```ocaml
(** A Monoid is an associative operation with an identity element.
  Examples include addition with 0, or multiplication with 1.
  In this signature, the operation is called op, the identity element id **)
  module type Monoid =
  sig
    type t
    (** id must be a left identity for op, i.e.
        [op id x = x]
        And id must also be a right identity, i.e.
        [op x id = x] **)
    val id : t
    (* op must be associative, i.e.
          [op (op x y) z = op x (op y z)] *)
    val op : t -> t -> t
  end

  (* The plus instance is as in the previous homework: *)

  type nat = Zero | S of nat

  module Plus =
  struct
      type t = nat
      let rec plus a b =
        match b with
        | Zero -> a
        | S i -> plus (S a) i
      let op = plus
      let id = Zero
  end

  (* The Max instance takes the maximum of two numbers: *)

  module Max =
  struct
      type t = nat
      let rec max a b =
        match (a, b) with
          | (Zero, x) -> x
          | (x, Zero) -> x
          | (S x, S y) -> S (max x y)
      let op = max
      let id = Zero
  end

  (* The Append instance has 'append' as its operation and the empty list as identity element:
*)

  module Append =
  struct
      type t = int list
      let rec append a b =
        match a with
          | [] -> b
          | h::tl -> h::(append tl b)
      let op = append
      let id = []
  end

  (* This is the lab exercise for October 24th:
      prove that Append satisfies the properties listed in the Monoid signature.
      The following takes care of the type-checking: *)
  let _ = (module Append : Monoid)
  (* We just need to add proofs that show that:
      - Append.op is associative (proof is in the slides!) Append.op a (Append.op b c) =
Append.op (Append.op a b) c
      - Append.id is a left identity (this one is easy) Append.op Append.id a = a
```

```
     - Append.id is a right identity (this one is a straightforward induction on lists)
Append.op a Append.id = a
     *)



  1) left identity
  proof: Append.op Append.id a = a

  case a = []

      Append.op Append.id a = a
  = { case }
      Append.op Append.id []
  = { Append.id def }
      let id = []
  = { apply Append.id }
      Append.op [] []
  = { Append def }
      type t = int list
      let rec append [] [] =
      match a with
        | [] -> b
        | h::tl -> h::(append tl b)
      let op = append
      let id = []
  = { apply match }
      []
  = { case }
      a

      case a = (h :: tl)

      Append.op Append.id a
  = { case }
      Append.op Append.id (h :: tl)
  = { Append.id def }
      let id = []
  = { apply Append.id }
      Append.op [] (h :: tl)
  = { Append def }
      type t = int list
      let rec append [] (h :: tl) =
      match a with
        | [] -> b
        | h::tl -> h::(append tl b)
      let op = append
      let id = []
  = { apply match }
      (h :: tl)
  = { case }
      a


  2) right identity
  proof: Append.op a Append.id = a

  case: a = []

      Append.op a Append.id
  = { case }
      Append.op [] Append.id
  = { Append.id def }
      let id = []
  = { apply Append.id }
      Append.op [] []
  = { Append def }
```

```
    type t = int list
    let rec append [] [] =
    match a with
      | [] -> b
      | h::tl -> h::(append tl b)
    let op = append
    let id = []
= { apply match }
    []
= { case }
    []


case: a = h :: tl
    Append.op a Append.id
= { case }
    Append.op (h :: tl) Append.id
= { Append.id def }
    let id = []
= { apply Append.id }
    Append.op (h :: tl) []
= { append def }
    type t = int list
    let rec append (h :: tl) [] =
    match a with
      | [] -> b
      | h::tl -> h::(append tl b)
    let op = append
    let id = []
= {apply append}
    h :: tl
= { case }
    a



3) associative
proof: Append (Append x y) z = Append x (Append y z)

{IH}: Append (Append (tl) y) z = Append (tl) (Append y z)

case: x = []

    Append (Append x y) z
= { case }
    Append (Append [] y) z
= { Append def }
    type t = int list
    let rec append a b =
    match a with
      | [] -> b
      | h::tl -> h::(append tl b)
    let op = append
    let id = []
= { apply Append }
    Append y z
= {Lemma: 1) }
    Append [] (Append y z)
= { case }
    Append x (Append y z)


case: x = h :: tl
    Append (Append x y) z
= { case }
    Append (Append (h :: tl) y) z
= { Append def }
```

```
    Append (
    type t = int list
    let rec append a b =
    match a with
      | [] -> b
      | h::tl -> h::(append tl b)
    let op = append
    let id = []
    )
= { apply Append }
    h :: Append (Append (tl) y) z
= { IH }
    h ::  Append (tl) (Append y z)
= { append def }
    type t = int list
    let rec append a b =
    match a with
      | [] -> b
      | h::tl -> h::(append tl b)
    let op = append
    let id = []
= { reverse match }
    Append (h :: tl) (Append y z)
= { case }
    Append (x) (Append y z)


let _ = (module Plus : Monoid)
  (* Proofs that this is true were in the previous homework,
    you don't have to repeat them in this homework.
    (On October 24th, I will include them myself.)
    *)

let _ = (module Max : Monoid) (* Proofs for this you have to write still *)

(* On associativity of Max.op:
    You will need some case distinction inside your inductive step.
    Consider these cases in the inductive step:
      - b = Zero
      - c = Zero
      - b = S b' and c = S c'
    (Why are these the only cases you need to consider?) *)

1) left identity
Proof: Max.op Max.id a = a

cast a = Zero

  Max.op Max.id a
= { case }
  Max.op Max.id Zero
= { Max.id def}
  Max.op Zero Zero
= { Max.op def }
  type t = nat
  let rec max Zero Zero =
    match (Zero, Zero) with
      | (Zero, x) -> x
      | (x, Zero) -> x
      | (S x, S y) -> S (max x y)
  let op = max
  let id = Zero
= { apply match }
  Zero
= { case }
  a
```

```
cast a = S a

  Max.op Max.id a
= { case }
  Max.op Max.id S a
= { Max.id def}
  Max.op Zero S a
= { Max.op def }
  type t = nat
  let rec max Zero (S a) =
    match (Zero, S a) with
      | (Zero, x) -> x
      | (x, Zero) -> x
      | (S x, S y) -> S (max x y)
  let op = max
  let id = Zero
= { apply match }
  S a
= { case }
  a

  type t = nat
  let rec max a b =
    match (a, b) with
      | (Zero, x) -> x
      | (x, Zero) -> x
      | (S x, S y) -> S (max x y)
  let op = max
  let id = Zero


2) right identity
Proof: Max.op a Max.id = a

cast a = Zero

  Max.op a Max.id
= { case }
  Max.op Zero Max.id
= { Max.id def}
  Max.op Zero Zero
= { Max.op def }
  type t = nat
  let rec max Zero Zero =
    match (Zero, Zero) with
      | (Zero, x) -> x
      | (x, Zero) -> x
      | (S x, S y) -> S (max x y)
  let op = max
  let id = Zero
= { apply match }
  Zero
= { case }
  a

cast a = S a

  Max.op a Max.id
= { case }
  Max.op (S a) Max.id
= { Max.id def}
  Max.op (S a) Zero
= { Max.op def }
  type t = nat
  let rec max (S a) Zero =
    match ((S a), Zero) with
      | (Zero, x) -> x
```

```
        | (x, Zero) -> x
        | (S x, S y) -> S (max x y)
    let op = max
    let id = Zero
  = { apply match }
    (S a)
  = { case }
    a




  3) associative
  Proof: Max.op a (Max.op b c) = Max.op (Max.op a b) c
    (* On associativity of Max.op:
      You will need some case distinction inside your inductive step.
      Consider these cases in the inductive step:
        - b = Zero
        - c = Zero
        - b = S b' and c = S c'
      (Why are these the only cases you need to consider?) *)
```

```
  lemma:
  prove: Max.op (S a) b = S(Max.op a b)

  base case: given that b == Zero; Proof Max.op (S a) b = S(Max.op a b)

    Max.op (S a) b
  = { case }
    Max.op (S a) Zero
  = { Max.op def }
    type t = nat
    let rec max a b =
      match (a, b) with
        | (Zero, x) -> x
        | (x, Zero) -> x
        | (S x, S y) -> S (max x y)
    let op = max
    let id = Zero
  = { apply match }
      S(a)
  = { reverse match }
    S(
      type t = nat
      let rec max a Zero =
        match (a, Zero) with
          | (Zero, x) -> x
          | (x, Zero) -> x
          | (S x, S y) -> S (max x y)
      let op = max
      let id = Zero
    )
  = { reverse Max.op }
    S(Max.op a Zero)
  = { case }
    S(Max.op a b)


  Induction: for any given a, let b == (S b)
  { IH }: Max.op (S a) b = S(Max.op a b)

    Max.op (S a) b
  = { case }
    Max.op (S a) (S b)
  = { Max.op def }
```

```
    type t = nat
    let rec max (S a) (S b) =
      match ((S a), (S b)) with
        | (Zero, x) -> x
        | (x, Zero) -> x
        | (S x, S y) -> S (max x y)
    let op = max
    let id = Zero
  = { apply match }
    S( Max.op a b )
```

```
  base case b = Zero

    Max.op a (Max.op b c)
  = { case }
    Max.op a (Max.op Zero c)
  = { Max.op def }
    Max.op a (
      type t = nat
      let rec max Zero c =
        match (Zero, c) with
          | (Zero, x) -> x
          | (x, Zero) -> x
          | (S x, S y) -> S (max x y)
      let op = max
      let id = Zero
    )
  = { apply match }
    Max.op a c
  = { reverse match }
    Max.op (
      type t = nat
      let rec max a Zero =
        match (a, Zero) with
          | (Zero, x) -> x
          | (x, Zero) -> x
          | (S x, S y) -> S (max x y)
      let op = max
      let id = Zero
    ) c
  = { reverse Max.op}
    Max.op (Max.op a Zero) c
  = { case }
    Max.op (Max.op a b) c

  base case c = Zero

    Max.op a (Max.op b c)
  = { case }
    Max.op a (Max.op b Zero)
  = { Max.op def }
    Max.op a (
      type t = nat
      let rec max b Zero =
        match (b, Zero) with
          | (Zero, x) -> x
          | (x, Zero) -> x
          | (S x, S y) -> S (max x y)
      let op = max
      let id = Zero
    )
  = { apply match }
    Max.op a b
  = { right identity }
    Max.op (Max.op a b) Zero
```

```
 = { case }
   Max.op (Max.op a b) c


 Induction: given any a, let b = (S b) and c = (S c)
 { IH }: Max.op a (Max.op b c) = Max.op (Max.op a b) c

   Max.op a (Max.op b c)
 = { case }
   Max.op a (Max.op (S b) (S c))
 = { Max.op def }
   MAx.op a (
     type t = nat
     let rec max (S b) (S c) =
       match ((S b), (S c)) with
         | (Zero, x) -> x
         | (x, Zero) -> x
         | (S x, S y) -> S (max x y)
     let op = max
     let id = Zero
   )
 = { apply match }
   Max.op a (S (max b c))
 = { apply lemma }
   S(Max.op a (Max.op b c))
 = { IH }
   S(Max.op (Max.op a b) c)
 = { reverse match }
   type t = nat
   let rec max S(Max.op a b) (S c) =
     match (S(Max.op a b), (S c)) with
       | (Zero, x) -> x
       | (x, Zero) -> x
       | (S x, S y) -> S (max x y)
   let op = max
   let id = Zero
 = {reverse Max.op }
   Max.op S(Max.op a b) (S c)
 = {reverse lemma }
   Max.op (Max.op a (S b)) (S c)
 = { case }
   Max.op (Max.op a b) c


 module Combine (M : Monoid) = struct
     let rec combine_r lst =
       match lst with
       | []   -> M.id
       | h :: t -> M.op h (combine_r t)

     let rec combine_l acc lst =
       match lst with
       | []   -> acc
       | h :: t -> (combine_l (M.op acc h) t)
 end

4) proof: Combine_r lst = Combine_l M.id lst

 (* To prove that [combine_r lst = combine_l M.id lst], you need to prove a stronger lemma.
     The lemma is that [M.op a (combine_r lst) = combine_l a lst] for any a.
     You can prove this by induction on lst.
     Using this lemma, you can prove the original theorem by setting a = M.id.
     *)

 case lst = []

   Combine_r lst
```

```
  = { case }
    Combine_r []
  = { Combine_r def }
    let rec combine_r lst =
      match [] with
      | []   -> M.id
      | h :: t -> M.op h (combine_r t)
  = { apply match }
    M.id
  = { reverse match }
    let rec combine_l acc lst =
      match [] with
      | []   -> acc
      | h :: t -> (combine_l (M.op acc h) t)
  = { reverse Combine_l }
    Combine_l M.id []
    = { case }
    Combine_l M.id lst
```

```
lemma 2:
Proof: M.op M.id a = a

case M = Plus

  M.op M.id a = a
= { case }
  Plus.op Plus.id a = a
= { Plus.id = Zero }
  Plus.op Zero a
= { proved in last week's hw's Plus's left identity }
  a

case M = Append
  M.op M.id a = a
= { case }
  Append.op Append.id a = a
= { Append.id = [] }
  Append.op [] a
= { proved in this week's hw's Append's left identity }
  a

case M = Max
  M.op M.id a = a
= { case }
  Max.op Max.id a = a
= { Max.id = Zero }
  Max.op Zero a
= { proved in this week's lab's Max's left identity }
  a
```

```
lemma 3:
Induction: let M = Append, Plus or Max
{ IH }: combine_r M.id = M.id

case M = Append
  combine_r M.id
= { case }
  combine_r Append.id
= { Append.id def }
  combine_r []
= { combine_r def }
  let rec combine_r lst =
    match [] with
    | []   -> M.id
```

```
    | h :: t -> M.op h (combine_r t)
= { apply match }
  Append.id
= { case }
  M.id


case M = Plus

  combine_r M.id
= { case }
  combine_r Plus.id
= { Plus.id def }
  combine_r Zero
= { combine_r def }
  let rec combine_r lst =
    match Zero with
    | []   -> M.id
    | h :: t -> M.op h (combine_r t)
= { apply match }
  Plus.op Zero (combine_r [])
= { combine_r def }
  Plus.op Zero (
    let rec combine_r lst =
      match [] with
      | []   -> M.id
      | h :: t -> M.op h (combine_r t)
  )
= { apply match }
  Plus.op Zero (combine_r Plus.id)
= { IH }
  Plus.op Zero Plus.id
= { Plus.id def }
  Plus.op Zero Zero
= { Plus.op def }
  type t = nat
  let rec plus a b =
    match b with
    | Zero -> a
    | S i -> plus (S a) i
  let op = plus
  let id = Zero
= { apply match }
  Zero
= { Plus.id def }
  Plus.id
= { case }
  M.id


case M = Max

  combine_r M.id
= { case }
  combine_r Max.id
= { Max.id def }
  combine_r Zero
= { combine_r def }
  let rec combine_r lst =
    match Zero with
    | []   -> M.id
    | h :: t -> M.op h (combine_r t)
= { apply match }
  Max.op Zero (combine_r [])
= { combine_r def }
  Max.op Zero (
    let rec combine_r lst =
```

```
      match [] with
      | []   -> M.id
      | h :: t -> M.op h (combine_r t)
  )
= { apply match }
  Max.op Zero (combine_r Max.id)
= { IH }
  MAx.op Zero Max.id
= { Max.id def }
  Max.op Zero Zero
= { Max.op def }
  type t = nat
  let rec max a b =
    match (a, b) with
      | (Zero, x) -> x
      | (x, Zero) -> x
      | (S x, S y) -> S (max x y)
  let op = max
  let id = Zero
= { apply match }
  Zero
= { Max.id def }
  Max.id
= { case }
  M.id
```

```
Induction: let lst = (h :: tl)
{ IH }: Combine_r lst = Combine_l M.id lst

    Combine_r lst
  = { case }
    Combine_r (h :: tl)
  = { Combine_r def }
    let rec combine_r lst =
      match [] with
      | []   -> M.id
      | h :: t -> M.op h (combine_r t)
  =  { apply match }
    M.op h (combine_r tl)
  = { after going through the whole list }
    M.op h (M.op tl (combine_r M.id))
  = { lemma 3}
    M.op h (M.op tl M.id)
  = { right identity }
    M.op h tl
  = { right identity }
    combine_l (M.op h tl) M.id
  = { reverse lemma 2}
    combine_l (M.op (M.op M.id h) tl) M.id
  = { after reversing through the whole list }
    combine_l (M.op M.id h) tl
  = { reverse match }
    let rec combine_l acc lst =
      match (h :: tl) with
      | []   -> acc
      | h :: t -> (combine_l (M.op acc h) t)
  = { Combine_l def }
    Combine_l M.id (h :: tl)
  = { case }
    Combine_l M.id lst


(* doing from the right *)
    (* Combine_l M.id lst
  = { case }
```

```
    Combine_l M.id (h :: tl)
  = { Combine_l def }
    let rec combine_l acc lst =
      match (h :: tl) with
      | []   -> acc
      | h :: t -> (combine_l (M.op acc h) t)
  = { apply match }
    combine_l (M.op M.id h) tl
  = { after going through the whole list }
    combine_l (M.op (M.op M.id h) tl) M.id
  = { lemma 2 }
    combine_l (M.op h tl) M.id *)


  (*
  Testing associativity and identity element properties:
  *)
  module type MonoidWithValues =
  sig
      include Monoid
      val values : (t*t*t)
  end

  module AppendV = struct
      include Append
      let values = ([2;3;4], [5;6], [7;8;9])
  end
  module MaxV = struct
      include Max
      let values = (S (S Zero), S (S (S Zero)), S (S (S (S Zero))))
  end
  module PlusV = struct
      include Plus
      let values = (S (S Zero), S (S (S Zero)), S (S (S (S Zero))))
  end

  let is_assoc op (v1,v2,v3)
    = assert (op (op v1 v2) v3 = op v1 (op v2 v3));
      assert (op (op v1 v3) v2 = op v1 (op v3 v2));
      assert (op (op v1 v2) v2 = op v1 (op v2 v2));
      assert (op (op v1 v3) v3 = op v1 (op v3 v3));
      assert (op (op v2 v1) v3 = op v2 (op v1 v3));
      assert (op (op v2 v3) v1 = op v2 (op v3 v1));
      assert (op (op v3 v1) v2 = op v3 (op v1 v2));
      assert (op (op v3 v2) v1 = op v3 (op v2 v1))

  let is_id op idt (v1,v2,v3)
    = assert (op idt v1 = v1);
      assert (op idt v2 = v2);
      assert (op idt v3 = v3);
      assert (op v1 idt = v1);
      assert (op v2 idt = v2);
      assert (op v3 idt = v3)

  let test_monoidV (module M : MonoidWithValues) =
      is_assoc M.op M.values;
      is_id M.op M.id M.values

  let _ = test_monoidV (module AppendV)
  let _ = test_monoidV (module MaxV)
  let _ = test_monoidV (module PlusV)

  (*
  Testing combine functions:
  *)
  let test_combine (module M : MonoidWithValues) =
      let module C = Combine(M) in
```

```
    let (v1,v2,v3) = M.values in
    assert (C.combine_r [v1;v2;v3] = C.combine_l M.id [v1;v2;v3]);
    assert (C.combine_r [v3;v2;v3] = C.combine_l M.id [v3;v2;v3]);
    assert (C.combine_r [v2;v2;v1] = C.combine_l M.id [v2;v2;v1]);
    assert (C.combine_r [v1;v2;v3] = M.op (M.op v1 v2) v3);
    assert (C.combine_l M.id [v1;v2;v3] = M.op (M.op v1 v2) v3)

let _ = test_combine (module AppendV)
let _ = test_combine (module MaxV)
let _ = test_combine (module PlusV)
```