コンピュータ・サイエンス1

第7回
コンピュータでの情報の扱い(4)

人間科学科コミュニケーション専攻
白銀 純子

第7回の内容

- コンピュータでの情報の扱い方(4)

前回の復習

負の数の表現[1](p. 9)

- コンピュータでの計算は、全て足し算
  - コンピュータでの計算は、電気回路で実行
    - 電気回路: 電気が通る線を組み合わせて、様々な処理をするためのもの
      (コンピュータを構成する最も基本的な部品)
  - 足し算, 引き算, かけ算, 割り算をするには、それぞれのために専用の回路が必要
    - 足し算専用回路, 引き算専用回路, かけ算専用回路, 割り算専用回路

経済的に良くない

足し算専用回路(加算器)を
組み合わせて他の計算をカバー

負の数の表現[2](p. 9)

- 足し算の組み合わせで他の計算も行う
  - 引き算:「a-b」を、「a+(-b)」(bを負数と考える)
  - かけ算: 足し算の繰り返しとして計算
  - 割り算: 引き算の繰り返しとして計算

コンピュータでも負数を扱う
- 方法1: 真数表現
- 方法2: 2の補数表現

真数表現(p. 9)

- 数を表す2進数に符号(+ or -)を表す1ビットを付加
  - 数の先頭のビットで符号を表す
  - 0が「+」, 1が「-」を表す

「符号ビット」と呼ぶ

10進数	2進数
-3	1011
-2	1010
-1	1001
-0	1000
+0	0000
+1	0001
+2	0010
+3	0011

符号ビット

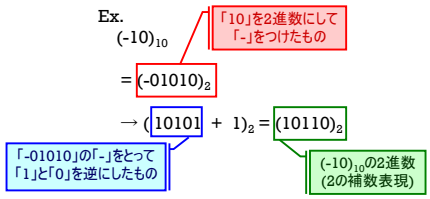
数を表す部分

0が「+0」と「-0」の
2種類できてしまう

具合が悪いので
真数表現はあまり使われない

2の補数表現[1](p. 9)

- 負の数Nを、正の数N(2進数)の0と1を反転させて1を加えた数で表現する方法
  - 0と正の整数(自然数)は、そのまま表現(この計算はしない)



2の補数表現[2](p. 9)

- 2の補数 = 負の数を2進数で表現したもの(コンピュータの世界では)
- 計算方法(例: -20を10桁の2進数に直す)
  1. 2の補数に直したい10進数のマイナスを取り除く
     - $(-20)_{10} \rightarrow (20)_{10}$
  2. 1. の結果を2進数に直す
     - $(20)_{10} = (0000010100)_2$
  3. 2. の結果の0と1を逆にする(0の桁を1、1の桁を0にする)

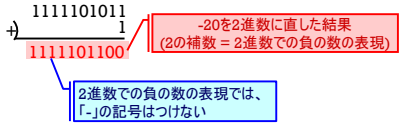
0 0 0 0 0 1 0 1 0 0

⇩

1 1 1 1 1 0 1 0 1 1

2の補数表現[2](p. 9)

- 2の補数 = 負の数を2進数で表現したもの(コンピュータの世界では)
- 計算方法(例: -20を10桁の2進数に直す)
  4. 3. の結果に1を足し算する



2の補数表現の利点(p. 10)

- 引き算(符号付きの足し算)をそのまま足し算として処理できる(自然数と同様に処理できる)

Ex.

$(10 + 3)_{10} = (01010 + 00011)_2 = (01101)_2$

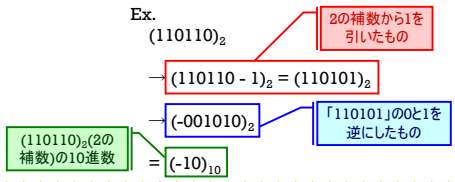
$(-6 + 3)_{10} = (11010 + 00011)_2 = (11101)_2$

⇕

真数表現: 符号付きの足し算を処理するには、別の回路が必要(単純に足すことはできない)

2の補数を10進数に変換[1]

- 2の補数から1を引き、0と1を反転させて10進数になおして「-」をつける
  - 負の数を2の補数に変換するときの逆
  - この計算は、負の数だけ



2の補数を10進数に変換[2]

- 計算方法(例: 1111101100を10進数に直す)
  1. 2の補数から1を引き算する

1111101100

-) 1

1111101011

  2. 1. の結果の0と1を逆にする(0の桁を1、1の桁を0にする)

1111101011

⇩

0000010100

※2の補数→10進数の方法は、10進数→2の補数の逆

2の補数を10進数に変換[2]

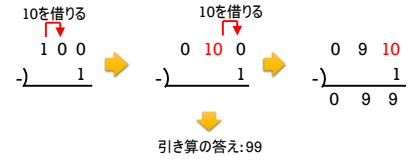
- 計算方法(例: 1111101100を10進数に直す)
  1. 2. の結果を10進数に直す
     - $(0000010100)_2 = (20)_{10}$
  2. 3. の結果に-(マイナス)をつける
     - $(20)_{10} \rightarrow (-20)_{10}$

1111101100を10進数に直した数

※2の補数→10進数の方法は、10進数→2の補数の逆

2進数の引き算[1]

- 10進数の引き算だと…
  - ある桁の引かれる数が引く数より小さければ、1つ大きな桁から10を借りる
    - 10を借りる: 貸した桁から1を引き、借りた桁に10を足す



引き算の答え: 99

2進数の引き算[2]

- 2進数の引き算だと…
  - ある桁の引かれる数が引く数より小さければ、1つ大きな桁から $(10)_2$(10進数で2)を借りる
    - 2を借りる: 貸した桁から1を引き、借りた桁に2を足す

引き算の答え: 011

※コンピュータ的には引き算はしないので、人間が2の補数→10進数の計算をするための引き算

正の数と負の数の見分け方[1]

- 大前提: 数を表す2進数の桁数は決まっている
  - 普通のコンピュータで32桁(or 64桁)

ということは…例えば $(10)_{10}$ は、コンピュータ的には…

0000….00001010 と考えている

28個の「0」

※授業のスライド中では32桁分も書けないので、そのときどきで適当なところで割愛

正の数と負の数の見分け方[2]

- 負の数(2の補数)の計算方法: 負の数Nを、正の数N(2進数)の0と1を反転させて1を加える

コンピュータ的には32桁で数を表すので…

$(-10)_{10} \rightarrow (10)_{10}$

$= (0000\ldots00001010)_2$

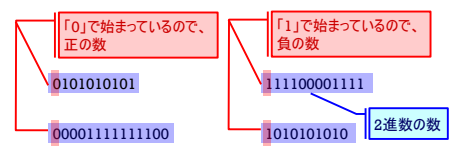
$\rightarrow (1111\ldots11110101 + 1)_2 = (1111\ldots11110110)_2$

28個の「0」も全て「1」に反転される

- 負の数は結果的に一番大きな桁(一番左の桁)が「1」になる
- 一番大きな桁(一番左の桁)が「0」であれば正の数、「1」であれば負の数として扱う

正の数と負の数の見分け方[3]

- 2進数を見たときに…(2の補数を考える場合)
  - 「2の補数を考える」という場合は、先頭の桁を見て、正の数か負の数かを判断
  - 「2の補数を考える」と書かれていない場合は、負の数を考えなくてOK



正の数と負の数の見分け方[3]

- 2進数で表された数は、一番大きな桁(一番左の桁)が「0」であれば正の数、「1」であれば負の数
- 10進数で表された数は、普通に正の数、負の数として計算
  - 正の数であれば、割り算だけで2進数に変換
    - Ex.「+5」と書かれていれば、割り算だけで2進数に変換
  - 負の数であれば、2の補数の方法で2進数に変換
    - Ex.「-5」と書かれていれば、2の補数の方法で2進数に変換
  - ただし、足し算や引き算をした結果を、2の補数を含めて計算すること
    - 計算の結果、一番大きな桁が「0」であれば正の数
    - 計算の結果、一番大きな桁が「1」であれば負の数

桁あふれ(オーバーフロー)(p. 11)

- 2の補数に関係した桁あふれ(オーバーフロー)が起こりうる

Ex. 2進数5桁の計算(10進数で14+5の計算)

```
  0 1 1 1 0
+ 0 0 1 0 1
-----------
  1 0 0 1 1
```

先頭の桁が1になってしまった

先頭の桁が1の場合は、2進数で負の数として扱う

1 0 0 1 1

負の数を表す

計算結果: $(-13)_{10}$(負の数)

2の補数に関係した桁あふれ(オーバーフロー)

桁あふれ[まとめ][1]

- 桁あふれの分類(その1)
  - 足し算等の何らかの計算の結果、コンピュータが扱うことのできる数の桁数の限界を超えてしまう場合
    - Ex. 4桁の数「0110+0110+0110」の計算
      - →本来の計算結果は「10010」で5桁になってしまうので、5桁目が無視されてコンピュータが出す結果は「0010」
      - →コンピュータが出す結果と本来の結果が違うことになる現象

ある意味、コンピュータの性能の限界を超えてしまうということで、その結果として、本来の計算結果とは違う結果がでる現象

※人間やコンピュータがミスをした、という現象ではない

桁あふれ[まとめ][2]

- 桁あふれの分類(その2)
  - 足し算等の何らかの計算の結果、数の正と負が違ってしまう場合
    - Ex. 4桁の数「0110+0110」の計算
      - →本来の計算結果は「1100」で、1桁目が1なので、コンピュータは計算結果を負の数(-4)として取り扱い
      - →本来の計算結果は正の数(12)
      - →コンピュータが出す結果と本来の結果が違うことになる現象

本来の計算結果は正(負)の数なのに、計算結果が負(正)の数になってしまう現象

※人間やコンピュータがミスをした、という現象ではない

桁あふれ[まとめ][3]

- どのような数を計算に使っても、計算結果を見て…
  1. 計算結果が決められた桁数を超えていれば、超えた分の桁の数を削除
     - 桁数を超えた部分 = 削除

     4桁の2進数の計算結果: $(100110)_2$ → 計算結果: $(0110)_2$
  2. 計算結果の先頭の桁が0か1かで、正か負を判断
     - 正の数(先頭の桁が0)であれば、普通に10進数に直す
     - 負の数(先頭の桁が1)であれば、2の補数の方法で10進数に直す

正の数と判断

4桁の2進数の計算結果: $(0101)_2$ → 計算結果: $(5)_{10}$

負の数と判断

4桁の2進数の計算結果: $(1010)_2$ → 計算結果: $(-6)_{10}$

やってみよう!

- (+10)+(+8)を5桁の2の補数として計算し、10進数として表現
- (-10)+(+8)を5桁の2の補数として計算し、10進数として表現

桁あふれも考慮すること

小数の表現方法

整数以外の数を表現するには?

- コンピュータが表現できる数: 整数, 小数
- 整数以外の数
  - 小数
  - 分数
  - n乗根(平方根, 立方根, etc)
  - π(円周率)
  - etc.

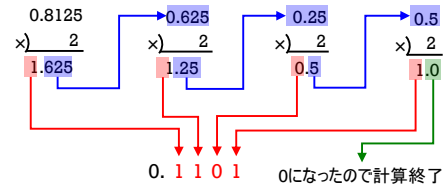
全て小数として表現

2進数の小数を10進数に直す[1]

1. 10進数の整数部分は、通常の方法で2進数に直す
2. 10進数の小数部分に2をかけ算する
3. 2. の結果、整数部分を小数点第1桁にする
4. 3. の結果、小数部分をまた10進数の数とする
   - 小数部分が0になるまで繰り返す
     - ✓ 無限小数になることも(10進数できりのいい数も2進数では無限小数になりえる)
   - 2. の整数部分を小数点第2桁、第3桁、...と置いていく
   - 1. の整数部分と2. の整数部分を並べたものが2進数での小数になる

2進数の小数を10進数に直す[2]

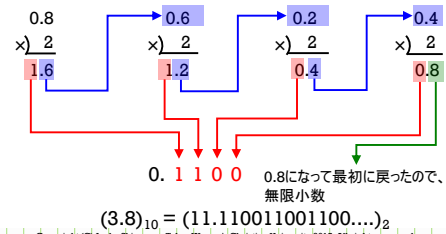
- 10進数の小数を2進数に直すには?(例1)
  - 10進数の0.8125を2進数に直す



$(0.8125)_{10} = (0.1101)_2$

2進数の小数を10進数に直す[3]

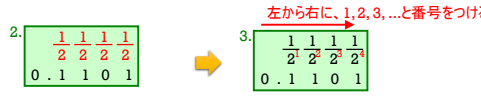
- 10進数の小数部分を2進数に直すには?(例2)
  - 10進数の3.8を2進数に直す(整数部分の3は2進数で11)



$(3.8)_{10} = (11.1100110011001100....)_2$

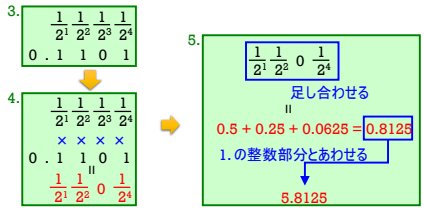
10進数の小数を2進数に直す[1]

- 例: 101.1101 (整数部分101は10進数で5)
1. 2進数の整数部分は、通常の方法で10進数に直す
2. 10進数の小数部分各桁の上に「1/2」を書く
3. 2. で書いた「1/2」の「2」の右肩に左から1, 2, 3, ...と書いていく
   - $1/2^0, 1/2^1, 1/2^2, ...$ができていく



10進数の小数を2進数に直す[2]

4. 各桁の上の「$1/2^n$」と、それぞれの桁の数をかけあわせる
5. 4.の結果を足し合わせ、1.の整数部分とあわせる



小数の表現方式

小数を表現する方法(p. 10)

- 固定小数点方式
- 浮動小数点方式

固定小数点方式[1](p. 10)

- 小数部分の桁数をあらかじめ決めておく方法

Ex. 2進数で表現した数の右から2ビットを小数部分とする場合

10進数の小数	2進数の小数	
0.00	00	00
0.25	00	01
0.50	00	10
0.75	00	11
1.00	10	00
1.25	10	01
1.50	10	10
…	…	
3.75	11	11

⇒ 小数部分は$1/2^2$刻み(0.25刻み)で表現

固定小数点方式[2](p. 10)

- 小数部分の桁数がnの場合:2進数では、小数部分が$1/2^n$刻みで表現

⇓

nを大きくすると、それだけ小数部分を細かく表現可能

※ただし、実際コンピュータは小数も2進数で考えているが、人間が考えるときの便宜上、10進数で考えることが多い

固定小数点方式[3](p. 10)

- 小数を表す桁数が決まっていると…

Ex. 右から2桁を小数部分とすると…

$(2 \div 1000)_{10} = (0.002)_{10} \rightarrow (0.00)_{10}$

小数を正確に表現できない

何桁分の小数部分を持っているかは数値によって異なる
=固定小数点方式で小数を表せる場合は少ない

⇓

浮動小数点方式

浮動小数点方式[1](p. 10)

- 小数: D × 10^nと表現できる
- Ex.
  - $0.5 = 5 \times 10^{-1}$
  - $-0.0625 = -6.25 \times 10^{-2}$
  - $0.0000000084 = 8.4 \times 10^{-9}$

どの数でも「$\times 10^n$」の「10」の部分は同じ

小数を「$D \times 10^n$」の形と考え、「D」と「n」だけ記憶しておく

浮動小数点方式[2](p. 10)

- 浮動小数点方式:
  小数を「$D \times 10^n$」と考え、「D」と「n」を記憶することで小数を表す方式

Ex.
D = 6.25, n = -3の場合: 0.00625
D = 6.25, n = -2の場合: 0.0625
D = 6.25, n = -1の場合: 0.625
D = 6.25, n = 0の場合: 6.25

nの数値が何かで、小数点が仮数の中を動くように見えるから「浮動小数点」と名づけられた

- D: 仮数部
- n: 指数部

と呼ぶ

浮動小数点方式[3](p. 10)

- 仮数部
  - 符号は「0」が「+」、「1」が「-」
  - 固定小数点方式
- 指数部
  - 符号は「0」が「+」、「1」が「-」(ただし、2の補数表現とは別の特殊な形で表現される)

浮動小数点方式[4](p. 10)

- コンピュータでは、指数部は2の累乗
  →小数を「$D \times 2^n$」として考え、「D」と「n」を記憶

Ex.
- $0.5 = 1.0 \times (1/2) = 1.0 \times 2^{-1}$
- $-0.0625 = -1.0 \times (1/16) = 1.0 \times 2^{-4}$

大きな数の表現(p. 10)

- 浮動小数点方式を利用して表現

Ex.
- $2000000000000 = 2 \times 10^{12}$
- $-42500000000000000 = -4.25 \times 10^{17}$

指数部が「+」の数になる

コンピュータは「2」と「+12」,「-4.25」と「+17」を記憶しておく
(実際には、「$\times 10^n$」ではなく「$\times 2^n$」で表現)

大きな数の表現[例](p. 10)

- Windowsの電卓のあるモード(オーバーフローをなかなかしないモード)で…

Ex. 1000000000000000 × 1000000000000000
= 1.e+30

1.0×10^{30}という意味
(浮動小数点方式での表現)

※オーバーフローしにくいモードは、大きな数を浮動小数点方式で表現する
= それだけたくさんの桁がある数を表現できるので、オーバーフローしにくい

桁落ち(1)

- 小数部分が無限のものを扱えるわけではない
  - 例えば割り算で割り切れない数や円周率
    - 小数部分を適当なところで切り捨てる
      (四捨五入ではない)

例えば…1÷3: コンピュータは「0.3333…333」と考える

本来はこの後も無限に続く

コンピュータが扱える小数の桁数:
「有効桁数」と呼ぶ

桁落ち(2)

- 小数部分が無限のものは適当なところまでで切り捨てられる

本来の数よりも、小数の桁数が小さくなってしまう現象

桁落ち

桁落ちの例

- Windowsの電卓のあるモード(オーバーフローをなかなかしないモード)で…

Ex. 電卓での計算: 9999999999999999 × 9999999999999999
= 9.999999999999998e+31 = 9.9999999999999$8 \times 10^{31}$

9999999999999999
×) 9999999999999999
…91
+) …91
…01

桁落ち

つまり本当の計算では…
9999999999999999 × 9999999999999999
= x.xxxx…01

桁落ちが起こると…

- 数が本来の数よりも小さくなってしまう
  - 微妙な数値が必要な場合には要注意
- 桁落ちをした数に大きな数をかけると、本来の数に大きな数をかけたときとの差が大きくなる

例えば…1÷3の結果が桁落ちし、0.333になるとすると

- 0.333に100000をかけると33300
- 1 × 100000を3で割ると(計算の順序を変えると)33333.333
- 本来の1/3に100000をかけると、33333.333….

- コンピュータで計算をするときは、計算の順番に注意
  (割り算はなるべく後にすること)
- 例えば「1÷3 × 100000」の計算は、「1 × 100000」をしてから3で割る

やってみよう![1]

- 2進数1.101を10進数に変換
  (2010年度ITパスポート春季試験問題)
- 2進数に変換した時、有限小数で表現できる10進数は、
  以下のうちどれか
  - 0.1
  - 0.2
  - 0.4
  - 0.5

  (2012年度ITパスポート秋季試験問題)

やってみよう![2]

- 0.0000055を浮動小数点方式で表現
- 0.000000001234を浮動小数点方式で表現
- 456000000000000を浮動小数点方式で表現
  ※3つとも仮数部は小数点第2位の小数とすること
- (10÷7) × 10000を計算
  - 小数点第2位までが有効桁数
  - 桁落ちを考えて計算すること
- 10÷7 × 10000を計算
  - 小数点第2位までが有効桁数
  - 桁落ちの影響がなるべく少ないように計算すること

文字の表現

文字の符号化(p. 13)

- 文字: コンピュータは整数に置き換えて扱う(番号をつけて扱う)
  - 文字を2進数で表現する(「符号化」と呼ぶ)
- 2進数で表現される文字集合
  - 半角英数文字
    - 図形文字
    - 制御文字
  - 多バイト文字
    - 図形文字

※文字集合: 文字の集まり

図形文字と制御文字(p. 13)

- 図形文字: 通常、画面に表示される文字
  - 人間が明示的に書いたり読んだりする文字
  - アルファベット, 数字, ひらがな, 漢字, 記号, etc.
- 制御文字: 通常、画面に表示されない文字
  - コンピュータに何らかの制御をするための文字
  - 改行, TAB, ESC, etc.

ASCII文字

ASCII文字(p. 13)

- ASCII: American Standard Code for Information Interchange
- 半角文字を表す文字集合
  - アルファベット大文字(26文字)
  - アルファベット小文字(26文字)
  - 数字(10文字)
  - 記号(スペース,「,」,「.」, etc.)

1文字を表すために、最低限7ビット必要
(6ビット: 64種類の情報, 7ビット: 128種類の情報)

※1文字を表す2進数の桁数(ビット数)は、どの文字でも同じ(つまり7ビット)

図形文字(p. 13)

- ASCII: 情報量が7ビットで収まるように、扱う文字を取り決めた文字集合
  - アルファベット(大文字・小文字): 52文字
  - 数字: 10文字
  - 記号(スペースを含む): 33文字

図形文字: 合計95文字

番号例(p. 14)

番号	文字	番号	文字	番号	文字
47	0	65	A	97	a
48	1	67	B	98	b
49	2	68	C	99	c
50	3	69	D	100	d
51	4	70	E	101	e
52	5	71	F	102	f
53	6	72	G	103	g
54	7	73	H	104	h
55	8	74	I	105	I
56	9	75	J	106	j

「数」としての0～9ではなく、「文字」としての0～9

番号の決まり(p. 14)

- 7ビットで1文字を表現 = 128文字表現可能
  - 図形文字: 95文字
    - 32番～126番までが図形文字
  - 残り33文字: 制御文字を表現
    - 0番～31番と127番が制御文字
      ※32番のスペースを、制御文字と考えることもある
- アルファベットの大文字と小文字の番号に規則
  - 小文字の番号は、大文字の番号に32(=$(2^5)_{10}$=$(100000)_2$)を足したもの
    - Ex. A: $(65)_{10} = (01000001)_2$, a: $(97)_{10} = (01100001)_2$
  - 大文字⇔小文字の変換はやりやすい

制御文字例(p. 14)

- BS(8番): Back Space
- HT(9番): TAB
- LF(10番): UNIX系OSでの改行
- CR(13番): Mac OSでの改行
  - Windowsでの改行は、13番と10番を組み合わせて「CRLF」の2つの制御文字で表現される
    - ✓ つまり、Windowsでは1つの改行が2文字分(2ビット)
- DEL(127番): Delete

※OS: Operating System(オペレーティングシステム)

ビット数[1](p. 14)

- コンピュータでは8ビットを1つの単位として扱うことが多い
  → ASCII文字も8ビットで表現すると扱いやすい
  - 8ビットのうち、7ビット分(2進数で7桁目まで)で文字を表現する
  - 残りビット(2進数で8桁目)に常に0を入れておく
    - ASCII文字としては無駄なビット
    - 日本語を表現するときに利用

例えば…
A: 65番(10進数)
= 1000001番(2進数)
= 01000001番(2進数, コンピュータ内での表現)
← ASCII的には無駄な(何も利用していない)ビット

ビット数[2](p. 14)

- 8ビットで1文字を表現 = 1バイトで1文字を表現

「1バイト文字」と呼ばれる

Ex.「Hello, my name is John.」
- アルファベット: 17文字
- 記号: 2文字
- スペース: 4文字

} 23文字 = 23バイト

やってみよう!

- 半角英数字の文章のバイト数を考えてみよう!
  - Mac OSだと?
  - Windowsだと?
  - UNIX系OSだと?

それぞれ、何バイト?

Hello, everybody!
Welcome to Tokyo Woman's Christian University!

※「Hello, everybody!」の後に改行が1つ、その他に改行はなし

多バイト文字

背景[1](p. 15)

- コンピュータは主にアメリカで作られ発展
  - 使う人も、専門家だけだった
  - 当初は扱う文字はアルファベット・数字・いくつかの記号でよかった
- コンピュータが全世界に普及
  - 使う人も、専門家だけではなくなった
  - 英語圏以外の言語圏に対応する必要が出てきた
    - 様々な言語圏の文字に対応

背景[2](p. 15)

- 様々な言語圏の文字: 英語圏の文字と同様に2進数で表現する必要性
  - 英語圏の文字: 128文字で表現可能
    - 1バイト分(256文字分)のうち、128文字分は英語圏の文字
  - 英語圏以外の文字: 128文字以上必要な場合も
    - 日本語
    - 中国語
    - 韓国語
    - etc.

    ➡ 128文字では収まらない

1文字を複数のバイト(多バイト)で表現

文字化け(p. 15)

- 多バイト文字の出現により、文字化けが発生
- 文字化けの原因
  - フォントの問題
  - 文字集合の符号化方式の問題

「文字コード」と呼ぶ

詳細な理由は何であれ…
要は文字を表す2進数(0と1の並び)を、コンピュータが理解していないために発生
- その2進数をどのような形でディスプレイに表示して良いかをコンピュータが理解していないため

フォントの問題[3](p. 15)

- 機種依存文字: コンピュータによって表現のしかたが違う文字
  - それぞれの文字を表現するビット列が、コンピュータによって異なる
    - 1文字1文字を表現するビット列は、JIS(日本の国家規格)などで決まっている
      → コンピュータの環境に依存しない
    - 規格で決められた文字に含まれていない文字もある

      機種依存文字
    - Ex. 丸付き数字(①, ②, …)、ローマ数字(Ⅰ, Ⅱ, …), etc.
- 外字: 登録されていない文字を、利用者が作ったもの
  - 人名漢字などを作ることが多い
  - 作ったコンピュータでしか使えない

符号化方式の問題[1](p. 15)

- 符号化: 1つの文字を2進数(ビット列)として表現すること
- ある1つの文字を表現するビット列が複数通り存在する場合
  - 半角英数の文字はASCIIの1通りだけ
    - 他にも存在するが、ASCIIが世界標準
    - 大部分のコンピュータはASCIIを利用
  - 日本語は複数通り存在

符号化方式の問題[2](p. 15)

- ある文書で使われている符号化方式をコンピュータが判別できないときに文字化け
  - 判別できなければ、コンピュータが普段使っている符号化方式で表示しようとすることが多い

Ex. Windowsで文書を開いた場合
- 文書はJISで書いてあり、WindowsはJISであると判別できなかった
- WindowsはShift JISとして開こうとする
- 文書は文字化けして表示される

※ソフトウェアによっても、符号化方式を判別できるものとできないものがある
(Ex. Windowsのメモ帳は、JISなどは判別できない)

日本語の符号化方式

日本語の文字(p. 15)

- 日本語固有の文字
  - ひらがな
  - カタカナ
  - 漢字
  - かぎカッコ
  - 句読点
  - etc.

日本語の文字(p. 15)

- ASCII
  - 1文字を8ビットで表現→全部で256文字分表現可能
  - 現状で128文字存在(128文字分利用されている)
- 日本語
  - ひらがな: あ～ん(ゐ, ゑなどの旧字を含む), 濁音・半濁音, 小文字(「ぁ」, 「ぃ」など)
  - カタカナ: ア～ン(ヰ, ヱなどの旧字を含む), 濁音・半濁音(ヴを含む), 小文字(「ァ」, 「ィ」, 「ヵ」, 「ヶ」など)

169文字

ひらがな・カタカナだけでもASCIIでは表現できない

日本語文字集合の規格(p. 16)

- 現状での日本語文字集合の規格: JIS X 0208:1997
  - ひらがな・カタカナ・漢字・非漢字文字で6879個

JIS第1水準(使用頻度の高い漢字): 2965個
JIS第2水準(使用頻度の低い漢字): 3390個

- 2^{13} = 8192なので、13ビットで表現可能
- コンピュータ処理では、バイト単位(8ビット単位)が好都合

16ビット(2バイト)で日本語1文字を表現

ASCII文字との区別(p. 16)

- 日本語の文書
  - 日本語の2バイト文字
  - ASCIIの1バイト文字

混在

日本語の2バイト文字(JIS X 0208)とASCIIの1バイト文字は区別する必要
(1つの文書の中で、どれが2バイト文字でどれが1バイト文字か)

- モード切り替えによる区別方法
- ASCII文字の番号を避ける区別方法

モード切り替え(p. 16)

- 文字集合切り替えのための特別な記号を用意
  - ここから先はASCII文字
  - ここから先は日本語文字
  - ここから先は中国語漢字
  - etc.

エスケープシーケンス

通常の文書では頻繁に文字集合が切り替わることがなく、同じ文字集合に属する文字が現れることが多いという性質を利用

- 国際標準規格:ISO-2022
- 日本語に適用したもの:ISO-2022-JP

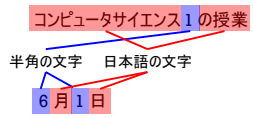
ISO-2022-JPの例(p. 16)

ESC $B	F|	K¥	$N	ESC (B	JP	ESC $B	$@	!#	ESC (B	¥n
	日	本	は		JP		だ	。		

- 「ESC $B」や「ESC (B」、「¥n」などがエスケープシーケンス
- 「F|」や「K¥」、「$N」などは、2バイト文字をASCII文字で表現した場合の文字
  (2バイト文字は、1バイト文字2文字の組み合わせで表現できる)
- エスケープシーケンス「ESC $B」や「ESC(B」は、3バイトずつ
- 「¥n」は改行を表し、半角文字の扱いなので、改行の前に2バイト文字がある場合は、改行と2バイト文字との間にもエスケープシーケンス
- 文章の開始(終了)が1バイト文字の場合は、文章の先頭(終了)にエスケープシーケンスはなし
- 文章の開始が2バイト文字の場合は、文章の先頭にエスケープシーケンスあり

モード切り替えの考え方[1](p. 16)

- 同じ文字集合に属する文字が現れることが多い



- 上側の文章: 日本語文字がいくつか続いた後、半角文字が少しあり、また日本語文字が続く
- 下側の文章: 日本語文字と半角の文字が交互にある

モード切り替えの考え方[2](p. 16)

- 普通の日本語の文章は、日本語の文字がずっと続き、たまに半角文字が出てくるということが多い

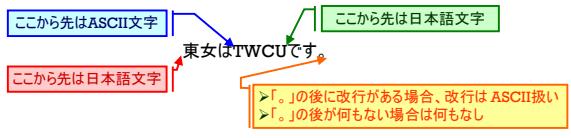
ある言語の文章では、その言語の文字がずっと続き、別の言語の文字はところどころに出てくる

頻繁に文字集合が切り替わるわけではない(ある言語と別の言語の文字が1文字ずつ交互に出てきたり、ということは少ない)
→文字集合がどこで切り替わっているか、わかるようにしておけば良い

モード切り替えの考え方[3](p. 16)

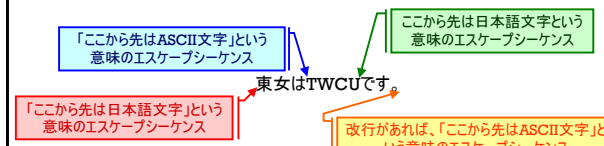
- 文字集合切り替えのための特別な記号を用意
  - ここから先はASCII文字
  - ここから先は日本語文字
  - ここから先は中国語漢字
  - etc.

エスケープシーケンス



モード切り替えの考え方[4](p. 16)

- 文字集合の切り替わりにエスケープシーケンスを入れる
  - 文章の開始(終了)がASCII文字の場合は、文章の先頭(終了)にエスケープシーケンスはなし
  - 文章の開始が日本語文字の場合は、文章の先頭にエスケープシーケンスあり



モード切り替えの考え方[5](p. 16)

- エスケープシーケンスは、3バイトずつ
- 改行は半角文字の扱いなので、改行の前に2バイト文字がある場合は、改行と2バイト文字との間にもエスケープシーケンス
- 文章の開始(終了)が1バイト文字の場合は、文章の先頭(終了)にエスケープシーケンスはなし
- 文章の開始が2バイト文字の場合は、文章の先頭にエスケープシーケンスあり

モード切り替えの考え方[6](p. 16)

- 日本語文字: 1文字2バイト
- ASCII文字: 1文字1バイト
  - 改行はASCII文字扱い
    - Mac OSやUNIX系OSだとASCII文字1文字分 = 1バイト
    - WindowsだとASCII文字2文字分 = 2バイト
- エスケープシーケンス: 1つ3バイト

東女はTWCUです。
- 日本語文字: 6文字 = 12バイト
- ASCII文字: 4文字 = 4バイト
- エスケープシーケンス: 3個 = 9バイト

} 25バイト

※この文章は改行なしとする

モード切り替えの問題(p. 17)

- 文書を先頭から順番に見ていく場合には問題ない
- 文書を途中から見ていくときに問題が生じる
  - 見始めた途中の文字が、ASCII文字か日本語文字か、エスケープシーケンスかが判別できない

Ex. 見始めた途中の文字が「70」番だった場合
- ASCII文字の「F」?
- 日本語文字の一部?
- 韓国語の一部?

検索や置換などの文書処理に時間がかかる

ASCII文字の番号を避ける(p. 17)

- ASCIIで使われていない番号を2バイト文字の番号にあてる方法
  - EUC(日本語のものをEUC-JP)
    - 第1バイト(前半の8ビット)と第2バイト(後半の8ビット)両方でASCII領域が避けられている
  - SJIS(Shift JIS)
    - 第2バイト(後半の8ビット)ではASCII領域も使われている
  - 文章のバイト数は、単純に、日本語文字で2バイト、ASCII文字で1バイトで数えれば良い

EUCとSJIS[1](p. 17)

- ASCII文字: 8個の0と1で、1文字分を表現
  - 実際には、7個の0と1で1文字分を表現
  - 8ビット目は必ず0

必ず0 — XXXXXXXX — 0000000から1111111まで、フルに使ってASCII文字を1文字ずつ表現

= ASCII文字は、0XXXXXXXという形
Ex.「a」を0と1で表現すると: 01100001

8ビット目が1になる番号(1XXXXXXXという形の番号)はASCII文字ではない

EUCとSJIS[2](p. 17)

- ASCIIで使われていない番号を2バイト文字の番号にあてる方法
  - EUC(日本語のものをEUC-JP)
    - 第1バイト(前半の8ビット)と第2バイト(後半の8ビット)両方でASCII領域が避けられている
  - SJIS(Shift JIS)
    - 第2バイト(後半の8ビット)ではASCII領域も使われている
  - 文章のバイト数は、単純に、日本語文字で2バイト、ASCII文字で1バイトで数えれば良い

例えばある文章が…
00110110100101101010010101101010

1から始まっているから日本語文字

0から始まっているからASCII文字

Webやメールでの文字化け(p. 18)

- Webページや電子メール
  - 文字コードの指示が文書中に書かれている場合
    「charset=iso-2022-jp」や「charset=Shift_JIS」など
    ➡ ソフトウェアは指示通りに文字コードを解釈し、表示
  - 文字コードの指示が文書中に書かれていない場合
    ➡ ソフトウェアは文書のデータの特徴から文字コードを判別

⬇

文字化けが発生する場合
- 文書中の文字コードの指示が間違っている場合
- 文字コードの指示がなく、文字コードを判別できなかった場合

やってみよう!

- 日本語の文章のバイト数を考えてみよう!
  - ISO-2022-JPだと?
  - EUC-JPだと?
  - Shift JISだと?

日本はJPだ。 ← アルファベットは半角,文末に1バイトの改行あり(p. 32の問3)

こんにちは!
東京女子大学へようこそ!Welcome!!

アルファベットと記号は半角,「こんにちは!」の後に「CRLF」の改行,「Welcome!!」の後には何も文字はなし

Unicode

言語圏ごとの文字コード(p. 18)

- これまでの多バイト文字の扱い:
  異なる言語圏ごとに文字集合を作成
  様々な文字集合ができてしまって不便
  - コンピュータネットワークの国際化が進んだ
  - コンピュータの資源が豊富になった

⬇

国際文字集合規格として各文字集合を統一化

統一文字コード(p. 18)

- Unicode
  - ASCII
  - ラテン文字
  - 日本語
  - 韓国語
  - 中国語
  - ベトナム語
  - ギリシャ文字
  - 記号
  - etc.

Unicodeバージョン5.2.0で
107361文字

UTF-8(p. 18)

- Unicodeでの代表的な符号化方式
- 1文字を1～6バイトの可変長(文字によってバイト数が異なる)で符号化する方式
  - ASCIIやISO-2022-JP、Shift JIS、EUC-JPは1文字を全て同じバイト数で表現している
- OS(WindowsやMacなどのオペレーティングシステム)でファイル名などの内部処理に利用
  - 半角英数を符号化した結果が、ASCII文字と全く同じになるため、従来のシステムと相性が良い

現在、Unicodeへの移行が急速に進んでいる
- ただし、以前から使われてきたファイルを移行するのは大変なので、完全移行には時間がかかる