# 「計算と論理」
# Software Foundations
# その3

五十嵐 淳
cal15@fos.kuis.kyoto-u.ac.jp

京都大学

October 27, 2015

# Lists.v

- 自然数のペア (ふたつ組)
- 自然数リスト
- リストに関する推論
- オプション型
- 「辞書」のデータ表現 (省略)

# 自然数のペア

引数がふたつ(以上)のコンストラクタを使った型定義

```
Inductive natprod : Type :=
| pair : nat → nat → natprod.
```

- コンストラクタがひとつだけの型
- pair: 自然数ふたつをとって natprod を作る
  - `pair 1 2         : natprod`
  - `pair (4 + 3) 2 : natprod`
  - natprod 型の式(の値)は必ず `pair ... ...` の形をしている
- product ⋯ (集合の)デカルト積

# 射影: 要素の取り出し関数

```
Definition fst (p : natprod) : nat :=
  match p with
  | pair x y => x (* パターンの新記法! *)
  end.
Definition snd (p : natprod) : nat :=
  match p with
  | pair x y => y
  end.
```

- fst ··· 第一射影 (first projection)
- snd ··· 第二射影 (second projection)

# Notationによる見慣れた表記の導入

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

Definition fst' (p : natprod) : nat :=
  match p with
  | (x,y) => x  (* パターンでも使える! *)
  end.

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

# ペアに関する簡単な性質の証明

**定理: Surjectivity of pairing**

任意のペアは，その第一射影と第二射影の組と等しい (すなわち，組を作る操作は全射になっている．)

**Coq による表現その 1**

```
Theorem surjective_pairing' :
  forall (n m : nat),
    (n,m) = (fst (n,m), snd (n,m)).
Proof. reflexivity. Qed.
```

# より自然な表現

## その2

```
Theorem surjective_pairing :
  forall (p : natprod), p = (fst p, snd p).
Proof.
  intros p. destruct p as [n m]. reflexivity.
Qed.
```

- ひとつしかないけれど場合分け
  - natprod なら必ず組の形 (n,m) をしている
- 変数を複数導入するイントロパターン

# Lists.v

- 自然数のペア (ふたつ組)
- 自然数リスト
- リストに関する推論
- オプション型
- 「辞書」のデータ表現 (省略)

# リストとは？

「もの」(要素) を一列に並べたような集まりを表すデータ

リストの作り方:

- 空リスト (`nil`) ← 全てのリストの種 (たね)
- 既存のリストの先頭へ要素を追加する (`cons`)

# 自然数リストの型定義

```
Inductive natlist : Type :=
  | nil : natlist
  | cons : nat -> natlist -> natlist.
```

(自然数) リストの作り方:

- 空リスト (`nil`) はリストである
- 自然数 `n` を自然数リスト `l` の先頭に追加したもの (`cons n l`) はリストである

自然数との構造の類似に注意!

# リスト表記

- cons の代わりの右結合中置演算子 **n :: l**
- 要素を列挙する表記 [**n**; **m**; ...]
  - .v を直接読むとちょっと紛らわしい

以下は全て同じリストを定義している:

```
Definition mylist1 := 1 :: (2 :: (3 :: nil)).
Definition mylist2 := 1 :: 2 :: 3 :: nil.
Definition mylist3 := [1;2;3].
```

# リスト操作関数(1): repeat

**n** が **count** 個並んだリスト

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

参考:

```
(define (repeat n count)
  (if (= count 0) '()
      (cons n (repeat n (- count 1)))))
```

# リスト操作関数(2): length

リストの長さ:

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

参考:

```
(define (length l)
  (if (null? l) 0
      (+ 1 (length (cdr l)))))
```

# リストを消費する関数を定義するコツ

- プログラムを書く前に，入力例を沢山考えて，それぞれ出力が何になるべきかを理解する
  - 教科書であれば `Example` が提供されていることも
- 基本は `nil` の場合と `h :: t` の場合分け
  - リスト引数が複数ある場合，どちらで場合分けをするか悩ましいことがある $\Longrightarrow$ 色々な可能性を探る
- `h :: t` の場合，`t` に対して再帰呼び出しをした結果 (の意味) をプログラムは見ないでよく考える

# リスト操作関数(3): app(end)

リストの連結

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

参考:

```
(define (append l1 l2)
  (if (null? l1) l2
      (cons (car l1) (append (cdr l1) l2))))
```

## `app l1 l2` の(右結合)中置記法: `l1 ++ l2`

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

# リスト操作関数(4): hd, tl

```
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.
```

- 引数が nil の場合もエラーにできないので適当な値 (default) を返す

# Lists.v

- 自然数のペア (ふたつ組)
- 自然数リスト
- リストに関する推論
- オプション型
- 「辞書」のデータ表現 (省略)

# リスト vs 自然数

```
Inductive natlist : Type :=
  | nil : natlist
  | cons : nat -> natlist -> natlist.
```

と

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

- 要素を無視して，構造だけ見れば同じ!

# リスト vs 自然数 (2)

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

と

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

# 単純化による証明

```
Theorem nil_app : forall l:natlist,
  [] ++ l = l.
Proof.
  intros l.  reflexivity.  Qed.
```

自然数の足し算と同じで以下はそう簡単ではない.

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

# 場合わけによる証明

```
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.
```

- イントロパターンで `l = n :: l'` を表現している

# リストに関する帰納法

**P(l)** を (自然数) リスト **l** について述べた命題とする

## リストに関する帰納法の原理

「任意のリスト **l** について **P(l)**」は以下と同値

- **P**(`nil`) かつ
- 任意の自然数 **n**, リスト **l′** について **P(l′)** ならば **P(n : : l′)**

単なる場合分けと違って，**P(n : : l′)** を示すのに，ひとつ短かいリストでは **P** が成立していること (つまり **P(l′)**) を仮定してよい

- **P(l′)** を「帰納法の仮定」(induction hypothesis, IH) と呼ぶ

# 復習・比較: 数学的帰納法

**P(n)** を自然数の性質について述べた命題とする

## 数学的帰納法の原理

「任意の自然数 **n** について **P(n)**」は以下と同値

- **P(0)** かつ
- 任意の自然数 **n′** について **P(n′)** ならば **P(S n′)**

単なる場合分けと違って，**P(S n′)** を示すのに，ひとつ小さい数では **P** が成立していること (つまり **P(n′)**) を仮定してよい

- **P(n′)** を「帰納法の仮定」(induction hypothesis, IH) と呼ぶ

# ++ の結合律

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

- 足し算の結合律の証明と比較してみよう!

# app の結合律の日本語による証明

**定理: 任意の l1, l2, l3 について (l1 ++ l2) ++ l3 = l1 ++ (l2 ++ l3) である**

証明: l1 についての帰納法.

- l1 = [] とする.

$$([]\ \texttt{++}\ l2)\ \texttt{++}\ l3 = []\ \texttt{++}\ (l2\ \texttt{++}\ l3)$$

を示す必要があるが，これは ++ の定義より明らか.

- $\mathbf{l1} = \mathbf{n}::\mathbf{l1'}$ ただし,

$$(\mathbf{l1'}\ \mathbf{++}\ \mathbf{l2})\ \mathbf{++}\ \mathbf{l3} = \mathbf{l1'}\ \mathbf{++}\ (\mathbf{l2}\ \mathbf{++}\ \mathbf{l3})$$

とする.

$$((\mathbf{n}::\mathbf{l1'})\ \mathbf{++}\ \mathbf{l2})\ \mathbf{++}\ \mathbf{l3} = (\mathbf{n}::\mathbf{l1'})\ \mathbf{++}\ (\mathbf{l2}\ \mathbf{++}\ \mathbf{l3})$$

を示す必要があるが，++ の定義より，これは

$$\mathbf{n}::((\mathbf{l1'}\ \mathbf{++}\ \mathbf{l2})\ \mathbf{++}\ \mathbf{l3}) = \mathbf{n}::(\mathbf{l1'}\ \mathbf{++}\ (\mathbf{l2}\ \mathbf{++}\ \mathbf{l3}))$$

と同値．これは帰納法の仮定より明らか．(証明終)

# 数学的帰納法による証明の雛形

## 定理: 任意の自然数 n について P(n)

証明: n に関する数学的帰納法による.

- $n = 0$ の場合:

  ⋯⋯ P(0) の証明 ⋯⋯

- $n = S(n')$ の場合，ただし $P(n')$ とする:

  ⋯⋯ $P(S(n'))$ の証明 ⋯⋯
  (⋯帰納法の仮定より⋯)

# リストに関する帰納法による証明の雛形

**定理: 任意のリスト l について P(l)**

証明: l に関する帰納法による.

- $l = []$ の場合:

  …… P([]) の証明 ……

- $l = n :: l'$ の場合，ただし $P(l')$ とする:

  …… $P(n :: l')$ の証明 ……
  (…帰納法の仮定より…)

# 例をもうひとつ

```
Theorem app_length : forall l1 l2 : natlist,
  length (l1 ++ l2) = (length l1) + (length l2).
Proof.
  intros l1 l2. induction l1 as [| n l1'].
  - (* l1 = nil *)
    reflexivity.
  - (* l1 = cons n l1' *)
    simpl. rewrite -> IHl1'. reflexivity.
Qed.
```

# もう少し複雑な例: リストの反転

```
(* 尻尾に追加するので（cons を後ろから読んで） snoc *
Fixpoint snoc (l:natlist) (v:nat) : natlist :=
  match l with
  | nil    => [v]
  | h :: t => h :: (snoc t v)
  end.

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

```
Theorem rev_length_firsttry :
  forall l : natlist,
    length (rev l) = length l.
Proof.
  intros l. induction l as [| n l'].
  - (* l = [] *)
    reflexivity.
  - (* l = n :: l' *)
    simpl.
(* 無理っぽいゴール:
  length (snoc (rev l') n) = S (length l') *)
```

- 我々は(rev から呼ばれる)snoc に関して何も示していない!

# snoc に関する補題…

```
Theorem length_snoc :
  forall (n : nat) (l : natlist),
    length (snoc l n) = S (length l).
Proof.
  intros n l. induction l as [| n' l'].
  - (* l = nil *)
    reflexivity.
  - (* l = cons n' l' *)
    simpl. rewrite -> IHl'. reflexivity.
Qed.
```

- つまったゴールより少し一般化(?)した定理
  - c.f. snoc の第一引数

# …を使えば突破できる!

```
Theorem rev_length : forall l : natlist,
  length (rev l) = length l.
Proof.
  intros l. induction l as [| n l'].
  - (* l = nil *)
    reflexivity.
  - (* l = cons *)
    simpl. rewrite -> length_snoc.
    rewrite -> IHl'. reflexivity.  Qed.
```

# 非形式証明(ヴァージョン1)

「雛形」に沿った冗長バージョン

**補題: 任意の n と l に対し length (snoc l n) = S(length l) である.**

証明: l に関する帰納法.

- l = [] とする.

$$\text{length (snoc [] n)} = \text{S(length [])}$$

を示す必要があるが，これは **length, snoc** の定義より明らか.

- $\mathsf{l} = \mathsf{n}' :: \mathsf{l}'$ ただし,

$$\textbf{length (snoc l}'\textbf{ n)} = \textbf{S(length l}'\textbf{)}$$

とする．ここで

$$\textbf{length (snoc (n}'::\textbf{l}'\textbf{) n)} = \textbf{S(length (n}'::\textbf{l}'\textbf{))}$$

を示す必要があるが，**length, snoc** の定義より，これは

$$\textbf{S(length (snoc l}'\textbf{ n))} = \textbf{S(S(length l}'\textbf{))}$$

と同値であり，これは帰納法の仮定より明らか．(証明終)

## 定理: 任意のリスト l に対し length (rev l) = length l

証明: l についての帰納法.

- l = [] とする.

$$\textbf{length (rev [ ]) = length [ ]}$$

  を示す必要があるが，これは **rev, length** の定義より明らか.

- $\mathsf{l} = \mathsf{n} :: \mathsf{l}'$ ただし，$\mathsf{length}\ (\mathsf{rev}\ \mathsf{l}') = \mathsf{length}\ \mathsf{l}'$ とする．

$$\mathsf{length}\ (\mathsf{rev}\ (\mathsf{n} :: \mathsf{l}')) = \mathsf{length}\ (\mathsf{n} :: \mathsf{l}')$$

を示す必要があるが，$\mathsf{rev}, \mathsf{length}$ の定義より，これは

$$\mathsf{length}\ (\mathsf{snoc}\ (\mathsf{rev}\ \mathsf{l}')\ \mathsf{n}) = \mathsf{S}\ (\mathsf{length}\ \mathsf{l}')$$

と同値．前の補題より，これは

$$\mathsf{S}\ (\mathsf{length}\ (\mathsf{rev}\ \mathsf{l}')) = \mathsf{S}\ (\mathsf{length}\ \mathsf{l}')$$

と同値で，これは帰納法の仮定より明らか．

# 非形式証明(ヴァージョン2)

わかっている人向けの短縮バージョン

**定理: 任意のリスト l に対し**
**length (rev l) = length l**

まず,

$$\textbf{length (snoc l n)} = \textbf{S (length l)}$$

である(これは l に関する帰納法による)ことに注意すると，この定理は l に関する帰納法で示すことができる．特に $\mathbf{l = n :: l'}$ の場合で，上の性質を帰納法の仮定と組み合わせて使う．

どちらがいいかは状況・読み手によるが，ひとまず本当に慣れるまでは冗長なスタイルを使ってください．

# 便利コマンド: `SearchAbout`

- 前に証明した定理の名前なんて覚えていられない!
- `SearchAbout foo` とかすると `foo` に関する定理を検索してくれる!
- proofgeneral なら `C-c C-a C-a`

# Lists.v

- 自然数のペア (ふたつ組)
- 自然数リスト
- リストに関する推論
- オプション型
- 「辞書」のデータ表現 (省略)

# オプション型

「〜かもしれない型」

```
Inductive natoption : Type :=
  | Some : nat -> natoption
  | None : natoption.
```

- Some 5
- Some 42
- None
- ⋮

# オプション型の使い道

リストの **n** 番目の要素を返す関数 `index`

- **n** が大きすぎる時にどうしたらいい？

```
Fixpoint index_bad (n:nat) (l:natlist) : nat :=
  match l with
  | nil => 42  (* arbitrary! *)
  | a :: l' => match beq_nat n 0 with
               | true => a
               | false => index_bad (pred n) l'
               end
  end.
```

# オプション型を使うと…

- ふつうの返り値を示す Some
- 適当な返り値がないことを示す None

```
Fixpoint index (n:nat) (l:natlist)
  : natoption :=
  match l with
  | nil => None
  | a :: l' => match beq_nat n O with
               | true => Some a
               | false => index (pred n) l'
               end
  end.
```

# 条件式: if–then–else

```
...
  | a :: l' => if beq_nat n O then Some a
               else index (pred n) l'
...
```

- 実は bool だけでなく，コンストラクタがふたつの inductive type なら何でも使える!
  - 定義での順番依存
    - 一番目のコンストラクタなら then 節，二番目なら else 節
- パターンによる値の取り出しはできない

# Lists.v

- 自然数のペア (ふたつ組)
- 自然数リスト
- リストに関する推論
- オプション型
- 「辞書」のデータ表現 (省略)
    - Dictionary, または連想リスト (association list) のデータ構造に関する定義と練習問題

# 宿題： 11/　午前10:30 締切

- Exercise: snd_fst_is_swap (1), list_funs (2), list_exercises (3), beq_natlist (2), hd_opt (2)
- その他は随意課題
- 解答を書き込んだ Basics.v, Induction.v, Lists.v を含む zip ファイルをオンライン提出システムを通じて提出
- 以下をコメント欄に明記:
  - 講義・演習に関する質問，わかりにくいと感じたこと，その他気になること．（「特になし」はダメです．）
  - 友達に教えてもらったら、その人の名前，他の資料 (web など) を参考にした場合，その情報源 (URL など).

# 宿題のヒント

- list_exercises の rev_involutive と distr_rev は難しい!
- 補題をうまく設定するのがコツ (ゴールの意味をよく考えて!)
  - rev_involutive では rev と snoc の関係について
  - distr_rev では snoc と append の関係について