ICOT Technical Memorandum: TM-0634

TM-0634

自動カスタマイズ構造エディタについて

高田裕志，榊原康文(富士通)

December, 1988

ICOT

Mita Kokusai Bldg. 21F
4 28 Mita 1 Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

自動カスタマイズ構造エディタについて *

高田裕志　　榊原康文
Yuji Takada　Yasubumi Sakakibara

富士通（株）国際情報社会科学研究所
〒 410-03 静岡県沼津市宮本 140 富士フォーラム
E-mail: {*yuji,yasu*}*@iias.fujitsu.junet*

概要

従来の構造エディタでの構造とは、プログラム言語の構文によって決まる構造を意味した。しかし、言語の構文によって決まる構造と、プログラマ自身が想定する構造は必ずしも一致しない。そして、その場合、プログラマにとって構造エディタは使いにくいものになってしまう。そこで、我々は入力テキストにもとづいて、プログラム言語の文法を、プログラマが想定する構造を持つ文法に自動的にカスタマイズし、カスタマイズされた文法によって決まる構造にもとづいて編集することが可能な構造エディタ ACE を提案する。ACE は、編集作業を通じてカスタマイズされた文法をもとに、プリティプリント・リードを行い、編集単位を定める。

1　はじめに

プログラムの構造は、形式的には、それを記述するプログラム言語の構文と意味によって決まる。従来の構造エディタでの構造とは、プログラム言語の構文によって決まる構造を意味した。しかし、言語の構文によって決まる構造と、プログラマ自身が想定する構造は必ずしも一致しない。なぜならば、一般に同じ言語を定義する構文はいくらでもあり、プログラマは異なる構文を想定することが可能だからである。

たとえば、今、C 言語で書かれた自然数の階乗を計算する関数のプログラム（図 -1）を考える。一般に、C 言語の構文によれば、関数定義は次のようなバッカス記法による文法で定義される（この文法を G_1 とする）：

*本研究は第五世代コンピュータプロジェクトの一環として行ったものである。

```
<関数定義>     :=  <関数宣言子> <関数本体>
<関数宣言子>   :=  <宣言子> ( <引数リスト> )
<関数本体>     :=  <型宣言並び> <関数文>
<関数文>       :=  { <宣言並び> <文の並び> }
                   ⋮
```

しかし、文法

```
<関数定義>     :=  <関数宣言子> <型宣言並び> <関数文>
<関数宣言子>   :=  <宣言子> ( <引数リスト> )
<関数文>       :=  { <本体> }
<本体>         :=  <宣言並び> <文の並び>
                   ⋮
```

で定義することもできる（これを G_2 とする）。これら2つの文法が定義する「記号列」は同じであるが、構造は異なる。図 1のプログラムは、文法 G_1 によると図 -2のような木構造、つまり、構文解析木を持つが、文法 G_2 によると図 -3のような木構造を持つ。そこで、プログラマは、自分が想定する文法によって決まる木構造にもとづいて、プリティプリントしながらプログラムを書いていると仮定する。つまり、その木構造にもとづいて、改行やタブなどを挿入してプログラムを書くものとする。ここでは、部分木全体を段下げ (indentation) すると仮定する。図 -1のプログラムでは、2つのかっこ “{” と “}” で囲まれた宣言並びと文の並びが段下げされている。文法 G_1 による木構造では、段下げされているテキスト全体に対応する1つの部分木はないが、G_2 による木構造では <本体> を根とする部分木が対応する。したがって、C の関数定義を図 -1のように書くプログラマが想定する構造は、G_1 よりもむしろ G_2 によって決まる。

プログラム言語の構文とプログラマが想定している構文が異なる場合、そのプログラマにとって構造エディタは使いにくいものになってしまう。たとえば、編集の単位が構文解析木の部分木である構造エディタで、2つのかっこ “{” と “}” で囲まれた宣言並びと文の並びの部分を削除することを考える。文法 G_1 による構文解析木では、削除コマンドを2度用いなければならないが、G_2 の場合には1度でよい。したがって、その部分を1つの編集単位と見なしているプログラマには、G_1 にもとづく構造エディタは使いにくいものになってしまう。その他にも、プリティプリント・リードの段下げが気に入らない、テンプレートが気に入らない、といった使いにくさも生じる。

そこで、我々は、入力テキストにもとづいて、プログラム言語の文法を、プログラマが想定する構造を持つ文法に自動的にカスタマイズし、カスタマイズされた文法によって決まる構造にもとづいて編集することが可能な構造エディタ ACE(Auto-Customizing syntax-directed Editor) を提案する。ACE は編集作業を通じてカスタマイズを行う。したがって、編集作業が進めば進むほど、エディタの持つ構造はプログラマのものに近くなる。カスタマイズされた文法をもとに、プリティプリント・リードを行い、編集単位を定める。

```
fact(n)
int n;
{
    int i;

    i=n--;
    while(n>0)
        i=i*n--;
    return(i);
}
```

図 1: 自然数の階乗を計算する C プログラム

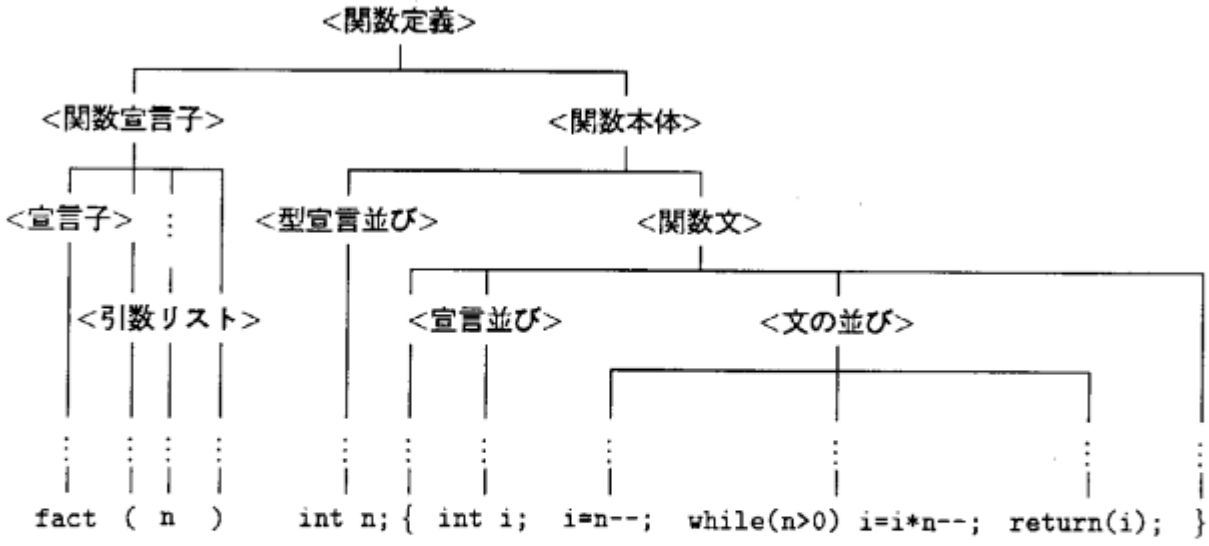

図 2: 文法G_1による構造

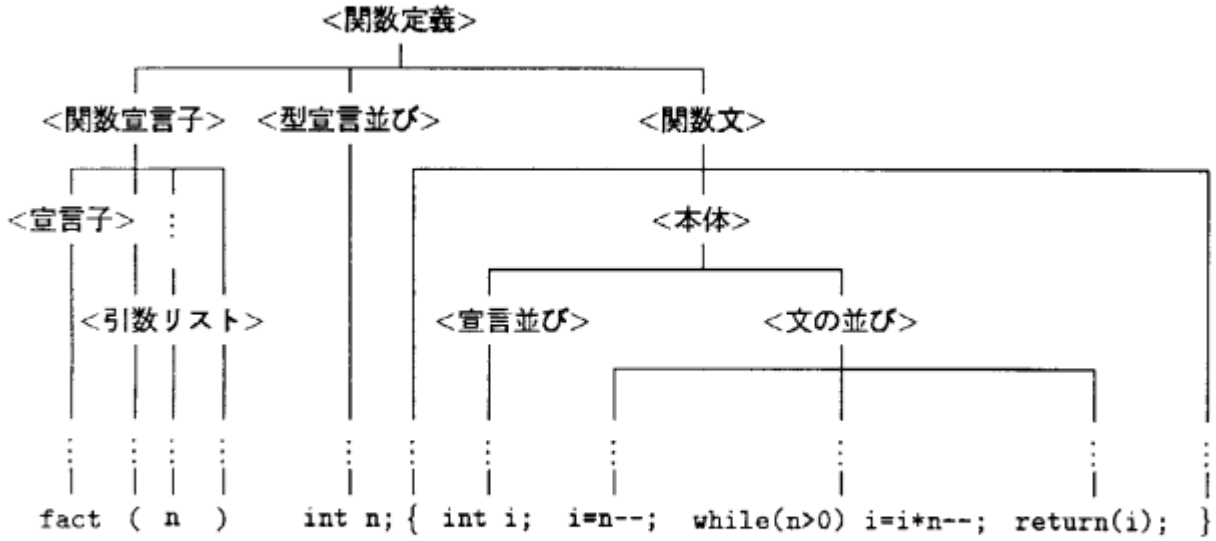

図 3: 文法G_2による構造

2 文脈自由文法と構造木

ACE はプログラム言語の構文によってテキストの構造を決定するが、ACE が扱える文法は文脈自由文法 (context-free grammar) である。ほとんどすべてのプログラム言語の構文は文脈自由文法で定義されるので、事実上、ACE はどのようなプログラム言語をも扱える構造エディタである。

文脈自由文法 G とは 4 項組 (N, Σ, P, S) である。ここで、N は非終端記号 (nonterminal symbol) の空でない有限集合、Σ は終端記号 (terminal symbol) の空でない有限集合である。Σ はアルファベットとも言われる。また、Σ 上の記号列の集合を Σ^* で表す。N と Σ は共通要素を持たないと仮定し、$N \cup \Sigma$ を V で表す。P は、$A \to x$ の形をした生成規則 (production) から成る空でない有限集合である。ここで、A は非終端記号、x は V 上の記号列である。特に、$A \to B$ $(A, B \in N)$ の形をした生成規則を、**単位 (unit) 生成規則**と言う。S は特別な非終端記号で、開始記号と呼ばれる。

次の 2 つの条件を満たす文脈自由文法 $G = (N, \Sigma, P, S)$ を**可逆 (reversible) 文脈自由文法**という:

- $A \to x \in P$ かつ $B \to x \in P$ ならば、$A = B$、
- 任意の 2 つの非終端記号 B, C と 記号列 $x, y \in V^*$ に対して、$A \to xBy \in P$ かつ $A \to xCy \in P$ ならば、$B = C$。

$G = (N, \Sigma, P, S)$ を文脈自由文法とする。V^* の記号列間の関係 $\Longrightarrow$ を次のように定義する。任意の V^* の記号列 x と y に対して、$x = uAv$、$y = uzv$、かつ $A \to z$ が G の生成規則であるならば、$x \Longrightarrow y$ である。次に、$x_0, x_1, \ldots, x_k$ を V^* の記号列とする。

$$x_0 \Longrightarrow x_1,\ x_1 \Longrightarrow x_2,\ \ldots,\ x_{k-1} \Longrightarrow x_k$$

であるならば、$x_0 \stackrel{*}{\Longrightarrow} x_k$ と表し、x_0 から x_k への**導出** (derivation) と言う。

文法 G の任意の非終端記号 A に対して、集合 $L(A,G)$ を

$$L(A,G) = \{w \in \Sigma^* \mid A \stackrel{*}{\Longrightarrow} w\}$$

と定義する。文法 G によって生成される言語 $L(G)$ とは、集合

$$L(G) = L(S,G) = \{w \in \Sigma^* \mid S \stackrel{*}{\Longrightarrow} w\}$$

である。記号列 w が $w \in L(A,G)$ であるとき、w は G によって生成されると言う。

文脈自由文法における導出は、図-2のような木構造として表現することができる。まず、**木** (tree) とは、方向のつけられた線分（**矢** (arrow)）によって結ばれた、有限個の**節** (node) の集合であって、次の条件を満たすもののことである（以下、ある矢印が節1から節2に向かっているとき、その矢は節1から**出て**節2に**入る**と言う）：

1. どの矢もそこに入らない節がただ1つある。この節は**根** (root) と呼ばれる。
2. 木の中のどの節にも、根からその節に至る矢の列がある。
3. 根以外のどの節にも、ちょうど1つの矢が入る。その結果、木の中には矢のループがない。

ある節 m から出る矢が入ってゆく節 n を、m の**子** (direct descendant) と言う。特に、子を持たない節を**葉** (leaf) と言う。節 n が節 m の**子孫** (descendant) であるというのは、節 $n_1, \ldots, n_k$ の列があって、次の条件が満たされることである: $n_1 = m$、$n_k = n$ であって、各 i について n_{i+1} は n_i の子である。

また、木の各節について、その子を1列に並べることができる。節 n_1 と n_2 が、同じ節 n の子であって、その並び方が n_1 のほうが n_2 より前であるとする。このとき、n_1 とその子孫は、n_2 とその子孫の**左**にあると言う。

文脈自由文法 $G = (N, \Sigma, P, S)$ の**導出木** (derivation tree) とは、次のような木のことである:

1. 各節に、V の記号が1つ対応させられている。その記号を、その節の**ラベル** (label) と言う。
2. 根のラベルは S である。
3. 節 n が子孫を少なくとも1つ持つならば、そのラベル A は N の要素である。
4. 節 $n_1, \ldots, n_k$ が、節 n の子を左から並べたものとすると、それらのラベルを $A_1, \ldots, A_k$ とするとき、

   $$A \to A_1 \cdots A_k$$

   は P の生成規則である。

```
[   [ [fact] ( [n] ) ]
    [   [int n;]
        [   {
            [int i;]

            [   [i=n--;]
                [while(n>0) i=i*n--;]
                [return(i);]   ]
        } ]   ]   ]
```

図 4: かっこ記号を用いて表現された構造木

このように、文脈自由文法 G において、導出 $S \stackrel{*}{\Longrightarrow} a_1 a_2 \cdots a_k$ に対して、葉のラベルが左から $a_1, a_2, \ldots, a_k$ である導出木が得られる。

アルファベット Σ 上の**構造木** (skeleton) とは、Σ の要素を葉のラベルとし、葉以外の節にはラベルがない木である。文脈自由文法 G の**構造木** (skeleton) とは、G の導出木から葉以外の節のラベルを取り除いた木である。

構造木は、2 つのかっこ記号 "[" と "]" を用いて、記号列としても表現できる。たとえば、図 -2の構造木は、図 -4のように表現できる。

3 自動カスタマイズ

ACE は、与えられた入力テキストのプリティプリントのフォーマットがそのテキストの構造を表していると解釈し、構造木を生成する。そして、その構造木が自分の持つ文脈自由文法 G のものでないとき、G に対して生成規則を削除・追加することによってカスタマイズを行う。カスタマイズは、次の 2 つの処理によって行われる:

1. 入力テキストのフォーマットにもとづいて構造木を生成する、
2. その構造木が自分の持つ文法のものとなるように、新たな生成規則を学習する。

特に、生成規則の学習は、文脈自由文法の帰納推論の理論にもとづいて行われる。カスタマイズを繰り返すことで、ACE の持つ文法によって決まる構造は、ユーザが想定する構造に近いものになる。

3.1 構造木の生成

まず、入力されたテキストから、構造木を生成する。構造木の生成は、テキストのプリティプリントのフォーマットにもとづいて行われる。ここでは、テキストは、スペース、改行、タブの3つの記号によってプリティプリントされているものとする。

行頭から改行までを、**文**と呼ぶ。したがって、テキストとは文の列である。また、2つの空行にはさまれた文の列を**ブロック**と呼ぶ。

$G = (N, \Sigma, P, S)$を文脈自由文法とする。基本的には、テキストは、次の規則にもとづいてプリティプリントされていると仮定する:

1. 各文 s に対して、$s \in L(A, G)$ である非終端記号 $A \in N$ が存在する。
2. 各ブロック t に対して、$t \in L(A, G)$ である非終端記号 $A \in N$ が存在する。
3. 段下げの深さが、ある固定された深さ以上の文の列 $s_1 s_2 \cdots s_k$ に対して、$s_1 s_2 \cdots s_k \in L(A, G)$ かつ $B \to A \in P$ である2つの非終端記号 $A, B \in N$ が存在する。

このように、ACEでは、単位生成規則はプリティプリントにおける段下げを表すと解釈する。

そこで、まず、入力テキストの構文解析には用いられているが、段下げを表していない単位生成規則を、次の方法を繰り返し適用することで、すべて削除する:

> P から段下げを表さない単位生成規則 $A \to B$ を削除し、P において現れる A を B に置き換える。

このようにしてできた文法を $G' = (N, \Sigma, P', S)$ とする。明らかに、$L(G) = L(G')$ である。

ACEは上記の3つの規則を満たすように、先ほどの前処理を行った文法 G' にもとづき、構造木を生成する。この際、規則3の適用を、ほかの2つの規則よりも優先させる。まず、文を構文解析する。文 s が規則2を満たす場合、ラベルが A である葉を生成する。このように、ACEが生成する構造木は、V 上の構造木である。文 s_1 が規則2を満たさない場合には、それを満たすように s_1 以降の文 $s_2, \ldots, s_k$ $(k \geq 2)$ を加えて、1つの文として扱う。その際、規則3が成立するようにしなければならない。同様に、ブロック t_1 が規則1を満たさない場合にも、規則1を満たすように、t_1 以降のブロック $t_2, \ldots, t_k$ までを1つのブロックとして扱う。構造木の生成においては、段下げは特に重要な意味を持つ。規則3が成立するかどうかにかかわらず、段下げの深さがある深さ以上の文の列 $s_1 s_2 \cdots s_k$ に対しては、単位生成規則の存在を仮定し、規則3を満たすように構造木を生成する。したがって、文法 G' が規則3を満たせない場合、G' のカスタマイズが生じる。

今、次のような生成規則からなる文脈自由文法 $G = (N, \Sigma, P, S)$ がACEに与えられている

ものと仮定する:

$$
\begin{aligned}
P \ = \{ \ & S \rightarrow A_1 A_2 \\
& A_1 \rightarrow B_1 \ (\ B_2\ ) \quad B_1 \rightarrow F \quad B_2 \rightarrow F \\
& A_2 \rightarrow C_1 C_2 \quad C_1 \rightarrow D \\
& C_2 \rightarrow \ \{\ DE\ \} \\
& D \rightarrow \texttt{int n;} \mid \texttt{int i;} \\
& E \rightarrow \texttt{i=n--;} \mid \texttt{return(i);} \mid \texttt{i=i*n--;} \mid \texttt{n>0} \mid E\,E \mid \texttt{while(}\ E\ \texttt{)}\ E \\
& F \rightarrow \texttt{fact} \mid \texttt{n}
\end{aligned}
$$

文法 G は図-1のプログラムを生成するが、その図でのプリティプリントのフォーマットを表現していない。そこで、ACE は G をカスタマイズする。まず、段下げを表していない単位生成規則を削除する。単位生成規則 $B_1 \rightarrow F$、$B_2 \rightarrow F$、$C_1 \rightarrow D$ は、入力テキストの導出に用いられるが、段下げを表していないので、削除される。これらの単位生成規則を削除した文法 G' は次の生成規則を持つ:

$$
\begin{aligned}
P \ = \{ \ & S \rightarrow A_1 A_2 \\
& A_1 \rightarrow F\ (\ F\ ) \\
& A_2 \rightarrow DC_2 \\
& C_2 \rightarrow \ \{\ DE\ \} \\
& D \rightarrow \texttt{int n;} \mid \texttt{int i;} \\
& E \rightarrow \texttt{i=n--;} \mid \texttt{return(i);} \mid \texttt{i=i*n--;} \mid \texttt{n>0} \mid E\,E \mid \texttt{while(}\ E\ \texttt{)}\ E \\
& F \rightarrow \texttt{fact} \mid \texttt{n}
\end{aligned}
$$

ACE は、G' にもとづいて、図-5のような構造木を生成する。この構造木をかっこ記号 "[" と "]" を用いて表現すると、図-6のようになる。ただし、葉の部分には、非終端記号ではなく、テキストを表示した（図において、左端に示された番号は行番号である）。図-6において、入力テキストは5行目の空行によって2つのブロックに分割されるが、最初のブロックは規則2を満たさないので、入力テキスト全体が1つのブロックとして扱われる。また、3行目の文は、規則1を満たさないので、4行目から10行目の文を加えて1つの文として扱われる。同様に、7行目の文も8行目とともに1つの文として扱われる。ただし、8行目は段下げが行われているので、規則3を満たすように、単位生成規則があるものとして構造木が生成される。さらに、4行目から9行目までも、段下げが行われている。したがって、この場合も同じように、単位生成規則があるものとして構造木を生成する。このように、生成された構造木は、もはや G' の構造木ではない。そこで、G' のカスタマイズを行うことになる。

3.2 生成規則の学習

入力されたテキストから生成された構造木が、ACE の持つ文法の構造木ではない場合、ACE は新たな生成規則を追加する。生成規則の追加は、文脈自由文法の帰納的学習の理論にもとづいて行われる。

榊原 [3] は、具体例から効率良く可逆文脈自由文法を帰納的に学習するアルゴリズム（以後

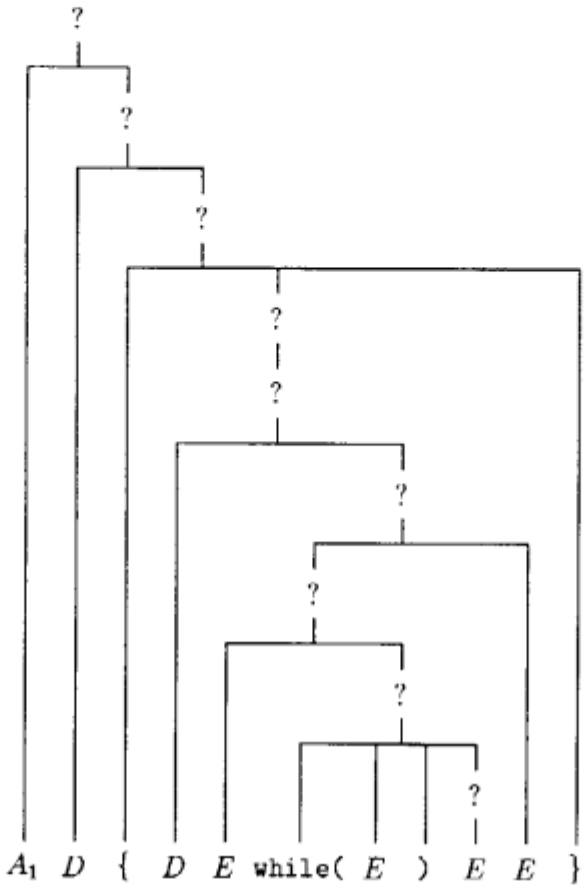

図 5: 生成された構造木

```
1:  [   [fact(n)]
2:      [int n;]
3:      [ {
4:        [   [int i;]
5:
6:            [i=n--;]
7:            [while(n>0)
8:              [   [i=i*n--;]   ]   ]
9:            [return(i);]   ]
10:     } ]   ]
```

図 6: 生成された構造木（かっこを用いた表現）

これを **TAID**と呼ぶ）を示した。ここでは、その概略だけを述べる（詳しくは、文献 [3] を参照）。

文法 G をアルファベット Σ 上の可逆文脈自由文法、$E(G)$ を言語 $L(G)$ の有限部分集合、$T(E(G))$ を集合

$$T(E(G)) = \{t \mid t \text{ は } E(G) \text{ の要素の導出木から得られる構造木}\}$$

とする（集合 $T(E(G))$ の要素は、G の構造木だけであることに注意）。集合 $T(E(G))$ を文法 G の**具体例**という。

文法 G の具体例 $T(E(G))$ が与えられたとき、アルゴリズム **TAID**は次の処理を有限回実行することによって学習する:

1. すべての構造木のすべての節に新しいラベルをつける。
2. ラベルづけられた構造木から、生成規則を作る。その際、葉以外のラベルはすべて新しい非終端記号である。
3. 可逆文脈自由文法の定義に適うように、新たに作られた非終端記号のうち、同じものと見なせるものは、1 つの非終端記号に置き換える。

アルゴリズム **TAID**は、常に、$E(G) \subseteq L(G') \subseteq L(G)$ なる可逆文脈自由文法 G' を、入力のサイズの多項式時間で帰納的に学習する。具体例 $T(E(G))$ が十分多くの要素を含んでいるとき、$L(G) = L(G')$ となる。

ACE はアルゴリズム **TAID**にもとづいて、新しい生成規則を学習する。今、ACE は文法 $G = (N, \Sigma, P, S)$ を持つと仮定する。前節で述べたように、入力テキストから生成された構造木は、V 上の構造木である。この入力テキストから、ACE は **TAID**にもとづいて、可逆文脈自由文法 $G_r = (N_r, V, P_r, S')$ を学習する。ただし、ここで、$N_r \cap N = \emptyset$ である。学習後、ACE は $G' = (N \cup N_r, \Sigma, P \cup P_r, S')$ をカスタマイズされた文法として持つ（ここで、G の開始記号 S は、生成規則の右辺に現れないかぎり、S' に書き換える）。

前節の例では、次の生成規則が文法 G' に追加される:

$$\begin{aligned}
&H_1 \rightarrow E \\
&H_1 \rightarrow \texttt{while(}\ E\ \texttt{)}\ H_1 \\
&H_2 \rightarrow DH_1 \\
&H_3 \rightarrow H_2 \\
&H_4 \rightarrow \{\ H_3\ \}
\end{aligned}$$

明らかに、学習された生成規則を追加した文法 G'' が生成する言語は、元の文法が生成する言語と同じ、つまり、$L(G) = L(G'')$ である。このように、ACE では、カスタマイズ前の文法が生成する言語とカスタマイズ後の文法が生成する言語は常に同じである。しかし、生成規則を追加したことによって、複数個の導出木を持つテキストが存在することになる。そこで、ACE では、新しく学習された生成規則から先に用いて構文解析を行う。

4 おわりに

ACE は、入力されたテキストのプリティプリントにもとづいて、プログラム言語の文法をカスタマイズする。カスタマイズされた文法にもとづいて構文解析木を生成し、プリティプリント・リードや編集の単位を定める。したがって、カスタマイズを繰り返すうちに、プログラマが想定する構造によって、編集を行うことが可能になる。さらに、プリティプリント・リードも、プログラマが想定する構造にもとづいて行われるので、見やすいものになる。

ここでは、構造木を生成するのに、フォーマット情報としては段下げの情報しか用いなかった。しかし、段下げの方法にも多様性があり、プログラマは異なる段下げごとに異なる構造を想定していると考えられる。したがって、今後、その他のフォーマット情報も用いて、より細かい構造を抽出する必要がある。

また、フォーマット情報に対して、構文上では単位生成規則のみを用いた。しかし、単位生成規則だけでは、多様な段下げなどの複雑な構造を表現できない。複雑なフォーマット情報を表現するには、より詳細に生成規則の型を決めなければならない。さらには、属性文法などのより表現力のある方法を用いることを検討し、学習方法を解明する必要もある。

さらに、テキストのフォーマット情報以外にも、編集の手順や編集単位などの情報を用いてカスタマイズすることも考えられる。

謝辞

日頃、有益な助言を頂く国際研・学習システム研究グループの横森研究員に感謝します。なお、本研究は第五世代コンピュータプロジェクトの一環として行ったものである。

参考文献

[1] J. E. Hopcroft and J. D. Ullman （野崎 他訳）. **オートマトン 言語理論 計算論** I, II. サイエンス社, 1984.

[2] B. W. Kernighan and D. M. Ritchic （石田晴久 訳）. **プログラミング言語** C. 共立出版, 1981.

[3] Y. Sakakibara. An efficient learning of context-free grammars for bottom-up parsers. 1988. To appear in Proceedings of FGCS '88.

[4] 和田英一. **エディタとテキスト処理**. bit, Vol. 14–15, 1982–1983.