```
Dark gray:       Sample solution to the given task
Light gray:      Changes from the "original" skeleton
```

```
(* ------------------------       Interpreter for CL      ----------------------- *)

  open TextIO;

(* ------------------------            Data Types         ----------------------- *)

  datatype Bulk = Set | Bag | Seq

  datatype Operator = Union | Intersect | Minus

  datatype Coll = SetColl of int list
  |               BagColl of (int * int) list
  |               SeqColl of int list

  type Store   = (string * Coll) list

  datatype Expression = DECLexpr of string * Bulk
  |                     INSERTexpr of int * string
  |                     DELETEexpr of int * string
  |                     CLEARexpr of string
  |                     OPexpr of Operator * Expression * Expression
  |                     IDENTexpr of string
  |                     NUMBERexpr of int
  |                     HALTexpr

  datatype Token = TokOPENBR
  |                TokCLOSEBR
  |                TokBULK of Bulk
  |                TokOP of Operator
  |                TokDECL
  |                TokINSERT
  |                TokDELETE
  |                TokCLEAR
  |                TokIN
  |                TokIDENT of string
  |                TokEQUALS
  |                TokNUMBER of int
  |                TokHALT


(* ------------------------           Exceptions          ----------------------- *)

  exception NotImplemented
  exception Nth
  exception Lexical of string
  exception SyntaxError of Token list
  exception NoSuchIdent of string
  exception IllegalIdent of string
  exception UnknownBulk of string
  exception NotANumber of string


(* ------------------------        Lexical Analyser       ----------------------- *)

    (* function to input a line and return it as a list of chars as string values *)
    fun inputLineCs istream =
        let val char = inputN(istream,1)
        in
          if char = "\n" then []
          else char :: inputLineCs istream
        end

(* function to check whether there exists element in list for which f is true *)
fun exists f []             = false
|   exists f (first::rest) = f first orelse exists f rest

(* function to convert string to list of chars represented as strings *)
fun sexplode s = map Char.toString (explode s)

(* function to find nth element in a list *)
fun nth nil n      = raise Nth
|   nth (hd::tl) 0 = hd
|   nth (hd::tl) n = nth tl (n-1)

(* function to find the length of a list *)
fun length []       = 0
|   length (hd::tl) = 1 + length tl


(* functions to determine type of char in input *)

fun BadLetter char = (char < "a" orelse char > "z")
                     andalso (char < "A" orelse char > "Z")

fun IsASpace str = str <= " "

fun IsAlphanum "" = false
|   IsAlphanum ch = (ch >= "a" andalso ch <= "z")
                    orelse (ch >= "A" andalso ch <= "Z")
                    orelse (ch >= "0" andalso ch <= "9")

fun Solo sym = exists (fn x => x = sym) ["(", ")", "+", "-", "*"]

(* function to form "words" from "chars" in input by "glueing" chars *)
fun Glue accum (this::rest) =
       if IsASpace this then
          (if accum = "" then Glue "" rest
                         else accum::(Glue "" rest))
       else if (IsAlphanum accum <> IsAlphanum this) then
          (if accum = "" then Glue this rest
                         else accum::(Glue this rest))
       else if Solo this orelse Solo accum then
          (if accum = "" then Glue this rest
                         else accum::(Glue this rest))
       else Glue (accum^this) rest
|   Glue accum nil = if accum = "" then [] else [accum]


(* functions to construct a number from digits *)

fun IsNumber s = not(exists (fn char => char < "0" orelse char > "9") (sexplode s))

fun MakeNumber digits =
   let fun MakeNumber'(d::drest, result) =
              MakeNumber'(drest, result * 10 + ord(d) - ord(#"0"))
       |   MakeNumber'(nil, result) = result
   in  MakeNumber'(explode digits, 0)
   end

(* function to check whether a word is a legal identifier *)
fun IsIdent(s) = not(exists BadLetter (sexplode s))
```

```
    (* function to generate tokens from words *)
    fun MakeToken("(")       = TokOPENBR
    |   MakeToken(")")       = TokCLOSEBR
    |   MakeToken("+")       = TokOP(Union)
    |   MakeToken("*")       = TokOP(Intersect)
    |   MakeToken("-")       = TokOP(Minus)
    |   MakeToken("val")     = TokDECL
    |   MakeToken("insert")  = TokINSERT
    |   MakeToken("delete")  = TokDELETE
    |   MakeToken("clear")   = TokCLEAR
    |   MakeToken("in")      = TokIN
    |   MakeToken("=")       = TokEQUALS
    |   MakeToken("halt")    = TokHALT
    |   MakeToken("set")     = TokBULK(Set)
    |   MakeToken("bag")     = TokBULK(Bag)
    |   MakeToken("seq")     = TokBULK(Seq)
    |   MakeToken(s)         = if IsNumber(s) then TokNUMBER(MakeNumber s)
                              else if IsIdent(s) then TokIDENT(s)
                              else raise Lexical(s)

    (* functions to perform lexical analysis of input, generating token list  *)
    (* the list of input characters are first "glued" into words and from the *)
    (* words, lexical tokens of the language are generated                    *)

    fun Lex(input) = Glue "" input

    fun lexical () =
        let val LexStrings = Lex (inputLineCs(stdIn))
        in
          map MakeToken LexStrings
        end


(* -------------------------            Parser            ----------------------- *)

    fun ParseExpr(TokDECL::TokIDENT(ident)::TokEQUALS::TokBULK(bulk)::rest) =
            (DECLexpr(ident,bulk), rest)

    |   ParseExpr(TokINSERT::TokNUMBER(number)::TokIN::TokIDENT(ident)::rest) =
             (INSERTexpr(number,ident), rest)

    |   ParseExpr(TokDELETE::TokNUMBER(number)::TokIN::TokIDENT(ident)::rest) =
             (DELETEexpr(number,ident), rest)

    |   ParseExpr(TokCLEAR::TokIDENT(ident)::rest) =
             (CLEARexpr(ident), rest)

    |   ParseExpr(TokHALT::rest) =
             (HALTexpr, rest)

    |   ParseExpr(TokIDENT(ident)::rest) =
             ParseExprTail(IDENTexpr(ident), rest)

    |   ParseExpr(TokOPENBR::rest) =
             ( case ParseExpr(rest) of
                 ( Expr,TokCLOSEBR::rest' ) => ParseExprTail(Expr,rest')
               | ( _,rest') => raise SyntaxError(rest')
             )
```

```
    (** task 2 **)
    |   ParseExpr(TokDECL::TokIDENT(ident)::TokEQUALS::TokIDENT(s)::rest) =
             raise UnknownBulk(s)

    |   ParseExpr(TokINSERT::TokIDENT(s)::TokIN::TokIDENT(ident)::rest) =
             raise NotANumber(s)

    |   ParseExpr(TokDELETE::TokIDENT(s)::TokIN::TokIDENT(ident)::rest) =
             raise NotANumber(s)

    |   ParseExpr(junk) = raise SyntaxError(junk)

    and

        ParseExprTail(Expr,TokOP(Op)::rest) =
             let val (Expr',rest') = ParseExpr(rest)
             in (OPexpr(Op,Expr,Expr'),rest')
             end

    |   ParseExprTail(Expr,rest) = (Expr,rest)


    fun parser () =
       (  case ParseExpr(lexical()) of
             (tree,[])   => tree
          |  (tree,rest) => raise SyntaxError(rest)
       )


(* -------------------------     Collection Operations     ----------------------- *)

    fun isMember elm nil = false
    |   isMember elm (hd::tl) = (hd = elm) orelse (isMember elm tl)

    (** task 1 **)
    fun setToBag nil = []
    |   setToBag (hd::tl) = (hd,1)::setToBag tl

    (* set operations *)
    fun setUnion nil lst2 = lst2
    |   setUnion (hd::tl) lst2 =
          if isMember hd lst2 then setUnion tl lst2
                              else hd::setUnion tl lst2

    fun setIntersect nil lst2 = nil
    |   setIntersect (hd::tl) lst2 =
          if isMember hd lst2 then hd::setIntersect tl lst2
                              else setIntersect tl lst2

    fun setMinus nil lst2 = nil
    |   setMinus (hd::tl) lst2 =
          if isMember hd lst2 then setMinus tl lst2
                              else hd::setMinus tl lst2


    (* bag helper functions *)
    fun memberCnt elm nil = 0
    |   memberCnt elm ((x,cnt)::tl) =
          if (x=elm) then cnt
                     else memberCnt elm tl
```

```
    (* convert bag to set, stripping off the counts *)
    fun member (x,cnt) = x
    fun bagToSet lst = map member lst

    (* reduce a list *)
    fun reduce f nil a = a
    |   reduce f (hd::tl) a = f (hd, reduce f tl a)


    (* combine two bags                                                    *)
    (* (fNewCnt computes the new count for each distinct member of         *)
    (* lst1 and lst2)                                                      *)
    (*                                                                     *)
    (* how it works:                                                       *)
    (* - lst1 and lst2 are converted to sets and the union is taken        *)
    (*    => "master list" with all distinct members of both bags          *)
    (* - master list is reduced, building up the resulting bag along the way  *)

    fun combine fNewCnt lst1 lst2 = reduce
                (fn (x,a) => (x, fNewCnt(memberCnt x lst1, memberCnt x lst2))::a)
                (setUnion (bagToSet lst1) (bagToSet lst2))
              []


    (* remove all elements with count zero *)
    fun clean [] = []
    |   clean ((x,cnt)::tl) = if (cnt=0) then clean tl else (x,cnt)::(clean tl)

    fun max (a,b) = if (a > b) then a else b
    fun min (a,b) = if (a < b) then a else b


    (* bag operations *)
    fun bagUnion lst1 lst2 =
            combine (fn (c1,c2) => max(c1,c2)) lst1 lst2

    fun bagIntersect lst1 lst2 =
            clean (combine (fn (c1,c2) => min(c1,c2)) lst1 lst2)

    fun bagMinus lst1 lst2 =
            clean (combine (fn (c1,c2) => if (c1-c2)>0 then c1-c2 else 0) lst1 lst2)
```

```
(* ------------------------------------------------------------------------- *)
```

```
    fun collInsert (elm, SetColl(set)) =
           if (isMember elm set) then SetColl(set)
                                 else SetColl(elm::set)

    |   collInsert (elm, BagColl(bag)) =
           let fun insert e nil = [(e, 1)]
               |   insert e ((num, cnt)::tl) =
                      if (e=num) then (e, cnt+1)::tl
                                 else (num, cnt)::(insert e tl)
           in BagColl(insert elm bag) end

    |   collInsert (elm, SeqColl(seq)) = SeqColl(seq@[elm])


    fun setDelete elm nil = nil
    |   setDelete elm (hd::tl) =
          if (hd = elm) then tl
                        else hd::(setDelete elm tl)
```

```
    fun bagDelete elm nil = nil
    |   bagDelete elm ((num, cnt)::tl) =
          if (num = elm) then if (cnt > 1) then (num, cnt-1)::tl
                                           else tl
                         else (num, cnt)::(bagDelete elm tl)

    fun seqDelete elm nil = nil
    |   seqDelete elm (hd::nil) = if (hd = elm) then nil else (hd::nil)
    |   seqDelete elm (hd::tl) = hd::(seqDelete elm tl)

    fun collDelete (elm, SetColl(set)) = SetColl(setDelete elm set)
    |   collDelete (elm, BagColl(bag)) = BagColl(bagDelete elm bag)
    |   collDelete (elm, SeqColl(seq)) = SeqColl(seqDelete elm seq)


    fun convert(SetColl(set), BagColl(bag)) =
          ( BagColl(setToBag(set)), BagColl(bag) )
    |   convert(BagColl(bag), SetColl(set)) =
          ( BagColl(bag), BagColl(setToBag(set)) )
    |   convert(coll1, coll2) = (coll1, coll2)


    fun union(SetColl(set1), SetColl(set2)) =
          SetColl(setUnion set1 set2)
    |   union(BagColl(bag1), BagColl(bag2)) =
          BagColl(bagUnion bag1 bag2)
    |   union(_, _) = raise NotImplemented

    fun intersect(SetColl(set1), SetColl(set2)) =
          SetColl(setIntersect set1 set2)
    |   intersect(BagColl(bag1), BagColl(bag2)) =
          BagColl(bagIntersect bag1 bag2)
    |   intersect(_, _) = raise NotImplemented

    fun minus(SetColl(set1), SetColl(set2)) =
          SetColl(setMinus set1 set2)
    |   minus(BagColl(bag1), BagColl(bag2)) =
          BagColl(bagMinus bag1 bag2)
    |   minus(_, _) = raise NotImplemented
```

```
(* ------------------------             Storage           ------------------------ *)
```

```
    fun modify action elm ident nil = nil
    |   modify action elm ident ((id, coll)::tl) =
          if (id = ident) then (id, (action(elm,coll)))::tl
                          else (id, coll)::(modify action elm ident tl)

    fun insert elm ident S = modify collInsert elm ident S
    fun delete elm ident S = modify collDelete elm ident S

    fun lookup ident nil = raise NoSuchIdent(ident)
    |   lookup ident ((id, coll)::tl) =
           if (ident = id) then coll
                           else lookup ident tl

    fun check ident nil = (ident <> "it")
    |   check ident ((id, _)::tl) = (ident <> id) andalso check ident tl
```

```
    (** task 3 **)
    fun clear ident nil = raise NoSuchIdent(ident)
    |   clear ident ((id, coll)::tl) =
          if (ident = id) then tl
                          else (id, coll)::clear ident tl


(* ------------------------             Output            ----------------------- *)

    fun printSet nil = ()
    |   printSet (hd::tl) = ( print(Int.toString(hd:int)); print(" "); printSet tl )

    fun printBag nil = ()
    |   printBag ((elm, cnt)::tl) =
          ( print("("); print(Int.toString(elm)); print(",");
            print(Int.toString(cnt)); print(") "); printBag tl )

    fun printColl (SetColl(lst)) = printSet lst
    |   printColl (BagColl(lst)) = printBag lst
    |   printColl (SeqColl(lst)) = printSet lst

    fun printBulk(ident, b) =
          (print("> "); print(ident);
           print(" = [ "); printColl b; print("]\n"))

    fun printBinding ident S = (printBulk(ident,(lookup ident S)); S)


(* ------------------------            Evaluator          ----------------------- *)

    fun create ident bulk =
        ( case bulk of
            Set => (ident,SetColl([]))
          | Bag => (ident,BagColl([]))
          | Seq => (ident,SeqColl([]))
        )


    fun evaluateOp (IDENTexpr(ident)) S =
          lookup ident S
    |   evaluateOp (OPexpr(Union, exp1, exp2)) S =
          union(convert((evaluateOp exp1 S), (evaluateOp exp2 S)))
    |   evaluateOp (OPexpr(Intersect, exp1, exp2)) S =
          intersect(convert((evaluateOp exp1 S), (evaluateOp exp2 S)))
    |   evaluateOp (OPexpr(Minus, exp1, exp2)) S =
          minus(convert((evaluateOp exp1 S), (evaluateOp exp2 S)))
    |   evaluateOp (_) (_) = raise NotImplemented


    fun evaluate (HALTexpr) S = (true,S)
    |   evaluate (DECLexpr(ident,bulk)) S =
          if check ident S then
             ( print("> "); print(ident); print(" = [ ]\n");
               (false, (create ident bulk)::S)
             )
          else raise IllegalIdent(ident)
    |   evaluate (INSERTexpr(num, ident)) S =
          (false, printBinding ident (insert num ident S))
    |   evaluate (DELETEexpr(num, ident)) S =
          (false, printBinding ident (delete num ident S))
```

```
    (** task 3 **)
    |   evaluate (CLEARexpr(ident)) S =
          (print("> clearing "); print(ident); print("\n"); (false, clear ident S) )
    |   evaluate (IDENTexpr(ident)) S =
          (false, printBinding ident S)
    |   evaluate (OPexpr(operation, exp1, exp2)) S =
          (printBulk("it", evaluateOp (OPexpr(operation, exp1, exp2)) S); (false, S) )
    |   evaluate (_) S = raise NotImplemented


(* ------------------------           Interpeter          ----------------------- *)

    fun interpreter(S) =
        ( case evaluate (parser()) S
            handle
               NotImplemented =>
                  ( print("> sorry, not implemented\n"); (false, S) )
             |  Overflow =>
                  ( print("> number too big\n"); (false, S) )
             (** task 2 **)
             |  NoSuchIdent(s) =>
                  ( print("> Collection "); print(s); print(" does not exist!\n");
                    (false, S) )
             |  UnknownBulk(s) =>
                  ( print("> Bulk "); print(s); print(" does not exist!\n");
                    (false, S) )
             |  NotANumber(s) =>
                  ( print("> "); print(s); print(" is not a number\n");
                    (false, S) )
             |  SyntaxError(junk) =>
                  ( print("> Syntax error: Not a valid CL statement!\n");
                    (false, S) )
             |  IllegalIdent(s) =>
                  ( print("> "); print(s); print(" is already defined!\n");
                    (false, S) )
             |  Lexical(s) =>
                  ( print("> "); print(s); print(" is an illegal character!\n");
                    (false, S) )
             |  Nth =>
                  ( print("> "); print("Internal error (Nth exception)!!\n");
                    (false, S) )
         of
            (true,S)   => print("\n> goodbye \n")
          | (false,S') => interpreter(S')
        )

    fun CL () = interpreter(nil)


(* ------------------------             Usage             ----------------------- *)
(*                                                                               *)
(*   val "name" = [set|bag|seq]          -  creates a new binding                *)
(*   insert "number" in "name"           -  inserts the number into the binding  *)
(*   delete "number" in "name"           -  deletes the number in the binding    *)
(*   clear "name"                        -  removes the binding                  *)
(*   "name"                              -  displays the binding                 *)
(*   expression                          -  evaluates the expression             *)
(*   halt                                -  stops the interpreter                *)
(*                                                                               *)
(* ------------------------------------------------------------------------------ *)
```