信州大学工学部

学士論文

文法圧縮における標準形を符号化するアルゴリズム

指導教員　西新 幹彦 准教授

学科　電気電子工学科
学籍番号　05T2048H
氏名　杉山 雅紀

2009 年 2 月 28 日

目次

1 序章

1.1 研究の背景と目的

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ことをデータ圧縮という. データ圧縮は、データ転送におけるコスト削減や, データを保存するのに必要な記憶容量の削減するのに用いられる. データ列には何度も出現する文字列が多く存在しているので, その文字列を別の文字列に置き換えることによってデータ量を小さくすることができる. データ圧縮には大きく分けて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できるが, 不可逆圧縮では戻すことができない. 可逆圧縮のひと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り, そのルールを繰り返し用いることでデータ量を削減することができる. 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 [3] などがなされている.

文法圧縮について Yang and Kieffer が提案した圧縮方法は, まず元のデータにリダクションルールを適用し, 文字列の置き換えをする. これを何回も繰り返し, これ以上リダクションルールを適用することができない形である中間コードを作る. この中間コードは符号化するには最適な形ではないので, 出現する順に変数を並べ替え, 初めて出現する変数を s と書き換える. これによって中間コードは符号化するのに最も適した形である標準形に直される. その標準形を符号化することで圧縮される. 標準形に表れる特徴を調べてみたところ,2 つの特徴を見つけた. 一つ目は, 文字列の長さが 3 以上の変数が並んでいる箇所が多く存在しているということである. つまりこれは, 長さ 3 以上の変数のときに用いる文字 b と e について,e の後ろに b が現れる箇所が多いということに等しい. e の後ろに b が現れる頻度はおよそ 3 分の 1 であることから. Yang and Kieffer が提案した方法では b の相対頻度で算術符号化を行うが, e の後に現れる文字が b である確率を条件付き確率 1/3 で符号化すれば圧縮率の向上につながるのではないかと考えた. 二つ目は, 長さ 3 以上の変数は標準形の前半に多く後半に少なくなるということである. Yang and Kieffer の提案法では出現した文字の頻度を増やしていくのだが, 長さ 3 以上の変数のときに用いる文字 b と e については, 後半になるにつれて出現頻度が小さくなる. よって頻度を増やさないことと,e の後に現れる文字が b である確率を条件付き確率 1/3 で符号化することを組み合わせて算術符号化を行った. これによって,e と b が多く出現する前半は 1/3 の確率で符号化でき, 後半では小さな確率で符号化することによって圧縮率の向上につながるのではないかと考えた.

本研究では, 文法圧縮の性能を向上させるようなアルゴリズムを提案し検証することを目的

とする.

1.2 本論文の構成

本論文では, 次のような構成をとる. 2 章では文法圧縮について説明する. 3 章では提案法についての説明と, 結果と評価を示す. 4 章ではまとめをする.

2 文法圧縮

文法圧縮は, 圧縮したい情報源系列を, 中間コード, 標準形, 符号語と形を変える作業が必要になる. これらの変形について,[2] を元に説明をする.

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し, 論文中では s_i などと表す. 終端文字や変数などの文字の並びを文字列と呼ぶ. 文法とは, 変数から文字列への写像のことであり, 記号 G で表す.

文法圧縮とは, 元のデータ列を中間コード, 標準形, 符号語と変形しながら圧縮されるデータ圧縮である.

2.1 生成規則

生成規則 [1] とは, 個々の変数 s_i と文字列 $G(s_i)$ の対応のことであり、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.

2.2 リダクションルール

文法圧縮をするとき, まず圧縮したいデータ列全体をメモリ上に展開して, 変数 s_0 とデータ列全体を対応させる. この 1 つの生成規則から成る文法を作る.s_0 が表す終端文字列を変えずに, リダクションルールという文法の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. ここで文法の大きさとは, 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 は 5 つのリダクションルールを提案した. それらを以下に示す. ここで α β γ は,2 文字以上の文字列を表す.

[ルール 1] 一度しか現れない生成規則を削除する.

$$\begin{aligned} s_i &\rightarrow \alpha s_j \beta \\ s_j &\rightarrow \ \gamma \\ &\Downarrow \\ s_i &\rightarrow \alpha\gamma\beta \end{aligned}$$

[ルール 2] 1 つの生成規則から重複文字列を見つけ, その重複文字列を表す生成規則を作る.

$$s_i \rightarrow \alpha_1\beta\alpha_2\beta\alpha_3$$

$$
\begin{array}{l}
\Downarrow \\
s_i \rightarrow \alpha_1 s_j \alpha_2 s_j \alpha_3 \\
s_j \rightarrow \quad \beta
\end{array}
$$

[ルール 3] 2 つの生成規則から重複文字列を見つけ, その重複文字列を表す生成規則を作る.

$$
\begin{array}{l}
s_i \rightarrow \alpha_1 \beta \alpha_2 \\
s_j \rightarrow \alpha_3 \beta \alpha_4 \\
\quad \Downarrow \\
s_i \rightarrow \alpha_1 s_k \alpha_2 \\
s_j \rightarrow \alpha_3 s_k \alpha_4 \\
s_k \rightarrow \quad \beta
\end{array}
$$

[ルール 4] 生成規則が次の生成規則の文字列の一部分であるとき, 文字列を変数で置き換え, 文字列の長さを短くする.

$$
\begin{array}{l}
s_i \rightarrow \alpha_1 \beta \alpha_2 \\
s_j \rightarrow \quad \beta \\
\quad \Downarrow \\
s_i \rightarrow \alpha_1 s_j \alpha_2 \\
s_j \rightarrow \quad \beta
\end{array}
$$

[ルール 5] 式 (1) より,2 つの生成規則の表す終端文字列が等しいとき, 片方の生成規則を削除する.

リダクションルール 1 から 5 のいずれも適用することができないような文法のことを,irreducible な文法という. irreducible な文法は, 文字列の長さや変数の数を減らすことができないので, 文法の大きさが最小になった形である.

Yang and Kieffer は 5 つのリダクションルールを提案したが、本研究ではリダクションルール 1 と 5 を必要としないアルゴリズム [2] を用いた. 以下にその手順を示す.

1. ルール 2 または 3 を適用できるかを検索する. 適用できる場所がなければ, 終了する.
2. 重複部分の長い方のルールを適用する. 長さが同じ場合はルール 2 を適用する.
3. これまでに作成された全ての生成規則に対して, ルール 4 を可能な限り適用する.
4. 手順 1 に戻る.

3 つのリダクションルールだけで,5 つ全てを使ったものと同じ能力を発揮できる. なぜなら,3 つのリダクションルールで irreducible な文法を作ることができるからである.

irreducible な文法を一列に連結することで中間コードが作られる.

2.3 算術符号

中間コードを算術符号化することで文法圧縮は完成する.

以下に算術符号についての説明を示す.

2.3.1 算術符号の符号化

算術符号は, 記号列を 0 から 1 の区間を用いて表す. 例えば, 記号は 1 と 0 の 2 種類があり, その出現確率をそれぞれ 0.2,0.8 とした場合, 区間を記号の出現確率に比例した小区間に分割していくことで符号化を行う. ここで例として, 記号列 01111 を符号化する. この過程を図 1 に示す. x 以上 y 未満の区間を $[x, y)$ と表すこととする. 区間の初期値は,[0,1) である. 記号

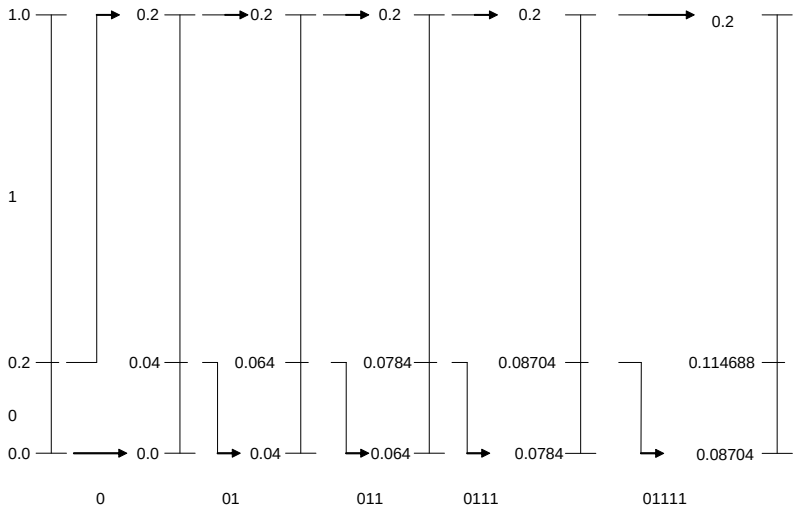

図 1　算術符号の符号化の過程

を読み込んだら区間 [0,1) を分割する. 記号が 1 ならば区間の 0 から 0.2 までの部分, 0 ならば 0.2 から 1.0 までの部分に分割する. 本論文では, 記号 1 と 0 の境界値のことを算術符号の境界値と呼ぶこととする.

最初の記号は 1 なので区間は,[0,0.2) である. 次の記号は 0 なので, 区間 [0,0.2) の 0.2 から 1.0 の部分 [0.04,0.20) が新しい区間である. このように, 記号を読み込むたびに区間を分割していくと, 記号列 01111 を表す区間は, [0.08704,0.2) となる. そして, 実際の算術符号の符号語は, この区間に含まれる一つの実数を指定する.

ここで符号語を 2 進数で表して, 区間内で小数点以下のビット数の少ない値を選ぶことにす

る. 例えば,0.125 を 2 進数で表すと次のようになる.

$$0.125 = 1/8 = (0.001)_2$$

$(001)_2$ の 3 ビットを符号語として出力すれば, 記号列 01111 の 5 文字を 3 ビットに圧縮することができるわけである.

2.3.2 算術符号の復号化

次に復号について説明する. 復号器が符号語 $(001)_2$ を受け取ったとき, 値を十進数の 0.125 に変換する. 0.125 は,[0,0.2) の間にあるので, 最初の記号は 0 であることがわかる. 次に 0 を表す区間 [0,0.2) を [0,1.0) になるように正規化すると, 符号語は次のように変換できる.

$$\begin{aligned} \text{正規化された算術符号の境界値} &= (\text{符号語} - \text{記号の下限値})/\text{記号の区間幅} \\ &= (0.125 - 0)/0.2 \\ &= 0.625 \end{aligned}$$

正規化された算術符号の境界値 0.625 は [0.2,1.0) の間にあるので、次の記号は 1 であることがわかる. このような操作を繰り返すことにより記号列 01111 を復号することができる.

2.3.3 適応的算術符号

算術符号は, あらかじめ与えられた出現確率に基づいて入力記号列を符号化していく方法である. 本研究では, データの読み込みと符号化を同時に進行させるので, 出現確率を調べておくことができない. そこで, 入力記号列の符号化を行いながら記号の出現確率を変化させる動的符号化である適応的算術符号を用いる. 最初は, どの記号も同じ確率で出現すると仮定して, 記号列を読み込む. 記号を読み込むごとに, その記号の出現頻度を更新し, 記号の符号化確率を修正し, その時点での符号化確率に基づいて記号の符号化を行う. この操作を繰り返すことで符号化される. 適応的算術符号の復号化は, 符号化を行ったときと同様の出現頻度に設定しておく. 符号語を復号するごとに, 復号された記号の出現頻度を更新し, 次の符号語はその時点での符号化確率に基づいて復号化される.

詳しい方法については次の標準形で説明する.

2.4 標準形

中間コードの意味する irreducible な文法の例を以下に示す.

$$\begin{aligned} s_0 &\rightarrow s_1 s_3 s_2 s_3 s_4 s_4 s_3 \\ s_1 &\rightarrow 100 \\ s_2 &\rightarrow s_1 0 \\ s_3 &\rightarrow s_4 s_2 \\ s_4 &\rightarrow 11 \end{aligned}$$

本研究では, 中間コードを標準形 (canonical form[1]) という形に直しながら算術符号化を実行する. irreducible な文法の例を用いて, 以下にその手順を示す.

1. 変数を出現する順に並べ替え, このとき変数の添え字を出現する順に大きくなるように置き換える. つまり s_i は s_{i+1} よりも必ず先に出現することになる.

$$\begin{array}{lcl} s_0 & \rightarrow & s_1 s_2 s_3 s_2 s_4 s_4 s_2 \\ s_1 & \rightarrow & 100 \\ s_2 & \rightarrow & s_4 s_3 \\ s_3 & \rightarrow & s_1 0 \\ s_4 & \rightarrow & 11 \end{array}$$

2. 変数 s_0 から s_i を添え字の小さい順に連結する. 変数の長さは, ほとんどが 2 になるので, 長さ 3 以上の変数は最初に b 最後に e の文字を入れて挟み, 長さ 2 の変数は何も挟まずに繋げる. ただし s_0 に関しては長さには関係なく文字列の終わりに b を入れる.

$$s_1 s_2 s_3 s_2 s_4 s_4 s_2 e b 100 e s_4 s_3 s_1 011$$

3. それぞれの添え字が最初に現れるとき, 添え字の番号を取る.

$$ssss_2 ss_4 s_2 e b 100 e s_4 s_3 s_1 011$$

4. 適応化算術符号を実行する. ここでは, すでに符号化したシンボルの出現頻度に応じて, 確率分布を符号化の途中で更新していく適応的算術符号を使う. 初期設定は変数以外の文字と b,e,s を 1 としておく. 変数 s_1 から s_i は 0 としておく. 連結した文字列を読んでいき, 現れた文字の頻度に 1 を加える.s が現れたときは, その s が添え字を取る前の s_x の文字も 1 を加える.

$$\begin{array}{c} c(1)=1, c(0)=1, c(b)=1, c(e)=1, c(s)=1, c(s_1)=0, c(s_2)=0, c(s_3)=0, c(s_4)=0 \\ \Downarrow \\ c(1)=1, c(0)=1, c(b)=1, c(e)=1, c(s)=\mathbf{2}, c(s_1)=\mathbf{1}, c(s_2)=0, c(s_3)=0, c(s_4)=0 \\ \Downarrow \\ c(1)=1, c(0)=1, c(b)=1, c(e)=1, c(s)=\mathbf{3}, c(s_1)=1, c(s_2)=\mathbf{1}, c(s_3)=0, c(s_4)=0 \\ \Downarrow \\ c(1)=1, c(0)=1, c(b)=1, c(e)=1, c(s)=\mathbf{4}, c(s_1)=1, c(s_2)=1, c(s_3)=\mathbf{1}, c(s_4)=0 \\ \Downarrow \\ c(1)=1, c(0)=1, c(b)=1, c(e)=1, c(s)=4, c(s_1)=1, c(s_2)=\mathbf{2}, c(s_3)=1, c(s_4)=0 \\ \Downarrow \\ \vdots \end{array}$$

この操作を繰り返し行う. 符号化確率は,

$$c(\beta)/\sum_i |c(\alpha)|$$

である. ここで $c(\beta)$ は読み込んだ記号の出現頻度で, $\sum_i |c(\alpha)|$ は全ての記号の出現頻度の総和である.

3 標準形の符号化

3.1 本研究の提案 1

表 1 に標準形に現れた特徴を示す. ここに示したファイルはカルガリーコーパスファイル [8] である. 表 1 の結果は book1, book2, news, pic, progc, progl, progp, trans のファイルには CPU:Intel Pentium (3.40GHz), メモリ:1GB のコンピュータを使用し, それ以外のファイルには CPU:Intel Pentium(2.80GHz), メモリ:1GB, のコンピュータを使用した. $p(b|e)$ は e が出現した後に b が出現する条件付確率であり $p(b)$ は b が出現する確率である. book1,book2 の欄が空白になっている理由は, 中間コードへの変形に非常に時間がかかるため, 実験を断念したためである. ちなみに book1,book2 の中間コードへの変換には,30 日以上かかることは分かった.

表 1　中間コードに現れた特徴

ファイル名	サイズ [byte]	生成規則数	文法の大きさ	処理時間	長さ 3 以上の変数	$p(b)$	$p(b\|e)$
bib	111261	5199	28180	7 時間 12 分	1471	0.05	0.31
book1	768771						
book2	610856						
geo	102400	6499	52868	18 時間 5 分	661	0.01	0.11
news	377109	17603	99567	252 時間 28 分	4586	0.04	0.31
obj1	21504	1241	9569	5 分	326	0.03	0.33
obj2	246814	10469	65761	61 時間 51 分	3091	0.05	0.36
paper1	53161	3137	16707	1 時間 14 分	999	0.06	0.32
paper2	82199	4741	24883	5 時間 45 分	1335	0.05	0.29
paper3	46526	3053	16260	1 時間 11 分	958	0.06	0.36
paper4	13286	1002	5579	2 分	332	0.06	0.36
paper5	11954	941	5196	1 分	320	0.06	0.40
paper6	38105	2342	12633	28 分	750	0.06	0.36
pic	513216	7486	44844	67 時間 29 分	1753	0.04	0.30
progc	39611	2262	12438	26 分	714	0.06	0.38
progl	71646	2789	14978	1 時間 13 分	963	0.06	0.41
progp	49379	1907	10598	13 分	597	0.06	0.37
trans	93695	3088	16985	1 時間 13 分	974	0.06	0.39

この表から,e の後に b が出現する確率は b が出現する相対頻度に比べて大きいことがわかった. 算術符号での符号化は, 圧縮率に出現頻度が大きく関係してくるので, この特徴を利用すれば圧縮率を向上させることができるのではないかと考えた. 表 1 の e の後に b が出現する条件付き確率がおよそ 3 分の 1 になっていることから, e の後に b が出現する条件付き確率は 3 分の 1 とし, e の後に現れる b 以外の文字も $p(a|e) = \frac{2}{3}\ p(a)$ の条件付き確率で符号化することを提案した. ここで a は b 以外の全ての文字を表す.

3.2 提案 1 の検証結果

どの程度の圧縮の向上が期待できるか, 中間コードに現れた結果と下式から見積もった. N は e の次に b が出た回数であり,M は e が出た回数である. a は b 以外の全ての文字を表す.

$$\begin{aligned}\text{減少サイズの見積もり値} &= N\left(\log\frac{1}{p(b|e)} - \log\frac{1}{p(b)}\right)\\ &\quad +(M-N)\left(\log\frac{1}{p(a|e)} - \log\frac{1}{p(a)}\right)\\ &= N\left(\log 3 - \log\frac{1}{p(b)}\right) + (M-N)\log\frac{3}{2}\end{aligned}$$

見積もった値と算術符号化を実行した結果を表 2 に示した. news については, 算術符号の精度が足りなくなったため結果が出なかった.

表 2　提案 1 の検証結果

ファイル名	見積もり値 [byte]	実際の減少サイズ [byte]
bib	76.2667	122
geo	0.2470	0
news	311.4574	−
obj1	28.0559	46
obj2	244.7795	356
paper1	49.5636	96
paper2	56.2029	90
paper3	64.6065	114
paper4	21.8606	44
paper5	24.6356	47
paper6	48.5932	90
pic	114.0978	133
progc	52.8604	100
progl	75.3979	136
progp	43.0364	76
trans	75.9231	128

表 2 から, 提案 1 の方法によって圧縮サイズが減少することがわかった. 減少サイズを見積り値よりも実際に提案方法で圧縮した値のほうが圧縮されている. これは, 見積もりは標準形の値から求めた値であるが, 実際の圧縮は標準形の文字列を読み込み見ながら逐次的に操作をするので, その時点での最適な値で符号化されるために, 見積もり値よりも圧縮されたと考えられる.

3.3 提案 2

文字列の長さが 3 以上の変数に用いる文字 e と b の分布から, これら 2 つの文字の頻度のカウントについて工夫することによって圧縮率を向上できないかと考えた. 実験を行ったカルガリーコーパスファイルについて, 文字列の長さが 3 以上の変数が現れる分布を, 図 2〜17 に示す.

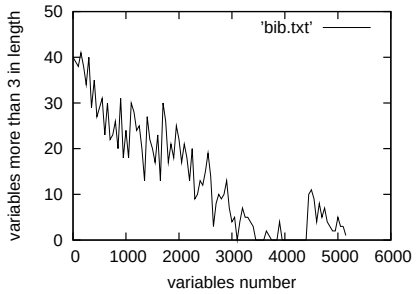

図 2　長さ 3 以上の変数が現れる分布 (bib)

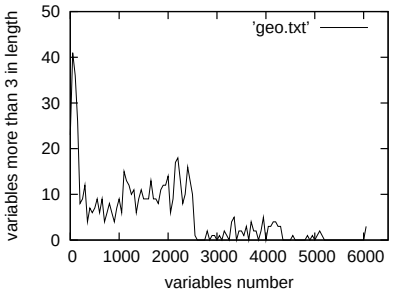

図 3　長さ 3 以上の変数が現れる分布 (geo)

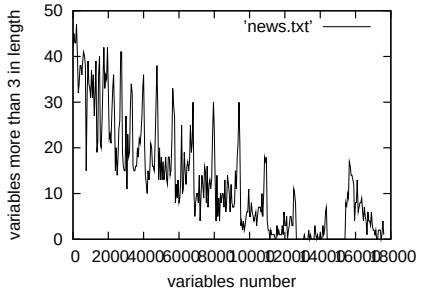

図 4　長さ 3 以上の変数が現れる分布 (news)

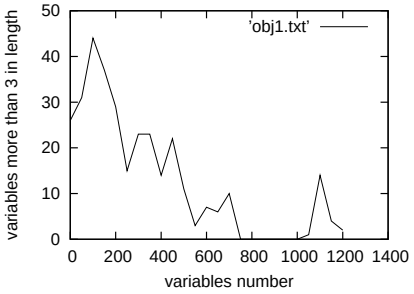

図 5　長さ 3 以上の変数が現れる分布 (obj1)

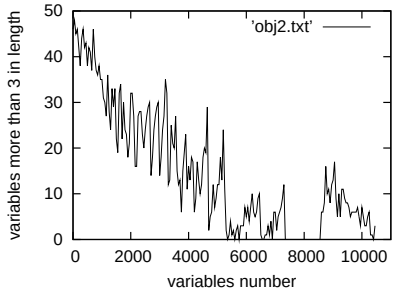

図 6　長さ 3 以上の変数が現れる分布 (obj2)

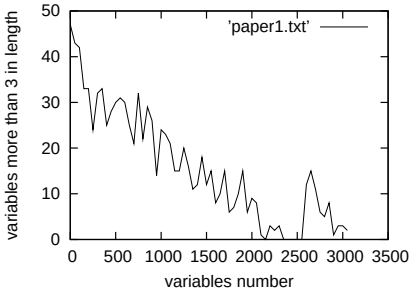

図 7　長さ 3 以上の変数が現れる分布 (paper1)

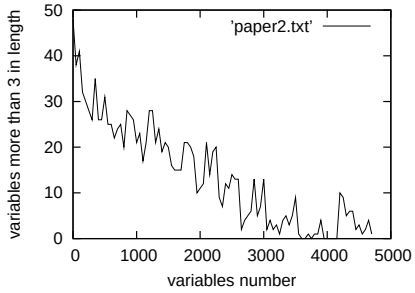

図 8　長さ 3 以上の変数が現れる分布 (paper2)

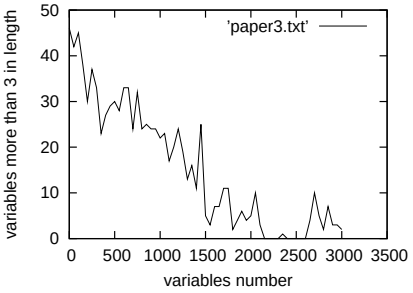

図 9　長さ 3 以上の変数が現れる分布 (paper3)

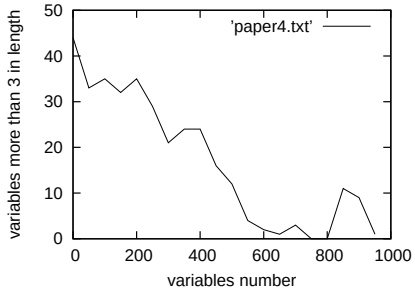

図 10　長さ 3 以上の変数が現れる分布 (paper4)

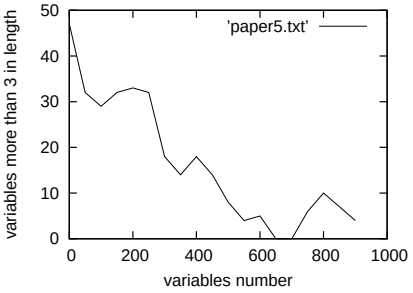

図 11　長さ 3 以上の変数が現れる分布 (paper5)

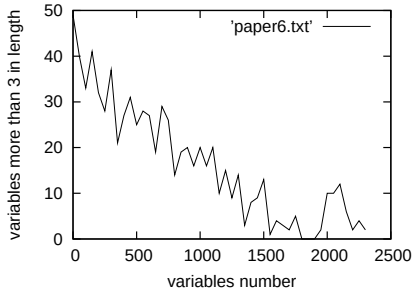

図 12　長さ 3 以上の変数が現れる分布 (paper6)

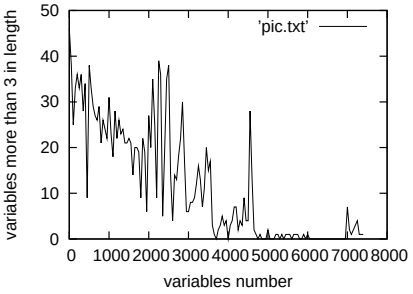

図 13　長さ 3 以上の変数が現れる分布 (pic)

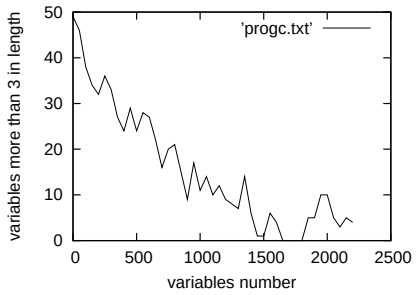

図 14　長さ 3 以上の変数が現れる分布 (progc)

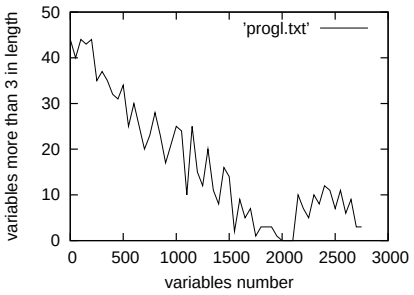

図 15　長さ 3 以上の変数が現れる分布 (progl)

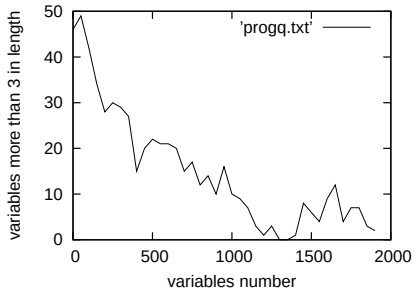

図 16　長さ 3 以上の変数が現れる分布 (progq)

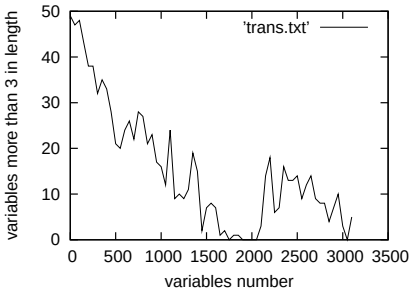

図 17　長さ 3 以上の変数が現れる分布 (trans)

これらの図から, 長さ 3 以上の変数は前半に多く後半に少なくなることがわかった. 符号化をする際にこの特徴を利用し, 長さ 3 以上の変数のときにしか使用されない e と b の頻度のカウントを工夫することによって, 圧縮率を向上させることができるのではないかと考えた. 提案 1 と b の頻度を 1 で固定することを組み合われて符号化することを提案した. こうすることで, 長さ 3 以上の変数が多く出現する前半では 3 分の 1 の頻度で符号化し, 後半では小さい頻度で符号化することができると考えた.

3.4 提案 2 の検証結果

表 3　提案 2 の検証結果

ファイル名	提案 1 での圧縮ファイルサイズ [byte]	提案 2 での圧縮ファイルサイズ [byte]
bib	34677	37149
geo	64722	65764
news	−	160657
obj1	10842	11290
obj2	87351	92224
paper1	19762	21394
paper2	29997	32275
paper3	19063	20578
paper4	5997	6438
paper5	5560	5971
paper6	14635	15806
pic	55004	56146
progc	14484	15578
progl	17805	19311
progp	12287	13161
trans	20629	22125

提案 2 の検証結果を表 3 に示した. 提案 2 の方法では圧縮サイズは減少しないことがわかった. 図 2 から, 長さ 3 以上の変数の数が中盤から大きく減少しているが, 終わる直前では長さ 3 以上の変数が増えていることがわかるので, この部分で損をしてしまっていることが考えられた. news が提案 1 では算術符号の精度が足りなくなってしまったために符号語を作ることができなかったが, 提案 2 では,b の頻度を数えなかったために精度が足りなくなることはなく, 符号語が作られた.

3.5 提案 3,4

実際に実験は行わなかったが, あと 2 つ圧縮の向上につながるのではないかと考えた方法があるので紹介する.

1. 長さ 3 以上の変数の最初と最後に挟む文字である b と e について,b が現れたあとにのみ e が現れるので, b が現れたときにだけ e の確率を考える.
2. 提案 1 において,e が出た後の条件付き確率で,b が出る確率と b 以外が出る確率を考えるときに, b 以外の確率に b の確率が含まれてしまっているので,b 以外の確率の時には b の確率は考えない.

この 2 つの方法について, 標準形からわかる数値を元にどれくらいの減少が期待できるかを見積もった. その値を表 4 に示す.

表 4　提案 2 の検証結果

ファイル名	提案 3 での減少サイズ [bit]	提案 4 での減少サイズ [bit]
bib	0.0773	0.0479
geo	0.0182	0.0110
news	0.0663	0.0383
obj1	0.0500	0.0306
obj2	0.0695	0.0429
paper1	0.0890	0.0553
paper2	0.0796	0.0493
paper3	0.0876	0.0545
paper4	0.0885	0.0550
paper5	0.0917	0.0571
paper6	0.0883	0.0549
pic	0.0575	0.0354
progc	0.0853	0.0530
progl	0.0959	0.0598
progp	0.0836	0.0519
trans	0.0844	0.0524

表からも分かるように, どのファイルでも 1bit 以下の減少しか見込まれないことがわかった. そのために実験は行わなかった.

4 まとめ

文法圧縮の標準形の形に現れた特徴を用いて, 圧縮率を向上させることについて実験を行った. 1 つ目の提案は, 長さ 3 以上の変数が並んでいる箇所が多く存在することに着目し, 長さ 3 以上の変数を表すときに用いる文字 b と e について,e の後に現れる文字が b かその他の文字であるかの条件付き確率で符号化すれば圧縮率の向上につながるのではないかと考えた. 結果, 標準形に現れた値からどの程度減少するかを見積もった値よりも圧縮サイズが減少した. 2

つ目の提案は, 長さ 3 以上の変数は標準形の前半に多く後半に少なくなる点に着目した. 長さ 3 以上の変数を表すときに用いる文字 b と e について, 頻度の数え方を工夫すれば圧縮率の向上につながるのではないかと考えた. この提案方法では圧縮サイズを減少させることはできなかった.

今後の課題としては, 元のファイルから標準形に変換するまでの処理時間の短縮が挙げられる. 文法圧縮には,Yang and Kieffer[1] が提案した方法で Hirarchical アルゴリズムと Sequential アルゴリズムがあり, 今回は Hirarchical アルゴリズムを用いた. Hirarchical アルゴリズムは, ファイル全体を一度に読み込んで文字列を操作するものであるために処理時間が長くなってしまうという欠点があった. Sequential アルゴリズムはファイルの読み込んだ部分から文字列操作をするアルゴリズムであるので, 処理時間を短縮することができると考えられる.

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] 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.

[4] 湯淺太一, コンパイラ, 昭晃堂, 2005.

[5] John C. Kieffer, En-hui Yang,“ Grammar-Based Codes: A New Class of Universal Lossless Source Codes, ” IEEE Transactions on Information Theory, vol.46, no.3, pp.737–754, 2000.

[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　元のデータから中間コード作成用プログラム

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int rulelen(int *rule)
{
    int num;
    for (num = 0; rule[num] != 256; num++){  /* rule の長さのカウント */
        ;
    }
    return(num);
}

/* int 型の表示ルールの宣言 */
void printrule(int *s)
{
    int n;
    do {
        if (*s < 256){
            if (iscntrl((char)*s)){
                printf(" %02x", (char)*s);
            } else {
                printf("  %c", (char)*s);
            }
        } else {
            printf(" %d%c", *s - 256, '*');
        }
    } while (*s++ != 256);
    printf("\n");
return;
}

/* *s の先頭文字から長さ n だけを表示する */
void print_len(int *s, int n)
{
    for (;n > 0; n--, s++){
        if(*s < 256){
            if(iscntrl((char)*s)){
                printf(" %02x", (char)*s);
            } else {
                printf("  %c", (char)*s);
            }
        } else {
            printf(" %d%c", *s - 256, '*');
        }
    }
    printf("\n");
}

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;
}

/* 重複パターンを表示するプログラム */
int same_pat2(int *line, int **x2, int **y2, int *n)
{
    int len1 = 0, len2 = 0;
    int i, i_mem, i_len;
    int dist, dist_mem;
    i_len = rulelen(line);
    *n = 1;

    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 > *n){
                    *n = 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 > *n){
                        *n = 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(*n == dist){
                break;
            }
            len1 = len2 = 0;
```

```
        }
    if(*n < 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 - *n];
    *y2 = &line[i_mem + 1 - *n + dist_mem];
    return (*n);
}

/* リダクションルール 2 の適用 */
void reduct_2(int i, int j, int *x2, int *y2, int n)
{
    int k, l;
    int *s0, *s1;
    int i_len;

    i_len = rulelen(array[i]);
    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 (array[i] + l == x2 || array[i] + l == y2){
            s0[k] = 256 + j;
            l += n - 1;
        } else {
            s0[k] = *(array[i] + l);
        }
    }

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

    free(array[i]);

    array[i] = s0;
    array[j] = s1;

    return;
}

/* KMP 法による文字列検索 */
int kmp_match(int i, int j, int n)
{
    int pt = 1, pp = 0;
    int i_len, j_len;
    int *skip;

    i_len = rulelen(array[i]);
    j_len = rulelen(array[j]);

    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 (*(array[j] + pt) != 256){
        if(*(array[j] + pt) == *(array[j] + 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 < n; ){
        if(*(array[i] + pt) == *(array[j] + pp)){
            pt++; pp++;
        } else if(pp == 0){
            pt++;
        } else {
            pp = skip[pp];      /* ここがポイント */
        }
    }
    free(skip);
    if(pp == n){
        return (pt - pp);
    }
    return (-1);
}

/* リダクションルール 4 の適用 */
void reduct_4(int i, int j, int m)
{
    int k, l, i_len, j_len;
    int *t0;

    i_len = rulelen(array[i]);
    j_len = rulelen(array[j]);

    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] = 256 + j;
            l += j_len - 1;
        } else {
            t0[k] = *(array[i] + l);
        }
    }
    free(array[i]);
    array[i] = t0;

    return;
}

int maxl;

/* 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 same_pat3(int i, int j, int **x3, int **y3)
{
    int pt, pp, pl;
    int i_len, j_len, samelen = 0;
```

```
int max_pt, max_pp;

i_len = rulelen(array[i]);
j_len = rulelen(array[j]);

maxl = 1;

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(*(array[i] + pt) == *(array[j] + 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(*(array[i] + pt) == *(array[j] + 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(*(array[i] + pt) == *(array[j] + 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(*(array[i] + pt) == *(array[j] + 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(*(array[i] + pt) == *(array[j] + 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(*(array[i] + pt) == *(array[j] + 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 = array[i] + max_pt - maxl + 1;
        *y3 = array[j] + max_pp - maxl + 1;
        return(maxl);
    } else {
        return(-1);
    }
}

/* リダクションルール 3 の適用 */
void reduct_3(int i, int j, int k, int *x3, int *y3, int n)
{
    int i_len, j_len;
    int *u0, *u1, *u2;
    int rl, rk;
```

```
i_len = rulelen(array[i]);
j_len = rulelen(array[j]);

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(array[i] + rl == x3){
                if(j_len == n){
                    u0[rk] = 256 + j;
                } else {
                    u0[rk] = 256 + k;
                }
                rl += n - 1;
            } else {
                u0[rk] = *(array[i] + 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(array[j] + rl == y3){
                u1[rk] = 256 + k;
                rl += n - 1;
            } else {
                u1[rk] = *(array[j] + rl);
            }
        }
        for(rl = 0; rl < n; rl++){
            u2[rl] = y3[rl];
        }
        u2[n] = 256;

        free(array[j]);
        array[j] = u1;
        array[k] = u2;
    } else if(j_len == n){
        free(u2);
    }
    free(array[i]);
    array[i] = 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(array[j] + rl == y3){
                if(i_len == n){
                    u1[rk] = 256 + i;
                } else {
                    u1[rk] = 256 + k;
                }
                rl += n - 1;
            } else {
                u1[rk] = *(array[j] + 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(array[i] + rl == x3){
                    u0[rk] = 256 + k;
                    rl += n - 1;
                } else {
                    u0[rk] = *(array[i] + rl);
                }
            }
            for(rl = 0; rl < n; rl++){
                u2[rl] = x3[rl];
            }
            u2[n] = 256;

            free(array[i]);
            array[i] = u0;
            array[k] = u2;
        } else if(i_len == n){
            free(u2);
        }
        free(array[j]);
        array[j] = u1;
    }

    return;
}

void
check_no_rule1(char *output)
{
    int i, k;
    int count[100000];
    FILE *fp;

    fp = fopen(output, "w");
    if(fp == NULL){
        printf("file open err\n");
        exit(-1);
    }

    for (i = 0; i < 100000; i++){
count[i] = 0;
    }

    /* ルールが使われた回数をカウントする */
    for(i = 0; array[i] != NULL; i++){
        for(k = 0; *(array[i] + k) != 256; k++){
            if(*(array[i] + k) > 256){
                count[*(array[i] + k) - 256] += 1;
            }
        }
    }

    for(i = 1; array[i] != NULL; i++){
        fprintf(fp, "%5d* : %3d 回\n", i, count[i]);
        if(count[i] < 2){
            fprintf(fp, "       1 回のみ使われるルールが見つかりました\n");
        }
    }
    for(i = 1; array[i] != NULL; i++){
        if(count[i] < 2){
            fprintf(fp, "%5d* : %3d 回", i, count[i]);
            fprintf(fp, "       1 回のみ使われるルールが見つかりました\n");
        }
    }

    fclose(fp);

    return;
}
```

```
void
printrule_rec(FILE *fp, int *s)
{
    int n;

    for (; *s != 256; s++){
        if (*s < 256){
            fprintf(fp, " %02x", *s);
        } else {
            printrule_rec(fp, array[*s - 256]);
        }
    }
    return;
}

void
check_no_rule5(char *output)
{
    int i;
    FILE *fp;

    fp = fopen(output, "w");
    if(fp == NULL){
        printf("file open err\n");
        exit(-1);
    }

    /* ルールを代入する */
    for(i = 1; array[i] != NULL; i++){
        fprintf(fp, "a[%5d] :", i);
        printrule_rec(fp, array[i]);
        fprintf(fp, "\n");
    }

    fclose(fp);

    return;
}

/* 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;
}

/* 圧縮する直前の文字列の表示 */
```

```
int character(char *outfile)
{
    FILE *fp;
    int b = 257;
    int e = 258;
    int c;
    int i;
    int j;

    printf("DEBUG %d\n", __LINE__);
    fp = fopen(outfile, "wb");

    if (fp == NULL){
        perror("ファイルのオープンに失敗しました.\n");
        exit(1);
    }else{
        printf("ファイルを正常にオープンしました.\n");
    }

    for (i = 0; array[i] != NULL; i ++){

        if(rulelen(array[i]) > 2 && i != 0){
            if(fwrite(&b,sizeof(int),1,fp) != 1){
                printf("書き込みに失敗しました.\n");
                exit(1);
            }
        }
        for(j = 0; array[i][j] != 256; j ++){
            if(array[i][j] < 256){
                c = array[i][j];
            } else {
                c = array[i][j]+2;
            }

            if(fwrite(&c,sizeof(int),1,fp) != 1){
                printf("書き込みに失敗しました.\n");
                exit(1);
            }

        }
        if(rulelen(array[i]) > 2){
            if(fwrite(&e,sizeof(int),1,fp) != 1){
                printf("書き込みに失敗しました.\n");
                exit(1);
            }
        }
    }
    fclose(fp);

    return 0;
}



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]);
    */

    /* 圧縮する直前の文字列の表示 */
    character(argv[2]);
    return 0;
}
```

A.2 中間コードから標準形に変換するプログラム

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_VAR_NUM 10000

int *array1[MAX_VAR_NUM];
int array2[MAX_VAR_NUM];
int *array3[MAX_VAR_NUM];

int rulelen(int *rule)
{
    int num;

    for (num = 0; rule[num] != 256; num++){  /* rule の長さのカウント */
;
    }
    return(num);
}


/* 中間コードを変数と文字列の形に戻す */
void sequence(FILE *fp)
{
    int n = 257;
    int m;
    int i = 0;
    int j;

    do {
        if (i >= MAX_VAR_NUM){
```

```
            printf("MAX_VAR_NUM を増やしてください.\n");
            exit(1);
        }
        if(n == 257){   /* S の長さが 3 以上のとき */
            for(m = 1; fread(&n, sizeof(int), 1, fp), n != 258; m++){    /* S 0 の長さを測る */
                ;
            }
            array1[i] = (int*)calloc(m, sizeof(int));  /* メモリの確保 */
            if(array1[i] == NULL){
                perror("メモリの動的確保に失敗しました\n");
                exit(1);
            }
            fseek(fp, sizeof(int) * (-m), SEEK_CUR);   /* 長さを数えた分だけ n を巻き戻す */
            for (j = 0; m > 1; m--, j++){
                fread(&n, sizeof(int), 1, fp);
                array1[i][j] = n;
            }
            fread(&n, sizeof(int), 1, fp); /* e を読み飛ばす */
            array1[i][j] = 256;
        } else {   /* S の長さが 2 のとき */
            array1[i] = (int*)calloc(3, sizeof(int));  /* メモリの確保 */
            if(array1[i] == NULL){
                perror("メモリの動的確保に失敗しました\n");
                exit(1);
            }
            array1[i][0] = n;
            fread(&n, sizeof(int), 1, fp);
            array1[i][1] = n;
            array1[i][2] = 256;
        }
        i++;
    } while (fread(&n, sizeof(int), 1, fp) == 1);
    return;
}

/* canonical form に変換 */

void canonical(FILE *cwfp)
{
    int i;
    int j;
    int n = 1;
    int c;
    int b = 257;
    int e = 258;

    array3[0] = array1[0];

    for(i = 0; array3[i] != 0; i++){  /* array1 と array3 の対応表を見る */
        int len = rulelen(array3[i]);
        if (len > 2 && i != 0){
            if (fwrite(&b, sizeof(int), 1, cwfp) != 1){
                printf("書き込みに失敗しました\n");
                exit(1);
            }
        }
        for(j = 0; array3[i][j] != 256; j++){
            if(array3[i][j] > 258){
                if(array2[array3[i][j] - 258] == 0){
                    array2[array3[i][j] - 258] = n;
                    array3[n] = array1[array3[i][j] - 258];
                    c = 256;
                    n++;
                }else{
                    c = array2[array3[i][j] - 258] + 258;
                }
            }else{
                c = array3[i][j];
            }
```

```
            if(fwrite(&c, sizeof(int), 1, cwfp) != 1){
                printf("書き込みに失敗しました\n");
                exit(1);
            }
        }
        if (len > 2){
            if (fwrite(&e, sizeof(int), 1, cwfp) != 1){
                printf("書き込みに失敗しました\n");
                exit(1);
            }
        }
    }
    return;
}


int
main(int argc, char *argv[])
{
    FILE *fp;
    FILE *cwfp;

    if (argc != 3){
        perror("引数の数が違います.\n");
        exit(1);
    }
    fp = fopen(argv[1], "rb");
    if (fp == NULL){
        perror("入力ファイルのオープンに失敗しました.\n");
        exit(1);
    }else{
        printf("入力ファイルを正常にオープンしました.\n");
    }
    cwfp = fopen(argv[2], "wb");
    if (fp == NULL){
        perror("出力ファイルのオープンに失敗しました.\n");
        exit(1);
    }else{
        printf("出力ファイルを正常にオープンしました.\n");
    }

    sequence(fp);
    canonical(cwfp);
    fclose(fp);
    fclose(cwfp);

    return 0;
}
```

A.3 適応的算術符号の符号化

```
#include <stdio.h>
#include <stdlib.h>
#include "arithmetic_coding.h"
#include "model.h"

static int buffer;
static int bits_to_go;
static int garbage_bits;
FILE *cwfp; /* 出力ファイル */

/* ビットのバッファ */
```

```
static int buffer;
static int bits_to_go;

/* 先頭のビット出力 */
start_outputing_bits(){
    buffer = 0;
    bits_to_go = 8;
}

/* ビットの出力 */
output_bit(int bit)
{   buffer >>= 1;
    if (bit) buffer |= 0x80;
    bits_to_go -= 1;
    if(bits_to_go==0){       /* バッファがいっぱいになったら出力する */
        putc(buffer, cwfp);
        bits_to_go = 8;
    }
}

/* 最後のビットを取り除く */
done_outputing_bits(){
    putc(buffer>>bits_to_go, cwfp);
}

/* 算術符号のアルゴリズム */
static void bit_plus_follow();


/* 符号化の状態の流れ */
static code_value low, high;
static long bits_to_follow;

/* シンボルの符号化のスタート */
start_encoding(){
    low = 0;
    high = Top_value;
    bits_to_follow = 0;
}


/* シンボルの符号化 */
void
encode_symbol(int symbol, unsigned long *cum)
{
    long range;                                 /* 符号の区域の大きさ */

    range = (long)(high-low)+1;
    high = low +
        (range*cum[symbol])/cum[0]-1;
    low = low +
        (range*cum[symbol+1])/cum[0];
    if ( cum[symbol] != cum[symbol+1] ){
        if (low == high + 1){
            printf("精度が足りません.\n");
            exit(1);
        }
    }else{
        printf("頻度が0です.%d\n", symbol);
        exit(1);
    }

    for (;;) {                          /* ビットを出力するためのループ */

        if (high<Half) {
            bit_plus_follow(0);         /* 下半分なら0を出力 */
        }
        else if (low >= Half) {         /* 上半分なら1を出力 */
            bit_plus_follow(1);
```

```
            low -= Half;
            high -= Half;
        }
        else if (low >= First_qtr && high<Third_qtr){
            bits_to_follow += 1;
            low -= First_qtr;
            high -= First_qtr;
        }
        else break;
        low = 2*low;
        high = 2*high+1;      /* 符号の列の目盛りを拡大する */
    }
    return;
}

/* 符号化の終わり */
done_encoding(){
    bits_to_follow += 1;
    if (low<First_qtr) bit_plus_follow(0);
    else bit_plus_follow(1);
}


/* ビットの出力 */
static void bit_plus_follow(bit)
    int bit;
{   output_bit(bit);
    while (bits_to_follow>0){
        output_bit(!bit);
        bits_to_follow -= 1;
    }
}



/* 適応化算術符号のモデル */
/* cum_freq[0〜255] = 文字
   cum_freq[256] = s
   cum_freq[257] = b
   cum_freq[258] = e
   cum_freq[259〜] = 変数 */

/* モデルの先頭 */
void
start_model(){
    int i;

    for(i = 0; i < 259; i++){      /* 1,0,e,b,s に初期設定の１を入れておく */
        cum_freq[i] = 259 - i;
    }
    for(i = 259; i < No_of_rules+259 ;i++){    /* 変数に 0 を代入 */
        cum_freq[i] = 0;
    }
    next_blank = 259;
    return;
}

/* void cum(){
    int i;
    for(i = 0; i < 267; i++){
        printf("%d\n", cum_freq[i]);
    }
    return;
}*/

/* 新しいシンボルの解釈のためのモデルのアップデート */

int antiunderflow = 0;
```

```
void
update_model(int symbol)
{

    if (cum_freq[0] >= Max_frequency){     /* 頻度の回数が最高値を超えたときの処理 */
        int freq[No_of_chars + No_of_rules];
        int i;
        int cum = 0;

        for(i = No_of_chars + No_of_rules - 1; i >= 0; i--){
            freq[i] = (cum_freq[i] - cum_freq[i+1] + 1)/2;
        }
        for (i = No_of_chars + No_of_rules - 1; i >= 0; i--){
            cum += freq[i];
            cum_freq[i] = cum;
        }
        antiunderflow++;
        /*
        printf("Max_frequency を超えた回数は %d です.\n", antiunderflow);
        */
    }


    if (symbol > 258){
        if (next_blank >= No_of_rules + 259){
            printf("あふれました (%d)\n", __LINE__);
            exit(1);
        }
    }

    if (symbol == 256){
        int k;
        for (k = next_blank; k >= 0; k--){
            cum_freq[k]++;
        }
        next_blank++;
    }
    for (symbol; symbol >= 0; symbol--){
        cum_freq[symbol]++;
    }
    return;
}

/* 符号化の MAIN PROGRAM */
main(int argc, char *argv[]){
    FILE *fp;
    int symbol;
    int prev_symbol = -1;

    start_model();
    start_outputing_bits();
    start_encoding();

    if (argc != 3){
        perror("引数の数が違います.\n");
        exit(1);
    }
    fp = fopen(argv[1], "rb");
    if (fp == NULL){
        perror("入力ファイルのオープンに失敗しました.\n");
        exit(1);
    }else{
        printf("入力ファイルを正常にオープンしました.\n");
    }
    cwfp = fopen(argv[2], "wb");
    if (fp == NULL){
        perror("出力ファイルのオープンに失敗しました.\n");
        exit(1);
    }else{
```

```
        printf("出力ファイルを正常にオープンしました.\n");
    }

    while (fread(&symbol, sizeof(int), 1, fp) == 1){        /* 文字のループ */
        if(prev_symbol == 258){
            encode_symbol((symbol == 257)? 0: 1, cum_eb);
            if (symbol != 257){
                encode_symbol(symbol, cum_freq);
            }
        } else {
            encode_symbol(symbol, cum_freq);       /* シンボルの符号化 */
        }
        update_model(symbol);           /* アップデートする */
        prev_symbol = symbol;
    }
    printf("Max_frequency を超えた回数は %d です.\n", antiunderflow);
    fclose(fp);
    printf("DEBUG %d\n", __LINE__);
    done_encoding();
    done_outputing_bits();

    fclose(cwfp);
    exit(0);
}
```

A.4 適応的算術符号の復号化

```
#include <stdio.h>
#include <stdlib.h>
#include "arithmetic_coding.h"
#include "model.h"

/* ビットバッファ */

static int buffer;
static int bits_to_go;
static int garbage_bits;
FILE *cwfp;
FILE *fp;


/* 先頭のビット入力 */

start_inputing_bits(){
    bits_to_go = 0;     /* バッファ中にビットがない状態でのバッファのスタート */
    garbage_bits = 0;
}

/* ビットを入力 */
int input_bit(){
    int t;
    if (bits_to_go==0){
        buffer = getc(cwfp);
        if (buffer == EOF){
            garbage_bits += 1;
            if (garbage_bits > Code_value_bits-2){
                fprintf(stderr, "Bad input file\n");
                exit(1);
            }
        }
        bits_to_go = 8;
    }
    t = buffer & 1;
```

```
    buffer >>= 1;
    bits_to_go -= 1;
    return t;
}

/* 算術復号化のアルゴリズム */

/* 復号化の流れ */

static code_value value;
static code_value low, high;

/* 復号化のスタート */
start_decoding()
{
    int i;
    value = 0;
    for(i = 1; i <= Code_value_bits; i++){     /* Code value がいっぱいになったらビットを入力 */
        value = 2*value+input_bit();
    }
    low = 0;                                   /* 最大コードの並び */
    high = Top_value;
}

/* 次のシンボルの復号化 */
int decode_symbol(unsigned long *cum)      /* シンボルの累計頻度 */
{
    long range;         /* 符号化の範囲の大きさ */
    int total;      /* 累計頻度の計算値 */
    int i;      /* シンボルの複合化 */
    range = (long)(high-low) + 1;
    total = (((long)(value-low) + 1)*cum[0] - 1) / range;      /* cum_freq の値を見つける */
    for(i = 1; cum[i] > total; i++);      /* シンボルを見つける */
    i--;
    high = low + (range*cum[i]) / cum[0]-1;      /* 割り当てられた符号の列を狭める */
    low = low + (range*cum[i+1]) / cum[0];
    for(;;){
        if(high < Half){
            /* Nothing */
        }
        else if(low >= Half){
            value -= Half;
            low -= Half;
            high -= Half;
        }
        else if(low >= First_qtr && high < Third_qtr){
            value -= First_qtr;
            low -= First_qtr;
            high -= First_qtr;
        }
        else break;

        low = 2*low;
        high = 2*high+1;
        value = 2*value+input_bit();
    }
    return(i);
}

/* 適応化算術符号のモデル */


/* モデルの先頭 */
void
start_model(){
    int i;

    for(i = 0; i < 259; i++){
cum_freq[i] = 259 - i;
```

```
    }
    for(i = 259; i < No_of_rules+259 ;i++){
        cum_freq[i] = 0;
    }

    next_blank = 259;
    return;
}


/* 新しいシンボルのためのモデルのアップデート */

int antiunderflow = 0;

void
update_model(int symbol)
{
    if (cum_freq[0] >= Max_frequency){                  /* 頻度が最高値を超えたときの処理 */
        int freq[No_of_chars + No_of_rules];
        int i;
        int cum = 0;

        for(i = No_of_chars + No_of_rules - 1; i >= 0; i--){
            freq[i] = (cum_freq[i] - cum_freq[i+1] + 1)/2;
        }
        for (i = No_of_chars + No_of_rules - 1; i >= 0; i--){
            cum += freq[i];
            cum_freq[i] = cum;
        }
        antiunderflow++;
        /*
        printf("Max_frequency を超えた回数は %d です.\n", antiunderflow);
        */
    }
    if (symbol > 258){
        if (next_blank >= No_of_rules + 259){
            printf("あふれました (%d)\n", __LINE__);
            exit(1);
        }
    }
    if (symbol == 256){
        int k;
        for (k = next_blank; k >= 0; k--){
            cum_freq[k]++;
        }
        next_blank++;
    }
    for (symbol; symbol >= 0; symbol--){
        cum_freq[symbol]++;
    }
    return;
}

/* 復号化のメインプログラム */
int
main(int argc, char *argv[]){

    int symbol = 257;  /* s0 を特別扱いするための初期値 */
    int a = 258; /* ルールのカウントをする変数 */
    int first_frag = 1;
    int end_frag;

    if (argc != 3){
        perror("引数の数が違います.\n");
        exit(1);
    }
    cwfp = fopen(argv[1], "rb");
    if (cwfp == NULL){
        perror("入力ファイルのオープンに失敗しました.\n");
```

```
        exit(1);
    }else{
        printf("入力ファイルを正常にオープンしました.\n");
    }
    fp = fopen(argv[2], "wb");
    if (fp == NULL){
        perror("出力ファイルのオープンに失敗しました.\n");
        exit(1);
    }else{
        printf("出力ファイルを正常にオープンしました.\n");
    }

    start_model();
    start_inputing_bits();
    start_decoding();

    for(;;){   /* 文字のループ */
        if(symbol == 257){
            for(;;){
                if(first_frag == 1){
                    first_frag = 0;
                }else{
                    fwrite(&symbol, sizeof(int), 1, fp);
                    if(symbol == 258) {
                        a++;
                        end_frag = 1;
                        break;
                    }
                    update_model(symbol);
                    /* printf("DEBUG %d\n", __LINE__); */
                }
                symbol = decode_symbol(cum_freq);
            }
        }else{
            fwrite(&symbol, sizeof(int), 1, fp);
            update_model(symbol);
            symbol = decode_symbol(cum_freq);
            fwrite(&symbol, sizeof(int), 1, fp);
            a++;
        }
        update_model(symbol);
        if (a >= next_blank) break;
        if (symbol == 258){
            if (decode_symbol(cum_eb) == 0){
                symbol = 257;
            } else {
                symbol = decode_symbol(cum_freq);
            }
            printf("DEBUG %d, %d\n", __LINE__, symbol);
        } else {
            symbol = decode_symbol(cum_freq);
        }
    }
    fclose(cwfp);
    fclose(fp);
    exit(0);
}
```

A.5 付録 C・D のヘッダファイル 1

```
/* 算術符号、復号のために用いる宣言 */


/* 算術符号の価値の大きさ */
#define Code_value_bits 16                              /* 符号のビット数 */
```

```
typedef long code_value;                                /* 算術符号の数の型 */

#define Top_value (((long)1<<Code_value_bits)-1)        /* 最大の符号 */


/* 符号の価値の列の中の 1/2 と 1/4 のポイント */

#define First_qtr (Top_value/4+1)                       /* 1/4 の後のポイント */
#define Half      (2*First_qtr)                         /* 1/2 の後のポイント */
#define Third_qtr (3*First_qtr)                         /* 3/4 の後のポイント */
```

A.6 付録 C・D のヘッダファイル 2

```
/* モデルのインターフェイス（交流）*/

/* 符号化されるシンボルのセット（位置）*/

#include <stdio.h>

#define No_of_chars 256              /* 文字シンボルの数 */
#define No_of_rules 100000          /* シンボルの数の合計 */


/* 文字とシンボルの見出しの間の表の翻訳 */

int sx_to_sX[No_of_rules];
int next_blank;

/* 累計された頻度の表 */
#define Max_frequency 16383             /* 回数を数える最大値 */
                                                /* 2^14-1 */
unsigned long cum_freq[No_of_chars + No_of_rules + 1];          /* 累積されたシンボルの回数 */
unsigned long cum_eb[] = {3, 2, 0};  /* e が出た後の b の確率 */
```

A.7 中間コードから元のファイルに変換するプログラム

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_VAR_NUM 10000

int *array[MAX_VAR_NUM];


/* 中間コードを変数の形に変換 */
void change_character(FILE*cwfp)
{
    int n = 257;
    int m;
    int i = 0;
    int j;
    int k = 3;

    do {
        if (i >= MAX_VAR_NUM){
            printf("MAX_VAR_NUM を増やしてください.\n");
            exit(1);
        }
```

```
        if(n == 257){   /* S の長さが 3 以上のとき */
            for(m = 1; fread(&n, sizeof(int), 1, cwfp), n != 258; m++){    /* Sの長さを測る */
                ;
            }
            array[i] = (int*)calloc(m, sizeof(int));  /* メモリの確保 */
            if(array[i] == NULL){
                perror("メモリの動的確保に失敗しました\n");
                exit(1);
            }
            fseek(cwfp, sizeof(int) * (-m), SEEK_CUR);   /* 長さを数えた分だけ n を巻き戻す */
            for (j = 0; m > 1; m--, j++){
                fread(&n, sizeof(int), 1, cwfp);
                if (n == 256){   /* n が s のとき s に添え字をつける */
                    n = n + k;
                    k++;
                }
                array[i][j] = n;
            }
            fread(&n, sizeof(int), 1, cwfp); /* e を読み飛ばす */
            array[i][j] = 256;
        } else {   /* S の長さが 2 のとき */
            array[i] = (int*)calloc(3, sizeof(int));  /* メモリの確保 */
            if(array[i] == NULL){
                perror("メモリの動的確保に失敗しました\n");
                exit(1);
            }
            if (n == 256){   /* n が s のとき s に添え字をつける */
                n = n + k;
                k++;
            }
            array[i][0] = n;
            fread(&n, sizeof(int), 1, cwfp);
            if (n == 256){   /* n が s のとき s に添え字をつける */
                n = n + k;
                k++;
            }
            array[i][1] = n;
            array[i][2] = 256;
        }
        i++;
    } while (fread(&n, sizeof(int), 1, cwfp) == 1);
    return;
}


/* 変数をデータ列に変換 */

void return_rule(FILE *fp, int n)
{
    int i;

    for (i = 0; array[n][i] != 256; i++){
        if (array[n][i] < 256){
            fputc(array[n][i], fp);
        } else {
            return_rule(fp, array[n][i] - 258);
        }
    }
    return;
}

void return_source(FILE *fp)
{
    return_rule(fp, 0);
    return;
}

int main(int argc, char *argv[])
{
```

```
    FILE *cwfp;
    FILE *fp;

    if (argc != 3){
        perror("引数の数が違います.\n");
        exit(1);
    }
    cwfp = fopen(argv[1], "rb");
    if (cwfp == NULL){
        perror("入力ファイルのオープンに失敗しました.\n");
        exit(1);
    }else{
        printf("入力ファイルを正常にオープンしました.\n");
    }
    fp = fopen(argv[2], "wb");
    if (cwfp == NULL){
        perror("出力ファイルのオープンに失敗しました.\n");
        exit(1);
    }else{
        printf("出力ファイルを正常にオープンしました.\n");
    }
    change_character(cwfp);
    return_source(fp);
    fclose(cwfp);
    fclose(fp);
    return 0;
}
```