```
(*

This exercise is based on a model of an emailing system. The system
is modeled through two state variables, "sent" (a set of messages) and
"inbox" (a dictionary mapping users to sets of messages - individual
inboxes). The overall goal is to complete the *specifications* of the
two functions "send" and "receive", so that the type invariant
provided is respected, and also the execution functions scenario1 and
scenario2 can be fully verified. Scenario3 should not be fully
verified, as it contains an error - it plays the role of a sanity
check.

You should write adequate preconditions and postconditions for "send"
and "receive". If required, feel free to strengthen the type
invariant, including additional information. All the verification
conditions generated should be provable in the TryWhy3 platform, with
the exception of some VCs generated for Scenario3.
*)


module Email

  use int.Int

  type user
  type content
  type message = { from : user;
                   tto : user;
                   content : content }

  clone set.SetApp with type elt = message

  clone fmap.MapApp with type key = user

    (*
    sent : conjunto de mensagens
    inbox : map user - conjunto de menssagens

    invariante:
    - uma mensagem ou está no conjunto "sent" ou está no conjunto "inbox"
    - se uma mensagem está no conjunto "sent", então, tem de estar endereçada
    por um utilizador
    *)

  type statetype = { mutable sent : set ; mutable inbox : t set }
    invariant { (forall u :user, m :message.
            mem u inbox /\ SetApp.mem m (inbox u) -> m.tto = u)
            /\
            true }
    by { sent = empty() ; inbox = create() }


  val state : statetype


    (*
    Send a message from user f to user t, with content c.
    The message is added to the set of sent messages

    Pre:
    - um utilizador tem uma caixa de entrada: t has an inbox
    - mensagem não está enviada: m is not in sent

    Pos:
    - se uma mensagem está em sent, então é porque foi enviada antes, ou é uma nova mensagem
    - se um utilizador está na inbox atual, então esteve na anterior
    - a mensagem está no novo conjunto de mensagens enviadas
    - o numero de mensagens enviadas incrementa
    *)


  let send (f t :user) (c :content) : ()
    requires { mem t state.inbox }
    requires { not SetApp.mem { from=f ; tto=t ; content=c } state.sent }
    ensures  { forall m :message.
                    SetApp.mem m state.sent <->
                    SetApp.mem m (old state.sent) \/ m = { from=f ; tto=t ; content=c } }
    ensures  { forall u :user.
                    mem u state.inbox <->
                    mem u (old state.inbox) }
    ensures  { SetApp.mem { from=f ; tto=t ; content=c } state.sent }
    ensures  { cardinal (state.sent) = cardinal (old state.sent) + 1 }
    writes   { state.sent }
  = let m = { from=f ; tto=t ; content=c } in
    state.sent <- SetApp.add m state.sent

    (*
    User t receives a message m. The message is removed from the set of sent messages,
    and added to the inbox of user t.

    Pre:
    - m está no sent
    - t tem inbox
    - m endereçada a t

    Pos:
    - se um usuário está na nova caixa de entrada, então estava na caixa de entrada antiga
    - se uma mensagem foi enviada, então foi enviada antes e não é a mensagem recebida
    - se a mensagem estiver na nova caixa de entrada do usuário t, então estava na caixa de entrada antiga do usuário t ou é a nova mensagem
    - a mensagem não está no novo conjunto de mensagens enviadas
    - a mensagem está no novo conjunto de mensagens enviadas do utilizador t
    - o numero de mensagens enviadas decrementa
    *)

(* WARNING : podem ser necessárias até 25000 iteracções para verificar o invariante *)
  let receive (t :user) (m :message) : ()
    requires { SetApp.mem m state.sent /\ m.tto = t /\ mem t state.inbox }
    ensures  { forall u :user.
                    mem u state.inbox <->
                    mem u (old state.inbox) }
    ensures  { forall m' :message
```

```
    ensures  { forall m' :message.
                   SetApp.mem m' state.sent <->
                   SetApp.mem m' (old state.sent) /\ m' <> m }
    ensures  { forall m' :message.
                   SetApp.mem m' (find t state.inbox) <->
                   SetApp.mem m' (find t (old state.inbox)) \/ m' = m }
    ensures  { not SetApp.mem m state.sent }
    ensures  { SetApp.mem m (find t state.inbox)}
    ensures  { cardinal (state.sent) = cardinal (old state.sent) - 1 }
    writes   { state.sent, state.inbox }
  = let umsgs = SetApp.add m (find t state.inbox) in
    state.inbox <- add t umsgs state.inbox;
    state.sent <- SetApp.remove m state.sent




  val u :user
  val f :user
  val t :user
  val c :content


  let scenario1 ()
    requires { let m = { from=u ; tto=u ; content=c }
               in not SetApp.mem m state.sent }
    requires { mem u state.inbox }
    ensures  { cardinal (state.sent) = cardinal (old state.sent) }
  = send u u c ; receive u { from=u ; tto=u ; content=c }


  let scenario2 ()
    requires { SetApp.is_empty(state.sent) }
    requires { mem u state.inbox }
    requires { mem t state.inbox }
    requires { u<>t }
    ensures  { SetApp.is_empty(state.sent) }
  = send f u c ; send f t c ; receive u { from=f ; tto=u ; content=c } ; receive t { from=f ; tto=t ; content=c }


  (* FAIL - checks inconsistencies *)
  let scenario3 ()
    requires { let m = { from=u ; tto=u ; content=c } in not SetApp.mem m state.sent }
    requires { mem u state.inbox }
    ensures  { cardinal (state.sent) = cardinal (old state.sent) }
  = send u u c ; receive u { from=f ; tto=u ; content=c }




end
```