# CSE 341, Winter 2005, Assignment 2
# Due: Thursday, January 27, 10:00 PM

Last updated: 01-20-05

You will continue with the theme of dominoes by writing several functions dealing with actual game play. There are many variations on dominoes, but you will be implementing the one described here, which is a combination of several variations designed for this assignment. In this game, each player has a "hand" of bones available to them to play. Players take turns placing bones end to end into a chain called a "layout." When the game first starts, the layout is empty, and any bone may be played. After that, the layout will have a left and a right side on which additional bones may be played, with the restriction that bones played end to end must have the same number of pips on the touching sides of the bones. A deck of bones is also used during the game. If a player does not have a bone that can be played, or wishes to not play a bone, they may draw another bone from the top of the deck and add it to their hand. If no bones remain in the deck, the player can simply pass instead, doing nothing on their turn. In this variation, a player can either draw a single bone from the deck or play a bone in a turn, but not both. The object of the game is to be the first player to get rid all bones in their hand. All other information and terminology you need to know can be found in homework 1.

You will use the following definitions in your solutions. A bone is represented as it was in homework 1:

```
type bone = int * int
```

A hand is a list of bones, where neither the orientation of the bones nor the order of the bones matters. A deck is also a list of bones, where the orientation of the bones is unimportant, but the order of the bones matters. A layout is a list of bones as well, where both orientation and order of the bones matters. To distinguish these three concepts, we will use 3 different types. Keep in mind that these types are all synonymous; the names are only for convenience and clarity.

```
type hand = bone list
type deck = bone list
type layout = bone list
```

A move is either playing the first bone of the game, or playing a bone on the left or right side of the layout, or drawing/passing when no bone can be played.

```
datatype move = PlayFirst of bone
              | PlayLeft of bone
              | PlayRight of bone
              | PassDraw
```

Sometimes a move will be illegal during game play. In such a case, a function might wish to indicate this by raising an exception:

```
exception BadMove
```

Using these definitions, do the following:

1. Write a function `find_playable` that takes a hand `h` and a suit `s` and returns a value of type `bone option`. If no bone of suit `s` exists in `h`, `find_playable` should return `NONE`. If at least one bone of suit `s` exists in `h`, then `find_playable` should return `SOME` of that bone. If more than one such bone exists in suit `s`, pick an arbitrary bone to return.

2. Write a function `without_bone` that takes a hand `h` and a bone `b`, and returns a hand with one instance of `b` removed, if any exist. For this function, orientation of the bones is unimportant, so `without_bone([(1,2)], (2,1))` must return the empty list `[]`. Your implementation must be tail recursive. This means you will need to use an accumulator-style recursive helper function. Your solution may only call this helper function as well as built-in operators, except for the list append operator, which you may not use. *Hint: the order of bones in a hand is unimportant, so the hand you return doesn't have to be in the same order as you hand you receive.*

3. Write a function `layout_summary` that takes a layout `l` and returns a value of type `(int * int) option`. If the layout is empty, `layout_summary` should return `NONE`. Otherwise, it should return `SOME (x,y)`, where `x` is the number of pips on the left side of the leftmost bone in the layout, and `y` is the number of pips on the right side of the rightmost bone in the layout.

4. Write a function `best_move` that takes a layout `l` and a hand `h` and returns a value of type `move` using the following criteria:

   - If the layout is empty, `best_move` must return `PlayFirst` of an arbitrary bone in `h`. If `h` is also empty, `best_move` must return `PassDraw`.
   - If the layout is not empty, and a bone `b` exists in `h` that could be played on either end of the layout, `best_move` must return `PlayLeft(b)` or `PlayRight(b)` as appropriate.
   - If the layout is not empty, but no bone exists in `h` that could be played on either end of the layout, or `h` is empty, `best_move` must return `PassDraw`.

   This function need not implement any sort of strategy. Use `find_playable` and `layout_summary` in your solution.

5. Write a function `do_move` that takes a layout `l`, a deck `d`, a hand `h`, and a move `m`, and returns a tuple `(lprime, dprime, hprime)` of type `layout * deck * hand`. `l`, `d`, and `h` represent the layout, deck, and player's hand before a move is made. `lprime`, `dprime`, and `hprime` must represent the layout, deck, and player's hand after the move `m` is made. Specifically:

   - If the move `m` is `PassDraw`, and the deck is not empty, then the first bone in the deck must be removed and placed in the player's hand. Otherwise, no change occurs.
   - If the move `m` is `PlayFirst` of bone `b`, and the layout is not empty, or `b` does not exist in the player's hand, then the exception `BadMove` must be raised. Otherwise, the bone `b` must be removed from the player's hand and placed as the only bone in the layout.
   - If the move `m` is a `PlayLeft` or `PlayRight` of bone `b`, and bone `b` does not exist in the player's hand, or bone `b` is not playable on the indicated side of the layout, `BadMove` must be raised. If the layout is empty, `BadMove` must be raised. Otherwise, the bone `b` must be removed from the player's hand and placed on the left or right side of the layout as appropriate, *in the correct orientation*. You may use the built in append operator `@`.

   Remember that you are not using mutation, so when it says that you must "remove" a bone from the player's hand and place it in the layout, for example, it means that the bone should appear in `lprime` but be absent from `hprime`. To give a concrete example, `do_move([(1,2)],[],[(5,2),(3,0)],PlayRight(5,2))` must return `([(1,2)(2,5)],[],[(3,0)])`, which is what the state of the layout, player's hand, and deck would be after playing `(5,2)` on the right side of the layout. Make use of `without_bone` and possibly `layout_summary` in your solution. **This function is complex**; use helper functions and pattern match aggressively.

# Type Summary

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

```
type bone = int * int
type hand = bone list
type deck = bone list
type layout = bone list
datatype move
  = PassDraw
  | PlayFirst of int * int
  | PlayLeft of int * int
  | PlayRight of int * int
exception BadMove
val find_playable = fn : hand * int -> (int * int) option
val without_bone = fn : hand * bone -> (int * int) list
val layout_summary = fn : layout -> (int * int) option
val best_move = fn : layout * hand -> move
val do_move = fn : layout * deck * hand * move -> layout * deck * hand
```

Getting these bindings does not necessarily mean your solution is correct. Also, the bindings you get may not look exactly like these due to type synonyms. Also, depending on how you solve problem 2, it might have a polymorphic type. Be sure to test your code.

## Assessment

- Your solution should generate correct bindings and give correct results.
- For this assignment, you must use pattern matching. Do not use the functions `null`, `hd`, `tl`, `isSome` or anything starting with `#`. You may use the function `valOf` when it is impossible to call it on a `NONE` value. *Hint: this is useful in conjunction with `layout_summary`, but you must be sure that `layout_summary` did not return `NONE`. Remember that you can't use `isSome`, so you must know something about what `layout_summary` was called with...*
- Do not use mutation, even if you know how.

## Turn-in Instructions

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