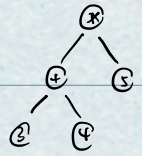
Continuation

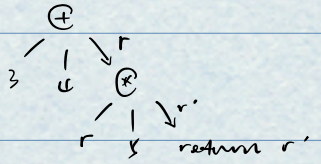
(3 + 4) * 5

add 3 4 and pass the result r to the computation

mult r 5 and pass the result r' to the computation

return r'





① fun add x y k = k(x+y)

fun mult x y k = k(x*y)

↑ continuation

add : int -> int -> (int -> 'a) -> 'a

eg. add 3 4 (fn r => mult r 5 (fn r' => r'))

=> (fn r => ---)(3 + 4)

=> [7/r] mult r 5 (fn r' => r')

=> (fn r' => r')(7 * 5)

=> [35/r'] r'

=> 35

"tail recursive"

② (* sum : int list -> int *)

fun sum [] = 0

| sum (x::xs) = x + (sum xs)

(* tsum : int list -> int

ENS: tsum(L, acc) = sum L + acc

*)

fun tsum ([], acc) = acc

| tsum (x::xs, acc) = tsum (xs, x + acc)

no explicit accumulator

(* csum : int list -> (int -> 'a) -> 'a

ENS: csum L k ≅ k (sum L)

*)

fun csum [] k = k 0

| csum x::xs k = csum xs (fn s => k(x+s))

s @ (x::xs) ?

CPS means roughly call functions tail-recursively with functions as accumulator

initial continuation

csum [2, 3] Int.toString -> "5"

csum [2, 3] : (int -> 'a) -> 'a

csum [2, 3] Int.toString

=> csum [3] (fn s => Int.toString (2+s)) ≅ s'

=> csum [] (fn s' => Int.toString (3+s'))

=> (fn s' => Int.toString (3+s')) 0

=>

② datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree

(* inorder : 'a tree * 'a list → 'a list

ENS : inorder (T, acc) ≅ L @ acc where L consists of the elements of T as seen in an inorder traversal

*)

```
fun inorder (Empty, acc) = acc
  | inorder (Node (l, x, r), acc) = inorder (l, x :: inorder (r, acc))
```

(* same : int list * int list → bool

ENS : same (X, Y) ⇒ { true, if X ≅ Y
                    { false otherwise

*)

```
fun same ([], []) = true
  | same (x::xs, y::ys) = x=y andalso same (xs, ys)
  | same _ = false
```

⇒ len X ≠ len Y

(* treematch : int tree * int list → bool

ENS : treematch (T, L) ⇒ { true, if the inorder traversal of T is L
                         { false, otherwise

```
fun treematch (T, L) = same (inorder (T, []), L)
```

problem : if the first elements are different, still scan through the entire tree in inorder

(* prefix : int tree → int list → (int list → bool) → bool

REQ : k is total

ENS : prefix T L k ⇒ { true, if ∃ lists L_1, L_2 s.t. $L \cong L_1 @ L_2$, the inversed traversal of T is L_1, k(L_2) ≅ true;
                     { false, otherwise

```
fun prefix Empty L k = k(L)   ≅ L1 = [] ⇒ L2 = L
  | prefix (Node (left, x, right)) L k =
      prefix left L (fn [] => false → len(T) > len(L)
                      | (y::ys) => (x=y) andalso (prefix right ys k))
```

```
fun treematch (T, L) = prefix T L List.null
```

'a list → bool (aka null)

```
fun null [] = true
  | null _ = false
```

② success & failure continuation

(* search : ('a → bool) → 'a tree → ('a → 'b) → (unit → 'b) → 'b

success continuation failure continuation

REQ : p is total

ENS : search p T sc fc ⇒ { sc(x) if x is in the tree & p(x) ≅ true
                         { fc (), otherwise

*)

```
fun search p Empty sc fc = fc ()   cannot have x to call sc!
  | search p (Node (l, x, r)) sc fc = if p(x) then sc(x)
      else search p l sc (fn () => search p r sc fc)
```

```
fun findevens (T : int tree) : int option =
    search (fn x => x mod 2 = 0) T SOME (fn () => NONE)
                                   (fn x => SOME x)
```

Exception & handle

exception Silly

Silly : exn (exception or extensible)

raise Silly : 'a → type? can accomodate! eg. (if 3 = 4 then raise Silly else 0) : int
(if 3 = 4 then raise Silly else "0") : string

```
fun f(x) = f(x)      f : 'a -> 'b
fun g(x) = raise Silly    g : 'a -> 'b
(if 3 = 4 then g(0) else "0") : string      g : int -> string
```

exception Rdiv of real constructors are also functions!

Rdiv : real → exn Rdiv(3.14) : exn

raise Rdiv(3.14) : 'a

a keyword, not function

```
fun rdivide (r1, r2) =
    if Real.abs(r2) ≤ 0.00005 then raise Rdiv(r1)      rdivide : real * real -> real
    else r1/r2
```

(real)

rdivide(3.14, 0.0) => --- raise Rdiv(3.14) not a value! error message at top level

```
( rdivide(3.14, 0.0)
  Block of code ---
  ----
) handle Rdiv(r) => "oops, you tried to divide " ^ (Real.toString(r)) ^ " by 0!"
```

suppose this has type string [3.14/r] need return a string

```
( Block of code
  [expression]
) handle Rdiv(r) => 9.999999 * r
       | Silly   => 1.0
       | _       => 0.0
```

```
( e0 handle p1 => e1
          | p2 => e2
          ⋮
          | pn => en ) : t
```

e0, e1, ..., en must all have the same type t, which is then the type of the expression & the patterns p1, ..., pn must match exceptions.

If e0 evaluates to v then that is the value of the overall expression.

n - Queens

local decisions are sometimes bad => back tracing!

exception Conflict (i,j) represent positions on n×n board

col row 1 ≤ i,j ≤ n

```
(* Threat : int * int -> int * int -> bool *)
fun Threat (x,y) (a,b) = x=a orelse y=b orelse x+y=a+b orelse x-y=a-b

(* conflict : int * int -> (int * int) list -> bool *)
fun conflict p = List.exists (threat p)
```

threat p : int * int → bool

('a → bool) → 'a list → bool

'a : int * int

returns int * int list → bool

1st attemp raise exception

```
(* addqueen : int * int * (int * int) list -> (int * int) list
   col   n   existing queen placements   all queen placements

   & raise Conflict if it is not possible to place the remaining queens
   queens : int -> (int * int) list or raise Conflict
*)
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

fun queens n = addqueen (1, n, [])
```

check if can move queen up

alternative : fun queens n = SOME (addqueen (1, n, [])) handle Conflict => NONE

eg. queens 4 => [(4,3), (3,1), (2,4), (1,2)] in reverse order !

queens 1 => [(1,1)]

queens 2 => raise an unhandled conflict

2nd attemp : success / failure continuation

```
(* addqueen : int * int * (int * int) list -> ((int * int) list -> '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 (i,j)::Q
                  else addqueen (i+1, n, (i,j)::Q) sc fcnew
                end
  in
    try 1
  end

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

3rd attemp : use options

```
fun try =
  case (if conflict (i,j) Q then NONE
        else if i=n then (i,j)::Q
        else addqueen (i+1, n, (i,j)::Q)) of
```

```
    NONE => if j=n then NONE else try (j+1)
  | result => result
```

Multi-purpose Continuation

```
(* find : int tree -> (int -> (unit -> 'a) -> 'a) -> (unit -> 'a) -> 'a *)
fun find Empty se fe = fe()
  | find (Node(l,x,r)) se fe =
    let
      fun fenew() = find l se (fn () => find r se fe)
    in
      if x mod 2 = 0 then se x fenew else fenew()
    end

fun findfirst T = find T (fn x => fn fe => SOME(x)) (fn () => NONE)
fun findall T = find T (fn x => fn fe => x::(fe())) (fn () => [])
fun count T = find T (fn x => fn fe => 1+fe()) (fn () => 0)
```