Prologによる構文解析

田中穂積

1. はじめに

Prologはフランスのマルセイユ大学のColmerauerが開発した言語である。Colmerauerは，かつてカナダのモントリオール大学でTAUMとよばれる機械翻訳システム開発プロジェクトに従事していたことがある。そこで彼はq-systemsとよばれる非決定的プログラミング言語（non-deterministic programming language）を開発し，機械翻訳システムで必要となる自然言語処理プログラムを作成している。

その後彼は，意味の問題と機械的な定理証明法（導出原理；resolution principle）に興味を持ち，Prologを考案，それを用いて，仏語対話システムの開発に取り組む。彼は当初，このシステムの中で演繹的な推論を行う部分の記述にPrologが適すと考えていたが，しばらく実験を続けるうちにPrologは，ロボティクスや音声認識の分野にも応用可能であることが明らかとなった。しかし，構文解析についてはq-systemsが優れていることが判明した。そこで，Prologの枠組みに，q-systemsの持つ構文解析機能を埋め込むことを考えた。その結果生れたものがmetamorphosis grammarである[1]。metamorphosisを辞書で調べると「変質，形態の変化」という意味がある。これは3.1で説明するようにmetamorphosis grammarの形式で書かれた文法規則がPrologプログラムに変換され，形態の変化を遂げることをあらわしたかったからだと思われる。

その後，イギリスのエジンバラ大学では，WarrenとPereiraが，通称Dec-10 Prologとよばれるプログラムを開発した。そしてその上に，Colmerauerのmetamorphosis grammarの考え方を一層洗練したDCG（Definite Clause Grammar）を開発した。このDCGで書かれた各文法規則は，metamorphosis grammarと同様に，ほぼ相似なProlog節に変換され実行されるが(3.1)，これは，自然言語処理でよく使われるATNGの動作と等価でより高速であるだけでなく，Prologに内蔵された強力なパターン照合機能が利用できるため，文法体系を簡潔に記述することが可能なことを示した[2]。

その後Pereiraは，英語の疑問文や関係代名詞文にみられる左外置（left extraposition）を扱うために，DCGを拡張したXG（extraposition grammar）を開発している[3]。

ここで注意すべきことは，DCGで書かれた文法規則を変換してできたPrologプログラムは，3.1で説明するようにトップダウンに構文解析するプログラムとして動作する。したがって，DCGに左回帰的な文法規則があり，それが適用されると無限ループに陥る。我が国の電子技術総合研究所の松本らは，この欠点を回避する技術を開発した。これは，DCGで書かれた文法規則を変換して，ボトムアップに構文解析するPrologプログラムを作り出す技術である。この変換を行うシステムはBUPとよばれている[4]（ただし，BUP使用者であっても，文法の記述はDCGで行うこと注

意)。

以下では,DCGとBUPについて説明する。XGについては文献［3］を参考にされたい。

2．文脈自由文法規則とホーン節

英語や日本語などの自然言語の文法記述に，言語学者は文脈自由文法規則の形式をよく用いる。自然言語処理の研究者も，文法記述に文脈自由文法規則を良く用いる。これは，文脈自由文法規則に対して効率の良い構文解析アルゴリズムが発見されていること，文法規則の作成が比較的見通し良く行いうることによると思われる。英語のサブセットに対する文脈自由文法規則の例を以下に示す。

$s \rightarrow np\ \ vp$ (1)

$np \rightarrow det\ \ noun$ (2)

$np \rightarrow noun$ (3)

$vp \rightarrow vi$ (4)

$vp \rightarrow vt\ \ np$ (5)

辞書項目も同様に，次の(6)〜(10)に示す。

$noun \rightarrow John$ (6)

$noun \rightarrow children$ (7)

$vi \rightarrow walk$ (8)

$vi \rightarrow walks$ (9)

$det \rightarrow a$ (10)

以上の文脈自由文法規則の形式の際立つ特徴は，記号「→」の左に，唯一の文法的記号（非終端記号）しかあらわれないことである。したがってこの形式は，ホーン節（Horn clause）によるPrologプログラムと非常に良い対応をなす。たとえば(1)に対して，次のホーン節を対応させることができる（s if np & vp と読む)。

$s :- np,\ vp.$

しかし，このホーン節表現では，s が np と vp にまとめられて，しかも np としてまとめられる部分文と，vp としてまとめられる部分文とが重複なく連続して，全体として一つの文を構成するという事実が表現できない（次節参照)。

3．DCG (Definite Clause Grammar)

2に示した文脈自由文法による文法規則の記述はDCGの最も単純な形式である。DCGによる文法記述の特徴は，

(イ) 非終端記号を述語とみなし，これに任意個の引数を持たせることができる。

(ロ) 記号「→」の右側の任意の場所に｛｝で囲まれたPrologプログラムが挿入できる。（これは補強項と以下ではよぶ。)

(イ),(ロ)により，DCGは補強文脈自由文法である。これらの補強により，文脈自由文法規則としてあらわしにくい構文上の制約条件（たとえば，時制の一致，呼応や，主語が三人称単数の場合の一般動詞現在形の語尾変化など）を容易に規則化できるだけでなく，(ロ)の補強項として，意味処理を行うプログラムをも記述することができる。

DCGの本質は(イ),(ロ)にあるが，説明を簡単にするために，しばらく，(イ),(ロ)が含まれない最も単純なDCGによる文法について説明する。ここで，2に示した文脈自由文法規則のDCGによる正確な記述を図1に示す。

$s \rightarrow np,\ vp.$	(1)	$noun \rightarrow [john].$	(6)
$np \rightarrow det,\ noun.$	(2)	$noun \rightarrow [children].$	(7)
$np \rightarrow noun.$	(3)	$vi \rightarrow [walk].$	(8)
$vp \rightarrow vi.$	(4)	$vi \rightarrow [walks].$	(9)
$vp \rightarrow vt,\ np.$	(5)	$det \rightarrow [a].$	(10)

図1　DCG形式の文脈自由文法規則

図1の記号「→」の意味について考えてみよう。たとえば矢印の向きに沿って(1)を読むと，文(s)を名詞句（np）と動詞句（vp）とに分解せよ，また矢印の向きとは逆に（右から左に）(1)を読むと，np と vp とから s を構成せよ，と読むことができる。前者は，より抽象的なものをより詳細なものへと分解するトップダウンの読み，後者は，より詳細かつ具体的なものから，より抽象的なものを構成するボトムアップの読みである。図1の(1)か

ら(10)のDCGを用いた構文解析法として，両者のそれぞれの読みに対応するトップダウン法とボトムアップ法がある。これについて次に説明する。

3.1 トップダウン法

Colmerauerのmetamorphosis grammarも，Pereira & WarrenのDCGも，トップダウンに構文解析するためのPrologプログラムに変換される。変換されたPrologプログラムは，(1)〜(10)にほぼ相似で1対1に対応するホーン節の集合である(図6参照)。

ここで「ほぼ相似」といった意味は，変換後のPrologプログラムの各ホーン節には，解析すべき(部分)文が述語の引数として埋め込まれているからである。それにより，たとえば(1)に対してsが過不足なく二つの部分文npとvpに分割され，しかも前半の部分文がnpで，後半がvpであることを保証する。これは図1の(1)を次のホーン節に変換することで実現できる。

$$s(X, Z) \ :- \ np(X, Y), vp(Y, Z). \qquad (1)'$$

ここで変数X，Y，Zには，部分文がユニファイされる。たとえば，

$$s([john, walks], [\ ]) \ :- \ np([john, walks], [walks]), vp([walks], [\ ]).$$

では，$X=[john, walks]$，
$Y=[walks]$，
$Z=[\ ]$
となる。

これは次のように解釈される。

(i) リストXからリストZを差し引いた残りのリスト$[john, walks]$がsである。

(ii) リストXからリストYを差し引いた残りのリスト$[john]$がnpである(図2参照)。

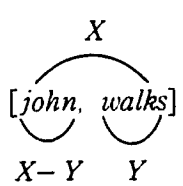


図2 差分リストによる文の分割

(iii) リストYからリストZを差し引いた残りのリスト$[walks]$がvpである。

ここで，(1)から(10)までのDCGによる文法が，どのようなPrologプログラムに変換されるかを，図3の(1-1)から(1-10)に示す。(この変換を行うトランスレータを以下ではDCGトランスレータとよぶ(図6参照))。

```
s(X, Z)   :- np(X, Y), vp(Y, Z).        (1-1)
np(X, Z)  :- det(X, Y), noun(Y, Z).     (1-2)
np(X, Z)  :- noun(X, Z).                (1-3)
vp(X, Z)  :- vi(X, Z).                  (1-4)
vp(X, Z)  :- vt(X, Y), np(Y, Z).        (1-5)

noun([john | X], X).                    (1-6)
noun([children | X], X).                (1-7)
vi([walk | X], X).                      (1-8)
vi([walks | X], X).                     (1-9)
det([a | X], X).                        (1-10)
```

図3 図1の文法をPrologプログラムに変換した結果

図3の(1-1)から(1-10)のPrologプログラムを用いて，*john walks*という文が，どのように構文解析されるかを以下に示す。解析は，次のゴールを実行することから始まる。

① ?− $s([john, walks], [\ ])$.

①により，(1-1)が選択され，新しいゴール②を実行する。

② ?− $np(john, walks], Y), vp(Y, [\ ])$.

②の最初の述語npに対して，(1-2)が選択され，新しいゴール③を実行する。

③ ?− $det([john, walks], Y'), noun(Y', Z)$.

③の最初の述語detに対して，(1-10)を選択しようとするが，第1引数のユニフィケーションに失敗するので，バックトラックが生じる。そこで②の述語npに対して(1-2)の次の(1-3)が選択され，新しいゴール④を実行する。

④ ?− $noun([john, walks], Y)$.

④の述語$noun$に対して，(1-6)が選択される。このとき$Y=[walks]$とすると④は成功し，した

がって②の最初の述語 *np* が成功する。そこで *np* の次の述語 *vp* を新しいゴールとして実行する。

⑤ ? − *vp*([*walks*], [ ]).

⑤の述語 *vp* に対して，(1-9)が選択され，*vp* は成功し，結局②が成功して①が立てたゴールが成功する。

以上の動作を追跡すると，与えられた文法規則に従って，*s* を *np* と *vp* に分解し，*np* を *noun* に，*noun* を *john* に分解し，*vp* を *vi* に，*vi* を *walks* に分解しながらプログラムが実行されていることがわかる。これは，トップダウンで縦型探索(depth-first)戦略を採用した構文解析を行うことと等価である。

したがって，次のことが理解できる。

(i) DCG形式の文法規則（図1）を，それとほぼ相似で1対1に対応するPrologプログラム(図3)に(自動的に)変換する。

(ii) ゴールとして *s*(〈入力文のリスト〉, [ ]) を実行すると図3のPrologプログラムが動作し，トップダウンで縦型探索による構文解析が行われる。

(iii) したがって，構文解析用の特別なプログラム（パーザ）が不要。

本章の冒頭で，DCGの本質(イ)，(ロ)を述べておいた。これまでは，構文解析の骨子を説明するために，最も単純なDCGを取り上げた。そこで(イ)で述べたように図1のDCG形式の規則にあらわれる非終端記号が，幾つかの引数を持つDCG形式の文法規則について説明する。

我々は構文解析結果として，構文解析木を得たいことがある。この場合には，図1の規則を次のものに変える。

$$
\begin{array}{rcll}
s([s,\ P,\ Q]) & \rightarrow & np(P),\ vp(Q). & (1)' \\
np([np,\ P,\ Q]) & \rightarrow & det(P),\ noun(Q). & (2)' \\
np([np,\ P]) & \rightarrow & noun(P). & (3)' \\
vp([vp,\ P]) & \rightarrow & vi(P). & (4)' \\
\vdots & & & \vdots \\
noun([noun,\ john]) & \rightarrow & [john] & (6)' \\
\vdots & & & \vdots \\
vi([vi,\ walks]) & \rightarrow & [walks] & (9)' \\
det([det,\ a]) & \rightarrow & [a] & (10)'
\end{array}
$$

これらをPrologプログラムに変換するのであるが，たとえば(1)'と(6)'を次のように変換する。

```
s([s, P, Q], X, Z) :- np(P, X, Y),
                      vp(Q, Y, Z).
noun([noun, john], [john | X], X).
```

以上のように変換してから次のゴール，

? − *s*(*TREE*, [*john*, *walks*], [ ]). (11)

を実行すれば，構文解成終了後の変数 *TREE* には，構文解析結果として，次の値が返される。

[*s*, [*np*, [*noun*, *john*]], [*vp*, [*vi*, *walks*]]]

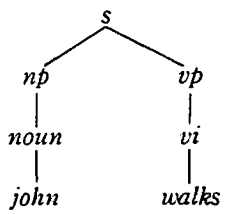


図4 *John walks* の構文解析結果（構文解析木）

これを図示したものが図4の構文解析木である。

DCG形式の文法規則では，文脈自由文法規則としてはあらわしにくい構文上の制約条件が簡単に記述できる。たとえば，主語が三人称単数の場合に一般動詞現在形に語尾変化（*s* などが付く）があるという事実は，(1)″から(10)″の規則であらわすことができる。

$$
\begin{array}{rcll}
s([s,\ P,\ Q]) & \rightarrow & np(P,\ Syn),\ vp(Q,\ Syn). & (1)'' \\
np([np,\ P,\ Q],\ Syn) & \rightarrow & det(P,\ Syn), & \\
 & & noun(Q,\ Syn). & (2)'' \\
np([np,\ P],\ Syn) & \rightarrow & noun(P,\ Syn). & (3)'' \\
vp([vp,\ P],\ Syn) & \rightarrow & vi(P,\ Syn). & (4)'' \\
\vdots & & & \vdots \\
noun([noun,\ john],\ singular\ 3) & \rightarrow & [john]. & (6)'' \\
noun([noun,\ children],\ plural) & \rightarrow & [children]. & (7)'' \\
\vdots & & & \vdots \\
vi([vi,\ walks],\ singular\ 3) & \rightarrow & [walks]. & (9)'' \\
det([det,\ a],\ singular\ 3) & \rightarrow & [a]. & (10)''
\end{array}
$$

以上のように，構文解析木を得るための引数の他に，もう一つ引数を用意する。(1)″では，述語 *np*

と *vp* の第2引数が同一変数 *Syn* になっている。これは，*np* の第2引数に返される値が，*vp* の第2引数に返される値と同じであることを要請している。以前と同様に(11)をゴールとして実行すると，(6)″の *noun* の第2引数である *singular*3 が(3)″を介して(1)″の *np* の第2引数 *Syn* に返される。同様にして，(9)″の *walks* からは，*vi* の第2引数である *singular*3 が(4)″を介して(1)″の *vp* の第2引数 *Syn* に返され同一であるので成功する。こうして，(1)″の *np*，すなわち主語が三人称単数(*singular*3)の場合（*john* の場合）に，動詞 *walk* の語尾に *s* が付いているかどうかが調べられる。

一方もし，

$?- \quad s(TREE, [children, walks])$ (12)

というゴールを実行すれば，(1)″の *np* の第2引数 *Syn* には((6)″より) *plural* が返され，*vp* の第2引数 *Syn* には ((9)″より) *singular*3 が返されるので不一致が起こり，(12)のゴールは失敗する。これは構文解析に失敗したことを意味している。

同様な理由から(2)″は，*det* が *singular* で *noun* が *plural* の場合（たとえば *a children* など）を排除することをあらわしたものであることがわかる。

ここでもし解析すべき文が，*I walk* であるとき，*I*，*walk* の辞書記述をどのようにすべきだろうか。たとえば次のようにしてみる。

$noun([noun, i], singular1) \rightarrow [i].$ (11)″†

$vi([vi, walk], X) \rightarrow [walk].$ (8)″

こうすると，(1)″の *np* の *Syn* には(11)″の *singular*1 が，また *vp* の *Syn* には(8)″の変数 *X* が返され，*X*＝*singular*1 として成功する。

ところがこれでは，解析すべき文が *john walk* のときも，*I walk* と同様な理由（ただし(1)″の *np* の *Syn* の値は *singular*3 となる）で成功する。このジレンマをどのようにして解決したら良いだろうか。これは，本章の冒頭の(ロ)で述べた DCG 形式，すなわち{ }で囲まれた補強項によって容易に解決することができる。(1)″，(8)″をそれぞれ(1)‴，(8)‴のようにする。

$s([s, P, Q]) \rightarrow np(P, Syn\,1),$
$\qquad vp(Q, Syn\,2),$ (1)‴
$\qquad \{concord(Syn\,1, Syn\,2)\}.$

$vi([vi, walk], present) \rightarrow [walk].$ (8)‴

そして補強項の述語 *concord* を次のように定義する。

$concord(singular3, singular3) \ :- \ !.$ (13)

$concord(X, singular3) \ :- \ !, fail.$ (14)

$concord(X, Y).$ (15)

(13)は，主語が三人称単数なら，一般動詞の現在形の語尾に *s* が付加されていなければならないという制約条件を，(14)は，主語が三人称単数ではないのに，一般動詞の現在形の語尾に *s* が付加されることはないという制約条件をあらわしている。

{ } で囲まれた補強項には，意味解析を行う述語を記述することもできる。それにより，構文解析過程と意味解析過程とを融合することができるが，これは本稿の範囲を越える。

以上で説明した，トップダウンと縦型探索法による構文解析には一つの欠点がある。それは，左回帰規則を適用すると無限ループに陥るという欠点である(図5)。これは，ボトムアップ法による構文解析では避けることができる。次節では，これについて説明する。

左回帰規則 : $np \rightarrow np, srel.$

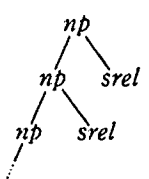


図5 左回帰規則と無限ループ

3.2 ボトムアップ法

トップダウン法による構文解析では，3.1の最後に述べたように，左回帰規則を適用すると無限

† Prolog では，大文字で始まる文字列は変数とみなされる。そこで，入力された文に含まれる大文字はすべて小文字に直す。たとえば $I \rightarrow i$ のように。

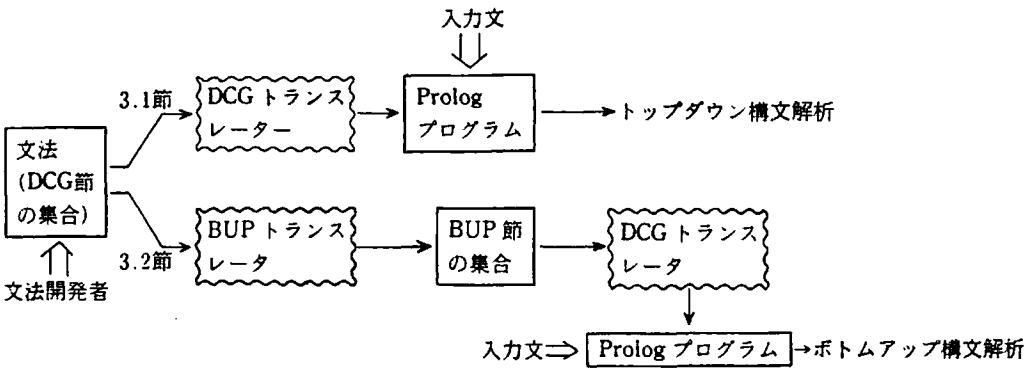


図6　DCGトランスレータとBUPトランスレータ

ループに陥る。これを避けるための一つの方法は，左回帰文法規則を右回帰文法規則に変換することである。詳しく説明することはできないが，こうすると，一つの左回帰文法規則から複数個の右回帰文法規則が得られ，文法規則の自然さが損われるため，文法開発の見通しが悪くなる。もう一つの方法は，ボトムアップ法による構文解析を行うことである。

3.1のDCG形式の文法を変換して，ボトムアップに構文解析するPrologプログラムを作り出すことはできないだろうか。電子技術総合研究所の松本らは，それを可能にする技術を開発している[4]。そこで以下ではその基本的な考え方を説明する。まずDCG形式の文法（DCG節の集合）を別のDCG節（以後これをBUP節とよぶ）の集合に変換する。これはBUPトランスレータで行う。変換して得たBUP節の集合はDCGトランスレータにより，部分文についての差分リスト（図2参照）を付加したPrologプログラムを作り出す。このPrologプログラムを実行することによりボトムアップに構文解析が行われる。図6にその概要を示す。

ここで，図6のDCG節の集合が，どのようなBUP節に変換されるかを説明し，それがPrologプログラムに変換された後，どのようにボトムアップ（＋縦型探索）構文解析が行われるかを説明する。使用する文法は，図1に示すものとする。また解析すべき文は *john walks* であるとする。

BUPトランスレータにより，図1は図7のように変換される。図7には，以下の説明で必要なBUP節のみが示されている。

$np(G) \rightarrow \{link(s, G)\}, goal(vp), s(G).$ (A)
⋮
$noun(G) \rightarrow \{link(np, G)\}, np(G).$ (C)
$vi(G) \rightarrow \{link(vp, G)\}, vp(G).$ (D)
⋮
$dict(noun) \rightarrow [john].$ (F)
⋮
$dict(vi) \rightarrow [walks]$ (I)
$dict(det) \rightarrow [a]$ (J)

$link(np, s).$	(K)	$s(s, [\ ], [\ ]).$	(P)
$link(noun, np).$	(L)	$np(np, [\ ], [\ ]).$	(Q)
$link(noun, s).$	(M)	$noun(noun, [\ ], [\ ]).$	(R)
$link(vi, vp).$	(N)	$vp(vp, [\ ], [\ ]).$	(S)
$link(X, X).$	(O)	$vi(vi, [\ ], [\ ]).$	(T)

図7　図1の文法をBUPトランスレータで変換して得たBUP節の集合

図7の(A)から(J)がそれぞれ図1の(1)から(10)と1対1に対応している。(K)以降はBUPトランスレータが(A)から(J)に自動的に付加したユニット節である。

link 節は，$a \rightarrow b, \cdots$ なる文法規則があれば，$link(b, a)$ が成り立ち，*link* は反射的かつ推移的関係であるから，すでに存在する *link* 節を用いて，(文法規則が与えられると)一意に決定できる。*link* 節は，b を根とする部分構文解析木が成長し将来 a を根とする部分構文解析木ができる可能性（b から a への到達可能性）を調べるために使われ

る。aが与えられれば，aはトップダウン的な予測としてはたらくために，ボトムアップに成長する木のうち無効なものを事前に排除でき，効率良く構文解析が行われる。

図7の(P)から(T)は停止条件節とよばれ，その機能については後述する。

図7のBUP節には，述語*goal*の定義が与えられていない。それを以下に示す。

```
goal(G, X, Z) :- dict(C, X, Y),    (U-1)
                 link(C, G),       (U-2)
                 P=..[C, G, Y, Z], call(P).   (U-3)
```
(U)

図7の(A)から(J)のBUP節はDCGの形式であるから，DCGトランスレータにより，次の(A)'から(J)'のPrologプログラムに変換する（図6参照）。

```
np(G, X, Z) :- link(s, G), goal(vp, X, Y),
               s(G, Y, Z).                    (A)'
        ⋮                                      ⋮
noun(G, X, Z) :- link(np, G), np(G, X, Z).
                                              (C)'
vi(G, X, Z) :- link(vp, G), vp(G, X, Z).
                                              (D)'
        ⋮                                      ⋮
dict(noun, [john | X], X).                    (F)'
        ⋮                                      ⋮
dict(vi, [walks | X], X).                     (I)'
dict(det, [a | X], X).                        (J)'
```

上記した(A)'～(J)'，図7の(K)～(T)，先の(U)((U-1)～(U-3))の*goal*節とを用いて，*john walks*をボトムアップに構文解析することができる。解析は，次のゴールを実行することから始まる。

① ?- *goal*(*s*, [*john*, *walks*], [ ]).

①により(U)が選択され，新しいゴール②が作り出される。

② ?- *dict*(*C*, [*john*, *walks*], *Y*),
link(*C*, *s*),
P=..[*C*, *s*, *Y*, [ ]], *call*(*P*).

②の最初の述語により(F)'が選ばれ，*john*に対する辞書引きを行う。これによりC=*noun*, Y=[*walks*]となり，Pに渡される。

goal(*s*, [*john*, *walks*], [ ])

noun
|
john *walks*

次に*link*(*noun*, *s*)を実行するが，これは図7の(M)により成功する。

goal(*s*, [*john*, *walks*], [ ])
link(*noun*, *s*)
noun
|
john *walks*

さらに②の最後で(P=)*noun*(*s*, [*walks*], [ ])を*call*する。その結果(C)'が選択され，(C)'の*link*(*np*, *s*)が(K)で成功して，(C)'の最後の*np*(*s*, [*walks*], [ ])を実行する。

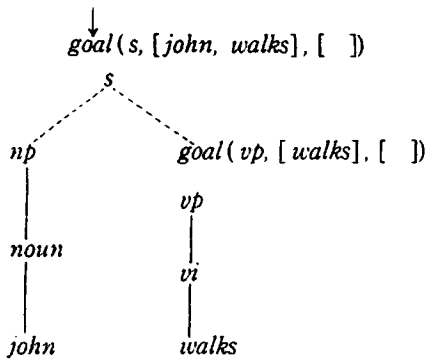
goal(*s*, [*john*, *walks*], [ ])
link(*np*, *s*)
np
|
noun
|
john *walks*

np(*s*, [*walks*], [ ])の実行により，(A)'が選ばれ，*link*(*s*,*s*)が成功((O))するから，(A)'のボディーにある*goal*(*vp*, [*walks*], *Y*)を実行する。

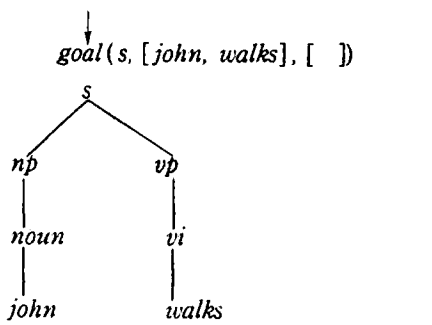
goal(*s*, [*john*, *walks*], [ ])
s
np *goal*(*vp*, [*walks*], [ ])
|
noun
|
john *walks*

以下同様にして，*goal*のボディーの実行が進み，*walks*の辞書引きをして，最終的に*vi*(*vp*, [ ],

[ ])を実行する。それにより(D)'が選択され，結局(D)'の最後の *vp*(*vp*, [ ], [ ]) を実行する。



vp(*vp*, [ ], [ ]) は停止条件節の(S)があるから成功し，最終的に *goal*(*vp*, [*walks*], [ ]) が成功したことになる。



上の状態で *s*(*s*, [ ], [ ]) なる停止条件節を実行し，ボトムアップの構文解析が成功したことにやる。

以上をまとめると次のようになる（図6）。

(i) 文法作成はDCG形式で行う。

(ii) BUPトランスレータにより，(i)をBUP節の集合に変換し，これをPrologプログラムに変換する。

(iii) *goal* 節(U)を定義し，ゴールとして *goal*(*s*, 〈入力文のリスト〉, [ ]) を実行すると，図7のPrologプログラムが動作し，(*link* 節による）トップダウン予測を利用したボトムアップで縦型探索による構文解析が行われる。

(iv) したがって，構文解析用の特別なプログラム（パーザ）が不要。

(v) 左回帰文法規則が許される。

繰り返して強調しておくが，文法開発者は，DCG形式の文法を扱うだけで良く，これまで説明したBUP節に対応するPrologプログラムの動作を知る必要はない。またBUPトランスレータを用いる場合にも，3の冒頭で述べた(イ)，(ロ)はそのまま利用できる。

最近，*goal* 節をわずかに変更して，BUP節の実行速度を大幅に向上させるアルゴリズムが開発された。これについては文献［4］を参照してほしい。

参考文献

［1］ Colmerauer, A.: Metamorphosis Grammar, in Bolc ed.: *Natural Language Communication with Computers*, Springer-Verlag, 1978, 133-190.

［2］ Pereira, F. & Warren, D.: Definite Clause Grammar for Language Analysis——A Survey of the Formalism and Comparison with Augmented Transition Networks, *Artificial Intelligence*, 13, 1980, 231-278.

［3］ Pereira, F.: Extraposition Grammar, *American Journal of Computational Linguistics*, 7, 4, 1981, 243-256.

［4］ Matsumoto, Y., Tanaka, H. et. al.: BUP——A Bottom-up Parser Embedded in Prolog, *New Generation Computing*, 1, 2, 1983, 145-158.

［田中穂積 TANAKA, Hozumi:
東京工業大学工学部情報工学科］