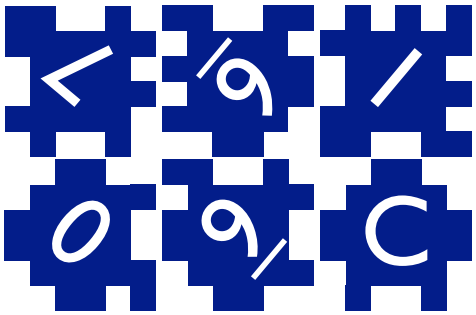
Prolog Assessed Exercise

Lecturers: Nik Sultana
Exercise originally written by David Evans and updated by Alastair Beresford and Andrew Rice

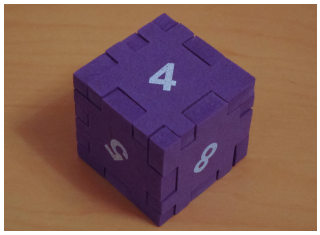
2nd January 2017

In this exercise you will write a Prolog program to solve a six-piece jigsaw puzzle. The pieces lock together to make a cube. Here is an example:



Each piece is labelled for ease of identification. The label consists of the number or character written on both sides. In the case of the example above, the labels of the pieces, clockwise from the top left, are 74, 65, 13, Cc, 98, and 02.

The goal of your Prolog program 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 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.

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.

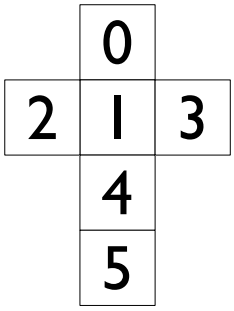
In general, we encode each piece as a list. The first element is a label for the piece. The second element of the piece data structure is a list of edges, with edge 0 coming first, edge 1 next, and so on.

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 Supporting predicates

The names of arguments to predicates in this handout are prefixed with a *mode indicator*, which is a character specifying how the argument is used. There is no universal agreement on the symbols used for mode indicators; here we adopt those used in SWI Prolog, namely:

+ Argument must be fully instantiated to a term that satisfies the required argument type. Think of the argument as an *input*.

- Argument must be unbound. Think of the argument as an *output*.

? Argument must be bound to a partial term of the indicated type. Note that a variable is a partial term for any type. Think of the argument as either an *input* or an *output* or both.

For example, `length(?List, ?Int)` is a built-in predicate which takes two arguments, `List` and `Int`. The predicate is true if `Int` is the length of the `List`. The mode indicators can be thought of indicating that the arguments can be used as either an input or as an output. Mode indicators are used in documentation, not in the source code of a Prolog program itself.

2.1 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.

2.2 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.

2.3 Reversing lists

Write a predicate `reverse(?A, ?B)` which is true iff elements in list `A` are in reverse order when compared to elements in list `B`.

2.4 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.)

2.5 Exclusive-OR list

Write a predicate `xorlist(?A, ?B)` which is true iff all the pairwise elements of lists `A` and `B` are true under the `xor` predicate defined above. For example,

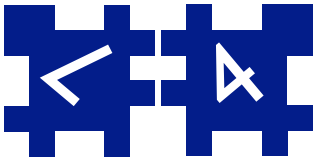
```
?- xorlist([1,0],[0,1]).
true.
?- xorlist([1,0,1],[1,1,0]).
false.
?- xorlist([1,0],X).
X = [0, 1].
```

2.6 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.

3 Piece orientation

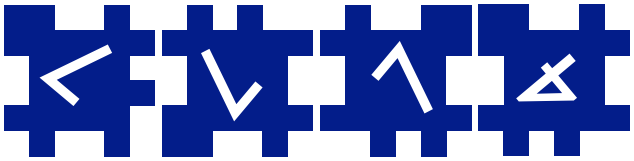
Suppose that a piece is in its canonical orientation as defined in section 1. If we flip it along its vertical axis, we end up with a new piece. This looks like the following for piece 74:



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!

4 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!)

When three pieces are combined together with a common corner, then exactly one of the three pieces must contain a finger in the corner. For example, edge 2 of the piece in slot 0 must be compatible with edge 0 of a piece in slot 1 and edge 1 of the piece in slot 2, and therefore the corner at the junction of slots 0, 1 and 2 must contain exactly one finger.

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.

5 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. The `?` in front of `S` is significant: your predicate should be able to test as well as generate solutions. When generating solutions you should constrain the first piece to be in orientation 0 as well as position 0 in order to remove duplicates (think about why this works).

`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`.

6 Deliverables and Deadlines

You should submit a single Prolog source file named `crsid.pl` (of course replacing `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. Your solution must pass the test program provided on the course website.

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. In your oral viva examination 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.

6.1 Important Dates

Email submission deadline	Monday 23rd January 12:00 noon
Release of electronic sign-up sheet for vivas on course website	Monday 23rd January 12:00 noon
Viva examinations (Intel Lab)	Thursday 26th and Friday 27th January (afternoons)

6.2 Tick Checklist

In order to achieve your tick you must accomplish the following:

1. implement and test the clauses described above, providing comments where requested;
2. check your solution passes the test program as provided on the course website;
3. ensure your solution will pass visual inspection (is it readable?);
4. check your solution works with different example puzzles;
5. submit your tick by email before before the deadline above; and
6. attend your examination and answer questions about your submission to the examiner's satisfaction: *be prepared and be punctual*.

6.3 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