Programming Languages and Techniques (CIS120)

Lecture 21

March 2nd, 2016

GUI: notifiers

Transition to Java

What is the type of xs ?

```
let r = {contents = 3}
let xs = [(fun () -> r.contents <- 5);
          (fun () -> print_int r.contents)]
```

1. unit -> unit
2. int -> unit
3. (unit -> unit) list
4. (unit -> int) list
5. (int -> unit) list
6. unit -> unit list

What should go in the blank to make the console print "5" ?

```
let rec iter (f:'a -> unit)(xs:'a list):unit=
   begin match xs with
   | [] -> ()
   | h :: t -> (f h ; iter f t)
   end
let r = {contents = 3}
let xs = [(fun () -> r.contents <- 5);
          (fun () -> print_int r.contents)]
;; iter (_____________) xs
```

1. fun () -> print_int 5
2. fun () -> ()
3. fun f -> f ()
4. fun f -> f

How far are you on HW 5?

1. Haven't started yet
2. Working on Tasks 1-4 (layout, drawing)
3. Working on Checkboxes
4. Working on Something Cool
5. I'm done!ç

Event Listeners

How to react to events in a modular way?

Listeners

widget.ml

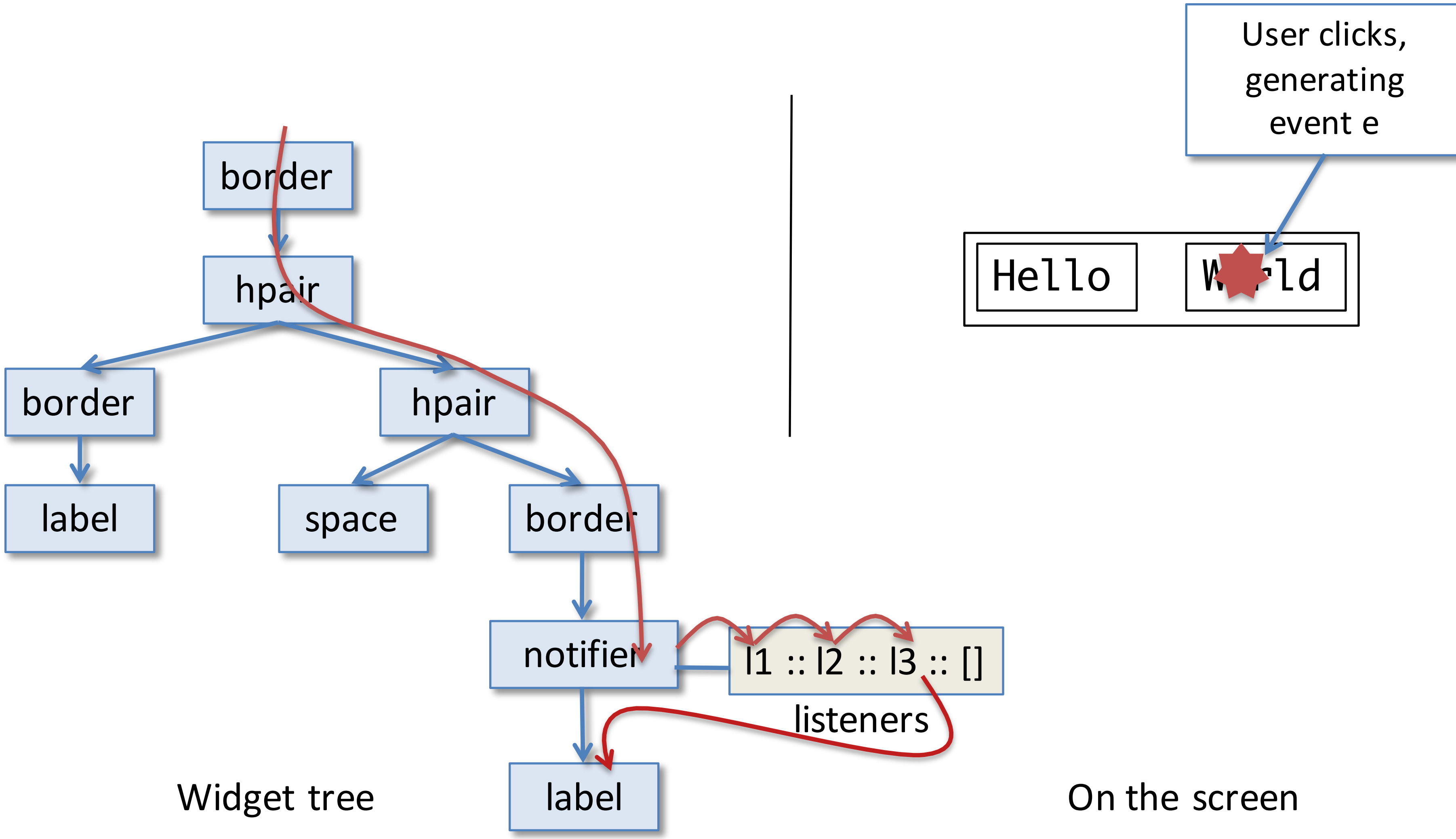
```
type event_listener = Gctx.gctx -> Gctx.event -> unit

(* Performs an action upon receiving a mouse click. *)
let mouseclick_listener (action: unit -> unit)
                        : event_listener =
  fun (g:Gctx.gctx) (e: Gctx.event) ->
    if Gctx.event_type e = Gctx.MouseDown
    then action ()
```

Event Listeners

- Problem: *Widgets may want to react to many different sorts of events*
- Example: Button
  - button click: changes the state of the paint program and button label
  - mouse movement: tooltip? highlight?
  - key press: provide keyboard access to the button functionality?
- These reactions should be independent
  - Each sort of event handled by a different *event listener* (i.e. a first-class function)
  - Reactive widgets may have *several* listeners to handle a triggered event
  - Listeners react in sequence, all have a chance to see the event
- Solution: notifier

Listeners and Notifiers Pictorially



Notifiers

- A *notifier* is a container widget that adds event listeners to a node in the widget hierarchy
  - Note: this way of structuring event listeners is based on Java’s Swing Library design (we use Swing terminology).
- The *event listeners* “eavesdrop” on the events flowing through the node
  - The event listeners are stored in a list
  - They react in order, if one of them handles the event the later ones do not hear it
  - If none of the listeners handle the event, then the event continues to the child widget
- List of event listeners can be updated by using a notifier_controller

Notifiers and Notifier Controllers

widget.ml

```
type notifier_controller =
     { add_listener : event_listener -> unit }

let notifier (w: widget) : widget * notifier_controller =
  let listeners = { contents = [] } in
  { repaint = w.repaint;
    handle =
      (fun (g: Gctx.gctx) (e: Gctx.event) ->
          List.iter (fun h -> h g e) listeners.contents;
          w.handle g e);
    size = w.size
  },
  { add_event_listener =
      fun (newl: event_listener) ->
          listeners.contents <-
                 newl :: listeners.contents
  }
```

Loop through the list of listeners, allowing each one to process the event. Then pass the event to the child.

The notifier_controller allows new listeners to be added to the list.

Buttons (at last!)

widget.ml

```
(* A text button *)
let button (s: string) : widget
                       * label_controller
                       * notifier_controller =
  let (w, lc)  = label s in
  let (w', nc) = notifier w in
    (w', lc, nc)
```

- A button widget is just a label wrapped in a notifier
- Add a mouseclick_listener to the button using the notifier_controller
- (For aesthetic purposes, you can but a border around the button widget.)

Demo: onoff.ml

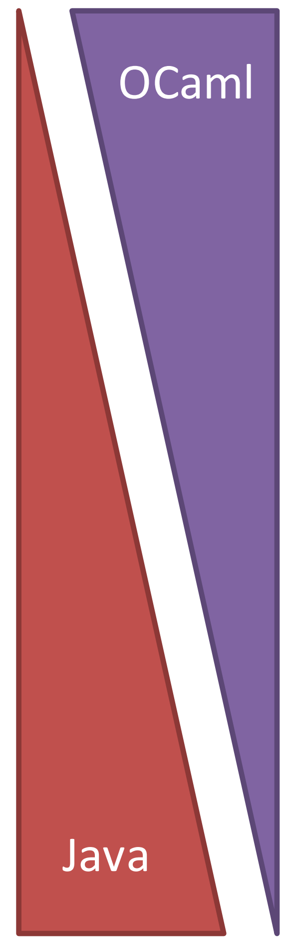
Changing the label on a button click

Goodbye OCaml…

…hello Java!

CIS 120 Overview

- Declarative (Functional) programming
  - *persistent* data structures
  - *recursion* is main control structure
  - frequent use of functions as data
- Imperative programming
  - *mutable* data structures (that can be modified "in place")
  - *iteration* is main control structure
- Object-oriented (and reactive) programming
  - mutable data structures / iteration
  - heavy use of functions (objects) as data
  - pervasive "abstraction by default"



Java and OCaml together

Xavier Leroy, one of the principal designers of OCaml

Guy Steele, one of the principal designers of Java

Moral: Java and OCaml are not so far apart…

Recap: The Functional Style

- Core ideas:
  - immutable (persistent / declarative) data structures
  - recursion (and iteration) over tree structured data
  - functions as data
  - generic types for flexibility (i.e. ‘a list)
  - abstract types to preserve invariants (i.e. BSTs)
  - *simple model of computation (substitution)*

- Good for:
  - elegant descriptions of complex algorithms and/or data
  - small-scale compositional design
  - “symbol processing” programs (compilers, theorem provers, etc.)
  - parallelism, concurrency, and distribution

Functional programming

OCaml

- Immutable lists primitive, tail recursion
- Datatypes and pattern matching for tree structured data
- First-class functions, transform and fold
- Generic types
- Abstract types through module signatures

Java (and C, C++, C#)

- No primitive data structures, no tail recursion
- Trees must be encoded by objects, mutable by default
- No first-class functions.* Must encode first-class computation with objects
- Generic types
- Abstract types through public/private modifiers

*until recently (Java 8)

OCaml vs. Java for FP

```
type 'a tree =
  | Empty
  | Node of ('a tree) * 'a * ('a tree)

let is_empty (t:'a tree) : bool =
  begin match t with
    | Empty -> true
    | Node(_,_,_) -> false
  end

let t : int tree = Node(Empty,3,Empty)
let ans : bool = is_empty t
```

```
interface Tree<A> {
  public boolean isEmpty();
}
class Empty<A> implements Tree<A> {
  public boolean isEmpty() {
     return true;
  }
}
class Node<A> implements Tree<A> {
  private final A v;
  private final Tree<A> lt;
  private final Tree<A> rt;

  Node(Tree<A> lt, A v, Tree<A> rt) {
    this.lt = lt; this.rt = rt; this.v = v;
  }

  public boolean isEmpty() {
    return false;
  }
}

class Program {
  public static void main(String[] args) {
    Tree<Integer> t =
    new Node<Integer>(new Empty<Integer>(),
     3, new Empty<Integer>());
    boolean ans = t.isEmpty();
  }
}
```

More FP

- Type inference
- Modules and support for large scale programming
- Objects (real, but different)
- Many other extensions
- Growing ecosystem
- Real World OCaml, OPAM

Most similar to OCaml,
Shares libraries with C#

Swift
iOS programming

Haskell (CIS 552)
Purity and laziness

Scala
Java / OCaml hybrid

Recap: Imperative programming

- Core ideas:
  - computation as change of state over time
  - distinction between primitive and reference values
  - aliasing
  - linked data-structures and iteration control structure
  - generic types for flexibility (i.e. ‘a queue)
  - abstract types to preserve invariants (i.e. queue invariant)
  - *Abstract Stack Machine model of computation*

- Good for:
  - numerical simulations
  - implicit coordination between components (queues, GUI)

Imperative programming

OCaml

- No null. Partiality must be made explicit with **options**.
- Code is an **expression** that has a value. Sometimes computing that value has other effects.
- References are **immutable** by default, must be explicitly declared to be mutable

Java (and C, C++, C#)

- Null is contained in (almost) every type. Partial functions can return **null**.
- Code is a sequence of **statements** that do something, sometimes using expressions to compute values.
- References are **mutable** by default, must be explicitly declared to be constant

Explicit vs. Implicit Partiality

OCaml variables

- Cannot be changed once created, must use mutable record

```
type 'a ref = { mutable contents: 'a }
let x = { contents = counter () }
;; x.contents <- counter ()
```

- Cannot be null, must use options

```
let y = { contents = Some (counter ())}
;; y.contents <- None
```

- Accessing the value requires pattern matching

```
;; match y.contents with
     | None -> failwith "NPE"
     | Some c -> c.inc ()
```

Java variables

- Can be assigned to after initialization

```
Counter x = new Counter ();
x = new Counter ();
```

- Can always be null

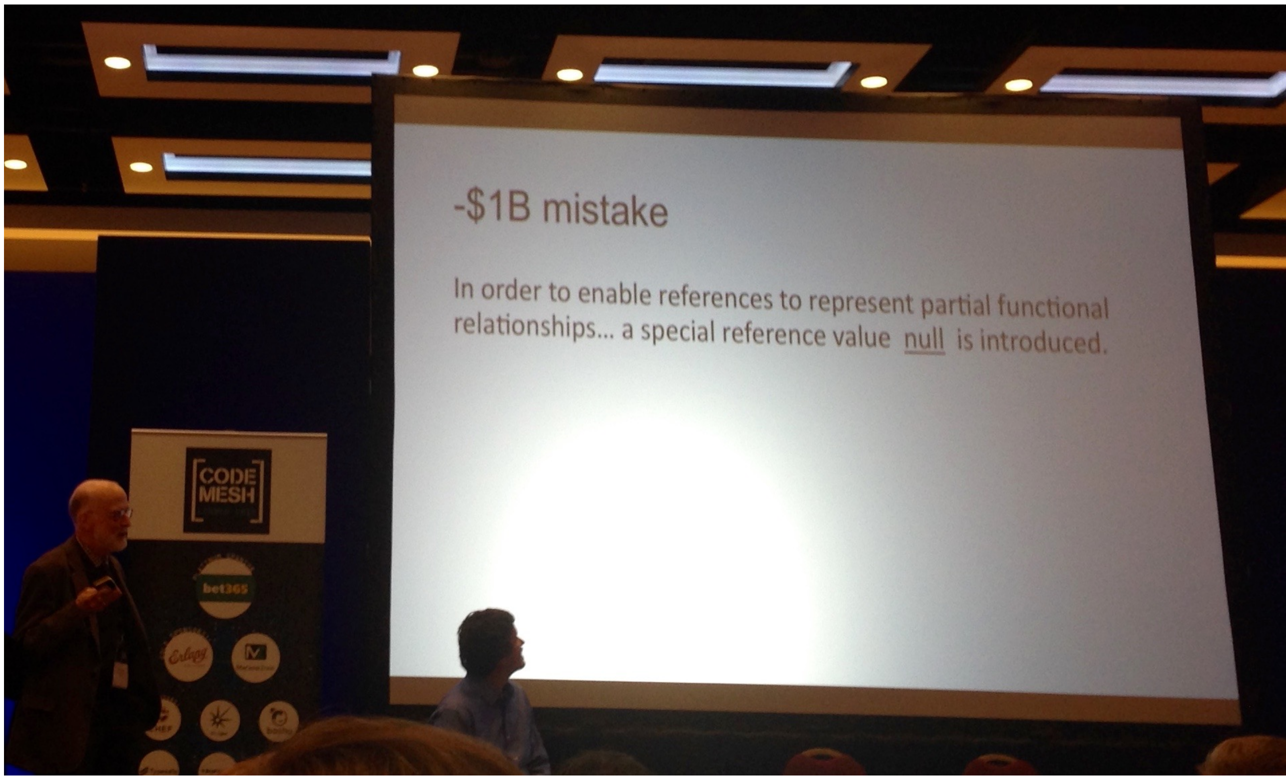
```
Counter y = new Counter ();
y = null;
```

- Check for null is implicit whenever a variable is used

```
y.inc();
```

- If null is used as an object (i.e. with a method call) then a **NullPointerException** occurs

The Billion Dollar Mistake



Sir Tony Hoare, QCon, London 2009