The goal of this assignment is to program the algorithm W for MINI-ML. We will consider a version of a destructive unification.

Type and type variables

We begin by introducing the abstract syntax of types

```
type typ =
  | Tint
  | Tarrow   of typ * typ
  | Tproduct of typ * typ
  | Tvar     of tvar

and tvar =
  { id : int;
    mutable def : typ option }
```

The type `typ` of MINI-ML types is mutually recursive with the type `tvar` of type variables. Indeed, a type variable contains a possible definition (obtained during unification), which is a type. Thus, `{ id = 1; def = None }` is "still" a type variable, but `{ id = 2; def = Some Tint }` is an old type variable that has been defined as being equal to the type `Tint` and must therefore now be treated exactly as if it were the type `Tint`.

To test this, you can use the following print function:

```
let rec pp_typ fmt = function
  | Tproduct (t1, t2) -> Format.fprintf fmt "%a *@ %a" pp_atom t1 pp_atom t2
  | Tarrow (t1, t2) -> Format.fprintf fmt "%a ->@ %a" pp_atom t1 pp_typ t2
  | (Tint | Tvar _) as t -> pp_atom fmt t
and pp_atom fmt = function
  | Tint -> Format.fprintf fmt "int"
  | Tvar v -> pp_tvar fmt v
  | Tarrow _ | Tproduct _ as t -> Format.fprintf fmt "@[<1>(%a)@]" pp_typ t
and pp_tvar fmt = function
  | { def = None; id } -> Format.fprintf fmt "'%d" id
  | { def = Some t; id } -> Format.fprintf fmt "@[<1>('%d := %a)@]" id pp_typ t
```

Next, we introduce a module `V` encapsulating the `tvar` type with a comparison function and a function that creates a new variable of type.

```
module V = struct
  type t = tvar
  let compare v1 v2 = Stdlib.compare v1.id v2.id
  let equal v1 v2 = v1.id = v2.id
```

```
  let create = let r = ref 0 in fun () -> incr r; { id = !r; def = None }
end
```

1 Normalization

To facilitate the management of type variables, we will introduce two functions that normalize a type expression by following the definitions contained in the type variables.

1.1

Write a function

```
val head: typ -> typ
```

which normalizes a head type, i.e. , `head t` returns a type equal to `t` that is not of the form `Tvar { def = Some _}`. In other words, `head t` follows the variable definitions of head types of `t` , as long as there are any.

1.2

Write a function

```
val canon: typ -> typ
```

which normalizes a type in its entirety, i.e. , which applies the `head` function above in depth. This second function will be used only for displaying types (in the result of a type checking operation or in error messages).

We can test with

```
let () =
  let a = V.create() in
  let b = V.create() in
  let ta = Tvar a in
  let tb = Tvar b in
  assert (head ta == ta);
  assert (head tb == tb);
  let ty = Tarrow (ta, tb) in
  a.def <- Some tb;
  assert (head ta == tb);
  assert (head tb == tb);
  b.def <- Some Tint;
  assert (head ta = Tint);
```

```
  assert (head tb = Tint);
  assert (canon ta = Tint);
  assert (canon tb = Tint);
  assert (canon ty = Tarrow (Tint, Tint))
```

2 Unification

To signal unification errors, we define the following exception and function:

```
exception UnificationFailure of typ * typ

let unification_error t1 t2 = raise (UnificationFailure (canon t1, canon
↪  t2))
```

2.1

Write a function

```
val occurs: tvar -> typ -> bool
```

which tests for the occurrence of a variable of type `int` within a type ('occur-check'). We can assume that the variable is undefined. However, the type may contain defined variables, and it will be necessary to apply `head` to it before examining it.

2.2

Write a function

```
val unify: typ -> typ -> unit
```

achieving the unification of two types. Here again, it will be necessary to use the `head` function on the types passed as arguments before examining them.

We can test with

```
let () =
  let a = V.create() in
  let b = V.create() in
  let ta = Tvar a in
  let tb = Tvar b in
  assert (occur a ta);
  assert (occur b tb);
  assert (not (occur a tb));
```

```
  let ty = Tarrow (ta, tb) in
  assert (occur a ty);
  assert (occur b ty);
  (* unifies 'a-> 'b and int->int *)
  unify ty(Tarrow(Tint, Tint));
  assert (canon ta = Tint);
  assert(canon ty = Tarrow(Tint, Tint));
  (* unifies 'c' and int->int *)
  let c = V.create() in
  let tc = Tvar c in
  unify tc ty;
  assert (canon tc = Tarrow (Tint, Tint))

let cant_unify ty1 ty2 =
  try let _ = unify ty1 ty2 in false with UnificationFailure _ -> true

let () =
  assert(cant_unify Tint(Tarrow(Tint, Tint)));
  assert(cant_unify Tint(Tproduct(Tint, Tint)));
  let a = V.create() in
  let ta = Tvar a in
  unify ta(Tarrow(Tint,Tint));
  assert (can_unify ta Tint)
```

3 Free variable of a type

To represent the set of free variables of a type, we use the Set module of Caml.

```
module Vset = Set.Make(V)
```

3.1

Write a function

```
val fvars : typ -> Vset.t
```

which calculates the set of free variables of a type. Once again, it is important to use the head function judiciously to consider only the variables that are not defined.

We can test with

```
let () =
  assert(Vset.is_empty(fvars(Tarrow(Tint,Tint))));
  let a = V.create() in
```

```
  let ta = Tvar a in
  let ty = Tarrow (ta, ta) in
  assert(Vset.equal(fvars ty)(Vset.singleton a));
  unify ty(Tarrow(Tint, Tint));
  assert(Vset.is_empty(fvars ty))
```

4 Typing environment

The following OCaml type is defined for type schema:

```
type schema = { vars: Vset.t; typ : typ }
```

Dictionaries using character strings as keys are then introduced.

```
Smap module = Map.Make(String)
```

then the following OCaml type for typing environments:

```
type env = { bindings: schema Smap.t; fvars: Vset.t }
```

The first field, ‘bindings‘ , contains the elements of the type environment. The second, ‘fvars‘ , contains the set of all free-type variables in the entries of ‘ bindings‘ . The ‘fvars‘ field is there only to avoid recalculating the set of free-type variables in the environment each time during the generalization operation. Note: some of these variables may be defined between the time they are included in this set and the time the set is used. Therefore, this set should be updated before use (for example, by applying ‘fvars(Tvar v) ‘ to each variable ‘v‘ and unioning the results).

Define the empty typing environment:

```
val empty : env
```

4.1

Write the function

```
val add: string -> typ -> env -> env
```

This adds an element to a typesetting environment without generalization (it will be used for function typing). Remember to update the fvars field of the environment.

4.2

Write the function

```
val add_gen: string -> typ -> env -> env
```

which adds an element to a typing environment by generalizing its type with respect to all its free type variables not appearing in the environment (it will be used in the typing of a let).

4.3

Write the function

```
val find: string -> env -> typ
```

which returns the type associated with an identifier in an environment, after replacing all variables in the corresponding schema with variables of fresh types. Note: there may be multiple occurrences of the same variable in the type, and it is essential to replace them all with the same fresh variable. Hint: You can introduce a module ‘Vmap = Map.Make(V)‘ and use a value of type ‘tvar Vmap.t‘ to represent this substitution.

5 Algorithm W

We define the following type for MINI-ML expressions:

```
type expression =
  | Var of string
  | Const of int
  | Op of string
  | Fun of string * expression
  | App of expression * expression
  | Pair of expression * expression
  | Let of string * expression * expression
```

5.1

Write the function

```
val w: env -> expression -> typ
```

implementing algorithm W. For the operators (Op), we can simply use Op "+" of type int * int -> int .

To test, we will use the following function

```
let typeof e = canon (w empty e)
```

Some positive tests (the expression and expected type are indicated in the comments):

```
(* 1 : int *)
let() = assert(typeof(Const 1) = Tint)

(* fun x -> x : 'a -> 'a *)
let () = assert (match typeof (Fun ("x", Var "x")) with
  | Tarrow (Tvar v1, Tvar v2) -> V.equal v1 v2
  | _ -> false)

(* fun x -> x+1: int -> int *)
let () = assert (typeof (Fun ("x", App (Op "+", Pair (Var "x", Const 1))))
                 = Tarrow (Tint, Tint))

(* fun x -> x+x: int -> int *)
let () = assert (typeof (Fun ("x", App (Op "+", Pair (Var "x", Var "x"))))
                 = Tarrow (Tint, Tint))

(* let x = 1 in x+x : int *)
let () =
  assert (typeof (Let ("x", Const 1, App (Op "+", Pair (Var "x", Var "x"))))
          (Tint)

(* let id = fun x -> x in id 1 *)
let () =
  assert (typeof (Let ("id", Fun ("x", Var "x"), App (Var "id", Const 1)))
          (Tint)

(* let id = fun x -> x in id id 1 *)
let () =
  assert (typeof (Let("id", Fun("x", Var "x"),
                      App(App(Var "id", Var "id"), Const 1)))
          (Tint)

(* let id = fun x -> x in (id 1, id (1,2)): int * (int * int) *)
let () =
  assert (typeof (Let("id", Fun("x", Var "x"),
                      Pair(App(Var "id", Const 1),
                           App(Var "id", Pair(Const 1, Const 2)))))
          = Tproduct(Tint, Tproduct(Tint, Tint)))

(* app = fun fx -> let y = fx in y: ('a -> 'b) -> 'a -> 'b *)
```

```
let () =
  let ty =
    typeof(Fun("f", Fun("x", Let("y", App(Var "f", Var "x"), Var "y"))))
  in
  assert (match ty with
    | Tarrow (Tarrow (Tvar v1, Tvar v2), Tarrow (Tvar v3, Tvar v4)) ->
        V.equal v1 v3 && V.equal v2 v4
    | _ -> false)
```

Some negative tests:

```
let cant_type e =
  try let _ = typeof e in false with UnificationFailure _ -> true

(* 1 2 *)
let () = assert (cant_type (App (Const 1, Const 2)))

(* fun x -> xx *)
let () = assert (cant_type (Fun ("x", App (Var "x", Var "x"))))

(* (fun f -> +(f 1)) (fun x -> x) *)
let() = assert(cant_type)
                  (App (Fun ("f", App (Op "+", App (Var "f", Const 1))),
                        Fun("x", Var "x"))))

(* fun x -> (x 1, x (1,2)) *)
let() = assert(cant_type)
                  (Fun ("x", Even (App (Var "x", Const 1),
                                   App (Var "x", Even (Const 1, Const
                                   ↪  2))))))

(* fun x -> let z = x in (z 1, z (1,2)) *)
let() = assert(cant_type)
                  (Fun ("x",
                        Let ("z", Var "x",
                             Pair (App (Var "z", Const 1),
                                   App (Var "z", Even (Const 1, Const
                                   ↪  2)))))))

(* let distr_pair = fun f -> (f 1, f (1,2)) in distr_pair (fun x -> x) *)
let () =
  assert (cant_type
            (Let ("distr_pair",
```

```
                Fun ("f", Even (App (Var "f", Const 1),
                                App(Var "f", Even(Const 1, Const 2)))),
                App (Var "distr_pair", (Fun ("x", Var "x"))))))
```