Notes for CSC 2/454, Feb. 27 and Mar. 4, 2025

==========

Functional programming

Functional languages such as Lisp/Scheme and ML/Haskell/OCaml/F# are an attempt to realize Church's lambda calculus in practical form as a programming language.

The key idea: do everything by composing functions.
No mutable state; no side effects.

So how do you get anything done?

Recursion

Takes the place of iteration.

Some tasks are “naturally” recursive. Consider for example the function

$$\text{gcd}(a, b) = \begin{cases} a & \text{if } a = b \\ \text{gcd}(a-b, b) & \text{if } a > b \\ \text{gcd}(a, b-a) & \text{if } b > a \end{cases}$$

(Euclid's algorithm).

We might write this in C as

```
int gcd(int a, int b) {
   /* assume a, b > 0 */
   if (a == b) return a;
   else if (a > b) return gcd(a-b, b);
   else return gcd(a, b-a);
}
```

Other tasks we're used to thinking of as naturally iterative:

```
typedef int (*int_func) (int);
   int summation(int_func f, int low, int high) {
      /* assume low <= high */
      int total = 0;
      int i;
      for (i = low; i <= high; i++) {
         total += f(i);
      }
      return total;
   }
```

$$\sum_{low \le i \le high} f(i)$$

But there's nothing sacred about this “natural” intuition.
Consider:

```
   int gcd(int a, int b) {
      /* assume a, b > 0 */
      while (a != b) {
         if (a > b) a = a-b;
         else b = b-a;
      }
      return a;
   }

   typedef int (*int_func) (int);
   int summation(int_func f, int low, int high) {
      /* assume low <= high */
      if (low == high) return f(low);
      else return f(low) + summation(f, low+1, high);
   }
```

More significantly, the recursive solution doesn't have to be any more expensive than the iterative solution. In OCaml, the gcd function would be written

```
   let rec gcd a b =
     if a = b then a
     else if a > b then gcd (a - b) b
     else gcd a (b - a);;
```

Things to notice in this code:

- top-level forms, let
- rec
- necessity of else
- application via justaposition, use of parentheses

double semicolons (tells REPL (read-eval-print loop) that you're done and it should interpret)

Note that the recursive call is the ***last*** thing `gcd` does — no further computation after the return. This is called ***tail recursion***. Functional language compilers will translate this as, roughly:

```
gcd(a, b) {
top:
   if a == b return a
   elsif a > b
      a := a – b
      goto top
   else
      b := b – a
      goto top
}
```

Functional programmers get good at writing functions that are naturally tail recursive. For example, instead of

```
let rec sum1 f low high =
  if low = high then f low
  else (f low) + (sum1 f (low + 1) high);;
```

we could write

```
let rec sum2 f low high st =
  if low = high then st + (f low)
  else sum2 f (low + 1) high (st + (f low));;
```

Here ‘`st`’ is a subtotal that accumulates what we've added up so far.

Things to notice in this code:

Function application groups more tightly than addition.

We could have left off the parentheses around “`f low`”.

In general, “normal” functions group left to right; operators have precedence.

Unfortunately, now we have to provide an extra zero parameter to the initial call:

```
# sum1 (fun x -> x*x) 1 10;;
- : int = 385

# sum2 (fun x -> x*x) 1 10 0;;
- : int = 385
```

Things to notice in this code:

`fun` is a ***lambda expression*** — a function definition

To get rid of that extra parameter, we can wrap it:

```
let sum3 f low high =
  let rec helper low st =
   let new_st = st + (f low) in
   if low = high then new_st
   else helper (low + 1) new_st in
  helper low 0;;
```

Things to notice in this code:

internal let

lexical nesting

lack of rec on declaration of sum3

(compiler wouldn't have complained; just unnecessary)

NB: This tail recursive code exploits the associativity of addition; a compiler is unlikely to do that for us automatically. There exist automatic mechanisms to turn non-tail-recursive functions into tail-recursive ones, using what's known as ***continuation passing style***, but that wouldn't be as efficient in this case.

Sometimes you'll hear someone argue that recursion is algorithmically inferior to iteration. Fibonacci numbers are sometimes given as an example:

```
let rec fib1 n =
  match n with
  | 0 -> 1
  | 1 -> 1
  | _ -> fib1 (n-1) + fib1 (n-2);;
```

This takes $O(2^n)$ time, where O(n) is possible. In a von Neumann language we are taught to write

```
int fib(int n) {
   int f1 = 1; int f2 = 1;
   int i;
   for (i = 2; i <= n; i++) {
      int temp = f1 + f2;
      f1 = f2; f2 = temp;
   }
   return f2;
}
```

But there's no reason why we have to do it the slow way in OCaml.
We can write the following instead:

```
let fib2 n =
  let rec helper f1 f2 i =
   if i = n then f2
   else helper f2 (f1 + f2) (i + 1) in
  helper 0 1 0;;
```

Thinking about recursion as a direct, mechanical replacement for iteration is the wrong way to look at things. One has to get used to thinking in a recursive style.

NB: One can actually do better than O(n) for Fibonacci numbers. In particular, F(n) is the nearest whole number to $\varphi^n/\text{sqrt}(5)$, where $\varphi = (1 + \text{sqrt}(5))/2$, but this has high constant-factor costs and problems with numeric precision. For modest n, the O(n) algorithm is perfectly respectable.

NB2: OCaml has imperative features, so we ***can*** write the iterative version. It runs against the grain of the language, however (like writing C-like code in C++, only worse), and you won't be allowed to do it in this course.

NB3: Recursion isn't enough by itself to create a really useful functional language. You also need of ***higher-order functions*** (functional forms). More on this later.

A more complete list of necessary features for functional programming, many of which are missing in some imperative languages:

recursion
1st class and high-order functions (including unlimited extent)
serious polymorphism
powerful list facilities
fully general aggregates
structured function returns
garbage collection

Lisp also has
homoiconography
self-definition
read-eval-print

ML/Haskell/F# have
Milner type inference
pattern matching
implicit currying
syntactic sugar: list comprehensions, monads

These are not necessarily present in other functional langs.

There are lots of functional programming languages. Lisp and ML are the roots of the two main trees.

Lisp
- dates from about 1960.
- originally developed by John McCarthy, who received the Turing Award in 1971.
- inspired by the lambda calculus, Alonzo Church's mathematical formulation of the notion of computation (which you may have seen a bit of in 173).
- two most widely used dialects today are Common Lisp (big, full-featured) and Scheme/Racket (smaller and more elegant, but getting bigger).

ML
- dates from the mid-to-late 1970s.
- originally developed by Robin Milner, who received the Turning Award in 1991.
- intended to be safer and more readable than Lisp
- two most widely used dialects today are SML and OCaml.
  Many academics consider SML more elegant, but OCaml is more "practical" -- it has a better toolchain and is widely used in industry.

- Microsoft's F# is an OCaml descendant.
- Haskell is an ML descendant (through Miranda); it's the leading language for research in functional programming, and is increasingly popular in industry as well. Haskell is distinguished by being ***purely*** functional (no imperative features at all) and by using lazy (normal-order) evaluation.

Advantages of functional languages:
- lack of side effects makes programs easier to understand
- lack of explicit evaluation order (in some languages) offers possibility of parallel evaluation (e.g. MultiLisp)
- lack of side effects and explicit evaluation order simplifies some things for a compiler (provided you don't blow it in other ways)
- programs are often surprisingly short
- language can be small yet very "powerful"

Challenges:
- difficult (but not impossible!) to implement efficiently on von Neumann machines
  - lots of copying of data through parameters
  - (apparent) need to create a whole new array in order to change one element
  - very heavy use of references (space and time and locality problem)
  - frequent subroutine calls
  - heavy space use for (non-tail) recursion
    - but anything you can write with a loop in an imperative language is straightforward to write as tail recursion
  - requires garbage collection

- difficult to integrate I/O into purely functional model
  leading approach is the monads of Haskell — sort of an imperative wrapper around a purely functional program; allows functions to be used not only to calculate values, but also to decide on the order in which imperative actions should be performed.

Requires a different mode of thinking by the programmer.

==========

Introduction to OCaml

ML dialect developed and maintained by researchers at INRIA,
the French national CS research institute
compiler (ocamlc) or interpreter (ocaml) — your choice

Interpreter runs a read-eval-print loop (REPL) much like Scheme or Python.

```
#use "file.ml";;          load source code
#load "library.cma";;     load binary library
```

(That's ocaml syntax; if you're using utop, say `#require "library";;`)

simple data types

`bool`, `int`, `float`, strings, tuples (pairs &c), lists

+, ∗, etc. vs `+.`, `∗.`, etc.

`float` constants must contain a decimal point

^ (string concatenation)

`fst` & `snd`

work ***only*** on two-element tuples (else type error)

hd, `tl` (deprecated: prefer pattern matching)

`::` and `@` (cons and append)

Lists are delimited with square brackets; elements are separated by semicolons.
Tuples are delimited with parentheses; elements are separated by commas.
Records (more later) are delimited with braces; elements are separated by semicolons.
Arrays are delimited with [| and |]; elements are separated by semicolons.

"structural" (same value; aka "deep") vs
"physical" (same instance; aka "shallow") equality

=, <> structural

```
2 = 2; "foo" = "foo"; [1;2;3] = [1;1+1;5−2]
```

==, != physical

```
2 == 2; "foo" != "foo"; [1;2;3] != [1;1+1;5−2]
```

Orderings (<, >, <=, >=) are defined on all non-function types. They do what you'd expect on arithmetic types, Booleans, characters, strings, and tuples, but may not make much sense on others.

type inference -- more on this in Chapter 7

- Type declarations are optional.
- Compiler ***infers*** types when declarations are omitted.
- Type checking amounts to checking for consistent usage.
  - Can't treat something as a string in one place and a number or a list somewhere else.

lexical conventions

- identifiers made from a–zA–Z0–9_'
  - must start with a letter or underscore
  - constructors, variant names, modules, and exceptions have to start with an upper case letter
  - everything else starts with a lower case letter or underscore

```
(* (* comments *) nest *)
```

Top-level forms terminated by **;;**

- This tells the REPL to interpret.

functions

```
let f a1 a2 a3 = ...
let f (a1:t1) (a2:t2) (a3:t3) : rt = ...
let f: t1 -> t2 -> t3 -> rt = fun a1 a2 a3 -> ...
```

Those three versions are equivalent, though the first is implicitly typed.

```
let rec f = ...

let rec g = ...    (* for mutually
and h = ...             recursive functions *)
```

pattern matching

```
match expr with
  var1 -> expr1
| var2 when pred2 -> expr2
```

```
| ...
| _ -> exprN
```

As in Rust, match is sort of like `case` or `switch` on steroids. Also works in other contexts— e.g., "`let (s, t, f) = my_tuple;;`" or function definitions: the (bad) Fibonacci example above

```
let rec fib1 n =
  match n with
  | 0 -> 1
  | 1 -> 1
  | _ -> fib1 (n-1) + fib1 (n-2);;
```

can be rewritten

```
let rec fib1 = function
  | 0 -> 1
  | 1 -> 1
  | n -> fib1 (n-1) + fib1 (n-2);;
```

arrays

```
let primes5 = [| 2; 3; 5; 7; 11 |];;
```

`.()` subscripting

```
primes5.(2)        => 5
```

elements are mutable (unlike those of lists and tuples)

assignment uses left arrow:

```
primes5.(2) <- 12345;;      => ()
```

strings

like arrays of characters, but with double-quoted literals.

Were mutable in older versions of the language. That's now deprecated.

If you need mutability, use **`bytes`** instead.

records

like tuples, but with fields that are named instead of positional

can declare fields to be mutable (immutable by default)

```
type widget = {name: string; sn: int; mutable price: float};;
let g = {name = "gear"; sn = 12345; price = 23.45};;
   g.name => "gear"
g.price <- 34.56;;    (* inflation *)
```

variants

```
type 'a bin_tree = Empty
                 | Node of 'a * 'a bin_tree * 'a bin_tree;;
...
let rec inorder = function
| Empty -> []
| Node(v, lft, rht) -> inorder lft @ [v] @ inorder rht;;
```

side effects

<– (mutable) record field assignment (not allowed in project)

:= and ! refs (like pointers; also not allowed in project)

I/O

```
read_line, read_int, read_float
print_int, print_float,
   print_char, print_string, print_newline,
prerr_int, prerr_float, prerr_string, prerr_newline
```

Printf module

```
printf
sprintf
```

Sys.argv

exceptions

```
exception Foo of string;;
raise (Foo "ouch")
try expr1 with Foo(s) -> expr2
```

Extended example from the text: simulation of a DFA.

We'll invoke the program by calling a function called 'simulate', passing it a DFA description and an input string.
The automaton description is a record with three fields: the start state, the transition function, and a list of the one or more final states. We can trivially make it polymorphic in the type of input symbols:

```
type state = int;;
type 'a dfa = {
  current_state : state;
  transition_function : (state * 'a * state) list;
  final_states : state list;
};;
```

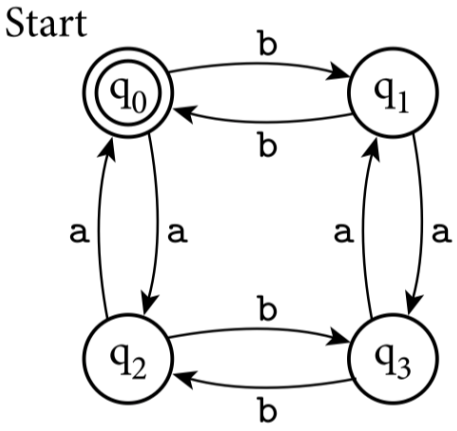
```
type decision = Accept | Reject;;
```

We've named the first field "`current_state`" instead of "`start_state`" for reasons that will become apparent in a minute.

The transition function is represented by a list of triples. The first element and third elements of each triple are the from and to states; the second element is the input symbol that transitions between them.

For example, consider the DFA

```
let a_b_even_dfa : char dfa =  (* input symbols are chars *)
  { current_state = 0;
   transition_function =
     [ (0, 'a', 2); (0, 'b', 1); (1, 'a', 3); (1, 'b', 0);
      (2, 'a', 0); (2, 'b', 3); (3, 'a', 1); (3, 'b', 2) ];
   final_states = [0];
  };;
```



This machine accepts strings containing an even number of a's and an even number of b's.

If we type

```
simulate a_b_even_dfa
         ['a'; 'b'; 'b'; 'a'; 'b'];;
```

then the OCaml interpreter (read-eval-print loop) will print

```
- : state list * decision = ([0; 2; 3; 2; 0; 1], Reject)
```

If we change the input string to abaaba it will print

```
- : state list * decision = ([0; 2; 3; 1; 3; 2; 0], Accept)
```

Here is the program:

```
open List;;     (* includes rev, find, and mem functions *)

let move (d:'a dfa) (x:'a) : 'a dfa =
  { current_state = (
      let (_, _, q) =
        find (fun (s, c, _) -> s = d.current_state && c = x)
             d.transition_function in
      q);
    transition_function = d.transition_function;
    final_states = d.final_states;
  };;

let simulate (d:'a dfa) (input:'a list)
   : (state list * decision) =
  let rec helper moves d2 remaining_input
     : (state option * state list) =
    match remaining_input with
    | [] -> (Some d2.current_state, moves)
    | hd :: tl ->
        let new_moves = d2.current_state :: moves in
          try helper new_moves (move d2 hd) tl
          with Not_found -> (None, new_moves) in
  match helper [] d input with
  | (None, moves) -> (rev moves, Reject)
  | (Some last_state, moves) ->
     ( rev (last_state :: moves),
      if mem last_state d.final_states
        then Accept else Reject);;
```

The basic idea is this: simulate takes a DFA and an input string as argument. If the input string is empty, it checks to see if the start state of the DFA is a final state. If the input string is not empty, simulate calls itself recursively, passing a one-symbol-shorter input string and a DFA that has been modified to have a different start state, namely the one that the old DFA moved to when given the initial input symbol.

==========

Higher-order functions (aka “functional forms”)

Take a function as argument, or return a function as a result.

We saw some examples in previous lectures. In the DFA simulation program, for example, the `find` function took a predicate as its first argument.

```
find (fun x -> x*x > 100) [7; 9; 11; 13]  => 11
```

If you ask OCaml to print the type of `find` it will say

```
('a -> bool) -> 'a list -> 'a
```

That is, `find`’s first argument is a function from `'a` to `bool` (i.e., a predicate), its second argument is a list of `'a` objects, and its return value is the first object in the list for which the predicate returns true.

Other examples:

```
map (fun x -> x*x) [2; 3; 5; 7]    => [4; 9; 25; 49]
```

`compose` (not pre-defined in some implementations)

```
let compose f g = fun x -> f (g x);;
(* or just *)
let compose f g x = f (g x);;

(compose hd tl) [1; 2; 3]   => 2
```

Folding (reduction)

```
fold_left ( * ) 1 [2; 3; 5; 7]     => 210
   (* note the spaces around * -- so it's not a comment *)
```

As is typical in uses of this function, 1 is the identity element for multiplication; in nested calls the corresponding argument will be a subtotal.

Folding is predefined, but we could have declared it as

```
let rec fold_left f i l =
  match l with
  | [] -> i
  | h :: t -> fold_left f (f i h) t;;
```

There's also a `fold_right`, but it isn't as used as much, because it isn't tail recursive.

Higher-order functions are great for building other functions:

```
let total l = fold_left ( + ) 0 l;;
(* or just *)
let total = fold_left ( + ) 0;;

total [1; 2; 3; 4; 5]          => 15

let total_all ll = map total ll;;
(* or just *)
let total_all = map total;;

total_all  [[1; 2; 3; 4; 5];
            [2; 4; 6; 8; 10];
            [3; 6; 9; 12; 15]]    => [15; 30; 45]
```

Currying

Applying a function to only some of its arguments in order to produce a function that expects the rest of the arguments.

Named after the logician Haskell Curry (the same guy Haskell is named after).

Automatic in ML family languages (takes some effort in Lisp)

```
let total = fold (+) 0;;

let plus3 = ( + ) 3;;

plus3 4     => 7
```

```
let plusn n = fun k -> n + k;;
let inc = plusn 1;;

let plus3 = plusn 3;;
inc 5              => 6

let comb a b = fun x y -> a * x + b * y;;
let comb23 = comb 2 3;;
comb23 5 6               => 28
```

NB: `plusn` and `comb` require ***unlimited extent*** (covered in chapter 3)

So what is going on here?

```
let ave a b = (a +. b) /. 2.0;;
```

is shorthand for

```
let ave = fun a -> fun b -> (a +. b) /. 2.0;;
```

This explains why OCaml says that the type of ave (even with the first definition) is "`float -> float -> float`".

```
val ave : float -> float -> float = <fun>
```

Juxtaposition helps makes things really clean.
When I say

```
ave a b
```

do I mean (in mathematical notation)

```
ave (a b)
```

or

```
(ave (a)) (b)    ?
```

The answer is, it doesn't really matter!
I do need to be aware what's going on, though, because if I give a function two few arguments the error message will usually be "type clash", rather than "too few arguments".

As an assignment using functional programming, I often provide students with a scanner and a parser generator in OCaml, and ask them to extend it to build a simple interpreter or compiler. As an example of the use of functional forms, here's an except from the parser generator.

Grammars are represented as a list of pairs, in which the first element of each pair is a nonterminal and the second element is a list of right-hand sides for productions with that nonterminal on the left-hand side. Each right-hand side is itself a list of symbols.

Here's the calculator grammar:

```
# calc_gram;;
- : (String.t * String.t list list) list =
[("P", [["SL"; "$$"]]);
("SL", [["S"; "SL"]; []]);
("S", [["id"; ":="; "E"]; ["read"; "id"]; ["write"; "E"]]);
("E", [["T"; "TT"]]);
("T", [["F"; "FT"]]);
("TT", [["ao"; "T"; "TT"]; []]);
("FT", [["mo"; "F"; "FT"]; []]);
("ao", [["+"]; ["-"]]);
("mo", [["*"]; ["/"]]);
("F", [["id"]; ["num"]; ["("; "E"; ")"]])]
```

Parse tables (for a table-driven top-down parser) are also a list of pairs, and again the first element of each pair is a nonterminal. The second element of each pair is more complicated: it's a nested list of pairs, in which the first element of each pair is a predict set (a list of terminals) and the second element is the right-hand side to predict when an element of the predict set is seen on the input.

Here's the table for the calculator grammar:

```
# get_parse_table calc_gram;;
- : (String.t * (String.t list * String.t list) list) list =
[("P", [(["$$"; "id"; "read"; "write"], ["SL"; "$$"])]);
("SL", [(["id"; "read"; "write"], ["S"; "SL"]); (["$$"], [])]);
("S",
[(["id"], ["id"; ":="; "E"]);
```

```
 ([“read”], [“read”; “id”]);
 ([“write”], [“write”; “E”])
]);
(“E”, [([“(“; “id”; “num”], [“T”; “TT”])]);
(“T”, [([“(“; “id”; “num”], [“F”; “FT”])]);
(“TT”,
[([“+”; “–”], [“ao”; “T”; “TT”]);
 ([“$$”; “)”; “id”; “read”; “write”], [])
]);
(“FT”,
[([“*”; “/”], [“mo”; “F”; “FT”]);
 ([“$$”; “)”; “+”; “–”; “id”; “read”; “write”], [])
]);
(“ao”, [([“+”], [“+”]); ([“–”], [“–”])]);
(“mo”, [([“*”], [“*”]); ([“/”], [“/”])]);
(“F”, [([“id”], [“id”]); ([“num”], [“num”]); ([“(“], [“(“; “E”;
“)”])])
]
```

Now suppose we have a parse table and we'd like to extract the original grammar. Here's a function that does so:

```
let grammar_of parse_tab =
  map (fun p -> (fst p,
                 (fold_left (@)
                            []
                            (map (fun (a, b) -> [b]) (snd p)))))
      parse_tab;;
```

If you understand how it does that, you're probably in good shape for an OCaml project. If you don't understand it, you should study it carefully, review Sec. 10.5 in the text, bring it up in workshop, or talk to the TA or instructor.

==========

Evaluation order (more in Chapters 6 and 9)

Applicative order

what you're used to in imperative languages
evaluate all arguments before passing to function
usually faster

Normal order

don't evaluate arg until you need it
sometimes faster -- more overhead to remember ***how*** to compute things, but avoid computing if you never need them
terminates if anything will (Church-Rosser theorem)
in naive form, can require re-evaluating something multiple times

Lazy evaluation gives the best of both worlds, so long as you stay purely functional: it evaluates only when it has to, but remembers the value so it doesn't compute it more than once. The implementation is said to use ***memoization*** (as in, it creates a memo). It can give unexpected answers in the presence of side effects.

A ***strict*** language requires all arguments to be well-defined, so applicative order can be used. A ***non-strict*** language does not require all arguments to be well-defined; it requires normal-order or lazy evaluation.

Lisp and ML are strict by default. Haskell is non-strict by default (you can ask for eager evaluation when you want it).

So in OCaml the following will result in a `Division_by_zero` exception:

```
let choose c t e =
  if c then t else e in
choose (3 < 4) 5 (6 / 0);;
```

But in Haskell the following evaluates just fine:

```
choose c t e =
  if c then t else e          -- declaration of choose
choose (3 < 4) 5 (6 / 0)        => 5
```

Note that in both languages `if (3 < 4) then 5 else (6 / 0)` evaluates to 5. That's because `if..then..else` is a ***special*** *form*, built into the language, whose arguments are lazily evaluated. In a similar vein, the following evaluate without error in both OCaml and Haskell—neither throws an exception in either language:

```
(3 < 4) or ((6 / 0) > 5)        => true
```

```
(3 > 4) and ((6 / 0) > 5)       => false
```

We talk more about lazy evaluation in Chapter 6.

In a language like OCaml you can build lazy evaluation for contexts in which it isn't otherwise provided using higher-order functions—in this case, functions embedded in data structures:

```
type 'a stream =
  Cons of 'a * (unit -> 'a stream);;
let hd : 'a stream -> 'a = function Cons(h, _) -> h;;
let tl : 'a stream -> 'a stream =
  function Cons(_, t) -> t ();;
```

We can use this machinery to create an "infinite list" of, say, the squares of the natural numbers:

```
let squares =
   let rec next n = Cons (n*n, fun () -> next (n+1)) in
   next 1;;

let rec nth n s =
   if n = 1 then hd s
   else nth (n-1) (tl s);;

let rec take n s =
   if n = 0 then []
   else (hd s) :: take (n-1) (tl s);;

hd squares;;              => 1
hd (tl squares);;         => 4
hd (tl (tl squares));;    => 9
nth 5 squares;;           => 25
take 5 squares;;          => [1; 4; 9; 16; 25]
```

Unfortunately, if we evaluate "`nth 123 squares`" multiple times, we'll compute the whole list multiple times. Wouldn't it be nice to ***remember*** the computed part of the list, and keep it around?
There's a standard library called `Lazy` that does this for you.

```
let a = lazy (expensive_function x);;
let b = if unlikely_condition then Lazy.force a else 0;;
```

or

```
let b = match unlikely_condition, a with
| true, lazy v -> v
| _ -> 0;;
```

The second version incorporates a ***lazy pattern match***. It will happen only if `unlikely_condition` evaluates to true.

Lazy works by creating functions that compute the values you want, but only when called. The '`lazy`' keyword creates the function; '`force`' calls it (and remembers the result for future reference).

Here's a re-write of the stream type:

```
type 'a stream =
  Cons of 'a * 'a stream Lazy.t;;
let hd : 'a stream -> 'a = function Cons(h, _) -> h;;
let tl : 'a stream -> 'a stream =
  function Cons(_, t) -> Lazy.force t;;

let squares =
  let rec next n =
    Cons (n*n, lazy (next (n+1))) in next 1;;

(* nth and take as before *)
```

Because of memoization, the list will actually be fleshed out in memory, and kept, as needed. If computing the next value was much more expensive than adding 1 each time, we'd only compute each explored value once.