COMP 302 – Section 01 / 02 – Fall 2017
Midterm Examination, 10 Oct 2017

Brigitte Pientka and Francisco Ferreira

—— Model Solution——

Name:

McGill ID#:

This examination is closed book, but you are allowed a **cheat sheet (one page - single sided, minimum of 12pt font)**. No calculators, laptops, cell phones, or any other device that could conceivably run OCaml are allowed. You have **80 minutes** (start: 18:05pm – 19:25 pm).

Please write your answers on the exam paper. Try to write in the blanks provided. If you cannot, you may write your answer in the scratch space at the end.

There are **100 points** total. This midterm is worth 10% of the course grade.

The questions are **not** ordered by difficulty.

Good luck!

Question 1 [20 points]: Short Answers

For each of the following expressions, say whether OCaml accepts the6cm expression as well-typed or gives a type error. If the expression is well-typed, give its most general type. If the expression is syntactically ill-formed, return parse error. Further, give the result of evaluating the expressions that OCaml returns. If OCaml does not evaluate the expression, mark it with ✗.

i. `fun y -> 3 * y`

Type : `int -> int` Result of evaluation: <**fun**>

ii. `fun (x , y)-> if x then y else 2.0`

Type : `(bool * float)-> float` Result of evaluation: <**fun**>

iii. `let x = 4 in let y = x * 2 in let x = 3 in x + y`

Type : `int` Result of evaluation: `11`

iv. `fun x y -> [ x ; y ]`

Type : `'a -> 'a -> 'a list` Result of evaluation: <**fun**>

v. `fun l -> match l with [] -> true | _ -> false`

Type : `'a list -> bool` Result of evaluation: <**fun**>

Question 2 [25 points]: Let's play cards!

In class we defined cards using `suit` and `rank`, and a `hand` of cards as follows:

```
type suit = Clubs | Spades | Hearts | Diamonds
type rank = Two | Three | Four | Five | Six | Seven | Eight | Nine | Ten |
            Jack | Queen | King | Ace

type card = rank * suit

type hand = Empty | Hand of card * hand
```

We want to sort the cards in our hand. For this purpose, we will use mergesort as a strategy. Furthermore, we do not want to fix how the cards must be ordered. Therefore our `sort` function is parameterized by a function `p:card -> card -> bool` which allows us to compare two cards in a `hand` of cards. It returns a sorted `hand` of cards.

```
let rec sort (p:card -> card -> bool) (h:hand) = match h with
  | Empty -> Empty
  | Hand (c, Empty) -> h
  | _ -> let (h1, h2) = split h in merge p (sort p h1) (sort p h2)
```

The function `merge p h1 h2` which we refer to in the code above allows us to merge two sorted hands `h1` and `h2`. You can assume it is given.

Your task is to implement the function `split: hand -> hand * hand`**.**

The function `split` takes a `hand` of cards and will put cards at odd position into the first hand and cards at even positions into the second hand.

```
# split (Hand ((Queen, Hearts), Hand ((Queen, Diamonds), Hand ((Ace, Hearts), Empty))));;
- : hand * hand =
(Hand ((Queen, Hearts), Hand ((Ace, Hearts), Empty)),
 Hand ((Queen, Diamonds), Empty))

# split Empty;;
- : hand * hand = (Empty, Empty)

# split (Hand((Queen, Hearts), Empty));;
- : hand * hand = (Hand ((Queen, Hearts), Empty), Empty)
```

Model Solution:

```
let rec split h = match h with
  | Empty -> (Empty , Empty)
  | Hand (c , Empty) -> (h , Empty)
  | Hand (c1, Hand (c2, h)) ->
      let h1, h2 = split h in
        (Hand (c1, h1) , Hand (c2, h2))
```

Question 3 [30 points]: Let's have cake!

Let's model a cup cake by its price, its weight, its calories and its ingredients as follows:

```
type price = float
type weight = float
type calories = int
type ingredient = Nuts | Gluten | Soy | Dairy

type cup_cake = Cup_cake of price * weight * calories * ingredient list

 let c1 = Cup_cake (2.5 , 80.3, 250, [Dairy ; Nuts]) ;;
 let c2 = Cup_cake (2.75, 90.5, 275, [Dairy ; Soy]) ;;
 let c3 = Cup_cake (3.05, 100.4, 303, [Nuts ; Dairy ; Gluten]) ;;
 let c4 = Cup_cake (3.25, 120.4, 330, [Gluten; Dairy]) ;;
```

Your task is to implement a function `allergy_free: ingredient list -> cup_cake list -> cup_cake list` . It takes a list of ingredients, `allergens`, and list of cup cakes, `l`, as input. It returns only those cup cakes that do not contain `allergens`. Note that list of ingredients are not ordered.

```
# allergy_free [Nuts] [c1 ; c2 ; c3 ; c4];;
- : cup_cake list =
[Cup_cake (2.75, 90.5, 275, [Dairy; Soy]);
 Cup_cake (3.25, 120.4, 330, [Gluten; Dairy])]
# allergy_free [Nuts ; Gluten] [c1 ; c2 ; c3 ; c4];;
- : cup_cake list = [Cup_cake (2.75, 90.5, 275, [Dairy; Soy])]
```

Use the following higher-order function to give a succinct answer:

- `List.filter: ('a -> bool)-> 'a list -> 'a list`
- `List.exists: ('a -> bool)-> 'a list -> bool`
- `List.for_all: ('a -> bool)-> 'a list -> bool`

If you do not know how to use these functions, write a program without them! Use the spare page at the end, if you need more space. A correct implementation without higher-order functions will receive a maximum of 20 points.

Model Solution:

```
let allergy_free allergens cupcakes =
List.filter (function Cup_cake (p,w,c,il) ->
              List.for_all (fun a -> not (List.exists (fun x -> x = a) il)) allergens
            ) cupcakes
```

Question 4 [25 points]: Heartache for Everyone (Inductive Proof)

Below we give two functions which both count the number of cards in a `hand`.

```
let rec count h = match h with
  | Empty -> 0
  | Hand (c, h) -> count h + 1

let rec count_tr h acc = match h with
  | Empty -> acc
  | Hand (c, h) -> count_tr h (1+acc)
```

Prove that for all `h:hand`, `count h = count_tr h 0`. As we cannot prove this statement directly, we prove the following more general statement.

Theorem 1. *For all hands* `h` *and accumulators* `acc`, `count h + acc = count_tr h acc`.

Proof. By structural induction on `h`.

- **Base Case:** `h = Empty`.

  `count Empty + acc` $\Longrightarrow^*$ `0 + acc` $\Longrightarrow^*$ `acc`

  `count_tr Empty acc` $\Longrightarrow^*$ `acc`

  Hence we are done.

- **Step Case:** `h = Hand(c,h')`.

  State your induction hypothesis:

  For all `acc'`. `count h' + acc' = count_tr h' acc'`

  Proof: To show: `count (Hand(c,h')) + acc = count_tr (Hand(c,h'))acc'`

  RHS: `count (Hand(c,h'))+ acc` $\Longrightarrow^*$ `(count h' + 1) + acc` by program `count`

  $\Longrightarrow^*$ `count h' + (1 + acc)` by assoc. of `+`

  $\Longrightarrow^*$ `count_tr h' (1 + acc)` by IH choosing `1 + acc` for `acc'`

  LHS: `count_tr (Hand(c,h'))acc` $\Longrightarrow^*$ `count_tr h' (1 + acc)` by program of `count_tr`

  Therefore, we are done.

□

Continue on the next page

Theorem 2 (Main Result). *For all hands* `h`, `count h = count_tr h 0`.

Proof.

By Theorem 1, choosing `0` for `acc`, we have: `count h + 0 = count_tr h 0`.

Since also `count h + 0` $\Longrightarrow^*$ `count h`, we have `count h = count_tr h 0`.

□

Scratch space