```
module Dict = Map.Make(String);;
type name = string;;

(* Special functions, 'native' to POPPL *)
type o =
    | Time_of (* () -> time of most recent message *)
    | Most_recent (* (name, ?limit) -> message *)
    | Message_type_is (* (message, name) -> Float *)
    | Message_payload (* message -> e *)
    | Time_passed (* stamp_start, stamp_end, delay -> Float *)
    | Bind (* name,value -> binds the value to the variable in the context *)
    | Is_more_than (* > *)
    | Is_less_than (* < *)
    | Is_equal_to (* = *)
    | Is_in (* < _ < *)
    | Not (* Boolean not, 1. -> 0. and 0. -> 1. *)
;;

(* Grammar *)
type e =
    | Add of handler (* Adds a handler *)
    | Remove of name (* Removes a handler *)
    | Send of name * (e list) (* Sends a message with the given name,
                          and the given expressions as data *)
    | Native of o * (e list) (* Native function use, with parameters *)
    | Variable of name (* Variable of context *)
    | Begin of e list (* Begin *)
    | If of e * e * e (* If (condition) (true) (false) *)
    | Value of value
    | Expr of expr (* An operation with parameters *)
and expr = operation * (e list)
and operation =
    | OpAdd
    | OpSubtract
    | OpMultiply
    | OpDivide
    | OpExp
    | OpLog
and value =
    | Measure of measure
    | Float of float (* A float *)
    | String of string (* A string *)
    | Message of message
    | True (* True boolean *)
    | False (* False boolean *)
    | Record of record
    | Log of log
    | Void
and unit = string
and units = (unit * int) list
and measure = float * units
and message = name * (value list) (* A message is an identifier and data as a list of values *)
and record = message * float (* A record is a message and a timestamp as a float *)
and queue = record Queue.t
and log = record list (* List of messages with a time stamp*)
and handler = name * (log -> e) (* A handler is an identifier and a function of the log *)
and handler_set = (log -> e) Dict.t (* An actor is a set of handlers *)
and context = value Dict.t (* A list of name * value couples, associating a variable name to a value *)

;;

(* --- Context handling --- *)

(* Update the given couple (name,value) in the given context *)
let update_value (name : name) (v : value) (env : context) =
    Dict.add name v env
;;

(* Returns the variable associated to the given name in the given context *)
let get_variable_value (name : name) (env : context) =
    try Dict.find name env
    with Not_found -> Void
;;


(* --- Log handling --- *)


(* Retrieve the newest time message in order to get the current system time *)
let rec now (log : log) = match log with
    | [] -> 0.
    | ((a,[Float(time)]),_)::q when a = "time_message" -> time
```

```
    | a::q -> now q
;;

(* Returns the number of registered messages *)
let log_length (log : log) = List.length log
;;


let rec append_queue_to_list l q =
    if Queue.is_empty q then
        l
    else
        let m = Queue.pop q in
        append_queue_to_list (m::l) q
;;

let queue_to_list q =
    let rec aux q l = begin
        if Queue.is_empty q then l
        else let m = Queue.pop q in aux q (m::l)
    end in aux q []
;;

(* --- Printing utils --- *)


let rec string_of_dict d =
    Dict.fold (fun name v s -> s^"["^(name)^"] ") d ""
;;

let string_of_handler h = match h with
    | (name,f) -> name
;;

let rec string_of_value v = match v with
    | Float(f) -> "Float: "^(string_of_float f)
    | Measure(v,u) -> (string_of_float v)^(List.fold_left (fun s (u,i) -> s^u^(string_of_int i)^" ") "" u)^" "
    | String(s) -> "String: "^s
    | Message(name,l) -> "Message: "^name^" "^(string_of_value_list l)
    | True -> "True"
    | False -> "False"
    | Void -> "Void"
    | Record(m,t) -> "Record: ("^(string_of_value (Message(m)))^") "^(string_of_float t)
    | Log(a::q) -> "Log"
and string_of_value_list l = List.fold_left (fun b a -> b^(string_of_value a)) "" l;;

let rec string_of_log (log : log) = match log with
    | [] -> ""
    | ((name,data),time)::q -> (string_of_log (q))^" ["^name^" at "^(string_of_float time)^" ("^(string_of_value_list data
;;

(* --- Mathematical expressions and units handling --- *)

(* Multiply two unit lists *)
let rec multiply_units (u1 : units) (u2 : units) =
    let rec update_units (u1,i1) units acc = match units with
        | [] -> [(u1,i1)]@(List.rev acc)
        | (u2,i2)::q -> begin
                    (* Case 1 : the unit both exists in the two units : m times m gives m^2 *)
                    if u1 = u2 then
                        begin
                            if (i1+i2) <> 0 then (List.rev acc)@[(u1,i2+i1)]@q
                            else (List.rev acc)@q (* The units cancel themselves so we ignore them (ex: m/s times m*s is r
                        end
                    (* Case 2 : the unit has to be created somewhere (we decided to create it in the second units argument
                    else if u1 > u2 then
                        (List.rev acc)@[(u1,i1);(u2,i2)]@q (* We insert the new unit here to preserve lexicographical orde
                    else update_units (u1,i1) q ((u2,i2)::acc) (* The unit has to be added further to preserve lexicograph
                end

    in match u1 with
    | [] -> u2
    | (u,i)::q -> multiply_units q (update_units (u,i) u2 [])
;;

(* Multiply two measures *)
let multiply (m1 : measure) (m2 : measure) =
    let (v1,u1) = m1 and (v2,u2) = m2 in (v1*.v2,multiply_units u1 u2)
;;

(* Inverts a measure *)
let invert (m1 : measure) =
```

```
    let (v1,u1) = m1 in (1./.v1, List.map (fun (u,i) -> (u,-i)) u1)
;;

(* Evaluates a mathematical operation *)
let eval_op (op : operation) (values : value list) =
    (* (v1,u1) stands for (value of first variable, units of first variable) *)
    let ((v1,u1),(v2,u2)) = match values with
        | [(Measure(m1));(Measure(m2))] -> (m1,m2)
        | [(Measure(m1))] -> (m1,(0.,[])) in
    match op with
        | OpAdd ->      if u1 = u2 then Measure(v1+.v2,u1)
                        else failwith "Adding two quantities of different units"
        | OpSubtract -> if u1 = u2 then Measure(v1-.v2,u1)
                        else failwith "Subtracting two quantities of different units"
        | OpMultiply -> Measure(multiply (v1,u1) (v2,u2))
        | OpDivide ->   Measure(multiply (v1,u1) (invert (v2,u2)))
        | OpExp ->      if u1 = [] then Measure(exp v1,[])
                        else failwith "Evaluating a math function with a dimensioned value"
        | OpLog ->      if u1 = [] then Measure(log v1,[])
                        else failwith "Evaluating a math function with a dimensioned value"
;;



(* --- Handlers handling --- *)


(* Adds the (name,h) handler to the given handler set. *)
(* name : The handler's name *)
(* h : The handler's function of log *)
(* handler_set : The handler_set to modify *)
(* Returns : The given handler_set to which was added the couple (name,h) *)
let add_handler (name : name) (h : log -> e) (handler_set : handler_set) =
    Dict.add name h handler_set
;;

(* Removes the (name,h) handler to the given handler set. *)
(* name : The handler's name *)
(* handler_set : The handler_set to modify *)
(* Returns : The given handler_set to which was removed the couple (name,h) *)
let remove_handler (name : name) (handler_set : handler_set) =
    Dict.remove name handler_set
;;

let handler_set_from_list (handler_list : handler list) =
    List.fold_left (fun set (name,h) -> Dict.add (name) h set) Dict.empty handler_list
;;

let queue_from_list (l : record list) =
    let queue = Queue.create () in
    List.iter (fun x -> Queue.push x queue) l;
    queue
;;

(* Finds the latest message with the given identifier in the log. *)
(* log : The current log *)
(* n : The wanted message identifier to be found *)
(* Returns : A log containing the one wanted message, or Void if no message has been found *)
let rec find_message (log : log) (n : string) = match log with
    | [] -> Void
    | ((s,d),time)::q -> if s = n then Record((s,d),time) else find_message q n
;;

(* --- Evaluation --- *)

(* Evaluates the given expression and returns the next state. *)
(* e : The expression to evaluate *)
(* handler_set : The current handler list *)
(* log : The current log *)
(* outgoing_messages : The current outgoing messages queue*)
(* context : The current context storing the current set of variables *)
(* Returns : a 4-uplet (next_expression_to_evaluate, new_handler_set, new_outgoing_messages_set, new_context) *)
let rec eval (e : e) (handler_set : handler_set) (log : log) (outgoing_messages : queue) (context : context) =


    (* Evaluates each parameter of the parameters list before evaluating the native expression *)
    let rec eval_params_and_get_context params context acc = match params with
        | [] -> (List.rev acc,context)
        | a::q -> let (v,_,context) = eval a handler_set log outgoing_messages context in eval_params_and_get_context q c

    (* Native(o,params) constructor evaluation *)
    let rec eval_native (o : o) (params : e list) (handler_set : handler_set) (log : log) (outgoing_messages : queue) (co
```

```
      let (values,context) =  if params = [] then ([],context)
                              else eval_params_and_get_context params context [] in
      match o with
      | Time_of -> begin match values with
          | [] | [Void]           ->  (Void,context)
          | [Record(((s,d),time))]    ->  (Float(time),context)
          end
      | Most_recent   -> begin match log with
          | []    -> (Void,context)
          | ((s,d),t)::q  -> begin match values with
              | []            ->  (Record((s,d),t),context) (* No specified identifier, returns the most recent message
              | [String(n)]   ->  (find_message log n,context) (* Find the most recent message with the given identifie
              end
          end
      | Message_type_is -> begin match log with
          | []            ->  (Void,context) (* No current message *)
          | ((s,d),time)::q ->    begin match values with
              | [String(p)]   ->  if s = p then (True,context) else (False,context)
              end
          end
      | Message_payload -> begin match values with
          | [] | [Void]               ->  (Void,context) (* No given message or no found message *)
          | [Record((s,v::b),time)]   ->  (v,context)
          end
      | Not -> begin match values with
          | [Void]    ->  (Void,context)
          | [True]    ->  (False,context)
          | [False]   ->  (True,context)
          end
      | Bind -> begin match values with
          | [] | [Void] -> (Void,context)
          | [String(identifier);v] -> (Void, update_value identifier v context)
          end
      | Is_in -> begin match values with
          | []                                    ->  (Void,context)
          | [Void;_;_] | [_;Void;_] | [_;_;Void]  ->  (False,context) (* See p.13 of article *)
          | [Float(time);Float(s);Float(f)]        ->  if time < f && time >= s then (True,context) else (False,context)
          end
      | Is_less_than -> begin match values with
          | []                    ->  (Void,context)
          | [Float(s);Float(f)]   ->  if s <= f then (True,context) else (False,context)
          end
      | Is_more_than -> begin match values with
          | []                    ->  (Void,context)
          | [Float(s);Float(f)]   ->  if s >= f then (True,context) else (False,context)
          end
      | Is_equal_to -> begin match values with
          | []                    ->  (Void,context)
          | [Float(s);Float(f)]   ->  if s = f then (True,context) else (False,context)
          end
      | Time_passed -> begin match values with
          | []                                                    ->  (Void,context)
          | [_;Void;_]                                            ->  (True,context)
          | [Float(current_time);Float(start_time);Float(delay)]  ->  if start_time +. delay <= current_time then (True,
          end
      in
    match e with
    | Value(v) -> (v, handler_set, context)
    | Variable(name) -> (get_variable_value name context, handler_set, context)
    | Send(n,d) ->  begin let (data,context) = eval_params_and_get_context d context [] in
                    Queue.push ((n,data),(now log)) outgoing_messages;
                    (* print_endline ("Added, queue size is now:"^(string_of_int (Queue.length outgoing_messages))); *)
                    (Void,handler_set,context) end
    | Add(name,h) -> (Void, (add_handler name h handler_set),context)
    | Remove(name) -> (Void, (remove_handler (name) handler_set), context)
    | Begin(l) -> begin let rec browse_list l handler_set log context = match l with
        | [] -> (Void, handler_set, context)
        | a::q -> let (e, handler_set, context) = eval a handler_set log outgoing_messages context in browse_list q handle
        in browse_list l handler_set log context
        end
    | If(c, t, f) -> begin match c with
        | Value(True) | Value(Float(0.)) -> eval t handler_set log outgoing_messages context (* "True" *)
        | Value(False) | Value(Message(_, _)) | Value(Void)  -> eval f handler_set log outgoing_messages context (* "False
        | x -> let (v,handler_set,context) = eval x handler_set log outgoing_messages context in
               eval (If(Value(v), t, f)) handler_set log outgoing_messages context
        end
    | Native(o, params) -> let (v,context) = eval_native o params handler_set log outgoing_messages context in (v, handler
    | Expr(op,params) ->
          let (values,context) =  if params = [] then ([],context)
                                  else eval_params_and_get_context params context [] in
          let r = eval_op op values in (r, handler_set,context)
;;
```

```
(* Evaluates one handler, and returns the next set of handler H and the outgoing messages *)
(* handler : The handler to evaluate *)
(* handler_set : The current handler list *)
(* log : The current log *)
(* outgoing_messages : The current outgoing messages set *)
(* context : The current context storing the current set of variables *)
(* Returns : a 4-uplet (Void, new_handler_set, new_outgoing_messages_set, new_context) *)
let eval_handler (handler : handler) (handler_set : handler_set) (log : log) (outgoing_messages : queue) (context : conte
    let (_,f) = handler in
    eval (f log) handler_set log outgoing_messages context
;;

(* Triggers all given handlers for a given log, while registering and keeping up to date the future
    set of handler h_f, the outgoing messages that need to be sent and the
    evaluation context *)
(* handler_set : The current handler list *)
(* log : The current log *)
(* outgoing_messages : The current outgoing messages set *)
(* context : The current context storing the current set of variables *)
(* Returns : a 3-uplet (new_handler_set, new_log, new_context) *)
let browse_handlers (h : handler_set) (log : log) (outgoing_messages : queue) (context : context) =
    Dict.fold   ( fun name f (h_f,context) ->   let (_, h_f, context) = eval_handler (name,f) h_f log outgoing_messages c
                                                (h_f,context) )
                h
                (h,context)

;;

let start (h0 : handler_set) =
    let h = ref h0 in
    let log = ref [(("initialisation",[]),0.)] in
    let context = ref Dict.empty in
    let stopped = ref false in
    let outgoing_messages = (queue_from_list ([])) in
    print_endline ("starting handler list : "^(string_of_dict h0));
    while not !stopped do
        let (nh, nc) = browse_handlers (!h) (!log) (outgoing_messages) (!context) in
        h:=nh;
        log:=append_queue_to_list (!log) (outgoing_messages); (* We add the outgoing message to the log *)
        context:=nc;
        print_endline ("handler list : "^(string_of_dict nh));
        print_endline ("log : \n"^(string_of_log (!log)));
        print_endline ("");
        print_string ("> ");
        let input = read_line () in
        print_endline ("");
        let parameters = String.split_on_char ' ' input in match parameters with
        | ["wait";t] -> let time = float_of_string t in
                        log:=(("time_message",[Float((now !log) +. time)]),(now (!log))+.time)::(!log);
        | ["aptt";t] -> let value = float_of_string t in
                        log:=(("aPTTResult",[Float(value)]),(now (!log)))::(!log);
        | ["stop"] ->   stopped:= true;
        | _ -> ()
    done
;;

(* --- Syntax shortcuts --- *)

let o_add (m1 : measure) (m2 : measure) = Expr(OpAdd,[Value(Measure(m1));Value(Measure(m2))]);;
let o_sub (m1 : measure) (m2 : measure) = Expr(OpSubtract,[Value(Measure(m1));Value(Measure(m2))]);;
let o_mul (m1 : measure) (m2 : measure) = Expr(OpMultiply,[Value(Measure(m1));Value(Measure(m2))]);;
let o_div (m1 : measure) (m2 : measure) = Expr(OpDivide,[Value(Measure(m1));Value(Measure(m2))]);;
let o_exp (m1 : measure) = Expr(OpExp,[Value(Measure(m1))]);;
let o_log (m1 : measure) = Expr(OpLog,[Value(Measure(m1))]);;
let nat1 (o : o) a = Native(o,[a]);;
let nat2 (o : o) a b = Native(o,[a;b]);;
let nat3 (o : o) a b c = Native(o,[a;b;c]);;

let e_test_1 = o_add (2.,[("m",1)]) (3.,[("m",1)]);; (* 2 m + 3 m = 5 m*)
let e_test_2 = o_add (2.,[("m",1)]) (3.,[("s",1)]);; (* 2 m + 3 s = error*)
let e_test_3 = o_mul (2.,[("m",1)]) (3.,[("s",1)]);; (* 2 m times 3 s = 6 m s*)
let e_test_4 = o_mul (2.,[("m",1)]) (3.,[("m",-1)]);; (* 2 m times 3 m^-1 = 6 *)
let e_test_5 = o_div (2.,[("m",1)]) (3.,[("m",1)]);; (* 2 m divided by 3 m = 2/3 *)
let e_test_6 = o_exp (2.,[("m",1)]);; (* exp 2 m = error *)
let e_test_7 = Expr(OpMultiply,[Expr(OpAdd,[Value(Measure(2.,[("m",1)]));Value(Measure(3.,[("m",1)]))]);Value(Measure(3.,

(* Whenever a message is received *)
let whenever_message_type message_identifier bound_variable_name body =
    If(
```

```
        Native(Message_type_is,[Value(String(message_identifier))]),
            (Begin(
                [Native(Bind,[Value(String(bound_variable_name));Native(Message_payload,[Native(Most_recent,[Value(String
                )
            ),Value(Void)
    )
;;

(* ~Log query *)
let whenever_last_messages_in message_identifier bound_variable_name a b body =
    If(
        Native(Is_in,[Native(Message_payload,[Native(Most_recent,[Value(String(message_identifier))])]);a;b]),
            (Begin(
                [Native(Bind,[Value(String(bound_variable_name));Native(Message_payload,[Native(Most_recent,[Value(String
                )
            ),Value(Void)
    )
;;

(* ~Log query *)
let whenever_last_messages_outside_of message_identifier bound_variable_name a b body =
    If(
        Native(Not,[Native(Is_in,[Native(Message_payload,[Native(Most_recent,[Value(String(message_identifier))])]);a;b])
            (Begin(
                [Native(Bind,[Value(String(bound_variable_name));Native(Message_payload,[Native(Most_recent,[Value(String
                )
            ),Value(Void)
    )
;;

(* After instruction *)
(* time : The time after wich the given body has to be triggered *)
(* body : The expression evaluated when the condition is verified *)
(* log : The current log *)
let after (start_time : float) (delay : float) (body : e) (log : log) =
    let n = "after_"^(string_of_int (log_length log)) in
    Add(n,
        fun log -> If(
            Native(Time_passed,[Value(Float(now log));Value(Float(start_time));Value(Float(delay))]),
            Begin([body; Remove(n)]),
            Value(Void)
        )
    )
;;

(* Every instruction *)
(* delay : Duration from which a new message can be sent *)
(* body : The expression evaluated when the condition is verified *)
(* message : The message (identifier,data) that is used to get the latest date at wich it was sent in the log *)
(* log : The current log *)
let every (delay : float) (body : e) message (log : log) =
    let (s,d) = message in
    If(Native(Time_passed, [Value(Float(now log));Native(Time_of,[Native(Most_recent,[Value(String(s))])]);Value(Float(de
        body,
        Value(Void)
    )
;;

(* --- Snippets --- *)

let initially =
    ("initially", fun log ->
        Begin(
            [
                Send(("giveBolus",[Value(Measure(80.,[("u",1);("kg",-1)]));Value(String("HEParin"));Value(String("iv"))])
                Send(("start",[Value(Measure(3.,[("u",1);("kg",-1);("h",-1)]));Value(String("HEParin"))]));
                Remove("initially")
            ]
        )
    )
;;

let infusion =
    ("infusion", fun log -> (whenever_message_type "aPTTResult" "aPTT"
        (Begin(
            [
                If(
                    Native(Is_less_than,[Variable("aPTT");Value(Float(45.))]),
                    Begin(
                        [
                            Send(("giveBolus",[Value(Measure(80.,[("u",1);("kg",-1)]));Value(String("HEParin"));Value(Str
                            Send(("increase",[Value(Measure(3.,[("u",1);("kg",-1);("h",-1)]));Value(String("HEParin"))]))
```

```
                        ]
                    ),
                    Value(Void)
                );
                If(
                    Native(Is_in,[Variable("aPTT");Value(Float(45.));Value(Float(59.))]),
                    Begin(
                        [
                            Send(("giveBolus",[Value(Measure(40.,[("u",1);("kg",-1)]));Value(String("HEParin"));Value(Str
                            Send(("increase",[Value(Measure(1.,[("u",1);("kg",-1);("h",-1)]));Value(String("HEParin"))]))
                        ]
                    ),
                    Value(Void)
                );
                If(
                    Native(Is_in,[Variable("aPTT");Value(Float(101.));Value(Float(123.))]),
                    Begin(
                        [
                            Send(("decrease",[Value(Measure(1.,[("u",1);("kg",-1);("h",-1)]));Value(String("HEParin"))]))
                        ]
                    ),
                    Value(Void)
                );
                If(
                    Native(Is_more_than,[Variable("aPTT");Value(Float(123.))]),
                    Begin(
                        [
                            Send(("hold",[Value(String("HEParin"))]));
                            after (now log) 1. (Begin(
                                [
                                    Send(("restart",[Value(String("HEParin"))]));
                                    Send(("decrease",[Value(Measure(3.,[("u",1);("kg",-1);("h",-1)]));Value(String("HEPar
                                ]
                            )) log
                        ]
                    ),
                    Value(Void)
                )
            ]
            )
        )
    )
)
;;

let apttchecking =
    ("aPTTChecking", fun log ->
        Begin([
            every 6. (whenever_last_messages_outside_of "aPTTResult" "aPTT" (Value(Float(59.))) (Value(Float(101.))) (Send
            every 24. (whenever_last_messages_in "aPTTResult" "aPTT" (Value(Float(59.))) (Value(Float(101.))) (Send(("che
        ])
    )
;;


let h0 = handler_set_from_list [initially;infusion;apttchecking];;
```