# Coq Summer School, Session 2 : Basic programming with numbers and lists

Pierre Letouzey, Pierre Castéran
Paris, Beijing

# Predefined data structures

- "Predefined" types are actually declared to Coq at load time [1]:

```
Inductive bool := true | false.

Inductive nat := O : nat | S : nat -> nat.

Inductive list A :=
 | nil : list A
 | cons : A -> list A -> list A.
```

Nota: `a::b` is a notation for `(cons a b)`.

[1] see Init/Datatypes.v

# Constructors

- `true` and `false` are the *constructors* of type `bool`
- `O` and `S` are the *constructors* of type `nat`
- `nil` and `cons` are the *constructors* of type `list` $A$

```
Check S (S (S O)).
```

*3 : nat*

```
Require Import List.
Check cons 2 (cons 3 (cons 5 (cons 7 nil))).
```

*2 :: 3 :: 5 :: 7 :: nil      : list nat*

```
Check plus :: mult :: minus :: nil.
```

*plus :: mult :: minus :: nil    : list (nat -> nat -> nat)*

# Pattern matching

- Any boolean is either `true` or `false`. Thus, we can analyse an expression and handle all possible cases:

```
Definition negb b :=
 match b with
  | true => false
  | false => true
 end.
```

- Most common situation: one pattern for each constructor.
- **Note:** for `bool`, an alternative syntactic sugar is `if b then false else true`.

```
Eval vm_compute in negb true.
```

*= false : bool*

```
Eval vm_compute in negb (negb true).
```

*= true : bool*

# Pattern matching

- Similarly, for natural numbers:

```
Definition pred (n:nat) :=
 match n with
  | S p => p
  | O => O
 end.

Definition iszero n :=
 match n with
  | O => true
  | S _ => false
 end.
```

# Pattern matching

- Similarly, for natural numbers:

```
Definition pred (n:nat) :=
 match n with
  | S p => p
  | O => O
 end.

Definition iszero n :=
 match n with
  | O => true
  | S _ => false
 end.

eval vm_compute in pred 43.
```

*= 42 : nat*

```
Eval vm_compute in is_zero (pred 0).
```

*= true : bool*

# More complex pattern matching

- We can use deeper patterns, combined matchings, as well as wildcards:

```
Definition istwo n :=
 match n with
  | S (S O) => true
  | _ => false
 end.

Definition andb b1 b2 :=
 match b1, b2 with
  | true, true => true
  | _, _ => false
 end.
```

Such complex matchings are not atomic, but rather expansed internally into nested matchings:

```
Print andb.
andb =
fun b1 b2 : bool =>
   if b1
   then if b2
        then true
        else false
    else false
     : bool -> bool -> bool
```

```
Print istwo.
istwo =
fun x : nat =>
match x with
| 0 => false
| 1 => false
| 2 => true
| S (S (S _)) => false
end
     : nat -> bool
```

# Recursion

- When using `Fixpoint` instead of `Definition`, recursive sub-calls are allowed (at least some of them).

```
Fixpoint div2 n :=
 match n with
  | S (S n') => S (div2 n')
  | _ => O
 end.
```

# Recursion

- When using `Fixpoint` instead of `Definition`, recursive sub-calls are allowed (at least some of them).

```
Fixpoint div2 n :=
 match n with
  | S (S n') => S (div2 n')
  | _ => O
 end.
```

- Here, `n'` represents a *structural sub-term* of the inductive argument `n`. For instance, if `n` is bound to 7, *i.e.* the term S (S (S (S (S (S (S O)))))), then `n'` is bound to 5, *i.e.* S (S (S (S (S 0)))), which is a subterm of the former one. This way, termination of computations is (syntactically) ensured.

# Three examples of badly written Fixpoint definitions

```
Fixpoint loop n := loop (S n).
(* BAD *)
```

# Three examples of badly written Fixpoint definitions

```
Fixpoint loop n := loop (S n).
(* BAD *)

Fixpoint loop n := loop n.
(* BAD *)
```

# Three examples of badly written Fixpoint definitions

```
Fixpoint loop n := loop (S n).
(* BAD *)

Fixpoint loop n := loop n.
(* BAD *)

Fixpoint log2 (n : nat) : nat :=
 match n with 0 => 0
            | 1 => 0
            | p => S (log2 (Div2.div2 p))
 end. (* BAD *)
```

**Note:** There are other ways to define recursive functions like `log2` in coq, but we will need to make interactive proofs.

In general, you may write recursive calls on variables introduced by pattern matchings.

```
Fixpoint foo (n:nat) : nat :=
 match n with 0 => 0
            | 1 => 0
            | S (S p) => 1 + foo (S p)
 end.(* BAD *)
```

In general, you may write recursive calls on variables introduced by pattern matchings.

```
Fixpoint foo (n:nat) : nat :=
 match n with 0 => 0
            | 1 => 0
            | S (S p) => 1 + foo (S p)
 end.(* BAD *)

Fixpoint foo (n:nat) : nat :=
 match n with 0 => 0
            | 1 => 0
            | S (S p as q) => 1 + foo q
 end.(* GOOD *)
```

# Some other recursive functions over nat

```
Fixpoint plus n m :=
 match n with
  | O => m
  | S n' => S (plus n' m)
 end.
```

```
Notation : n + m for plus n m
```

# Some other recursive functions over nat

```
Fixpoint plus n m :=
 match n with
  | O => m
  | S n' => S (plus n' m)
 end.

Notation : n + m for plus n m

Fixpoint minus n m := match n, m with
  | S n', S m' => minus n' m'
  | _, _ => n
 end.
Notation : n - m for minus n m
```

```
Fixpoint mult (n m :nat) : nat :=
match n with
| O => O
| S p => m + mult p m
end.
Notation : n * m for mult n m

Fixpoint beq_nat n m := match n, m with
  | S n', S m' => beq_nat n' m'
  | O, O => true
  | _, _ => false
 end.
```

## Recursion over lists

- With recursive functions over lists, the main novelty is *polymorphism* :

```
Fixpoint length A (l : list A) :=
 match l with
  | nil => 0
  | _ :: l' => S (length l')
 end.
```

# Recursion over lists

- With recursive functions over lists, the main novelty is *polymorphism* :

```
Fixpoint length A (l : list A) :=
 match l with
  | nil => 0
  | _ :: l' => S (length l')
 end.

Fixpoint app A (l1 l2 : list A) : list A :=
 match l1 with
  | nil => l2
  | a :: l1' => a :: (app l1' l2)
 end.
```

- NB: `(app l1 l2)` is noted `l1++l2`.

# Applying a function to every element of a list

```
Fixpoint map A B (f : A -> B)(l : list A) : list B :=
 match l with
  | nil => nil
  | a::l' => f a :: map f l'
 end.

Eval compute in map (fun n => n * n) (1::2::3::4::5::nil).
```

*1::4::9::16::25::nil : list nat*

# Applying a function to every element of a list

```
Fixpoint map A B (f : A -> B)(l : list A) : list B :=
 match l with
  | nil => nil
  | a::l' => f a :: map f l'
 end.

Eval compute in map (fun n => n * n) (1::2::3::4::5::nil).
```

*1::4::9::16::25::nil : list nat*

# Reversing a list

First version:

```
Set Implicit Arguments.

Fixpoint naive_reverse (A:Type)(l: list A) : list A :=
 match l with
  | nil => nil
  | a::l' => naive_reverse l' ++ (a::nil)
 end.

Eval compute in naive_reverse (1::2::3::4::5::6::nil).
```

*= 6 :: 5 :: 4 :: 3 :: 2 :: 1 :: nil*
*: list nat*

# Why "naïve_reverse" ?

Problem: a lot of recursive calls to app :

```
nil ++ (6::nil) 1 calls
(6::nil) ++ (5:: nil)  2 calls
(6::5::nil) ++ (4::nil)  3 calls
...
(6::5::4::3::2::nil)++(1::nil) 6 calls
```

$n(n+1)/2$ recursive calls ($n$ being the list's length) !

# A more efficient function

```
Fixpoint rev_app (A:Type)(l l1: list A) : list A :=
  match l with
   | nil => l1
   | a::l' => rev_app l' (a::l1)
  end.

Eval compute in rev_app (4::5::6::nil) (3::2::1::nil).
```

*= 6::5::4::3::2::1::nil*

```
Definition rev A (l:list A) := rev_app l nil.
```

# Same approach, with a local recursive function

```
Definition rev'  A (l:list A) :=
  (fix  aux (l1 l2: list A) :=
     (* appends the reverse of l1 to l2 *)
     match l1 with
      | nil => l2
      | a::l'1 => aux l'1 (a::l2)
     end) l nil.
```

# Fold on the right

- The intention:
  `fold_right f init (a::b::...::z::nil)`
  `= (f a (f b (...(f z init))))`
- The code:

```
Fixpoint fold_right A B (f:B->A->A)(init:A)(l:list B)
 : A :=
 match l with
  | nil => init
  | x :: l' => f x (fold_right f init l')
 end.
```

```
Eval vm_compute in fold_right mult 1 (1::2::3::4::nil).
```

*= 24 : nat*

```
Eval vm_compute in
 map (fold_right mult 1)
     ((1::2::3)::(4::nil)::nil::(4::2::3::nil)).
```

*= 6::4::1::24::nil : list nat*

# Yet another example of ill-formed recursive definition

Merging two sorted lists.

```
Fixpoint merge (u v : list nat) : list nat :=
 match u,v with
  nil,v => v
| u,nil => u
| a::u', b::v' => if leb a b
                           then a::(merge u' v)
                           else b::(merge u v')
 end.
```

# Yet another example of ill-formed recursive definition

Merging two sorted lists.

```
Fixpoint merge (u v : list nat) : list nat :=
 match u,v with
  nil,v => v
| u,nil => u
| a::u', b::v' => if leb a b
                          then a::(merge u' v)
                          else b::(merge u v')
 end.
```

*Error: Cannot guess decreasing argument of fix.*

# A first solution

```
Fixpoint merge_aux (n:nat) (u v : list nat)
  : list nat :=
 match n,u,v with
  | 0,_,_=> nil
  | S _, nil,v => v
  | S _, u,nil => u
  | S p, a::u', b::v' =>
    if leb a b then a::(merge_aux p u' v)
               else b::(merge_aux p u v')
 end.
```

# A first solution

```
Fixpoint merge_aux (n:nat) (u v : list nat)
  : list nat :=
 match n,u,v with
  | O,_,_=> nil
  | S _, nil,v => v
  | S _, u,nil => u
  | S p, a::u', b::v' =>
    if leb a b then a::(merge_aux p u' v)
               else b::(merge_aux p u v')
 end.

Definition merge u v :=
  merge_aux (length u + length v) u v.

Eval compute in merge (1::3::5::nil) (1::2::2::6::nil).
```

*1::1::2::2::3::5::6::nil : list nat*

# Remarks

- This solution is not fully satisfactory (because of extra computations).
- Other solutions exist, relying on interactive proofs. See documentation on Function.
- **Extra exercise:** define a polymorphic version of merge.