Functional Programming

# Zippers, Reactivity, GUIs

**Exercise 1.** Introduce operators $-, /$ into the context rewriting "pull out subexpression" example. Remember that they are not commutative.

**Exercise 2.** Add to the *paddle game* example:

1. game restart,
2. score keeping,
3. game quitting (in more-or-less elegant way).

**Exercise 3.** Our numerical integration function roughly corresponds to the rectangle rule. Modify the rule and write a test for the accuracy of:

1. the trapezoidal rule;
2. the Simpson's rule. http://en.wikipedia.org/wiki/Simpson%27s_rule

**Exercise 4.** Explain the recursive behavior of integration:

1. In *paddle game* implemented by stream processing – `Lec10b.ml`, do we look at past velocity to determine current position, at past position to determine current velocity, both, or neither?
2. What is the difference between `integral` and `integral_nice` in `Lec10c.ml`, what happens when we replace the former with the latter in the `pbal` function? How about after rewriting `pbal` into pure style as in the following exercise?

**Exercise 5.** Reimplement the *Froc* based paddle ball example in a pure style: rewrite the `pbal` function to not use `notify_e`.

**Exercise 6.** * Our implementation of flows is a bit heavy. One alternative approach is to use continuations, as in `Scala.React`. OCaml has a continuations library *Delimcc*; for how it can cooperate with *Froc*, see http://ambassadortothecomputers.blogspot.com/2010/08/mixing-monadic-and-direct-style-code.html

**Exercise 7.** Implement `parallel` for flows, retaining coarse-grained implementation and using the event queue from *Froc* somehow (instead of introducing a new job queue).

**Exercise 8.** Add quitting, e.g. via a `'q'` key press, to the *painter* example. Use the `is_cancelled` function.

**Exercise 9.** Our calculator example is not finished. Implement entering decimal fractions: add handling of the `dots` event.

**Exercise 10.** The `Flow` module has reader monad functions that have not been discussed on slides:

```
let local f m = fun emit -> m (fun x -> emit (f x))
let local_opt f m = fun emit ->
  m (fun x -> match f x with None -> () | Some y -> emit y)
val local : ('a -> 'b) -> ('a, 'c) flow -> ('b, 'c) flow
val local_opt : ('a -> 'b option) -> ('a, 'c) flow -> ('b, 'c) flow
```

Implement an example that uses this compositionality-increasing capability.