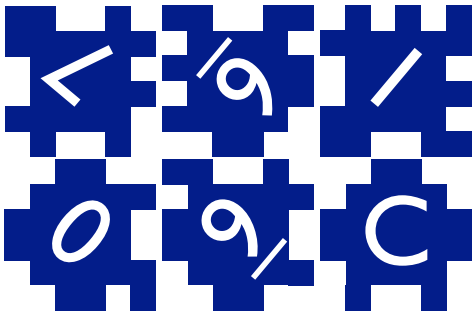
Prolog Assessed Exercise

Lecturer and Assessor: David Evans <david.evans@cl.cam.ac.uk>

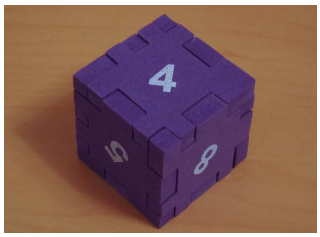
21st November 2011

This exercise is about foam[1] puzzles. Here's one:



Each piece is labelled for ease of identification, a label consisting of what is written on the piece's two sides. In this case the labels of these pieces, clockwise from the top left, are 74, 65, 13, Cc, 98, and 02.

The goal is to assemble these six pieces, by interlocking the "fingers" that are on each edge, to form a cube. The result looks something like this



In this solution to the puzzle, the sides of the pieces that show 7, 6, 1, C, 9, and 0 are on the inside of the cube. This is why 4, 5, and 8 are visible in the picture; the invisible three exterior faces show 3, 2, and c.

You will develop a Prolog program to solve these sorts of puzzles.

You should work through and complete all steps of this document. Ensure that each implemented predicate behaves correctly under backtracking. You may make reasoned use of the cut operator where appropriate, although if you find yourself using it a lot then you are on the wrong track—a correct solution exists that does not use cut at all. Frequently we will need to represent true and false. By convention we shall use `0` for false and `1` for true.

The date above indicates the document version. Check the subject web page for updates. The changes made between document versions will be listed on the subject web page. Details about submission and marking are at the end of this document. This document consists of 8 pages.

[1] Of course they don't *need* to be made of foam but often they are.

1 Representing the puzzle

In common with the puzzles that we have discussed in lectures, the first task is to devise a representation of the problem. Doing this requires encoding the individual pieces and how they may be placed.

Consider piece 74 in the above diagram. This piece has 4 sides, which we denote 0 through 3 starting at the top and counting clockwise. Each side has six fingers. We start in the upper left corner (which is side 0) and write a 1 if a finger is present and a 0 if it is not. Side 0 can therefore be represented by the list `[1,1,0,0,1,0]`. Continuing clockwise, side 1 corresponds to the list `[0,1,0,1,0,0]` and so on for sides 2 and 3.

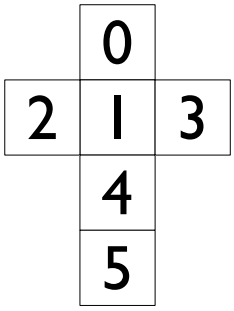
In general, we encode each piece as a list. The first element is a label for the piece. The second element of the piece data structure is a list of edges, with edge 0 coming first, edge 1 next, and so on.

Based on this, the puzzle above would be encoded as follows:

```
['74', [[1,1,0,0,1,0], [0,1,0,1,0,0], [0,1,0,0,1,0], [0,1,0,0,1,1]]]
['65', [[1,1,0,0,1,1], [1,0,1,1,0,0], [0,0,1,1,0,0], [0,1,0,1,0,1]]]
['13', [[0,1,0,1,0,1], [1,1,0,1,0,1], [1,1,0,0,1,1], [1,1,0,0,1,0]]]
['Cc', [[0,0,1,1,0,0], [0,0,1,1,0,0], [0,1,0,0,1,0], [0,0,1,1,0,0]]]
['98', [[1,1,0,0,1,0], [0,1,0,0,1,0], [0,0,1,1,0,0], [0,0,1,1,0,1]]]
['02', [[0,0,1,1,0,0], [0,1,0,0,1,1], [1,0,1,1,0,0], [0,0,1,1,0,0]]]
```

(The labels here were chosen to correspond to the physical puzzle used for the photo.) We say that these pieces are in their *canonical* orientations, where the first edge is at the top (edge 0), the next is on the right-hand side (edge 1), and so on.

Alongside the pieces we have to encode a configuration of the pieces. The cube we are trying to construct has six sides, so we label those sides as follows:



A candidate solution to the puzzle consists of filling each of these six slots with a piece that is flipped or not and is in a specific orientation. A solution is valid if adjacent pieces fit together, so the bottom edge of what is in slot 0 has to plug into the top edge of what is in slot 1 and, similarly, the left edge of what is in slot 2 has to plug into the left edge of what is in slot 5.

2 Library predicates

Your solution will be much simpler if you use some SWI-Prolog predicates that were not discussed in lectures. These are explained in this section and part of the exercise is learning how to use them. That said, none should be mysterious.

2.1 bagof/3

The `bagof(+Template, :Goal, -Bag)` predicate gathers all solutions to `Goal`. You have not seen the notation `:Goal` before; it indicates that `Goal` is a "meta-argument", for our purposes meaning that it contains some Prolog code. `bagof/3` works as follows. `Goal` will be solved repeatedly, each resulting in a binding for `Template`. `Bag` will be unified with a list of these bindings. Recall from lectures that the `perm/2` predicate computes list permutations. We can use `bagof/3` to gather all the permutations of the list `[1, 2, 3]` as follows:

```
?- bagof(P, perm([1, 2, 3], P), Ps).
Ps = [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2],
      [3, 2, 1]].
```

2.2 lists:reverse/2

`lists:reverse(?A, ?B)` is true iff[2] the elements in list `A` are in reverse order compared to list `B`. For example,

```
?- lists:reverse([1, 2, 3], [1, 3, 2]).
false.

?- lists:reverse([1, 2, 3], B).
B = [3, 2, 1].
```

`lists:reverse/2` is in the `lists` library module, so you should include

```
:- use_module(library(lists)).
```

in your source file.

2.3 maplist/3

The `maplist` family of predicates makes it easy to apply a given predicate to all the elements of a list. The one with arity 3, `maplist(:Goal, ?A, ?B)`, is true iff `Goal` can be successfully applied to successive elements of the lists `A` and `B`. You can think of the call

```
maplist(pred, [A1, A2, A3], [B1, B2, B3])
```

being translated to

```
pred(A1, B1), pred(A2, B2), pred(A3, B3)
```

As an example, in lectures we talked about how the > operator is really the predicate `>(+A, +B)`, true iff `A` is numerically greater than `B`:

```
?- >(3,4).
false.

?- >(9,4).
true.
```

[2] "if and only if"

Using `maplist/3` we can check whether every element in one list is great than its corresponding element in another

```
?- maplist(>, [1,2,3], [4,1,2]).
false.

?- maplist(>, [5,2,3], [4,1,2]).
true.
```

`maplist/3` is in the `apply` library module, so you should include

```
:- use_module(library(apply)).
```

in your source file.

3 Supporting predicates

3.1 Piece generation

Using the above representation for pieces, write a predicate `piece(?P)` that is true iff the piece `P` is an encoded piece known to Prolog. Your predicate may consist of a simple set of facts.

Use `bagof/3` to produce a list of the puzzle pieces known to Prolog and use the `length/2` predicate in a test, included in your source file, that there are in fact 6 (six) of them.

3.2 Rotating lists

In the lectures we discussed `rotate(?A, ?B)` which is true iff the list `B` is the list `A` rotated left by one element. Write an enhanced version of this, `rotate(+A, +N, ?B)`, which is true iff list `B` is the list `A` rotated left by `N` elements. Include comments in your source file that explain how your predicate works and tests that demonstrate that it is correct.

3.3 Exclusive-OR

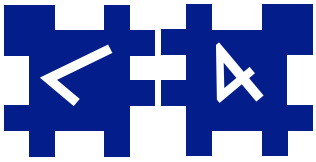
Write a predicate `xor(?A, ?B)` which is true iff `A xor B`. Remember that `A` exclusive-OR (`xor`) `B` iff either `A` or `B` is true (`1` in our context) but not both. (As a hint, this really is as easy as it seems because of Prolog's closed-world assumption.)

3.4 Number ranges

In the course of building your solution you will need to generate integer values within a range. This was visited in your supervision problems. Implement the predicate `range(+Min, +Max, -Val)` which unifies `Val` with `Min` on the first evaluation and then all values up to `Max - 1` on backtracking.

4 Piece orientation

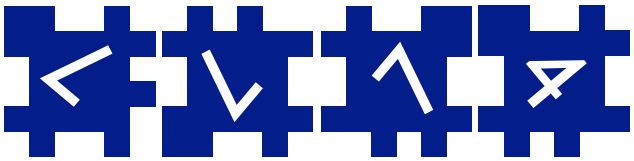
Suppose that a piece is in its canonical orientation as defined in section 1. If we flip it along its vertical axis, we end up with a new piece. This looks like the following for piece 74



Write a predicate `flipped(+P, ?FP)` which is true iff piece `FP` is piece `P` flipped. The flipped piece doesn't have a new label—it is still piece 74, for example, after all. Ensure that your predicate can *generate* flipped pieces as well as check them. Include in your source file comments describing how your predicate works.

We consider a piece to be in orientation `Or` if starting in its canonical orientation we rotate it `Or` times *anticlockwise*. Therefore the canonical orientation is orientation `0`. By convention we say that a piece is in orientation `-Or` if that piece is placed in its canonical orientation, flipped (according to the scheme described above), and then rotated `Or` times anticlockwise. Make sure that you are comfortable with the fact that a piece in orientation `-Or` does *not* lead to the same shape as that piece rotated `Or` times clockwise!

Here is piece 74 in, from left to right, orientations 0, 1, 3, and -1.



Write a predicate `orientation(+P, ?O, -OP)` which is true iff `OP` is piece `P` in orientation `O`. In effect, `OP` is a new piece manufactured by `orientation/3`. So, for example,

```
?- orientation(['74', [[1, 1, 0, 0, 1, 0], [0, 1, 0, 1, 0, 0],
     [0, 1, 0, 0, 1, 0], [0, 1, 0, 0, 1, 1]]], 3, OP).
OP = ['74', [[0, 1, 0, 0, 1, 1], [1, 1, 0, 0, 1, 0],
     [0, 1, 0, 1, 0, 0], [0, 1, 0, 0, 1, 0]]]
```

The `?` in front of `O` is critical: this predicate *must* be able to generate oriented pieces as well as check them!

5 Piece compatibility

The edges of two pieces may be plugged into each other if their fingers interlock, meaning that each position contains exactly one finger. For example, edge 0 of piece 65 is compatible with edge 2 of piece 98. (But remember that for this to work one of the edges has to be reversed!)

However, this is insufficient to describe a solution to the puzzle. Clearly edge 2 of the piece in slot 0 must be compatible with edge 0 of that in slot 1. However, slots 0, 1, and share a corner. The fact that this corner must contain exactly one finger from the three pieces is an extra constraint.

Write a predicate `compatible(+P1, +Side1, +P2, +Side2)` which is true iff `Side1` of piece `P1` can be plugged into `Side2` of `P2`. (`compatible/4` does not need to be able to unify its arguments.) How are you taking into account the issue of corners, as discussed above? Include in your source file comments that indicate how your predicate works.

Write a predicate `compatible_corner(+P1, +Side1, +P2, +Side2, +P3, +Side3)` that is true iff there is exactly one finger in the first position of each given side of each given piece.

6 Putting it all together

We are ready! Write a predicate `puzzle(+Ps, ?S)` where `Ps = [P0, P1, P2, P3, P4, P5]` and `P0`–`P5` are puzzle pieces in the form described in section 1. This defines the puzzle in question. Note that your `puzzle/2` predicate is *not* expected to handle malformed piece descriptions or pieces that are physically impossible. `S` specifies a solution to the puzzle and is a list of six elements, element n describing the piece and its orientation to go into slot n (the slots are defined in section 1), These elements are each a list of two elements, the first being a piece from `Ps` and the second being an integer from -4 to 3 specifying an orientation. See the test case below for an example of the layout of `Ps` and `S`. The `?` in front of `S` is significant: your predicate should be able to test as well as generate solutions.

`puzzle/2` has the following declarative meaning.

> `puzzle(+Ps, ?S)` is true iff the pieces P0, P1, P2, P3, P4, and P5 drawn from `Ps` in orientations O0, O1, O2, O3, O4, O5 (yielding pieces OP0 through OP5) have the following edges compatible
>
> 1. 2 of OP0 and 0 of OP1,
> 2. 3 of OP0 and 0 of OP2,
> 3. 3 of OP1 and 1 of OP2,
> 4. 1 of OP0 and 0 of OP3,
> 5. 1 of OP1 and 3 of OP3,
> 6. 2 of OP1 and 0 of OP4,
> 7. 2 of OP2 and 3 of OP4,
> 8. 2 of OP3 and 1 of OP4,
> 9. 2 of OP4 and 0 of OP5,
> 10. 3 of OP2 and 3 of OP5,
> 11. 0 of OP0 and 2 of OP5, and
> 12. 1 of OP3 and 1 of OP5
>
> and the following edge triplets represent compatible corners
>
> 1. 3 of OP0, 0 of OP1, 1 of OP2;
> 2. 2 of OP0, 1 of OP1, 0 of OP3;
> 3. 2 of OP2, 3 of OP1, 0 of OP4;
> 4. 3 of OP3, 2 of OP1, 1 of OP4;
> 5. 0 of OP5, 3 of OP4, 3 of OP2;
> 6. 1 of OP5, 2 of OP4, 2 of OP3;
> 7. 2 of OP5, 1 of OP0, 1 of OP3; and
> 8. 3 of OP5, 0 of OP0, 0 of OP2

Upon finding a solution, your predicate will produce bindings for P0–P5 and O0–O5. Display these using the following Prolog code:

```
format('~w at ~w~n', [P0, O0]),
format('~w at ~w~n', [P1, O1]),
format('~w at ~w~n', [P2, O2]),
format('~w at ~w~n', [P3, O3]),
format('~w at ~w~n', [P4, O4]),
format('~w at ~w~n', [P5, O5]).
```

Multiple solutions are to be returned upon backtracking over `puzzle/2`.

7 Test case

Append the following test clause to your file:

```
test :-
  P0 = ['74', [[1,1,0,0,1,0], [0,1,0,1,0,0], [0,1,0,0,1,0], [0,1,0,0,1,1]]],
  P1 = ['65', [[1,1,0,0,1,1], [1,0,1,1,0,0], [0,0,1,1,0,0], [0,1,0,1,0,1]]],
  P2 = ['13', [[0,1,0,1,0,1], [1,1,0,1,0,1], [1,1,0,0,1,1], [1,1,0,0,1,0]]],
  P3 = ['Cc', [[0,0,1,1,0,0], [0,0,1,1,0,0], [0,1,0,0,1,0], [0,0,1,1,0,0]]],
  P4 = ['98', [[1,1,0,0,1,0], [0,1,0,0,1,0], [0,0,1,1,0,0], [0,0,1,1,0,1]]],
  P5 = ['02', [[0,0,1,1,0,0], [0,1,0,0,1,1], [1,0,1,1,0,0], [0,0,1,1,0,0]]],
  puzzle([P0, P1, P2, P3, P4, P5], S),
  (
    S = [[P0, 0], [P1, 2], [P4, 3], [P2, 1], [P3, 0], [P5, 2]];
    S = [[P0, 0], [P1, 2], [P4, 3], [P2, 1], [P3, -4], [P5, 2]];
    S = [[P0, 0], [P1, 2], [P4, 3], [P2, 1], [P5, 3], [P3, 3]];
    S = [[P0, 0], [P1, 2], [P4, 3], [P2, 1], [P5, 3], [P3, -3]]
  ).
```

(Note that `test/0` uses the "or" predicate `;/2` for tidiness but you don't need to understand it any more than you want to.) Prolog will reply "true" to the `test/0` predicate if the solutions returned by your implementation are correct for the chosen puzzle.

Each of these four solutions has piece 74 in position 0. Why is this sufficient to represent every solution to the puzzle, meaning that there are only four? You will be asked this in your viva!

8 Deliverables and Deadlines

You should submit a single Prolog source file named `CRSID-prolog11.pl` (replace CRSID with your CRSID). This file should contain all the clauses above along with appropriate tests. The file should compile and load in SWI-Prolog without errors, warnings about singleton variables, or failed clauses. For the avoidance of doubt, your code is expected to work correctly on the SWI-Prolog version running on PWF Linux.

Email your submission to `prolog-tick@cl.cam.ac.uk` .

Examination will take the form of a visual inspection of your source code, a test using different puzzles from that above, and an oral examination. Your oral viva examination will last for seven and a half minutes and you will be expected to explain the functioning of your code and resolve any issues that are raised by your examiner. Ensure that you have re-familiarised yourself with your submission prior to attending your exam. You will be told at the end of your viva whether you have passed your tick.

8.1 Important Dates

Viva sign-up sheets placed outside Student Administration in the William Gates Building. Write your CRSID in an empty slot.	Fri 20-Jan-2012 12:00 noon
Submission deadline for your tick (by email)	Fri 27-Jan-2012 12:00 noon
Viva sign-up sheets taken down	Fri 27-Jan-2012 12:00 noon
Viva Examinations	Thu 9-Feb-2012 13:00-16:00
Viva Examinations	Fri 10-Feb-2012 13:00-16:00

8.2 Tick Checklist

In order to achieve your tick you must have accomplished the following:

1. Implement and test the clauses described above providing comments where requested;
2. Your submitted code must pass visual inspection and a further test on a different example puzzle;
3. Sign up for a viva examination before Friday 27-Jan-2012 12:00 noon;
4. Submit your tick by email before Friday 27-Jan-2012 12:00 noon;
5. Attend your examination and answer questions about your submission to the examiner's satisfaction: *be prepared and be punctual*.

8.3 Alternative: C & C++ Assessed Exercise

You need only complete either the Prolog tick or the C & C++ tick but you may complete both if you wish. No further examination credit is available for completing both ticks. The examination procedure for the C & C++ tick is of similar form to the above and will run concurrently with the Prolog tick examinations.

END OF TICK