信州大学工学部

学士論文

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

指導教員　西新 幹彦 准教授

学科　電気電子工学科
学籍番号　06T2031G
氏名　國安 隆治

2010 年 3 月 17 日

目次

1 **序章** 1

1.1 研究の背景と目的 1

1.2 本論文の構成 1

2 **文法圧縮とリダクションルール** 2

2.1 文字と文字列 2

2.2 生成規則と文法 2

2.3 文法圧縮 2

2.4 リダクションルール 2

2.5 逐次的アルゴリズム 4

3 **ルールの適用法** 5

3.1 提案法 5

3.2 提案法の狙い 5

3.3 実験とその結果 1 6

3.4 検証 1 6

3.5 実験とその結果 2 6

3.6 検証 2 8

3.7 階層的アルゴリズムと逐次的アルゴリズム 8

4 **まとめ** 9

謝辞 9

参考文献 9

付録 A ソースコード 11

A.1 リダクションルールを高速に適用するプログラム 11

1 序章

1.1 研究の背景と目的

近年, 情報化社会と言われるほど情報は重要なものであり情報を持つデータで溢れかえっている. 従って, データ圧縮が大きな意味を持つ. データ圧縮を行うことでデータを保存するのに必要な記憶容量の削減やデータ転送におけるコスト削減が出来る. データ圧縮とは, あるデータが持つ情報を保持したままデータ量を減らし別のデータに変換することである. 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, その文字列を別の文字列に置換することによってデータ量を小さくすることが出来る. データ圧縮は大きく分けると, 可逆圧縮と不可逆圧縮がある. 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出来るが, 不可逆圧縮では戻すことが出来ない. その可逆圧縮の中の一つに文法圧縮がある. 文法圧縮には,Yang and Kieffer[1] が提案した 5 つのリダクションルールがある. この 5 つのルールを繰り返し文法に適用することで圧縮出来るというものである. 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] や標準形を符号化するアルゴリズム [3] の研究, 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 [4] などがなされている. 本研究では, 逐次的に文法から 1 文字ずつ圧縮していくという逐次的アルゴリズムを採用し, ルールの適用の仕方によってより早く文法圧縮が出来るのではないかという考えに基づいて研究を行った. まずはじめに本論文の概要について述べる.

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. また, 本研究の主旨はリダクションルールの高速な適用なので圧縮率については深く言及しない.

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 [4] 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. その関係を式 (1) のように表す.

$$s_i \rightarrow G(s_i) \qquad (i \geq 0) \tag{1}$$

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

リダクションルールの文法圧縮は, 文法内の同じ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のにあたって 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,s_0 が表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.

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

[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,$s \to \alpha_1\beta\alpha_2\beta\alpha_3$ (α_1,α_2,α_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.

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

[ルール 3] s と s' の右辺に現れる長さ 2 以上の文字列 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.

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

[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.

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

[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' を全て 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.

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

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ということが [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 文字読み込んでは適用するためデータ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 ルールの適用法

3.1 提案法

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 [9] である. 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, ルール 2→3→4→1→5 とするものとする. また, ルールの適用法について以下にその手順を示す.

1. ルール 2 を s_0 にのみ適用.
2. ルール 3 を s_0 と他の生成規則の組に適用. (全ての生成規則が適用出来なくなるまで適用する. 追加された生成規則にも適用する.)
3. ルール 1 を適用.
4. ルール 5 を適用.
5. これまでの手順 1,2,3,4 においてルールが 1 度も適用されなかったら,irreducible な文法なので次の文字を読み込む.
6. ルール 2 を全ての生成規則に適用.
7. ルール 3 を全通り適用.
8. 手順 3,4,6,7 を irreducible な文法になるまで繰り返す.

プログラムの実装上, ルール 3 がルール 4 と同じ役割を果たしているためルール 4 は省略している.

3.2 提案法の狙い

手順 1,2,3,4(すなわち, ルール 2,3,1,5) を行うことで手順 1 においては s_0 以外のルール検索を, 手順 2 においては s_0 と他の生成規則以外のルール検索を行わないので余分な検索をする手間が省ける. また, 手順 1〜4 までにルールの適用が 1 度もない場合は次の文字を読み込むため必要以上のルール検索がなくなる. その結果, 全体の処理時間が短縮されるというのが提案法の狙いである.

3.3 実験とその結果 1

提案法を用いた場合とそうではない 2 つのプログラムを用いて生成規則数, 文法の大きさ, 処理時間を計る. そしてその結果を比較する. 実験結果を表 1 と表 2 に示す.

3.4 検証 1

表 1 と表 2 を比べてみると, 生成規則及び文法の大きさは変わらないのに対し, 処理時間は全てではないが提案法を適用した方が半分ほどになっている. ルールが適用された文法とそうでない文法の大きさはまったく同じで生成規則も同じである. このことから, 処理時間の短縮のために圧縮率を犠牲にしていないことがわかる. また, 必要ないルールの適用場所検索を行わないために処理時間が半分になったと考えられる. 各文法ファイルの大きさや中身にもよるが, 全ての処理時間が半分になることから文法の半分ほどはルールの適用が必要ではないと思われる.

表 1　提案法を用いた結果

ファイル名	ファイルサイズ [Byte]	生成規則数	文法の大きさ	処理時間 [s]
bib	111261	4619	26634	38928.968
book1	768771	–	–	–
book2	610856	–	–	–
geo	102400	4722	45919	119319.640
news	377109	–	–	–
obj1	21504	1299	8288	1145.656
obj2	246814	–	–	–
paper1	53161	2947	15853	9676.671
paper2	82199	3945	23278	24432.031
paper3	46526	2682	15387	7142.343
paper4	13286	1016	5432	359.546
paper5	11954	961	5040	287.687
paper6	38105	2359	12299	4606.265
pic	513216	–	–	–
progc	39611	2338	12221	4522.343
progl	71646	2988	14792	9317.765
progp	49379	2027	10005	3038.015
trans	93695	3239	16106	16355.234

3.5 実験とその結果 2

検証 1 より, 文法の半分ほどはルールの適用が必要ではないという予想を検討するために以下の実験を行った. 各ファイルにおけるルールが適用された回数, ルール適用の検索を行った回数を計測し, そこから文法の冗長度を求める. 文法の冗長度の求め方は式 (2) のようである. 実験結果を表 3 に示す.

表 2　提案法を用いない結果

ファイル名	ファイルサイズ [Byte]	生成規則数	文法の大きさ	処理時間 [s]
bib	111261	4619	26634	81534.265
book1	768771	−	−	−
book2	610856	−	−	−
geo	102400	4722	45919	178148.421
news	377109	−	−	−
obj1	21504	1299	8288	2039.359
obj2	246814	−	−	−
paper1	53161	2947	15853	19847.359
paper2	82199	3945	23278	47040.703
paper3	46526	2682	15387	14432.890
paper4	13286	1016	5432	713.875
paper5	11954	961	5040	577.531
paper6	38105	2359	12299	9323.468
pic	513216	−	−	−
progc	39611	2338	12221	8816.953
progl	71646	2988	14792	19428.234
progp	49379	2027	10005	6217.078
trans	93695	3239	16106	30290.656

表 3　実験結果

ファイル名	ファイルサイズ [Byte]	ルールが適用された回数	ルール適用の検索を行った回数	文法の冗長度 [%]
bib	111261	6852	28868	23.7
book1	768771	−	−	−
book2	610856	−	−	−
geo	102400	4848	46046	10.5
news	377109	−	−	−
obj1	21504	2107	9097	23.2
obj2	246814	−	−	−
paper1	53161	4468	17375	25.7
paper2	82199	5127	24461	21.0
paper3	46526	3631	16337	22.2
paper4	13286	1451	5868	24.7
paper5	11954	1422	5502	25.8
paper6	38105	3654	13595	26.9
pic	513216	−	−	−
progc	39611	3756	13640	27.5
progl	71646	5718	17523	32.6
progp	49379	4287	12266	34.5
trans	93695	8668	21536	40.2

$$\text{文法の冗長度} = \frac{\text{ルールが適用された回数}}{\text{ルール適用の検索を行った回数}} \times 100 \tag{2}$$

3.6　検証 2

表 3 を見てみると, ほとんどのファイルの文法の冗長度が 30 %に満たないことがわかる. これは, 文法の 6 から 7 割程度はルールの適用がされていないということである. また, サイズの大きいファイルでも冗長度が高いものや低いものもあり, その逆で小さいファイルでも冗長度

が高いものや低いものもある．このことから, 文法の冗長度において文法の大きさは関係がないことがわかる．そして, ルールが適用しやすい文法と適用しにくい文法があるといことが考えられる．これは圧縮率にも影響していると考えられる. つまり冗長度が高いと圧縮率が高く, 冗長度が低いと圧縮率は低くなると予想できる．

3.7 階層的アルゴリズムと逐次的アルゴリズム

階層的アルゴリズムと逐次的アルゴリズムの違いについて述べる．表 4 に文献 [2] による階層的アルゴリズムの実験結果を引用する．

表 4　階層的アルゴリズム [2] の実験結果

ファイル名	ファイルサイズ [Byte]	生成規則数	文法の大きさ	処理時間
bib	111261	5199	28180	18900
book1	768771	−	−	−
book2	610856	−	−	−
geo	102400	6499	52868	62820
news	377109	−	−	794160
obj1	21504	1241	9569	180
obj2	246814	10469	65761	153960
paper1	53161	3137	16707	3300
paper2	82199	4741	24883	15000
paper3	46526	3053	16260	3120
paper4	13286	1002	5579	60
paper5	11954	941	5196	60
paper6	38105	2342	12633	1200
pic	513216	7486	44844	215820
progc	39611	2262	12438	1200
progl	71646	2789	14978	2820
progp	49379	1907	10598	780
trans	93695	3088	16985	3180

表 1 と表 4 を比較してみると, 生成規則数はそれほど大きな違いはないが文法の大きさは全てのファイルにおいて表 1 の方が小さくなっている．しかし, 処理時間においては表 1 は表 4 に比べ非常に時間がかかっている．これは, 階層的アルゴリズムと逐次的アルゴリズムではまったく別のものであるからである．例を挙げるなら, 毎日届く新聞をデータ化して送るのに 365 日分を一気に送るのと,1 日分ずつ毎日送るのではまったく違う．つまり, 階層的アルゴリズムは文法全体を一度展開し一気にルールを適用するのに対し, 逐次的アルゴリズムは 1 文字読み込んで irreducible な文法になるまで検索し適用するので時間がかかってしまったと考えられる．

4 まとめ

本研究ではルールを高速に適用できないかと考え, ルールの適用法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, 逐次的アルゴリズ

ムという逐次的に文法から 1 文字ずつ圧縮するアルゴリズムを用いた．提案法は, 次のような順で文法から 1 文字読んでルールを適用していく. ルール 2 を s_0 にのみ適用, ルール 3 を s_0 と他の生成規則に適用, ルール 1 を適用, ルール 5 を適用し, これまでに 1 度もルールの適用がなかったら次の文字を読み込む．ルールの適用があったらルール 2 を全ての生成規則に適用, ルール 3 を全通り適用, あとは irreducible な文法になるまでルール 1,5,2,3 を繰り返すというものである．結果は, 提案法を用いることで各ファイルとも処理時間が半分ほどになった．これは, 必要ないルールの適用場所検索を行わないために処理時間が半分になったためと考えられる．各ファイルの大きさや中身にもよるが, 文法の半分ほどはルールの適用が必要でないことがわかった．また, リダクションルールを削減するアルゴリズムと実験結果を比較してみると処理時間が圧倒的にかかっていた．これは, 階層的アルゴリズムは文法全体を一度展開し一気にルールを適用するのに対し, 逐次的アルゴリズムは 1 文字読み込んで irreducible な文法になるまで検索し適用するので時間がかかってしまったと考えられる．

今後の課題として, 今以上の処理時間の短縮化が挙げれる．リダクションルールを削減するアルゴリズムを用いて適用するルールを減らすことで今以上に時間の短縮が可能であると考えられる．

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] 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.

[5] 湯淺太一, コンパイラ, 昭晃堂, 2005.

[6] 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.
[7] 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.
[8] ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus

付録 A　ソースコード

A.1　リダクションルールを高速に適用するプログラム

```
#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;
} 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);
      }

      *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];
		}
	}

	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);
}


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;

/* ファイルから１文字読み込む */
int
getsymbol()
{
      int symbol;

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

/* ファイルに１文字返す */
void
ungetsymbol(int symbol)
{
      if (stack_idx >= STACK_MAX){
            printf("overflow at %d\n", __LINE__);
            exit(1);
      }
      stack[stack_idx] = symbol;
      stack_idx++;
      return;
}


#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 = 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;
    }
    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){
			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;

	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++;
		}
	}

	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);
}


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

	input_fp = fopen(argv[1], "rb");
	if (input_fp == NULL){
		printf("open error on %d\n", __LINE__);
		exit(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();
	}

	printgrammar();

	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);

}
```