```
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 name * (e -> e) (* Adds a handler  with a name and a function *)
    | Remove of name (* Removes a handler *)
    | Send of name * (e list) (* Sends a message *)
    | Lambda of (e -> e) (* Function *)
    | List of e list (* List of expressions *)
    | Native of o * (e list) (* Native function use *)
    | Variable of name (* Variable of context *)
    | Message of name * (e list) (* A message is an identifier and data as a list of expressions *)
    | Log of (name * (e list) * float) list (* List of messages with a time stamp*)
    | Begin of e (* Begin *)
    | If of e * e * e (* If (condition) (true) (false) *)
    | Float of float
    | String of string
    | True (* True boolean *)
    | False (* False boolean *)
    | Void
;;

type message = name * (e list);;
type handler = name * (e -> e);;
type actor = handler list;;
type context = (name * e) list;;

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

(* Returns the list of current registered variables *)
let context_to_name_list context =
    List.map (fun (a,b) -> a) context
;;

(* Update the given couple (name,value) in the given context *)
let update_value (name,v) context =
    let rec aux l acc = match l with
        | [] -> List.rev acc
        | (n,v2)::q -> if name = n then aux q ((name,v)::acc) else aux q ((name,v2)::acc)
    in aux context []
;;

(* Applies the values e to the variables names in the given context *)
let apply_values variables e context =
    let rec aux zipped context = match zipped with
        | [] -> context
        | (name,v)::q ->
                if not(List.mem name (context_to_name_list context)) then
                    aux q ((name,v)::context) (* If var doesn't exist, we create it *)
                else
                    aux q (update_value (name,v) context) (* Else we update it *)
    in aux (List.combine variables e) context
;;

let rec get_variable_value name context = match context with
    | [] -> Void
    | (n,v)::q -> if n = name then v else get_variable_value name q
;;

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

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

```
(* Returns the number of registered messages *)
let log_length (log : e) = match log with
    | Log(l) -> List.length l
;;


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

let rec string_of_handler_list handler_list = match handler_list with
    | [] -> ""
    | (name,f)::q -> "["^name^"] "^(string_of_handler_list q)
;;

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


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

(* Adds the (name,h) handler to the given handler_list. *)
let add_handler (name : name) (h : e -> e) (handler_list : handler list) =
    let rec browse l = match l with
        | [] -> (name,h)::handler_list
        | (n,_)::q -> if n = name then handler_list else browse q in
    browse handler_list
;;

(* Adds the (name,e) handler to the given handler_list. *)
let remove_handler (name : name) (handler_list : handler list) =
    let rec browse l = match l with
        | [] -> l
        | (n,e)::q -> if n = name then browse q else (n,e)::(browse q) in
    browse handler_list
;;

(* 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 : e) (n : string) = match log with
    | Log([]) -> Void
    | Log((s,d,time)::q) -> if s = n then (Log([(s,d,time)])) else find_message (Log(q)) n
;;


(* Evaluates the given expression and returns the next state. *)
(* e : The expression to evaluate *)
(* handler_list : 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 (next_expression_to_evaluate, new_handler_list, new_outgoing_messages_set, new_context) *)
let rec eval (e : e) (handler_list : handler list) (log : e) (outgoing_messages : e) (context : context) =
    let rec eval_native (o : o) (params : e list) (handler_list : handler list) (log : e) (outgoing_messages : e) (contex
        let Log(l) = log in
        match o with
        | Time_of -> begin match params with
            | [] | [Void]           ->  (Void,context)
            | [Log((s,d,time)::q)]  ->  (Float(time),context)
            | [e]                   ->  let (v,_,_,context) = (eval e handler_list log outgoing_messages context) in
                                        eval_native o [v] handler_list log outgoing_messages context
            end
        | Most_recent   -> begin match l with
            | []    -> (Void,context)
            | (s,d,t)::q    -> begin match params with
                | []            ->  (Log([(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
                | [e]           ->  let (v,_,_,context) = (eval e handler_list log outgoing_messages context) in
                                    eval_native o [v] handler_list log outgoing_messages context
                end
            end
        | Message_type_is -> begin match l with
            | []            ->  (Void,context) (* No current message *)
            | (s,d,time)::q ->  begin match params with
                | [String(p)]   ->  if s = p then (True,context) else (False,context)
                | [e]           ->  let (v,_,_,context) = (eval e handler_list log outgoing_messages context) in
                                    eval_native o [v] handler_list log outgoing_messages context
                end
            end
```

```
        | Message_payload -> begin match params with
            | [] | [Void]               ->  (Void,context) (* No current message *)
            | [Log((s,e::b,time)::q)]   ->  (e,context)
            | [e]                       ->  let (v,_,_,context) = (eval e handler_list log outgoing_messages context) in
                                            eval_native o [v] handler_list log outgoing_messages context
            end
        | Not -> begin match params with
            | [Void]    ->  (Void,context)
            | [True]    ->  (False,context)
            | [False]   ->  (True,context)
            | [e]       ->  let (v,_,_,context) = (eval e handler_list log outgoing_messages context) in
                            eval_native o [v] handler_list log outgoing_messages context
            end
        | Bind -> begin match params with
            | [] | [Void] -> (Void,context)
            | [String(identifier);v] -> (Void, apply_values [identifier] [v] context)
            | [a;v] -> let (a,_,_,context) = eval a handler_list log outgoing_messages context in
                       let (v,_,_,context) = eval v handler_list log outgoing_messages context in
                       eval_native o [a;v] handler_list log outgoing_messages context
            end
        | Is_in -> begin match params 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)
            | [c;a;b]                               ->  let (v,_,_,context) = eval c handler_list log outgoing_messages co
                                                        let (s,_,_,context) = eval a handler_list log outgoing_messages co
                                                        let (f,_,_,context) = eval b handler_list log outgoing_messages co
                                                        eval_native o [v;s;f] handler_list log outgoing_messages context
            end
        | Is_less_than -> begin match params with
            | []                    ->  (Void,context)
            | [Float(s);Float(f)]   ->  if s <= f then (True,context) else (False,context)
            | [a;b]                 ->  let (s,_,_,context) = eval a handler_list log outgoing_messages context in
                                        let (f,_,_,context) = eval b handler_list log outgoing_messages context in
                                        eval_native o [s;f] handler_list log outgoing_messages context
            end
        | Is_more_than -> begin match params with
            | []                    ->  (Void,context)
            | [Float(s);Float(f)]   ->  if s >= f then (True,context) else (False,context)
            | [a;b]                 ->  let (s,_,_,context) = eval a handler_list log outgoing_messages context in
                                        let (f,_,_,context) = eval b handler_list log outgoing_messages context in
                                        eval_native o [s;f] handler_list log outgoing_messages context
            end
        | Is_equal_to -> begin match params with
            | []                    ->  (Void,context)
            | [Float(s);Float(f)]   ->  if s = f then (True,context) else (False,context)
            | [a;b]                 ->  let (s,_,_,context) = eval a handler_list log outgoing_messages context in
                                        let (f,_,_,context) = eval b handler_list log outgoing_messages context in
                                        eval_native o [s;f] handler_list log outgoing_messages context
            end
        | Time_passed -> begin match params with
            | []                                                    ->  (Void,context)
            | [_;Void;_]                                            ->  (True,context)
            | [Float(current_time);Float(start_time);Float(delay)]  ->  if start_time +. delay <= current_time then (True,
            | [c;a;b]                                               ->  let (v,_,_,context) = eval c handler_list log outg
                                                                        let (s,_,_,context) = eval a handler_list log outg
                                                                        let (f,_,_,context) = eval b handler_list log outg
                                                                        eval_native o [v;s;f] handler_list log outgoing_me
            end
        in
    let Log(m) = outgoing_messages in
    match e with
    | Void | True | False | Float(_) | String(_) | Log(_) -> (e, handler_list, outgoing_messages, context)
    | Variable(name) -> (get_variable_value name context, handler_list, log, context)
    | Send(n, d) -> eval (Void) handler_list log (Log((n,d,now log)::m)) context
    | Add(name,e) -> eval (Void) (add_handler name e handler_list) log outgoing_messages context
    | Remove(name) -> eval (Void) (remove_handler (name) handler_list) log outgoing_messages context
    | Begin(List(l)) -> begin let rec browse_list l handler_list log outgoing_messages context = match l with
        | [] -> (Void, handler_list, outgoing_messages, context)
        | a::q -> let (e, handler_list, outgoing_messages, context) = eval a handler_list log outgoing_messages context in
        in browse_list l handler_list log outgoing_messages context
        end
    | If(c, t, f) -> begin match c with
        | True | Float(0.) -> eval t handler_list log outgoing_messages context (* "True" *)
        | False | Message(_, _) | Log(_) | Void | Lambda(_) -> eval f handler_list log outgoing_messages context (* "False
        | x -> let (e,handler_list,outgoing_messages, context) = eval x handler_list log outgoing_messages context in
               eval (If(e, t, f)) handler_list log outgoing_messages context
        end
    | Native(o, a) -> let (e,context) = eval_native o a handler_list log outgoing_messages context in eval e handler_list

;;
```

```
(* Evaluates one handler, and returns the next set of handler H and the outgoing messages *)
(* handler : The handler to evaluate *)
(* handler_list : 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_list, new_outgoing_messages_set, new_context) *)
let eval_handler (handler : handler) (handler_list : handler list) (log : e) (outgoing_messages : e) (context : context) =
    let (_,f) = handler in
    eval (f log) handler_list 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 *)
let browse_handlers (h : handler list) (log : e) (context : context) =
    let rec build_future_handler_set (h : handler list) (h_f : handler list) (log : e) (outgoing_messages : e) (context :
        | [] -> (h_f, outgoing_messages, context)
        | a::q -> let (_, h_f, outgoing_messages, context) = eval_handler a h_f log outgoing_messages context in
                  build_future_handler_set q h_f log outgoing_messages context
    in build_future_handler_set h h log (Log([])) context (* We build the future handler set based on the current one, wi
;;

(* Looping over outgoing messages, and for each one we trigger all the handlers *)
(* handler_list : 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_list, new_log, new_context) *)
let rec browse_outgoing_messages (handler_list : handler list) (log : e) (outgoing_messages : e) (context : context) =
    (* print_endline ("browsing outgoing messages with handler list : "^(string_of_handler_list handler_list)^" and log :
    let Log(l) = log in
    match outgoing_messages with
    | Log([]) -> (handler_list, log, context)
    | Log(a::q) -> let (h_f, Log(outgoing_messages_to_add), context) = browse_handlers handler_list (Log(a::l)) context in
                browse_outgoing_messages h_f (Log(a::l)) (Log(q@outgoing_messages_to_add)) context
;;

(* Runs once over a handler set, from scratch. *)
(* h0 : The starting handler set *)
(* Returns : a 3-uplet (new_handler_list, new_log, new_context) *)
let run (h0 : handler list) =
    browse_outgoing_messages h0 (Log([])) (Log([("initialisation",[],0.)])) []
;;

let start (h0 : handler list) =
    let h = ref h0 in
    let log = ref (Log([])) in
    let context = ref [] in
    let outgoing_messages = ref (Log([("initialisation",[],0.)])) in
    let stopped = ref false in
    print_endline ("starting handler list : "^(string_of_handler_list h0));
    while not !stopped do
        let (nh, nl, nc) = browse_outgoing_messages (!h) (!log) (!outgoing_messages) (!context) in
        h:=nh;
        log:=nl;
        context:=nc;
        outgoing_messages:=(Log([]));
        print_endline ("handler list : "^(string_of_handler_list nh));
        print_endline ("log : "^(string_of_log nl));
        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
                        outgoing_messages := (Log([("time_message",[Float((now !log) +. time)],(now !log) +. time)]));
        | ["aptt";t] -> let value = float_of_string t in
                        outgoing_messages := (Log([("aPTTResult",[Float(value)],(now !log))]));
        | ["stop"] ->   stopped:= true;
        | _ -> ()
    done
;;

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

(* Whenever a message is received *)
let whenever_message_type message_identifier bound_variable_name body =
    If(
        Native(Message_type_is,[String(message_identifier)]),
            (Begin(
                List([Native(Bind,[String(bound_variable_name);Native(Message_payload,[Native(Most_recent,[String(message_
```

```
            ),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,[String(message_identifier)])]);a;b]),
            (Begin(
                List([Native(Bind,[String(bound_variable_name);Native(Message_payload,[Native(Most_recent,[String(message_
            ),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,[String(message_identifier)])]);a;b])]),
            (Begin(
                List([Native(Bind,[String(bound_variable_name);Native(Message_payload,[Native(Most_recent,[String(message_
            ),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 : e) (delay : e) (body : e) (log : e) =
    let n = "after_"^(string_of_int (log_length log)) in
    Add(n,
        fun log -> If(
            Native(Time_passed,[Float(now log);start_time;delay]),
            Begin(List([body; Remove(n)])),
            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 : e) =
    let (s,d) = message in
    If(Native(Time_passed, [Float(now log);Native(Time_of,[Native(Most_recent,[String(s)])]);Float(delay)]),
        body,
        Void
    )
;;

(* --- Snippets --- *)

let initially =
    ("initially", fun log ->
        Begin(
            List([
                Send(("giveBolus",[Float(80.);String("HEParin");String("iv")]));
                Send(("start",[Float(3.);String("HEParin")]));
                Remove("initially")
            ])
        )
    )
;;

let infusion =
    ("infusion", fun log -> (whenever_message_type "aPTTResult" "aPTT"
        (Begin(
            List([
                If(
                    Native(Is_less_than,[Variable("aPTT");Float(45.)]),
                    Begin(
                        List([
                            Send(("giveBolus",[Float(80.);String("HEParin");String("iv")]));
                            Send(("increase",[Float(3.);String("HEParin")]));
                        ])
                    ),
                    Void
                );
                If(
                    Native(Is_in,[Variable("aPTT");Float(45.);Float(59.)]),
```

```
                    Begin(
                        List([
                            Send(("giveBolus",[Float(40.);String("HEParin");String("iv")]));
                            Send(("increase",[Float(1.);String("HEParin")]));
                        ])
                    ),
                    Void
                );
                If(
                    Native(Is_in,[Variable("aPTT");Float(101.);Float(123.)]),
                    Begin(
                        List([
                            Send(("decrease",[Float(1.);String("HEParin")]));
                        ])
                    ),
                    Void
                );
                If(
                    Native(Is_more_than,[Variable("aPTT");Float(123.)]),
                    Begin(
                        List([
                            Send(("hold",[String("HEParin")]));
                            after (Float(now log)) (Float(1.)) (Begin(
                                List([
                                    Send(("restart",[String("HEParin")]));
                                    Send(("decrease",[Float(3.);String("HEParin")]));
                                ])
                            )) log
                        ])
                    ),
                    Void
                )
            ])
        ))
    )
)
;;

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


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