```
(*  Starting point for CS 442/642 W13 Assignment 3

   These datatype declarations form an abstract syntax for Milner
   expressions.  All inputs to the type-inferencer are assumed to be
   syntactically valid Milner programs.  *)

datatype prim = Add | Neg | Mult | Div | And | Or | Not | Eq | Lt | Gt

datatype milner = Var of string
                | Abs of string * milner
                | App of milner * milner
                | If of milner * milner * milner
                | Let of string * milner * milner
                | Fix of string * milner
                | Int of int
                | Bool of bool
                | Prim of prim
                | Raise of milner                       (* Part D *)
                | Letex of string * milner              (* Part D *)
                | Handle of milner * milner * milner (* Part D *)

datatype mtype = TInt | TBool | TVar of string | TAbs of string * mtype
               | Arrow of mtype * mtype | TExc (*this is the exception type*)

(* Generating Type Variables:
   We reserve varables of the form Zn, n an integer, n>=0. *)
val counter = ref 0

fun newtype () =
   "Z" ^ Int.toString(!counter before counter := !counter + 1)


(* Environment: mapping from names to types *)

type env = string -> mtype


(* Start of Part A *)
fun pptype TInt = "int"
|   pptype TBool = "bool"
|   pptype TExc = "expn"
|   pptype (TVar x) = "'" ^ x
|   pptype (Arrow(a, b)) = "(" ^ pptype(a) ^ " -> " ^ pptype(b) ^ ")"
|   pptype (TAbs (x, e)) = "(FORALL " ^ x ^ " " ^ pptype(e) ^ ")"
(* End of Part A *)

(* Start of Part B, Part D *)
fun updateEnv ENV (x:string) (t:mtype) name =
   if x = name then t else (ENV name)

exception LookupFailedNameNotFound

fun emptyenv(x:string):mtype = raise LookupFailedNameNotFound

(*
1. addition: add: int -> (int -> int)
2. numeric negation: neg: int -> int
3. multiplication: mult: int -> (int -> int)
4. division: div: int -> (int -> int)
5. conjunction: and: bool -> (bool -> bool)
6. disjunction: or: bool -> (bool -> bool)
7. logical negation: not: bool -> bool
8. numeric equality: eq: int -> (int -> bool)
9. less than: lt: int -> (int -> bool)
10. greater than: gt: int -> (int -> bool)
11. raise exception: raise: FORALL alpha (alpha -> alpha)  (* Part D *)
*)

val intToIntInt = Arrow(TInt, Arrow(TInt, TInt))
val intToInt = Arrow(TInt, TInt)
val boolToBool = Arrow(TBool, TBool)
val boolToBoolBool = Arrow(TBool, Arrow(TBool, TBool))
val intToIntBool = Arrow(TInt, Arrow(TInt, TBool))
val generalType =
   let
      val newVarStr = (newtype())
   in
      TAbs(newVarStr, (TVar newVarStr))
   end

val initenv = updateEnv (updateEnv (updateEnv (updateEnv (updateEnv (updateEnv (updateEnv
(updateEnv (updateEnv (updateEnv (updateEnv emptyenv "add" intToIntInt) "neg" intToInt) "mult"
intToIntInt) "div" intToIntInt) "and" boolToBoolBool) "or" boolToBoolBool) "not" boolToBool) "eq"
intToIntBool) "lt" intToIntBool) "gt" intToIntBool) "raise" generalType;

fun emptySubst(t:mtype) = t

```

```
85  fun subst t a TInt = TInt
86  |   subst t a TBool = TBool
87  |   subst t a TExc = TExc
88  |   subst t a (TVar b) =
89        if a=b then t else (TVar b)
90  |   subst t a (TAbs (typevar_string, mtype)) =
91        if a=typevar_string then   (TAbs (typevar_string, mtype)) else (TAbs (typevar_string,
    (subst t a mtype)))
92  |   subst t a (Arrow (t1, t2)) = (Arrow ((subst t a t1), (subst t a t2)))
93
94
95  exception OccursCheckFailedCircularity
96  exception UnificationFailedTypeMismatch
97
98  fun occurs_check(t, TVar a) = (a = t)
99  |   occurs_check(t, TAbs (a, mtype)) = false
100 |   occurs_check(t, Arrow(a1, a2)) = occurs_check(t, a1) orelse occurs_check(t, a2)
101 |   occurs_check(t, (TBool|TInt|TExc)) = false
102
103 (* TAbs is impossible to occur in unify because lookup will never return a TAbs type so t in (s,
    t) will never contain a TAbs, and subst is therefore not possible to have a TAbs as parameter. So
    don't need to worry about e.g. s1 o A - a bound var in A will never be replaced. For example,
    (for all a, a->b->a)[c/a] => (for all a, a->b->a); but (for all a, a->b->a)[c/b] => (for all a, a-
    >c->a) *)
104 fun unify(TInt, TInt) = emptySubst
105 |   unify(TBool, TBool) = emptySubst
106 |   unify(TExc, TExc) = emptySubst
107 |   unify(TVar a, t) =
108       if TVar a = t then emptySubst
109       else if occurs_check(a, t) then raise OccursCheckFailedCircularity
110       (*else if t = TExc then raise UnificationFailedTypeMismatch*)
111       else subst t a (* substitute t for a: [t/a] *)
112 |   unify(t, TVar a) = unify(TVar a, t)
113 |   unify(Arrow(t1, t2), Arrow(t3, t4)) =
114       let
115          val s1=unify(t1, t3)
116          val s2=unify(s1 t2, s1 t4)
117       in
118          s2 o s1
119       end
120 |   unify(_, _) = raise UnificationFailedTypeMismatch
121
122 (* a list of TVars (maybe TInts and TBools) *)
123 val freevarlist:mtype list = [];
124
125 fun eq (tar:mtype) (elem:mtype) = (tar = elem)
126
127 fun exists f l =
128    if (null l) then false
129    else if (f (hd l)) then true
130    else exists f (tl l)
131
132 fun pushTVars TVAR FREEVARLIST =
133    if (exists (eq TVAR) FREEVARLIST)=false then TVAR :: FREEVARLIST
134    else FREEVARLIST
135
136 (* output a list of variables that not occuring free (string in TVars) for use of quantify *)
137 fun getNotOccuringFreeVars (TVar a) FREEVARLIST =
138       if (exists (eq (TVar a)) FREEVARLIST) then []
139       else [a]
140 |   getNotOccuringFreeVars (TInt|TBool|TExc) FREEVARLIST = []
141 (* the following case is impossible *)
142 |   getNotOccuringFreeVars (TAbs (x, e)) FREEVARLIST = x :: (getNotOccuringFreeVars e FREEVARLIST)
143 |   getNotOccuringFreeVars (Arrow(t1, t2)) FREEVARLIST = (getNotOccuringFreeVars t1 FREEVARLIST)
    @ (getNotOccuringFreeVars t2 FREEVARLIST)
144
145 fun quantify t notoccuringfreevarlist =
146    if (null notoccuringfreevarlist) then t
147    else (TAbs (hd notoccuringfreevarlist, quantify t (tl notoccuringfreevarlist)))
148
149 fun disquantify (TAbs (x, e)) = ((subst (TVar (newtype())) x) (disquantify e))
150 |   disquantify (TVar a) = (TVar a)
151 |   disquantify TInt = TInt
152 |   disquantify TBool = TBool
153 |   disquantify TExc = TExc
154 |   disquantify (Arrow (t1, t2)) = (Arrow (t1, t2))
155
156 fun typeAssert (t1:mtype,t2:mtype) =
157    if t1 = t2 then true
158    else raise UnificationFailedTypeMismatch
159
160 (* Helper for W *)
161 fun W' A F (Var x) =
162    let
163       val sigma = (A x)
164       val t = (disquantify sigma)
165    in
```

```
166         (emptySubst, t)
167       end
168     | W' A F (Abs (x, e)) =
169         let
170           val newVar = (TVar (newtype()))
171           val (s, t) = (W' (updateEnv A x newVar) (pushTVars newVar F) e)
172         in
173           (s, Arrow(s newVar, t))
174         end
175     | W' A F (App (e1, e2)) =
176         let
177           val newVar = (TVar (newtype()))
178           val (s1, t1) = (W' A F e1)
179           val (s2, t2) = (W' (s1 o A) (map s1 F) e2)
180           val s3 = unify(s2 t1, Arrow(t2, newVar))
181         in
182           (s3 o s2 o s1, s3 newVar)
183         end
184     | W' A F (If (e1, e2, e3)) =
185         let
186           val (s1, t1) = (W' A F e1)
187           val s2 = unify(TBool, t1)
188           val (s3, t3) = (W' (s2 o s1 o A) (((map s2) o (map s1)) F) e2)
189           val (s4, t4) = (W' (s3 o s2 o s1 o A) (((map s3) o (map s2) o (map s1)) F) e3)
190           val s5 = unify(s4 t3, t4)
191         in
192           (s5 o s4 o s3 o s2 o s1, s5 t4)
193         end
194     | W' A F (Fix (x, e)) =
195         let
196           val newVar = (TVar (newtype()))
197           val (s1, t1) = (W' (updateEnv A x newVar) (pushTVars newVar F) e)
198           val s2 = unify(s1 newVar, t1)
199         in
200           (s2 o s1, (s2 o s1) newVar)
201         end
202     | W' A F (Let (x, e1, e2)) =
203         let
204           val (s1, t1) = (W' A F e1)
205           val sigma = (quantify t1 (getNotOccuringFreeVars t1 (map s1 F)))
206           val (s2, t2) = (W' (updateEnv (s1 o A) x sigma) (map s1 F) e2)
207         in
208           (s2 o s1, t2)
209         end
210     | W' A F (Handle (e1, e2, e3)) =
211         let
212           val (s1, t1) = (W' A F e1)
213           val isSame = typeAssert (TExc, t1) (* if t1 is not TExc, an exception will raise *)
214           val s2 = unify(TExc, t1)
215           val (s3, t3) = (W' (s2 o s1 o A) (((map s2) o (map s1)) F) e2)
216           val (s4, t4) = (W' (s3 o s2 o s1 o A) (((map s3) o (map s2) o (map s1)) F) e3)
217           val s5 = unify(s4 t3, t4)
218         in
219           (s5 o s4 o s3 o s2 o s1, s5 t4)
220         end
221     | W' A F (Letex (x, e)) =
222         let
223           val (s, t) = (W' (updateEnv A x TExc) F e)
224         in
225           (s, t)
226         end
227     | W' A F (Int (i:int)) = (emptySubst, TInt)
228     | W' A F (Bool (b:bool)) = (emptySubst, TBool)
229     | W' A F (Prim Add) = (W' A F (Var "add"))
230     | W' A F (Prim Neg) = (W' A F (Var "neg"))
231     | W' A F (Prim Mult) = (W' A F (Var "mult"))
232     | W' A F (Prim Div) = (W' A F (Var "div"))
233     | W' A F (Prim And) = (W' A F (Var "and"))
234     | W' A F (Prim Or) = (W' A F (Var "or"))
235     | W' A F (Prim Not) = (W' A F (Var "not"))
236     | W' A F (Prim Eq) = (W' A F (Var "eq"))
237     | W' A F (Prim Lt) = (W' A F (Var "lt"))
238     | W' A F (Prim Gt) = (W' A F (Var "gt"))
239     | W' A F (Raise e) =
240         let
241           val (s1, t1) = (W' A F e)
242           val isSame = typeAssert (TExc, t1) (* if t1 is not TExc, an exception will raise *)
243         in
244           (W' A F (Var "raise"))
245         end
246
247   (* W:  Accepts the arguments A (environment) and E (expression).
248       Returns the type of E in A.  *)
249
250   fun W A E = (W' A freevarlist E)
251
252
```