コンピュータサイエンス入門第二

永藤 直行

4th quarter

Part I

Prologue

Prologue

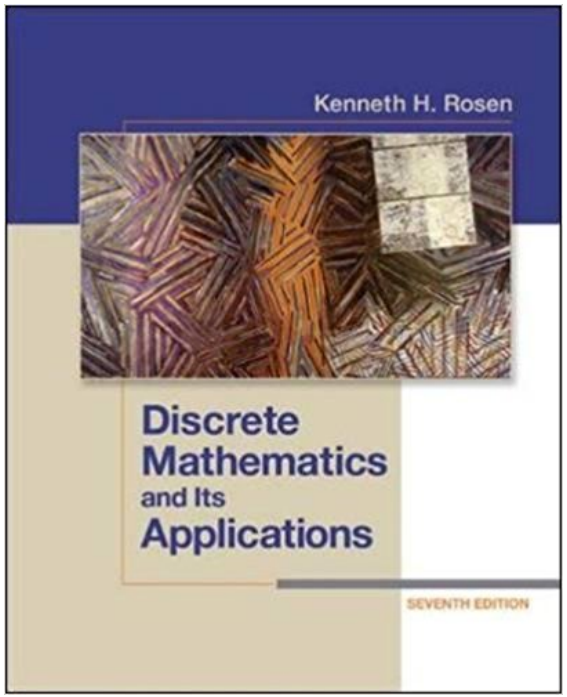
Outline

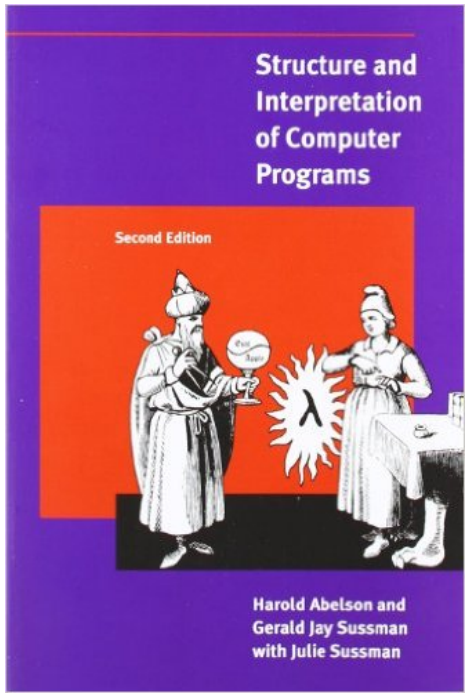
参考図書

- Discrete Mathematics and its Applications
- Structure and Interpretation of Computer Programs

  http://web.mit.edu/alexmv/6.037/sicp.pdf
- 計算機プログラムの構造と解釈 (日本語訳)

  http://sicp.iijlab.net/fulltext/xcont.html





Outline

講義概要

- この講義も Zoom で行います
- 講義資料は https://sites.google.com/presystems.xyz/elementaryCS/ に置いてあります
- 講義スケジュール:
  1. 再帰
  2. よいアルゴリズム，わるいアルゴリズム，ふつうのアルゴリズム

目標

計算でものを表すとき便利な道具と注意すべきことを会得すること

Outline

評価基準

- 講義は全 7 回
- 課題: 3 回 (計 85 点) と特別課題 (15 点)
- 課題提出:
  - 講義時間中に課題を出します
  - 提出方法はその都度指定します

Part II

再帰

再帰

Outline

再帰 (Recursion)
再帰への導入
再帰 (Recursion)
再帰的定義
木構造
プログラムとしての再帰
引数の有効範囲 (Scope)
再帰プログラム
帰納法 (Induction)
繰り返しと再帰
末尾再帰 (Tail Recursion)
コンピュータの中では
再帰のまとめ
課題 S

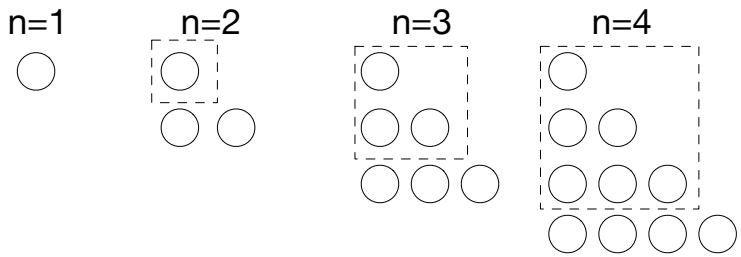
再帰への導入

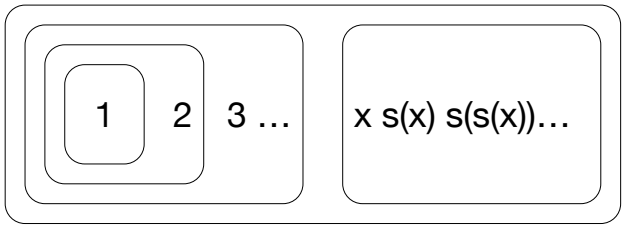
- 対象（問題）を簡潔に明示的に表す方法
- たとえば下の絵，ひとつ絵を書いてその中央にその絵を書いている（Droste 効果)
- 対象を定義するときにより小さい部分を参照して定義する

再帰の例

- 数列 $a_n = \frac{n(n+1)}{2}$, $n \leq 1$
- 初項を 1, n 項を $a_n = n + a_{n-1}$ となる
- n 項を決めるのに $n-1$ 項から決める
- まだ決まっていない最小の項を既に知っている項から決定する
- 再帰的定義:

$$a(n) = \begin{cases} 1 & \text{if } n = 1 \\ n + a(n-1) & \text{otherwise} \end{cases}$$





再帰的定義の原理

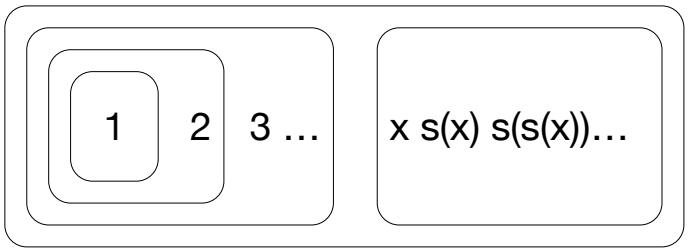
- 関数 f の定義に x より小さい要素についての評価値を利用して定義することを原始再帰 (primitive recursion) と呼ぶ
- f が x より小さい値について計算可能であり，いつも同じ値と仮定し，$f \upharpoonright x$ と書く
- 定義できた最大の x のつぎの値について定義するというのを繰り返すと全体について定義できる (δ-近似)

再帰の原理

順序数 (自然数は順序数) x として，

$$f(x) = G(f \upharpoonright x)$$

ここで，$f \upharpoonright x$ は f を x より小さい数に制限したもの，G は計算のしかたを表したもの



関数の再帰的定義

加算，乗算の再帰的定義

- G に相当するのが succ
- Basic step に近づけるように recursive step を定義
  - $a + \mathrm{succ}(b) = \mathrm{succ}(a + b)$

Basic step:	$a + 0$	if $b = 0$
Recusive step:	$\mathrm{succ}(a + b)$	otherwise

Listing 1: 加算

```
1 # Recursive definition
2 import os
3 import time
4
5 def succ (x):
6   return(x+1)
7 def pred (x):
8   return(x-1)
9 def add (a,b):
10  if (b==0):
11    return(a)
12  else:
13    return(succ(add(a,pred(b))))
```

Listing 2: 乗算

```
15 # Multiplication
16 #
17 def mult (a,b):
18   if (b==0):
19     return(0)
20   else:
21     return(add(add(0,a),mult(a,pred(b))))
```

再帰的定義の例 1

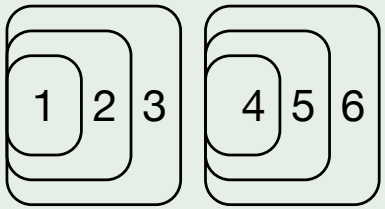
Factorial

- $n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$
- $n!$ を決めるのに $(n-1)!$ の値をつかう
- G に相当するのが $n \cdot (n-1)!$ ということ
- まだ決まっていない最小の値は決まっている値で最大の値のつぎになる

Listing 3: 階乗

```
1 def fact (n):
2   if (n == 1):
3     return(1)
4   else:
5     return(n*(fact(n-1)))
```

Example (4!)



フィボナッチ数

- 突然ですが次の問題を考えてみてください

うさぎとフィボナッチ数

- n ヶ月後のうさぎのつがいは何組？
- 最初一組のつがいだけ
- 2 ヶ月経つと メス 1 匹を生んでそのつがいが 1 組増える
- うさぎは決して死なない

月	生後 0 ヶ月	生後 2 ヶ月以下	合計
1	0	1	1
2	0	1	1
3	1	1	2
4	1	2	3

再帰的定義の例 2

- フィボナッチ数

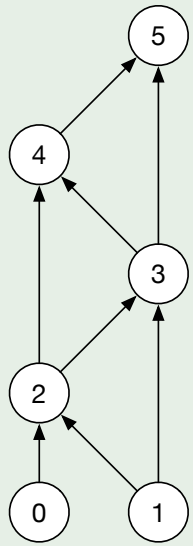
0	1	2	3	4	5···
1	1	2	3	5	8···

- fib($S(n)$) を決めるのに fib(n) と fib($n-1$) を使う
- G に相当するのが n と $n-1$ のフィボナッチ数を足し合わせるということ

Listing 4: フィボナッチ数

```
1 def fib (n):
2   if (n==0):
3     return(0)
4   else:
5     if (n==1):
6       return(1)
7     else:
8       return(fib(n-1)+fib(n-2))
```

Example (fib(5))



集合の再帰的定義

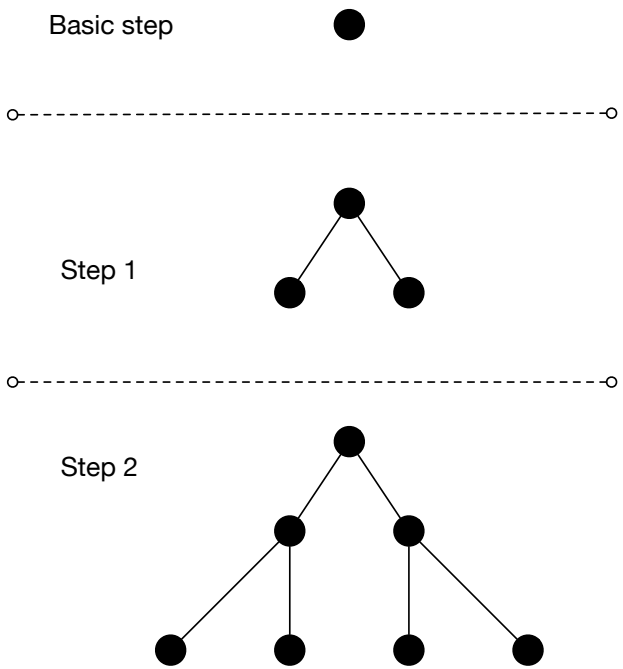
- Basic step: 集合の初期要素を定義
- Recursive step: 既にわかっている要素から新しい要素を定義する規則

Example (文字列の集合 Σ^*)

- Basic step: $\epsilon \in \Sigma^*$ (空列 ϵ も Σ^* に含まれる)
- Recursive step: $a \in \Sigma, w \in \Sigma^*$ ならば $aw \in \Sigma^*$
- E.g.: $\{\epsilon,\ a,\ aa,\ aaa, \cdots\}$

Tree (木構造)

- 数以外も
- Basic step: vertex r は tree
- Recursive step: $T_1, T_2, \cdots, T_n$ それぞれ root を $r_1, r_2, \cdots, r_n$ とする tree として，r から $r_1, r_2, \cdots, r_n$ への edge 追加したものもまた tree である



再帰は強力な道具

- 最初の元から順番に対象を構成
- 対象が順番に並べられるなら
- 問題を解く時も，最も小さい問題の解から順番に全体の解を構成することができる
- （いつもではないけど）

宿題 1

- 自然数上の四則演算を再帰で定義し直してみてください
- フィボナッチ数を求める再帰的な計算を繰り返しに直してみてください
- 先の文字列の集合を参考に文字列の長さを求める関数を再帰的に定義せよ.
  - Hint: Basic step を空列は長さ 0 として，recursive step は一文字短い文字列より 1 長い
  - Python では str[1:] とすると文字列の 2 番目以降の文字列を得ることができる
  - 空列は not str あるいは len(str) で判定

Outline

再帰 (Recursion)
再帰への導入
再帰 (Recursion)
再帰的定義
木構造
プログラムとしての再帰
引数の有効範囲 (Scope)
再帰プログラム
帰納法 (Induction)
繰り返しと再帰
末尾再帰 (Tail Recursion)
コンピュータの中では
再帰のまとめ
課題 S

仮引数と実引数

- CS 第 1 では手続きに名前をつけて抽象化することをみた
- 関数は 0 個以上の仮引数というものをもつ
  - 下の例では n，eps などが仮引数
- 仮引数は関数の本体で有効である
- 関数を呼び出したときの値に束縛 (bind) されて，関数の本体では呼び出し時の値に置き換えられる
- 呼び出し時の値を実引数という
- 一般に変数は有効範囲 (scope) が決まっている
- 仮引数は関数本体が有効範囲である

Listing 5: Newton 法

```
1  ### Newton's method
2  def sqrt_iter (guess,n,eps,previous):
3    def is_enough (guess,eps,previous):
4      return(abs(previous-guess)<(2*eps))
5    def improve (guess,n):
6      return((guess+(n/guess))/2.0)
7    if is_enough(guess,eps,previous):
8       return(guess)
9    else:
10      return(sqrt_iter(improve(guess,n),n
              ,eps,guess))
11 def sqroot1 (n,eps):
12   return(sqrt_iter(1.0,n,(2*eps),0.0))
```

Listing 6: Newton 法

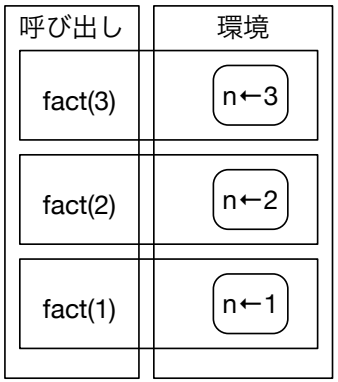
```
13 def sqroot (n):
14   ### Machine epsilon
15   def eps_m ():
16     epsilon, old, prod = 1.0, 0.0,
            0.0
17     cnt=0
18     while (prod!=1.0):
19       old = epsilon
20       cnt=cnt+1
21       epsilon=epsilon/2.0
22       prod=epsilon+1.0
23     return(old)
24   return(sqroot1(n,eps_m()))
```

プログラムとしての再帰

- プログラムでも関数を定義するときに自身をもちいることができる
- 下のプログラムを実行してみるので引数の値に注意して見ていてください
- 関数は定義しただけでは実行されず，呼び出したときはじめて活性化され，仮引数が束縛される
  - n は 3,2,1 と束縛される

Listing 7: 階乗 (引数表示版)

```
1 # Factorial
2 def fact (n):
3   if (n == 1):
4     return(1)
5   else:
6     return(n*(fact(n-1)))
```



関数の評価と生成プロセス

fact(4) の生成プロセス

```
  fact(4)
=>(4 * fact(3))
=>(4 * (3 * fact(2)))
=>(4 * (3 * (2 * fact(1))))
=>(4 * (3 * (2 * 1)))
=>(4 * (3 * 2 ))
=>(4 * 6)
=>24
```

帰納法の原理

- 帰納法は再帰的に定義されたものの性質を証明するテクニック
- 再帰的アルゴリズムの正当性を証明するのにも使える
- 下に帰納法の原理をあげておきます
- succ(y) は y のつぎの数という意味です (CS 入門第一の +1 に相当)
- 証明したいことがらを ϕ とします
- 原理の前提条件を充たすことを示します
  $\phi(0) \wedge (\phi(n) \Rightarrow \phi(\mathrm{succ}(n)))$

帰納法の原理

集合 $X = \{n : \phi(n)\}$ について，もし $0 \in X$ かつ
$\forall y \in X :\ \mathrm{succ}(y) \in X$ であるなら，X はすべての自然数を含む

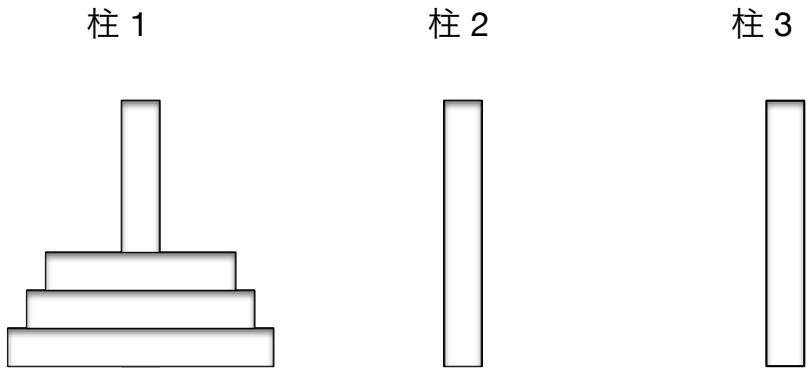
Outline

再帰 (Recursion)
再帰への導入
再帰 (Recursion)
再帰的定義
木構造
プログラムとしての再帰
引数の有効範囲 (Scope)
再帰プログラム
帰納法 (Induction)
繰り返しと再帰
末尾再帰 (Tail Recursion)
コンピュータの中では
再帰のまとめ
課題 S

QUIZ 1

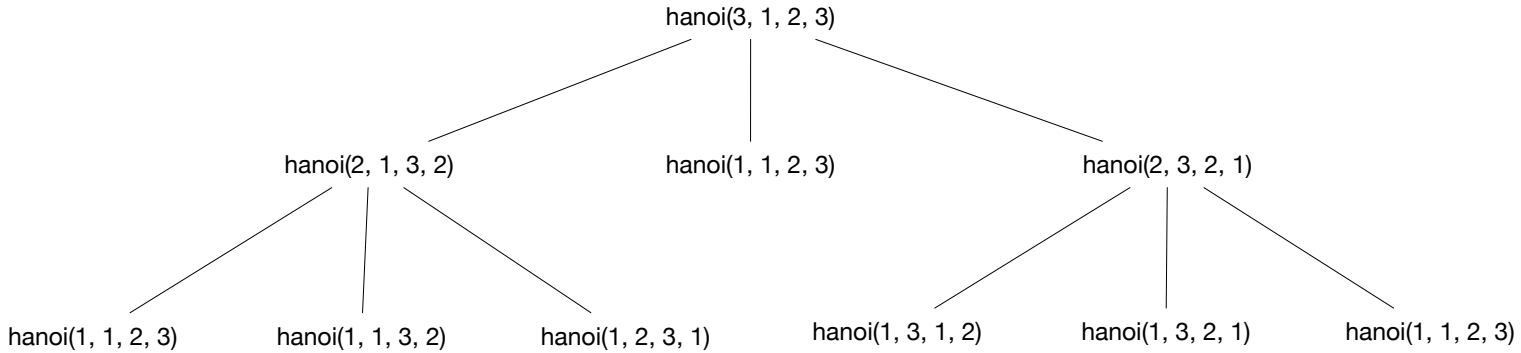
- ハノイの塔 (Tower of Hanoi) というパズルを解くプログラムを作成してください
- 目的:
  - 柱 1 から柱 2 へ移動
- 制約:
  - 柱から柱への移動は一度に 1 枚だけ
  - 大きい円盤をそれより小さい円盤の上にのせてはいけない
- 4b(CS2) クラスのサイト:

  https://sites.google.com/a/presystems.xyz/sample/home/elementary-computer-science から hanoi-skeleton.py をダウンロードして使ってください



Quiz 1 の hint

- Basic step: 1 枚の移動
- Recursive step: n 枚から $n-1$ 枚の移動
- hanoi(n,a,b,c) は n 枚のディスクを a から b へ c を使って移動する関数
- Python でリストに要素を追加するときは append() を使う



繰り返しと再帰

- 以前，繰り返しでいろいろな計算を実現しました
- 再帰は繰り返しに変換できることがあります
- ここでは繰り返しと再帰の関係について見ていきます

末尾再帰呼び出し (Tail Recursive Call)

- どちらも再帰的定義ですが計算プロセスに違いがあります
- Listing 8 は fact(n) を得るために fact(n-1) の後の計算 (n ×ロ) を覚えてなければならない
- Listing 9 はその必要はなく，計算の状態 “カウンタと途中までの積” を覚えておくだけで良い
- Listing 9 のような形を末尾再帰的といいます

Listing 8: 再帰プロセス版

```
1 # Factorial
2 def fact (n):
3   if (n == 1):
4     return(1)
5   else:
6     return(n*(fact(n-1)))
```

Listing 9: 繰り返しプロセス版

```
16 # Factorial
17 def fact_iter (n):
18   def fact_iter1 (prod, cnt, max):
19     if cnt > max:
20       return(prod)
21     else:
22       return(fact_iter1(prod*cnt,cnt+1,
                 max))
23   return(fact_iter1(1,1,n))
```

末尾呼び出し (Tail Call)

- 末尾再帰は繰り返し `while`, `for` に書き換えられます
- 繰り返しは再帰関数の特別な場合と見なせる
- 再帰は関数を呼び出すたびに資源を消費するが，繰り返しは一定

再帰プロセス

```
  fact(4)
=>(4 * fact(3))
=>(4 * (3 * fact(2)))
=>(4 * (3 * (2 * fact(1))))
=>(4 * (3 * (2 * 1)))
=>(4 * (3 * 2 ))
=>(4 * 6)
=>24
```

繰り返しプロセス

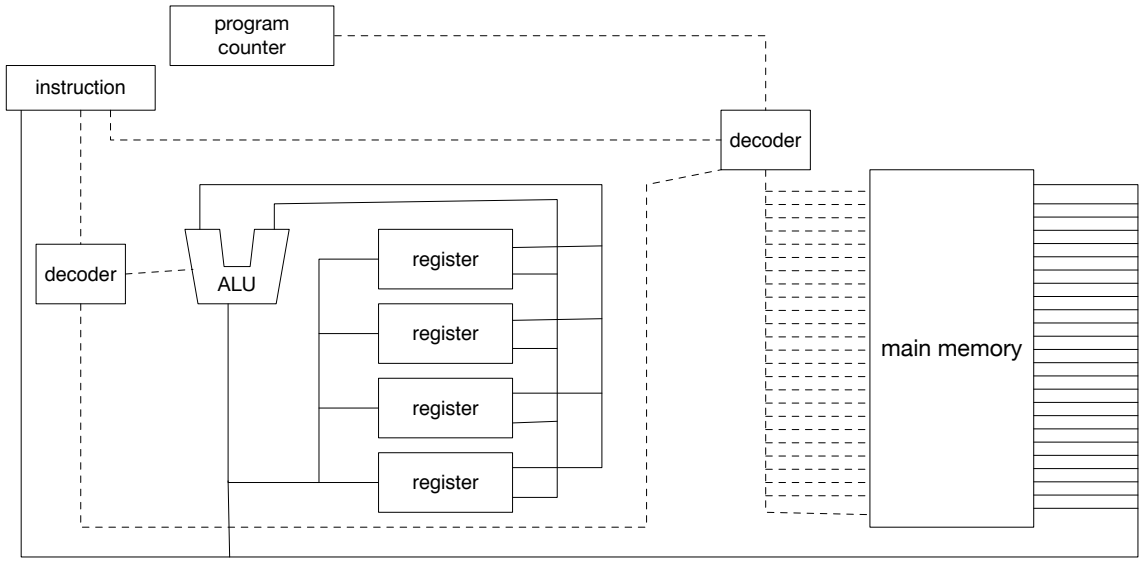
```
  fact-iter(4)
=>fact-iter1( 1,1,4)
=>fact-iter1( 1,2,4)
=>fact-iter1( 2,3,4)
=>fact-iter1( 6,4,4)
=>fact-iter1(24,5,4)
=>24
```

Outline

再帰 (Recursion)
再帰への導入
再帰 (Recursion)
再帰的定義
木構造
プログラムとしての再帰
引数の有効範囲 (Scope)
再帰プログラム
帰納法 (Induction)
繰り返しと再帰
末尾再帰 (Tail Recursion)
コンピュータの中では
再帰のまとめ
課題 S

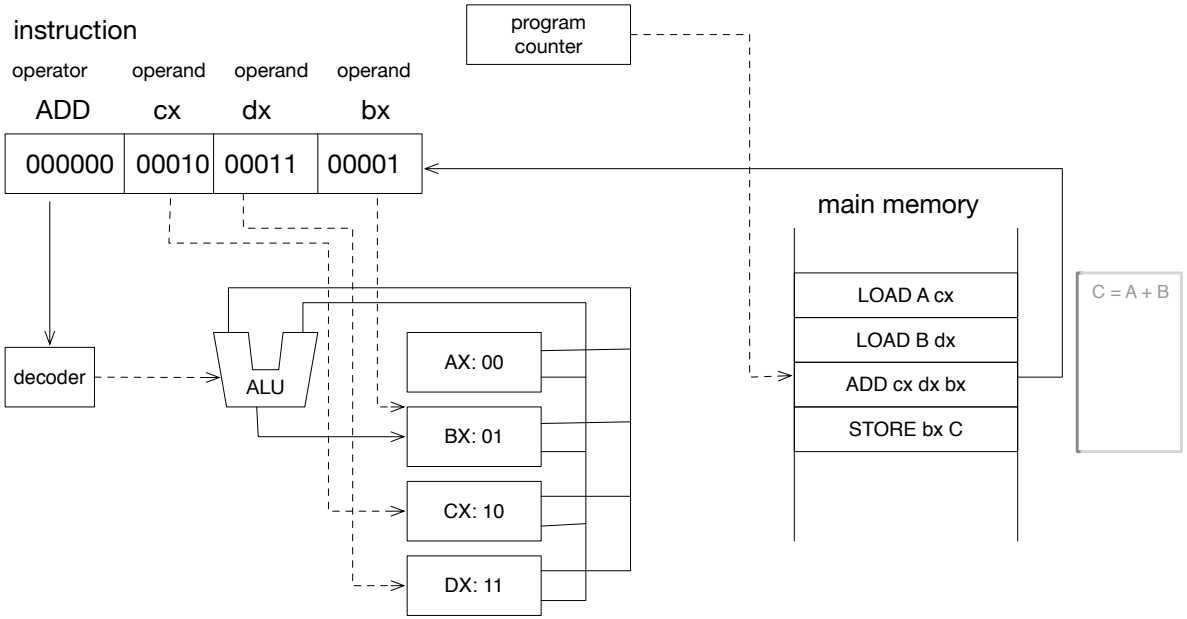
CPU とメモリ

- コンピュータの命令自体も符号化されてます
- CPU (Central Processing Unit) ごとに命令セットも符号も異なっています
- ここでは CPU が命令を実行するサイクルについて見てみます



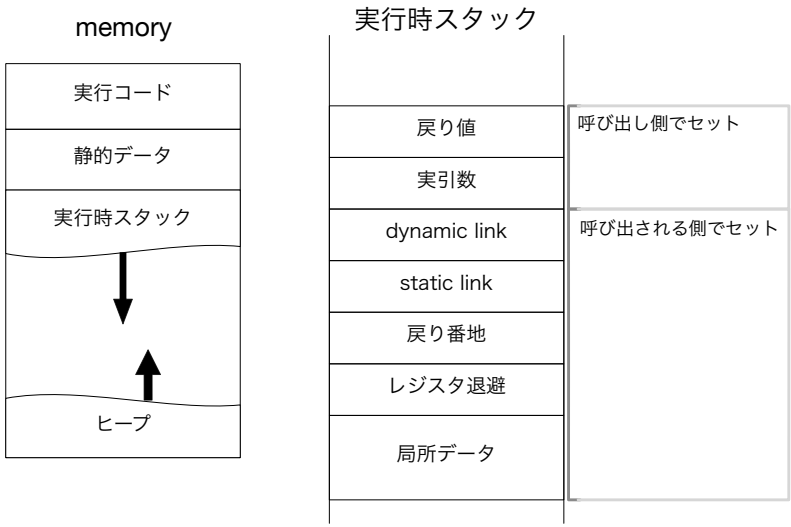
演算のサイクル

1. Instruction に命令をフェッチ
2. メインメモリからレジスタにデータを移動
3. ALU (Arithmetic and Logic Unit) がレジスタからデータを取り出す
4. ALU で演算
5. 結果をレジスタに書き込む
6. レジスタからメインメモリにデータを移動
7. ADD cx dx bx という命令を例にすると下図のようになります



関数を呼び出したとき

- 関数を呼び出すたびに下図の右側のように実行時スタックの領域に保存する
  - 活性レコード (activation record) という
- 終了すれば活性レコードは開放される
- 多くのプログラミング言語は末尾再帰を繰り返しに変換してくれないので繰り返し構文が用意されている



Outline

再帰 (Recursion)
再帰への導入
再帰 (Recursion)
再帰的定義
木構造
プログラムとしての再帰
引数の有効範囲 (Scope)
再帰プログラム
帰納法 (Induction)
繰り返しと再帰
末尾再帰 (Tail Recursion)
コンピュータの中では
再帰のまとめ
課題 S

再帰のまとめ

- 再帰とはある問題をそれより簡単な同じ問題に分解して問題を解く方法
- 既にわかっているそれより前のものから求める
- 一見複雑な問題も再帰的に定義すると単純な計算で解ける
- それ以上分割出来ない問題を解き，結果からより大きい問題の結果を得るというもの (分割統治法 (Divide and Conquer) と呼ばれる)
- E.g. Quiz 1 では複雑な問題をより小さい同じ問題に分解して解いていっている
- 再帰は非常に強力な解法になる

課題 S

- https://sites.google.com/presystems.xyz/elementaryCS/ から fib-skeleton.py をダウンロード
- フィボナッチ数を求める問題です
- まだ，お話していない部分も含まれるのですぐに取り掛からなくていいです
- ソースコード中のコメントを参照して (2)-(4) の問に答えて見てください
- 提出は出来たところまででいいです
- 出来なかったところはコメントにしてください