, Board),
   look_up_board(1, 11, 22, Board, Z0),
   look_up_board(3, 11, 22, Board, Z)
  )).

set_x_pentomino(X, _, _, Board, Z0, Z) :-
  X =:= 8, !,
  (place_pent('X', 3, 1, 10, Board);
   place_pent('X', 4, 1, 10, Board);
   place_pent('X', 3, 2, 10, Board),
   look_up_board(2, 4, 10, Board, Z0),
   look_up_board(4, 2, 10, Board, Z)
  ).

set_x_pentomino(X, Y, Col, Board, Z0, Z) :-
  4 =< X, X =< 6, !,
  M is (X-1)//2,
  N is (Y-1)//2,
for(J0, 1, N),
   for(I0, 1, M),
   I is I0+1,
   J is J0,
   place_pent('X', I, J, Col, Board),
   { 
      X := 5, I := 3 ->
      look_up_board(1, J, Col, Board, Z0),
      look_up_board(5, J, Col, Board, Z)
   ;
   true
   },
   { 
      X := 4, I := 2, J := 7 ->
      look_up_board(1, 7, Col, Board, Z0),
      look_up_board(1, 9, Col, Board, Z)
   ;
   true
   ).

look_up_board(I, J, Col, B, X) :-
P is Col*I+J+1,
arg(P, B, X).

pent_board(I, J, Board) :- 3 =< I, I =< 6, !,
   J is 60//I,
mak
J1 is J+1,
frame(I, J1, M, N, Board).

frame(I, J, M, N, Board) :-
  (I =:= 1 ; I =:= M ; J =:= 1 ; J =:= N),
  !,
  L is (I-1)*N+J,
  arg(L, Board, '*'),
  I1 is I+1,
  J1 is J,
  frame(I1, J1, M, N, Board).

frame(I, J, M, N, Board) :-
  I1 is I+1,
  J1 is J,
  frame(I1, J1, M, N, Board).

show_result(B) :-
  board_size(B, H, W),
  Col is W+2,
  for(I, 1, H),
  nl,
  for(J, 1, W),
  look_up_board(I, J, Col, B, P),
  write_pent(P),
  fail.
show_result(_) :- nl.

write_pent(P) :- var(P), !, write('_ ').
write_pent(P) :- write(P), write(' ').

board_size(B, H, W) :-
  board_width(B, W),
  board_height(W, H).

board_width(B, W) :-
  count_flame(B, 1, W).

count_flame(B, N, W) :-
  arg(N, B, P), P == '*',
  !,
  N1 is N+1,
  count_flame(B, N1, W).

count_flame(_, N, W) :-
  W is N-4.

board_height(W, H) :-
  10 =< W, W =< 20,
H is 60//W.

board_height(8, 8).

/**
 * place_pent(P, I, J, Col, B)
 * P  : %N$r#1$DG[CV$9$k (B
 * I  : %N$sHV9f (B
 * J  : %N$shHV9f (B
 * Col: %N$hn?t (B+2(+2 %N$JoI,MW (B)
 * B  : %I$\!<%IS9=BSBI (B
 */

place_pent(P, I, J, Col, B) :-
  C is Col*I+J+1,
  place_pent(P, C, Col, B).

% Pentmino = 'X'
% Number = 1
%  D
%  C E G
%  F
place_pent('X', C, Col, Board) :-
  arg(C, Board, 'X'),
  D is C-Col+1,
  arg(D, Board, 'X'),
  E is C+1,
  arg(E, Board, 'X'),
  F is E+Col,
  arg(F, Board, 'X'),
  G is E+1,
  arg(G, Board, 'X').

% Pentmino = 'F'
% Number = 1
%  D F
%  C E
%  G
place_pent('F', C, Col, Board) :-
  arg(C, Board, 'F'),
  D is C-Col+1,
  arg(D, Board, 'F'),
  E is C+1,
  arg(E, Board, 'F'),
  F is D+1,
  arg(F, Board, 'F'),
  G is E+Col,
  arg(G, Board, 'F').

% Pentmino = 'F'
% Number = 2
%  D
%  C E F
%  G
place_pent('F', C, Col, Board) :-
  arg(C, Board, 'F'),
  D is C-Col+1,
  arg(D, Board, 'F'),
  E is C+1,
  arg(E, Board, 'F'),
  G is E+Col,
  arg(G, Board, 'F'),
  arg(1, Board, 'F'),
place_pent('F', C, Col, Board) :-
    arg(C, Board, 'F'),
    D is C+(2*Col)-1,
    arg(D, Board, 'F'),
    E is C+Col,
    arg(E, Board, 'F'),
    F is D+1,
    arg(F, Board, 'F'),
    G is E+1,
    arg(G, Board, 'F').

% Pentmino = 'F'
% Number = 4
% C
% D E G
% F

place_pent('F', C, Col, Board) :-
    arg(C, Board, 'F'),
    D is C+Col,
    arg(D, Board, 'F'),
    E is D+1,
    arg(E, Board, 'F'),
    F is E+Col,
    arg(F, Board, 'F'),
    G is E+1,
    arg(G, Board, 'F').

% Pentmino = 'F'
% Number = 5
% C D
% E G
% F

place_pent('F', C, Col, Board) :-
    arg(C, Board, 'F'),
    D is C+1,
    arg(D, Board, 'F'),
    E is D+Col,
    arg(E, Board, 'F'),
    F is E+Col,
    arg(F, Board, 'F'),
    G is E+1,
    arg(G, Board, 'F').

% Pentmino = 'F'
% Number = 6
% E
% C D G
% F

place_pent('F', C, Col, Board) :-
    arg(C, Board, 'F'),
    D is C+1,
arg(D, Board, 'F'),
E is C-Col+2,
arg(E, Board, 'F'),
F is D+Col,
arg(F, Board, 'F'),
G is D+1,
arg(G, Board, 'F').
% Pentmino = 'F'
% Number = 7
%   D
%   C E
%   F G
place_pent('F', C, Col, Board) :-
  arg(C, Board, 'F'),
  D is C-Col+1,
  arg(D, Board, 'F'),
  E is C+1,
  arg(E, Board, 'F'),
  F is E+Col,
  arg(F, Board, 'F'),
  G is F+1,
  arg(G, Board, 'F').
% Pentmino = 'F'
% Number = 8
%   D
%   C F G
place_pent('F', C, Col, Board) :-
  arg(C, Board, 'F'),
  D is C-Col+1,
  arg(D, Board, 'F'),
  E is C+Col,
  arg(E, Board, 'F'),
  F is E+Col,
  arg(F, Board, 'F'),
  G is F+Col,
  arg(G, Board, 'F').
% Pentmino = 'I'
% Number = 1
% C D E F G
place_pent('I', C, Col, Board) :-
  arg(C, Board, 'I'),
  D is C+Col,
  arg(D, Board, 'I'),
  E is D+Col,
  arg(E, Board, 'I'),
  F is E+Col,
  arg(F, Board, 'I'),
  G is F+Col,
  arg(G, Board, 'I').
% Pentmino = 'I'
% Number = 2
% C D E F G
place_pent('I', C, _, Board) :-
    arg(C, Board, 'I'),
    D is C+1,
    arg(D, Board, 'I'),
    E is D+1,
    arg(E, Board, 'I'),
    F is E+1,
    arg(F, Board, 'I'),
    G is F+1,
    arg(G, Board, 'I').

% Pentmino = 'L'
% Number = 1
% C
% D
% E
% F G
place_pent('L', C, Col, Board) :-
    arg(C, Board, 'L'),
    D is C+Col,
    arg(D, Board, 'L'),
    E is D+Col,
    arg(E, Board, 'L'),
    F is E+Col,
    arg(F, Board, 'L'),
    G is F+1,
    arg(G, Board, 'L').

% Pentmino = 'L'
% Number = 2
% C E F G
% D
place_pent('L', C, Col, Board) :-
    arg(C, Board, 'L'),
    D is C+Col,
    arg(D, Board, 'L'),
    E is C+1,
    arg(E, Board, 'L'),
    F is E+1,
    arg(F, Board, 'L'),
    G is F+1,
    arg(G, Board, 'L').

% Pentmino = 'L'
% Number = 3
% C D
% E
% F
% G
place_pent('L', C, Col, Board) :-
    arg(C, Board, 'L'),
    D is C+1,
    arg(D, Board, 'L'),
    E is D+Col,
    arg(E, Board, 'L'),
    F is E+Col,
    arg(F, Board, 'L'),
    G is F+Col,
    arg(G, Board, 'L').

% Pentmino = 'L'
\% Number = 4
\%       F
\% C D E G
place_pent('L', C, Col, Board) :-
    arg(C, Board, 'L'),
    D is C+1,
    arg(D, Board, 'L'),
    E is D+1,
    arg(E, Board, 'L'),
    F is E-Col+1,
    arg(F, Board, 'L'),
    G is E+1,
    arg(G, Board, 'L').
\% Pentmino = 'L'
\% Number = 5
\%   C
\%   D
\%   F
\% E G
place_pent('L', C, Col, Board) :-
    arg(C, Board, 'L'),
    D is C+Col,
    arg(D, Board, 'L'),
    E is D+(2*Col)-1,
    arg(E, Board, 'L'),
    F is D+Col,
    arg(F, Board, 'L'),
    G is E+1,
    arg(G, Board, 'L').
\% Pentmino = 'L'
\% Number = 6
\%   C
\%   D E F G
place_pent('L', C, Col, Board) :-
    arg(C, Board, 'L'),
    D is C+Col,
    arg(D, Board, 'L'),
    E is D+1,
    arg(E, Board, 'L'),
    F is E+1,
    arg(F, Board, 'L'),
    G is F+1,
    arg(G, Board, 'L').
\% Pentmino = 'L'
\% Number = 7
\%   C E
\%   D
\%   F
\% G
place_pent('L', C, Col, Board) :-
    arg(C, Board, 'L'),
    D is C+Col,
    arg(D, Board, 'L'),
    E is C+1,
    arg(E, Board, 'L'),
    F is D+Col,
    arg(F, Board, 'L'),
G is F+Col,
arg(G, Board, 'L').
% Pentmino = 'L'
% Number = 8
% C D E F
% G
place_pent('L', C, Col, Board) :-
  arg(C, Board, 'L'),
  D is C+1,
  arg(D, Board, 'L'),
  E is D+1,
  arg(E, Board, 'L'),
  F is E+1,
  arg(F, Board, 'L'),
  G is F+Col,
  arg(G, Board, 'L').

% Pentmino = 'N'
% Number = 1
% C
% D
% E F
% G
place_pent('N', C, Col, Board) :-
  arg(C, Board, 'N'),
  D is C+Col,
  arg(D, Board, 'N'),
  E is D+Col,
  arg(E, Board, 'N'),
  F is E+1,
  arg(F, Board, 'N'),
  G is F+Col,
  arg(G, Board, 'N').

% Pentmino = 'N'
% Number = 2
% D F G
% C E
place_pent('N', C, Col, Board) :-
  arg(C, Board, 'N'),
  D is C-Col+1,
  arg(D, Board, 'N'),
  E is C+1,
  arg(E, Board, 'N'),
  F is D+1,
  arg(F, Board, 'N'),
  G is F+1,
  arg(G, Board, 'N').

% Pentmino = 'N'
% Number = 3
% C
% D E
% F
% G
place_pent('N', C, Col, Board) :-
  arg(C, Board, 'N'),
  D is C+Col,
  arg(D, Board, 'N'),
  E is D+1,
arg(E, Board, 'N'),
  F is E+Col,
arg(F, Board, 'N'),
  G is F+Col,
arg(G, Board, 'N').
% Pentmino = 'N'
% Number = 4
  E  G
place_pent('N', C, Col, Board) :-
  arg(C, Board, 'N'),
  D is C+1,
  arg(D, Board, 'N'),
  E is C-Col+2,
  arg(E, Board, 'N'),
  F is D+1,
  arg(F, Board, 'N'),
  G is E+1,
  arg(G, Board, 'N').
% Pentmino = 'N'
% Number = 5
  C
% D  G
place_pent('N', C, Col, Board) :-
  arg(C, Board, 'N'),
  D is C+(2*Col)-1,
  arg(D, Board, 'N'),
  E is C+Col,
  arg(E, Board, 'N'),
  F is D+Col,
  arg(F, Board, 'N'),
  G is D+1,
  arg(G, Board, 'N').
% Pentmino = 'N'
% Number = 6
  C  D
% E  F  G
place_pent('N', C, Col, Board) :-
  arg(C, Board, 'N'),
  D is C+1,
  arg(D, Board, 'N'),
  E is D+Col,
  arg(E, Board, 'N'),
  F is E+1,
  arg(F, Board, 'N'),
  G is F+1,
  arg(G, Board, 'N').
% Pentmino = 'N'
% Number = 7
  D
% C  F
% E
% G
place_pent('N', C, Col, Board) :-
  arg(C, Board, 'N'),
D is C-Col+1,
arg(D, Board, 'N'),
E is C+Col,
arg(E, Board, 'N'),
F is C+1,
arg(F, Board, 'N'),
G is E+Col,
arg(G, Board, 'N').

% Pentmino = 'N'
% Number = 8
% C D E
%     F G

place_pent('N', C, Col, Board) :-
  arg(C, Board, 'N'),
  D is C+1,
  arg(D, Board, 'N'),
  E is C+1,
  arg(E, Board, 'N'),
  F is C+1,
  arg(F, Board, 'N'),
  G is E+Col,
  arg(G, Board, 'N').

% Pentmino = 'P'
% Number = 1
% C E
% D G
% F

place_pent('P', C, Col, Board) :-
  arg(C, Board, 'P'),
  D is C+Col,
  arg(D, Board, 'P'),
  E is C+1,
  arg(E, Board, 'P'),
  F is D+Col,
  arg(F, Board, 'P'),
  G is D+1,
  arg(G, Board, 'P').

% Pentmino = 'P'
% Number = 2
% C D F
% E G

place_pent('P', C, Col, Board) :-
  arg(C, Board, 'P'),
  D is C+1,
  arg(D, Board, 'P'),
  E is D+Col,
  arg(E, Board, 'P'),
  F is D+1,
  arg(F, Board, 'P'),
  G is E+1,
  arg(G, Board, 'P').

% Pentmino = 'P'
% Number = 3
% D
% C F
% E G

place_pent('P', C, Col, Board) :-
  arg(C, Board, 'P'),
  D is C+1,
  arg(D, Board, 'P'),
  E is D+1,
  arg(E, Board, 'P'),
  F is D+1,
  arg(F, Board, 'P'),
  G is E+1,
  arg(G, Board, 'P').
arg(C, Board, 'P'),
D is C-(Col-1),
arg(D, Board, 'P'),
E is C+Col,
arg(E, Board, 'P'),
F is C+1,
arg(F, Board, 'P'),
G is E+1,
arg(G, Board, 'P').
% Pentmino = 'P'
% Number = 4
% C E
% D F G
place_pent('P', C, Col, Board) :-
  arg(C, Board, 'P'),
  D is C+Col,
  arg(D, Board, 'P'),
  E is C+1,
  arg(E, Board, 'P'),
  F is D+1,
  arg(F, Board, 'P'),
  G is F+Col,
  arg(G, Board, 'P').
% Pentmino = 'P'
% Number = 5
% C E
% D F
% G
place_pent('P', C, Col, Board) :-
  arg(C, Board, 'P'),
  D is C-(Col-1),
  arg(D, Board, 'P'),
  E is C+1,
  arg(E, Board, 'P'),
  F is D+1,
  arg(F, Board, 'P'),
  G is E+1,
  arg(G, Board, 'P').
% Pentmino = 'P'
% Number = 6
% C E G
% D F
place_pent('P', C, Col, Board) :-
  arg(C, Board, 'P'),
  D is C+(Col-1),
  arg(D, Board, 'P'),
  E is C+1,
  arg(E, Board, 'P'),
  F is D+1,
  arg(F, Board, 'P'),
  G is E+1,
place_pent('P', C, Col, Board) :-
  arg(C, Board, 'P'),
  D is C+Col,
  arg(D, Board, 'P'),
  E is D+Col,
  arg(E, Board, 'P'),
  F is D+1,
  arg(F, Board, 'P'),
  G is E+1,
  arg(G, Board, 'P').
% Pentmino = 'P'
% Number = 8
% C E G
% D F

place_pent('P', C, Col, Board) :-
  arg(C, Board, 'P'),
  D is C+Col,
  arg(D, Board, 'P'),
  E is C+1,
  arg(E, Board, 'P'),
  F is D+1,
  arg(F, Board, 'P'),
  G is E+1,
  arg(G, Board, 'P').
% Pentmino = 'T'
% Number = 1
% C D F
% E
% G

place_pent('T', C, Col, Board) :-
  arg(C, Board, 'T'),
  D is C+1,
  arg(D, Board, 'T'),
  E is D+Col,
  arg(E, Board, 'T'),
  F is D+1,
  arg(F, Board, 'T'),
  G is E+Col,
  arg(G, Board, 'T').
% Pentmino = 'T'
% Number = 2
% E
% C D F
% G

place_pent('T', C, Col, Board) :-
  arg(C, Board, 'T'),
  D is C+1,
  arg(D, Board, 'T'),
  E is C-(Col-2),
  arg(E, Board, 'T'),
  F is D+1,
  arg(F, Board, 'T'),
  G is F+Col,
  arg(G, Board, 'T').
% Pentmino = 'T'
% Number = 3
% C
place_pent('T', C, Col, Board) :-
    arg(C, Board, 'T'),
    D is C+(Col*2)-1,
    arg(D, Board, 'T'),
    E is C+Col,
    arg(E, Board, 'T'),
    F is D+1,
    arg(F, Board, 'T'),
    G is F+1,
    arg(G, Board, 'T').

place_pent('U', C, Col, Board) :-
    arg(C, Board, 'U'),
    D is C+Col,
    arg(D, Board, 'U'),
    E is D+1,
    arg(E, Board, 'U'),
    F is C+2,
    arg(F, Board, 'U'),
    G is E+1,
    arg(G, Board, 'U').

place_pent('U', C, Col, Board) :-
    arg(C, Board, 'U'),
    D is C+Col,
    arg(D, Board, 'U'),
    E is C+1,
    arg(E, Board, 'U'),
    F is D+Col,
    arg(F, Board, 'U'),
    G is F+1,
    arg(G, Board, 'U').
% Number = 3
% C E F
% D   G
place_pent('U', C, Col, Board) :-
arg(C, Board, 'U'),
D is C+Col,
arg(D, Board, 'U'),
E is C+1,
arg(E, Board, 'U'),
F is E+1,
arg(F, Board, 'U'),
G is D+2,
arg(G, Board, 'U').

% Pentmino = 'U'
% Number = 4
% C D
%   F
% E G
place_pent('U', C, Col, Board) :-
arg(C, Board, 'U'),
D is C+1,
arg(D, Board, 'U'),
E is C+(2*Col),
arg(E, Board, 'U'),
F is D+Col,
arg(F, Board, 'U'),
G is E+1,
arg(G, Board, 'U').

% Pentmino = 'U'
% Number = 1
% C
% D
% E F G
place_pent('V', C, Col, Board) :-
arg(C, Board, 'V'),
D is C+Col,
arg(D, Board, 'V'),
E is D+Col,
arg(E, Board, 'V'),
F is E+1,
arg(F, Board, 'V'),
G is F+1,
arg(G, Board, 'V').

% Pentmino = 'V'
% Number = 2
% C E G
% D
% F
place_pent('V', C, Col, Board) :-
arg(C, Board, 'V'),
D is C+Col,
arg(D, Board, 'V'),
E is C+1,
arg(E, Board, 'V'),
F is D+Col,
arg(F, Board, 'V'),
G is E+1,
arg(G, Board, 'V').
% Pentmino = 'V'
% Number = 3
% C D E
%     F
%     G
place_pent('V', C, Col, Board) :-
  arg(C, Board, 'V'),
  D is C+1,
  arg(D, Board, 'V'),
  E is D+1,
  arg(E, Board, 'V'),
  F is E+Col,
  arg(F, Board, 'V'),
  G is F+Col,
  arg(G, Board, 'V').
% Pentmino = 'V'
% Number = 4
%     D
%     F
%     C E G
place_pent('V', C, Col, Board) :-
  arg(C, Board, 'V'),
  D is C-(2*Col)+2,
  D > 0,  %%% arg/3
  arg(D, Board, 'V'),
  E is C+1,
  arg(E, Board, 'V'),
  F is D+Col,
  arg(F, Board, 'V'),
  G is E+1,
  arg(G, Board, 'V').
% Pentmino = 'W'
% Number = 1
% C
% D E
%     F G
place_pent('W', C, Col, Board) :-
  arg(C, Board, 'W'),
  D is C-Col+1,
  arg(D, Board, 'W'),
  E is D+1,
  arg(E, Board, 'W'),
  F is E+Col,
  arg(F, Board, 'W'),
  G is F+1,
  arg(G, Board, 'W').
% Pentmino = 'W'
% Number = 2
% D G
% C F
% E
place_pent('W', C, Col, Board) :-
  arg(C, Board, 'W'),
  D is C-(Col-1),
  arg(D, Board, 'W'),
  E is C+Col,
arg(E, Board, 'W'),
F is C+1,
arg(F, Board, 'W'),
G is D+1,
arg(G, Board, 'W').
% Pentmino = 'W'
% Number = 3
% C D
%    E F
%      G
place_pent('W', C, Col, Board) :-
  arg(C, Board, 'W'),
  D is C+1,
  arg(D, Board, 'W'),
  E is D+Col,
  arg(E, Board, 'W'),
  F is E+1,
  arg(F, Board, 'W'),
  G is F+Col,
  arg(G, Board, 'W').
% Pentmino = 'W'
% Number = 4
%    E
%  D G
% C F
place_pent('W', C, Col, Board) :-
  arg(C, Board, 'W'),
  D is C-(Col-1),
  arg(D, Board, 'W'),
  E is D-(Col-1),
  arg(E, Board, 'W'),
  F is C+1,
  arg(F, Board, 'W'),
  G is D+1,
  arg(G, Board, 'W').
% Pentmino = 'Y'
% Number = 1
%    E
%  C D F G
place_pent('Y', C, Col, Board) :-
  arg(C, Board, 'Y'),
  D is C+1,
  arg(D, Board, 'Y'),
  E is C-(Col-2),
  arg(E, Board, 'Y'),
  F is D+1,
  arg(F, Board, 'Y'),
  G is F+1,
  arg(G, Board, 'Y').
% Pentmino = 'Y'
% Number = 2
%    C
%    D
%    E G
%    F
place_pent('Y', C, Col, Board) :-
  arg(C, Board, 'Y'),
D is C+Col,
arg(D, Board, 'Y'),
E is D+Col,
arg(E, Board, 'Y'),
F is E+Col,
arg(F, Board, 'Y'),
G is E+1,
arg(G, Board, 'Y').
% Pentmino = 'Y'
% Number = 3
% C D F G

place_pent('Y', C, Col, Board) :-
arg(C, Board, 'Y'),
D is C+1,
arg(D, Board, 'Y'),
E is D+Col,
arg(E, Board, 'Y'),
F is D+1,
arg(F, Board, 'Y'),
G is F+1,
arg(G, Board, 'Y').
% Pentmino = 'Y'
% Number = 4
% D
% C E
% F
% G

place_pent('Y', C, Col, Board) :-
arg(C, Board, 'Y'),
D is C-(Col-1),
arg(D, Board, 'Y'),
E is C+1,
arg(E, Board, 'Y'),
F is E+Col,
arg(F, Board, 'Y'),
G is F+Col,
arg(G, Board, 'Y').
% Pentmino = 'Y'
% Number = 5
% C
% D F
% E

place_pent('Y', C, Col, Board) :-
arg(C, Board, 'Y'),
D is C+Col,
arg(D, Board, 'Y'),
E is D+Col,
arg(E, Board, 'Y'),
F is D+1,
arg(F, Board, 'Y'),
G is E+Col,
arg(G, Board, 'Y').
% Pentmino = 'Y'
% Number = 6
% C D E G
place_pent('Y', C, Col, Board) :-
  arg(C, Board, 'Y'),
  D is C+1,
  arg(D, Board, 'Y'),
  E is D+1,
  arg(E, Board, 'Y'),
  F is E+Col,
  arg(F, Board, 'Y'),
  G is E+1,
  arg(G, Board, 'Y').
% Pentmino = 'Y'
% Number = 7
%   C
%   E
% D F
%   G

place_pent('Y', C, Col, Board) :-
  arg(C, Board, 'Y'),
  D is C+(2*Col)-1,
  arg(D, Board, 'Y'),
  E is C+Col,
  arg(E, Board, 'Y'),
  F is D+1,
  arg(F, Board, 'Y'),
  G is F+Col,
  arg(G, Board, 'Y').
% Pentmino = 'Y'
% Number = 8
%   D
% C E F G

place_pent('Y', C, Col, Board) :-
  arg(C, Board, 'Y'),
  D is C-(Col-1),
  arg(D, Board, 'Y'),
  E is C+1,
  arg(E, Board, 'Y'),
  F is E+1,
  arg(F, Board, 'Y'),
  G is F+1,
  arg(G, Board, 'Y').
% Pentmino = 'Z'
% Number = 1
%   C D
%    E
%  F G

place_pent('Z', C, Col, Board) :-
  arg(C, Board, 'Z'),
  D is C+1,
  arg(D, Board, 'Z'),
  E is D+Col,
  arg(E, Board, 'Z'),
  F is E+Col,
  arg(F, Board, 'Z'),
  G is F+1,
  arg(G, Board, 'Z').
% Pentmino = 'Z'
place_pent('Z', C, Col, Board) :-
    arg(C, Board, 'Z'),
    D is C+Col,
    arg(D, Board, 'Z'),
    E is C+1,
    arg(E, Board, 'Z'),
    F is C-(Col-2),
    arg(F, Board, 'Z'),
    G is E+1,
    arg(G, Board, 'Z').

place_pent('Z', C, Col, Board) :-
    arg(C, Board, 'Z'),
    D is C+(2*Col)-1,
    arg(D, Board, 'Z'),
    E is C+Col,
    arg(E, Board, 'Z'),
    F is C+1,
    arg(F, Board, 'Z'),
    G is D+1,
    arg(G, Board, 'Z').

place_pent('Z', C, Col, Board) :-
    arg(C, Board, 'Z'),
    D is C+Col,
    arg(D, Board, 'Z'),
    E is D+1,
    arg(E, Board, 'Z'),
    F is E+1,
    arg(F, Board, 'Z'),
    G is F+Col,
    arg(G, Board, 'Z').

% Utilities
for(M, M, N) :- M =< N.
for(I, M, N) :- M =< N, M₁ is M + 1, for(I, M₁, N).
pent_select(X, [X|Xs], Xs).
pent_select(X, [Y|Ys], [Y|Zs]) :- pent_select(X, Ys, Zs).

read_int(N) :-
    skip_spaces(C),
readint(C, N).

readint(C, N) :-
  C =:= "-", !,
  get0(C1),
  readint1(C1, 0, N1),
  N is -N1.
readint(C, N) :-
  readint1(C, 0, N).

readint1(C, N0, N) :-
  C >= "0", C =< "9",
  !,
  N1 is 10*N0+C-'0',
  get0(C1),
  readint1(C1, N1, N).
readint1(10, N, N) :- !.   % end of line
readint1(_, N, N) :- !,
  skip(10).    % skip to end of line

skip_spaces(C) :- get0(C0), skip_spaces1(C0, C).

skip_spaces1(9, C) :- !, skip_spaces(C).
skip_spaces1(10, C) :- !, skip_spaces(C).
skip_spaces1(12, C) :- !, skip_spaces(C).
skip_spaces1(13, C) :- !, skip_spaces(C).
skip_spaces1(32, C) :- !, skip_spaces(C).
skip_spaces1(C, C).

read_line(Str) :-
  get0(C),
  readline(C, Str).
readline(10, []) :- !.   % LF means end_of_line
readline(13, Str) :- !, read_line(Str). % CR is ignored
readline(C, [C|Str]) :- !, read_line(Str).

/**************************
Count All Solutions
**************************/
count_all(N) :-
  findall(B, solve_pentomino(N, B), L),
  length(L, C),
  write(C), write(' solutions'), nl.

/**************************
For applet
**************************/
pentomino_applet(X, B) :-
  solve_pentomino(X, B0),
  remove_asterisk(B0, B).

remove_asterisk(B0, B) :-
  B0 =.. [[]|As],
  rm_aster(As, B).
rm_aster([], []) :- !.
rml_aster([A|As], Ps) :- A == '*', !,
    rm_aster(As, Ps).
rml_aster([A|As], [A|Ps]) :-
    rm_aster(As, Ps).