# オートマトンの圧縮配列表現と言語処理系への応用

前田敦司[†]　水島宏太[†]

決定性有限状態オートマトン (DFA) は言語処理系の字句解析器や文字列のパターンマッチ・検索アルゴリズムなどの基礎として広く用いられている．DFA をエミュレートする速度を落とすことなくメモリ効率よく表現する手法もさまざまに工夫されてきた．

本稿では，DFA の状態遷移表を，複数の配列を用いて効率よく表現する手法のバリエーションを紹介する．この手法は，trie を 2 つの一次元配列に圧縮して格納する青江のダブル配列法と似て，状態遷移表の遷移リンクを 2 つの一次元配列に圧縮して格納する．ただし，ダブル配列法とは違って木構造だけでなく，DFA の遷移を表現するのに必要な一般のグラフ構造も表現できる．

筆者らはこの手法を，ネットワークを経由する攻撃を検知する侵入検知システム (NIDS) のパターン検索エンジンに用いるために開発したが，本稿では同様の手法を言語処理系の字句解析器に応用し，その効果を実験を用いて確かめた．

実験の結果，提案手法は，従来の圧縮手法と同等のメモリ効率を持ちながら，より高速であった．

# A compressed-array representation of automata and its application to programming language.

MAEDA Atusi[†] and Kouta Mizushima[†]

Deterministic finite automata (DFA) serves as a basic tool for wide variety of computer algorithms including lexical analysis of programming languages and string pattern matching/searching. Various methods are proposed for representing DFA compactly, without sacrificing emulation speed.

In this paper, we describe a variation of multi-array representation of DFA transition tables. Like Aoe's double-array method, which stores a trie into two linear arrays, our method stores transition link of DFA into two linear arrays. Unlike double-array method, however, our method can express general graph structures, which is required to express DFA transitions.

Authors developed this method primary for use in pattern matching engines in network intrusion detection system (NIDS). In this paper, we applied the method to lexical analyzers and performed some performance measurement.

Experiments shows that our method has spece efficiency comparative to existing methods, while achieving superior performance.

## 1. はじめに

決定性有限状態オートマトン (DFA)[4] は，ある種の状態機械の定式化であり，その受理する言語である正規言語および，その文法である正規文法などとともに，様々な文字列処理アルゴリズムの理論的基礎となっている．与えられた正規文法から，その文法が生成する言語を受理する DFA を構成する手法や，DFA の状態数を最小化する手法はよく調べられており，また，DFA はソフトウェアによって高速にエミュレートできるため，言語処理系の字句解析[7] や文字列のパターンマッチ・検索[1] などの広い範囲で応用されている．

DFA をソフトウェアでエミュレートする際の実装には，メモリ効率と速度のトレードオフが存在するが，本稿ではこのトレードオフを改善する手法を示し，字句解析器として用いた場合について性能の評価を行なう．

速度を大きく低下させることなくメモリ効率を改善するために，lex[5] や flex[6] などの生成する字句解析器が用いている手法に，状態遷移表内の冗長な遷移を省略し，疎になった状態遷移表を一次元の配列に圧縮して格納する方法がある． 本稿で述べる手法は，この方法をもとにして，速度を向上させるものである．

これらに関連の深い手法は，デジタル検索木の一種である trie[3] の圧縮手法として用いられており，説明を用意にする都合上，本稿ではまず trie の圧縮技法について述べる．

† 筑波大学大学院システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba

## 2. 辞書 trie とその圧縮表現

複数の文字列を格納した辞書を木構造で表す際に，文字によってラベル付けされた辺を根からたどった時のパスによって格納された文字列を表すことにする．木構造のそれぞれの節には，その節を終端とする文字列が辞書にあるかどうかを示す 1 ビットの情報をもたせる．たとえば，**図 1** は，この約束にしたがって文字列「f」「face」「fat」「hi」を含む辞書を表す木構造である．二重丸で表した節は，その節を終端とする文字列が辞書中に存在することを表している．

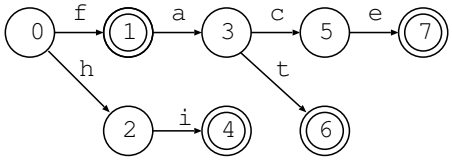


**図 1** 辞書を表す木構造

このような木構造の節を，入力シンボルに一対一対応する自然数（たとえば文字コード）でインデックス付けした配列で表し，配列の要素として次の節へのリンクを格納することにしたものが trie である．trie の節にも自然数を対応付け，リンクを節番号の数値で表現することにすると，trie は 2 次元配列で容易に表現できる (**図 2**)．

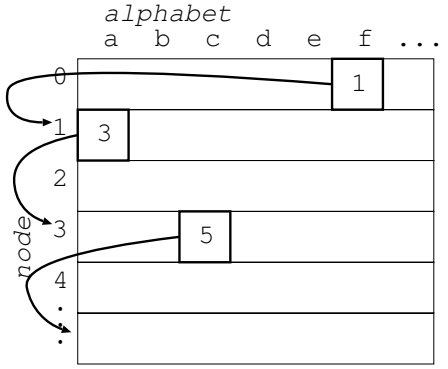


**図 2** Trie を表す 2 次元配列

図 2 の配列 `table[M][N]`(ただし，`M` は節の数，`N` はアルファベットのサイズ) において，節 $i$ から文字コード $j$ に対応するラベルの付いた辺が指す節は `table[i][j]` で高速にアクセスできる．そのような辺が存在しない場合には，`table[i][j]` に節番号と区別できる値 (たとえば負の値) を入れておけばよい．しかし，一般に，有効な辺の数は trie 全体の容量と比べてかなり少ないため，メモリ効率が悪い．

### 2.1 簡易 Johnson 法による trie の圧縮

後に述べる Johnson の方法[7] は，複数の一次元配列を用いて状態遷移表を圧縮して格納するもので，`yacc` や `lex` などの言語処理系で用いられている．ここでは，3 つの配列を用いる簡略化した形で，trie のリンクを格納する方法について概要を述べる．この方法では，もとの 2 次元表の内容を格納する配列 `next[K]` と，もとの行が `next` 中のどこに移ったかを示す `base[M]`，`next` の値がもとはどの行から来たものかを記録する `check[K]` の 3 つの配列を用いる．`next` と `check` は，互いに等しい大きさの配列で，もとの行の中の必要なデータが衝突しないように，またなるべくデータが密になるように `next` にデータを格納していく（図 3）．K は，このように詰めた結果を格納するのに十分な大きさの値でなければならない．

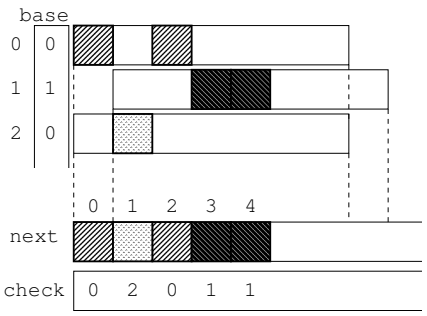


**図 3** Johnson の方法 (簡易版)

もとの $i$ 行目のデータは `next[base[i]]` から `next[base[i] + N - 1]` の範囲に格納されているので，もとの 2 次元配列の要素 `table[i][j]` のかわりに `next[base[i]+j]` をアクセスすればよい．ただし，他の行からのデータでないことを確認するため，`check[base[i]+j]` が `i` に等しいかどうかを確かめる必要がある．もし等しくない場合は，その辺は存在しないと判定する．

### 2.2 青江の方法

青江のダブル配列法[2] は，上述した簡易 Johnson 法から配列の数を 2 つに減らし，アクセス速度も改善した方法である．木構造では，行 $i$ を指す `next` 中の要素は（根を除き）ちょうど 1 つだけ存在し，`base` 配列の要素と一対一に対応する．すなわち，簡易 Johnson 法において `next[`$n_i$`]=i` であるような $n_i$ は (根の行

を 0 とすると) 任意の $i > 0$ についてただ 1 つ定まるので，ここに `base[i]` の値 $b_i$ を格納して，`next` と `base` をまとめてしまう（図 4）．

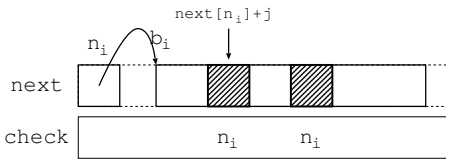


**図 4** 青江の方法

`table` の行 $i$ に対応する `next` 中の位置 $n_i$ から入力文字コード $j$ に対応する辺をたどった先は `next[n`$_i$`]+j` で求まる☆．辺が存在するかどうかのチェックは `check[next[n`$_i$`]+j]` と `n`$_i$ の比較による．配列アクセスが 1 回減るので，Johnson の方法より速度も向上する．もとの 2 次元配列の行番号 $i$ と，青江の方法で節を identify するために用いる番号 $n_i$ は異なる番号になるが，一対一に対応するので根から順にたどる限りは問題なくアクセスできる．ただし，この方法は Johnson 法と違って，木構造にしか適用できない．

## 3. DFA の状態遷移表の圧縮表現

### 3.1 Johnson 法による状態遷移表の圧縮

DFA の状態遷移関数 $\delta(s, a)$ を表現する際にも，素朴に 2 次元配列 (状態遷移表) を用いて状態番号 $i$ からシンボルのコード $j$ での遷移先 $\delta(i, j)$ を 2 次元配列の要素 `table[i][j]` で表すことができる．

DFA の状態遷移関数は全域的であり，`table` の全ての要素に有効な遷移先が格納されているが，通常は冗長な遷移先が多く含まれるため，これを利用して格納する情報を減らし，表のサイズを圧縮することができる．

状態遷移表に含まれる冗長性には以下のようなものがある．

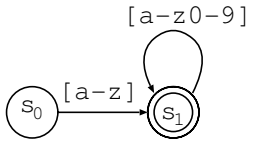


**図 5** DFA の状態遷移図の例

(1) 多くの DFA には，受理状態への経路を持たない状態 (jam 状態) が含まれる．たとえば，**図 5** は正規表現 $[a-z][a-z0-9]*$ で表される言語を受理する DFA の状態遷移図であるが，受理状態へ至る可能性のある遷移以外は省略されている．通常は省略されている遷移や jam 状態も明示的に描くと**図 6** のようになる．アルファベット $\Sigma$ のサイズが大きい時，この DFA の状態遷移表はほとんどが jam 状態 J への遷移を含むものになる．

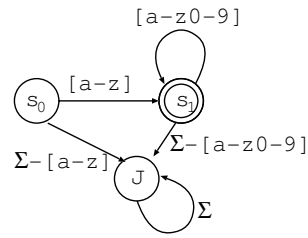


**図 6** jam 状態を省略せずに描いた状態遷移図の例

(2) DFA の状態には，ほとんどのシンボルについて同じ遷移先へ遷移するような組がある．たとえば図 6 の状態 $s_0$ と $s_1$ からの遷移先は，$[0-9]$ に関する遷移を除いて同じになっている．すなわち，2 次元配列に格納した場合，2 つの行の内容はほとんど同じになる．

(3) シンボルの中には，DFA 全体を通じて常に同じ扱いがなされるものがある．たとえば図 6 の DFA で，どの状態においてもシンボル a と z に関する遷移先は同じである．

オリジナルの Johnson の方法は，上の 2 番めの冗長性を利用したものである．2.1 節で述べた 3 つの配列 `base[M]`, `next[K]`, `check[K]` に加え，ある状態と似た「デフォルトの状態」を表す配列 `def[M]` を用いる．`def[i]` には，状態 $i$ と遷移先の多くが共通する状態の番号を入れておき，$i$ 行めのデータのうち `def[i]` と同じものについては `next` に格納しないことにする．これにより，2.1 節と同様に配列を圧縮して格納することが可能になる．

また，flex では，ある状態 $i$ が受理状態かどうかと，受理状態である場合に実行されるアクションの番号を格納する配列 `accept[M]` を用いている．

**図 7** に，Johnson 法を用いて DFA をエミュレートするプログラムの概略を示す．Flex で圧縮オプション `-C` を指定した場合には，これと同等のコードが生成される．

☆ 文献 2) では，ここでいう `next` を `base` と呼んでいる．

```
i = start_state;
do {
  j = *input_ptr++;
  if (accept[i]) {
    last_pos = input_ptr - 1;
    last_accept = i;
  }
  while (check[base[i] + j] != i) i = def[i];
  i = next[base[i] + j];
} while (i != JAMSTATE);
```

**図 7** Johnson 法を用いた DFA のエミュレート

### 3.2 Equivalent character class

Flex では，3.1 節で述べた冗長性のうちの 3 番めのものを用いた圧縮法も利用できる．DFA 中で常に同等に扱われる文字の集合 (equivalent character class; EC) に 0 から始まる番号をつけ，文字コードから ec 番号への対応表 `ec[N]` を用いることで，2 次元配列 `table` の列数を `N` から EC の数にまで減らすことができる．

この方法を Johnson 法と組み合わせることもでき，その場合には図 7 のコード中の `j = *input_ptr++;` を `j = ec[*input_ptr++];` と変更するだけですむ．

さらに flex では，すべての状態ではないが多くの状態で同等に扱われる文字の集合 (meta EC) を用いることもできるようになっている．この場合，ある番号より大きい番号を持つ状態ではすべて meta EC 内の文字を同等に扱うように状態を並べ替え，実行時には状態番号の大きさを判別してから配列 `meta` による変換を適用するかどうか決めるコードを生成する．

## 4. グラフ表現可能なダブル配列

本稿で実験した状態遷移表の新たな表現方法は，青江の方法と似て Johnson 法から `base` 配列を取り除く手法である．青江の方法では，ある状態番号 $i$ について `next[n`$_i$`]=i` となる $n_i$ がただ一つ存在することが必要であったが，状態遷移表が表すグラフ構造ではこれは満たされない．

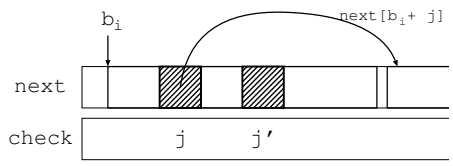


**図 8** 提案手法

我々の方法は，状態番号 $i$ のかわりに `base[i]` の値 $b_i$ を直接 `next` に格納するものである (**図 8**)．青江の方法で用いていた $n_i$ に代えて，この方法では $b_i$ を用いて状態を identify する．入力シンボル番号 $j$ を用いて次の状態へ遷移するには，`check[b`$_i$ `+ j]` をチェックした後に `next[b`$_i$ `+ j]` を読み出して新たな $b_i$ とすればよい．

ここで，Johnson 法や青江の方法では，もとの `table` の複数の行が `next` 中の同じ位置にマップされることがありうることに注意が必要である．我々の手法では `next` 中の行データの開始位置で行を識別する必要があるため，このような位置の重複を許容できない．データを `next` に詰め込む時に，すでに使用した位置を再利用しないように制約して行データのマップ先を決定する必要がある．このため，`next` および `check` 配列のサイズ K は，Johnson 法や青江の方法よりも大きくなる可能性がある．

一方，行データの位置が重複しないことから，`check` に格納する確認用の値として，行の識別子 $b_i$ でなく，オフセット $j$ を用いることが可能になる．状態数に比べてアルファベットのサイズが小さい場合，`check` の要素サイズを小さくできる可能性がある．特に，EC や meta EC を用いた場合には効果があると思われる．

```
b_i = start_state;
do {
  j = *input_ptr++;
  if (next[b_i - 1]) {
    last_pos = input_ptr - 1;
    last_accept = i;
  }
  while (check[b_i + j] != j) b_i = check[b_i - 1];
  i = next[b_i + j];
} while (b_i != JAMSTATE);
```

**図 9** 提案手法を用いた DFA のエミュレート

この手法を用いて DFA をエミュレートするコードは**図 9** のようになる．ここで，Johnson 法における `accept[i]` と `def[i]` の値は，それぞれ `next[b`$_i$ `- 1]` および `check[b`$_i$ `- 1]` に格納して，`accept` 配列と `def` 配列を用いないようにしている．`accept` や `def` の添字に状態番号を用いることはもはやできないので，`accept` や `def` を別に用意するとすれば，これらのサイズも `next` や `check` と同じ大きさになってしまうからである．

Flex では，`-CF` オプションを用いることでこの手

法に似たデータ構造を用いるコードを出力する．ただし，-CF で生成されるコードは 3.1 節で述べた冗長性の 1 番めのものだけを利用して jam 状態への遷移を表から取り除いているが，デフォルト状態を用いた圧縮を行なわない (その分，高速になっている)．また，meta EC を用いることができないという制約がある．

## 5. 評　　価

**表 1** 実験結果

| オプション | サイズ (B) | 実行時間 (ms) | 速度 (MB/s) |
|---|---|---|---|
| -Cnem | 13827 | 868 | 59,6 |
| -Cem | 13911 | 1124 | 46.0 |
| -Cne | 14475 | 892 | 58.0 |
| -Ce | 14820 | 1108 | 46.7 |
| -Cnm | 18097 | 828 | 62.5 |
| -Cm | 18115 | 1076 | 48.1 |
| -C | 23406 | 1004 | 51.6 |
| -Cn | 23527 | 780 | 66.4 |
| -Cfe | 54966 | 776 | 66.7 |
| -CFe | 67597 | 812 | 63.8 |
| -CF | 105476 | 676 | 76.6 |
| -Cf | 178625 | 660 | 78.4 |
| -CFa | 203116 | 676 | 76.6 |
| -Cfa | 348273 | 628 | 82.4 |

本稿で述べた手法を含むさまざまな手法の組合せによって，字句解析器のサイズと実行速度がどのように変化するかを調べるために実験を行なった．実験の環境は Athlon 64 X2 4600+(2.4GHz), RAM 1.0GB, Debian/GNU Linux 4.0, カーネル 2.6.18(amd64), gcc 4.1.1 である．gcc に与えたオプションはすべて-g -O2 である．

Flex 2.5.33 に提案するデータ形式の字句解析器を生成する機能を加え，予約語を含む C の全ての字句要素を解析する字句解析器 (ルール数 101, DFA 状態数 329) を生成して，そのサイズと実行時間を測定した．字句解析器に与えるデータには，ruby-1.8.6 の全ソースコードをプリプロセスし，1 つのファイルに結合し，さらにそれを 10 回繰り返したファイル (大きさ 52MB) を用いた．生成した字句解析器は，全て 8 ビット文字コードを扱うもの (flex に-8 オプションを与えたもの) である．

サイズとして示したのは，生成された字句解析器 (lex.yy.c) をコンパイルしたオブジェクトファイルの text 領域の大きさである．いずれのファイルも，bss 領域は 60 バイトであり，data 領域は 4~ 24 バイトであった．

Flex の圧縮方法を指定するオプションのうち，基本的なデータ構造を指定するのは-C(Johnson 法), -Cn(提案した手法), -Cf (圧縮しない 2 次元配列を用いる方法), -CF(提案手法に似た，full speed table と呼ばれる手法を用いる方法) の 4 種類である．これに，-Ce(EC を用いる), -Cm(meta EC を用いる), -Ca (データを 4 バイト境界に整列する) などの付加的な指定を組み合わせることができる．Flex のマニュアルには時間と空間のトレードオフに関して，以下のような記述がある．

```
slowest & smallest
      -Cem
      -Cm
      -Ce
      -C
      -C{f,F}e
      -C{f,F}
      -C{f,F}a
fastest & largest
```

上記の全ての組合せに，-Cnem, -Cne, -Cnm, -Cn の 4 通りを加えた計 14 通りの組合せについて実験を行なった結果を**表 1** に示す．表の行の順序は，サイズの小さい順である．また，この結果のサイズと速度についてプロットした散布図を**図 10** に示す．散布図で，提案手法に関する結果を表す記号には+を，その他の結果の記号には x を用いている．

## 6. 考　　察

結果のうち，提案手法 (-Cn) を用いた 4 通りは，いずれも Johnson 法を用いた結果とほぼ同等のサイズと，Johnson 法を 3 割程度上回る速度を得ている．また，-Cn と-Cfe の比較では，1/2 未満のメモリサイズでほぼ同等の速度が得られている．

この結果から，Johnson 法を用いるべき理由はほとんど見当たらなくなったと考えられる．

## 7. おわりに

本稿では，DFA の状態遷移表をコンパクトに格納し，優れた実行速度が得られる配列表現について述べた．この手法は木構造に限定されず，また他の圧縮手法と組み合わせる際の制限もない．実験による評価の結果によれば，本手法は同等のメモリサイズの既存手法に比べて常に高い性能が得られており，DFA を用いた字句解析器の速度とメモリサイズのトレードオフを改善するものといえる．

この手法は，ネットワークトラフィックの増大とネッ

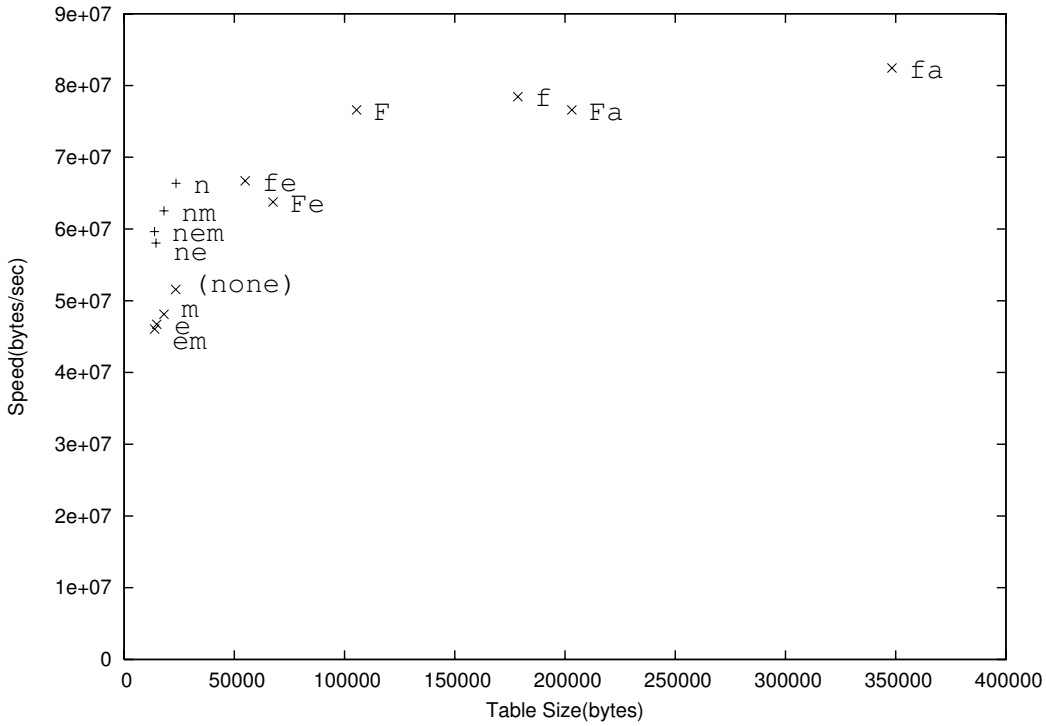


**図 10** 速度とメモリサイズのトレードオフ

トワーク経由の攻撃の増加に対処するために筆者らが提案している，全てのサーバ及びクライアントに配置できる軽量な侵入検知システムに用いる目的で当初の実装を行なったものである[8] が，字句解析器にも応用が可能であると思われたため本稿の実験・評価を行なったものである．

今後は，より広い範囲の文法に対する性能を確認し，Johnson 法にかわる手法として一般的に用いることが可能かを確かめていきたいと考えている．

## 謝　　辞

本研究の一部は，科学研究費補助金特定領域研究「情報爆発 IT 基盤」(領域番号 456) 「通信端点における分散検知モジュールによる侵入防止機構」(課題番号 19024008) をうけて行われた．

## 参 考 文 献

1) Aho, A. V. and Corasick, M. J.: Efficient string matching: an aid to bibliographic search, *Communications of the ACM*, Vol. 18, No. 6, pp. 333–340 (1975).

2) Aoe, J.-I., Morimoto, K. and Sato, T.: An Efficient Implementation of Trie Structures, *Software — Practice and Experience*, Vol. 22, No. 9, pp. 695–721 (1992).

3) Fredkin, E.: Trie Memory, *Communications of the ACM*, Vol. 3, No. 9, pp. 490–499 (1960).

4) Hopcroft, J. E., Ullman, J. D. and Motowani, R.: オートマトン・言語理論・計算論, サイエンス社 (2003).

5) Johnson, W. L., Porter, J. H., Ackley, S. I. and Ross, D. T.: Automatic generation of efficient lexical processors using finite state techniques, *Commun. ACM*, Vol. 11, No. 12, pp. 805–813 (1968).

6) Paxson, V.: *Flex – a fast scanner generator*, Free Software Foundation (1990).

7) Ram, M. S., Sethi, R., Ullman, J. D. and Aho, A. V.: *Compilers: Principles, Techniques and Tools*, Addison-Wesley, second edition (2006).

8) 前田敦司, 渡辺祐介, 西孝王, 山口喜教: 通信端点における軽量侵入検知モジュールの試作, 電子情報通信学会技術研究報告 CPSY, Vol. 106, No. 436, pp. 13–18 (2006).