(op o) : ('a -> 'b) * ('c -> 'a) -> ('c -> 'b)   ; (f o g)(x) = f(g(x)),   (op o) o (op o) is ill-typed
- foldl (op ^) "" ["h","e","l","l","o"]   ; val it = "olleh" : string
- foldr (op ^) "" ["h","e","l","l","o"]   ; val it = "hello" : string

Staging : SML functions are pass by value, i.e. function arguments evaluated at bind

fun f3 (x:int) : int -> int = let val z = horriblecomputation(x) in (fn y => z + y) end

Proofs

In CPS proofs, usually we apply inductive hypothesis early. Don't reason about when s or k is called because that may assume your code is correct. If we have exceptions, we want to case on whether exception was raised or not

Regex: (SOUND: matches => in language, COMPLETE: in language => matches)

Sound: assume cs, k such that match(regex) is true. NTS that there exist p, s such that p@s = cs, p is in the language of the regex, and k s is true.
Complete: converse of Sound

Writing Continuation Functions

• In base cases, we apply the continuation instead of directly returning a value
• In recursive cases, the continuation acts as a functional accumulator.
• In exceptional cases, we may discard (or duplicate) the continuation to circumvent the normal control flow.equivalence on basic types != equivalence on function expressions

Basic types: "both evaluate to same value, raise the same exception, or fail to terminate"
For function expressions, they just need to evaluate to extensionally equivalent function values (e.g. fn x => x + x, fn y => y * 2)

Exception
```
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
```

Option types
```
fun addqueen(i, n, Q) = let
  fun try j=
    case (if conflict (i,j) Q then NONE
             else if i=n then SOME((i,j)::Q)
             else addqueen(i+1, n, (i,j)::Q))
    of NONE => if (j=n)
                  then NONE else try(j+1)
     | result => result
  in try 1 end
```

Failure continuation
```
fun addqueen(i, n, Q, fc) = let
  fun try j =
    if j=n+1 then fc()
    else if (conflict (i,j) Q) then try(j+1)
    else if i=n then (i,j)::Q
    else
addqueen(i+1, n, (i,j)::Q, fn () => try(j+1))
    in try 1 end
```

		type of recurrence	big-O class
$T(n)$	$=$	$T(n \text{ div } 2)+c$	$O(\log n)$
$T(n)$	$=$	$T(n-1)+c$	$O(n)$
$T(n)$	$=$	$2T(n \text{ div } 2)+c$	$O(n)$
$T(n)$	$=$	$T(n \text{ div } 2)+c_1 n+c_0$	$O(n)$
$T(n)$	$=$	$2T(n \text{ div } 2)+c_1 n+c_0$	$O(n \log n)$
$T(n)$	$=$	$T(n-1)+c_1 n+c_0$	$O(n^2)$
$T(n)$	$=$	$T(n-1)+c_2 n^2+c_1 n+c_0$	$O(n^3)$
$T(n)$	$=$	$2T(n-1)+c$	$O(2^n)$

Summation formula

Geometric series:

$\sum_{k=1}^{n} r^k = \frac{r(1-r^n)}{1-r}$, (and if $|r| < 1$), $\sum_{k=1}^{\infty} r^k = \frac{r}{1-r}$

Some trees: $\sum_{i=1}^{d} i2^i = (d-1)2^{d+1} + 2$ (homework 4), $\sum_{i=0}^{d} 2^i = 2^{d+1} - 1$ (exam review)

Interface

```
datatype 'a list = nil | :: of 'a * 'a list
exception Empty

val null : 'a list -> bool
val length : 'a list -> int
val @ : 'a list * 'a list -> 'a list
val hd : 'a list -> 'a
val tl : 'a list -> 'a list
val last : 'a list -> 'a
val getItem : 'a list -> ('a * 'a list) option
val nth : 'a list * int -> 'a
val take : 'a list * int -> 'a list
val drop : 'a list * int -> 'a list
val rev : 'a list -> 'a list
val concat : 'a list list -> 'a list
val revAppend : 'a list * 'a list -> 'a list
val app : ('a -> unit) -> 'a list -> unit
val map : ('a -> 'b) -> 'a list -> 'b list
val mapPartial : ('a -> 'b option) -> 'a list -> 'b list
val find : ('a -> bool) -> 'a list -> 'a option
val filter : ('a -> bool) -> 'a list -> 'a list
val partition : ('a -> bool)
                  -> 'a list -> 'a list * 'a list
val foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
val foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
val exists : ('a -> bool) -> 'a list -> bool
val all : ('a -> bool) -> 'a list -> bool
val tabulate : int * (int -> 'a) -> 'a list
val collate : ('a * 'a -> order)
                -> 'a list * 'a list -> order
```

```
fun lazy_fib a b = Cons(a, fn () => Cons(b, fn () =>
    let val L = lazy_fib a b in zip_with (op +) L (tail L) end))
fun sieve s = delay (fn () => sieve' (expose s))
and sieve' (Empty) = Empty
  | sieve' (Cons(p, s)) = Cons(p, sieve (filter (notDivides p) s))

signature ARITHMETIC =
sig
  type integer
  (* converts type int into the specified integer type *)
  val rep : int -> integer
  (* converts type integer to int *)
  val toInt : integer -> int
  (* allows you to view the integer as a string *)
  val display : integer -> string
  (* add two integers together *)
  val add : integer * integer -> integer
  (* multiply two integers together *)
  val mult : integer * integer -> integer
end

functor AlphaBeta (Settings : SETTINGS) : PLAYER =
```

:> is opaque, hides valueai

Seq.length : 'a Seq.seq -> int
Seq.empty : unit -> 'a Seq.seq
Seq.singleton : 'a -> 'a Seq.seq
Seq.append : 'a Seq.seq * 'a Seq.seq -> 'a Seq.seq
Seq.tabulate : (int -> 'a) -> int -> 'a Seq.seq
Seq.nth : 'a Seq.seq -> int -> 'a
Seq.filter : ('a -> bool) -> 'a Seq.seq -> 'a Seq.seq
Seq.map : ('a -> 'b) -> 'a Seq.seq -> 'b Seq.seq
Seq.reduce : (('a * 'a) -> 'a) -> 'a -> 'a Seq.seq -> 'a
Seq.reduce1 : (('a * 'a) -> 'a) -> 'a Seq.seq -> 'a
Seq.mapreduce : ('a -> 'b) -> 'b -> ('b * 'b -> 'b) -> 'a Seq.seq -> 'b
Seq.toString : ('a -> string) -> 'a Seq.seq -> string
Seq.repeat : int -> 'a -> 'a Seq.seq
Seq.flatten : 'a Seq.seq Seq.seq -> 'a Seq.seq
Seq.flatten ss is equivalent to reduce append (empty ()) ss
Seq.zip : ('a Seq.seq * 'b Seq.seq) -> ('a * 'b) Seq.seq
Seq.split : 'a Seq.seq -> int -> 'a Seq.seq * 'a Seq.seq
Seq.take : 'a Seq.seq -> int -> 'a Seq.seq
Seq.drop : 'a Seq.seq -> int -> 'a Seq.seq
Seq.cons : 'a -> 'a Seq.seq -> 'a Seq.seq
Seq.update : 'a Seq.seq * int * 'a -> 'a Seq.seq
Seq.toList : 'a seq -> 'a list
Seq.fromList : 'a list -> 'a seq

Function	Work	Span
`Seq.length S`	$O(1)$	$O(1)$
`Seq.empty ()`	$O(1)$	$O(1)$
`Seq.singleton x`	$O(1)$	$O(1)$
`Seq.append (S1, S2)`	$O(\lvert S1\rvert + \lvert S2\rvert)$	$O(1)$
`Seq.tabulate f n`	$O(n)$	$O(1)$
`Seq.nth S i`	$O(1)$	$O(1)$
`Seq.filter p S`	$O(\lvert S\rvert)$	$O(\log \lvert S\rvert)$
`Seq.map f S`	$O(\lvert S\rvert)$	$O(1)$
`Seq.reduce c b S`	$O(\lvert S\rvert)$	$O(\log \lvert S\rvert))$
`Seq.reduce1 c b S`	$O(\lvert S\rvert)$	$O(\log \lvert S\rvert))$
`Seq.mapreduce l e n S`	$O(\lvert S\rvert)$	$O(\log \lvert S\rvert))$
`Seq.toString ts S`	$O(\lvert S\rvert)$	$O(\log \lvert S\rvert)$
`Seq.repeat n x`	$O(n)$	$O(1)$
`Seq.flatten S`	$O(\lvert S\rvert + \sum_{s\in S} \lvert s\rvert)$	$O(\log \lvert S\rvert)$
`Seq.zip (S1,S2)`	$O(\min(\lvert S1\rvert, \lvert S2\rvert)$	$O(1)$
`Seq.split S i`	$O(\lvert S\rvert)$	$O(1)$
`Seq.take S i`	$O(i)$	$O(1)$
`Seq.drop S i`	$O(\lvert S\rvert - i)$	$O(1)$
`Seq.cons x S`	$O(\lvert S\rvert)$	$O(1)$
`Seq.update (S, i, x)`	$O(\lvert S\rvert)$	$O(1)$
`Seq.toList S`	$O(\lvert S\rvert)$	$O(\lvert S\rvert)$
`Seq.fromList L`	$O(\lvert L\rvert)$	$O(\lvert L\rvert)$

```
                    .
                  / |   \
                 /  |    \
 .              f x1  f x2 .... f xn
 |               \  |    /
                  \ |   /
 .   length,nth     .             map,tabulate
```

```
          .
   / ......\     (n/2 branches wide here)
  . .      . .
  \ /      \ /      (width n/4 here)
   . .     . .
   \ /     \ /
     . .  . .
         .
         .
         .           (log n levels deep)
        \ /
         .                         reduce
```

Lazy:
datatype 'a stream = Stream of unit -> 'a front
and 'a front = Empty | Cons of 'a * 'a stream
delay : (unit -> 'a front) -> 'a stream
expose : 'a stream -> 'a front

```
fun filter p s = delay (fn ()  => filter' p (expose s))
and filter' p (Empty) = Empty
   | filter' p Cons(x, xs) =
     if p(x) then Cons(x, filter p xs) else filter' p (expose xs)
```

Imperative:
ref : 'a -> 'a ref
! : 'a ref -> 'a
(op :=) : 'a ref * 'a -> unit

```
fun reachable (g:graph) (x:int, y:int) : bool =
 let val visited = ref [ ]
 fun dfs (n:int) : bool = (n = y) orelse
    let val V = !visited
    in (not (mem n V)) andalso (visited := n::V; exists dfs (G n)) end
 in dfs x end
```

Red/Black Tree Representation (RBT) Invariants:

1. Tree is a binary search tree
2. The children of a red node are black.
3. Every path from the root to a leaf has the same number of black nodes, called the black height of the tree.

We can temporarily violate 2: red node children are black except maybe at root: the root and one of its children may both be red.