```
exception NotImplemented of string
exception TypeError of Ex.Expression * string

fun tc (TE: Ty.TypeScheme TE.Environment, exp: Ex.Expression): Ty.Type =
 (case exp of
    Ex.BOOLexpr b => Ty.mkTypeBool()
  | Ex.NUMBERexpr _ => Ty.mkTypeInt()
  | Ex.SUMexpr(e1,e2)  => checkIntBin(TE,e1,e2)
  | Ex.DIFFexpr(e1,e2) => checkIntBin(TE,e1,e2)
  | Ex.PRODexpr(e1,e2) => checkIntBin(TE,e1,e2)
  | Ex.LISTexpr [] =>
       let val new = Ty.freshTyvar ()
        in Ty.mkTypeList(Ty.mkTypeTyvar  new)
       end
  | Ex.LISTexpr(e::es) => tc (TE, Ex.CONSexpr(e,Ex.LISTexpr es))
  | Ex.CONSexpr(e1,e2) =>
      let val t1 = tc(TE, e1)
          val t2 = tc(TE, e2)
          val new = Ty.freshTyvar ()
          val newt= Ty.mkTypeTyvar new
          val t2' = Ty.mkTypeList newt
          val S1 = Unify.unify(t2, t2')
                   handle Unify.Unify=>
                   raise TypeError(e2,"expected list type")

          val S2 = Unify.unify(S1 on newt,S1 on t1)
                   handle Unify.Unify=>
                   raise TypeError(exp,"element and list have different types")
       in S2 on (S1 on t2)
      end
  | Ex.EQexpr _ => raise NotImplemented "(equality)"
  | Ex.CONDexpr _ => raise NotImplemented "(conditional)"
  | Ex.DECLexpr(x,e1,e2) =>
       let val t1 = tc(TE,e1);
           val typeScheme = Ty.close(t1)
        in tc(TE.declare(x,typeScheme,TE), e2)
       end
  | Ex.RECDECLexpr _ => raise NotImplemented "(rec decl)"
  | Ex.IDENTexpr x   =>
       (Ty.instance(TE.retrieve(x,TE))
       handle TE.Retrieve _ =>
        raise TypeError(exp,"identifier " ^ x ^ " not declared"))
  | Ex.LAMBDAexpr _  => raise NotImplemented "(function)"
```

```
    | Ex.APPLexpr _ => raise NotImplemented    "(application)"

   )handle Unify.NotImplemented msg => raise NotImplemented msg

  and checkIntBin(TE,e1,e2) =
    let val t1 = tc(TE,e1)
        val _  = Ty.unTypeInt t1
                 handle Ty.Type=> raise TypeError(e1,"expected int")
        val t2 = tc(TE,e2)
        val _  = Ty.unTypeInt t2
                 handle Ty.Type=> raise TypeError(e2,"expected int")
     in Ty.mkTypeInt()
    end;

  fun typecheck(e) = tc(TE.emptyEnv,e)

end; (*TypeChecker*)
```

Then we extend the Type functor to match the TYPE signature:

```
functor Type():TYPE =
struct
  ...
  datatype TypeScheme = FORALL of tyvar list * Type

  fun instance(FORALL(tyvars,ty))=
  let val old_to_new_tyvars = map (fn tv=>(tv,freshTyvar())) tyvars
      exception Find;
      fun find(tv,[])= raise Find
      |   find(tv,(tv',new_tv)::rest)=
          if tv=tv' then new_tv else find(tv,rest)
      fun ty_instance INT = INT
      |   ty_instance BOOL = BOOL
      |   ty_instance (LIST t) = LIST(ty_instance t)
      |   ty_instance (TYVAR tv) =
             TYVAR(find(tv,old_to_new_tyvars)
                   handle Find=> tv)

  in
     ty_instance ty
  end
  fun close(ty)=
  let fun fv(INT) = []
      |   fv(BOOL)= []
      |   fv(LIST t) = fv(t)
      |   fv(TYVAR tv) = [tv]
   in FORALL(fv ty,ty)
  end

end;
```

Finally, the system is re-built as in Version 2, except that we have to provide and link in an `Environment` functor which matches `ENVIRONMENT`.

Exercise 7 Extend Version 4 with `if .. then .. else`. (This extension has no subtle implications for the type checking.)

Exercise 8 [For the extra keen] Extend Version 4 to cope with lambda abstraction (`fn`) and application. First, you have to introduce arrow types with constructors and destructors. Then you have to change the type of `close` so that it takes two arguments, namely a type environment and a type. It should return the type scheme that is obtained by quantifying on all the type variables that occur in the type but do not occur free in the type environment.

Then you can modify the type checker. When you type check a lambda abstraction, you just bind the formal parameter to the trivial type scheme which is just a fresh type variable (no quantified variables). Thus the type environment can now contain type schemes with free type variables.

An application `tc(TE,e)` now yields two arguments, namely a type t and a substitution S; the idea is that if you apply the substitution S to the type environment `TE`, which now can contain free type variables, the expression `e` has the type t. When an expression consists of more than one subexpression, the type environment gradually becomes more and more specific by applying the substitutions produced by the checking of the subexpressions one by one. Moreover, the substitution returned from the whole expression is the composition of these individual substitutions. (You have to extend the `TYPE` signature (and the `Type` functor) with composition of substitutions.

Finally, you can extend the unification algorithm to cope with arrow types. (This will also use composition of substitutions.)

4.5 Acknowledgement

The parser and evaluator and all the signatures related to them are due to Nick Rothwell.

Appendix A: The bare Interpreter

```
(* interp1.sml : VERSION 1: the bare interpreter *)

signature INTERPRETER=
   sig
      val interpret: string -> string
      val eval: bool ref
      and tc  : bool ref
   end;

                  (* syntax *)

signature EXPRESSION =
   sig
      datatype Expression =
         SUMexpr of Expression * Expression   |
         DIFFexpr of Expression * Expression   |
         PRODexpr of Expression * Expression   |
         BOOLexpr of bool   |
         EQexpr of Expression * Expression   |
         CONDexpr of Expression * Expression * Expression   |
         CONSexpr of Expression * Expression   |
         LISTexpr of Expression list   |
         DECLexpr of string * Expression * Expression   |
         RECDECLexpr of string * Expression * Expression   |
         IDENTexpr of string   |
         LAMBDAexpr of string * Expression   |
         APPLexpr of Expression * Expression   |
         NUMBERexpr of int
   end


              (* parsing *)

signature PARSER =
   sig
      structure E: EXPRESSION

      exception Lexical of string
      exception Syntax of string
      val parse: string -> E.Expression
   end


                        (* environments *)

signature ENVIRONMENT =
   sig
      type 'object Environment

      exception Retrieve of string

      val emptyEnv: 'object Environment
      val declare: string * 'object * 'object Environment
          -> 'object Environment
      val retrieve: string * 'object Environment -> 'object
   end

                        (* evaluation *)
signature VALUE =
   sig
      type Value
      exception Value

      val mkValueNumber: int -> Value
          and unValueNumber: Value -> int

      val mkValueBool: bool -> Value
          and unValueBool: Value -> bool

      val ValueNil: Value
      val mkValueCons: Value * Value -> Value
          and unValueHead: Value -> Value
          and unValueTail: Value -> Value

      val eqValue: Value * Value -> bool
      val printValue: Value -> string
   end


signature EVALUATOR =
   sig
```