Artemis 6.6.1 Course Overview ge43fij

Courses > Retake Exam: Functional Programming and Verification (IN0003) SS23 > Exams > Retake Exam: Functional Programming and Verification (IN0003) SS23

Exam Results

General Information

Module number:	IN0003
Course:	Functional Programming and Verification
Examiner:	Prof. Dr. Helmut Seidl
Exam Title:	Retake Exam: Functional Programming and Verification (IN0003) SS23
Date:	Oct 10, 2023
Working Time:	13:35 - 15:35
Duration:	2h
Review Timespan:	Oct 20, 2023 09:30 - Oct 25, 2023 11:59 Review is open
Examined student:	Qichen Liu

Result Overview

#	Exercise	Your Points	Achievable Points	Achieved Percentage
1	Quiz: Weakest Preconditions	6	16	37.5%
2	Quiz: OCaml	6	8	75%
3	Equational Reasoning	10.5	18	58.3%
4	Tail Recursion	0	18	-
5	Modules and Functors	12	28	42.9%
6	Recursive Datatypes	0	32	0%
Total		**34.5**	**120**	**28.7%**

Grade: 5.0

Grade	Interval (%)
5.0	[0 - 30)
4.7	[30 - 35)
4.3	[35 - 40)
4.0	[40 - 45)
3.7	[45 - 50)
3.3	[50 - 55)
3.0	[55 - 60)
2.7	[60 - 65)
2.3	[65 - 70)
2.0	[70 - 75)
1.7	[75 - 80)
1.3	[80 - 85)
1.0	[85 - ∞)

Intervals:

- **[a, b)**: Left boundary is included in and right is excluded from the interval
- **(a, b]**: Left boundary is excluded from and right is included in the interval
- **[a, b]**: Both boundaries are included in the interval

Exercises

#1 Quiz: Weakest Preconditions [6 / 16 Points] 37.5%

1) WP

Your Score: 0/2

Which of the following assertions is logically equivalent to the weakest precondition for the statement `x = x - 1;`, given the postcondition $\forall n.((n > 0 \wedge x > 0) \vee n \leq 0)$?

Please choose the correct answer option

Answer	Solution	You
$x > 1$	Correct	○
$false$	Wrong	○
$n > 0 \wedge x > 1$	Wrong	○
$(n > 0 \wedge x > 1) \vee n \leq 0$	Wrong	◉

2) WP

Your Score: 0/2

Which of the following assertions is logically equivalent to the weakest precondition for the statement `x = 10;`, given the postcondition $(y = x \implies x \leq 5) \wedge (y \neq x \implies x > 5)$?

Please choose the correct answer option

Answer	Solution	You
$y \neq 10$	Correct	○
$x = 10$	Wrong	○
$(y = 10 \implies x \leq 5) \wedge (y \neq 10 \implies x > 5)$	Wrong	◉
$(y = x \wedge x \leq 5) \vee (y \neq x \wedge x > 5)$	Wrong	○

3) WP

Your Score: 0/2

Which of the following assertions is logically equivalent to the weakest precondition for the statement `x = read();`, given the postcondition $i > 0 \wedge y = 2 \cdot i \wedge x > 0 \wedge k \neq 0$?

Please choose the correct answer option

Answer	Solution	You
$false$	Correct	○
$i > 0 \wedge y = 2 \cdot i \wedge k \neq 0$	Wrong	○
$\forall x.(i > 0 \wedge y = 2 \cdot i \wedge k \neq 0)$	Wrong	◉
$\exists x.(i > 0 \wedge y = 2 \cdot i \wedge x > 0 \wedge k \neq 0)$	Wrong	○

4) WP

Your Score: 2/2

Which of the following assertions is logically equivalent to the weakest precondition for the statement `x = y + 1;`, given the postcondition $y > 0 \land x > 0$?

Please choose the correct answer option

Answer	Solution	You
$y > 0$	Correct	◉
$x > 0$	Wrong	○
$x = y + 1$	Wrong	○
$y > 1$	Wrong	○

5) WP

Your Score: 0/2

Which of the following assertions is logically equivalent to the weakest precondition for the condition `i != n`, given the postconditions $B_{true} \equiv q = i! \land i < n$ in the true-case and $B_{false} \equiv q = n!$ in the false-case?

Please choose the correct answer option

Answer	Solution		You
$i \leq n \land q = i!$	Correct	⚠	○
$q = i!$	Wrong	⚠	◉
$i \neq n \land q = i!$	Wrong		○
$q = n!$	Wrong		○

6) WP

Your Score: 2/2

Which of the following assertions is logically equivalent to the weakest precondition for the condition `n >= 0`, given the postconditions $B_{true} \equiv x = 5 \cdot i \land i < n$ in the true-case and $B_{false} \equiv false$ in the false-case?

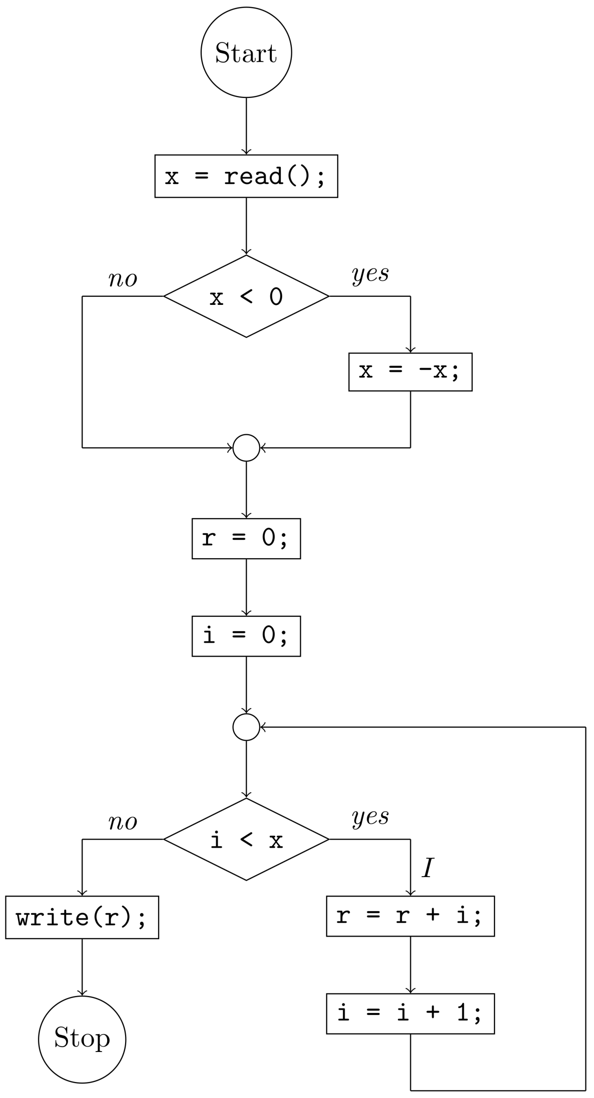
Please choose the correct answer option

Answer	Solution	You
$n >= 0 \land (x = 5 \cdot i \land i < n)$	Correct	◉
$false \lor (x = 5 \cdot i \land i < n)$	Wrong	○
$true \implies (x = 5 \cdot i \land i < n)$	Wrong	○
$n >= 0 \implies (x = 5 \cdot i \land i < n)$	Wrong	○

7) Loop Invariant

Your Score: 0/2

Select the assertion that holds for the following loop at the program point annotated with I:



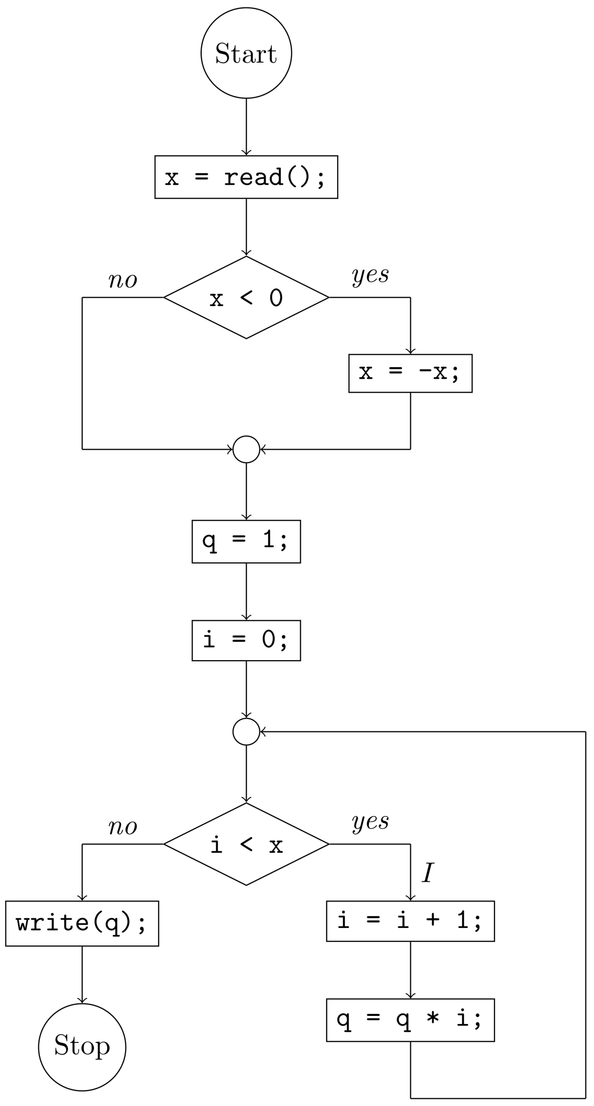
Please choose the correct answer option

Answer	Solution	You
$i < x \wedge r = \sum_{a=0}^{i-1} a$	Correct	○
$i < x \wedge r = \sum_{a=0}^{i} a$	Wrong	◉
$r = \sum_{a=0}^{x} a$	Wrong	○
$i = x$	Wrong	○

8) Loop Invariant

Your Score: 2/2

Select the assertion that holds for the following loop at the program point annotated with I:



Please choose the correct answer option

Answer	Solution	You
$i < x \wedge q = i!$	Correct	◉
$i < x \wedge q = \sum_{a=0}^{i} a$	Wrong	○
$i < x \wedge q = x!$	Wrong	○
$q = x!$	Wrong	○

#2 Quiz: OCaml [6 / 8 Points] 75%

1) Lists and Folding

Your Score: 0/2

Consider the following expression using `List.fold_left`:

```
let ys = List.fold_left (fun a y -> if p y then y :: a else a) [] xs in
List.fold_left (fun a y -> if q y then Some y else a) None ys
```

Which of the following expressions (using standard `List`-module functions) evaluates to the same value as the above expression, for all functions `p`, `q`, and lists `xs`?

Assume that all expressions terminate without raising exceptions.

Please choose the correct answer option

Answer		Solution		You
`List.find_opt q (List.filter p xs)`	Explanation	Correct	⚠	○
`List.find_opt (fun y -> p y || q y) xs`	Explanation	Wrong		○
`List.find_opt (fun y -> p y && q y) (List.rev xs)`	Explanation	Wrong	⚠	◉
`List.filter q (List.filter p xs)`	Explanation	Wrong		○

2) Tail Recursion

Your Score: 2/2

Given the following type definition for binary trees:

```
type 'a tree = Leaf | Node of 'a tree * 'a * 'a tree
```

Which of the following functions is tail recursive?

Please choose the correct answer option

Answer	Solution	You

`let rec size acc = function`
`  | Leaf -> acc`
`  | Node (l, x, r) -> size (size (acc + 1) l) r`
	Wrong	○

```
let rec to_list acc = function
  | Leaf -> List.rev acc
  | Node (l, x, r) ->
      let xs = to_list acc l in
      to_list (x :: xs) r
```

Wrong

```
let rec find_along path t = match t, path with
  | Leaf, _ -> []
  | _, [] -> []
  | Node (l, x, r), b :: xs ->
      if b then x :: find_along xs r
      else x :: find_along xs l
```

Wrong

```
let rec insert acc y = function
  | Leaf -> acc
  | Node (l, x, r) ->
      if y < x then insert ((true, x, r) :: acc) y l
      else insert ((false, x, l) :: acc) y r
```

Correct

3) Input/Output

Your Score: 2/2

Which of the following functions never raises a `Sys_error` exception to its caller?

Hint: The functions `open_out`, `output_string` and `close_out` can raise `Sys_error`.

Please choose the correct answer option

Answer	Solution	You

```
let output_twice s path =
  match (try Some (open_out path) with Sys_error _ -> None) with
  | Some ch ->
      (try output_string ch s; output_string ch s with Sys_error _ -> ());
      close_out ch
  | None -> ()
```

Wrong

```
let output_twice s path =
  try
    let ch = open_out path in
    try
      output_string ch s;
      output_string ch s
    with Sys_error _ -> close_out ch
  with
    Sys_error _ -> ()
```

Correct

```
let output_twice s path =
  let helper f =
    let ch = open_out path in
    try (f ch; close_out ch)
    with Sys_error _ -> close_out ch
  in
  helper (fun ch -> try (output_string ch s; output_string ch s) with Sys_error
_ -> ())
```

Wrong

```
let output_twice s path =
  let helper f =
    let ch = open_out path in
    try f ch with Sys_error _ -> ();
    try close_out ch with Sys_error _ -> ()
  in
  helper (fun ch -> output_string ch s; output_string ch s)
```

Wrong ◯

4) Modules and Functors

Your Score: 2/2

Consider the following incomplete program:

```
module type A = ...

module M : A = struct
  type t = L | R
  let dup x = [x; x]
end
```

Assuming the program compiles, which of the following must hold?

Please choose the correct answer option

Answer	Solution	You
The signature `A` must contain the value `dup`.	Wrong	◯
The signature `A` is allowed to specify `dup` to be of type `int -> int list`.	Correct	◉
The signature `A` is allowed to be a functor module type (`functor ... -> ...`).	Wrong	◯
It is possible to match on values of type `M.t` outside the definition of `M`, as long as the signature `A` contains `type t`.	Wrong	◯

#3 A Equational Reasoning [10.5 / 18 Points] ⊗ 58.3%

You are viewing the example solution

Problem Statement

Equational Reasoning

Given the following definitions (as in the tutorials, we write `let rec` in place of `and` for clarity of presentation):

```
let rec length ls =
  match ls with
  | [] -> 0
  | l :: lst -> 1 + length lst

let rec count k cs =
  match cs with
  | [] -> k
  | c :: cst ->
      count
        (match c with true -> k + 1 | false -> k)
        cst

let rec map f ms =
  match ms with
  | [] -> []
  | m :: mst -> f m :: map f mst
```

Show that the statement:

```
length xs = count 0 (map (fun z -> true) xs)
```

holds for all lists `xs`.

Use equational reasoning. If you prove a generalized claim, show that the generalization can be instantiated to the original claim.

Submission Format

Submissions may only be in the form of plain text.

Your submission **must follow** the following format. Copy this template into your submission and then complete it by replacing the `<...>` with your answers. Leave any fields you don't need blank.

```
[Generalization]
Generalized statement (if necessary) (*): <...>

[Base Case]
Statement being proven in base case: <...>
Proof of base case:
<...>

[Inductive Step]
Induction hypothesis (or hypotheses): <...>
Statement being proven in inductive step: <...>
Proof of inductive step:
<...>

[Instantiation]
Instantiation of generalization (if necessary):
<...>

QED
```

If you need to instantiate a generalized statement, use `*` to refer to the generalized statement.

For all equational proofs that show the equivalence of two MiniOCaml expressions, annotate each step as follows:

```
           e_1
(rule 1) = e_2
(rule 2) = e_3
      ...
(rule n) = e_n
```

For each step, when you:

- apply the definition of a function `f`, `rule` must be the name of that function, `f`
- apply the rule for function application, `rule` must be `fun`
- apply an induction hypothesis, `rule` must be `I.H.`
- simplify an arithmetic expression, `rule` must be `arith`
- select a branch in a match expression, `rule` must be `match`
- expand a `let` definition, `rule` must be `let`
- apply a lemma that you have already proven in the exercise, `rule` must be the name you gave to the lemma

In each step, apply only a single rule. Write each step on its own line.

Submissions that do not use the template or do not follow this format for equational proofs may receive 0 points.

▶ Assessment Guidelines

Example Solution

```
[Generalization]
Generalized statement (if necessary) (*): a + length xs =
count a (map (fun z -> true) xs)


[Base Case]
Statement being proven in base case: a + length [] =
count a (map (fun z -> true) [])
Proof of base case:

a + length []
(length) = a + match [] with [] -> 0 | l :: lst -> 1 +
length lst
(match) = a + 0
(arith) = a

count a (map (fun z -> true) [])
(map) =
  count a
    (match [] with
    | [] -> []
    | m :: mst -> (fun z -> true) m :: map (fun z ->
true) mst)
(match) = count a []
(count) =
  match [] with
  | [] -> a
  | c :: cst ->
      count
        (match c with true -> a + 1 | false -> a)
        cst
(match) = a


[Inductive Step]
Induction hypothesis (or hypotheses): a + length xs =
count a (map (fun z -> true) xs)
Statement being proven in inductive step: a + length (x
:: xs) = count a (map (fun z -> true) (x :: xs))
Proof of inductive step:

a + length (x :: xs)
(length) = a + (match x :: xs with [] -> 0 | l :: lst ->
1 + length lst)
(match) = a + (1 + length xs)

count a (map (fun z -> true) (x :: xs))
(map) =
  count a
    (match x :: xs with
    | [] -> []
    | m :: mst -> (fun z -> true) m :: map (fun z ->
true) mst)
(match) =
  count a ((fun z -> true) x :: map (fun z -> true) xs)
(fun) =
  count a (true :: map (fun z -> true) xs)
(count) =
  match true :: map (fun z -> true) xs with
  | [] -> a
  | c :: cst ->
      count
        (match c with true -> a + 1 | false -> a)
        cst
(match) =
  count
    (match true with true -> a + 1 | false -> a)
    (map (fun z -> true) xs)
(match) = count (a + 1) (map (fun z -> true) xs)
(IH) = (a + 1) + length xs
(arith) = a + (1 + length xs)


[Instantiation]
Instantiation of generalization (if necessary):
length xs
(arith) = 0 + length xs
(*) = count 0 (map (fun z -> true) xs)

QED
```

#4 Tail Recursion [0 / 18 Points]

You didn't submit any solution for this exercise.

#5 Modules and Functors [12 / 28 Points] 42.9%

Your Submission

The submission is linked to commit bede12741f4

Assessment

Wrong (32)

Test Case · feedback failed

```
> (See more) Total: 12P
 ListMonoid:
3P   (max 3P)
  type t:
1P   PASS
  zero:
1P   PASS
  plus:
1P   PASS
 FunctionMonoid:
4P   (max 4P)
  type t:
1P   PASS
  zero:
1P   PASS
  plus:   [...] ...
```

Test Case · points:26 failed

Test Case · points:27 failed

Test Case · points:24 failed

Test Case · points:25 failed

Test Case · 0:core:5:PairMonoid:2:plus:1:prop:0:all failed

```
where P = PairMonoid
(ListMonoid):
P.plus was not defined

File "probe-
data/PAIR_PLUS_EX/probe.ml"
, line 3, characters 8-14:
3 | let _ = M.plus
            ^^^^^^
Error: Unbound value M.plus
```

Problem Statement

Tasks:

Modules and Functors: Modular Monoids

In this exercise, we will implement monoids as modules in OCaml. A monoid groups together a type `t`, an associative binary operation `plus`, and an identity element `zero`. That means that:

- `plus (plus x y) z` is equal to `plus x (plus y z)`
- `plus zero x` and `plus x zero` both return `x`

We will represent monoids as modules with the following signature:

```
module type Monoid = sig
  type 'a t

  val zero : 'a t
  val plus : 'a t -> 'a t -> 'a t
end
```

0. **Grading** No results
   Check the results of this task to see how your submission was graded.

1. **ListMonoid** No results
   Lists form a monoid. The identity element (`zero`) is the empty list, and the binary operation (`plus`) is list concatenation.

   Implement the module `ListMonoid`, which conforms to the signature `Monoid` where the type `'a t` is `'a list`.

2. **FunctionMonoid** No results
   Functions of type `'a -> 'a` form a monoid. The identity element is the identity function (i.e. the function that always returns its input). The binary operation is function composition, i.e. `plus f g` is a function that returns `f (g x)` given an input `x`.

   Implement the module `FunctionMonoid`, which conforms to the signature `Monoid` where the type `'a t` is `'a -> 'a`.

3. **Operations on Monoids**

   Further operations can be implemented based on an existing implementation of a monoid. Implement the functor `MonoidOperations`, conforming to the following signature:

   ```
   module type MonoidOperations = functor (M : Monoid) -> sig
     val fold : 'a M.t list -> 'a M.t
     val mul : int -> 'a M.t -> 'a M.t
   end
   ```

   You may not assume anything about the monoid `M`, except that `plus` is associative and `zero` is an identity element, as described above.

   1. **fold** No results
      Given a list `xs`, the function `fold` combines the elements in order using `M.plus`. Thus, for a list `[`x_1`;`x_2`; ... ;`x_{n-1}`;`x_n`]`, the result is equal to `M.plus` x_1 `(M.plus` x_2 `(... (M.plus` x_{n-1} x_n`)))`. For empty lists, it returns

Test Case · 0:core:5:PairMonoid:1:zero:0:zero failed

```
where P = PairMonoid
(ListMonoid):
P.zero was not defined

File "probe-
data/PAIR_ZERO_EX/probe.ml"
, line 3, characters 8-14:
3 | let _ = M.zero
            ^^^^^^
Error: Unbound value M.zero
```

Test Case · 0:core:6:PairListFlippedListMonoid:2:plus:1:prop:0:all failed

```
test `all` failed on ≥ 1
cases:
PairListFlippedListMonoid.p
lus ([], []) ([], [""])
(after 6 shrink steps)
Expected:
([], [""])
But got:
([], [])
```

Test Case · 0:core:5:PairMonoid:2:plus:0:fixed:0:examples failed

```
where P = PairMonoid
(ListMonoid):
P.plus was not defined

File "probe-
data/PAIR_PLUS_EX/probe.ml"
, line 3, characters 8-14:
3 | let _ = M.plus
            ^^^^^^
Error: Unbound value M.plus
```

Test Case · 0:core:2:MonoidOperations:1:mul:0:fixed:0:examples failed

```
where O = MonoidOperations
(ListMonoid):
O.mul was not defined

File "probe-
data/OPS_MUL_EX/probe.ml",
line 2, characters 8-13:
2 | let _ = M.mul
            ^^^^^
Error: Unbound value M.mul
```

Test Case · points:22 failed

Test Case · points:23 failed

Test Case · points:20 failed

`M.zero`.

2. **mul** No results

   Given a non-negative integer `n` and a value `x` from `M`, the function `mul` starts with `M.zero` and then adds `x` to it `n` times. When `n` is `0`, it returns `M.zero`.

   For example, the result of `mul 3 x` would be equal to `M.add x (M.add x (M.add x M.zero))`.

4. **FlipMonoid** No results

   Given an existing monoid, a new monoid may be formed by swapping the order of the arguments to `plus`.

   Implement `FlipMonoid`. The functor `FlipMonoid` takes a module `M` as an argument, which conforms to the `Monoid` signature. It returns a module which conforms to the signature `Monoid`, where the type `'a t` is `'a M.t`.

5. **OptionMonoid** No results

   Given an existing monoid, a monoid can be defined on optional values from the `option` datatype. The identity element is `None`.

   The `plus` operation is defined as follows:

   - `plus (Some x) (Some y)` returns `Some (Base.plus x y)`, where `Base` is the existing monoid
   - `plus (Some x) None` and `plus None (Some x)` both return `Some x`
   - `plus None None` returns `None`

   Implement `OptionMonoid`. The functor `OptionMonoid` takes a module `M` as an argument, which conforms to the `Monoid` signature. It returns a module which conforms to the signature `Monoid`, where the type `'a t` is `'a M.t option`.

6. **PairMonoid** No results

   Given two existing monoids, a new monoid may be defined over pairs. Each element is a pair of a value from the first monoid and a value from the second monoid. The identity element is the pair consisting of the identity element from the first monoid and the identity element of the second monoid. The binary operation is also defined by applying the first binary operation to the first element from each pair, and the second binary operation to the second elements.

   Implement `PairMonoid`. The functor `PairMonoid` takes two modules, `L` and `R`, as arguments, each conforming to the `Monoid` signature. It returns a module which conforms to the signature `Monoid`, where the type `'a t` is `('a L.t * 'a R.t)`.

7. **PairListFlippedListMonoid** No results

   Define the module `PairListFlippedListMonoid`, a `Monoid` where the type `'a t` is `('a list * 'a list)`. The monoid is a pair monoid: the first monoid in the pair is the list monoid, and the second is also the list monoid, but with the `plus` operation flipped.

   For your definition of `PairListFlippedListMonoid`, you may assume all other modules and functors from previous parts of the exercise are correctly defined.

Test Case · points:21 failed

Test Case · 0:core:3:FlipMonoid:2:plus:1:prop:0:all failed

```
where F = FlipMonoid
(ListMonoid):
F.plus was not defined

File "probe-
data/FLIP_PLUS_EX/probe.ml"
, line 2, characters 8-14:
2 | let _ = M.plus
            ^^^^^^
Error: Unbound value M.plus
```

Test Case · 0:core:3:FlipMonoid:1:zero:0:zero failed

```
where F = FlipMonoid
(ListMonoid):
F.zero was not defined

File "probe-
data/FLIP_ZERO_EX/probe.ml"
, line 2, characters 8-14:
2 | let _ = M.zero
            ^^^^^^
Error: Unbound value M.zero
```

Test Case · 0:core:3:FlipMonoid:2:plus:0:fixed:0:examples failed

```
where F = FlipMonoid
(ListMonoid):
F.plus was not defined

File "probe-
data/FLIP_PLUS_EX/probe.ml"
, line 2, characters 8-14:
2 | let _ = M.plus
            ^^^^^^
Error: Unbound value M.plus
```

Test Case · 0:core:2:MonoidOperations:0:fold:1:prop:0:all failed

```
where O = MonoidOperations
(ListMonoid):
O.fold was not defined

File "probe-
data/OPS_FOLD_EX/probe.ml",
line 2, characters 8-14:
2 | let _ = M.fold
            ^^^^^^
Error: Unbound value M.fold
```

Test Case · points:19 failed

Test Case · points:17 failed

Test Case · points:18 failed

Test Case · points:15 failed

Test Case · points:16 failed

Test Case · points:13 failed

Test Case · points:14 failed

Test Case · 0:core:4:OptionMonoid:2:plus:1:prop:0:all failed

```
where O = OptionMonoid
(ListMonoid):
O.plus was not defined

File "probe-
data/OPTION_PLUS_EX/probe.m
l", line 2, characters 8-
14:
2 | let _ = M.plus
            ^^^^^^
Error: Unbound value M.plus
```

Test Case · 0:core:4:OptionMonoid:1:zero:0:zero failed

```
where O = OptionMonoid
(ListMonoid):
O.zero was not defined

File "probe-
data/OPTION_ZERO_EX/probe.m
l", line 2, characters 8-
14:
2 | let _ = M.zero
            ^^^^^^
Error: Unbound value M.zero
```

Test Case · 0:core:4:OptionMonoid:2:plus:0:fixed:0:examples failed

```
where O = OptionMonoid
(ListMonoid):
O.plus was not defined

File "probe-
data/OPTION_PLUS_EX/probe.m
l", line 2, characters 8-
14:
2 | let _ = M.plus
            ^^^^^^
Error: Unbound value M.plus
```

Test Case · 0:core:6:PairListFlippedListMonoid:2:plus:0:fixed:0:examples failed

```
test `examples` failed on ≥
1 cases:
```

```
PairListFlippedListMonoid.p
lus (["a"], ["x"; "y"])
(["b"; "c"], ["z"])
Expected:
(["a"; "b"; "c"], ["z";
"x"; "y"])
But got:
([], [])
```

Test Case · 0:core:2:MonoidOperations:0:fold:0:fixed:0:examples failed

```
where O = MonoidOperations
(ListMonoid):
O.fold was not defined

File "probe-
data/OPS_FOLD_EX/probe.ml",
line 2, characters 8-14:
2 | let _ = M.fold
            ^^^^^^
Error: Unbound value M.fold
```

Test Case · 0:core:2:MonoidOperations:1:mul:1:prop:0:all failed

```
where O = MonoidOperations
(ListMonoid):
O.mul was not defined

File "probe-
data/OPS_MUL_EX/probe.ml",
line 2, characters 8-13:
2 | let _ = M.mul
            ^^^^^
Error: Unbound value M.mul
```

Test Case · points:12 failed

Correct (27) **12P**

Test Case · 0:core:4:OptionMonoid:0:t passed

Test Case · 0:core:6:PairListFlippedListMonoid:0:t passed

Test Case · 0:core:1:FunctionMonoid:2:plus:0:fixed:0:example passed

Test Case · 0:core:3:FlipMonoid:0:t passed

Test Case · 0:core:0:ListMonoid:1:zero:0:zero passed

Test Case · 0:core:1:FunctionMonoid:1:zero:0:fixed:0:examples

passed

Test Case · 0:core:0:ListMonoid:2:plus:0:fixed:0:examples passed

Test Case · 0:core:1:FunctionMonoid:1:zero:1:prop:0:all passed

Test Case · 0:core:1:FunctionMonoid:0:t passed

Test Case · 0:core:0:ListMonoid:2:plus:1:prop:0:all passed

Test Case · points:0 passed 1P

Test Case · points:1 passed 1P

Test Case · 0:core:5:PairMonoid:0:t passed

Test Case · points:2 passed 1P

Test Case · points:3 passed 1P

Test Case · points:4 passed 1P

Test Case · points:5 passed 1P

Test Case · points:6 passed 1P

Test Case · points:7 passed 1P

Test Case · points:8 passed 1P

Test Case · points:9 passed 1P

Test Case · 0:core:6:PairListFlippedListMonoid:1:zero:0:zero passed

Test Case · 0:core:0:ListMonoid:0:t passed

Test Case · points:11 passed **1P**

Test Case · build passed

Test Case · points:10 passed **1P**

Test Case · 0:core:1:FunctionMonoid:2:plus:1:prop:0:all passed

You can submit one complaint for each manually assessed exercise in this exam.

#6 Recursive Datatypes [0 / 32 Points] 0%

Your Submission

The submission is linked to commit No commit was made

Assessment

Wrong (1)

Test Case · {{ name }} failed

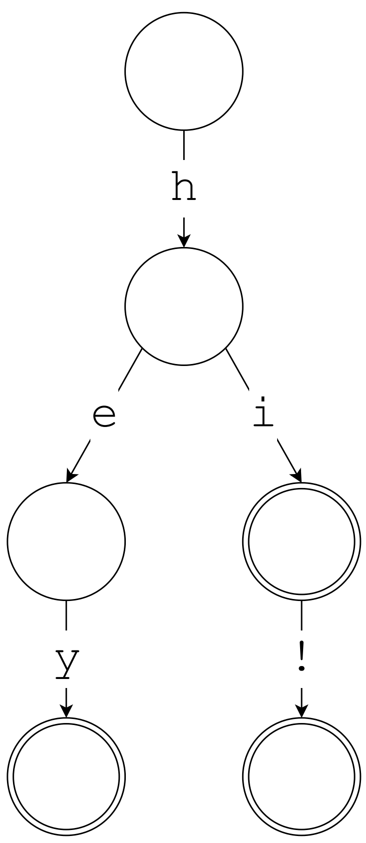
`Empty submission`

Problem Statement

Tries, Tries, Tries

A `trie` is a tree that stores words in a compact way. In OCaml, we implement a `trie` as a type with the constructor `Trie of (bool * (char * trie) list)`. The `bool` specifies whether the sequence of characters given by the path from the root to the current node is a word stored in the `trie`. The `(char * trie) list` stores the outgoing edges in an association list, where the letter of the outgoing edge is associated with its sub-`trie`.

As an example, we fill an empty `trie` with the words `hey`, `hi`, and `hi!`. Below, there is a visualization where the sub-`tries` are labeled and `tries` where the `bool` is set to `true` have a double outline. Additionally, it is shown how the `trie` is represented in OCaml using the defined `trie`-type.




```
Trie
  ( false,
    [
      ( 'h',
        Trie
          ( false,
            [
              ('e', Trie (false, [ ('y', Trie
(true, [])) ]));
              ('i', Trie (true, [ ('!', Trie
(true, [])) ]));
            ] ) );
    ] )
```

1: The Trie-Module

The `Trie`-Module contains the main implementation for your `tries`.
The type `Trie.trie` and the value `Trie.empty` are already given and **should not** be changed.

All words the `Trie`-Module works with are represented as `char list`s so that you can use the known functions from the `List`-Module to access and modify the characters of the words.

Hint: You may use the `List` module for this exercise.

Trie.contains trie word No results

Returns `true` if the passed `trie` contains the passed `word`; otherwise returns `false`

Trie.insert trie word No results

Inserts the passed `word` into the given `trie`.

Trie.remove trie word No results

Removes the passed `word` from the given `trie`.

To save used memory of the `trie`s, subtries that do not store any words (i.e., every `bool` is `false`) should be removed from the returned `trie`. You may assume all passed `tries` already follow this invariant.

2: Shared Tries as a service

The `Trie_db`-Functor wraps the `Trie`-Module and allows shared access to a `Trie` using the Reppy-system discussed in the lecture.

The `Trie_db`-Functor should not implement the `Trie` functionality by itself, but use the methods from the passed `Trie`-module.
This allows you to implement the `Trie_db` without having implemented the `Trie`.

The `Trie_db` already contains the type `t` representing a `channel` similar to the exercises from the lecture.

Trie_db(Trie).create () No results

Creates a new `Trie_db`-server in a new thread.

Trie_db(Trie).insert trie_server word No results

Inserts the `word` into the shared `trie_server`.

Trie_db(Trie).remove trie_server word No results

Removes the `word` from the shared `trie_server`.

Trie_db(Trie).contains trie_server word No results

Returns `true` if the passed `trie` contains the passed `word`; otherwise returns `false`. This should block and return the resulting `bool`.

Examples No results

Examples for the use of the specified `trie`-type, the `Trie`-Module and the `Trie_db`-Functor can be seen with the `example_trie_*`-functions at the end of the file.
The order of the `tries` sub-`trie`-list is not defined, so you may not be able to test equality directly. However, public tests checking these examples are provided.
Additionally, while local testing of your `Trie_db` may require an implemented `Trie`, the (public) tests test your `Trie_db` with a fully functional `Trie`.

Note: The hidden tests not assigned to any task give no points, as they are already covered by other tests.

You can submit one complaint for each manually assessed exercise in this exam.