First Exam
CS 1103 Computer Science 1 Honors

Fall 2019

Friday September 27, 2019
Instructor Muller

KEY

Before reading further, please arrange to have an empty seat on either side of you. Now that you are seated, please write your name **on the back** of this exam.

This is a closed-notes and closed-book 50-minute exam. Computers, calculators, and books are prohibited. You may use print-outs of the OCaml and Standard Library cheat-sheets. Work on this exam should stop in 50 minutes.

- Partial credit will be given so be sure to show your work.
- Feel free to write helper functions if you need them.
- **Please write neatly.**

Problem	Points	Out Of
Part 1		4
Part 2.1		4
Part 2.2		4
Part 2.3		4
Part 2.4		4
Part 2.5		4
Total		20

Table 1: Do only 4 of 5 problems in Part 2.

Part 1 (4 Points): Short Answer

1. (1 Point) Is the following well-formed? If not, what's wrong? If so, indicate the type and show the step-by-step simplification.

```
let ns = [2 * 3; [4]] in (ns, ns)
```

Answer: This is ill-formed, items in a list must have the same type.

2. (1 Point) Let's say we have the following type definition `type t = {a : int * int; b : bool}`. Is the following well-formed? If not, what's wrong? If so, indicate the type and show the step-by-step simplification.

```
let pair = (2 * 3, 4) in {b = not(true); a = pair}
```

Answer: This is well-formed and is of type `t`.

```
let pair = (2 * 3, 4) in {b = not(true); a = pair} =>
let pair = (6, 4) in {b = not(true); a = pair} =>
{b = not(true); a = (6, 4)} =>
{b = false; a = (6, 4)}
```

3. (1 Point) Let's say we have the following definition `let double n = n * 2`. Is the following well-formed? If not, what's wrong? If so, indicate the type and show the step-by-step simplification.

```
match double(4) != 16 with | true -> 1024 | false -> false
```

Answer: This is ill-formed. The branches of a match must return values of the same type.

4. (1 Point) Is the following well-formed? If not, what's wrong? If so, indicate the type and show the step-by-step simplification.

```
let x = (let y = not(false) in false || y) in not(x && x)
```

Answer: This is well-formed and of type `bool`.

```
let x = (let y = not(false) in false || y) in not(x && x) =>
let x = (let y = true in false || y) in not(x && x) =>
let x = (false || true) in not(x && x) =>
let x = true in not(x && x) =>
not(true && true) =>
not(true) =>
false
```

Part 2 (16 Points): Writing Functions

Do **exactly four** of the following five 4-point problems. Please circle the numbers of the problems you wish to be graded.

1. (4 Points) Write a function `allTheSame : 'a list -> bool` such that a call `(allTheSame xs)` returns `true` if all of the elements of `xs` are the same, as judged by the `=` operator. For example, `allTheSame` should return `true` for the following inputs: `[]`, `["Alice"]` and `["Alice"; "Alice"]`. It should return `false` for the following input: `["Alice"; "Bob"]`. How much work does your solution do?

   Answer:

   ```
   let rec allTheSame xs =
     match xs with
     | [] | [only] -> true
     | x :: y :: ys -> (x = y) && allTheSame (y :: ys)
   ```

   This function is linear in the length of `xs`.

2. (4 Points) Write a function `powersOfTwo : int -> int list` such that a call `(powersOfTwo n)` returns a list of the powers of 2 running from 2^0 up to 2^{n-1}. For example, the call `(powersOfTwo 6)` should return the list `[1; 2; 4; 8; 16; 32]`. You may assume that n is non-negative. How much work does your solution do?

Answer:

```
let intPow2 n = int_of_float (2.0 ** (float n))

let powersOfTwo n =
  let rec loop powers =
    match powers with
    | [] -> []
    | power :: powers -> (intPow2 power) :: (loop powers)
  in
  loop (Code.range n)
```

This function requires N steps.

3. (4 Points) Write a function `allIn : 'a list -> 'a list -> bool` such that a call `(allIn xs ys)` returns `true` if every element of `xs` is an element of `ys`. Otherwise, `allIn` should return `false`. How much work does your solution do?

Answer:

```
let rec allIn xs ys =
  match xs with
  | [] -> true
  | x :: xs -> (List.mem x ys) && (allIn xs ys)
```

This function calls `List.mem` for each of the N elements of `xs`. The `List.mem` function has one call for each of the M elements of `ys`. So the work is $M \cdot N$.

4. (4 Points) Write a function `digits : int -> int list` such that a call `(digits n)` returns a list of the digits in `n`. For example, the call `(digits 548)` should return the list `[5; 4; 8]`. How much work does your solution do?

Answer:

```
let digits n =
  let rec loop n answer =
    match n = 0 with
    | true  -> answer
    | false -> loop (n / 10) (n mod 10 :: answer)
  in
  loop n []
```

This function requires $\log_{10} N$ steps.

5. (4 Points) A *dictionary* is a list of *key*/*value* pairs `(key, value)`. For example, a key might be your friend's name and the associated value might be their contact information. As another example, the list of pairs in

```
let dictionary = [(10, "Alice"); (14, "Carmen"); (12, "Joe")];;
```

is a 3-element dictionary with integer keys and string values. The value of key `14` is the string `"Carmen"`.

Write a function `find : 'a -> 'a * 'b list -> 'b` such that a call `(find key dictionary)` returns the value of `key` in `dictionary` if it exists. For example, with `dictionary` defined as above, the call `(find 12 dictionary)` should return the string `"Joe"`. If the `key` doesn't occur in `dictionary` then `find` should fail using `failwith`. How much work does your solution do?

Answer:

```
let rec find key dictionary =
  match dictionary with
  | [] -> failwith "The key is not in the list"
  | (key1, value) :: dictionary ->
    (match key = key1 with
     | true  -> value
     | false -> find key dictionary)
```

This function is linear in the length of `dictionary`.