OCAML SIMULATION

As usual … the AST

```
type expr =
  | EInt of int
  | EBool of bool
  | Var of string
  | Let of string * expr * expr              (* let x = e1 in e2 *)
  | Prim of string * expr * expr              (* binop e1 e2 *)
  | If of expr * expr * expr                 (* if e1 then e2 else e3 *)
  | Fun of string * expr                     (* param identifier * funct body *)
  | Call of expr * expr (Apply)             (* fun identifier * param *)
  | GetInput of expr                         (* function that takes input, taint source*)
```

- Static scope discipline

Run Time Values … standard

```
type value =
  | Int of int
  | Bool of bool
  | Closure of string * expr * value env
```

Environment: handling bindings and taint

```
(* environment *)
type 'v env = (string * 'v * bool) list

(* binding *)
let rec lookup env x =
  match env with
  | [] -> failwith (x ^ "not found")
  | (y, v, _) :: r -> if x = y then v else lookup r x

(* taintness of a variable *)
let rec t_lookup env x =
  match env with
  | [] -> failwith (x ^ "not found")
  | (y, _, t) :: r -> if x = y then t else t_lookup r x
```

The environment maps variables to pairs consisting of a value and taint status

Interpreter: putting together the rules we described

Program takes exp., environment and default false taint status

```
let rec eval (e : expr) (env:value env) (t : bool) : value * bool =
 match e with
 | EInt n -> (Int n, t)
 | EBool b -> (Bool b, t)
 | Var x -> (lookup env x, t_lookup env x)
 | Prim (op, e1, e2) ->
  begin
   let v1, t1 = eval e1 env t in
   let v2, t2 = eval e2 env t in
    match (op, v1, v2) with
    (* taintness of binary ops is given by the OR of the taintness of the args *)
    | "*", Int i1, Int i2 -> (Int (i1 * i2), t1 || t2)
    | "+", Int i1, Int i2 -> (Int (i1 + i2), t1 || t2)
    | "-", Int i1, Int i2 -> (Int (i1 - i2), t1 || t2)
    | "=", Int i1, Int i2 -> (Bool (if i1 = i2 then true else false), t1 || t2)
    | "<", Int i1, Int i2 -> (Bool (if i1 < i2 then true else false), t1 || t2)
    | ">", Int i1, Int i2 -> (Bool (if i1 > i2 then true else false), t1 || t2)
    | _, _, _ -> failwith "Unexpected primitive."
  end
```

Interpreter (cont.)

```
| If (e1, e2, e3) ->
    begin
     let v1, t1 = eval e1 env t in
     match v1 with
       | Bool true -> let v2, t2 = eval e2 env t in (v2, t1 || t2)
       | Bool false -> let v3, t3 = eval e3 env t in (v3, t1 || t3)
       | _ -> failwith "Unexpected condition."
    end
```

evaluation of guard

→ body, and we take or of guard taint and body taint.

Interpreter (cont)

```
| Fun (f_param, f_body) -> (Closure (f_param, f_body, env), t)
| Call (f, param) ->
  let f_closure, f_t = eval f env t in
  begin
   match f_closure with
   | Closure (f_param, f_body, f_dec_env) ->
     let f_param_val, f_param_t = eval param env t in
      let env' = (f_param, f_param_val, f_param_t)::f_dec_env in
       let f_res, t_res = eval f_body env' t
        in (f_res, f_t || f_param_t || t_res)

   | _ -> failwith "Function expected error"
  end
```

Interpreter (cont)

```
| Fun (f_param, f_body) -> (Closure (f_param, f_body, env), t)
| Call (f, param) ->
  let f_closure, f_t = eval f env t in
  begin
   match f_closure with
   | Closure (f_param, f_body, f_dec_env) ->
      let f_param_val, f_param_t = eval param env t in
       let env' = (f_param, f_param_val, f_param_t)::f_dec_env in
        let f_res, t_res = eval f_body env' t
         in (f_res, f_t || f_param_t || t_res)

   | _ -> failwith "Function expected error"
  end
```

SPOT
THE ERROR

Imagine that value associated to Closure is tainted.
You are applying a function that is possibly
supplied by an attacker.

Interpreter (cont.)

```
| GetInput(e) -> eval e env true
```