Section 2 - Recognizing/Programming w/SML Types

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/SML Types

a) For each of the following examples, determine if t2 is a more general type than t1. A type t2 is more general than the type t1 if you can take t2, replace its type variables consistently, and get t1.

i) \[ t1: \text{string list} \times \text{int list} \rightarrow \text{int list} \]
   \[ t2: \ 'a \text{ list} \times \ 'b \text{ list} \rightarrow \ 'b \text{ list} \]

ii) \[ t1: \text{string list} \times \text{string list} \rightarrow \text{string list} \]
    \[ t2: \ 'a \text{ list} \times \ 'b \text{ list} \rightarrow \ 'b \text{ list} \]

iii) \[ t1: \text{string list} \times \text{string list} \rightarrow \text{int list} \]
    \[ t2: \ 'a \text{ list} \times \ 'b \text{ list} \rightarrow \ 'b \text{ list} \]

iv) \[ \text{type foo = int} \times \text{int} \]
   \[ t1 = \text{foo} \rightarrow \text{bool} \]
   \[ t2 = \ 'a \times \ 'a \rightarrow \text{bool} \]

b) Write each of the following SML functions. Once you have written the function, try to reason about the most general type the SML type checker would assign to the function binding.

i) Write a function swap_pair that takes the values a and b and returns a pair that has the given values in the reverse order they were passed in.

ii) Write a function swap_pairs_list that takes a list of pairs and returns a list of pairs with each of the original pairs’ values swapped.

iii) Write a function size that takes a list and returns the number of elements in that list.

iv) Write a function contains that takes a value and a list and returns true if the given value is in the list (false otherwise).

v) Write a function remove_all that takes a value and a list and returns a list of the values in the original list not equal to the given value.
**Programming w/simple datatypes**

Answer questions a - c using the following bindings:

```plaintext
type cart = real * real
datatype shape =
    Circle of cart * real (* coordinates and radius *)
| Square of cart * real (* coordinates and side length *)
| Rectangle of cart * real * real (* coordinates and side lengths *)
```

a) Write a function `area` that takes a shape and calculates the area of the shape. You may use 3.14 for pi.

b) Write a function `quadrant_one_only` that takes a list of shapes and returns a list of the shapes in the given list that are in quadrant one (positive x and y coordinates).

c) Write a function `construct_squares` that takes a list of `ints` and returns a list of `Squares`. The values of the `Squares` should be related to the corresponding `int` in the given list, with the x and y coordinates of the `Square` being the value of the `int`, and the side length of the `Square` being the absolute value of the `int`.

d) Fill in the question marks in the following `exp` datatype binding. Then write a function `eval` which takes an `exp` and returns an `int` that represents the result of evaluating the given `exp`.

```plaintext
datatype exp = Constant of int
    | Negate of ?
    | Add of ? * ?
    | Multiply of ? * ?
```

**More complex programming with datatypes/pattern matching**

Use the following datatype binding to solve the problems in this section. **Disclaimer: These problems may be more approachable after Dan’s lecture on Friday. They will serve as good practice for homework 2**

```plaintext
datatype dessert =
    IceCream of (string * int) (* flavor * num scoops *)
| Pie of (string * int) (* flavor * num slices *)
| Brownie of (int) (* number of brownies *)
| WhippedCream
| Feast of dessert list (* collection of desserts *)
```

1. Write a function `add_whipped_cream` which takes a list of desserts and returns a list of pairs, where the first value in each pair is the dessert from the given list and the second is WhippedCream.

2. Write a function `ice_cream_feast` which takes a list of strings that are flavors and returns a Feast of corresponding IceCreams, each having one scoop of a given flavor.
3. Write a function `flatten` which takes a dessert and returns a dessert list of all the individual non-Feast desserts recursively contained in the given dessert. For example, given a dessert

```scala
def flatten(d: Dessert): List[Dessert] = d match {
  case Feast(deserts) => flatten(deserts)
  case IceCream(flavor, scoops) => List(flavor, scoops)
  case Brownie(slices) => List(slices)
  case WhippedCream => List()
  case Pie(fruit, slices) => List(fruit, slices)
}
```

Given a dessert

```scala
val d = Feast([IceCream("vanilla", 2), Feast([Brownie(1), WhippedCream]), Pie("apple", 4)])
```

a call of `flatten(d)` would return the list

```scala
[IceCream("vanilla", 2), Brownie(1), WhippedCream, Pie("apple", 4)]
```

Note how when coming across a Feast, the Feast itself is not added to the resulting list, but rather its desserts are recursively merged into the result list. `flatten` should also work for any dessert passed to it, not just Feasts.

4. Write a function `num_scoops` that takes a dessert and a flavor as a string and returns the number of scoops of ice cream of the given flavor that are contained in the dessert.

5. Write a function `flavors` that takes a dessert and returns a list of strings that are all of the flavors in the given dessert. The flavor of Pie or IceCream should have the flavor with the appropriate dessert appended to it (e.g. `IceCream("huckleberry", 2)` has the flavor "huckleberry ice cream"). Brownies should have the flavor "brownie" and WhippedCream has the flavor "whipped cream".

6. Write a function `enough_ice_cream` that takes in a dessert and returns true if the dessert contains enough scoops of ice cream for other desserts, and false otherwise. "Enough scoops of ice cream" is defined as having a scoop of ice cream for every pie slice and every brownie in the dessert. The flavors of the scoops of ice cream do not matter.

7. Write a function `dessert_equal` that takes two desserts and determines if they are equal or not. Two desserts that are not Feasts are considered equal if they are the same type with exactly the same information. For example, `IceCream("vanilla", 2)` is only considered equal to `IceCream("vanilla", 2)` and it would not be considered equal to `IceCream("vanilla", 3)` or `Pie("vanilla", 2)`. Two Feasts are considered equal if they have exactly the same number of elements, and all elements at corresponding positions of the Feasts are equal.

8. Write a function `no_whipped_cream_allowed` that takes a dessert and returns a dessert option. The function should return `SOME (d)` where `d` is the same as the given dessert with all instances of WhippedCream removed. If the given dessert was made completely of WhippedCream, `NONE` should be returned.
Practice w/SML Types

a) Explanations:

i) \( t_2 \) is more general than \( t_1 \). \( 'a \) can be replaced by string, \( 'b \) can be replaced by int

ii) \( t_2 \) is more general than \( t_1 \). \( 'a \) can be replaced by string, \( 'b \) can be replaced by string

iii) \( t_2 \) is not more general than \( t_1 \). \( 'a \) can be replaced by string, but \( 'b \) cannot be both string and int

iv) \( t_2 \) is more general than \( t_1 \). \( 'a \) can be replaced by int because foo is a synonym for \( \text{int} \times \text{int} \)

b) Solutions for each problem given, followed by the binding produced upon evaluation:

i) 

fun swap_pair \( (a, b) \) = 
  (b, a)

val swap_pair = fn : ('a * 'b) -> ('b * 'a)

ii) 

fun swap_pairs_list ps =
  case ps of
    [] => []
  | p :: ps' => swap_pair (p) :: swap_pairs_list(ps')

val swap_pairs_list = fn : ('a * 'b) list -> ('b * 'a) list

iii) 

fun size xs =
  case xs of
    [] => 0
  | x :: xs' => 1 + size (xs')

val size = fn : 'a list -> int

iv) 

fun contains \( (x, xs) \) =
  case xs of
    [] => false
  | x' :: xs' => (x = x') orelse contains (xs')

val contains = fn : ('a * 'a list) -> bool

v) 

fun remove_all \( (x, xs) \) =
  case xs of
    [] => []
  | x' :: xs' => if x = x'
    then remove_all (x, xs')
    else x' :: remove_all (x, xs')

val remove_all = fn : ('a * 'a list) -> 'a list
Programming w/simple datatypes

a) fun area sh =
  case sh of
    Circle(_ , r) => 3.14 * r * r
  | Square(_ , s) => s * s
  | Rectangle(_ , w , l) => w * l

b) fun quadrant_one_only shs =
  let
    fun is_quadrant_one sh =
      case sh of
        Circle((x , y), _) => x > 0.0 andalso y > 0.0
      | Square((x , y), _) => x > 0.0 andalso y > 0.0
      | Rectangle((x , y), _, _) => x > 0.0 andalso y > 0.0
    in
      case shs of
        [] => []
      | sh :: shs’ => if is_quadrant_one (sh)
        then sh :: quadrant_one_only (shs’)
        else quadrant_one_only (shs’)
    end

c) fun construct_squares xs =
  case xs of
    [] => []
  | x :: xs’ => Square((x , x), abs(x)) :: construct_squares(xs’)

d) datatype exp = Constant of int
  | Negate   of exp
  | Add      of exp * exp
  | Multiply of exp * exp

  fun eval (Constant i)  = i
  | eval (Add(e1, e2)) = (eval e1) + (eval e2)
  | eval (Negate e1)   = ~ (eval e1)
  | eval (Multiply(e1, e2)) = (eval e1) * (eval e2)
More complex programming with datatypes/pattern matching

1) fun add_whipped_cream ds =
   case ds of
       [] => []
     | d :: ds' => (d, WhippedCream) :: add_whipped_cream (ds')

2) fun ice_cream_feast fs =
   let
       fun help fs =
           case fs of
               [] => []
             | f :: fs' => IceCream(f, 1) :: help(fs')
   in
       Feast(help(fs))
   end

3) fun flatten d =
   case d of
       Feast (ds) =>
         let
             fun help (ds) =
                 case ds of
                     [] => []
                   | d :: ds' => flatten (d) @ help (ds')
         in
             help (ds)
         end
     | _ => [d]

4) fun num_scoops (d, f) =
   case d of
       IceCream (f, i) => i
     | Feast (ds) =>
         let
             fun help ds =
                 case ds of
                     [] => 0
                   | d' :: ds' => num_scoops(d', f) + help(ds')
         in
             help (ds)
         end
     | _ => 0
5) fun flavors d =
   case d of
     IceCream (f, _) => [f ^ " ice cream"]
   | Pie (f, _) => [f ^ " pie"]
   | Brownie (_) => ["brownie"]
   | WhippedCream => ["whipped cream"]
   | Feast (ds) =>
     let
       fun help ds =
         case ds of
           [] => []
         | d' :: ds' => flavors (d') @ help (ds')
     in
       help (ds)
     end

6) fun enough_ice_cream d =
   let
     fun plus_minus d =
       case d of
         IceCream (_, i) => i
       | Pie (_, i) => ~i
       | Brownie (i) => ~i
       | Feast (ds) =>
         let
           fun help ds =
             case ds of
               [] => 0
             | d' :: ds' => plus_minus(d') + help(ds')
         in
           help (ds)
         end
     in
     plus_minus (d) >= 0
   end
7)  fun dessert_equal (d1, d2) of
    case (d1, d2) of
      (IceCream(f1, i1), IceCream(f2, i2)) => f1 = f2 andalso i1 = i2
    |  (Pie(f1, i1), Pie(f2, i2)) => f1 = f2 andalso i1 = i2
    |  (Brownie(i1), Brownie(i2)) => i1 = i2
    |  (WhippedCream, WhippedCream) => true
    |  (Feast (ds1), Feast(ds2)) =>
        let
          fun help (ds1, ds2) =
            case (ds1, ds2) of
              ([], []) => true
            |  (d1' :: ds1', d2' :: ds2') =>
                dessert_equal(d1', d2') andalso help(ds1', ds2')
              |  _ => false
            in
            help (ds1, ds2)
        end
      |  _ => false

8)  fun no_whipped_cream_allowed d =
    case d of
      WhippedCream => NONE
    |  Feast (ds) =>
        let
          fun help ds =
            case ds of
              [] => []
            |  d' :: ds' =>
                let
                  val rest = help(ds')
                in
                  case no_whipped_cream_allowed(d') of
                    NONE => rest
                  |  SOME(e) => e :: rest
                end
          in
          help ds
        end
    in
    case result of
      [] => NONE
    |  _ => SOME (Feast (result))
    end
    |  _ => SOME(d)