２００６年度　情報数学Ｃ

— 有限オートマトンと言語理論 —

九州大学理学部数学科 3 年

2006 年 12 月 01 日

1　有限オートマトンとは？

1.1　アルファベット, 記号, 言語

「正しい英文を判定する機械を作りたい！」

Σ	$=$	$\{A, B, \cdots, Z, a, b, \cdots, z, \underline{(空白)}, ,, ., ! \}$	(アルファベット)
Σ^+	$=$	$\{"abc\_de", "this\_is", "aa", "aaa", "aabc", \cdots\}$	(文字列の集合)
Σ^*	$=$	$\{\varepsilon\} \cup \Sigma^+$	(空文字列 (ε) を含む文字列の集合)

L を正しい英文の集合とすると

$$L = \{"This\_is\_a\_pen.", "I\_am\_a\_boy.", \cdots\}$$

であり, $L \subset \Sigma^*$ である. 文字列 x の長さを $|x|$ で表す.

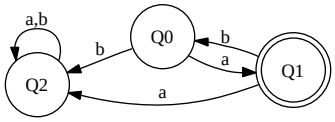
$$|this\_is| = 7, \quad |aaa| = 3, \quad |\varepsilon| = 0$$

「Σ^* の部分集合を言語という！」

アルファベット (Σ) と言語 (L) が与えられたとき, $w \in L$ を判定する機械 M を作りたい.

1.2　定義

簡単な有限オートマトンの例



1. 最初は → ○ の頂点 Q0 から始めます.
2. 自分の持つ文字列の最初の文字の矢印に沿って進みます.
3. 最後の頂点が◎なら, "Yes", ○なら, "No"を答えます.

$$
\begin{aligned}
\Sigma &= \{a, b\} \\
L &= \{a, aba, ababa, abababa, \cdots\} \\
&= \{a(ba)^n | n \geq 0\}
\end{aligned}
$$

$w = aba$ のとき, $w \in L$ だろうか?
$w = baa$ のとき, $w \in L$ だろうか?

「機械 M を (数式を用いて) 厳密に定義する.」

$$
\begin{array}{lcll}
K & = & \{q_0, q_1, q_2\} & :\text{状態集合} \\
F & = & \{q_1\} \quad (F \subset K) & :\text{最終状態集合} \\
q_0 & \in & K & :\text{初期状態} \\
\delta & : & K \times \Sigma \to K & :\text{遷移関数}
\end{array}
$$

$$
\begin{aligned}
\delta(q_0, a) &= q_1 \\
\delta(q_0, b) &= q_2 \\
\delta(q_1, a) &= q_2 \\
\delta(q_1, b) &= q_0 \\
\delta(q_2, a) &= q_2 \\
\delta(q_2, b) &= q_2
\end{aligned}
$$

有限オートマトンの定義

5 つ組 $M = (K, \Sigma, \delta, q_0, F)$ のことを 有限オートマトン(FA) という. 但し, K, Σ は空でない有限集合, $q_0 \in K$, $F \subset K$, $\delta : K \times \Sigma \to K$ とする.

δ は以下のように $\delta^* : K \times \Sigma^* \to K$ へ自然に拡張出来る.

$$
\left\{
\begin{array}{lll}
(1) & \delta^*(q, \varepsilon) = q & \\
(2) & \delta^*(q, ax) = \delta^*(\delta(q, a), x) & (a \in \Sigma, x \in \Sigma^*)
\end{array}
\right. \tag{1}
$$

拡張された δ^* の計算例

$$
\begin{aligned}
\delta^*(q_0, aba) &= \delta^*(\delta(q_0, a), ba) \\
&= \delta^*(q_1, ba) \\
&= \delta^*(\delta(q_1, b), a) \\
&= \delta^*(q_0, a) \\
&= \delta^*(\delta(q_0, a), \varepsilon) \\
&= \delta^*(q_1, \varepsilon) = q_1 \in F
\end{aligned}
$$

受理言語の定義

$$
L(M) = \{w \in \Sigma^* | \delta^*(q_0, w) \in F\} \tag{2}
$$

を有限オートマトン M によって受理される言語という.

正則言語 (RL) の定義

言語 $L(\subset \Sigma^*)$ に対して, $L = L(M)$ となる有限オートマトンが存在するとき L を 正則言語(RL) という.

1.3 正則言語の演算

正則言語の性質

命題 1 L が正則言語のとき, $\bar{L} = \Sigma^* - L$ は正則言語である.

(証明のヒント. L を受理する有限オートマトンを $M = (K, \Sigma, \delta, q_0, F)$ とするとき, $\bar{M} = (\bar{K}, \Sigma, \bar{\delta}, \bar{q_0}, \bar{F})$ で, $\bar{L} = L(\bar{M})$ となる有限オートマトンが作れれば, $\bar{L}$ が正則言語であることが示される.)

命題 2 L_1, L_2 を正則言語とするとき,

(i) $L_1 \cup L_2$ は正則言語である.

(ii) $L_1 \cap L_2$ は正則言語である.

練習問題 1 $\Sigma = \{0, 1\}$, $M_1 = \{\{p_0, p_1\}, \Sigma, \delta_1, p_0, \{p_1\}\}$, $M_2 = \{\{q_0, q_1\}, \Sigma, \delta_2, q_0, \{q_1\}\}$ とする. ただし, $\delta_i : K_i \times \Sigma \to K_i$ ($K_1 = \{p_0, p_1\}$, $K_2 = \{q_0, q_1\}$) は,

$$\begin{array}{llll} \delta_1(p_0, 0) = p_0, & \delta_2(q_0, 0) = q_1 \\ \delta_1(p_0, 1) = p_1, & \delta_2(q_0, 1) = q_0 \\ \delta_1(p_1, 0) = p_1, & \delta_2(q_1, 0) = q_0 \\ \delta_1(p_1, 1) = p_0, & \delta_2(q_1, 1) = q_1 \end{array}$$

とする.

(i) M_1, M_2 の受理する言語は, それぞれ, どのような言語か?

(ii) $L(M_1) \cup L(M_2)$ を受理する有限オートマトン $M_\cup$ を作成せよ.

(iii) $L(M_1) \cap L(M_2)$ を受理する有限オートマトン $M_\cap$ を作成せよ.

1.4 Haskell 言語による有限オートマトンの実現

1.4.1 集合をリストで表現

```
import Data.List
```

Haskell 言語では集合はリストで表わす. 和集合は関数 `union` で実行するが, 重複元がなければリストの結合関数 (`++`や `concat`) で代用出来る. 積集合は関数 `intersect` で実行出来, 重複元を消去するには関数 `nub` を使う.

例 1
```
*Main> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
*Main> concat [[1,2,3],[4,5],[6,7,8,9]]
[1,2,3,4,5,6,7,8,9]
*Main> intersect [1..6] [2,4..10]
[2,4,6]
*Main> [1..6] ++ [2,4..10]
[1,2,3,4,5,6,2,4,6,8,10]
*Main> nub ([1..6] ++ [2,4..10])
[1,2,3,4,5,6,8,10]
*Main> union [1..6] [2,4..10]
[1,2,3,4,5,6,8,10]
```

1.4.2 和集合の計算

関数 concat や関数 union は最初のリストの後に次のリストを接続しますが, 最初のリストが無限リストの場合, 2番目のリスト以降は意味がない. 集合の和集合を最初はリスト中の集合の先頭の要素だけを集めて, その後で, 2番目以降の要素の集合たちの和集合を計算する関数で実現する. 理由は, 無限集合の和集合も計算出来るようにするためである. それが次の関数 union' である.

```
union':: Eq a =>[[a]]->[a]
union' [] = []
union' (x:xs) | x == []   = union' xs
              | otherwise = [head x]++(union' (xs++[tail x]))
```

例 2
```
*Main> union' ["abc","12","4560","789"]
"a147b258c690"
*Main> union' [[1,2,3], [4,5], [6,7,8,9]]
[1,4,6,2,5,7,3,8,9]
*Main> take 10 [0..]
[0,1,2,3,4,5,6,7,8,9]
*Main> take 10 $ union' [[100..], [200..]]
[100,200,101,201,102,202,103,203,104,204]
```

1.4.3 文字列の無限集合 (Σ^*)

集合 Σ に対して, Σ^* は次のように定義される.

(i) $\varepsilon \in \Sigma^*$

(ii) $x \in \Sigma$, $w \in \Sigma^*$ のとき, $xw \in \Sigma^*$

(iii) (i),(ii) だけで Σ^* は定義される.

この定義は, Σ^* が次の再帰集合方程式:

$$S = \{\varepsilon\} \cup \bigcup_{x \in \Sigma} \{xw | w \in S\} \tag{3}$$

の最小解 S であることを意味している.

例 3 $\Sigma = \{a, b\}$ のとき, Σ^* の満たす集合方程式 *(3)* を和集合関数 union' を用いて *Haskell* で記述したのが下の例 abstar である.

```
abstar::[[Char]]
abstar = [""] ++ (union' [["a"++ w | w <-abstar], ["b"++ w | w <-abstar ]])
```

```
*Main> take 10 abstar
["","a","b","aa","ab","ba","bb","aaa","aab","aba"]
```

Haskell の関数定義は, 必要なときに必要なだけ計算されるので, このように無限集合を *Haskell* で記述することが出来る. 結果を表示する際には, 必要な個数を take 関数により切り出せば良い.

上の例を一般化し, 一般のアルファベット集合 Σ に対して, Σ^* を求める関数 sstar を考える.. 集合 $s = \Sigma$ に対して, $sstar(s) = \Sigma^*$ となる関数定義は, 式 (**??**) より下のようになる.

```
sstar :: [Char] -> [[Char]]
sstar [] = []
sstar s  = [""] ++ (union' [ [[x] ++ w | w <- (sstar s)] | x <- s ])
```

例 4 $\Sigma = \{a, b, c\}$ の場合の Σ^* の例を以下に示す.

```
*Main> take 30 $ sstar ['a','b','c']
  ["","a","b","c","aa","ab","ac","ba","bb","bc","ca","cb","cc",
   "aaa","aab","aac","aba","abb","abc","aca","acb","acc","baa",
   "bab","bac","bba","bbb","bbc","bca","bcb"]
```

1.4.4 決定性有限オートマトン

アルファベットの型は Char であり, 文字列の型は String であるが, 状態集合 K を整数の有限集合で表わすこととし, その型を State とする. δ に対して, δ^* を作る関数 dstar は式 (1) より下のようになる.

```
type State  = Int
type States = [State]
-- type String = [Char]

dstar::(State->Char->State)->State->String->State
dstar d s [] = s
dstar d s (a:w) = dstar d (d s a) w
```

Σ^* の部分集合 A に対して, 受理言語集合 $L(M) \cap A$ を計算する関数 accepts は式 (**??**) より以下のように定義される. $A = \Sigma^*$ とした場合, すなわち, Σ が s のとき, accepts d s0 f (sstar s) が, $L(M)$ を与える. 有限オートマトン m に対して, $L(m)$ を計算する関数を language とする.

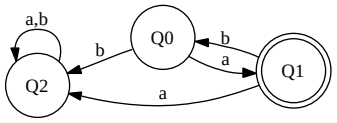
```
type Automaton = (States, [Char], State->Char->State, State, States)

accepts::Automaton->[String]->[String]
accepts (q, s, d, s0, f) ss = [w | w <- ss, (dstar d s0 w) `elem` f]

language::Automaton ->[String]
language (q, s, d, s0, f) = accepts (q, s, d, s0, f) (sstar s)
```

例 5 *(決定性)* 有限オートマトン $M_1 = (\Sigma_1, Q_1, \delta_1, s_1 0, F_1)$ に対して, $\Sigma_1 = \{a, b\}$, $Q_1 = \{0, 1, 2\}$, $s_1 0 = 0$, $F_1 = \{1\}$ とし, $\delta_1 : Q_1 \times \Sigma_1 \to Q_1$ は下表の通り定義する.

$\delta_1(s,c)$	$c = `a'$	$c = `b'$
$s = 0$	*1*	*2*
$s = 1$	*2*	*0*
$s = 2$	*2*	*2*




```
m1::Automaton
m1 = ([0,1,2], ['a','b'], d, 0, [1])
     where d 0 'a' = 1
           d 0 'b' = 2
           d 1 'a' = 2
           d 1 'b' = 0
           d 2 'a' = 2
           d 2 'b' = 2
```

$L(M)$ すなわち, `language m1`, ならびに, $L(M) \cap \{b\}^*$ と $L(M) \cap \{a\}^*$ を計算する.

```
*Main> take 5 $ language m1
["a","aba","ababa","abababa","ababababa"]
*Main> accepts m1 $ take 5 (sstar ['b'])
[]
*Main> accepts m1 $ take 5 (sstar ['a'])
["a"]
```

$L(M) = \{a(ba)^n | n \geq 0\}$, $L(M) \cap \{b\}^* = \phi$, ならびに, $L(M) \cap \{a\}^* = \{a\}$ であることが確認出来る.

1.4.5 補集合を受理する有限オートマトン

```
m_complement::Automaton -> Automaton
m_complement (k, s, d, s0, f) = (k, s, d, s0, fc)
      where fc = [x | x<-k, not (x `elem` f) ]
```

例 6 例 *5* のオートマトン $m1$ に対して, $L(m1)$ の補集合 $\overline{L(m1)}$ を受理するオートマトンを考える. $\overline{L(m1)}$, $\overline{L(m1)} \cap \{b\}^*$, ならびに, $\overline{L(m1)} \cap \{a\}^*$ を計算してみる.

```
*Main> take 10 $ language $ m_complement m1
["","b","aa","ab","ba","bb","aaa","aab","abb","baa"]
*Main> accepts (m_complement m1) $ take 5 (sstar ['b'])
["","b","bb","bbb","bbbb"]
*Main> accepts (m_complement m1) $ take 5 (sstar ['a'])
["","aa","aaa","aaaa"]
```

例 *5* の結果と比べて, 確かに補集合が受理されていることが確認出来る.

1.4.6 整数の対を整数で表わす

```
natpairs::[(Int,Int)]
natpairs = [(x,z-x) | z <- [0..], x<- [0..z]]
pxy::(Int,Int)->Int
pxy (i,j) = (div ((i+j) * (i+j+1)) 2) + i
px::Int->Int
px z = fst (natpairs!!z)
py::Int->Int
py z = snd (natpairs!!z)
```

変数 natpairs は自然数の対全体の集合を表わす. 関数 pxy (i,j) は対 (i, j) が natpairs の何番目に現れるかを計算している. 逆に natpairs の z 番目の x 座標を返すのが px, 同じく y 座標を返すのが py である. 関数 pxy の逆関数が関数 px と関数 py の対である.

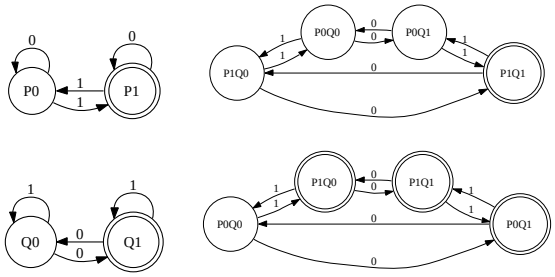
例 7
```
*Main> take 10 natpairs
[(0,0),(0,1),(1,0),(0,2),(1,1),(2,0),(0,3),(1,2),(2,1),(3,0)]
*Main> take 10 [pxy (i,j) | (i,j)<-natpairs]
[0,1,2,3,4,5,6,7,8,9]
*Main> pxy (4,2)
25
*Main> px $ pxy (4,2)
4
*Main> py $ pxy (4,2)
2
```

1.5 積集合と和集合を受理する有限オートマトン

```
m_join::Automaton -> Automaton -> Automaton
m_join (kp, sp, dp, sp0, fp) (kq, sq, dq, sq0, fq) = (kpq, spq, dpq, spq0, fpq)
      where kpq = [pxy (i,j) | i<-kp, j<-kq]
            spq = sp
            dpq z c = pxy ((dp (px z) c),(dq (py z) c))
            spq0 = pxy (sp0,sq0)
            fpq = [pxy (i,j) | i<-fp, j<-kq] ++ [pxy (i,j) | i<-kp, j<-fq]

m_meet::Automaton -> Automaton -> Automaton
m_meet (kp, sp, dp, sp0, fp) (kq, sq, dq, sq0, fq) = (kpq, spq, dpq, spq0, fpq)
      where kpq = [pxy (i,j) | i<-kp, j<-kq]
            spq = sp
            dpq z c = pxy ((dp (px z) c),(dq (py z) c))
            spq0 = pxy (sp0,sq0)
            fpq = [pxy (i,j) | i<-fp, j<-fq]
```

例 8 練習問題 *1* にある 2 つのオートマトン, $m1$, $m2$ を考え, 受理言語 $L(m1)$, $L(m2)$ の和集合や積集合を受理するオートマトンを作成する.




```
mp::Automaton
mp = ([0,1], ['0','1'], dp, 0, [1])
    where dp 0 '0' = 0
          dp 0 '1' = 1
          dp 1 '0' = 1
          dp 1 '1' = 0

mq::Automaton
mq = ([0,1], ['0','1'], dq, 0, [1])
    where dq 0 '0' = 1
```

```
dq 0 '1' = 0
dq 1 '0' = 0
dq 1 '1' = 1
```

```
*Main> take 10 $ language mp
["1","01","10","001","010","100","111","0001","0010","0100"]
*Main> take 10 $ language mq
["0","01","10","000","011","101","110","0001","0010","0100"]
*Main> take 5 $ language $ m_meet mp mq
["01","10","0001","0010","0100"]
*Main> take 10 $ language $ m_join mp mq
["0","1","01","10","000","001","010","011","100","101"]
```

上の例から

$$
\begin{aligned}
L(m1) &= \{w|w \text{ 中の } 1 \text{ の個数が奇数個 }\} \\
L(m2) &= \{w|w \text{ 中の } 0 \text{ の個数が奇数個 }\} \\
L(m1) \cap L(m2) &= \{w|w \text{ 中の } 0 \text{ と } 1 \text{ の個数がともに奇数個 }\} \\
L(m1) \cup L(m2) &= \{w|w \text{ 中の } 0, \text{または}, 1 \text{ の個数が奇数個 }\} \\
&= \{0,1\}^* - \{w|w \text{ 中の } 0 \text{ と } 1 \text{ の個数がともに偶数個 }\}
\end{aligned}
$$

であることが確認出来る.

1.6 レポート課題

Haskell での関数定義 `m_complement`, `m_join`, `m_meet` を参考にして, 命題 1, 命題 2 の証明を行いなさい. 式変形だけでなく, 証明すべきことの道筋を丁寧に記述すること.

参考文献

[1] 有川節夫, 宮野悟: オートマトンと計算可能性, 培風館 (1986).

[2] The Grasgow Haskell Compiler, `http://www.haskell.org/ghc/`

[3] Graphviz –Graph Visualization Software, `http://www.graphviz.org/`