	Recurrence	Asymptotic Upper Bound
$T(n) =$	$T(n-1)+c$	$O(n)$
$T(n) =$	$T(n/2)+c$	$O(\log n)$
$T(n) =$	$2T(n/2)+c$	$O(n)$
$T(n) =$	$T(n/2)+c_1 n+c_0$	$O(n)$
$T(n) =$	$2T(n/2)+c_1 n+c_0$	$O(n\log n)$
$T(n) =$	$T(n-1)+c_1 n+c_0$	$O(n^2)$
$T(n) =$	$2T(n-1)+c$	$O(2^n)$

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

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

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 combine functions without referring explicitly to arguments of the functions: 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
```

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

- `raise Div : 'a` (not a value! but is an expression)
- `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: type that follows form of another type structure

"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`

```
fun match (r : regexp) (cs : char list)
    (k : char list -> bool) : bool = case r of
  | 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
```

$L(a) = \{a\}$	(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)
$L(r_1+r_2) = \{s \mid s \in L(r_1) \text{ or } s \in L(r_2)\}$	
$L(r_1 r_2) = \{s_1 s_2 \mid s_1 \in L(r_1) \text{ and } s_2 \in L(r_2)\}$	
$L(r^*) = \{s \mid s = s_1 s_2 \ldots s_n, \text{ some } n \ge 0, \text{ with each } s_i \in L(r)\}$	

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

Proof of **correctness**
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
- `type age = int` – type abbreviations
- `datatype A = B | C` – new datatype
- "by *type inference*, the first argument to v should have type int, but not x has type bool; no value since ill-typed"
- **extensional equivalence** (expressions): (1) reduce to same value, (2) raise same exception, (3) loop forever (function-type exp.): for all `e1 ≅ e2`, `f e1 ≅ g e2`
- ephemeral (not persistent): mutable
- type checking happens before evaluation

in a **signature**	in **both**	in a **structure**
type specification	type declaration	val/fun declaration
val spec.	datatype decl.	structure decl.
structure spec.	exception decl.	

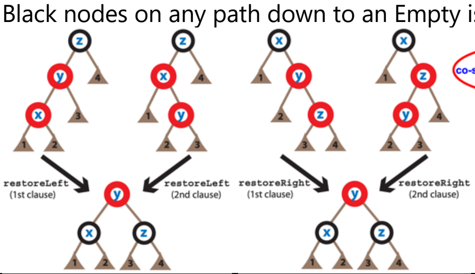
```
functor MkStruct (S1 : SIG1)
 :> SIG2 where type S2.t = S1.t
= struct ... end
MkStruct (structName)
```

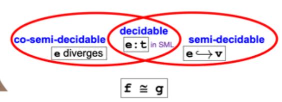
**functors take in exactly 1 structure*
-> structure ascribing to anon. sig.

```
sig
 type t        (* parameter *)
 type 'a dict (* abstract *)
end
```

```
functor MkSugar (
  structure A : SIG1
  structure B : SIG1
) : SIG2 = ...
MkSugar (
  structure A = ...
  structure B = ...)
struct
  type t = int
  datatype 'a dict =
     Empty | Node of …
```

red-back tree invariants: (roughly balanced: $d \le 2\log(n+1)$) (1) tree is sorted on key part of entries, (2) children of a Red node are Black, (3) each node has well-defined Black height: number of Black nodes on any path down to an Empty is the same





almost RBT: Red root may have Red child

productive streams: `Stream.expose S => Stream.Empty`, or `=> Stream.Cons (x, s')`, where `s'` is productive (i.e., doesn't loop forever or contains raised exceptions)

"constructors of datatype aren't declared in the signature: user external to the structure cannot pattern match on or use the constructors."

```
datatype 'a stream = Stream of unit -> 'a front
     and 'a front = Empty | Cons of 'a * 'a stream
fun delay f : (unit->'a front)->'a stream = stream f
fun expose (Stream d) : 'a stream -> 'a front = d ()
empty : 'a stream, val : 'a * 'a stream -> 'a stream
fun interleave s1 s2 =
     S.delay (fn () => interleave' (S.expose s1, s2))
and interleave' (S.Empty, s2) = S.expose s2
| interleave' (S.Cons(x, s1'), s2) =
     S.Cons(x, interleave s2 s1')
(* iterate f x => f0(x), f1(x), f2(x) *)
fun iterate F x = S.delay (fn () => iterate' F x)
and iterate' F x = S.Cons (x,
     S.delay (fn () => S.expose (iterate F(F(x)))
fun cycle s og n i = if i >= n then cycle og og n 0
  else S.delay(fn () => case S.expose s of
     S.Empty => S.Empty
   | S.Cons(x, s') => S.Cons(x, cycle s' og n (i+1)))
```

`empty, cons(e, st), tabulate(int -> 'a), null, hd, take (st, n), drop, append (s1, s2), map, filter, zip`

Brent's theorem: an expression e with work W and span S can be evaluated on a p-processor machine in time $\Omega(W/p, S)$.

- work of cost graph G: number of nodes in G
- span: num. of nodes on longest path from source to sink

using recursion/CPS	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`

context-free grammar G is specified by a (1) finite alphabet Σ of terminals (disjoint with V), (2) set V of non-terminals, (3) start symbol in V (often S), (3) and set of expansion rules, each of the form $N \to \omega$, with $N \in V, \omega \in (\Sigma \cup V)^*$. $L(G) = \{\omega \in \Sigma^* | S \Rightarrow \omega\}$
leftmost derivation: each step expands current leftmost non-terminal (if more than one leftmost derivation: G is *ambiguous*)
pumping lemma (for regex): let L be an infinite regular language (L = L(r) for some regex r), then ∃ strings α, ω, β such that $\omega \neq \varepsilon$ and $\alpha\omega^k\beta \in L$ for every $k \ge 0$. ex.: rules R of G such that $L(G) = L(G_1) \cup L(G_2)$ is $R : S \to S_1 | S_2$ along with rules R_1 and R_2

```
! : 'a ref -> 'a.      (op :=) : 'a ref * 'a -> unit
!e : t if e : t ref    ref e : t ref if e : t
e1 := e2 : unit if e1 : t ref and e2 : t
```

- evaluate e1, if e1 ↪ cell c, evaluate e2, if e2 ↪ v, change contents of c to be v and return ()

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

WTS 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.

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

Let n be the number of Branches in the tree.

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

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

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

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

constant-time: `empty ()`, `singleton x`, `nth S i`, `null S`, `length S`, `subseq S (i, l)`, `take/drop/split S i`

constant-span: `tabulate f n` `n`, `rev S` `n`, `append (S1, S2)` `|S1| + |S2|`, `map f S` `|S|`, `zip (S1, S2)` `min(|S1|, |S2|)`, `enum S` `|S|`, `update (S, (i, x))` `|S|`, `inject (S, U)` `|S| + |U|`

`flatten S` `|S|+sum|s|`, `log|S|`, `filter p S` `|S|`, `log|S|`, `toString ts S` `|S|`, `log|S|`, `reduce g z S` `|S|`, `log|S|`, `sort cmp S` `|S|log|S|`, `log²|S|`, `equal f (S1, S2)` `min|S1|+|S2|`, `log(…)`, `merge cmp (S1, S2)` `|S1|+|S2|`, `log(…)`, `search cmp x S` `log|S|`