4

ソフトウェア工学

Software Engineering

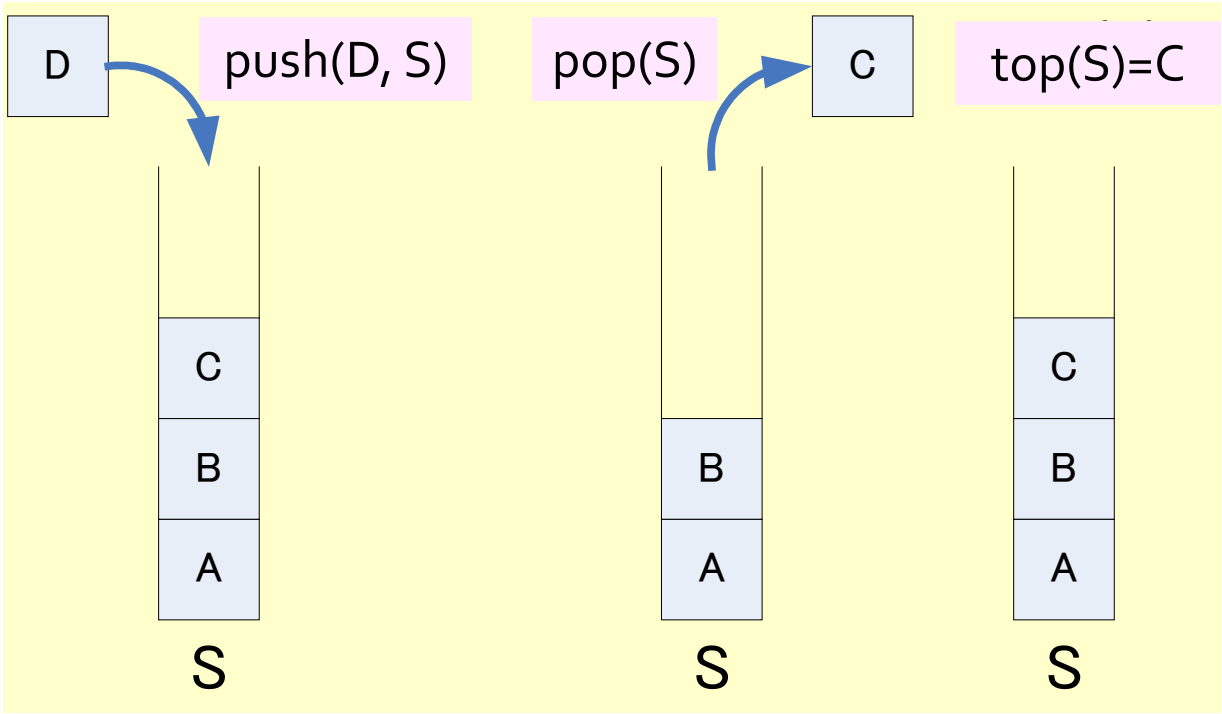
抽象データ型

ABSTRACT DATA TYPE

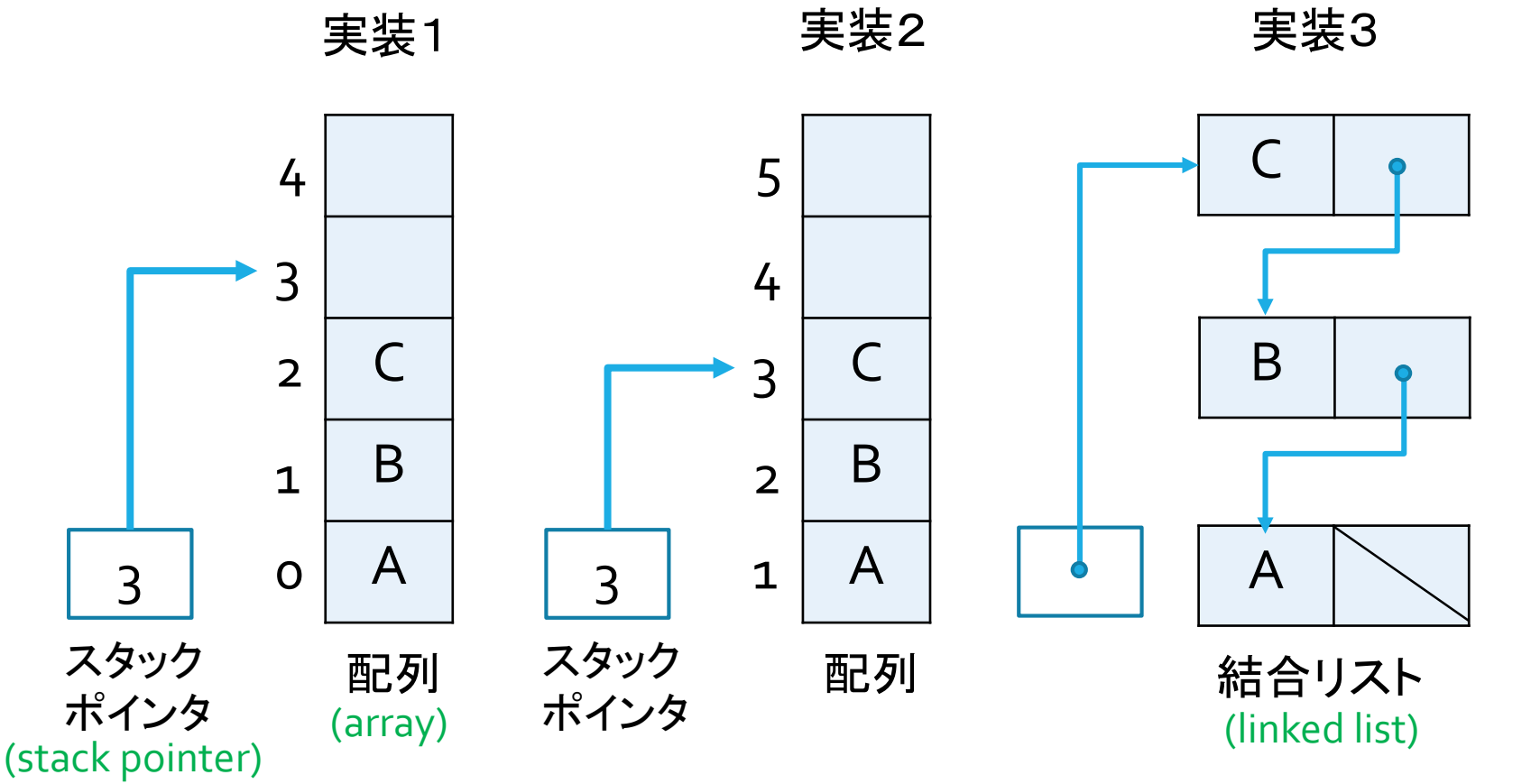
データ抽象　(data abstraction)

■目的：データ構造を（実装に依存せずに）抽象的に定義
■方法：データにアクセス(read, write)する関数の仕様のみを記述

スタック(stack)の例



スタックの実装のいろいろ



実装がいろいろある
実装は今後も変化する

→ 複数の開発者間で再利用しにくい

インタフェースと実装の分離

インタフェースを
抽象的に理解

アプリケーション
プログラマー

API: Application Programming Interface

push(D, S)　pop(S)　top(S)

(interface)
インタフェース

分　　離

実　装
(implementation)

スタック

実装は
隠蔽

システムプログラマー

インタフェースと実装の分離の利点

- 情報隠蔽，抽象化：細部を隠してわかりやすくする
- 契約（インタフェース）を変えずに実装を進化させられる
- アプリケーション（利用者）とシステム（提供者）の分離

データ型の概念（初心者向け）

プログラミングで使用される概念で、データの種類のこと.
文字型・整数型・実数型・文字列型などがある.

【例】基本データ型

- boolean　真偽値（1ビット）
- char　文字（16ビット）
- byte　整数（8ビット）
- short　整数（16ビット）
- int　整数（32ビット）
- long　整数（64ビット）
- float　浮動小数点数（32ビット）
- double　浮動小数点数（64ビット）

【例】ユーザー定義型

- String　文字列
- Stack　スタック
- List　リスト

古い考え方：データ型 ＝ データの集合

int = 全32ビットデータの集合 ？

このデータのデータ型は？　　32ビット

01000001011000000000000000000000

- int　　1,097,072,640？
- float　　14.25？
- String　　”槍” とNULL文字？
- その他　　白黒画像の一部？

●データを見ても，データ型はわからない
なぜならば：　データを見ても，データ型の「集合」に属するか判断不可能

→ データ型は単なるデータの集合としては定義できない

現代的な考え方：
データ型＝データの集合＋演算・処理の集合

01000001011000000000000000000000

- int　　CPUが整数演算に使うなら 1,097,072,640
- float　　CPUが実数演算に使うなら 14.25
- String　　CPUが文字列処理に使うなら "槍" とNULL文字
- その他　　CPUが画像処理に使うなら 白黒画像の一部

高級プログラミング言語では，データの種類ごとに
適用可能な演算や入出力等の処理を混在させず，固定

→ データ型＝〈データの集合，演算や処理の集合〉

intの印字

int = 〈全32ビットデータの集合，{+, −, *, / , putIntなど}〉

float = 〈全32ビットデータの集合，{+, −, *, / , putFloatなど}〉

→ この考えがやがて オブジェクト に発展

抽象データ型の考え方：データ型＝関数の集合

データ型の定義に現れるデータ，演算，処理は，
理論的には，すべて関数から定義できると考える

■ nat（自然数）型の例

4 は，s(s(s(s(o( ))))) で定義される

- データ
  o( ) : 0 を返す関数(zero function)
  s(x) : xのつぎの数(x+1)を返す関数(successor function)
- 演算
  plus(x, y): x+yを返す関数
  times(x, y): x*yを返す関数
- 処理
  putInt(x): xを10進数として印字する作用をもつ「関数」

データの具体的な表現を示さず，関数だけを示している

抽象データ型
(abstract data type)

ソフトウェアの複雑性を抑制
する一般的な手段のひとつ

代数的仕様記述

■ nat（自然数）の例

- データ
  o : 0 を返す関数(zero function)　(o( ) を単に o と書くことにする)
  s(x) : xのつぎの数(x+1)を返す関数(successor function)
- 演算
  plus(x, y): x+yを返す関数

関数の定義（仕様）を日本語で（非形式的に）記述

数学的（形式的）に記述　形式的仕様記述
(formal)　(formal specification)

関数の間の関係（制約）を等式によって記述

代数的仕様記述　（形式的仕様記述の一種）

plus(o, y) = y
plus(s(x), y) = s(plus(x, y))

関数の型の記述

plus : nat nat → nat

アリティ (arity)
（引数のデータ型）

ソート (sort)
（戻り値のデータ型）

関数の型

アリティ = 1 → unary

アリティ = 2 → binary

自然数 (nat: natural number)の記述

仮想的な**代数的仕様記述言語**による定義

```
abstract data type nat

constructors
  o :        → nat
  s  : nat → nat

operators
  plus : nat nat → nat
  times : nat nat → nat

equations
  plus(o, y) = y
  plus(s(x), y) = s(plus(x, y))
  times(o, y) = o
  times(s(x), y) = plus(times(x, y), y)
```

抽象データ型の名前

構成子の型
（データを作り出す関数）

演算子の型

演算子の定義

直接実行

```
plus(o, y) = y
plus(s(x), y) = s(plus(x, y))
times(o, y) = o
times(s(x), y) = plus(times(x, y), y)
```

プログラミング言語で実装する前に
仕様を用いた式の書換え(rewriting)で実行(シミュレーション)

2 * 2 = 4 を求めてみる

```
times(s(s(o)), s(s(o)))
= plus(times(s(o), s(s(o))), s(s(o)))
= plus(plus(times(o, s(s(o))), s(s(o))), s(s(o)))
= plus(plus(o, s(s(o))), s(s(o)))
= plus(s(s(o)), s(s(o)))
= s(plus(s(o), s(s(o))))
= s(s(plus(o, s(s(o)))))
= s(s(s(s(o))))
```

⇒ データの具体的な表現を決めなくても関数の計算ができた

記述の完全性

- データ : nat
  - o : → nat
  - s(x) : nat → nat

o は自然数
x が自然数ならば，s(x)は自然数
すべての自然数は，o と s(x) を用いて構成できる

} 無矛盾（変な定義でない）
} 完全（抜けがない）

- 演算 : plus(x, y)
  - plus(o, y) = y
  - plus(s(x), y) = s(plus(x, y))

} 無矛盾（変な定義でない）

完全（すべての引数の組合せを網羅）

第2引数は，すべての自然数を網羅．
第1引数は，自然数が 0 か 1以上か（必ずどちらかになっている）
に応じて，どちらかの等式で網羅．

抽象データ型の代数的仕様記述と実装

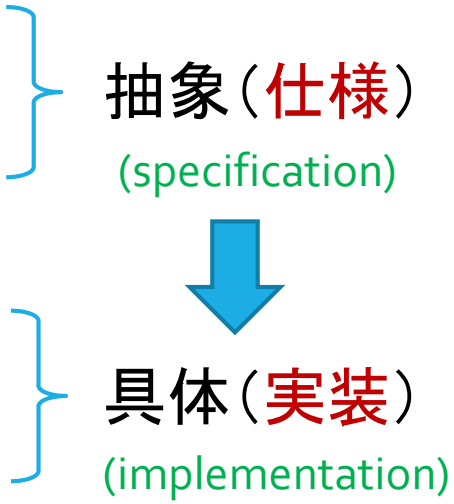
■抽象データ型を 無矛盾で完全な 代数的仕様記述 により定義
■仕様を満たすように実装（プログラミング）

■ nat の例

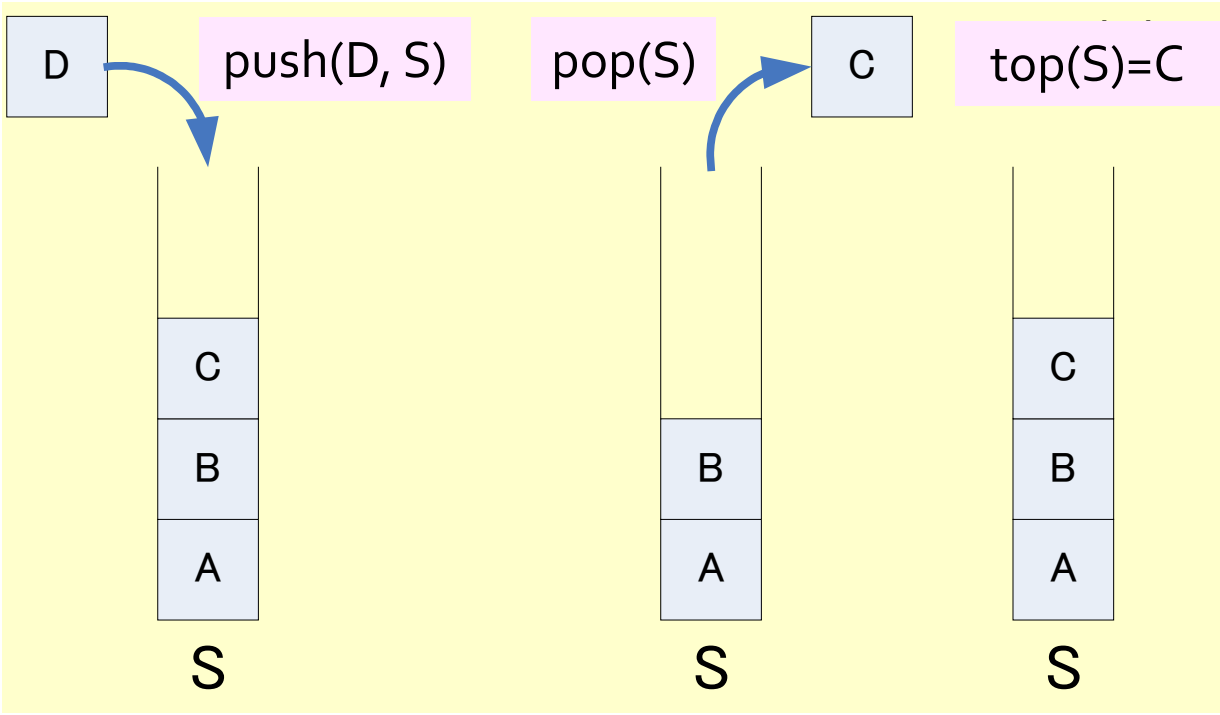
データの集合＝{o, s(o), s(s(o)), s(s(s(o))), ...}
演算の集合　＝{plus, times}

■ nat の実装

データの集合＝{0000, 0001, 0010, 0011, ...}
演算の集合　＝{加算, 乗算}



スタック
(stack)



自然数のスタック(natStack)の記述

abstract data type natStack

constructors

empty_natStack : → natStack

push : nat natStack → natStack

operators

pop : natStack → natStack

top : natStack → nat

equations

pop(empty_natStack) = empty_natStack

pop(push(x, S)) = S

top(empty_natStack) = o

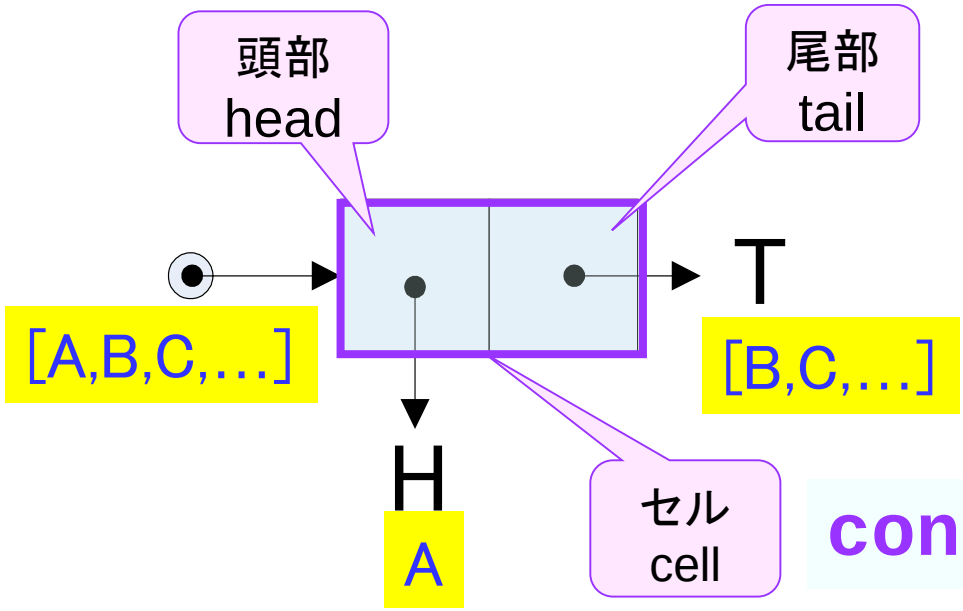
top(push(x, S)) = x

← 本当は，エラーにしたいところ

記述が無矛盾で完全になっていることを確かめよう

リスト

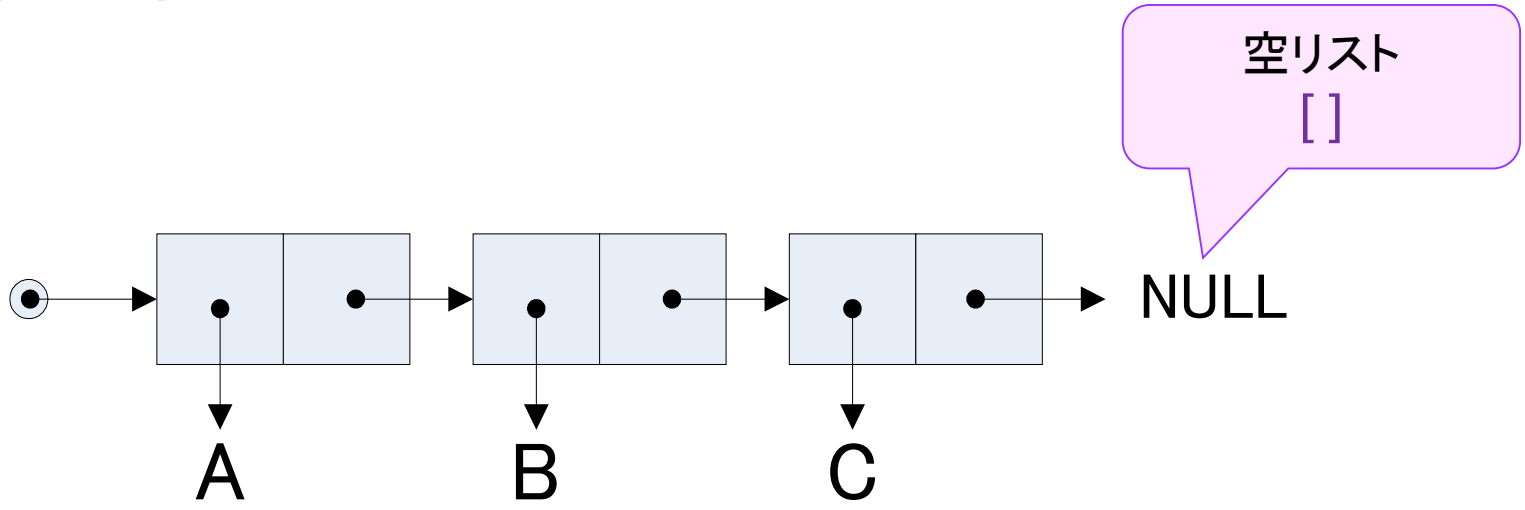
リスト: 任意の長さのデータの列 [A, B, C, …]
(list) を表現するデータ構造



cons(H, T) で表現

簡易記法: **H : T**

リスト



A: (B: (C: [ ]))

= A: B: C: [ ]

: は右結合演算子

= [A, B, C]

（この簡易記法が使われることもある）

リストの演算

head(A : B : C : D : E : [ ]) = A

tail(A : B : C : D : E : [ ]) = B : C : D : E : [ ]

append(A : B : C : [ ], D : E : [ ])　　結合
= A : B : C : D : E : [ ]

reverse(A : B : C : D : E : [ ])　　反転
= E : D : C : B : A : [ ]

自然数のリスト(natList)の記述(1/2)

abstract data type natList

constructors

null : → natList

cons : nat natList → natList

operators

head : natList → nat

tail : natList → natList

append : natList natList → natList

reverse : natList → natList

自然数のリスト(natList)の記述(2/2)

equations

head([ ]) = null

head(x : L) = x

tail([ ]) = null

tail(x : L) = L

append([ ], L) = L

append(x : L1, L2) = x : append(L1, L2)

reverse([ ]) = [ ]

reverse(x: L) = append(reverse(L), x : [ ])

リスト処理の直接実行

append([ ], L)　　= L
append(x : L1, L2) = x : append(L1, L2)
reverse([ ])　= [ ]
reverse(x: L) = append(reverse(L), x : [ ])

append(1 : 2 : [ ], 3 : 4 : [ ])
= 1 : append(2 : [ ], 3 : 4 : [ ])
= 1 : 2 : append([ ], 3 : 4 : [ ])
= 1 : 2 : 3 : 4 : [ ]

reverse(3 : 2 : 1 : [ ])
= append(reverse(2 : 1 : [ ]), 3 : [ ])
= append(append(reverse(1 : [ ]), 2 : [ ]), 3 : [ ])
= append(append(append(reverse([ ]), 1 : [ ]), 2 : [ ]), 3 : [ ])
= append(append(append([ ], 1 : [ ]), 2 : [ ]), 3 : [ ])
= append(append(1 : [ ], 2 : [ ]), 3 : [ ])
= . . . . . = append(1 : 2 : [ ], 3 : [ ])
= . . . . . = 1 : 2 : 3 : [ ]

関数型プログラミング言語による実装

【Haskell による実装の例】

```
head          :: [a] → a
head (x : _)  =  x
```

a は任意のデータ型を表す

[a] は a 型データのリストを表す

_ は 重要でない変数名を省略したもの

```
tail          :: [a] → [a]
tail (_ : ys) =  ys
```

[a] と [a] を受け取り [a] を返す関数の型

```
append              :: [a] → [a] → [a]
append [ ] ys       =  ys
append (x : xs) ys  =  x : (append xs ys)
```

関数の引数をくくるカッコを省略可

```
reverse           :: [a] → [a]
reverse [ ]       =  [ ]
reverse (x : xs)  =  append (reverse xs) [x]
```

演習問題 4

(1) 自然数の代数的仕様 nat を拡張するために，減算 minus を等式で定義しなさい．
　ただし，x < y のときには，minus(x, y) の値はゼロであると定義する．

ヒント：　左辺が minus(o, y)，minus(s(x), o)，minus(s(x),s(y)) からなる3つの等式を記述する．

(2) 自然数のスタックの代数的仕様 natStack の直接実行を用いて，つぎの式の計算をしなさい．
　ただし，自然数 s(s(s(o))) を 3 などと略記してよい．
　　top(pop(push(3, push(2, push(1, empty_natStack)))))

(3) 自然数のリストの代数的仕様 natList を拡張するために，関数 sum(L) を等式で定義しなさい．
　ただし，sum(L) は，演算 plus を用いて，自然数のリスト L の中のすべての自然数の総和を求めるものとする．
　実行例：sum(s(s(o)) : o : s(o) : [ ]) = s(s(s(o)))

ヒント：　左辺が sum([ ])，sum(x : L) からなる2つの等式を記述する．