```
type environment = (string * mlvalue ref) list
(* note: value part is mlvalue ref (to support LET REC) *)

(* miniML values *)
and mlvalue =
    Int of int
  | Bool of bool
    Cons of mlvalue * mlvalue
    Pair of mlvalue * mlvalue
    Nil
    Closure of (pattern * expr) list * environment
  | Undefined

(* miniML identifiers *)
and ident = string

(* miniML expressions *)
and expr =
    Const of mlvalue
    Var of ident
    Plus of expr * expr
    Minus of expr * expr
    Times of expr * expr
    Div of expr * expr
    Equal of expr * expr
    PairExp of expr * expr
    ConsExp of expr * expr
    IfExp of expr * expr * expr
    LambdaExp of (pattern * expr) list
    App of expr * expr
    MatchExp of expr * (pattern * expr) list
    LetExp of (pattern * expr) list * expr
  | LetRecExp of (pattern * expr) list * expr
(* following two constructor is not used *)
    TopLetExp of (pattern * expr) list
  | TopLetRecExp of (pattern * expr) list

(* miniML pattern expressions *)
and pattern =
    ConstPtn of mlvalue
    IdentPtn of ident
    ConsPtn of pattern * pattern
  | PairPtn of pattern * pattern
```



```
%{
open MiniML
(* 1 [2; 3] -> App(1,2) [3] -> A(A(1,2),3) [] *)
let rec make_appl x = function
    [] -> x
  | y::ys -> make_appl (App(x,y)) ys
let rec make_list = function
    [] -> Const(Nil)
  | y::ys -> ConsExp(y,make_list ys)
let rec make_list_ptn = function
    [] -> ConstPtn(Nil)
  | y::ys -> ConsPtn(y,make_list_ptn ys)
let rec make_fun e = function
    [] -> e
  | y::ys -> LambdaExp([y, make_fun e ys])

let rec pat_vars = function
    ConstPtn _ -> []
  | IdentPtn "_" -> []
  | IdentPtn i -> [i]
  | ConsPtn(p1,p2) | PairPtn(p1,p2) ->
      pat_vars p1 @ pat_vars p2

let rec check_unique = function
    [] -> ()
  | h :: t ->
      if List.mem h t then
        failwith ("duplicated variable " ^ h ^ " in bindings")
      else
        check_unique t

(* checkes well-formedness of single pattern *)
let check_pattern p =
  check_unique (pat_vars p)

(* checkes well-formedness of let bindings *)
let check_let bindings =
  let vars =
    List.fold_right
      (fun (patt, _) acc ->
        pat_vars patt @ acc) bindings []
  in
  check_unique vars

(* checkes well-formedness of let-rec bindings *)
let check_letrec bindings =
  let vars =
    List.fold_right
      (fun (patt, _) acc ->
        match patt with
          IdentPtn "_" -> acc
        | IdentPtn x -> x :: acc
        | _ ->
            failwith "only variables are allowed as left-hand side of let rec")
      bindings []
  in
  check_unique vars;
  let rec subtract l s =
    match l with
      [] -> []
    | h::t ->
        if List.mem h s then subtract t s else h :: subtract t s
```

```
  in
  let rec subtract_vars u = function
      [] -> u
    | h::t ->
        subtract_vars (subtract u (pat_vars h)) t
  in
  let rec iter u = function
      Const _ -> ()
    | Plus(x1,x2) | Minus(x1,x2) | Times(x1,x2) | Div(x1,x2)
    | Equal(x1,x2) | PairExp(x1,x2) | ConsExp(x1,x2)
    | App(x1,x2) ->
        iter u x1; iter u x2
    | IfExp(x1,x2,x3) -> iter u x1; iter u x2; iter u x3
    | Var(i) ->
        if List.mem i u then
          failwith ("invalid occurence of variable " ^ i ^ " in right-hand side
of let rec")
    | LambdaExp(l) -> ()
    | LetExp(l,e) ->
        List.iter (fun (pat, exp) -> iter u exp) l;
        iter (subtract_vars u (List.map fst l)) e
    | LetRecExp(l,e) ->
        let vars = (subtract_vars u (List.map fst l)) in
        List.iter (fun (pat, exp) -> iter vars exp) l;
        iter vars e
    | MatchExp(e1,l) ->
        iter u e1;
        List.iter
          (fun (pat, exp) ->
            iter (subtract_vars u [pat]) exp) l
    | TopLetExp _ | TopLetRecExp _ ->
        assert false
  in
  List.iter (fun (_,e) -> iter vars e) bindings
%}

%token <int> INT
%token <string> IDENT

%token LPAREN RPAREN
%token LBRACKET RBRACKET
%token BAR
%token PLUS MINUS TIMES DIV EQUAL
%token EOD
%token IF THEN ELSE
%token LET REC IN AND
%token CONS SEMICOLON COMMA
%token FUN FUNCTION ARROW
%token MATCH WITH
%token FALSE TRUE

%right prec_let
%right prec_fun prec_match prec_try
%right prec_list
%right prec_if
%left BAR
%left COMMA
%left EQUAL
%right CONS
%left PLUS MINUS
%left TIMES DIV
%left prec_appl
```



```
%start main
%type <MiniML.expr> main
%%
main:
  toplevel_phrase EOD { $1 }
| error { failwith("parse failed.") }
;
toplevel_phrase:
  expr { $1 }
| LET let_bindings
    { TopLetExp(List.rev $2) }
| LET REC let_bindings
    {
      check_letrec $3;
      TopLetRecExp(List.rev $3)
    }

simple_expr:
  INT { Const(Int $1) }
| TRUE { Const(Bool true) }
| FALSE { Const(Bool false) }
| IDENT { Var($1) }
| LPAREN expr RPAREN { $2 }
| LBRACKET RBRACKET { Const(Nil) }
| LBRACKET expr_semi_list RBRACKET { make_list(List.rev $2) }
;
expr:
  simple_expr { $1 }
| expr COMMA expr { PairExp($1,$3) }
| expr CONS expr { ConsExp($1,$3) }
| expr PLUS expr { Plus($1,$3) }
| expr MINUS expr { Minus($1,$3) }
| expr TIMES expr { Times($1,$3) }
| expr DIV expr { Div($1,$3) }
| expr EQUAL expr { Equal($1,$3) }
| FUN patterns ARROW expr %prec prec_fun
    {
      make_fun $4 (List.rev $2)
    }
| FUNCTION multiple_matches %prec prec_fun
    {
      LambdaExp(List.rev $2)
    }
| simple_expr expr_list %prec prec_appl
    { make_appl $1 (List.rev $2) }
| IF expr THEN expr ELSE expr %prec prec_if
    { IfExp($2,$4,$6) }
| MATCH expr WITH multiple_matches %prec prec_match
    {
     MatchExp($2,List.rev $4)
   }
| LET let_bindings IN expr %prec prec_let
    {
     check_let $2;
     LetExp(List.rev $2,$4)
   }
| LET REC let_bindings IN expr %prec prec_let
    {
      check_letrec $3;
      LetRecExp(List.rev $3,$5)
    }
;
```

```
let_bindings:
  let_binding { [$1] }
| let_bindings AND let_binding { $3 :: $1 }
let_binding:
  patterns EQUAL expr {
    match List.rev $1 with
      [p] -> p, $3
    | (IdentPtn i as p1)::ps -> p1, make_fun $3 ps
    | _ -> failwith "parse failed at let_binding" }
multiple_matches:
  multiple_matches BAR sgl_match { $3 :: $1 }
| sgl_match { [$1] }
;
sgl_match:
  pattern ARROW expr { $1, $3 }
;
expr_semi_list:
  expr { [$1] }
| expr_semi_list SEMICOLON expr { $3 :: $1 }
;
expr_list:
  simple_expr { [$1] }
| expr_list simple_expr { $2 :: $1 }
;
patterns:
  pattern { [$1] }
| patterns pattern { $2 :: $1 }
;
pattern:
  pattern_internal { check_pattern $1; $1 }

pattern_internal:
  LPAREN pattern_internal RPAREN { $2 }
| INT { ConstPtn(Int $1) }
| TRUE { ConstPtn(Bool true) }
| FALSE { ConstPtn(Bool false) }
| LBRACKET RBRACKET { ConstPtn(Nil) }
| IDENT { IdentPtn($1) }
| pattern_internal CONS pattern_internal { ConsPtn($1,$3) }
| pattern_internal COMMA pattern_internal { PairPtn($1,$3) }
| LBRACKET pattern_semi_list RBRACKET { make_list_ptn(List.rev $2) }

pattern_semi_list:
  pattern_internal { [$1] }
| pattern_semi_list SEMICOLON pattern_internal { $3 :: $1 }
;
```



```
{
 open MiniML
 open MiniMLParser
 exception Eof
 let identifier_of s = List.assoc s
     [ "if", IF;
       "then", THEN;
       "else", ELSE;
       "fun", FUN;
       "function", FUNCTION;
       "match", MATCH;
       "with", WITH;
       "false", FALSE;
       "true", TRUE;
       "let", LET;
       "rec", REC;
       "in", IN;
       "and", AND;
     ]
}
let lowercase = ['a'-'z' '_']
let uppercase = ['A'-'Z']
let identchar =
  ['A'-'Z' 'a'-'z' '_' '\'' '0'-'9']

rule token = parse
  [' ' '\t' '\n'] { token lexbuf }
| ";;" { EOD }
| ['0'-'9']+ { INT(int_of_string(Lexing.lexeme lexbuf)) }
| "::" { CONS }
| ";" { SEMICOLON }
| "|" { BAR }
| "[" { LBRACKET }
| "]" { RBRACKET }
| "->" { ARROW }
| lowercase identchar * { let s = Lexing.lexeme lexbuf in
                          try identifier_of(s) with _ -> IDENT(s) }
| '+' { PLUS }
| '-' { MINUS }
| '*' { TIMES }
| '/' { DIV }
| '=' { EQUAL }
| '(' { LPAREN }
| ')' { RPAREN }
| ',' { COMMA }
| eof { raise Eof }
```

```
open MiniML

(* let failwith s = raise (Failure s) *)

let rec get id = function
    [] -> raise (Failure ("Unbound value " ^ id))
  | (i, v) :: ls ->
      if i = id then
        (match !v with
          Undefined -> failwith "using undefined value with letrec"
        | x -> x)
      else get id ls

let get_int = function
    Int(v1) -> v1 | _ -> failwith "not an int value"

let get_bool = function
    Bool(v1) -> v1 | _ -> failwith "not a bool value"

exception MatchFailed

(* val eval : env * expr -> mlvalue *)

let rec eval env = function
    Const(v) -> v
  | Var(id) -> get id env
  | Plus(e1, e2) ->
      Int(get_int (eval env e1) + get_int (eval env e2))
  | Minus(e1, e2) ->
      Int(get_int (eval env e1) - get_int (eval env e2))
  | Times(e1, e2) ->
      Int(get_int (eval env e1) * get_int (eval env e2))
  | Div(e1, e2) ->
      Int(get_int (eval env e1) / get_int (eval env e2))
  | Equal(e1, e2) ->
      Bool((eval env e1) = (eval env e2))
  | ConsExp(e1, e2) ->
      Cons(eval env e1, eval env e2)
  | IfExp(ec, e1, e2) ->
      if get_bool (eval env ec) then eval env e1 else eval env e2
  | LambdaExp [p1, e] ->
      Closure([p1, e], env)
  | LambdaExp ps -> (* only for optional *)
      Closure(ps, env)
  | App(e1, e2) -> begin
      match eval env e1 with
        Closure([IdentPtn arg, body], i_env) -> ... (* required *)
      | Closure(pats, i_env) ->
          failwith "unsupported expression" (* optional *)
      | _ -> failwith "applying value to non-closure"
  end
  | MatchExp(exp,pats) -> ... (* required *)
  | LetExp([IdentPtn id, e1], e2) ->
      let v1 = eval env e1 in
      eval ((id, ref v1) :: env) e2
  | LetExp(ptns, e2) -> failwith "unsupported expression" (* optional *)
  | LetRecExp([IdentPtn id, e1], e2) -> ... (* required *)
  | LetRecExp(ptns, e2) ->
      failwith "unsupported expression" (* optional, slightly difficult *)
  | _ -> failwith "invalid expression"
```



```
type id = int

type mltypes =
    (* concrete types *)
    Tint
  | Tbool
  | Tlist of mltypes
  | Tpair of mltypes * mltypes
  | Tarrow of mltypes * mltypes
    (* intermediate types for type synthesis *)
  | TVar of vartype
  | TUnknown
and vartype = { id : int ; mutable v : mltypes }

(* type schema *)
type schema = Forall of id list * mltypes

(* type environment *)
type typeenv = ( string * schema ) list

open MiniML

let new_id =
  let v = ref 2 in
  fun () -> (incr v; !v : id)

let new_type_variable () = TVar { id = new_id (); v = TUnknown }

let occurs { id = n; v = _ } v2 =
  let rec iter = function
      TVar { id = m; v = _ } -> (n = m)
    | Tint | Tbool -> false
    | Tarrow(t1,t2) -> iter t1 || iter t2
    | Tpair(t1,t2) -> iter t1 || iter t2
    | Tlist(t1) -> iter t1
    | TUnknown -> failwith "occurs"
  in iter v2;;

let rec shorten t = match t with
  TVar ({ id = _; v = tv } as tv2) -> begin
    match tv with
      TUnknown -> t
    | TVar { id = _; v = TUnknown } -> tv
    | TVar tv1 -> tv2.v <- tv1.v; shorten t
    | t' -> shorten t'
  end
| TUnknown -> failwith "shorten: should not happen"
| t -> t

let vars_of_type t =
  let rec iter vs = function
      TVar { id = n; v = TUnknown } ->
        if List.mem n vs then vs else n::vs
    | TVar { id = _; v = t } -> iter vs t
    | Tarrow(t1, t2) -> iter (iter vs t1) t2
    | Tpair(t1, t2) -> iter (iter vs t1) t2
    | Tint | Tbool -> vs
    | Tlist t2 -> iter vs t2
    | TUnknown -> failwith "vars_of_type: should not happen"
  in iter [] t;;

let rec subtract_list a b =
```

```
  match a with
    [] -> []
  | hd::tl ->
      if List.mem hd b
      then subtract_list tl b
      else hd::(subtract_list tl b)

let rec unique = function
    [] -> []
  | hd::tl ->
      if List.mem hd tl
      then unique tl
      else hd::(unique tl)

let vars_of_tenv tenv =
  List.flatten
    (List.map
       (fun (_, Forall(gvars, t)) ->
         subtract_list (vars_of_type t) gvars)
       tenv)

let generalize tenv t =
  let genvars =
    unique (subtract_list (vars_of_type t)
              (vars_of_tenv tenv))
  in Forall(genvars, t)

let instanciate = function
    Forall([], t) -> t
  | Forall(gvars, t) ->
      let new_vars =
        List.map(fun n -> (n, new_type_variable ())) gvars
      in
      let rec iter = function
          (TVar { id = n; v = TUnknown } as t) ->
            (try List.assoc n new_vars
            with _ -> t)
        | TVar { id = _; v = t } -> iter t
        | Tint -> Tint
        | Tbool -> Tbool
        | Tlist(t1) -> Tlist(iter t1)
        | Tarrow(t1,t2) -> Tarrow(iter t1, iter t2)
        | Tpair(t1,t2) -> Tpair(iter t1, iter t2)
        | TUnknown -> failwith "instanciate: should not happen"
      in
      iter t

let rec unify t1 t2 = match (shorten t1, shorten t2) with
  (TVar ({ id = n; v = TUnknown } as tv1),
   (TVar { id = m; v = TUnknown } as t2))
  -> if n != m then tv1.v <- t2 else ()
| (t1, TVar ({ id = _; v = TUnknown } as tv2))
  -> if not (occurs tv2 t1) then tv2.v <- t1
    else failwith "unify: recursive type occured"
| (TVar ({ id = _; v = TUnknown } as tv1), t2)
  -> if not (occurs tv1 t2) then tv1.v <- t2
  else failwith "unify: recursive type occured"
| (Tint, Tint) -> ()
| (Tbool, Tbool) -> ()
| (Tarrow (t1, t2), Tarrow(t'1, t'2))
  -> unify t1 t'1; unify t2 t'2;
| (Tpair (t1, t2), Tpair(t'1, t'2))
```



```
  -> unify t1 t'1; unify t2 t'2;
| (Tlist t1, Tlist t2) -> unify t1 t2;
| _ -> failwith "unify: type does not match"

let rec type_of_expr tenv = function
    Const (Int _) -> Tint
  | Const (Bool _) -> Tbool
  | Const (Nil) ->
      Tlist(new_type_variable ())
  | Const (_) -> failwith "type_of_expr: should not happen"
  | Plus (e1, e2) ->
      unify (type_of_expr tenv e1) Tint;
      unify (type_of_expr tenv e2) Tint; Tint
  | Minus (e1, e2) ->
      unify (type_of_expr tenv e1) Tint;
      unify (type_of_expr tenv e2) Tint; Tint
  | Times (e1, e2) ->
      unify (type_of_expr tenv e1) Tint;
      unify (type_of_expr tenv e2) Tint; Tint
  | Div (e1, e2) ->
      unify (type_of_expr tenv e1) Tint;
      unify (type_of_expr tenv e2) Tint; Tint
  | Equal (e1, e2) ->
      failwith "exercise1"
  | ConsExp (e1, e2) ->
      failwith "exercise1"
  | PairExp (e1, e2) ->
      Tpair (type_of_expr tenv e1, type_of_expr tenv e2)
  | IfExp (e1, e2, e3) ->
      unify (type_of_expr tenv e1) Tbool;
      let u = new_type_variable () in
      unify (type_of_expr tenv e2) u;
      unify (type_of_expr tenv e3) u;
      u
  | Var s ->
      let tscheme = try
        List.assoc s tenv
      with _ -> failwith ("Unbound variable: " ^ s)
      in instanciate tscheme
  | LambdaExp([IdentPtn x, e]) ->
      let alpha = new_type_variable () in
      let ts = Forall([], alpha) in
      Tarrow(alpha, type_of_expr ((x, ts) :: tenv) e)
  | App(e1, e2) ->
      failwith "exercise2-1"
  | LetExp([IdentPtn s,e1],e2) ->
      let t1 = type_of_expr tenv e1 in
      let ts = generalize tenv t1 in
      type_of_expr ((s,ts)::tenv) e2
  | LetRecExp([IdentPtn s,e1],e2) ->
      failwith "exercise2-2"
(* extensions *)
  | LetExp(ptns, e2) ->
      failwith "Unimplemented (optional excesise 4-1)"
  | LetRecExp(ptns, e2) ->
      failwith "Unimplemented (optional excesise 4-2)"
  | LambdaExp(ptns) ->
      failwith "Unimplemented (optional excesise 3)"
  | MatchExp(e1,ptns) ->
      failwith "Unimplemented (optional excesise 3)"
  | TopLetExp _ | TopLetRecExp _ ->
      failwith "type_of_expr: top level phrase not supported"
```

```
and pattern_type = function
    (* for optional excesises *)
    ConstPtn v -> (type_of_expr [] (Const v), [])
  | IdentPtn "_" ->
      (new_type_variable (), [])
  | IdentPtn s ->
      let alpha = new_type_variable () in
      (alpha, [s, alpha])
  | ConsPtn(v1, v2) ->
      let (t1, b1) = pattern_type v1 in
      let (t2, b2) = pattern_type v2 in
      unify t2 (Tlist(t1));
      (t2, b1 @ b2)
  | PairPtn(v1, v2) ->
      let (t1, b1) = pattern_type v1 in
      let (t2, b2) = pattern_type v2 in
      (Tpair(t1, t2), b1 @ b2)

(* for trial *)
let rec shorten_all t = match t with
  TVar ({ id = _; v = TUnknown }) -> t
| TVar tv -> shorten_all tv.v
| TUnknown -> failwith "shorten: should not happen"
| Tarrow(t1, t2) -> Tarrow(shorten_all t1, shorten_all t2)
| Tlist(t1) -> Tlist(shorten_all t1)
| t -> t

let test_tenv =
  let alpha = TVar { id = 1; v = TUnknown } in
  let beta = TVar { id = 2; v = TUnknown } in
  let hd_typescheme =
    Forall([1], Tarrow(Tlist alpha, alpha)) in
  let tl_typescheme =
    Forall([1], Tarrow(Tlist alpha, Tlist alpha)) in
  let fst_typescheme =
    Forall([2; 1], Tarrow(Tpair(alpha, beta), alpha)) in
  let snd_typescheme =
    Forall([2; 1], Tarrow(Tpair(alpha, beta), beta)) in
  ["hd", hd_typescheme;
   "tl", tl_typescheme;
   "fst", fst_typescheme;
   "snd", snd_typescheme;
 ];;

let type_of_expr e = shorten_all (type_of_expr test_tenv e);;

(* you can use this function as a blackbox *)
open Format
let print_mltypes t =
  let n = ref (-1) in
  let vars = ref [] in
  let rec iter level ppf =
    function
        Tint -> fprintf ppf "int"
      | Tbool -> fprintf ppf "bool"
      | Tlist x -> fprintf ppf "@[%a list@]" (iter 2) x
      | Tpair(x,y) as z ->
          if level >= 2 then
            fprintf ppf "(@[%a@])" (iter 0) z
          else fprintf ppf "@[<1>%a@ * %a@]" (iter 2) x (iter 2) y
      | Tarrow(x,y) as z ->
```



```
          if level >= 1 then
            fprintf ppf "(@[%a@])" (iter 0) z
          else
            fprintf ppf "@[<1>%a ->@ %a@]" (iter 1) x (iter 0) y
      | TVar { id = id; v = TUnknown } ->
          let varnum =
            try List.assoc id !vars
            with _ -> (incr n;
                       vars := (id, !n)::!vars;
                       !n)
          in
          fprintf ppf "'%c" (char_of_int (int_of_char 'a' + (varnum mod 26)));
          if (varnum > 26) then
            fprintf ppf "%d" (varnum / 26)
      | TVar { id = _; v = v } -> iter level ppf v
      | TUnknown -> failwith "print_mltypes: should not happen"
  in
  printf "%a" (iter 0) (shorten_all t)

(*

#load "miniML.cmo";;
#load "miniMLLexer.cmo";;
#load "miniMLParser.cmo";;
#load "miniMLReader.cmo";;
open MiniMLReader;;

(* if you have compiled this file *)
#load "miniMLTyping.cmo";;
open MiniMLTyping;;

(* if you want formatted outputs like "'a -> 'a" *)
#install_printer print_mltypes;;

*)
```

```
let read () =
  let lexbuf = Lexing.from_channel stdin in
  MiniMLParser.main MiniMLLexer.token lexbuf

let mlexp_of_string s =
  let lexbuf = Lexing.from_string (s ^ ";;") in
  MiniMLParser.main MiniMLLexer.token lexbuf

(*

#load "miniMLLexer.cmo";;
#load "miniMLParser.cmo";;
#load "miniMLReader.cmo";;
open MiniML;;
open MiniMLReader;;

*)
```



```
SOURCES = miniMLParser.mly miniMLLexer.mll miniML.ml \
        miniMLReader.ml miniMLTyping.ml
A2PSOPTIONS = -2 -A fill $(A2PSPRINTER)

all: miniMLReader.cmo miniMLLexer.cmo miniMLParser.cmo \
        miniMLTyping.cmo

%.mli %.ml : %.mly
        ocamlyacc $<

%.ml : %.mll
        ocamllex $<

%.cmi : %.mli
        ocamlc -c $<

%.cmo : %.ml
        ocamlc -c $<

depend:
        ocamldep $(SOURCES) > .depend

miniMLLexer.ml: miniMLLexer.mll
miniMLParser.ml: miniMLParser.mly

miniMLLexer.cmo: miniML.cmo miniMLParser.cmi
miniMLParser.cmo: miniML.cmo


clean:
        rm -f *.cm? miniMLLexer.ml miniMLParser.mli miniMLParser.ml

a2ps:
        a2ps $(A2PSOPTIONS) $(SOURCES) Makefile

miniMLReader.cmo: miniMLLexer.cmo miniMLParser.cmo
miniMLReader.cmx: miniMLLexer.cmx miniMLParser.cmx
miniMLInterp-full.cmo: miniML.cmo
miniMLInterp-full.cmx: miniML.cmx
miniMLTyping.cmo: miniML.cmo
miniMLTyping.cmx: miniML.cmx
```