論理型言語による構文解析法 YAP について†

林 達也††

自然言語を対象とした論理型言語による構文解析法としては，これまでに DCG パーサ（以下単に DCG と略称する），BUP, SAX が存在する．本論文では，これらとは異なる新しい方法 YAP について提案する．YAP の特徴は次のとおりである．（1）DCG, BUP, SAX がそれぞれ縦型下降式，縦型上昇式，横型上昇式に基づいているのに対して，YAP は横型下降式を採っている．（2）左回帰規則，ε 規則を共に許している．(3)Knuth の LR 手法を曖昧さを含む一般の CFG にまで適用できるように改めた Earley の手法と，基本的には等価であるが，入力解析と解析木作成の一体化や解析表でなくスタックの使用，それに処理の効率化などの点で改良を行っている．（4）YAP の時間的，空間的能率は，入力の長さを n とした時に，曖昧さを含まない文法の場合は $O(n^3)$ である．また，解析パス数が定数オーダの場合は $O(n)$ である．（5）論理型言語によるパーサの動作は，基本的には SAX と同様に文法規則の右辺の要素ごとに対応したホーン節の集合として定義される．ただしホーン節は，右辺の要素が規則のどの位置に存在するかによって，左端型，中間型，右端型の３種類に分けられる．そしてスタック情報の授受によって相互に関連づけられる．

1. まえがき

自然言語を対象とした，論理型言語による構文解析法としては，これまでに DCG[1], BUP[2),3)], SAX[4),5)] が開発されている．本論文では，これらとは異なる新しい方法について提案する．この新しい解析法を，この論文では，YAP (Yet Another efficient Parsing method for general context free grammars based on the logic programing language) と呼ぶことにする．YAP と他の方法との相違点は次のとおりである．

（1） DCG, BUP, SAX がそれぞれ縦型下降方式，縦型上昇方式，横型上昇方式に基づいているのに対して，YAP は横型下降方式を採っている．

（2） YAP は下降方式に基づいてはいるが，左回帰の構文規則を許している．

（3） 実用上，必要性が稀ではない ε 規則も，自由に使用することができる．

YAP は，Knuth の LR 手法[6)~8)]を曖昧さを含む一般の CFG にまで適用できるように改めた Earley の手法[9),10)]と，基本的には等価であるが以下の点で異なる．

（1） Earley の手法は，入力列を走査してまず解析表（複数）を作成し，その後で，解析表に基づき解析木ないし right parse[10)]を作成する．YAP は，これに対して，入力解析と併行して解析木の作成を行う．このために可能な解析パスをすべて抽出し分離している．

（2） YAP では，解析表を用いる代りに，それと等価なマルチスタックを使用している．

（3） スタックの効率を上げるために，YAP では，必要最少限の情報のみを格納するように考慮している．

以降では，２章で YAP の基本原理について説明した後，３章で具体的解析法について述べる．次いで動作例を４章で述べ，５章で時間的空間的能率を考察することにする．

2. YAP の基本原理

YAP ではスタックを用いて解析を行うので，まず最初に，スタックに保持すべき情報について考察する．次に，パーサのアクションの種類について考察する．パーサは，文法規則の右辺を構成する構文要素に対応して，モジュール化されて構成される．そして，論理型言語によるパーサは後戻りなしに動作して，すべての解析木（もしあれば）を同時に求めることができる．

2.1 保持すべき情報

YAP では解析表の代りにマルチスタックを用いている．スタックは解析の進行に伴い，必要に応じて動的に生成されたり消去されたりする．各スタックは，入力列の先頭から現在までに走査した入力部分列に対して，互いに異なる可能な解析パスを表すために用いられる．

† YAP: Yet Another Efficient Parsing Method for General Context Free Grammars Based on the Logic Programing Language by TATSUYA HAYASHI (Fujitsu Laboratories Ltd.).

†† (株)富士通研究所

#		
1	s	—→ 0 np 1 vp
2	np	—→ 2 det 3 noun 4 relc
3	np	—→ 5 name
4	vp	—→ 6 vt 7 np
5	vp	—→ 8 vi
6	relc	—→ 9 [that] 10 vp
7	relc	—→ 11 ε
8	det	—→ 12 [every]
9	noun	—→ 13 [man]
10	name	—→ 14 [Mary]
11	vt	—→ 15 [loves]
12	vi	—→ 16 [lives]

図 1 文法例G1
Fig. 1 A context free grammar G1.

解析パスを表現する上で必要最少限の情報は，結論を先に言えば，文法規則上の復帰位置と解析の現在位置のみである．例えば，**図1**のような文法において，入力列「Every man that lives loves Mary」が与えられた場合，入力部分列「Every man that」に対する解析パスを考えよう．なお，文法規則の右辺の位置を一意に示すために，図のように，構文要素の直前にそれぞれ異なる番号を与えておく．

そうすれば今の場合，解析パスは（この場合1通りしかない）[⊢, 0, 4, 10] (*8, *9) で十分表現できる．ここで[ ]がスタックを表し（10が先頭要素，⊢は入力列の左端を示す記号），( ) はそれに付随する規則列である．スタックの内容は，規則 *1, *2, *6 が順次トップダウンに適用されて，現在*6の「10」の位置にいること，つまり「10」が現在位置であることを示す．また，*6の右辺のすべての要素が入力部分列と対応付けられると「10」は捨てられ，*6が規則列に格納される．そして解析過程は*2の「4」に戻る．つまり「4」が直接の復帰位置で，同様に「0」がその先の復帰位置である．

なお，*2の右辺要素 *det*（「2」）および *noun*（「3」）に関しては，既に入力部分列と対応付けられており，適用された*8, *9が規則列に格納されているわけである（規則列は最終的に right parse になる）．

その際例えば「2」，「3」，「9」のような位置情報は冗長なので，独立にはスタックに格納する必要はないのである．

こうして，Earley や他の LR 手法とは異なって，原則として位置情報は，適用してきた各規則について1個だけ保持すればよいわけである．

なお，上の例で，入力部分列が「Every man」の場合は，非終端記号 *relc* がε規則を持つため解析パス（スタック）は一時的に2個となり，それぞれ，[⊢, 0, 4, 9] (*8, *9), [⊢, 1] (*8, *9, *7) で表される．そして，次の「that」を走査した時点で後者は消滅するわけである．

2.2 アクションの種類

パーサの動作は，基本的には SAX と同様に文法規則右辺の要素ごとに，ホーン節として定義される．

しかし，BUP や SAX の場合は左隅解析法をとっているため，規則右辺の左端要素に対応するアクションではトップダウン予測が効率上必要になる．そのため，現在のゴール（目標構文要素）に，当該左端要素を含む文法規則の左辺（要素）が到達可能であるか否かを，実行時に link, tp-filter, tp-out などの述語を用いて，チェックしている．また，SAX は上記の理由から，右辺の構文要素（に対応するホーン節）を二つのタイプに分け，左端要素（に対応するホーン節）はタイプ1，それ以外はタイプ2としている．タイプ1では，トップダウン予測を動的に行うわけである．

YAP の場合，復帰位置を把握するという観点からは，構文要素をタイプ分けする必要はない．むしろ，その後の動作の方が規則上の位置によって異なるので，YAP ではこれに基づいてタイプを三つに分けている．第1のアクションタイプは左端要素に対応するもの，第2のタイプは規則上の中間に位置する要素に対応するもの，第3のタイプは右端要素に対応するものである．言うまでもなく，ある構文要素が複数個所に現れる場合は，全体のアクションは個別のアクションの集合になる．

さて，左端要素に対応するアクションはどうなるであろうか．この場合は解析パスが，直前まで適用していた規則のある位置から左端要素を含む規則へいったん分岐するので，前の規則上の位置（スタックの先頭にある）を復帰位置としてプッシュして，新しい規則の左端要素の右隣の位置を先頭に乗せればよい．例えば，図1において，*2 の *det* の場合には，スタックをプッシュして「3」を先頭に乗せることになる．

次に中間要素の場合は，解析パスは単に同一規則上を左から右へ1ステップ位置が進むだけであるから，スタックの先頭にある現在位置を，要素の左位置から右位置に更新するだけでよい．例えば，図1の*2の *noun* の場合は，スタックの先頭要素を「3」から「4」に置き換えればよい．

最後に右端要素の場合は，当該規則から導出された（と思われる）入力部分列が検出されたわけなので，解析木の生成を行い，それからスタックをポップして復帰位置を先頭に出して，規則の左辺要素に対応するアクションへ移ればよい．例えば，図1の*2の *relc* の場合は，スタックの先頭要素「4」をポップして，*np* に対応するアクションへ移ることになる．

ところで，これまでは説明を単純化していたが，スタックの要素は実は単一の位置番号ではなく，一般に1個以上の位置番号からなる集合（リストで表現）である．例えば，*4の *vt* の場合，スタックの先頭には復帰位置として*，[1, 6, 8, 15, 16] が格納されているはずなのである．*vt*（に対応するアクション）は，「6」がその中に含まれていることを調べた後左端タイプとしてのアクションをとる．この [1, 6, 8, 15, 16] は，*1の *np* によって実は作成されたのである．

np はスタックをプッシュする際に「1」だけでなく，「1」の直後が非終端要素 *vp* なのでさらに *vp* を左辺に持つ*4, *5を参照して「6」,「8」を含める．さらにまた，「6」,「8」の直後が非終端要素 *vt*, *vi* なのでそれらを左辺に持つ*11, *12を参照して「15」,「16」を集合に含めるのである（「6」,「8」,「15」,「16」を「1」から最左導出された位置と呼ぶことにする）．したがって，結局 [1, 6, 8, 15, 16] を現在位置として，*np* はスタックの先頭に格納したのである．

なお，*4の *vt* は，復帰位置を確認した後，*np* の場合と同様にしてスタックをプッシュし，先頭に [7, 2, 5, 12, 14] を格納する．

3. 具体的方法

ここでは，論理型言語 PROLOG による YAP の実現法について具体的に述べる**．最初のうちは簡単のため，解析木の作成部分は省略し，後で独立して取り上げることにする．また，表記を簡潔にするために，差分リストやカット記号の導入等効率上の配慮は割愛する．

3.1 タイプ別アクション

（1） 左端タイプの場合

構文要素を A とし，対応する述語を aL とすると，アクションは次のようなホーン節で表される．

```
aL([C|X], Y) :-aL1(C, D), (D≠[ ],
                Y=[[D, C|X]]; Y=[ ]).
aL1([ ], [ ]).
aL1([n|C1], [n1, ···, np|D]) :-aL1(C1, D)***.
                ⋮
aL1([_|C1], D) :-aL1(C1, D).
```

ここで aL の第1引数，第2引数はそれぞれ操作前，操作後のスタック（リスト形式）を示す．スタックは単純スタックとしマルチスタックの場合は後述する．n は A の左位置，n_1〜n_p は A の右位置またはその最左導出とする．

（2） 中間タイプの場合

構文要素 A に対応する述語を aC とすると，ホーン節は次のようになる．

```
aC([C|X], Y) :-aC1(C, D), (D≠[ ],
                Y=[[D|X]]; Y=[ ]).
aC1([ ], [ ]).
aC1([n|C1], [n1, ···, np|D]) :-aC1(C1, D)***.
                ⋮
aC1([_|C1], D) :-aC1(C1, D).
```

スタックの先頭を置換する以外は，（1）の処理と同様である．

（3） 右端タイプの場合

構文要素 A に対応する述語を aR とすると，ホーン節は次のようになる．

```
aR([C|X], Y) :-aR1(C, [C|X], Y).
aR1([ ], _, [ ]).
aR1([n|C1], [C|X], Y) :-b(X, Y1),
    aR1(C1, [C|X], Y2), append(Y1, Y2, Y)***.
                ⋮
aR1([_|C1], X, Y) :-aR1(C1, X, Y).
```

ここで n は A の左位置，b は当該規則の左辺要素 B に対応する述語とする．

（4） タイプ混在の場合

構文要素 A が複数のタイプを持つ場合は，次のホーン節を追加する必要がある．

```
a(X, Y) :-aL(X, Y1), aC(X, Y2),
    aR(X, Y3), append(Y1, Y2, Y4),
    append(Y4, Y3, Y).
```

（5） マルチスタックの場合

A が非終端記号の場合には，入力としてマルチスタックを直接受け取ることはないが，終端記号の場合には一般にマルチスタックとなる．そこで，次のホーン

---

* *11のように右辺が1要素からなる場合には，現在位置は同時に復帰位置でもある．

** 言語仕様上，スタックはこれまでの説明と逆向きに表現される点に注意．

*** n が A の唯一の左位置の場合には，右辺の再帰呼出し部分は不要である．

節を追加して，スタック単位に逐次的に処理を行う必要がある．

$a([\ ],[\ ])$.

$a([X|XX],Y)$:– $aS(X,Y1)$, $a(XX,Y2)$,
append $(Y1,Y2,Y)$.

3.2 左回帰規則を含む場合

例えば，$A \longrightarrow \alpha^k B^l \beta$,

$B \longrightarrow {}^m B^n \gamma$,

で，解析位置が「k」を含む場合，「k」からの最左導出で得られる位置（「m」など）は異なるものだけ集合に加えればよい．こうすれば，後は非回帰要素と全く同様の扱いでよい．すなわち，左回帰要素Bが認識された時には，解析パスはBの繰り返し回数に関係なく，

$[\cdots,[l,\cdots]]$,

$[\cdots,[k,m,\cdots],[n,\cdots]]$

の2通りとなる（$\alpha=\varepsilon$ の時は1通りだが問題はない）．そして，Bが繰り返される時は後者が，そうでない時は前者が残り，以下同様にしてBから導出される入力部分列を正しく検出することができる．

3.3 ε 規則を含む場合

例えば，$A \longrightarrow \alpha^k B^l \beta$, （$\alpha=\varepsilon$ でもよい），

$B \longrightarrow {}^m \varepsilon$,

で，解析位置が「k」を含む場合は，解析パスとして，

$[\cdots,[k,\cdots]]$ に加えて，

$[\cdots,[l,\cdots]]$

を追加する必要がある．このためには，3.1(1)，(2)で述べた左端タイプ，中間タイプのアクションを次のように修正すればよい．

$Y=[[D,C|X]]$ または $[[D|X]] \Longrightarrow$

$Y1=[[D,C|X]]$ または $[[D|X]](\equiv[Y1'])$,
$b(Y1',Y2)$, append $(Y1,Y2,Y)$.

3.4 1要素規則を含む場合

構文要素Aが，ある規則上で同時に左端でかつ右端の場合には，右端タイプとして扱い，3.1(3)を次のように修正すればよい（現在位置が同時に復帰位置でもある）．

$b(X,Y1) \Longrightarrow b([C|X],Y1)$

3.5 入力形式

パーサの動作は上述したように，モジュール化されたホーン節の集合として記述され，見かけ上構文要素対応で互いに独立している．

そこで YAP では，入力列 $w_1 w_2 \cdots w_n$ から次のようなホーン節を作成して，これを YAP への入力とし，モジュール化されたアクション間の必要な関連付けを行っている．

:–open (I_1), $w_1(I_1,I_2)$, $w_2(I_2,I_3)$,
$\cdots$, $w_n(I_n,I_{n+1})$, close (I_{n+1}).

ここで，open, close はそれぞれ前処理，後処理（後述）である．また I_i は，語 w_1 から w_{i-1} までの入力部分列の走査により得られた解析パス（マルチスタック）を示し，I_{i+1} は語 w_1 から w_i までの入力部分列に対する解析パスである．I_{i+1} は述語 w_i によって作成され，述語 w_{i+1} へ渡される．

3.6 前処理／後処理

YAP では，入力文の第1語の走査に入る前に，次のホーン節で示される前処理が行われる．

open $([[[\vdash,\cdots]]])$.

これは，初期解析パス（位置）として「$\vdash$」およびそれから最左導出される位置をスタックにあらかじめ格納するものである．ただし，与えられた文法は，$S_0 \rightarrow {}^l S^1 \dashv$ を第0規則として持つものと仮定する．上記のホーン節は入力節中の open (I_1) によって起動され，値が I_1 にユニファイされる．

また，YAP では，解析が無事終了したかどうか，あるいは生成された解析木の数や解析木そのものの出力などは後処理 close 述語を適宜用意して行う．

3.7 解析木の生成

入力列を走査しながら解析木を生成し，走査終了後に生成された解析木を出力するためには，解析パス用スタック1対のほかに，これと連動するスタック1対を用意する必要がある．

記述を簡潔にするために，ここでは 3.1 で示したホーン節の変更部分のみを以下に示すことにする．また，構文要素 A は最初のうち，非終端記号だとしておく．

（1） 左端／中間タイプの場合

$a*([C|X],Y,L,M)$:–
$a*1(C,D)$, $(D \neq [\ ], Y=[[D,C|X]]$
または $[[D|X]], M=[L]$; $Y=[\ ]$,
$M=[\ ])$.

ここで，L, M が解析木用のそれぞれ操作前，操作後のスタックである．また「$*$」は「L」または「C」を表す．

（2） 右端タイプの場合

$aR([C|X],Y,L,M)$:–$aR1(C,[C|X],Y,L,M)$.

$aR1([\ ],\_,[\ ],\_,[\ ])$.

$aR1([n|C1],[C|X],Y,[B_m,\cdots,B_1|L1],M)$:–
$b(X,Y1[b(B_1,\cdots,B_m)|L1],M1)$,
$aR1(C1,[C|X],Y2,[B_m,\cdots,B_1|L1]),M2)$,
append $(Y1,Y2,Y)$, append $(M1,M2,M)$.
$aR1([\_|C1],X,Y,L,M)$:–
$aR1(C1,X,Y,L,M)$.

ここで，$B_1,\cdots,B_m$ は当該構文規則の右辺で，言うまでもなく，$B_m=A$ である．

上記のように，スタック L, M にはトリーリストが格納される．

（3） 終端記号の場合

構文要素 A が終端記号の場合には，スタック L には A はまだ格納されていない．したがって，次のように修正する必要がある．

（a） 左端／中間タイプの場合
$M=[L]\Longrightarrow M=[[A|L]]$

（b） 右端タイプの場合
$aR1([n|C1],[C|X],$
$Y,[B_{m-1},\cdots,B_1|L1],M)$:–
$b(X,Y1,[b(B_1,\cdots,B_{m-1},A)|L1],M1)$,
$aR1(C1,[C|X],$
$Y2,[B_{m-1},\cdots,B_1|L1],M2)\cdots$.

4. 動 作 例

YAP の動作を具体例にもとづいて見てみよう．図2のような曖昧性を持った文法が与えられたとする．また入力列を「failing students looked hard」とする．YAP への入力は，

open ($I1$, $M1$), failing ($I1$, $I2$, $M1$, $M2$),
students ($I2$, $I3$, $M2$, $M3$), looked ($I3$, $I4$, $M3$, $M4$),
hard ($I4$, $I5$, $M4$, $M5$), close ($I5$, $M5$).

である．ただし，簡潔のため $M1$～$M5$ には，解析木

#			
1	s	→	0 np 1 vp
2	np	→	2 a 3 n
3	np	→	4 prp 5 n
4	vp	→	6 v 7 a
5	vp	→	8 v 9 av
6	a	→	10 [failing]
7	prp	→	11 [failing]
8	a	→	12 [hard]
9	av	→	13 [hard]
10	n	→	14 [students]
11	v	→	15 [looked]

図 2 曖昧性のある文法例G2
Fig. 2 An ambiguous contex free grammar G2.

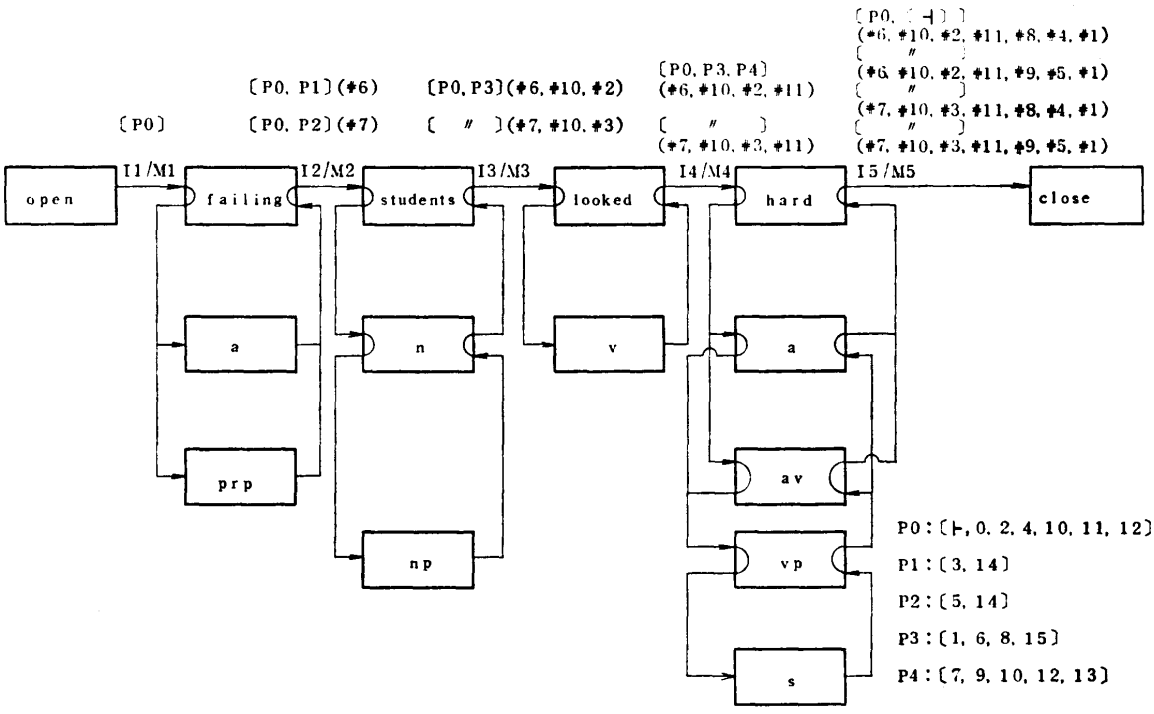

図 3 G2に基づく YAP の解析モデル
Fig. 3 YAP parsing model for the grammar G2.

ではなく right parse を格納するものとする．すると**図3**に示すように，まず最初に前処理 open が動き，初期解析パス [[⊢, 0, 2, 4, 10, 11, 12]] ([$P0$] とする) を作成し，Mへの初期値を空にして，$I1$, $M1$ を通じて failing にこれらの値を渡す．図において，四角は述語，実線は I/M 情報の転送路を示す．述語 a は簡潔にするため2個に分割してある．また，「⊣」は入力列の右端を示す記号である．failing は $P0$ の中に「10」が含まれていることを確認して，適用規則 *6 をスタックに格納し a を呼び出す．同時にまた failing は，$P0$ の中に「11」も含まれていることを確認する．そこで *7 を格納（別スタックになる）し prp を呼び出す．この時点で，I およびMの対 (I/M) は，

[$P0$] (*6), [$P0$] (*7)

の2通りになる．

次に，a, prp は，$P0$ にそれぞれ「2」,「4」が含まれていることを確認し，I/M をそれぞれ，

[$P0$, [3, 14]($\equiv P1$)] (*6),
[$P0$, [5, 14]($\equiv P2$)] (*7)

に変更する．この I/M は PROLOG のユニファイ機能により $I2$, $M2$ を通じて students に渡る．

students は，$P1$ および $P2$ に「14」が含まれているのを確認し，I/M を，

[$P0$, $P1$] (*6, *10),
[$P0$, $P2$] (*7, *10)

とする．そして n を呼び出す．

さて n は，$P1$ には「3」が，$P2$ には「5」が含まれていることを知って，I/M を，

[$P0$] (*6, *10, *2),
[$P0$] (*7, *10, *3)

として，np を呼び出す．np は $P0$ に「0」が含まれているので，I/M を，

[$P0$, [1, 6, 8, 15]($\equiv P3$)] (*6, *10, *2),
[$P0$, [1, 6, 8, 15]($\equiv P3$)] (*7, *10, *3)

とする．これが $I3$, $M3$ を通じて looked に渡る．以下同様にして，$I4$, $M4$ を通じて，

[$P0$, $P3$, [7, 9, 10, 12, 13]($\equiv P4$)] (*6, *10, *2, *11),
[$P0$, $P3$, [7, 9, 10, 12, 13]($\equiv P4$)] (*7, *10, *3, *11)

が hard に渡される．そして結局，$I5$, $M5$ を通じて，

[$P0$, [⊣]] (*6, *10, *2, *11, *8, *4, *1),
[$P0$, [⊣]] (*6, *10, *2, *11, *9, *5, *1),
[$P0$, [⊣]] (*7, *10, *3, *11, *8, *4, *1),
[$P0$, [⊣]] (*7, *10, *3, *11, *9, *5, *1)

が，後処理 close に渡される．したがって，close はこれら4通りの解析結果を出力すればよいわけである．

なお，図2の文法に対する YAP のプログラム（解析木用スタックは省略）を付録に示す．その際「⊢」,「⊣」の代りに「b」,「e」を用いている．

5. 時間的・空間的能率

YAP は元来，横型下降式の構文解析法を論理型言語で実現しようとする試みから生まれたものである．しかし言うまでもなく，その手法は任意のプログラム言語で実現することができる．

ここでは，YAP の時間的空間的能率について考察しよう．YAP は Earley の方法と基本的には等価なので，それとの対比において考えよう．

まず，入力列（長さ n）を i 語目まで走査した時点で作成される Earley の解析表を T_i とする．また YAP のマルチスタックの状態を MS_i，その中に含まれるスタックを S_k^i とする．

T_i の要素は $(A\rightarrow\alpha\cdot\beta, j)$ $(j=0\sim i)$ なので，T_i のサイズは $o(i)$ である．一方 S_k^i の要素は $(A\rightarrow\alpha\cdot\beta)$ の集合に相当するので，S_k^i の各要素のサイズは $o(C)$ すなわち定数で抑えられる．

次に，YAP の左端型や中間型のアクションは，Earley の scanner および predictor と同様に，所要時間は $o(C)$ である[9]．一方，右端型のアクションは completer に相当するが，S_k^i の要素のサイズが $o(C)$ なので右端型の所要時間も $o(C)$ である（ただしここでは completer に倣って，適用規則の左辺に対応する述語呼出しまでを1アクションと考える）．S_k^i の深さは $o(i)$ なので結局 S_k^i 作成の時間的能率は $o(i)$ となる．これに対して，completer の所要時間は曖昧な文法の場合には $o(i)$ となり，したがって T_i の作成に要する時間は $o(i^2)$ となる[9),10)]．これは T_i の各要素が解析パス数の増大により $o(i)$ の回数重複登録される可能性があり，それを避けるためのチェックが必要になるからである．しかし，曖昧性を含まない文法の場合には重複登録の可能性はない[9),10)]．つまり，その時の解析パス数は $o(i)$ である．

YAP では MS_i が解析パスを陽に分離して保持しているので，曖昧でない場合 S_k^i の数は $o(i)$ になる．したがって結局，全体の時間的能率は $o(n^2)$ となる．

次に所要記憶量であるが，S_k^i の深さは $o(i)$ であり，したがって S_k^i の記憶量も $o(i)$ である．そこで結局，曖昧でない場合全体の空間的能率も $o(n^2)$ とな

る．なお，解析パスの規則列への規則番号の登録や解析木の生成は，原則として S_k^i のポップ動作と対応するので，このための所要時間および所要記憶量は i の時点で累計 $o(i)$ である．したがって最終的に $o(n)$ となり，全体の能率には影響を与えない（右辺の長さが1の規則の場合は S_k^i のポップ動作を伴わないが，そのような規則の適用回数は S_k^i の各要素に対して高々定数で抑えられる）．

Earley の方法は上述したように，与えられた入力が文か非文かを識別することを主目的としているために解析パスを分離していない．これに対して YAP は，入力文のすべての解析木を求めることを目的にしている．そのため，CFG 一般に対して，Earley の方法では $o(n^3)$（文／非文の識別）であるが，本方式ではすべての解析木や right parse の生成に $o(n^3)$ を越える場合があり得る．ただし，上述したことから明らかなように，曖昧な文法の場合でも解析パス数が $o(i)$ の時は能率は $o(n^2)$ である．また，文法の曖昧性に関係なく解析パス数が $o(C)$（$LR(0)$ や先読み[9)~11)]を行えば $LR(k)$ もこれに該当）の時は能率は $o(n)$ となる．

6. あとがき

本論文では，論理型言語による横型下降方式の構文解析法 YAP について提案した．また YAP で用いた方式は，基本的には Earley の手法と等価で，これに若干の改良を加えたものであり，同じ良好な能率を有することを示した．一方，論理型言語においては，適切な引数や述語の導入により，意味処理と構文処理の整合性を高めることが可能である．そこで今後は，YAP の性能評価，言語の外置現象に対応した拡張 CFG とその解析法，意味処理と構文処理を統合した自然言語処理のモデル化等の検討を進め，順次報告する予定である．

謝辞 本研究の機会を与えていただいた富士通研究所常務山田 博氏に感謝する．また，本論文をまとめるに当りいろいろと手伝ってくれた富士通研究所情報処理研究部門小部正人，小野越夫，泉田義男，杉山健司，小野美由紀，上原三八および ICOT の田中裕一の諸氏に厚くお礼を申し上げる．

参考文献

1) Pereira, F. C. N. and Warren, D. H.: Definite Clause Grammars for Larguage Analysis—A Survey of the Formalism and a Comparison with Augmented Transition Networks, *Artif. Intell.*, Vol. 13, pp. 231-278 (1980).
2) Matsumoto, Y., Tanaka, H., Hirakawa, H., Miyoshi, H. and Yasukawa, H.: BUP: A Bottom-Up Parser Embedded in PROLOG, *New Generation Computing*, Vol. 1, No. 2, pp. 145-158 (1983).
3) Matsumoto, Y., Kiyono, M. and Tanaka, H.: Facilities of the BUP Parsing System, *Proc. Natural Language Understanding and Logic Programming* (1984).
4) Matsumoto, Y.: Parallel Syntax Analysis, SIGNLP, *IPS Japan*, Vol. 86, No. 6, p. 8 (1986).
5) Matsumoto, Y. and Sugimura, R.: SAX: A Parsing System Based on Logic Programming Languages: *Computer Software*, Vol. 3, No. 4, pp. 4-11 (1986) (Japanese).
6) Knuth, D. E.: On the Translation of Languages from Left to Right, *Information and Control*, Vol. 8, pp. 607-639 (1965).
7) Hayashi, T.: On the Construction of *LR(k)* Analyzers, *Proc. ACM 71*, pp. 538-553 (1971).
8) Hayashi, T.: On the Translation from CFG to Production Language, *Trans. IPS Japan*, Vol. 12, No. 3, pp. 145-153 (1971) (Japanese).
9) Earley, J.: An Efficient Context-Free Parsing Algorithm, *CACM*, Vol. 13, No. 2, pp. 94-102 (1970).
10) Aho, A. V. and Ullman, J. D.: *The Theory of Parsing, Translation, and Compiling*, Prentice-Hall, Englewood Cliffs (1972).
11) Yoshida, S. et al.: Parsing Methods for Machine Translation, *Trans. IPS Japan*, Vol. 26, No. 10, pp. 1157-1164 (1985) (Japanese).

林 達也（正会員）

1937年生．1960年早稲田大学第一理工学部応用物理学科卒業．同年富士通(株)入社．以来，ソフトウェア工学(コンパイラ自動作成，設計支援，部品化)，データベース(リレーショナル，分散，マルチメディア)，自然言語処理(機械翻訳，自然言語インタフェイス，情報検索)，人工知能（エキスパートシェル，知識表現）等の研究開発に従事．富士通研究所情報処理研究部長代，ソフトウェア研究部長を経て現在情報処理研究部門所属．元本学会編集幹事．1973年度本学会論文賞受賞．ACM，日本ソフトウェア科学会，日本モーツァルト協会各会員．AAI (Applied Artificial Intelligence) (Hemisphere) Editorial Board メンバ．

付録 文法 G2 に対する YAP 解析プログラム

Appendix YAP implementation for the grammer G2

```
a(X,Y) :-
      aL(X,Y1), aR(X,Y2), append(Y1,Y2,Y) .

aL([C|X],Y) :-
      aL1(C,D),
        (D=/=[], Y=[[D,C|X]];
         Y=[]) .

aL1([],[]) .
aL1([2|C1],[3,14]) .
aL1([_|C1],D) :-
      aL1(C1,D) .

aR([C|X],Y) :-
      aR1(C,[C|X],Y) .

aR1([],_,[]) .
aR1([7|C1],[C|X],Y) :-
      vp(X,Y) .
aR1([_|C1],X,Y) :-
      aR1(C1,X,Y) .

prp([C|X],Y) :-
      prpL1(C,D),
        (D=/=[], Y=[[D,C|X]];
         Y=[]) .

prpL1([],[]) .
prpL1([4|C1],[5,14]) .
prpL1([_|C1],D) :-
      prpL1(C1,D) .

n([C|X],Y) :-
      nR1(C,[C|X],Y) .

nR1([],_,[]) .
nR1([3|C1],[C|X],Y) :-
      np(X,Y1), nR1(C1,[C|X],Y2), append(Y1,Y2,Y) .
nR1([5|C1],[C|X],Y) :-
      np(X,Y1), nR1(C1,[C|X],Y2), append(Y1,Y2,Y) .
nR1([_|C1],X,Y) :-
      nR1(C1,X,Y) .

np([C|X],Y) :-
      npL1(C,D),
        (D=/=[], Y=[[D,C|X]];
         Y=[]) .

npL1([],[]) .
npL1([0|C1],[1,6,8,15]) .
npL1([_|C1],D) :-
      npL1(C1,D) .

v([C|X],Y) :-
      vL1(C,D),
        (D=/=[], Y=[[D,C|X]];
         Y=[]) .

vL1([],[]) .
vL1([6|C1],[7,10,12|D]) :-
      vL1(C1,D) .
vL1([8|C1],[9,13|D]) :-
      vL1(C1,D) .
vL1([_|C1],D) :-
      vL1(C1,D) .

av([C|X],Y) :-
      avR1(C,[C|X],Y) .

avR1([],_,[]) .
avR1([9|C1],[C|X],Y) :-
      vp(X,Y) .
avR1([_|C1],X,Y) :-
      avR1(C1,X,Y) .

vp([C|X],Y) :-
      vpR1(C,[C|X],Y) .

vpR1([],_,[]) .
vpR1([1|C1],[C|X],Y) :-
      s(X,Y) .
vpR1([_|C1],X,Y) :-
      vpR1(C1,X,Y) .

s([C|X],Y) :-
      sL1(C,D),
        (D=/=[], Y=[[D,C|X]];
         Y=[]) .

sL1([],[]) .
sL1([b|C1],[e]) .
sL1([_|C1],D) :-
      sL1(C1,D) .

fS([C|X],Y) :-
      fR1(C,[C|X],Y) .

fR1([],_,[]) .
fR1([10|C1],X,Y) :-
      a(X,Y1), fR1(C1,X,Y2), append(Y1,Y2,Y) .
fR1([11|C1],X,Y) :-
      prp(X,Y1), fR1(C1,X,Y2), append(Y1,Y2,Y) .
fR1([_|C1],X,Y) :-
      fR1(C1,X,Y) .

stS([C|X],Y) :-
      stR1(C,[C|X],Y) .

stR1([],_,[]) .
stR1([14|C1],X,Y) :-
      n(X,Y) .
stR1([_|C1],X,Y) :-
      stR1(C1,X,Y) .

lS([C|X],Y) :-
      lR1(C,[C|X],Y) .

lR1([],_,[]) .
lR1([15|C1],X,Y) :-
      v(X,Y) .
lR1([_|C1],X,Y) :-
      lR1(C1,X,Y) .

hS([C|X],Y) :-
      hR1(C,[C|X],Y) .

hR1([],_,[]) .
hR1([12|C1],X,Y) :-
      a(X,Y1), hR1(C1,X,Y2), append(Y1,Y2,Y) .
hR1([13|C1],X,Y) :-
      av(X,Y1), hR1(C1,X,Y2), append(Y1,Y2,Y) .
hR1([_|C1],X,Y) :-
      hR1(C1,X,Y) .

failing([],[]) .
failing([X|XX],Y) :-
      fS(X,Y1), failing(XX,Y2), append(Y1,Y2,Y) .

students([],[]) .
students([X|XX],Y) :-
      stS(X,Y1), students(XX,Y2), append(Y1,Y2,Y) .

looked([],[]) .
looked([X|XX],Y) :-
      lS(X,Y1), looked(XX,Y2), append(Y1,Y2,Y) .

hard([],[]) .
hard([X|XX],Y) :-
      hS(X,Y1), hard(XX,Y2), append(Y1,Y2,Y) .

open([[[b,0,2,4,10,11,12]]]) .

close(I) :-
      write(I), nl .

append([],Y,Y) .
append([X|T],Y,[X|TT]) :-
      append(T,Y,TT) .
```

(昭和 62 年 4 月 27 日受付)
(昭和 63 年 7 月 15 日採録)