



软件理论基础与实践

# Lists: Working with Structured Data

熊英飞  
北京大学



# 本节内容

- 采用前面教授的知识定义一系列常用数据结构
- 这些数据结构也是函数式程序设计语言的标准数据结构
- 这些数据结构将在后续课程中反复使用



# Pair

```
Inductive natprod : Type :=  
  | pair (n1 n2 : nat).
```

```
Check (pair 3 5) : natprod.
```

```
Definition fst (p : natprod) : nat :=  
  match p with  
  | pair x y => x  
  end.
```

```
Definition snd (p : natprod) : nat :=  
  match p with  
  | pair x y => y  
  end.
```

```
Compute (fst (pair 3 5)).  
(* ==> 3 *)
```



# Pair

Notation `"( x , y )"` := `(pair x y)`.

Compute `(fst (3,5))`.

```
Definition fst' (p : natprod) : nat :=  
  match p with  
  | (x,y) => x  
  end.
```

```
Definition swap_pair (p : natprod) : natprod :=  
  match p with  
  | (x,y) => (y,x)  
  end.
```



# 注意区分Pair和多参数匹配

```
Fixpoint minus (n m:nat) : nat :=  
  match n, m with  
  | 0 , _      => 0  
  | S _ , 0    => n  
  | S n' , S m' => minus n' m'  
  end.
```

```
Definition bad_minus (n m : nat) : nat :=  
  match n, m with  
  | (0 , _ ) => 0  
  | (S _ , 0 ) => n  
  | (S n' , S m' ) => bad_minus n' m'  
  end.
```



# Lists

```
Inductive natlist : Type :=  
  | nil  
  | cons (n : nat) (l : natlist).
```

```
Definition mylist := cons 1 (cons 2 (cons 3 nil)).
```

```
Notation "x :: l" := (cons x l)  
  (at level 60, right associativity).
```

```
Notation "[ ]" := nil.
```

```
Notation "[ x ; .. ; y ]" := (cons x .. (cons y nil) ..).
```

```
Definition mylist1 := 1 :: (2 :: (3 :: nil)).
```

```
Definition mylist2 := 1 :: 2 :: 3 :: nil.
```

```
Definition mylist3 := [1;2;3].
```

```
Definition mylist4 := 1 :: 1 + 1 :: 3 :: nil.
```

优先级低  
于加减



# 常用函数

```
Fixpoint repeat (n count : nat) :  
natlist :=  
  match count with  
  | 0 => nil  
  | S count' => n :: (repeat n count')  
  end.
```

```
Fixpoint length (l:natlist) : nat :=  
  match l with  
  | nil => 0  
  | h :: t => S (length t)  
  end.
```



# 常用函数

```
Fixpoint app (l1 l2 : natlist) : natlist :=  
  match l1 with  
  | nil      => l2  
  | h :: t => h :: (app t l2)  
  end.
```

Notation "x ++ y" := (app x y)  
 (right associativity, at level 60).

```
Example test_app1: [1;2;3] ++ [4;5] = [1;2;3;4;5].  
Proof. reflexivity. Qed.  
Example test_app2: nil ++ [4;5] = [4;5].  
Proof. reflexivity. Qed.  
Example test_app3: [1;2;3] ++ nil = [1;2;3].  
Proof. reflexivity. Qed.
```





# 常用函数

```
Definition hd (default : nat) (l : natlist) : nat :=  
  match l with  
  | nil => default  
  | h :: t => h  
  end.
```

```
Definition tl (l : natlist) : natlist :=  
  match l with  
  | nil => nil  
  | h :: t => t  
  end.
```

```
Example test_hd1:  
Proof. reflexivity. Qed.
```

```
Example test_hd2:  
Proof. reflexivity. Qed.
```

```
Example test_tl:  
Proof. reflexivity. Qed.
```

```
hd 0 [1;2;3] = 1.
```

```
hd 0 [] = 0.
```

```
tl [1;2;3] = [2;3].
```



# 证明列表的性质

**Theorem** `nil_app` : forall l : natlist,  
 [] ++ l = l.

**Proof.** `reflexivity. Qed.`

**Theorem** `tl_length_pred` : forall l:natlist,  
 pred (length l) = length (tl l).

**Proof.**

`intros l. destruct l as [| n l'].`

`- (* l = nil *)`

`reflexivity.`

`- (* l = cons n l' *)`

`reflexivity. Qed.`

为什么可以不用  
induction?



# 证明列表的性质

**Theorem** `nil_app` : forall l : natlist,  
 [] ++ l = l.

**Proof.** `reflexivity. Qed.`

**Theorem** `tl_length_pred` : forall l:natlist,  
 pred (length l) = length (tl l).

**Proof.**

```
intros l. destruct l as [| n l'].
```

```
- (* l = nil *)
```

```
  reflexivity.
```

```
- (* l = cons n l' *)
```

```
  (* Nat.pred (length (n :: l')) = length (tl (n :: l')) *)
```

```
  simpl.
```

```
  (* length l' = length l' *)
```

```
  reflexivity. Qed.
```



# 采用归纳证明列表的性质

**Theorem** `app_assoc` : forall l1 l2 l3 : natlist,  
 (l1 ++ l2) ++ l3 = l1 ++ (l2 ++ l3).

**Proof.**

```
intros l1 l2 l3. induction l1 as [| n l1' IHl1'].  
- (* l1 = nil *)  
  reflexivity.  
- (* l1 = cons n l1' *)  
  (* IHl1': (l1' ++ l2) ++ l3 = l1' ++ l2 ++ l3 *)  
  (* Goal: ((n :: l1') ++ l2) ++ l3 = (n :: l1') ++ l2 ++ l3 *)  
  simpl. rewrite -> IHl1'. reflexivity. Qed.
```

对l1做归纳不  
影响l2和l3



# 倒转列表

```
Fixpoint rev (l:natlist) : natlist :=  
  match l with  
  | nil      => nil  
  | h :: t => rev t ++ [h]  
  end.
```

```
Example test_rev1:  
Proof. reflexivity. Qed.
```

```
Example test_rev2:  
Proof. reflexivity. Qed.
```

```
rev [1;2;3] = [3;2;1].
```

```
rev nil = nil.
```



# 证明倒转列表的性质

**Theorem** `rev_length` : forall l : natlist,  
length (rev l) = length l.

**Proof.**

```
intros l. induction l as [| n l' IHl'].  
(* [Coq Proof View]  
* 2 subgoals  
*  
* =====  
* length (rev [ ]) = length [ ]  
*  
* subgoal 2 is:  
* length (rev (n :: l')) = length (n :: l')  
*)
```



# 证明倒转列表的性质

```
- reflexivity.
- (** [Coq Proof View]
* 1 subgoal
*
*   n : nat
*   l' : natlist
*   IHl' : length (rev l') = length l'
*   =====
*   length (rev (n :: l')) = length (n :: l')
*)
  simpl.
(** length (rev l' ++ [n]) = S (length l') *)
  rewrite -> IHl'.
(** length (rev l' ++ [n]) = S (length (rev l')) *)
```



# 证明辅助定理

**Theorem** `app_length` : forall l1 l2 : natlist,  
length (l1 ++ l2) = (length l1) + (length l2).

**Proof.**

```
intros l1 l2. induction l1 as [| n l1' IHl1'].  
- (* l1 = nil *)  
  reflexivity.  
- (* l1 = cons *)  
  simpl. rewrite -> IHl1'. reflexivity. Qed.
```





# 证明倒转列表的性质

**Theorem** `rev_length` : forall l : natlist,  
length (rev l) = length l.

**Proof.**

```
intros l. induction l as [| n l' IHl'].  
- (* l = nil *)  
  reflexivity.  
- (* l = cons *)  
  simpl. rewrite -> app_length.  
  simpl. rewrite -> IHl'. rewrite add_comm.  
  reflexivity.
```

**Qed.**



# 搜索定理

- 按名称搜索：
  - Search rev.
  - 输出：
    - test\_rev2: rev [ ] = [ ]
    - rev\_length: forall l : natlist, length (rev l) = length l
    - test\_rev1: rev [1; 2; 3] = [3; 2; 1]
- 按定理形式搜索：
  - Search ( \_ + \_ = \_ + \_ ).
- 限定搜索的模块：
  - Search ( \_ + \_ = \_ + \_ ) inside Induction.
- 按变量模式匹配：
  - Search (?x + ?y = ?y + ?x).



# Options: 处理例外情况

- 图灵奖Tony Hoare: 我发明Null是一个错误, 造成十亿美元的损失
- Null的问题: 不处理null值编译器也不报警
- 如何让编译器报警?

```
Inductive natoption : Type :=  
  | Some (n : nat)  
  | None.
```



# Options: 处理例外情况

```
Fixpoint nth (l:natlist) (n:nat) : natoption :=
  match l with
  | nil => None
  | a :: l' => match n with
                | 0 => Some a
                | S n' => nth l' n'
              end
  end.
```

```
Definition option_elim (d : nat) (o : natoption) :
nat :=
  match o with
  | Some n' => n'
  | None => d
  end.
```



# Partial Map

```
Inductive id : Type :=  
  | Id (n : nat).
```

```
Inductive partial_map : Type :=  
  | empty  
  | record (i : id) (v : nat) (m : partial_map).
```

```
Definition update (d : partial_map)  
  (x : id) (value : nat)  
  : partial_map :=  
  record x value d.
```



# Partial Map

```
Definition eqb_id (x1 x2 : id) :=  
  match x1, x2 with  
  | Id n1, Id n2 => n1 =? n2  
  end.
```

```
Fixpoint find (x : id) (d : partial_map) : natoption :=  
  match d with  
  | empty          => None  
  | record y v d' => if eqb_id x y  
                     then Some v  
                     else find x d'  
  end.
```



# 作业

- 完成Lists.v中standard非optional的11道习题
  - 请使用最新英文版教材
  - 下下周一之前提交