	Recurrence	Asymptotic Upper Bound
$T(n) =$	$T(n-1)+c$	$O(n)$
$T(n) =$	$T(n/2)+c$	$O(\log n)$
$T(n) =$	$2T(n/2)+c$	$O(n)$
$T(n) =$	$T(n/2)+c_1n+c_0$	$O(n)$
$T(n) =$	$2T(n/2)+c_1n+c_0$	$O(n\log n)$
$T(n) =$	$T(n-1)+c_1n+c_0$	$O(n^2)$
$T(n) =$	$2T(n-1)+c$	$O(2^n)$

$$\sum_{i=0}^{n} 2^i = 2^{n+1}-1 \qquad \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=0}^{\log n} \frac{1}{2^i} \le \sum_{i=0}^{\infty} \frac{1}{2^i} = 2 \in O(1) \qquad \sum_{i=0}^{n-1} a^i = \frac{1-a^n}{1-a}$$

function application is left associative: `f x y means (f x) y`
arrows right associative: `t1 -> t2 -> t3 = t1 -> (t2 -> t3)`
HOFs: take functions as arguments and/or return HOFs
combinators (HOFs): functions that combine small pieces of code into larger pieces of code – e.g., composition `f o g`
point-wise principle: specify what a particular combination of functions means by writing out explicitly how the combinator evaluates code for a given argument (currying makes this easy) -> use the combinator to <u>combine functions without referring explicitly to arguments of the functions</u>: take in function values and return a function value: **point-free programming**

point-specific:
```
fun (f ++ g) x = f(x) + g(x)
fun (f ++ g) = fn x => f(x) + g(x)
```
point-free:
```
fun quadratic = square ++ double
```
staging: move parts of the computation close to where the arguments required for the computation appear; perform useful work prior to receiving all its arguments
```
fun g x y = let val z = hc(x) in z + y end
```
staging **does not** occur, lambda expression not applied:
```
g 5 => [5 / x] fn y => let val z = hc(x) in z + y end
g5 2 => [5/x, 2/y] let val z = hc(x) in z + y end
     => [5/x, 2/y, someint/z] z + y (*takes 10 months*)
```
staging: hc doesn't depend on x:
```
fun h x = let val z = hc(x) in (fn y => z + y) end
```

map: replace constituent values (*defined over general datatypes*)
```
(* map : ('a -> 'b) -> 'a list -> 'b list *)
map f [x1, …, xn] = [f x1, …, f xn]
```
fold: replace (n-ary) constructors with (n-ary) functions
```
(* ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)
foldl f z [x1, …, xn] = f(xn, …, f(x2, f(x1, z)))
foldr f z [x1, …, xn] = f(x1, …, f(xn-1, f(xn, z)))
L = [1, 2, 3, 4]
fun foldr f z [] = z
  | foldr f z (x :: xs) = f(x, foldr f z xs)
foldr (op -) 0 L = 4 - (3 - (2 - (1 - 0))) = 2
foldr (op ::) [] L = [1, 2, 3, 4]
fun foldl f z [] = z
  | foldl f z (x :: xs) = foldl f f(x, z) xs
foldl (op -) 0 L = 1 - (2 - (3 - (4 - 0))) = ~2
foldl (op ::) [] L = [4, 3, 2, 1]

op o : ('b -> 'c) * ('a -> 'b) -> ('a -> 'c)
val filter : ('a -> bool) -> 'a list -> 'a list
val zip : 'a list * 'b list -> ('a * 'b) list
```

Proof of <u>**correctness**</u>

1. prove **termination** – (match r cs k) returns a value for all arguments r, cs, k satisfying REQUIRES specifications
2. given termination, simplify ENSURES, prove **soundness** ("only if") and **completeness** ("if") via structural induction *(match r cs k) = true if and only if there exists p and s s.t.…*

- Non-value expressions (2 + 2) are not valid patterns
```
val () = Test.string_int("test_1", ("two", 2), f 2)
val () = Test.int_list_eq("test_2", [1, 3], g 2)
```
- `div`, `mod`: integer division, mod, `/`: real division
- "by <u>*type inference*</u>, the first argument to v should have type int, but not x has type bool; no value since ill-typed"
- **extensional equivalence** (expressions/functions): (1) reduce to same value, (2) raise same exception, (3) loop forever (function-type exp.): reduce to function values $f_1 \cong f_2$, (2), (3)

a function, `f`, is written in continuation passing style (CPS) if:

1. f takes at least one continuation as an argument
2. if f makes a call to a recursive function g, then this call is a tail call and g must be in CPS
3. if f makes a call to a function g, which itself has continuations, then this call is a tail call and g must be in CPS
4. f calls its continuation(s) and does so in tail call(s)

*in a CPS function, you **may**…*	*you **may not**…*
• case on value of input • use let…in…end • use non-recursive helper functions in non-tail calls • use/case on result of predicates in non-tail calls • pass in modified continuations, predicates, and/or other arguments to the recursive calls you make	• manipulate, case on, or otherwise use the result of either recursive calls, CPS helper functions, or continuation calls ○ break the tail-call requirement • use non-CPS recursive helper function

```
fun size' Empty k = k 0
  | size' (Node (L, x, R)) k = size' L
      (fn resL => size' R (fn resR => k (resL + resR + 1)))
```

polymorphism:

- `raise Div : 'a` (not a value!)
- `loop 0 : 'a`, where `fun loop x : 'a -> 'b = loop x`
- `[] : 'a list`
- datatypes cannot have base type! `datatype 'a tree`
  - polymorphic must be in scope where defined
- instance of type: some type that follows the form of another type structure – a → (b → c) * a is an instance of a → b * a, but a → b (too general) and a → b * c (too specific) are not

built-in **exceptions** (`exception Div`)
"extensible" datatype, 'extend' upon exception datatype & declare new exception constructors using `exception` *keyword*

- cannot declare top-level polymorphic exceptions
  - no possible binding in scope
  - allowed: `fun f (x : 'a) =`
    `let exception Poly of 'a in () end`
- `Div : exn, raise : exn -> 'a`
- `Fail of string` (Unimplemented)
- `Match` (non-exhaustive casing)
  - pattern-**match** → cannot find a clause that matches
- `Bind` (pattern matching failed in `val` binding)
  - **bind** values to variables, cannot create valid binding
- **handling**: `(expr : t) handle exn1 => e1 : t`
  `| exn2 => e2 : t`
  where exn1 are exception constructors
- `case (raise exn : t) of`
  `(exn1 : t) => "A" | (exn2 : t) => B`

$L(a) = \{a\}$	(singleton set) for every character $a \in \Sigma$
$L(0) = \{\}$	(the empty language, no strings)
$L(1) = \{\varepsilon\}$	(the language consisting of the empty string)
$L(r_1 + r_2) = \{s \| s \in L(r_1) \text{ or } s \in L(r_2)\}$	
$L(r_1 r_2) = \{s_1 s_2 \| s_1 \in L(r_1) \text{ and } s_2 \in L(r_2)\}$	
$L(r^*) = \{s \| s = s_1 s_2 \dots s_n, \text{ some } n \geq 0, \text{ with each } s_i \in L(r)\}$	

`{ε, "a", "b", "ab"}`	`(a+1)(b+1)`
set of all strings with no consecutive "b"	`(a+ba)*(b+1)`

```
fun rev (Times (r1, r2)) = Times (rev r2, rev r1)
  | rev (Plus (r1, r2)) = Plus (rev r1, rev r2)
  | rev (Star r) = Star (rev r)
fun ap (Char c) = Plus (One, Char c) (* all prefixes *)
  | ap (Times (r1, r2)) = Plus (ap r1, Times (r1, ap r2))
  | ap (Plus (r1, r2)) = Plus (ap r1, ap r2)
  | ap (Star r) = Times (Star r, ap r)
fun nub One = Zero (* remove One *)
  | nub Plus (r1,r2) = Plus (nub r1, nub r2)
  | nub Times (r1,r2) =
        Plus (Times (nub r1, r2), Times (r1,nub r2))
  | nub Star r = Times (nub r, Star r)

fun match (r : regexp) (cs : char list)
    (k : char list -> bool) : bool = case r of
    Zero   => false
  | One    => k cs
  | Char c => (case cs of [] => false
                 | c' :: cs' => c' = c andalso k cs')
  | Plus  (r1,r2) => match r1 cs k orelse match r2 cs k
  | Times (r1,r2) => match r1 cs (fn cs' => match r2 cs' k)
  | Star r =>
     k cs orelse match r cs (fn cs' => match (Star r) cs' k)
  | Star r =>
   let fun m' cs' = k cs' orelse match r cs' m' in m' cs end
fun accept r s = match r (String.explode s) List.null
```

```
fun flatten ([] : int list list) : int list = []
  | flatten (L :: LS) = (case L of
       [] => flatten LS
     | x :: xs => x :: (flatten (xs :: LS)))
```

We want to show that for all values `LL : int list list`, `flatten LL ≅ oldFlat LL`.

Proof: We proceed by structural induction on LL

Base Case: `LL = []` … Hence, `flatten [] ≅ oldFlat []`

Inductive Case: `LL = L :: LS` for some values `L : int list` and `LS : int list list`

Inductive Hypothesis: Assume `flatten LS ≅ oldFlat LS`

Want to show: `flatten L :: LS ≅ oldFlat L :: LS`

We then proceed by structural induction on L

Inner Base Case: Let `L ≅ []` …

Inner Inductive Case: Let L ≅ `x::xs` for some x : int and xs : int list

Inner IH: `flatten (xs :: LS) ≅ oldFlat (xs :: LS)`

WTS: `flatten((x::xs)::LS) ≅ oldFlat ((x::xs)::LS)`

[cite outer/inner IH, **totality**, clause # of function]

Since `flatten L :: LS ≅ oldFlat L :: LS`, by structural induction on LL, for all LL : int list list, flatten LL ≅ oldFlat LL.

mapping and combinator examples:

```
(* tmap : ('a -> 'b) -> 'a tree ->g 'b tree *)
fun tmap f Empty = Empty
  | tmap f (Node(l, x, r)) = Node(tmap f l, f x, tmap f r)
(* tfold: ('b * 'a * 'b -> 'b) -> 'b -> 'a tree -> 'b *)
fun tfold f z Empty = z
  | tfold f z Node(l,x,r) = f (tfold f z l, x, tfold f z r)
val stringify : int tree -> string tree = tmap Int.toString
val treesum = tfold (fn(a, x, b) => a + x + b) 0

fun mapEnum (f : int * 'a -> 'b) (L : 'a list) =
  let
    val comb_fn = fn (x, (i, L')) => (i + 1, f(i, x) :: L')
    val (_, result) = foldl comb_fn (0, []) L
  in
    foldl (op ::) [] result
  end

scanl f z [x1, x2, ..., xn] ==>
[z, f(x1, z), f(x2, f(x1, z))…f(xn, …, f(x2, f(x1, n))…)]
fun scanl f z L =
 let
  fun comb (x, (fxz, L)) = (f (x, fxz), f (x, fxz) :: L)
  val (_, res) = foldl comb (z, [z]) L
 in
  foldl (op ::) [] res
 end
fun scanlCPS f z [] k = k ([], z)
  | scanlCPS f z (x :: xs) k = f (x, z)
       (fn z' => scanlCPS f z' xs
       (fn (L', x') => k (z::L', x')))
```

using recursion	using HOFs

`fun ptn p [] = ([], [])`
`  | ptn p (x :: xs) =`
`let`
` val (R1, R2) = ptn p xs`
`in`
` if (p x) then`
`    (x :: R1, R2) else`
`    (R1, x :: R2) end`
	`fun partition' p L =`
	`foldr (fn (x, (R1, R2))`
	`   => if (p x) then`
	`      (x :: R1, R2) else`
	`      (R1, x :: R2))`
	`([], []) L`

```
fun leaves Nub = []
  | leaves (Branch (L, C, R, v)) =
      (case (leaves L, leaves C, leaves R) of
          ([] ,[] ,[]) => [v]
        | (resL, resC, resR) => (resL @ resC) @ resR)
```

Let n be the number of Branches in the tree.

- $W_{\text{leaves}}(0) = k_0,\ W_{\text{leaves}}(1) = k_1$
- $W_{\text{leaves}}(n) = W_{\text{leaves}}(n_l) + W_{\text{leaves}}(n_c) + W_{\text{leaves}}(n_r) + W_{@}(n_l) + W_{@}(n_l + n_c) + k_2 = 3W_{\text{leaves}}\left(\frac{n}{3}\right) + kn$

There are $\log_3 n$ levels, the work/node at level i is $\frac{kn}{3^i}$.

There are 3^i nodes at level i.

$$\sum_{i=0}^{\log_3 n} 3^i \cdot \frac{kn}{3^i} = kn \log_3 n \Longrightarrow W_{\text{leaves}}(n) \in O(n \log n)$$

- $S_{\text{leaves}}(n) = \max\left(S_{\text{leaves}}(n_l), S_{\text{leaves}}(n_c), S_{\text{leaves}}(n_r)\right) + \max\left(S_{@}(n_l), S_{@}(n_l + n_c)\right) + k_2 = S_{\text{leaves}}\left(\frac{n}{3}\right) + kn$

There are $\log_3 n$ levels, the work/node at level i is $\frac{kn}{3^i}$.

There is 1 node at level i.

$$\sum_{i=0}^{\log_3 n} \frac{kn}{3^i} \leq \sum_{i=0}^{\infty} \frac{kn}{3^i} = \frac{kn}{1 - \frac{1}{3}} = \frac{3kn}{2} \Longrightarrow S_{\text{leaves}}(n) \in O(n)$$