Lec 13

Exceptions

handle & raise are both keyword

Defining exception:

convention: capital

```
exception Silly   ← type exn

if 1 = 2 then raise Silly else 42   : int
```

raise Silly: type 'a — so we can assign it some vacuous type. it never will have value.

42: matches whatever type

Handle exception:

```
(if 1 = 2 then raise Silly else 42) handle Silly ⇒ 100   ↪ 42
```

In general:

```
((e0) handle p1 ⇒ e1
           | p2 ⇒ e2
           ⋮
           | pn ⇒ en)
```

all need be same type

all need match exn

* exception propagates up until something handles it, else uncaught exception

```
(if 1 = 2 then raise Silly else 1 div 0) handle Silly ⇒ 100   ↪ 1000
                                                | Div ⇒ 1000
```

Exception with data

```
exception Rdiv of real          Rdiv : real → exn
                                Rdiv 1.0 : exn

raise (Rdiv 1.0)                : 'a

fun rdivide (x, y) =
   if Real.abs (y) ≤ 0.0001 then
      raise (Rdiv y)
   else
      x / y

fun f(x, y, z) = (x + rdivide (y, z)) handle (Rdiv r) ⇒ 1000.0 + r
f(2.0, 2.0, 2.0) ↪ 3.0
f(2.0, 2.0, 0.0) ↪ 1000.0
```

N-Queen problem – backtracking

```
int * int    (* board coordinate *)

(* threat : int * int → int * int → bool ; true if two position threaten each other *)
fun threat (a, b) (c, d) = a = c orelse b = d orelse a+b = c+d orelse a-b = c-d

(* conflict  int * int → (int * int) list → bool  if piece can go there *)
fun conflict p = List.exists (threat p)
                 ↑ point     ↑ ('a → bool) → 'a list → bool

exception Conflict
```

Exception approach

col boardsize existing pieces solution

```
(* addqueen : int * int * (int * int) list -> (int * int) list
fun addqueen (i, n, Q) =
  let
    fun try j = (if conflict (i,j) Q then raise Conflict
                 else if i = n then (i,j) :: Q
                 else addqueen (i+1, n, (i,j) :: Q))
                handle Conflict => if j=n then raise Conflict
                else try (j + 1)
  in
    try 1
  end

(* nqueens *)
fun nqueens n = (SOME addqueens (1, n, [])) handle Conflict => NONE
```

Continuation approach

```
(* addqueen : int * int * (int * int) list -> ((int * int) list -> 'a) -> (unit -> 'a) -> 'a *)
fun addqueen (i, n, Q) sc fc =
  let
    fun try j =
      let
        fun fcnew () = if j=n then fc() else try (j+1)
      in
        if conflict (i,j) Q then fcnew()
        else if i = n then sc ((i,j) :: Q)
        else addqueen (i+1, n, (i,j)::Q) sc fcnew
      end
  in
    try 1
  end

fun nqueens n = addqueen (1, n, []) (fn Q => SOME Q) (fn () => NONE)
```

continuation nor exception approach

→ Use NONE, omitted.

When continuation much better than exception

↓ embed fc in sc, so caller can reject answer

```
(* search : ('a -> bool) -> 'a tree -> ('a -> (unit -> 'b) -> 'b) -> (unit -> 'b) -> 'b
fun search p Empty sc fc = fc ()
  | search p (Node(L,x,R)) sc fc =
      let
        fun fcnew = search p L sc (fn () => search p R sc fc)
      in
        if p x then
          sc x fcnew
        else
          fcnew ()
      end

fun even x = x mod 2 = 0
                                  (* like normal find
fun findeven T = search even T (fn x => fn _ => SOME x) (fn () => NONE)
                                  (* finds all without extra work
fun findevens T = search even T (fn x => fn k => x::k()) (fn () => [])
```