Functional Programming

by Łukasz Stafiniak

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

Lecture 7: Laziness

Lazy evaluation. Stream processing.

M. Douglas McIlroy *"Power Series, Power Serious"*

Oleg Kiselyov, Simon Peyton-Jones, Amr Sabry
"Lazy v. Yield: Incremental, Linear Pretty-Printing"

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

Laziness

- Today's lecture is about lazy evaluation.
- Thank you for coming, goodbye!
- But perhaps, do you have any questions?

Evaluation strategies and parameter passing

- **Evaluation strategy** is the order in which expressions are computed.
  - For the most part: when are arguments computed.
- Recall our problems with using *flow control* expressions like `if_then_else` in examples from λ-calculus lecture.
- There are many technical terms describing various strategies. Wikipedia:

  Strict evaluation. Arguments are always evaluated completely before function is applied.

  Non-strict evaluation. Arguments are not evaluated unless they are actually used in the evaluation of the function body.

  Eager evaluation. An expression is evaluated as soon as it gets bound to a variable.

  Lazy evaluation. Non-strict evaluation which avoids repeating computation.

Call-by-value. The argument expression is evaluated, and the resulting value is bound to the corresponding variable in the function (frequently by copying the value into a new memory region).

Call-by-reference. A function receives an implicit reference to a variable used as argument, rather than a copy of its value.

- In purely functional languages there is no difference between the two strategies, so they are typically described as call-by-value even though implementations use call-by-reference internally for efficiency.
- Call-by-value languages like C and OCaml support explicit references (objects that refer to other objects), and these can be used to simulate call-by-reference.

Normal order. Start computing function bodies before evaluating their arguments. Do not even wait for arguments if they are not needed.

Call-by-name. Arguments are substituted directly into the function body and then left to be evaluated whenever they appear in the function.

Call-by-need. If the function argument is evaluated, that value is stored for subsequent uses.

- Almost all languages do not compute inside the body of un-applied function, but with curried functions you can pre-compute data before all arguments are provided.
  - Recall the `search_bible` example.
- In eager / call-by-value languages we can simulate call-by-name by taking a function to compute the value as an argument instead of the value directly.
  - "Our" languages have a `unit` type with a single value `()` specifically for use as throw-away arguments.
  - Scala has a built-in support for call-by-name (i.e. direct, without the need to build argument functions).
- ML languages have built-in support for lazy evaluation.
- Haskell has built-in support for eager evaluation.

Call-by-name: streams

- Call-by-name is useful not only for implementing flow control
  - ```
    let if_then_else cond e1 e2 =
      match cond with true -> e1 () | false -> e2 ()
    ```

  but also for arguments of value constructors, i.e. for data structures.
- **Streams** are lists with call-by-name tails.

  ```
  type 'a stream = SNil | SCons of 'a * (unit -> 'a stream)
  ```
- Reading from a stream into a list.

  ```
  let rec stake n = function
   | SCons (a, s) when n > 0 -> a::(stake (n-1) (s ()))
   | _ -> []
  ```
- Streams can easily be infinite.

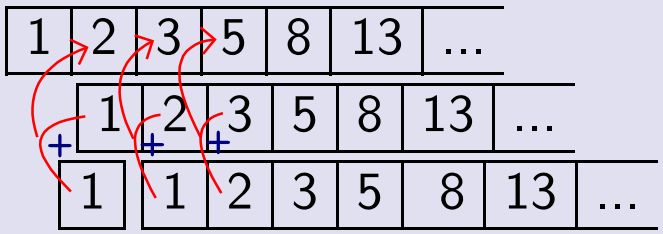
  ```
  let rec s_ones = SCons (1, fun () -> s_ones)
  let rec s_from n =
    SCons (n, fun () ->s_from (n+1))
  ```
  ```

- Streams admit list-like operations.

```
let rec smap f = function
 | SNil -> SNil
 | SCons (a, s) -> SCons (f a, fun () -> smap f (s ()))
let rec szip = function
 | SNil, SNil -> SNil
 | SCons (a1, s1), SCons (a2, s2) ->
     SCons ((a1, a2), fun () -> szip (s1 (), s2 ()))
 | _ -> raise (Invalid_argument "szip")
```

- Streams can provide scaffolding for recursive algorithms:

```
let rec sfib =
  SCons (1, fun () -> smap (fun (a,b)-> a+b)
    (szip (sfib, SCons (1, fun () -> sfib))))
```



- Streams are less functional than could be expected in context of input-output effects.

```
let file_stream name =
  let ch = open_in name in
  let rec ch_read_line () =
    try SCons (input_line ch, ch_read_line)
    with End_of_file -> SNil in
  ch_read_line ()
```

- *OCaml Batteries* use a stream type `enum` for interfacing between various sequence-like data types.
  - The safest way to use streams in a *linear* / *ephemeral* manner: every value used only once.
  - Streams minimize space consumption at the expense of time for recomputation.

Lazy values

- Lazy evaluation is more general than call-by-need as any value can be lazy, not only a function parameter.
- A *lazy value* is a value that "holds" an expression until its result is needed, and from then on it "holds" the result.
  - Also called: a *suspension*. If it holds the expression, called a *thunk*.
- In OCaml, we build lazy values explicitly. In Haskell, all values are lazy but functions can have call-by-value parameters which "need" the argument.
- To create a lazy value: `lazy expr` – where `expr` is the suspended computation.
- Two ways to use a lazy value, be careful when the result is computed!
  - In expressions: `Lazy.force l_expr`
  - In patterns: `match l_expr with lazy v -> ...`
    - Syntactically `lazy` behaves like a data constructor.

- Lazy lists:

```
type 'a llist = LNil | LCons of 'a * 'a llist Lazy.t
```

- Reading from a lazy list into a list:

```
let rec ltake n = function
 | LCons (a, lazy l) when n > 0 -> a::(ltake (n-1) l)
 | _ -> []
```

- Lazy lists can easily be infinite:

```
let rec l_ones = LCons (1, lazy l_ones)
let rec l_from n = LCons (n, lazy (l_from (n+1)))
```

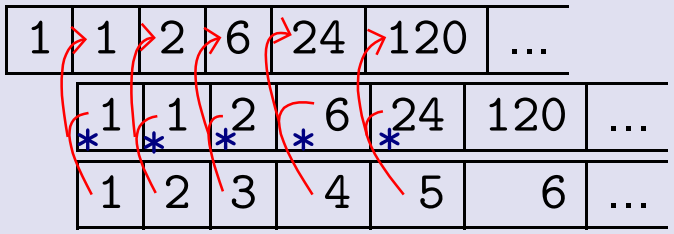
- Read once, access multiple times:

```
let file_llist name =
  let ch = open_in name in
  let rec ch_read_line () =
    try LCons (input_line ch, lazy (ch_read_line ()))
    with End_of_file -> LNil in
  ch_read_line ()
```

- ```
  let rec lzip = function
    | LNil, LNil -> LNil
    | LCons (a1, ll1), LCons (a2, ll2) ->
        LCons ((a1, a2), lazy (
          lzip (Lazy.force ll1, Lazy.force ll2)))
    | _ -> raise (Invalid_argument "lzip")

  let rec lmap f = function
    | LNil -> LNil
    | LCons (a, ll) ->
      LCons (f a, lazy (lmap f (Lazy.force ll)))
  ```

- ```
  let posnums = lfrom 1
  let rec lfact =
    LCons (1, lazy (lmap (fun (a,b)-> a*b)
                         (lzip (lfact, posnums))))
  ```



Power series and differential equations

- Differential equations idea due to Henning Thielemann. **Just an example.**
- Expression $P(x) = \sum_{i=0}^{n} a_i x^i$ defines a polynomial for $n < \infty$ and a power series for $n = \infty$.
- If we define

```
let rec lfold_right f l base =
  match l with
    | LNil -> base
    | LCons (a, lazy l) -> f a (lfold_right f l base)
```

  then we can compute polynomials

```
let horner x l =
  lfold_right (fun c sum -> c +. x *. sum) l 0.
```

- But it will not work for infinite power series!
  - Does it make sense to compute the value at x of a power series?
  - Does it make sense to fold an infinite list?

- If the power series converges for $x > 1$, then when the elements a_n get small, the remaining sum $\sum_{i=n}^{\infty} a_i x^i$ is also small.
- `lfold_right` falls into an infinite loop on infinite lists. We need call-by-name / call-by-need semantics for the argument function `f`.

```
let rec lazy_foldr f l base =
  match l with
    | LNil -> base
    | LCons (a, ll) ->
      f a (lazy (lazy_foldr f (Lazy.force ll) base))
```

- We need a stopping condition in the Horner algorithm step:

```
let lhorner x l =
  let upd c sum =
    if c = 0. || abs_float c > epsilon_float
    then c +. x *. Lazy.force sum
    else 0. in
  lazy_foldr upd l 0.

let inv_fact = lmap (fun n -> 1. /. float_of_int n) lfact
let e = lhorner 1. inv_fact
```

This is a bit of a hack, we hope to "hit" the interval $(0, \varepsilon]$.

Power series / polynomial operations

- ```
  let rec add xs ys =
    match xs, ys with
      | LNil, _ -> ys
      | _, LNil -> xs
      | LCons (x,xs), LCons (y,ys) ->
        LCons (x +. y, lazy (add (Lazy.force xs) (Lazy.force ys)))
  ```
- ```
  let rec sub xs ys =
    match xs, ys with
      | LNil, _ -> lmap (fun x-> ~-.x) ys
      | _, LNil -> xs
      | LCons (x,xs), LCons (y,ys) ->
        LCons (x-.y, lazy (add (Lazy.force xs) (Lazy.force ys)))
  ```
- ```
  let scale s = lmap (fun x->s*.x)
  ```
- ```
  let rec shift n xs =
    if n = 0 then xs
    else if n > 0 then LCons (0. , lazy (shift (n-1) xs))
    else match xs with
      | LNil -> LNil
      | LCons (0., lazy xs) -> shift (n+1) xs
      | _ -> failwith "shift: fractional division"
  ```

- ```
  let rec mul xs = function
    | LNil -> LNil
    | LCons (y, ys) ->
      add (scale y xs) (LCons (0., lazy (mul xs (Lazy.force ys))))
  ```

- ```
  let rec div xs ys =
    match xs, ys with
    | LNil, _ -> LNil
    | LCons (0., xs'), LCons (0., ys') ->
      div (Lazy.force xs') (Lazy.force ys')
    | LCons (x, xs'), LCons (y, ys') ->
      let q = x /. y in
      LCons (q, lazy (divSeries (sub (Lazy.force xs')
                                     (scale q (Lazy.force ys'))) ys))
    | LCons _, LNil -> failwith "divSeries: division by zero"
  ```

- ```
  let integrate c xs =
    LCons (c, lazy (lmap (uncurry (/.)) (lzip (xs, posnums))))
  ```

- ```
  let ltail = function
    | LNil -> invalid_arg "ltail"
    | LCons (_, lazy tl) -> tl
  ```

- ```
  let differentiate xs =
    lmap (uncurry ( *.)) (lzip (ltail xs, posnums))
  ```
  ```

Differential equations

- $\frac{\mathrm{d}\sin x}{\mathrm{d}x} = \cos x, \frac{\mathrm{d}\cos x}{\mathrm{d}x} = -\sin x, \sin 0 = 0, \cos 0 = 1.$
- We will solve the corresponding integral equations. *Why?*
- We cannot define the integral by direct recursion like this:

```
let rec sin = integrate (of_int 0) cos
and cos = integrate (of_int 1) ~-:sin
```

Unary op. `let (~-:) = lmap (fun x-> ~-.x)`

unfortunately fails:

```
Error: This kind of expression is not allowed as right-
hand side of ‘let rec’
```

  - Even changing the second argument of `integrate` to call-by-need does not help, because OCaml cannot represent the values that `x` and `y` refer to.

- We need to inline a bit of `integrate` so that OCaml knows how to start building the recursive structure.

```
let integ xs = lmap (uncurry (/.)) (lzip (xs, posnums))
let rec sin = LCons (of_int 0, lazy (integ cos))
and cos = LCons (of_int 1, lazy (integ ~-:sin))
```

- The complete example would look much more elegant in Haskell.
- Although this approach is not limited to linear equations, equations like Lotka-Volterra or Lorentz are not "solvable" – computed coefficients quickly grow instead of quickly falling...

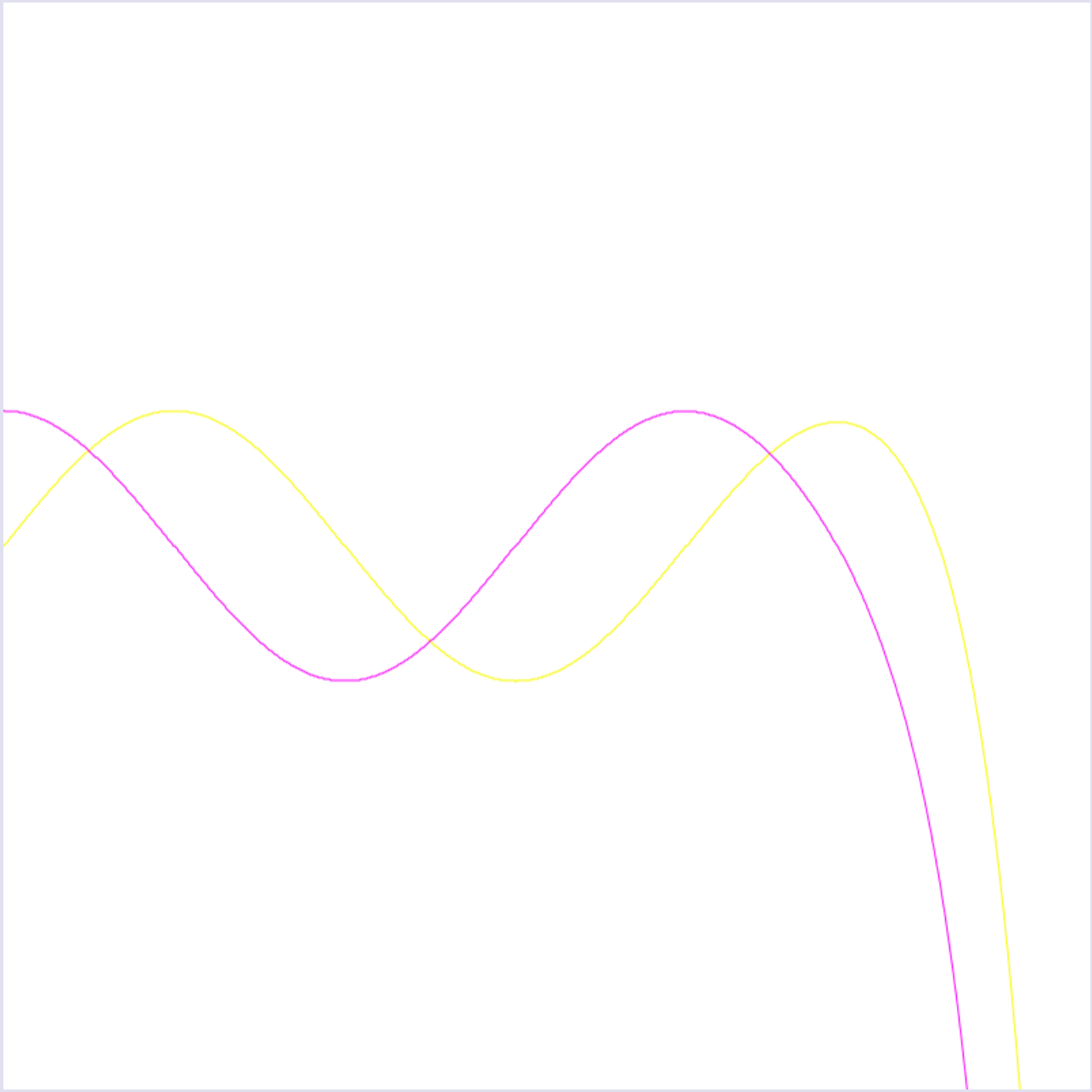
- Drawing functions are like in previous lecture, but with open curves.
- ```
  let plot_1D f ~w ~scale ~t_beg ~t_end =
    let dt = (t_end -. t_beg) /. of_int w in
    Array.init w (fun i ->
      let y = lhorner (dt *. of_int i) f in
      i, to_int (scale *. y))
  ```
  ```

Arbitrary precision computation

- Putting it all together reveals drastic numerical errors for large x.

```
let graph =
  let scale = of_int h /. of_int 8 in
  [plot_1D sin ~w ~h0:(h/2) ~scale
      ~t_beg:(of_int 0) ~t_end:(of_int 15),
   (250,250,0);
   plot_1D cos ~w ~h0:(h/2) ~scale
     ~t_beg:(of_int 0) ~t_end:(of_int 15),
   (250,0,250)]
let () = draw_to_screen ~w ~h graph
```

  - Floating-point numbers have limited precision.
  - We break out of Horner method computations too quickly.



- For infinite precision on rational numbers we use the `nums` library.
  - It does not help – yet.
- Generate a sequence of approximations to the power series limit at x.

```
let infhorner x l =
  let upd c sum =
    LCons (c, lazy (lmap (fun apx -> c+.x*.apx)
                         (Lazy.force sum))) in
  lazy_foldr upd l (LCons (of_int 0, lazy LNil))
```

- Find where the series converges – as far as a given test is concerned.

```
let rec exact f = function
  | LNil -> assert false
  | LCons (x0, lazy (LCons (x1, lazy (LCons (x2, _)))))
      when f x0 = f x1 && f x0 = f x2 -> f x0
  | LCons (_, lazy tl) -> exact f tl
```

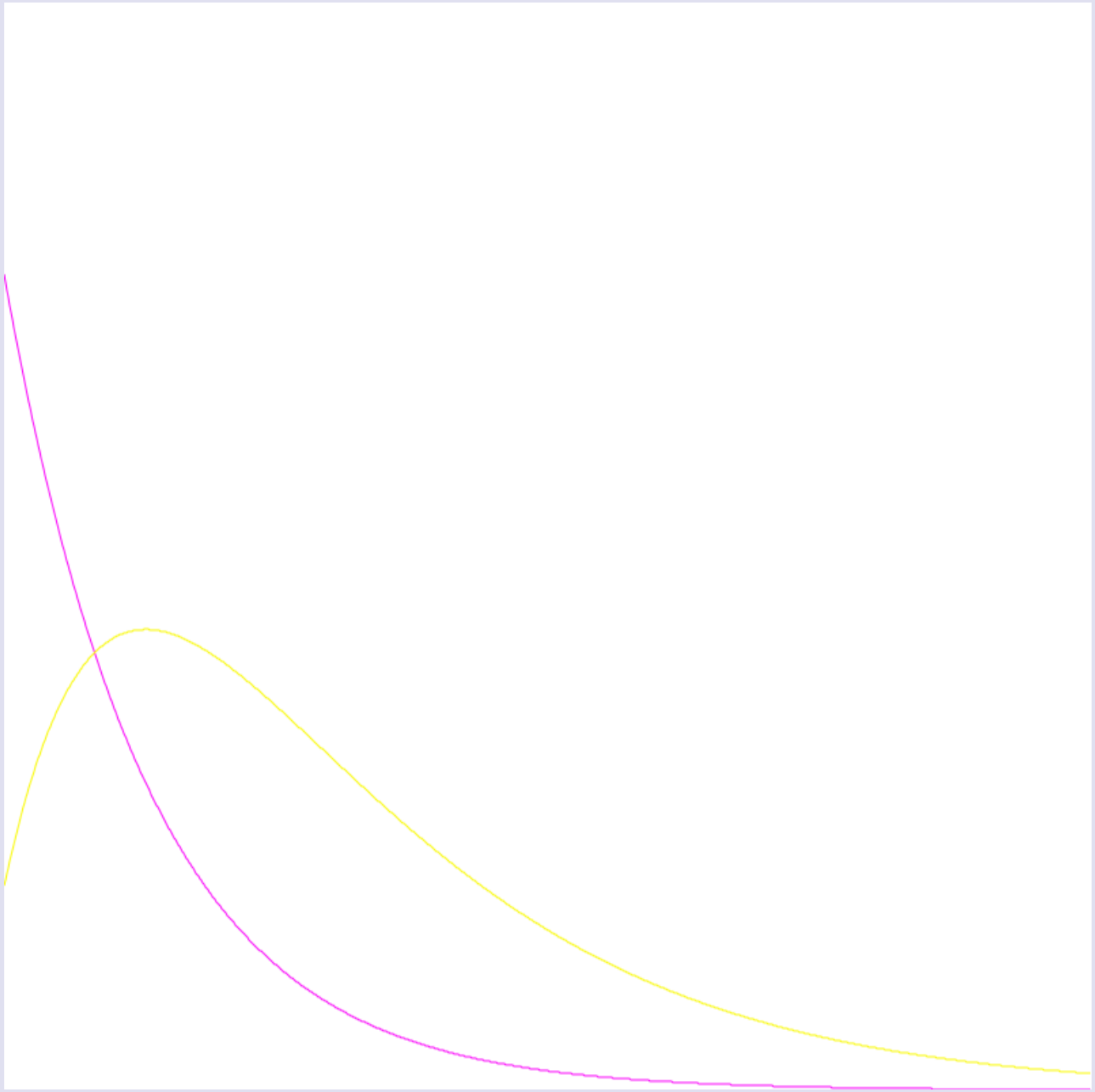
We arbitrarily decide that convergence is when three consecutive results are the same.

- Draw the pixels of the graph at exact coordinates.

```
let plot_1D f ~w ~h0 ~scale ~t_beg ~t_end =
  let dt = (t_end -. t_beg) /. of_int w in
  let eval = exact (fun y-> to_int (scale *. y)) in
  Array.init w (fun i ->
    let y = infhorner (t_beg +. dt *. of_int i) f in
    i, h0 + eval y)
```

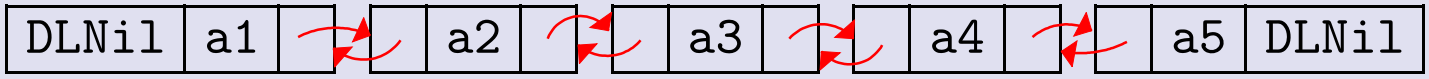
- Success! If a power series had every third term contributing we would have to check three terms in the function `exact`...
  - We could like in `lhorner` test for `f x0 = f x1 && not x0 =. x1`
- Example `n_chain`: nuclear chain reaction–*A decays into B decays into C*
  - http://en.wikipedia.org/wiki/Radioactive_decay#Chain-decay_processes

```
let n_chain ~nA0 ~nB0 ~lA ~lB =
  let rec nA =
    LCons (nA0, lazy (integ (~-.lA *:. nA)))
  and nB =
    LCons (nB0, lazy (integ (~-.lB *:. nB +: lA *:. nA))) in
  nA, nB
```



Circular data structures: double-linked list

- Without delayed computation, the ability to define data structures with referential cycles is very limited.
- Double-linked lists contain such cycles between any two nodes even if they are not cyclic when following only *forward* or *backward* links.



- We need to "break" the cycles by making some links lazy.
- ```
  type 'a dllist =
    DLNil | DLCons of 'a dllist Lazy.t * 'a * 'a dllist
  ```
- ```
  let rec dldrop n l =
    match l with
      | DLCons (_, x, xs) when n>0 ->
        dldrop (n-1) xs
      | _ -> l
  ```

- ```
  let dllist_of_list l =
    let rec dllist prev l =
      match l with
        | [] -> DLNil
        | x::xs ->
          let rec cell =
            lazy (DLCons (prev, x, dllist cell xs)) in
          Lazy.force cell in
    dllist (lazy DLNil) l
  ```
- ```
  let rec dltake n l =
    match l with
      | DLCons (_, x, xs) when n>0 ->
        x::dltake (n-1) xs
      | _ -> []
  ```
- ```
  let rec dlbackwards n l =
    match l with
      | DLCons (lazy xs, x, _) when n>0 ->
        x::dlbackwards (n-1) xs
      | _ -> []
  ```
  ```

Input-Output streams

- The stream type used a throwaway argument to make a suspension

```
type 'a stream = SNil | SCons of 'a * (unit -> 'a stream)
```

  What if we take a real argument?

```
type ('a, 'b) iostream =
  EOS | More of 'b * ('a -> ('a, 'b) iostream)
```

  A stream that for a single input value produces an output value.

- `type 'a istream = (unit, 'a) iostream`
  Input stream produces output when "asked".

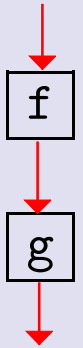
  `type 'a ostream = ('a, unit) iostream`
  Output stream consumes provided input.

  - Sorry, the confusion arises from adapting the *input file / output file* terminology, also used for streams.

- We can compose streams: directing output of one to input of another.

```
let rec compose sf sg =
  match sg with
  | EOS -> EOS                                          No more output.
  | More (z, g) ->
    match sf with                                                 No more
    | EOS -> More (z, fun _ -> EOS)                  input "processing power".
   | More (y, f) ->
      let update x = compose (f x) (g y) in
      More (z, update)
```

  - Every box has one incoming and one outgoing wire:



  - Notice how the output stream is ahead of the input stream.

Pipes

- We need a more flexible input-output stream definition.
  - Consume several inputs to produce a single output.
  - Produce several outputs after a single input (or even without input).
  - No need for a dummy when producing output requires input.
- After Haskell, we call the data structure `pipe`.

```
type ('a, 'b) pipe =
  EOP
| Yield of 'b * ('a, 'b) pipe
| Await of 'a -> ('a, 'b) pipe
```

For incremental streams change to lazy.

- Again, we can have producing output only *input pipes* and consuming input only *output pipes*.

```
type 'a ipipe = (unit, 'a) pipe
type void
type 'a opipe = ('a, void) pipe
```

  - Why `void` rather than `unit`, and why only for `opipe`?

- Composition of pipes is like "concatenating them in space" or connecting boxes:

```
let rec compose pf pg =
  match pg with
  | EOP -> EOP                                          Done producing results.
  | Yield (z, pg') -> Yield (z, compose pf pg')          Ready result.
  | Await g ->
    match pf with
    | EOP -> EOP                                        End of input.
    | Yield (y, pf') -> compose pf' (g y)               Compute next result.
    | Await f ->
      let update x = compose (f x) pg in
      Await update                                      Wait for more input.

let (>->) pf pg = compose pf pg
```

- Appending pipes means "concatenating them in time" or adding more fuel to a box:

```
let rec append pf pg =
  match pf with
  | EOP -> pg                                   When pf runs out, use pg.
  | Yield (z, pf') -> Yield (z, append pf' pg)
  | Await f ->                  If pf awaits input, continue when it comes.
    let update x = append (f x) pg in
    Await update
```

- Append a list of ready results in front of a pipe.

```
let rec yield_all l tail =
  match l with
  | [] -> tail
  | x::xs -> Yield (x, yield_all xs tail)
```

- Iterate a pipe (**not functional**).

```
let rec iterate f : 'a opipe =
  Await (fun x -> let () = f x in iterate f)
```

Example: pretty-printing

- Print hierarchically organized document with a limited line width.

```
type doc =
  Text of string | Line | Cat of doc * doc | Group of doc
```

- ```
  let (++) d1 d2 = Cat (d1, Cat (Line, d2))
  let (!) s = Text s
  let test_doc =
    Group (!"Document" ++
               Group (!"First part" ++ !"Second part"))
  ```

```
# let () = print_endline (pretty 30 test_doc);;
Document
First part Second part
# let () = print_endline (pretty 20 test_doc);;
Document
First part
Second part
# let () = print_endline (pretty 60 test_doc);;
Document First part Second part
```
```

- Straightforward solution:

```
let pretty w d =                                    Allowed width of line w.
  let rec width = function                          Total length of subdocument.
    | Text z -> String.length z
    | Line -> 1
    | Cat (d1, d2) -> width d1 + width d2
    | Group d -> width d in
  let rec format f r = function                     Remaining space r.
    | Text z -> z, r - String.length z
    | Line when f -> " ", r-1                       If not f then line breaks.
    | Line -> "\n", w
    | Cat (d1, d2) ->
      let s1, r = format f r d1 in
      let s2, r = format f r d2 in
      s1 ^ s2, r                                    If following group fits, then without line breaks.
    | Group d -> format (f || width d <= r) r d in
  fst (format false w d)
```

- Working with a stream of nodes.

```
type ('a, 'b) doc_e =                Annotated nodes, special for group beginning.
  TE of 'a * string | LE of 'a | GBeg of 'b | GEnd of 'a
```

- Normalize a subdocument – remove empty groups.

```
let rec norm = function
  | Group d -> norm d
  | Text "" -> None
  | Cat (Text "", d) -> norm d
  | d -> Some d
```

- Generate the stream by infix traversal.

```
let rec gen = function
  | Text z -> Yield (TE ((),z), EOP)
  | Line -> Yield (LE (), EOP)
  | Cat (d1, d2) -> append (gen d1) (gen d2)
  | Group d ->
    match norm d with
    | None -> EOP
    | Some d ->
      Yield (GBeg (),
             append (gen d) (Yield (GEnd (), EOP)))
```

- Compute lengths of document prefixes, i.e. the position of each node counting by characters from the beginning of document.

```
let rec docpos curpos =
  Await (function                                  We input from a doc_e pipe
  | TE (_, z) ->
    Yield (TE (curpos, z),          and output doc_e annotated with position.
           docpos (curpos + String.length z))
  | LE _ ->                        Spice and line breaks increase position by 1.
    Yield (LE curpos, docpos (curpos + 1))
  | GBeg _ ->                                 Groups do not increase position.
    Yield (GBeg curpos, docpos curpos)
  | GEnd _ ->
    Yield (GEnd curpos, docpos curpos))

let docpos = docpos 0                          The whole document starts at 0.
```

- Put the end position of the group into the group beginning marker, so that we can know whether to break it into multiple lines.

```
let rec grends grstack =
  Await (function
  | TE _ | LE _ as e ->
    (match grstack with
    | [] -> Yield (e, grends [])                    We can yield only when
    | gr::grs -> grends ((e::gr)::grs))             no group is waiting.
  | GBeg _ -> grends ([]::grstack)                  Wait for end of group.
  | GEnd endp ->
    match grstack with                              End the group on top of stack.
    | [] -> failwith "grends: unmatched group end marker"
    | [gr] ->                                       Top group – we can yield now.
      yield_all
        (GBeg endp::List.rev (GEnd endp::gr))
        (grends [])
    | gr::par::grs ->                               Remember in parent group instead.
      let par = GEnd endp::gr @ [GBeg endp] @ par in
      grends (par::grs))                            Could use catenable lists above.
```

- That's waiting too long! We can stop waiting when the width of a group exceeds line limit. `GBeg` will not store end of group when it is irrelevant.

```
let rec grends w grstack =
  let flush tail =                          When the stack exceeds width w,
    yield_all                                 flush it – yield everything in it.
      (rev_concat_map ~prep:(GBeg Too_far) snd grstack)
      tail in          Above: concatenate in rev. with prep before each part.
  Await (function
  | TE (curp, _) | LE curp as e ->
    (match grstack with        Remember beginning of groups in the stack.
    | [] -> Yield (e, grends w [])
    | (begp, _)::_ when curp-begp > w ->
      flush (Yield (e, grends w []))
    | (begp, gr)::grs -> grends w ((begp, e::gr)::grs))
  | GBeg begp -> grends w ((begp, [])::grstack)
```

```
| GEnd endp as e ->
  match grstack with
  | [] -> Yield (e, grends w [])
  | (begp, _)::_ when endp-begp > w ->
    flush (Yield (e, grends w []))
  | [_, gr] ->
    yield_all
      (GBeg (Pos endp)::List.rev (GEnd endp::gr))
      (grends w [])
  | (_, gr)::(par_begp, par)::grs ->
    let par =
      GEnd endp::gr @ [GBeg (Pos endp)] @ par in
    grends w ((par_begp, par)::grs))
```

No longer fail when the stack is empty – could have been flushed.

If width not exceeded, work as before optimization.

- Initial stack is empty:

```
let grends w = grends w []
```

- Finally we produce the resulting stream of strings.

```
let rec format w (inline, endlpos as st) =
  Await (function
  | TE (_, z) -> Yield (z, format w st)
  | LE p when List.hd inline ->
    Yield (" ", format w st)
  | LE p -> Yield ("\n", format w (inline, p+w))
  | GBeg Too_far ->
    format w (false::inline, endlpos)
  | GBeg (Pos p) ->
    format w ((p<=endlpos)::inline, endlpos)
  | GEnd _ -> format w (List.tl inline, endlpos))

let format w = format w ([false], w)
```

State: the stack of "group fits in line"; position where end of line would be.

After return, line has w free space.

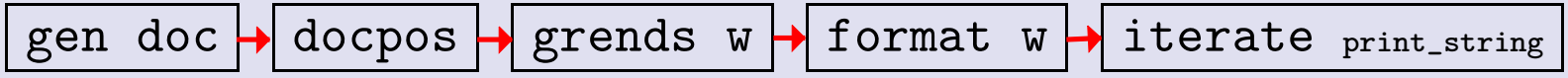
Group with end too far is not inline.

Group is inline if it ends soon enough.

Break lines outside of groups.

- Put the pipes together:

```
let pretty_print w doc =
```



- Factorize `format` so that various line breaking styles can be plugged in.

```
let rec breaks w (inline, endlpos as st) =
  Await (function
  | TE _ as e -> Yield (e, breaks w st)
  | LE p when List.hd inline ->
    Yield (TE (p, " "), breaks w st)
  | LE p as e -> Yield (e, breaks w (inline, p+w))
  | GBeg Too_far as e ->
    Yield (e, breaks w (false::inline, endlpos))
  | GBeg (Pos p) as e ->
    Yield (e, breaks w ((p<=endlpos)::inline, endlpos))
  | GEnd _ as e ->
    Yield (e, breaks w (List.tl inline, endlpos)))

let breaks w = breaks w ([false], w)
```

```
let rec emit =
  Await (function
  | TE (_, z) -> Yield (z, emit)
  | LE _ -> Yield ("n", emit)
  | GBeg _ | GEnd _ -> emit)

let pretty_print w doc =
  gen doc >-> docpos >-> grends w >-> breaks w >->
  emit >-> iterate print_string
```

- Tests.

```
let (++) d1 d2 = Cat (d1, Cat (Line, d2))
let (!) s = Text s
let test_doc =
  Group (!"Document" ++
           Group (!"First part" ++ !"Second part"))

let print_e_doc pr_p pr_ep = function
  | TE (p,z) -> pr_p p; print_endline (": "^z)
  | LE p -> pr_p p; print_endline ": endline"
  | GBeg ep -> pr_ep ep; print_endline ": GBeg"
  | GEnd p -> pr_p p; print_endline ": GEnd"
let noop () = ()
let print_pos = function
  | Pos p -> print_int p
  | Too_far -> print_string "Too far"

let _ = gen test_doc >->
  iterate (print_e_doc noop noop)
let _ = gen test_doc >-> docpos >->
  iterate (print_e_doc print_int print_int)
let _ = gen test_doc >-> docpos >-> grends 20 >->
  iterate (print_e_doc print_int print_pos)
let _ = gen test_doc >-> docpos >-> grends 30 >->
  iterate (print_e_doc print_int print_pos)
let _ = gen test_doc >-> docpos >-> grends 60 >->
  iterate (print_e_doc print_int print_pos)
let _ = pretty_print 20 test_doc
let _ = pretty_print 30 test_doc
let _ = pretty_print 60 test_doc
```