Predefined data structures

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

  ```coq
  Inductive bool := true | false.
  Inductive nat := 0 : 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).
  - Every natural number is data constructed with S and 0.
  ```

Constructors

- true and false are the constructors of type bool
- 0 and S are the constructors of type nat
- nil and cons are the constructors of type list A for any type A

Check S (S 0).
3 : nat

Open Scope list_scope.
Check cons 2 (cons 3 (cons 5 (cons 7 nil))).
2 :: 3 :: 5 :: 7 :: nil : list nat

Pattern matching

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

  ```coq
  Definition negb b :=
  match b with
  | true => false
  | false => true
  end.
  
  Nota: for bool, an alternative syntactic sugar is
  if b then false else true.
  ```

Compute negb true.
= false : bool

Compute negb (negb true).
= true : bool

Pattern matching

- Similarly, for lists

  ```coq
  Definition tail (A : Type) (l:list A) :=
  match l with
  | x::tl => tl
  | nil => nil
  end.
  
  Definition isempty (A : Type) (l : list A) :=
  match l with
  | nil => true
  | _ :: _ => false
  end.
  ```
Pattern matching

- Similarly, for lists
  ```coq
  Definition tail (A : Type) (l:list A) :=
  match l with
  | x::tl => tl
  | nil => nil
  end.
  ```

  ```coq
  Definition isempty (A : Type) (l : list A) :=
  match l with
  | nil => true
  | _ :: _ => false
  end.
  ```

  Compute tail nat (1::2::3::nil).
  = 2::3::nil : list nat
  Compute isempty nat (1::nil).
  = false : bool

More complex pattern matching

- We can use deeper patterns, combined matchings, as well as wildcards:
  ```coq
  Definition has_two_elts (A : Type) (l : list A) :=
  match l with
  | _ :: _ :: nil => true
  | _ => false
  end.
  ```

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

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

- Print has_two_elts.
  has_two_elts =
  fun (A : Type) (l : list A) =>
  match l with
  | nil => false
  | _ :: nil => false
  | _ :: _ :: nil => true
  | _ :: _ :: _ :: _ => false
  end
  :forall A : Type, list A -> bool

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

  Note also that "if-then-else" is just a pattern-matching construct.

Recursion

- When using Fixpoint instead of Definition, recursive sub-calls are allowed (at least some of them).
  ```coq
  Fixpoint every_other (A : Type) (l : list A):=
  match l with
  | _ :: a :: l' => a :: every_other l'
  | _ => nil
  end.
  ```

Recursion

- When using Fixpoint instead of Definition, recursive sub-calls are allowed (at least some of them).
  ```coq
  Fixpoint every_other (A : Type) (l : list A):=
  match l with
  | _ :: a :: l' => a :: every_other l'
  | _ => nil
  end.
  ```

- Here, l' represents a structural sub-term of the inductive argument l. For instance, if l is bound to 1 :: 2 :: 3 :: nil, then n' is bound to 3 :: nil, which is a subterm of the former one. This way, termination of computations is ensured.
Three examples of badly written Fixpoint definitions

Fixpoint loop l := loop (1 :: l).
(* BAD *)

Fixpoint loop (A : Type) (l : list A) := loop l.
(* BAD *)

Fixpoint log_like (l : list nat) :=
  match l with
  | nil => nil
  | a::nil => nil
  | a::l' => a::log_like (every_other (a::l'))
end.

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

Fixpoint foo (l:list nat) : nat :=
  match l with
  | nil => 0
  | a::nil => 0
  | a::b::l' => a + foo (b::l')
end.(* BAD *)

Fixpoint foo (l:list nat) : nat :=
  match l with
  | nil => 0
  | a::nil => 0
  | a::(b::l' as l2) => a + foo l2
end.(* GOOD *)

Pattern matching and recursion over nat

- nat is an inductive type, with two constructor 0 and S
- every natural number is either of the form S p where p is another natural number, or 0
- Pattern-matching expressions analyze according to these cases
- The numeric notation is misleading: when you see 3 the system handles S (S (S O))
- the display engine creates the number
- The function S is not a complex operation, just a constructor
Simple functions over nat

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

Fixpoint div2 (n : nat) :=
    match n with
    | S (S p) => S (div2 p)
    | _ => 0
end.

Fixpoint fact (n : nat) :=
    match n with
    | 0 => 1
    | S p => n * fact p
end.

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) :=
    match n with
    | O => O
    | S p => m + mult p m
end.

Notation : n*m for mult n m

Fixpoint beq_nat nm := 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 => O
    | a :: l' => S (length l')
end.

Recursion over lists

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

  Fixpoint length A (1 : list A) :=
    match 1 with
    | nil => 0
    | _ :: 1' => S (length 1')
end.

Fixpoint app A (11 12 : list A) : list A :=
    match 11 with
    | nil => 12
    | a :: 11' => a :: (app 11' 12)
end.

- NB: (app 11 12) is noted 11++12.
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' => aux l'1 (a::l2)
end) l nil.
Fold on the right

The intention:
\[
\text{fold_right } f \text{ init } (a::b::...::z::\textbf{nil}) = (f a (f b (...(f z \text{ init}))))
\]

The code:
\[
\text{Fixpoint } fold\_right : B \rightarrow A \rightarrow (\text{list } B) \rightarrow A =
\begin{align*}
\text{match } l \text{ with} \\
\mid \text{nil} & \Rightarrow \text{init} \\
\mid x :: l' & \Rightarrow f x (f \text{ fold_right } f \text{ init } l')
\end{align*}
\]

Yet another example of ill-formed recursive definition

Merging two sorted lists.

Fixpoint merge : list nat \rightarrow list nat :=
\[
\begin{align*}
\text{match } u, v \text{ with} \\
\mid \text{nil}, v & \Rightarrow v \\
\mid u, \text{nil} & \Rightarrow u \\
\mid a::u', b::v' & \Rightarrow \text{if } \text{leb } a b \\
\mid \text{then } a::(\text{merge } u' v) \\
\mid \text{else } b::(\text{merge } u v')
\end{align*}
\]

A first solution

Fixpoint merge_aux : list nat \rightarrow list nat :=
\[
\begin{align*}
\text{match } n, u, v \text{ with} \\
\mid 0, _, _, & \Rightarrow \text{nil} \\
\mid S n, \text{nil}, v & \Rightarrow v \\
\mid S n, u, \text{nil} & \Rightarrow u \\
\mid S p, a::u', b::v' & \Rightarrow \\
\mid \text{if } \text{leb } a b & \text{ then } a::(\text{merge_aux } p u' v) \\
\mid \text{else } b::(\text{merge_aux } p u v')
\end{align*}
\]

Definition merge uv :=
\[
\text{merge_aux (length } u + \text{ length } v) u v.
\]

Eval compute in merge (1::3::5::\textbf{nil}) (1::2::2::6::\textbf{nil}).
\[
1::1::2::2::3::5::6::\textbf{nil} : \text{list nat}
\]
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.