CS 312 Recitation 8
ADT Examples: Stacks Queues, and Dictionaries

In this recitation, we will see more examples of structures and signatures that implement functional data structures.

Stacks and Queues

In **recitation 7**, we discussed stacks and queues. We repeat the signature for stacks here, adding a notation for representing the abstract contents.

```
  signature STACK =
    sig
      (* Overview: an 'a stack is a stack of elements of type 'a.
       * We write |e1, e2, ... en| to denote the stack with e1
       * on the top and en on the bottom. *)
      type 'a stack
      exception EmptyStack

      val empty : 'a stack
      val isEmpty : 'a stack -> bool

      val push : ('a * 'a stack) -> 'a stack
      val pop : 'a stack -> 'a stack
      val top : 'a stack -> 'a
      val map : ('a -> 'b) -> 'a stack -> 'b stack
      val app :  ('a -> unit) -> 'a stack -> unit
      (* note: app traverses from top of stack down *)
    end
```

Now we present a signature for queues; first-in, first-out data structures. Again, we introduce a notation for discussing the abstract contents of the queue.

```
signature QUEUE =
    sig
      (* Overview: an 'a queue is a FIFO queue of elements of type 'a.
       * We write <e1, e2, ... en> to denote the queue whose front
       * is e1 and whose back is en. Elements are enqueued at the back
       * and dequeued from the front. *)
      type 'a queue
      exception EmptyQueue

      val empty : 'a queue
      val isEmpty : 'a queue -> bool

      (* enqueue(x, q) is q with x enqueued at the back.
       * Example: enqueue(3, <1,2>) = <1,2,3> *)
      val enqueue : ('a * 'a queue) -> 'a queue

      (* dequeue(q) is q with its front element removed.
       * Requires: q is nonempty. *).
      val dequeue : 'a queue -> 'a queue

      (* front(q) is the element at the front. Requires: q is nonempty. *)
      val front : 'a queue -> 'a

      val map : ('a -> 'b) -> 'a queue -> 'b queue
      val app : ('a -> unit) -> 'a queue -> unit
    end
```

The simplest possible implementation for queues is to represent a queue via two stacks: one stack A on which to enqueue elements, and one stack B from which to dequeue elements. When dequeuing, if stack B is empty, then we reverse stack A and consider it the new stack B.

Here is an implementation for such queues. It uses the stack structure Stack, which is rebound to the name S inside the structure to avoid long identifier names.

```
structure Queue :> QUEUE =
    struct

      structure S = Stack

      type 'a queue = ('a S.stack * 'a S.stack)
      (* AF: The pair (|e1, e2, ... en|, |e'1, e'2, ..., e'n|) represents
       *     the queue <e'1, e'2, ..., e'n, en, ..., e2, e1>.
       *)

      exception EmptyQueue
```

```
      val empty : 'a queue = (S.empty, S.empty)
      fun isEmpty ((s1,s2):'a queue) =
        S.isEmpty (s1) andalso S.isEmpty (s2)

      fun enqueue (x:'a, (s1,s2):'a queue) : 'a queue =
        (S.push (x,s1), s2)

      fun rev (s:'a S.stack):'a S.stack = let
        fun loop (old:'a S.stack, new:'a S.stack):'a S.stack =
          if (S.isEmpty (old))
            then new
          else loop (S.pop (old), S.push (S.top (old),new))
      in
        loop (s,S.empty)
      end

      fun dequeue ((s1,s2):'a queue) : 'a queue =
        if (S.isEmpty (s2))
          then (S.empty, S.pop (rev (s1)))
                    handle S.EmptyStack => raise EmptyQueue
        else (s1,S.pop (s2))

      fun front ((s1,s2):'a queue):'a =
        if (S.isEmpty (s2))
          then S.top (rev (s1))
                   handle S.EmptyStack => raise EmptyQueue
        else S.top (s2)

      fun map (f:'a -> 'b) ((s1,s2):'a queue):'b queue =
        (S.map f s1, S.map f s2)

      fun app (f:'a -> unit) ((s1,s2):'a queue):unit =
        (S.app f s2;
         S.app f (rev (s1)))

         end
```

Fractions

Another simple data type is a fraction, a ratio of two integers. Here is a possible signature.

```
signature FRACTION =
    sig
        (* A fraction is a rational number *)
        type fraction
        (* first argument is numerator, second is denominator *)
        val make : int -> int -> fraction
        val numerator : fraction -> int
        val denominator : fraction -> int
        val toString : fraction -> string
        val toReal : fraction -> real
        val add : fraction -> fraction -> fraction
        val mul : fraction -> fraction -> fraction
    end
```

Here's one implementation of fractions -- what can go wrong here?

```
structure Fraction1 :> FRACTION =
    struct
        type fraction = { num:int, denom:int }
        (* AF: The record {num, denom} represents fraction (num/denom) *)

        fun make (n:int) (d:int) = {num=n, denom=d}

        fun numerator(x:fraction):int = #num x

        fun denominator(x:fraction):int = #denom x

        fun toString(x:fraction):string =
            (Int.toString (numerator x)) ^ "/" ^
            (Int.toString (denominator x))

        fun toReal(x:fraction):real =
            (Real.fromInt (numerator x)) / (Real.fromInt (denominator x))

        fun mul (x:fraction) (y:fraction) : fraction =
            make ((numerator x)*(numerator y))
                 ((denominator x)*(denominator y))

        fun add (x:fraction) (y:fraction) : fraction =
            make ((numerator x)*(denominator y) +
                  (numerator y)*(denominator x))
                 ((denominator x)*(denominator y))
    end
```

There are several problems with this implementation. First, we could give 0 as the denominator -- this is a bad fraction. Second, we're not reducing to smallest form. So we could overflow faster than we need to.

Third, we're not consistent with the signs of the numbers. Try `make ~1 ~1`.

We need to pick some representation invariant that describes how we're going to represent legal fractions. Here is one choice that tries to fix the bugs above.

```
structure Fraction2 :> FRACTION =
    struct
        type fraction = { num:int, denom:int }
        (* AF: represents the fraction num/denom
         * RI:
         *   (1) denom is always positive
         *   (2) always in most reduced form
         *)

        fun gcd (x:int) (y:int) : int =
        (* Algorithm due to Euclid:  for positive numbers x and y,
         * find the greatest-common-divisor. *)
            if (x = y) then x
            else if (x < y) then gcd x (y - x)
                 else gcd (x - y) y

        exception BadDenominator

        fun make (n:int) (d:int) : fraction =
            if (d < 0) then raise BadDenominator
            else let val g = gcd (abs n) (abs d)
                     val n2 = n div g
                     val d2 = d div g
                 in
                     if (d2 < 0) then {num = ~n2, denom = ~d2}
                     else {num = n2, denom = d2}
                 end

        fun numerator(x:fraction):int = #num x

        fun denominator(x:fraction):int = #denom x

        fun toString(x:fraction):string =
            (Int.toString (numerator x)) ^ "/" ^
            (Int.toString (denominator x))

        fun toReal(x:fraction):real =
            (Real.fromInt (numerator x)) / (Real.fromInt (denominator x))

        (* notice that we didn't have to re-code mul or add --
         * they automatically get reduced because we called
         * make instead of building the data structure directly.
         *)
        fun mul (x:fraction) (y:fraction) : fraction =
            make ((numerator x)*(numerator y))
                 ((denominator x)*(denominator y))

        fun add (x:fraction) (y:fraction) : fraction =
            make ((numerator x)*(denominator y) +
                  (numerator y)*(denominator x))
                 ((denominator x)*(denominator y))
    end
```

Dictionaries

A very useful type in programming is the **dictionary**. A dictionary is a mapping from strings to other values. A more general dictionary that maps from one arbitrary key type to another is usually called a **map** or an **associative array**, although sometimes “dictionary” is used for these as well. In any case, the implementation techniques are the same. Here's a signature for dictionaries:

```
signature DICTIONARY =
    sig
        (* An 'a dict is a mapping from strings to 'a.
           We write {k1=>v1, k2=>v2, ...} for the dictionary which
           maps k1 to v1, k2 to v2, and so forth. *)

        type key = string
        type 'a dict

        (* make an empty dictionary carrying 'a values *)
        val make : unit -> 'a dict

        (* insert a key and value into the dictionary *)
        val insert : 'a dict -> key -> 'a -> 'a dict

        (* Return the value that a key maps to in the dictionary.
         * Raise NotFound if there is not mapping for the key. *)
        val lookup : 'a dict -> key -> 'a
        exception NotFound

        end
```

Here's an implementation discussed in **recitation 6**.

```
structure FunctionDict :> DICTIONARY =
  struct
    type key = string
    type 'a dict = string -> 'a
    (* The function f represents the mapping in which x is mapped to
     * f(x), except for x such that f raises NotFound, which are not
     * in the mapping.
     *)
    exception NotFound
    fun make () = fn _ => raise NotFound
    fun lookup (d: 'a dict) (key: string) : 'a = d key
    fun insert (d:'a dict) (k:key) (x:'a) : 'a dict =
      fn k' => if k=k' then x else d k'
```

Here is another implementation: an association list **`[(key1,x1),...,(keyn,xn)]`**

```
structure AssocList :> DICTIONARY =
    struct
        type key = string
        type 'a dict = (key * 'a) list
        (* AF: The list [(k1,v1), (k2,v2), ...] represents the dictionary
         *     {k1 => v1, k2 => v2, ...}, except that if a key occurs
         *     multiple times in the list, only the earliest one matters.
         * RI: true.
         *)

        fun make():'a dict = []

        fun insert (d:'a dict) (k:key) (x:'a) : 'a dict = (k,x)::d

        exception NotFound

        fun lookup (d:'a dict) (k:key) : 'a =
            case d of
                [] => raise NotFound
              | ((k',x)::rest) =>
                    if (k = k') then x
                    else lookup rest k
                    end
```

This next implementation seems a little better for looking up values. Also note that the abstraction function does not need to specify what duplicate keys mean.

```
structure SortedAssocList :> DICTIONARY =
    struct
        type key = string
        type 'a dict = (key * 'a) list
        (* AF: The list [(k1,v1), (k2,v2), ...] represents the dictionary
         *     {k1 => v1, k2 => v2, ...}
         * RI: The list is sorted by key and each key occurs only once
         *     in the list. *)

        fun make():'a dict = []

        fun insert (d:'a dict) (k:key) (x:'a) : 'a dict =
            case d of
                [] => (k,x)::nil
              | (k',x')::rest =>
                    (case String.compare(k,k') of
                         GREATER => (k',x')::(insert rest k x)
                       | EQUAL => (k,x)::rest
                       | LESS => (k,x)::(k',x')::rest)

        exception NotFound

        fun lookup (d:'a dict) (k:key) : 'a =
            case d of
                [] => raise NotFound
              | ((k',x)::rest) =>
                    (case String.compare(k,k') of
                         EQUAL => x
                       | LESS => raise NotFound
                       | GREATER => lookup rest k)
    end
```

Here is another implementation of dictionaries. This one uses a binary tree to keep the data -- the hope is that inserts or lookups will be proportional to log(n) where n is the number of items in the tree.

```
structure AssocTree :> DICTIONARY =
    struct
        type key = string
        datatype 'a dict = Empty | Node of {key: key,datum: 'a,
                                            left: 'a dict,right: 'a dict}
        (* AF: Empty represents the empty mapping {}
         *     Node {key, datum, left, right} represents the union of the
```

```
     *      mappings {key => datum}, AF(left), and AF(right).
     * RI: for Nodes, data to the left have keys that
     *      are LESS than the datum and the keys of
     *      the data to the right. *)

    fun make():'a dict = Empty

    fun insert (d:'a dict) (k:key) (x:'a) : 'a dict =
        case d of
            Empty => Node{key=k, datum=x, left=Empty, right=Empty}
          | Node {key=k', datum=x', left=l, right=r} =>
                (case String.compare(k,k') of
                     EQUAL =>
                         Node{key=k, datum=x, left=l, right=r}
                   | LESS =>
                         Node{key=k',datum=x',left=insert l k x,
                              right=r}
                   | RIGHT =>
                         Node{key=k',datum=x',left=l,
                              right=insert r k x})

    exception NotFound

    fun lookup (d:'a dict) (k:key) : 'a =
        case d of
            Empty => raise NotFound
          | Node{key=k',datum=x, left=l, right=r} =>
                (case String.compare(k,k') of
                     EQUAL => x
                   | LESS => lookup l k
                   | RIGHT => lookup r k)
end
```