

4

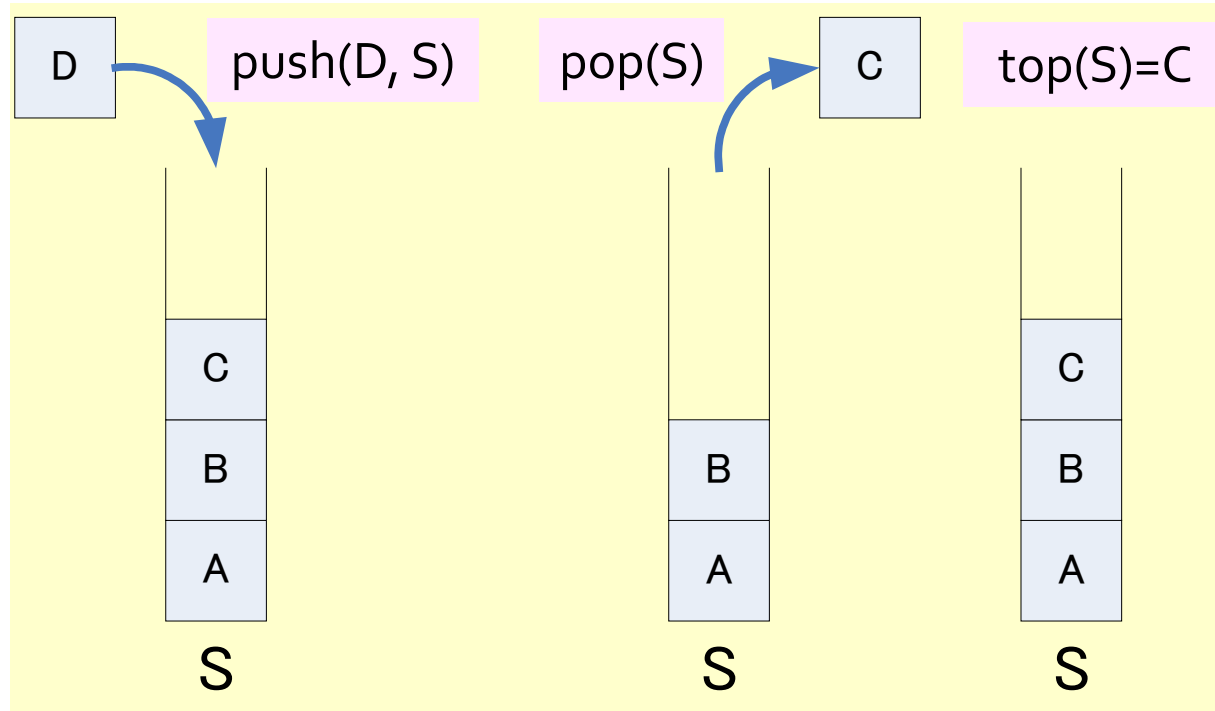
ソフトウェア工学

Software Engineering

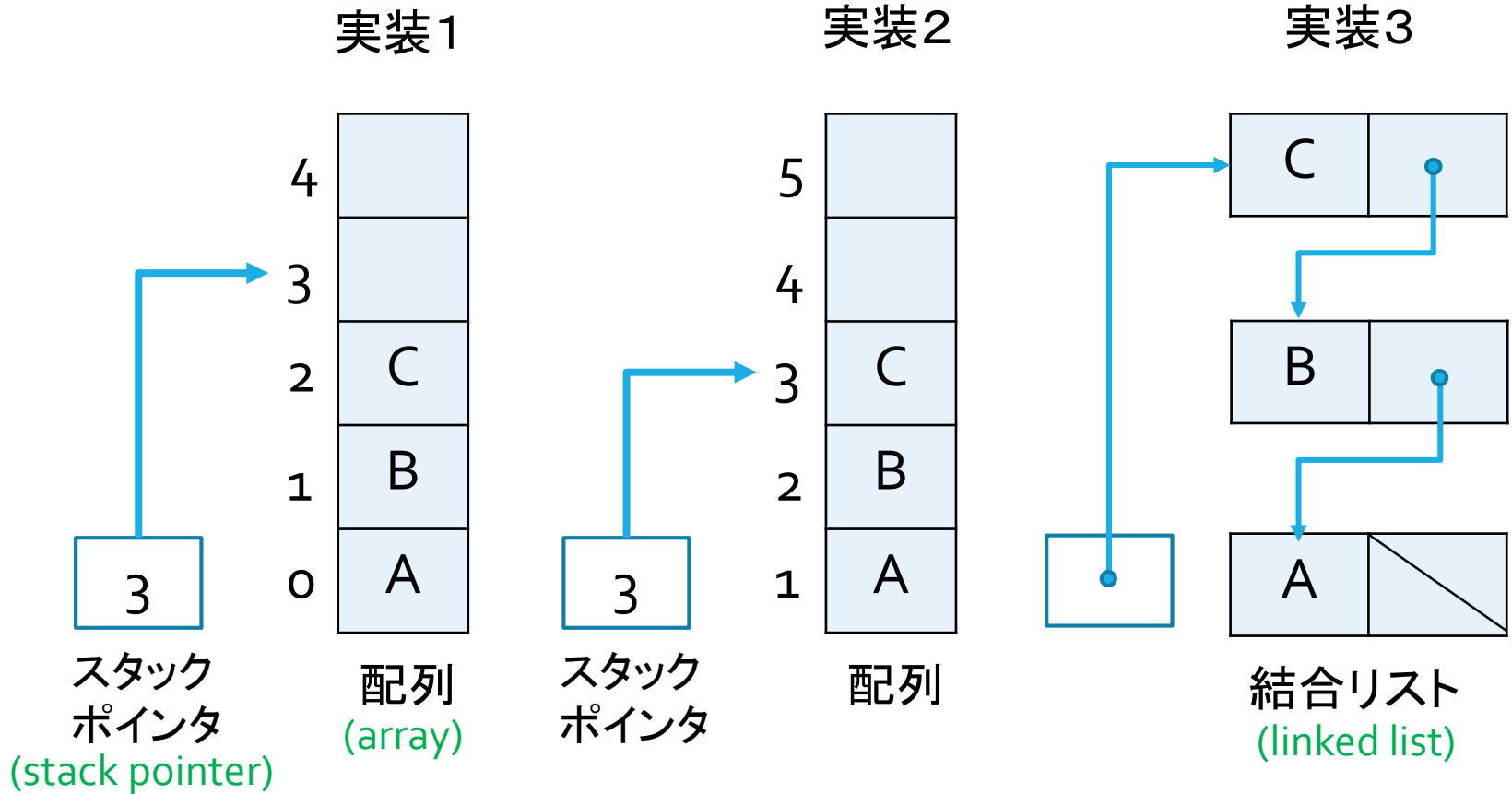
抽象データ型

ABSTRACT DATA TYPE

データ抽象の動機：スタックの例 (data abstraction) (stack)



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

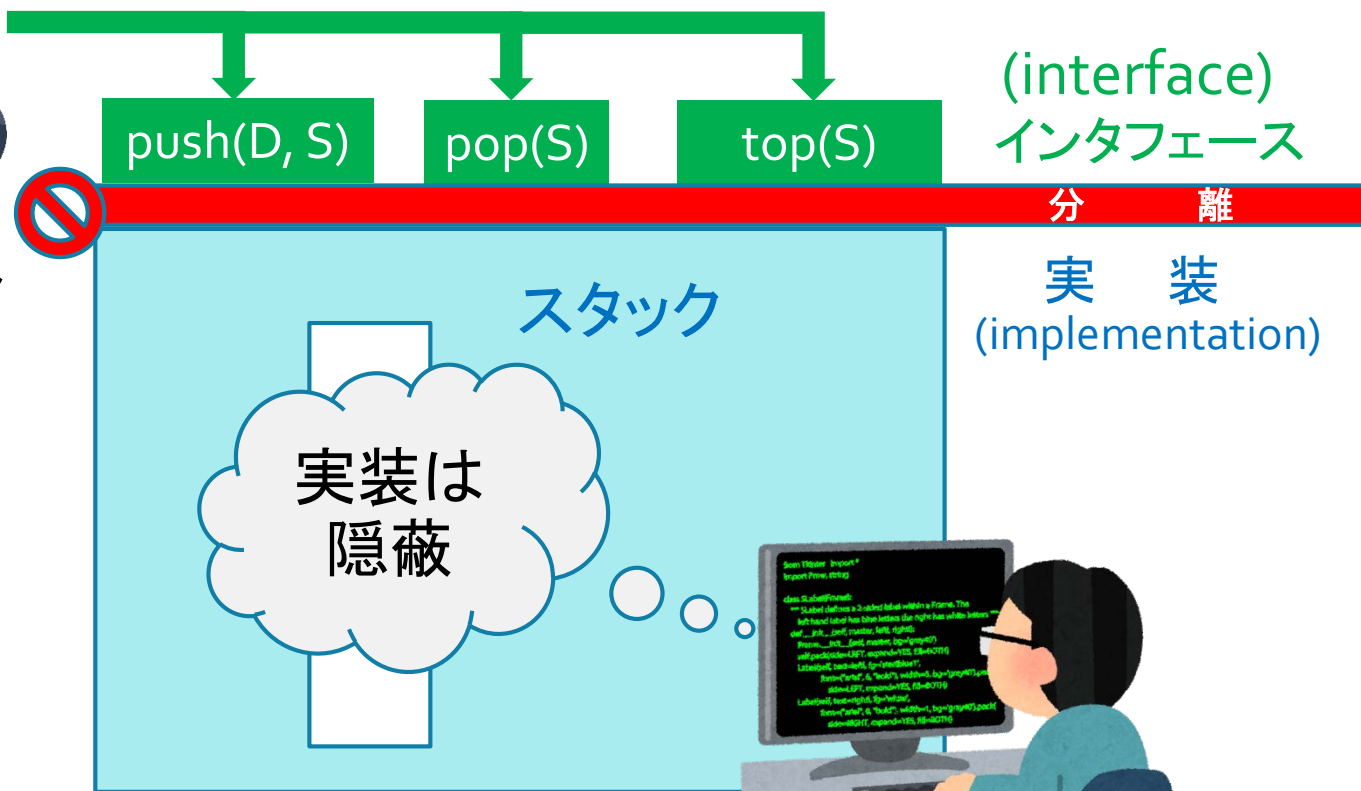
インタフェースを
抽象的に理解

インタフェースと実装の分離

API: Application Programming Interface



アプリケーション
プログラマー



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

データ型はどこで定義されているか

- CPUで定義しているのではない
- プログラミング言語で定義されている（基本データ型）
- プログラマが定義できる（ユーザ定義型）



データ型は、プログラミング言語とプログラムで定義される

- 使えるデータ型は、言語ごとに異なる.
- 使えるデータ型は、ライブラリごとに異なる.
- 使えるデータ型は、アプリケーションごとに異なる.

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
- データを見ても、データ型の「集合」に属するか判断できない

➡ データ型は単なるデータの集合としては定義できない

データ型 = データの集合 + 演算・処理の集合

010000010110000000000000000000000000

- 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

データ型 = 関数の集合 という考え方もある

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

• 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)

4 は, $s(s(s(s(o()))))$ で定義される

• 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



データ抽象

(data abstraction)

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
(formal)

形式的仕様記述
(formal specification)

関数間の関係 (制約) を等式によって記述



代数的仕様記述

$$plus(o, y) = y$$

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

関数の型の記述

plus : nat nat → nat

アリティ (arity) ソート (sort)
(引数のデータ型) (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

$o : \quad \rightarrow \text{nat}$

$s : \text{nat} \rightarrow \text{nat}$

operators

$\text{plus} : \text{nat} \text{ nat} \rightarrow \text{nat}$

$\text{times} : \text{nat} \text{ nat} \rightarrow \text{nat}$

equations

$\text{plus}(o, y) = y$

$\text{plus}(s(x), y) = s(\text{plus}(x, y))$

$\text{times}(o, y) = o$

$\text{times}(s(x), y) = \text{plus}(\text{times}(x, y), y)$

抽象データ型の名前

構成子の型

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
仕様を用いた式の変換で実行(シミュレーション)

$2 * 2 = 4$ を求めてみる

$\text{times}(\text{s}(\text{s}(\text{o})), \text{s}(\text{s}(\text{o})))$	$2 * 2$
$= \text{plus}(\text{times}(\text{s}(\text{o}), \text{s}(\text{s}(\text{o}))), \text{s}(\text{s}(\text{o})))$	$1 * 2 + 2$
$= \text{plus}(\text{plus}(\text{times}(\text{o}, \text{s}(\text{s}(\text{o}))), \text{s}(\text{s}(\text{o}))), \text{s}(\text{s}(\text{o})))$	$0 * 2 + 2 + 2$
$= \text{plus}(\text{plus}(\text{o}, \text{s}(\text{s}(\text{o}))), \text{s}(\text{s}(\text{o})))$	$0 + 2 + 2$
$= \text{plus}(\text{s}(\text{s}(\text{o})), \text{s}(\text{s}(\text{o})))$	$2 + 2$
$= \text{s}(\text{plus}(\text{s}(\text{o}), \text{s}(\text{s}(\text{o}))))$	$1 + (1 + 2)$
$= \text{s}(\text{s}(\text{plus}(\text{o}, \text{s}(\text{s}(\text{o}))))$	$1 + (1 + (0 + 2))$
$= \text{s}(\text{s}(\text{s}(\text{s}(\text{o}))))$	$1 + (1 + 2)$

 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

$$0 : \text{nat}$$

$$s(x) : \text{nat} \rightarrow \text{nat}$$

0 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, 0 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 : $\text{plus}(x, y)$

$$\text{plus}(0, y) = y$$

$$\text{plus}(s(x), y) = s(\text{plus}(x, y))$$

} 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

抽象データ型 (abstract data type) : 無矛盾で完全な代数的仕様記述により定義されるデータ型

■ nat の例

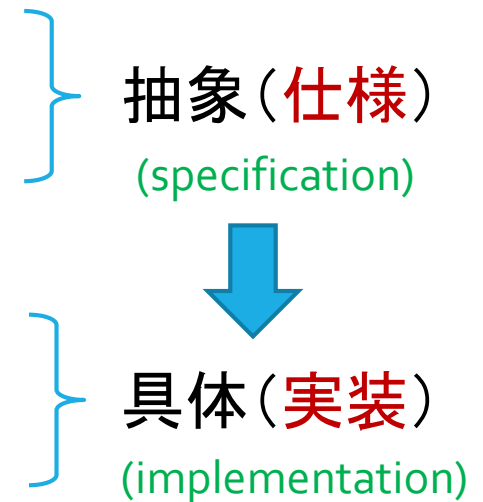
データの集合 = $\{o, s(o), s(s(o)), s(s(s(o))), \dots\}$

演算の集合 = $\{\text{plus}, \text{times}\}$

■ nat の実装

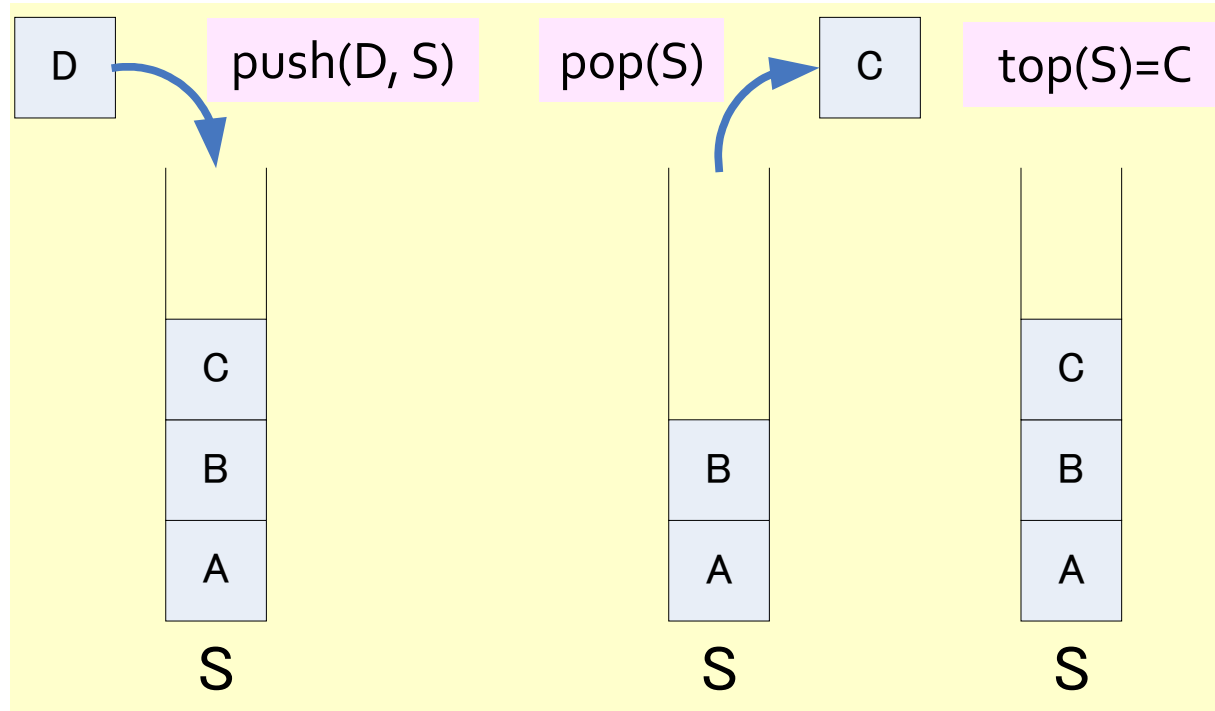
データの集合 = $\{0000, 0001, 0010, 0011, \dots\}$

演算の集合 = $\{\text{加算}, \text{乗算}\}$



スタック

(stack)



自然数のスタック (natStack) の記述

abstract data type natStack

constructors

empty_natStack : \rightarrow natStack

push : nat natStack \rightarrow natStack

operators

pop : natStack \rightarrow natStack

top : natStack \rightarrow nat

equations

pop(empty_natStack) = empty_natStack

pop(push(x, S)) = S

top(empty_natStack) = 0

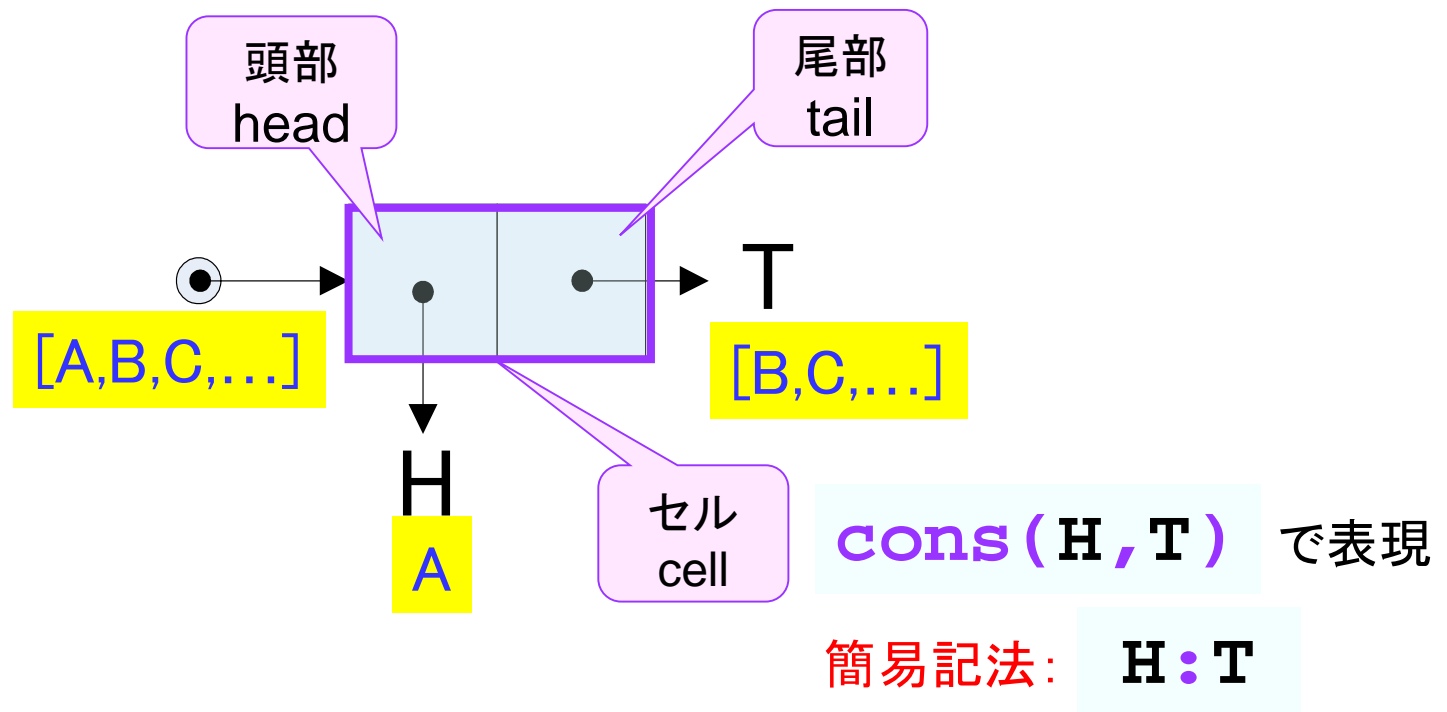
top(push(x, S)) = x

← 本当は, エラーにしたいところ

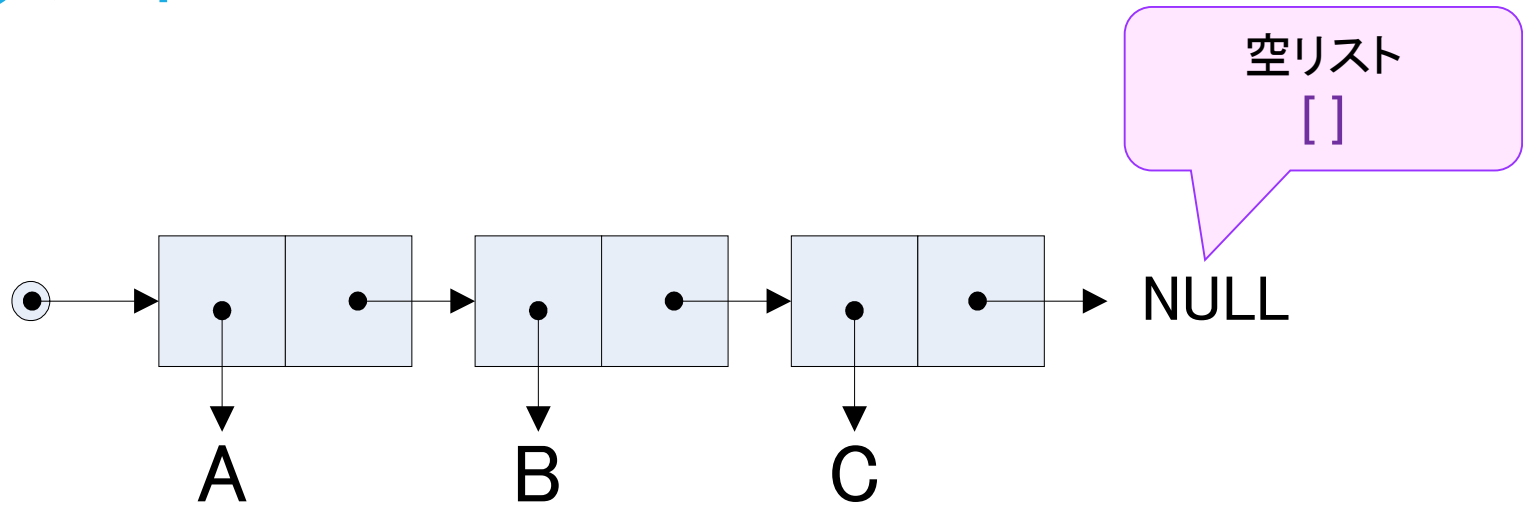
記述が無矛盾で完全になっていることを確かめよう

リスト

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



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 : [])`
`= A : B : C : D : E : []`

結合

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

反転

自然数のリスト(natList)の記述(1/2)

abstract data type natList

constructors

null : \rightarrow natList

cons : nat natList \rightarrow natList

operators

head : natList \rightarrow nat

tail : natList \rightarrow natList

append : natList natList \rightarrow natList

reverse : natList \rightarrow natList

自然数のリスト(natList)の記述(2/2)

equations

$\text{head}([\])$ = null

$\text{head}(x : L)$ = x

$\text{tail}([\])$ = null

$\text{tail}(x : L)$ = L

$\text{append}([\], L)$ = L

$\text{append}(x : L_1, L_2)$ = $x : \text{append}(L_1, L_2)$

$\text{reverse}([\])$ = $[\]$

$\text{reverse}(x : L)$ = $\text{append}(\text{reverse}(L), x : [\])$

リスト処理の直接実行

$\text{append}([], L) = L$
 $\text{append}(x : L1, L2) = x : \text{append}(L1, L2)$
 $\text{reverse}([]) = []$
 $\text{reverse}(x : L) = \text{append}(\text{reverse}(L), x : [])$

```
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` 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。