MP 2 – Pattern Matching and Recursion

CS 421 – Fall 2011

Revision 1.1

Assigned August 30, 2011
Due September 6, 2011 23:59
Extension 48 hours (20% penalty)

1 Change Log

1.1 Fix typo (Problem 5) and remove needless instruction (Problem 6).

1.0 Initial Release.

2 Objectives and Background

The purpose of this MP is to help the student master:

- pattern matching
- higher-order functions
- recursion

3 Instructions

The problems below have sample executions that suggest how to write answers. You have to use the same function name, but the name of the parameters that follow the function name need not be duplicated. That is, you are free to choose different names for the arguments to the functions from the ones given in the example execution. We will sometimes use `let rec` to begin the definition of a function that is allowed to use `rec`. You are not required to start your code with `let rec`, and you may use `let rec` when we do not.

For all these problems, you are allowed to write your own auxiliary functions, either internally to the function being defined or externally as separate functions. In fact, you will find it helpful to do so on several problems. In this assignment, **you may not use any library functions other than @** except where explicitly noted.

4 Problems

1. (3 pts) Rational numbers can be represented as a pair of integers, the numerator and a nonzero denominator. Equality in this representation is defined as $\frac{a}{b} = \frac{c}{d} \iff ad = bc$. Write `rat_eq : (int * int) * (int * int) -> bool` that takes two rational numbers as a pair of pairs of integers, and decides their equality. We will not test the behavior of your function for invalid inputs (e.g., $\frac{0}{0}$).

```
# let rat_eq r = ...;;
val rat_eq : (int * int) * (int * int) -> bool = <fun>
# rat_eq ((1, 2), (3, 4));;
- : bool = false
```

2. (3 pts) The Ackermann function, $A : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is defined as:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \wedge n = 0 \\ A(m-1, A(m, n-1)) & \text{otherwise} \end{cases}$$

Write a function `ackermann : int * int -> int` that implements $A(m,n)$. We will not test your function on invalid inputs (i.e., when $m < 0$ or $n < 0$).

```
# let rec ackermann (m,n) = ...;;
val ackermann : int * int -> int = <fun>
# ackermann (3,4);;
- : int = 125
```

3. (3 pts) Write a function `list_ex : ('a -> bool) -> 'a list -> bool` that takes a predicate and a list and decides if at least one element of the list satisfies the predicate.

```
# let rec list_ex p xs = ...;;
val list_ex : ('a -> bool) -> 'a list -> bool = <fun>
# list_ex (fun x -> x < 0) [1;-1;0;4;-2;5];;
- : bool = true
```

4. (3 pts) Write a function `elem : 'a -> 'a list -> bool` that decides if the first argument is an element of the given list.

```
# let rec elem x xs = ...
val elem : 'a -> 'a list -> bool = <fun>
# elem 5 [1;34;42;6];;
- : bool = false
```

5. (3 pts) Write `to_float_list : int list -> float list` that takes a list of integers and converts it to a list of floats. You may use `float_of_int : int -> float` (a built-in function in OCaml) to convert an `int` into the corresponding `float`.

```
# let rec to_float_list xs = ...;
val to_float_list : int list -> float list = <fun>
# to_float_list [1;2;3];;
- : float list = [1.; 2.; 3.]
```

6. (3 pts) Write `sum : float list -> float` that returns the sum of all elements in the list, or 0.0 if the list is empty.

```
# let rec sum fs = ...;;
val sum : float list -> float = <fun>
# sum [0.; 2.; 5.];;
- : float = 7.
```

7. (5 pts) Write a function `zip : 'a list -> 'b list -> ('a * 'b) list` that takes two lists and returns a list of pairs where the pair in position i is the ith elements of the first and second list. If one list is longer than the other, ignore the extra elements while zipping.

```
# let rec zip xs ys = ...;;
val zip : 'a list -> 'b list -> ('a * 'b) list = <fun>
# zip [1;2] ["hello"; "world"];;
- : (int * string) list = [(1,"hello"); (2,"world")]
```

8. (5 pts) Write a function `split : ('a * 'b) list -> 'a list * 'b list` that takes a list of pairs and outputs a pair of lists with the elements of first list the first element of each pair and the elements of the second list those of the second element of each pair.

```
# let rec split xs = ...;;
val split : ('a * 'b) list -> 'a list * 'b list = <fun>
# split [(1,"hello"); (2,"world")];;
- : int list * string list = ([1; 2], ["hello"; "world"])
```

9. (6 pts) Write a function `cross : 'a list -> 'b list -> ('a * 'b) list` that takes two lists and forms a list consisting of all the pairs of the form (a_i, b_j) where a_i is an element of the first list and b_j is an element of the second list. Note this allows for duplicate pairs in the result. The elements of the resulting list should be ordered as follows (a_i is the ith element of the first list and b_i is the ith element of the second list)

- For all x and y, (a_i, b_x) is before (a_j, b_y) if $i < j$
- (a_i, b_j) is before (a_i, b_k) if $j < k$

```
# let cross xs ys = ...
val cross : 'a list -> 'b list -> ('a * 'b) list = <fun>
# cross [1;2] ["hello"; "world"];;
- : (int * string) list = [(1, "hello"); (1, "world"); (2, "hello"); (2, "world")]
```

10. (7 pts) Suppose that we have a record of discrete events, ordered 1 through n, occurring in a given sample space. We may record the frequency distribution of these events in that space by a list $[f_1, \ldots, fn]$ where f_i is the number of occurrences of the i^{th} event in the sample space. Then, the probability p_i of the i^{th} event occurring in the sample space is given by

$$p_i = \frac{f_i}{\Sigma_i f_i}$$

Write a function `freq_to_prob : int list -> float list` that takes a list of frequencies of events and returns a list of the corresponding probabilities for those events. You may assume that the list of frequencies contains only non-negative values, and the there is at least one positive value in it. You can convert an integer into a float using the built-in ocaml function `float_of_int`.

```
# let rec freq_to_prob fs = ...;;
val freq_to_prob fs : int list -> float list = <fun>
# freq_to_prob [3;7;9];;
-: float list = [0.157894736842105254; 0.368421052631578927; 0.473684210526315763]
```

11. (5 pts) For a discrete random variable X that takes on values x_i, with each x_i occurring with probability p_i, the expected value of X is $\Sigma_i p_i x_i$. Assume you are given a list of pairs (x_i, p_i) of a float, being the i^{th} value a random variable X may take on, together with a float representing its probability of occurrence. Write a function `expected : (float * float) list -> float` that calculates the expected value of a discrete random variable, the probability distribution of whose is described by the input list. You should return `0.0` for the empty list.

```
# let rec expected xs = ...;;
val expected : (float * float) list -> float = <fun>
# expected [(1.,0.5);(2.,0.5)];;
- : float = 1.5
```

12. (7 pts) Recall that a Deterministic Finite Automata (DFA) is a tuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q is a set of states,
- Σ is an input alphabet,
- $\delta : Q \times \Sigma \to Q$ is a transition function,
- $q_0 \in Q$ is an initial state,
- $F \subseteq Q$ is a set of accepting states.

When running a DFA we extend δ to $\hat{\delta} : Q \times \Sigma^* \to Q$ by repeatedly applying δ starting at the initial state and first "letter" of the input. Input i is accepted if $\hat{\delta}(q_0, i) \in F$.

In this problem we will use lists to represent sets. While lists allow for duplicate elements none of the test inputs will contain duplicates. Write a function `accept : 'a list * 'b list * ('a -> 'b -> 'a) * 'a * 'a list -> 'b list -> bool` that takes a DFA as a tuple and an input word and decides if the DFA accepts the word. You may use `List.fold_left` in this problem. Note OCaml's type inference may infer a more general type for your solution than the type listed above, do not panic.

```
# let accept (qs,ss,d,q0,fs) i = ...
val accept : 'a * 'b * ('c -> 'd -> 'c) * 'c * 'c list -> 'd list -> bool = <fun>
# accept ([0; 1; 2],["a"; "b"], (fun q _ -> (q + 1) mod 3), 0, [2])
         ["a"; "a"; "a"];;
- : bool = false
```

4.1 Extra Credit

13. (5 pts) Write a function `valid_dfa : 'a list * 'b list * ('a -> 'b -> 'a) * 'a * 'a list -> bool` that checks that a DFA represented as above satisfies the conditions:

- $\forall q \in Q, s \in \Sigma. \delta(q, s) \in Q$.
- $q_0 \in Q$
- $F \subseteq Q$

You may use `List.fold_left` in this problem. Note OCaml's type inference may infer a more general type for your solution, do not panic.

```
# let rec valid_dfa (qs,ss,d,q0,fs) = ...
val valid_dfa : 'a list * 'b list * ('a -> 'b -> 'a) * 'a * 'a list -> bool = <fun>
# valid_dfa ([0; 1; 2],["a"; "b"], (fun q _ -> (q + 1) mod 3), 0, [3]);;
- : bool = false
```