

# I216: 計算量の理論と離散数学

担当: 上原隆平 & 宮地充子

前半のテキスト:

「計算可能性・計算の複雑さ入門」

渡辺治著, 近代科学社

# I216: Computational Complexity & Discrete Mathematics

- Prof. R. UEHARA and Prof. A. Miyaji

- **Text for former half:**

**"Introduction to Computability and  
Computational Complexity"**

**by Osamu Watanabe,**

**Kindai-Kagaku-sha (in Japanese)**

# 1. 問題とアルゴリズム

1.1. 問題とは, アルゴリズムとは, そして手に負えない問題とは  
問題とは何か

= 関数の計算問題 : 入力  $\rightarrow$  出力  
(数値計算だけでない)

## ソート問題

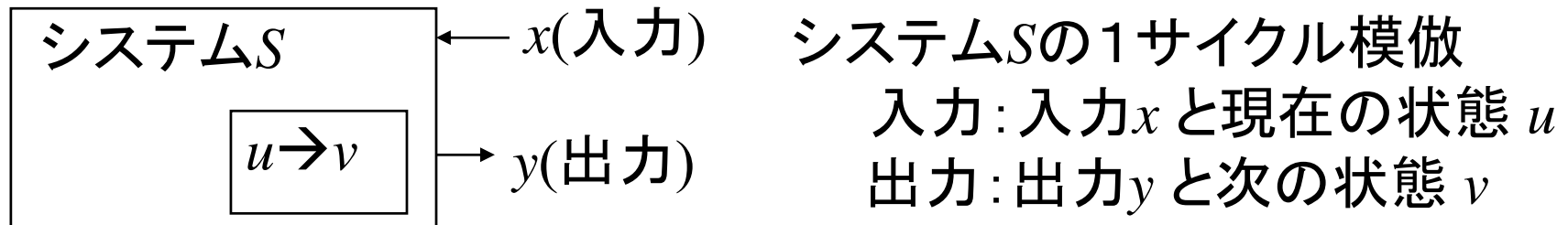
入力: 自然数の列  $a_1, a_2, \dots, a_n$

出力: 入力列を小さい順に並べた列  $a_{i_1}, a_{i_2}, \dots, a_{i_n}$ .

入力と出力が数学的に明確に定義されていること

~~出力: 最高の料理法~~

## 例1.1. コンピュータシステムの働き



$(x, u)$ を $(y, v)$ に対応させる関数  $f_S$  の計算問題

# Chap. 1 Problems and Algorithms

## 1.1. What are problems and algorithms? Intractable problems?

### Problem

= Problem of computing a function: input  $\rightarrow$  output  
(not only numerical computation)

### Sorting Problem

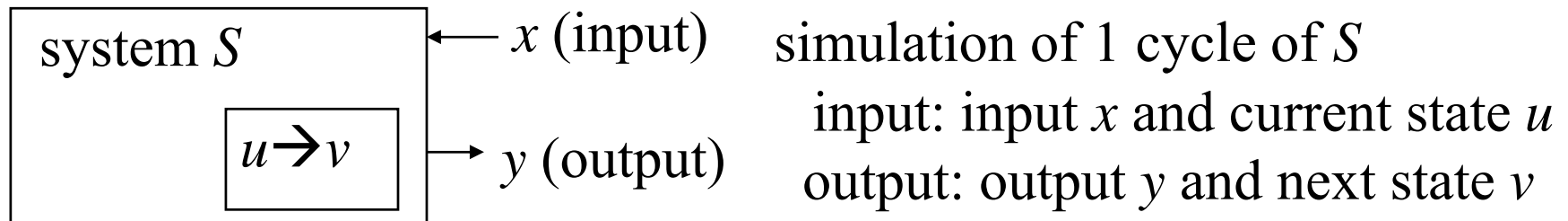
input: sequence of natural numbers  $a_1, a_2, \dots, a_n$

output: increasing order  $a_{i_1}, a_{i_2}, \dots, a_{i_n}$ .

Input and output must be mathematically defined

~~output: the best recipe~~

### Example: Performance of a computer system



problem of computing a function to map  $(x, u)$  to  $(y, v)$

**仮定:** どんな入力に対しても関数は何か値を返す  
 たとえば, 異常入力に対しては ? を返す  
 = 全域関数の立場

システム  $S$  が入力  
 10を仮定していな  
 いなら  $f_S(10)=?$  と  
 定義

## 問題を解くアルゴリズム (algorithm)

入力に対して問題が規定している出力を求める方法.

何を求めるか      の違い  
 如何に求めるか

### 2次方程式の根計算問題

入力: 有理数  $a, b, c$

出力:  $ax^2+bx+c=0$ を満たす  $x$ の一つ

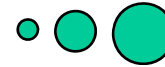
出力が何かは明確だが, 出力を求める方法は?

「アルゴリズム = 方法」

→ アルゴリズム = プログラムとして実現できる計算方法

**Assumption**: system returns some value for any input

e.g. ? is returned for an abnormal input  
 = standpoint of total function



We define  $f_S(10)=?$   
 if the system S  
 cannot handle the  
 input 10.

**Algorithm for solving a problem**

a method for computing an output specified in the problem.

What to be computed?

How to compute?

> **difference**

Problem of calculating a root of a quadratic equation

input: rational number  $a, b, c$

output: an  $x$  that satisfies  $ax^2+bx+c=0$

Output is clearly defined but how can we find it?

"Algorithm = method"

→ algorithm = a method that can be realized as a program

## 難しい問題とやさしい問題

→計算の複雑さ

前半では、「理論的に計算不可能な問題」を扱う

後半では、「実質的に計算できない問題」を扱う

## 手に負えない問題(intractable problems)

「計算可能性の理論」「帰納関数論」

### 例1.2. 計算不可能な問題の例

停止問題(停止性判定問題)

入力:プログラム $A$ (1入力) とそれへの入力  $x$

出力:  $A \rightarrow x$  を与えて実行させると停止するか?

停止するならYES, しないならNO.

この問題は計算不可能であることが証明できる

→後述

## Hard and Easy Problems

→ Complexity of Computation

Former half of the lecture deals with “incomputable” problems.

Latter half of the lecture deals with “hard” problems.

## Intractable problems

"Theory of Computability"

"Theory of Recursive Functions"

### Example 1.2. Incomputable problems

Halting Problem (Problem of deciding halting)

input: a one-input program  $A$  and an input  $x$

output: whether does it terminate if  $x$  is given to  $A$ ?

YES if it terminates and NO otherwise.

We can prove that this problem is incomputable

→ to be explained later



正しいプログラムを作るのは“難しい”が、計算は“やさしい”問題

・コラッツの予想は正しいか？

入力: なし

出力: Yes か No

コラッツの予想:  
どんな自然数でも  
「偶数ならば2で割り、奇数  
ならば3倍して1を加える」  
ことを繰り返すと1になる

計算可能であっても難しい問題

・計算時間がかかり過ぎる

・多量のメモリーが必要である

・計算コストを考えた上での計算可能性

→「計算の複雑さの理論」 → 後述

事実上計算不可能の基準

= 多項式時間で計算することが不可能

→ 手に負えない問題

(多項式時間は手に負える問題の基準ではないことに注意)

否定的な結果を示すための基準

A correct problem is hard to program, but can be computed easily

- Is Collatz Problem true?

input: nothing

output: Yes or No.

**Collatz Conjecture:**  
Iterating “ $\div 2$  if it is even,  
and  $\times 3+1$  if it is odd”  
always returns 1 for any  
positive integer.

### Computable but hard problems

- too much time for computation
- too much space for storage
- computability based on computation cost  
→ "Theory of computational complexity" → later

### Criterion on practical incomputability

= impossible to be computed in polynomial time

→ intractable problems

(Note that polynomial time is not the criterion to be tractable.)

## $NP$ 完全問題

- (1) 解を教えてもらえば, それが解の条件を満たしているか否かは簡単にチェックできる.
- (2) しかし, 解の候補数が(入力のサイズに関して)指数関数的に増大するので, 「一つ一つの候補をチェックする」という単純なプログラムでは, 時間計算量が指数関数的に増大してしまう.

1950年代から研究開始

### 例1.3. 箱詰め問題(bin packing problem)

- $n$  個の棒状の荷物: 長さは  $a_1, a_2, \dots, a_n$
- 長さがそれぞれ  $b$  の  $k$  個の箱にうまく収まるように詰めることができるか?

単純な方法では, 少なく見積もっても指数関数的な時間がかかってしまう.

## *$\mathcal{NP}$ Complete Problem*

- (1) Given a solution to the problem, it can be easily checked whether it satisfies the condition for solution.
- (2) But, a simple program checking every solution candidate takes exponential time since the number of candidates grows exponentially.

The study starts in 1950's.

### **Example 1.3.** Bin packing problem

- $n$  items of lengths  $a_1, a_2, \dots, a_n$
- Is it possible to pack all the items into  $k$  boxes of length  $b$ ?

A simple algorithm takes at least exponential time.

$\mathcal{P} \neq \mathcal{NP}$ 予想

100万ドルの懸賞金つき!!

「 $\mathcal{NP}$ 完全問題は多項式時間では解けない。」

例1.4. どんな多項式も指数関数よりは緩やかに増加する.

$p(n)$ を任意の多項式,  $e(n)$ を任意の指数関数とすると

→十分大きな  $n$  に対して,  $p(n) \ll e(n)$  が成り立つ.

(例) 十分大きな  $n$  については  $n^{10000} \ll 1.0000001^n$

定義1.2.

- (1) その問題を解くプログラムが存在しない問題を, (計算不可能という意味で) 手に負えない問題という.
- (2) その問題を多項式時間以内で解くプログラムが存在しない問題を, (計算困難という意味で) 手に負えない問題という.

## $\mathcal{P} \neq \mathcal{NP}$ Conjecture

One of \$1000000 Millennium prize problems

Any  $\mathcal{NP}$  complete problem cannot be solved in polynomial time.

**Example 1.4.** Any polynomial function grows more slowly than an exponential function.

Let  $p(n)$  be any polynomial function and  $e(n)$  be any exponential function

→ for sufficiently large  $n$ ,  $p(n) \ll e(n)$ .

e.g.,  $n^{10000} \ll 1.0000001^n$  for sufficiently large  $n$ .

### **Definition 1.2.**

- (1) A problem for which there is no program to solve it is called "intractable" in the sense that it cannot be computed.
- (2) A problem for which there is no program to solve it in polynomial time is called "intractable" in the sense that it is hard to be computed.

## 1.2. 準備

### 1.2.1. 集合, 関数, 述語など

#### (1) 数

特に断らない限り, 自然数(0を含む)のみを扱う.

$x$  が実数のとき,  $[x]$  で  $x$  の整数部を表す(切り捨て)

#### (2) 集合

標準的な記号:  $A \cup B, A \cap B, \bar{A}, A \subseteq B$

$A \times B$ :  $A$  と  $B$  の要素の順序対全体の集合

$\|A\|$ : 集合  $A$  の要素数

原則として, 大文字アルファベットで集合を表す. 例外は

$\Gamma$  我々のプログラミング言語で文字として許される記号

$\Sigma$   $\{0, 1\}$

$\mathbb{N}$  自然数の全体(0を含む)

## 1.2. Preparation

### 1.2.1. Set, function, predicate, and etc.

#### (1) Number

Only natural numbers (including 0) are considered.

$[x]$  represents the integral part of  $x$  (rounding off)

#### (2) Set

standard notations:  $A \cup B, A \cap B, \bar{A}, A \subseteq B$

$A \times B =$  a set of all pairs of elements of  $A$  and  $B$

$\|A\|$ : number of elements of the set  $A$

In principle, sets are denoted by capital letters.

Exception:  $\Gamma$  symbols used in programs

$\Sigma \quad \{0, 1\}$

$\mathbf{N}$  a set of all natural numbers (including 0)



$X$ : 任意の有限集合

$X$ 上の文字列 =  $X$ の各要素を“文字”とみなし, その文字  
を有限個(0個を含む)並べて得られたもの

文字列の長さ = 文字列を構成する文字の数

$|x|$ : 文字列  $x$  の長さ

文字列 0100 の長さは4.

長さ0の文字列を空列といい,  $\varepsilon$  という記号で表す.

$\Sigma^*$ : 0 と1を並べてできる文字列全体の集合(空列を含む)

辞書式順序(もどき): 長さ優先の辞書式順序

$x, y$ :  $\Sigma^*$  上の文字列

$x < y \Leftrightarrow$  (a)  $|x| < |y|$ , あるいは,

(b)  $|x| = |y|$  で最初に異なる文字を  $x_i, y_i$  とするとき

$$x_i < y_i.$$

(例)  $101 < 0011 < 0100$

通常辞書式順序との違いは何か?

なぜ, このような順序を導入するのか?

$X$ : any finite set

strings on  $X$  = a finite sequence of elements of  $X$  (each element of  $X$  is regarded as a "letter")

length of a string = the number of letters in the string

$|x|$ : length of a string  $x$

the length of a string "0100" is 4.

A string of length 0 is called an empty string, denoted by  $\epsilon$ .

$\Sigma^*$ : a set of all strings consisting of 0 and 1 (including empty string)

**(Pseudo-)Lexicographical Order**: (with length preferred)

$x, y$ : strings on  $\Sigma^*$

$x < y \iff$  (a)  $|x| < |y|$ , or otherwise,

(b) for the first different letters in  $x$  and  $y$  be  $x_i, y_i$ ,

$x_i < y_i$ .

(example)  $101 < 0011 < 0100$

What is the difference from usual lexicographical order?

What is the reason of introducing such an order?

## 論理記号

## 用例

$P \wedge Q$

$P \vee Q$

$\neg P$

$P \rightarrow Q$

$P \leftrightarrow Q$

$\exists x \in L[R(x)]$

$\forall x \in L[R(x)]$

$\overset{\infty}{\exists} x \in L[R(x)]$

$\overset{\infty}{\forall} x \in L[R(x)]$

## 意味

 $P$  かつ  $Q$  $P$  または  $Q$  $P$  でない $P$  ならば  $Q$  ( $\neg P \vee Q$  と同値) $P$  ならば  $Q$  かつ  $Q$  ならば  $P$  $L$  に属するある  $x$  で  $R(x)$  $L$  に属する任意の  $x$  で  $R(x)$  $R(x)$  となる  $x$  が  $L$  の中に無限個ある $L$  の中の有限個を除いたすべての  $x$  で  $R(x)$ 

演習:

$\overset{\infty}{\forall} x \in L[R(x)]$  ならば必ず  $\overset{\infty}{\exists} x \in L[R(x)]$   
 だが、逆は真ではない。なぜか。

## Logic symbols

### example

$$P \wedge Q$$

meaning

$P$  and  $Q$

$$P \vee Q$$

$P$  or  $Q$

$$\neg P$$

not  $P$

$$P \rightarrow Q$$

if  $P$  then  $Q$

$$P \leftrightarrow Q$$

if  $P$  then  $Q$  and if  $Q$  then  $P$

$$\exists x \in L[R(x)]$$

for some  $x$  in  $L$ ,  $R(x)$  holds

$$\forall x \in L[R(x)]$$

for any  $x$  in  $L$ ,  $R(x)$  holds

$$\overset{\infty}{\exists} x \in L[R(x)]$$

there are infinitely many  $x$  in  $L$  with  $R(x)$

$$\overset{\infty}{\forall} x \in L[R(x)]$$

for any  $x$  except finitely many elements in  $L$ ,  
 $R(x)$  holds

Exercise:

$$\overset{\infty}{\forall} x \in L[R(x)] \text{ implies } \overset{\infty}{\exists} x \in L[R(x)] \quad .$$

However, opposite direction is not true. Why?

## 命題論理式

命題変数と論理記号( $\wedge, \vee, \neg$ )から成る式

例:  $F(X_1, X_2) = [X_1 \vee X_2] \wedge \neg X_1$

## 真偽値の割り当て

与えられた命題論理式の各命題変数に真偽値を代入すること。  
上の例では,

$(0,0), (0,1), (1,0), (1,1)$

の4通りの割り当てが存在。(0:偽, 1:真)

## 命題論理式の分類

**リテラル**: 命題変数あるいはその否定(記号は $\neg$ )

**和項**: リテラルをOR(記号は $\vee$ )でつないだ項

**和積式**: 和項をAND(記号は $\wedge$ )でつないだ式

**二和積式**: 和積式の形の命題論理式で, しかも各和項が  
ちょうど2個のリテラルからなるもの

**三和積式**: 和積式の形の命題論理式で, しかも各和項が  
ちょうど3個のリテラルからなるもの

**拡張命題論理式**: 論理記号として,  $\rightarrow, \leftrightarrow$  も許したものの

## Propositional Logic Expression

Expression consisting of propositional variables and logic symbols  $\wedge, \vee, \neg$  e.g.  $F(X_1, X_2) = [X_1 \vee X_2] \wedge \neg X_1$

## Truth assignment

Assigning truth value to each propositional variable in each logic expression. e.g. there are 4 different assignments (0,0), (0,1), (1,0), (1,1) for the expression above. (0: false, 1: true)

## Classification of propositional expressions

**literal:** logic variable or its negation

**sum term:** term in which literals are connected by OR

**sum-multiply expression:** expression in which sum terms are connected by AND

**2-sum expression:** logic expression in the sum-multiply form and each sum term consists of exactly two literals

**3-sum expression:** logic expression in the sum-multiply form and each sum term consists of exactly three literals

**extended logic expression:** one that may include  $\rightarrow, \leftrightarrow$  as well.

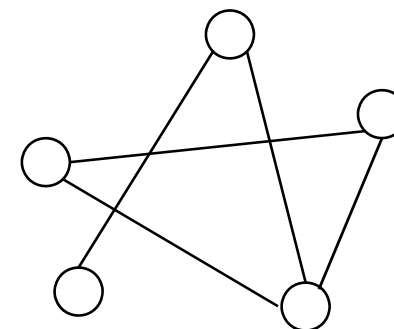
## グラフの表現

グラフの各頂点に1から順に番号をふる.

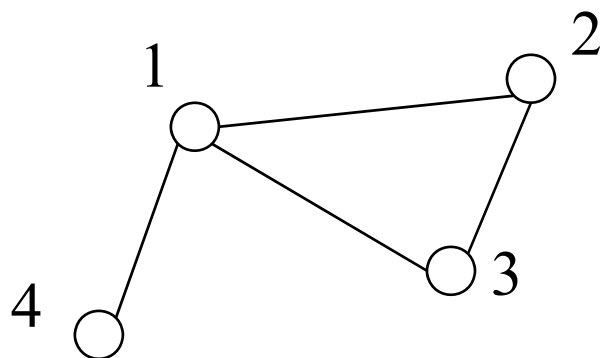
グラフの辺:  $(i, j)$

グラフの表現  $G = (n, E)$

$n$ : 頂点数,  $E$ : 辺の集合



無向グラフの例:

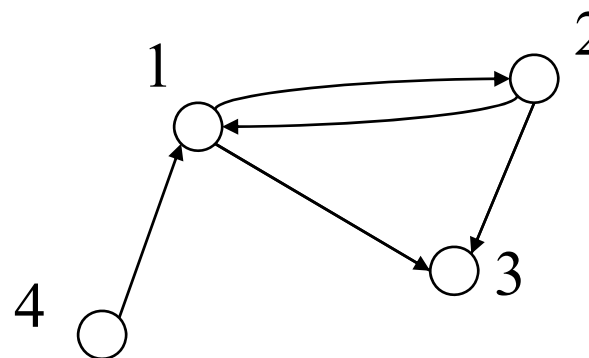


$E = \{(1,2), (1,3), (2,3), (1,4)\}$

$G = (4, \{(1,2), (1,3), (2,3), (1,4)\})$

例えば  $(1,2)$  と  $(2,1)$  は区別しない

有向グラフの例:



$E = \{(1,2), (2,1), (1,3), (2,3), (4,1)\}$

$G = (4, \{(1,2), (2,1), (1,3), (2,3), (4,1)\})$

辺の向きを区別する

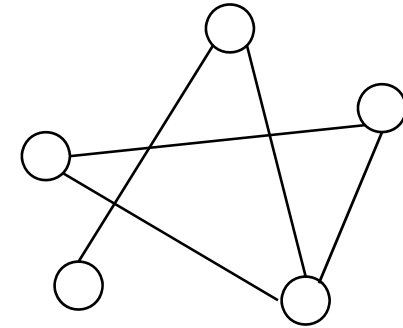
## Expression of a graph

graph vertices are numbered sequentially

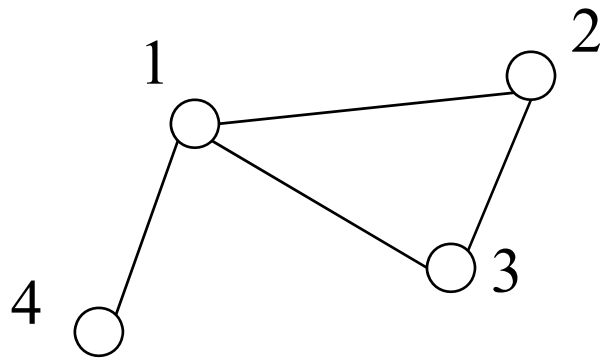
graph edge:  $(i, j)$

expression of a graph  $G = (n, E)$

$n$ : number of vertices,  $E$ : set of edges



Example of a graph:

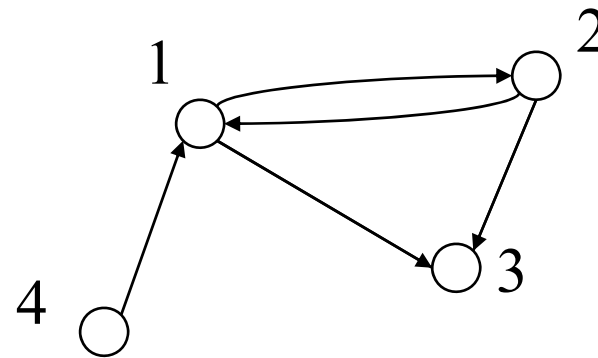


$E = \{(1,2), (1,3), (2,3), (1,4)\}$

$G = (4, \{(1,2), (1,3), (2,3), (1,4)\})$

Do not distinguish  $(1,2)$  from  $(2,1)$

Example of a directed graph:



$E = \{(1,2), (2,1), (1,3), (2,3), (4,1)\}$

$G = (4, \{(1,2), (2,1), (1,3), (2,3), (4,1)\})$

$(1,2)$  and  $(2,1)$  are different arcs



## 1.2.2. アルゴリズムの記述方法

### PASCAL風の手続き型プログラミング言語

例: 2進表現で与えられた自然数を通常 of 自然数に変換

1. prog TR(input x: string on  $\Sigma$ ): integer;
2. label LOOP;
3. var n: num; c: string;
4. %単にstringと型指定したときはstring on  $\Gamma$ 型を意味する.
5. begin
6.   if  $x \neq 0 \wedge \text{head}(x) = 0$  then LOOP: goto LOOP: end-if;
7.   %2進表記でないものが入力されると無限ループに入る.
8.   n:=0;
9.   while  $x > \varepsilon$  do %  $\varepsilon$ は空列を表す定数
10.     c:=head(x);
11.     if c=1 then n:=2\*n+1
12.       else n:=2\*n end-if;
13.     x:=right(x)
14.   end-while;
15.   halt(n)
16. end.

## 1.2.2. Algorithm Description

PASCAL-like procedural programming language

Ex. Conversion from a binary natural number into an ordinary one.

```
1.  prog TR(input x: string on  $\Sigma$ ): integer;
2.  label LOOP;
3.  var n: num; c: string;
4.  % string implies a type of string on  $\Gamma$ .
5.  begin
6.    if  $x \neq 0 \wedge \text{head}(x) = 0$  then LOOP: goto LOOP: end-if;
7.    %if non-binary expression is input then goto infinite loop
8.    n:=0;
9.    while  $x > \varepsilon$  do %  $\varepsilon$  is a constant for an empty string
10.     c:=head(x);
11.     if c=1 then n:=2*n+1
12.       else n:=2*n end-if;
13.     x:=right(x)
14.   end-while;
15.   halt(n)
16. end.
```

**注意事項:**

- ・入出力に関する記述は省く.
- ・TR: プログラム名 ( )内が入力変数とその型指定,  
( )の右が出力の型
- ・ $f_{TR}$ : プログラムTRが計算する(部分)関数
- ・正常終了と無限ループ
  - ・出力が得られるのはhalt文で正しく停止するときのみ.
  - ・出力が得られない場合, プログラムが計算する関数値は未定義とみなす.

$$f_{TR}(001) = \perp$$

**Remarks:**

- description concerning input and output are omitted.
- TR: program name (input variable and its type declaration)  
the type of output follows
- $f_{TR}$ : the (partial) function computed by the program TR
- normal termination and infinite loop
  - Output is obtained only when it terminates correctly by a halt sentence.
  - When an output is obtained, the function value computed by the program is considered as "undefined"

$$f_{TR}(001) = \perp$$

## 変数の型

自然数型: num型

文字列型: string型

文字列を構成する“文字”として許される記号 $0, 1, 2, \dots, a, b, \dots$ の全体を $\Gamma$ とする.

## 文字列型データの基本演算

$\text{head}(x)$   $x$  の最初の1文字

$\text{right}(x)$   $x$  の2文字目から右の部分

$\text{tail}(x)$   $x$  の最後の1文字

$\text{left}(x)$   $x$  の先頭から最後の2文字目までの部分

$x \# y$   $x$  と  $y$  の接続

$x \leq y$  長さ優先の辞書式順序による大小比較

ただし,  $\text{head}(\varepsilon)=\text{right}(\varepsilon)=\text{tail}(\varepsilon)=\text{left}(\varepsilon)=\varepsilon$

## Types of variables

natural number type: type num

string type:

Let  $\Gamma$  be a set of all symbols  $0, 1, 2, \dots, a, b, \dots$  used in strings

## Elementary operations on strings

$\text{head}(x)$  the first letter of  $x$

$\text{right}(x)$  the part of  $x$  after its first letter

$\text{tail}(x)$  the last letter of  $x$

$\text{left}(x)$  the part of  $x$  before its last letter

$x \# y$  concatenation of  $x$  and  $y$

$x \leq y$  comparison based on lexicographic order with length preferred

where,  $\text{head}(\varepsilon)=\text{right}(\varepsilon)=\text{tail}(\varepsilon)=\text{left}(\varepsilon)=\varepsilon$

## 2.2. 計算の基本要素

「データ」や「プログラム」を最小限の資源で表現  
...対象を絞ることで議論を単純化する

### 2.2.1. データ表現のための基本要素

データ表現のためには文字列型だけで十分.

構造型などを含め,

すべてのデータ(型)は $\Sigma(=\{0,1\})$ 上の文字列型で代用可能

補題2.1.すべての基本データ型は $\Sigma^*$ 型と構造型で実現できる.

自然数型, 整数型, 実数型, 論理値型, 文字列型

(略証)自然数の基本演算(加減乗除, 大小比較)に対応する  
 $\Sigma^*$ 上での関数を用意すればよい。

## 2.2. Elements of Computation

String data type suffices to represent data. All data types can including structured type be represented by strings on  $\Sigma$ .

**Lemma 2.1: All elementary data types can be represented by  $\Sigma^*$  types and structured type.**

types for natural numbers, integers, reals, truth values, strings

(Outline of Proof) It is sufficient to prepare functions on  $\Sigma^*$  for elementary operations on natural numbers (e.g., plus, minus, multiply, divide, compare).



### 自然数の1進表記

自然数  $n \rightarrow 0$  を  $n$  個並べる

$[n]$ : 自然数  $n$  の2進表記      $[4] \rightarrow 100$

$\bar{n}$  : 自然数  $n$  の1進表記      $\bar{4} \rightarrow 00000$

例2.2. 一般の文字列 ( $\Gamma$ 上の文字列) も  $\Sigma$ 上の文字列で表現可能.

e.g. 8ビットの2進列でのコード化(ASCIIコードなど)

補題2.2. すべての構造型は  $\Sigma^*$ 型で表現できる.

## Unary representation of a natural number

natural number  $n \rightarrow$  sequence of  $n$  0s

$\lceil n \rceil$ : binary representation  $\lceil 4 \rceil \rightarrow 100$

$\bar{n}$ : unary representation  $\bar{4} \rightarrow 00000$

Ex. 2.2: Ordinary letters are also represented by binary strings  
e.g. each letter is coded in 8 bits

**Lemma 2.2. All structure types are represented by  $\Sigma^*$  type.**

定理2.3. われわれのプログラミング言語のすべてのデータ型とその上の基本演算は $\Sigma^*$ 型とその上の基本演算だけで実現できる.

### 「われわれのコード化法」

$[x]$ : データ  $x$  を表す  $\Sigma^*$  の元 ( $x$  のコード)

$[w]$ :  $\Sigma^*$  の元  $w$  が表しているデータ

例2.6. プログラムも(改行コード入りの)文字列と見なしてコード化.

```

prog A ...      A = 0111000 01110010 01101111 ...
begin          p          r          o ...
:
end.           01100101 01101110 00101110 ...
               e          n          d

```

もっと使いやすい  
コード化もあるが,  
当面はこれで.

**Theorem 2.3. All the data types and elementary operations in our programming language can be realized on  $\Sigma^*$ .**

“Our encoding method”

$\lceil x \rceil$ : an element of  $\Sigma^*$  representing a data  $x$  (a code of  $x$ )

$\lfloor w \rfloor$ : a data represented by an element  $w$  of  $\Sigma^*$

**Ex.2.6.** Programs are also coded by considering them as strings

prog A ...	A =	0111000	01110010	01101111	....
begin		p	r	o	....
:					
end.		01100101	01101110	00101110	...
		e	n	d	

We could use a different coding method, but ...

## 2.2. 計算の基本要素

「データ」や「プログラム」を最小限の資源で表現  
...対象を絞ることで議論を単純化する

### 2.2.2. 制御機構のための基本要素

補題2.4. 関数プログラム(関数定義と関数呼び出し)は,  
すべてif文とgoto文によって実現できる.

(略証)

フローチャート → if文とgoto文

再帰呼び出し → スタックを用いて書きなおす

補題2.5. すべての制御構造はif文とgoto文によって実現できる.

定理2.6. すべての制御構造はif文とwhile文によって実現できる.  
(例に基づいて証明)



プログラムの  
「標準形」

## 2.2.2. Elements for Control Mechanism

**Lemma 2.4: A function (definition and call of function) can be implemented by if and goto statements.**

(Proof sketch)

flowchart → if statement and goto statement

recursive call → can be rewritten using a stack

**Lemma 2.5. All the control mechanisms can be realized by if and goto statements.**

**Theorem 2.6. All the control structures can be realized by if and while statements.**

(Proof based on examples)



**“Standard Form”  
of a program**

```
% xが0*かどうかを判定するプログラム
prog A(input x:  $\Sigma^*$ ):  $\Sigma^*$ ;
label LOOP; var a:  $\Sigma^*$ ;
begin
LOOP: if x=  $\epsilon$  then halt(1) end-if;
      a:=head(x); x:=right(x);
      if a=1 then halt(0) else goto LOOP end-if
end.
```

これを次のように変形する.

- (1) プログラムの各行は次のいずれか.
  - (a) 代入文とgoto文
  - (b) if  $\Sigma^*$ 上の比較 then goto ... else goto ... end-if
  - (c) halt(変数)
- (2) プログラム本体の各行には, L1から始まり, L2, L3,...と順にラベルづけされている.
- (3) ただし, (c)の形の行はプログラムの最後に1箇所しか現れず, それはL0とラベル付けされている.

```
% program to determine whether x is 0* or not
prog A(input x:  $\Sigma^*$ ):  $\Sigma^*$ ;
label LOOP; var a:  $\Sigma^*$ ;
begin
LOOP: if x=  $\epsilon$  then halt(1) end-if;
      a:=head(x); x:=right(x);
      if a=1 then halt(0) else goto LOOP end-if
end.
```

**Convert it as follows.**

- (1) Each line of a program is one of the followings:
  - (a) substitution, goto statement
  - (b) if comparison on  $\Sigma^*$  then goto ... else goto ... end-if
  - (c) halt(variable)
- (2) Each line in the program body is labeled as L1, L2, ...
- (3) The line of the form (c) above appears only once in the program and it is labeled as L0.



```

prog A(input x:  $\Sigma^*$ ):  $\Sigma^*$ ;
label LOOP; var a:  $\Sigma^*$ ;
begin
LOOP: if x=  $\epsilon$  then halt(1) end-if;
      a:=head(x); x:=right(x);
      if a=1 then halt(0) else goto LOOP end-if
end.

```



```

prog B(input x:  $\Sigma^*$ ):  $\Sigma^*$ ;
label L0, L1, L2, L3, L4, L5, L6;
var a,c:  $\Sigma^*$ ;
begin
L1: if x=  $\epsilon$  then goto L5 else goto L2 end-if;
L2: a:=head(x); goto L3;
L3: x:=right(x); goto L4;
L4: if a=1 then goto L6 else goto L1 end-if;
L5: c:=1; goto L0;
L6: c:=0; goto L0;
L0: halt(c)
end.

```

(3-2) goto 文で次に実行する行に移動

(3-1) 通常の処理+次に実行する行を決める

(2) haltの値を設定

(1) halt文を追加

```

prog A(input x:  $\Sigma^*$ ):  $\Sigma^*$ ;
label LOOP; var a:  $\Sigma^*$ ;
begin
LOOP: if x=  $\epsilon$  then halt(1) end-if;
      a:=head(x); x:=right(x);
      if a=1 then halt(0) else goto LOOP end-if
end.

```



```

prog B(input x:  $\Sigma^*$ ):  $\Sigma^*$ ;
label L0, L1, L2, L3, L4, L5, L6;
var a,c:  $\Sigma^*$ ;
begin
L1: if x=  $\epsilon$  then goto L5 else goto L2 end-if;
L2: a:=head(x); goto L3;
L3: x:=right(x); goto L4;
L4: if a=1 then goto L6 else goto L1 end-if;
L5: c:=1; goto L0;
L6: c:=0; goto L0;
L0: halt(c)
end.

```

(3-2) Jump to the next line indicated by goto

(3-1) Usual process + goto next line

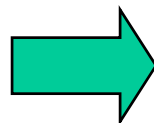
(2) Set values of halt

(1) Add halt

```

prog B(input x:  $\Sigma^*$ ):  $\Sigma^*$ ;
label L0, L1, L2, L3, L4, L5, L6;
var a,c:  $\Sigma^*$ ;
begin
L1: if x=  $\varepsilon$  then goto L5 else goto L2 end-if;
L2: a:=head(x); goto L3;
L3: x:=right(x); goto L4;
L4: if a=1 then goto L6 else goto L1 end-if;
L5: c:=1; goto L0;
L6: c:=0; goto L0;
L0: halt(c)
end.

```



```

prog C(input x:  $\Sigma^*$ ):  $\Sigma^*$ ;
var pc: num; a,c: $\Sigma^*$ ;
begin
  pc:=1;
  while pc != 0 do
    case pc of
      1: if x=  $\varepsilon$  then pc:=5 else pc:=2 end-if;
      2: a:=head(x); pc:=3;
      3: x:=right(x); pc:=4;
      4: if a=1 then pc:=6 else pc:=1 end-if;
      5: c:=1; pc:=0;
      6: c:=0; pc:=0;
    end-case;
  end-while;
  halt(c)
end.

```



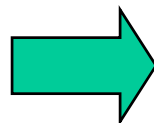
goto Lk  $\rightarrow$  pc:=k;

ただし、case文は  
実際にはif文の  
組み合わせで実現。

```

prog B(input x:  $\Sigma^*$ ):  $\Sigma^*$ ;
label L0, L1, L2, L3, L4, L5, L6;
var a,c:  $\Sigma^*$ ;
begin
L1: if x=  $\varepsilon$  then goto L5 else goto L2 end-if;
L2: a:=head(x); goto L3;
L3: x:=right(x); goto L4;
L4: if a=1 then goto L6 else goto L1 end-if;
L5: c:=1; goto L0;
L6: c:=0; goto L0;
L0: halt(c)
end.

```



```

prog C(input x:  $\Sigma^*$ ):  $\Sigma^*$ ;
var pc: num; a,c: $\Sigma^*$ ;
begin
pc:=1;
while pc != 0 do
  case pc of
    1: if x=  $\varepsilon$  then pc:=5 else pc:=2 end-if;
    2: a:=head(x); pc:=3;
    3: x:=right(x); pc:=4;
    4: if a=1 then pc:=6 else pc:=1 end-if;
    5: c:=1; pc:=0;
    6: c:=0; pc:=0;
  end-case;
end-while;
halt(c)
end.

```



goto Lk  $\rightarrow$  pc:=k;

Remark: case statement  
is realized by combination  
of if and goto

**単純プログラム:** 下の要素のみで構成されるプログラム

データ型:  $\Sigma$ 上の文字列型 ( $\Sigma$ 型,  $\Sigma^*$ 型)

基本演算: 文字列型の基本演算

実行文: 代入文, if文(case文), while文, halt文

**定理2.7.** どんなプログラムもそれと同値な単純プログラムに書換えることができる. しかも次のような標準形プログラムに書き直せる

```

prog プログラム名(input ...);
var pc:  $\Sigma^*$ ; ...  $\Sigma$ ; ...  $\Sigma^*$ ; %pcの値は自然数の2進表記
begin
  pc:=1;
  while pc != 0 do
    case pc of
      1: (文);
      2: (文);
      :
      k: (文);
    end-case
  end-while;
  halt(c)
end.

```

各(文)の形は

- ・ if 比較文 then pc:=k1 else pc:=k2 end-if
- ・ 代入文 ; pc:=k;

のいずれか

**Simple program:** a program consisting only of the following elements.

data type: string type on  $\Sigma$  ( $\Sigma$  type,  $\Sigma^*$  type)

elementary operations: elementary operations on strings

execution statements: substitution, if (case), while, halt

**Theorem 2.7 Any program can be rewritten into its equivalent simple program of the following form:**

```

prog Program name(input ...);
var pc:  $\Sigma^*$ ; ...  $\Sigma$ ; ...  $\Sigma^*$ ; % value of pc is a binary representation of an integer
begin
  pc:=1;
  while pc != 0 do
    case pc of
      1: (statement);
      2: (statement);
      :
      k: (statement);
    end-case
  end-while;
  halt(c)
end.

```

each statement is one of the two:

- if comparison then pc:=k1 else pc:=k2 end-if
- substitution; pc:=k;

定理2.8. すべての計算可能関数に対し,  
それを計算する標準形プログラムが存在する.

プログラムカウンタの働きを考えてみよう.

更なる制約 (テキスト101ページ)

「各文は高々定数時間で実行できるものだけ」

$u, u'$ :  $\Sigma$ 型の変数,       $v, v'$ :  $\Sigma^*$ 型の変数

$c$ :  $\Sigma$ 型の定数,       $s$ :  $\Sigma^*$ 型の定数

(代入文)

- |                            |                                    |   |
|----------------------------|------------------------------------|---|
| (1) $u:=c$ ;               | (2) $u:=u'$ ;                      |   |
| (3) $u:=\text{head}(v)$ ;  | (4) $u:=\text{tail}(v)$ ;          |   |
| (5) $v:=s$ ;               | <del>(6) <math>v:=v'</math>;</del> | ? |
| (7) $v:=\text{right}(v)$ ; | (8) $v:=\text{left}(v)$ ;          |   |
| (9) $v:=u \# v$ ;          | (10) $v:=v \# u$ ;                 |   |

(比較文)

- |            |            |
|------------|------------|
| (11) $u=c$ | (12) $v=s$ |
|------------|------------|

**Theorem 2.8** For every computable function, there is a program in the standard form.

Consider a behavior of program counter.

**Further constraints** (refer to 101 page of the textbook)

“each statement must be implemented in constant time”

$u, u'$ : variables of  $\Sigma$  type,       $v, v'$ : variables of  $\Sigma^*$  type

$c$ : constant of  $\Sigma$  type,       $s$ : constant of  $\Sigma^*$  type

**(Substitution)**

- |                             |  |
|-----------------------------|--|
| (1) $u := c;$               | (2) $u := u';$                         |
| (3) $u := \text{head}(v);$  | (4) $u := \text{tail}(v);$             |
| (5) $v := s;$               | <del>(6) <math>v := v';</math></del> ? |
| (7) $v := \text{right}(v);$ | (8) $v := \text{left}(v);$             |
| (9) $v := u \# v;$          | (10) $v := v \# u;$                    |

**(Comparison)**

- |              |              |
|--------------|--------------|
| (11) $u = c$ | (12) $v = s$ |
|--------------|--------------|