解 説

基般

文法圧縮最前線

田部井 靖生（科学技術振興機構・さきがけ）

文法圧縮の概要

文法圧縮とは，入力文字列を小さい文法で表現することによる圧縮法である．文法圧縮では，文字列中の共通する部分列を同じ生成規則で表現する．よって，共通する部分列を多く含む文字列ほど，サイズの小さい文法を構築することができる．共通する部分列を多く含む文字列は，反復文字列と呼ばれ，実例としては同じ種族からなるゲノム配列の集合，バージョン管理されたテキスト，レポジトリ中でのソフトウェアなどがある．特に，ヒトゲノムの個人間での違いは 0.1% ほどといわれていている．現在のシーケンシング技術の発達によりゲノムを読むコストは減少していて，処理すべきゲノム配列はデータベース中に増加している．このため，文法圧縮はゲノムを処理する有力な手法であると期待されている．その他，転置索引，グラフ，フィンガープリントなど，文字列以外のデータでも文法圧縮の応用が提案されている．

文法圧縮の主な目的は，圧縮した状態で文字列を効率よく処理することである．通常，得られた文字列は圧縮され保持される．そして，文字列を処理する際，圧縮された文字列をいったんすべて解凍した後に処理される．処理すべき文字列が小さいときはこのような手法でもよいが，文字列が巨大になった際，ディスクスペースが大量消費され，その後の計算で無駄が生じる．文字列を圧縮して処理した方が，処理時間は圧縮サイズのみに依存した時間で処理できるため，速度的な有効性が期待できる．よって，圧縮された文字列は圧縮された状態で処理したほうが計算効率がよいといえる．そのため，圧縮された文字列上で達成したいタスクに応じたさまざまな操作が必要である．文法圧縮では，圧縮した状態でデータに対するさまざまな操作を行うことが可能である．**表 -1** に文法圧縮上の主な操作の一覧を示す．

文法圧縮には，大きく 2 種類の研究課題がある．1 つ目の問題は，入力文字列からどのようにして文法を構築するかという問題である．文字列から最小の文法を構築する問題は NP 困難問題であり，このため，これまでにさまざまな近似アルゴリズムが提案されている．2 つ目の問題は，構築した文法を符号化する問題である．タスクに依存して文法に対して必要となる操作は異なる．このため，これまでにさまざまな文法の符号化法が提案されている．本稿では代表的な文法圧縮アルゴリズムとその上の文字列検索法の概要を述べる．

タスク	説明
文字列検索	文字列中でのクエリ文字列の出現位置を求める
編集距離計算	文字列同士の編集距離を計算する
q グラム頻度計算	文字列中に存在するすべての q グラムの頻度を計算する
アクセス計算	S の任意の位置 i $(1 \leq i \leq \|S\|)$ での $S[i]$ を求める
rank 計算	位置 i $(1 \leq i \leq \|S\|)$ に対して，$S[1, i]$ 中の文字 a の出現回数を求める
select 計算	S 中で文字 a の i 番目の出現位置を求める
自己索引	ランダムアクセスと文字列検索可能な索引構造

表 -1　文法圧縮された文字列上での操作一覧

準　　備

Σをサイズσの有限アルファベット，すなわち，$|\Sigma|=\sigma$とする．Σ^*の要素を文字列とする．長さkのすべての文字列の集合をΣ^kとする．文字列$S\in\Sigma^*$の長さを$|S|$で記述し，文字列$S\in\Sigma^*$の長さをu，そして，文字列$Q\in\Sigma^*$の長さをmとする．Sのi番目の文字を$S[i]$により記述する．そして，i番目に始まりj番目で終わるSの部分文字列を$S[i, j]$と記述する$(1\leq i\leq j\leq u)$．文字列$S=XYZ$に対して，X，Y，ZをそれぞれSの接頭辞，部分文字列，接尾辞と呼ぶ．Sの長さqの接頭辞を$pre(S, q)$，すなわち，$pre(S, q)=S[1, q]$，Sの長さqの接尾辞を$suf(S, q)$，すなわち，$suf(S, q)=S[u-q+1, u]$と記述する．Sの長さqの部分文字列をqグラムという．

本稿では，iterated logarithmを用いる．logの底は2とする．iterated logarithmは再帰的に定義され，$\log^{(1)}u=\log u$, $\log^{(i+1)}u=\log\log^{(i)}u$であり，iterated logarithmは，$\log^* u=\min\{i; \log^{(i)}u\leq 1\}$と定義される．実際は，$u\leq 2^{65536}$のとき$\log^* u\leq 5$であるので，$\log^* u=O(1)$である．

文脈自由文法（context-free grammar（CFG））とは4つ組$G=(\Sigma, V, D, X_s)$である．Σは有限アルファベットであり，Σの要素は終端記号と呼ばれる．Vは変数集合，ただし，$V\cap\Sigma=\emptyset$を満たす，そして，Vの要素は非終端記号と呼ばれる．Dは$V\times(V\cup\Sigma)^*$の部分集合であり辞書と呼ばれる．そして，Dの要素は生成規則と呼ばれる．$X_s\in V$は文法の開始記号である．

(i) 入力文字列

$S=abcbacabcb$

(ii) Sを表現するSLP

$\Sigma=\{a, b, c\}$
$V=\{X_1, X_2, X_3, X_4, X_5, X_6\}$
$D=\{X_1\rightarrow ab$
$X_2\rightarrow X_1c,$
$X_3\rightarrow X_2b,$
$X_4\rightarrow ac,$
$X_5\rightarrow X_3X_4,$
$X_6\rightarrow X_5X_3\}$
$X_s=X_6$

(iii) 構文木

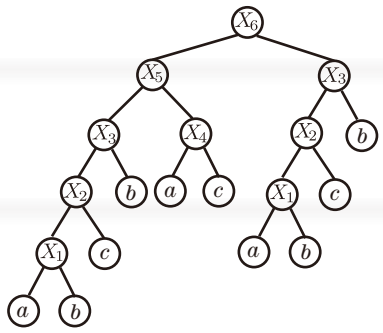


図-1　文法圧縮の例

文法圧縮とは

本章では，文法圧縮とその標準形である直線的プログラム（SLP）について述べる．文法圧縮とは，与えられた文字列Sのみを導出するCFGである．文法圧縮Gのサイズ$|G|$はGのすべての生成規則（非終端記号）の数である．文法圧縮アルゴリズムはその振舞いにより異なる文法を出力するが，以下の問題を解くという点においては同じである．

問題1（最小文法発見）

文字列Sが与えられたら，Sのみを表現するサイズ最小のCFGGを求める．

特に文法がチョムスキー標準形であるとき，すなわち，文法圧縮における各生成規則の右辺が2つの終端記号または非終端記号からなるとき，たとえば$A\rightarrow ab\in D$，CFGはSLPと呼ばれる．SLPは文法圧縮におけるさまざまな応用を展開する上での標準形である．

文法圧縮は，構文木としての等価な表現を持ち，構文木の各内部ノードはCFGGの非終端記号（すなわち，V要素）に対応し，葉ノードはGのアルファベットΣに対応する．各非終端記号$X_i\in V$は入力文字列Sの部分文字列を符号化し，構文木におけるX_iを根とする部分木の葉のラベル列に対応する．X_iが符号化する文字列を$val(X_i)$とする．文法がSLPのとき，SLPに対応する構文木は完全二分木となる（**図-1**）．文法圧縮の構文木としての表現は，文法圧縮を応用したアルゴリズムを設計する際の見通しのよい表現であり，簡潔データ構造などを用いることによりさまざまな符号化を行うことができる．どのような符号化を行うかは，目的のタス

クに依存する．

一般的には，構文木中での各レベルにおける各ノードの高さが一定であるとは限らない（すなわち，構文木はアンバランスである）が，構文木の各ノードの左右の子の高さがたかだか1違うだけ（すなわち，構文木はバランスしている）の方が扱いやすい場合が多い．文字列Sのn個の生成規則からなるSLP Gに対して，Gから$O(n \log u)$時間で$O(n \log u)$個の生成規則を持つGと等価なバランスされた構文木からなるSLP G'に変換するアルゴリズムが存在する．

問題1は，NP完全問題であることが知られている．よって，ほとんどのアルゴリズムは近似的に問題1を解く．次の章では，文法圧縮アルゴリズムの概要を述べる．

文法圧縮アルゴリズム

文法圧縮アルゴリズムには大きく分けてオフラインアルゴリズムとオンラインアルゴリズムがある．オフラインアルゴリズムは，入力文字列をディスクから一度メモリに読み込んだ後で文法を構築する．一般的に，オフラインアルゴリズムは圧縮率が高い．オンラインアルゴリズムは，入力文字列をディスクから1文字ずつ読みつつ文法を構築する．よって，オンラインアルゴリズムは，入力文字列をメモリに読む必要がないのでメモリ効率がよい．加えて，オンラインアルゴリズムはいったん文法を構築した後でも，後から追加された文字列に対して，文法を再構築することなく圧縮を行うことができるという利点がある．近年のデータ生成速度の高速化に伴い，アルゴリズムがオンライン型であることは，文法圧縮を大規模データに適応する際の利点であるといえる．アルゴリズムの評価尺度としては，近似率，時間，メモリがあり，これまで，多くの文法圧縮アルゴリズムが提案されている．アルゴリズムにはそれぞれ特徴があり，処理したい文字列の種類と状況に応じて，アルゴリズムを使い分けることが重要である．

初期の文法圧縮アルゴリズムは，貪欲法に基づくアルゴリズムであった．その後，RytterとCharikarらが，LZ77分解に基づく文法圧縮アルゴリズムを設計し，アルゴリズムの近似率を解析するための手法を開発したことが，その後の文法圧縮の発展に大きく貢献した．

文字列SのLZ77分解$LZ(S)$とは，Sの分解$f_1, f_2, ..., f_z$である．各f_ℓは，$\ell=1$に対しては$f_\ell=S[1]$，そして，$1<\ell\leq z$に対しては，$f_1 ... f_{\ell-1}$の部分列と一致する接尾辞$S[|f_1 ... f_{\ell-1}|+1, u]$の最長接頭辞である．各$f_\ell$はファクタと呼ばれる．$LZ(S)$のサイズ$|LZ(S)|$は，ファクタの個数$z$である．たとえば，$S=ababcabcab$に対する$LZ(S)$は，$f_1=a, f_2=b, f_3=ab, f_4=c, f_5=abc, f_6=ab$となる．

基本的なアイディアは，$LZ(S)$の各ファクタf_iに対して，バランスされた構文木を構築することである．$LZ(S)$の各ファクタf_iに対して，$\log u$個の非終端記号を用いて高さ$\log u$のバランスした構文木を構築できる．よって，全ファクタに対する文法のサイズと構築時間は，$O(z \log u)$となる．

文字列Sに対するLZ分解のサイズは，Sに対する任意の文法のサイズ以下となるという重要な事実が知られている．

定理1　文字列Sに対するLZ分解Sを$LZ(S)$とする．このとき，Sの任意の文法圧縮$G(S)$に対して，$z\leq|G(S)|$が成り立つ．

よって，Sに対する最小文法を$G_{opt}(S)$とするとき，文法のサイズは$O(G_{opt}(S)\log u)$で抑えることができ，近似率は$O(\log u)$となる．

これまでに$O(\log(u/G_{opt}(S)))$の近似率を達成するいくつかのアルゴリズムが提案されている．これらのアルゴリズムの基本的なアイディアは，文字列中に存在する部分文字列の文字ペアが，なるべく同じ非終端記号に置き換えられるように構文木の下から上へボトムアップに構文木を構築することである．次の節では，代表的なアルゴリズムであるRe-Pairの概要を述べる．

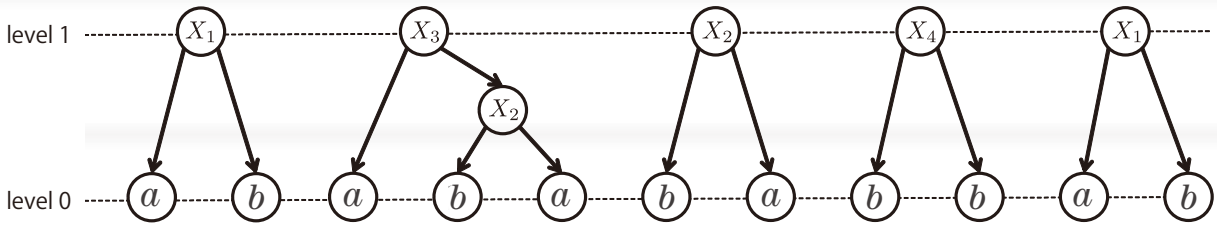


図 -2　ESP が文字列 $S^0=ababababbab$ から構文木を構築する例．ESP は，レベル 0 の文字列 S^0 から部分木を構築し，そして，レベル 1 の文字列 $S^1=X_1X_3X_2X_4X_1$ から部分木を段階的に構築する

■Re-Pair

Re-Pair[1] は，貪欲法に基づく文法圧縮アルゴリズムであり，アルゴリズムの各イテレーションで最頻出の記号ペアを非終端記号へ置き換える．Re-Pair の各イテレーションは次の 3 ステップからなる：(i) 現在の文字列 S で最頻出の隣接する記号ペア ab を発見する，(ii) 文字列 S 中に出現する最頻出の隣接する記号ペア ab を新しい非終端記号 A に置換する，(iii) 新しい生成規則 $A \to ab$ を辞書 D に追加する．文字列 S 中で隣接する文字ペアが 2 回以上出現しなくなるまでステップ (i) ～ (iii) を繰り返す．SLP を構築したい場合は，S の長さが 1 になるまでステップ (i) ～ (iii) を繰り返す．

図 -1 における入力文字列 S では，最頻出ペアは ab であり，S における ab は非終端記号 X_1 に置換され，生成規則 $X_1 \to ab$ が辞書 D に追加される．ステップ (ii) における記号ペア置換後の文字列は $S=X_1cbacX_1cb$ である．更新後の S 上でステップ (i) ～ (iii) は繰り返される．

文献 1）で述べられているように，Re-Pair は入力の文字列長に対して線形時間で動作するように実装することができる．線形時間で実装するためには，文字の更新を追跡するための入力文字列の各位置でポインタ，最頻出ペアを発見するための優先度付き順位キューなどのデータ構造が必要である．しかし，これらのデータ構造を利用した実装では大量のメモリを消費してしまう．改良点としては，ステップ (i) における最頻出の記号ペアを発見する代わりに頻出するトップ k 個の記号ペアを発見し，ステップ (ii) で文字列中に出現するこれら k 個の記号ペアを非終端記号で置換する．このとき，各文字ペアの出現を線形探索し文字列の更新を行う．このため，線形時間アルゴリズムではなくなるが，十分大きな k を設定すれば速度を犠牲にすることなく，省メモリで文法を構築することができることができる．

■Edit-sensitive parsing（ESP）に基づく文法圧縮アルゴリズム

FOLCA[2] は Edit-sensitve parsing（ESP）に基づくオンラインアルゴリズムである．ESP とは，文字列での同じ部分文字列の出現に対してノードのずれの上限を保証する構文木を構築するアルゴリズムである．ESP による構文木を ESP 木という．ESP が提案されたもともとの目的は，移動付き編集距離と呼ばれる文字列間の距離を近似的に計算することであった．これまでに，ESP は文法圧縮アルゴリズムに適応されている．

ESP は，完全 2 分木からなる ESP 木をレベルごとにボトムアップに構築する（**図 -2**）．ESP 木の各レベルでは 2 つのタイプの部分木が作られる．1 つ目のタイプは，$X \to AB$ の形の生成規則に対応する部分木であり，X に対応するノードと左右 2 つの子ノード AB からなる木（2 木）である．2 つ目のタイプは，2 つの生成規則 $X \to AY$ と $Y \to BC$ に対応する部分木であり，X に対応するノードと左右 2 つの子ノード AY，さらに Y に対するノードと左右 2 つの子ノード BC からなる部分木（2-2 木）である．

ESP の基本的なアイディアは，任意の 2 つの記号間の大小関係（たとえば，アルファベット順）を定義し，同じ部分文字列の異なる位置の出現に対してできるだけ多くの同じ部分木が構築されるように，この 2 つのタイプの部分木の選択を行うということ

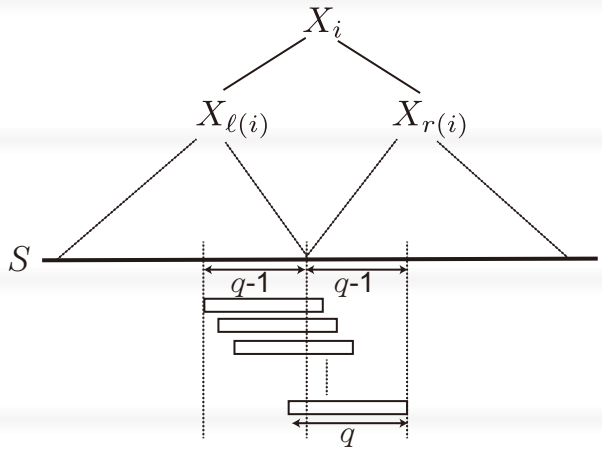


図 -3　X_i により串刺しされる q グラム $itv(X_i)$ の例

である．これにより，同じ部分文字列の異なる位置の出現に対して，ESP 木のノードのずれの上限を保証することができる．さらに，ESP が構築する文法圧縮の近似率は $O(\log u \log^* u)$ であることが示されている．ESP の計算時間は $O(u \log^* u)$ であり，構文木の高さは $O(\log u)$ となる．

FOLCA は ESP のオンライン文法圧縮への拡張であり入力文字列を読みつつ構文木を簡潔表現に符号化することができる．これによりオンラインかつ省メモリで文法圧縮を行うことができる．近年，FOLCA は定数領域で文法圧縮を行うアルゴリズムに拡張されている．FOLCA の改良版である Lossy/Freq-FOLCA[3] は定数領域で動作可能であり，100 人分のゲノム配列（約 300GB のテキストファイル）を 1 日で圧縮することができる．

文字列検索のための索引

文字列 S の自己索引とは，文字列 S のランダムアクセスとクエリ Q の出現する S 上での位置を検索可能なデータ構造である．文法圧縮上で索引構造を構築することの難しさは，文法圧縮では木のすべてのノードの非終端記号が全部分文字列を符号化しているわけではないところにある．よって，単純にクエリと非終端記号とのマッチングを行うだけでは，検索漏れが生じてしまう．文法圧縮を用いて q グラムの検索をするための手法の 1 つに，次に述べる文字列のカーネル化という手法がある．

S に対する SLP におけるある非終端記号 X_i に対して，構文木上で左のノードに対応する非終端記号を $X_{\ell(i)}$，右のノードの対応する非終端記号を $X_{r(i)}$ とする．すなわち，生成規則 $X_i \to X_{\ell(i)} X_{r(i)}$ に対応する．変数 X_i に対して，S の部分文字列 t_i を $val(X_{\ell(i)})$ の長さ $(q-1)$ の接頭辞と $val(X_{r(i)})$ の長さ $(q-1)$ の接尾辞の連結とする．すなわち，$t_i = suf(val(X_{\ell(i)}), q-1)pre(val(X_{r(i)})\ q-1)$．$X_i$ により表現される q グラムとは，t_i 中に存在するすべての q グラムの集合であり，そのような集合を $itv(X_i)$ とする（**図 -3**）．$itv(X_i)$ に含まれる q グラムは変数 X_i に串刺しされているかのように見えるので，$itv(X_i)$ の q グラムは変数 X_i に串刺しされているという．このとき，S 中に存在する q グラムに関して次の補題が成立する．

補題 1　S に対する SLP の変数集合 $\{X_i\}_{i=1}^n$ に対して，S の任意の q グラムは $\cup_{i=1}^n itv(X_i)$ に含まれる．

S のカーネルとは，$\cup_{i=1}^n itv(X_i)$ のすべての q グラムを連結した文字列である．カーネルは q グラムの検索に用いることができる．たとえば，カーネルを圧縮接尾木を用いて索引を構築することにより，任意の q グラムの S 中での存在をチェックすることができる．このときの検索時間は $O(q \log \sigma)$ であり，メモリは $O(nq(q-1) \log \sigma)$（σ はアルファベットサイズ）となる．

補題 1 は，文法圧縮をさまざまな応用に利用するための便利な道具であり，その他，q グラム頻度計算や次に述べる文字列検索に利用することができる．

■ SLP-index

SLP-index[4] は，補題 1 の串刺しの原理を用いた文字列 S の SLP 上での自己索引である．最終的には，クエリ検索は**図 -4** (v) の圧縮されたデータ構造上で実装されるが，説明の容易さのため文法の構文木上で説明を行う．SLP-index の基本的なアイディアは，クエリ Q の各位置 p で接頭辞 $Q[1, p]$

(i) 文法

$X_1 \to X_7X_8$	ab
$X_2 \to X_1X_9$	abc
$X_3 \to X_2X_8$	$abcb$
$X_4 \to X_7X_9$	ac
$X_5 \to X_3X_4$	$abcbac$
$X_6 \to X_5X_3$	$abcbacabcb$
$X_7 \to a$	a
$X_8 \to b$	b
$X_9 \to c$	c

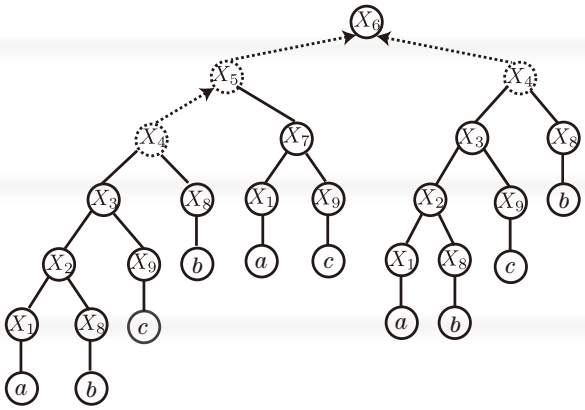
(ii) ソート

$X_7 \to a$	a
$X_1 \to X_7X_8$	ab
$X_2 \to X_1X_9$	abc
$X_3 \to X_2X_8$	$abcb$
$X_5 \to X_3X_4$	$abcbac$
$X_6 \to X_5X_3$	$abcbacabcb$
$X_4 \to X_7X_9$	ac
$X_8 \to b$	b
$X_9 \to c$	c

(iii) 変数名の付け替え

$X_1 \to a$	
$X_2 \to X_1X_8$	
$X_3 \to X_2X_9$	
$X_4 \to X_3X_8$	
$X_5 \to X_4X_7$	
$X_6 \to X_5X_4$	
$X_7 \to X_1X_9$	
$X_8 \to b$	
$X_9 \to c$	

(iv) クエリ cb を検索する例



(v) 2 次元グリッド符号化

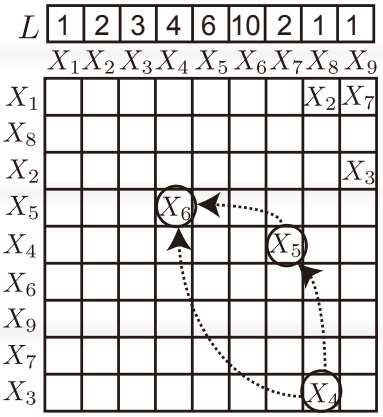


図 -4　SLP-index の構築とクエリ cb の検索例

と接尾辞 $Q[p+1, m]$ の 2 つの部分文字列に分け，$Q[1, p]$ は $val(X_{\ell(i)})$ の接尾辞，かつ，$Q[p+1, m]$ は $val(X_{r(i)})$ の接頭辞とする変数 X_i を求める．そのような X_i は，クエリ Q を串刺しする．次に，求めた X_i に対して，構文木上で X_i をラベルとして持つノードからルートへのパスを求める．求めたパスの個数は，文字列 S におけるクエリ Q の位置 p の分割における出現回数となる．同じ手続きを各 $p \in \{2, 3, \ldots, m-1\}$ に対して計算する．すべての p におけるパスの総和が Q の出現回数である．

図 -4 (iv) は，クエリ cb の検索例である．cb を串刺しする変数は X_4 であり，X_4 からルート X_6 へのパスは 2 つ存在する．よって，cb は S 上で 2 回出現する．

クエリの出現頻度だけでなく，文字列 S における出現位置を求めたいときは，補助データ構造として，各非終端記号が符号化する部分文字列の長さを配列 L に保持する．串刺しするノード X_i からルートに辿る際に，変数 $offset$ を用意し，$val(X_i)$ におけるクエリ Q の出現位置で初期化する．各ノード X_i で，ノード X_i が親ノード X_p の右の子，すなわち，$X_i = X_{r(p)}$ であるときのみ，$offset$ に左の子が符号化する部分文字列の長さ $L[X_{l(p)}]$ を足す．ルートに辿り着いたときの $offset$ の値が，クエリ Q の S における出現位置である．

たとえば，図 -4 (iv) における 2 つ目の cb の出現において，X_4 は cb を串刺しするので，$offset$ は cb の $val(X_4)$ における出現位置 $offset = 3$ で初期化される．次に X_4 は X_6 の右の子であるので，左の子 X_5 の長さ $L(X_5)$ を $offset$ に加える，すなわち $offset = offset + L(X_5) = 3 + 6 = 9$ となり cb の 2 番目の出現位置を求めることができる．

実際の検索は，2 次元グリッドに符号化された構文木上で行われる（図 -4 (v)）．文法は SLP であ

るが，説明しやすさのために，各終端記号に生成規則 $X_i \to \alpha(\alpha \in \Sigma)$ を追加する（図 -4 (i)）．はじめに，SLP の生成規則を各生成規則が符号化する文字列で辞書順にソートする（図 -4 (ii)）．次に，ソートされた順に変数名の名前の付け替えを行う（図 -4 (iii)）．最後に，各生成規則を 2 次元グリッドとして表現する（図 -4 (v)）．グリッド上で，各行は生成規則 $X_i \to X_{\ell(i)}X_{r(i)}$ における $X_{\ell(i)}$ であり，各列は $X_{r(i)}$ である．グリッドの要素は，行と列の変数を導出する非終端記号である．すなわち，生成規則 $X_i \to X_{\ell(i)}X_{r(i)}$ に対して，行が $X_{\ell(i)}$，列が $X_{r(i)}$ のとき，グリッドの要素は X_i である．2 次元グリッドは，wavelet 木を用いて，$2n \log n + o(n \log n)$ ビットで表現できる．

2 次元グリッド上で検索を行うときは，クエリ Q の各分割位置 p での接頭辞 $Q[1, p]$ と接尾辞 $Q[p+1, m]$ を，対応する 2 次元グリッドの列と行の区間を 2 分探索することに求める．行と列に対応する非終端記号はそれが符号化する文字列をもとに辞書順にソートされているので，$Q[1, p]$ と $Q[p+1, m]$ を含む区間を行と列でそれぞれ求めることができる．次に，見つかった非終端記号から根へのパスを 2 次元グリッド上で計算することにより，Q の分割 p での出現回数を計算することができる．

配列 L のサイズが，$n \log u$ ビットであるので，合計 $n \log \mathrm{u} + 2n \log n + o(n \log n)$ ビットとなる．occ をクエリ Q の S における出現回数，h を構文木の高さとするとき，SLP-index のサイズは $n \log u + O(n \log n)$ ビットであり，クエリの検索時間は $O(|m|^2 + h \log n(|m| + occ))$ である．

■その他の自己索引

ESP に基づく自己索引 ESP-index[5] が提案されている．ESP の構文木のずれの上限に関する保証を用いて索引を構築するので，クエリの出現を構文木上でトップダウンに検索することができる．よって，長いクエリに対して SLP-index よりも高速に検索することができる．

今後の展望

本稿では，文法圧縮とその上の操作を中心にその概要を述べた．文法圧縮は，文字列処理の分野で最も活発に研究されている分野の 1 つであり，その発展も目覚しい．一方で，群論や計算幾何学の応用も研究されている．より理論的な成果に興味のある読者は文献 6）が参考になる．

ビッグデータ時代では，データは多様かつ生成速度も早いことから，今後は動的更新をサポートする圧縮技術が研究の主流になると考えられる．動的更新とは，圧縮を再構築することなくデータの挿入，削除，追加をサポートすることである．本稿では，動的更新をサポートするデータ圧縮としてオンライン文法圧縮と自己索引を紹介した．文法圧縮は，反復文字列に対して有効な圧縮法であるが，データを圧縮した状態でさまざまな操作をサポートすることから，多様な実応用も期待されている．本稿をきっかけに，多くの読者がこの分野に興味を持っていただければ幸いである．

参考文献

1) Larsson, J. and Moffat, A. : Offline Dictionary-based Compression, In *Proceedings of DCC*, pp.296-305 (1999).
2) Maruyama, S., Tabei, Y., Sakamoto, H. and Sadakane, K. : Fully Online Grammar Compression, In *Proceedings of SPIRE*, pp.218-229 (2013).
3) Maruyama, S. and Tabei, Y. : Fully Online Grammar Compression in Constant Space, In *Proceedings of DCC*, pp.173-182 (2014).
4) Claude, F. and Navarro, G. : Self-indexed Grammar-based Compression, *Fundamental Informaticae*, 111:313-337 (2011).
5) Takabatake, Y., Tabei, Y. and Sakamoto. H. : Improved ESP-index : A Practical Self-index for Highly Repetitive Texts, In *Proceedings of SEA*, pp.338-350 (2014).
6) Lohrey, M. : Algorithmics on SLP-compressed Strings : A survey, *Groups Complexity Cryptology*, 4:241-299 (2012).

（2015 年 11 月 1 日受付）

田部井靖生　tabei.y.aa@m.titech.ac.jp

2009 年東京大学大学院新領域創成科学研究科情報生命科学専攻博士号（科学）取得．2010 年科学技術振興機構 ERATO 湊離散構造処理系プロジェクト研究員．2013 年現職．