```
functor MkAllShortestPaths(Table : TABLE) : ALL_SHORTEST_PATHS =
struct open Table

  type vertex = key
  type edge = vertex * vertex
  type graph = vertex seq table
  type asp = graph

  (* Task 2.1 *)
  val makeGraph : edge Seq.seq -> graph = collect

  (* Task 2.2 *)
  fun numEdges (G : graph) : int =
      Seq.reduce op+ 0 (Seq.map Seq.length (range G))

  local
    (* the set of vertices in G; the union of the set of vertices with
     * outgoing edges and the set of vertices with incoming edges.
     *)
    fun vertices (G : graph) : set =
        let
          open Set
          (* set of vertices with out-edges *)
          val U = domain G
          (* set of vertices with in-edges *)
          val V = Seq.reduce union (empty ()) (Seq.map fromSeq (range G))
        in
          union (U, V)
        end
  in
    fun numVertices (G : graph) : int =
        Set.size (vertices G)
  end

  (* Task 2.3 *)
  fun outNeighbors (G : graph) (u : vertex) : vertex Seq.seq =
      case find G u
        of NONE => Seq.empty ()
         | SOME nbrs => nbrs

  (* Task 2.4 *)
  fun makeASP (G : graph) (u : vertex) : asp =
      let
        fun reverseEdges v =
            Seq.map (fn u => (u, v)) (outNeighbors G v)

        (* extendASP F evaluates to a vertex seq table
         *
         *   { v -> { u : u \in N^-(v) & u \in F } : v \in N^+(F) }
         *
         * Each out-neighbor v of some vertex in F maps to
         * a sequence of vertices U where each vertex u \in U
         * is in the in-neighbors of v as well as in F.
         *)
        fun extendASP F =
            collect (Seq.flatten (range (tabulate reverseEdges F)))

        exception MergeConflict
        val mergeNoConflict = merge (fn _ => raise MergeConflict)

        fun bfs F uASP =
```

```
            if Set.size F = 0 then uASP
            else let
              val newASP = erase (extendASP F, domain uASP)
              val uASP' = mergeNoConflict (uASP, newASP)
              val F' = domain newASP
            in
              bfs F' uASP'
            end

        val $ = Set.singleton
      in bfs ($u) (singleton (u, Seq.empty ()))
      end

  fun report uASP dest =
      let
        open Seq
        fun buildPaths paths v =
            let
              val parents = valOf (Table.find uASP v)
              val paths' = map (fn p => v::p) paths
            in
              if length parents = 0 then paths'
              else flatten (map (buildPaths paths') parents)
            end
      in
        case Table.find uASP dest
          of NONE => empty ()
           | SOME _ => map % (buildPaths (singleton []) dest)
      end
end
```

```
functor MkThesaurusASP (ASP : ALL_SHORTEST_PATHS where type vertex = string)
  : THESAURUS =
struct
  structure Seq = ASP.Seq
  open Seq

  type thesaurus = ASP.graph

  (* Task 3.1 *)
  fun make (pairs : (string * string seq) seq) : thesaurus =
      let
        fun makeEdges (w1, s) = map (fn w2 => (w1, w2)) s
        val E = flatten (map makeEdges pairs)
      in ASP.makeGraph E
      end

  (* Task 3.2 *)
  (* computes the total number of words in the thesaurus *)
  fun numWords th = ASP.numVertices th

  (* Task 3.3 *)
  (* evaluates to a sequence of synonyms for a given word *)
  fun synonyms th = ASP.outNeighbors th

  (* reports the shortest path from word1 to word2 as a sequence
   * of strings with word1 first and word2 last.
   * evaluates to NONE if no such path exists.
   *)
  fun query th word1 = ASP.report (ASP.makeASP th word1)

end
```