ボトムアップ構文解析システムBUPの拡張と日本語文法の試作

田中穂積，小山晴生，奥村学
（東京工業大学・工学部）

1. はじめに

Prologは、一階述語論理を基礎とし、パターン・マッチングとバックトラックの機能を持っているために、構文解析とは相性がよい。DCG (Definite Clause Grammar) [Pereira80]で記述された文法をほぼ相似で1対1に対応するPrologプログラムに変換して実行する手法が、エジンバラ大学で開発されている。この変換されたプログラムを実行することは、トップダウン，デプスファーストに構文解析を進めることと等価であることが知られている。ところが、この手法を用いるとトップダウンに構文解析が進むため、左再帰的文法規則を扱えない問題がある。そこで、DCGで記述された文法規則に対応するProlog節を生成し、それらの実行によりボトムアップ，デプスファーストに構文解析する方法が電子技術総合研究所で開発された。これをBUPと呼ぶ。[松本他83]

ところで、日本語処理を行なう場合には自動分かち書きの機能は不可欠である。しかし、これまでのBUPシステムではこの点が考慮されていなかった。そこで本論文では、辞書に記述されていない未定義語を含む文の自動分かち書きの機能を付加するため、3つの方式を検討し、最良と思われる方式をBUPシステムに組み入れた。

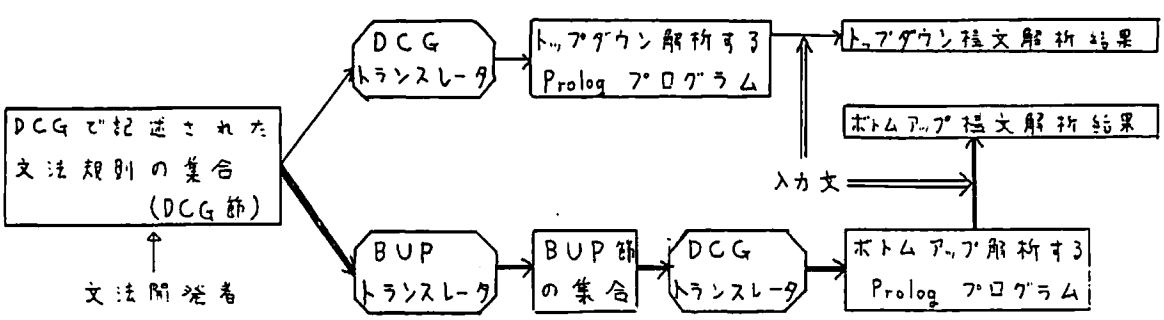
またこれまでのBUPシステムに組み込まれていた高速バージョンの、辞書引きアルゴリズム［松本，清野，田中'83］が、分かち書き、あるいは誤り処理を行なう場合には問題があることを指摘し、その解決策を検討する。

さらに、構文解析木の数を大幅に減らすことができる右方枝分かれの木構造を得る日本語文法規則を開発した。この場合、一つの構文解析木から複数個の意味解析結果を得る機構が必要になるが、Prologの非決定的プログラム技法を用いると、容易に実現可能であることが示される。

2. 自動分かち書きと構文解析

日本語文は通常分かち書きはなされていない。特に単語ごとの分かち書きは人間が行なっても一定に行なえないことが多い。また、動詞句の中の相を表わす部分など（例：行かせられたくなかったようだ）を基本要素に分解するには、いずれにせよ自動分かち書きを行なわざるうえない。

また現在の自然言語処理システムは、基本的にはChomsky以後の、rule（変形規則，文法規則）の集合を使って記述された形式的言語の理



論を基礎にしている。すなわち、これらは人間の言語能力における言語、完全な言語をその主な処理対象としている。しかし、この完全な言語の体系がいかなるものであるかは未解明であり、言語学の大きな問題となっている。しかし、現実にシステムで解析される文は、実際の言語運用における言語（自然言語）であるため、ここに多くの問題が発生する。

以上二点から、今回BUPシステムにある種の未定義語を含む文の自動分かち書きの機能を付け加える実験を3種行なった。

2.1 未定義語と文法規則

一般に未定義語と言っても多くの種類がある。タイプミスによるもの。固有名詞，専門用語，略語，新しい概念を表す造語。擬音語，擬態語。"半径R"の"R"のようなラベル，あるいは数字，数式等の特殊記号列。このようなものは辞書に記述されているとは考えにくいものが多く、未定義語として現れることが多いと考えられる。ここでは、例として数字とラベルを取り上げることにする。

ここでは、自動分かち書きを行なった後、あるいは分かち書きを行ないながら、構文解析を行なうことを考えている。したがって、未定義語に何らかのカテゴリを与え、その役割を文法規則に表現しておくことにする。すなわち、辞書的には未定義であっても、文法規則的には定義された状態にしておく。

さて、数表現は日本語においては次のような形態[*1]をとることが多いと考えられている。〔片佐原80〕

数表現 → (前置助数詞), ：約十年など
　　数,
　　(概数表示), ：十数年など
　　(単位), ：kg，冊など
　　(接度名詞). ：以上など

また、この数表現は多くの場合、量と関連する名詞と共に現れる。その代表的な例は次の2つと考える。

np → n, number型：震度3など
np → number型：手術が5cmなど

また、ラベルについても数表現と同じ位置に現れることが多い。また、この場合、2番目の形で固有名詞などの多くが吸収可能であると考えられる。

以上から次の文法規則を組み込んで実験を行なった。

np → (n), unknown.
unknown → label, (unit).
unknown → number, (unit).

2.2 自動分かち書きのアルゴリズム

(a) 辞書のみの情報による方法

入力された文字列を、parserにわたす前にリストに分解し分かち書きを行なう。この場合、最長一致法と字種の区切れ情報を使う方法がよく知られている。ここでは数字については字種情報を使った。ただし、"円O_1と円O_2"の"O_1"などとなる可能性は残した。その他の定義語については最長一致法を使用した。これを使用すると失敗する例が有りうるが、使用する情報が辞書のみの場合にはこの種のHeuristicsにたよらざるうえないようである。未定義語に関しては最も短かい切り出しを原則とした。さらに、未定義語の連続は禁止した。これによって"円$\underline{O_1}$"を"円O1"と切り出すことを防いだ。

〔脚注〕

*1 規則の記法についてはDCGを使用するが、本文中において省略可は（）で表す。すなわち、
np → (n), unknown. は正しくは、
np → (n;[]), unknown. である。

(b) 構文規則と最長一致法の併用による方法

(a)の方法の弱点の多くは辞書以外の情報を使っていないことに起因している。現在、我々のBUPシステム上の文法体系で利用できるのは構文情報までであるため、次にこれを使用することを考える。

BUPシステムによって変換されたプログラムで実際にパージングを行なう場合、入力された文は、goal節中の辞書引きによって左から順次カテゴリが決定されていく。そこで、この過程に分かち書きの機能を組み込むことによって、構文解析と同時に分かち書きを行なう。文法規則の適用とバックトラックに合わせて語の切り出しも同時進行し、構文解析終了時に分かち書きも終了する。

文法規則によって未定義語の連続が禁止されているので、(a)で用いたこのHeuristicsは必要ない。

(c) 最長一致法を使用しない方法

(b)の方法の弱点は最長一致法を使用したことによるものがめだつ。それは、これを使用したために可能な解釈の一部を捨ててしまうこと、及び、複雑なリストの操作を行なわなければならないことである。最長一致法では、あるリストで失敗した場合には、そのリストの末尾から一要素を取り出して次の解析を行なうことになる。

例：[h,a,n,a,w,o,……] … fail
[h,a,n,a,w,o] … fail
[h,a,n,a,w] … fail
[h,a,n,a] … success
(花あるいは鼻)

ところが、Prologのリスト処理は、先頭に要素を付け加えたり、取りはずしたりすることは容易であるが、末尾でこれを行なおうとすると効率が落ちる。差分リスト操作によってこれをさける方法もある[*2]が、副作用があるため使用に制限が加わる。

どちらにしても最長一致法を使用しなければ問題はなくなる。そしてこれは構文情報が使用できるため、十分に可能である。

ただし、未定義語部分の切り出しに、最短切り出しというHeuristicsを用いることは、必要なようである。これを除くと、後に続く定義語も含めた大きな未定義語を切り出して解析することがある。

例：[h,a,n,k,e,i][r,n,o,e,n][n,o]

羊	飼	八の	円	の
名詞		未定義語		助詞

2.3 結果と考察

以上3つにおいて、(a)の辞書のみによる方法はあまり良い方法とは言えない。(b)については、リスト操作が複雑なため(c)に比べて、よくても1割は遅くなるようである。(c)は最長一致法を使用せず、Prologのパターン・マッチングに任されるため、(a),(b)では出てこない解釈も可能でかつ速い。以上から(c)が最もすぐれていると考えられる。

この特徴は以下の通りである。

- 辞書の情報だけでなく、構文情報も利用している。
- 数字については字種情報を利用している。
- 定義語はPrologのパターン・マッチングによって切り出されるため可能な解釈のすべてが、語の長さにかかわらずに得られる。
- 入力文字列中の空白、句読点など明らかな区切り情報を利用できる。

また、未定義語に関して、

- 最短切り出しを原則とする。
- 未定義語の切り出しが起こるのは、文法規則からその存在が予測された時だけである。
- 定義語で木が作れる場合にはそれを優先する。

最後に、今回のプログラムの弱点は、未定義語の先頭が定義語のスペルと同じになっている場合に切り出せないことであるのを指摘しておく。このような場合、一度解析が完全に失敗するまでは、これを検出することはできないように思われる。

以下に成功例をあげておく。

```
| ?: ame10kowokatte3kowoutta.

400  msec.
No. 1

   sentence
    |-pp
    |   |-np
    |   |   |-n -- ame
    |   |   |-label
    |   |       |-numb -- 10
    |   |       |-unit -- ko
    |   |-p
    |       |-kp -- wo
    |-sentence
        |-vpt
        |   |-vp
        |   |   |-v -- katte
        |-sentence
            |-pp
            |   |-np
            |   |   |-label
            |   |       |-numb -- 3
            |   |       |-unit -- ko
            |   |-p
            |       |-kp -- wo
            |-sentence
                |-vp
                    |-v -- utta
```

(これは3.で述べる高速化アルゴリズムは使用していない例である。)

3. context を考慮に入れた辞書引きの高速化

BUP には、解析が進むにつれて行なわれる辞書引きの結果を残しておき、再計算を防ぐ高速化がなされている。ところがこのアルゴリズムは、2.で述べた自動分かち書きを行なう場合や、英熟語などの処理を行なう場合には問題がある。

3.1 従来の高速化辞書引きアルゴリズムの問題

問題のプログラムの概略を次に示す。

```
dictionary(C,X,Y):-
    wf_dict(_,X,_),!,          (1-a)
    wf_dict(C,X,Y).            (1-b)
dictionary(C,X,Y):-
    dict(C,X,Y),               (2-a)
    create_dlist(X,Y,U,V),     (2-b)
    assert(wf_dict(C,U,V)),    (2-c)
    fail.                      (2-d)
dictionary(C,X,Y):-
    wf_dict(C,X,Y).            (3)
```

dictionary という述語の呼び出しが辞書引きに相当する。X,Y は、入力文字列の差分リスト表現であり、C は辞書引きの結果得られるカテゴリである。

辞書引きの結果は、具体的には、wf_dict と呼ばれる assertion として assert されている。(1-a) でそれがあるかどうか調べる。あれば、(1-b) でそれによって結果を返す。カットシンボルがあるため、その他の可能性は追わない。もしない場合は辞書本体を調べる(2-a)。さらに、X,Y を、切り出し語のみの情報にして(2-b)、U,V[*3] とし、これらを assert する(2-c)。以下では、これを context なしの wf_dict と呼ぶ。さらに、(2-d) で fail がかかることによって、ここの過程が繰り返され、X にユニファイ可能なすべての辞書引き結果が wf_dict に assert される。その後、(3)によって結果が返される。

(脚注)

*2 bit 1983.5 bitplayer
「リストと差分リスト」上田和紀 参照

*3 例: この石は重い.
X = [石,は,重,い]
Y = [は,重,い]
U = [石|R]
V = R
また、以下では X を context と呼ぶ。

wf_dictは解析中に存在する短期記憶(STM)と考えている。それに対して、dictは辞書本体で、現在はメインメモリ上に存在しているが、大規模化した場合には2次記憶に移り、アクセスに時間がかかるようになると想定されている。

以下に例を示す。例文は、

"花子は花を持つ"　(花子も辞書に存在するとする。)

最初の辞書引きではSTMには何もないので、辞書本体を読んで、STMは次のようになる。

〔STM〕
- 花子
- 花

このうち、花は文法的にいずれ失敗するので、やがて花子が選択され、残りのリストは、"は花を持つ"になる。次も、STMにはユニフィケーション可能なものはないので、同様に、

〔STM〕
- は
- 花子
- 花

リストは、"花を持つ"になる。ここでは、すでに"花"がSTMに存在するため、辞書本体を調べに行く手間は省かれてすぐに結果を返す。リストは"を持つ"になる。

以下同様に解析がすすむが、これはバックトラックによって再度辞書引きが行なわれた場合にも辞書引きの手間を省き高速化がはかられる。

しかし例文が"花を花子は持つ"であると以上のアルゴリズムは不完全になる。最初に、

〔STM〕
- 花

次に、

〔STM〕
- を
- 花

ここで、リストは"花子は持つ"であるから、"花"とユニフィケーション可能であるため、あらためて辞書本体を読むことはなり。そのために"花子"が辞書にあってもこれは読み出されない。同じことは、

I take a bus from that bus stop.

のような英語語においても生じる。

3.2 完全な高速化辞書引きアルゴリズム

(a) アルゴリズム1

3.1で指摘した問題は、文中での単語の現れる位置の違いを(create_dictにより)無視した結果生じたものである。したがって、文中での単語の現れる位置をも含めたwf_dict(以下ではこれをcontext付きのwf_dictと呼ぶ)をassertすれば完全性は保たれるようになる。

この方法の欠点は、assertする情報が多くなることである。しかも、同一の切り出し語が異なる場所にあらわれれば別のwf_dictとしてassertされることになる。

(b) アルゴリズム2

(a)で述べた方法の欠点は次のようにして解決できる。例文は、

"花を花子が持つ"

とする。

wf_dictとしてassertする情報は、3.1で説明したcontextなしのwf_dictである。他にその場所で辞書引きが行なわれたことを示すためのcontextもがassertされる。

最初の辞書引きで

〔STM〕(wf_dict)
- 花

及び、

〔STM〕(context)
- 花を花子は持つ

がassertされる。

リストは、"を花子が持つ"になる。

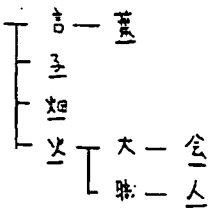
次に、

〔STM〕(wf_dict)
- を
- 花

(context)
- を花子は持つ
- 花を花子は持つ

ここでリストは、"花子は持つ" であるから "花" とユニフィケーション可能であるが、contextをみれば、初めての場所での辞書引きであることがわかる。したがって、ユニフィケーション可能なもののうち、最もスペルの長いもの（この場合は"花" しかない）よりさらに長い単語が辞書に隠れている可能性がある。

このような辞書引きを可能にするためここでは次のような辞書構造を採用する。（5章参照）



この辞書では、スペルの短かい単語から先に読み出される。したがって、これを次々と上側にassert（すなわちasserta）していけば、最も長い単語が上にくる。したがってSTM内でそのような単語を見つけ出すのは容易である。次にそれを使って、そこから先に辞書を読み進むことができる。

この辞書構造により、"花" で始まる単語のみの辞書引きが可能になる。

（STM）（wf_dict）
- 花子
- を
- 花

（context）
- 花子は持つ
- を花子は持つ
- 花を花子に持つ

以下同様に、解析が進められる。

"花子が花を持つ" の例では、contextなしのwf_dictによっているため以前と同じ性能が得られるようになっているが、しかしながら、単語ごとの辞書本体へのアクセス回数は減少したがcontextごとのアクセス回数は減っていない。

(c) アルゴリズム2の拡張

上記の辞書はtrie構造をしているので、終端がある。一度終端まで読み切ってしまえば、その系統の単語についてはcontextが詳しくなっても再び読みに行くのは無意味である。

そこで、wf_dictに終端か、非終端かを示すIDを付加する。これによって辞書本体へのアクセス回数が減少する。

3.3 結果と考察

(a)は、assertする切り出し語の後側部分が大きいため、(b)に比較して少くとも1割は速度が落ちるようである。使用するメモリも大きい。また冗長な部分が存在する。その点、(b)は何も高速化せず、ただ辞書本体を引きに行く場合と比較して、例文によってかなり差があるが、3割ぐらい速くなるようである。ただし、木を2本、3本と出していくとやがて、4本目では(a)と速度が変わらなくなる。(c)は、現状では、辞書がメイン・メモリ上に存在するため、(b)と大差はない。場合によっては、少し遅くなる。しかしながら、辞書が2次記憶に移った場合には、能力を発揮すると考えられる。

実行結果（単位はmsec）

例文
hanakotohanawotarouhamotu.
花子と（花鼻）を太郎は持つ.

	(a)	(b)	(c)
第1の木	769	431	423
第2の木	432	371	351
第3の木	772	541	536
第4の木	121	105	104
Total time	2285	1872	1835

4. 左方枝分かれ構造と右方枝分かれ構造〔井佐原，田中 81〕

日本語の構文解析では、埋め込み文をどのような構造の構文解析木とするかが問題である。多くの文法学者はこの埋め込みまを左方枝分かれ構造であると述べている。この左方枝分かれ構造は、人間の文理解の過程とうまく対応していることもあり、従来の文法においては自然なものとして受け入れられている。それでは、このような構造を得るための文法が、計算機処理の立場からも望ましいと言えるだろうか。これについて考察する。

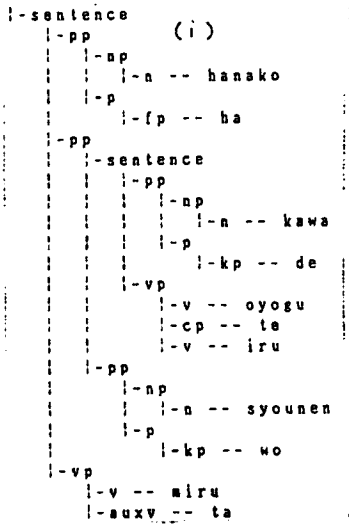
左方枝分かれ構造を得るための文法規則：

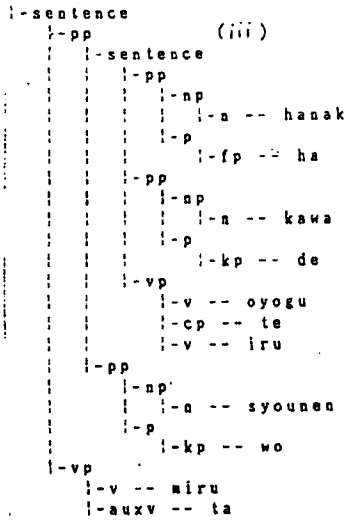
$$\begin{cases} S \longrightarrow PP^{*},\ VP. \\ PP \longrightarrow S,\ PP. \end{cases}$$

と右方枝分かれ構造を得るための文法規則：

$$\begin{cases} S \longrightarrow PP,\ S. \\ S \longrightarrow VP. \\ PP \longrightarrow VP,\ PP. \end{cases}$$

のそれぞれを用いて実際に構文解析した結果を表1，図1に示す。表1のように、左方枝分かれ構造を得るための文法規則では、文が長くなり、埋め込まれる文の数が多くなればなるほど、得られる構文解析木の数は指数関数的に増加する。構文解析木の数がこのよ





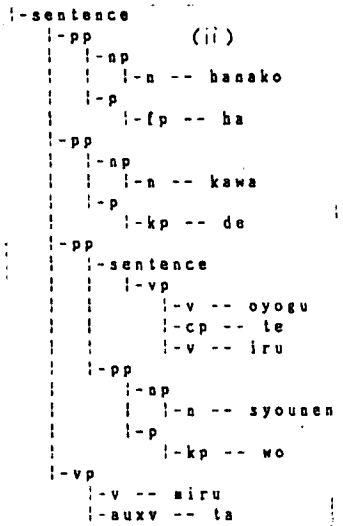
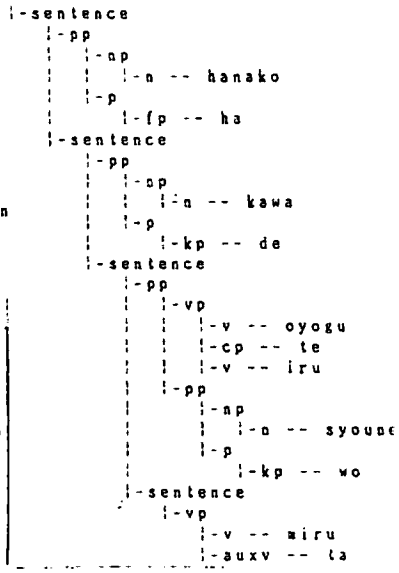


図1 (a)
左方枝分かれ構造の構文解析木

図1 (b)
右方枝分かれ構造の構文解析木



例文	単語数	左方枝分かれ構造		右方枝分かれ構造	
		構文解析木の数	構文解析に要した時間	構文解析木の数	構文解析に要した時間
花子は川で泳いだ。	6	1	499 msec	1	356 msec
花子は川で泳いでいる少年を見た。	10	3	1122 msec	1	750 msec
花子は池に落ちた犬を助けた少年にアメを上げた。	16	12	4764 msec	1	1614 msec
花子は池に落ちた犬を助けた少年に上げるアメを買いに行った。	19	55	23268 msec	1	2010 msec

表1　得られる構文解析木の数の比較

うに増加すると、構文解析に要する時間が増加するだけでなく、構文解析が終了した後で意味解析を行なう場合には、得られた構文解析木すべてに対して意味解析を施すわけであるから、効率が悪くなる。通常このような点が問題にならないのは、人間がその意味を理解しつつ、構文解析を行なっているからである。従って計算機の場合にも、意味解析・構文解析を並列に実行することが考えられるが、解析の初期には意味情報が不十分であることから、この手法では十分な効果が得られにくいと思われる。

一方、右方枝分かれ構造を得るための文法規則では、表1のような埋め込み文に関しては、得られる構文解析木はただ1つである[*4]。

4.1. 右方枝分かれ構造の構文解析木

埋め込み文には、意味的な曖昧さが含まれている場合がある。例えば、「花子は川で泳いでいる少年を見た。」という文では、「川で」は「泳いでいる」、「見た」の両方に係りうる。図1(a)がこの例文の構文解析結果であるが、このうち「川で泳いでいる」という関係を表わす(i)と、「川で見た」という関係を表わす(ii)とが、意味的に妥当な構造と言える。

このように右方枝分かれ構造の構文解析木群では、埋め込み文が含みうるすべての意味を、1つ1つの木が表わしているわけである。従ってこのような文法規則によって得られた個々の構文解析木には、意味的な曖昧さはないと一般に考えられている[*4]。その代り、複数の構文解析木が得られるという点に意味的な曖昧さが吸収される。

4.2. 右方枝分かれ構造の構文解析とその意味解析手法

図1(b)が右方枝分かれ構造の構文解析木の例である。右方枝分かれ構造の構文解析木の特徴は、非終端記号 sentence を根とする部分木がすべて、完成した1つの文の句構造を表わしているということである。例えば、図1(b)の、例文「花子は川で泳いでいる少年を見た。」に対する構文解析木では、

見た
泳いでいる少年を見た
川で泳いでいる少年を見た
花子は川で泳いでいる少年を見た

がそれぞれ非終端記号 sentence によって配されている。これは、この構文解析木の構造が人間の言語的直観から見ても、さほど奇異でないことを示している。

右方枝分かれ構造の場合、複数の構文解析木によって意味的な曖昧さは吸収されていたが、右方枝分かれ構造の場合には、基本的には構文解析木を1つに絞り込むことができるので、構文解析木の相違によって曖昧さは吸収できない。代りに、右方枝分かれ構造の場合には、その1つの構文解析木が、右方枝分かれ構造の複数の構文解析木の持つ意味構造を implicit に内包して、曖昧さを吸収しているのである。

それでは、複数の意味構造を implicit に内包したこの構造の構文解析木上で、どのように意味解析を実行すればよいであろうか。以下でそれを述べることにする。

すでに述べたように、この構造の構文解析木の特徴は、非終端記号 sentence を根とする部分木の各々が、完成した1つの文の句構造を表わしていることであった。この特徴を最も活かした意味解析の方法としては、図2に示したような、構文解析木の右端の小さな

(脚注) *4 しかし、句「花子を紹介した太郎」における「太郎」(紹介した人間, 紹介された人間のどちらの解釈が成り立つ)のような例は、実際には多数存在する。

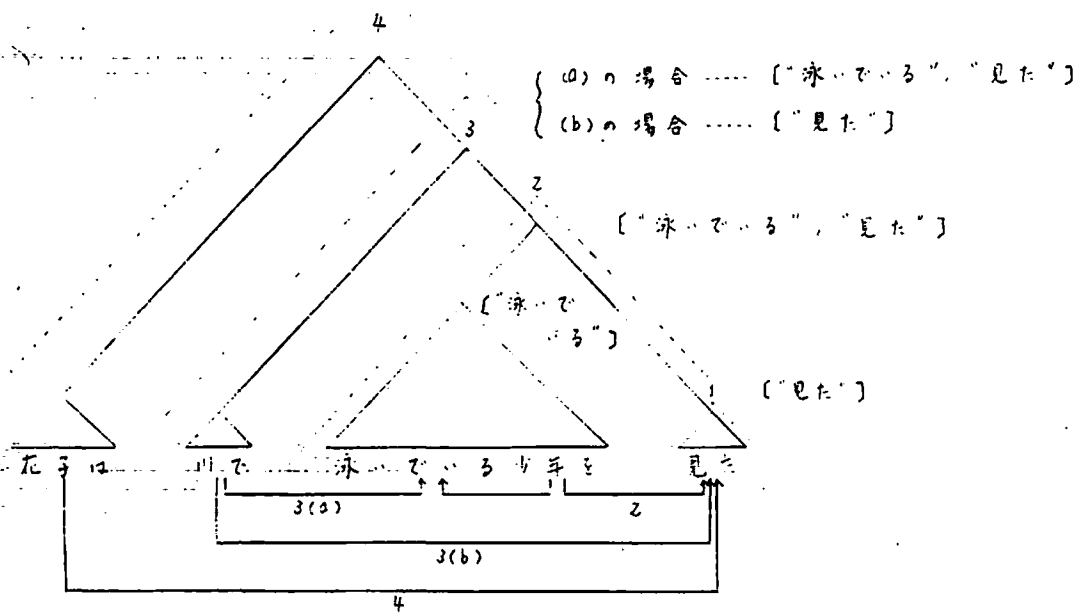


図2　意味解析

部分木から解析を開始し、その解析結果を用いて、その上の、より大きな部分木について解析を繰り返すという方法が考えられる。これにすなわち、末端から根へ意味解析を進めていくことを意味する。文中の他の構成要素を規定するかが名詞などに比べ強い助詞が後置されている日本語では、この意味解析は非常に効率よく行なえるものと思われる。この方法をPrologで実現するなら、その解析過程においてPrologのバックトラック機能により、1つの構文解析木からでも複数の意味構造を抽出することが可能になる。

例えば、図2では、「川で」の係り先として

「泳いでいる」……(a)

「見た」……(b)

の2通りの可能性がそえられるが、今係り先が(a)の方であるとして解析を進めていくと、

川で泳いでいる少年を花子が見た

という意味解析結果が得られる。ここで強制的にバックトラックを掛けると、「川で」の係り先の決定の地点まで自動的に戻り、今度は係り先が(b)の方であるとして解析を進め、

泳いでいる少年を花子は川で見た

という2つ目の意味解析結果も容易に得ることができる。

計算機処理の立場からは方投分かれ構造と右方投分かれ構造の様な解析木を比較してきたが、左方投分かれ構造は、得られる構文解析木の数などの点で問題があり、右方投分かれ構造と比べて一概に優れているとは言えないことがわかった。一方、右方投分かれ構造では、得られる構文解析木をごく少数に抑えることができ、また、その得られたごく少数の構文解析木からでも、Prologのバックトラック機能を用いることで、文の持つすべての意味構造が容易に抽出可能であることがわかった。

そこで今回は、Prologを用いることの利点を活かして、右方投分かれ構造を採用し、その文法を試作してみたが、その際、意味解析が比較的容易に行なえるような木構造が得られるよう考慮した。

5. 形態素解析

日本語構文解析システムにおいては、日言に対する形態素解析処理が必要不可欠である。本構文解析システムでも、三吉らによる形態素解析プログラ

ん〔三吉，他 83〕を基に形態素解析を行なっている。ただし、本構文解析システムでは、辞書構造ヒーて、辞書へのアクセスが非常に効率よく行なえ、また、将来的には辞書を外部記憶化する時に有効な "Trie" 構造〔田中，他 83〕，〔Aho, et al. 83〕を用いている。図3に "Trie" 構造辞書の例を示す。この "Trie" 構造辞書を用いた形態素解析処理の例を以下に示す。

ex) 〔〔ko〕，〔mai〕〕（来ない）

用いる文法規則：

$$\begin{cases} vp \longrightarrow v,\ aux,\ \{comlink\}. & \cdots\cdots ① \\ aux \longrightarrow auxv. \\ aux \longrightarrow auxv,\ aux,\ \{comlink\}. \end{cases}$$

まず、単語（"ko"）を文字のリスト（〔k,o〕）に分解して辞書を見に行く。同音に関しては、語幹しか辞書登録してないので、この辞書検索では語幹のみを取り出す。互語の存在すき時まで辞書の文字を送っていき、語幹"k"を切り出した後、語幹"k"の所に記述されている「来る」に関する情報

```
[verb,kahen,[],kuru]
```

と、リストの残り〔o〕を用いて語尾処理を行なう。語尾処理プログラムは、

```
sobi(mizen,verb,kahen,_) --> [o]. ------②
```

のような形で記述されているが、今の場合はこの②が適用され、語尾処理は成功する。単語"ko"についての辞書引きがこれで終了し、"ko"に関する情報として

```
[v,kahen,mizen,[],verb,kuru] ----- ③
```

が得られる。"mai"についても同様に辞書引きを行ない、その後、構文解析規則①中に記述された相互承接プログラムcomlinkにより、"mai"が③の情報を持つ動詞と承接可能かどうか調べる。

```
conlink(nai,[_,_,mizen,_,verb,X]) :-
    not(member(X,[aru])).
```

6. 補強項

右方枝分かれ構造を得るための文法規則では、埋め込み文に関しては、得

```
dict([[k,[[v,[[verb,kahen,[],kuru]]]],
       [a,[[v,[[verb,dan_5,s,kasu]]],
          [v,[[verb,dan_5,k,kaku]]]]],
       [i,[[v,[[verb,jouge_1,[],kiru]]]]]],
  [n,[],
     [o,[],
        [b,[],
           [o,[[v,[[verb,dan_5,r,noboru]]]]]]],
     [a,[[auxv,[[adj,0,[],nai]]]]],
  [t,[],
     [o,[[v,[[verb,dan_5,b,tobu]]]]],
     [a,[[auxv,[[tokushu,2,[],ta]]]]],
  [s,[[v,[[verb,sahen,[],suru]]]],
     [u,[],
        [t,[],
           [e,[[v,[[verb,jouge_1,[],suteru]]]]]]],
     [o,[],
        [u,[[auxv,[[adjv,0,[],souda]]]]]],
  [r,[],
     [e,[[auxv,[[verb,jouge_1,[],reru]]]]]]]).
```

図3　"Trie" 構造辞書

られる構文解析木は1つであるということはすでに述べた。しかし、その右方枝分かれ構造を得るための文法規則でも構文解析木が複数得られることがある。例えば、連体修飾句の係り先となる名詞句が複数存在する場合である。

ex)　小さな水の粒子の運動

この句を何も制約条件を加えず全く自由な文法規則で構文解析すると、連体修飾句の係り受けの点で曖昧さが生じ、図4のように構文解析木が5個得られる。そこで、文脈に依存した情報を用いた補強項と呼ばれるPrologプログラムを文法規則中に書き込み、得られる構文解析木を、(i)のような、係り受け関係を全く決定しないままの構文解析木1つに絞り込み、係り受け関係の決定は、意味解析過程に委ねる。なお、この意味解析過程における係り受け関係の決定は、埋め込み文からの意味構造の抽出と同様に、容易に実現可能である。

```
|-np                                  |-np
|  |-nlink      (i)                   |  |-nlink      (ii)
|  |  |-np                            |  |  |-np
|  |  |  |-nlink                      |  |  |  |-det -- tiisana
|  |  |  |  |-np                      |  |  |  |-np
|  |  |  |  |  |-det -- tiisana       |  |  |  |  |-n -- mizu
|  |  |  |  |  |-np                   |  |  |-kp -- no
|  |  |  |  |     |-n -- mizu         |  |-np
|  |  |  |-kp -- no                    |  |  |-nlink
|  |  |-np                            |  |  |  |-np
|  |  |  |-n -- ryuusi                |  |  |  |  |-n -- ryuusi
|  |-kp -- no                         |  |  |-kp -- no
|-np                                  |-np
|  |-n -- undou                       |  |-n -- undou
```

図4　補強項なしの文法規則によって得られる構文解析木

```
|-np                                    |-np
  |-nlink     (iii)                      |-det -- tiisana   (iv)
  |  |-np                                |-np
  |  |  |-det -- tiisana                   |-nlink
  |  |  |-np                               |  |-np
  |  |     |-nlink                         |  |  |-nlink
  |  |     |  |-np                         |  |  |  |-np
  |  |     |  |  |-n -- mizu               |  |  |  |  |-n -- mizu
  |  |     |  |-kp -- no                   |  |  |  |-kp -- no
  |  |     |-np                            |  |  |-np
  |  |        |-n -- ryuusi                 |  |     |-n -- ryuusi
  |  |-kp -- no                            |  |-kp -- no
  |-np                                     |-np
     |-n -- undou                             |-n -- undou

|-np
  |-det -- tiisana   (v)
  |-np
     |-nlink
     |  |-np
     |  |  |-n -- mizu
     |  |-kp -- no
     |-np
        |-nlink
        |  |-np
        |  |  |-n -- ryuusi
        |  |-kp -- no
        |-np
           |-n -- undou
```

図4 （続き）

```
|: houkaiseki ha, kimat ta houkou ni kireini
|: wareru seisitu ga aru.

812  msec.
No. 1
|-sentence
   |-pp
   |  |-np
   |  |  |-n -- houkaiseki
   |  |-p
   |     |-fp -- ha
   |-comma -- ,
   |-sentence
      |-pp
      |  |-np
      |  |  |-vp
      |  |  |  |-v -- kimaru
      |  |  |  |-auxv -- ta
      |  |  |-np
      |  |     |-n -- houkou
      |  |-p
      |     |-kp -- ni
      |-sentence
         |-vpt
         |  |-vp
         |     |-adjv -- kireida
         |-sentence
            |-pp
            |  |-np
            |  |  |-vp
            |  |  |  |-v -- wareru
            |  |  |-np
            |  |     |-n -- seisitu
            |  |-p
            |     |-kp -- ga
            |-sentence
               |-vp
                  |-v -- aru

Total Time = 1634  msec.

number of wf_goal was :       24.
number of wf_dict was :       21.
number of wf_fail_goal was :  49.
```

図5　解析例

7. 解析例

図5に今回試作した文法を用いて実際に構文解析した例を示す。今回は、日本語文法〔水谷，他83〕，〔西村，他78〕のコアとなる部分のみしかインプリメントしなかったため、文法規則数は約80程度であり、文法体系としてはごく小規模なものでしかない。しかし、この文法によって中学1年の理科の教科書数ページ分の例文が構文解析可能である。また、その例文に対して得られた構文解析木は、どれも数個程度であった。次に構文解析の効率であるが、1単語あたり1つ目の構文解析木を得るまでの時間が約80msec，構文解析に要したトータルな時間が約150 msec と、かなりの効率で構文解析が行なえることが示せた。

8. おわりに

本研究前半では、構文レベルの情報で未定義語を含む文の解析がどの程度可能か、という問題を検討し、SUPに対して辞書情報，構文情報及び未定義語部分の最短切り出しのHeuristicsを使って、未定義語を含む文の自動分かち書きを実現した。また、BUPの高速辞書引きアルゴリズムの不完全性を指摘し、新しいアルゴリズムを提案した。

その結果、Prologが、その強力なunificationとバックトラックの機能ゆえにこの種のプログラムの開発を行なう上で、非常に有効であることが理かめられた。

後半ではまず、左方枝分かれ構造の構文解析木と右方枝分かれ構造の構文解析木を比較検討した結果、右方枝分かれ構造にも、得られる構文解析木を少数に抑制することができるなどの有効な点があることを示した。次に、その右方枝分かれ構造の構文解析木を得る文法規則を試作し、右方枝分かれ構造では、得られる

構文解析本を少数に抑制することが可能であることを、実際に確認した。

〔謝辞〕

本研究の機会を与えていただいた淵一博ICOT研究所長に感謝致します。また、有益な御討論をしていただいた田中研究室の皆様、電子技術総合研究所推論システム研究室の諸氏、ICOT第2研究室の諸氏に感謝致します。

〔参考文献〕

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

松本裕治, 他 :
"Prologに埋め込まれたボトムアップパーサ: BUP",
Proc. of the Logic Programming Conference '83, 1983

松本裕治, 他 :
"BUPトランスレータ — 文法規則から構文解析プログラムの自動生成 —",
電子技術総合研究所彙報, 47, 8, 1983

松本裕治, 他 :
"BUPの高速化",
情報処理学会自然言語処理研究会 39-7, 1983

Y. Matsumoto and et al. :
"BUP: A Bottom-Up Parser Embedded in Prolog",
New Generation Computing, 1, 2, 1983

安川秀樹, 田中穂積 :
"Prologによる形態素処理と熟語処理について",
情報処理学会自然言語処理研究会 32-4, 1982

井佐原均 :
"日本語文法作成の試み",
電子技術総合研究所パターン情報部推論機構研究室研究討論会資料, 1980

井佐原均, 田中穂積 :
"日本語埋め込み文の構文解析における諸問題",
情報処理学会自然言語処理研究会 26-4, 1981

水谷静夫, 他 :
"朝倉日本語新講座3 文法と意味 I",
朝倉書店, pp 1-80, 1983

西村恕彦, 他 :
"日本語基本文法 — 単文篇 —",
電子技術総合研究所研究報告, 783, pp 30-35, 1978

三吉秀夫, 他 :
"Prologによる日本語文節DCGの生成",
情報処理学会自然言語処理研究会 39-4, 1983

田中穂積, 他 :
"LISPで学ぶ認知心理学3 言語理解",
東京大学出版会, pp 28-35, 1983

A. V. Aho and et al :
"data structures and algorithms",
ADDISON-WESLEY, pp 163-169, 1983