5 ソフトウェア工学
Software Engineering

構造的帰納法

STRUCTURAL INDUCTION

再帰関数
(recursive function)

関数の中からその関数自身を呼び出すような処理を行う関数.

この呼び出しの連鎖には必ず終点が必要で，引数を徐々に単純化しないといけない.

【階乗の例】

定義 $n! = 1 \times 2 \times \cdots \times n$

非再帰的な仕様
(specification)

プログラミング言語による実装
(implementation)

Haskell による再帰的定義

```
factorial        :: Int → Int
factorial 0      = 1
factorial (n+1)  = (n+1) * factorial n
```

境界条件 (boundary condition)
ここで再帰を終了

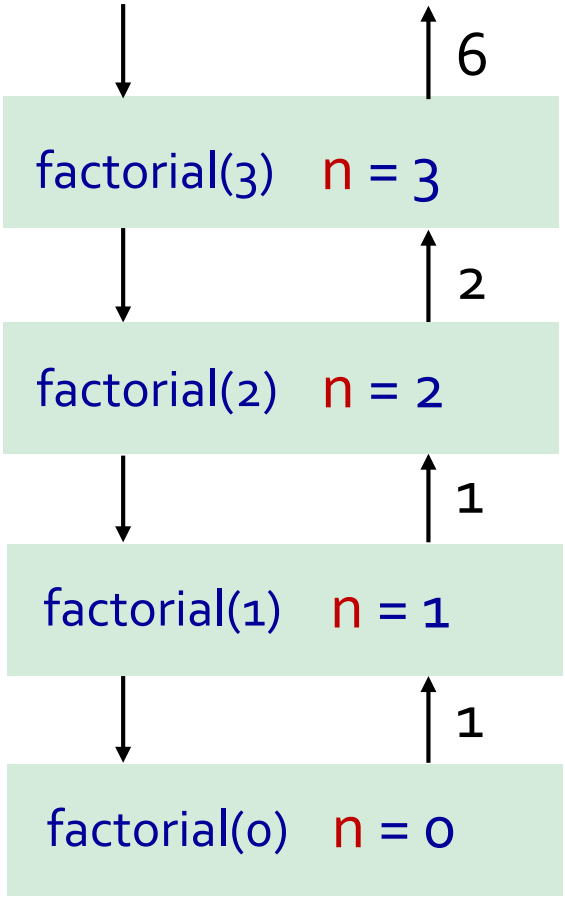
再帰呼び出し (recursive call)
factorialがfactorialを呼ぶ

注: $n = 0$ のときは, $n! = 1$. なぜなら:

$$n! = 1 \times (1 \times 2 \times \cdots \times n)$$

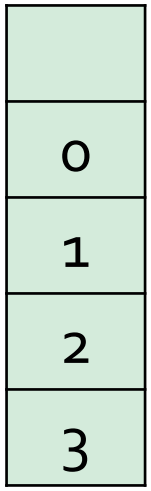
実装は仕様を満たしているのか？

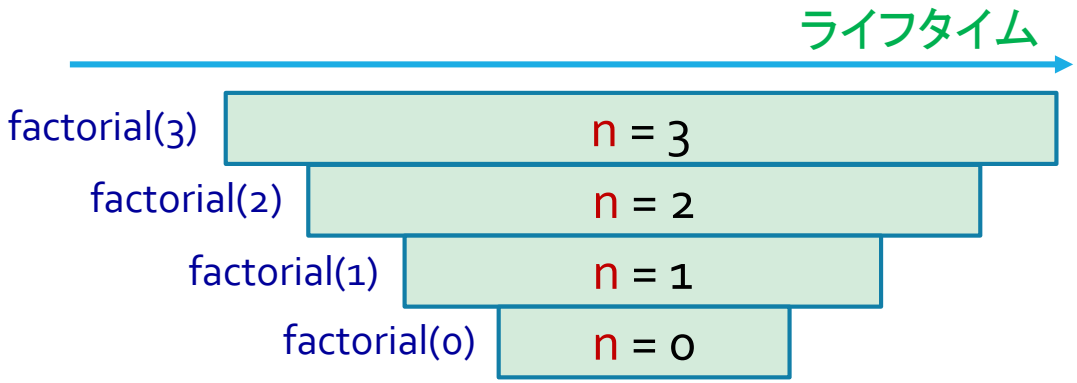
再帰関数 factorial(n) のふるまい



４つのローカル変数 n は，
名前は同じだが，異なる変数．

スタック上に別々のメモリ領域を割り当て．





内部の動作は複雑（実行を追ってはダメ）

数学的帰納法により理解

数学的帰納法
(mathematical induction)

すべての自然数 n について命題 P(n) が成り立つことの証明方法

(i) 基礎ケース: P(o)
(base case)

最小の自然数 n=o について命題 P(n) が成り立つことを示す

(ii) 帰納ステップ: P(k)⇒P(k+1)
(induction step)

k を任意の自然数とし: P(k) を仮定して, P(k+1)を示す

帰納法の仮定
(inductive hypothesis)

(i)(ii)が示されれば, すべての自然数 n について命題 P(n) が成り立つ

数学的帰納法による証明の例

定理　$1+2+\cdots+n=\frac{1}{2}n(n+1)$

(i) 基礎ケース: P(o)

$n=0$ のとき，左辺＝0，右辺=0.

注: $n=0$ のときは，$1+2+\cdots+n=0$.
なぜなら:
$(1+2+\cdots+n)=0+(1+2+\cdots+n)$

(ii) 帰納ステップ: P(k)⇒P(k+1)

帰納法の仮定: $1+2+\cdots+k=\frac{1}{2}k(k+1)$

$n=k+1$ のとき，

左辺 $=1+2+\cdots+k+(k+1)$

$=\frac{1}{2}k(k+1)+(k+1)$　　（帰納法の仮定より）

$=\frac{1}{2}(k+1)(k+2)$

右辺 $=\frac{1}{2}(k+1)(k+2)$

数学的帰納法による再帰関数の正当性の証明

任意の自然数について，実装が仕様を満たすことを証明

実装　　仕様

```
factorial 0       = 1
factorial (n+1)   = (n+1) * factorial n
```

定理　factorial n ＝ $1 \times 2 \times \cdots \times n$

(i) 基礎ケース

$n = 0$ のとき，左辺 ＝ factorial $0 = 1$

右辺 $= 1 \times 2 \times \cdots \times 0 = 1$

(ii) 帰納ステップ

帰納法の仮定：factorial k ＝ $1 \times 2 \times \cdots \times k$

$n = k + 1$ のとき

左辺 ＝ factorial $(k+1)$

$= (k+1) \times$ factorial k　　（定義より）

$= (k+1) \times (1 \times 2 \times \cdots \times k)$　　（帰納法の仮定）

$= 1 \times 2 \times \cdots \times (k+1) =$ 右辺

数学的帰納法の別バージョン

(i) 基礎ケース: P(o)
最小の自然数 n=o について命題 P(n) が成り立つことを示す

(ii) 帰納ステップ: (n>k ⇒ P(k)) ⇒ P(n)
n を任意の自然数とし:
n より小さなすべての k についてP(k) を仮定して, P(n)を示す

帰納法の仮定
「引数を小さくすると命題は正しい」
と仮定する

(i)(ii)が示されれば, すべての自然数 n について命題 P(n) が成り立つ

再帰的なデータ構造：リスト

リスト ＝ [ ]
（list）　| cons（データ, リスト）

空リスト
（極小元）

または

構成子
(constructor)

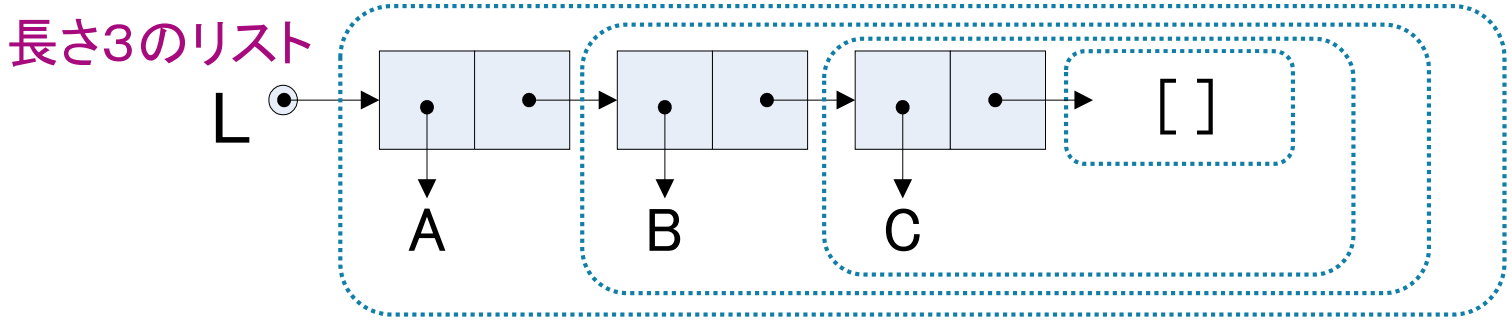
再帰

L :: リスト

変数の型宣言

L = cons(A, cons(B, cons(C, [ ])))

代入



【省略記法】　L = A : (B : (C : [ ]))

L = A : B : C : [ ]

再帰的なデータ構造：木

二分木 (binary tree) ＝ Leaf(整数)
　　| Node(二分木, 整数, 二分木)

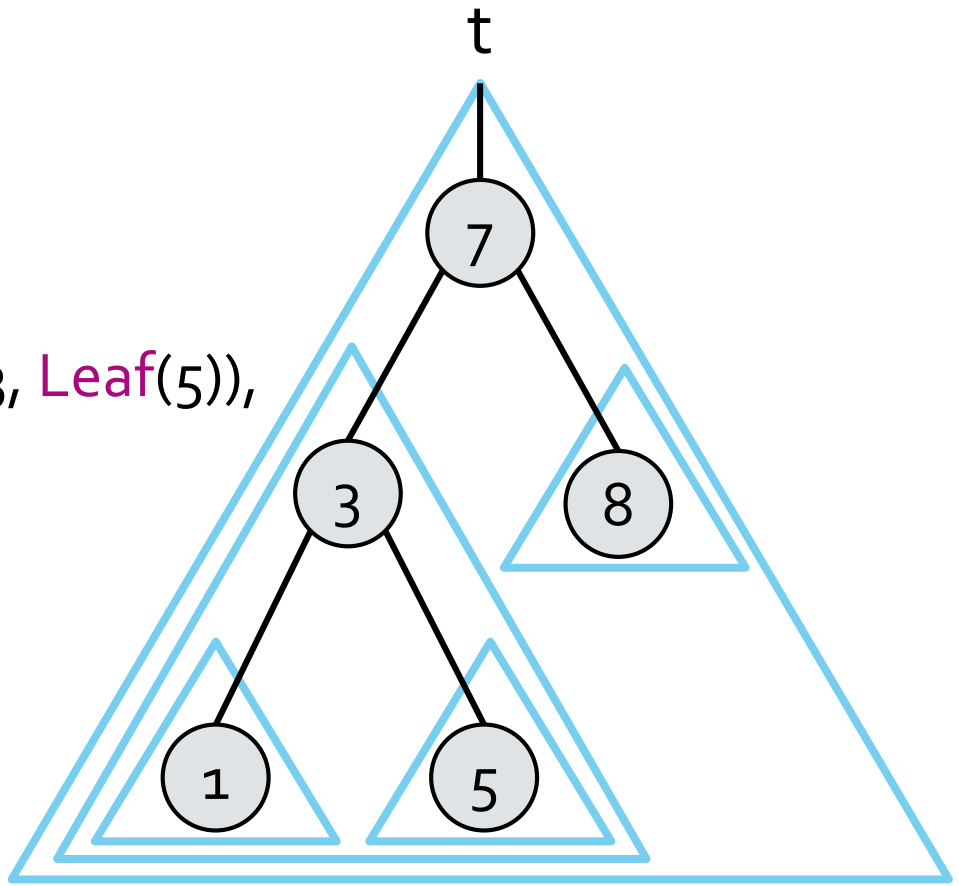
葉ノード（極小元）

非終端ノード（再帰）

（左部分木）　（右部分木）

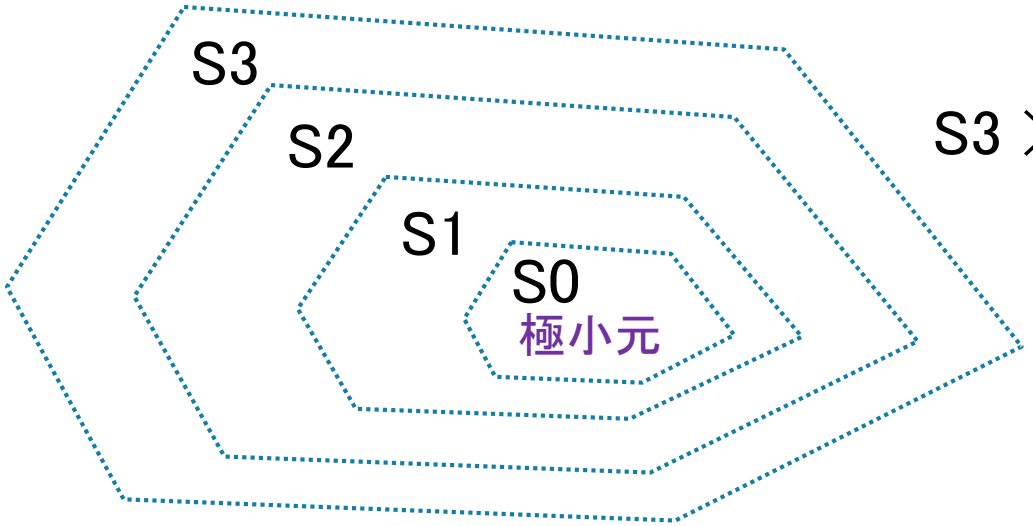
Leaf, Node: 構成子

```
t :: 二分木
t = Node( Node(Leaf(1), 3, Leaf(5)),
          7,
          Leaf( 8)
        )
```



再帰的なデータ構造：まとめ

再帰的データ構造 ＝ 極小元
　　｜ 構成子（再帰的データ構造）



S3 ＞ S2 ＞ S1 ＞ S0

⇒ 構造に大小関係（半順序）がある
（partial order）
ただし，整礎であること（無限の減少列が存在しない）
（well-founded）

構造的帰納法
(structural induction)

数学的帰納法を一般化した証明手法の一つ.

リスト構造や木など，再帰的なデータ構造に関する命題を証明する方法.

■すべての構造 S について命題 P(S) が成り立つことの証明方法

(i) 基礎ケース: P(μ)
極小元 μ について命題 P(μ) が成り立つことを示す

(ii) 帰納ステップ: (S>S′ ⇒ P(S′)) ⇒ P(S)
S を任意の構造とし:
S より小さなすべての構造 S′ についてP(S′) を仮定して，P(S)を示す

帰納法の仮定
「引数を小さくすると命題は正しい」
と仮定する

(i)(ii)が示されれば，すべての構造 S について命題 P(S) が成り立つ

構造的帰納法による再帰関数の正当性の証明(1/3)

【リストの長さ】

任意のデータ型 a のリスト

```
length          :: [a] → Int
length [ ]      =  0
length (x : xs) =  1 + length xs
```

定理：　length xs は xs の長さ（要素数）

(i) 基礎ケース

xs=[ ] のとき，length xs ＝length [ ] = 0 なので成り立つ.

(ii) 帰納ステップ

帰納法の仮定：x : xs（長さ $n+1$）より小さなリスト xs について
length xs は xs の長さ

length (x : xs) ＝ 1 ＋ length(xs)　　（定義より）
＝ 1 ＋（xs の長さ）　（帰納法の仮定より）
＝（x : xs）の長さ

構造的帰納法による再帰関数の正当性の証明(2/3)

【リストの最後の要素を削除】

```
butlast          :: [a] → [a]
butlast [ ]      = [ ]
butlast [ x ]    = [ ]
butlast (x : xs) = x : butlast xs
```

要素が0個の場合
要素が1個の場合
要素が2個以上の場合

定理：　butlast xs は xs の最後の要素を削除したリスト

(i) 基礎ケース

要素数が 0 のとき，butlast [ ] = [ ]
要素数が 1 のとき，butlast [a] = [ ]

} いずれも正しい.

(ii) 帰納ステップ　要素数が 2 以上(x : xs)のとき(xs は要素数 1 以上)

帰納法の仮定：　x : xs より小さなリスト xs について
butlast xs は xs の最後の要素を削除したリスト

butlast (x : xs) = x : butlast xs
= x : (xs から最後の要素を削除したリスト)
= (x : xs) から最後の要素を削除したリスト

構造的帰納法による再帰関数の正当性の証明(3/3)

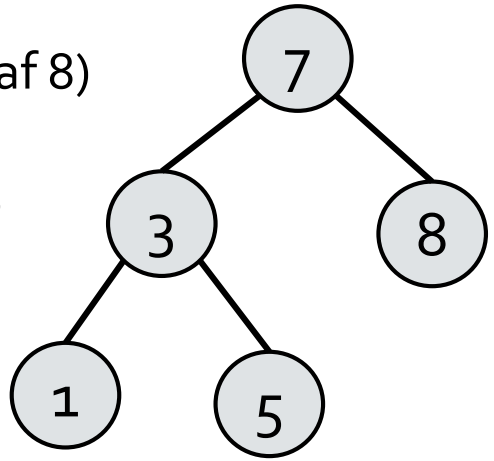
【木に含まれる整数の総和】

```
data Tree = Leaf Int
          | Node Tree Int Tree

sum                          :: Tree → Int
sum (Leaf n)                 = n
sum (Node left n right)      = sum (left) + n + sum (right)
```

```
t :: Tree
t = Node (Node (Leaf 1) 3 (Leaf 5)) 7 (Leaf 8)
```

ここで sum t を計算すると 24 が得られる



演習問題 5

整数と整数リストを受け取り
整数リストを返す関数

(1) つぎの関数定義を考える.

```
insert                          :: Int → [Int] → [Int]
insert x [ ]                    = [x]
insert x (y : ys) | x≤y         = x : y : ys
                  | otherwise   = y : (insert x ys)

isort           :: [Int] → [Int]
isort [ ]       = [ ]
isort (x : xs)  = insert x (isort xs)
```

ガード付き等式
（条件を満たすほうを実行）

| は such that と読むとよい

つぎの命題の証明の概略を示しなさい.

insert x xs は, 昇順にソートされた整数リスト xs の適切な位置に
x を挿入して, 昇順にソートされた整数リストを返す.
isort xs は, 整数リスト xs を昇順にソートする.

(2) 1つ前のスライドで示した sum t は木 t に含まれるすべての整数の和を求めることの証明の概略を示しなさい.