CSE 307, Constraint Logic Programming
TD3: Omnidirectional Spreadsheet

Sylvain Soliman

Bachelor of Science of École Polytechnique

Remember that if you need it the reference manual of SWI-Prolog is at
https://www.swi-prolog.org/pldoc/refman/

1 Preliminaries — CLP(ℚ)

In this lab session we will use the solver over rationals provided by the `clpq` library. This will allow us to post arithmetic constraints over rationals with the usual operators `=, =\=, <, >, =<, >=` and surrounding braces `{...}`.

```
?- use_module(library(clpq)).
true.

?- {1+1 = 2}.
true.

?- {1+X = 3}.
X = 2.

?- {1+X = Y}.
{Y=1+X}.

?- {1+2 = Y}.
Y = 3.
```

2 Preliminaries — Terms and unification

Prolog solves equality constraints between two terms by unification, i.e. finding values for the variables that make the two terms syntactically equal. It also provides a predicate for building or deconstructing terms: `=..`

```
?- f(g(a), X) = f(X, Y).
X = Y, Y = g(a).

?- f(g(a), X) =.. L.
L = [f, g(a), X].
```

As you see, the second argument of `=../2` is a list with the functor of the first argument as head, and its list of arguments as tail.

3 Reminders — functional list processing (optional)

Functional programming constructs like *map*, *fold* and *anonymous functions* do exist in Prolog. They can be useful especially when handling lists, but are only shortcuts…

```
?- use_module(library(apply)).    % map/fold
true.

?- maplist(last, [[a, b, c], [d, e, f]], Lasts).
Lasts = [c, f].

?- use_module(library(yall)).     % lambda expressions with last argument as return
true.

?- foldl([X,Y,Z]>>{Z=X+Y}, [1, 5, 3, 2, 4], 0, Sum).
Sum = 15.
```

4 Omnidirectional Spreadsheet

We will now construct a (text-based) spreadsheet that allows constraints to propagate in any direction between cells.

```
a2=2
c1=6
c1=b1*a2
a1=mean(a1:c1)
```

```
  |       a |         b |         c |
  ---------------------------------
 1|     9/2 |         3 |         6 |
 2|       2 |     _5248 |     _5254 |
  ---------------------------------
```

As you can see, the formulae are not **assignments** like in an usual spreadsheet, but **constraints** that propagate to all the variables.

All your work will be done in the `spreadsheet.pl` file that you will rename as usual and that contains some already given predicates.

1. To represent our spreadsheet in a simple way, we will use nested lists. Write the predicate that given a number of rows and columns ensures that the spreadsheet is a list of `NRows` rows, each being a list of `NColumns` variables.

   Hint: the predicate `length(List, Length)` might be useful.

   ```
   %! spreadsheet(+NRows:int, +NColumns:int, ?Spreadsheet:list) is det.
   %
   % Spreadsheet is a list of NRows rows, each is a list of NColumns elements.
   ?- spreadsheet(2, 3, L).
   L = [[_7712, _7718, _7724], [_7826, _7832, _7838]].
   ```

2. To access the elements of the spreadsheet we will use standard spreadsheet cell notation with lowercase letters (e.g. `b3`).

   Write the predicate `cell_xy/3` that associates to an atom as above its x and y coordinates.

   Hint: one cane use the `atom_chars/2` and `char_code/2` predicates.

```
%! cell_xy(+Cell:atom, -X:int, -Y:int) is det.
%
% Decompose a cell name into integer coordinates.
?- cell_xy(b3, X, Y).
X = 2,
Y = 3.
```

3. Now, using the above, let us access the real cell from its atomic description by writing `element/3`.

   Hint: `nth1/2` might come in handy.

```
%! element(?Spreadsheet:list, +Cell:atom, ?Elt) is det.
%
% Return the element Elt from Spreadsheet, at address given by atom Cell
?- element([[1, 2, 3], [4, 5, 6]], b2, Elt).
Elt = 5.
```

4. The main interest of our spreadsheet is to actually handle constraints. To add a constraint implement the predicate `formula/2`. Formulae are built from cell-names (like `a1`), and the usual operators of `clpq`.

   Hint: it might be useful to split that process into two parts, first transform the formula into a constraint on the real elements of the spreadsheet, then add that constraint by using `onstraintcall(C)` or simply `onstraintConstraint`.

```
%! formula(?Spreadsheet:list, +Formula) is nondet.
%
% Parse the term Formula, convert cell names to their contents and add the constraint.
```

5. To handle formulae that tackle more than one cell, we use the standard spreadsheet rectangle notation `a1:b2`.

   Write `elements/3` that gets a list of all cell contents from the rectangle.

```
%! elements(?Spreadsheet:list, Rectangle, Elts:list) is det.
%
% Return all the list of elements in the Rectangle = Cell1:Cell2
% where Cell1 is upper-left and Cell2 is lower-right
?- elements([[1, 2, 3], [4, 5, 6]], a1:b2, Elts).
Elts = [1, 2, 4, 5] .
```

6. Extend the `formula/2` predicate to allow for two operators over cell-rectangles: `sum/1` and `mean/1`.

   You should now be able to run the example shown at the very beginning.

```
?- spreadsheet(2, 3, S), formula(S, a2 = 2), formula(S, c1 = 6),
   formula(S, c1 = b1 * a2), formula(S, a1 = mean(a1:c1)), display_sheet(S).
a2=2
c1=6
c1=b1*a2
a1=mean(a1:c1)
  |       a |        b |        c |
  ---------------------------------
 1|     9/2 |        3 |        6 |
 2|       2 |    _5248 |    _5254 |
  ---------------------------------

S = [[9/2, 3, 6], [2, _5248, _5254]] .
```