プログラミング－言語・基礎・実践－ 15－10
（１９９４．１．２１）

効率の良いトライ / 状態遷移機械の構成方式

増井俊之

シャープ株式会社 技術本部

ソフトウェア研究所

masui@shpcsl.sharp.co.jp

正規表現による状態遷移や辞書の記述から効率良い状態遷移表とその解釈プログラムを生成するシステム Flex について述べる。状態遷移表やトライを効率良く表現する手法は数多く提案されているが、その多くは遷移表やトライのノードが疎であるという条件を仮定しているため文字端末シーケンス解析表のように大規模かつ疎でない遷移表を効率良く実現することができない。Flex では状態遷移表の冗長部分の圧縮方式として各種の手法を組みあわせることにより、大規模なトライや状態遷移表のコンパクトな遷移表への圧縮を実現している。

Compressing State Transition Tables and Tries

Toshiyuki MASUI

Software Laboratories

Corporate Research and Development Group

SHARP Corporation

masui@shpcsl.sharp.co.jp

An efficient table compression method for state transition tables and tries is introduced. Although many techniques for compressing sparse tables and tries have been proposed, most of them cannot compress non-sparse transition tables even when they have redundant entries. We implemented Flex, a *lex*-like program which can almost always generate compact state transition tables for a wide range of finite state machines and tries.

1 はじめに

有限状態遷移機械は符号の復号 / パタンマッチング / 文字列処理 / 構文解析など幅広い応用に使用することができ、また大規模なものが必要になることも多いので、各種の状態遷移機械の効率的実装手法が提案されている。一方、辞書を表現するデータ構造としてトライ (trie) 構造が使われることが多い。トライは文字列や数などの集合 (辞書) を n 分木で表現するデータ構造で、木の根から受理ノードまでの経路が辞書中の文字列や数を示すようにしたものである。木の各枝が文字に対応するようにCのキーワードの集合の一部を 26 分木のトライで表現したものを図 1に示す。トライは木構造の状態遷移機械と等価である。

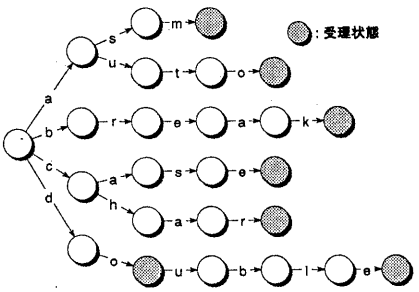


図 1: Cのキーワードを表現するトライ

状態遷移機械を作成する場合、UNIX では *lex* が広く使用されている。*lex* は構文解析プログラム作成システム *yacc* と組みあわせて用いられる字句解析プログラム生成システムであり、正規表現文字列を使用して状態遷移の記述を簡単に行なうことができるため一般の状態遷移機械生成器としても使うことができる。しかし *lex* は小規模で複雑な状態遷移機械を作成するには適しているが、もともとコンパイラの字句解析が主目的であるため入力文字を先読みして解釈するなど処理が複雑で動作も遅く遷移表も大きくなり、大規模な辞書トライなどの実装には適していない。

本稿では、状態遷移機械やトライを規模にかかわず効率良く実現するためのデータ構造及びその生成システム Flex [6][13] について報告する。Flex は *lex* と同様の正規表現によるトライや状態遷移機械の記述をメモリ効率 / 実行効率の良い状態遷移表に変換することができ、広い範囲のアプリケーションに使用することができる。

2 トライ及び状態遷移表の特徴

2.1 トライの統計的特徴

図 1の例にもみられるように、トライは以下のような特徴を持っているのが普通である。

1. 各ノードの子供の数は一般に少なく、ひとつだけのことが最も多い。
2. ひとつだけ子供を持つノードが連続することが多い。
   (例: 図 1の "r" → "e" → "a" → "k")

各種の辞書についてトライを構成したときの子ノードの数の割合を図 2に示す。

辞書	子ノード数			
	1	2	3	4
Cのキーワード (41 個)	88.51%	7.47%	2.87%	0.57%
UNIX のコマンド名 (250 個)	92.67%	4.11%	0.89%	0.54%
英単語 (25000 個)	84.47%	9.59%	2.76%	1.13%
ネットワーク記事 (2200 行) 中の単語	87.09%	7.78%	2.02%	0.84%

図 2: トライのノードの子供の数の割合

このように、自然言語やプログラム言語中の文字列の集合に対しトライを作成すると、辞書の規模によらず 85% 以上のノードが子供をひとつだけしか持たないことがわかる。図 1のような 26 分木の各ノードは最大 26 個のポインタを持つはずであるが、実際はほとんどのノードはポインタをひとつだけしか持つ必要がないため、実装方式を工夫することにより効率良くメモリ上に実現することができる。

2.2 状態遷移表の特徴

ANSI の文字端末制御シーケンス [2] の一部を解釈する状態遷移機械を図 3に示す。例えばカーソル上方移動を指令するシーケンスは "ESC"(0x1b), "["(0x5b), (移動量), "A"(0x41) であり、移動量 (省略可能) は "0" から "9" の数字の列で示される。

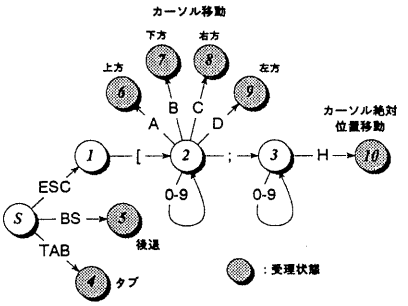
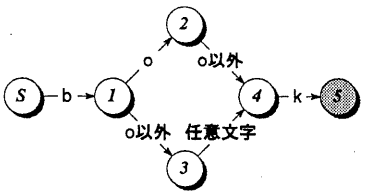


図 3: カーソル移動を認識する状態遷移機械

状態遷移表は (状態の数×入力文字種) の表で表現することができるが、上述の端末シーケンスのような場合、意味のあるシーケンスの数は文字種に比べ少ないため表は疎となる。LR 構文解析表のように自動的に生成される遷移表の場合においても同様に状態遷移表は疎となることが多いため、疎な表の圧縮手法を遷移表に適用して状態遷移機械を実現することが広く行なわれている。

2.3 既存の圧縮手法の問題点

既存のトライや状態遷移表の圧縮手法は、トライ及び状態遷移機械のひとつのノードから次のノードへの出力枝の数が少ないことが前提となっている。辞書トライの場合はこの前提はほぼ常に正しいと考えられるが、この条件を満たさない状態遷移機械が必要になることも多い。例えば「"b" で始まり "k" で終わる 4 文字の単語で "book" 以外のもの」を認識する状態遷移機械と遷移表は図 4のようになるが、このような遷移表は単純であるにもかかわらず遷移表が疎であることを仮定した手法ではうまく圧縮することができない[1]。



入力文字

状態	a	b	c	…	k	…	o	…	y	z
S	-	1	-	…	-	…	-	…	-	-
1	3	3	3	…	3	…	2	…	3	3
2	4	4	4	…	4	…	-	…	4	4
3	4	4	4	…	4	…	4	…	4	4
4	-	-	-	…	5	…	-	…	-	-

図 4: "b??k" という単語で "book" 以外のものを認識する状態遷移機械と遷移表

このような遷移表は疎ではないが単純で冗長性をもつため、その性質を利用すると表を圧縮することが可能である。

3 状態遷移機械生成器 Flex

Flex は正規表現による状態遷移機械やトライの表現を効率良い遷移表及びその解釈プログラムに変換するシステムである。トライや状態遷移機械の一般的な特徴を生かしているためほとんどの場合にコンパクトな遷移表を作成することができる。

3.1 Flex の構文

状態遷移及び受理時の動作は lex と似た構文で指定する。プログラミング言語としては C 及び Perl [10][13] が使用できる。Flex の構文を以下に示す。

```
<前方定義部>
%%
<パタン 1>      <アクション 1>
<パタン 2>      <アクション 2>
...
%%
<後方定義部>
```

ふたつの "%%" で囲まれた正規表現パタンとアクションの組の集合が遷移表及び解釈プログラムに変換されて出力され、前方 / 後方定義部はそのまま出力プログラムにコピーされる。アクション部は使用するプログラミング言語で記述する。パタン 1、パタン 2、... が認識されたとき対応するアクションが実行される。

3.2 Flex の使用例

Flex では図 4の状態遷移機械は以下のように記述する[2]。

```
%%
b(o[^o]|[^o].)k   { print "Nonbook\n"; }
%%
```

このソースを Flex に与えると以下のようなアクション実行関数、遷移表及び遷移表の解釈プログラムが出力される。

```
...
sub zzaction1 { print "Nonbook\n"; }
@out = ( # 遷移表
  0x61, 0x62, 0x51, 0x6f, 0x09, 0x71, 0x21,
  0x6b, 0x0d, 0x11, 0x6f, 0x00, 0x06,
);
sub zztrans { # 遷移表解釈関数
   local($c,$n) = @_;
...
```

このプログラムを実行すると、"b" で始まり "k" で終わる 4 文字の単語で "book" 以外のものを認識した場合のみ "Nonbook" が印刷される。

3.3 状態遷移表の表現法

Flex では 2節で述べたトライや状態遷移機械の特徴を利用し、広い範囲の応用において状態遷移表を圧縮できるようにするために以下のような遷移表の表現方式を採用している。

1. **遷移図のノードの性質に応じて異なる表現型式を用いる**

   2.1節で述べたように、トライの根に近い部分は各ノードの子供の数が多いのに対し、葉に近い部分ではほとんどのノードは子供をひとつしか持たず、またそのよ

[1] この例では文字を ["b", "k", "o", それ以外] のようにカテゴリ化することにより既存の手法を適用することも可能であるが、複雑な状態遷移機械の場合そのようなことは困難である。例えば図 3では "0" から "9" をまとめてひとつのカテゴリとして扱うことが可能であるが、遷移機械の別の部分においてある数字を特別に扱う必要があった場合はその数字はまた別のカテゴリとしなければならなくなる。ANSI 端末シーケンスのような場合結局ほとんど全てのコードを別カテゴリとしなければならなくなるためカテゴリ化のメリットはほとんど無い。

[2] 正規表現は UNIX の grep や *lex* などで使われる標準的なものを使用している。例えば "[^o]" は o 以外の任意の 1 文字を示し、"." は任意の 1 文字を示す。

うなノードが連続することが多いが、このように性質の異なるノードに対しては異なる表現方法を採用する方が効率が良い。例えば青江のトライ圧縮法 [12] では子供をひとつだけ持つノードの連続を文字列として表現する (例えば図 1の "r"' → "e"' → "a"' → "k" という遷移を "reak" という文字列で表現する) ことにより効率化を図っているし、パトリシア木 [7] でもこのようなノードの連続を特別に扱っている。

Flex ではノードの子 (出力枝) の数に応じて 3 種類の表現型式を使用している。子供の数がひとつだけの場合は [12] と同様に上述の文字列表現を用いる。子供の数が少ないときは入力文字とそれに対応する遷移先を組にしたものを並べる。子供の数が多いときは、遷移をおこす文字をビットフィールド表現したものの後に遷移先のリストを並べる (図 5)。ビットフィールド表現は Maly による圧縮トライ C-trie[5] や Purdin のトライ圧縮手法 [8] でも使用されている方法である。

aビット: ノードが受理状態であることを示す
pビット: ノードの最後がポインタか次のノードかを示す
ccccビット: 要素数(n)
ec: 要素数が16以上の場合に使用(このときccccは0)
Ck: k番目の文字
Pk: ノードへのポインタ
Nk: 次のノード
Pe/Ne: デフォルト遷移に対応するノード/ポインタ
dir: 文字集合のビットマップ表現

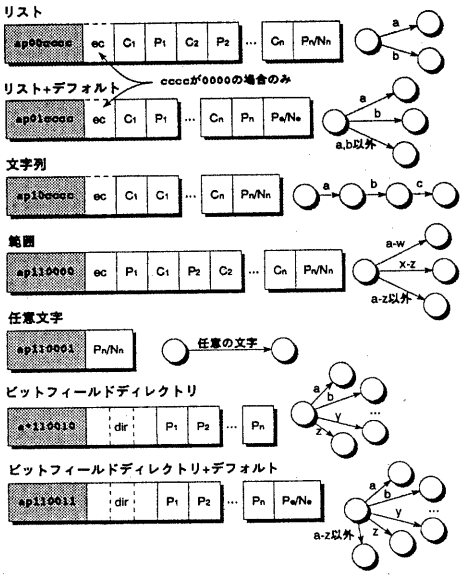


図 6: Flex のノード表現方式

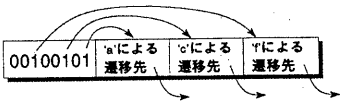


図 5: ビットフィールドによるトライのノード表現

2. **デフォルト遷移の導入**

状態遷移機械を構成する場合、図 4の例のように特定の文字以外の入力文字に対する遷移先を指定しなければならないことが多い。このような場合に特定の文字以外の全文字に対する遷移先を指定しなくてもよいようにするため、デフォルトの遷移先を指定できるようにする。

3. **文字集合 (文字クラス) 指定の導入**

状態遷移機械では「英字」「数字」のように文字の集合を指定して遷移を定義する必要があることも多いため、指定された範囲の文字に対する遷移先も指定できるようにする。

以上の方針を組みあわせて、Flex では図 6のようなノード表現方式を用いている。ノードの先頭バイトによりノードの種別を区別する。遷移先のノードがすぐ後に続く場合は先頭バイト中の "p" ビットを "1" として次ノードへのポインタを省略することにより圧縮効率を上げている。ポインタ P_k の値が表の大きさを上回るときは受理状態を示し、その値と表の大きさの差が受理番号となるようにしている。

3.2節の遷移表の解釈は図 7のようになる。第 1 バイトは 0x61 なので 1 文字 (ここでは "b") が後に続くことを示しており、かつ "p" ビットが "1" なのでその後のバイトが次のノードになっていることを示している。また第 3 バイトは 0x51 なので、遷移を起こす文字のリストとその遷移先及びデフォルトの遷移先ノードが後に続くことを示している。

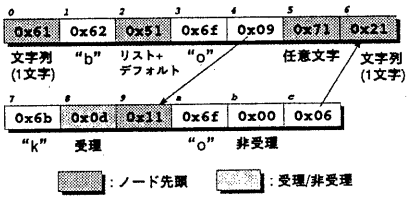


図 7: 3.2節の遷移表の解釈

4 既存の圧縮手法との比較

本節ではトライ及び状態遷移表の既存の圧縮方式を紹介し、Flex との比較を行なう。

4.1 トライの木構造の構成手法

"case" と "base" というふたつの文字列をトライで表現する場合、文字列の先頭を根とする場合と後尾を根とする場

合では、図8のように必要なノードの数に大きな違いが生じる。

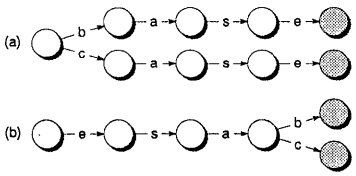


図 8: トライの経路の制御

このように、同じ集合を表現するトライの構築法は何種類もあるので、Comer[4] は効率良いトライを生成するためにどのような分岐を優先すればよいかについての各種のヒューリスティクスを提案している。またどういう方針で子供の分岐が行なわれるかの情報を親ノードの中に格納する O-Trie という手法も提案している。

4.2 トライのノードを圧縮する手法

2.1節で述べたように、トライの各ノードを素直に実装するとほとんどが空のポインタになってしまうため、各種のノード圧縮手法が提案されている。

Maly[5] は C-trie というデータ構造を提案している。トライのノードは子供のビットフィールド表現 / 終端であるかどうかのフラグ / 子供がいるかどうかのフラグ / 子供のオフセットから構成される。ノードの子供の位置は、ビットフィールドから子供の存在を確認した後オフセットとビット位置から計算する。

Purdin[8] はトライの子供ノードをビットフィールドで表現して圧縮する別の手法を提案している。英文字 26 文字のビットフィールド表現に 4 バイトを使用するが、これらは通常疎となるため、最初の 3 ビットを残りのバイトの有無の判定に使用する。次ノードへのポインタは相対値で表現し、ポインタも圧縮表現する。

青江 [12] は「ダブル配列」というデータ構造でトライのノードを効率良く表現する手法を提案している。ここでは図 9のように base[], check[] というふたつの同じ大きさの配列を使用する。

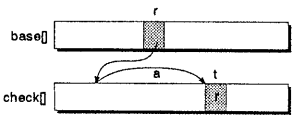


図 9: 青江の「ダブル配列」

トライのノードは配列のインデクス (図 9の r, t) で表現される。あるノードを示すインデクスが r であるとき、文字 a に対して子ノードのインデクス候補 t = base[r]+a を計算し、check[t] ≡ r が成立するとき、r の文字 a に対応する子ノードを t とする。ひとつのノードを表現するには、base[] を 1 エントリと、その子ノードへのポインタを表現するのに子ノードの数だけ check[] のエントリを使用する。使用されていない部分 (図 9の配列の白い部分) は別のノードのために使用可能であるため、灰色の部分が重なりあわないようにうまくずらして配置することにより効率良くトライを実装することができる。また、子供をひとつだけしか持たないノードの連続は文字列として特別に取り扱うことによりさらに効率化をはかっている。

4.3 疎行列の圧縮手法

Aho らの教科書 [1] では、図 10のような 3 個の配列を使用して状態遷移表をコンパクトに表現する手法が解説されている。ここでは状態 r が入力文字 a により状態 n に遷移可能であることを示している。青江の手法と同様に、灰色の部分を適当にずらして重ねあわせて配列に格納することにより、疎な遷移表を小さな配列に格納することができる。この手法は *yacc* の出力する構文解析表で使用されている。

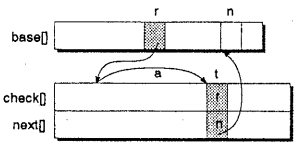


図 10: Aho らによる状態遷移表の圧縮方式

Tarjan[9] は一般の疎な表をコンパクトに表現する手法を提案している。基本的には上の Aho らの手法と同様に疎な表の行を重ねあわせて圧縮を行なうが、表の中に疎でない行が存在するためうまく行を配置できない場合でも桁データを行方向にシフトさせて疎にすることにより Aho らの手法が適用可能になるようにしている。

青江 [11] は [12] を拡張して一般の状態遷移機械の圧縮に適用している。[12] の手法 (4.2節参照) では全てのノードが別々のインデクス値を持つことが前提になっているため、複数の遷移経路を経て同じ状態に遷移することのある状態遷移 (木構造で表現できない状態遷移) を表現することはできなかったが、[11] では木構造でない状態遷移を木構造の状態遷移と ε 遷移の合成と考えることによりトライの場合と同様の手法を一般の状態遷移機械に適用している。ε 遷移を表現するために base[] の 1 エントリが余分に必要となる (図 11)。

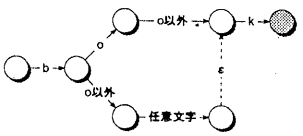


図 11: 図 4の遷移図の [11] による表現

4.4 Flexの手法との比較

上にあげた各種の手法は、ひとつの統一的な方式でトライや状態遷移機械をコンパクトに表現しようとしているため、遷移表が疎でない場合などのようにその方式における圧縮の前提が成立しない場合に効率がひどく悪くなるという欠点がある。Flexの手法は、圧縮効率が他の方式に比べて優れていることに加え、各種の表現方法を併用することによって、実際の様々な応用において必要となるトライや状態遷移機械のほとんどを効率良く圧縮表現することが可能となっている。青江の手法[12]においても、効率の良さはダブル配列と文字列表現を併用していることに因るところが大きい。

Flexでは4.1節で述べたようなヒューリスティクスは使用していない。このようなヒューリスティクスは、データベースをトライで表現する場合のように選択するキーの性質によりノードの性質が大きく変わる場合は有効であるが、Flexが対象とするような文字列集合を表現する場合には有効でないからである。

5 Flexの適用例

Flexにより図4のような小規模な状態遷移機械の効率良い遷移表が作成できることを既に述べたが、本節ではFlexが広い範囲の辞書や状態遷移機械の遷移表も効率良く作成できることを実例により示す。

5.1 トライへの適用

青江[3]はダブル配列と文字列表現を併用した方式で22個のコマンド名[3]を136バイトの領域(ダブル配列84バイト+文字列領域52バイト)に格納できることを示しているが、Flexの表現方式では同じデータを124バイトに格納することができる。大規模な辞書トライも同様にコンパクトに表現することができる。

5.2 ローマ字かな変換

Flexを使用するとローマ字かな変換のような文字列処理プログラムを以下のような簡単な記述から作成することができる。

```
%%
a    { 'あ'; }
ba   { 'ば'; }
bb   { &ungetc('b'), 'っ'; }
be   { 'べ'; }
bi   { 'び'; }
...
zyo  { 'じょ'; }
zyu  { 'じゅ'; }
zz   { &ungetc('z'), 'っ'; }
%%
```

[3]adb, apply, apropos, ar, as, at, awk, basename, bc, biff, binmail, cat, cc, ccat, cd, checkeq, checknr, chfn, chgrp, chmod, chown, chsh の22個

"b"のような子音文字をふたつ認識した場合は"っ"を出力するが、2個目の文字を先読みしているためungetc()で入力に戻すようにしている。変換の記述は168行であり遷移表は約600バイトとなる。

5.3 端末エスケープシーケンス解析

図3の状態遷移機械は以下のように記述できる。

```
%%
\x09                    { "TAB"; }
\x08                    { "BS"; }
\x1b\[[0-9]*A           { "UP"; }
\x1b\[[0-9]*B           { "DOWN"; }
\x1b\[[0-9]*C           { "RIGHT"; }
\x1b\[[0-9]*D           { "LEFT"; }
\x1b\[[0-9]*;[0-9]*H    { "MOVE"; }
%%
```

共通のプレフィクスはマージされて図3のように3個の内部状態と7個の受理状態が作られる。Flexの出力する遷移表は図12のように40バイトで表現される。

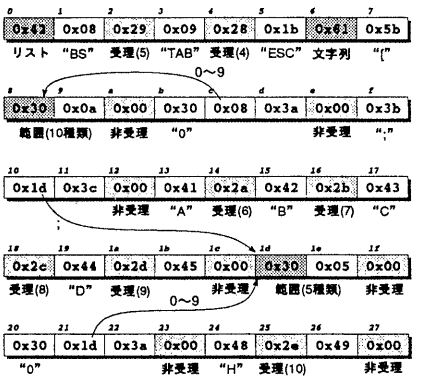


図12: ANSI端末シーケンスの遷移表

同じ種類の文字(e.g. 数字)をカテゴリ化した後で4.3節で述べたAhoの手法を適用するとbase[]に11バイト(状態数)、check[]とnext[]にそれぞれ17バイトが必要であり、加えてカテゴリ化のための表(単純に実現すると256バイト)が必要となる。また青江の手法[11]では最低でも枝の総数の大きさをもつ配列が2個必要となるが、図3の枝の総数は30なので最低60エントリが必要となる。大規模な端末シーケンスに適用する場合はさらに差は大きくなる。付録に示した69種の端末シーケンスから状態遷移機械を作成すると状態遷移のノード数は253個となるが、Flexの遷移表が1096バイトで構成できたのに対し上述のAhoの手法では2632バイトを要する。また枝の総数は37527個にもなるため青江の手法を単純に適用すると70000エントリ以上の領域が必要になってしまう。

5.4 逆アセンブラ

最後に8ビットマイコンの逆アセンブラをFlexで記述した例を示す。この例では228行の記述が1640バイトの遷移表に変換される。

```
%%
[\x08-\x09].        { "MV r,n"; }
[\x0a-\x0b]..       { "MV r,n"; }
[\x0c-\x0f]...      { "MV r,n"; }
[\x80-\x87].        { "MV r,(n)"; }
[\x88-\x8f]...      { "MV r,[lmn]"; }
[\x98-\x9e]\x00.    { "MV r,[(n)]"; }
[\x98-\x9e]\x80..   { "MV r,[(m)+n]"; }
[\x98-\x9e]\xc0..   { "MV r,[(m)-n]"; }
...                 ...
\xde                { "HALT"; }
\xdf                { "OFF"; }
%%
```

6 実装

FlexはPerlで約700行のプログラムである。Flexは正規表現の記述からまず非決定性状態遷移機械を作成し、決定性状態遷移機械に変換した後、各状態からの出力枝の数に応じてノードの表現型式を選択して遷移表を生成する。実装の詳細は[13]に解説されている。

7 結論

トライ及び状態遷移機械の一般的な特徴にもとづいて単一の表でこれらを効率よく実装する手法を示した。本手法はトライによる自然言語辞書、端末制御シーケンス解析表など幅広い分野において効率良い状態遷移機械を作成するのに有効である。

参考文献

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] American National Standard Institute, New York, NY. *Additional Controls For Use With American National Standard Code For Information Interchange*, ANSI X3.64–1979.

[3] Jun'ichi Aoe. An efficient implementation of string pattern matching machines for a finite number of keywords. *SIGIR Forum*, Vol. 23, No. 3,4, pp. 22–33, Spring/Summer 1989.

[4] D. Comer. Heuristics for trie index minimization. *ACM Transactions on Database Systems*, Vol. 6, No. 3, pp. 383–395, September 1979.

[5] Kurt Maly. Compressed tries. *Communications of the ACM*, Vol. 19, No. 7, pp. 409–415, July 1976.

[6] Toshiyuki Masui. User interface specifacation based on parallel and sequential execution specification. In *USENIX'91 Conference Proceedings*, pp. 117–125, January 1991.

[7] Donald R. Morrison. PATRICIA – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, Vol. 15, No. 4, pp. 514–534, October 1968.

[8] Titus D. M. Purdin. Compuressing tries for storing dictionaries. In H. Berghel, J. Talburt, and D. Roach, editors, *Proceedings of the 1990 Symposium on Applied Computing*, pp. 336–340. IEEE, ACM, April 1990.

[9] Robert Endre Tarjan and Andrew Chi-Chih Yao. Storing a sparse table. *Communications of the ACM*, Vol. 22, No. 11, pp. 606–611, November 1979.

[10] Larry Wall and Randal L. Schwartz. *Programming Perl*. A Nutshell Handbook. O'Reilly & Associates, Inc., Sebastopol, CA, 1991.

[11] 青江順一. ダブル配列による有限状態機械の効率的インプリメンテーション. 電子情報通信学会論文誌D, Vol. J70-D, No. 4, pp. 653–662, April 1987.

[12] 青江順一. ダブル配列による高速ディジタル検索アルゴリズム. 電子情報通信学会論文誌D, Vol. J71-D, No. 9, pp. 1592–1600, September 1988.

[13] 増井俊之. Perl書法. アスキー出版局, July 1993.

付録: 端末シーケンスと状態遷移表

5.3節で使用した端末シーケンスを以下に示す。

```
%%
\000                                { "Null"; }
\007                                { "Bell"; }
\010                                { "BS"; }
\011                                { "HT"; }
\012                                { "LF"; }
\015                                { "CR"; }
[\040-\177]                         { "ANK/JIS"; }
[\240-\337]                         { "Kana"; }
[\201-\237]|\340-\357][\100-\176|\200-\374]  { "SJIS"; }
[\360-\374][\100-\176|\200-\374]     { "SJIS"; }
[\200|\375-\377]                    { "F/F,etc."; }
\033\$.                             { "KanjiIn"; }
\033\(.                             { "KanjiOut"; }
[\021\023]                          { "DC1,DC3"; }
\033\[[0-9]*A                       { "Up"; }
\033\[[0-9]*B                       { "Down"; }
\033\[[0-9]*C                       { "Right"; }
\033\[[0-9]*D                       { "Left"; }
\033\[[0-9]*;[0-9]*[Hf]             { "Move"; }
\033\[[Hf]                          { "Home"; }
\033E                               { "NewLine"; }
\0337                               { "SaveCursorPos"; }
\0338                               { "RestoreCursorPos"; }
\033D                               { "CursorDown"; }
\033M                               { "CursorUp"; }
\033\[?25h                          { "CursorVisible"; }
\033\[?25l                          { "CursorInvisible"; }
\033\[0?K                           { "ClearLineFromCursor"; }
\033\[1K                            { "ClearLineToCursor"; }
\033\[2K                            { "ClearLine"; }
\033\[0?J                           { "ClearScreenFromCursor"; }
\033\[1J                            { "ClearScreenToCursor"; }
\033\[2J                            { "ClearScreen"; }
\033\[?3;[0-9]*;[0-9]*K             { "PartialEraseLine"; }
\033\[?3;[0-9]*;[0-9]*;[0-9]*;[0-9]*J  { "PartialEraseScreen"; }
\033\[;?[0-9]*m                     { "SetCharAttribute"; }
\033\[?5;[0-9]*;[0-9]*K             { "DrawKeisen"; }
\033\[[0-9]*;[0-9]*r                { "SetScrollArea"; }
\033\[r                             { "ResetScrollArea"; }
\033\[?5;[23].K                     { "Buzzer/LED"; }
\033&\[$\000[\200-\377]\201\000     { "ControlSensorPanel"; }
\033\\cA                            { "InitGraphics"; }
\033\\cC                            { "ClearScreen"; }
\033\\cU                            { "ResetColor"; }
\033\\cZ                            { "EndGraphics"; }
\033\\p[LE].....                    { "Line"; }
\033\\p[PRS]\000.(....)*\377\377    { "Rect,etc."; }
\033\\p[CFD].....                   { "Circle,etc."; }
\033\\pA....[\034-\376]*\033        { "Text"; }
\033\\pK....[\040-\376]*\033\033    { "Kanji"; }
\033\\s[IA]..                       { "Pickid"; }
\033\\s[LFMHGZT]..                  { "GType"; }
\033\\s[OPDEA]..                    { "Sopen,Pictr"; }
\033\\s[CF]                         { "Sclose,Delpca"; }
\033\\[RMGQ]....                    {}
\033\\sN.....                       {}
\033\\v[VWZ].....                   { "Tvport,Twind,Hzoom"; }
\033\\sM....                        { "CharSize"; }
\033\\sS..                          { "ChangeSpace"; }
\033\\b[MCE]....                    { "Sgvis,Sgbln,Sgdet"; }
\033\\sQ..                          {}
\033\\b[VBD]..                      {}
\033\\bT.....................       {}
\033\\nW.....                       { "Bdout"; }
\033\\iJ.....                       {}
\033\\jA..                          {}
\033\\bU.....................       { "Sgtr2"; }
\033\\b[XFRWCE]....                 { "Pcvis,Pedet"; }
\033\\iS....                        { "Setpic"; }
%%
```