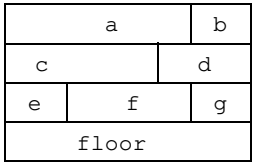
Example: Unstacking blocks

Consider a stack of blocks, each of which is uniquely labeled with a letter:



This arrangement could be represented with these facts:

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

Problem: Define a predicate `clean` that will print a sequence of blocks to remove from the floor such that no block is removed until nothing is on it.

A suitable sequence of removals for the above diagram is: `a`, `c`, `e`, `b`, `d`, `f`, `g`. Another is `a`, `b`, `c`, `d`, `e`, `f`, `g`.

Unstacking blocks, continued

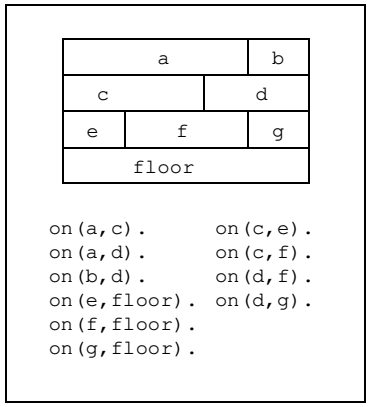
One solution: (`blocks.pl`)

```
removable(B) :- \+on(_,B).

remove(B) :-
    removable(B),
    retractall(on(B,_)),
    write('Remove '), write(B), nl.

remove(B) :-
  on(Above,B), remove(Above), remove(B).

clean :- on(B,floor), remove(B), clean.
clean.
```



Important: If we want to be able to `assert` and `retract` facts of a predicate defined in a file, we must use the `dynamic` directive:

```
% cat blocks.pl
:-dynamic(on/2).
on(a,c). on(a,d).
...
```


Unstacking blocks, continued

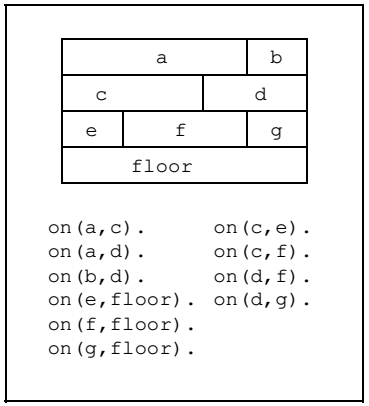
A more concise solution: (`blocks2.pl`)

```
clean :-
    on(A,_), \+on(_,A),
    write('Remove '), write(A), nl,
    retractall(on(A,_)), clean.
clean.
```

Output:

```
?- clean.
Remove a
Remove b
Remove c
Remove d
Remove e
Remove f
Remove g

Yes
```



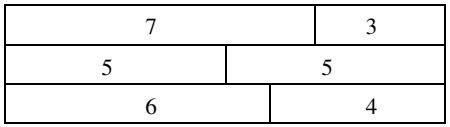
Puzzle solving

Brick laying

The Zebra Puzzle

Example: brick laying

Consider six bricks of lengths 7, 5, 6, 4, 3, and 5. One way they can be laid into three rows of length 10 is like this:



Problem: Write a predicate `laybricks/3` that produces a suitable sequence of bricks for three rows of a given length:

```
?- laybricks([7,5,6,4,3,5], 10, Rows).
Rows = [[7, 3], [5, 5], [6, 4]]

?- laybricks([7,5,6,4,3,5], 12, Rows).
No
```

In broad terms, how can we approach this problem?

Brick laying, continued

Here is a helper predicate `getone(X, List, Remaining)` that produces in `Remaining` a copy of `List` with `X` removed:

```
getone(X, [X|T], T).
getone(X, [H|T], [H|N]) :- getone(X, T, N).
```

Usage:

```
?- getone(X,[a,b,a,d],R).
X = a
R = [b, a, d] ;

X = b
R = [a, a, d] ;

X = a
R = [a, b, d] ;

X = d
R = [a, b, a] ;
```

What is the result of `?- getone(a, [a,b,a], R).`?

Brick laying, continued

`layrow` produces a sequence of bricks for a row of a given length:

```
layrow(Bricks, 0, Bricks, []).

layrow(Bricks, RowLen, Left, [Brick|MoreBricksForRow]) :-
   getone(Brick, Bricks, Left0),
   RemLen is RowLen - Brick, RemLen >= 0,
   layrow(Left0, RemLen, Left, MoreBricksForRow).
```

Usage:

```
?- layrow([3,2,7,4], 7, BricksLeft, Row).
BricksLeft = [2, 7]
Row = [3, 4] ;

BricksLeft = [3, 2, 4]
Row = [7] ;

BricksLeft = [2, 7]
Row = [4, 3] ;
No
```

What were the intermediate steps to arrive at the first row cited?

Brick laying, continued

Now we can write `laybricks`, which is hardwired for three rows:

```
laybricks(Bricks, RowLen, [Row1, Row2, Row3]) :-
   layrow(Bricks, RowLen, LeftAfter1, Row1),
   layrow(LeftAfter1, RowLen, LeftAfter2, Row2),
   layrow(LeftAfter2, RowLen, LeftAfter3, Row3),
   LeftAfter3 = [].
```

Usage:

```
?- laybricks([2,1,3,1,2], 3, Rows).
Rows = [[2, 1], [3], [1, 2]] ;

Rows = [[2, 1], [3], [2, 1]] ;

Rows = [[2, 1], [1, 2], [3]] ;

...many more...
```

What is the effect of `LeftAfter3 = []`?

Brick laying, continued

A more general predicate, to lay `Nrows` rows of length `RowLen`:

```
laybricks2([], 0, _, []).

laybricks2(Bricks, Nrows, RowLen, [Row|Rows]) :-
        layrow(Bricks, RowLen, BricksLeft, Row),
        RowsLeft is Nrows - 1,
        laybricks2(BricksLeft, RowsLeft, RowLen, Rows).
```

Usage:

```
?- laybricks2([5,1,6,2,3,4,3], 3, 8, Rows).
Rows = [[5, 3], [1, 4, 3], [6, 2]]

?- laybricks2([5,1,6,2,3,4,3], 8, 3, Rows).
No

?- laybricks2([5,1,6,2,3,4,3], 2, 12, Rows).
Rows = [[5, 1, 6], [2, 3, 4, 3]]

?- laybricks2([5,1,6,2,3,4,3], 4, 6, Rows).
Rows = [[5, 1], [6], [2, 4], [3, 3]]
```

As is, `laybricks2` requires that all bricks be used. How can we remove that requirement?

Brick laying continued

Some facts for testing:

```
b(3, [8,5,1,4,6,6,2,3,4,3,3,6,3,8,6,4]).    % 6x12
b(4, [8,5,1,4,6,6,2,3,4,3,3,6,3,8,6,4,1]). % 6x12 with extra 1
```

Usage:

```
?- b(3,Bricks), laybricks2(Bricks,6,12,Rows).
Bricks = [8, 5, 1, 4, 6, 6, 2, 3, 4|...]
Rows = [[8, 1, 3], [5, 4, 3], [6, 6], [2, 4, 3, 3], [6, 6], [8,
4]]
Yes
```

Is the `getone` predicate really necessary? Or could we just use `permute/2` and try all combinations of bricks?

Brick laying, continued

Let's try a set of bricks that can't be laid into six rows of twelve:

```
?- b(4,Bricks), laybricks2(Bricks, 6, 12, Rows).
...[the sound of a combinatorial explosion]...
^C
Action (h for help) ? a
% Execution Aborted

?- statistics.
8.05 seconds cpu time for 25,594,610 inferences
2,839 atoms, 1,854 functors, 1,451 predicates, 23 modules,
35,904 VM-codes
```

The speed of a Prolog implementation is sometimes quoted in LIPS—logical inferences per second.

The "Zebra Puzzle"

The Wikipedia entry for "Zebra Puzzle" presents a puzzle said to have been first published in the magazine *Life International* on December 17, 1962. Here are the facts:

There are five houses.
The Englishman lives in the red house.
The Spaniard owns the dog.
Coffee is drunk in the green house.
The Ukrainian drinks tea.
The green house is immediately to the right of the ivory house.
The Old Gold smoker owns snails.
Kools are smoked in the yellow house.
Milk is drunk in the middle house.
The Norwegian lives in the first house.
The man who smokes Chesterfields lives in the house next to the man with the fox.
Kools are smoked in the house next to the house where the horse is kept.
The Lucky Strike smoker drinks orange juice.
The Japanese smokes Parliaments.
The Norwegian lives next to the blue house.

The article asked readers, "Who drinks water? Who owns the zebra?"

The Zebra Puzzle, continued

We can solve this problem creating a set of goals and asking Prolog to find the condition under which all the goals are true.

A good starting point is these two facts:

The Norwegian lives in the first house.
Milk is drunk in the middle house.

That information can be represented as this goal:

```
Houses = [house(norwegian, _, _, _, _),   % First house
          _,
          house(_, _, _, milk, _),        % Middle house
          _, _]
```

Note that there are five variables: nationality, pet, smoking preference (remember, it was 1962!), beverage of choice and house color. We'll use instances of `house` structures to represent knowledge about a house. Anonymous variables are used to represent "don't-knows".

The Zebra Puzzle, continued

Some facts can be represented with goals that specify structures as members of `Houses` with unknown position:

The Englishman lives in the red house.

```
member(house(englishman, _, _, _, red), Houses)
```

The Spaniard owns the dog.

```
member(house(spaniard, dog, _, _, _), Houses)
```

Coffee is drunk in the green house.

```
member(house(_, _, _, coffee, green), Houses)
```

How can we represent *The green house is immediately to the right of the ivory house.*?

The Zebra Puzzle, continued

At hand:

The green house is immediately to the right of the ivory house.

Here's a predicate that expresses left/right positioning:

```
left_right(L, R, [L, R | _]).
left_right(L, R, [_ | Rest]) :- left_right(L, R, Rest).
```

Testing:

```
?- left_right(Left,Right,  [1,2,3,4]).
Left = 1
Right = 2 ;

Left = 2
Right = 3 ;
...
```

Goal:

```
left_right(house(_, _, _, _, ivory),
           house(_, _, _, _, green), Houses)
```

The Zebra Puzzle, continued

Remaining facts:

The man who smokes Chesterfields lives in the house next to the man with the fox.
Kools are smoked in the house next to the house where the horse is kept.
The Norwegian lives next to the blue house.

How can we represent these?

The Zebra Puzzle, continued

The man who smokes Chesterfields lives in the house next to the man with the fox.
Kools are smoked in the house next to the house where the horse is kept.
The Norwegian lives next to the blue house.

We can say that two houses are next to each other if one is immediately left or right of the other:

```
next_to(X, Y, List) :- left_right(X, Y, List).
next_to(X, Y, List) :- left_right(Y, X, List).
```

More goals:

```
next_to(house(_, _, chesterfield, _, _),
        house(_, fox, _, _, _), Houses)

next_to(house(_, _, kool, _, _),
        house(_, horse, _, _, _), Houses)

next_to(house(norwegian, _, _, _, _),
        house(_, _, _, _, blue), Houses)
```


The Zebra Puzzle, continued

A rule that comprises all the goals developed thus far:

```
zebra(Houses, Zebra_Owner, Water_Drinker) :-
  Houses = [house(norwegian, _, _, _, _), _,
            house(_, _, _, milk, _), _, _],
  member(house(englishman, _, _, _, red), Houses),
  member(house(spaniard, dog, _, _, _), Houses),
  member(house(_, _, _, coffee, green), Houses),
  member(house(ukrainian, _, _, tea, _), Houses),
  left_right(house(_,_,_,_,ivory), house(_,_,_,_,green), Houses),
  member(house(_, snails, old_gold, _, _), Houses),
  member(house(_, _, kool, _, yellow), Houses),
  next_to(house(_,_,chesterfield,_,_),house(_, fox,_,_,_), Houses),
  next_to(house(_,_,kool,_,_), house(_, horse, _, _, _), Houses),
  member(house(_, _, lucky_strike, orange_juice, _), Houses),
  member(house(japanese, _, parliment, _, _), Houses),
  next_to(house(norwegian,_,_,_,_), house(_,_,_,_, blue), Houses),
  member(house(Zebra_Owner, zebra, _, _, _), Houses),
  member(house(Water_Drinker, _, _, water, _), Houses).
```

Note that the last two goals ask the questions of interest.

The Zebra Puzzle, continued

The moment of truth:

```
?- zebra(_, Zebra_Owner, Water_Drinker).
Zebra_Owner = japanese
Water_Drinker = norwegian ;
No
```

The whole neighborhood:

```
?- zebra(Houses,_,_), member(H,Houses), writeln(H), fail.
house(norwegian, fox, kool, water, yellow)
house(ukrainian, horse, chesterfield, tea, blue)
house(englishman, snails, old_gold, milk, red)
house(spaniard, dog, lucky_strike, orange_juice, ivory)
house(japanese, zebra, parliment, coffee, green)

No
```

Credit: The code above was adapted from http://sandbox.rulemaker.net/ngps/119, by Ng Pheng Siong, who in turn apparently adapted it from work by Bill Clementson in Allegro Prolog.

Parsing and grammars

A very simple grammar

A very simple parser

Grammar rule notation

Credit: This section borrows heavily from chapter 12 in Covington, et al.

A very simple grammar

Here is a grammar for a very simple language:

```
Sentence => Article Noun Verb

Article    => the | a

Noun       => dog | cat | girl | boy

Verb       => ran | talked | slept
```

Here are some sentences in the language:

```
the dog ran

a boy slept

the cat talked
```


A very simple parser

Here is a simple parser for the grammar: (`parser0.pl`)

```
sentence(Words) :-
    article(Words, Left0), noun(Left0, Left1), verb(Left1, []).

article([the| Left], Left). article([a| Left],  Left).
noun([Noun| Left], Left) :- member(Noun, [dog,cat,girl,boy]).
verb([Verb|Left], Left) :- member(Verb, [ran,talked,slept]).
```

Usage:

```
?- sentence([the,dog,ran]).
Yes

?- sentence([the,dog,boy]).
No

?- sentence(S). % Generates all valid sentences
S = [the, dog, ran] ;
S = [the, dog, talked]
...
```


A very simple parser, continued

For reference:

```
sentence(Words) :-
    article(Words, Left0), noun(Left0, Left1), verb(Left1, []).

article([the|Left], Left). article([a| Left],  Left).
noun([Noun|Left], Left) :- member(Noun, [dog,cat,girl,boy]).
verb([Verb|Left], Left) :- member(Verb, [ran,talked,slept]).
```

Note that the heads for `article`, `noun`, and `verb` all have the same form.

Let's look at a clause for `article` and a unification:

```
article([the|Left], Left).

?- article([the,dog,ran], Remaining).
Remaining = [dog, ran]
```

If `Words` begins with `the` or `a`, then `article(Words, Remaining)` succeeds and unifies `Remaining` with the rest of the list. **The key idea: `article`, `noun`, and `verb` each consume an expected word and produce what's left.**

A very simple parser, continued

```
sentence(Words) :-
    article(Words, Left0), noun(Left0, Left1), verb(Left1, []).
```

A query sheds light on how `sentence` operates:

```
?- article(Words, Left0), noun(Left0, Left1),
    verb(Left1, Left2), Left2 = [].

Words = [the, dog, ran]
Left0 = [dog, ran]
Left1 = [ran]
Left2 = []

?- sentence([the,dog,ran]).
Yes
```

Each goal consumes one word. The remainder is then the input for the next goal.

Why is `verb`'s result, `Left2`, unified with the empty list?

A very simple parser, continued

For convenient use, let's add a predicate `s/1` that uses `concat_atom` to split up a string based on spaces:

```
s(String) :-
    concat_atom(Words, ' ', String), sentence(Words).

sentence(Words) :-
    article(Words, Left0), noun(Left0, Left1),
    verb(Left1, []).
```

Usage:

```
?- s('the dog ran').
Yes

?- s('ran the dog').
No
```


Grammar rule notation

Prolog's *grammar rule notation* provides a convenient way to express these stylized rules. Instead of this,

```
sentence(Words) :-
    article(Words, Left0), noun(Left0, Left1), verb(Left1, []).
article([the| Left], Left).
article([a| Left], Left).
noun([Noun| Left], Left) :- member(Noun, [dog,cat,girl,boy]).
verb([Verb|Left], Left) :- member(Verb, [ran,talked,slept]).
```

we can take advantage of grammar rule notation and say this,

```
sentence --> article, noun, verb.
article --> [a]; [the].
noun --> [dog]; [cat]; [girl]; [boy].
verb --> [ran]; [talked]; [slept].
```

Note that the literals are specified as singleton lists.

The semicolon is an "or". Alternative: `noun --> [dog]. noun --> [cat]. ...`

Grammar rule notation, continued

```
% cat parser1.pl
sentence --> article, noun, verb.
article --> [a]; [the].
noun --> [dog]; [cat]; [girl]; [boy].
verb --> [ran]; [talked]; [slept].
```

`listing` can be used to see the clauses generated by Prolog:

```
 ?- listing(sentence).
sentence(A, D) :-  article(A, B), noun(B, C), verb(C, D).

?- listing(article).
article(A, B) :-
        (   'C'(A, a, B)
        ;   'C'(A, the, B)
        ).

?- listing('C').
'C'([A|B], A, B).
```

Note that the predicates generated for `sentence`, `article` and others have an arity of 2.

Grammar rule notation, continued

```
% cat parser1a.pl
s(String) :-
    concat_atom(Words, ' ', String), sentence(Words,[]).

sentence --> article, noun, verb.
article --> [a]; [the].
noun --> [dog]; [cat]; [girl]; [boy].

verb --> [ran]; [talked]; [slept].
verb([Verb|Left], Left) :- verb0(Verb).

verb0(jumped). verb0(ate). verb0(computed).
```

Points to note:

`sentence`, `article`, `verb`, and `noun` are known as *non-terminals*. `dog`, `cat`, `ran`, talked, etc. are known as *terminals*.

The call to `sentence` in `s` now has two terms to match the generated rule.

`verb` has both a grammar rule and an ordinary rule. The latter makes use of `verb0` facts.