Data Structures and Functional Programming	**Problem Set 2**
CS 3110, Fall 2013	**Due at 11:59 PM, Thursday, September 19**
Version: 1	**Last Modified: September 11, 2013**

Read these instructions.

Compilation and Naming

All code you submit must compile, and all functions must be correctly named and typed.

Programs that do not compile will be heavily penalized. If your submission does not compile, we will notify you immediately. You will have 24 hours after the submission date to supply us with a patch. If you do not submit a patch, or if your patched code does not compile, you will receive an automatic zero.

You can ensure your submission complies with our scripts by following the instructions in the `test.ml` file provided as part of the release. This will only check compilation; it does not substitute for testing your code.

Folding

Folding is one of the most useful tools in functional programming. Because of its ubiquity in OCaml programs, this problem set is designed to get you comfortable with folds that perform useful computations on lists. **For full credit, your solutions must not contain the `rec` keyword unless the problem states otherwise.**

Style

Be aware that excessive use of `fold` is often bad style — in many cases recursive functions are more readable. However, the goal of this assignment is to familiarize you with folds, so you will be forced to push this particular boundary.

This means that you will have to apply extra effort to make your code clear and readable. Continue to abide by the CS 3110 style guide, and be sure to make good use of comments and well-named auxiliary definitions.

Loading modules

For part three, we have provided you with a helper module `Melodytree`. To compile the module, run the shell command

```
ocamlc melodytree.mli melodytree.ml
```

This will generate the file `melodytree.cmo`. To load this file into your toplevel, use the command `#load "melodytree.cmo";;`, or pass the filename as an argument to the `ocaml` command.

To make the values and types defined in `melodytree.mli` available to your code, use the ocaml command `open Melodytree;;`.

Part One: Folding

Exercise 1

Write a function

```
mean : float list -> float
```

which, given a list of rationals $[\alpha_0; \alpha_1; \cdots; \alpha_{n-1}]$, returns the arithmetic mean

$$m = \frac{1}{n} \sum_{k=0}^{n} \alpha_k$$

or fails if the list is empty.

Examples

```
# mean [3.];;
- : float = 3.

# mean [1.; 2.; 3.];;
- : float = 2.;

# mean [];;
Exception: Failure "mean: Empty list"
```

Exercise 2

Write a function

```
max2 : 'a list -> 'a
```

which, given a list of type `'a`, returns the second-greatest unique element in the list, or fails if the list contains fewer than two unique elements.

Examples

```
# max2 [0; 1; 4; -13];;
- : int = 1

# max2 [1.; 3.5; 3.5; 5.];;
- : float = 3.5

# max2 ["one"; "two"; "three"];;
- : string = "three"

# max2 ["whoops"; "whoops"];;
Exception: Failure "max2: Less than two unique elements"
```

```
# max2 [5; 5; 1; 2];;
- : int = 2
```

Exercise 3

Write a function

```
intersect_all : 'a list list -> 'a list
```

which, given a list of lists L returns the intersection of all of the lists $\ell \in L$. That is, it returns a list containing all the elements which are contained in each $\ell \in L$. The order of these elements does not matter.

Examples

```
# intersect_all [];;
- : 'a list = []

# intersect_all [[1; 2; 3]; [2; 3; 4]; [3; 4; 5]];;
- : int list = [3]

# intersect_all [["swag"; "cs3110"; "yolo"]; ["#hashtag"; "yolo"; "swag"]];;
- : string list = ["yolo"; "swag"]
```

Part Two: Big integers

`int`s in OCaml are represented using a fixed number of bits (either 31 or 63 depending on your platform). This means that unlike the actual integers, there is a largest `int`; this value has the built-in name `max_int`. If you evaluate `max_int + 1`, you will discover a potentially surprising result.

In some applications (such as geometry and cryptography), it is important to work with very large integers. In this part of the problem set, you will implement some basic operations for arbitrarily large (positive) integers.

We will represent large integers using lists of (positive) `int`s:

```
type bignum = int list
```

We will interpret the list $[a_0; \cdots; a_{n-1}; a_n]$ as the number

$$a_0 \cdot 2^{16n} + \cdots + a_{n-1} \cdot 2^{16} + a_n$$

Exercise 4

Since OCaml integers contain more than 16 bits, there are multiple ways to represent the same big integer. For example, the two lists `[1; 0]` and $[2^{16}]$ both represent the number 2^{16}.

In such situations, it is often useful to choose a **canonical representation**; a unique "preferred" way to represent each abstract value. For the representation of large integers that we have chosen, if we require the elements of a `bignum` to be between 0 and 2^{16}, including 0 but excluding 2^{16}, then there is exactly one way to represent each integer n. We will refer to this `bignum` as the canonical representation of n. In the following exercises, you may assume that all elements of the inputs are non-negative.

Write a function `canonicalize` which converts a bignum to its canonical form.

Examples

```
# canonicalize [65537];;
- : bignum = [1; 1]

# canonicalize [1236842; 0; 1789212];;
- : bignum = [18; 57194; 27; 19740]

# canonicalize [0; 0; 1];;
- : bignum = [1];;

# canonicalize [];;
- : bignum = [];;
```

> **Hint:** You may find the built-in functions for bitwise operations to be useful. The documentation for them can be found here.

Exercise 5

Write a function `bignum_plus` that adds two `bignums` and returns their canonical sum.

Examples

```
# bignum_plus [65537] [0];;
- : bignum = [1; 1]

# bignum_plus [1236842; 0; 1789212] [65536];;
- : bignum = [18; 57194; 28; 19740]

# bignum_plus [0; 0; 1] [1; 1; 0];;
- : bignum = [1; 1; 1];;

# bignum_plus [] [1];;
- : bignum = [1];;
```

> **Hint:** For this problem, you may find it to useful to use `List.fold_left2` or `List.fold_right2`.

Exercise 6

Write a function `bignum_mult` that multiplies two `bignums` and returns their canonical product.

Examples

```
# bignum_mult [65537] [0];;
- : bignum = []

# bignum_mult [1236842; 0; 1789212] [65536];;
- : bignum = [18; 57194; 27; 19740; 0]

# bignum_mult [0; 0; 1] [1; 1; 0];;
- : bignum = [1; 1; 0]

# bignum_mult [] [1];;
- : bignum = [];;
```

Part Three: Computer Generated Music

In this section, you will implement two simple search algorithms and use them to automatically generate music. Although we are applying these algorithms to generate music, we have created abstractions that will allow you to implement the project without any understanding of music. Feel free to ignore any footnotes in this document.

We will represent notes as integers[1], and melodies as lists of integers[2]:

```
type note   = int
type melody = note list
```

We will provide a web service that allows you to convert melodies in this form into pdf files containing written music and mp3 files that you can listen to.

Exercise 7

As a first exercise, you will convert a melody from the major scale to the melodic minor scale.

There are some simple rules to follow:

- Any input note that is congruent to 4 modulo 12 should be reduced by one (musically, this is called dropping a major third).
- Any input note that is congruent to 9 modulo 12 should be reduced by one, and any input note that is congruent to 11 modulo 12 should be reduced by one (musically, this corresponds to dropping the major sixth and seventh respectively)...
- **except** that subsequences formed by zero or more 9s (mod 12) followed by one or more 11s (mod 12) followed by a 0 (mod 12) should remain unchanged (musically, this is what differentiates the melodic minor scale from the natural minor scale).

Write a function

```
major_to_melodic_minor : melody -> melody
```

that implements these rules.

Examples

```
# major_to_melodic_minor [0; 2; 4; 5; 7; 9; 11; 0; 11; 9];;
- : melody = [0; 2; 3; 5; 7; 9; 11; 0; 10; 8]

# major_to_melodic_minor [7; 9; 11; 9; 11; 0; 11; 9; 0];;
- : melody = [7; 8; 10; 9; 11; 0; 10; 8; 0]

# major_to_melodic_minor [9; 7; 9; 9; 9; 11; 11; 0];;
```

[1] the integer representation of a note n is given by the number of semitones between n and middle C.
[2] this project does not contain any rhythm; all notes are whole notes.

```
- : melody = [8; 7; 9; 9; 9; 11; 11; 0]

# major_to_melodic_minor [9; 23; 0; 11; 21];;
- : melody = [9; 23; 0; 10; 20]
```

Exercise 8

> **Note:** You **may** use the **`rec`** keyword for this problem.

Many of the key principles of music composition are derived from a musical tool known as species counterpoint. The theory of species counterpoint was first published by Johann Joseph Fux in The Study of Counterpoint in 1725.

The theory of species counterpoint consists of a number of guidelines that distinguish compositions that "sound good" from those that "sound bad" (to a 17th century ear). We can quantify the "badness" of any composition by calculating how often and how egregiously the composition violates Fux's guidelines. Using this definition, we can automatically search for good compositions — those that minimize badness.

For the purposes of this assignment, we will only consider compositions that consist of two sequences of notes: a bassline and a melody. We will search for **complete** compositions — those that have exactly one note in the melody for each note in the bassline. Not all possible compositions are musically **valid**; some notes and progressions are strictly forbidden by the rules of species counterpoint.

We can think of partial compositions over a given bassline as nodes in a tree: the children of a given composition c are the valid compositions that can be generated by adding a single note to the melody of c.

We have provided you with an abstraction that allows you to traverse the tree rooted at a given bassline. In the file `melodytree.mli`, we have defined an opaque type `node` that represents a node in the tree. The function `children` allows you to find all of the children of a given node. You can The `badness` function[3] returns an integer representing the badness of a partial composition; compositions with larger badnesses are worse. The `is_goal` function can be used to determine whether the composition represented by a node is complete. Given a complete composition, you can extract the melody using the `counterpoint` function.

To create the root node, you should invoke the `start` function. This function takes in the initial bassline (represented using the `bassline` type, which is just an alias for `melody`). The `start` function also takes a **mode**, which changes the set of allowable notes. You can use the `Ionian` mode to generate major melodies, or experiment with the other modes to generate music with a different character.

Using these functions, write a function

```
find_best_counterpoint_naive : mode -> bassline -> melody
```

[3] our implementation of `badness` is based on the method described in the paper Automatic Species Counterpoint by Bill Schottstaedt.

that generates all possible complete compositions in the given mode over the given bassline, and returns the melody corresponding to the best composition.

Exercise 9

> **Note:** You **may** use the `rec` keyword for this problem.

The badness of a node is computed by accruing penalties for each note that is added to the melody. This means that for any node `n`, if `m` is a child of `n`, then `badness m >= badness n`.

We can make use of this fact to perform a **guided** search of the solution space in a manner analogous to Dijkstra's algorithm. If we maintain a "frontier" of discovered but unexplored nodes, and we explore the nodes in increasing order of badness, then the first complete composition that we discover will be the best complete composition. Once a complete composition is found, the search can return this composition immediately.

Write a function

```
find_best_counterpoint_guided : mode -> bassline -> melody
```

that uses this strategy to find the best melody over the given bassline.

> **Hint:** You may find the function `List.merge` helpful.

Exercise 10

For the purposes of this question, you may assume that the length of the list returned by the provided `children` function is always at least two. You may also assume that `children` runs in constant time.

In the file `part3.txt`, answer the following questions. Each answer should consist of one or two sentences.

(a) How large is the tree of partial compositions?

(b) The `find_best_counterpoint_guided` function explores the tree of partial compositions. Can this function run in less time than it takes to construct the tree of partial compositions? Why or why not?

(c) Find a bassline that causes `find_best_counterpoint_naive` to run for more than 5 minutes on your machine. How long does `find_best_counterpoint_guided` take to find the melody for this bassline?

(d) Do `find_best_counterpoint_naive` and `find_best_counterpoint_guided` necessarily return the same melody? Why or why not?