Functional Programming

by Łukasz Stafiniak

Email: `lukstafi@gmail.com`, `lukstafi@ii.uni.wroc.pl`
Web: `www.ii.uni.wroc.pl/~lukstafi`

Lecture 6: Folding and Backtracking

Mapping and folding. Backtracking using lists. Constraint solving.

Martin Odersky "Functional Programming Fundamentals" Lectures 2, 5 and 6

Bits of Ralf Laemmel "Going Bananas"

Graham Hutton "Programming in Haskell" Chapter 11 "Countdown Problem"

Tomasz Wierzbicki "*Honey Islands* Puzzle Solver"

If you see any error on the slides, let me know!

Plan

- `map` and `fold_right`: recursive function examples, abstracting over gets the higher-order functions.
- Reversing list example, tail-recursive variant, `fold_left`.
- Trimming a list: `filter`.
  - Another definition via `fold_right`.
- `map` and `fold` for trees and other data structures.
- The point-free programming style. A bit of history: the FP language.
- Sum over an interval example: $\sum_{n=a}^{b} f(n)$.
- Combining multiple results: `concat_map`.
- Interlude: generating all subsets of a set (as list), and as exercise: all permutations of a list.

- The Google problem: the `map_reduce` higher-order function.
  - Homework reference: modified `map_reduce` to
    1. build a histogram of a list of documents
    2. build an inverted index for a list of documents

    Later: use `fold` (?) to search for a set of words (conjunctive query).
- Puzzles: checking correctness of a solution.
- Combining bags of intermediate results: the `concat_fold` functions.
- From checking to generating solutions.
- Improving "generate-and-test" by filtering (propagating constraints) along the way.
- Constraint variables, splitting and constraint propagation.
- Another example with "heavier" constraint propagation.

Basic generic list operations

How to print a comma-separated list of integers? In module `String`:

```
val concat : string -> string list -> string
```

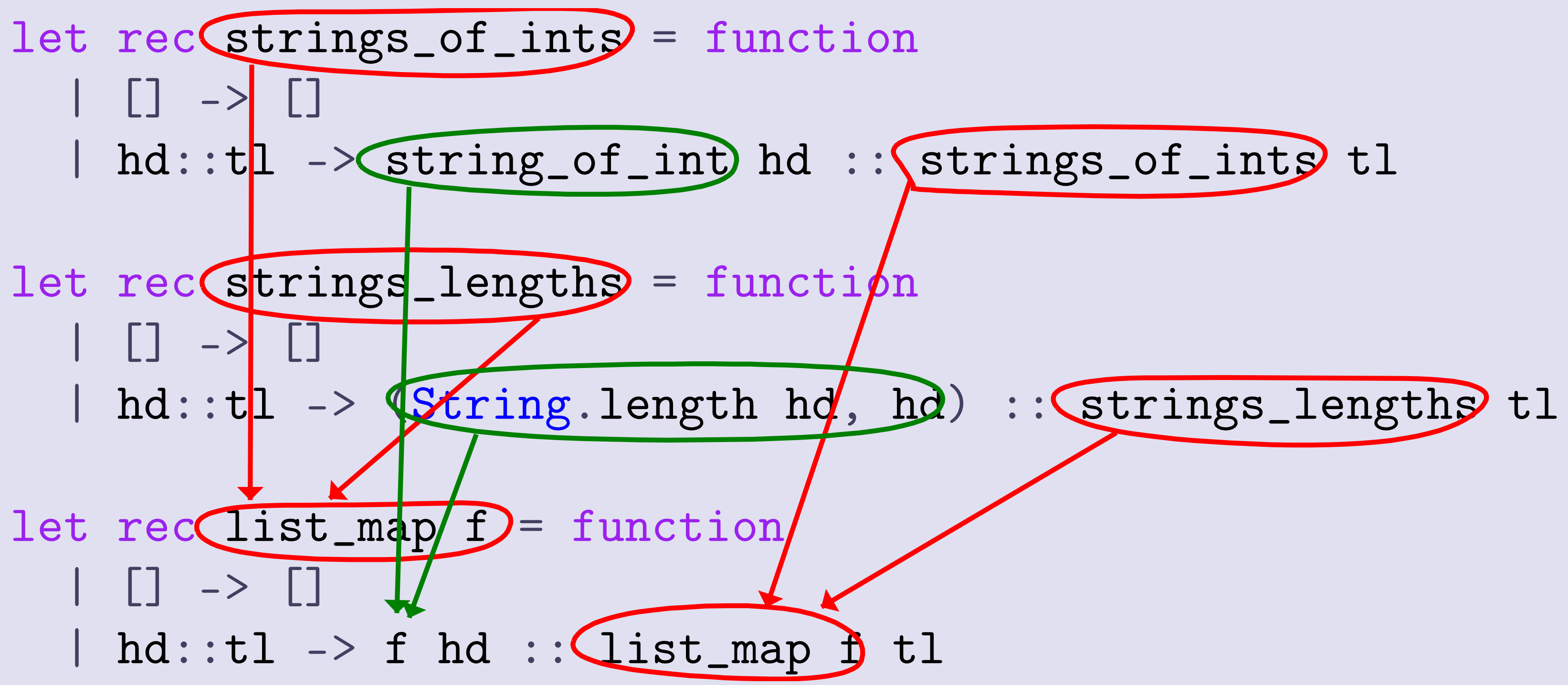
First convert numbers into strings:

```
let rec strings_of_ints = function
  | [] -> []
  | hd::tl -> string_of_int hd :: strings_of_ints tl
let comma_sep_ints = String.concat ", " -| strings_of_ints
```

How to get strings sorted from shortest to longest? First find the length:

```
let rec strings_lengths = function
  | [] -> []
  | hd::tl -> (String.length hd, hd) :: strings_lengths tl
let by_size = List.sort compare -| strings_lengths
```

Always extract common patterns



```
let rec strings_of_ints = function
  | [] -> []
  | hd::tl -> string_of_int hd :: strings_of_ints tl

let rec strings_lengths = function
  | [] -> []
  | hd::tl -> (String.length hd, hd) :: strings_lengths tl

let rec list_map f = function
  | [] -> []
  | hd::tl -> f hd :: list_map f tl
```

Now use the generic function:

```
let comma_sep_ints =
  String.concat ", " -| list_map string_of_int
let by_size =
  List.sort compare -| list_map (fun s->String.length s, s)
```

How to sum elements of a list?

```
let rec balance = function
  | [] -> 0
  | hd::tl -> hd + balance tl
```

How to multiply elements in a list?

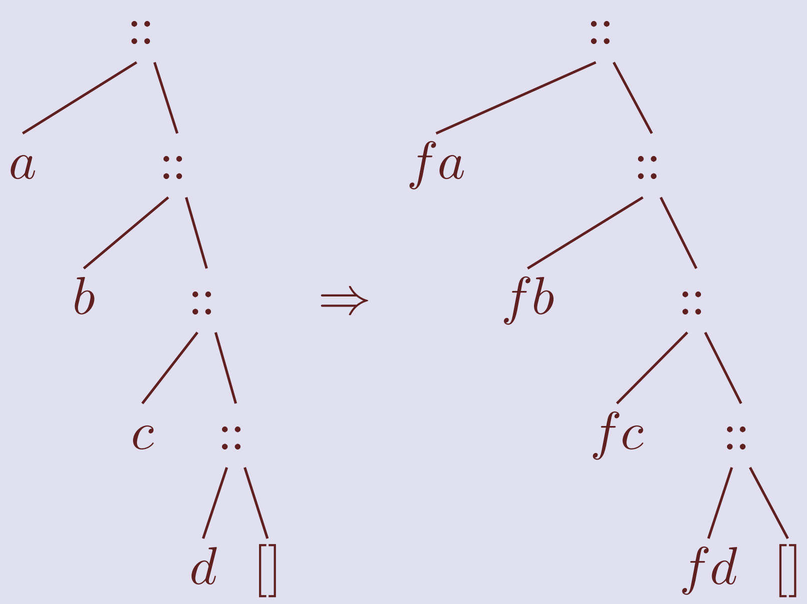
```
let rec total_ratio = function
  | [] -> 1.
  | hd::tl -> hd *. total_ratio tl
```

Generic solution:

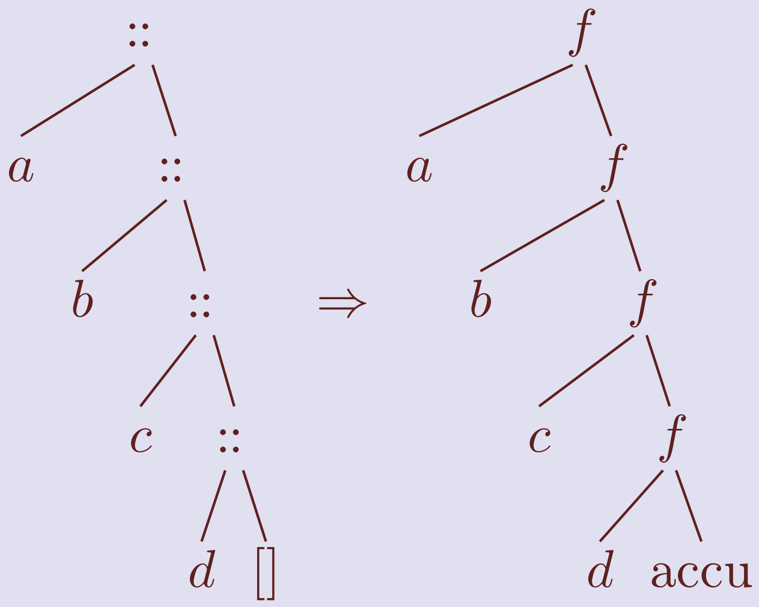
```
let rec list_fold f base = function
  | [] -> base
  | hd::tl -> f hd (list_fold f base tl)
```

Caution: `list_fold f base l` = `List.fold_right f l base`.

`map` alters the contents of data without changing the structure:

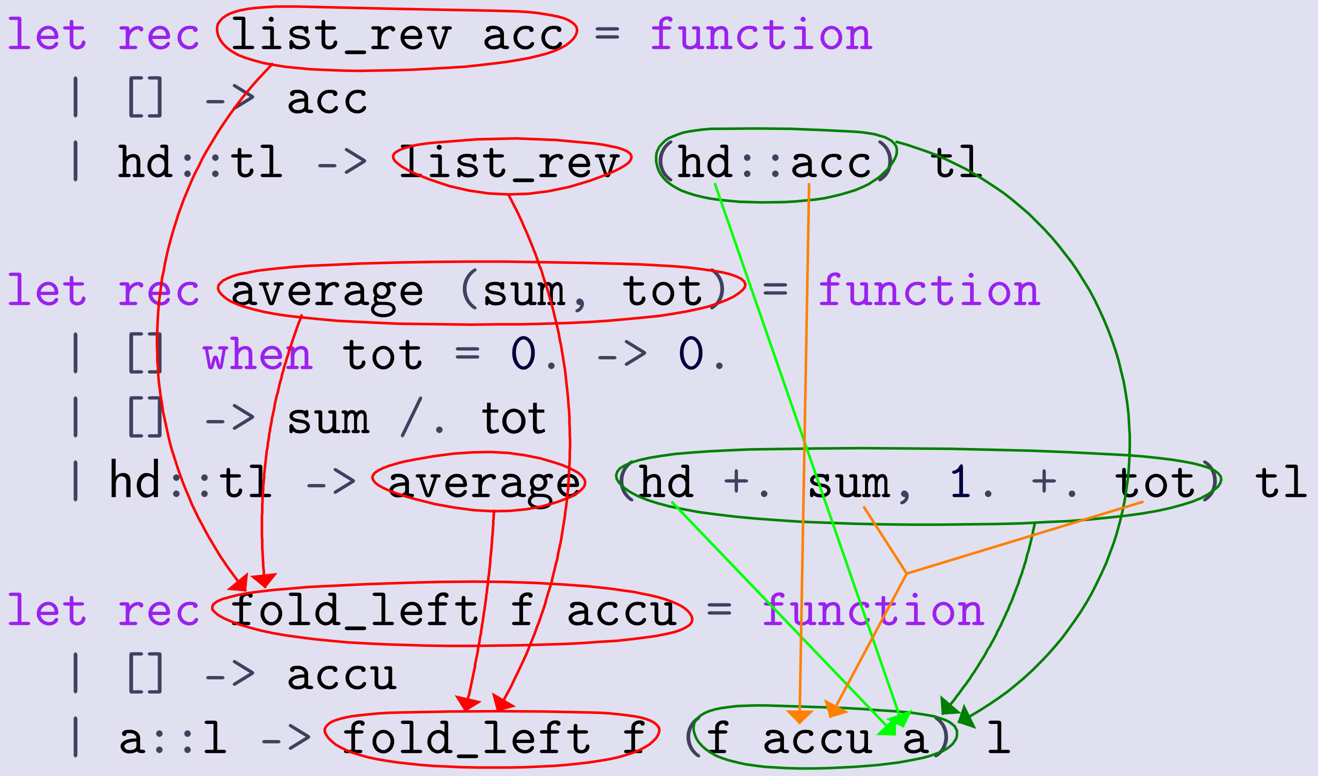


`fold` computes a value using the structure as a scaffolding:



Can we make `fold` tail-recursive?

Let's investigate some tail-recursive functions. (Not hidden as helpers.)



```
let rec list_rev acc = function
  | [] -> acc
  | hd::tl -> list_rev (hd::acc) tl

let rec average (sum, tot) = function
  | [] when tot = 0. -> 0.
  | [] -> sum /. tot
  | hd::tl -> average (hd +. sum, 1. +. tot) tl

let rec fold_left f accu = function
  | [] -> accu
  | a::l -> fold_left f (f accu a) l
```

- With `fold_left`, it is easier to hide the accumulator. The `average` example is a bit more tricky than `list_rev`.

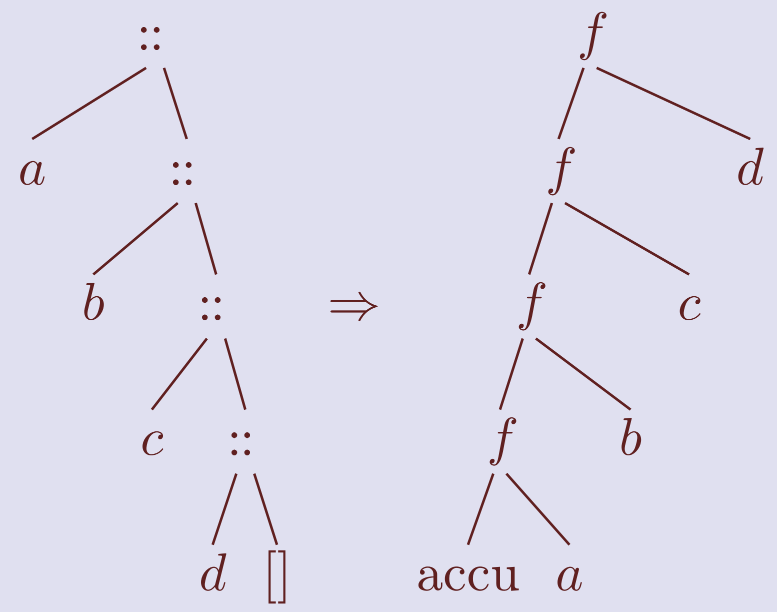
  ```
  let list_rev l =
    fold_left (fun t h->h::t) [] l
  let average =
    fold_left (fun (sum,tot) e->sum +. e, 1. +. tot) (0.,0.)
  ```

- The function names and order of arguments for `List.fold_right` / `List.fold_left` are due to:
  - `fold_right f` makes `f` *right associative*, like list constructor `::`

    `List.fold_right f [a1; ...; an] b` is `f a1 (f a2 (... (f an b) ...))`.
  - `fold_left f` makes `f` *left associative*, like function application

    `List.fold_left f a [b1; ...; bn]` is `f (... (f (f a b1) b2) ...) bn`.

- The "backward" structure of `fold_left` computation:



- List filtering, already rather generic (a polymorphic higher-order function)

```
let list_filter p l =
  List.fold_right (fun h t->if p h then h::t else t) l []
```

- Tail-recursive map returning elements in reverse order:

```
let list_rev_map f l =
  List.fold_left (fun t h->f h::t) [] l
```

`map` and `fold` for trees and other structures

- Mapping binary trees is straightforward:

```
type 'a btree = Empty | Node of 'a * 'a btree * 'a btree

let rec bt_map f = function
  | Empty -> Empty
  | Node (e, l, r) -> Node (f e, bt_map f l, bt_map f r)

let test = Node
  (3, Node (5, Empty, Empty), Node (7, Empty, Empty))
let _ = bt_map ((+) 1) test
```

- `map` and `fold` we consider in this section preserve / respect the structure of the data, they **do not** correspond to `map` and `fold` of *abstract data type* containers, which are like `List.rev_map` and `List.fold_left` over container elements listed in arbitrary order.
  - I.e. here we generalize `List.map` and `List.fold_right` to other structures.

- `fold` in most general form needs to process the element together with partial results for the subtrees.

```
let rec bt_fold f base = function
  | Empty -> base
  | Node (e, l, r) ->
    f e (bt_fold f base l) (bt_fold f base r)
```

- Examples:

```
let sum_els = bt_fold (fun i l r -> i + l + r) 0
let depth t = bt_fold (fun _ l r -> 1 + max l r) 1 t
```

`map` and `fold` for more complex structures

To have a data structure to work with, we recall expressions from lecture 3.

```
type expression =
     Const of float
   | Var of string
   | Sum of expression * expression    (* e1 + e2 *)
   | Diff of expression * expression   (* e1 - e2 *)
   | Prod of expression * expression   (* e1 * e2 *)
   | Quot of expression * expression   (* e1 / e2 *)
```

Multitude of cases make the datatype harder to work with. Fortunately, *or-patterns* help a bit:

```
let rec vars = function
  | Const _ -> []
  | Var x -> [x]
  | Sum (a,b) | Diff (a,b) | Prod (a,b) | Quot (a,b) ->
    vars a @ vars b
```

Mapping and folding needs to be specialized for each case. We pack the behaviors into a record.

```
type expression_map = {
  map_const : float -> expression;
  map_var : string -> expression;
  map_sum : expression -> expression -> expression;
  map_diff : expression -> expression -> expression;
  map_prod : expression -> expression -> expression;
  map_quot : expression -> expression -> expression;
}
```

Note how `expression` from above is substituted by `'a` below, explain why?

```
type 'a expression_fold = {
  fold_const : float -> 'a;
  fold_var : string -> 'a;
  fold_sum : 'a -> 'a -> 'a;
  fold_diff : 'a -> 'a -> 'a;
  fold_prod : 'a -> 'a -> 'a;
  fold_quot : 'a -> 'a -> 'a;
}
```

Next we define standard behaviors for `map` and `fold`, which can be tailored to needs for particular case.

```
let identity_map = {
  map_const = (fun c -> Const c);
  map_var = (fun x -> Var x);
  map_sum = (fun a b -> Sum (a, b));
  map_diff = (fun a b -> Diff (a, b));
  map_prod = (fun a b -> Prod (a, b));
  map_quot = (fun a b -> Quot (a, b));
}

let make_fold op base = {
  fold_const = (fun _ -> base);
  fold_var = (fun _ -> base);
  fold_sum = op; fold_diff = op;
  fold_prod = op; fold_quot = op;
}
```

The actual `map` and `fold` functions are straightforward:

```
let rec expr_map emap = function
  | Const c -> emap.map_const c
  | Var x -> emap.map_var x
  | Sum (a,b) -> emap.map_sum (expr_map emap a) (expr_map emap b)
  | Diff (a,b) -> emap.map_diff (expr_map emap a) (expr_map emap b)
  | Prod (a,b) -> emap.map_prod (expr_map emap a) (expr_map emap b)
  | Quot (a,b) -> emap.map_quot (expr_map emap a) (expr_map emap b)

let rec expr_fold efold = function
  | Const c -> efold.fold_const c
  | Var x -> efold.fold_var x
  | Sum (a,b) -> efold.fold_sum (expr_fold efold a) (expr_fold efold b)
  | Diff (a,b) -> efold.fold_diff (expr_fold efold a) (expr_fold efold b)
  | Prod (a,b) -> efold.fold_prod (expr_fold efold a) (expr_fold efold b)
  | Quot (a,b) -> efold.fold_quot (expr_fold efold a) (expr_fold efold b)
```

Now examples. We use `{record with field=value}` syntax which copies `record` but puts `value` instead of `record.field` in the result.

```
let prime_vars = expr_map
  {identity_map with map_var = fun x -> Var (x^"'")}

let subst s =
  let apply x = try List.assoc x s with Not_found -> Var x in
  expr_map {identity_map with map_var = apply}

let vars =
  expr_fold {(make_fold (@) []) with fold_var = fun x-> [x]}
let size = expr_fold (make_fold (fun a b->1+a+b) 1)

let eval env = expr_fold {
  fold_const = id;
  fold_var = (fun x -> List.assoc x env);
  fold_sum = (+.); fold_diff = (-.);
  fold_prod = ( *.); fold_quot = (/.);
}
```

Point-free Programming

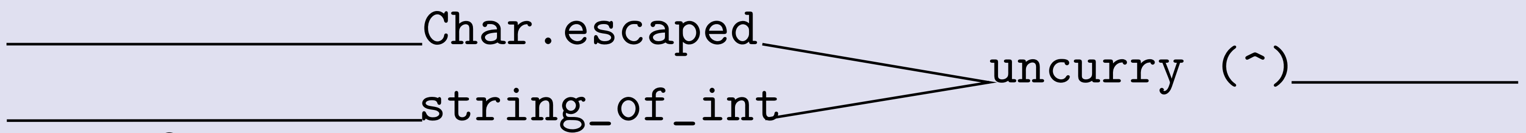
- In 1977/78, John Backus designed **FP**, the first *function-level programming* language. Over the next decade it evolved into the **FL** language.
  - ”Clarity is achieved when programs are written at the function level – that is, by putting together existing programs to form new ones, rather than by manipulating objects and then abstracting from those objects to produce programs.” *The FL Project: The Design of a Functional Language*
- For functionl-level programming style, we need functionals/combinators, like these from *OCaml Batteries*: `let const x _ = x`

```
let ( |- ) f g x = g (f x)
let ( -| ) f g x = f (g x)
let flip f x y = f y x
let ( *** ) f g = fun (x,y) -> (f x, g y)
let ( &&& ) f g = fun x -> (f x, g x)
let first f x = fst (f x)
let second f x = snd (f x)
let curry f x y = f (x,y)
let uncurry f (x,y) = f x y
```

- The flow of computation can be seen as a circuit where the results of nodes-functions are connected to further nodes as inputs.

  We can represent the cross-sections of the circuit as tuples of intermediate values.

-
  ```
  let print2 c i =
    let a = Char.escaped c in
    let b = string_of_int i in
    a ^ b
  ```



```
let print2 = curry
  ((Char.escaped *** string_of_int) |- uncurry (^))
```

- Since we usually work by passing arguments one at a time rather than in tuples, we need `uncurry` to access multi-argument functions, and we pack the result with `curry`.

  ◦ Turning C/Pascal-like function into one that takes arguments one at a time is called *currification*, after the logician Haskell Brooks Curry.

- Another option to remove explicit use of function parameters, rather than to pack intermediate values as tuples, is to use function composition, `flip`, and the so called **S** combinator:

  ```
  let s x y z = x z (y z)
  ```

  to bring a particular argument of a function to "front", and pass it a result of another function. Example: a filter-map function

  ```
  let func2 f g l = List.filter f (List.map g (l))
  ```

  Definition of function composition.

  ```
  let func2 f g = (-|) (List.filter f) (List.map g)
  let func2 f = (-|) (List.filter f) -| List.map
  ```

  Composition again, below without the infix notation.

  ```
  let func2 f = (-|) ((-|) (List.filter f)) List.map
  let func2 f = flip (-|) List.map ((-|) (List.filter f))
  let func2 f = (((|-) List.map) -| ((-|) -| List.filter)) f
  let func2 = (|-) List.map -| ((-|) -| List.filter)
  ```

Reductions. More higher-order/list functions

Mathematics has notation for sum over an interval: $\sum_{n=a}^{b} f(n)$.

In OCaml, we do not have a universal addition operator:

```
let rec i_sum_from_to f a b =
  if a > b then 0
  else f a + i_sum_from_to f (a+1) b
let rec f_sum_from_to f a b =
  if a > b then 0.
  else f a +. f_sum_from_to f (a+1) b
let pi2_over6 =
  f_sum_from_to (fun i->1. /. float_of_int (i*i)) 1 5000
```

It is natural to generalize:

```
let rec op_from_to op base f a b =
  if a > b then base
  else op (f a) (op_from_to op base f (a+1) b)
```

Let's collect the results of a multifunction (i.e. a set-valued function) for a set of arguments, in math notation:

$$f(A) = \bigcup_{p \in A} f(p)$$

It is a useful operation over lists with `union` translated as `append`:

```
let rec concat_map f = function
  | [] -> []
  | a::l -> f a @ concat_map f l
```

and more efficiently:

```
let concat_map f l =
  let rec cmap_f accu = function
    | [] -> accu
    | a::l -> cmap_f (List.rev_append (f a) accu) l in
  List.rev (cmap_f [] l)
```

List manipulation: All subsequences of a list

```
let rec subseqs l =
  match l with
    | [] -> [[]]
    | x::xs ->
      let pxs = subseqs xs in
      List.map (fun px -> x::px) pxs @ pxs
```

Tail-recursively:

```
let rec rmap_append f accu = function
  | [] -> accu
  | a::l -> rmap_append f (f a :: accu) l
```

```
let rec subseqs l =
  match l with
    | [] -> [[]]
    | x::xs ->
      let pxs = subseqs xs in
      rmap_append (fun px -> x::px) pxs pxs
```

In-class work: Return a list of all possible ways of splitting a list into two non-empty parts.

Homework:

Find all permutations of a list.

Find all ways of choosing without repetition from a list.

By key: `group_by` and `map_reduce`

It is often useful to organize values by some property.

First we collect an elements from an association list by key.

```
let collect l =
  match List.sort (fun x y -> compare (fst x) (fst y)) l with
  | [] -> []                                  Start with associations sorted by key.
  | (k0, v0)::tl ->
    let k0, vs, l = List.fold_left
      (fun (k0, vs, l) (kn, vn) ->          Collect values for the current key
        if k0 = kn then k0, vn::vs, l           and when the key changes
       else kn, [vn], (k0,List.rev vs)::l)  stack the collected values.
      (k0, [v0], []) tl in                  What do we gain by reversing?
    List.rev ((k0,List.rev vs)::l)
```

Now we can group by an arbitrary property:

```
let group_by p l = collect (List.map (fun e->p e, e) l)
```

But we want to process the results, like with an *aggregate operation* in SQL. The aggregation operation is called **reduction**.

```
let aggregate_by p red base l =
  let ags = group_by p l in
  List.map (fun (k,vs)->k, List.fold_right red vs base) ags
```

We can use the **feed-forward** operator: `let ( |> ) x f = f x`

```
let aggregate_by p redf base l =
  group_by p l
  |> List.map (fun (k,vs)->k, List.fold_right redf vs base)
```

Often it is easier to extract the property over which we aggregate upfront. Since we first map the elements into the extracted key-value pairs, we call the operation `map_reduce`:

```
let map_reduce mapf redf base l =
  List.map mapf l
  |> collect
  |> List.map (fun (k,vs)->k, List.fold_right redf vs base)
```

`map_reduce/concat_reduce` **examples**

Sometimes we have multiple sources of information rather than records.

```
let concat_reduce mapf redf base l =
  concat_map mapf l
  |> collect
  |> List.map (fun (k,vs)->k, List.fold_right redf vs base)
```

Compute the merged histogram of several documents:

```
let histogram documents =
  let mapf doc =
    Str.split (Str.regexp "[ t.,;]+") doc
  |> List.map (fun word->word,1) in
  concat_reduce mapf (+) 0 documents
```

Now compute the *inverted index* of several documents (which come with identifiers or addresses).

```
let cons hd tl = hd::tl
let inverted_index documents =
  let mapf (addr, doc) =
    Str.split (Str.regexp "[ t.,;]+") doc
  |> List.map (fun word->word,addr) in
  concat_reduce mapf cons [] documents
```

And now... a "search engine":

```
let search index words =
  match List.map (flip List.assoc index) words with
  | [] -> []
  | idx::idcs -> List.fold_left intersect idx idcs
```

where `intersect` computes intersection of sets represented as lists.

Tail-recursive variants

```
let rev_collect l =
  match List.sort (fun x y -> compare (fst x) (fst y)) l with
  | [] -> []
  | (k0, v0)::tl ->
    let k0, vs, l = List.fold_left
      (fun (k0, vs, l) (kn, vn) ->
        if k0 = kn then k0, vn::vs, l
        else kn, [vn], (k0, vs)::l)
      (k0, [v0], []) tl in
    List.rev ((k0, vs)::l)

let tr_concat_reduce mapf redf base l =
  concat_map mapf l
  |> rev_collect
  |> List.rev_map (fun (k,vs)->k, List.fold_left redf base vs)

let rcons tl hd = hd::tl
let inverted_index documents =
  let mapf (addr, doc) = ... in
  tr_concat_reduce mapf rcons [] documents
```

Helper functions for inverted index demonstration

Sets as **sorted** lists.

```
let intersect xs ys =
  let rec aux acc = function
    | [], _ | _, [] -> acc
    | (x::xs' as xs), (y::ys' as ys) ->
      let c = compare x y in
      if c = 0 then aux (x::acc) (xs', ys')
      else if c < 0 then aux acc (xs', ys)
      else aux acc (xs, ys') in
  List.rev (aux [] (xs, ys))
```

The `Scanf` library uses continuation passing.

```
let read_lines file =
  let input = open_in file in
  let rec read lines =
    try Scanf.fscanf input "%[^\r\n]\n"
          (fun x -> read (x :: lines))
    with End_of_file -> lines in
  List.rev (read [])
```

Index elements by their positions.

```
let indexed l =
  Array.of_list l |> Array.mapi (fun i e->i,e)
  |> Array.to_list

let search_engine lines =
  let lines = indexed lines in
  let index = inverted_index lines in
  fun words ->
    let ans = search index words in
    List.map (flip List.assoc lines) ans

let search_bible =
  search_engine (read_lines "./bible-kjv.txt")
let test_result =
  search_bible ["Abraham"; "sons"; "wife"]
```

Higher-order functions for the `option` type

Operate on an optional value:

```
let map_option f = function
  | None -> None
  | Some e -> f e
```

Map an operation over a list and filter-out cases when it does not succeed:

```
let rec map_some f = function
  | [] -> []
  | e::l -> match f e with
    | None -> map_some f l
    | Some r -> r :: map_some f l
```

Tail-recurively:

```
let map_some f l =
  let rec maps_f accu = function
    | [] -> accu
    | a::l -> maps_f (match f a with None -> accu
      | Some r -> r::accu) l in
  List.rev (maps_f [] l)
```

The Countdown Problem Puzzle

- Using a given set of numbers and arithmetic operators $\boxed{+}$, $\boxed{-}$, $\boxed{*}$, $\boxed{/}$, construct an expression with a given value.
- All numbers, including intermediate results, must be positive integers.
- Each of the source numbers can be used at most once when constructing the expression.
- Example:
  - numbers $\boxed{1}$, $\boxed{3}$, $\boxed{7}$, $\boxed{10}$, $\boxed{25}$, $\boxed{50}$
  - target $\boxed{765}$
  - possible solution $\boxed{(25-10) * (50+1)}$
- There are 780 solutions for this example.
- Changing the target to $\boxed{831}$ gives an example that has no solutions.

- Operators:

```
type op = Add | Sub | Mul | Div
```

- Apply an operator:

```
let apply op x y =
  match op with
  | Add -> x + y
  | Sub -> x - y
  | Mul -> x * y
  | Div -> x / y
```

- Decide if the result of applying an operator to two positive integers is another positive integer:

```
let valid op x y =
  match op with
  | Add -> true
  | Sub -> x > y
  | Mul -> true
  | Div -> x mod y = 0
```

- Expressions:

```
type expr = Val of int | App of op * expr * expr
```

- Return the overall value of an expression, provided that it is a positive integer:

```
let rec eval = function
  | Val n -> if n > 0 then Some n else None
  | App (o,l,r) ->
    eval l |> map_option (fun x ->
      eval r |> map_option (fun y ->
      if valid o x y then Some (apply o x y)
      else None))
```

- **Homework:** Return a list of all possible ways of choosing zero or more elements from a list – `choices`.

- Return a list of all the values in an expression:

```
let rec values = function
  | Val n -> [n]
  | App (_,l,r) -> values l @ values r
```

- Decide if an expression is a solution for a given list of source numbers and a target number:

  ```
  let solution e ns n =
    list_diff (values e) ns = [] && is_unique (values e) &&
    eval e = Some n
  ```

Brute force solution

- Return a list of all possible ways of splitting a list into two non-empty parts:

```
let split l =
  let rec aux lhs acc = function
    | [] | [_] -> []
    | [y; z] -> (List.rev (y::lhs), [z])::acc
    | hd::rhs ->
      let lhs = hd::lhs in
      aux lhs ((List.rev lhs, rhs)::acc) rhs in
  aux [] [] l
```

- We introduce an operator to work on multiple sources of data, producing even more data for the next stage of computation:

```
let ( |-> ) x f = concat_map f x
```

- Return a list of all possible expressions whose values are precisely a given list of numbers:

```
let combine l r =
  List.map (fun o->App (o,l,r)) [Add; Sub; Mul; Div]
```

Combine two expressions using each operator.

```
let rec exprs = function
  | [] -> []
  | [n] -> [Val n]
  | ns ->
    split ns |-> (fun (ls,rs) ->
      exprs ls |-> (fun l ->
        exprs rs |-> (fun r ->
          combine l r)))
```

For each split `ls,rs` of numbers,
for each expression `l` over `ls`
and expression `r` over `rs`
produce all `l ? r` expressions.

- Return a list of all possible expressions that solve an instance of the countdown problem:

```
let guard n =
  List.filter (fun e -> eval e = Some n)

let solutions ns n =
  choices ns |-> (fun ns’ ->
    exprs ns’ |> guard n)
```

- Another way to express this:

```
let guard p e =
  if p e then [e] else []

let solutions ns n =
  choices ns |-> (fun ns’ ->
    exprs ns’ |->
      guard (fun e -> eval e = Some n))
```

Fuse the generate phase with the test phase

- We seek to define a function that fuses together the generation and evaluation of expressions:
    - We memorize the value together with the expression – in pairs `(e, eval e)` – so only valid subexpressions are ever generated.

```
let combine' (l,x) (r,y) =
  [Add; Sub; Mul; Div]
  |> List.filter (fun o->valid o x y)
  |> List.map (fun o->App (o,l,r), apply o x y)

let rec results = function
  | [] -> []
  | [n] -> if n > 0 then [Val n, n] else []
  | ns ->
    split ns |-> (fun (ls,rs) ->
      results ls |-> (fun lx ->
        results rs |-> (fun ry ->
          combine' lx ry)))
```

- Once the result is generated its value is already computed, we only check if it equals the target.

```
let solutions' ns n =
  choices ns |-> (fun ns' ->
    results ns' |>
        List.filter (fun (e,m)-> m=n) |>
            List.map fst)
```

We discard the memorized values.

Eliminate symmetric cases

- Strengthening the valid predicate to take account of commutativity and identity properties:

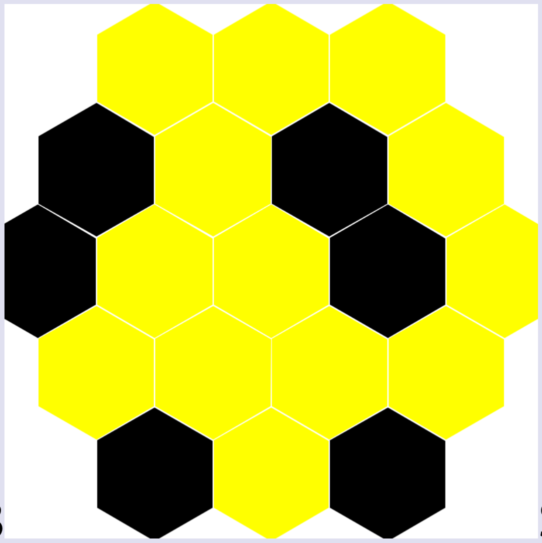
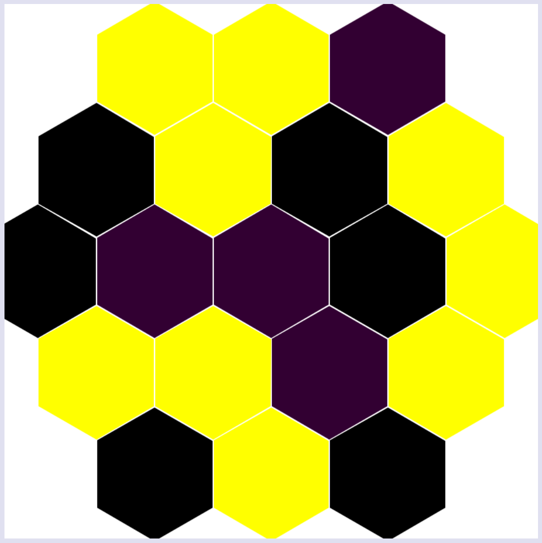
  ```
  let valid op x y =
    match op with
    | Add -> x <= y
    | Sub -> x > y
    | Mul -> x <= y && x <> 1 && y <> 1
    | Div -> x mod y = 0 && y <> 1
  ```

  - We eliminate repeating symmetrical solutions on the semantic level, i.e. on values, rather than on the syntactic level of expressions – it is both easier and gives better results.

- Now recompile `combine'`, `results` and `solutions'`.

The Honey Islands Puzzle

- Be a bee! Find the cells to eat honey out of, so that the least amount of honey becomes sour, assuming that sourness spreads through contact.
    - Honey sourness is totally made up, sorry.
- Each honeycomb cell is connected with 6 other cells, unless it is a border cell. Given a honeycomb with some cells initially marked as black, mark some more cells so that unmarked cells form `num_islands` disconnected components, each with `island_size` cells.

Task: 3 islands x 3  Solution: 

Representing the honeycomb

```
type cell = int * int
module CellSet =
  Set.Make (struct type t = cell let compare = compare end)
type task = {
  board_size : int;
  num_islands : int;
  island_size : int;
  empty_cells : CellSet.t;
}
```

We address cells using “cartesian” coordinates and store them in either lists or sets.

For board “size” N, the honeycomb coordinates range from $(-2N, -N)$ to $2N, N$.

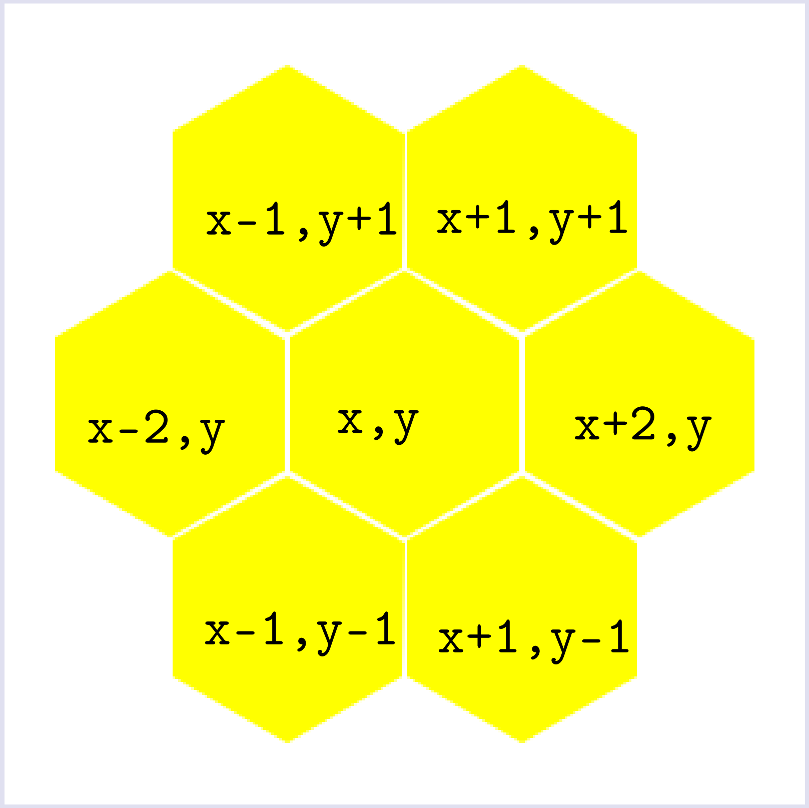
Required number of islands and required number of cells in an island.

The cells that are initially without honey.

```
let cellset_of_list l =
  List.fold_right CellSet.add l CellSet.empty
```

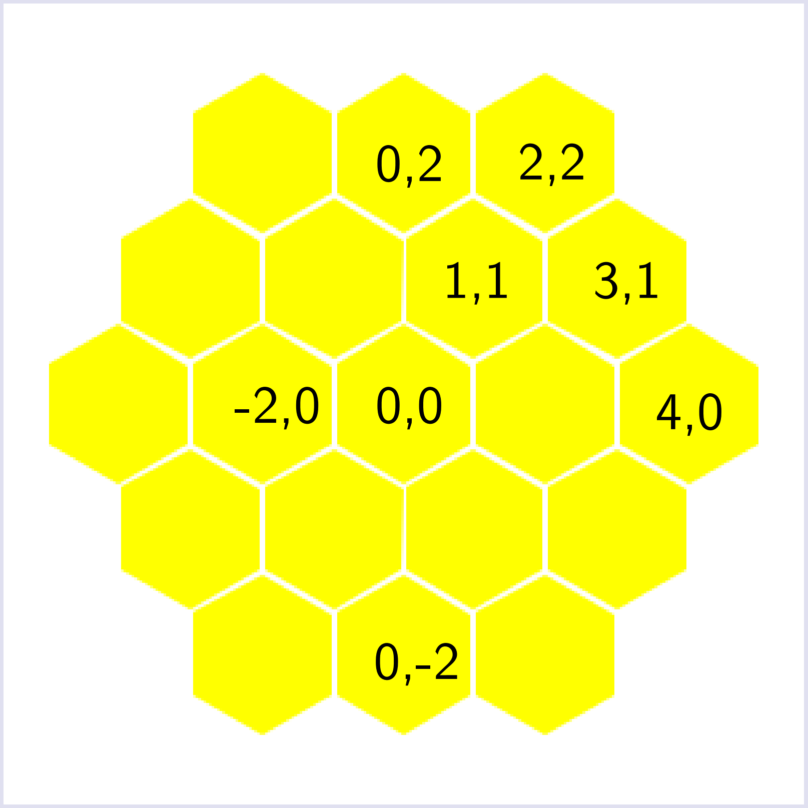
List into set, inverse of `CellSet.elements`

Neighborhood



```
let neighbors n eaten (x,y) =
  List.filter
    (inside_board n eaten)
    [x-1,y-1; x+1,y-1; x+2,y;
     x+1,y+1; x-1,y+1; x-2,y]
```

Building the honeycomb



```
let even x = x mod 2 = 0

let inside_board n eaten (x, y) =
  even x = even y && abs y <= n &&
  abs x + abs y <= 2*n &&
  not (CellSet.mem (x,y) eaten)

let honey_cells n eaten =
  from_to (-2*n) (2*n)|->(fun x ->
    from_to (-n) n |-> (fun y ->
     guard (inside_board n eaten)
        (x, y)))
```

Drawing honeycombs

We separately generate colored polygons:

```
let draw_honeycomb ~w ~h task eaten =
  let i2f = float_of_int in
  let nx = i2f (4 * task.board_size + 2) in
  let ny = i2f (2 * task.board_size + 2) in
  let radius = min (i2f w /. nx) (i2f h /. ny) in
  let x0 = w / 2 in
  let y0 = h / 2 in
  let dx = (sqrt 3. /. 2.) *. radius +. 1. in
  let dy = (3. /. 2.) *. radius +. 2. in
  let draw_cell (x,y) =
    Array.init 7
      (fun i ->
        let phi = float_of_int i *. pi /. 3. in
        x0 + int_of_float (radius *. sin phi +. float_of_int x *. dx),
        y0 + int_of_float (radius *. cos phi +. float_of_int y *. dy)) in
```

The distance between (x, y) and $(x+1, y+1)$.

We draw a closed polygon by placing 6 points evenly spaced on a circumcircle.

```
let honey =
  honey_cells task.board_size (CellSet.union task.empty_cells
                                (cellset_of_list eaten))
  |> List.map (fun p->draw_cell p, (255, 255, 0)) in
let eaten = List.map
    (fun p->draw_cell p, (50, 0, 50)) eaten in
let old_empty = List.map
    (fun p->draw_cell p, (0, 0, 0))
    (CellSet.elements task.empty_cells) in
honey @ eaten @ old_empty
```

We can draw the polygons to an *SVG* image:

```
let draw_to_svg file ~w ~h ?title ?desc curves =
  let f = open_out file in
  Printf.fprintf f "<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="%d" height="%d" viewBox="0 0 %d %d"
     xmlns="http://www.w3.org/2000/svg" version="1.1">
" w h w h;
  (match title with None -> ()
  | Some title -> Printf.fprintf f "  <title>%s</title>n" title);
  (match desc with None -> ()
  | Some desc -> Printf.fprintf f "  <desc>%s</desc>n" desc);
  let draw_shape (points, (r,g,b)) =
    uncurry (Printf.fprintf f "  <path d="M %d %d") points.(0);
    Array.iteri (fun i (x, y) ->
      if i > 0 then Printf.fprintf f " L %d %d" x y) points;
    Printf.fprintf f
      ""n        fill="rgb(%d, %d, %d)" stroke-width="3" />n"
      r g b in
  List.iter draw_shape curves;
  Printf.fprintf f "</svg>%!"
```

But we also want to draw on a screen window – we need to link the `Graphics` library. In the interactive toplevel:

```
#load "graphics.cma";;
```

When compiling we just provide `graphics.cma` to the command.

```
let draw_to_screen ~w ~h curves =
  Graphics.open_graph (" "^string_of_int w^"x"^string_of_int h);
  Graphics.set_color (Graphics.rgb 50 50 0);
  Graphics.fill_rect 0 0 (Graphics.size_x ()) (Graphics.size_y ());
  List.iter (fun (points, (r,g,b)) ->
    Graphics.set_color (Graphics.rgb r g b);
    Graphics.fill_poly points) curves;
  if Graphics.read_key () = 'q'
 then failwith "User interrupted finding solutions.";
  Graphics.close_graph ()
```

We draw a brown background.

We wait so that solutions can be seen as they're computed.

Testing correctness of a solution

We walk through each island counting its cells, depth-first: having visited everything possible in one direction, we check whether something remains in another direction.

Correctness means there are `num_islands` components each with `island_size` cells. We start by computing the cells to walk on: `honey`.

```
let check_correct n island_size num_islands empty_cells =
  let honey = honey_cells n empty_cells in
```

We keep track of already visited cells and islands. When an unvisited cell is there after walking around an island, it must belong to a different island.

```
let rec check_board been_islands unvisited visited =
  match unvisited with
  | [] -> been_islands = num_islands
  | cell::remaining when CellSet.mem cell visited ->
      check_board been_islands remaining visited    Keep looking.
  | cell::remaining (* when not visited *) ->
      let (been_size, unvisited, visited) =
        check_island cell                           Visit another island.
          (1, remaining, CellSet.add cell visited) in
      been_size = island_size
      && check_board (been_islands+1) unvisited visited
```

When walking over an island, besides the `unvisited` and `visited` cells, we need to remember `been_size` – number of cells in the island visited so far.

```
and check_island current state =
  neighbors n empty_cells current
  |> List.fold_left
    (fun (been_size, unvisited, visited as state)
      neighbor ->
      if CellSet.mem neighbor visited then state
      else
        let unvisited = remove neighbor unvisited in
        let visited = CellSet.add neighbor visited in
        let been_size = been_size + 1 in
        check_island neighbor
          (been_size, unvisited, visited))
    state in
```

Walk into each direction and accumulate visits.

Start from the current overall state (initial `been_size` is 1).

Initially there are no islands already visited.

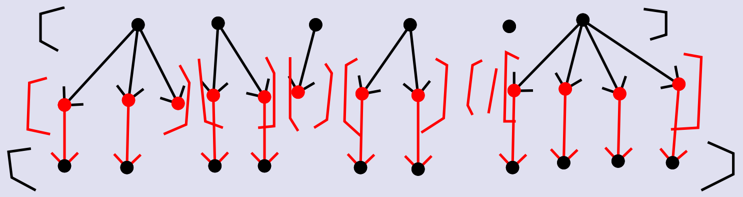
```
check_board 0 honey empty_cells
```

Interlude: multiple results per step

When there is only one possible result per step, we work through a list using `List.fold_right` and `List.fold_left` functions.

What if there are multiple results? Recall that when we have multiple sources of data and want to collect multiple results, we use `concat_map`:

```
concat_map
  f xs =
```



```
List.map f xs
 |> List.concat
```

We shortened `concat_map` calls using "`work |-> (fun a_result -> ...)`" scheme. Here we need to collect results once per step.

```
let rec concat_fold f a = function
  | [] -> [a]
  | x::xs ->
    f x a |-> (fun a' -> concat_fold f a' xs)
```

Generating a solution

We turn the code for testing a solution into one that generates a correct solution.

- We pass around the current solution `eaten`.
- The results will be in a list.
- Empty list means that in a particular case there are no (further) results.
- When walking an island, we pick a new neighbor and try eating from it in one set of possible solutions – which ends walking in its direction, and walking through it in another set of possible solutions.
  - When testing a solution, we never decided to eat from a cell.

The generating function has the same signature as the testing function:

```
let find_to_eat n island_size num_islands empty_cells =
  let honey = honey_cells n empty_cells in
```

Since we return lists of solutions, if we are done with current solution `eaten` we return `[eaten]`, and if we are in a "dead corner" we return `[]`.

```
let rec find_board been_islands unvisited visited eaten =
  match unvisited with
  | [] ->
    if been_islands = num_islands then [eaten] else []
  | cell::remaining when CellSet.mem cell visited ->
    find_board been_islands
      remaining visited eaten
  | cell::remaining (* when not visited *) ->
    find_island cell
      (1, remaining, CellSet.add cell visited, eaten)
    |->         Concatenate solutions for each way of eating cells around and island.
    (fun (been_size, unvisited, visited, eaten) ->
      if been_size = island_size
      then find_board (been_islands+1)
             unvisited visited eaten
      else [])
```

We step into each neighbor of a current cell of the island, and either eat it or walk further.

```
and find_island current state =
  neighbors n empty_cells current
  |> concat_fold
      (fun neighbor
        (been_size, unvisited, visited, eaten as state) ->
        if CellSet.mem neighbor visited then [state]
        else
          let unvisited = remove neighbor unvisited in
          let visited = CellSet.add neighbor visited in
          (been_size, unvisited, visited,
           neighbor::eaten)::
            (* solutions where neighbor is honey *)
          find_island neighbor
            (been_size+1, unvisited, visited, eaten))
      state in
```

Instead of `fold_left` since multiple results.

The initial partial solution is – nothing eaten yet.

```
check_board 0 honey empty_cells []
```

We can test it now:

```
let w = 800 and h = 800
let ans0 = find_to_eat test_task0.board_size test_task0.island_size
  test_task0.num_islands test_task0.empty_cells
let _ = draw_to_screen ~w ~h
  (draw_honeycomb ~w ~h test_task0 (List.hd ans0))
```

But in a more complex case, finding all solutions takes too long:

```
let ans1 = find_to_eat test_task1.board_size test_task1.island_size
  test_task1.num_islands test_task1.empty_cells
let _ = draw_to_screen ~w ~h
  (draw_honeycomb ~w ~h test_task1 (List.hd ans1))
```

(See `Lec6.ml` for definitions of test cases.)

Optimizations for *Honey Islands*

- Main rule: **fail** (drop solution candidates) **as early as possible**.
    - Is the number of solutions generated by the more brute-force approach above 2^n for n honey cells, or smaller?
- We will guard both choices (eating a cell and keeping it in island).
- We know exactly how much honey needs to be eaten.
- Since the state has many fields, we define a record for it.

```
type state = {
  been_size: int;                    Number of honey cells in current island.
  been_islands: int;                       Number of islands visited so far.
  unvisited: cell list;                        Cells that need to be visited.
  visited: CellSet.t;                                       Already visited.
  eaten: cell list;                               Current solution candidate.
  more_to_eat: int;          Remaining cells to eat for a complete solution.
}
```

We define the basic operations on the state up-front. If you could keep them inlined, the code would remain more similar to the previous version.

```
let rec visit_cell s =
  match s.unvisited with
  | [] -> None
  | c::remaining when CellSet.mem c s.visited ->
    visit_cell {s with unvisited=remaining}
  | c::remaining (* when c not visited *) ->
    Some (c, {s with
      unvisited=remaining;
      visited = CellSet.add c s.visited})

let eat_cell c s =
  {s with eaten = c::s.eaten;
    visited = CellSet.add c s.visited;
    more_to_eat = s.more_to_eat - 1}

let keep_cell c s =
  {s with been_size = s.been_size + 1;
    visited = CellSet.add c s.visited}
```

Actually c is not used...

We increase been_size at the start of find_island rather than before calling it.

```
let fresh_island s =
  {s with been_size = 0;
    been_islands = s.been_islands + 1}

let init_state unvisited more_to_eat = {
  been_size =                                0;
  been_islands = 0;
  unvisited; visited = CellSet.empty;
  eaten = []; more_to_eat;
}
```

We need a state to begin with:

```
let init_state unvisited more_to_eat = {
  been_size = 0; been_islands = 0;
  unvisited; visited = CellSet.empty;
  eaten = []; more_to_eat;
}
```

The “main loop” only changes because of the different handling of state.

```
  let rec find_board s =
    match visit_cell s with
    | None ->
      if s.been_islands = num_islands then [eaten] else []
    | Some (cell, s) ->
      find_island cell (fresh_island s)
      |-> (fun s ->
        if s.been_size = s.island_size
        then find_board s
        else [])
```

In the “island loop” we only try actions that make sense:

```
and find_island current s =
  let s = keep_cell current s in
  neighbors n empty_cells current
  |> concat_fold
      (fun neighbor s ->
        if CellSet.mem neighbor s.visited then [s]
        else
          let choose_eat =
            if s.more_to_eat = 0 then []
            else [eat_cell neighbor s]
          and choose_keep =
            if s.been_size >= island_size then []
            else find_island neighbor s in
          choose_eat @ choose_keep)
      s in
```

Guard against actions that would fail.

Finally, we compute the required length of `eaten` and start searching.

```
let cells_to_eat =
  List.length honey - island_size * num_islands in
find_board (init_state honey cells_to_eat)
```

Constraint-based puzzles

- Puzzles can be presented by providing the general form of solutions, and additional requirements that the solutions must meet.
- For many puzzles, the general form of solutions for a given problem can be decomposed into a fixed number of variables.
  - A domain of a variable is a set of possible values the variable can have in any solution.
  - In the *Honey Islands* puzzle, the variables correspond to cells and the domains are $\{\text{Honey}, \text{Empty}\}$ (either a cell has honey, or is empty – without distinguishing "initially empty" and "eaten").
  - In the *Honey Islands* puzzle, the constraints are: a selection of cells that have to be empty, the number and size of connected components of cells that are not empty. The neighborhood graph – which cell-variable is connected with which – is part of the constraints.

- There is a general and often efficient scheme of solving constraint-based problems. **Finite Domain Constraint Programming** algorithm:
  1. With each variable, associate a set of values, initially equal to the domain of the variable. The singleton containing the association is the initial set of partial solutions.
  2. While there is a solution with more than one value associated to some variable in the set of partial solutions, select it and:
     a. If there is a possible value for some variable, such that for all possible assignments of values to other variables, the requirements fail, remove this value from the set associated with this variable.
     b. If there is a variable with empty set of possible values associated to it, remove the solution from the set of partial solutions.
     c. Select the variable with the smallest non-singleton set associated with it (i.e. the smallest greater than 2 size). Split that set into similarly-sized parts. Replace the solution with two solutions where the variable is associated with either of the two parts.
  3. The final solutions are built from partial solutions by assigning to a variable the single possible value associated with it.

- This general algorithm can be simplified. For example, in step (2.c), instead of splitting into two equal-sized parts, we can partition into a singleton and remainder, or partition “all the way” into several singletons.
- The above definition of *finite domain constraint solving* algorithm is sketchy. Questions?
- We will not discuss a complete implementation example, but you can exploit ideas from the algorithm in your homework.