信州大学工学部

学士論文

逐次的文法圧縮における生成規則カウント法の改善

指導教員　西新 幹彦 准教授

学科　電気電子工学科
学籍番号　07T2059A
氏名　中尾 貴充

2011 年 2 月 25 日

目次

1 序論

1.1 研究の背景と目的

近年, 様々な分野で, 情報を持つデータが溢れかえっている. 従って, 大量のデータを扱うためのハードディスクの容量や, データ転送時のコストを削減するにあたって, あるデータに含まれる情報を保ったまま, データ量を減らした別のデータに変換するというデータ圧縮は必要不可欠である. データ列には, 何度も出現する文字列が多く存在するため, その文字列を別の変数に置換することでデータ量を圧縮することができる. データ圧縮は, 可逆圧縮と不可逆圧縮に大きく二分化することができる. 可逆圧縮は, 圧縮したデータを圧縮する前のデータに戻すことができるが, 不可逆圧縮では戻すことができない. その可逆圧縮のひとつに Yang and Kieffer[1] が提案した文法圧縮がある. 文法圧縮は,5 つのリダクションルールで構成される. このリダクションルールを繰り返し用いることで圧縮が実現される. 文法圧縮に関してはこれまでに, 文法圧縮におけるリダクションルールを削減するアルゴリズム [2], 文法圧縮における標準形を符号化するアルゴリズム [3], 文法圧縮におけるリダクションルールの高速な適用アルゴリズム [4] の研究や, 圧縮率は犠牲になるが使用メモリ量を減らすという省スペースな線形時間文法圧縮アルゴリズムの研究 [5] などが行われている. 文法圧縮アルゴリズムには, 文字列全体を一度に読み込み, 圧縮するという階層的アルゴリズムと, 逐次的に 1 文字ずつ圧縮していくという逐次的アルゴリズムが存在するが, 本研究では, 後者の逐次的アルゴリズムについて考察し, 出現確率を計算するためのカウンタの動作を工夫することにより, 圧縮率を向上させることを提案する.

1.2 本論文の構成

本論文では, 次のような構成をとる. 2 章では文法圧縮とリダクションルールについて説明する. 3 章では提案法についての説明と実験, 結果, 検証について示す. 4 章ではまとめをする.

2 文法圧縮とリダクションルール

2.1 文字と文字列

本研究では,1 バイトのデータのことを終端文字と呼ぶ. 終端文字とは別に変数と呼ばれる文字を用意する. これを論文中では s_i などと表す. 終端文字や変数を合わせて文字と呼ぶ. また, 文字の並びを文字列と呼ぶ.

2.2 生成規則と文法

生成規則 [5] とは, 個々の変数 s_i と文字列 $G(s_i)$ の対応のことである. その関係を

$$s_i \to G(s_i) \quad (i \geq 0)$$

と表す.

この関係を「変数 s_i が文字列 $G(s_i)$ を指す」という. 変数 s_i が指す文字列 $G(s_i)$ に含まれる各変数を再帰的にすべて終端文字列に直したものを「変数 s_i が表す終端文字列」という. そして, 生成規則の集合を文法という.

2.3 文法圧縮

文法圧縮とは, 与えられたデータ列を実質的な性質を保ちつつデータ量を減らした別のデータに変換することである. その過程で生成規則を増やしていき, データ列を小さくするというものである. そのための手順が次に説明するリダクションルールである.

2.4 リダクションルール

文法圧縮は, 変数が指す文字列（生成規則の右辺）内の同じ部分文字列をリダクションルールを用いて生成規則に置き換え圧縮していくものである. このリダクションルールを用いるにあたって 2 つのアルゴリズムがあり,1 つが階層的アルゴリズムで, もう 1 つが逐次的アルゴリズムである. この 2 つのアルゴリズムには共通する手順がある. それは,$G(s_0)$ があらわす終端文字列を変えずにリダクションルールの文法書き換え規則に従って文法の大きさを小さく書き換えるというものである. ここで文法の大きさ [1] とは, 生成規則の右辺にある文字列の長さの総和 $\sum_i |G(s_i)|$ である. Yang and Kieffer[1] は 5 つのリダクションルールを提案した. ルールとその例を以下に示す.

[*ルール 1*] 生成規則の右辺全体の中に一度のみ現れる変数を s_i と置く. 生成規則の右辺に現

れる s_i に対して, 生成規則 $s_i \to G(s_i)$ を適用する. そして,$s_i \to G(s_i)$ を削除する.

$$
\begin{aligned}
&s_0 \to s_1 s_1 \\
&s_1 \to s_2 1 \\
&s_2 \to 010 \\
&\quad \Downarrow \\
&s_0 \to s_1 s_1 \\
&s_1 \to 0101
\end{aligned}
$$

[ルール 2] s の右辺に二度現れる長さ 2 以上の文字列を β と置く. すなわち,$s \to \alpha_1\beta\alpha_2\beta\alpha_3$($\alpha_1$,$\alpha_2$,$\alpha_3$ は長さ 0 以上の文字列) と書ける.s の右辺の β を新しい変数 s' で置き換える. 生成規則 $s' \to \beta$ を追加する.

$$
\begin{aligned}
&s_0 \to s_1 01 s_1 01 \\
&s_1 \to 11 \\
&\quad \Downarrow \\
&s_0 \to s_1 s_2 s_1 s_2 \\
&s_1 \to 11 \\
&s_2 \to 01
\end{aligned}
$$

[ルール 3] s と s' の右辺に現れる長さ 2 以上の文字列を β と置く. すなわち,$s \to \alpha_1\beta\alpha_2$,$s' \to \alpha_3\beta\alpha_4$ と書ける.s,s' の右辺の β を新しい変数 s'' に置き換える. 生成規則 $s'' \to \beta$ を追加する.

$$
\begin{aligned}
&s_0 \to s_1 0 s_2 \\
&s_1 \to 10 \\
&s_2 \to 0 s_1 0 \\
&\quad \Downarrow \\
&s_0 \to s_3 s_2 \\
&s_1 \to 10 \\
&s_2 \to 0 s_3 \\
&s_3 \to s_1 0
\end{aligned}
$$

[ルール 4] s の右辺に現れる長さ 2 以上の文字列 β が s' の右辺と同じであるとする. すなわ

ち,$s \to \alpha_1\beta\alpha_2$,$s' \to \beta$ と書ける.s の右辺の β を s' で置き換える.

$$
\begin{aligned}
&s_0 \to s_2 01 s_1\\
&s_1 \to s_2 0\\
&s_2 \to 11\\
&\quad \Downarrow\\
&s_0 \to s_1 1 s_1\\
&s_1 \to s_2 0\\
&s_2 \to 11
\end{aligned}
$$

[ルール 5] 生成規則において, 右辺が表す終端文字列が同じ生成規則を s,s' とする. 右辺全体の中に現れる s' をすべて s に置き換える. それから,s' を削除する. また, 使用しない生成規則は削除する.

$$
\begin{aligned}
&s_0 \to 1 s_1 s_2 s_4\\
&s_1 \to 00\\
&s_2 \to 1 s_1 0\\
&s_3 \to 10\\
&s_4 \to s_3 s_1\\
&\quad \Downarrow\\
&s_0 \to 1 s_1 s_2 s_2\\
&s_1 \to 00\\
&s_2 \to 1 s_1 0\\
&s_3 \to 10\\
&\quad \Downarrow\\
&s_0 \to 1 s_1 s_2 s_2\\
&s_1 \to 00\\
&s_2 \to 1 s_1 0
\end{aligned}
$$

以上の 5 つのルールを文法に適用して, 文法の大きさを小さくしていく. そして, ルール 1 から 5 のいずれも適用することができない文法にする. このような文法を,irreducible な文法という.irreducible な文法は漸近的に最良な文法であることが [1] によって示されている.

2.5 逐次的アルゴリズム

すでに述べたように, 文法圧縮で用いるアルゴリズムは 2 つある.1 つは階層的アルゴリズムである. これはデータ列全体をメモリ上に展開して圧縮するというものである. もう 1 つは逐

次的アルゴリズムである. これは, データ列から 1 文字読み込んで s_0 が指す文字列の末尾に追加し, その結果として出来た文法に対してルールを適用し,irreducible な文法になったらまた次の 1 文字を読み込んでルールを適用するというアルゴリズムである. このアルゴリズムの利点は,1 文字読み込んではルールを適用するためデータ列全体を展開する必要がなく, より早くルールの適用が出来る. 本研究では, 逐次的アルゴリズムについて考えた.

3 カウンタと符号化

逐次的アルゴリズムを用いてデータ列から 1 文字読み込み,irreducible な文法になるまでルールを適用する. そのルールの適用順は [4] に基づき, ルール $2 \to 3 \to 4 \to 1 \to 5$ とする. 従来法及び提案法の全体の動作の流れについて以下にその手順を示す.

3.1 出現確率

文法圧縮では, 各終端文字と各変数の出てきた回数をカウントするカウンタと呼ばれるものを符号器と復号器に取り入れる. 終端文字（0,1）及び変数（s_i) の出現した個数を読み込んだ文字列全体の文字数で割ることによって確率を算出する. 文字 β の出現確率 $P(\beta)$ の関係式を以下に示す. また,$c(\beta)$ は文字 β のカウンタの値,$\mathcal{A}$ は終端文字全体を表す.

$$P(\beta) = c(\beta) / \sum_{\alpha \in s_i \cup \mathcal{A}} c(\alpha)$$

3.2 従来法

終端文字及び変数を検索し, リダクションルールが適用され, 新たな変数を生成した場合, 新たに生成された変数のみをカウントアップする方法で, 以下にその手順を示す.

1. 終端文字のカウンタをカウントアップする
2. 出現確率を算出する
3. s_0 の末尾に追加する文字を検索し, 見つかった文字を符号化する
4. 上で求めた文字を s_0 の末尾に追加し irreducible な文法にする
5. 文法が圧縮されなかったら, 追加された文字をカウントアップし手順 2 から再開
6. 文法が圧縮されたら, 新たに生成した変数をカウントアップし手順 2 から再開

3.3 提案法

終端文字及び変数を検索し, リダクションルールが適用され, 新たな変数を生成した場合, 文法全体の終端文字及び変数の個数を新たにカウントする方法で, 以下にその手順を示す.

1. 終端文字のカウンタをカウントアップする
2. 出現確率を算出する
3. s_0 の末尾に追加する文字を検索し, 見つかった文字を符号化する
4. 上で求めた文字を s_0 の末尾に追加し irreducible な文法にする
5. 文法が圧縮されなかったら, 追加された文字をカウントアップし手順 2 から再開
6. 文法が圧縮されたら, 文法全体の変数及び終端文字の個数を新たにカウントし直し, 手順 2 から再開

3.4 提案法の狙い

手順 1,2,3,4,5,6 を行うことにより文字の出現する確率を把握することができる. 手順 4 において, 新たに生成された変数が多いと, 手順 6 で文法全体の文字数を新たにカウントし直すために, 符号化確率がより正確なものと考えられる. 符号化確率と情報源の出現確率の誤差が小さいほど圧縮率は良くなるので, より正確な符号化確率を算出するのが提案法の狙いである.

3.5 実験とその結果

従来法を用いた場合と提案法を用いた場合の 2 つのプログラムを用いて圧縮後のファイルサイズを計測し, 各ファイルの圧縮率, 及び, 従来法の圧縮率と提案法の圧縮率の差を算出し, その結果を比較する. また, 本研究で処理の対象となるファイルはカルガリーコーパスファイル [6] である. 実験結果を表 1 に示す.

表 1　実験結果

ファイル名	圧縮率 [%] 従来法	圧縮率 [%] 提案法	改善 [ポイント]
obj1	52.20	52.18	0.02
paper1	38.79	38.52	0.26
paper3	41.14	40.83	0.31
paper4	45.49	45.32	0.17
paper5	47.80	47.51	0.29
paper6	45.50	45.32	0.17
progc	39.80	39.51	0.30
progl	29.25	28.97	0.28
progp	28.75	28.44	0.31

3.6　検証

表 1 から分かるように, 圧縮率にばらつきはあるが, 全てのファイルにおいて, 従来法よりも圧縮されていることが分かる．このことから, ファイルサイズが大きいほうが, リダクションルールの適用回数が多くなり, 出現確率に大きく違いが出てくるからだと考えられる．すなわち, ファイルサイズが大きい（即ち, ルールの適用回数が多い）ファイルには提案法は有効だが, ファイルサイズが小さい（即ち, ルールの適用回数が少ない）ファイルにはあまり有効でないことが予想される．

3.7　もうひとつの従来法

実際に実験は行わなかったが, 上で述べたように [1] で述べられているカウント方法がもう一つ存在するので, その方法をここで紹介する．もう一つのカウント方法は, 基本的には従来法と同じなのだが, 検索する文字で, 明らかにありえない文字が存在する場合, その文字のカウントを初めから 0 にするという方法である．簡単に例を用いて紹介する．

$$s_0 \rightarrow s_1 s_1$$
$$s_1 \rightarrow 0101$$

上の様な生成規則が出来ている場合に $s_1 s_1$ の後に 010 と読み込まれた場合, 次に 1 が読み込まれることはありえない．というのも,1 が読み込まれた場合, それは s_1 と同じであるので, 初めから s_1 が読み込まれる．即ち $s_1 s_1$ の後に 010 と読み込まれた場合,010 の後に読み込まれる文字は 0 か s_1 に限定される．このような状況になった場合に,1 のカウンタの値を 0 にセッ

トしなおし,010 の 1 文字後に読み込まれる文字の出現確率を算出するという方法が, もう一つの従来法である. こちらのカウント方法は従来法よりも圧縮率が良くなることが [1] より分かっており, 提案法はこちらにも適用できるので, 本研究で行った結果よりも圧縮率がさらに良くなることが予想される.

4 まとめ

本研究では, 圧縮率を高めることができないかと考え, カウンタの改良を提案し実験を行った. データ列全体をメモリ上に展開して圧縮する階層的アルゴリズムではなく,1 文字ずつ圧縮する逐次的アルゴリズムを用いた. またリダクションルールの適用順は國安隆治 [4] が提案した方法に従った. 提案法は, 次のような順で文法から 1 文字読み込んでカウントしていく. 初めに 0,1 のカウントを 1 にしておき, 読み込んだ文字を 1 つカウントアップする. その後ルールが適用され, 変数 s_i が作成されたときにもう 1 度文法全体の終端文字及び変数をカウントし直すという方法を用いた. 結果は, 提案法を用いることで圧縮率が良くなった. このことから, 提案法は文法圧縮において有効だったと考えられる. また, ファイルサイズが大きいものにはリダクションルールがより多く適用され, 何度も文法全体の文字数をカウントし直すことにより, 無駄な変数, 置換された文字のカウントを抑えることにより正確な符号化確率を求めることができたと考えられる. 今後の課題として, 今以上の圧縮率の縮小化やより大きなファイルサイズにも適用できるプログラミングの改良が挙げられる.

謝辞

本研究を行うにあたって, 細かく指導してくださった指導教員の西新幹彦准教授に感謝の意を表する.

参考文献

[1] En-hui Yang, John C. Kieffer, “ Efficient Universal Lossless Data Compression Algorithms Based on a Greedy Sequential Grammar Transform -Part One: Without Context Models, ” IEEE Transactions on Information Theory, vol.46, no.3, pp.755-777, 2000.

[2] 矢野裕幸, “ 文法圧縮におけるリダクションルールを削減するアルゴリズム ”, 信州大学工学部学士論文,2008.

[3] 杉山雅紀, “ 文法圧縮における標準形を符号化するアルゴリズム ”, 信州大学工学部学士論文,2009.

[4] 國安隆治, “ 文法圧縮におけるリダクションルールの高速な適用アルゴリズム ”, 信州大学工学部学士論文,2010.
[5] 喜田拓也, 坂本比呂志, 下薗真一, “ 省スペースな線形時間文法圧縮アルゴリズム, ” 電子情報通信学会,pp.1-7,2004.
[6] ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus
[7] 湯淺太一, コンパイラ, 昭晃堂, 2005.
[8] 粂井康孝, 猫でもわかる c 言語プログラミング第 2 版, ソフトバンククリエイティブ株式会社,2008.
[9] B.W. カーハニン,D.M. リッチー, プログラミング言語第 2 版, 共立出版株式会社,2005.
[10] 大川内隆朗, 大原竜男, 簡単 C 言語初版, 株式会社技術評論社,2010

付録 A　ソースコード

A.1　irreducible な文法に圧縮したときに, カウントの無駄を省くプログラム

```
#include <stdio.h>
#include <stdlib.h>
/* #include <string.h> */
#include <time.h>


enum checkType { VALID, PREFIX, VOID };

typedef struct prod_rule {
    int *string;
    int *represent;
    int counter;
    enum checkType check;
    int freq;
    int cum_freq;
} prod_rule;

prod_rule *grammar;
int max_rules;

int input_length;

int reduce_grammar_call;
int reduce_grammar_noapply;

/*****************************************************************************/

int
stringcmp(int *str1, int *str2)
{
    int i;

    for (i = 0; str1[i] == str2[i] && str1[i] != 256 && str2[i] != 256; i++){
        ;
    }
    return(str1[i] - str2[i]);
}

int
stringlen(int *string)
{
    int num;

    if (string == NULL){
        printf("%d\n", __LINE__);
        exit(1);
    }
    for (num = 0; string[num] != 256; num++){  /* rule の長さのカウント */
        ;
    }
    return(num);
}

/* 生成規則の右辺の長さを求める（末尾の 256 は数えない） */
int
rulelen(prod_rule *p_rule)
{
    int num;

    if (p_rule == NULL){
        printf("%d\n", __LINE__);
```

```
        exit(1);
    }
    num = stringlen(p_rule->string);
    return(num);
}

/* 生成規則を表示する */
void
printrule(prod_rule *p_rule)
{
    int *s = p_rule->string;

    do {
        if (*s < 256){
            if (' ' <= *s && *s <= '~'){
                printf("  %c", (char)*s);
            } else {
                printf(" %02x", (char)*s);
            }
        } else {
            printf(" %d%c", *s - 256, '*');
        }
    } while (*s++ != 256);
    if (p_rule->represent == NULL){
        printf(" NULL");
    } else {
        printf(" \"");
        for (s = p_rule->represent; *s < 256; s++){
            if (' ' <= *s && *s <= '~'){
                printf("%c", (char)*s);
            } else {
                printf("0x%02x ", (char)*s);
            }
        }
        printf("\"");
    }
    printf("\n");
    return;
}

/* 文法全体を表示する */
void
printgrammar()
{
    int i;

    for (i = 0; i < max_rules; i++){
        if (grammar[i].string != NULL){
            printf("%d%c -> ", i, '*');
            printrule(grammar + i);
        }
    }
    return;
}

/*************************************************************************/


/* 1 度しか使われていない生成規則の数を調べる */
int
reduct_rule_1_check()
{
    int i, j, single;

    for (i = 0; i < max_rules; i++){
        grammar[i].counter = 0;
    }

    for (i = 0; i < max_rules; i++){
```

```
        if (grammar[i].string != NULL){
            for (j = 0; grammar[i].string[j] != 256 ; j++){
                if (grammar[i].string[j] >= 257){
                    grammar[grammar[i].string[j] - 256].counter++;
                }
            }
        }
    }
    single = 0;
    for (i = 0; i < max_rules; i++){
        if (grammar[i].counter == 1){
            single++;
        }
    }
    return(single); /* 1 度しか使われていない生成規則の数 */
}

/* 1 度しか使われていない生成規則を削除する */
int
reduct_rule_1()
{
    int i, j;
    int rep = 0;

    for (i = 0; i < max_rules; i++){
        if (grammar[i].string != NULL){
            for (j = 0; grammar[i].string[j] != 256; j++){
                if (grammar[i].string[j] >= 257 && grammar[grammar[i].string[j] - 256].counter == 1){
                    int rule_from = grammar[i].string[j] - 256;
                    int newstrlen = rulelen(grammar + i) + rulelen(grammar + rule_from) - 1;
                    int *newstr = (int *)calloc(newstrlen + 1, sizeof(int));
                    int k_new, k_old;

                    for (k_new = 0, k_old = 0; k_new < newstrlen; k_old++){
                        if (grammar[i].string[k_old] == rule_from + 256){
                            int k_from;
                            for (k_from = 0; grammar[rule_from].string[k_from] != 256; k_from++){
                                newstr[k_new++] = grammar[rule_from].string[k_from];
                            }
                        } else {
                            newstr[k_new++] = grammar[i].string[k_old];
                        }
                    }
                    newstr[newstrlen] = 256;
                    free(grammar[i].string);
                    grammar[i].string = newstr;
                    free(grammar[rule_from].string);
                    grammar[rule_from].string = NULL;
                    j--;
                    rep++;
                }
            }
        }
    }
    return(rep); /* 削除した数 */
}

/* 重複パターンを検索するプログラム */
int
reduct_rule_2_check(prod_rule *p_rule, int **x2, int **y2)
{
    int len1 = 0, len2 = 0;
    int i, i_mem, i_len;
    int dist, dist_mem;
    int *line = p_rule->string;
    int match = 1;

    i_len = rulelen(p_rule);
```

```
    for(dist = i_len / 2; dist > 1; dist--){
            for(i = 0; i < i_len - dist; i++){
                if(line[i] == line[dist + i]){
                    len1++;
                    while (len1 > match){
                        match = len1;          /* 重複部分の最大の長さ */
                        i_mem = i;            /* 重複文字の最後の場所 */
                        dist_mem = dist;    /* 2 箇所の重複部分の距離 */
                    }
                    if(len1 > dist)
                        break;
                } else {
                    if(len1 < dist && len1 > 1){
                        /*print_len(&line[i - len1], len1);*/
                    }
                    len1 = 0;
                }
                if(len1 == dist){
                    break;
                } else if(i == i_len - dist - 1 && len1 > 1){
                    /*print_len(&line[i + 1 - len1], len1);*/
                }
            }

            if(i_len %2 == 0 && dist == i_len / 2){    /* 文字数が偶数の時、初期値 dist(最大値) のみ除く */
                ;
            } else {
                for(i = 0; i < dist; i++){
                    if(line[i] == line[i_len - dist + i]){
                        len2++;
                        while (len2 > match){
                            match = len2;
                            i_mem = i;
                            dist_mem = i_len - dist;
                        }
                        if(len2 > dist)
                            len2 = dist;
                    } else {
                        if(len2 < dist && len2 > 1){
                        /*    print_len(&line[i - len2], len2);*/
                        }
                        len2 = 0;
                    }
                    if(len2 == dist){
                        break;
                    } else if(i == dist - 1 && len2 > 1){
                        /*print_len(&line[i + 1 - len2], len2);*/
                    }
                }
            }
            if(match == dist){
                break;
            }
            len1 = len2 = 0;
    }
    if(match < 2){
        return (-1);
    }

/*    print_len(&line[i_mem + 1 - maxlen], maxlen);    重複の最大文字の部分
    printf("が最長重複文字である。\n");
*/
    *x2 = &line[i_mem + 1 - match];
    *y2 = &line[i_mem + 1 - match + dist_mem];

    return (match);
}

/* リダクションルール 2 の適用 */
```

```
void
reduct_rule_2(prod_rule *p_rule, prod_rule *new_rule, int *x2, int *y2, int n)
{
    int k, l;
    int *s0, *s1;
    int i_len;

    i_len = rulelen(p_rule);
    s0 = (int *) calloc(i_len + 3 - 2 * n, sizeof(int));
    s1 = (int *) calloc(n + 1, sizeof(int));
    if(s0 == NULL || s1 == NULL){
        printf("メモリが確保できません。\n");
        exit(-1);
    }

    for(l = k = 0; l < i_len + 1; l++, k++){
        if (p_rule->string + l == x2 || p_rule->string + l == y2){
            s0[k] = (new_rule - grammar) + 256;
            l += n - 1;
        } else {
            s0[k] = *(p_rule->string + l);
        }
    }

    for(l = 0; l < n; l++){
        s1[l] = x2[l];
    }
    s1[n] = 256;

    free(p_rule->string);

    p_rule->string = s0;
    new_rule->string = s1;
    return;
}

/* リダクションルール 3 が適用できるか検索（力まかせ法）*/
int
reduct_rule_3_check(prod_rule *p_rule1, prod_rule *p_rule2, int **x3, int **y3)
{
    int pt, pp, pl;
    int i_len, j_len, samelen = 0;
    int max_pt, max_pp;
    int maxl = 1;

    i_len = rulelen(p_rule1);
    j_len = rulelen(p_rule2);


    if(i_len >= j_len){        /* array[i] の方が長い、若しくは同じ場合 */
        if(j_len > 2){
            for(pl = 2; pl < j_len; pl++){
                for(pt = 0, pp = j_len - pl; pp < j_len; pt++, pp++){
                    if(*(p_rule1->string + pt) == *(p_rule2->string + pp)){
                        samelen++;
                        if(maxl < samelen){
                            maxl = samelen;
                            max_pt = pt;
                            max_pp = pp;
                        }
                        if(pp == j_len - 1){
                            samelen = 0;
                        }
                    } else {
                        samelen = 0;
                    }
                }
            }
        }
```

```
    for(pl = 0; pl < i_len - j_len + 1; pl++){
        for(pt = pl, pp = 0; pt < i_len && pp < j_len; pt++, pp++){
            if(*(p_rule1->string + pt) == *(p_rule2->string + pp)){
                samelen++;
                if(maxl < samelen){
                    maxl = samelen;
                    max_pt = pt;
                    max_pp = pp;
                }
                if(pp == j_len - 1){
                    samelen = 0;
                }
            } else {
                samelen = 0;
            }
        }
    }
    if(j_len > 2){
        for(pl = 0; pl < j_len - 2; pl++){
            for(pp = 0, pt = i_len - j_len + 1 + pl; pt < i_len && pp < j_len - 1 - pl; pt++, pp++){
                if(*(p_rule1->string + pt) == *(p_rule2->string + pp)){
                    samelen++;
                    if(maxl < samelen){
                        maxl = samelen;
                        max_pt = pt;
                        max_pp = pp;
                    }
                    if(pp == j_len - 2 - pl){
                        samelen = 0;
                    }
                } else {
                    samelen = 0;
                }
            }
        }
    }
} else {        /* array[j] の方が長い場合 */
    if(i_len > 2){
        for(pl = 2; pl < i_len; pl++){
            for(pp = 0, pt = i_len - pl; pt < i_len; pt++, pp++){
                if(*(p_rule1->string + pt) == *(p_rule2->string + pp)){
                    samelen++;
                    if(maxl < samelen){
                        maxl = samelen;
                        max_pt = pt;
                        max_pp = pp;
                    }
                    if(pt == i_len - 1){
                        samelen = 0;
                    }
                } else {
                    samelen = 0;
                }
            }
        }
    }

    for(pl = 0; pl < j_len - i_len + 1; pl++){
        for(pp = pl, pt = 0; pt < i_len && pp < j_len; pt++, pp++){
            if(*(p_rule1->string + pt) == *(p_rule2->string + pp)){
                samelen++;
                if(maxl < samelen){
                    maxl = samelen;
                    max_pt = pt;
                    max_pp = pp;
                }
                if(pt == i_len - 1){
                    samelen = 0;
```

```
                    }
                } else {
                    samelen = 0;
                }
            }
        }
        if(i_len > 2){
            for(pl = 0; pl < i_len - 2; pl++){
                for(pt = 0, pp = j_len - i_len + 1 + pl; pt < i_len - 1 - pl; pt++, pp++){
                    if(*(p_rule1->string + pt) == *(p_rule2->string + pp)){
                        samelen++;
                        if(maxl < samelen){
                            maxl = samelen;
                            max_pt = pt;
                            max_pp = pp;
                        }
                        if(pt == i_len - 2 - pl){
                            samelen = 0;
                        }
                    } else {
                        samelen = 0;
                    }
                }
            }
        }
    }

    if(maxl > 1){
        *x3 = p_rule1->string + max_pt - maxl + 1;
        *y3 = p_rule2->string + max_pp - maxl + 1;
        return(maxl);
    } else {
        return(-1);
    }
}

/* リダクションルール 3 の適用 */
void
reduct_rule_3(prod_rule *p_rule1, prod_rule *p_rule2, prod_rule *new_rule, int *x3, int *y3, int n)
{
    int i_len, j_len;
    int *u0, *u1, *u2;
    int rl, rk;

    i_len = rulelen(p_rule1);
    j_len = rulelen(p_rule2);

    u0 = (int *) calloc(i_len - n + 2, sizeof(int));
    u1 = (int *) calloc(j_len - n + 2, sizeof(int));
    u2 = (int *) calloc(n + 1, sizeof(int));
    if(u0 == NULL || u1 == NULL || u2 == NULL){
        printf("メモリが確保できません\n");
        exit(-1);
    }

    if(i_len >= j_len){
        /* array[i] に関する部分 */
        if(j_len >= n){
            for(rl = rk = 0; rl < i_len + 1; rl++, rk++){
                if(p_rule1->string + rl == x3){
                    if(j_len == n){
                        u0[rk] = (p_rule2 - grammar) + 256;
                    } else {
                        u0[rk] = (new_rule - grammar) + 256;
                    }
                    rl += n - 1;
                } else {
                    u0[rk] = *(p_rule1->string + rl);
                }
```

```
            }
        }
        /* array[j], array[k] に関する部分 */
        if(j_len > n){
            for(rl = rk = 0; rl < j_len + 1; rl++, rk++){
                if(p_rule2->string + rl == y3){
                    u1[rk] = (new_rule - grammar) + 256;
                    rl += n - 1;
                } else {
                    u1[rk] = *(p_rule2->string + rl);
                }
            }
            for(rl = 0; rl < n; rl++){
                u2[rl] = y3[rl];
            }
            u2[n] = 256;

            free(p_rule2->string);
            p_rule2->string = u1;
            new_rule->string = u2;
        } else if(j_len == n){
            free(u2);
        }
        free(p_rule1->string);
        p_rule1->string = u0;
    } else {
        /* array[j] に関する部分 */
        if(i_len >= n){
            for(rl = rk = 0; rl < j_len + 1; rl++, rk++){
                if(p_rule2->string + rl == y3){
                    if(i_len == n){
                        u1[rk] = (p_rule1 - grammar) + 256;
                    } else {
                        u1[rk] = (new_rule - grammar) + 256;
                    }
                    rl += n - 1;
                } else {
                    u1[rk] = *(p_rule2->string + rl);
                }
            }
        }
        /* array[i], array[k] に関する部分 */
        if(i_len > n){
            for(rl = rk = 0; rl < i_len + 1; rl++, rk++){
                if(p_rule1->string + rl == x3){
                    u0[rk] = (new_rule - grammar) + 256;
                    rl += n - 1;
                } else {
                    u0[rk] = *(p_rule1->string + rl);
                }
            }
            for(rl = 0; rl < n; rl++){
                u2[rl] = x3[rl];
            }
            u2[n] = 256;

            free(p_rule1->string);
            p_rule1->string = u0;
            new_rule->string = u2;
        } else if(i_len == n){
            free(u2);
        }
        free(p_rule2->string);
        p_rule2->string = u1;
    }

    return;
}
```

```
/* KMP 法による文字列検索 */
int
reduct_rule_4_check(prod_rule *p_rule1, prod_rule *p_rule2)
{
    int pt = 1, pp = 0;
    int i_len, j_len;
    int *skip;

    i_len = rulelen(p_rule1);
    j_len = rulelen(p_rule2);

    if(j_len == 0){
        printf("強制終了\n");
        exit(-1);
    }

    skip = (int *) calloc(j_len + 1, sizeof(int));
    if(skip == NULL){
        printf("メモリが確保できません\n");
        exit(-1);
    }

    /* 表の作成 */
    skip[0] = 0;
    skip[pt] = 0;
    while (*(p_rule2->string + pt) != 256){
        if(*(p_rule2->string + pt) == *(p_rule2->string + pp)){
            skip[++pt] = ++pp;
        } else if (pp == 0){
            skip[++pt] = 0;
        } else {
            pp = skip[pp];
        }
    }
/*    printf("表：");
    print_len(skip, j_len + 1);
*/
    for(pt = pp = 0; pt < i_len && pp < j_len; ){
        if(*(p_rule1->string + pt) == *(p_rule2->string + pp)){
            pt++; pp++;
        } else if(pp == 0){
            pt++;
        } else {
            pp = skip[pp];          /* ここがポイント */
        }
    }
    free(skip);
    if(pp == j_len){
        return (pt - pp);
    }
    return (-1);
}



/* リダクションルール 4 の適用 */
void
reduct_rule_4(prod_rule *p_rule1, prod_rule *p_rule2, int m)
{
    int k, l, i_len, j_len;
    int *t0;

    i_len = rulelen(p_rule1);
    j_len = rulelen(p_rule2);
    t0 = (int *) calloc(i_len - j_len + 2, sizeof(int));
    if(t0 == NULL){
        printf("メモリが確保できません\n");
        exit(-1);
    }
```

```
    for(l = k = 0; l < i_len + 1; l++, k++){
        if (l == m){
            t0[k] = (p_rule2 - grammar) + 256;
            l += j_len - 1;
        } else {
            t0[k] = p_rule1->string[l];
        }
    }
    free(p_rule1->string);
    p_rule1->string = t0;

    return;
}

int
rulelen_rec(prod_rule *p_rule)
{
    int i;
    int len = 0;

    for (i = 0; p_rule->string[i] != 256; i++){
        if (p_rule->string[i] < 256){
            len++;
        } else {
            len += rulelen_rec(grammar + (p_rule->string[i] - 256));
        }
    }
    return(len);
}

int *
set_represent(prod_rule *p_rule, int *dest)
{
    int i;

    for (i = 0; p_rule->string[i] != 256; i++){
        if (p_rule->string[i] < 256){
            *dest = p_rule->string[i];
            dest++;
        } else {
            dest = set_represent(grammar + (p_rule->string[i] - 256), dest);
        }
    }
    return(dest);
}

int
make_represent(prod_rule *p_rule)
{
    int len;
    int *term;

    len = rulelen_rec(p_rule);
    p_rule->represent = (int *)calloc(len + 1, sizeof(int));
    if (p_rule->represent == NULL){
        printf("error on %d\n", __LINE__);
        exit(1);
    }
    term = set_represent(p_rule, p_rule->represent);
    *term = 256;

    return(len);
}

/*リダクションルール 5*/
void
reduct_rule_5(prod_rule *replace, prod_rule *with)
{
```

```
    int i, j;
    int n_rep = replace - grammar + 256;
    int n_with = with - grammar + 256;

    for (i = 0; i < max_rules; i++){
        if (grammar[i].string != NULL){
            for (j = 0; grammar[i].string[j] != 256 ; j++){
                if (grammar[i].string[j] == n_rep){
                    grammar[i].string[j] = n_with;
                }
            }
        }
    }
    free(replace->string);
    replace->string = NULL;
    free(replace->represent);
    replace->represent = NULL;
    replace->counter = 0;
    return;
}


/************************************************************************/
#define STACK_MAX 100000
int stack_idx;
int stack[STACK_MAX];
FILE *input_fp;

int
getsymbol()
{
    int symbol;

    if (stack_idx > 0){
        stack_idx--;
        symbol = stack[stack_idx];
    } else {
        symbol = fgetc(input_fp);
        input_length++;
    }
    return(symbol);
}

#if 0

int
getsymbol_()
{
    static char input[] = "0100010101001001011001010010";
    static int i = 0;
    int symbol;

    if (stack_idx > 0){
        stack_idx--;
        symbol = stack[stack_idx];
    } else {
        if (i >= (int)strlen(input)){
            symbol = EOF;
        } else {
            symbol = input[i++];
        }
    }
printf("getsymbol(): %d\n", symbol);
    return(symbol);
}

#endif

void
```

```
ungetsymbol(int symbol)
{
    if (stack_idx >= STACK_MAX){
        printf("overflow at %d\n", __LINE__);
        exit(1);
    }
    stack[stack_idx] = symbol;
    stack_idx++;
    return;
}

void encode_symbol(int);
void calccumfreq(void);
void incfreq(int);

#define BUFFER_MAX 100000
int
extend_grammar()
{
    int buffer[BUFFER_MAX];
    int count_valid;
    int last_prefix = -1;
    int last_valid = -1;
    int append;
    int n;
    int i;
    int len_s0;

    for (i = 1; i < max_rules; i++){
        if (grammar[i].string != NULL){
            grammar[i].check = VALID;
            /* rules[i].represent はセットされているはず */
            if (grammar[i].represent == NULL){
                printf("programm error on %d\n", __LINE__);
                exit(1);
            }
        }
    }

    n = 0;
    do {
        if (n >= BUFFER_MAX){
            printf("overflow at %d (n=%d)\n", __LINE__, n);
            exit(1);
        }
        buffer[n] = getsymbol();
        count_valid = 0;
        for (i = 1; i < max_rules; i++){
            if (grammar[i].string != NULL && grammar[i].check == VALID){
                if (grammar[i].represent[n] == 256){
                    grammar[i].check = PREFIX;
                    last_prefix = i;
                } else if (grammar[i].represent[n] == buffer[n]){
                    count_valid++;
                    last_valid = i;
                } else {
                    grammar[i].check = VOID;
                }
            }
        }
        n++;
    } while (count_valid > 1 || (count_valid == 1 && stringlen(grammar[last_valid].represent) > n));

    if (count_valid == 1){
        append = last_valid + 256;
    } else {
        if (last_prefix < 0){
            append = buffer[0];
            while (--n > 0){
```

```
                ungetsymbol(buffer[n]);
            }
        } else {
            int len;
            append = last_prefix + 256;
            len = stringlen(grammar[last_prefix].represent);
            while (--n >= len){
                ungetsymbol(buffer[n]);
            }
        }
    }

    if (append != EOF){
        /* append を S0 の末尾に追加 */
        len_s0 = rulelen(grammar) + 2;
        grammar->string = (int *)realloc(grammar->string, len_s0 * sizeof(int));
        if (grammar->string == NULL){
            printf("out of memory on %d\n", __LINE__);
            exit(1);
        }
        grammar->string[len_s0 - 2] = append;
        grammar->string[len_s0 - 1] = 256;

        //printf("(");
        //for (i = 1; i < max_rules; i++){
        //    if (grammar[i].string != NULL){
        //        printf(" %d", i + 256);
        //    }
        //}
        //printf(")\n%d\n", append);

        calccumfreq();
/*        {
            int i;
            for (i = 0; i < max_rules + 257; i++){
                printf("cumfreq[%03d] = %d\n", i, cumfreqval(i));
            }
        } */
        encode_symbol(append);
        /* incfreq(append); */
    }
    return(append);
}


prod_rule *
blank_rule()
{
    int rule_idx;

    for (rule_idx = 1; rule_idx < max_rules; rule_idx++){
        if (grammar[rule_idx].string == NULL){
            grammar[rule_idx].string = NULL;
            grammar[rule_idx].represent = NULL;
            grammar[rule_idx].counter = 0;
            grammar[rule_idx].freq = 1; /* 頻度の初期化 */
            return(grammar + rule_idx);
        }
    }
    max_rules++;
    grammar = (prod_rule *)realloc(grammar, sizeof(prod_rule) * max_rules);
    if (grammar == NULL){
        printf("out of memory on %d\n", __LINE__);
        exit(1);
    }
    grammar[max_rules - 1].string = NULL;
    grammar[max_rules - 1].represent = NULL;
    grammar[max_rules - 1].counter = 0;
    grammar[max_rules - 1].freq = 1; /* 頻度の初期化 */
```

```
    return(grammar + (max_rules - 1));
}

int
reduce_grammar()
{
    int len, num;
    int *pat1;
    int *pat2;
    int rule_idx, rule_idx2;
    prod_rule *new_rule;
    int apply;

    reduce_grammar_call++;
    apply = 0;

    len = reduct_rule_2_check(grammar + 0, &pat1, &pat2);
    if (len >= 2){
        new_rule = blank_rule();
        reduct_rule_2(grammar + 0, new_rule, pat1, pat2, len);
        make_represent(new_rule);
        apply = 1;
    }

    rule_idx = 1;
    while (rule_idx < max_rules){
        if (grammar[rule_idx].string == NULL){
            rule_idx++;
            continue;
        }
        len = reduct_rule_3_check(grammar + 0, grammar + rule_idx, &pat1, &pat2);
        if (len >= 2){
            if (rulelen(grammar + rule_idx) == len){
                reduct_rule_4(grammar + 0, grammar + rule_idx, pat1 - grammar[0].string);
            } else {
                new_rule = blank_rule();
                reduct_rule_3(grammar + 0, grammar + rule_idx, new_rule, pat1, pat2, len);
                make_represent(new_rule);
            }
            rule_idx = 1;
            apply = 1;
        } else {
            rule_idx++;
        }
    }

/*
    reduct_rule_1_check();
    reduct_rule_1();
*/
    num = reduct_rule_1_check();
    if (num > 0){
        reduct_rule_1();
        apply = 1;
    }

    for (rule_idx = 1; rule_idx < max_rules - 1; rule_idx++){
        if (grammar[rule_idx].string == NULL){
            continue;
        }
        for (rule_idx2 = rule_idx + 1; rule_idx2 < max_rules; rule_idx2++){
            if (grammar[rule_idx2].string == NULL){
                continue;
            }
            if (stringcmp(grammar[rule_idx].represent, grammar[rule_idx2].represent) == 0){
                reduct_rule_5(grammar + rule_idx2, grammar + rule_idx);
                apply = 1;
            }
```

```
    }
}

if (apply == 0){
    reduce_grammar_noapply++;
}

while(apply == 1){
    apply = 0;
    rule_idx = 0;
    while (rule_idx < max_rules){
        if (grammar[rule_idx].string == NULL){
            rule_idx++;
            continue;
        }
        len = reduct_rule_2_check(grammar + rule_idx, &pat1, &pat2);
        if (len >= 2){
            new_rule = blank_rule();
            reduct_rule_2(grammar + rule_idx, new_rule, pat1, pat2, len);
            make_represent(new_rule);
            rule_idx = 0;
            apply = 1;
        } else {
            rule_idx++;
        }
    }

    rule_idx = 0;
    while (rule_idx < max_rules - 1){
        if (grammar[rule_idx].string == NULL){
            rule_idx++;
            continue;
        }
        rule_idx2 = rule_idx + 1;
        while (rule_idx2 < max_rules){
            if (grammar[rule_idx2].string == NULL){
                rule_idx2++;
                continue;
            }
            len = reduct_rule_3_check(grammar + rule_idx, grammar + rule_idx2, &pat1, &pat2);

            if (len >= 2){
                if (rulelen(grammar + rule_idx) == len){
                    reduct_rule_4(grammar + rule_idx2, grammar + rule_idx, pat2 - grammar[rule_idx2].string);
                } else if (rulelen(grammar + rule_idx2) == len) {
                    reduct_rule_4(grammar + rule_idx, grammar + rule_idx2, pat1 - grammar[rule_idx].string);
                } else {
                    new_rule = blank_rule();
                    reduct_rule_3(grammar + rule_idx, grammar + rule_idx2, new_rule, pat1, pat2, len);
                    make_represent(new_rule);
                }
                rule_idx = 0;
                rule_idx2 = 1;
                apply = 1;
            } else {
                rule_idx2++;
            }
        }
        rule_idx++;
    }

    num = reduct_rule_1_check();
    if (num > 0){
        reduct_rule_1();
        apply = 1;
    }

    for (rule_idx = 1; rule_idx < max_rules - 1; rule_idx++){
        if (grammar[rule_idx].string == NULL){
```

```
                continue;
            }
            for (rule_idx2 = rule_idx + 1; rule_idx2 < max_rules; rule_idx2++){
                if (grammar[rule_idx2].string == NULL){
                    continue;
                }
                if (stringcmp(grammar[rule_idx].represent, grammar[rule_idx2].represent) == 0){
                    reduct_rule_5(grammar + rule_idx2, grammar + rule_idx);
                    apply = 1;
                }
            }
        }
    }

    return(0);
}

/* 全ての生成規則の右辺の文字総数 */
int
rule_number()
{
    int sum = 0;
    int rule_idx;

    for (rule_idx = 0; rule_idx < max_rules; rule_idx++){
        if (grammar[rule_idx].string != NULL){
            sum += rulelen(grammar + rule_idx);
        }
    }

    return(sum);
}


/* 生成規則の総数 */
int
grammar_number()
{
    int num = 0;
    int rule_idx;

    for (rule_idx = 0; rule_idx < max_rules; rule_idx++){
        if (grammar[rule_idx].string != NULL){
            num += 1;
        }
    }

    return(num);
}


/*****************************************************************************/

int *
dupstring(int *string)
{
    int *dup;
    int i;

    dup = (int *)calloc(stringlen(string) + 1, sizeof(int));
    for (i = 0; string[i] != 256; i++){
        dup[i] = string[i];
    }
    dup[i] = 256;
    return(dup);
}

/*****************************************************************************/
#define CODE_VALUE_BITS 16
```

```
#define TOP_VALUE (((long)1 << CODE_VALUE_BITS) -1)
#define FIRST_QTR (TOP_VALUE / 4+1)
#define HALF (2 * FIRST_QTR)
#define THIRD_QTR (3 * FIRST_QTR)

int low = 0;
int high = TOP_VALUE;
int cum_freq[258];
int freq[257];

void
calcfreq()
{
    int i, j;

    for (i = 0; i < 258; i++){
        freq[i] = 1;
    }
    for (i = 0; i < max_rules; i++){
        grammar[i].freq = 0;
    }

    for (i = 0; i < max_rules; i++){
        if (grammar[i].string != NULL){
            for (j = 0; grammar[i].string[j] != 256 ; j++){
                if (grammar[i].string[j] < 257){
                    freq[grammar[i].string[j]]++;
                } else {
                    grammar[grammar[i].string[j] - 256].freq++;
                }
            }
        }
    }
    return;
}

void
calccumfreq()
{
    int i;
    int cum = 0;

    cum_freq[0] = 0;
    for (i = 1; i < 258; i++){
        cum += freq[i - 1];
        cum_freq[i] = cum;
    }
    for (i = 1; i < max_rules; i++){
        if (grammar[i].string != NULL){ /* i 番目の生成規則が使われている */
            cum += grammar[i].freq;
        }
        grammar[i].cum_freq = cum;
    }
    return;
}

int
cumfreqval(int index)
{
    if (index > max_rules + 256){
        printf("%d\n", __LINE__);
        exit(1);
    }
    if (index < 258){
        return(cum_freq[index]);
    }
    return(grammar[index - 257].cum_freq);
}
```

```
void
incfreq(int index)
{
    if (index < 257){
        freq[index]++;
    }
    else {
        grammar[index-256].freq++;
    }
    return;
}

/************************************************************************/
FILE *codewordfile3;
int bit_buffer;
int bits_to_go = 8;

void
output_bit(int bit)
{
       bit_buffer >>= 1;
       if (bit){
               bit_buffer |= 0x80;
       }
       bits_to_go --;
       if(bits_to_go == 0){
           putc(bit_buffer, codewordfile3);
           bits_to_go = 8;
       }
       return;
}

void
done_outputing_bits()
{
    putc(bit_buffer >> bits_to_go, codewordfile3);
    return;
}

/************************************************************************/
long bits_to_follow;

void bit_plus_follow(int bit)
{
    output_bit(bit);
    while(bits_to_follow > 0){
            output_bit(1 -bit);
            bits_to_follow--;
    }
    return;
}

void
encode_symbol(int symbol)
{
    long range;
    int num;
    num = grammar_number();

    range = (long)(high - low) + 1;
    high = low + (range * cumfreqval(symbol + 1)) / cumfreqval(max_rules + 256) - 1;
    low = low + (range * cumfreqval(symbol)) / cumfreqval(max_rules + 256);
    if (high < low){
        printf( "生成規則数:%d\n",num );
        printf("%d, %d\n", __LINE__, max_rules);
        exit(1);
    }
    for (;;){
            if (high <HALF) {
```

```
            bit_plus_follow(0);
        } else if (low >= HALF){
            bit_plus_follow(1);
            low -= HALF;
            high -= HALF;
        }else if (low >= FIRST_QTR && high < THIRD_QTR){
            bits_to_follow++;
            low -= FIRST_QTR;
            high -= FIRST_QTR;
        } else {
            break;
        }
        low = 2 * low;
        high = 2 *high +1;
    }
    return;
}

void
done_encoding()
{
    bits_to_follow++;
    if (low < FIRST_QTR){
        bit_plus_follow(0);
    }else{
        bit_plus_follow(1);
    }
    return;
}

/*****************************************************************************/

int
main(int argc,char **argv)
{
    int i;
    clock_t end;
    int sum, num;

    input_fp = fopen("paper5", "rb");
    if (input_fp == NULL){
        printf("open error on %d\n", __LINE__);
        exit(1);
    }
    codewordfile3 = fopen("codeword3.txt", "wb");
    if (codewordfile3 == NULL){
        printf("open error on %d\n", __LINE__);
        exit(1);
    }

    /* 終端文字の頻度の初期化 */
    for (i = 0; i < 258; i++){
        freq[i] = 1;
    }

    input_length = 0;

    reduce_grammar_call = 0;
    reduce_grammar_noapply = 0;

    grammar = (prod_rule *)malloc(sizeof(prod_rule));
    if (grammar == NULL){
        printf("%d\n", __LINE__);
        exit(1);
    }
    grammar[0].string = (int *)malloc(sizeof(int));
    if (grammar[0].string == NULL){
        printf("%d\n", __LINE__);
        exit(1);
```

```
    }
    grammar[0].string[0] = 256;
    max_rules = 1;
    make_represent(grammar + 0);

    while (extend_grammar() != EOF){
        reduce_grammar();
        calcfreq();
    }
    done_outputing_bits();

    end = clock();
    printf( "実行時間:%f[s]\n", (double)end / CLOCKS_PER_SEC);

    sum = rule_number();
    printf( "右辺の文字総数:%d\n", sum);
    num = grammar_number();
    printf( "生成規則数:%d\n", num);
    printf( "適用されない回数:%d\n", reduce_grammar_noapply);
    printf( "全体の回数:%d\n", reduce_grammar_call);
    printf( "長さ:%d\n", input_length);

    return(0);

}



#if 0

int
main()
{
    int s0[] = {'1', 256+1, 256+2, 256+4, 256};
    int s1[] = {'0', '0', 256};
    int s2[] = {'1', 256+1, '0', 256};
    int s3[] = {'1', '0', 256};
    int s4[] = {256+3, 256+1, 256};
    prod_rule rules[5];
    int num;
    int i;
    int j;
    int *x3;
    int *y3;

    rules[0].string = dupstring(s0);
    rules[1].string = dupstring(s1);
    rules[2].string = dupstring(s2);
    rules[3].string = dupstring(s3);
    rules[4].string = dupstring(s4);
    grammar = rules;
    max_rules = 5;

    printgrammar();

    for (i = 1; i < max_rules ; i++){
        make_represent(rules + i);
    }

    for (j = 1; j < max_rules -1; j++){
        for (i = j + 1; i < max_rules; i++){
            if (strcmp(rules[j].represent, rules[i].represent) == 0)
                printf("%d, %d\n", j, i);
        }
    }

    printf("[%d]\n", __LINE__);
```

```
    reduct_rule_5(rules + 2, rules + 4);
    max_rules = 5;

    printf("[%d]\n", __LINE__);
    printgrammar();

    return(0);
}



int
main()
{
    int str[] = {'1', '0','1','1','1', 256+1,'1','0','0','1', 256};
    prod_rule rule = {str, NULL, 0};
    int len;

    len = rulelen(&rule);
    printf("%d\n", len);
    return(0);
}









#endif









#if 0



int maxlen;
int **array, *line;
int *x2, *y2, *x3, *y3;
int array_size;


/* 配列 array の空きを探す */
int empty_array()
{
    int i;

    for(i = 1; i < array_size; i++){
        if(array[i] == NULL){
            break;
        }
    }
    return i;
}
```

```
/* 配列が不足の時、追加確保する */
void increse_array()
{
    int i, i_len, j;
    int **new_array, *s0;
    int new_array_size;

    new_array_size = array_size * 2;

    new_array = (int **) calloc(new_array_size, sizeof(int *));
    if(new_array == NULL){
        printf("メモリが確保できません\n");
        exit(-1);
    }

    /* 新しい配列に古い配列をコピー */
    for(i = 0; i < array_size; i++){
        new_array[i] = array[i];
    }

    free(array);
    array = new_array;

    array_size = new_array_size;
    return;
}

int main(int argc, char *argv[]){
    int c, i, j, k, l, m = 0;
    int rr2, rr3;
    int gram_len = 0, num_char;

    FILE *fp;
    fp = fopen(argv[1],"rb");
    if(fp == NULL){
        printf("file open err\n");
        return(1);
    }
    for (num_char = 0; (c = fgetc(fp)) != EOF; num_char++){
          ;         /* 入力文字数のカウント */
    }

    /* 文字数 +1 個の int 型のメモリを動的に確保 */
    line = (int *) calloc(num_char + 1, sizeof(int));
    if(line == NULL){
        printf("メモリが確保できません\n");
        exit(-1);
    }

    rewind(fp);        /* ファイルの先頭に戻す関数 */

    /* 確保したメモリ上に書き込む */
    while((c = fgetc(fp)) != EOF){
        line[m] = c;
        m++;
    }
    line[num_char] = 256;
    fclose(fp);

    array_size = 5000;

    array = (int **) calloc(array_size, sizeof(int *));
    if(array == NULL){
        printf("メモリが確保できません\n");
        exit(-1);
    }

    array[0] = line;
```

```
    if(array[1] == NULL){
        if(same_pat2(array[0], &x2, &y2, &maxlen) > 1){
            reduct_2(0, empty_array(), x2, y2, maxlen);
        } else {
            printf("圧縮できません。\n");
            exit(-1);
        }
        while ((m = kmp_match(0, 1, maxlen)) > -1){
            reduct_4(0, 1, m);
        }
    }

    for(i = 0; i < empty_array() - 1; i++){
        for(j = i + 1; j < empty_array(); j++){
            rr2 = same_pat2(array[i], &x2, &y2, &maxlen);
            if(rr2 > 1){
            /*    printf("x2 :"); print_len(x2, rr2);
                printf("y2 :"); print_len(y2, rr2);*/
            }

            rr3 = same_pat3(i, j, &x3, &y3);
            if(rr3 > 1){
            /*    printf("x3 :"); print_len(x3, rr3);
                printf("y3 :"); print_len(y3, rr3);*/
            }

            if(rr2 >= rr3 && rr2 > 1){
                reduct_2(i, empty_array(), x2, y2, rr2);
                for(l = 0; l < empty_array() - 1; l++){
                    while ((m = kmp_match(l, empty_array() - 1, rr2)) > -1){
                        reduct_4(l, empty_array() - 1, m);
                    }
                }
            } else if(rr2 < rr3 && rr3 > 1){
                reduct_3(i, j, empty_array(), x3, y3, rr3);
                for(l = 0; l < empty_array() - 1; l++){
                    while ((m = kmp_match(l, empty_array() - 1, rr3)) > -1){
                        reduct_4(l, empty_array() - 1, m);
                    }
                }
            } else { /* rr2 = rr3 = -1 の場合 */
                ;
            }

            if(empty_array() == array_size){
                increse_array();
            }
        }
    }

    /* 文法を全て表示する */
    for(i = 0; array[i] != NULL; i++){
        printf("array[ %d] :", i); printrule(array[i]);
    }

    /* 文法の長さをカウントする */
    for(i = 0; array[i] != NULL; i++){
        m = rulelen(array[i]);
        gram_len += m;
    }
    printf("文法の長さは %d である。\n", gram_len);

    check_no_rule1(argv[2]);
    check_no_rule5(argv[3]);

    return 0;
}
```

```
#endif
```