CSE 341, Winter 2005, Assignment 1
Due: Tuesday, January 18, 10:00 PM

Last updated: 01-13-05

You will write several SML functions having to do with dominoes. All domino knowledge necessary for this assignment will be provided. A domino is a small, rectangular tile (called a "bone") divided into two square halves. A number of small dots (called "pips"), from 0 to 6, occupy each half. Bones with a different number of pips on each side are called "singles", while bones with the same number of pips on both sides are called "doubles." During game play, players take turns placing their tiles end to end, with the restriction that touching ends must have the same number of pips. In this particular variation, only one bone may be played at each end of an already-played bone.

There are 7 "suits" of bones in the game for each possible number of pips on a bone. All singles belong to two suits, while all doubles belong to only one suit. For instance, the bone 3-5 belongs to suits 3 and 5, but the bone 2-2 belongs only to suit 2.

In ML, you will use pairs (type `int * int`) to represent the bones, with each number in the pair representing the number of pips on each side. Bones can be rotated in the game, so for most purposes, `(3,5)` is the same bone as `(5,3)`. However, in some problems the order will matter. The problem will explicitly state when this is the case. You may assume you will always be passed valid bones when writing your functions, except when the problem requires you to explicitly test this.

1. Write a function `legal_bone` that takes a bone `b`, returning `true` if both sides of the bone contain a valid number of pips, `false` otherwise.

2. Write a function `compatible_bones` that takes two bones, `b1` and `b2`, and returns `true` if both bones could be legally placed end-to-end in some orientation, and `false` otherwise.

3. Write a function `all_legal_bones` that takes a list of bones `blist`, returning `true` if all bones in the list are legal (or the list is empty) and `false` otherwise. Use a previous function in your solution.

4. Write a function `no_doubles` that takes a list of bones `blist`, returning `true` if all bones in the list are singles (or the list is empty) and `false` otherwise.

5. Write a function `make_suit` that takes an integer `s` and returns a list of all bones in that suit. *Hint: use a helper function.*

6. Write a function `all_same_suit` that takes a list of bones `blist`, returning `true` if all bones in the list share at least one common suit, and `false` otherwise (or if the list is empty). *Hint: This is not as obvious as it sounds. Use a helper function.*

7. Write a function `legal_order` that takes a list of bones `blist`, returning `true` if the list represents a line of bones laid out in legal order, and `false` otherwise (or if the list is empty). For this function, *the order of the integers within a pair matters.* The list `[(1,3),(3,5)]` represents a legal line of bones, but `[(3,1),(3,5)]` does not. *Hint: use a helper function.*

8. Write a function `make_line_between` that takes two integers, `a` and `b`, and returns a list of bones where the left side of the first bone is `a`, the right side of the last bone is `b`, and all numbers between between `a` and `b` (inclusive) are present on at least one side of one bone in the list. The result must represent a legal line of bones as defined in problem 7. For example, `make_line_between (1,4)` could return `[(1,2),(2,3),(3,4)]`, but *not* `[(1,3),(3,4)]` or `[(1,2),(3,4)]`.

9. **(Extra Credit)** Write a function `legal_order_r` that takes a list of bones `blist`, returning `true` if the list could represent a line of bones in legal order if bones can be rotated freely in place, and `false` otherwise (or if the list is empty). For example, `[(1,2)(3,2)(3,4)]` would pass the test because the `(3,2)` could be rotated to `(2,3)`, creating a legal line of bones.

Type Summary

A correct solution will cause these bindings to be printed in the read-eval-print loop:

```
val legal_bone = fn : int * int -> bool
val compatible_bones = fn : (int * int) * (int * int) -> bool
val all_legal_bones = fn : (int * int) list -> bool
val no_doubles = fn : (int * int) list -> bool
val make_suit = fn : int -> (int * int) list
val all_same_suit = fn : (int * int) list -> bool
val legal_order = fn : (int * int) list -> bool
val make_line_between = fn : int * int -> (int * int) list
```

Getting these bindings does not necessarily mean your solution is correct. Be sure to test your code.

Assessment

Your submission should:

- Be correct.
- Exhibit good style, particularly indentation. Comments are not strictly necessary, but use them if you wish to explain something.
- Not use any features not yet covered in class. You will need let bindings to define helper functions, but datatypes and case matches are not necessary.

Turn-in Instructions

Use the turn-in form linked from the course website.