Data Structures and Functional Programming	**Problem Set 1**
CS 3110, Spring 2014	**Thursday, February 6**
Version: 1	**Last Modified: January 31, 2014**

Overview

This assignment will take you through a sequence of small problems, culminating in a final program that simulates a rock-paper-scissors tournament.

Objectives

- Gain familiarity with basic OCaml features such as lists, tuples, functions, pattern matching, and data types.
- Practice writing programs in the functional style using immutable data, recursion, and higher-order functions.
- Introduce the basic features of the OCaml type system.
- Illustrate the impact of code style on readability, correctness, and maintainability.

Recommended reading

The following supplementary materials may be helpful in completing this assignment:

- Lectures 1 2 3
- Recitations 1 2 3
- The CS 3110 style guide
- The OCaml tutorial
- Real World OCaml, Chapters 1-3

What to turn in

Exercises marked [code] should be placed in `ps1.ml` and will be graded automatically. Exercises marked [written] should be placed in `written.txt` or `written.pdf` and will be graded by hand.

Warm-up

Exercise 1:

[written] Identify the types and values of the following expressions. If the expressions are not well typed, briefly explain why not.

> **Note:** Although the toplevel will give you the answers to these questions, we recommend that you try them on your own before checking them against the toplevel. Figuring out types is a good skill to have for reading OCaml code and for passing 3110 exams.

(a) `3 + 5`

(b) `("zardoz", 3+5)`

(c) `["zardoz"; 3+5]`

(d) `[List.hd] :: []`

(e) `fun x y -> if x then y else x`

(f) `fun a (b, c) -> a c (b c)`

Exercise 2:

[written] Give expressions having the following types.

(a) `int -> int -> int`

(b) `(int -> int) -> int`

(c) `int -> (int -> int)`

(d) `'a -> 'a`

(e) `'a -> 'b -> 'a`

(f) `('a * 'b -> 'c) -> ('a -> 'b -> 'c)`

Code Style

Exercise 3:

Rock-paper-scissors is a simple two-player game in which players independently choose one of three symbols (rock, paper, or scissors), and reveal their choice simultaneously. The winner of each round is selected using the following ordering: rock beats scissors, scissors beat paper, and paper beats rock. When the two players choose the same symbol, the round ends in a draw.

The following function simulates a single round of a rock-paper-scissors game, but is written with poor style.

```
(**
 * [rps_round a b] accepts two symbols, [a] and [b], and determines the
 * winner.  "rock" beats "scissors", "scissors" beats "paper", and "paper"
 * beats "rock"
 *
 * @param a b One of the choices "rock", "paper" or "scissors"
 * @return One of { -1, 0, 1, 4 }. -1 means [a] won, 1 means [b] won,
 *   0 indicates a tie.  4 indicates an invalid input.
 *)
let rps_round (a : string) (b : string) : int =
  if a = "rock" then (
    if b = "rock" then 0
    else if b = "paper" then 1
    else if b = "scissors" then -1
    else 4
  )
  else if a = "paper" then (
    if b = "rock" then -1
    else if b = "paper" then 0
    else if b = "scissors" then 1
    else 4
  )
  else if a = "scissors" then (
    if b = "rock" then 1
    else if b = "paper" then -1
    else if b = "scissors" then 0
    else 4
  )
  else 4
```

(a) [code] Modify `rps_round` using the following type definitions,

```
type move   = Rock | Paper | Scissors
type result = AWin | BWin  | Draw
```

so that `a` and `b` have type `move`, and the function returns a value of type `result`. The resulting function `rps_round_enum`.

[written] Briefly explain (in one or two sentences) why this function is less error-prone than the version using `string` and `int`.

(b) [code] Write a function `rps_round_nested_match` replacing each `if-then-else` expression with a `match` statement.

[written] Explain in one or two sentences why `rps_round_nested_match` is less error-prone than `rps_round_enum`

(c) [code] Simplify the function further in `rps_round_single_match` by replacing the entire body of the function with a single `match` statement (hint: match on a tuple).

(d) [code] Write a function `rps_round_with_helper` that uses a helper function

```
beats : move -> move -> bool
```

returning `true` if its first parameter wins against its second parameter and `false` otherwise. Use the catch-all pattern `_` in your implementation of `beats`.

[written] Explain one way in which the `rps_round_with_helper` implementation is less error-prone than `rps_round_single_match`. Explain one way in which `rps_round_with_helper` is *more* error-prone than `rps_round_single_match` (Hint: consider the code changes that would be needed to add a fourth move `Nuke` that beats all of the other `moves`).

OCaml programming

Exercise 4:

(a) [code] Write a function `all_pairs` that takes an `'a list` ℓ as input and returns a list containing all pairs of elements of ℓ. For example:

```
# all_pairs ['a'; 'b'];;
- : (char * char) list = [('a', 'a'); ('a', 'b');
                          ('b', 'a'); ('b', 'b')]

# all_pairs [0; 1; 2];;
- : (int * int) list = [(0, 0); (0, 1); (0, 2); (1, 0);
                        (1, 1); (1, 2); (2, 0); (2, 1); (2, 2)]
```

The order of the output list does not matter.

(b) [code] Use `all_pairs` to implement a function `test_rps_eq` that takes two of your implementations of `rps_round` as arguments, and returns true if they agree on all inputs. For example:

```
# test_rps_eq rps_round_enum rps_round_enum;;
- : bool = true

# test_rps_eq rps_round_enum rps_round_with_helper;;
- : bool = true

# let rps_round_bogus a b = AWin in
test_rps_eq rps_round_enum rps_round_bogus;;
- : bool = false

# let rps_round_bogus a b = AWin in
test_rps_eq rps_round_bogs rps_round_bogus;;
- : bool = true
```

(c) [code] Use `all_pairs` and `test_rps_eq` to implement a function `test_all_rps` that takes a list of `rps_round` implementations as arguments and returns true if all of them are equal on all inputs:

```
# test_all_rps [rps_round_enum; rps_round_single_match;
  rps_round_nested_match; rps_round_with_helper];;
- : bool = true

# let rps_round_bogus a b = AWin in
test_all_rps [rps_round_enum; rps_round_bogus];;
- : bool = false
```

Exercise 5:

A rock-paper-scissors player can be represented as a function that takes in a list of the opponent's previous moves, and returns a move for the next round:

```
type history = move list
type player  = history -> move
```

For example, the player that always chooses `Rock` would be represented as

```
let always_rock : player = fun history -> Rock
```

By convention, we will store the most recent move at the head of the list. For example, in the following game,

round	1	2	3	4	5
A's move	Rock	Paper	Paper	Scissors	?
B's move	Rock	Paper	Paper	Scissors	

the list passed to player A for the fifth round will be `[Scissors; Paper; Paper; Rock]`.

(a) [code] Implement a player `beats_last` that always beats the most recent move that the opponent made. `beats_last` should play Rock on the first move. For example:

```
# beats_last [Rock; Paper; Paper; Scissors];;
- : move = Paper

# beats_last [Paper];;
- : move = Scissors

# beats_last [];;
- : move = Rock
```

(b) [code] Write a function `always_plays` that takes a `move` and returns a player that always plays that move. For example:

```
# let always_paper = always_plays Paper;;
always_paper : player = <fun>

# always_paper [];;
- : move = Paper

# always_paper [Scissors; Scissors; Scissors];;
- : move = Paper
```

(c) [code] Write a function `rps_game` that takes two players and keeps running rounds until one of them wins. It should return `true` if the first wins; false otherwise. Use any of your implementations of `rps_round` from Exercise 1.

Exercise 6:

(a) [code] Write a function

```
pair_filter : ('a -> 'a -> 'a) -> 'a list -> 'a list
```

`pair_filter` takes two inputs: a function `compare` that returns one of two values, and a list `l` of values. The `pair_filter` function should break `l` into adjacent pairs, and apply `compare` to each pair. It should return a list containing the resulting values. For example,

```
pair_filter max [0; 5; 7; 2; 3; -10]
  ↓ [max 0 5; max 7 2; max 3 -10]
  ↓ [5; 7; 3]
```

If the input list has odd length, then the last element should be returned as well. For example,

```
pair_filter min [1; 2; 3; 4; 5]
  ↓ [min 1 2; min 3 4; 5]
  ↓ [1; 3; 5]
```

(b) [code] Write a function

```
tournament : ('a -> 'a -> 'a) -> 'a list -> 'a option
```

`tournament` should repeatedly call `pair_filter` on its input list until only a single element `w` remains. It should then return `Some w`. If the input list is empty, `tournament` should return `None`.

For example,

```
tournament max [1; 4; 3; 2; 7; 9; 0]
  ↓ tournament max [4; 3; 9; 0]
  ↓ tournament max [4; 9]
  ↓ tournament max [9]
  ↓ Some 9
```

Comments

[written] At the end of the file, please include any comments you have about the problem set, or about your implementation. This would be a good place to document any extra Karma problems that you did (see below), to list any problems with your submission that you weren't able to fix, or to give us general feedback about the problem set.

Release files

The accompanying release file `ps1.zip` contains the following files:

- `writeup/` contains this file and also the `.tex` file we used to generate it (in case you are interested)
- `release/ps1.ml` and `written.txt` are templates for you to fill in and submit.
- `ps1.mli` contains the interface and documentation for the functions that you will implement in `ps1.ml`
- `examples.ml` contains all of the examples from this writeup in the form of unit tests.

Karma Suggestions

> **Note:** We encourage you to think about different directions that you can take the problem sets or different parts of OCaml or functional programming that you are curious about.
>
> You may submit any extra work you do in the Karma section of each problem set. We will comment on any extra work that you turn in, but Karma **is completely optional and will not affect your grade in any way**.

Here are some suggestions that may pique your interest:

- Implement a player that randomly chooses what symbol to play.
- Implement a player that asks the user what to play.
- The `game` function will often run forever (e.g. if a deterministic player plays against itself). Write a function that runs a game but stops and returns `Draw` if the game runs for `k` rounds.
- Write a function that takes a (deterministic) player and returns a new player that will beat the first player on the third round.
- Write a best two of three, best three of five, or best k of $(2k-1)$ tournament.
- Use `tournament` and `rps_game` to implement a rock-paper-scissors tournament. Find a way to easily identify the winner.
- Use your code to implement a different game, such as tic-tac-toe.
- Use LaTeXwith the 3110 style files for your written questions.