Evaluation Order

- Different programming languages, evaluate in different orders.
- Some things are common. E.g.
  - The conditional `if b then x else y end`:
    First b
    Second x or y
  - Sequential composition `S1;S2`:
    First S1
    Second S2
- Some things are different or undefined
  - evaluation order of (sub-)expressions
  - evaluation order for argument to a function call

Differences

- In Java evaluation order is left-to-right
- Using gcc for C:
  - expression are evaluated left-to-right
  - function argument are evaluated right-to-left
- In OCaml evalutation order is right-to-left, except
  - S1;S2 (first S1 then S2)
  - let x=e1 in e2 (first e1 then e2)

Examples

```
# let m i = Printf.printf "[%d]" i;i;;
val m : int -> int = <fun>
# (m 1,m 2,m 3);;
[3][2][1]- : int * int * int = (1, 2, 3)
# (+) (m 1) (m 2) ;;
[2][1]- : int = 3
# m(1)+m(2);;
[2][1]- : int = 3
```

Some remarks

- Order is can be semantically irrelevant.
  (E.g. no side-effects, no exceptions caught).
- Order can have practical impact (E.g. memory use).
- Relying on evaluation order is best avoided:
  - porting code from one language to another becomes difficult
  - different compiler (version) may have different result
- For our equivalence proofs we assume well-behaved functions:
  - No side effects.
  - No exceptions thrown.
  - Terminate for all values.

Introduction

- Lazy Computation means delaying the evaluation of an expression until the result is needed for the first time (never evaluating it if the result is never needed).
- Always costs some time for testing if result has been previously computed.
- Can save memory if expression small and result big.
- Can save time if result is never needed.
- Can cost memory if expression is big and result is small.

Implementation 1

Encode delayed evaluation as a function:

```
# let d = fun () -> m(1);;
val d : unit -> int = <fun>
```

Problem: expression evaluated every time function is called:

```
# d();;
[1]- : int = 1
# d();;
[1]- : int = 1
```

Implementation 2

Encode delayed evaluation as a function and memoize:

```
# let d = let x = ref None in
  fun () -> match !x with
    | None -> let v = m(1) in x:= Some(v);v
    | Some(v) -> v
;;
val d : unit -> int = <fun>
```

Implementation 2

Better expression evaluated once:

```
# d();;
[1]- : int = 1
# d();;
- : int = 1
```

However: not concise and difficult for compiler.

Implementation 3

Use the built-in lazy feature and Lazy.force:

```
# let d = lazy (m(1));;
val d : int lazy_t = <lazy>
```

Concise and evaluated once:

```
# open Lazy;;
# force d;;
[1]- : int = 1
# force d;;
- : int = 1
```

Applications

- If we want to know if a (unique) solution exists then we do not need all solutions.
- Enumerating solutions on demand uses much less memory than generating them all at once.

Lists

- Normal list: compute all elements at once.
- Lazy list: compute elements on demand.

Lazy lists type in OCaml

```
open Lazy ;;

type 'a lazy_list = 'a list1 Lazy.t
 and 'a list1 = Nil | Cons of 'a*'a lazy_list
;;
```

Conversion functions

```
let rec lazy_of_list l = lazy (match l with
 | [] -> Nil
 | x::xs -> Cons(x,lazy_of_list xs)
);;

let rec list_of_lazy l = match force l with
 | Nil -> []
 | Cons(x,xs) -> x::(list_of_lazy xs)
;;
```

Map

```
let rec lmap f l = lazy (match force l with
 | Nil -> Nil
 | Cons(x,xs) -> Cons(f x,lmap f xs)
);;
```

Map step-by-step

We start with the normal version.

```
let rec map f l = match l with
  | Nil -> Nil
  | Cons(x,xs) -> Cons(f x,map f xs)
;;
```

Map step-by-step

When you match against a lazy list, you force it:

```
let rec map f l = match force l with
  | Nil -> Nil
  | Cons(x,xs) -> Cons(f x,map f xs)
;;
```

Map step-by-step

Every lazy argument of a constructor gets a lazy:

```
let rec map f l = match force l with
  | Nil -> Nil
  | Cons(x,xs) -> Cons(f x,lazy (map f xs))
;;
```

Map step-by-step

The first step must be lazy as well:

```
let rec map f l = match force l with
  | Nil -> Nil
  | Cons(x,xs) -> Cons(f x,lazy (map f xs))
and lmap f l = lazy (map f l)
;;
```

Map step-by-step

That is equivalent to:

```
let rec map f l = match force l with
  | Nil -> Nil
  | Cons(x,xs) -> Cons(f x,lmap f xs)
and lmap f l = lazy (map f l)
;;
```

Map step-by-step

Which is equivalent to:

```
let rec map f l = match force l with
  | Nil -> Nil
  | Cons(x,xs) -> Cons(f x,lmap f xs)
and lmap f l = lazy (match force l with
  | Nil -> Nil
  | Cons(x,xs) -> Cons(f x,lmap f xs))
;;
```

Map step-by-step

Which is equivalent to:

```
let rec lmap f l = lazy (match force l with
  | Nil -> Nil
  | Cons(x,xs) -> Cons(f x,lmap f xs)
);;
```

Filter step-by-step

We start with the normal version.

```
let rec filter p l = match l with
  | Nil -> Nil
  | Cons(x,xs) -> if p x then Cons(x,filter p xs)
                            else (filter p xs)
;;
```

Filter step-by-step

When you match against a lazy list, you force it:

```
let rec filter p l = match force l with
  | Nil -> Nil
  | Cons(x,xs) -> if p x then Cons(x,filter p xs)
                         else (filter p xs)
;;
```

Filter step-by-step

Every lazy argument of a constructor gets a lazy:

```
let rec filter p l = match force l with
  | Nil -> Nil
  | Cons(x,xs) -> if p x then Cons(x,lazy (filter p
                              else (filter p xs)
;;
```

Filter step-by-step

The first step must be lazy as well:

```
let rec filter p l = match force l with
  | Nil -> Nil
  | Cons(x,xs) -> if p x then Cons(x,lazy (filter p
                            else (filter p xs)
and lfilter p l = lazy (filter p l)
;;
```

Filter step-by-step

This is the same as:

```
let rec filter p l = match force l with
  | Nil -> Nil
  | Cons(x,xs) -> if p x then Cons(x,lfilter p xs)
                          else (filter p xs)
and lfilter p l = lazy (filter p l)
;;
```

Homework

- Lazy version of @ (lappend).
- Lazy version of concat (llconcat).
- Length of a lazy list (llength).
- Submission by email for grading is optional.

Guarded Recursion.

- A recursive call is guarded it it occurs as the argument of a constructor.
- Performance of list producing functions (map,filter,etc.):

list type	tail recursion	guarded recursion	problem
normal	good	bad	stack overflow
lazy	bad	good	tail recursion runs to completion before returning

- For element producing functions (e.g. length) tail recursion is best.