解　説

計算論的学習理論の応用システム†

高　田　裕　志††

1. はじめに

計算論的学習理論は，機械学習の基礎理論を研究する分野である．従来，「帰納推論」や「文法推論」という名で研究が行われてきており，工学の分野では主に，プログラムの自動合成やパターン認識などでその応用システムが研究されてきた．

プログラムの自動合成では，Summers[16]やBiermannら[4]のLISPプログラムの自動合成システム，Shapiro[13]のPROLOGプログラムの自動合成システムMISが有名である．文献6)や7)には，構造的（構文的）パターン認識に関する数多くの応用システムが述べられている．帰納推論の解説論文[2]にも，理論的結果とともに数多くの応用システムが解説されている．また，「計算論的学習理論」という一つの分野にまとまる機会をもたらしたValiantの新しい学習モデルPAC-学習に関しては，決定木や決定リストへの応用[5),10)]ニューラルネットへの応用[3]が研究されている．

本稿では，これらの応用を概説するのではなく，今まであまり紹介されていない，プログラム自動合成以外のソフトウェア工学における応用システムを解説する．解説する応用システムは，自動化が望まれる問題を理論的に十分解析することによって，効率のよい解決法をとっている．ここで解説する例から，応用システム作成において理論的解析がどのように貢献しているかを理解していただきたい．特に，学習の効率は応用システム開発において最も重要な問題であり，この解説を通じて，理論的解析にもとづく効率化の手法を理解していただければ幸いである．

```
$
Author : Angluin, D.
Title  : Inductive Inference of Formal Languages from
         Positive Data
Journal : Information and Control 45
Year   : 1980
$
Author : Ibarra, O. H.
Title  : On Two-Way Multihead Automata
Journal : Journal of Computer System and Sciences
Year   : 1973
$
⋮
```

図-1　入力データの例

2. データエントリシステムへの応用

篠原[14),15)]は，効率よく帰納推論可能なパターン言語のサブクラスとそのアルゴリズムを解明した．そのアルゴリズムを用いて，学習機能をもつデータエントリシステムを開発した．

2.1 データエントリの問題

一般に，テキストには構造がない場合が多いが，ある種のテキストデータは構造をもっている．たとえば，図-1のテキストの場合，$ではさまれた部分が一つのレコードであり，各レコードには著者(Author)，タイトル(Title)，雑誌名(Journal)，発行年(Year)のフィールドがあり，これらはこのテキストの構造である．このような構造をもつテキストは，単にこの文字列を入力するのではなく，各レコードごと，さらにフィールドごとに入力するのが望ましい．たとえば，データエントリシステム側からの"Author :"や"Title :"などのプロンプトにしたがって，フィールドごとに入力するほうが，全テキストを単なる文字列として入力するよりも望ましい．しかし，このようなデータエントリの書式は，データベースごとに異なる．

そこで，篠原はデータエントリの書式を推論する問題に，パターン言語の帰納推論を応用した．篠原のデータエントリシステムは，入力された具体例からパターンを帰納推論することによって，

† An Introduction on Application Systems of Computational Learning Theory by Yuji TAKADA (FUJITSU LABORATORIES LTD., International Institute for Advanced Study of Social Information Science).

†† (株)富士通研究所国際情報社会科学研究所

データエントリの書式を学習する．学習された書式を用いて，データをより簡単に入力することができる．

2.2 正則パターンと帰納推論

集合 Σ を少なくとも二つの異なる記号を含むアルファベット，X を変数の集合とする．ただし，X と Σ とは共通部分をもたないとする．このとき，**パターン** P とは $\Sigma \cup X$ 上の記号列である．

変数 $x_1, x_2, \cdots, x_n$ をもつパターン P とそれらの変数に対する代入 $[w_1/x_1, w_2/x_2, \cdots, w_n/x_n]$ が与えられたとき，この代入をパターン P に適用した結果はアルファベット Σ 上の記号列である．ただし，各 w_i は Σ 上の空でない記号列である．たとえば，パターン $x_1ax_2ax_1$ に代入 $[bb/x_1, ccc/x_2]$ を適用した結果は，Σ 上の記号列 *bbacccabb* である．パターン P が定義する言語とは，P に任意の代入を適用して得られる Σ 上の記号列の集合である．

パターン言語の帰納推論に関しては，Angluin[1] がさまざまな結果を示した．中でも最も重要な結果は，パターン言語が正の具体例，つまり，その言語の要素だけから学習できることである．これは，正則言語でさえも正の具体例だけから学習できないことを考えると，驚くべき結果である．しかし，残念ながら，パターン言語のクラス全体を効率よく推論するアルゴリズムはないと考えられている．そこで，篠原は効率よく推論できる二つのサブクラスを発見した．特に，学習機能をもつデータエントリシステムでは，**正則パターン**言語のクラスが用いられている．

正則パターン R とは，各変数が高々一回しか現れないパターンである．したがって，パターン x_1ax_2 は正則パターンであるが，$x_1ax_2ax_1$ は正則パターンでない．正則パターン言語のクラスは，正の具体例だけから効率よく帰納推論することができる．

2.3 学習機能をもつデータエントリシステム

先にも述べたように，図-1 のような構造のあるテキストに対しては，その構造にもとづく書式を用いたデータエントリシステムが望まれる．篠原のデータエントリシステムは，この書式を帰納推論する．

たとえば，図-1 のようなデータが与えられると，システムはパターン

"Author : x_1 Title : x_2 Journal : x_3 Year : x_4"

を帰納推論する．この推論されたパターンを用いて，データエントリシステムはプロンプト "Author :", "Title :" などを出力し，ユーザはこれらにもとづいてデータを入力することができる．

このようなマンマシン・インタフェースには，効率のよさが要求されるが，篠原のシステムはこの要求に対して，十分効率がよい．

3. テキストエディタへの応用

Nix の事例による編集システム (*EBE* システム)[8),9)] も，パターン言語の帰納推論の重要な応用例である．Nix は，テキスト編集におけるテキスト変換の問題を，パターン言語の帰納推論によって解決した．

3.1 テキスト変換の問題

EBE システムで対象となるテキスト変換の問題とは，あるテキストを置換によって別のテキストに変換する問題である．

例として，**表-1** の電話番号表を**表-2** のような書式に変換することを考える．多くのテキストエディタでは，置換コマンドを繰り返すか，カーソル移動コマンドと削除・挿入コマンドを繰り返して変換するが，このような小さいテキストでも相当数コマンドを繰り返さなければ変換することができない．Emacs などの機能の充実したエディタでは，正規表現による置換やマクロコマンドによって，比較的少ない操作で変換することができる．たとえば，Emacs では，正規表現による置換コマンド

表-1 電話番号表 (1)

(03) 123-1234.
(06) 234-1234.
(011) 123-4123.
(045) 234-4123.
(0482) 12-3456.
(0559) 23-4567.
(01462) 3-4567.

表-2 電話番号表 (2)

03 (123) 1234.
06 (234) 1234.
011 (123) 4123.
045 (234) 4123.
0482 (12) 3456.
0559 (23) 4567.
01462 (3) 4567.

M-x replace-regexp RET (\([0-9]+\))␣\([0-9]+\)−**RET**\1␣(\2)␣ **RET**

によって一回の置換で変換できる．ここで，"(\([0-9]+\))␣\([0-9]+\)-" と "\1␣(\2)␣" が正規表現である．ただし，正規表現やマクロはユーザが定義しなければならないが，複雑な変換に対してはかなりの熟練を要する．さらに，新たな変換の対象の増加（付加）にともない，置換の対象や操作が変化した場合，正規表現やマクロを定義しなおさなければならない．

EBE システムでは，**ギャッププログラム**と呼ばれるプログラムによって変換が行われる．*EBE* システムは，ユーザによる変換の例から，それを一般化した変換を表すギャッププログラムを自動的に合成する．ユーザは，この合成されたプログラムを用いて，残りの部分やほかのテキストに対して変換を実行することができる．したがって，ユーザがギャッププログラムをプログラミングする必要はなく，変換の例をいくつかシステムに示してさえやれば，全テキストを望ましい書式に変換することができる．

上記の例では，ユーザが実際に置換

(03)␣123-1234. ⇒ 03␣(123)␣1234.

(06)␣234-1234. ⇒ 06␣(234)␣1234.

(011)␣123-4123. ⇒ 011␣(123)␣4123.

を行い，これを変換の具体例として *EBE* システムに入力してやる．システムは入力された具体例を解析し，ギャッププログラム

(-1-)␣-2-−-3-. ⇒ -1-␣(-2-)␣-3-.

を合成する．ここで，-1-，-2-，-3- は**ギャップ**と呼ばれる変数である．このプログラムは，矢印⇒の左辺のパターンがマッチした部分を，右辺のパターンによって置き換える．ユーザはこのプログラムを実行することによって，残り4つのエントリを望ましい書式に変換することができる．

3.2 ギャッププログラムとテキスト変換

ギャッププログラムは，**ギャップパターン** G と**ギャップ置換** R からなり，$G \Rightarrow R$ で定義される．各 s_i $(0 \leq i \leq n)$ をアルファベット Σ 上の長さ1以上の記号列，各 g_i $(1 \leq i \leq n)$ をギャップとすると，Σ 上の**ギャップパターン** G は，記号列とギャップの列 $s_0 g_1 s_1 g_2 s_2 \cdots g_n s_n$ と定義される*．ただし，各ギャップは Σ と共通部分をもたないギャップ用のアルファベットから選ばれるものとし，$i \neq j$ ならば，$g_i \neq g_j$ でなければならない．したがって，一つのギャップパターンに同じギャップが二度現れることはない．記号列 s_0 は $n \neq 0$ ならば空記号列でもよい．また，各 s_i をギャップパターンの**定数**という．

Σ 上の記号列 w に対して，$w = s_0 w_1 s_1 w_2 s_2 \cdots w_n s_n$ となる代入 $[w_1/g_1, w_2/g_2, \cdots, w_n/g_n]$ が存在するとき，ギャップパターン $s_0 g_1 s_1 g_2 s_2 \cdots g_n s_n$ は w に**マッチする**という．ギャッププログラムのパターンマッチは，決定的に行われる．したがって，一つの記号列に対して，パターンマッチで得られる代入は高々一つである．

アルファベット Σ とギャップ $g_1, g_2, \cdots, g_n$ 上のギャップパターンに対して，**ギャップ置換** R はアルファベット $\Sigma \cup \{g_1, g_2, \cdots, g_n\}$ 上の任意の記号列と定義される．ギャップ置換 R に対して，代入 $[w_1/g_1, w_2/g_2, \cdots, w_n/g_n]$ を適用すると Σ 上の記号列が得られる．

ギャッププログラム $G \Rightarrow R$ によるテキスト変換は次の手順で行われる；(1)テキスト中から，ギャップパターン G にマッチする記号列 w を見つけ，代入 $[w_1/g_1, w_2/g_2, \cdots, w_n/g_n]$ を得る．(2)この代入をギャップ置換 R に適用し，記号列 w' を得る．(3)記号列 w を w' で置き換える．たとえば，前節の例では，ギャッププログラム

(-1-)␣-2-−-3-. ⇒ -1-␣(-2-)␣-3-.

を表-1 の4番目以降のエントリに対して実行すると，まずギャップパターンがエントリ "(045)␣234-4123." にマッチし，代入 [045/-1-, 234/-2-, 4123/-3-] が得られ，この代入をギャップ置換に適用した "045␣(234)␣4123." によって元のエントリが置き換えられる．残りのエントリも同様の手順で変換される．

3.3 具体例からのギャッププログラムの合成

EBE システムは，テキスト変換の具体例の集合からギャッププログラムを**帰納推論**によって自動合成する．システムに与えられる**具体例**は，変換前の記号列 w と変換後の記号列 w' の対 (w, w') である．与えられる具体例は，正の例だけである．つまり，ユーザが意図する変換である対だけを *EBE* システムに与える．*EBE* システムは与えられた具体例の集合 T からギャッププログ

* 最後が Σ 上の記号列であることに注意．したがって，前節の例にギャッププログラムを適用するには，各エントリの最後がピリオド(.)で終了する必要がある．

ラム $G \Rightarrow R$ を帰納推論する．

具体例からギャッププログラムを合成する問題は，計算量の点で非常に難しい問題である．そこで，Nix は問題を二つの部分問題に分割した．一つは，変換前の記号列の集合からギャップパターンを合成する問題で，もう一つは，変換後の記号列の集合と最初の問題で得られた代入からギャップ置換を合成する問題である．

3.3.1 ギャップパターンの合成

与えられた記号列の集合から望ましいギャップパターンを求める問題は，NP-困難な問題である．そこで，*EBE* システムは，以下の二つの発見的方法を用いて，ギャップパターンを合成する．

定数の発見　まず，具体例からギャップパターンの定数を求める．*EBE* システムは，二つの記号列に共通する最も長い部分記号列を求めるアルゴリズムを用いて，ギャップパターンの定数を求める．

n 個の記号列 $w_1, w_2, \cdots, w_n$ の最長共通部分を求める問題は NP-困難な問題であるが，二つの記号列の最長共通部分を求める問題は多項式時間で解ける．そこで，*EBE* システムは二つの記号列の最長共通部分を求めるアルゴリズムを繰り返し適用することによって，ギャップパターンの定数を近似する．たとえば，具体例 "(03)␣123-1234.", "(06)␣234-1234.", "(011)␣123-4123." が与えられたとき，システムは初めの二つの具体例から最長共通部分列 "(0", ")␣", "-1234." を求め，これと三つめの具体例から定数 "(0", ")␣", "-", "." を求める．

ギャップの挿入　定数を見つけたあと，ギャップを挿入し，ギャップパターンを作る．Nix は，ギャップを正しく挿入する問題も NP-困難な問題と予測している．そこで，*EBE* システムは，与えられた具体例にマッチするように必要な箇所すべてにギャップを挿入する．先の例では，三つの具体例とも三つめの記号が異なるので，ギャップ -1- を定数 "(0" と ")␣" の間に挿入する．さらに，定数 ")␣" と "-" にはさまれた記号列も三つの具体例すべてにおいて異なるので，ギャップ -2- を挿入する．同様の理由から，定数 "-" と "." の間にもギャップ -3- を挿入して，ギャップパターン "(0 -1-)␣-2--3-." が得られる．

このギャップパターン合成法は，与えられた具体例の集合から，ギャッププログラム合成に必要なギャップパターンを同定する．

3.3.2 ギャップ置換の合成

ギャップ置換の合成の場合でも，ギャップパターンの合成で得られた代入と具体例から単純にギャップ置換を合成する問題は，計算効率の点で手に負えない問題である．そこで，*EBE* システムでは次のようにギャップ置換を合成する：(1) 各具体例 (w, w') に対して，パターンマッチによって得られる代入を用いて記号列 w を w' に置き換えるすべての可能な置換を表現する有限状態オートマトンを構成する，(2) 構成されたすべてのオートマトンの共通部分を求める．この手続きによって得られたオートマトンの初期状態から最終状態までの経路を求めることによって，求めるギャップ置換が得られる．

たとえば，二つの具体例

(03)␣123-1234. ⇒ 03␣(123)␣1234.

(06)␣234-1234. ⇒ 06␣(234)␣1234.

が与えられたとき，変換前の二つの記号列にマッチするギャップパターンは "(0 -1-)␣-2--1234." である．このとき，それぞれの具体例に対して代入 [3/-1-, 123/-2-] と [6/-1-, 234/-2-] が得られる．この代入を用いて，**図-2** の二つの有限状態オートマトン(a)と(b)をそれぞれ構成し，その共通部分を求める．図-2 の有限状態オートマトン(c)が，(a)と(b)の共通部分を表す．これ

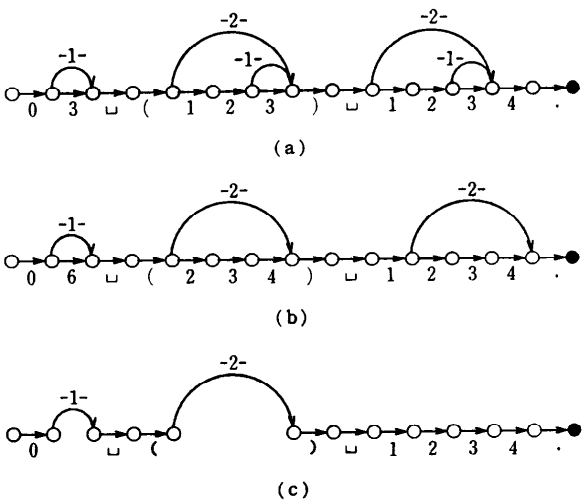


図-2 ギャップ置換の合成

```
%root
 <EXPRESSION>

%lexicon
 <IDENTIFIER>  ::=  [A-Za-z][A-Za-z0-9]*
 <CONSTANT>    ::=  [1-9][0-9]+

%grammar
 <EXPRESSION>        ::=  <EXPRESSION> * <EXPRESSION> |
                          <PARENTHESIS> |
                          <SUBEXPRESSION>
 <SUBEXPRESSION>     ::=  <CONSTANT> |
                          <IDENTIFIER>
 <PARENTHESIS>       ::=  ( <PLUSEXPRESSION> )
 <PLUSEXPRESSION>    ::=  <EXPRESSION> |
                          <PLUSEXPRESSION> + <PLUSEXPRESSION>
```

図-3　編集しやすい構造をもつ文法

より，ギャップ置換 “0-1-␣(-2-)␣1234.” が得られる．

このギャップ置換合成法は，理論的には効率よくないが，実用的には効率がよく，たいていの場合，具体例の長さと個数の多項式時間で，ギャップ置換を合成する．

先に述べたギャップパターンの合成法とここで述べたギャップ置換の合成法を用いることによって，ギャッププログラムを具体例から効率よく合成することができる．

実際の *EBE* システムでは，実用性を考慮して，ギャッププログラム合成のときに，トークン化*，パターン簡約，ギャップ制限などの発見的方法を用いる．これらの発見的方法を用いることで，より “もっともらしい” ギャッププログラムを合成する．

ギャップパターンで定義できる言語のクラスが正則言語の真部分クラスでしかないので，*EBE* システムで可能なテキスト変換は限られたものでしかない．しかし，パターン言語の帰納推論に関する最近の結果を用いることによって，より実用的なシステムにグレードアップすることが可能であろう．

4. 構造エディタへの応用

榊原[11),12)]は，構造例から文脈自由文法を効率よく学習するアルゴリズムを解明した．このアルゴリズムを用いて，筆者らは学習機能をもつ文法指導型構造エディタを開発し，編集構造のカスタマイズの問題を解決した．

4.1 編集構造のカスタマイズの問題

構造エディタ，特に構文指導型の構造エディタでは，プログラムは木構造で表現され，その木構造に従って編集が行われる．この木構造はプログラミング言語の文法によって決められ，構文的に正しい編集のみが許される．テキスト・エディタと比較して，構造エディタは次のような利点をもつ：(1)編集されたプログラムには構文エラーがない，(2)編集が構文にそってガイドされる．したがって，構造エディタを使用することで，プログラミングの生産性を向上させることが期待できる．しかし，これらの利点も，編集構造がユーザにとって「編集しやすい」ときのみ利点であり，そうでなければ大きな欠点となってしまう．従来の構造エディタでは，この「編集しやすい構造」がエディタ設計者によって決められてしまっていた．しかし，「編集しやすさ」はユーザの主観の問題であり，ユーザごとに異なる可能性がある．

たとえば，**図-3** と**図-4** は，ともに簡単な算術式を定義する文法である．これらの文法によると，式 “a*(b+c)” の構造は，それぞれ**図-5** の(a)，(b)のような木構造になる．ここで，各ノードを根とする部分木が編集の対象になる．図-5 の(b)では，“*(b+c)” や “+c)” などのユーザにとって意味のないテキストが編集対象になってしまうが，(a)では，どの部分をとってもユーザにとって意味のある編集構造となってい

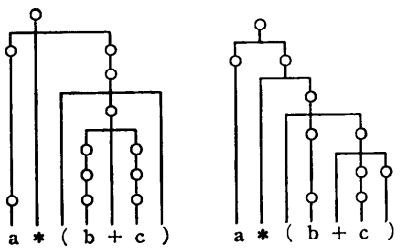
* この節で用いた例では，*EBE* システムは数字の列をトークンとして扱う．したがって，合成されたギャッププログラムでは，すべての具体例において2番目の文字が0であるにもかかわらず，ギャップパターン “(-1-)-2-.-3-.” を合成する．

```
%root
 <EXPRESSION>

%lexicon
 <IDENTIFIER>   ::=  [A-Za-z][A-Za-z0-9]*
 <CONSTANT>     ::=  [1-9][0-9]+

%grammar
 <EXPRESSION>       ::=  <SUBEXPRESSION> |
                         <SUBEXPRESSION> <REPEATTIMES>
 <SUBEXPRESSION>    ::=  ( <EXPRESSION> <REPEATPLUS> |
                         <IDENTIFIER> |
                         <CONSTANT>
 <REPEATTIMES>      ::=  * <SUBEXPRESSION> <REPEATTIMES> |
                         * <SUBEXPRESSION>
 <REPEATPLUS>       ::=  + <EXPRESSION> <REPEATPLUS> |
                         )
```

図-4 編集しにくい構造をもつ文法



(a) 望ましい編集構造 (b) 望ましくない編集構造

図-5 式 "a*(b+c)" の編集構造

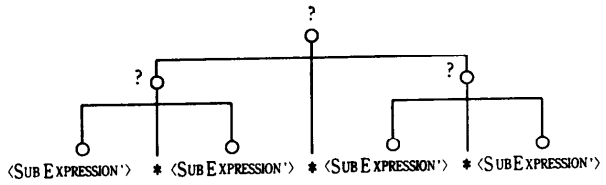


図-6 具体例

る．(b)のような編集構造で編集しなければならないとすると，ユーザは編集しづらいと感じるであろう．

エディタによって指定される構造が編集しづらい場合には，構造が編集しやすいものになるようにエディタをカスタマイズすることが望まれる．従来の構造エディタでは，エディタが使用している文法を修正することによってカスタマイズが行われていた．しかし，文法を修正するには文法に関する専門的な知識を必要とするので，一般のユーザには構造エディタのカスタマイズは非常に難しい．そこで，筆者らは，具体例と質問・応答から任意の文脈自由文法を学習する二つの文法学習アルゴリズムを基礎に，文法学習機能を構造エディタに組み込み，具体例と質問・応答から新しい文法を学習し，エディタをカスタマイズする方式を開発した．

4.2 文法学習によるカスタマイズ

カスタマイズは次の三つのセッションからなる：(1)具体例の作成，(2)学習機能の起動，(3)学習された文法ともとの文法のマージ．この3セッションを繰り返すことにより，ユーザの意図する構造で編集することが可能となる．

(1) 具体例の作成 エディタのもつ文法とは異なる木構造をもつプログラムが，学習機能に与える**具体例**となる．具体例は，エディタによる構造の制約を受けることなく作成され，内部ノードにラベルをもたない．ユーザにとって意図する構造をもつものを**正の**具体例，そうでないものを**負の**具体例という．たとえば，**図-6** は，図-4 の文法を図-3 にカスタマイズするために作成された正の具体例である．

(2) 学習機能の起動 作成された具体例を入力として学習機能を起動する．学習機能は以下の二つの学習アルゴリズムを用いるが，それらの正当性と効率は理論的に保証されている．この機能により，任意の文脈自由文法*を学習させることができる．

1.) 正の具体例によるカスタマイズ

第一の学習アルゴリズムは，正の具体例だけから学習する．一般に，ユーザにとっては正の具体例は想起しやすいので，正の具体例だけから学習できるというのは，カスタマイズにとって望ましい実用的な特徴である．また，このアルゴリズムは非常に高速である．しかし，学習可能な文法に制限がある，望まれる文法にカスタマイズするのに比較的多くの具体例を必要とする，などの欠点がある．

* BNF 記法で定義できる文法はすべて文脈自由文法である．したがって，C, PASCAL などのプログラミング言語の構文は文脈自由文法で定義できる．

2.）質問・応答によるカスタマイズ

第二の学習アルゴリズムは，具体例と質問・応答から学習する．アルゴリズムは木構造をユーザに見せ，それが正の具体例であるか負の具体例であるかをユーザに質問する．質問に対し，ユーザは「はい」または「いいえ」で答える．質問・応答の回数は，与えられた具体例の大きさによって決まる．このアルゴリズムでは，任意の文脈自由文法を学習することができる．また，比較的少数の具体例でカスタマイズすることができる．

（3） 文法のマージ 学習終了後，学習された文法とエディタがもっていた文法をマージする．学習された文法では，非終端記号* はすべて新しいものであるが，このマージの過程で，極力，もとの記号に対応させる．この対応づけは，生成規則の右辺の形の類似性にもとづいて行われる．たとえば，図-6 の具体例から学習機能は三つの生成規則

〈EXPRESSION'〉→〈SUBEXPRESSION'〉
　　＊〈SUBEXPRESSION'〉
〈EXPRESSION'〉→〈ACE0〉＊〈ACE0〉
〈ACE0〉→〈SUBEXPRESSION'〉
　　＊〈SUBEXPRESSION'〉

を生成するが，始めの生成規則と最後の生成規則は右辺が同じであるので，非終端記号〈ACE0〉は〈EXPRESSION'〉に対応づけられる．

このカスタマイズ法は，ユーザが正しく具体例を与え，正しく質問に答えるかぎり，正しく文法をカスタマイズする．したがって，その場合においては，カスタマイズされた後も，編集されたプログラムには構文エラーがない．

4.3 文法学習機能をもつ構造エディタ ACE

このカスタマイズの方法を用いて，学習機能をもつ構文指導型構造エディタ **ACE** を GNU Emacs 上で開発した[17),18)]．**ACE** の使用例を**図-7** に示す**．

ACE では，学習機能により具体例と質問・応答を通じて，編集構造をカスタマイズできる．ユーザが望む文法になるまでに，学習を数回繰り返す必要があるが，直接文法を修正する必要がないので，文法に関する専門的な知識がない初心者でもカスタマイズすることができる．

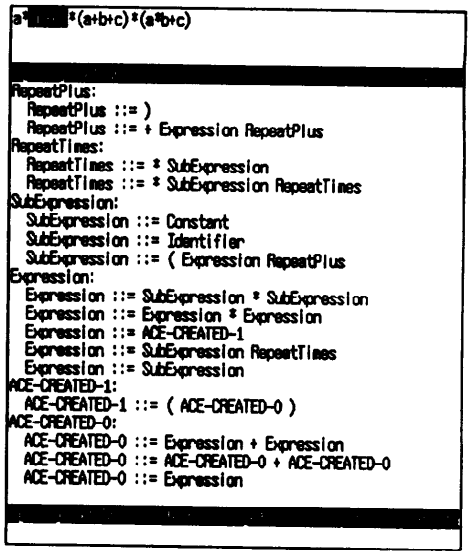


図-7 使用例（カスタマイズ終了後の画面）

また，**ACE** を，新しいプログラミング言語を設計する際の設計支援ツールとして用いることもできる．コンパイラ設計者などの文法設計の専門家にとっても文法を正しく定義するのは難しいが，**ACE** を用いることにより，具体例と質問・応答からインタラクティブに文法を設計することができる．プリティプリントのコマンドを文法に含めることで，プリティプリンタ設計にも使用することができる．

5. おわりに

計算論的学習理論は，機械学習の問題における学習可能性や学習の複雑さ，効率を理論的に解明しようとする研究である．特に，計算時間や必要な具体例の個数に関する効率の良さは，学習アルゴリズムを実際のシステムに応用するには欠かせない性質である．ここで解説した応用例でも，計算時間に関する学習効率を良くする目的で，問題を部分問題に制限したり，補助的な情報を用いたりしている．たとえば，パターン言語の場合，パターン言語全体のクラスは効率よく学習できないとされている．そこで，篠原のデータエントリシステムや Nix の *EBE* システムは効率よく学習可能な部分クラスを用いて問題を解いている．また，筆者らの **ACE** システムでも，文脈自由文法

---

* 木構造のノードのラベルにあたる．編集構造の名前になり，編集をガイドする役割をもつ．

** カスタマイズ終了後の画面．反転表示されたテキスト部分が現在の編集の対象．2番目のウィンドウの内容は現在の編集に用いられている文法．学習された文法である．

全体のクラスに対する効率よい学習アルゴリズムは解明されていないので，文法の構造情報をユーザに提供してもらうことによって，学習の効率をよくしている．この効率化の際に，理論的解析が非常に有効である．問題の複雑さを理論的に解析することで，発見的方法や補助情報の導入などの効率化を計る本質的なポイント，方法が明確になる．たとえば，*EBE* システムでは，ギャッププログラムの合成問題を詳細に解析することによって，発見的方法の導入のポイントと方法を明確にしている．

しかし，半面，理論的に効率よく学習できる対象というのはそれほど多くはない．また，たとえ問題を複雑にしているポイントが解明できたとしても，効率をあげるための発見的方法が簡単には見つからない場合も多い．

計算論的学習理論の研究は，ここ数年世界的に盛んになりつつある．とくに，PAC 学習では，かなりの基礎的研究が蓄積されてきており，ニューラルネットなどでの応用が期待されている．今後，これらの成果をもとに，より実用的な応用システムが開発されてくることがおおいに期待される．

参考文献

1) Angluin, D.: Finding Patterns Common to a Set of Strings, *Journal of Computer and System Sciences*, 21, pp. 46-62 (1980).
2) Angluin, D. and Smith, C. H.: Inductive Inference: Theory and Methods, *ACM Computing Surveys*, 15(3), pp. 237-269 (1983).
3) Baum, E. B. and Haussler, D.: What Size Net Gives Valid Generalization, *Neural Computation*, 1, pp. 151-160 (1988).
4) Biermann, A. W.: The Inference of Regular Lisp Programs from Examples, *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-8, pp. 585-600 (1978).
5) Ehrenfeucht, A. and Haussler, D.: Learning Decision Trees from Random Examples, *Information and Computation*, 82, pp. 231-246 (1989).
6) Fu, K. S. and Booth, T. L.: Grammatical Inference: Introduction and Survey, part 1 and 2, *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-5, pp. 95-111, 409-423 (1975).
7) Gonzalez, R. C. and Thomason, M. G.: *Syntactic Pattern Recognition: An Introduction*, Addison-Wesley, Reading, Mass. (1978).
8) Nix, R. P.: ***Editing by Example***, PhD thesis, Yale University Computer Science Dept. (1983).
9) Nix, R. P.: Editing by Example, *ACM Transactions on Programming Languages and Systems*, 7(4), pp. 600-621 (1985).
10) Rivest, R. L.: Learning Decision Lists, *Machine Learning*, 2, pp. 229-246 (1987).
11) Sakakibara, Y.: An Efficient Learning of Context-Free Grammars for Bottom-Up Parsers. In *Proceedings of FGCS '88*, pp. 447-454 (1988). To appear in *Information and Computation*.
12) Sakakibara, Y.: Learning Context-Free Grammars from Structural Data in Polynomial Time. In *Proceedings of 1st Workshop on Computational Learning Theory*, pp. 296-310 (1988). To appear in *Theoretical Computer Science*.
13) Shapiro, E. Y.: *Algorithmic Program Debugging*, PhD thesis, Yale University Computer Science Dept. 1982. Published by MIT Press (1983).
14) Shinohara, T.: Polynomial Time Inference of Pattern Languages and Its Applications. In *Proceedings of The Seventh IBM Symposium on Mathematical Foundations of Computer Science*, pp. 193-209 (1982).
15) Shinohara, T.: *Studies on Inductive Inference from Positive Data*, PhD thesis, Kyushu University (1986).
16) Summers, P. D.: A Methodology for Lisp Program Construction from Examples, *Journal of the ACM*, 24(1), pp. 161-175 (1977).
17) Takada, Y., Sakakibara, Y. and Ohtani, T.: **ACE**: A Syntax-Directed Editor Customizable from Examples and Queries, Research report, IIAS-SIS, FUJITSU LIMITED (1991).
18) 大谷 武，榊原康文，高田裕志：学習機能をもつ構造エディタ **ACE**，日本ソフトウェア科学会第7回大会論文集，pp. 29-32 (1990).

(平成2年10月17日受付)

高田 裕志（正会員）

昭和34年生．昭和58年北海道大学文学部行動科学科卒業．昭和60年同大学院文学研究科行動科学専攻修士課程修了．同年富士通(株)（現在，(株)富士通研究所）国際情報社会科学研究所入所，現在に至る．計算論的学習理論，形式言語理論などの研究に従事．EATCS, LA 学会，日本認知科学会各会員．