Section 4 - ML Modules, Currying

This handout was composed by Porter Jones. There are probably plenty of typos/incorrect solutions/etc for you to catch! Please email me with any issues, comments. or feedback at pbjones@cs.washington.edu. All thoughts are welcome :)

Practice w/Modules

a) Below on the left are various lines of code that belong in the signature and module skeletons on the right. Your job is to discern which lines belong in the RATIONAL signature and which belong in the Rational module. For sake of space, the full expression in the function bindings has been replaced with a comment of `(* function_name body *)`

Lines of Code	signature and module skeleton

a) `val make_frac : int * int -> rational`
 	
b) `fun toString (x,y) = (* to_string body *)`
 	
c) `type rational`
 	
d) `val toString : rational -> string`
 	
e) `fun Whole i = (i,1)`
 	
f) `exception BadFrac`
 	
g) `type rational = int * int`
 	
h) `fun add ((a,b),(c,d)) = (* add body *)`
 	
i) `val add : rational * rational -> rational`
 	
j) `val Whole : int -> rational`
 	
k) `fun make_frac (x,y) = (* make_frac body *)`
	`signature RATIONAL =`
	`sig`
	
	`end`
	
	`structure Rational :> RATIONAL =`
	`struct`
	
	`end`

b) Fill in the `ICECREAMSHOP` signature below so that clients can use all of the function bindings in `IceCreamShop` defined on the next page but aren't able to create "bad" orders. A "bad" order is one that contains a flavor that is not in `available_flavors` or that has a number of scoops less than 0 or greater than `max_scoops`.

```
signature ICECREAMSHOP =
sig
  exception BadOrder









end
```

```
structure IceCreamShop :> ICECREAMSHOP =
struct
  val max_scoops = 3
  val available_flavors = ["vanilla", "chocolate",
                           "huckleberry", "moose tracks"]
  exception BadOrder
  type order = (string * int)

  fun buy_order (flavor, scoops, money) =
    if money < scoops orelse scoops < 0 orelse scoops > max_scoops
       orelse not (isSome(List.find (fn x => x = flavor)
available_flavors))
    then raise BadOrder
    else (flavor, scoops)

  fun consume_scoop (f, s) =
    if s > 0
    then SOME(f,s - 1)
    else NONE

  fun num_scoops (_, s) = s

  fun has_scoops (_, s) = s > 0

end
```

c) Which of the above functions can be implemented by a client of `IceCreamShop` who doesn't have access to the module?

Practice w/Currying

a) Write a function `filter_by_example` that takes a function `f`, a value `x`, and a list `xs` in curried form. Upon applying the three arguments, the result of the function should be a new list that has all of the values from the original list that return the same result when `f` is applied to them as when `f` is applied to the given value.

b) Write a function `same_size_as` that takes a list and a list of lists in curried form and returns all of the lists in the second parameter that have the same size as the given list. Use `filter_by_example` in your answer.

c) Write a function `count_o` that takes a string and returns the number of occurrences of the lowercase letter `#"o"` in the given string. Our solution uses `List.filter` and `String.explode`

d) Write a function `silly_application` that takes a list of strings and returns a new list of strings of all the strings in the given list that have the same number of occurrences of the letter o as “dogsarecool”. Use `count_o` and `filter_by_example`.

e) Write a function `contains` that has type `''a -> ''a list -> bool` (notice the currying) and takes a first argument value, a second argument list, and returns true if the first argument is in the second argument.

f) Write a function `filter_unique` that takes a function, list of previous values, and an input list of values. If applying the given function to an input value results in a value not previously seen (not in the list of previous values), the input value should be added to the result list, and the result of applying the function should be added to the previous values list.

g) Write a function `unique_sums` that takes a list of lists of integers and returns a new list that contains lists that have unique summations. Use `filter_unique` in your answer.

h) Write a function `all_that_contain` that has type `''a -> ''a list list -> ''a list list` (notice the currying) which takes a value, and a list of lists, and returns a new list of all of the original lists that contain the given value.

i) Write a function `even_only` that takes a list of lists of ints and returns a new list of lists of ints that are the original lists with only even values. Use a val binding and some combination of `List.map` and `List.filter`

j) Write a function `even_only_not_empty` that returns the same thing as `even_only` except has no empty lists in its result. Our solution uses a fun binding, function composition, and calls to `List.filter` and `even_only`

The following questions assume that the current environment contains the following binding:

```
val names = (* some list of names *) : string list
```

k) Create a val binding `unique_size_non_empty` that is bound to a string list containing strings from `names` that all have different sizes and are not the empty string.

l) Create a val binding `all_pairs` that is bound to a list of lists of pairs, where the ith list contains all of the unique pairs with the ith value from `names` as the first string in the pair. Our solution uses two calls to map.

Section 4 - Solutions

This handout was composed by Porter Jones. There are probably plenty of typos/incorrect solutions/etc for you to catch! Please email me with any issues, comments. or feedback at pbjones@cs.washington.edu. All thoughts are welcome :)

Practice w/Modules

a)

```
signature RATIONAL =
sig
  type rational
  exception BadFrac

  val make_frac : int * int -> rational
  val toString : rational -> string
  val add : rational * rational -> rational
  val Whole : int -> rational

end

structure Rational :> RATIONAL =
struct
  type rational = int * int
  exception BadFrac

  fun toString (x,y) = (* to_string body *)
  fun Whole i = (i,1)
  fun make_frac (x,y) = (* make_frac body *)
  fun add ((a,b),(c,d)) = (* add body *)

end
```

b)

```
signature ICECREAMSHOP =
sig
  exception BadOrder
  type order
  val max_scoops : int
  val available_flavors : string list

  val buy_order : string * int * int -> order
  val consume_scoop : order -> order option
  val num_scoops : order -> int
  val has_scoops : order -> bool

end
```

c)

`has_scoops` is the only function that can be implemented outside of the IceCreamShop module. One possible implementation would be:

```
fun has_scoops_2 order = IceCreamShop.num_scoops order > 0;
```

Practice w/Currying

a)
```
fun filter_by_example f x =
    List.filter (fn x' => f x = f x')
```

b)
```
fun same_size_as xs = filter_by_example List.length xs
```

c)
```
fun count_o s =
    List.length (List.filter (fn x => x = #"o") (String.explode s))
```

d)
```
val silly_application = filter_by_example count_o "dogsarecool"
```

e)
```
fun contains x =
    List.foldl (fn (x', acc) => acc orelse x' = x) false
```

f)
```
fun filter_unique f prev xs =
    case xs of
         [] => []
       | x'::xs' =>
           let
             val result = f x'
           in
             if contains result prev
             then filter_unique f prev xs'
             else x' :: filter_unique f (result :: prev) xs'
           end
```

g)
```
fun unique_sums xs = filter_unique List.length [] xs
```

h)
```
fun all_that_contain x = (List.filter (contains x))
```

i)
```
val even_only =
    List.map (List.filter (fn x => x mod 2 = 0))
```

j)
```
fun even_only_not_empty xs =
    List.filter (not o List.null) (even_only xs)
```

k)
```
val unique_size_not_empty = filter_unique String.size [0]
```

l)
```
val all_pairs =
    List.map (fn x => List.map (fn y => (x, y)) names) names
```