第1章　はじめに

このテキストは、佐賀大学理工学部知能情報システム学科の必修科目「形式言語とオートマトン」のための講義資料です。本学科のカリキュラムにマッチするように書かれています。

この講義で学ぶことはコンピュータやプログラミングの理論に関する数学的な事柄です。そのため、この講義の内容が直ちにコンピュータの知識やプログラミング技術の向上に役立つことはありません。しかし、プログラミングの表層からは見えにくい深層（基礎）の数学的な構造を知ることはコンピュータの限界、プログラムの性質、プログラミングの難しさを理解する上で重要です。そのような理解を通して再度、現実の問題（やプログラムの要求仕様）を見直すならば、その問題の数学的な本質が自ずと見えてくることとなり、ひいてはその問題を解決するプログラムの効果的な開発につながってきます。

この講義では、主に以下の事項を学びます。なお、ここで言う数学は微分積分学や線形代数学とは異なり、グラフや記号論理に近い分野の数学です。

有限オートマトン　この講義の中心テーマです。単に理論的な興味だけではなく、実用上はたとえばロボットの動作アルゴリズムの表現、ワープロの文字列検索ツールなど様々な場面で応用されており、きわめて重要な概念です。

文脈自由文法　少し複雑な文字列処理を行うときにはこの概念が有効です。コンパイラを開発したり、通信プロトコルの記述などに応用されています。

チューリング・マシン [1]　現在私たちが使用しているコンピュータの基となった概念であり、コンピュータの限界を考える上で重要です。

授業中に紙面で配布する講義テキストでは、テキスト中に枠囲みの空白 [　　] があります。この空白に入る語句は、授業中に説明します。また、山下@講義担当者の授業用ホームページ：

`http://www.fu.is.saga-u.ac.jp/~yaman/yamane.html`

の pdf ファイルにはその語句が [語句] のように埋めてありますから、授業中に聞き逃した場合にはダウンロードしてチェックしてください。授業を欠席したときにテキストの配布があった場合にもこの pdf ファイルをダウンロードしてください。

[1] テューリング・マシンとも表記します。「マシン」は書き言葉であって、発音的には「マシーン」がより近い表記です。原語は Turing machine です。

第2章　復習

この章では、後に必要となる基礎的な数学の知識を復習します。

2.1　集合

「もの」の集まりを **集合** (set) と呼びます。集合に含まれる「もの」のことを、**要素、元** (element, member) などと呼びます。a を集合 A の要素とするとき、これを $a \in A$ (または $A \ni a$) と表します。集合に含まれる要素の数が有限であるとき、特に **有限集合** (finite set) と呼び、無限であるとき **無限集合** (infinite set) と呼びます。集合 A について、$|A|$ は A に含まれる要素の数を表します。A の濃度 (cardinality) とも言います。A が有限集合であるときには $|A|$ は有限値です [1]。

集合を、要素を列挙して表すときには、全要素を括弧 $\{\}$ で囲んで表します。たとえば「絶対値が 4 以下の自然数の集合」と言えば、$\{1,\ 2,\ 3,\ 4\}$ となります。この表記法は一般には無限集合には適用できませんが、このテキストでは集合の要素を常識的な一定の順序で列挙できるときにも用います。たとえば「偶数の集合」は $\{2,\ 4,\ 6, \cdots\}$ のように表してもよいこととします。

集合を、各要素 x が満たす性質 $P(x)$ を示して表すときには、以下のように表現します。

$$\{x \mid P(x)\}$$

さらに性質 $P(x)$ が「$x \in A$ かつ $Q(x)$」であるときには、$\{x \mid x \in A \text{ かつ } Q(x)\}$ と表してもよいのですが、しばしばこれを

$$\{x \in A \mid Q(x)\}$$

と表します。たとえば $\mathbf{N}$ を全ての自然数の集合とするとき、偶数の集合は

$$\{a \in \mathbf{N} \mid \text{ある自然数 } b \text{ について } a = 2b \text{ が成り立つ}\}$$

と表すことがあります。

集合 A の任意の要素が集合 B の要素であるとき、$A \subseteq B$、$A \subset B$ (または $B \supseteq A$、$B \supset A$) と表し、「A は B に含まれる」、「A は B の部分集合 (subset) である」などと言います。$A \subseteq B$ か

[1] A が無限集合であるときには $|A|$ を有限値で表すことができません。この辺りの数学はオートマトン理論と全く関係ないという訳ではないので、興味のある人は「カントールの集合論」などを調べてみてください。

つ $A \neq B$ であるとき、特に $A \subsetneq B$（あるいは $B \supsetneq A$）と表し、「A は B の真部分集合（proper subset）である」などと言います。

注意　包含関係 $A \subset B$ の意味は、高校数学で習う事柄と異なるようですから注意してください。関係 $A \subset B$ は $A \subseteq B$ と同じ意味です。$A \subsetneq B$ ではありません。

集合の演算

A と B を任意の集合とするとき、この集合に対する演算として以下のものがよく使われます。

和　$A \cup B = \{x \mid x \in A \text{ あるいは } x \in B\}$.

共通部分　$A \cap B = \{x \mid x \in A \text{ かつ } x \in B\}$.

差　$A - B = \{x \mid x \in A \text{ かつ } x \notin B\}$.

直積　$A \times B = \{(x, y) \mid x \in A \text{ かつ } y \in B\}$.

ベキ集合　$2^A = \{B \mid B \subseteq A\}$.

このうち、和、共通部分、差については高校でも学んでいる事柄ですから説明を省略します。

直積、ベキ集合には馴染みのない人が多いでしょう。この二つの演算は、単純な構造の集合から複雑な構造の集合を構築するための演算と見なすこともできます[2]。たとえば A、B を自然数の集合とするとき、和 $A \cup B$、共通部分 $A \cap B$、差 $A - B$ の演算で得られる集合はやはり自然数の集合です。しかし直積 $A \times B$ は自然数の集合ではなく、自然数の二項組（ペア）の集合です。ベキ集合 2^A は、自然数の集合の集合です。

簡単な例として、$A = \{1, 3, 5\}$、$B = \{2, 4\}$ と仮定します。このとき直積は以下の通りです。

$$A \times B = \{(1, 2), (1, 4), (3, 2), (3, 4), (5, 2), (5, 4)\}$$

すなわち、A からひとつの要素 a を取り出し、B からもひとつの要素 b を取り出し、そのペア (a, b) を作ります。そのようなペア全てを含む集合が $A \times B$ です。A の要素の数を m、B の要素の数を n とするとき、$A \times B$ の要素の数は $m \times n$ です。つまり

$$|A \times B| = |A| \times |B|$$

[2] プログラミング言語的な発想で述べれば、簡単なデータ構造を組み合わせて複雑なデータ構造を作り出すことと似ています。

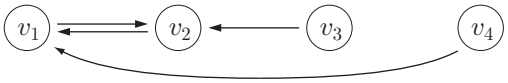


図 2.1: グラフの例

という関係が成り立ちます。直積の演算子記号に乗算記号 $\times$ を用いる理由は上の式に由来すると思われます。上の例では $m = 3$、$n = 2$ であり、$A \times B$ の要素の数は確かに $m \times n = 3 \times 2 = 6$ です。

$A = \{1, 3, 5\}$ の場合のベキ集合は以下の通りです。

$$2^A = \{\emptyset, \{1\}, \{3\}, \{5\}, \{1, 3\}, \{1, 5\}, \{3, 5\}, \{1, 3, 5\}\}$$

すなわち、A の部分集合全てを要素とする集合が 2^A です。空集合は任意の集合の部分集合ですから 2^A に含まれることに注意してください。A それ自身も A の部分集合ですから 2^A に含まれます。A の要素の数を m とするとき、2^A の要素の数は 2^m です。つまり

$$|2^A| = 2^{|A|}$$

という関係が成り立ちます。ベキ集合を表すために 2 のベキ乗を用いる理由は上の式に由来すると思われます。上の例では 2^A の要素の数は $|2^A| = 2^{|A|} = 2^3 = 8$ です。その他の例として、$B = \emptyset$ ならば、$2^B = \{\emptyset\}$ です [3]。$C = \{1\}$（すなわち C は要素をひとつだけ持つ集合）ならば、$2^C = \{\emptyset, \{1\}\}$ です。

2.2　グラフ

グラフ（graph）G は、有限個の **頂点**（vertex, node）の集合 V と、有限個の **枝**（edge, arc）の集合 E で定義されます。枝 e は二つの頂点 $v, v' (\in)$ を用いて $v \to v'$ と表します。v を枝 e の **始点**（start vertex）、v' を e の **終点**（end vertex）と呼びます。また v を v' の先行頂点（predecessor）、v' を v の後続頂点（successor）とも呼びます。たとえば以下はグラフの例です。

$$\begin{aligned} G_1 &= (V_1, E_1) \\ V_1 &= \{v_1, v_2, v_3, v_4\} \\ E_1 &= \{v_1 \to v_2,\ v_2 \to v_1,\ v_3 \to v_2,\ v_4 \to v_1\} \end{aligned}$$

このグラフを図示すると、図 2.1 の通りです。グラフの表現は図示による表現の方が直感的で一般的かもしれません。このテキストでもグラフはほとんどの場合に図示しますが、形式的な扱いを行う場合には頂点と枝の表現を用います。

[3] ついでに B のベキ集合のベキ集合を考えてみると、$2^{2^B} = \{\emptyset, \{\emptyset\}\}$ となります。B のベキ集合のベキ集合のベキ集合は $2^{2^{2^B}} = \{\emptyset, \{\emptyset\}, \{\{\emptyset\}\}, \{\emptyset, \{\emptyset\}\}\}$ となります。

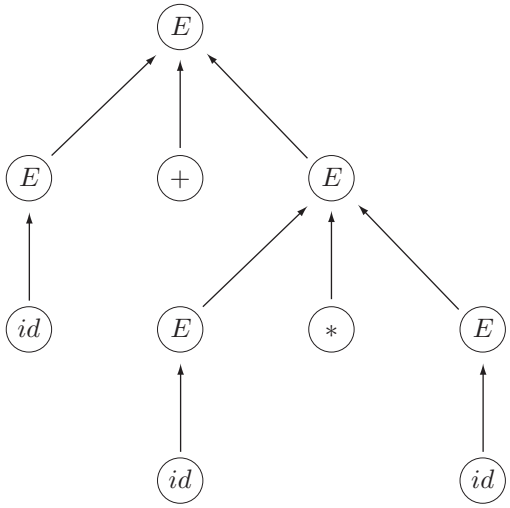


図 2.2: 木構造の例

グラフ $G = (V, E)$ について、$v \to v' \in E$ ならば $v' \to v \in E$ であるとき、グラフは 無向グラフ（undirected graph）であると言います。さもなくば 有向グラフ（directed graph）と言います。このテキストで扱うほとんどのグラフは有向グラフです。

グラフ $G = (V, E)$ が枝 $v_1 \to v_2$, $v_2 \to v_3$, ..., $v_{k-1} \to v_k$ を持つとき、G には v_1 から v_k への 路、道（path）があると言います。特に路の始点と終点が一致する（すわなち $v_1 = v_k$ となる）とき、その道（路）を 閉路、循環路（closed path, cyclic path）と呼びます。

ある集合 L が存在し、グラフの頂点 $v \in V$ から $l \in L$ への写像 f が存在するとき、$f(v)$ を v の ラベル（label）と呼びます。同様に、グラフの枝 $e \in E$ から $l \in L$ への写像 g が存在するとき、$g(e)$ を v の ラベル（label）と呼びます。頂点や枝にラベルの付いたグラフをラベル付きグラフ（labelled[4] graph）と呼びます。このテキストでは枝にラベルの付いたグラフが多数現れます。

木

コンピュータサイエンスでは、グラフの特殊な形状として、いわゆる木構造[5]をよく用います。正確には以下の性質を満たすものを **木**（tree）と呼びます。

1. 後続頂点を持たない頂点が唯一存在します。その頂点を特に **根**（root）と呼びます。グラフ内の任意の頂点から根への路が必ずひとつ存在します。
2. 根以外の頂点は後続頂点をひとつ持ちます。
3. 各頂点の先行頂点には、頂点間の接続関係とは別に順序が決められています。

木の各頂点の後続頂点のことをしばしば **親**（parent）と呼び、先行頂点のことを **子供**（child, children）と呼びます。親の親、親の親の親などを先祖（ancestorr）と呼びます。子供の子供、子供の子供の子供などを子孫（descendant）と呼びます。先行頂点を持たない頂点を **葉**（leaf）とも呼びます。根でも葉でもない頂点を **内部頂点**（internal vertex）と呼びます。

たとえば図 2.2 は、12 章で考察する木構造の例です。頂点の中の記号 $E, +, *, id$ は頂点のラベルです。この図では、図の最上部の頂点が根です[6]。下部の子供を持たない頂点が葉です。上の図において、根は三つの子供を持ち、そのラベルを左から順に列挙すると $E + E$ となります。$+EE$ や $EE+$ ではありません。上の条件 3. の「後続頂点には... 順序が決められています」とは、このことを意味しています。同様に、根の最右の子供の三つの子供のラベルを左から列挙すると、$E * E$ です。このように、木は単なるグラフではなく、子供の左右の位置関係 — これを兄弟関係とも呼びます — も木が有する重要な情報です。

2.3　関数の型

集合 D_1、D_2 について、**定義域**（domain）を D_1 とし、**値域**（codomain, range）[7] を D_2 とする関数（function）の集合を $D_1 \to D_2$ と表します。また $D_1 \to D_2$ を関数の型（type）とも呼びます。関数 f が D_1 を定義域とし、D_2 を値域とするとき、しばしば

$$f : D_1 \to D_2$$

[4] あるいは labeled。

[5] 単に「木」と呼ぶよりも「木構造」(tree structure) と、「木」と「構造」をセットにして呼ぶことが多いように思われます。

[6] 常識的に考えれば、根が上部にあり、葉が下部にある木は逆さまです。しかし、コンピュータサイエンスの「木」に慣れてしまうと図 2.2 が木であることに何の疑問も感じなくなってしまいます。そのせいか、コンピュータ関係者の中には、葉が上で根が下に描いてある図をとっさに「逆木」と呼んでしまう粗忽者がたまに見かけられます。

[7] range と codomain はほとんど同じですが、厳密には異なります。その違いについては、この講義は数学の講義ではないので追求しません。気になる人は「codomain range 値域」などのキーワードでインターネット検索してください。

と表し、「f は $D_1 \to D_2$ という 型 (type) を持つ」などとも言います。これを C 言語で考えれば、以下のようなプロトタイプ宣言と類似しています。

```
D2 f(D1);
```

定義域 D_1 が集合 D_3、D_4 の直積であるような場合も当然考えられ、そのような関数の型は $D_3 \times D_4 \to D_2$ と表されます。これも、関数 f について C 言語のプロトタイプ宣言風に表せば、以下の通りです。

```
D2 f(D3,D4);
```

値域が D_2 が集合 D_5、D_6 の直積であるような場合も当然考えられます。そのような関数の型は $D_3 \times D_4 \to D_5 \times D_6$ と表されます。これに対応する C 言語のプロトタイプ宣言はありません。なぜならば、C 言語では複数の値を一度には `return` できないからです。そのような関数を C 言語で実装するときには構造体を用いるのが一般的です。

値域 D_2 が集合 D_5 のベキ集合であるような場合も考えられ、そのような関数の型は $D_1 \to 2^{D_5}$ と表されます。これに類似する C 言語のプロトタイプ宣言もありません。そのような関数を C 言語で実装する手法は各人各様ですが、たとえば線形リストや bit 列を用いて 2^{D_5} の要素を表すことができます。

第3章　記号列と言語

この章では、この後の章全てに関係する基本的な概念を定義します。

3.1　記号列

日本語のひらがな/カタカナ/漢字や英語のアルファベット 26 文字、数学における数記号（0〜9）のように、情報を表す不可分の基本単位としてしばしば **記号**（symbol）が用いられます。そして様々な情報は、ちょうど日本文がひらがな/カタカナ/漢字の並びで表されるように、あるいは英文がアルファベットの並びで表されるように、自然数が数記号の並びで表されるように、記号の並びとして表現されます。この記号の並びを一般に **記号列**（symbol string）あるいは単に **列**（string）と呼びます。記号列はこの講義で論じる内容の最も重要な操作対象です。

記号列を論じるときには、その記号列に含まれる記号の集合を、ある有限集合に固定して論じるのが一般的です。その記号の集合のことを **アルファベット**（alphabet）[1] と呼び、しばしばギリシャ文字 Σ を用いて表します。たとえばアルファベットを $\Sigma = \{a, b, c\}$ とするとき、$ababc$、$aaabbbccc$ などの、Σ に含まれる記号で構成される列を Σ の上の 記号列と呼びます。

記号列 u について、$|u|$ は記号列の **長さ**（length）を表します。たとえば $|ababc| = 5$、$|aaabbbccc| = 9$ です。記号列の中には長さが 0 であるような記号列が存在します。そのような記号列を特に **空列**（empty string）と呼び、ギリシャ文字 ε で表します。もちろん $|\varepsilon| = 0$ が成り立つと考えます。

注意　空列を空集合と混同しないように注意してください。空列はれっきとした列です。空集合は列とは直接関係しません。

記号列 u の後に記号列 v を続けて書いた記号列 uv を u と v の **連接**（concatenation）と呼びます。$|u| = m$、$|v| = n$ ならば、$|uv| = m + n$ が成り立ちます。たとえば aaa の後に bbb を連接した記号列は $aaabbb$ です。

[1] アルファベットと言えば英文字 26 文字を連想しがちですが、それはアルファベットの一例に過ぎません。

空列 ε は連接演算について単位元として振る舞います。すなわち、任意の記号列 u について

$$u\varepsilon = \varepsilon u = u$$

が成り立ちます。（これは加算の単位元 0、乗算の単位元 1 が

$$x + 0 = 0 + x = x, \quad x \times 1 = 1 \times x = x$$

を満たすことと同じです。）単位元はその演算の数学的な性質を論じる際に様々な重要な役割を演じますが、空列もこの講義の様々な場面で重要な役割を演じることとなります。

同じ記号 a が n 回続けて現れるような列 $\underbrace{aaa\cdots aaa}_{n}$ は a^n と表すと約束します。ただし a^0 は空列とします。同様に、同じ記号列 u が n 回続けて現れるような列 $\underbrace{uuu\cdots uuu}_{n}$ は u^n と表します。これを再帰的に定義すれば、以下の通りです。

$$\begin{aligned} u^0 &= \varepsilon、 \\ u^n &= uu^{n-1} \end{aligned}$$

必要ならば括弧を用いて $(u)^n$ と表してもよいとします [2]。

記号列 u、v、w について、$u = vw$ が成り立つとき、v は u の **接頭語**（prefix）であると言います。また w は u の **接尾語**（postfix）であると言います。たとえば $ababc$ の接頭語は、ε、a、ab、aba、$abab$、$ababc$ です。空列や記号列それ自身も接頭語であることに注意してください。接頭語の中から記号列それ自身を除いたものを **真の**（proper）接頭語と呼んで、記号列それ自身と区別する場合があります。真の接尾語も同様です。

注意　このテキストでは、誤解、曲解を避けるために、記号列を二重引用符 “ ” で囲み、たとえば “$ababc$”、“$aaabbbccc$” などと記述する場合があります。しかし、二重引用符の有無は記号列の内容に影響しません。$ababc$=“$ababc$” と考えてください。

3.2　言語

アルファベット Σ の上の記号列を要素とする集合を Σ の上の **形式言語**（formal language）または単に **言語**（language）と呼びます。たとえば $\Sigma = \{@\}$ とするとき、奇数個の@からなる列の集合:

$$L_1 = \{@^{2n+1} \mid n \geq 0\} = \{@, @@@, @@@@@, @@@@@@@, \cdots\}$$

[2] このとき、括弧 () は記号列を構成する記号とは見なしません。

は言語です。また素数個の@からなる列の集合:

$$L_2 = \{@^n \mid n \text{ は素数 }\} = \{@@, @@@, @@@@@, @@@@@@@, \cdots\}$$

も言語です。$\Sigma = \{0, 1, 2, \cdots, 8, 9\}$ とするとき、以下は 10 進数による素数の集合であって、やはり言語です。

$$L_3 = \{n \mid n \text{ は 10 進数の素数 }\} = \{2, 3, 5, 7, 11, 13, \cdots\}$$

空集合 $\emptyset$ も言語です。空列のみを要素とする集合 $\{\varepsilon\}$ も言語です。$\emptyset$ と $\{\varepsilon\}$ を混同しないように注意してください。

　このように、形式言語とは特定の数学的条件を満たす記号列の集合のことです。そうすると、与えられた記号列 u が言語 L に属するか否かという判定問題 — これを 所属問題 (membership problem) とも呼びます — が興味深い問題として想起されてきます。実際、このテキストの主な目的は様々な言語の所属問題を解くことです。

　たとえば、記号列 @@ が上に示した言語 L_1 に所属するか否か（@@ の長さが奇数であるか否か）を知るにはどのようなアルゴリズムを用いればよいでしょうか。またそれは、@@ が言語 L_2 に所属するか否か（@@ の長さが素数であるか否か）という問題と比較して難しい問題でしょうか、あるいは易しい問題でしょうか。このテキストでは、このような所属問題をいくつかのカテゴリーに分類し、各カテゴリーの性質を明らかにしていきます。

　所属問題は、一見、コンピュータとは無関係に思えるかもしれません。しかし、所属の是非を判定するプログラムをコンピュータに実装することを考えてみましょう。そのプログラムの複雑さは所属問題の性質によって分類できるのです。所属問題を知ることはコンピュータプログラムの複雑さを知ることに関係しており、プログラムの理論的な構造な性質について理解を深めてくれます。

　ところで、これ以降の便利のために、Σ の上の全ての記号列の集合を Σ^* と表すと約束します。そうすると、Σ の上の任意の言語は Σ^* の部分集合ということになります。もちろん、Σ^* 自身も Σ^* の部分集合であり、Σ の上の言語です。アルファベット Σ の右肩に星印 $*$ を付けて全ての記号列の集合を表す理由は、4 章で明らかになります。

注意　「言語」というと、日本語、英語、中国語のような自然言語を想起する人がいるかもしれません。また、この授業では自然言語のコンピュータ処理（日英自動翻訳など）について学ぶものと期待している人もいるかもしれません。しかし残念ながら、この授業で扱う「言語」はとても単純です。

　上に定義したように、記号列の集合のことをこの授業では言語と呼んでいます。この授業では、記号列の集合に関する所属問題を論じることが主なテーマであって、日英自動翻訳などについては全く触れません。というのも、自然言語のコンピュータ処理を行うには単に所属問題を解くだけでは足りず、構文論/意味論などの難解な問題に取り組まねばならないからです。

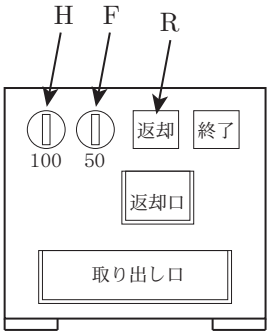


図 3.1: 缶ジュースの自動販売機

3.3　例題：缶ジュースの自動販売機

次章以降、形式言語に関して議論を具体的に進めていきますが、正規表現と有限状態オートマトンに関する章で一貫して用いる例題をここであらかじめ紹介しておきます。

缶ジュースの自動販売機　ひなびた温泉旅館に缶ジュースの自動販売機が置かれているとします。この販売機は旧式のため、以下に述べる単純な仕組みで動作します。

1. 缶ジュースは 1 本 100 円です。
2. 販売機には、
   (a) 100 硬貨投入口、
   (b) 50 円硬貨投入口、
   (c) 返却ボタン、
   (d) 硬貨返却口、
   (e) 操作終了ボタン

   が付いています（図 3.1 参照)。
3. 100 円硬貨、50 円硬貨を 100 円分以上投入した後に、操作終了ボタンを押すと缶ジュース 1 本が出てきて、その 1 回の販売処理が終了します。ただしお釣りは戻りません。お金の投入を間違えると利用者が一方的に損をしますが、文句は言えないことになっています。
4. 投入した硬貨総額が 100 円に満たないまま操作終了ボタンを押すと缶ジュースは出てこず、その回の販売処理がそのまま終了します。投入した硬貨は返却されません。この場合も利用者は損をしますが、やはり文句は言えないことになっています。

5. 操作終了ボタンを押す前に返却ボタンを押すと、それまでに投入した硬貨全てが返却口に戻り、それまでの投入金額が 0 円にリセットされます。幸いなことに、この場合は利用者は損をしません。

さて、利用者が行う自動販売機の操作を以下の三つの記号で表します。

H ... 100 円硬貨を投入する。

F ... 50 円硬貨を投入する。

R ... 返却ボタンを押す。

利用者が缶ジュースを 1 本得るためには、上の H、F、R の操作を何度か繰り返した後に操作終了ボタンを押すと想定されています。

さて、ここで問題です。

問題　操作終了ボタンを押したときに缶ジュースが自動販売機から出てくるような操作の列を全て示しなさい。ただし、利用者の損を関知する必要はありません。つまり、1000 円分の硬貨を投入してお釣り 900 円が戻らなくとも気せず、そのような場合も缶ジュースが出てくる操作列と見なします。

たとえば HFRHHF はこの自動販売機の操作列の例です。この列は、

H 100 円硬貨を投入し、
F 50 円硬貨を投入し、
R 返却ボタンを押し、
H 100 円硬貨を投入し、
H 100 円硬貨を投入し、
F 50 円硬貨を投入する

操作の列を表しています。このとき、途中で硬貨が返却されますが、最終的には 250 円を投入していますから、ジュースは出てきます。

次に、操作列 HHFFRF ではジュースは出てきません。何故ならば、この列では、

H 100 円硬貨を投入し、
H 100 円硬貨を投入し、
F 50 円硬貨を投入し、
F 50 円硬貨を投入し、
R 返却ボタンを押し、
F 50 円硬貨を投入する

操作を行います。この場合、最終的には 50 円しか投入していませんから、ジュースは出てきません。

上の問題に答えるには、前者の HFRHHF のような、ジュースが出てくる操作列の性質を示せばよいのです。

次のように考えてみましょう。ジュースを得るには、操作を開始した後、返却ボタンを最後に押してから操作終了ボタンを押すまでの間に少なくとも 100 円硬貨 1 枚を投入するか、50 円硬貨 2 枚を投入する必要があります。このことから、ジュースを得るための操作列を以下の三つの操作列 u、v、w に分けて考えます。

1. u は H、F、R からなる任意の列、
2. v は H または FF、
3. w は H、F からなる任意の列

ここに「... からなる任意の列」というときには空列も含むとこの講義では約束します。

列 u は 返却操作 R を含むため、ジュースを得るには無効かもしれない操作列を意味します。u は余分な硬貨を投入する操作列かもしれませんが、その余分な硬貨でジュースが出てくるとは確定できないとします。あるいは u は空列かもしれません。

これに対して、v はジュースを確実に得るための操作列です。

w は余分な硬貨を投入する操作列です。w は空列かもしれません。

そしてジュースが出てくる操作列とは、u、v、w がこの順番に連結された列 uvw であれば必要十分です。よって、求める操作列の集合を L_J と表すとき、それは以下の通りです。

$$L_J = \{uvw \mid u \in \{\mathrm{H},\mathrm{F},\mathrm{R}\}^*,\ v \in \{\mathrm{H},\mathrm{FF}\},\ w \in \{\mathrm{H},\mathrm{F}\}^*\}$$

これが問題の答えです。

第4章　正規表現

この授業では様々な記号列を扱います。そこでまず、この章では、記号列あるいは記号列の集合（＝言語）を簡単に表わすことができる **正規表現** と呼ばれる数学的な記法を学びます。

前章の「ジュースが出てくる記号列の集合」を思い出しましょう。それは以下のように表現できました。

$$L_J = \{uvw \mid u \in \{\mathrm{H},\mathrm{F},\mathrm{R}\}^*,\ v \in \{\mathrm{H},\mathrm{FF}\},\ w \in \{\mathrm{H},\mathrm{F}\}^*\}$$

しかし、正規表現を用いるならば、これを

$$(\mathrm{H}|\mathrm{F}|\mathrm{R})^*(\mathrm{H}|\mathrm{FF})(\mathrm{H}|\mathrm{F})^*$$

と表わすことができます。前者に比べ後者の方が簡素な表現になっており、数学的な取り扱いに便利です。この便利さから正規表現は様々な分野に利用されています。最も有名な例として、多くのワープロの文字列検索処理では、検索すべき文字列を正規表現で表わすことができます。また、様々なプログラミング言語のコンパイラではソース・プログラム中の単語[1]を認識する部分の形式的記述には正規表現を用いています。

この章では正規表現を定義し、正規表現を用いて記号列を表わす例をいくつか示します。そして次章以降の講義内容を紹介します。

4.1　準備

正規表現を定義する前に 3 章の内容を拡張定義します。

任意の記号列の集合 L_1 と L_2 について、その **連接**（concatenation）L_1L_2 を以下のように定義します。

$$L_1L_2 = \{xy \mid x \in L_1, y \in L_2\}$$

たとえば、記号列の集合 $\{ab, bb\}$ と $\{aa, ccc\}$ について以下の通りです。

$$\{ab, bb\}\{aa, ccc\} = \{abaa, abccc, bbaa, bbccc\}$$

[1] 正確には「字句」と呼びます。

特殊な場合として、空集合 $\emptyset$ や空列のみの集合 $\{\varepsilon\}$ との連接では、任意の記号列の集合 L について以下が成り立つことを確認しましょう。

$$\emptyset L = L\emptyset = \emptyset, \quad \{\varepsilon\}L = L\{\varepsilon\} = L$$

記号列の集合 L について、L^n は L を n 個連接した集合 $\underbrace{L \cdots L}_{n}$ を表します。これを再帰的に定義すれば、以下の通りです [2]。

$$\begin{aligned} L^0 &= \{\varepsilon\}, \\ L^n &= LL^{n-1} \quad (n \geq 1) \end{aligned}$$

記号列の集合 L について、L^* を L の **クリーネ閉包** (Kleenean closure) と言い、以下のように定義します。

$$L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots = \{\varepsilon\} \cup L \cup LL \cup LLL \cup \ldots$$

L^* は常に空列を含むことに注意してください。ほとんどの場合に L^* は無限集合です。

考察　L^* が有限集合となるような L を示しなさい。

さて、3 章の最後に「Σ 上の全ての記号列の集合を Σ^* と表す」と約束したことを思い出しましょう。3 章では星印 * の意味を述べませんでしたが、実は上のクリーネ閉包がその定義だったのです。何故ならば、今、Σ 上の長さ n の記号列を $a_1a_2...a_n$ とします。ここに各 a_i は Σ の要素です。そうすると、$a_1a_2...a_n$ は Σ^n に含まれます。Σ^n は Σ^* の部分集合ですから、$a_1a_2...a_n$ は Σ^* に含まれます。つまり、Σ 上の任意の記号列は Σ^* に含まれるのです。

4.2　定義

以下の 8 個の規則によって再帰的に構成される表現形式のことをアルファベット Σ の上の **正規表現** [3] (regular expression) と呼びます。また、正規表現 r が表現する記号列の集合 — これを r の **言語** (language) と呼びます — を $L(r)$ と表すとき、$L(r)$ は以下のように再帰的に定義されます。

1. アルファベット Σ に属する記号 a は正規表現であり、$L(a) = \{a\}$ と定義します。

[2] 3.1 節に記号列に関する類似した定義があったことを思い出しましょう。
[3] 正規表現を正則表現とも呼びます。いずれも regular expression の訳語です。

2. 記号 ε は正規表現であり、$L(\varepsilon) = \{\varepsilon\}$ と定義します。すなわち、空列を表す記号 ε の言語は空列のみを含む集合です。

3. 記号 $\emptyset$ は正規表現であり、$L(\emptyset) = \emptyset$ と定義します[4]。

4. r を正規表現とするとき、r を括弧で囲んだ (r) も正規表現であり、$L((r)) = L(r)$ と定義します。括弧 () は、演算子の結合順序を変更したり、明示するために用います。

5. r_1、r_2 を正規表現とするとき、それらを連接した r_1r_2 も正規表現であり、$L(r_1r_2) = L(r_1)L(r_2)$ と定義します。ここに、r_1 が ε ならば $L(r_1r_2) = L(\varepsilon)L(r_2) = \{\varepsilon\}L(r_2) = L(r_2)$ が成り立ち、r_2 が ε ならば $L(r_1r_2) = L(r_1)L(\varepsilon) = L(r_1)\{\varepsilon\} = L(r_1)$ が成り立つことに注意しましょう。

6. r_1、r_2 を正規表現とするとき、$r_1|r_2$ も正規表現であり、$L(r_1|r_2) = L(r_1) \cup L(r_2)$ と定義します。なお、$L(r_1|r_2)$ が $L(r_1)$ と $L(r_2)$ の和集合であることから、講義では $r_1|r_2$ を直感的に「r_1 または r_2」と読み下すことがあります。

7. r を正規表現とするとき、r^* も正規表現であり、$L(r^*) = L^*(r)$（すなわち $L(r^*)$ は $L(r)$ のクリーネ閉包）と定義します。なお、クリーネ閉包の意味をかんがみ、講義では r^* を直感的に「r の 0 回以上の繰り返し」と読み下すことがあります。

8. r を正規表現とするとき、r^+ も正規表現であり、$L(r^+) = L(r)L^*(r)$ と定義します。なお、講義では r^+ を直感的に「r の 1 回以上の繰り返し」と読み下すことがあります。

なお、正規表現に現れる演算子の **優先順位** は以下のように仮定します。

$$(\ ) \ > \ * \ > \ + \ > \ \text{連接} \ > \ |$$

たとえば $\Sigma = \{\mathrm{H}, \mathrm{F}, \mathrm{R}\}$ とするとき、この章の冒頭に示した $(\mathrm{H}|\mathrm{F}|\mathrm{R})^*(\mathrm{H}|\mathrm{FF})(\mathrm{H}|\mathrm{F})^*$ は正規表現です。そして、その言語は以下の通りです。

$$L((\mathrm{H}|\mathrm{F}|\mathrm{R})^*(\mathrm{H}|\mathrm{FF})(\mathrm{H}|\mathrm{F})^*) = \{\mathrm{H}, \mathrm{F}, \mathrm{R}\}^*\{\mathrm{H}, \mathrm{FF}\}\{\mathrm{H}, \mathrm{F}\}^*$$

考察　$(\mathrm{H}|\mathrm{F}|\mathrm{R})^*\mathrm{H}|\mathrm{FF}(\mathrm{H}|\mathrm{F})^*$、$\mathrm{H}|\mathrm{F}|\mathrm{R}^*(\mathrm{H}|\mathrm{FF})(\mathrm{H}|\mathrm{F})^*$、$\mathrm{H}|\mathrm{F}|\mathrm{R}^*\mathrm{H}|\mathrm{FFH}|\mathrm{F}^*$ は正規表現です。これら三つの正規表現の構造（演算子と被演算子の入れ子関係）と言語を示しなさい。

[4]通常、“$\emptyset$” は空集合を表わす記号として用いられます。つまり、空集合を表わす記号は空集合を表わすと定義している訳です。

4.3　正規表現の例

この節では、$\Sigma = \{a, b\}$ と仮定し、いくつかの正規表現の例を示します。これらの例は後章でも継続して用います。

例 **1:** $(a|b)^*$

これは Σ 上の 任意の記号列 を表す正規表現です。この正規表現の言語は $L((a|b)^*) = L(a|b)^* = \{a, b\}^* = \Sigma^*$ です。

例 **2:** ε

定義から $L(\varepsilon) = \{\varepsilon\}$ です。後章において重要な例題となりますから、敢えてここで挙げておきます。

例 **3:** $\emptyset$

定義から $L(\emptyset) = \emptyset$ です。つまり、記号 $\emptyset$ は如何なる記号列も表しません。また任意の正規表現 r について、$r\emptyset$、$\emptyset r$ は共に正規表現であり、$L(r\emptyset) = L(\emptyset r) = \emptyset$ が成り立ちます [5]。

なお、言語 $L(\varepsilon)$ と $L(\emptyset)$ は異なる集合であることに注意してください。ε と $\emptyset$ は共に空なものを表わしますが、前者 $L(\varepsilon)$ は空列のみを含む集合、後者 $L(\emptyset)$ は空集合です。全く別ものであり、混同してはいけません。

例 **4:** a

アルファベットに属するひとつの記号 a も正規表現であり $L(a) = \{a\}$ が成り立ちます。この例はあまりにも自明ですが、後章で重要な例題となりますので、ここで挙げておきます。

例 **5:** aba

アルファベットに属する記号のみから作られた記号列 aba も正規表現であり、$L(aba) = \{a\}\{b\}\{a\} = \{aba\}$ が成り立ちます。

[5] 何故ならば、$L(r\emptyset) = L(r)L(\emptyset) = L(r)\emptyset = \emptyset$。$L(\emptyset r)$ も同様。

例 **6:** $\varepsilon|b|(a|b)(a|b)^+$

実はこの正規表現の言語は $\Sigma^* - \{a\}$ であり、上の例 4 の言語 $L(a)$ の 補集合 です。言語が $\{a\}$ であるような正規表現 a に比べ、かなり複雑な形であることに注意してください。正規表現には集合差演算 “–”（2.1 節参照）に相当する演算が定義されていないため、補集合の表現は複雑になるのです。しかし、複雑にはなるものの、任意の正規表現 r について $L(r') = \Sigma^* - L(r)$ が成り立つような正規表現 r' は必ず存在します。

例 **7:** a^*

特定のパターンが 繰り返す ような記号列を表すことは正規表現の最も得意とするところです。この例では $L(a^*) = \{a^i \mid i \geq 0\}$ です。

言語が $\{a^i \mid i \geq 1\}$ であるような正規表現は、a^+ です。あるいは aa^* または a^*a です。言語が $\{a^i \mid i \geq 2\}$ であるような正規表現は、aa^+ または a^+a、あるいは aaa^*、aa^*a、a^*aa と表わすことができます。

例 **8:** $a^*|b^*$

言語の集合和は演算子 ‘|’ を用いて簡単に表現できます。この例は、a^* の言語と b^* の言語の和集合を表わします。何故ならば、$L(a^*|b^*) = L(a^*) \cup L(b^*) = \{a^i \mid i \geq 0\} \cup \{b^i \mid i \geq 0\}$ が成り立つからです。

例 **9:** $(ab)^*$

この正規表現は a と b が交互に現れる列 $(ab)^n = abab...ab$ を表します。正確に言えば、$L((ab)^*) = \{(ab)^n \mid n \geq 0\} = \{\varepsilon, ab, abab, ababab, ...\}$ となります。

例 **10:** $(b^*ab^*a)^*b^*$

この正規表現は、記号 a と b の出現順序は問わず、記号列の中に現れる a の数が必ず偶数（0 は偶数とします）であるような列、たとえば ε、bbb、aab、$abba$、... を表わします。何故ならば、まず正規表現 $(aa)^*$ は a が偶数個現れる a のみからなる列を表します。記号列中に b は任意個任意の場所に現れて構わないので、$(aa)^*$ の中に b^*（0 個以上の b を表します）を配置すると、表題の正規表現になります。

例 **11:** $(a|b)^*aa$

この正規表現は末尾が記号列 aa で終わるような記号列を表します。何故ならば、$(a|b)^*$ は任意の記号列を表わし（例 1 参照）、その後ろに aa が連接されているからです。正確に言えば、$L((a|b)^*aa) = \{uaa \mid u \in \{a,b\}^*\}$ となります。

考察　列の 途中 に aa を含むような記号列の集合を表す正規表現を示しなさい。

例 **12:** 正規表現で表すことができない言語

実は正規表現の表現力は非常に限定的で、正規表現では決して表わすことができない言語が無限に存在します。

たとえば言語 $\{a^nb^n \mid n \geq 0\}$ を表すような正規表現は存在しません。何故ならば、この言語に含まれる記号列では a の数と b の数が同じでなければなりません。数が同じか否か判断するには、a が現れた個数を一旦どこかに記憶しておき、その後に b の個数と比較せねばなりません。しかし、正規表現では数の記憶に相当する機能が十分ではないのです。

a の数が、ある定数 N 以下に限定されている場合には正規表現が存在します。言語 $\{a^nb^n \mid N \geq n \geq 0\}$ の正規表現は $\varepsilon|ab|aabb|aaabbb|...|a^Nb^N$ です [6]。逆に言えば、a の数が定数 N に制限されない場合が問題になる訳です。そのような場合には理論上、無限の大きさの記憶領域が必要となりますが、正規表現はそのような記憶の機能を持っていません。

なお、敢えて言えば、正規表現 a^*b^* は言語 $\{a^nb^n \mid n \geq 0\}$ を表わすために比較的近い表現とも言えますが、しかし、正規表現 a^*b^* の言語は $\{a^mb^n \mid m,n \geq 0\}$ であって、m と n の大きさが 異なる 場合も含みますから、$L(a^*b^*) \supsetneq \{a^nb^n \mid n \geq 0\}$ の関係にあります。

次にもう少し実用的な例題を挙げましょう。

例 **13:** $0|(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$

この正規表現は、任意の正整数を表しています。ただし $\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$ とします。これを直感的に述べれば、「正整数とは、数字 0 であるか、数字 1 から 9 の後に数字 0 から 9 の 0 個以上並んだものである」を表しています。よって、この表現では 00 や 0123 などは正整数とは見なしません。先頭に 0 がくるような正整数は唯一 0 に限られるからです。この正規表現については後章で詳しく考察します。

[6] N が有限ならば正規表現は存在しますが、しかし $N = 100000000$ などという非常に大きな数を想定することは現実的ではありません。コンピュータに実装することを考えるならば、N はせいぜい数十程度でしょう。それを超える場合には、正規表現ではなく、後に述べる文脈自由文法を用いるべきです。

例 **14:**$(0|1)^+.(0|1)^*|(0|1)^*.(0|1)^+$

この正規表現は、2 進数の浮動小数点数を表しています。ただし $\Sigma = \{0, 1, .\}$ とします。‘.’ は小数点を表わす記号です。直感的に述べれば、「2 進数の浮動小数点数とは、数字 0 または 1 が 1 個以上あり、その後に小数点. があり、その後に数字 0 または 1 が 0 個以上あるか、あるいは数字 0 または 1 が 0 個以上あり、その後に小数点. があり、その後に数字 0 または 1 が 1 個以上ある」ことを表しています。よって、この表現では小数点のみ “.” という記号列は浮動小数点数とは見なされません。小数点の前または後ろに少なくとも 1 個の数字が現れる必要があります。

例 **15:**$(+|-|\varepsilon)((0|1)^+.(0|1)^*|(0|1)^*.(0|1)^+)$

この正規表現は、2 進数の符号付き浮動小数点数を表しています。ただし $\Sigma = \{0, 1, ., +, -\}$ とします。直感的に述べれば、「2 進数の符号付き浮動小数点数とは、先頭に $+$ または $-$ が現れるか、あるいは何も現れず、その後に例 14 の浮動小数点数が続く」ことを表しています。“+100.1”、“−.001”、“1.1” などがその例です。この場合、記号列 “+.”、“−.”、“.” は 2 進数の符号付き浮動小数点数ではありません。

さて、正規表現 r_1、r_2 について、もし $L(r_1) = L(r_2)$ ならば、r_1、r_2 は等価である（equivalent）と言います。

たとえば任意の正規表現 r_1、r_2 について、$r_1|r_2$ は $r_2|r_1$ と等価です。何故ならば、$L(r_1|r_2) = L(r_1) \cup L(r_2) = L(r_2) \cup L(r_1) = L(r_2|r_1)$ が成り立つからです。その他にも、正規表現の定義に基づき、様々な等価性が知られています。

考察　$\Sigma = \{a, b\}$ とするとき、以下のそれぞれ二つの正規表現が等価であるか否か答えなさい。

(1) $a|b$ と $b|a$、　(2) $a|\varepsilon$ と $a|\emptyset$、　(3) $a^*|\varepsilon$ と $a^*|\emptyset$、　(4) aa^*b と a^*ab、

(5) $(aa)^*$ と a^*a^*、　(6) $(a|b)^*$ と $(a^*b^*)^*$

4.4　正規表現を応用したソフトウェア・ツール

既に見たように、正規表現は文字列のパターンを表現するときに便利です。ここでは正規表現が実用化されている例を紹介します。

4.4.1　grep による文字列検索

grep（または egprep）は、以下の形式：

```
> grep -E 正規表現 ファイル名
```

で使用する UNIX コマンドで、テキストファイル中から、正規表現で指定された文字列パターンを見つけ出し、その文字列が含まれる行を全て表示します。膨大なデータから特定の情報を検索するときに非常に便利で、UNIX 環境下の作業ではよく使用されています。

　たとえば以下は、ファイルの各行に F、H、R からなる様々な文字列[7] を 20 行を格納しているファイルの内容です。ただし、記号列の先頭と末尾を明確に知るために、各行の先頭と末尾に@を付けることにします[8]。

```
@HFFFRRRHFHHHHFHRHHFHF@
@RRRRHFFHHFHHR@
@RF@
@FH@
@FHRRHFRRFRRRF@
@HFRRF@
@RHF@
@RHHHFHFHHRRFHRFFFHF@
@HHHHFHFRHFFRHFFFFHRHRFFRRRHRFFRHRRRHRRHHFFHFR@
@HHRFHFF@
@HHR@
@HRHFHHRRFHRR@
@RRHRFHHHFFRFFF@
@HFFRHRFR@
@RHRR@
@RFRR@
@RHFH@
@FFRRRFFH@
@HH@
@FH@
```

　上の記号列について缶ジュースが出てくる記号列を grep で見つけることができます。ファイル名を `sample.dat` とするとき、以下のようにコマンド実行します。

```
> grep -E '@(F|H|R)*(FF|H)(F|H)*@' sample.dat
```

これに対する出力は以下の通りとなります。

```
@HFFFRRRHFHHHHFHRHHFHF@
@FH@
@RHF@
@RHHHFHFHHRRFHRFFFHF@
@HHRFHFF@
@RRHRFHHHFFRFFF@
@RHFH@
@FFRRRFFH@
@HH@
@FH@
```

[7] 講義担当者が乱数を用いて作成しました。

[8] grep では行頭は ∧、行末は $ と表すことができますから、@の代わりにそれを用いてもいいのですが、後章との関連からここでは ∧、$ は用いないことにしました。

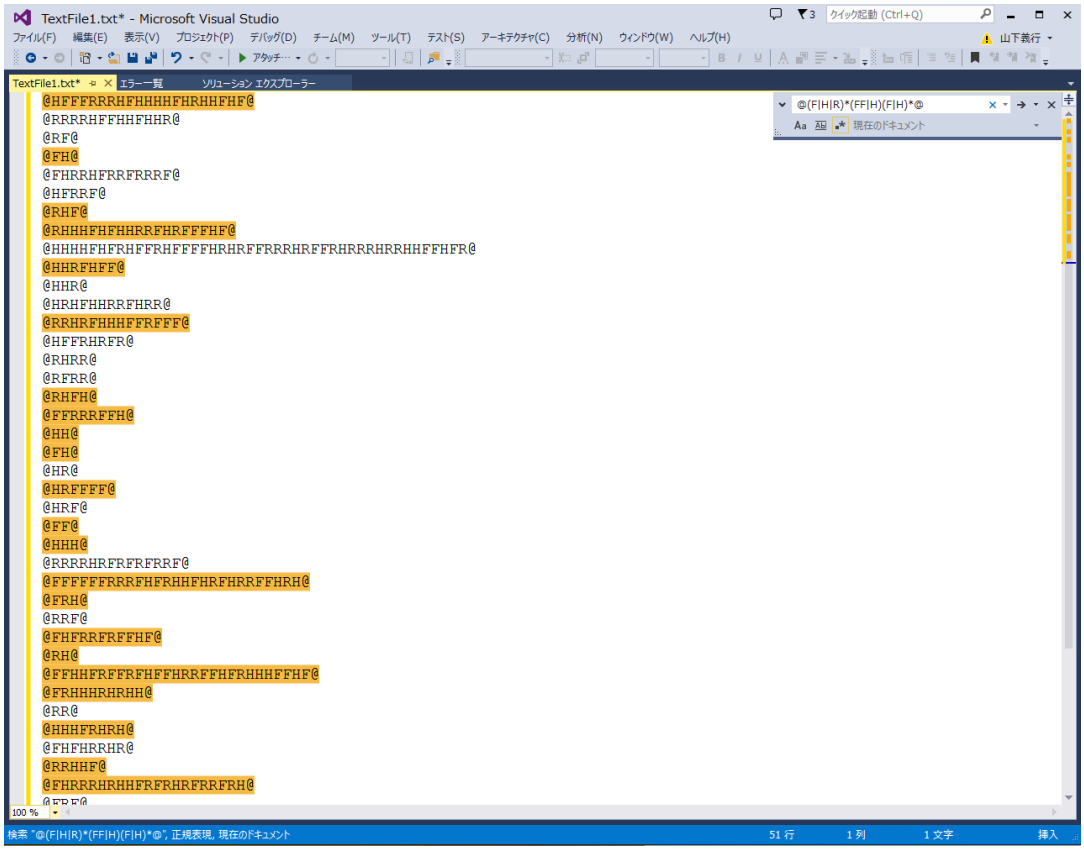


図 4.1: Visual Studio の正規表現による検索例

これらの文字列で缶ジュースが出てくることを確認してください。また、この出力に残らなかった文字列では缶ジュースが出てこないことを確認してください。

さて、grep はどのような仕組みで文字列と正規表現のマッチングを行っているのでしょうか。

4.4.2　Visual Studio による文字列検索

本学科で使用している統合プログラム開発環境 Visual Studio にも文字列検索機能が備わっています。変数名や関数名をキーワードとしてプログラム中から特定部位を検索する機能は、プログラム・サイズが少し大きくなれば必要不可欠です。単純な文字列ではなく、文字列のパターンを検索したい場合も出てくるでしょう。そのようなときに正規表現による検索は便利です。

図 4.1 は、先に紹介した grep と同じ検索を Visual Studio で行った例です。これを行うには、まずメニューを編集→検索と置換→クイック検索と選び、検索ボックスを開きます（図中の右上）。次に、ボックス左下の三つのトグルスイッチ  の中から右側のスイッチ  をオンにします。これは正規表現を用いるか否かを選択するスイッチです。最後に正規表現をボックスに入力すれば、図 4.1 のように正規表現とマッチする（正規表現の表す形式言語に含まれる）文字列が全てハ

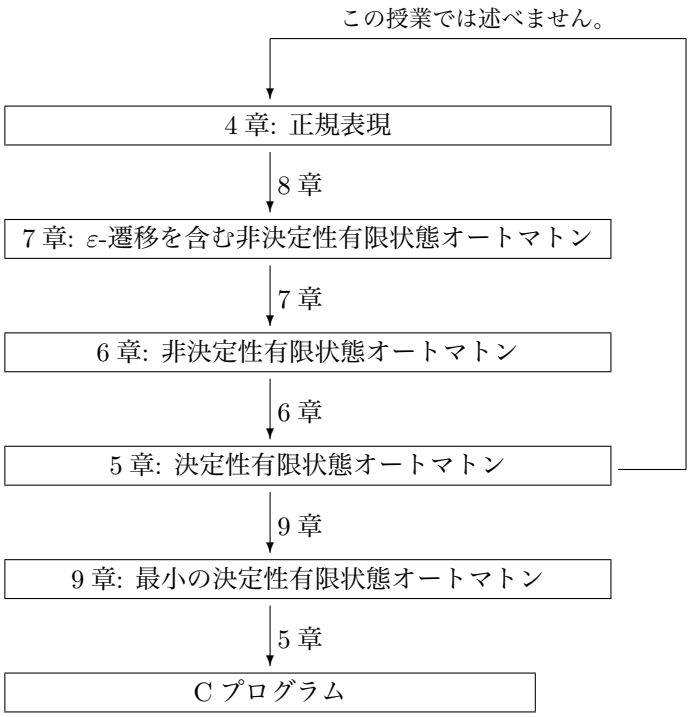


図 4.2: 有限状態オートマトンと正規表現の変換関係

イライトされます。

さて、Visual Studio はどのような仕組みで文字列と正規表現のマッチングを行っているのでしょうか。

4.5　後章の展望（正規表現と有限状態オートマトンの相互変換）

この授業の目的のひとつは、前節で紹介した grep や Visual Studio の例のように、与えられた正規表現を文字列照合に利用し、実際に照合プログラムをコンピュータ上で動かす原理を学ぶことです。言い換えれば、任意に与えられた正規表現についてその言語の所属問題を解くプログラムを生成する原理を学びます。

たとえば、缶ジュースが出ていくる正規表現 $r = (\mathrm{H}|\mathrm{F}|\mathrm{R})^*(\mathrm{H}|\mathrm{FF})(\mathrm{H}|\mathrm{F})^*$ が与えられたとします。私たちの目的は、別途、任意に与えられる文字列（たとえば “FHRH” や “FHFRF”）がその言語 $L(r)$ に含まれるか否かを判定するプログラムを作ることです。そのプログラムは「“FHRH” はその言語に含まれる」、「“FHFRF” はその言語に含まれない」などと判定できねばなりません。実はそのようなプログラムを作るには特別な創意工夫も直感も必要ありません。任意に与えられた正規表現についてその所属問題を解くプログラムは機械的に作ることができます。実際、grep や Visual Studio はそれを行っています。その手法を解説することがこの授業の目的のひとつです。そ

して、その手法を通してコンピュータやアルゴリズム、プログラムに関する深い理解を得ること、背後に潜むプログラムの数理的な構造を知ることが最終的な目的です。

図 4.2 は、この章と後章の内容の関係を正規表現と各種オートマトンの相互変換という観点からまとめたものです。まだオートマトンについては学んでいないため、以下の説明だけは分かりにくいかもしれませんが、講義の全体像をあらかじめ示し、講義の途中で目標を見失わないようにこの図を用意しました。

まず、図の一番上にこの章で定義した 正規表現 を置きました。この正規表現を図の下方に向かって順次変換していきます。次章（5 章）では図中の真ん中やや下にある 決定性有限状態オートマトン を学びます。最終的には、任意の正規表現がそれと等価な決定性有限状態オートマトンへ機械的に変換できることが示すのがこの授業の大きな目的です。

しかし、正規表現と決定性有限状態オートマトンは相当に異なる数学的体系ですから、両者の溝を埋めるべく、6 章では 非決定性 有限状態オートマトンをそれぞれ学びます。そして、非決定性有限状態オートマトンから決定性有限状態オートマトンへの変換を学びます。図 4.2 では下方への矢印を用いて変換を表わしています。次に、7 章では ε-遷移を含む 非決定性有限状態オートマトンを学びます。そして、ε-遷移を含む非決定性有限状態オートマトンから（6 章の ε-遷移を含まない）非決定性有限状態オートマトンへの変換を学びます。さらに 8 章では正規表現から ε-遷移を含む非決定性有限状態オートマトンへの変換を学びます。

これら一連の変換を連結することで、正規表現から決定性有限状態オートマトンまでの変換の道が完成します。次章で述べますが、決定性有限状態オートマトンを C プログラムへ変換することは容易です。結果、正規表現から C プログラムへの機械的な変換が可能になります。もう少し正確に言うならば、任意の正規表現 r について、その言語 $L(r)$ に関する所属問題を解くプログラムが機械的に導出できるのです。

ところで、機械的な変換では、変換前のオートマトンに比べて変換後のオートマトンが 冗長な構成に なりがちであることが知られています。ここで言う「冗長」とは、くだけた言い方をすれば、同じ機能を持つ機械（オートマトン）を作るにしても、機械に無駄な部品が多く使用されている状況を指します。そのような無駄は機械の実行効率を落としますから望ましくありません。そこでここでは、決定性有限状態オートマトンから C プログラムを導出する直前に、3 回の変換において累積された冗長性を削減する処理を行います。それが、 最小 の決定性有限状態オートマトンを求める処理です。これを 9 章で学びます。

有限状態オートマトンの理論では決定性有限状態オートマトンを正規表現に変換する手法も知

られています（図 4.2 の下から上への矢印です）。そうすると、結局、正規表現、ε-遷移を含む非決定性有限状態オートマトン、（ε-遷移を含まない）非決定性有限状態オートマトン、決定性有限状態オートマトンの四者が相互に変換可能になります。言い換えれば、あるひとつの表現形式で表された言語（記号列の集合）は必ず他の三つの表現形式でも表現可能なのです。その意味から「これらは同じ言語クラスを表す」と言い、特にこの言語クラスを **正規言語**（regular language）と呼んでいます[9]。正規表現、3 種類の有限状態オートマトンはいずれも正規言語を表すための数学的体系なのです。

　相互変換可能であることの理論的意義は大きいのですが、しかし有限状態オートマトンから正規表現への変換に実用上の意義を見出すのは困難です。正規言語の理論で実用上最も重要な性質は、任意の正規表現が決定性有限状態オートマトンに変換できることです。正規表現はコンパクトであり、オートマトンに比べ、人にとって読みやすく、書きやすい表現形式です。それに対して、次章で述べる決定性有限状態オートマトンは入力記号列が機械によって認識される様子を素直に表しており、プログラム化し易いという特徴を持ちます。そこで、特定の記号列の集合を表現するためには正規表現を用い、その記号列を認識するプログラム（所属問題を解くプログラム）を作るために、正規表現を決定性有限状態オートマトンに変換し、最後にそれを C プログラムへ変換するという応用を行います。

[9] あるいは正則言語とも呼びます。

第5章　決定性有限状態オートマトン

図 4.2 のシナリオに従い、この章では決定性有限状態オートマトン（deterministic finite state automaton）を学びます。

このオートマトンには「決定性有限状態」と冠されていますが、その大まかな意味は次の通りです。

「決定性」とはオートマトンの動作が全ての状況で一意に定まることを意味します。これは次章で「非決定性」と対比させて学ぶ予定です。

「有限状態」とは、オートマトンの 状態数 が有限であることを意味します。オートマトンは動作によってその状態を変えていきます。その状態を全て数え上げても高々有限個の状態しか持たないオートマトンを有限状態オートマトンと呼びます。たとえばエレベータを考えましょう。エレベータが階から階へ移動している途中の状態を無視し、各階へ一時停止した状態だけを数え上げれば、エレベータは高々有限個の状態を持ち、有限状態オートマトンと見なすことができます。何故ならば、エレベータの状態を決めるパラメータは、それが上/下のどちらの方向に移動している途中なのか、あるいは全く停止中（誰もエレベータを使用していない）なのか（計 3 状態）、エレベータが今、どの階に止まっているか（n 階建てならば n 状態）、エレベータのドアは今、開いているか/閉まっているか（2 状態）、各階のエレベータのドアの側に付いている上階行きのボタン △ は押されているか否か[1]（$2 \times n$ 状態）、下階行きのボタン ▽ は押されているか否か[2]（$2 \times n$ 状態）です。これはエレベータの状態数が $24n^3$ 状態（$=3 \times n \times 2 \times (2 \times n) \times (2 \times n)$）であることを意味し、状態数は有限です。実は世の中の多くの事象や機械は有限状態オートマトンと見なせることが知られています。一見すると有限状態と思えないような事象も、細部を捨象すると、有限状態と見なして問題のない場合があります。

この章から 7 章までは 3 種類の有限状態オートマトンの性質を詳しく論じていきます。特にこの章では 3 種類の中で最も単純な決定性の動作を行うオートマトンの性質を論じます。

5.1　状態と遷移

3.3 節の缶ジュースの自動販売機を思い出しましょう。そして次の問題を考えましょう。

[1] 正確に言えば、最上階には上階行きのボタン △ はありませんから、その分だけ状態数は減ります。
[2] 正確に言えば、最下階には下階行きのボタン ▽ はありませんから、その分だけ状態数は減ります。

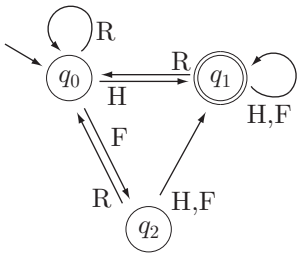


図 5.1: 缶ジュースの自動販売機の状態遷移図

問題　この自動販売機を有限状態オートマトンとして設計しなさい。

この問題の答えである有限状態オートマトンを **状態遷移図** (state transition graph、または state transition diagram) と呼ばれるグラフを用いて表わしてみましょう。図 5.1 が自動販売機を状態遷移図で表したものです。

　グラフの頂点 q_0、q_1、q_2 を **状態** (state) と呼びます。グラフの枝にはラベル H、F、R が付いていますが、これらはそのラベル (記号) を読んで状態を遷移することを表しています。3 個の状態の中で状態 q_1 のみ二重丸で表されていますが、このような状態を **最終状態** (final state) と呼びます。図 5.1 には最終状態は 1 個だけです。必要ならば最終状態は複数個設けても構いません。オートマトンが動作を開始する際の状態を特に **初期状態** (initial state) と呼びます。このテキストでは、状態遷移図の中で矢印 ↘ で指された状態をひとつだけ設け、それを初期状態とします。図 5.1 では q_0 が初期状態です。

　いま、ある人が列 HFRFHF の操作を行ったとします。このとき、オートマトンは与えられた操作列の記号を左からひとつずつ読みながら、初期状態から状態を変化 (遷移) させていきます。以下は列 HFRFHF に対応する遷移の模式図です。

$$q_0 \xrightarrow{\mathrm{H}} q_1 \xrightarrow{\mathrm{F}} q_1 \xrightarrow{\mathrm{R}} q_0 \xrightarrow{\mathrm{F}} q_2 \xrightarrow{\mathrm{H}} q_1 \xrightarrow{\mathrm{F}} q_1$$

初期状態 q_0 から動作を始め、記号を読む毎に状態を遷移し、そして全ての記号を読み終えたところで動作を停止します。記号列を全て読み込んだときに、もしオートマトンが最終状態で停止したならば、読み込まれた記号列は **受理された** (accepted) と言います。上の動作例では状態 q_1 で動作を停止し、q_1 は最終状態ですから、記号列 HFRFHF は受理されました。この自動販売機の例では、受理は「自動販売機から缶ジュースが出てくる」ことに対応します。

　逆に、もしオートマトンが最終状態で動作を停止しなかったならば、読み込まれた記号列は受理されなかった、あるいは拒否された (rejected) と言います。もうひとつの例題として、別の人が列 FFRFHRF の操作を行ったとします。このときのオートマトンの動作は以下の通りです。

$$q_0 \xrightarrow{\mathrm{F}} q_2 \xrightarrow{\mathrm{F}} q_1 \xrightarrow{\mathrm{R}} q_0 \xrightarrow{\mathrm{F}} q_2 \xrightarrow{\mathrm{H}} q_1 \xrightarrow{\mathrm{R}} q_0 \xrightarrow{\mathrm{F}} q_2$$

この場合、オートマトンの動作は状態 q_2 で終わっており、q_2 は最終状態ではないため、列 FFRFHRF は受理されません。「ジュースは出てこない」ということです。

さて、3.3 節の問題で求めた言語 L_J に含まれる記号列は図 5.1 のオートマトンで受理されることが確かめられます。逆に L_J に含まれない記号列は受理されないことも確かめられます。その意味で図 5.1 のオートマトンは言語 L_J の性質を表現したものになっています。

考察　缶ジュースの価格が 150 円の場合について、決定性有限状態オートマトンを求めなさい。

5.2　5 項組による定義

有限状態オートマトンを状態遷移図を用いて表すことは直感的で分かりやすい方法です。オートマトンの議論をするにはこの表現方法で十分と思えますし、実際ほとんどの場合にそれで十分です。しかし、オートマトンを数学的に厳密に扱う場合やオートマトンのプログラム化を検討する場合には、図示よりも集合と関数による定義が適しています。この節ではその定義の方法を解説します。

ひとつの有限状態オートマトン M は、5 項組 $(Q, \Sigma, \delta, q_0, F)$ で定義されます。ここに Q は状態の有限集合です。Σ は入力アルファベット（入力記号の有限集合）です。δ は、（現在の）状態と（今まさに読み込もうとしている）入力記号から（次の）状態を求める関数です。関数の定義域と値域を明示すると、$\delta : Q \times \Sigma \to Q$ となります。この関数を **遷移関数** (transition function) と呼びます。q_0（$\in Q$）は初期状態です。F（$\subseteq Q$）は最終状態の集合です。5 項組が与えられたならば、それから状態遷移図を作ることができることに注意しましょう。逆に、状態遷移図が与えられたならば、それから 5 項組を作ることができます。

さて、遷移関数 δ を拡張し、以下のように定義される関数 $\hat{\delta} : Q \times \Sigma^* \to Q$ を作ります。

$$\begin{aligned} \hat{\delta}(q, \varepsilon) &= q, \\ \hat{\delta}(q, au) &= \hat{\delta}(\delta(q, a), u) \end{aligned}$$

ただし $a \in \Sigma$、$u \in \Sigma^*$ とします。遷移関数 δ は ひとつの記号 を読んで状態を遷移する様子を表していますが、$\hat{\delta}$ は 記号列 を読んで遷移する様子を表していることに注意してください。ある記号列 u（$\in \Sigma^*$）について、$\hat{\delta}(q_0, u) \in F$ であるならば、すなわち初期状態 q_0 から記号列 u を読んで到達した状態 $\hat{\delta}(q_0, u)$ が最終状態であるならば、u はオートマトン M によって受理されると言います。そして M の **受理言語** $L(M)$ を M によって受理される記号列の全体と定

義します。以下は、それを式で表したものです。

$$L(M) = \{u \in \Sigma^* \mid \hat{\delta}(q_0, u) \in F\}$$

たとえば、缶ジュースの自動販売機（図 5.1）を M_J とするとき、それは 5 項組を用いて以下のように表すことができます。

$$M_J = (Q_J, \Sigma, \delta_J, q_0, F_J)$$

ここに Q_J、Σ、δ_J、F_J は以下の通りです。

$$\begin{array}{l} Q_J = \{q_0, q_1, q_2\}, \\ \Sigma = \{\mathrm{H}, \mathrm{F}, \mathrm{R}\}, \\ \delta_J(q_0, \mathrm{H}) = q_1, \quad \delta_J(q_0, \mathrm{F}) = q_2, \quad \delta_J(q_0, \mathrm{R}) = q_0, \\ \delta_J(q_1, \mathrm{H}) = q_1, \quad \delta_J(q_1, \mathrm{F}) = q_1, \quad \delta_J(q_1, \mathrm{R}) = q_0, \\ \delta_J(q_2, \mathrm{H}) = q_1, \quad \delta_J(q_2, \mathrm{F}) = q_1, \quad \delta_J(q_2, \mathrm{R}) = q_0, \\ F_J = \{q_1\}. \end{array}$$

あるいは Q_J、Σ、F_J は用いず、直接、以下のように書いてもいいでしょう。

$$M_J = (\{q_0, q_1, q_2\}, \{\mathrm{H}, \mathrm{F}, \mathrm{R}\}, \delta_J, q_0, \{q_1\})$$

M_J の受理言語は定義より以下の通りです。

$$L(M_J) = \{u \in \{\mathrm{H}, \mathrm{F}, \mathrm{R}\}^* \mid \hat{\delta}_J(q_0, u) \in \{q_1\}\}$$

たとえば記号列 HRHF がこの 5 項組によって受理されるか否かを調べるため、実際に遷移関数の値 $\hat{\delta}_J(q_0, \mathrm{HRHF})$ を計算してみましょう。計算は定義に則り以下のように進みます。

$$\begin{array}{rcll} \hat{\delta}_J(q_0, \mathrm{HRHF}) & = & \hat{\delta}_J(\delta_J(q_0, \mathrm{H}), \mathrm{RHF}) & \\ & = & \hat{\delta}_J(q_1, \mathrm{RHF}) & \\ & = & \hat{\delta}_J(\delta_J(q_1, \mathrm{R}), \mathrm{HF}) & \\ & = & \hat{\delta}_J(q_0, \mathrm{HF}) & \\ & = & \hat{\delta}_J(\delta_J(q_0, \mathrm{H}), \mathrm{F}) & \\ & = & \hat{\delta}_J(q_1, \mathrm{F}) & \\ & = & \hat{\delta}_J(\delta_J(q_1, \mathrm{F}), \varepsilon) & (\because\ \mathrm{F} = \mathrm{F}\varepsilon) \\ & = & \hat{\delta}_J(q_1, \varepsilon) & \\ & = & q_1 & \end{array}$$

よって、$\hat{\delta}_J(q_0, \mathrm{HRHF}) \in \{q_1\} = F_J$ となり、HRHF は受理されます。

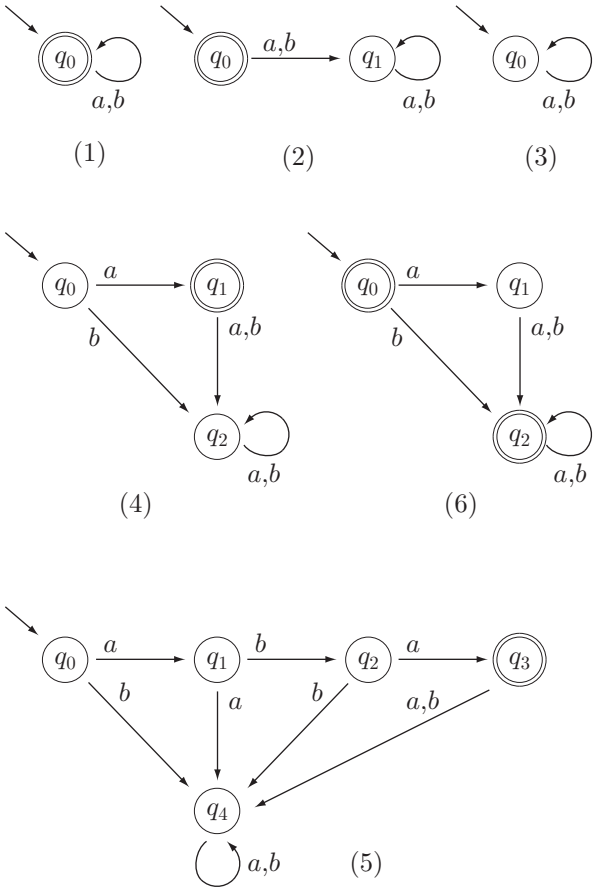


図 5.2: 様々な有限状態オートマトン（その 1）

5.3　有限状態オートマトンの例

この節では様々な種類の有限状態オートマトンの例を挙げて、その特徴を解説します。ただし、以下の全てのオートマトンについて $\Sigma = \{a, b\}$ とします。

ここに示す例 1 から例 12 のオートマトンは 4.3 節の例 1 から例 12 の正規表現にそれぞれ対応します。対応するものどうしの言語は同じです。比較のために、各例題の冒頭には対応する正規表現も記載しました。

例 1: $L((a|b)^*) = L(M_1) = \Sigma^*$

受理言語が $\Sigma^* = \{a, b\}^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, ...\}$、すなわち Σ 上の全ての記号列の集合であるようなオートマトン M_1 の状態遷移図は図 5.2 (1) の通りです。5 項組で表すと、$M_1 = (\{q_0\}, \Sigma, \delta_1, q_0, \{q_0\})$ となります。ここに遷移関数は以下の通りです。

$$\delta_1(q_0, a) = q_0, \quad \delta_1(q_0, b) = q_0.$$

すなわち、どのような記号列を読もうとも、常に最終状態 q_0 にいるようなオートマトンです。

例 **2:** $L(\varepsilon) = L(M_2) = \{\varepsilon\}$

空列のみを受理するオートマトン（受理言語が空列のみからなる集合であるオートマトン）M_2 の状態遷移図は図 5.2 (2) の通りです。5 項組で表すと、$M_2 = (\{q_0, q_1\}, \Sigma, \delta_2, q_0, \{q_0\})$ となります。ここに遷移関数は以下の通りです。

$$\begin{aligned} \delta_2(q_0, a) &= q_1, \quad \delta_2(q_0, b) = q_1, \\ \delta_2(q_1, a) &= q_1, \quad \delta_2(q_1, b) = q_1. \end{aligned}$$

このオートマトンでは記号を全く読まないでオートマトンの実行が終了するときに空列を受理します。もし記号をひとつ読んだならば、状態は q_1 へ遷移します。q_1 ではどんな記号を読もうとも状態は遷移せず、q_1 のままです。q_1 は最終状態ではありませんから、記号列は受理されません。結局、状態が一度 q_1 に遷移したならば、その後にどんな記号列を読もうとも状態は遷移せず、q_1 のままで、かつ決して記号列が受理されることはありません。このような状態を **死状態**（dead state）と呼びます。死状態は、記号列を積極的に受理しない機構をオートマトンに実現するためのテクニックのひとつです。

例 **3:** $L(\emptyset) = L(M_3) = \emptyset$

受理言語が空集合であるようなオートマトン M_3 の状態遷移図は図 5.2 (3) の通りです。5 項組で表すと、$M_3 = (\{q_0\}, \Sigma, \delta_3, q_0, \emptyset)$ となります。ここに遷移関数は以下の通りです。

$$\delta_3(q_0, a) = q_0, \quad \delta_3(q_0, b) = q_0.$$

このオートマトンは最終状態を持ちません。よって、どのような遷移を行うとも記号列が受理されることはなく、受理言語は空です。このオートマトンの状態 q_0 も死状態のひとつであり、しかもこのオートマトンは死状態のみからなる特殊なオートマトンです。

例 **4:** $L(a) = L(M_4) = \{a\}$

長さが 1 の記号列 a のみを受理するオートマトン M_4 の状態遷移図は図 5.2 (4) の通りです。5 項組で表すと、$M_4 = (\{q_0, q_1, q_2\}, \Sigma, \delta_4, q_0, \{q_1\})$ となります。ここに遷移関数は以下の通りです。

$$\begin{aligned} \delta_4(q_0, a) &= q_1, \quad \delta_4(q_0, b) = q_2, \\ \delta_4(q_1, a) &= q_2, \quad \delta_4(q_1, b) = q_2, \\ \delta_4(q_2, a) &= q_2, \quad \delta_4(q_2, b) = q_2. \end{aligned}$$

状態 q_2 が死状態であることに注意してください。

例 **5:** $L(aba) = L(M_5) = \{aba\}$

長さが 3 の記号列 aba のみが受理されるオートマトン M_5 の状態遷移図は図 5.2 (5) の通りです。5 項組で表すと、$M_5 = (\{q_0, q_1, q_2, q_3, q_4\}, \Sigma, \delta_5, q_0, \{q_3\})$ となります。ここに遷移関数は以下の通りです。

$$\begin{aligned} \delta_4(q_0, a) &= q_1, \quad \delta_4(q_0, b) = q_4, \\ \delta_4(q_1, a) &= q_4, \quad \delta_4(q_1, b) = q_2, \\ \delta_4(q_2, a) &= q_3, \quad \delta_4(q_2, b) = q_4, \\ \delta_4(q_3, a) &= q_4, \quad \delta_4(q_3, b) = q_4, \\ \delta_4(q_4, a) &= q_4, \quad \delta_4(q_4, b) = q_4. \end{aligned}$$

状態 q_4 が死状態であることに注意してください。

例 **6:** $L(\varepsilon|b|(a|b)(a|b)^+) = L(M_6) = \Sigma^* - \{a\}$

受理言語が $L(M_4)$ の 補集合 $\Sigma^* - \{a\}$ であるようなオートマトン M_6 の状態遷移図は図 5.2 (6) の通りです。5 項組で表すと、$M_6 = (\{q_0, q_1, q_2\}, \Sigma, \delta_6, q_0, \{q_0, q_2\})$ となります。ここに遷移関数は以下の通りです。

$$\begin{aligned} \delta_6(q_0, a) &= q_1, \quad \delta_6(q_0, b) = q_2, \\ \delta_6(q_1, a) &= q_2, \quad \delta_6(q_1, b) = q_2, \\ \delta_6(q_2, a) &= q_2, \quad \delta_6(q_2, b) = q_2. \end{aligned}$$

このオートマトンの状態遷移図は M_4 のそれと全く同じ構造をしています。M_4 と異なるのは、最終状態が M_4 と M_6 では相補的になっている点です。つまり、M_4 で受理する記号列は M_6 では受理せず、M_4 で受理しない記号列は M_6 では受理するのです。一般に、受理言語 $L(M)$ の補集合を受理するオートマトンを作るには単に最終状態の集合を補集合にすれば十分です。実はオートマトン M_1 と M_3 もそのような関係になっています。

ここで 4.3 節の例 4 と例 6 を思い出してください。例 4 の正規表現 a が自明であったのに対して、その補集合 $\Sigma^* - L(a)$ を言語とする例 6 の正規表現 $\varepsilon|b|(a|b)(a|b)^+$ の形は非常に複雑でした。しかし、それぞれに対応するオートマトンは同程度の複雑さを持っています。正規表現とオートマトンのこのような違いは、単純に数学的体系の構造の違いに由来します。多くの場合に正規表現は記号列の集合をコンパクトに表現できますが、万能ではなく、補集合のような不得手な場合もあるのです。

考察　受理言語が $\{a, b\}$ であるようなオートマトンを作りなさい。

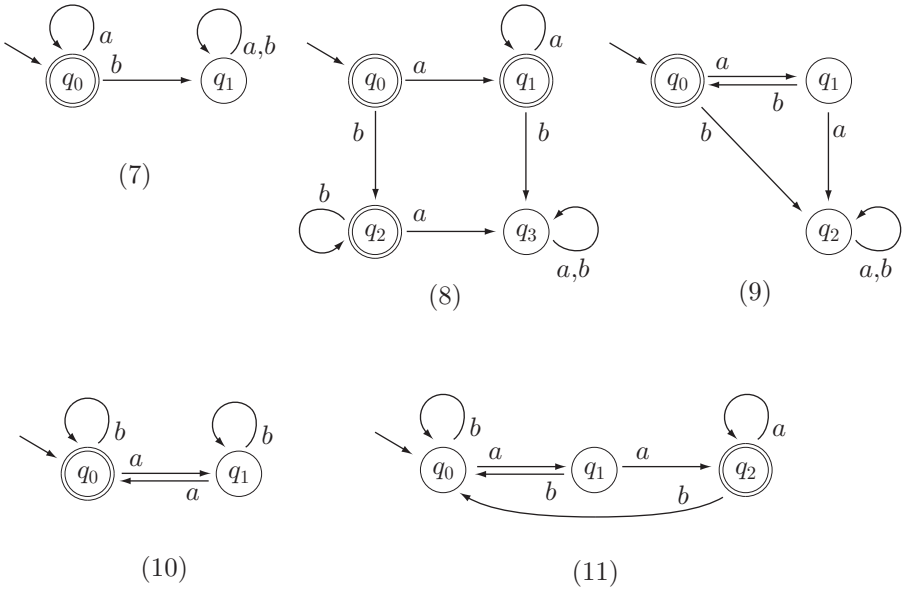


図 5.3: 様々な有限状態オートマトン（その 2）

例 **7:** $L(a^*) = L(M_7) = \{a^n \mid n \geq 0\}$

0 個以上の a からなる列を受理するオートマトン M_7 の状態遷移図は図 5.3 (7) の通りです。5 項組で表すと、$M_7 = (\{q_0, q_1\}, \Sigma, \delta_7, q_0, \{q_0\})$ となります。ここに遷移関数は以下の通りです。

$$\begin{aligned} \delta_7(q_0, a) &= q_0, \quad \delta_7(q_0, b) = q_1, \\ \delta_7(q_1, a) &= q_1, \quad \delta_7(q_1, b) = q_1. \end{aligned}$$

受理する記号列が a^n（a が n 回繰り返す）のような構造を持っているときには状態遷移図には 閉路 が存在します。図 5.3 (7) の場合は $q_0 \to q_0$ がその閉路です。なお、状態 q_1 は死状態です。

考察　受理言語が $\{a^n \mid n \geq 1\}$ であるようなオートマトンを作りなさい。この受理言語には空列 ε が含まれないことに注意しなさい。

例 **8:** $L(a^*|b^*) = L(M_8) = \{a^n \mid n \geq 0\} \cup \{b^n \mid n \geq 0\}$

0 個以上の a からなる列、または0 個以上の b からなる列を受理するオートマトン M_8 の状態遷移図は図 5.3 (8) の通りです。5 項組で表すと、$M_8 = (\{q_0, q_1, q_2, q_3\}, \Sigma, \delta_8, q_0, \{q_0, q_1, q_2\})$ となります。ここに遷移関数は以下の通りです。

$$\begin{aligned} \delta_8(q_0, a) &= q_1, \quad \delta_8(q_0, b) = q_2, \\ \delta_8(q_1, a) &= q_1, \quad \delta_8(q_1, b) = q_3, \\ \delta_8(q_2, a) &= q_3, \quad \delta_8(q_2, b) = q_2, \\ \delta_8(q_3, a) &= q_3, \quad \delta_8(q_3, b) = q_3. \end{aligned}$$

状態 q_1、q_2 はそれぞれ a^n、b^n（$n \geq 1$）を受理するための状態です。空列は初期状態 q_0 で受理されます。q_3 は死状態です。

例 **9:** $L((ab)^*) = L(M_9) = \{(ab)^n \mid n \geq 0\}$

a と b が交互に現れる列を受理するオートマトン M_9 の状態遷移図は図 5.3 (9) の通りです。5 項組で表すと、$M_9 = (\{q_0, q_1, q_2\}, \Sigma, \delta_9, q_0, \{q_0\})$ となります。ここに遷移関数は以下の通りです。

$$\begin{aligned} \delta_9(q_0, a) &= q_1, \quad \delta_9(q_0, b) = q_2, \\ \delta_9(q_1, a) &= q_2, \quad \delta_9(q_1, b) = q_0, \\ \delta_9(q_2, a) &= q_2, \quad \delta_9(q_2, b) = q_2 \end{aligned}$$

このオートマトンでは、閉路 $q_0 \to q_1 \to q_0$ の遷移を繰り返すことで記号列 $(ab)^n$（$n \geq 0$）を受理します。

例 **10:** $L((b^*ab^*a)^*b^*) = L(M_{10}) = \{u \in \Sigma^* \mid u \text{ に現れる } a \text{ の数が偶数 }\}$

記号 a と b の出現順序は問わず、記号列の中に現れる a の数が偶数（0 は偶数とします）であるような列、たとえば ε、bbb、aab、$abba$、... を受理するオートマトン M_{10} の状態遷移図は図 5.3 (10) の通りです。5 項組で表すと、$M_{10} = (\{q_0, q_1\}, \Sigma, \delta_{10}, q_0, \{q_0\})$ となります。ここに遷移関数は以下の通りです。

$$\begin{aligned} \delta_{10}(q_0, a) &= q_1, \quad \delta_{10}(q_0, b) = q_0, \\ \delta_{10}(q_1, a) &= q_0, \quad \delta_{10}(q_1, b) = q_1. \end{aligned}$$

状態 q_0 は a を偶数個読んだ状態、q_1 は奇数個読んだ状態です。記号 b を読むことは受理と関係しないため、b を読んでも状態が変化しません。

なお、この例も、正規表現が不得手とする場合です。$(b^*ab^*a)^*b^*$ が a を偶数個含む記号列を表わすと理解することは決して容易ではありません。

考察　集合 $\{u \in \Sigma^* \mid u \text{ に現れる } a \text{ の数が偶数かつ } u \text{ に現れる } b \text{ の数が偶数 }\}$ を受理集合とするようなオートマトンを作りなさい。

例 **11:** $L((a|b)^*aa) = L(M_{11}) = \{uaa \mid u \in \Sigma^*\}$

u を Σ の上の任意の列とするとき、uaa は aa で終わる列を意味します。そのような列を受理するオートマトン M_{11} の状態遷移図は図 5.3 (11) の通りです。5 項組で表すと、$M_{11} = (\{q_0, q_1, q_2\}, \Sigma, \delta_{11}, q_0, \{q_2\})$ となります。ここに遷移関数は以下の通りです。

$$\begin{aligned} \delta_{11}(q_0, a) &= q_1, \quad \delta_{11}(q_0, b) = q_0, \\ \delta_{11}(q_1, a) &= q_2, \quad \delta_{11}(q_1, b) = q_0, \\ \delta_{11}(q_2, a) &= q_2, \quad \delta_{11}(q_2, b) = q_0. \end{aligned}$$

q_0 は直前に読んだ記号が a ではなかった状態、q_1 は 2 個前の記号は a ではなく、直前の記号が a であった状態、q_2 は 2 個前と直前に読んだ記号が共に a であった状態を表します。途中で b を読んだ場合には、現在の状態に依らず、必ず q_0 へ遷移します。この「現在の状態に依らず、必ず q_0 へ遷移する」ことはある種の リセット 機構と考えることができます。リセットボタンが押されたときに機械がそのときの状態に関わらず初期状態へ戻るという仕組みです。同様のリセット機構は、3.3 節の自動販売機の返却ボタンの動作、言い換えれば、図 5.1 の自動販売機のオートマトンの「記号 R を読んだときには、必ず q_0 へ遷移する」動作にも見ることができます。リセット機構はオートマトンを設計する上でのテクニックのひとつです。

考察　集合 $\{uaav \mid u, v \in \Sigma^*\}$ を受理集合とするようなオートマトンを作りなさい。すなわち、aa という部分列を含む列、たとえば aa、baa、$abaa$、$abbaabbb$ などを受理するオートマトンを作りなさい。

例 **12:** 有限状態オートマトンで表わすことができない言語

　如何なる有限状態オートマトンでも表現できない言語が存在します。これは 4.3 の例 12 と同じ議論です。たとえば、$\{a^n b^n \mid n \geq 0\}$ は正規表現で表現できませんでしたが、有限状態オートマトンでもやはり表現できません。

考察　適当な正数 N （たとえば $N = 2, 3, 4, ...$）について、集合 $\{a^n b^n \mid N \geq n \geq 0\}$ を受理集合とするようなオートマトンを作りなさい。

5.4　C 言語による実装

　決定性有限状態オートマトンを C 言語で実装することはとても簡単です。

　たとえば記号 a、b を `char` 型の ASCII 文字 ‘a’、‘b’ で代用することにします。そして ‘a’、‘b’ からなる記号列を文字配列（`char []` 型）で表します。次に文字配列 `char in[]` を引数とする関数を考え、その引数がオートマトンによって受理されたならば整数値 `1` を返し、さもなくば `0` を返すように関数を定義すればよいのです。

　たとえば前節の M_{10} の場合、プログラムは図 5.4 の通りです。関数 `int M10(char in[])` が M_{10} に対応するオートマトンです。この関数では状態をラベルに対応させ、状態遷移を `goto` を用いて行います。配列要素の値がヌル文字 ’\0’ になったときに状態 q_0（ラベル `q_0` の位置）に居たならば受理と見なし、`1` を返します。配列要素の値がヌル文字 ’\0’ になったときに状態 q_1（ラベル `q_1` の位置）に居たならば不受理と見なし、`0` を返します。配列要素の値が想定外の文字

```
int M10(char in[]){
        int n = 0;

        q_0:    switch(in[n++]){
                case 'a': goto q_1;
                case 'b': goto q_0;
                case '\0': return 1;
                default: return -1;
                }

        q_1:    switch(in[n++]){
                case 'a': goto q_0;
                case 'b': goto q_1;
                case '\0': return 0;
                default: return -1;
                }
}

int main(){
        char* input = "aabaaaba";
        printf("\n%s : %d\n",input,M10(input));
}
```

図 5.4: オートマトン M_{10} の C プログラムによる実装（goto による）

であった（すなわち、原因不明のエラーが発見された）場合には、-1 を返します。main 関数では、入力文字配列 input を初期設定し、関数 M10() を呼び出しています。

図 5.4 のプログラムは、構造化プログラミングで忌み嫌われる goto を使ってオートマトンを実現しています。これを goto を用いず、再帰呼び出し (recursive call) を用いて実装することも可能です。図 5.5 は、各状態をひとつの関数として実現し、状態の遷移を関数呼び出しで実現しています。しかし、一般に関数呼び出しは実行時間とメモリを大量に消費します。しかも図 5.5 のプログラムが図 5.4 に比べて可読性に優れているという訳でもありません。よって総合的に見て図 5.4 のプログラムの方が良い実装と言えるでしょう。

考察　様々な有限状態オートマトンを C/C++/Java のプログラムで実装してみなさい。

5.5　練習問題

1. アルファベット Σ が以下の (a)〜(c) であるときに、言語

$$
\begin{aligned}
L &= \{(aaa)^n \mid n \geq 0\} \\
&= \{\varepsilon, aaa, aaaaaa, aaaaaaaaa, ...\}
\end{aligned}
$$

```
int q_0(char in[]){
        switch(in[0]){
        case 'a': return q_1(in+1);
        case 'b': return q_0(in+1);
        case '\0': return 1;
        default: return -1;
        }
}
int q_1(char in[]){
        switch(in[0]){
        case 'a': return q_0(in+1);
        case 'b': return q_1(in+1);
        case '\0': return 0;
        default: return -1;
        }
}
int M10(char in[]){
        return q_0(in);
}
```

図 5.5: オートマトン M_{10} の C プログラムによる実装（再帰呼び出しによる）

を受理する決定性有限状態オートマトン（の状態遷移図）をそれぞれ示しなさい。

(a) $\Sigma = \{a\}$

(b) $\Sigma = \{a, b\}$

(c) $\Sigma = \{a, b, c\}$

2. 上記 1. の三つのオートマトンを 5 項組で定義しなさい。

3. $\Sigma = \{a, b\}$ として、以下の言語を受理する決定性有限状態オートマトン（の状態遷移図）をそれぞれ示しなさい。

(a) 先頭と末尾にのみ b が現れる記号列の全体（bb, bab, $baaaab$ など）

(b) 記号列 bb を含み、その他の箇所に b は現れない記号列の全体（bb, abb, $aaabba$ など）

(c) 2 個の b を含む記号列の全体（b は連続して現れる必要はない。bb, $abab$, $aabaaabaa$, など）

(d) 記号列 bb を含む記号列の全体（b が 3 個以上現れてもよい。bb, $abbb$, $abaabbab$ など）

(e) a が偶数個（0 個を含む）、b が 3 の倍数個（0 個を含む）現れる記号列の全体（ε, aa, bbb, $ababb$, $abababa$ など）

(f) その記号列中の、長さが 3 の任意の部分記号列中に a が高々1 個しか現れないような記号列の全体（abb, $bbbabbbbabbabb$ など。$babbaba$ は末尾の aba に 2 個の a を含むから受理しない。）

第6章　非決定性有限状態オートマトン

この章では決定性有限状態オートマトンを機能拡張し、非決定性の動作を行う有限状態オートマトンを解説します。

「非決定性」（あるいは「非決定的」）とは、ある状態において次の動作が一意に定まらず、複数の動作が可能であることを言います。そのようなオートマトン（機械）を用いることで、形式言語の記述力はさらに向上します。しかも興味深いことに、非決定性有限状態オートマトンと決定性有限状態オートマトンが本質的には等価な記述力を持つことが明らかになります。一般に非決定的性質と決定的性質が等価になることはほとんどありません。たとえば現実世界の事象において「どちらを選択してもよい」（非決定性）という状況と「必ずこれを選択せよ」（決定性）という状況を比べれば、両者は全く異なります。それが等価になるのは、オートマトンの状態数が有限であるために成り立つ性質です。

6.1　非決定性

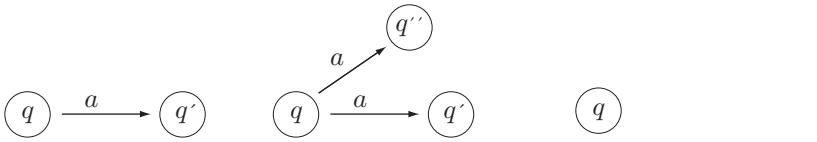
この節では非決定性の概念を説明します。ここの内容が理解できていないと、これ以降のこの授業の内容がほとんど最後まで理解できませんから、注意して学習してください。

6.1.1　非決定的な動作とは

前章の決定性オートマトンでは、状態 q において記号 a を読んだときの遷移先状態 q' は必ず一意に決まりました（図 6.1 (1) 参照）。このようなオートマトンの動作は決定性であると言います。これに対して、状態 q において記号 a を読んだときの遷移先状態が q', q'', ... と複数あってもよい（図 6.1 (2) 参照）とするオートマトンの動作を非決定的である（nondeterministic）と言います。

また、決定性オートマトンでは、状態 q において記号 a を読んだときの遷移先状態が存在しないということは決してありえませんが、非決定性オートマトンでは遷移先状態が無い（図 6.1 (3) 参照）場合も許されます。

非決定性の動作、特に遷移先が複数あるような（図 6.1 (2) 参照）動作は決定性の動作とは全く異なります。決定性ではオートマトンの動作が厳密に規定されており、ある特定の言語を受理するオートマトンを作るときに、どのような構造にすればよいのか、なかなかうまく発想できずに苦労



(1) 決定性　　(2) 非決定性（その1）　　(3) 非決定性（その 2, a による遷移がない）

図 6.1: 決定性と非決定性の違い

することがあります（5.5 節の練習問題を思い出しましょう）。それに対して非決定性の場合には、その特徴をうまく活かせば[1]、様々な言語を受理するオートマトンを容易に作ることができます。

オートマトンでは入力記号の読み込みを繰り返し、全ての入力記号を読み込んだ後に、その入力記号列が受理されるか否かを判定します。もしオートマトンの動作が非決定的ならば、ある遷移先を選択したときには入力記号列を受理できるものの、別の遷移先を選択したときにはその同じ入力記号列を受理できないという場合がありえます。では、複数の選択肢から遷移先をどのように選べばよいのでしょうか。

実は、非決定性有限状態オートマトンではどの遷移先を選ぶかは重要な問題ではありません。

非決定性有限状態オートマトンでは、初期状態から到達可能な 全て の路を調べ、その中に最終状態へ到達する路が 少なくともひとつ あるならば、その記号列は 受理 されると考えます。逆に、初期状態から到達可能な全ての路を調べ、その中に最終状態へ到達する路が全くないならば、記号列は受理されません。重要なことは、特定の動作経路について受理/不受理を判断するのではなく、全ての経路を尽くして判断するという点です。

よって、受理/不受理を判断するには到達可能な全ての経路を調べればよいのですが、有限状態オートマトンでは状態数が有限であり、また入力記号列も有限の長さであるため、到達可能な経路の総数も有限の範囲です。後に詳細を述べますが、このことが有限状態オートマトンの扱いを簡単にしています。

ほとんど全ての学生が前章の決定性オートマトンの概念を難なく理解します。しかし、この章に入って、非決定性の概念が理解できず、つまずく学生が少なくありません。これ以降を読み進めていく中で「非決定性」の意味が分からなくなったときには、必ずここに立ち戻って再確認してください。

では、前置きはこれくらいにして、非決定性有限状態オートマトンについて本題に入って行きましょう。慣れないうちは非決定性の便利さが理解しづらいかもしれませんが、しだいにその表現能

[1]非決定性をうまく利用できればいいのですが、初学者では決定性オートマトンよりも非決定性オートマトンの方が逆に難しく感じるようです。単に定義を理解するだけではなく、例題や練習問題で勘所を掴み、よく馴染んでおきましょう。

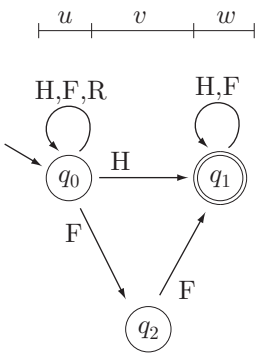


図 6.2: 缶ジュースの自動販売機の状態遷移図

力の高さに気付くはずです。

なお、これ以降は、記述の便利のために、決定性有限状態オートマトン（deterministic finite state automaton）を **DFA** と略記し、非決定性有限状態オートマトン (nondeterministic finite state automaton）を **NFA** と略記します。

6.1.2　缶ジュースの自動販売機

5.1 節の缶ジュースの自動販売機を再度検討します。

缶ジュースが出てくる記号列の集合 L_J を 5.1 節では以下のように求めました。

$$L_J = \{uvw \mid u \in \{\mathrm{H},\mathrm{F},\mathrm{R}\}^*,\ v \in \{\mathrm{H},\mathrm{FF}\},\ w \in \{\mathrm{H},\mathrm{F}\}^*\}$$

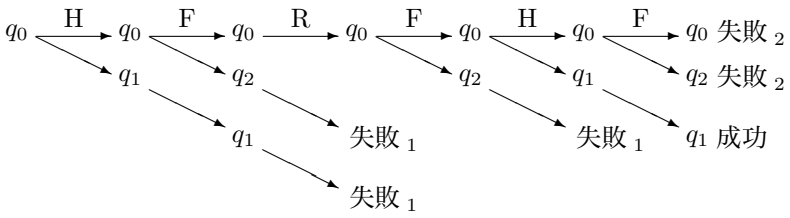
L_J に含まれる記号列は部分列 u、v、w を連接したものでした。しかし、この集合を受理する DFA（図 5.1）の形状は列 uvw の構造を反映する形にはなっておらず、集合とオートマトンの対応関係を読み取ることは容易ではありません。NFA を用いるならば、非決定的な状態遷移を用いた記述が可能であるため、記号列の構造を反映したオートマトンの設計が可能です。

図 6.2 は缶ジュースの自動販売機を NFA（の状態遷移図）で表現したものです。図の上部には、部分列 u、v、w とオートマトンの各部位の対応関係を示しました。状態 q_0 から H、F、R を読んで q_0 へ戻る閉路が u $(\in \{H,F,R\}^*)$ に対応します。状態 q_0 から状態 q_1 へ至る二つの経路が v $(\in \{H,FF\})$ に対応します。状態 q_1 から H、F を読んで q_1 へ戻る閉路が w $(\in \{H,F\}^*)$ に対応します。

前章の DFA と異なるのは、状態 q_0 において H を読んだときに q_0 と q_1 の複数の状態へ遷移できることです。同様に q_0 において F を読んだときに q_0 と q_2 へ遷移できます。このような遷移は DFA ではありえません。また q_2 において H、R を読んだときにはどこへも遷移できません。このような「どこへも遷移できない」という状況も DFA ではありえません。DFA では、ある状態 q に

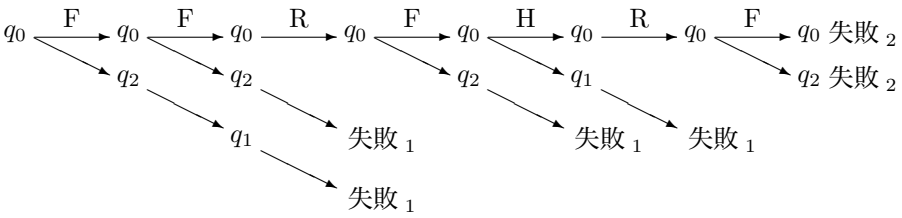
おいてある記号 a を読んだ時にどこへ遷移すべきか、その遷移先は常にひとつ定まっていました。しかし、NFA では遷移先は 複数 あっても構いませんし、全く 無くても 構いません。

図 6.2 のオートマトンで記号列 HFRFHF を読んだときの遷移は以下の図の通りです。



初期状態 q_0 において記号 H を読んだときの遷移可能な状態は q_0 と q_1 の二つです。このとき、どちらか一方の遷移が優先されるということは一切ありません。そこで、たとえば q_1 へ遷移してみます。そうすると、次に q_1 において記号 F を読むと q_1 へ遷移します。さらに q_1 において記号 R を読もうとしますが、このとき遷移する先がありません。よってこの路は行き止まりとなり、受理動作 失敗 したこととなります。上の図には遷移先を誤って途中で行き止まりになる路が三つあります。上の図ではこのような路の終端に「失敗 $_1$」を付けました。また、記号列全体を読み終えたものの最終状態に至らなかった路が二つあります。上の図ではこのような路の終端に「失敗 $_2$」を付けました。しかし、記号列全体を読み終え、最終状態に至る路がひとつ存在します。この路を上の図では「 成功 」で表しました。ひとつだけとは言え、成功する路がありますから、結局、記号列 HFRFHF はこのオートマトンによって受理されると考えます。もちろん成功する路が複数個あっても構いません。

もうひとつ例を示します。以下の図は、図 6.2 のオートマトンで記号列 FFRFHRF を読んだときの全ての状態遷移です。



この場合、全ての路が失敗しています。よって記号列 FFRFHRF は受理されません。

NFA における受理/不受理は 成功する路が少なくともひとつ存在するか否か で判断されます。受理される記号列による状態遷移であっても、失敗する路もあります。しかし、ひとつでも成功する路があれば、その記号列は受理されると判断します。

6.2　5 項組による定義

NFA は 5 項組 $M = (Q, \Sigma, \delta, q_0, F)$ で定義されます。ここに Q、Σ、q_0、F は DFA と同じです。すなわち Q は状態の有限集合、Σ は入力アルファベット、q_0（$\in Q$）は初期状態、F（$\subseteq Q$）は最終状態の集合です。遷移関数 δ は、DFA とは異なり、状態と入力記号から状態の集合への関数（$\in Q \times \Sigma \to 2^Q$）です。

遷移関数 δ を拡張し、以下のように定義される関数 $\hat{\delta} : Q \times \Sigma^* \to 2^Q$ を作ります。

$$\begin{aligned} \hat{\delta}(q, \varepsilon) &= \{q\}, \\ \hat{\delta}(q, au) &= \bigcup_{p \in \delta(q,a)} \hat{\delta}(p, u) \end{aligned}$$

ただし $a \in \Sigma$、$u \in \Sigma^*$ とします。そして記号列 u について $\hat{\delta}(q_0, u) \cap F \neq \emptyset$ であるとき、言い換えれば、初期状態 q_0 から記号列 u を読んで到達した状態のひとつ（$\in \hat{\delta}(q_0, u)$）が最終状態であるとき、u はオートマトン M によって受理されると定義します。すなわち、M の **受理言語** — これを $L(M)$ と表します — は M によって受理される記号列の全体です。

$$L(M) = \{u \in \Sigma^* \mid \hat{\delta}(q_0, u) \cap F \neq \emptyset\}$$

たとえば、缶ジュースの自動販売機（図 6.2）を $M_{J,2}$ とするとき、その 5 項組は以下の通りです。

$$M_{J,2} = (\{q_0, q_1, q_2\}, \{\mathrm{H}, \mathrm{F}, \mathrm{R}\}, \delta_{J,2}, q_0, \{q_1\})$$

ここに

$$\begin{array}{lll} \delta_{J,2}(q_0, \mathrm{H}) = \{q_0, q_1\}, & \delta_{J,2}(q_0, \mathrm{F}) = \{q_0, q_2\}, & \delta_{J,2}(q_0, \mathrm{R}) = \{q_0\}, \\ \delta_{J,2}(q_1, \mathrm{H}) = \{q_1\}, & \delta_{J,2}(q_1, \mathrm{F}) = \{q_1\}, & \delta_{J,2}(q_1, \mathrm{R}) = \emptyset, \\ \delta_{J,2}(q_2, \mathrm{H}) = \emptyset, & \delta_{J,2}(q_2, \mathrm{F}) = \{q_1\}, & \delta_{J,2}(q_2, \mathrm{R}) = \emptyset. \end{array}$$

$M_{J,2}$ の受理言語は定義より

$$L(M_{J,2}) = \{u \in \{\mathrm{H}, \mathrm{F}, \mathrm{R}\}^* \mid \hat{\delta}_{J,2}(q_0, u) \cap \{q_1\} \neq \emptyset\}$$

となります。

記号列 HRHF が $M_{J,2}$ で受理されるか否か、$\hat{\delta}_{J,2}(q_0, \mathrm{HRHF})$ を計算してみましょう。

$$\begin{aligned}
\hat{\delta}_{J,2}(q_0, \mathrm{HRHF}) &= \bigcup_{p \in \delta_{J,2}(q_0, \mathrm{H})} \hat{\delta}_{J,2}(p, RHF) \\
&= \hat{\delta}_{J,2}(q_0, RHF) \cup \hat{\delta}_{J,2}(q_1, RHF) \\
&= \bigcup_{p \in \delta_{J,2}(q_0, \mathrm{R})} \hat{\delta}_{J,2}(p, HF) \quad \cup \quad \bigcup_{p \in \delta_{J,2}(q_1, \mathrm{R})} \hat{\delta}_{J,2}(p, HF) \\
&= \hat{\delta}_{J,2}(q_0, HF) \quad \cup \quad \emptyset \\
&= \bigcup_{p \in \delta_{J,2}(q_0, \mathrm{H})} \hat{\delta}_{J,2}(p, F) \\
&= \hat{\delta}_{J,2}(q_0, F) \cup \hat{\delta}_{J,2}(q_1, F) \\
&= \bigcup_{p \in \delta_{J,2}(q_0, \mathrm{F})} \hat{\delta}_{J,2}(p, \varepsilon) \quad \cup \quad \bigcup_{p \in \delta_{J,2}(q_1, \mathrm{F})} \hat{\delta}_{J,2}(p, \varepsilon) \\
&= (\hat{\delta}_{J,2}(q_0, \varepsilon) \cup \hat{\delta}_{J,2}(q_2, \varepsilon)) \quad \cup \quad \hat{\delta}_{J,2}(q_1, \varepsilon) \\
&= (\{q_0\} \cup \{q_2\}) \quad \cup \quad \{q_1\} \\
&= \{q_0, q_1, q_2\}
\end{aligned}$$

そして $\hat{\delta}_{J,2}(q_0, \mathrm{HRHF}) \cap \{q_1\} = \{q_1\} \neq \emptyset$ となりますから、HRHF は受理されます。

さて、缶ジュースが出てくる記号列の集合 $L_J = \{uvw \mid ...\}$ の各部分列 u、v、w は、図 6.2 のオートマトンの遷移とそれぞれ厳密な対応を見ることができます。まず、$u \in \{\mathrm{H}, \mathrm{F}, \mathrm{R}\}^*$ に対しては以下の部分が対応します。

$$\delta_{J,2}(q_0, \mathrm{H}) \ni q_0, \quad \delta_{J,2}(q_0, \mathrm{F}) \ni q_0, \quad \delta_{J,2}(q_0, \mathrm{R}) \ni q_0.$$

$v \in \{\mathrm{H}, \mathrm{FF}\}$ に対しては以下の部分です。

$$\delta_{J,2}(q_0, \mathrm{H}) \ni q_1, \quad \delta_{J,2}(q_0, \mathrm{F}) \ni q_2, \quad \delta_{J,2}(q_2, \mathrm{F}) \ni q_1$$

そして、$w \in \{\mathrm{H}, \mathrm{F}\}^*$ について以下の部分です。

$$\delta_{J,2}(q_1, \mathrm{H}) \ni q_1, \quad \delta_{J,2}(q_1, \mathrm{F}) \ni q_1.$$

このように、NFA は受理言語を柔軟に記述できる点で DFA よりも強力であり、正規表現との親和性[2] も高くなっています。

さて、任意 の DFA は NFA です。何故ならば、決定性の動作は非決定性の動作の特殊なケースと考えられるからです。たとえば、前章の缶ジュースの自動販売機の DFA M_J は 5 項組を用いて以下のように表わしました（5.2 節参照）。

$$M_J = (\{q_0, q_1, q_2\}, \{\mathrm{H}, \mathrm{F}, \mathrm{R}\}, \delta_J, q_0, \{q_1\})$$

[2] 正規表現で言い表した記号列の性質をオートマトンの構造に素直に反映できること。

$$\begin{array}{lll} \delta_J(q_0,\mathrm{H})=q_1, & \delta_J(q_0,\mathrm{F})=q_2, & \delta_J(q_0,\mathrm{R})=q_0, \\ \delta_J(q_1,\mathrm{H})=q_1, & \delta_J(q_1,\mathrm{F})=q_1, & \delta_J(q_1,\mathrm{R})=q_0, \\ \delta_J(q_2,\mathrm{H})=q_1, & \delta_J(q_2,\mathrm{F})=q_1, & \delta_J(q_2,\mathrm{R})=q_0. \end{array}$$

これをそのまま NFA と読み替えるには以下のような 5 項組 M'_J を作ればよいのです。

$$M'_J = (\{q_0,q_1,q_2\},\{\mathrm{H},\mathrm{F},\mathrm{R}\},\delta'_J,q_0,\{q_1\})$$

$$\begin{array}{lll} \delta'_J(q_0,\mathrm{H})=\{q_1\}, & \delta'_J(q_0,\mathrm{F})=\{q_2\}, & \delta'_J(q_0,\mathrm{R})=\{q_0\}, \\ \delta'_J(q_1,\mathrm{H})=\{q_1\}, & \delta'_J(q_1,\mathrm{F})=\{q_1\}, & \delta'_J(q_1,\mathrm{R})=\{q_0\}, \\ \delta'_J(q_2,\mathrm{H})=\{q_1\}, & \delta'_J(q_2,\mathrm{F})=\{q_1\}, & \delta'_J(q_2,\mathrm{R})=\{q_0\}. \end{array}$$

このようにして、任意の DFA はそのまま NFA に読み替えることができるのです。

6.3　様々な NFA

前章で示した DFA の言語の例（5.3 節参照）の中には、NFA を用いることで、より簡素に表現できるものがあります。以下、それを示し、前章との比較を通して DFA と NFA の違いを明らかにします。もちろん、前章と同様に $\Sigma=\{a,b\}$ と約束します。なお、前章の例 1 から例 10 のうち、NFA を用いても DFA から簡単化できない例は以下に載せていません。

例 **2:** $L(\varepsilon)=L(M_2)=L(M_{12})=\{\varepsilon\}$

空列のみを受理する NFA M_{12} の状態遷移図は図 6.3 (1) の通りです。5 項組で表すと、$M_{12}=(\{q_0\},\Sigma,\delta_{12},q_0,\{q_0\})$ となります。ここに遷移関数は以下の通りです。

$$\delta_{12}(q_0,a)=\emptyset, \quad \delta_{12}(q_0,b)=\emptyset$$

前章の図 5.2 (2) のオートマトンでは死状態が必要でしたが、NFA では必要ありません。NFA では動作に行き詰まった場合には受理しないと約束されているからです。

例 **3:** $L(\emptyset)=L(M_3)=L(M_{13})=\emptyset$

受理言語が空集合であるような NFA M_{13} の状態遷移図は図 6.3 (2) の通りです。5 項組で表すと、$M_{13}=(\{q_0\},\Sigma,\delta_{13},q_0,\emptyset)$ となります。ここに遷移関数は以下の通りです。

$$\delta_{13}(q_0,a)=\emptyset, \quad \delta_{13}(q_0,b)=\emptyset$$

この 5 項組は例 2 の M_{12} の 5 項組と似ています。違いは、初期状態が最終状態か否かです。また、前章の図 5.2 (3) の DFA では自分自身へ戻る枝が必要でしたが、NFA ではそのような枝を付ける必要がありません。たとえ q_0 から q_0 へ遷移したところで受理しない事実に変わりないからです。

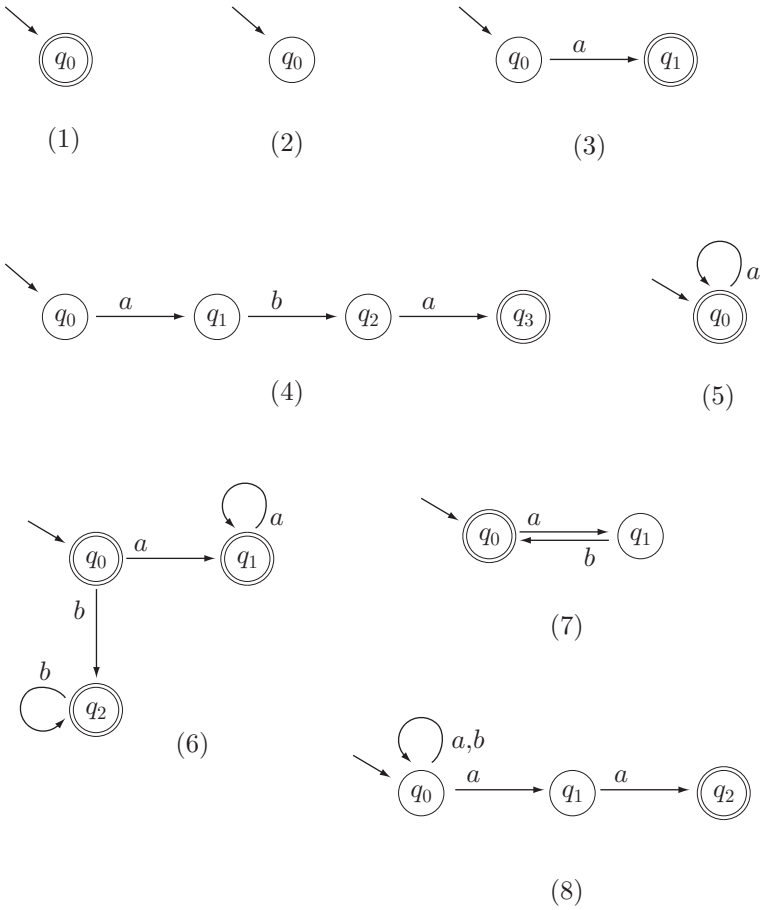


図 6.3: 様々な NFA

例 **4:** $L(a) = L(M_4) = L(M_{14}) = \{a\}$

受理言語が記号列 a だけからなる集合 $\{a\}$ であるような NFA M_{14} の状態遷移図は図 6.3 (3) の通りです。5 項組で表すと、$M_{14} = (\{q_0, q_1\}, \Sigma, \delta_{14}, q_0, \{q_1\})$ となります。ここに遷移関数は以下の通りです。

$$\begin{aligned} \delta_{14}(q_0, a) &= \{q_1\}, \quad & \delta_{14}(q_0, b) &= \emptyset, \\ \delta_{14}(q_1, a) &= \emptyset, & \delta_{14}(q_1, b) &= \emptyset. \end{aligned}$$

前章のオートマトン M_4 （図 5.2 (4)）では死状態を導入し、受理されないことが分かった記号列を途中から無駄読みしました。しかし NFA はそのような本質的でない部分の記述を必要としません。

なお、この言語の補集合を受理する NFA は前章の例 6 と同じものです。

考察　例 4 の言語の補集合 $\Sigma^* - \{a\}$ を受理する NFA の中で状態数が 2 個のものが存在するか否か、検討しなさい。

状態数を限定しなければ、同じ受理言語も持つ DFA、NFA は無数に存在する場合があることに

注意してください。最少の状態数のオートマトンに比べて状態数が多いオートマトンは冗長であると言います。前章の例 6（図 5.2 (6)）は状態数が 3 でした。

例 **5:** $L(aba) = L(M_5) = L(M_{15}) = \{aba\}$

受理言語が集合 $\{aba\}$ であるような NFA M_{15} の状態遷移図は図 6.3 (4) の通りです。5 項組で表すと、$M_{15} = (\{q_0, q_1, q_2, q_3\}, \Sigma, \delta_{15}, q_0, \{q_3\})$ となります。ここに遷移関数は以下の通りです。

$$\begin{array}{ll} \delta_{15}(q_0, a) = \{q_1\}, & \delta_{15}(q_0, b) = \emptyset, \\ \delta_{15}(q_1, a) = \emptyset, & \delta_{15}(q_1, b) = \{q_2\}, \\ \delta_{15}(q_2, a) = \{q_3\}, & \delta_{15}(q_2, b) = \emptyset, \\ \delta_{15}(q_3, a) = \emptyset, & \delta_{15}(q_3, b) = \emptyset. \end{array}$$

前章の DFA M_5（図 5.2 (5)）に対応しますが、やはり NFA では死状態を必要としません。

例 **7:** $L(a^*) = L(M_7) = L(M_{16}) = \{a^n \mid n \geq 0\}$

0 個以上の a からなる列を受理する NFA M_{16} の状態遷移図は図 6.3 (5) の通りです。5 項組で表すと、$M_{16} = (\{q_0\}, \Sigma, \delta_{16}, q_0, \{q_0\})$ となります。ここに遷移関数は以下の通りです。

$$\delta_{16}(q_0, a) = \{q_0\}, \quad \delta_{16}(q_0, b) = \emptyset.$$

このオートマトンでも死状態を必要としませんから、前章の例 7 の M_7 （図 5.3 (7)）よりも状態数が減っています。

例 **8:** $L(a^*|b^*) = L(M_8) = L(M_{17}) = \{a^n \mid n \geq 0\} \cup \{b^n \mid n \geq 0\}$

前章の例 8 が例 7 の拡張であったように、この章の例 8 も例 7 の拡張として作ることができます。この NFA M_{17} の状態遷移図は図 6.3 (6) の通りです。5 項組で表すと、
$M_{17} = (\{q_0, q_1, q_2\}, \Sigma, \delta_{17}, q_0, \{q_1, q_2\})$ となります。ここに遷移関数は以下の通りです。

$$\begin{array}{ll} \delta_{17}(q_0, a) = \{q_1\}, & \delta_{17}(q_0, b) = \{q_2\}, \\ \delta_{17}(q_1, a) = \{q_1\}, & \delta_{17}(q_1, b) = \emptyset, \\ \delta_{17}(q_2, a) = \emptyset, & \delta_{17}(q_2, b) = \{q_2\}. \end{array}$$

NFA の構造が a と b について対称的であることに注意してください。このオートマトンも死状態が必要ありませんから、前章の M_8 （図 5.3 (8)）よりも状態数が減っています。

例 **9:** $L((ab)^*) = L(M_9) = L(M_{18}) = \{(ab)^n \mid n \geq 0\}$

この NFA M_{18} の状態遷移図は図 6.3 (7) の通りです。5 項組で表すと、
$M_{18} = (\{q_0, q_1\}, \Sigma, \delta_{18}, q_0, \{q_0\})$ となります。ここに遷移関数は以下の通りです。

$$\begin{aligned} \delta_{18}(q_0, a) &= \{q_1\}, \quad & \delta_{18}(q_0, b) &= \emptyset, \\ \delta_{18}(q_1, a) &= \emptyset, & \delta_{18}(q_1, b) &= \{q_0\}. \end{aligned}$$

この NFA には DFA M_9 （図 5.3 (9)）の死状態 q_2 が必要ないことに注意してください。

実はこの節のここまで説明してきた NFA では死状態が必要ないという点が DFA との主な相違点でした。しかし、NFA と DFA の最も大きな違いは、NFA では複数の選択肢が可能という点です。その特長をうまく活かした例が以下です。

例 **11:** $L((a|b)^*aa) = L(M_{11}) = L(M_{19}) = \{uaa \mid u \in \Sigma^*\}$

記号列の末尾が必ず aa で終わるような列を受理する NFA M_{19} の状態遷移図は図 6.3 (8) の通りです。5 項組で表すと、$M_{19} = (\{q_0, q_1, q_2\}, \Sigma, \delta_{19}, q_0, \{q_2\})$ となります。ここに遷移関数は以下の通りです。

$$\begin{aligned} \delta_{19}(q_0, a) &= \{q_0, q_1\}, \quad & \delta_{19}(q_0, b) &= \{q_0\}, \\ \delta_{19}(q_1, a) &= \{q_2\}, & \delta_{19}(q_1, b) &= \emptyset, \\ \delta_{19}(q_2, a) &= \emptyset, & \delta_{19}(q_2, b) &= \emptyset. \end{aligned}$$

前章の DFA M_{11} （図 5.3 (11)）では b を読む度に q_0 へ遷移する構造になっていました。これはオートマトンの状態をリセットすることに相当します。しかし、NFA にはそのようなリセットの仕組みは必要ありません。

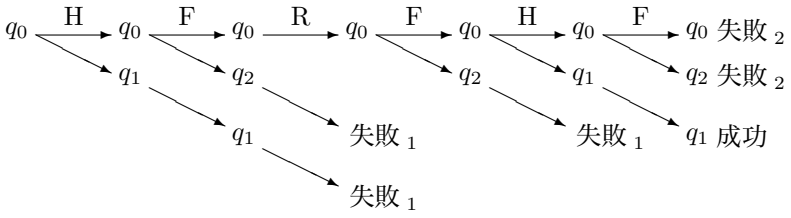
この NFA では $\delta_{19}(q_0, a) = \{q_0, q_1\}$ に複数の選択肢が存在します。a を読んだときに q_0 に戻るか、q_1 へ移動するか、どちらを選択しても構いません。全ての記号を読み込んだときにちょうど最終状態 q_2 へ到達しているような遷移が存在すればよいのです。

考察（**5** 章からのつづき）　集合 $\{uaav \mid u, v \in \Sigma^*\}$ を受理集合とするような NFA を作りなさい。すなわち、aa という部分列を含む列、たとえば aa、baa、$abaa$、$abbaabbb$ などを受理する NFA を作りなさい。

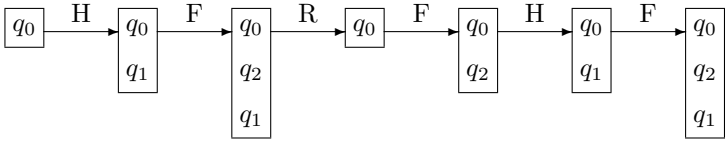
6.4　NFA から DFA への等価変換

NFA の最も興味深い性質は、任意 の NFA には 等価 な DFA が存在し、その NFA を DFA に変換する簡単な方法が知られていることです。

自動販売機の NFA（図 6.2）を思い出しましょう。記号列 HFRFHF による NFA の状態の遷移は以下の通りでした。



記号を読み進めるに従って、初期状態から到達可能な状態が増減を繰り返していきます。そして、記号を読み終えたときに、最終状態がひとつでも存在すれば、その記号列はこの NFA によって受理されたと考えました。この NFA の動作を DFA でシミュレートするには、NFA において初期状態から到達した状態の 集合 を DFA のひとつの状態と見なします。そうするとオートマトンの動作は読み込む記号に応じて常に一意に（決定的に）定まります。以下の図は、上の NFA の動作例について状態の集合を四角の枠で囲み、決定的な動作として表現したものです。



上の図の初期状態は $\boxed{q_0}$ です。ここで記号 H を読むと、NFA では状態 q_0 または q_1 に遷移しますが、それを DFA では状態 $\boxed{q_0\ q_1}$ に遷移すると解釈します。次に記号 F を読むと、NFA では q_0 から q_0 または q_2 へ遷移し、q_1 からは q_1 へ遷移します。それを DFA では状態 $\boxed{q_0\ q_1}$ から状態 $\boxed{q_0\ q_2\ q_1}$（あるいは $\boxed{q_0\ q_1\ q_2}$）へ遷移すると解釈します。これを繰り返し、結局、DFA は初期状態は $\boxed{q_0}$ から記号列 HFRFHF を読んで状態 $\boxed{q_0\ q_2\ q_1}$（あるいは $\boxed{q_0\ q_1\ q_2}$）へ遷移します。これを NFA に言い換えれば、q_0、q_1 または q_2 へ遷移することを意味し、この中に最終状態 q_1 が含まれていますから記号列は受理されたと考えます。DFA について言えば、$\boxed{q_0\ q_2\ q_1}$ の中には NFA の最終状態 q_1 が含まれますから、$\boxed{q_0\ q_2\ q_1}$ を DFA の最終状態と見なします。そうすると、DFA の最終状態に到達したこととなり、この DFA によって記号列は受理されます。

NFA の 状態の集合 を DFA の ひとつの状態 と見なし、NFA を DFA でシミュレートするアイディアはうまく行きそうです。実際うまく行きます。なぜならば、NFA の状態数は有限ですから状態の部分集合の数も高々有限です。よって NFA は DFA に変換可能です。今、NFA の状態数を n とするとき、変換後の DFA の状態の数は高々 2^n となります。よって変換後の DFA の状態数は元の NFA の状態数に比べて膨大なものになるかもしれません [3] が、しかし、状態数は有限あり、原理上は DFA を構成可能です。

以下は、このアイディアをまとめたものです。

[3] $n \geq 6$ のときには、$2^n \geq 64$ となりますから、人手で扱うことは困難かもしれません。

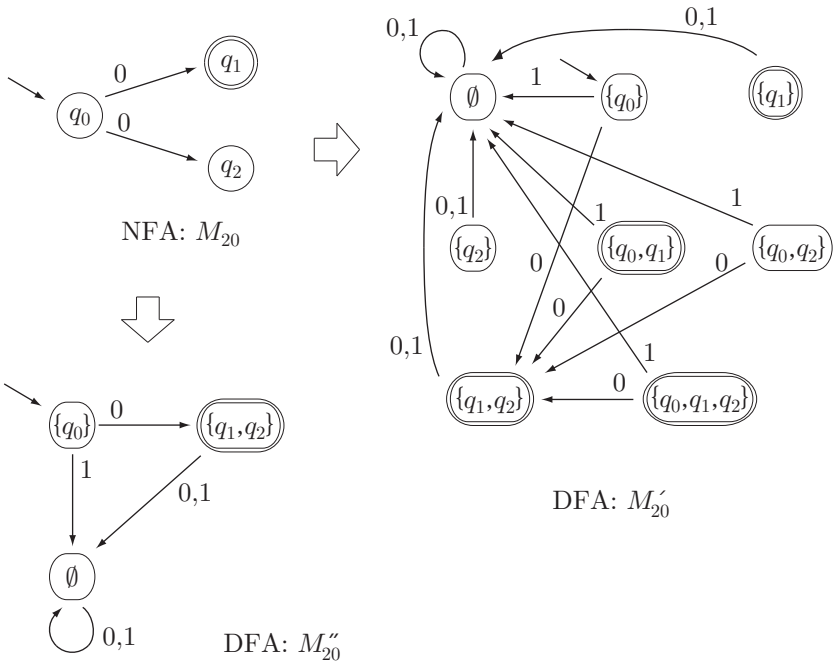


図 6.4: NFA から DFA への等価変換例（その 1）

変換規則 1　$M = (Q, \Sigma, \delta, q_0, F)$ を任意の NFA とします。そのとき、M と同じ 受理言語 を持つ DFA: $M' = (Q', \Sigma, \delta', q'_0, F')$ を必ず作ることができます。すなわち、記号列 u について、$u \in L(M)$ ならば $u \in L(M')$ であり、$u \notin L(M)$ ならば $u \notin L(M')$ であるような M' が存在し、そのような M' を実際に作ることができます。

まず、Q' を $Q' = 2^Q$ と設定します。つまり、Q の各部分集合を M' のひとつの状態とします。次に M の遷移関数 $\delta : Q \times \Sigma \to 2^Q$ を用いて、M' の遷移関数 $\delta' : 2^Q \times \Sigma \to 2^Q$ を以下のように定義します。

$$\delta'(P, a) = \bigcup_{p \in P} \delta(p, a)$$

さらに $q'_0 = \{q_0\}$ とします。M' の最終状態 $P \in F' \subseteq 2^Q$ は、$P \cap F \neq \emptyset$ を満たす Q の部分集合とします。すなわち、Q の部分集合であって、その中の要素が M の最終状態をひとつ以上含む集合の全体集合が F' です。　□

この変換の正当性はこの章の最後に示します。

まず、非常に簡単な例題で検証しましょう。

たとえば図 6.4 の NFA: M_{20} は 5 項組 $M_{20} = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\})$ で定義されます。ここに遷移関数 δ は以下の通りです。

$$
\begin{array}{ll}
\delta(q_0,0)=\{q_1,q_2\}, & \delta(q_0,1)=\emptyset,\\
\delta(q_1,0)=\emptyset, & \delta(q_1,1)=\emptyset,\\
\delta(q_2,0)=\emptyset, & \delta(q_2,1)=\emptyset
\end{array}
$$

この NFA を上の変換方式に従って DFA に変換したものが図 6.4 の M'_{20} です。5 項組で表すと、以下の通りです [4]。

$$
\begin{aligned}
M'_{20} &= (Q'_{20},\{0,1\},\delta'_{20},q'_0,F'_{20})\\
Q'_{20} &= 2^{\{q_0,q_1,q_2\}}\\
&= \{\emptyset,\{q_0\},\{q_1\},\{q_2\},\{q_0,q_1\},\{q_0,q_2\},\{q_1,q_2\},\{q_0,q_1,q_2\}\}\\
q'_0 &= \{q_0\}\\
F'_{20} &= \{\{q_1\},\{q_0,q_1\},\{q_1,q_2\},\{q_0,q_1,q_2\}\}
\end{aligned}
$$

ここに

$$
\begin{array}{ll}
\delta'_{20}(\emptyset,0)=\emptyset, & \delta'_{20}(\emptyset,1)=\emptyset,\\
\delta'_{20}(\{q_0\},0)=\{q_1,q_2\}, & \delta'_{20}(\{q_0\},1)=\emptyset,\\
\delta'_{20}(\{q_1\},0)=\emptyset, & \delta'_{20}(\{q_1\},1)=\emptyset,\\
\delta'_{20}(\{q_2\},0)=\emptyset, & \delta'_{20}(\{q_2\},1)=\emptyset,\\
\delta'_{20}(\{q_0,q_1\},0)=\{q_1,q_2\}, & \delta'_{20}(\{q_0,q_1\},1)=\emptyset,\\
\delta'_{20}(\{q_0,q_2\},0)=\{q_1,q_2\}, & \delta'_{20}(\{q_0,q_2\},1)=\emptyset,\\
\delta'_{20}(\{q_1,q_2\},0)=\emptyset, & \delta'_{20}(\{q_1,q_2\},1)=\emptyset,\\
\delta'_{20}(\{q_0,q_1,q_2\},0)=\{q_1,q_2\}, & \delta'_{20}(\{q_0,q_1,q_2\},1)=\emptyset.
\end{array}
$$

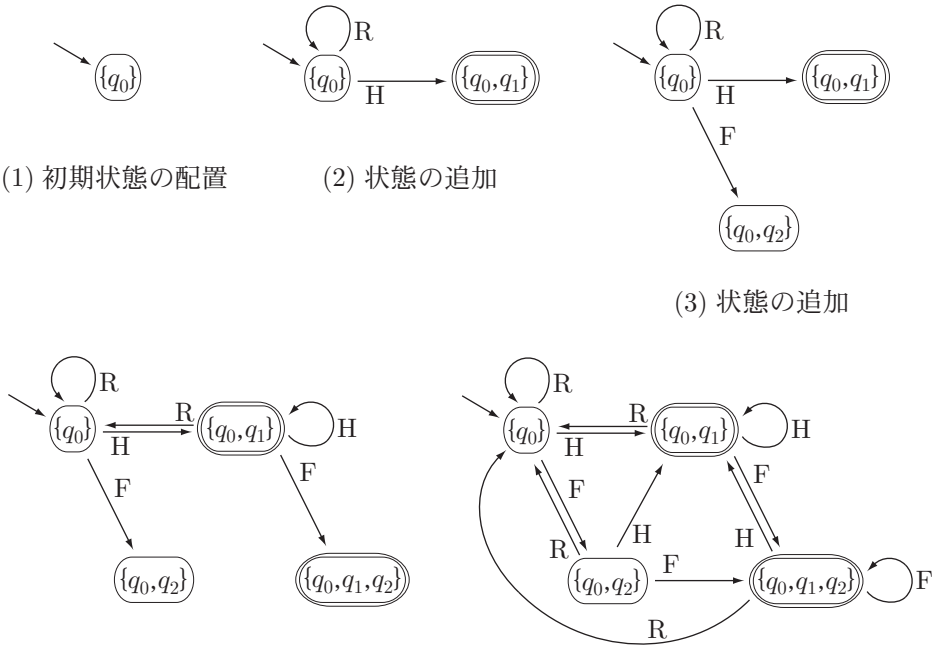
ところで、図 6.4 の M'_{20} を見ると、初期状態 $q'_0=\{q_0\}$ から適当な記号列を読んで遷移可能な状態は、初期状態 $\{q_0\}$ と $\emptyset$、$\{q_1,q_2\}$ のみです。その他の状態へは如何なる記号列を読んでも決して遷移することはありません。そこで、初期状態から 到達可能 な状態 $\{q_0\}$、$\emptyset$、$\{q_1,q_2\}$ のみを用いて M'_{20} を再構成したものが、図 6.4 の M''_{20} です。これを 5 項組で表すならば、

$$
\begin{aligned}
M''_{20} &= (Q''_{20},\{0,1\},\delta''_{20},q''_0,F''_{20})\\
Q''_{20} &= \{\emptyset,\{q_0\},\{q_1,q_2\}\}\\
q''_0 &= \{q_0\}\\
F''_{20} &= \{\{q_1,q_2\}\}
\end{aligned}
$$

ここに

$$
\begin{array}{ll}
\delta''_{20}(\emptyset,0)=\emptyset, & \delta''_{20}(\emptyset,1)=\emptyset,\\
\delta''_{20}(\{q_0\},0)=\{q_1,q_2\}, & \delta''_{20}(\{q_0\},1)=\emptyset,\\
\delta''_{20}(\{q_1,q_2\},0)=\emptyset, & \delta''_{20}(\{q_1,q_2\},1)=\emptyset.
\end{array}
$$

[4] オートマトンの教科書の中には、Q'_{20} の各状態を $\emptyset$, $\{q_0\}$, $\{q_1\}$, $\{q_2\}$, $\{q_0,q_1\}$, $\{q_0,q_2\}$ ではなく、[], $[q_0]$, $[q_1]$, $[q_2]$, $[q_0,q_1]$, $[q_0,q_2]$ と表記しているものがあります。というのは、括弧 { } を使う表記方法では、それが単なる集合を表わしているのか、ひとつの状態と見なすべき集合を表わしているのか、区別しにくいためです。しかし、[ ] を使うことでますます混乱する学生もいるようですから、このテキストでは集合の表記をそのまま用いました。



(1) 初期状態の配置　(2) 状態の追加　(3) 状態の追加

(4) 状態の追加　(5) DFA $M'_{J,2}$ の完成

図 6.5: NFA（図 6.2）から DFA への等価変換例（その 2、DFA の構築過程）

M''_{20} が M'_{20} と同じ受理集合を持つことは明らかです。

この例から分かるように、変換方式にそのまま従ったのでは初期状態から到達不能な状態を多数含んでしまいます。しかし、それらは全く無駄です。そこでこれ以降は 初期状態から到達可能な状態 のみを取り扱うと約束します。

ところで、空集合 $\emptyset$ もひとつの状態です。初期状態 $\{q_0\}$ から $\emptyset$ への遷移があるとき、$\emptyset$ は前章でいう死状態の役割をすることに注意してください。何故ならば、任意の記号 a について変換後の DFA の遷移関数 δ' は常に $\delta'(\emptyset, a) = \emptyset$ を満たします。よって一旦、状態 $\emptyset$ に入ると二度と状態 $\emptyset$ から抜け出せません。また状態 $\emptyset$ が DFA の最終状態になることはありません。これらの事実は状態 $\emptyset$ が **死状態** に他ならないことを意味しています。前節の例題で見たように、NFA ではわざわざ死状態を作る必要がありません。ある状態で記号を読めなくなったならば、その NFA の動作は失敗であったと判断すればよいのです。しかし、その NFA を変換した DFA では死状態 $\emptyset$ が自動的に作られるのです。

もうひとつの例として、缶ジュースの自動販売機 $M_{J,2}$（図 6.2）を DFA に変換します。

前の例で述べたように、変換後の DFA に必要な状態は $2^{\{q_0,q_1,q_2\}}$ の一部に過ぎません。そこで、実際に DFA を作るときには、初期状態 $\{q_0\}$ から到達可能な状態を順次追加していきます。

まず DFA の状態遷移図に初期状態だけを配置します（図 6.5 (1)）。次に、初期状態から記号 H

を読んで遷移できる状態 $\{q_0, q_1\}$ を遷移図に加えたものが図 6.5 (2) です。ここに状態 $\{q_0, q_1\}$ はもとの NFA の最終状態 q_1 を含みますから、DFA の最終状態です。よって状態遷移図の $\{q_0, q_1\}$ を二重丸で表します。さらに、初期状態から記号 F を読んで遷移できる状態 $\{q_0, q_2\}$ を遷移図に加えたものが図 6.5 (3) です。さらに、状態 $\{q_0, q_1\}$ から記号 F を読んで遷移できる状態 $\{q_0, q_1, q_2\}$ を遷移図に加えたものが図 6.5 (4) です。この状態も最終状態です。さて、同様の処理を繰り返しても、これ以上新たに遷移図に付け加えることのできる状態は見つかりません。そこで、残る作業は図 6.5 (4) に遷移の枝を書き加え、オートマトンを完成させることです。このようにして完成した DFA の状態遷移図が図 6.5 (5) です。

このオートマトン $M'_{J,2}$ を 5 項組で表すならば以下の通りです。

$$M'_{J,2} = (Q'_{J,2}, \{\mathrm{H}, \mathrm{F}, \mathrm{R}\}, \delta'_{J,2}, q'_0, F'_{J,2})$$

ここに、

$$\begin{aligned} Q'_{J,2} &= \{\{q_0\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_0, q_1, q_2\}\}, \\ q'_0 &= \{q_0\}, \\ F'_{J,2} &= \{\{q_0, q_1\}, \{q_0, q_1, q_2\}\}. \end{aligned}$$

遷移関数は以下の通りです。

$$\begin{array}{lll} \delta'_{J,2}(\{q_0\}, \mathrm{H}) = \{q_0, q_1\}, & \delta'_{J,2}(\{q_0\}, \mathrm{F}) = \{q_0, q_2\}, & \delta'_{J,2}(\{q_0\}, \mathrm{R}) = \{q_0\}, \\ \delta'_{J,2}(\{q_0, q_1\}, \mathrm{H}) = \{q_0, q_1\}, & \delta'_{J,2}(\{q_0, q_1\}, \mathrm{F}) = \{q_0, q_1, q_2\}, & \delta'_{J,2}(\{q_0, q_1\}, \mathrm{R}) = \{q_0\}, \\ \delta'_{J,2}(\{q_0, q_2\}, \mathrm{H}) = \{q_0, q_1\}, & \delta'_{J,2}(\{q_0, q_2\}, \mathrm{F}) = \{q_0, q_1, q_2\}, & \delta'_{J,2}(\{q_0, q_2\}, \mathrm{R}) = \{q_0\}, \\ \delta'_{J,2}(\{q_0, q_1, q_2\}, \mathrm{H}) = \{q_0, q_1\}, & \delta'_{J,2}(\{q_0, q_1, q_2\}, \mathrm{F}) = \{q_0, q_1, q_2\}, & \delta'_{J,2}(\{q_0, q_1, q_2\}, \mathrm{R}) = \{q_0\}. \end{array}$$

考察　図 6.3 の各 NFA をそれぞれ DFA に変換しなさい。

変換規則 1 の正当性

変換規則 1 の NFA $M = (Q, \Sigma, \delta, q_0, F)$ とそれを変換した DFA $M' = (2^Q, \Sigma, \delta', q'_0, F')$ の受理集合が等しいことを証明します。

これを証明するには、任意の記号列 u について、

$$\hat{\delta}(q_0, u) = \hat{\delta'}(\{q_0\}, u)$$

であることを証明すれば十分です。これには u の長さ $|u|$ に関する数学的帰納法を用います。

そして、これを証明するには、任意の状態集合 P について

$$\bigcup_{q \in P} \hat{\delta}(q, u) = \hat{\delta'}(P, u)$$

を証明すれば十分です。

基底段階: $u = \varepsilon$（すなわち、$|u| = 0$）とします。このとき、任意の状態 q について、$\hat{\delta}$ の定義（6.2 節）より以下が成り立ちます。

$$\bigcup_{q \in P} \hat{\delta}(q, \varepsilon) = \bigcup_{q \in P} q = P$$

また、$\hat{\delta'}$ の定義（5.2 節）より以下が成り立ちます。

$$\hat{\delta'}(P, \varepsilon) = P$$

よって、以下の等式が成り立ちます。

$$\bigcup_{q \in P} \hat{\delta}(q, \varepsilon) = \hat{\delta'}(P, \varepsilon)$$

帰納段階: 任意の記号列の集合 P'、長さ n の記号列 u について

$$\bigcup_{q \in P'} \hat{\delta}(q, u) = \hat{\delta'}(P', u)$$

が成り立つと仮定します。$\hat{\delta}$ の定義（6.2 節）より以下が成り立ちます。

$$\bigcup_{q \in P} \hat{\delta}(q, au) = \bigcup_{q \in P} \bigcup_{q' \in \delta(q,a)} \hat{\delta}(q', u)$$

また、$\hat{\delta'}$ の定義（5.2 節）、仮定、δ' の定義（変換規則 1）より以下が成り立ちます。

$$\begin{aligned} \hat{\delta'}(P, au) &= \hat{\delta'}(\delta'(P, a), u) \\ &= \bigcup_{q \in \delta'(P,a)} \hat{\delta}(q, u) \\ &= \bigcup_{q' \in P} \bigcup_{q \in \delta'(q',a)} \hat{\delta}(q, u) \end{aligned}$$

よって

$$\bigcup_{q \in P} \hat{\delta}(q, au) = \hat{\delta'}(P, au)$$

以上、基底段階と帰納段階の証明から全体が証明できました。

6.5　C 言語による実装

NFA の動作をコンピュータ上に実装する方法はいくつか知られています。

ひとつは、変換規則 1 を用いて NFA を DFA に等価変換し、その DFA を C プログラムに変換する方法です。この方法の利点はプログラムが最も高速に動作することです。逆に欠点は等価変換に手間が掛かる点です。変換規則 1 によれば、変換前の NFA の状態数を n として、変換後の DFA

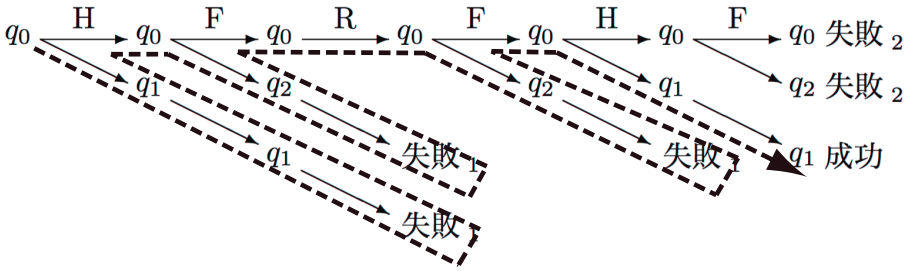


図 6.6: 後戻り付き深さ優先探索による NFA の動作のシミュレーション（その 1）

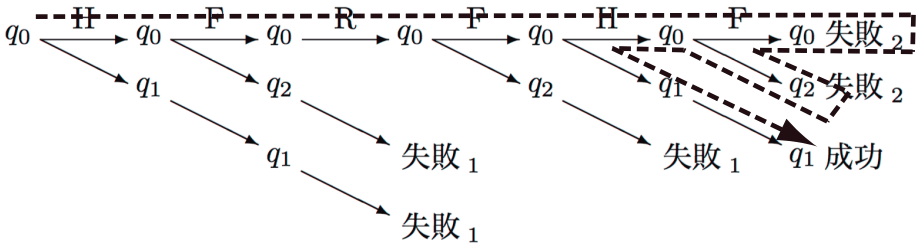


図 6.7: 後戻り付き深さ優先探索による NFA の動作のシミュレーション（その 2）

の状態数は最悪 2^n になります [5]。2^n 個の状態を作るために掛かる時間は、最悪 2^n に比例して爆発的に増えていきます。使用するメモリも同様に 2^n に比例します。4.4 節の検索の例の場合、正規表現を入力したならば直ちに検索を開始してほしいところですが、等価変換に時間が掛かると利用者はストレスを感じるかもしれません。また、使用メモリが CPU のキャッシュサイズを超えるようなことになれば実行速度の低下も起きます [6]。

2 番目の方法は、NFA を DFA に変換せず、NFA の動作を後戻り付き深さ優先探索でシミュレートする方法です。今、NFA の動作においてある状態から複数の状態遷移が可能であるとします。この状態を分岐点と呼びましょう。そのとき、複数の選択肢の中のひとつの状態遷移を選び、先へ進むことにします。そしてさらに状態遷移が進み、実行途中で受理に成功しないと判明ならば直前の分岐点まで後戻りし、先ほどとは別の状態遷移を選びます。もしその分岐点の全ての状態遷移を試した後ならば、さらにひとつ前の分岐点へ戻ります。このように、分岐点における選択・後戻りを繰り返して成功する路が存在するか否かを調べていく方法です。図 6.6 と図 6.7 に、6.4 節で取り上げた状態遷移例についてこの手法で探索する経路の例を示します。なお、入力記号列および状態数は有限ですから、この探索は必ず有限時間内に終了します。この方法の利点は、NFA から DFA

[5] 変換規則 1 自体は 2^n 個の状態を用意するような定式化になっていますが、図 6.5 辺りで見たように、実際には初期状態から到達可能な状態のみを扱えばよいので、2^n 個になることはまれです。

[6] CPU とメモリの間のデータ転送速度はコンピュータの実行速度に大きく影響します。メモリは 1 次キャッシュ、2 次キャッシュ、メインメモリ等に階層化されており、1 次キャッシュは高速小容量、メインメモリは低速大容量のメモリです。実感する機会は少ないかもしれませんが、コンピュータの動作は使用メモリが全て 1 次キャッシュに収まるときに最速です。

への等価変換を要しないことです。直ちに実行が開始できます。使用メモリサイズも元の NFA の状態数に比例する程度であり、極少です。欠点は、探索が試行錯誤を繰り返しながら進むため、無駄な探索がありうる点です。よって、一般には DFA へ変換する最初の方法が高速です。

3 番目の方法として、後戻りしない幅優先探索を応用し、NFA が通過する全ての状態を集合として管理しながら実行する方法です。これは 1 番目の方法と 2 番目の方法の折衷案とも考えられますが、詳細は省略します。

6.6　練習問題

1. アルファベット Σ が以下の (a)〜(c) であるときに、言語

$$\begin{aligned} L &= \{(aaa)^n \mid n \geq 0\} \\ &= \{\varepsilon, aaa, aaaaaa, aaaaaaaaa, ...\} \end{aligned}$$

を受理する、状態数が 3 以下であるような NFA（の状態遷移図）をそれぞれ示しなさい。

   (a) $\Sigma = \{a\}$

   (b) $\Sigma = \{a, b\}$

   (c) $\Sigma = \{a, b, c\}$

2. 上記 1. の三つのオートマトンを 5 項組で定義しなさい。

3. $\Sigma = \{a, b\}$ として、以下の言語を受理する NFA（の状態遷移図）をそれぞれ示しなさい。ただし、いずれの NFA の状態数も 3 以下とし、枝の数はできるだけ少なくすること。次にその NFA を DFA へ等価変換しなさい。

   (a) 先頭と末尾にのみ b が現れる記号列の全体（*bb*, *bab*, *baaaab* など）。

   (b) 先頭と末尾に b が現れる記号列の全体（先頭と末尾以外に b が現れてもよい。*bb*, *bab*, *bbb*, *babaab*, *bbbab* など）。

   (c) 記号列 *bb* を含む記号列の全体（b が 3 個以上現れてもよい。*bb*, *abbb*, *abaabbab* など）。

第7章　ε-遷移を含む非決定性有限状態オートマトン

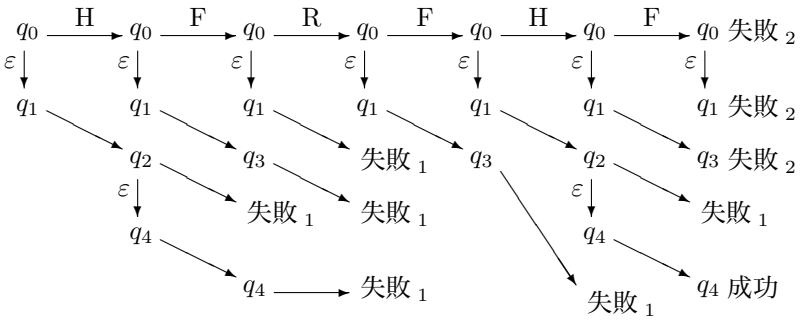
有限状態オートマトンをさらに機能拡張します。この拡張が正規表現と有限状態オートマトンの間の直接の架け橋となります。

7.1　ε-遷移

ε-遷移（ε-transition）とは、空列 ε による状態遷移のことです。前章までの状態遷移とは記号をひとつ読んで行うものでした。しかし、記号を読まなくとも — すなわち空列 ε によって — 状態遷移が可能であるように拡張したものが ε-遷移を含む非決定性有限状態オートマトン（nondeterministic finite state automaton with ε-transition）です。ε-遷移が可能な状態で ε-遷移を行っても構いませんし、行わなくても構いません。その意味で、このオートマトンは非決定性です。

以下では便宜上、このオートマトンを **ε-NFA** と略記します。

図 7.1 は、缶ジュースの自動販売機を ε-NFA を用いて記述したものです。初期状態 q_0 では記号を読むことなく状態 q_1 へ遷移可能です。初期状態 q_2 でも記号を読むことなく状態 q_3 へ遷移可能です。このオートマトンで記号列 HFRFHF を読んだときの遷移は以下の通りです。



この図の中の下向きの枝 $\varepsilon\downarrow$ は、ε-遷移を表します。たとえば初期状態 q_0 では先頭の記号 H を読む前に ε-遷移によって q_1 へ遷移可能です。先頭の H を読んだ直後にも状態 q_0、q_2 から ε-遷移可能です。よって q_0 から q_1 へ、そして q_2 から q_4 へ枝 $\varepsilon\downarrow$ が延びています。

結局、このオートマトンでは記号列 HFRFHF によって 10 種類の遷移の路が存在します。この中の 6 本の路はそれ以上記号を読み進めることができなくなって遷移に失敗した路です（失敗 $_1$ の

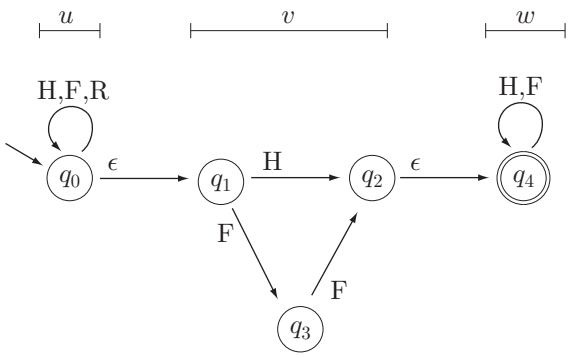


図 7.1: 缶ジュースの自動販売機の状態遷移図

路)。3 本の路は記号列を最後まで読んだものの最終状態には至らず、失敗した路です（失敗 $_2$ の路)。成功した路は唯一、以下のものだけです。

$$q_0 \xrightarrow{\mathrm{H}} q_0 \xrightarrow{\mathrm{F}} q_0 \xrightarrow{\mathrm{R}} q_0 \xrightarrow{\mathrm{F}} q_0 \xrightarrow{\varepsilon} q_1 \xrightarrow{\mathrm{H}} q_2 \xrightarrow{\varepsilon} q_4 \xrightarrow{\mathrm{F}} q_4$$

しかし成功路がひとつ存在するのですから、記号列は HFRFHF はこのオートマトンによって受理されると考えます。この受理の概念は前章の NFA と同じです。

NFA（図 6.2 参照）では、ジュースが出る記号列 $L_J = \{uvw \mid ...\}$ の各部分列 u、v、w に対応した部分を見ることができました。ε-NFA（図 7.1 参照）では、さらに鮮明にその部分を表現できます。というのも、ε-遷移は、オートマトンの部分と部分をつなぐ糊（のり）のような役目をするためです。実際、図 7.1 の ε-NFA の中の部分列 u、v、w に対応する部分（状態の集合）はそれぞれ $\{q_0\}$、$\{q_1, q_2, q_3\}$、$\{q_4\}$ です。ε-遷移は、q_0 と q_1 を結んで u と v を連結し、q_2 と q_4 を結んで v と w を連結しています。

7.2　5 項組による定義

ε-NFA は 5 項組 $M = (Q, \Sigma, \delta, q_0, F)$ で定義されます。ここに Q、Σ、q_0、F は NFA と同じです。すなわち、Q は状態の有限集合、Σ は入力アルファベット、q_0 $(\in Q)$ は初期状態、F $(\subseteq Q)$ は最終状態の集合です。遷移関数 δ は、NFA と異なり、状態と入力記号 または空列記号 ε から状態の集合への関数 $(\in Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q)$ です。

各状態 q について、q から ε-遷移だけで到達可能な状態の集合を ε-CLOSURE(q) $(\subseteq Q)$ と表します。形式的にはこの集合は以下の等式で定義されます。

$$\varepsilon\text{-CLOSURE}(q) = \{q\} \cup \{q'' \in Q \mid \varepsilon\text{-CLOSURE}(q) \ni q',\ \ \delta(q', \varepsilon) \ni q''\}$$

噛み砕いて言えば、ε-CLOSURE(q) とは少なくとも q 自身を含み、q' が ε-CLOSURE(q) に含まれ、q' から ε-遷移で状態 q'' へ遷移可能ならば、q'' も含むような集合です。

さて、前章、前々章と同様に、遷移関数 δ について以下のように定義される関数 $\hat{\delta}$ ：

$\boxed{Q \times \Sigma^* \to 2^Q}$ を作ります。

$$\begin{aligned}\hat{\delta}(q,\varepsilon) &= \varepsilon\text{-CLOSURE}(q),\\ \hat{\delta}(q,au) &= \bigcup_{q'\in\varepsilon\text{-CLOSURE}(q)}\left[\bigcup_{q''\in\delta(q',a)}\hat{\delta}(q'',u)\right]\end{aligned}$$

ただし $a \in \Sigma$、$u \in \Sigma^*$ とします。そして記号列 u について、$\hat{\delta}(q_0,u)\cap F \neq \emptyset$ であるとき、すなわち、初期状態 q_0 から記号列 u を読んで到達した状態のひとつ $(\in \hat{\delta}(q_0,u))$ が最終状態であるとき、u はオートマトン M によって受理されると言います。M の受理言語 $L(M)$ は M によって受理される記号列の全体です。すなわち

$$L(M) = \{u \in \Sigma^* \mid \hat{\delta}(q_0,u)\cap F \neq \emptyset\}$$

自動販売機 $M_{J,3}$ を 5 項組で表現すると以下の通りです。

$$M_{J,3} = (\{q_0,q_1,q_2,q_3,q_4\},\{\mathrm{H},\mathrm{F},\mathrm{R}\},\delta_{J,3},q_0,\{q_4\})$$

ここに

$$\begin{array}{llll}
\delta_{J,3}(q_0,\mathrm{H})=\{q_0\}, & \delta_{J,3}(q_0,\mathrm{F})=\{q_0\}, & \delta_{J,3}(q_0,\mathrm{R})=\{q_0\}, & \delta_{J,3}(q_0,\varepsilon)=\{q_1\},\\
\delta_{J,3}(q_1,\mathrm{H})=\{q_2\}, & \delta_{J,3}(q_1,\mathrm{F})=\{q_3\}, & \delta_{J,3}(q_1,\mathrm{R})=\emptyset, & \delta_{J,3}(q_1,\varepsilon)=\emptyset,\\
\delta_{J,3}(q_2,\mathrm{H})=\emptyset, & \delta_{J,3}(q_2,\mathrm{F})=\emptyset, & \delta_{J,3}(q_2,\mathrm{R})=\emptyset, & \delta_{J,3}(q_2,\varepsilon)=\{q_4\},\\
\delta_{J,3}(q_3,\mathrm{H})=\emptyset, & \delta_{J,3}(q_3,\mathrm{F})=\{q_2\}, & \delta_{J,3}(q_3,\mathrm{R})=\emptyset, & \delta_{J,3}(q_3,\varepsilon)=\emptyset,\\
\delta_{J,3}(q_4,\mathrm{H})=\{q_4\}, & \delta_{J,3}(q_4,\mathrm{F})=\{q_4\}, & \delta_{J,3}(q_4,\mathrm{R})=\emptyset, & \delta_{J,3}(q_4,\varepsilon)=\emptyset,
\end{array}$$

$M_{J,3}$ の受理言語は定義より

$$L(M_{J,3}) = \{u \in \{\mathrm{H},\mathrm{F},\mathrm{R}\}^* \mid \hat{\delta}_{J,3}(q_0,u)\cap\{q_4\} \neq \emptyset\}$$

となります。

様々な ε-NFA の例は、以下の節の議論を通して紹介します。

7.3　ε-NFA から NFA への等価変換

NFA から DFA への等価変換が可能である（6.4 節）のと同じく、ε-NFA から NFA への等価変換が可能です。

変換規則 2　$M = (Q, \Sigma, \delta, q_0, F)$ を任意の ε-NFA とします。そのとき、M と同じ受理言語を持つ ε-遷移を含まない NFA: $M' = (Q', \Sigma, \delta', q_0', F')$ を必ず作ることができます。すなわち記号列 u について、$u \in L(M)$ ならば $u \in L(M')$ であり、$u \notin L(M)$ ならば $u \notin L(M')$ であるような M' が存在し、そのような M' を実際に作ることができます。

まず、$Q' = Q$、$q_0' = q_0$ とします。つまり変換の前後で同じ状態を用います。初期状態も同じです。この点は変換規則 1 とは全く異なります（6.4 節参照）。次に M の遷移関数 $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$ を用いて、M' の遷移関数 $\delta' : Q \times \Sigma \to 2^Q$ を以下のように定義します。

$$\begin{aligned} \delta'(q, a) &= \hat{\delta}(q, a) \\ &= \bigcup_{q' \in \varepsilon\text{-CLOSURE}(q)} \left[\bigcup_{q'' \in \delta(q', a)} \varepsilon\text{-CLOSURE}(q'') \right] \end{aligned}$$

すなわち、M' において状態 q から a を読んで遷移する先は、M において q から ε-遷移で遷移し、さらに a を読んで遷移し、さらに ε-遷移で遷移する先を全て集めたものです。最終状態の集合 F' は以下の通りです。

$$F' = F \cup \{q \in Q' \mid \varepsilon\text{-CLOSURE}(q) \cap F \neq \emptyset\}$$

すなわち、ε-遷移だけで最終状態に到達可能な状態 q は新たに 最終状態 に追加します[1]。 □

変換規則 2 の内容を理解するための簡単な変換例をいくつか示します。ただし全ての例題について $\Sigma = \{a, b\}$ と仮定します。

例 1

図 7.2 (1) 左の ε-NFA M_{16} は初期状態 q_0 から最終状態 q_1 へ ε-遷移が可能です。よって、変換後のオートマトンでは、q_0 も最終状態に加わります。変換後のオートマトン M_{16}' には q_0 から q_1 への有向枝がないことに注意してください。よって変換後のオートマトンでは状態 q_1 は不要です。変換前後のオートマトンの受理集合は共に $\{\varepsilon\}$ です。

この例題を 5 項組を用いて議論してみましょう。M_{16} は $(\{q_0, q_1\}, \Sigma, \delta_{16}, \{q_1\})$ と表すことができます。ここに

$$\begin{aligned} &\delta_{16}(q_0, a) = \emptyset, \quad \delta_{16}(q_0, b) = \emptyset, \quad \delta_{16}(q_0, \varepsilon) = \{q_1\}, \\ &\delta_{16}(q_1, a) = \emptyset, \quad \delta_{16}(q_1, b) = \emptyset, \quad \delta_{16}(q_1, \varepsilon) = \emptyset. \end{aligned}$$

[1] 実際には、最終状態への追加は初期状態だけについて行えば十分です。よって

$$F' = \begin{cases} F \cup \{q_0\}, & \text{if } \varepsilon\text{-CLOSURE}(q_0) \cap F \neq \emptyset \\ F, & \text{otherwise} \end{cases}$$

と定義する教科書もあります。変換規則 2 の正当性の証明には上の関係式のみを用います。

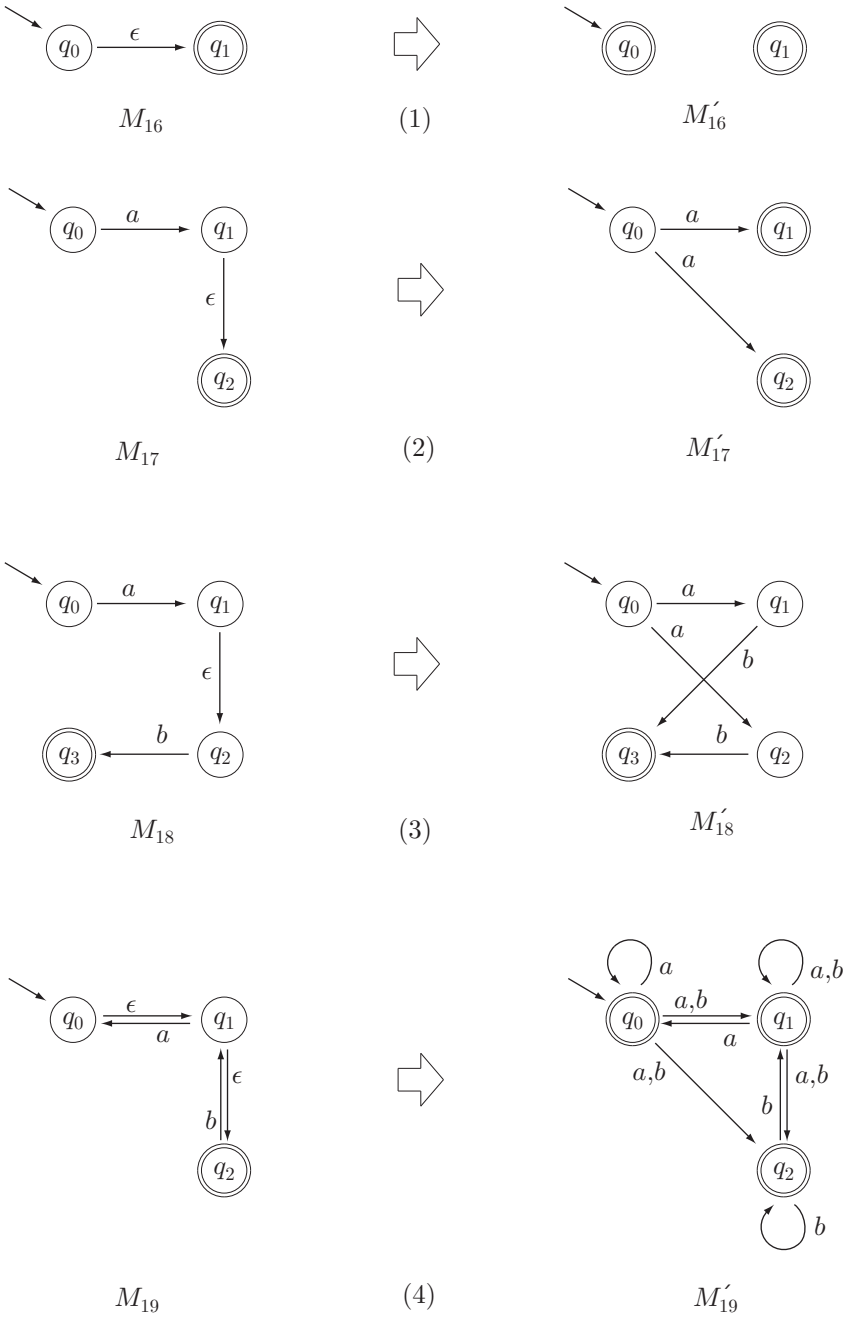


図 7.2: ε-NFA から NFA への変換

よって、変換後のオートマトン M'_{16} は $(\{q_0, q_1\}, \Sigma, \delta'_{16}, F'_{16})$ という形になります。ここに δ'_{16} と F'_{16} を具体的に決める必要があります。まず、δ'_{16} は以下の通りです。

$$\begin{aligned} \delta'_{16}(q_0, a) = \emptyset, \quad \delta'_{16}(q_0, b) = \emptyset, \\ \delta'_{16}(q_1, a) = \emptyset, \quad \delta'_{16}(q_1, b) = \emptyset. \end{aligned}$$

すなわち全ての関数値が空集合になります。次に最終状態の集合 F'_{16} は $\{q_0, q_1\}$ です。何故ならば、変換規則 2 に従えば、

$$\begin{aligned} F'_{16} &= F_{16} \cup \{q \in Q' \mid \varepsilon\text{-CLOSURE}(q) \cap F_{16} \neq \emptyset\} \\ &= \{q_1\} \cup \{q \in Q' \mid \varepsilon\text{-CLOSURE}(q) \cap \{q_1\} \neq \emptyset\} \end{aligned}$$

ですが、ε-CLOSURE$(q_0) = \{q_1\}$ であるため、F'_{16} は q_0 を含むことになるからです。

例 **2**

図 7.2 (2) 左の ε-NFA M_{17} では、$q_0 \xrightarrow{a} q_1 \xrightarrow{\varepsilon} q_2$ という遷移が可能です。よって、変換後の (2) 右の NFA M'_{17} では $q_0 \xrightarrow{a} q_2$ を持っています。また M_{17} では q_1 から ε-遷移で最終状態 q_2 へ到達できますから、M'_{17} では q_1 も最終状態です。変換前後のオートマトンの受理集合は $\{a\}$ です。

例 **3**

図 7.2 (3) の例題は (2) の例題を少し複雑化したものです。右のオートマトン M'_{18} が記号列 ab を読む遷移には $q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_3$ と $q_0 \xrightarrow{a} q_2 \xrightarrow{b} q_3$ の二種類があります。これに対して、左の変換前のオートマトン M_{18} が ab を読む遷移は $q_0 \xrightarrow{a} q_1 \xrightarrow{\varepsilon} q_2 \xrightarrow{b} q_3$ と一本道であって一見すると決定性のようにも見えます。しかし、ε-遷移には遷移する/しないの非決定性があるため、それが M'_{18} では二種類の遷移となって現れています。

例 **4**

図 7.2 (4) の例題は、(1) から (3) に比べると複雑です。状況を複雑にしている原因のひとつは、左のもとのオートマトン M_{19} に $q_0 \xrightarrow{\varepsilon} q_1 \xrightarrow{a} q_0$、$q_1 \xrightarrow{\varepsilon} q_2 \xrightarrow{b} q_1$ という ε-遷移を含む閉路がある点です。このような場合には、変換後のオートマトンに思いのほか多くの枝が加わります。この変換例では、オートマトン M_{19} の枝の数は 4 ですが、M'_{19} のそれは 12 です。変換後のオートマトンでは q_0、q_1 も最終状態である点にも注意が必要です。

人手で変換するときには落ち着いて丁寧に変換しないと、何本かの枝を忘れそうになります。忘れないためには、状態遷移を表で表してみるのも一案です。今、記号 a について 状態 q_i から状態

表 7.1: ε-NFA M_{19} の状態遷移図表（M_{19} において、状態 q_i から a または b を読んで状態 q_j へ遷移できるか否かのチェック）

a	q_0	q_1	q_2
q_0	○	○	○
q_1	○	○	○
q_2	×	×	×

b	q_0	q_1	q_2
q_0	×	○	○
q_1	×	○	○
q_2	×	○	○

q_j へ以下のような遷移が可能か否かを調べます。ここに $q_{i'}$、$q_{i''}$、$q_{j'}$、$q_{j''}$ などは任意の状態です。

$$
\begin{aligned}
q_i &\xrightarrow{a} q_j,\\
q_i &\xrightarrow{\varepsilon} q_{i'} \xrightarrow{a} q_j,\\
q_i &\xrightarrow{a} q_{j'} \xrightarrow{\varepsilon} q_j,\\
q_i &\xrightarrow{\varepsilon} q_{i'} \xrightarrow{\varepsilon} q_{i''} \xrightarrow{a} q_j,\\
q_i &\xrightarrow{\varepsilon} q_{i'} \xrightarrow{a} q_{j'} \xrightarrow{\varepsilon} q_j,\\
q_i &\xrightarrow{a} q_{j''} \xrightarrow{\varepsilon} q_{j'} \xrightarrow{\varepsilon} q_j,\\
&\vdots\\
q_i &\xrightarrow{\varepsilon} \ldots \xrightarrow{\varepsilon} q_{i'} \xrightarrow{a} q_{j'} \xrightarrow{\varepsilon} \ldots \xrightarrow{\varepsilon} q_j.
\end{aligned}
$$

もし上のような遷移が可能ならば、行および列が状態であるような表を作り、表の q_i の行と q_j の列の交点に○、さもなくば×を記します。このようにして作った表が表 7.1 の左側の表です。この表から q_2 からは a を読んでどこにも遷移できないことが分かります。同様に状態 q_i から b を読んで状態 q_j へ同様に遷移できるか否かを調べた表が表 7.1 の右側です。この表からどんな状態からも b を読んで q_0 へ遷移できないことが分かります。

考察　図 7.2 の各オートマトン（M_{16} と M'_{16} 以外）を 5 項組で表しなさい。また、変換の妥当性を、状態遷移図ではなく 5 項組を用いて示しなさい。

例 **5:** 缶ジュースの自動販売機

図 7.3 は、図 7.1 を ε-遷移を含まない NFA に変換したものです。

考察　図 7.3 の状態 q_0 から H を読んで q_1、q_2、q_4 へ遷移可能です。その理由をそれぞれ説明しなさい。

考察　図 7.3 を変換規則 1 に基づいて DFA に変換しなさい。

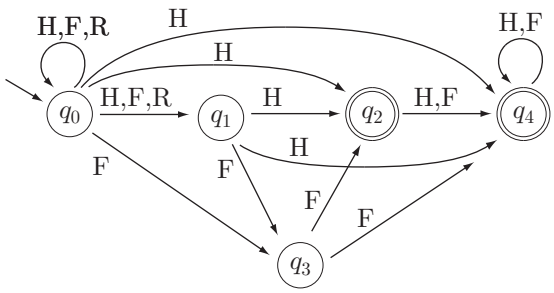


図 7.3: ε-NFA（図 7.1）から NFA への変換

変換規則 **2** の正当性

変換規則 2 の ε-NFA $M = (Q, \Sigma, \delta, q_0, F)$ とそれを変換した NFA $M' = (2^Q, \Sigma, \delta', q_0', F')$ の受理集合が等しいことを証明するには以下の二つを証明すれば十分です。

1. もし ε-CLOSURE$(q_0) \cap F \neq \emptyset$ が成り立つならば、そのときに限り、$q_0' \in F'$ が成り立つ。

2. 任意の記号列 u（$|u| \geq 1$）について、以下の等式が成り立つ。

$$\hat{\delta}(q_0, u) = \hat{\delta'}(q_0', u)$$

1. は変換規則の定義から明らかです。2. の証明には u の長さに関する数学的帰納法を用います。具体的な帰納法の用い方は前章の変換規則 1 の証明と類似していますから、省略します。

考察　余裕のある人は、任意の記号列 u（$|u| \geq 1$）について、$\hat{\delta}(q_0, u) = \hat{\delta'}(q_0', u)$ が成り立つことを自力で証明してみなさい。

7.4　C 言語による実装

前章同様に ε-NFA の動作をコンピュータ上に実装する方法は 2 つに大別されます。

ひとつは、ε-NFA を NFA に等価変換し、さらにその NFA を DFA に等価変換してその DFA を C プログラムに変換する方法です。利点は前章と同じく、変換後の C プログラムは高速に動作することです。欠点は、2 回の等価変換によってオートマトンの状態数、枝の数が増えるかもしれないことです。最悪、指数的に増大するかもしれません。

2 番目の方法は、ε-NFA を DFA に変換せず、ε-NFA の動作を後戻り付き深さ優先探索でシミュレートする方法です。基本的な動作は前章の方法と同様です。ただし、ε-遷移では記号を読まないことに注意が必要です。そのため、ε-NFA が図 7.4 のような構造を含む場合、$q_1 \to q_2 \to q_1 ...$ を

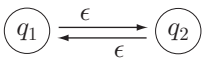


図 7.4: ε-NFA を深さ優先探索で直接実行する場合の問題点

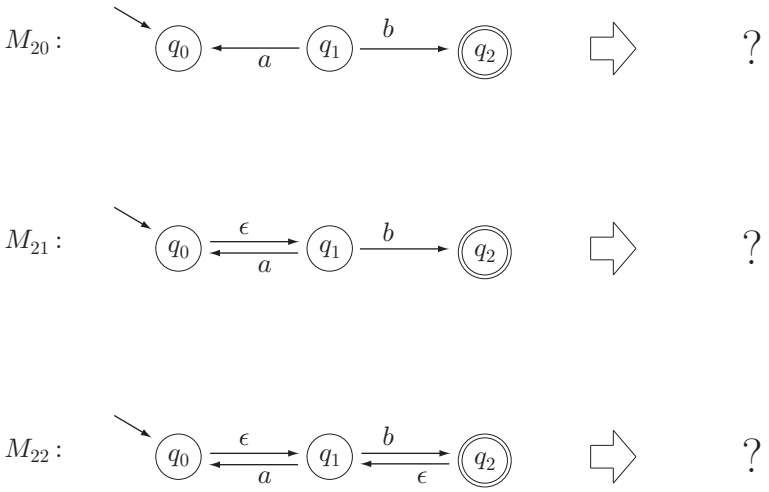


図 7.5: ε-NFA から NFA への変換の練習問題

記号を読むことなく無限に繰り返し、動作が止まらなくなってしまいます。記号を読まないで遷移可能な閉路を含まないことのチェックが必要です。含む場合にはそれを削除する変換が必要です。

　前章でも述べたように、文字列検索などに利用する場合には即時性が重要ですから、後者の方法がよいかもしれませんが、高速に動作させるには前者の方法が確実です。この講義テキストでは前者を想定した講義を行っています。

7.5　練習問題

1. 図 7.5 の 3 個の ε-NFA M_{20}, ..., M_{22} の受理言語を示しなさい。M_{20} から M_{22} へと、オートマトンの枝が 1 本ずつ増えていることを注意しなさい。

2. 図 7.5 の 3 個の ε-NFA をそれぞれ NFA へ変換しなさい。変換前のオートマトンの枝の違いによって、変換後のオートマトンがどのように違ってくるのか、注意しなさい。

3. 2. で求めた NFA を DFA へ変換しなさい。なお、$\Sigma = \{a, b\}$ とする。

第8章　正規表現（再）

図 4.2 を思い出しましょう。6 章に示した変換規則 1、7 章に示した変換規則 2、そしてこれから解説する変換規則 3 によって、正規表現 から DFA への機械的な等価変換が可能になります。

8.1　正規表現から ε-NFA への等価変換

任意の正規表現から、その正規表現の言語を受理言語とする ε-NFA を作ることができます。

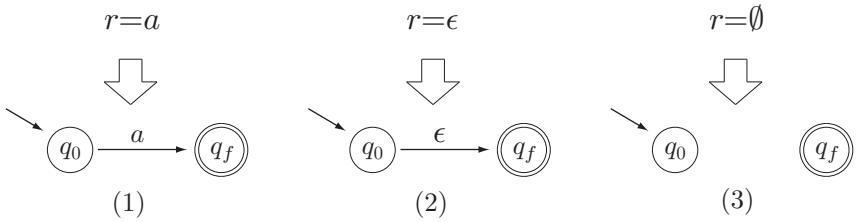
変換規則 3　r を Σ の上の任意の正規表現とします。そのとき、$L(r)$ を受理言語とする ε-NFA: $M = (Q, \Sigma, \delta, q_0, F)$ を必ず作ることができます。すなわち、記号列 u について、$u \in L(r)$ ならば $u \in L(M)$ であり、$u \notin L(r)$ ならば $u \notin L(M)$ であるような M が存在し、そのような M を実際に作ることができます。

具体的な変換方法として、正規表現 r が、$r = a$ $(a \in \Sigma)$、$r = \varepsilon$、$r = \emptyset$ の場合の変換方法を初めに示します。次に r_1、r_2 は正規表現であって、それぞれに対応する ε-NFA: M_1、M_2 が既に構成されていると仮定します。このとき、$r = r_1 \mid r_2$、$r = r_1 r_2$、$r = r_1^*$、$r = (r_1)$、$r = r_1^+$ の形をしているときのオートマトンの構成法を示します。ただし、議論を簡単にするため、この変換規則によって構成される全てのオートマトンでは、

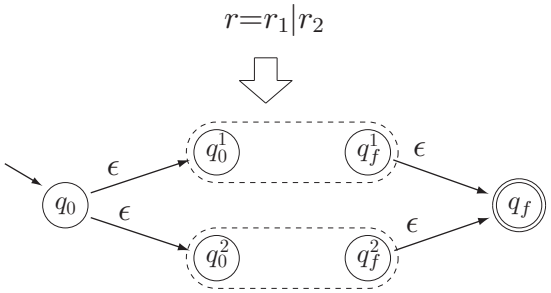
1. 最終状態はひとつしか存在せず、
2. 最終状態を始点とする枝はなく、
3. 初期状態と最終状態は異なる

ように構成すると約束します。

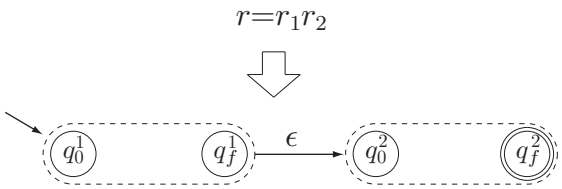
1. まず、$r = a$ $(a \in \Sigma)$ ならば、対応する ε-NFA を図 8.1(1) のように作ります。この ε-NFA を 5 項組 M_a で示すと、$M_a = (\{q_0, q_f\}, \Sigma, \delta_a, q_0, \{q_f\})$ です。ただし $\delta_a(q_0, a) = \{q_f\}$ とし、それ以外の δ_a の値は全て $\emptyset$ です。
2. もし $r = \varepsilon$ ならば、対応する ε-NFA を図 8.1(2) のように作ります。この ε-NFA を 5 項組 M_ε で示すと、$M_\varepsilon = (\{q_0, q_f\}, \Sigma, \delta_\varepsilon, q_0, \{q_f\})$ です。ただし $\delta_\varepsilon(q_0, \varepsilon) = \{q_f\}$ とし、それ以外の δ_ε の値は全て $\emptyset$ です。



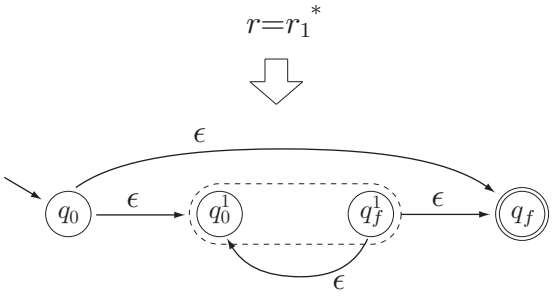
(1)　(2)　(3)



(4)



(5)



(6)

図 8.1: 正規表現から ε-NFA への変換

3. もし $r = \emptyset$ ならば、対応する ε-NFA を図 8.1(3) のように作ります。この ε-NFA を 5 項組 $M_\emptyset$ で示すと、$M_\emptyset = (\{q_0, q_f\}, \Sigma, \delta_\emptyset, q_0, \{q_f\})$ です。ただし遷移関数 $\delta_\emptyset$ の値は全て $\emptyset$ です。

次に、r_1、r_2 が正規表現であって、それぞれに対応する ε-NFA: M_1、M_2 が既に構成されていると仮定します。ここに 5 項組を $M_1 = (Q_1, \Sigma, \delta_1, q_0^1, \{q_f^1\})$、$M_2 = (Q_2, \Sigma, \delta_2, q_0^2, \{q_f^2\})$ と仮定しておきます。

1. もし $r = r_1|r_2$ ならば、対応する ε-NFA を図 8.1(4) のように作ります。この ε-NFA を 5 項組 M_A で示す[1] と、$M_A = (Q_1 \cup Q_2 \cup \{q_0, q_f\}, \Sigma, \delta_A, q_0, \{q_f\})$ です。ただし $\delta_A(q_0, \varepsilon) = \{q_0^1, q_0^2\}$、$\delta_A(q_f^1, \varepsilon) = \{q_f\}$、$\delta_A(q_f^2, \varepsilon) = \{q_f\}$、任意の記号 $a \in \Sigma$ について $\delta_A(q_0, a) = \delta_A(q_f^1, a) = \delta_A(q_f^2, a) = \emptyset$ です。さらに M_1 の任意の状態 $q \in Q_1$、および任意の記号 $a \in \Sigma \cup \{\varepsilon\}$ について $\delta_A(q, a) = \delta_1(q, a)$、$M_2$ の任意の状態 $q \in Q_2$、および任意の記号 $a \in \Sigma \cup \{\varepsilon\}$ について $\delta_A(q, a) = \delta_2(q, a)$ とします。

2. もし $r = r_1 r_2$ ならば、対応する ε-NFA を図 8.1(5) のように作ります。この ε-NFA を 5 項組 M_C で示す[2] と、$M_C = (Q_1 \cup Q_2, \Sigma, \delta_C, q_0^1, \{q_f^2\})$ です。ただし $\delta_C(q_f^1, \varepsilon) = \{q_0^2\}$ です。さらに任意の状態 $q \in Q_1 - \{q_f^1\}$、および任意の記号 $a \in \Sigma \cup \{\varepsilon\}$ について $\delta_C(q, a) = \delta_1(q, a)$、任意の状態 $q \in Q_2$、および任意の記号 $a \in \Sigma \cup \{\varepsilon\}$ について $\delta_C(q, a) = \delta_2(q, a)$ とします。

3. もし $r = r_1^*$ ならば、対応する ε-NFA を図 8.1(6) のように作ります。この ε-NFA を 5 項組 M_K で示す[3] と、$M_K = (Q_1 \cup \{q_0, q_f\}, \Sigma, \delta_K, q_0, \{q_f\})$ です。ただし $\delta_K(q_0, \varepsilon) = \{q_0^1, q_f\}$、$\delta_K(q_f^1, \varepsilon) = \{q_0^1, q_f\}$ です。さらに任意の状態 $q \in Q_1 - \{q_f^1\}$、および任意の記号 $a \in \Sigma \cup \{\varepsilon\}$ について $\delta_K(q, a) = \delta_1(q, a)$ とします。

4. もし $r = (r_1)$ ならば、対応する ε-NFA は M_1 と同じです。

5. もし $r = r_1^+$ ならば、これを $r = r_1 r_1^*$ または $r = r_1^* r_1$ に変形し、変換を行います。　□

例として正規表現 $(\mathrm{H}|\mathrm{F}|\mathrm{R})^*$ を ε-NFA に変換することを考えましょう。まず、三つの正規表現 H、F、R に対応する ε-NFA はそれぞれ図 8.2 (1)、(2)、(3) です。ただし煩雑ですから、各状態にラベル (q_0、q_f など）は付けないことにします。次に、正規表現 H|F に対応する ε-NFA は、図 8.2(1)、(2) の二つの ε-NFA をもとに図 8.1(4) のように組み上げて、図 8.2(4) です。正規表現 H|F|R は (H|F)|R と見なせますから、これに対応する ε-NFA は、図 8.2(4)、(3) のふたつの ε-NFA を図 8.1(4) のように組み上げて、図 8.2(5) です。そして正規表現 $(\mathrm{H}|\mathrm{F}|\mathrm{R})^*$ に対応する ε-NFA は、図 8.2(5) の ε-NFA をもとに図 8.1(6) のように組み上げて、図 8.2(6) です。これが求めるオートマト

[1]添字 A は Alternation（選択）に由来して付けました。
[2]添字 C は Concatenation（連接）に由来して付けました。
[3]添字 K は Kleene（クリーネ:人名）に由来して付けました。

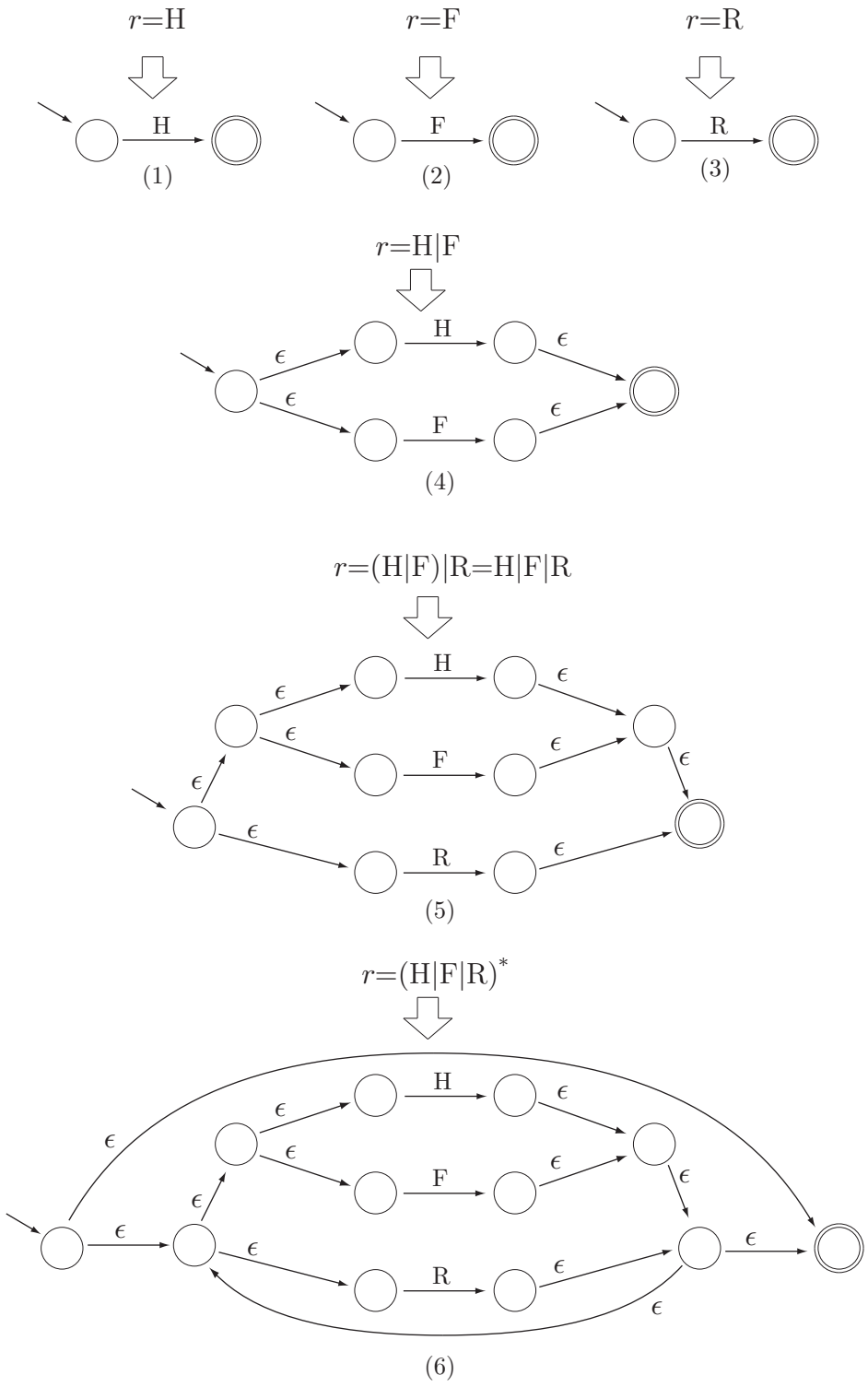


図 8.2: 正規表現 (H|F|R)* の ε-NFA への変換

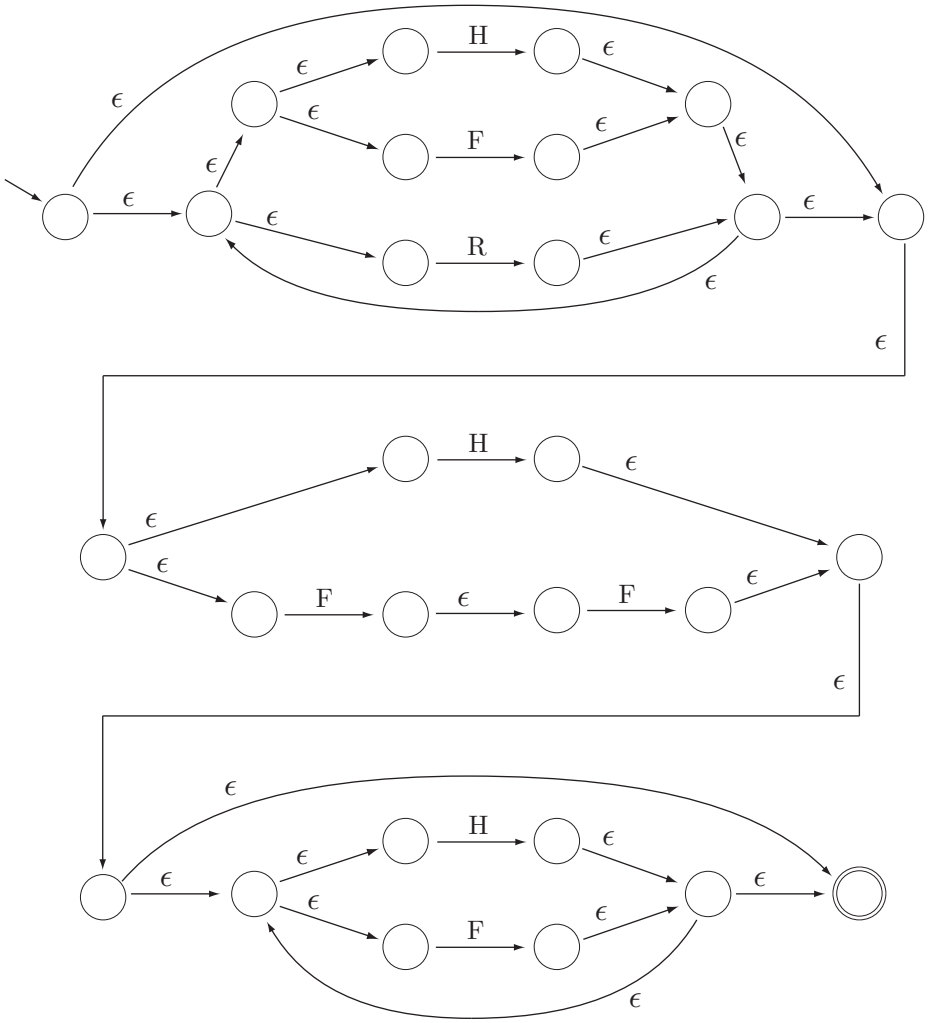


図 8.3: 正規表現 (H|F|R)*(H|FF)(H|F)* の ε-NFA への変換

ンです。オートマトンを構築していく操作が変換規則 3 に基づいた全く機械的なものであることに注意してください。

　同様の機械的な操作を繰り返すことで、正規表現 (H|F|R)*(H|FF)(H|F)* に対応する ε-NFA を図 8.3 のように得ます。

　ところで、正規表現 (H|F|R)*(H|FF)(H|F)* は缶ジュースが出てくる記号列の集合を表現したものでした。7 章では、そのような記号列の集合を表す ε-NFA として図 7.1 を示しました。このオートマトンを変換規則 3 に基づいて求めた図 8.3 と比較すると、その形状は相当に異なります。一言で言えば、図 8.3 の方が多くの ε-遷移を含み、冗長です。6 章の変換規則 1、7 章の変換規則 2 もそうでしたが、自動変換して導出されたオートマトンは、人手で求めたオートマトンよりも冗長です。冗長ですが、しかし受理言語は同じです。

また、図 8.3 には多くの ε-遷移が使用されていますが、それらがオートマトンの部品と部品をつなぐ役割を果たしている事に気づくでしょう。ε-遷移は、正規表現から有限状態オートマトンへの変換を容易にするために導入されました [4]。

変換規則 1、変換規則 2 がそうであったように、変換規則 3 が等価変換であることも証明せねばなりません。それにはやはり数学的帰納法 [5] を用います。この証明は図 8.1 からほとんど明らかであるため、このテキストでは省略します。

8.2　DFA、NFA、ε-NFA、正規表現の相互変換

5 章から始まった有限状態オートマトンと正規表現の話は、この章でひとつの区切りとなります。ここまでの流れをまとめたものが、4 章に示した図 4.2 です。

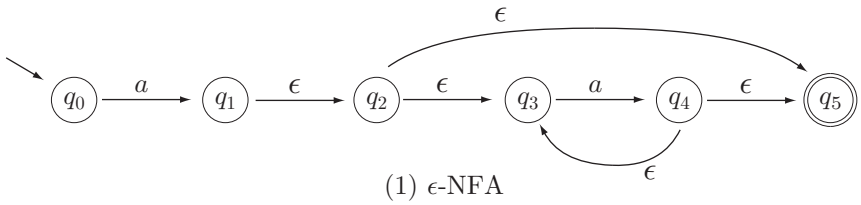
ここまで決定性有限状態オートマトン（DFA）、非決定性有限状態オートマトン（NFA）、ε-遷移を含む非決定性有限状態オートマトン（ε-NFA）、そして正規表現を学んできました。そしてこれらの間には等価変換が存在し、正規表現から ε-NFA へ、ε-NFA から NFA へ、NFA から DFA へと変換可能です。この授業では述べませんが、DFA は、その DFA の受理言語を表す正規表現に変換できます。そうすると、正規表現、ε-NFA、NFA、DFA は相互に等価変換可能になります。言い換えれば、ひとつの表現形式で表された言語（記号列の集合）は必ず他の三つの表現形式でも表現可能なのです。その意味から、これらは同じ言語クラスであると言い、そのクラスを正規言語と呼んでいます。

正規言語の理論の実用上の重要な点は、任意の正規表現が DFA に変換できることです。正規表現はコンパクトであり、人間にとって理解しやすい表現形式です。それに対して、DFA は入力記号列が機械によって認識される様子をそのまま表していますから、プログラム化しやすいという特徴を持ちます。そこで、特定の言語（記号列の集合）を表現するためには正規表現を用い、その記号列を認識するプログラムを作るためには、正規表現を等価な DFA に変換し、さらにその DFA をプログラム化するという応用が考えられます。実際、この方法はプログラミング言語のコンパイラを作る上で利用されています。

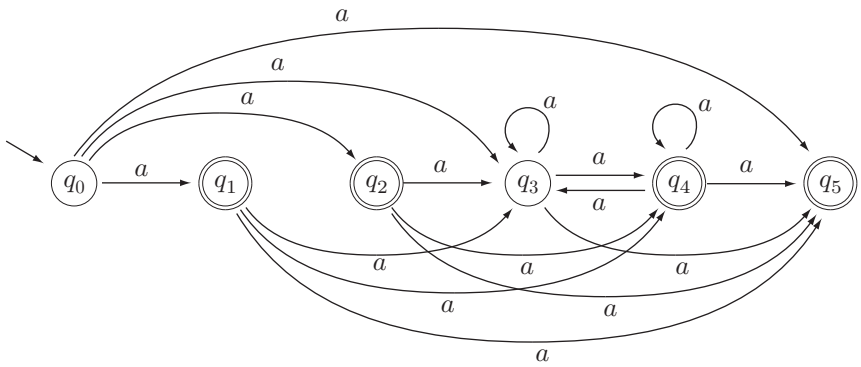
ここで正規表現から DFA への変換の例を示しましょう。$\Sigma = \{a\}$ とするとき、正規表現 aa^* は、変換規則 3 によって図 8.4 (1) のような ε-NFA へ変換されます。次にそれを変換規則 2 に従って NFA へ変換すると、図 8.4 (2) のようになります。さらにそれを変換規則 1 に従って DFA へ変換すると、図 8.4 (3) のようになります。この DFA を 5.4 節の方法で C プログラムに変換すれば、記号列 a^n $(n \geq 1)$ を認識するプログラムが完成する訳です。この変換過程は全く機械的であ

[4] 正規表現から ε-遷移のない非決定性有限状態オートマトン、あるいは決定性有限状態オートマトンへダイレクトに変換することを考えてみてください。その変換規則は相当に複雑になるはずです。

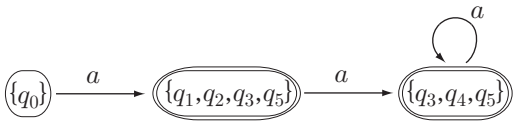
[5] この場合、正規表現に含まれる演算子数に基づく帰納法を用いればよいでしょう。



(1) ϵ-NFA



(2) NFA



(3) DFA

図 8.4: 正規表現 aa^* の ε-NFA、NFA、DFA への変換

るため、実際にはわざわざ人手で変換する必要はありません。コンピュータを用いて自動変換できます。その変換を行うために作られたソフトウェアが 10 章で解説する `lex` です。

ところで、変換によって得られる DFA は一般に冗長です。たとえば図 8.4 (3) の DFA は 3 個の状態を持ちますが、それと等価な DFA は 2 個の状態だけで作ることができます。次章で述べるように、任意の DFA はそれと等価で、かつ状態数が最少の DFA に変換できますから、通常は C プログラムを導出する前にはあらかじめ状態数を最少化し、冗長性を取り除く処理を行います。その方法を次章で解説します。

8.3　練習問題

1. 4.3 節の例 1 から例 11 について、それを順に ε-NFA、NFA、DFA に変換しなさい。導出された DFA を 5.3 節の DFA と比較しなさい。

第9章　決定性有限状態オートマトンの最小化

しかし変換によって得られるオートマトンは多くの場合に冗長です。受理集合が同じであるようなオートマトンならば、状態数の少ない方がCプログラムに実装した際により小さなメモリで動作可能です。使用するメモリが少ない場合にはプログラムの高速実行も期待できます[1]。

9.1　最小化アルゴリズム

DFA $M = (Q, \Sigma, \delta, q_0, F)$ に含まれる二つの状態 p, q $(\in Q)$ について、以下の性質が成り立つとき、「p と q は **同値** （equivalent）である」と言い、$p \equiv q$ と表します。

- 任意の記号列 u $(\in \Sigma^*)$ について、$\delta(p, u)$ が最終状態であるならば、そのときに限り、$\delta(q, u)$ も最終状態である。

もし二つの状態が同値であるならば、その二つの状態を通過した後の受理/不受理の判定は全く同一です。よって、その二つの状態をひとつに **合併** してもよいと考えられます。これが最小化のアイディアです。

ところで、ここの議論について言えば、実は同値性（$p \equiv q$）の判定を行うよりもむしろ逆に非同値性（$p \not\equiv q$）の判定を行った方が分かりやすく簡単です。「p と q が同値でない」とは、上の性質の逆、すなわち以下の性質を満たすことです。

- ある記号列 u $(\in \Sigma^*)$ について、$\delta(p, u)$ と $\delta(q, u)$ のいずれか一方が最終状態、他方が最終状態でない。

これは以下のような条件に言い換えることができます。

1. $p \in F$ かつ $q \notin F$、または $p \notin F$ かつ $q \in F$ ならば、$p \not\equiv q$.
2. ある記号 a $(\in \Sigma)$ について、$\delta(p, a) \not\equiv \delta(q, a)$ ならば、$p \not\equiv q$.

[1] 使用メモリが少ない場合、キャッシュミスが減ることが期待できます。またレジスタ上に常駐可能なデータの割合が増え、メモリアクセスが減ることも期待できます。

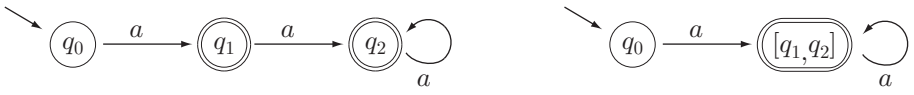


図 9.1: 最小化の例題 DFA（その 1）

そこで、上の判定を繰り返し、任意の二つの状態について非同値性を次々と追加していきます。そして、二つの状態 p と q について あらゆる場合について非同値であると判定できない ならば、p と q は同値であると判定します [2]。

そして同値であるものどうしは合併し、オートマトンの状態数を減らします。状態 p と q の合併後の状態を $[p,q]$ と表しましょう。具体的に状態 p と q の合併とは、p、q に入る枝は全て $[p,q]$ に入り、p、q から出る枝は全て $[p,q]$ から出るように設定し、p と q は状態遷移図から削除します。

9.2　例題

いくつかの例題で検証しましょう。

図 9.1 (1) の DFA は、前章、図 8.4 (3) の DFA の再掲です。この DFA の三つの状態の中の任意の二つの組み合わせに対して、上に示した非同値性の条件 1. を適用すると、直ちに

$$q_0 \not\equiv q_1,\quad q_0 \not\equiv q_2$$

が成り立ちます。次に q_1 と q_2 に条件 2. の適用を試みますが、$\delta(q_1,a)$ $(=q_2)$ と $\delta(q_2,a)$ $(=q_2)$ は $\delta(q_1,a) \not\equiv \delta(q_2,a)$ を満たさないため、$q_1 \not\equiv q_2$ は成り立ちません。このことから、逆に $q_1 \equiv q_2$ が成り立ちます。そこで、状態 q_1 と状態 q_2 をひとつの状態 $[q_1,q_2]$ に合併すると、図 9.1(2) の DFA が求められます。図 9.1 (1) と図 9.1 (2) のオートマトンが等価であることは明らかです。

図 9.2 は最小化の例題として様々な教科書で取り上げられている有名な DFA です。このオートマトンには状態数が 8 個あり、状態の対の数は全部で ${}_8C_2$ $(= 8!/6!2! = 28)$ 通りもあるため、非同値性を場当たり的に調べることは効果的ではありません。そこでまず、全ての状態の対の組み合わせを表に表します。表 9.1 はその組み合わせです。

この表の組み合わせの中で、条件 1. を満たすものに $\times$ を付けたものが表 9.2 です。図 9.2 の DFA の最終状態は q_2 だけですから、q_2 とその他の最終状態でない状態 q_i（$i \neq 2$）の間に $q_2 \not\equiv q_i$ が成り立ちます。そこで、これらの間に $\times$ を付けました。

次に、条件 2. を用いて、表 9.2 に $\times$ を追加していきます。条件 2. は、ある記号 a について、もし $\delta(q,a) = p$、$\delta(q',a) = p'$ であり、かつ $p \not\equiv p'$ であるならば $q \not\equiv q'$ が成り立つことを言ってい

[2] 詳細は述べませんが、非同値性をこれ以上追加できない状態において、p と q が非同値ではないならば、p と q は同値であることが証明できます。

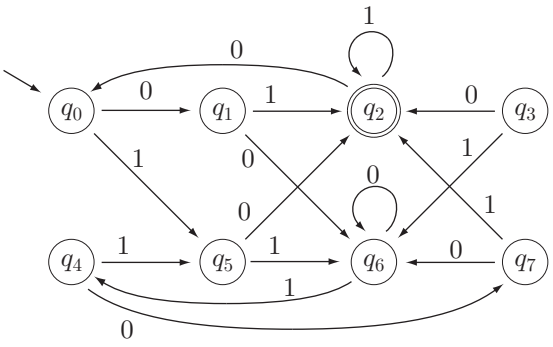


図 9.2: 最小化の例題 DFA（その 2）

表 9.1: 非同値表（初期設定）

q_1							
q_2							
q_3							
q_4							
q_5							
q_6							
q_7							
	q_0	q_1	q_2	q_3	q_4	q_5	q_6

ます。この関係を

$$q \not\equiv q' \quad \Leftarrow \quad \delta(q,a) = p \not\equiv p' = \delta(q',a)$$

と表すことにしましょう。そうすると、表 9.2 の関係から以下の非同値性が帰結されます。

$$q_0 \not\equiv q_1 \quad \Leftarrow \quad \delta(q_0,1) = q_5 \not\equiv q_2 = \delta(q_1,1)$$

$$q_0 \not\equiv q_3 \quad \Leftarrow \quad \delta(q_0,0) = q_0 \not\equiv q_3 = \delta(q_3,0)$$

$$q_0 \not\equiv q_5 \quad \Leftarrow \quad \delta(q_0,0) = q_1 \not\equiv q_2 = \delta(q_5,0)$$

$$q_0 \not\equiv q_7 \quad \Leftarrow \quad \delta(q_0,1) = q_5 \not\equiv q_2 = \delta(q_7,1)$$

$$q_1 \not\equiv q_3 \quad \Leftarrow \quad \begin{array}{l} \delta(q_1,0) = q_2 \not\equiv q_6 = \delta(q_3,0) \\ \delta(q_1,1) = q_6 \not\equiv q_2 = \delta(q_3,1) \end{array}$$

$$q_1 \not\equiv q_4 \quad \Leftarrow \quad \delta(q_1,1) = q_2 \not\equiv q_5 = \delta(q_4,1)$$

$$q_1 \not\equiv q_5 \quad \Leftarrow \quad \delta(q_1,1) = q_2 \not\equiv q_6 = \delta(q_5,1)$$

$$q_1 \not\equiv q_6 \quad \Leftarrow \quad \delta(q_1,1) = q_2 \not\equiv q_4 = \delta(q_6,1)$$

$$q_3 \not\equiv q_4 \quad \Leftarrow \quad \delta(q_3,0) = q_2 \not\equiv q_7 = \delta(q_4,0)$$

表 9.2: 非同値表（条件 1 チェック後）

q_1							
q_2	×	×					
q_3			×				
q_4			×				
q_5			×				
q_6			×				
q_7			×				
	q_0	q_1	q_2	q_3	q_4	q_5	q_6

表 9.3: 非同値表（条件 2、1 回目チェック後）

q_1	×						
q_2	×	×					
q_3	×	×	×				
q_4		×	×	×			
q_5	×	×	×		×		
q_6		×	×	×		×	
q_7	×		×	×	×	×	×
	q_0	q_1	q_2	q_3	q_4	q_5	q_6

$$q_3 \not\equiv q_6 \quad \Leftarrow \quad \delta(q_3,0) = q_2 \not\equiv q_6 = \delta(q_6,0)$$

$$q_3 \not\equiv q_7 \quad \Leftarrow \quad \begin{array}{l} \delta(q_3,0) = q_6 \not\equiv q_2 = \delta(q_7,0) \\ \delta(q_3,1) = q_2 \not\equiv q_6 = \delta(q_7,1) \end{array}$$

$$q_4 \not\equiv q_5 \quad \Leftarrow \quad \delta(q_4,0) = q_7 \not\equiv q_2 = \delta(q_5,0)$$

$$q_4 \not\equiv q_7 \quad \Leftarrow \quad \delta(q_4,1) = q_5 \not\equiv q_2 = \delta(q_7,1)$$

$$q_5 \not\equiv q_6 \quad \Leftarrow \quad \delta(q_5,0) = q_2 \not\equiv q_6 = \delta(q_6,0)$$

$$q_5 \not\equiv q_7 \quad \Leftarrow \quad \begin{array}{l} \delta(q_5,0) = q_2 \not\equiv q_6 = \delta(q_7,0) \\ \delta(q_5,1) = q_6 \not\equiv q_2 = \delta(q_7,1) \end{array}$$

$$q_6 \not\equiv q_7 \quad \Leftarrow \quad \delta(q_6,1) = q_4 \not\equiv q_2 = \delta(q_7,1)$$

これらを表に書き加えると、表 9.3 の通りです。

再度、条件 2. を表 9.3 の関係に適用すると、以下の非同値性が帰結されます。

$$q_0 \not\equiv q_6 \quad \Leftarrow \quad \begin{array}{l} \delta(q_0,0) = q_1 \not\equiv q_6 = \delta(q_6,0) \\ \delta(q_0,1) = q_5 \not\equiv q_4 = \delta(q_6,1) \end{array}$$

$$q_4 \not\equiv q_6 \quad \Leftarrow \quad \begin{array}{l} \delta(q_4,0) = q_7 \not\equiv q_6 = \delta(q_6,0) \\ \delta(q_4,1) = q_5 \not\equiv q_4 = \delta(q_6,1) \end{array}$$

表 9.4: 非同値表（条件 2、2 回目チェック後）

q_1	×						
q_2	×	×					
q_3	×	×	×				
q_4		×	×	×			
q_5	×	×	×		×		
q_6	×	×	×	×	×	×	
q_7	×		×	×	×	×	×
	q_0	q_1	q_2	q_3	q_4	q_5	q_6

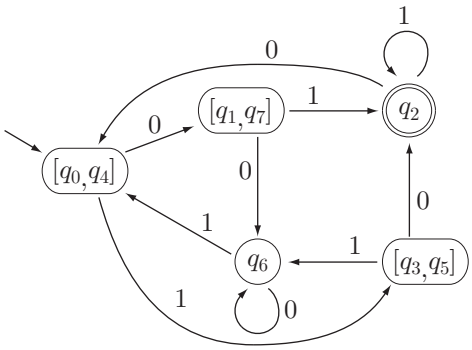


図 9.3: 最小化の例題 DFA（その 2、合併後）

よって、これらを表に書き加えたものが表 9.4 です。再度、条件 2. を表 9.4 の関係に適用しても、もはや新しい非同値性は検出されません [3]。そこで、表 9.4 の中の × の付いていない 3 箇所から $q_0 \equiv q_6$、$q_1 \equiv q_7$、$q_3 \equiv q_5$ を結論します。

同値性が分かったならば、同値な状態どうしを合併します。図 9.3 は、図 9.2 の中の状態 q_0 と q_4、q_1 と q_7、q_3 と q_5 を合併して得られたオートマトンです。なお、二つの状態 p と q の合併後の状態を $[p, q]$ と表わしました。合併後は状態数が 3 個減りました。

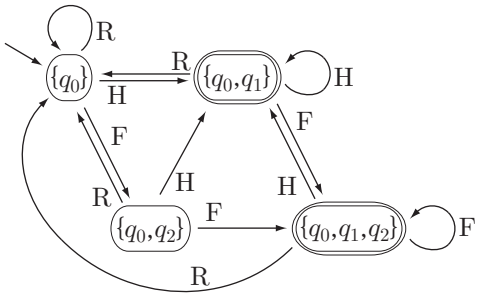
例題をもうひとつ示します。

図 9.4 のオートマトンは、図 6.5 (5) の DFA です。NFA から等価変換されたものであり、冗長です。この DFA を最小化しましょう。

まず、全ての状態の対の組み合わせを表に表します。表 9.5 はその組み合わせです。

この表の組み合わせの中で、条件 1. を満たすものに × を付けたものが表 9.2 です。図 9.2 の DFA の最終状態は q_2 だけですから、q_2 とその他の最終状態でない状態 q_i（$i \neq 2$）の間に $q_2 \not\equiv q_i$ が成り立ちます。そこで、これらの間に × を付けました。

[3] 変化が起きなくなるまで一定の処理を繰り返す方法はよくある方法です。



DFA $M'_{J,2}$

図 9.4: 最小化の例題 DFA（その 3）

表 9.5: 非同値表（初期設定）

$\{q_0, q_1\}$			
$\{q_0, q_2\}$			
$\{q_0, q_1, q2\}$			
	$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

次に、条件 2. を用いて、表 9.6 に $\times$ を追加していきます。そうすると、表 9.6 の関係から以下の非同値性のみが帰結されます。

$$\{q_0\} \not\equiv \{q_0, q_2\} \quad \Leftarrow \quad \delta(\{q_0\}, F) = \{q_0, q_2\} \not\equiv \{q_0, q_1, q_2\} = \delta(\{q_0, q_2\}, F)$$

よって、これらを表に書き加えると、表 9.7 の通りです。再度、条件 2. を表 9.7 の関係に適用しても、もはや新しい非同値性は検出されません。そこで、表 9.7 の中の $\times$ の付いていないエントリから、$\{q_0, q_1\} \equiv \{q_0, q_1, q_2\}$ が成り立つと見なします。そこで、それら状態を合併すると、図 9.5 の DFA を得ます。合併後は状態数が 1 個減りました。これは 5 章の図 5.1 の DFA に他なりません。

9.2.1　最小化アルゴリズムの正当性

以上の方法によって、必ず最小の DFA を導出できます。しかも、ある受理言語 L を持つ DFA は必ず一意な形に最小化されます。

なぜならば、まず、非同値表を用いて $p \not\equiv q$ と判別 できなかった ならば、$p \equiv q$ が成り立つことを証明します。これは背理法を用いて証明できます。次に、ある DFA M_1 を最小化した結果が M'_1 であるとき、M_1 と M'_1 の受理言語は等しく、かつ M'_1 の状態数は最小であることが証明できます。これも背理法で証明できます。最後に、同じ受理言語 L を持つ 2 つの DFA M_1、M_2 を最小化した結果をそれぞれ M'_1、M'_2 とするならば、M'_1 と M'_2 は同じ形であることが背理法で証明できます。

表 9.6: 非同値表（条件 1 チェック後）

$\{q_0, q_1\}$	×		
$\{q_0, q_2\}$		×	
$\{q_0, q_1, q2\}$	×		×
	$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

表 9.7: 非同値表（条件 2、1 回目チェック後）

$\{q_0, q_1\}$	×		
$\{q_0, q_2\}$	×	×	
$\{q_0, q_1, q2\}$	×		×
	$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

余力のある人は具体的な証明に挑戦してみてください。

9.2.2　正規言語の同値性判定問題

（具体的な証明は載せていませんが）前節の結果「ある受理言語 L を持つ DFA は必ず一意な形に最小化される」ことから、正規言語に関する以下のような興味深い性質が導出できます。

1. 任意の 2 つの正規表現 r_1、r_2 が等価であるか否かは 有限時間内 に判定できる。

   何故ならば、それぞれの正規表現を変換規則 1〜3、および最小化アルゴリズムによって DFA M_1、M_2 へ変換したとしましょう。もし $L(r_1) = L(r_2)$ が成り立つならば、M_1 と M_2 は同じ形になり、さもなくば異なる形になるからです。

2. 任意の正規表現 r について、$L(r)$ が 空集合 か否かは有限時間内に判定できる。

   何故ならば、上記 1 の性質から、2 つの正規表現 r と $\emptyset$ が等価であるか否かを判定すればいいからです。

3. 任意の正規表現 r について、$L(r)$ が Σ^* か否かは有限時間内に判定できる。

   何故ならば、上記 1 の性質から、2 つの正規表現 r と $(a_1|a_2|...|a_n)^*$ が等価であるか否かを判定すればいいからです。ただし、$a_1, a_2, ..., a_n \in \Sigma$ とします。

4. 任意の 2 つの正規言語 L_1、L_2 が、正規表現、ε-NFA、NFA、DFA のいずれかを用いて定義されているとき、$L_1 = L_2$ が成り立つか否かは 有限時間内に判定できる。

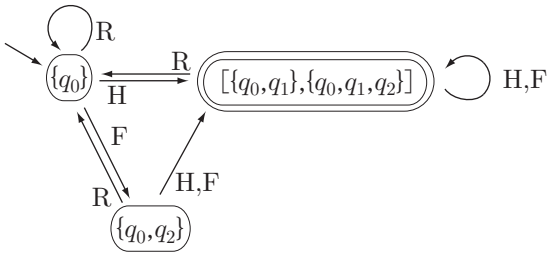


図 9.5: 最小化の例題 DFA（その 3、合併後）

何故ならば、L_1、L_2 を最小化された DFA に変換し、比較すればよいからです。

ところで、上の文章に出てきた「(任意の... について）... が成り立つか否か」という形の問いを一般に決定問題（decision problem）と呼びます。決定問題について、その成否を正しく答えることのできるアルゴリズムが存在するとき、その問題は 決定可能 （decidable）であると言います。逆に、対応するアルゴリズムが存在しないとき、その問題は決定不能（undecidable）であると言います。決定問題は計算機科学の発展と相呼応して理解が深まっていきました。

上に挙げた 4 つの決定問題はいずれも決定可能です。よって、アルゴリズムが存在し、プログラムとして実装可能です。

しかし、世の中には決定不能であるような問題も多数、知られています。そのような問題については、問題が数学的に厳密に定義されているにも関わらず、どのようなアルゴリズム/プログラムを用いても有限時間内に正しい答えを出すことができません。これはコンピュータによる計算の理論的な限界を示しています。決定不能な問題の事例は最終章で示します。

9.3　練習問題

1. 第 6 章、図 6.5 の変換後の DFA $M'_{J,2}$ を最小化しなさい。そして最小化された DFA が 5 章の図 5.1 の DFA と同じものであることを確認しなさい。

第10章　字句解析器生成系lex

有限状態オートマトンと正規表現の理論は単なる数学理論ではなく、様々に応用されています。その中で最も重要な応用のひとつが字句解析器の自動生成です。UNIX システム上の lex はそのような自動生成プログラムとして有名です[1]。ここでは lex の簡単な使用方法を通して、この授業で学んできた内容がどのように利用されているか紹介します。

10.1　例1：二つの@で囲まれた正整数の認識

4.3 節の例 12 を思い出しましょう。

正整数は、0 または 1 から 9 の数の後に 0 から 9 の数が 0 個以上続く文字列であると定義するとき、たとえば

$$0, \quad 10, \quad 99, \quad 123456789$$

などは正整数です。しかし

$$00, \quad 01, \quad 12a, \quad +0, \quad -0, \quad -888, \quad 12+66$$

などは正整数ではありません。このように定義された正整数は以下の正規表現で表わすことができました。

$$0|(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$$

これに少し手を加え、「二つの@で囲まれた正整数」を考えましょう。わざわざ、@ で囲むのは、lex の動作を単純化するためです[2]。たとえば

$$@0@, \quad @10@, \quad @99@, \quad @123456789@$$

などがその例です。これを正規表現で表すならば、以下のようになります。

$$@(0|(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*)@$$

このような正規表現が与えられたならば、これを 4 章、7 章、6 章の変換方法に従って決定性有限状態オートマトンに変換し、さらに 9 章の方法で状態数を最少化することができます。その決定

[1] Windows で lex を利用するには、フリーの UNIX ライクな環境 Cygwin をインストールすることで可能です。興味のある人は講義担当者まで連絡してください。講義担当者が自前で作成したインストール資料（pdf）を提供します。

[2] 実は lex は単に正規表現の所属問題をプログラム化するツールではなく、様々な高度な機能が備わっています。しかし、そのような高度な機能はしばしば初学者を混乱させるため、ここではわざと@で囲むことを行います。

```
//ここが宣言部

%%

//ここが正規表現部

@(0|[1-9][0-9]*)@ { return 1; }
.                 { return 0; }

%%

//ここがプログラム部

int main(){
        if(yylex()) printf("YES, %s is accepted\n",yytext);
        else printf("NO, rejected\n");
}
int yywrap(){ return 1; }
```

図 10.1: 二つの@で囲まれた正整数の lex 記述

性有限状態オートマトンから C プログラムを作成すれば（5.4 節参照)、「二つの@で囲まれた正整数」を認識するプログラムを自動的に作り出すことができるのです。

lex はこの変換過程を自動化するツールです。以下、lex の使い方を簡単に紹介します。

10.2　例 1 の lex 記述

lex へ入力するソースファイルは、正規表現とそれに関連する C プログラムからなります。図 10.1 は、前節の「二つの@で囲まれた正整数」を lex で記述したものです。以下に、この記述の内容を箇条的に概説します。

1. lex 記述は二つの “%%” によって三つの部分に大別されます。以下、それらの部分を宣言部、正規表現部、プログラム部と呼ぶこととします。

2. 宣言部では正規表現部やプログラムにおいて使用されるデータの型や大域変数を宣言します。図 10.1 の記述例では宣言部は空（から）です。というのも、今回の例題はとても簡単なため、宣言を必要としないからです。

3. 正規表現部には正規表現とそれが認識されたときのアクション（action）の対を記述します。図 10.1 の記述例では `@(0|[1-9][0-9]*)@` が、「@で囲まれた正整数」を表す正規表現です。4 章で述べた正規表現の書き方とは異なることを注意してください。この違いについては後に詳しく述べます。{ return 1; } は、入力文字列が「@で囲まれた正整数」と認識されたときのアクションを表します。この場合には整数値 1 を返します。次の行の先頭のドット ‘.’

は、@(0|[1-9][0-9]*)@ に当てはまらない、その他の全ての文字列の先頭の文字を表します。これは、想定外の入力に対する例外処理（exception handling）の記述と考えてよいでしょう。入力文字列が「@で囲まれた正整数」に当てはまらないと認識されたときには、アクション { return 0; } によって整数値 0 を返します。

4. 正規表現部は lex によって C 関数 int yylex() に変換されます。yylex() は決定性有限状態オートマトンを C プログラムとして表現したものです。3. のアクションとは、実はこの関数のリターン文に他なりません。

5. プログラム部には C のプログラムを記述します。上の記述例では main 関数を定義しています。この main 関数は「@で囲まれた正整数」を認識する関数 yylex() を呼び出し、もしその戻り値が真値（C 言語では非 0 値）ならば、

```
YES, ... is accepted
```

と標準出力へ出力します。... には実際に認識された文字列が入ります。なお、関数 yylex() は認識した文字列を大域変数 yytext に自動的に格納します。もし戻り値が偽値（C 言語では 0 値）ならば

```
NO, rejected
```

と出力します。関数 printf() は、C の標準出力関数です。lex は C++との相性があまり良くないので、ここでは出力のために、C++の出力ストリーム cout ではなく、C の出力関数 printf() を用いました [3]。

6. 上の記述例のプログラム部には関数 int yywrap() が定義されていますが、この関数の役割については当面知る必要はありません。

さて、この lex 記述で最も重要な部分は「@で囲まれた正整数」を表す正規表現

```
@(0|[1-9][0-9]*)@
```

です。前節では、これを

$$@(0|(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)@$$

と書きました。実は 4 章の正規表現と lex の正規表現では異なる点が多々あります。それを列挙すると以下の通りです。

[3] C++向きの字句解析器生成系 lex++（あるいは flex++）というものも知られています。あるいは lex に “-+” というオプションを付けると、C++のプログラムが出力されますが、ここでは用いません。

星印 ‘ *’ と ‘*’ : 4 章の正規表現でクリーネ閉包（0 回以上の繰り返し）を表すときには、r^* のように、星印 $*$ を正規表現 r の右肩に付けました。しかし、通常の ASCII テキスト・ファイルでは肩付き文字を表すことができないため、lex では単に星印を正規表現の後ろに付け、r* と表します。

括弧 [ ] とハイフン ‘-’ : 丸括弧 () の意味は 4 章の正規表現と lex の正規表現で同じです。4 章の正規表現では大括弧（角括弧）[] を用いませんでしたが、lex の大括弧 [ ] は複数個の | の省略記号として用いることができます。しかもハイフン - と共に使用することで、記述量を大幅に削減できます。たとえば lex の正規表現 [0-9] は、(0|1|2|3|4|5|6|7|8|9) を略記したものと見なされます。すなわち「0 から 9 までの任意の 1 文字」を表します。一般的には [a-b] という記述によって「ASCII 文字の並びに順において a から b までの任意の 1 文字」を表します。よって lex の正規表現 @(0|[1-9][0-9]*)@ は「@ で始まり、その後に 0 が続くか、あるいは 1 から 9 の中の 1 文字が続き、さらに 0 から 9 の中の任意の文字が 0 回以上続き、最後に @ が続く文字列」を表現していることになります。

10.3　lex 記述のコンパイルと実行

前節の lex 記述のファイル名を example1.l としましょう。UNIX のセッション下で

```
> lex example1.l
```

または

```
> flex example1.l
```

とコマンド実行することで、lex.yy.c という C プログラムが自動的に生成されます。そこで、さらに

```
> cc lex.yy.c
```

または

```
> gcc lex.yy.c
```

によってコンパイルすると、a.out という実行可能ファイル（プログラム）が生成されます。これを実行するには

```
> ./a.out
```

と入力します。そうすると、プログラムが標準入力（キーボード）からの入力待ち状態に入ります。ここでたとえば

```
@10@
```

と入力すると、

```
YES, @10@ is accepted
```

と出力され、プログラムの実行は終了します。すなわち @10@ が「@ で囲まれた正整数」と認識されました。もし

```
@00@
```

と入力したならば、これは「@ で囲まれた正整数」ではないので、

```
NO, rejected
```

と出力されます。

10.4　例 2: 二つの @ で囲まれた a の加減算式の認識

二つ目の例題は、@ で囲まれた a の加減算式です。ただし a は変数名です。「a の加減算式」とは、たとえば

$$a, \quad a+a, \quad a-a, \quad a+a-a, \quad a-a-a+a$$

などのように、加減算演算子 $+$ と $-$ を用いて a を連結したものです。これらを @ で囲めば、「二つの @ で囲まれた a の加減算式」になります。正規表現で表すと、以下の通りです。

$$@a((+|-)a)^*@$$

上の正規表現は直感的には「@ で始まり、その後に a が続き、さらにその後に $+$ または $-$ に続く a が 0 回以上現れ、最後に @ が現れる記号列」を表現しています。

上の正規表現を用いた lex 記述の例は図 10.2 の通りです。図 10.1 の例 1 の記述とは正規表現部のみが異なります。ここに lex 正規表現は以下の通りとしました。

```
@a(("+"|"-")a)*@
```

既に述べたように、記号 $+$ や $-$ は lex の正規表現では特別な意味を持つ演算子です。そこで、普通の記号としての $+$ や $-$ を用いたいときには、" " で囲まねばならないと約束されています。

例 1 と同様に、この lex 記述をコンパイル、実行し、入力文字列 @a+a@ や @a-a+a@ などが受理されることを確かめましょう。

```
%%

@a(("+"|"-")a)*@          { return 1; }
.                         { return 0; }

%%

int main(){
        if(yylex()) printf("YES, %s is accepted\n",yytext);
        else printf("NO, rejected\n");
}
int yywrap(){ return 1; }
```

図 10.2: 二つの@で囲まれた正整数の lex 記述

10.5　例 3: 二つの@で囲まれた、缶ジュースが出てくる記号列

三つ目の例題は、この講義テキストで継続して用いている例題：缶ジュースが自動販売機から出てくる記号列です。ここでも lex の仕様の都合上、@で囲まれた記号列を考えます。それを表わす正規表現は以下の通りです。

$$@(H|F|R)^*(H|FF)(H|F)^*@$$

上の正規表現を lex の正規表現で表わすと以下のようになります。

```
@[HFR]*(H|FF)[HF]*@
```

ここに括弧 [] で囲まれた [HFR] および [HF] は、「H または F または R」、「H または F」を表わします。ハイフンを含まない括弧 [] はその中に現れる文字のひとつを表わすと約束されています。別の書き方として、

```
@(H|F|R)*(H|FF)(H|F)*@
```

も可能ですが、慣れると前者の書き方がコンパクトです。

lex 記述は図 10.3 の通りです。この lex 記述をコンパイル、実行し、入力文字列 @HF@ や @HFHFRFH@などが受理されることを確かめましょう。また逆に、@HHRF@ や @F@ が受理されないことも確かめましょう。

```
%%

@[HFR]*(H|FF)[HF]*@      { return 1; }
.                        { return 0; }

%%

int main(){
        if(yylex()) printf("YES, %s is accepted\n",yytext);
        else printf("NO, rejected\n");
}
int yywrap(){ return 1; }
```

図 10.3: 二つの@で囲まれた正整数の lex 記述

第11章　出力付きオートマトン（試験範囲外）

この章では有限状態オートマトンの応用例を紹介します。

有限状態オートマトンは入力された記号列の 受理／不受理 を調べる機械です。既に見てきたように、オートマトンの構造は決して単純なものではないのですが、しかし、動作結果が二者択一であるため、応用例が限られ、オートマトンが世の中の何の役に立つのか、あるいはコンピュータとどういう関係にあるのか、などという疑問が払拭できない学生もいるかもしれません。そこで、オートマトンの 出力動作 を拡張したモデルを紹介します。類似の概念を 1 年生必修科目「論理回路」で学んだはずです。

11.1　ムーア機械

ムーア機械（Moore machine）[1] はオートマトンが状態を遷移する毎にあるひとつの記号を出力する機械です。

たとえば入力アルファベットを $\{0,1\}$ とし、入力記号列 01011 が与えられたと仮定します。このとき、入力記号列中の 0 と 1 を反転した記号列 10100 を出力するような機械（拡張された DFA）を図 11.1 のように作ることができます。

この機械は次の点で通常の DFA と異なります。

1. 最終状態が定義されません。また、入力記号列が終端することは仮定されておらず、記号が入力される限り、動作を続けます。

2. 各状態に出力記号が付加されており、状態遷移が起る度毎に、遷移先の状態に対応する出力記号が出力されます。また、機械が動作を始めた直後の最初の入力記号を読む前には初期状態の出力記号が出力されます。出力記号は、状態遷移図では状態の真上に記載すると約束します（図 11.1 では、q_0 に **0** が、q_1 に **1** が付加されています）。

よって、図 11.1 の機械に入力記号列 01011 が与えられた場合、状態遷移は以下のように起こり、結果、記号列 110100 が出力されます。

[1]「ムーア」とはこの機械を提案した数学者 エドワード・ムーア の名前です。

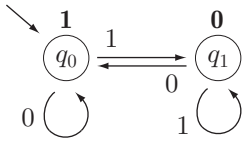


図 11.1: 入力記号を反転するムーア機械

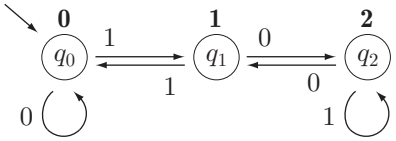
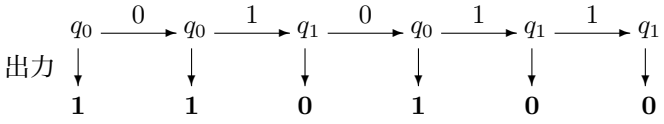
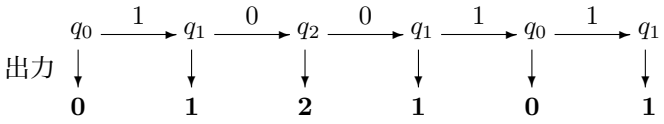


図 11.2: 入力 2 進数を 3 で割った余りを出力するムーア機械



出力記号列 110100 の先頭の 1 文字を無視するならば、これが入力記号列 01011 の bit 反転になっています。

やや複雑な例題として、その時点まで読み込んだ入力記号列を 2 進数 v と解釈したときに、v を 3 で割った余り $v \mod 3$ を出力するムーア機械を図 11.2 のように紹介します。この機械に入力列 10011 を与えた場合の動作は以下の通りです。



この出力が正しいことは、各時点までの入力記号列、その記号列の数値解釈、3 で割った余り、を以下のように列挙してみれば確認できます。

入力記号列	10 進数値	3 で割った余り	出力記号
ε	0	0	**0**
1	1	1	**1**
10	2	2	**2**
100	4	1	**1**
1001	9	0	**0**
10011	19	1	**1**

考察　図 11.2 のムーア機械が正しい答えを出力する理由を述べなさい。（ヒント：数値 v を 3 で割った余りを p （$p = 0, 1, 2$）とするとき、$2v$ を 3 で割った余りは $2p \mod 3$、$2v + 1$ を 3 で割った余りは $2p + 1 \mod 3$ になることを確認しなさい。）

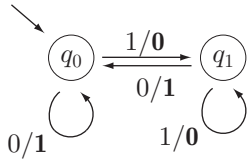


図 11.3: 入力記号を反転するミーリー機械

なお、例題からも分かるように、ムーア機械の動作は決定性オートマトンを前提としており、非決定性オートマトンには馴染みません。

ムーア機械は 6 項組 $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$ でも定義できます。ここに、Q、Σ、$\delta : Q \times \Sigma \to Q$、$q_0 \in Q$ は DFA と同じです。Δ は出力アルファベット（出力記号の有限集合）であり、$\lambda : Q \to \Delta$ は各状態の出力記号を求める関数です。

11.2　ミーリー機械

ミーリー機械（Mealy machine）[2] はオートマトンが状態を遷移するときにあるひとつの記号を出力する機械です。ムーア機械は遷移先の状態で記号を出力しますが、ミーリー機械では遷移の途中で出力するという点が異なります。

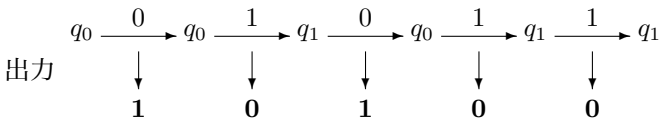
ムーア機械と同様に、入力記号列中の 0 と 1 を反転した記号列を出力するミーリー機械を図 11.3 に示します。

この機械は次の点で DFA や図 11.1 のミーリー機械と異なります。

1. 最終状態が定義されません。（この点はムーア機械と同様です。）

2. 各有向枝に出力記号が付加されており、状態遷移が起る度毎に、遷移に用いる枝に付随する出力記号が出力されます。機械が動作を始めた直後の最初の遷移前には如何なる出力も起きません。出力記号 **b** は、状態遷移図では有向枝に入力記号 a と共に $a/\mathbf{b}$ と記載すると約束します。

よって、図 11.3 の機械に入力記号列 01011 が与えられた場合、状態遷移は以下のように起こり、結果、記号列 110100 が出力されます。



同様に、読み込んだ入力記号列を 3 で割った余りを出力するミーリー機械は図 11.4 の通りです。

[2]「ミーリー」とはこの機械を提案した数学者 ジョージ・H・ミーリー の名前です。なお、日本語では「ミーリー」ではなく「ミーリ」と記述している文献も多々あります。

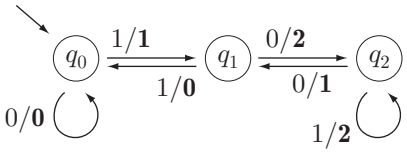


図 11.4: 入力 2 進数を 3 で割った余りを出力するミーリー機械

11.3　ムーア機械とミーリー機械の等価性

図 11.1 と図 11.3、図 11.2 と図 11.4 を比べてみてください。ムーア機械とミーリー機械の類似性に気づくでしょう。

ムーア機械から出力される最初の 1 個の記号（機械が動作開始直後に初期に出力する記号）を無視するならば、図 11.1 と図 11.3 の機械は同じ入力記号列に対して同じ出力記号列を出力します。図 11.2 と図 11.4 の機械も同様です。このような関係が成り立つとき、それらムーア機械とミーリー機械は 等価 であると言います。このとき、次の事柄が知られています。

1. 任意のムーア機械に等価なミーリー機械を作ることができる。

2. 任意のミーリー機械に等価なムーア機械を作ることができる。

1. の具体的な変換手法を与えることは比較的容易です。余裕のある人は図 11.1〜図 11.4 を参考に考えてみてください。

2. の具体的な変換手法は少し複雑です。というのは、ミーリ機械は遷移途中で出力記号を出力しますが、ムーア機械は遷移先の状態で出力記号を出力します。ふたつの機械の出力のタイミングがずれており [3]、これを解決するには変換後のムーア機械の状態数を増やす必要があります。

[3]出力のタイミングがずれている事情は 1. も同様なのですが、ムーア機械よりもミーリー機械が出力の融通が利くため、タイミングのずれは 1. では大きな問題になりません。それに対して 2. では融通の利きにくいムーア機械でミーリー機械をシミュレートせねばならず、工夫が必要です。

第12章　文脈自由文法

第 4 章、第 5 章の例 12 を思い出しましょう。次の言語（記号列の集合）：

$$\{a^n b^n \mid n \geq 0\} = \{\varepsilon, ab, aabb, aaabbb, ...\}$$

は如何なる DFA の 受理言語 でもありませんでした（5.3 節参照）。また如何なる正規表現を持ってしても、この集合を表現することはできません（4.3 節参照）。というのも、この集合は正規表現や有限状態オートマトンの表現できる言語クラス（正規言語）には属さないからです。

言語学者チョムスキー（Chomsky）[1] は様々な言語のクラスを四つの階層に分けました（第 14 章参照）。その階層で言えば、上の集合は、正規言語のひとつ上の言語クラスである文脈自由言語に属するのです。

この言語クラスは特殊なクラスではありません。言語 $\{a^n b^n \mid n \geq 0\}$ の a を左括弧 '(' に置き換え、b を右括弧 ')' に置き換え、記号列の真ん中に数字の 1 を置いてみます。そのようにして作られた言語:

$$\{({}^n 1)^n \mid n \geq 0\} = \{1, (1), ((1)), (((1))), ((((1)))), ...\}$$

に含まれる記号列は、整数 1 を括弧で幾重にも囲んだ形をしており、C/C++言語の代入文の右辺として許される式です。実は、任意の四則演算式の形 — たとえば `a+b`, `a+b+c`, `a*b+c`, `a-b*c` など — も文脈自由言語のクラスであることが知られています。C 言語や C++、Java などの プログラミング言語 のプログラムの形も文脈自由言語のクラスであることが知られています [2]。

12.1　定義

文脈自由文法では 2 種類の記号を用います。

ひとつは、有限状態オートマトン、正規表現にも用いたアルファベット Σ です。Σ の上の言語を扱うという意味では、文脈自由文法も正規表現も同じです。ただし、文脈自由文法では Σ に属

[1] チョムスキー（アメリカ人）は批評家としても有名です。

[2] 厳密に言えば、プログラムを論じる際には、構文と意味を論じなければなりません。ここでいう「正しいプログラムの形が文脈自由言語のクラスである」とは、構文が文脈自由言語のクラスであることを意味します。プログラムの意味は文脈自由言語のクラスではありません。そもそもプログラムの意味を言語クラスに当てはめて論じることは、あまり有益ではありません。

する記号のことを主に**終端記号**（terminal symbol あるいは単に terminal）と読んでいます。終端記号は「導出が終端する記号」という意味から命名されました。あるいは「他の記号に書き換えられることのない記号」と言ってもよいでしょう。導出の定義はすぐ後で行います。

文脈自由文法で用いるもうひとつの記号は**非終端記号**（nonterminal symbol または単に nonterminal）です。非終端記号は「(導出が) 終端しない記号」という意味から命名されたものです。すなわち、非終端記号は「他の記号に書き換えられる記号」であって[3]、導出がそこで終わることはありません。

文脈自由文法ではアルファベットを Σ ではなく、T で表します。そうすると、T^*（T のクリーネ閉包）は終端記号列の全体を表します。さらに非終端記号の有限集合を N とするとき、$(T \cup N)^*$ は終端記号と非終端記号からなる列の全体を表します。また、$u \in T^*$ を満たす u は任意の終端記号列、$\alpha \in (T \cup N)^*$ を満たす α は終端記号と非終端記号からなる任意の列です。この章のこれ以降では、記号 T、N、T^*、$(T \cup N)^*$ が何度も出てきますが、全てこの意味と約束しておきます。

非終端記号を書き換える規則を**生成規則**（production rule あるいは単に production）または**構文規則**（syntax rule）と呼びます。ひとつの生成規則は、

$$X \to \beta$$

という形をしており、ここに X は非終端記号（$\in N$）、β は任意の記号列（$\in (T \cup N)^*$）です。β が空列 ε の場合もありうることに注意してください。その場合の生成規則は $X \to \varepsilon$ という形です。

記号列 $\alpha X \gamma$（$X \in N$, $\alpha, \gamma \in (T \cup N)^*$）と生成規則 $X \to \beta$ が与えられたとき、「$\alpha X \gamma$ は $X \to \beta$ によって $\alpha\beta\gamma$ に書き換えられる」、あるいは単に「$\alpha X \gamma$ は $\alpha\beta\gamma$ に書き換えられる」と言い、この書き換えを

$$\alpha X \gamma \Rightarrow \alpha\beta\gamma$$

と表します。また「書き換え」のことを「導出」とも言います。生成規則の有限集合 P に含まれる適当な生成規則を用いることで記号列 α_1、α_2、...、α_k（$k \geq 1$）が

$$\alpha_1 \Rightarrow \alpha_2, \quad \alpha_2 \Rightarrow \alpha_3, \quad \cdots, \quad \alpha_{k-1} \Rightarrow \alpha_k$$

という関係を満たすとき、これを

$$\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_k$$

[3]「書き換えられる」ということから、かつては変数（variable）とも呼ばれましたが、プログラミング言語で使う「変数」と混乱するため、現在ではその呼び方はしません。

と表し、**導出列**（derivation sequence）と呼びます。またこの導出列は、途中を省略し、

$$\alpha_1 \Rightarrow^* \alpha_k$$

とも表し、「0 回以上[4] の書き換え操作によって α_1 から α_k が導出される」、あるいは単に「α_1 から α_k が導出される」と言います。

文脈自由文法（context-free grammar）G は 4 項組 (N, T, P, S) によって定義されます。ここに N は非終端記号の有限集合、T は終端記号の有限集合、P は生成規則の有限集合、S は開始記号 S $(\in N)$ です。開始記号 S から導出される記号列（$\in (T \cup N)^*$）を文形式（sentential form）と呼びます。特に、開始記号 S から導出される終端記号列（$\in T^*$）を文（sentence）と呼びます。G の定義する**文脈自由言語**（context-free language）$L(G)$ とは、G の文の全体集合であって、形式的には以下の等式の通りです。

$$L(G) = \{u \in T^* \mid S \Rightarrow^* u\}$$

このとき、G は $L(G)$ を生成するとも言います。

この章の冒頭に出てきた言語 $\{a^n b^n \mid n \geq 0\}$ は以下の文脈自由文法によって生成可能です。

$$\begin{aligned} G_1 &= (\{S\}, \{a, b\}, P_1, S) \\ P_1 &= \{S \to \varepsilon,\ S \to aSb\} \end{aligned}$$

なぜならば、この文法では以下のような様々な導出が可能です。

$$S \Rightarrow \varepsilon$$

$$S \Rightarrow aSb \Rightarrow ab$$

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$$

$$\ldots$$

よって、

$$L(G_1) = \{u \in T^* \mid S \Rightarrow^* u\} = \{a^n b^n \mid n \geq 0\}$$

また、言語 $L(G_1) - \{\varepsilon\}$ を生成する文脈自由文法は以下の通りです。

$$\begin{aligned} G_2 &= (\{S\}, \{a, b\}, P_2, S) \\ P_2 &= \{S \to ab,\ S \to aSb\} \end{aligned}$$

[4] 4 章においてアスタリスク $*$ が「0 回以上」を表したことを思い出しましょう。この文法やオートマトンの分野では、しばしば $*$ をそのような意味で用います。

もう少し複雑な例として、以下の文脈自由文法は、終端記号 a と b が同じ個数だけ現れるような全ての記号列の集合を生成します。

$$\begin{aligned} G_3 &= (\{S, A, B\}, \{a, b\}, P_3, S) \\ P_3 &= \{S \to aB,\ S \to bA, \\ &\quad A \to a,\ A \to aS,\ A \to bAA, \\ &\quad B \to b,\ B \to bS,\ B \to aBB\} \end{aligned}$$

考察（少し難しい）　文法 G_3 が a と b が同じ個数だけ現れるような全ての記号列を生成できることを証明しなさい。ヒント: 非終端記号 A は a の個数が b よりも 1 個だけ多い記号列を導出し、B は b の個数が a よりも 1 個だけ多い記号列を導出します。それを考慮し、文に長さに関する数学的帰納法を用いて証明します。

ところで、文脈自由文法を構成する四つの要素: 終端記号、非終端記号、生成規則、開始記号の中で、文法定義の核になるものは、言うまでもなく、生成規則です。そこで簡単のため、以下では、文脈自由文法を例示する際には生成規則のみを示すと約束します。使用する非終端記号は、生成規則の左辺に現れるもの全てです。使用する終端記号は、生成規則の右辺に現れる記号から非終端記号を除いた全てです。開始記号は、最初の行に示された生成規則の左辺の記号と約束します。

たとえば図 12.1 の文法 G_4 は 6 個の生成規則からなりますが、E のみが非終端記号であり、かつ E が開始記号です。終端記号は、E 以外の記号 $+$、$-$、$*$、$/$、id、$($、$)$ の 7 個です。この文法では、以下のような導出が可能です。ここに下線部の非終端記号が次に書き換える記号を表します。

$$\begin{aligned} \underline{E} &\Rightarrow E + \underline{E} \\ &\Rightarrow E + \underline{E} * E \\ &\Rightarrow E + id * \underline{E} \\ &\Rightarrow \underline{E} + id * id \\ &\Rightarrow id + id * id \end{aligned}$$

文 $id + id * id$ が導出されました。終端記号 id をプログラミング言語でいう変数と見なすと、この文は代入文の右辺に現れそうな四則演算式に他なりません。

12.2　導出

開始記号からある特定の終端記号列を導くとき、その導出の方法は 唯一 ではありません。前節最後の導出例を振り返ってみましょう。2 回目の書き換えの直後の場面:

1: $E \to E \ + \ E$
2: $E \to E \ - \ E$
3: $E \to E \ * \ E$
4: $E \to E \ / \ E$
5: $E \to id$
6: $E \to (\ E\ )$

図 12.1: 四則演算式を表す曖昧な文法: G_4

$$\Rightarrow \ E \ + \ E \ * \ E$$

では文形式に含まれる三つの非終端記号 E のいずれもが書き換え可能です。導出では「次にどの非終端記号を書き換えるか」という点について任意性が残されており、実際、その選択は全く自由です。どの非終端記号から先に書き換えようとも、もし終端記号列がその文法の言語に属するならば、必ずその終端記号列を導出することができます。というのも、文脈自由文法では文脈に依存せずに書き換えることができる ため、非終端記号を書き換える順序は最終的に導出される記号列（文）に影響しません。

この任意性は、しかし理論を展開する上で邪魔になることが多いため、通常は書き換えの順序に制限を設けます。最もよく用いられる制限は「文形式の一番左に現れる非終端記号を書き換える」というものです。これを **最左導出** （leftmost derivation）と呼びます。たとえば前節の、文 $E \ + \ E \ * \ E$ を導出した例題を最左導出で行うと以下の通りです。左側から終端記号列が確定していく様子が分かるでしょう。

$$\begin{aligned}
\underline{E} &\Rightarrow \underline{E} \ + \ E \\
&\Rightarrow id \ + \ \underline{E} \\
&\Rightarrow id \ + \ \underline{E} \ * \ E \\
&\Rightarrow id \ + \ id \ * \ \underline{E} \\
&\Rightarrow id \ + \ id \ * \ id
\end{aligned}$$

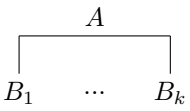
最左導出とは逆に、「文形式の一番右に現れる非終端記号を書き換える」という制限を設けた導出を **最右導出** （rightmost derivation）と呼びます。以下はその導出例です。右側から終端記号列が確定していく様子が見えるでしょう。

$$\begin{aligned}
\underline{E} &\Rightarrow E \ + \ \underline{E} \\
&\Rightarrow E \ + \ E \ * \ \underline{E} \\
&\Rightarrow E \ + \ \underline{E} \ * \ id \\
&\Rightarrow \underline{E} \ + \ id \ * \ id \\
&\Rightarrow id \ + \ id \ * \ id
\end{aligned}$$

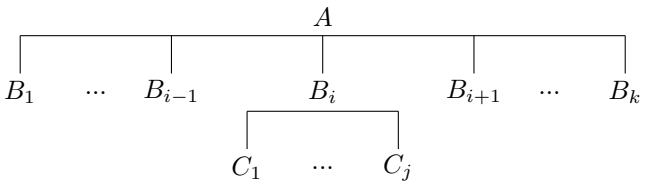
導出方法が異なると導出列も異なりますが、最終的に同じ終端記号列が導出できる点に注意してください。

12.3　構文木と曖昧性

非終端記号 A を生成規則 $A \to B_1...B_k$ によって書き換えるとき、A と $B_1...B_k$ について、以下のような親子関係の木を作るとします。



さらに $B_1...B_k$ の中の非終端記号 B_i（$1 \leq i \leq k$）を生成規則 $B_i \to C_1...C_j$ によって書き換え、$B_1...B_{i-1}C_1...C_jB_{i+1}...B_k$ を得るとき、これに対応して、木を以下のように成長させます。



これを繰り返すことで、開始記号 S からの文 $t_1t_2...t_n$ の導出 $S \Rightarrow^* t_1t_2...t_n$ に対応する木を作ることができ、このような木のことを **構文木**（syntax tree）と呼びます。この木の根（root）は開始記号 S、葉（leaf）は各終端記号 t_1、t_2、...、t_n です。構文木は文の構文構造を表現したものと考えることができます。

文法 G について、文 $t_1t_2...t_k$ に対応する構文木が複数個存在するとき、その文法は **曖昧（あいまい）** である（ambiguous）と言います。逆に構文木が唯一ならば、その文法は曖昧ではない（unambiguous）と言います。

文法が曖昧であるとは、簡単に言えば、文の構造の解釈が唯一に定まらないことを意味します。たとえば、日本語の文法が文脈自由文法で与えられたとしましょう。与えられた日本語の文章の構文木が複数個存在するならば、その文の意味解釈が複数個存在することとなり、不都合です。日英翻訳を行う場合に大きな問題となります。

コンピュータが曖昧性を嫌うことはいうまでもありませんが、しかし曖昧な文脈自由文法は珍しいものではありません。理論的には、任意に与えられた文法が曖昧か否か検査するアルゴリズムは存在しないことが知られており、これは曖昧な文法の扱いが一筋縄では行かないことを示唆しています。

図 12.1 は、よく知られた曖昧な文法の例です。例として、先に導出例で取り上げた文 $id + id * id$ を考えてみましょう。この文に対応する構文木を作ると、図 12.2 に示すように 2 種類の構文木が

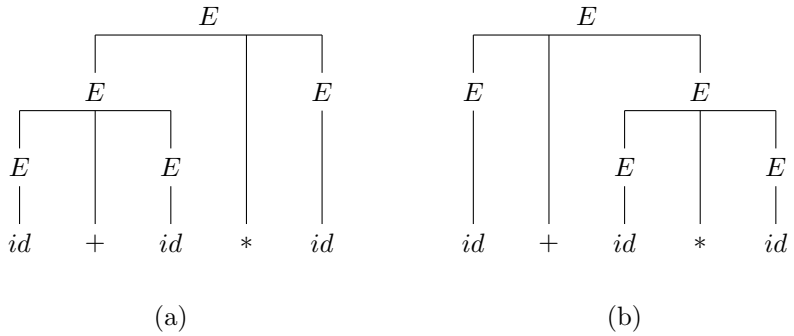


図 12.2: $id\ +\ id\ *\ id$ の二つの構文木

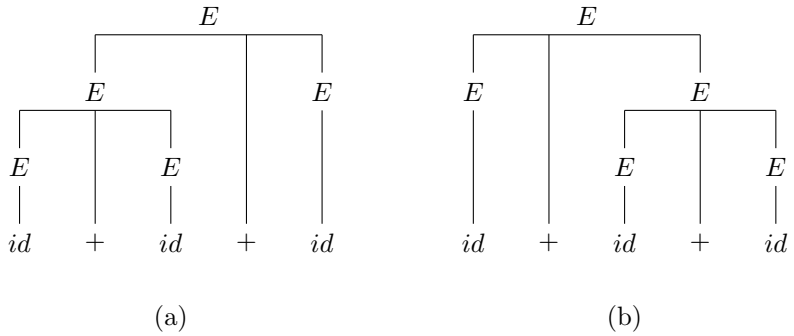


図 12.3: $id\ +\ id\ +\ id$ の二つの構文木

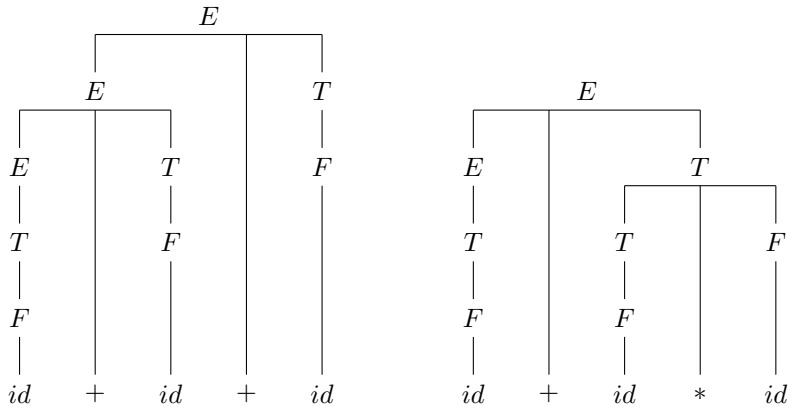
可能です。また、文 $id\ +\ id\ +\ id$ に対応する構文木を作ると、図 12.3 に示すように 2 種類の構文木が可能です。このように、ひとつの終端記号列に二つの構文木が存在するため、図 12.1 の文法 G_4 は曖昧です。

補足すると、一般に乗算演算子は加算演算子よりも演算子順位（operator precedence）が高い（あるいは演算の結合性が強い）と考えますから、$id\ +\ id\ *\ id$ という四則演算式は通常は $id\ +\ (id\ *\ id)$ と等価です。その意味では図 12.2 の二つの木のうち、(b) の木が正しい演算子順位を正しく反映しています。また、加算演算子は通常、左結合性（left associative）を持つと考えますから、$id\ +\ id\ +\ id$ という式は $(id\ +\ id)\ +\ id$ と等価です。その意味では図 12.3 の二つの木のうち、(a) の木が左結合性を正しく反映しています。

図 12.1 による四則演算式の定義は曖昧ですが、四則演算式は曖昧性なく定義できることが知られており、図 12.4 がその曖昧でない文法です。ここに非終端記号名 T は term（項）に、F は factor（因子）に由来します。この文法の特徴は、非終端記号 E、T、F の定義が 3 層の階層を成していることです。E の階層には $+$、$-$ が現れ、T の階層には $*$、$/$ が現れ、F の階層に id、(、) が現

1: $E \to E\ +\ T$	4: $T \to T\ *\ F$	7: $F \to id$
2: $E \to E\ -\ T$	5: $T \to T\ /\ F$	8: $F \to (\ E\ )$
3: $E \to T$	6: $T \to F$	

図 12.4: 四則演算式を表す曖昧でない文法: G_5



(a) $id\ +\ id\ +\ id$　　(b) $id\ +\ id\ *\ id$

図 12.5: 一意に定まる構文木

れることを注意してください。下の階層ほど演算子順位が高いため、この文法は演算子順位:

$$id,\ ()\ \ >\ \ *,\ /\ \ >\ \ +,\ -$$

を自然に表現できるように工夫されています。この文法の下での文 $id\ +\ id\ *\ id$、文 $id\ +\ id\ +\ id$ に対応する構文木を図 12.5 に示します。この文法は曖昧ではありませんから、ひとつの終端記号列には高々ひとつの構文木しか存在しません。ただし、この文法の欠点として、構築される構文木が図 12.1 の文法の構文木に比べ、高く（深く）なってしまうことです。図 12.3 の木の高さ（深さ）は 3 ですが、図 12.5 (a) の木の高さは 5 です。

考察　図 12.5 の文法の中の規則 1 から規則 6 を以下の規則と置き換えたとき、演算子の左結合性/右結合性についてどのような変化が起きるか、調べなさい。

1: $E \to T\ +\ E$	4: $T \to F\ *\ T$
2: $E \to T\ -\ E$	5: $T \to F\ /\ T$
3: $E \to T$	6: $T \to F$

第13章　プッシュダウン・オートマトン

正規言語を受理する機械として有限状態オートマトンがあるように、文脈自由言語を受理する機械としてプッシュダウン・オートマトン（pushdown automaton）が考案されています。「プッシュダウン」とは「最後に記憶された情報が最初に検索される」ことを意味します[1]。このようなデータ構造としてすぐに **スタック**（stack）が連想されますが、プッシュダウン・オートマトンとはまさに スタックを持つオートマトン に他なりません。

なお、以下ではプッシュダウン・オートマトンを簡単に PDA と呼ぶこととします。

13.1　スタックを用いた動作

正規言語ではない言語 $\{a^n b^n \mid n \geq 1\}$ を受理する PDA を検討しましょう。ただし、$n \geq 0$ でないことに注意してください。

先に述べたように、PDA はスタックを持つ有限状態オートマトンです（図 13.1 参照）。PDA は、有限状態オートマトンと同じように入力記号を読んで状態を遷移します。と同時に、スタック上の頂上（図 13.1 では最右の記号）の記号をひとつ取り出して（pop）、読み、その記号の種類によって遷移先を選ぶことが可能です。また状態遷移時にスタックに記号を積む（push）こともできます。図 13.1 では、スタックの底を左、頂上を右に表しています（下図参照）。

底 ... 頂上

さて、次のようなオートマトンを考えます。

1. 三つの状態 q_0、q_1、q_2 を作ります。q_0 は初期状態であり、かつ記号 a を読み進めている状態です。q_1 は記号 b を読み進めている状態です。q_2 は最終状態です。

2. スタックの底にはあらかじめ記号 $\#$ を置いておきます。このような記号を **スタック初期記号**（initial stack symbol）と呼びます。もしスタックから取り出した記号が $\#$ ならば、スタックの底が見えたことを意味します。

3. 状態 q_0 において

[1] 同様の術語として、FILO（first in, last out）または LIFO（last in, first out）が知られています。

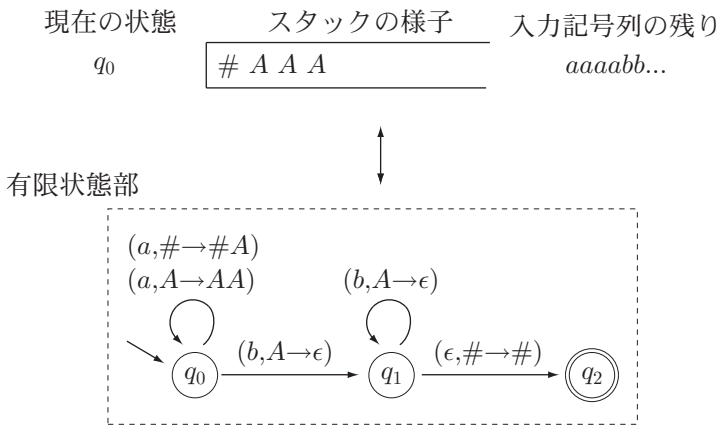


図 13.1: $a^n b^n$ を認識するプッシュダウン・オートマトン

(a) 記号 a を読んだときには、スタックの状態に関わらず、記号 A をスタックに積みます。

(b) 記号 b を読んだときに、スタックの頂上の記号が A ならば、A をスタックから降ろします。そして q_1 へ遷移します。

(c) 上の二つに当てはまらない場合には、オートマトンの動作は失敗と見なし、入力記号列は受理されません。

4. 状態 q_1 において

(a) 記号 b を読んだときに、スタックの頂上の記号が A ならば、A をスタックから降ろします。

(b) スタックの頂上の記号が $\#$ ならば、入力記号を読まないで、状態 q_2 へ遷移します（これは ε 遷移です)。

(c) 上の二つに当てはまらない場合には、オートマトンの動作は失敗と見なし、入力記号列は受理されません。

5. 状態 q_2 において入力記号を全て読み終えたならば、オートマトンの動作は成功です。さもなくば失敗です。

このオートマトンの動作のポイントは、読み込んだ入力記号 a と同じ個数の A をスタックに積むことです。そして次に入力記号 b を読むときには A をスタックからひとつずつ降ろしていきます。そして、a と同じ個数の b を読み終えたならば、スタックの頂上には $\#$ が見えているはずです。つまり、スタックを用いて記号 a の個数を記憶するのです。このような記憶装置は DFA、NFA、ε-NFA にはありませんでした。

ところで、ある状態 p において、入力記号が a であり、かつスタックの頂上の記号が Z であるときに、状態 q へ遷移し、スタックから Z を下ろし、新たにスタックに記号 Z_1、Z_2、...、Z_k $(k \geq 0)$ をこの順に積むという動作を行うと仮定します。これをオートマトンの状態遷移図に表すために、状態遷移図の p から q への枝に以下のラベルを付けると約束します。

$$(a,\ Z \to Z_1 Z_2 ... Z_k)$$

記号を読まない ε 遷移の場合には以下の通りとします。

$$(\varepsilon,\ Z \to Z_1 Z_2 ... Z_k)$$

スタックに何も積まない場合には以下の通りとします。

$$(a,\ Z \to \varepsilon)$$

図 13.1 の状態遷移図には上のようなラベルを付けています。

図 13.1 に基づいて、実際に入力記号列 $aaaabbbb$ を読み込んでみましょう。

その準備として、オートマトンの実行の様子を的確に知るために、オートマトンの状態 q、スタックの内容 γ、(まだ読み込まれていない) 入力記号列の残り u を以下のように並べて表します。

状態	スタック	入力列の残り
q	γ	u

これを一般に **時点表示** (instantaneous description) [2] と呼びます。そうすると、例題の PDA で入力記号列 $aaaabbbb$ を読むときの初期の時点表示は以下の通りです。

状態	スタック	入力列の残り
q_0	$\#$	$aaaabbbb$

ここでオートマトンが記号 a を読み、上の規則 3.(a) によって時点表示は以下のように変化します。

状態	スタック	入力列の残り
q_0	$\#A$	$aaabbbb$

同じく、残りの三つの a を読むと、変化は以下の通りです。

状態	スタック	入力列の残り
q_0	$\#AA$	$aabbbb$
q_0	$\#AAA$	$abbbb$
q_0	$\#AAAA$	$bbbb$

次に b を読むと、上の 3.(b) の規則によって、状態は q_1 に遷移し、スタック上の記号 A がひとつ減ります。

[2] 読んで字のごとく、オートマトンの各時点での状態を表した式のことです。

状態	スタック	入力列の残り
q_0	$\#A$	$aaabbbb$
q_0	$\#AA$	$aabbbb$
q_0	$\#AAA$	$abbbb$
q_0	$\#AAAA$	$bbbb$
q_1	$\#AAA$	bbb
q_1	$\#\mathrm{AA}$	
q_1	$\#\mathrm{A}$	
q_1	$\#$	
q_2	$\#$	

図 13.2: 図 13.1 のプッシュダウン・オートマトンの動作例（まとめ）

状態	スタック	入力列の残り
q_1	$\#AAA$	bbb

さらに 4.(a) の規則を 3 回適用し、時点表示は以下のように変化します。

状態	スタック	入力列の残り
q_1	$\#\mathrm{AA}$	
q_1	$\#\mathrm{A}$	
q_1	$\#$	

最後に 4.(b) の規則を適用すると、

状態	スタック	入力列の残り
q_2	$\#$	

となり、状態 q_2 において入力列は全て読み終えました。よって記号列 $aaaabbbb$ は受理されました。この一連の動作を図 13.2 にまとめました。

考察　入力記号列が ε、$aaabb$、$aabbb$ のときのオートマトンの動作をそれぞれ調べなさい。

考察　言語 $\{a^n b^n \mid n \geq 0\}$ を受理する PDA を検討しなさい。ここに上の例とは異なり、$n = 0$ の場合が含まれることを注意する。

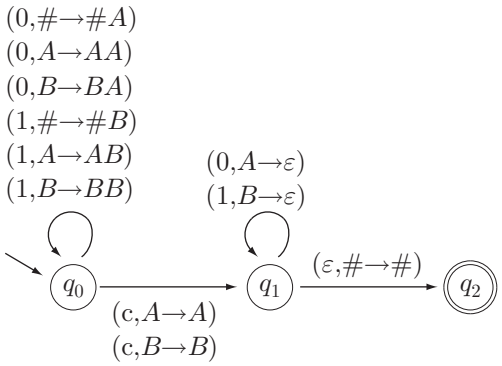


図 13.3: 言語 $\{ucu^R \mid u \in \{0,1\}^+\}$ を受理するプッシュダウン・オートマトン

13.2　決定性と非決定性

実は、前節で紹介した PDA は非決定性の動作を暗黙に含んでいます。つまり、ある状態 q において以下の二つの条件のいずれかを満たすとき、その PDA は非決定的な動作をします。

1. 入力記号が a、スタックの頂上の記号が Z であるときの動作が複数定義されている。
2. スタックの頂上の記号が Z であるとき、入力記号 a を読む動作と入力記号を読まない動作（ε 遷移）の両方が定義されている。

逆に言えば、上の二つの条件を共に満たさないとき、PDA は決定的な動作をします。

有限状態オートマトンでは決定性オートマトン（DFA）と非決定性オートマトン（NFA）の受理言語のクラスに 本質的な差 はありませんでした。すなわち、任意の NFA には、それと同じ受理言語を持つ DFA が常に存在します。しかし、PDA ではそのような性質は成り立ちません。非決定性 PDA の受理言語の中には、それと同じ受理言語を持つ決定性 PDA が存在しない場合があります。

これを理解するための例題として二つの言語とその PDA を紹介します。

ひとつは、$\{0,1,\mathrm{c}\}$ 上の言語 $\{ucu^R \mid u \in \{0,1\}^+\}$ です。ここに、u^R は記号列 u の中の記号の並びを左右逆転した記号列を表します[3]。たとえば、0c0, 1c1, 01c10, 10c01, 001c100, 011c110 などがこの言語に含まれます。ちなみにこの言語を表す文脈自由文法は以下の通りです。

$$S \to 0\mathrm{c}0, \quad S \to 1\mathrm{c}1, \quad S \to 0S0, \quad S \to 1S1$$

この記号列を受理する PDA は図 13.3 の通りです。この PDA の動作は、図 13.1 の PDA と似ています。記号列の前半部 u を読むときには、読んだ記号の情報をスタックに記録していきます。

[3]簡単のために、ここでは u は空列ではないと仮定していますが、空列を含むように拡張することは難しくありません。

状態	スタック	入力列の残り
q_0	$\#$	011c110
q_0	$\#A$	11c110
q_0	$\#AB$	1c110
q_0	$\#ABB$	c110
q_1	$\#ABB$	110
q_1	$\#AB$	10
q_1	$\#A$	0
q_1	$\#$	
q_2	$\#$	

図 13.4: 図 13.3 のプッシュダウン・オートマトンの動作例

次に記号 c を読んだところで動作が切り替わります。記号列の後半部 u^R を読むときには、スタック上の情報を読み取りながら、読んだ記号とスタック上の情報を突き合わせていきます。

図 13.4 に、記号列 011c110 を受理する動作を示します。最終状態 q_2 へ到達できましたから、この記号列は受理されました。

次に、$\{0,1\}$ 上の言語 $\{uu^R \mid u \in \{0,1\}^+\}$ を検討します。この言語は、上の言語にとてもよく似ていますが、この言語では記号 c が使用されていません。この言語に対応する文脈自由文法は以下の通りです。

$$S \to 00, \quad S \to 11, \quad S \to 0S0, \quad S \to 1S1$$

記号列 uu^R は、上の例 ucu^R と同様に左右対称です。しかし、ucu^R では記号列のちょうど真ん中に記号 c がありますが、uu^R にはそのような記号はありません。たったこれだけの違いがオートマトンの性質を劇的に変えてしまいます。

この記号列を受理する PDA は図 13.5 の通りです。このオートマトンは非決定的です。何故ならば、状態 q_0 において、入力記号が 0 であり、スタックの頂上が A であるときの動作が 2 種類あります。また入力記号が 1 であり、スタックの頂上が B であるときの動作も 2 種類あるからです。

図 13.6 に、記号列 011110 を受理する動作（受理に成功する動作例）を示します。最終状態 q_2 へ到達できましたから、この記号列は受理されました。

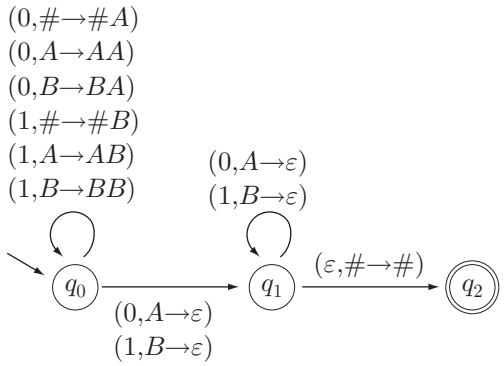


図 13.5: 言語 $\{uu^R \mid u \in \{0,1\}^+\}$ を受理するプッシュダウン・オートマトン

しかし、図 13.7 のように動作することも可能です。これは失敗した例ですが、では、uu^R という形の記号列について成功するような動作を意図的に行うことは可能でしょうか。

答えは簡単です。入力記号列が uu^R という形ならば、u を読み終えた段階、つまり入力記号列のちょうど半分を読み終えたところで状態 q_0 から状態 q_1 へ遷移すればよいのです。では、半分を読み終えたことをどうやって知ればよいでしょうか。

実はそれは不可能なのです。オートマトンは入力記号を順に読み取っていき、入力記号が無くなったときに動作を終了します。しかし、オートマトンの動作が始まる前に入力記号列の全体の長さを知ることはできません。よって、もちろん、入力列をちょうど半分を読み終えたタイミングを知ることはできません。原理的に不可能なのです。

このことは、非決定性オートマトンと決定性オートマトンが本質的に異なること、すなわち、変換によって非決定性オートマトンを決定性オートマトンに変換できないことを意味しています。有限状態オートマトンで成り立った非決定性と決定性の等価性（6.4 節を思い出しましょう）は実は異例なことなのです。

13.3　7 項組による定義

有限状態オートマトンが 5 項組で定義できたように、PDA は 7 項組で数学的に厳密に定義できます。

その PDA の定義には 2 種類の方法が知られています。ひとつは、有限状態オートマトンと同様に、入力記号列を読み終えたときに最終状態に入っていたならば、その記号列を受理すると定義する方法です。もうひとつは、入力記号列を読み終えたときにスタックが空になっていたならば、その記号列を受理すると定義する方法です。

前節までの議論は前者に基づくものでした。この節では前者の定義方法を解説します。後者の定義方法については類書を参考にしてください。

状態	スタック	入力列の残り
q_0	$\#$	011110
q_0	$\#A$	11110
q_0	$\#AB$	1110
q_0	$\#ABB$	110
q_1	$\#AB$	10
q_1	$\#A$	0
q_1	$\#$	
q_2	$\#$	

図 13.6: 図 13.5 のプッシュダウン・オートマトンの動作例（成功例）

13.3.1　最終状態による定義

ひとつの PDA M は、7 項組 $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ によって定義されます。ここに Q、Σ、q_0、F は有限状態オートマトンと同じものです。すなわち、Q は状態の有限集合、Σ は入力アルファベット、q_0 $(\in Q)$ は初期状態、F $(\subseteq Q)$ は最終状態の集合です。Γ は有限状態オートマトンには無かったもので、記号の有限集合です。これは **スタック** ・アルファベット（stack alphabet）と呼ばれます。各記号は スタック記号（stack symbol）と呼ばれ、スタックに載せて利用します。Z_0 $(\in \Gamma)$ は **スタック初期記号** （stack start symbol または stack bottom symbol）と呼ばれ、初期状態のスタックの底に置かれる記号です。遷移関数 δ は、状態とスタック記号と入力アルファベットまたは空列記号から、状態とスタック記号列のペアの有限集合を求める関数（$\in Q \times \Gamma \times (\Sigma \cup \{\varepsilon\}) \to 2^{Q \times \Gamma^*}$）です。

PDA の実行の様子を見るには、前節で導入した時点表示を用います。

この節で用いる時点表示を状態、スタック記号列、入力記号列の 3 項組 (q, γ, u) $(\in Q \times \Gamma^* \times \Sigma^*)$ で表します。スタック記号列を $s_1 s_2 ... s_k$ とするとき、このテキストでは s_1 をスタックの底とし、s_k をスタックの頂上と見なします。スタックの底は常にスタック初期記号 Z_0 ですから、その意味では、より正確な時点表示は $(q, Z_0\gamma, u) \in Q \times \{Z_0\}\Gamma^* \times \Sigma^*$ と言ってもよいでしょう。

PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ の初期時点表示は以下の通りです。

$$(q_0, Z_0, u)$$

状態	スタック	入力列の残り
q_0	$\#$	011110
q_0	$\#A$	11110
q_0	$\#AB$	1110
q_0	$\#ABB$	110
q_0	$\#ABBB$	10
q_0	$\#ABBBB$	0
q_0	$\#ABBBBA$	

図 13.7: 図 13.5 のプッシュダウン・オートマトンの動作例（失敗例）

ここに u はまだ全く読まれていない入力記号列です。M の時点表示が $(q, \gamma Z, aw)$ $(q \in Q,\ \gamma \in \Gamma^*,\ Z \in \Gamma,\ \ a \in \Sigma,\ w \in \Sigma^*)$ であって、遷移関数が $\delta(q, Z, a) \ni (p, \gamma')$ であるとき、「M（の時点表示）は $(q, \gamma Z, aw)$ から $(p, \gamma\gamma', w)$ に遷移する」と言い、これを以下のように表します。

$$(q, \gamma Z, aw) \vdash_M (p, \gamma\gamma', w)$$

すなわち、「M が状態 q に居て、スタックの頂上の記号が Z であり、次の入力記号が a であるときには、M は a を読み、状態は p へ遷移し、スタックには Z の代わりに新たに γ' を載せる」のです。ただし、a は入力記号（$\in \Sigma$）または空列記号 ε です。a が ε の場合には入力記号を読まないで遷移をするのですが、これは 7 章の ε-遷移と同じです。

時点表示 $D_1, D_2, \cdots, D_k$ $(k \geq 1)$ が関係:

$$D_1 \vdash_M D_2,\ D_2 \vdash_M D_3,\ \cdots,\ D_{k-1} \vdash_M D_k$$

を満たすとき、これを簡略化して、

$$D_1 \vdash_M^* D_k$$

と表します。なお、混乱の恐れが無い場合には、それぞれ $\vdash_M$、$\vdash_M^*$ を単に $\vdash$、$\vdash^*$ と表します。

この PDA M の受理言語 $L(M)$ は、以下のように定義される入力記号列の集合です。

$$L(M) = \{u \in \Sigma^* \mid (q_0, Z_0, u) \vdash_M^* (p, \gamma, \varepsilon),\ p \in F,\ \gamma \in \Gamma^*\}$$

すなわち、「M が入力記号列 u を全て読み終えたときに最終状態に居るような u の全体」が M の言語です。

$$
\begin{aligned}
(q_0, \#, aaaabbbb) &\vdash_{M_1} (q_0, \#A, aaabbbb) \\
&\vdash_{M_1} (q_0, \#AA, aabbbb) \\
&\vdash_{M_1} (q_0, \#AAA, abbbb) \\
&\vdash_{M_1} (q_0, \#AAAA, bbbb) \\
&\vdash_{M_1} (q_1, \#AAA, bbb) \\
&\vdash_{M_1} (q_1, \#AA, bb) \\
&\vdash_{M_1} (q_1, \#A, b) \\
&\vdash_{M_1} (q_1, \#, \varepsilon) \\
&\vdash_{M_1} (q_2, \#, \varepsilon)
\end{aligned}
$$

図 13.8: PDA M_1 の動作例

たとえば、図 13.1 の例題オートマトンを 7 項組で表すと、以下の通りです。

$$M_1 = (\{q_1, q_1, q_2\}, \{a, b\}, \{\#, A\}, \delta_1, q_0, \#, \{q_2\})$$

ここに

$$
\begin{array}{lll}
\delta_1(q_0, \#, a) = \{(q_0, \#A)\}, & \delta_1(q_0, \#, b) = \emptyset, & \delta_1(q_0, \#, \varepsilon) = \emptyset, \\
\delta_1(q_0, A, a) = \{(q_0, AA)\}, & \delta_1(q_0, A, b) = \{(q_1, \varepsilon)\}, & \delta_1(q_0, A, \varepsilon) = \emptyset, \\
\delta_1(q_1, \#, a) = \emptyset, & \delta_1(q_1, \#, b) = \emptyset, & \delta_1(q_1, \#, \varepsilon) = \{(q_2, \#)\}, \\
\delta_1(q_1, A, a) = \emptyset, & \delta_1(q_1, A, b) = \{(q_1, \varepsilon)\}, & \delta_1(q_1, A, \varepsilon) = \emptyset, \\
\delta_1(q_2, \#, a) = \emptyset, & \delta_1(q_2, \#, b) = \emptyset, & \delta_1(q_2, \#, \varepsilon) = \emptyset, \\
\delta_1(q_2, A, a) = \emptyset, & \delta_1(q_2, A, b) = \emptyset, & \delta_1(q_2, A, \varepsilon) = \emptyset.
\end{array}
$$

入力記号列 $aaaabbbb$ に関する動作例を図 13.8 に示します。

考察　図 13.3、図 13.5 の PDA を 7 項組で表わしなさい。

13.4　練習問題

1. （難問） 以下の文脈自由文法に対応する PDA を検討しなさい。

$$
\begin{array}{ll}
1\text{: } E \to E \ + \ F & 3\text{: } F \to id \\
2\text{: } E \to F & 4\text{: } F \to (\ E\ )
\end{array}
$$

第14章　チョムスキー階層

文脈自由文法は、言語学者チョムスキーが自然言語の数理を研究するときに考案した形式的文法のひとつです。チョムスキーは言語の複雑さに応じて、4種類の文法を考えました。それらはそれぞれ0型文法、1型文法、2型文法、3型文法とも呼ばれていますが、むしろ**句構造文法**(phrase structure grammar)、**文脈依存文法**(context-sensitive grammar)、**文脈自由文法**(12章参照)、**正規文法**[1](regular grammar)という呼び方が一般的です。言語クラスの包含関係は以下の通りです。

$$\text{句構造言語} \supsetneq \text{文脈依存言語} \supsetneq \text{文脈自由言語} \supsetneq \text{正規言語}$$

いずれの文法も生成規則

$$\alpha \to \beta$$

($\alpha \in (T \cup N)^* N (T \cup N)^*$, $\beta \in (T \cup N)^*$) によって定義されますが、左辺αおよび右辺βの形式に制限を加えることによって文法のクラスが変化します。

また、それぞれの文法には、その文法の言語を受理するオートマトンが考案されています。それらは、順に**チューリング・マシン**(Turing machine)、**線形拘束オートマトン**(linear bounded automaton)、**プッシュダウン・オートマトン**(13章参照)、**有限状態オートマトン**(5章参照)です。チューリング・マシンは次章で概説します。

この章では、各文法クラスとオートマトンを小さなクラスから順に概説します。

14.1　正規文法

任意の正規言語は4章で学んだ正規表現によって表現できますが、文脈自由文法を用いても表現できます。

[1] 正則文法とも呼びます。

たとえば $\Sigma = \{a, b, c\}$ 上の言語 $\{a^i b^j c^k \mid i, j, k \geq 0\}$ は正規表現では $a^* b^* c^*$ と表現されますが、文脈自由文法では

$$A \to aA, \quad A \to B,$$

$$B \to bB, \quad B \to C,$$

$$C \to cC, \quad C \to \varepsilon.$$

と表現すればよいのです（なお、この文法の開始記号は A です）。

また、言語 $\{a^m c^n \mid m, n \geq 0\} \cup \{b^m c^n \mid m, n \geq 0\}$ は、正規表現では $a^* c^* | b^* c^*$ と表現されますが、文脈自由文法では

$$S \to \varepsilon, \quad S \to a\ A, \quad S \to b\ B,$$

$$A \to C, \quad A \to a\ A,$$

$$B \to C, \quad B \to b\ B,$$

$$C \to \varepsilon, \quad C \to c\ C$$

と表現すればよいのです。

任意の正規言語は文脈自由文法で表現可能です。しかも生成規則を以下のように制限してもなお任意の正規言語が表現可能です。

1. 生成規則の右辺が終端記号または空列である。たとえば $X \to a$（$X \in N,\ a \in T \cup \{\varepsilon\}$）
2. 生成規則の右辺が終端記号または空列とそれに続くひとつの非終端記号からなる。たとえば
   $X \to a\ Y$（$X, Y \in N,\ a \in T \cup \{\varepsilon\}$）

このように制限された文法を**右正規文法**（right regular grammar）と呼びます[2]。上の二つの例題はまさに右正規文法です。

任意の正規言語は右線形文法で表現可能であることを証明するには、有限状態オートマトンの各状態と右線形文法の各非終端記号の一対一関係を調べるのが最も簡単です。

$X \to a\ Y$ を $X \to Y\ a$ で置き換えた文法を**左正規文法**（left regular grammar）と呼びます[3]。左正規文法もやはり任意の正規言語を表現可能です。右正規文法と左正規文法を合わせて正規文法と呼びます[4]。

考察 缶ジュースが出てくる記号列の集合（5 章参照）を右正規文法で表しなさい。

[2] $X \to a$、$X \to a\ Y$ の右辺の a を終端記号列 u に拡張した文法を右線形文法（right linear grammar）と呼びます。右正規文法と右線形文法は等価な表現力を持ちます。

[3] 左正規文法の右辺の a を終端記号列 u に拡張した文法を左線形文法（left linear grammar）と呼びます。

[4] 右線形文法と左線形文法を合わせて正規文法と呼ぶことがあります。

14.2　文脈自由文法

文脈自由文法については 12 章で述べた通りですから省略します。

14.3　文脈依存文法

文脈依存文法とは、生成規則 $\alpha \to \beta$（$\alpha, \beta \in (T \cup N)^*$）について、$\alpha$ の長さが β の長さ以下（すなわち $|\alpha| \leq |\beta|$）であるように制限[5]を加えた文法です。

12 章の冒頭で述べたように、$\Sigma = \{a, b, c\}$ 上の言語 $\{a^n b^n \mid n \geq 1\}$ は正規表現では表現できませんが、文脈自由文法では表現できました。これと類似した関係で、言語 $\{a^n b^n c^n \mid n \geq 1\}$ は文脈自由文法では表現できませんが、文脈依存文法では表現できる例として有名です。以下は、その言語を生成する文脈依存文法の例です。

$$S \to abc, \quad S \to aSBc, \quad cB \to Bc, \quad bB \to bb$$

この文法を用いれば以下のような導出が可能です。

$$S \Rightarrow aSBc \Rightarrow aaSBcBc \Rightarrow aaabcBcBc \Rightarrow \cdots \Rightarrow aaabbbccc$$

考察　上の導出列中の省略された部分「$\cdots$」を埋めて、導出列を完成させなさい。

結果として、任意の n について記号列 $a^n b^n c^n$ が導出可能です。

しかし、上の文法を見て、これが言語 $\{a^n b^n c^n \mid n \geq 1\}$ を生成するとすぐに理解できる人がいるでしょうか。文脈依存文法では、終端記号であっても 前後の文脈に依って 生成規則によって書き換えられて消えてしまうことがあります。同じ非終端記号であっても 前後の文脈に依って 全く別の記号列に書き換えられる場合があります。このような性質は文法の読解性を著しく低下させます。これに対して文脈自由文法では終端記号が別の記号に書き換えられることはありません。非終端記号の書き換えは文脈に依存しません（よって文脈自由と命名されています）。確かに文脈依存文法の表現力は文脈自由文法のそれを超えています。しかし文脈依存文法は人が扱うには複雑過ぎる表現形式と考えられています。形式文法の理論をコンピュータサイエンスに応用するには、文脈自由文法を応用する辺りが限界です[6]。

文脈依存文法に対応するオートマトンは線形拘束オートマトンと呼ばれています。このオートマトンは、任意の入力記号列が文脈依存文法の生成言語（文脈依存言語）に属するか否かを判定する機械です。これは、次章で述べるチューリング・マシンのテープの長さを、無限長ではなく、入力記号列の定数倍の長さに制限したものです。詳細は省略します。

[5] 厳密にはこのような文法を単調文法（monotonic grammar）と呼びますが、通常、文脈依存文法と同一視します。

[6] 数少ない応用例として、日英/英日機械翻訳などの自然言語処理では日本語や英語を定義するために文脈依存文法や次節で述べる句構造文法が用いられる場合があります。

14.4　句構造文法

句構造文法とは、文脈依存文法から $|\alpha| \leq |\beta|$ という制約を取り除いた文法であり、最も大きな表現力を持ちます。

句構造文法に対応するオートマトンはチューリング・マシンと呼ばれています。このオートマトンは、任意の入力記号列が句構造文法の生成言語（句構造言語）に属するか否かを判定する機械です。チューリング・マシンについては次章に概説します。

第15章　チューリング・マシン

チューリング・マシンはイギリス人数学者アラン・チューリングが提案した計算機械です。構造が簡単であるにも関わらず、高い表現能力を持っており、コンピュータで計算可能な問題は全てチューリング・マシンで表現できることが知られています。チューリング・マシンは、現在のコンピュータが世に出てくる前に提案されたにも関わらず、現在のコンピュータとよく似た実行方式であるため、現在のコンピュータのモデルと見なされます。

この授業の最後の章ではこのチューリング・マシンとそれにまつわる話題を概説します。

15.1　定義

チューリング・マシンには様々なヴァリエーションがありますが、典型的なチューリング・マシンは**テープ**（tape）、**テープヘッド**（tape head）、**有限状態部**（finite control part）の三つからなります（図 15.1 参照）。

テープは**セル**（cell）の並びで構成されています。 左端から方向に向かって無限個のセルが並んでおり、理論上、テープは半無限長[1]であると仮定します。テープ・ヘッドはテープのセルの上を左右へ 1 回に 1 セル分だけ移動できます。そして、ヘッドが指しているセルの記号を読み込み、かつそのセルに新たに記号を書き込むことができます。有限状態部は有限状態オートマトンと類似しています。有限個の状態を持ち、各状態ではテープ・ヘッドの左右への移動と記号の読み込み/書き込みの動作を制御します。

具体的なチューリング・マシンの動作は以下の通りです。

1. チューリング・マシンの動作開始直前の初期の状態は以下の通りです。
   (a) 長さ n の入力記号列 $a_1a_2...a_n$ が、テープの最左のセルから n 番目のセルまでにあらかじめ書き込まれています。それ以外の全てのセルには空白文字（blank symbol）B が書き込まれています。
   (b) テープ・ヘッドの初期位置は最左のセルです。
   (c) 有限状態部の状態は、あらかじめ定められた初期状態 q_0 です。

[1] テープが両側に無限に伸びるような定義（モデル化）もありますが、半無限長の場合と等価です。

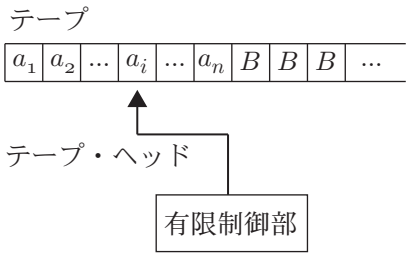


図 15.1: チューリング・マシンの構造

2. チューリング・マシンは、有限状態部の現在の状態 q と、ヘッドが現在、指しているセルに書き込まれている記号 a を参照して、以下の一連の動作を行います。この動作は q と a について決定的です。

   (a) 状態を q から q' へ遷移します。

   (b) ヘッドが指しているセルの内容を a から a' へ変更します。

   (c) ヘッドをひとつ左、またはひとつ右のセルへ移動します。

   この動作を以下では

   $$\delta(q,a) = (q', a', L) \quad \text{または} \quad \delta(q,a) = (q', a', R)$$

   で表すこととします[2]。ここに L はヘッドが左へひとつ移動することを、R はヘッドが右へひとつ移動することを表します。

3. 上の動作を次のいずれかを満たすまで繰り返します。

   (a) $\delta(q,a)$ が定義されていないときには、入力列は受理されなかった と判断し、チューリング・マシンの動作を停止します。

   (b) q が最終状態であるときには、入力列は受理された と判断し、チューリング・マシンの動作を停止します。

チューリング・マシンには様々なヴァリエーションが考えられています。たとえば、非決定的な動作を行うもの、テープを複数個持つもの、1 本のテープに複数のテープ・ヘッドを持つもの、テープが左右に無限長のもの、などなど様々です。しかし、いずれも上の基本的な定義のチューリング・マシンと等価な表現力を持つことが知られています。

14.3 節において触れた線形拘束オートマトンはチューリング・マシンのテープ長を有限のサイズに制限したもの — より正確には、テープ長を入力記号列の長さの定数倍（たとえば 2 倍など）

[2] δ は有限状態オートマトンでいう遷移関数です。

に制限したものです。テープ上の入力記号が書き込まれていないセルは作業領域として利用できます。線形拘束オートマトンではその領域が有限のサイズに制限されますから、結果としてオートマトンの動作がチューリング・マシンよりも制限されることになります。

関連して、線形拘束オートマトンに対応する文脈依存文法（14.3 節参照）では生成規則 $\alpha \to \beta$ について α の長さが β の長さ以下であるが、チューリング・マシンに対応する句構造文法（14.4 節参照）ではその制限がないと述べました。実はこの制限が線形拘束オートマトンのテープ長の制限の裏返しになっています。

15.2　動作例

チューリング・マシンの例として $\Sigma = \{a, b\}$ 上の言語 $\{a^m b^n |\ m > n\}$ を受理する場合を定義してみましょう。この言語は文脈自由言語です [3] からわざわざチューリング・マシンを用いるまでもありませんが、分かりやすい簡単な例題として取り上げてみます。

1. この言語の記号列には a が必ず 1 個以上含まれます。そこで、まず最左のセルに書き込まれている a を補助記号 x に書き換えます。もしそのような a が存在しなければ記号列を受理しません。

2. 次にテープ・ヘッドを右へ移動し、最左の b を見つけます。もしそのような b が見つからないならば（すなわち空白記号 B が先に見つかったならば)、a の個数が b の個数よりも多いことになりますから、記号列を受理します。もしそのような b が見つかったならば、その b を補助記号 x に書き換えます。

3. 次にテープ・ヘッドを左へ移動し、最左の a を見つけます。以降の処理は 1. と同じです。

そこで、まず有限状態部に状態 q_0、q_1、q_2、q_3、q_f を用意します。ここに q_0 が初期状態、q_f が最終状態です。チューリング・マシンの動作は以下のように定義します。ここに $-$ は遷移が定義されていないことを表します。なお、q_f は最終状態ですから、$\delta(q_f, ...)$ を定義する必要はありません。チューリング・マシンの動作を決める作業はプログラミングに相当します。

$$
\begin{array}{llll}
\delta(q_0, a) = (q_1, x, R), & \delta(q_0, b) = -, & \delta(q_0, x) = -, & \delta(q_0, B) = - \\
\delta(q_1, a) = (q_1, a, R), & \delta(q_1, b) = (q_2, x, L), & \delta(q_1, x) = (q_1, x, R), & \delta(q_1, B) = (q_f, B, L) \\
\delta(q_2, a) = (q_3, a, L), & \delta(q_2, b) = -, & \delta(q_2, x) = (q_2, x, L), & \delta(q_2, B) = - \\
\delta(q_3, a) = (q_3, a, L), & \delta(q_3, b) = -, & \delta(q_3, x) = (q_0, x, R), & \delta(q_3, B) = -
\end{array}
$$

実際に動かしてみましょう。入力記号列 $aaabb$ について、チューリング・マシンの動作の様子を図示したものが図 15.2 です。チューリング・マシンは初期状態から動作を始め、テープ上を何度か行き来しながら、解析を進めています。有限状態オートマトンが入力記号列を 1 回しか解析で

[3] $m > n$ という制約から、オートマトンは a の現れた個数 m を記憶しておき、b の個数 n はその m よりも小さくなければなりません。この記憶の必要性からこの言語は DFA などでは表現できず、PDA ならば表現できることが分かります。

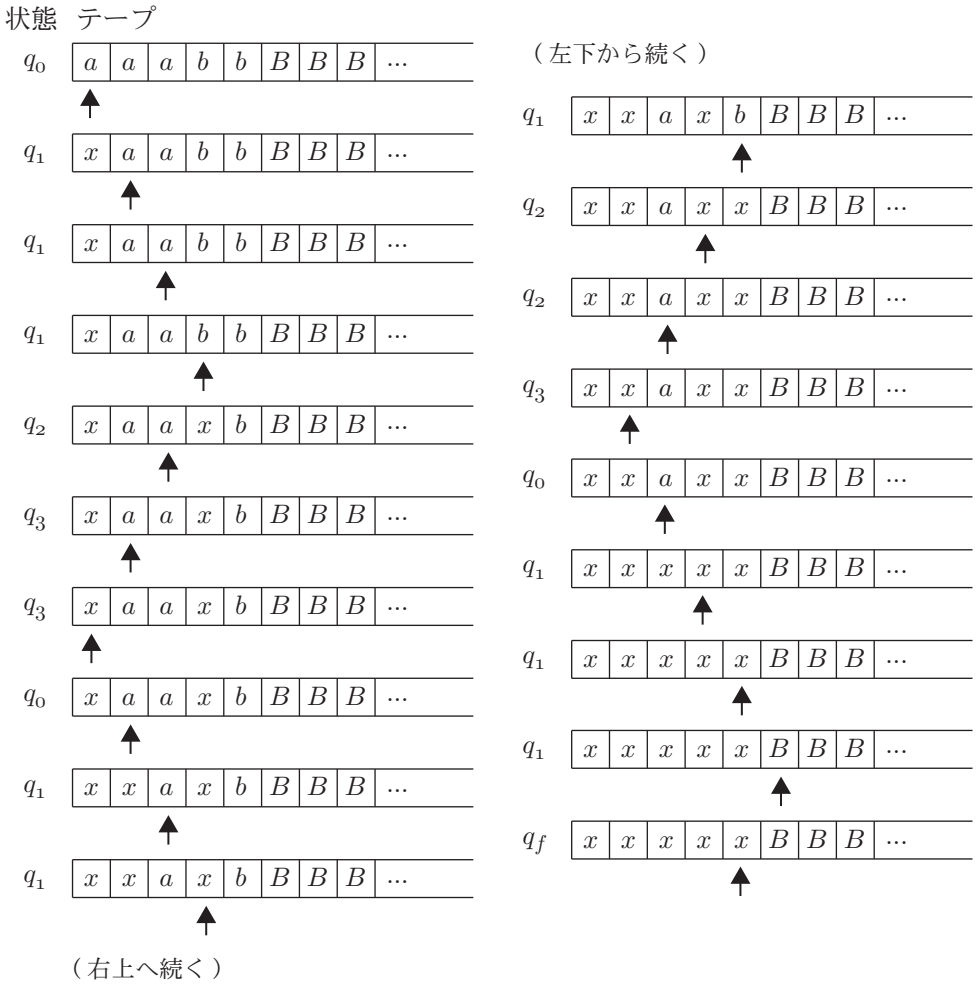


図 15.2: *aaabb* の受理過程

きなかったのに対して、チューリング・マシンでは何度も解析できます。またテープ上の記号を書き換えることで、解析途中の結果をメモしておくこともできます。

15.3　チューリング・マシンの表現力

15.3.1　四則演算

この章の冒頭において、チューリング・マシンはコンピュータで計算可能な問題は全て表現可能と述べました。よって、たとえば正整数の四則演算もチューリング・マシンを用いて実行可能なのですが、ではどのような仕組みで演算を実現すればよいのでしょうか。

前節で見たように、チューリング・マシンの構造は非常に単純です。そのため、四則演算を実現するには少し工夫が必要です。

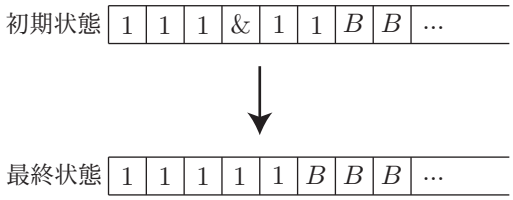


図 15.3: チューリング・マシンによる $3+2=5$ の計算

まず、正整数 n をチューリング・マシンに表現せねばなりません。ここではテープ上に記号 1 を n 個並べて表すと約束します[4]。

そして、m と n の加算のための入力記号列を、左から m 個の 1、区切り記号の &、n 個の 1 を並べて表すと約束します。となれば、加算の実行後には、左から $m+n$ 個の 1 が並ぶようにチューリング・マシンが動作すればよいのです。図 15.3 に、$3+2=5$ に対応するチューリング・マシンの初期状態と最終状態を示します。

考察　図 15.3 のような加算を行うチューリング・マシンを設計しなさい。

減算も同様です。これには、前節の言語 $\{a^m b^n | \ m > n\}$ を受理する例題が参考になるでしょう。

乗算 $m \times n$ では、m を n 回加えるように設計します。これには加算を行うチューリング・マシンの動作を利用します。

除算 m/n では、m から n を引いた回数をテープ上にメモしておけばよいでしょう。これには減算を行うチューリング・マシンの動作を利用します。

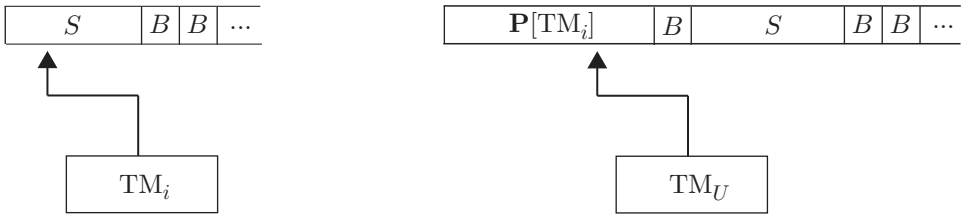
数の表現形式が私たちの通常の感覚からすれば単純過ぎるため違和感があるかもしれませんが、上のような方法でチューリング・マシンで四則演算を実現することができます。四則演算ができるならば、それを組み合わせることで指数や平方根の計算も可能です。さらに複雑な計算も可能であり、結果としてチューリング・マシンを用いて任意の算術計算を実行可能です。

15.3.2　万能チューリング・マシン

チューリング・マシンでは多種多様な処理を記述することができます。それぞれの処理をそれぞれチューリング・マシンとして設計し、実行した場合、そのチューリング・マシンはその処理のための 専用 コンピュータ（special-purpose computer）として働きます。

興味深いことに、チューリング・マシンの中にはそのような様々な専用チューリング・マシンをシミュレートできるものが存在し、特に任意のチューリング・マシンをシミュレートできるチューリング・マシンは 万能 チューリング・マシン（universal Turing machine）と呼ばれています。

[4]つまり、1 進法による数の表現を用いる訳です



(a) 専用マシーンによる実行
TM$_i(S)$

(a) 万能マシーンによる実行
TM$_U(\mathbf{P}[\mathrm{TM}_i],S)$

図 15.4: 専用チューリング・マシンと万能チューリング・マシン

今、専用コンピュータであるチューリング･マシンを処理内容に応じ、番号を付けて TM$_1$、TM$_2$、... と区別することにします。次に、その中のひとつのマシン TM$_i$ への入力記号列を S とします。このとき、S を入力とする TM$_i$ の実行を TM$_i(S)$ と表します。図 15.4 (a) がそれに対応します。

次に、万能チューリング・マシンを TM$_U$ と表します。その入力は、TM$_i$ の設計図 — これを $\mathbf{P}[\mathrm{TM}_i]$ と表します — とその TM$_i$ への入力 S です。設計図である $\mathbf{P}[\mathrm{TM}_i]$ は入力ですから、テープの上に記号列として載っていなければなりません。たとえば、この章の最初のチューリング・マシンには

$$\delta(q_0,a)=(q_1,x,R),\ \delta(q_1,a)=(q_1,a,R),\ \delta(q_2,a)=(q_3,a,L),\ ...$$

という状態遷移が定義されていました。これらを記号列として表すには、テープ上の各セルに

0	a	1	x	R	1	a	1	a	R	2	a	3	a	L	...

と配置することができます。ただし、$0,1,2,3,a,x,R,L\in\Sigma$ と仮定します。図 15.4 (b) は、記号列 $\mathbf{P}[\mathrm{TM}_i]$ と S を空白記号 B を挟んでテープ上に初期配置し、そのテープ上をテープ・ヘッドが左右に動きながら動作する万能チューリング・マシン TM$_U$ を図式化しました。この実行を TM$_U(\mathbf{P}[\mathrm{TM}_i],S)$ と表現しましょう。この実行では、TM$_U$ は、S の上の記号をひとつ読み込んだ後に $\mathbf{P}[\mathrm{TM}_i]$ の中から次の状態遷移にマッチする規則を検索し、その規則に則った遷移を行い、S 上の該当箇所の記号を書き換え、さらに S の上の次の記号を読み込み ... と動作を行ないます。専用マシンの実行 TM$_i(S)$ をひとつひとつシミュレートしていくため、万能マシンの実行 TM$_U(\mathbf{P}[\mathrm{TM}_i],S)$ は TM$_i(S)$ に比べ、非常に手間の掛かる遅い動作になりますが、等価な動作が可能です（機械が停止したときのテープの状態は TM$_U(\mathbf{P}[\mathrm{TM}_i],S)$ と TM$_i(S)$ で厳密に等しくなります)。逆に言えば、任意の TM$_i$、S について TM$_i(S)$ と等価な動作が可能なチューリング・マシンを万能チューリング・マシンと呼びます。

もし万能マシンを用意できたならば、任意の専用マシンと等価な動作ができる訳ですから、いちいち特定用途の専用マシンを作る必要はありません。専用マシンに比べて実行ステップが長くなるでしょうが、万能マシン 1 台で任意の計算ができるのです。

今日、私たちがコンピュータと呼んでいるものは、万能チューリング・マシン TM_U に相当します。コンピュータのメモリあるいはハードディスクなどの記憶装置は、チューリング・マシンのテープに相当します。そして、私たちがプログラム（あるいはソフトウェア）と呼んでいるものは、$\mathbf{P}[\mathrm{TM}_i]$ に相当します。入力データは、S に相当します。コンピュータ上で入力データを与えてプログラムを実行することは、$\mathrm{TM}_U(\mathbf{P}[\mathrm{TM}_i], S)$ に相当します。現在のコンピュータは万能チューリング・マシンを数学的基礎とするものです。この方式の特徴は、プログラムをメモリ上にデータと記憶しておく点です。それゆえ、メモリ上のプログラムを書き換えるだけで、様々な計算を実行できます。この方式は**プログラム内蔵方式**（stored program method）と呼ばれ、コンピュータの重要な基本原理のひとつとなっています。

1943 年に世界で最初に開発された電子式コンピュータ ENIAC では、プログラムとデータは峻別されていました。入力データはメモリに格納されたものの、プログラムは多数のスイッチの ON/OFF と部品間のケーブルの接続によって実現されていました。そのため、プログラムを入れ替えるにはスイッチの切り替えとケーブルの張り替えが必要とされ、1 回のプログラムの入れ替えに数日を要したようです（出展: Wikipedia）。プログラム内蔵方式は、ENIAC の次の世代のコンピュータ EDVAC によって実装されました。ドイツ系ユダヤ人研究者フォン・ノイマンがこの方式を提案したとされ、それに由来して、プログラム内蔵方式の今日のコンピュータをしばしばノイマン型コンピュータと呼んでいます。

15.3.3　チューリング完全

チューリング・マシンは有限状態オートマトン、プッシュダウン・オートマトンよりも高い記述力を有し、多種多様な処理を記述することができますが、チューリング・マシンよりもさらに大きな記述力、表現力を持つ機械は存在するのでしょうか。

20 世紀前半に多くの数学者によって様々な機械、数学体系が提案され、その表現力が研究されました。代表的なものとして以下のようなものがあります[5]。

チューリング・マシン ... 本章の主題です。

帰納的関数 ... 自然数の上の関数に対して、いくつかの計算上の仕組みを導入し、関数の表現力を論じるものです。

ラムダ計算 ... ラムダ式（λ 式）と呼ばれる表現形式に対して、いくつかの変形操作を定義し、様々な式の変形に関する性質を論じるものです。

タグシステム ... 文脈自由文法の生成規則に類似した規則を用いて記号列の変形を繰り返し、変形の性質を論じるものです。

[5]個々について詳細な述べませんので、興味を持った人は自身で調べてみてください。

興味深いことに、それぞれは独立して提案されたにも関わらず、これらはいずれも等価な記述力を持つことが分かっています。このことは、どのような手順や操作を用いようとも、機械的な手順や操作で記述できる内容[6]には共通の限界が存在することを示唆しています。ここで言う「記述力」は計算能力、計算可能性（computability）と換言することもできます。そこで、

計算可能である　＝　チューリング・マシンで計算できる

と見なそうという提案がなされました。この提案を、上記の等価性の研究に貢献した研究者の名前を冠してチャーチ＝チューリングの提唱（Church-Turing thesis）またはチャーチの提唱（Church's thesis）と呼んでいます。この提案は定理ではありませんから証明できません。「計算可能である」ことの定義を与えている（そう見なそうと提案しているに過ぎない）ことに注意してください。

チャーチ＝チューリングの提唱は、任意の計算可能な内容を実際に記述し、実行するにはチューリング・マシンを用いればよい、あるいはチューリング・マシンと等価なシステムを用いればよいことを意味しています。そのような等価なシステムのことを**チューリング完全**（Turing complete）[7]であると言います。

前節で述べたように、私たちが日頃使用しているノート PC（のハードウェアとそれを動かす機械語）はチューリング完全です。

プログラミング言語も計算を記述できるシステムと見なすことができます。まず、C/C++/Java/Ruby などのプログラミング言語はチューリング完全です。何故ならば、C++でチューリング・マシン（たとえばこの章で取り上げたチューリング・マシン）をシミュレートするプログラムを作ることは容易です。その意味で C++はチューリング完全です。世の中に知られているプログラミング言語のほとんどはチューリング完全であり、逆にチューリング完全でないものは一般にはプログラミング言語とは呼びません。ただしいくつかの反例として、Web 文書作成用の HTML（HyperText Markup Language）やデータベース言語 SQL （Structured Query Language が語源とされています）は一見するとプログラミング言語に似た構造を持っていますが、チューリング完全ではありません。

15.3.4　計算可能/計算不能

システムがチューリング完全であるならば、任意の計算可能な内容はそのシステムで実行できることが保証されます。ところで「計算可能」な内容とは具体的にどのようなものでしょうか。その答えは、既にコンピュータについて知識のある私たちにとっては単純明快です。「アルゴリズムが存在すること」です。「アルゴリズム」については 1 年生の「情報基礎概論」において以下のように学びました。

[6]「内容」とは、数学的には集合や関数を意味します。
[7]同じ意味で、チューリング計算可能（Turing computable）と呼ぶこともあります。

1. 手順通りに実行すれば、正しい答えを求めることができる。

2. 手順にあいまいさがない。また個々人の経験や感性に応じて手順の解釈が異なるということがない。

3. 手順の実行は、必ず有限回の操作で終了/停止する。無限に実行し続けることはない。

4. 手順は、特定のコンピュータ、プロセッサ、プログラミング言語を想定して作られていない。

1.、2. は当然の条件です[8]。3. は重要です。手順の中にはこの条件を部分的にしか満たさないものも考えられ、それをアルゴリズムと区別して、セミアルゴリズム（semi-algorithm）と呼ぶことがあります。ただし、4. の 条件は、この章の議論には馴染みません。この章では 4. の代わりに「手順をチューリング完全な特定のプログラミング言語で記述できる」という条件を用いることとします。

アルゴリズムの例として、任意に与えられた二次方程式 $ax^2 + bx + c = 0$ が実数解を持つか否かを判定する問題を考えましょう。これは計算可能な問題です。何故ならば、判別式 $b^2 - ac$ が 0 以上ならば YES、さもなくば NO と回答すればよく、明らかに有限の手順で答えが出るからです。別の例として、任意に与えられた自然数 a と b が互いに素であるか否かを判定する問題を考えましょう。これも計算可能な問題です。何故ならば、互いに素か否かはユークリッドの互除法を用いて a と b の最大公約数を求め、それが 1 であるか否かを判定すればよいからです。もちろんユークリッドの互除法は有限の手順で終了します。このように、計算可能な問題とそれを解決するアルゴリズムは膨大に存在します。

上に上げた二つの例はいずれも判定問題です。計算結果が YES/NO、1/0、真/偽などの 2 値で求められる問題 — 広い意味の判定問題 — のことを決定問題（decision problem）とも呼びます。ある決定問題に対するアルゴリズムが存在する場合にはその問題は決定可能（decidable）であると言い、さもなくば決定不能（undecidable）[9] であると言います。世の中には膨大な数の決定問題が存在し、その多数は決定可能です。上に挙げた 2 例以外にも、任意に与えられた命題論理式が恒真か否かを判定する問題、任意に与えられた自然数がメルセンヌ素数か否か判定する問題などは決定問題であり、いずれも決定可能です。

具体的な計算値 n を求める問題はそのままでは決定問題ではないのですが、それを「計算結果 n は定数 x 以下か否か」という問題へ置き換えることで決定問題へ読み替えることができます。そして、x を様々に変えて複数の決定問題を用意することで、n の値をいくらでも精密に求めることができます。たとえば $\sqrt{2}$ の値を計算する場合、まず「$\sqrt{2}$ は 2 以下か？」という問題を考えます。答えは YES です。そこで次に「$\sqrt{2}$ は 1 以下か？」という問題を考えます。答えは NO です。次に

[8] この授業では非決定性オートマトンを学びました。その動作はある意味で曖昧ですが、非決定的な動作を並列実行と解釈すれば、曖昧さは無くなります。

[9] 計算結果が YES、1、真の場合には停止するものの、NO、0、偽の場合には停止しないかもしれないアルゴリズム（これが上述のセミアルゴリズム）が存在する問題を半決定可能（semi-decidable）と言います。

「$\sqrt{2}$ は 1.5 以下か？」という問題を考えます。答えは YES です。次に「$\sqrt{2}$ は 1.25 以下か？」という問題を考えます。答えは NO です。このように、$\sqrt{2}$ の計算精度はいくらでも小さくすることが可能です。計算可能性を論じることは決定問題を論じることと本質的に同じなのです。

ところで、決定不能な問題、チューリング・マシンを用いても計算できない問題は世の中に存在します。たとえば 12.3 節において触れた文脈自由文法の曖昧性に関して、任意に与えられた文脈自由文法が曖昧か否かを判定する問題は決定不能であることが知られています。今日では様々な決定不能問題が存在することが知られています。このテキストでは決定不能問題の中で最も有名なチューリング・マシンの停止性問題（Turing Machine Halting Problem）[10] について解説し、この章およびこのテキスト全体を終えることにします。

15.3.5　停止性問題

プログラムのデバッグで難渋する場合のひとつが、開発したプログラムの実行が無反応になる場合です。正しい計算を行なっている途中なのか（よって、未だ答えが出ていないが、もう少し待てば計算結果が得られるか)、あるいは無限ループに陥っていて、今後、永遠にプログラムの実行が止まらないのか、判断が難しい場合があり、デバッグに苦労します。開発中のプログラムが無限ループを持つのか否かを自動的に発見してくれるソフトウェア・ツールがあると助かるのですが、計算可能性の理論はそのような便利なツールは理論上、存在しないことを明らかにしています。以下、そのことをチューリング・マシンをベースに背理法で証明します。

今、入力 S を受け取って、ある計算を行なうチューリング・マシン TM を考え、その実行を $TM(S)$ と表します。任意に与えられた S と TM について、$TM(S)$ が有限停止するか否かを判定できるチューリング・マシン H が存在すると仮定し、その実行を、ちょうど万能チューリング・マシンの場合と同様に、$\mathrm{H}(\mathbf{P}[TM], S)$ と表すことにします。なお、$\mathbf{P}[TM]$ は、TM を記号列に変換したものです（15.3.2 節参照)。ここで、$\mathrm{H}(\mathbf{P}[TM], S)$ の実行は有限停止し、

1. もし $TM(S)$ が停止するならば（そのときに限り） ... $\mathrm{H}(\mathbf{P}[TM], S)$ は YES を出力する
2. もし $TM(S)$ が停止しないならば（そのときに限り） ... $\mathrm{H}(\mathbf{P}[TM], S)$ は NO を出力する

と仮定します。上に述べたように、そのような H があると便利なのですが、残念ながらそのような H は理論上、存在しえないことを証明していきます。

まず、上の入力記号列 S は任意ですから、S の代わりに $\mathbf{P}[TM]$ を用いることもできるはずです。その場合、 H の実行は $\mathrm{H}(\mathbf{P}[TM], \mathbf{P}[TM])$ という奇妙な実行になりますが、それが現実世界でどのような意味を持つのかなどということは当面、気に掛けないことにします。

[10] 厳密に言えば、チューリング・マシンの停止性問題は半決定問題です。

次に、 H を改造した以下のチューリング・マシン M を作ります。この改造はわずかなので、もし H を作ることができる（H が存在する）ならば、必ず M も作ることができます。

1. もし H($\mathbf{P}[TM], \mathbf{P}[TM]$) が YES を出力するならば（そのときに限り） ... M($\mathbf{P}[TM]$) は停止しない [11,12]。

2. もし H($\mathbf{P}[TM], \mathbf{P}[TM]$) が NO を出力するならば（そのときに限り） ... M($\mathbf{P}[TM]$) は停止する [13]。

次に、M への入力として M を記号列化した $\mathbf{P}$[M] を与える実行 M($\mathbf{P}$[M]) を考えてみます。これも現実世界では奇妙な実行ですが、そこに気を取られること無く、数学的な意味のみを追求して行きます。

さて、私たちは、M($\mathbf{P}$[M]) が停止するか否かに注目します。これについて M の定義から

1. もし M($\mathbf{P}$[M]) が停止するならば ... H($\mathbf{P}$[M], $\mathbf{P}$[M]) が NO を出力します。

2. もし M($\mathbf{P}$[M]) が停止しないならば ... H($\mathbf{P}$[M], $\mathbf{P}$[M]) が YES を出力します。

次に、もとの H の定義から

1. もし H($\mathbf{P}$[M], $\mathbf{P}$[M]) が NO を出力するならば ... M($\mathbf{P}$[M]) は停止しません。（参考：もし H($\mathbf{P}[TM], S$) が NO を出力するならば ... $TM(S)$ は停止しません。）

2. もし H($\mathbf{P}$[M], $\mathbf{P}$[M]) が YES を出力するならば ... M($\mathbf{P}$[M]) は停止します。（参考：もし H($\mathbf{P}[TM], S$) が YES を出力するならば ... $TM(S)$ は停止します。

上の２段階の論理を縮めると

1. もし M($\mathbf{P}$[M]) が停止するならば ... M($\mathbf{P}$[M]) は停止しません。

2. もし M($\mathbf{P}$[M]) が停止しないならば ... M($\mathbf{P}$[M]) は停止します。

これは矛盾です。よって、任意に与えられた S と TM について、$TM(S)$ が有限停止するか否かを判定できるチューリング・マシン H が存在すると仮定したことが間違いであると結論されます。

停止性問題においてチューリング・マシンであることは問題の本質ではありません。プログラムをプログラムの引数にできるならば、どんなチューリング完全なプログラミング言語についても — もちろん、C/C++についても — 同様の性質が成り立ちます。

（以上）

[11]第一の改造は引数の数に関するものです。H の引数は二つですが、M の引数はひとつです。しかし、これは問題ではありません。H($\mathbf{P}[TM], \mathbf{P}[TM]$) の実行では二つの引数が同一であることに注意してください。M は実行直後に受け取った引数をテープ上にコピーすればよいのです。

[12]第二の改造では、H が YES を出力する動作部分を M では無限ループ（同じ状態を無限に遷移し続けるような有限状態部）に置き換えます。

[13]第三の改造では、H が NO を出力するときには M は停止するように改造します。