CSCI 4150 Introduction to Artificial Intelligence, Fall 2002
Assignment 2 (72 points): out Friday September 6, due Friday September 13

Notes on this assignment

- All problems are to be turned in electronically through the web tester. See the Assignment 2 Information page for details.
- There will be two files for this assignment on the assignment information page:
  - `assign2.scm` — a file of stubs for the procedures you must write. (You may also create "helper procedures" for any problem; these can have any name you wish so long as you don't create two procedures with the same name or redefine support code procedures.)
  - `a2code.scm` — containing support code
- You may always assume (on this and future assignments) that your procedures will receive valid inputs. I will not repeat this statement on future assignments.
- There is one "bonus" problem on this assignment which is marked by a "⋆".

Recursion problems

All section references are to "How to Solve Problems Using Scheme".

1. (10 points) Do exercise 10 in Section 9.4 [`one-to-n`]
2. (10 points) Do exercise 16 in Section 9.4 [`positions`]
3. (12 points) Do exercise 13 in Section 9.4 [`flatten`]
4. (12 points) Write a procedure `(corresponding-min values Lst)` where `values` and `Lst` are lists of the same length. `values` must be a list of numbers, but `Lst` can have any type of elements. This procedure should return the element of `Lst` that corresponds to the element of `values` that is the smallest number in the `values` list.

   ```
   > (corresponding-min '(3.5 7    2     9.6)
                        '(red blue green yellow))
   ;Value: green
   ```

   There is one technical detail about Scheme you need to be aware of for this (and future) problems — the difference between *exact* and *inexact* numbers. Here's something that might seem surprising:

   ```
   > (= 2.0 2)
   ;Value: #t

   > (equal? 2.0 2)
   ;Value: #f
   ```

   The reason for this apparent anomaly is that `2.0` is an inexact number, and `2` is an exact number. When considered mathematically (by the `=` procedure) they represent the same value. However, when considered as Scheme objects (by the `equal?` procedure), they are different.

   If a mathematical procedure is given only exact numbers, it will produce an exact number. If *any* argument is an inexact number, the resulting value will be an inexact number. For example:

```
> (min 3.5 7 2 9.6)
;Value: 2.
```

(The decimal point after the `2` indicates that this is an inexact value.) You can convert an exact value to an inexact value with the procedure `exact->inexact`.

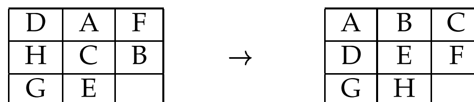
Recall that the `member` procedure uses `equal?` to compare elements. Thus:

```
> (member 2. '(3.5 7 2 9.6))
;Value: #f
```

Sliding block puzzle problems

One topic of the next assignment is "sliding block puzzles." We will write some procedures in this assignment that will be used on the next.

A well-known example of sliding block puzzles is the 8-puzzle, which consists of a 3×3 array with one empty cell and a block in all other cells; you can slide one of the horizontally or vertically adjacent blocks into the empty cell. The goal is to rearrange the blocks into a given order:

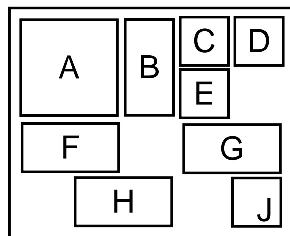


D	A	F
H	C	B
G	E	

→

A	B	C
D	E	F
G	H	

For the next assignment, however, we will be using different sized blocks on an arbitrary sized rectangular grid of cells. For example:



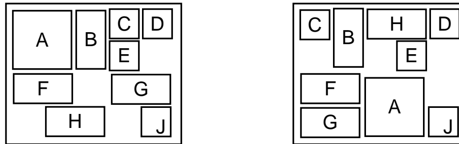
This happens to be a 5×4 grid with 2×2, 1×2, 2×1, and 1×1 blocks. The representation of the above puzzle state we'll use in Scheme is:

```
((   A     (A)     B       C      D)
 ( (A)    (A)    (B)      E   empty)
 (   F     (F)  empty     G     (G))
 (empty    H     (H)   empty   J)))
```

This is a list containing one element for each row. Each row is represented by a list with one element for each cell. If the cell is empty, that element is the symbol `empty`. If the cell is occupied by a block, then the element is either the name of the block (a single-character symbol) or a list containing the name of the block. The convention is that the upper left cell of the block is represented by the symbol and that other cells that make up the block are represented by the list with the block name. To help you visualize a puzzle from its representation in Scheme, I will provide a procedure `(print-sbp s)` which will print an arbitrary sized puzzle.

In the following problems, do not make any assumptions about the puzzle size, block size, or block shape.

5. (12 points) Write the procedure `(block-distance sbp-a sbp-b block)` which takes two sliding block puzzle states and the name of a block. It should return a list of the form: `(row-dist col-dist)` that indicate the difference in rows and columns of the given block from `sbp-a` to `sbp-b`. For example, suppose we define two states `example-a` and `example-b` to be the states:



(These variables will be defined in the support code file.)

Then we would get:

```
> (block-distance example-a example-b 'c)
;Value: (0 3)
> (block-distance example-a example-b 'h)
;Value: (3 1)
> (block-distance example-a example-b 'j)
;Value: (0 0)
```

6. (16 points) Write the procedure `(valid-right? sbp block)` which takes a sliding block puzzle state and the name of a block and returns `#t` if that block can be moved 1 cell to the right. For example:

```
> (valid-right? example-a 'f)
;Value: #t
> (valid-right? example-a 'b)
;Value: #f
```

⋆7. (5 points) Write the procedure `(valid-down? sbp block)` which takes a sliding block puzzle state and the name of a block and returns `#t` if that block can be moved 1 cell down. For example:

```
> (valid-down? example-a 'b)
;Value: #t
> (valid-down? example-a 'g)
;Value: #f
```