CSCI–4150 Introduction to Artificial Intelligence, Fall 2004
Assignment 2 (76 points), out Thursday September 9, due Thursday September 16

Notes on this assignment

- All problems are to be turned in electronically through the web tester. See the Assignment 2 information page for details. The standard late deadlines apply. (Henceforth, I will only explicitly mention exceptions to the standard late deadlines.)
- 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

Recursion problems

1. (6 points) Write a recursive procedure `(even-numbers j k)` where `j` and `k` are integers such that `j` $\leq$ `k`. It should return a list of the even integers (in order) from `j` to `k` inclusive. For example:

```
> (even-numbers 2 11)
;Value: (2 4 6 8 10)
> (even-numbers -3 -1)
;Value: (-2)
```

2. (6 points) Write a procedure `(remove-leading-zeros Lst)` which is given a list of integers `Lst`. It should repeatedly remove zeros from the front of the list and return a list that starts with a nonzero number. If the list consists of all zeros, then it should return the empty list. For example:

```
> (remove-leading-zeros '(0 0 1 0 5))
;Value: (1 0 5)
> (remove-leading-zeros '(0 0 0))
;Value: ()
```

3. (6 points) Write a recursive procedure `(add-front-element e L)` where `e` is any Scheme value and `L` is a list of lists. This procedure should add `e` to the front of every list (i.e., element) in L. For example:

```
> (add-front-element 'b '((a t) (e t) (i t) (o t) (u t)))
;Value: ((b a t) (b e t) (b i t) (b o t) (b u t))
> (add-front-element #t '((1 2 3) (4 5 6)))
;Value: ((#t 1 2 3) (#t 4 5 6))
```

4. (6 points) Section 9.5, Exercise 8 from "How to Solve Problems Using Scheme" [`all-greater`]

5. (8 points) Section 9.5, Exercise 14 from "How to Solve Problems Using Scheme" [`flatten`]

6. (8 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 (which may be exact, inexact, or a mix), but `Lst` can have any kind 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 are several ways to do this problem. If you choose to use the `member` procedure (described in Section 7.2.4), then be sure to read about equality predicates and inexact numbers in Sections 7.2.2 and 7.2.3.

7. (8 points) Write a procedure `(move-leftmost s)` which takes a list `s` of nonnegative integers where the first element is nonzero. This procedure should return a list where each element is a slightly modified copy of `s`. The basic idea is that you subtract 1 from the first element of the list and add 1 to the second element. If the second element in the original list is 0, then you make another copy of `s`, subtracting 1 from the first element and adding 1 to the third element. This continues until you reach a nonzero element (or the end) of the original list. If the list consists of a single number, it should return the empty list. For example:

```
> (move-leftmost '(1 0 0 3))
;Value: ((0 1 0 3) (0 0 1 3) (0 0 0 4))
> (move-leftmost '(4 0 1 0 7))
;Value: ((3 1 1 0 7) (3 0 2 0 7))
> (move-leftmost '(2 0 0))
;Value: ((1 1 0) (1 0 1))
> (move-leftmost '(5 4))
;Value: ((4 5))
> (move-leftmost '(3))
;Value: ()
```

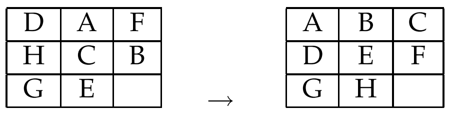
   More formally, the list returned by this procedure should have j elements, where j is the index of the second nonzero element in the list `s`. (The first nonzero element will be the first element, i.e., element 0.) If there is only one nonzero element, then j is the length of the list `s` minus 1. Element i of the returned list should be a copy of `s` where 1 is subtracted from the first element and added to element $(i+1)$.

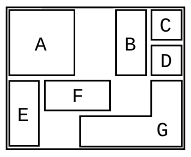
   Hint: use your `add-front-element` procedure from Problem 3.

Sliding block puzzle problems

In Assignment 3, we will be solving "sliding block puzzles." The procedures your write in these exercises will be used in that assignment.

A well-known example of sliding block puzzles is the 8-puzzle, which consists of a 3×3 grid 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. For example (on the left):





On the right is shown the type of sliding block puzzle that we will be using; it has different sized rectangular and L-shaped blocks on an arbitrary sized rectangular grid of cells. This is a picture of `example-1` from the support code.

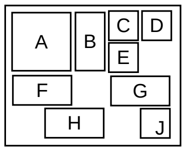
The L-shaped blocks can have any orientation or size. You can think of them as the union of two rectangular blocks that share one corner. The key assumption about these blocks is that in any row or column, the cells that belong to that block are in contiguous cells. See the support code file for more examples.

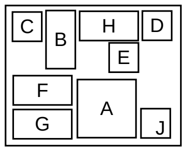
As an example of the Scheme representation for sliding block puzzle states, the above puzzle state would be:

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

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 the name of the block (a single-character symbol).

Here are two more sliding block puzzle states, `example-a` and `example-b` from the support code:





These are used in the example output for the problems in this section.

To help you visualize a puzzle from its Scheme representation, I am providing a procedure in the support code `(print-sbp s)` which will print an arbitrary sized puzzle.

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

8. (10 points) Write the procedure `(valid-right?  sbp block)` which takes a sliding block puzzle state and the name of a block. It should return `#t` if that block can be moved 1 cell to the right and `#f` otherwise. For example:

```
> (valid-right? example-1 ’a)
;Value: #t
> (valid-right? example-b ’b)
;Value: #f
```

9. (10 points) Write the procedure `(empty:right-neighbors sbp)` that takes a sliding block puzzle as an argument. It should return a list of the blocks that are on the immediate right of an empty cell in any row. There should be no duplicates in the list that you return. For example:

```
> (empty:right-neighbors example-1)
;Value: (b g)
> (empty:right-neighbors example-a)
;Value: (g h j)
> (empty:right-neighbors example-b)
;Value: (b e)
```

The order of the elements in the list you return is not important.

10. (8 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 distance in rows and columns (separately) of the given block from `sbp-a` to `sbp-b`. (The list returned should contain two nonnegative integers.) For example:

```
> (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)
```

One way to think of this is to pick any “reference” cell on the block and use it to measure the row and column distances.