平成3年(1991)1.29

「自然言語処理における統合」シンポジウム論文集

(電子情報通信学会/日本ソフトウェア科学会)

増進的曖昧性解消モデルに基づいた統合的日本語解析

An Integrated Japanese Analysis based on Incremental Disambiguation Model

秋葉友良　伊藤克亘　奥村 学　田中穂積

AKIBA Tomoyosi　ITOU Katunobu　OKUMURA Manabu　TANAKA Hozumi

東京工業大学

Tokyo Institute of Technology

あらまし　本稿では、増進的曖昧性解消モデルに基づいた日本語解析の一手法について述べる。この手法では日本語の特徴を考慮して、文節内ではLR表を用いた構文解析を行ない、文節間では意味主導に、動詞の選択制限に関する一般化弁別ネットワークを用いて係り受け解析を行なう。LR表と一般化弁別ネットワーク(GDN: Generalized Discrimination Network)を共にスタックを操作する知識源として扱うことで同一視して、処理の統合を実現する。構文レベルと意味レベルを統合することにより増進的な日本語解析ができることを示す。

1　はじめに

自然言語解析における大きな問題の一つは、解析の途中で生じる様々な曖昧性をどのように解消するかということである。自然言語に曖昧性が遍在するのは、文を構成する各部分の情報がどれも部分的で[橋田,1988]、文の他の部分の情報を用いないと曖昧性を解消できないことに起因している。文を解析するとき、曖昧性を文末まで解消しない方法をとると、可能性の個数は組合せ的に爆発してしまう。したがって、曖昧性を解消するための望ましい方法は、文を解析する過程で得られる情報(制約)を増進的に蓄積し、できるかぎり早期の段階で曖昧性を段階的に解消していくことである[Mellish,1985]。我々は、意味の曖昧性を増進的に解消するモデルを提案している[奥村,1989]。増進的曖昧性解消モデルは、文を左から右へ読み進む過程で、バックトラックせずに、情報を得られた時点で処理しながら解析を進めていくモデルである。

さらに、形態素・構文・意味・文脈などの各解析レベルの処理では、他のレベルの情報を用いなければ曖昧性を十分に解消できないことがある。例えば、構文解析における曖昧性は、意味レベルの情報を用いて不適切な解析木を排除することで部分的に解消できる。しかし、各レベルの処理を独立して行ない、処理終了後、次のレベルに結果を渡すという方法では、各レベルで部分的な解析しか行なうことができないため、曖昧性が組合せ的に爆発してしまう。望ましい方法は、各解析レベルの処理が入力文に対して同時的に行なわれ、各処理が相互に他の処理の解析結果を利用して曖昧性を解消することである。すなわち、増進的曖昧性解消モデルにとって、解析レベルの統合は不可欠な方向である。

本稿では、増進的曖昧性解消モデルに基づいて、各解析レベルを統合する問題について検討する。構文解析・意味解析のレベルについて、このような立場にたった一解析手法を提案する。この手法では、構文解析レベルとして一般化LR構文解析法[沼崎,1989]を、意味解析レベルとして一般化弁別ネットワーク(GDN)[Okumura,1990]を使用し、日本語の構文と意味の解析を統合し、増進的に曖昧性解消を行なう。

2節では、日本語の特徴を考察し、本稿で提案する手法の基本的な方針を明らかにする。

2　日本語の構造的特徴

日本語の最小構成要素としての語は、自立語と付属語に大別される。「文節」は自立語+付属語列と定義される。文は文節の並びで表現できる。

日本語には、文節の出現順序が比較的自由であるという特徴がある。さらに、文脈から容易に補うことができる文節は頻繁に省略される。そのため、文の構造を反映する形式で記述した文節間の文法規則は曖昧性解消の強い制約とはならず、構文解析過程で解析結果が数多く得られてしまう。この問題を避けるために、日本語の文節間関係の解析には、より意味の情報を強く反映している係り受け解析を行なうことが多い[杉村,1988, 赤坂,1990]。

一方、文節内の語の接続についてはほぼ正規文法で表現できることが知られている[日高,1989]。そのため、構文解析による効果が期待できる。

本稿で提案する手法では、文節内の語の接続解析に一般化LR構文解析アルゴリズム[沼崎,1989]を用いる。一般化LR構文解析アルゴリズムは、解析の曖昧性に対して複数のスタックを保持し、探索を横型に行なうことで、文脈自由文法を扱えるようにLR構文解析法を一般化したアルゴリズムである。並列横型探索なので、増進的曖昧性解消モデルとの整合性が良い。

文節間は、文節内のような文法を用いず、意味情報を主導的に用いて係り受け解析を行なう。その際、増進的曖昧性解消モデルに基づき、増進的に係り受け解析を行なう。意味解析を増進的に行なうモデルとして、一般化弁別ネットワーク(GDN)を利用する(3節)。GDNは格フレームの一表現である。4節では増進的に係り受け解析を行なう手法について概要を述べる。増進的係り受け解析にGDNを用いることによって、解析過程で生じる曖昧性を早期に解消することができることを示す。5節では、文節内の語の接続解析と文節間の係り受け解析を統合する手法について述べる。

3 GDNを用いた増進的意味曖昧性解消

意味的な曖昧性解消を、弁別ネットワークを下向きにたどる過程として実現する研究がある。弁別ネットワークを用いた曖昧性解消過程は、ネットワークを根ノードから葉ノードへ向けて一段づつ下向きにたどる操作に対応する。根ノードから下向きにたどると、到達可能な葉ノードの候補が減少していくことから、曖昧性が増進的に解消される。

弁別ネットワークを用いた意味的曖昧性解消手法には以下のような利点がある。

- 弁別ネットワークは、単語の複数の意味を互いに無関係であるとして独立に扱うのではなく、単語の複数の意味間に存在する関係を考慮した表現形式である。
- 単語の複数の意味を互いに無関係であるとして、候補となる単語の意味をリスト形式で保持していると、曖昧性が増大し候補数が多くなると、リストの線形探索は非常に効率が悪くなる。それに対し、弁別ネットワークを用いた手法は、葉ノードのそれぞれの単語の意味に向けて根ノードからネットワークを下向きにたどる操作になるので、線形探索に比べ効率がよい。

しかし、弁別ネットワークには、前もって決定された順序で制約が入力されないとたどれないという問題がある。我々は[Okumura,1990]で、非決定的な順序で制約が入力された場合でも弁別ネットワークを増進的にたどれる手法を提案している。制約の非決定的な順序に対処できる我々の弁別ネットワークを一般化弁別ネットワーク(GDN)と呼ぶ。

2節で述べたように、日本語の解析では、制約の得られる順序の非決定性が大きく、また制約はしばしば省略されるため、GDNの果たす役割は重要である。

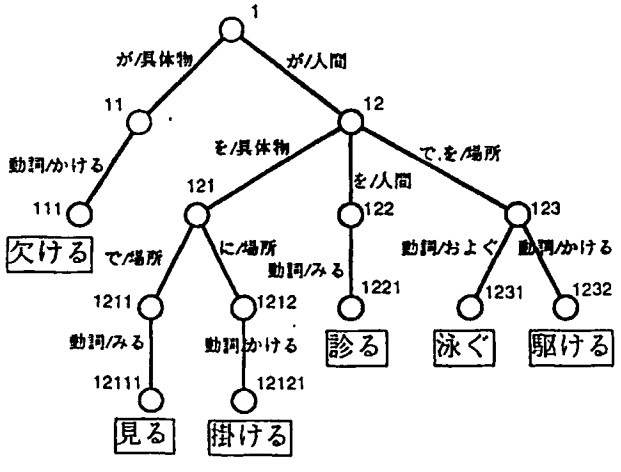

図1：一般化弁別ネットワーク

動詞全体の格フレームに対してGDNを構築することによって、増進的な意味曖昧性解消を行なう。図1は、動詞「みる(見る、診る)」「およぐ(泳ぐ)」「かける(掛ける、駆ける、欠ける)」によって構築したGDNの一例である。ネットワークのアークには格に対する制約(選択制限)および、動詞の見出し語に関する制約が付加される。(図中の'が/人間'や'動詞/かける'などがこれにあたる。)葉ノードは動詞の各語義に対応する。(例えば、葉ノード12121は語義「掛ける」に対応している。)

図1のネットワークを用い、「橋を川にかける」という文を解析することを考える。従来の格フレームを用いた解析法では、動詞「かける」が現れてから解析を始め、「橋を」「川に」がそれぞれ「掛ける」の格フレームの格スロットを満足することから解析に成功し、「かける」の意味が「掛ける」に決まる。すなわち、動詞「かける」が現れるまで曖昧性の解消は行なわれない。

GDNを用いた解析では、従来の解析法と比べ早期に曖昧性解消が行なえる。GDNの処理過程は、ノードに対してユニークに付けられた識別子と、根ノードからそのノードまでのパスに含まれるアークに付加された制約のうち満足していない制約の位置を表現するビットベクトルの組(状態)で表される。初期状態として、GDNの根ノード1と空のベクトルを表す0^1の組(1,0)から解析が始められる。「橋を」が入力されると、ネットワークの枝に付けられている制約のうち、'を/場所'と'を/具体物'が満足されるので、ノード121あるいは123までネットワークをたどることができる。ビットベクトルは共に010であり、根ノードからこれらのノードまでのパス中でビットが1の位置の制約('が/人間')が満足されていないことを表す。すなわち、状態{(121,010)(123,010)}に遷移する。この状態から到達可能な葉ノードは、「見る」「掛ける」「泳ぐ」「駆ける」であり、動詞の候補がこれらに絞り込まれたことを意味する。次に「川に」は、制約'に/場所'を満足する。ノード123からはこれ以上ネットワークをたどることができないが、ノード121からはノード1212にたどることができる。ビットベクトルは0100である。状態は(1212,0100)に遷移する。この状態において次のことがわかる。

- この状態から到達可能な葉ノードは、12121(掛ける)のみであり、動詞の意味の候補は「掛ける」に絞り込まれた。
- この状態から葉ノードにたどり着くのにまだ満足されていない制約は'が/人間'と'動詞/かける'である。したがって次の入力はおそらくこれらの制約のどちらかを満たすものであろうという予測が得られる。

以下、入力「かける」に対して制約'動詞/かける'が満足され、状態(12121,01000)にたどり着く。このように、GDNを用いた意味的な曖昧性解消の過程では、動詞が現れる前に増進的に動詞の意味の曖昧性を解消することができる。

また、GDNを使うことによって、意味的な曖昧性だけでなく、文節間の係り受け関係の曖昧性も増進的に解消することができる(4.3節)。

4 増進的係り受け解析

日本語の二文節間の係り受け関係を、文を左から右に読み進む過程で増進的に解析することを考える。図2のように係り受け関係を、修飾する文節から修飾される文節へ引いたアークで表現することにする。このとき、係り受け関係には、

- 文節は、後続の文節のいずれか一つに係る。ただし最後の文節は係り先がない。
- どの二つの係り受け関係も交差しない。[2]

[1] ビットベクトルの左端の0は、識別子と桁数を合わせるためのもので、対応する制約は存在しない。

[2] この係り受け非交差の原則を破る文が存在することが知られている。しかし、本稿ではこの原則を厳守する方針で議論を進める。

などの制約が存在する。したがってアークによる表現は、

- 最後の文節を除く文節からは、必ず一つのアークが出て、後続の文節の一つに入る。
- どのアークも交差しない。

という制約を満足する必要がある。

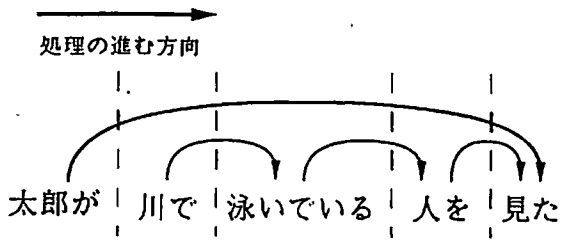

図2：文節間の係り受け

4.1 スタックを用いた状態表現

係り受け解析の処理過程を表す状態表現は、解析中の文節の上を通過しているアークの集合で表すことができる。このアーク集合に対する操作は、上述した制約を考慮すると最後に付加したアーク (最も下に位置する) から取り出しが行なわれるので、アークの集合はスタックを使って表現できる。

スタックは曖昧性に応じて複数用意されるが、その内スタックのトップが同じものを木構造化することで、同じ処理が何度も行なわれることを避けることができる [沼崎,1989]。例えば、解析中に曖昧性が生じ次の二つのスタックが得られたとする。

```
(top)    [e,d,c,b,a]          (bottom)
         [e,b,a]
```

ここで、スタックトップが共通なので、木構造化を行なうと、

```
[e,[d,c,b,a],
   [b,a]]
```

となる。これ以降の処理は、この木構造化スタックのスタックトップ e に対して一つのプロセスで進めればよく、再計算を避けることができる。スタックの分岐点を越えてポップが行なわれた場合、再びスタックは 2 つに分かれる。

また、同じ動詞に係る文節から出るアークは一本にまとめることにする。これは、4.3節で述べるように同じ動詞に係る文節を GDN を用いて増進的に処理するためである。スタック表現では、アークに対応するスタックの要素同士を融合して、一要素とすることで表す。

以上のような状態表現を用い、「太郎が川で泳いでいる人を見た」という文を増進的に係り受け解析し、「太郎が川で泳いでいる...」まで処理した状態のスタックを図 3 に示す。スタックに積まれている {} で囲まれた要素は、それぞれ対応する文節 (の意味表現) を、アークが一本にまとまっているものに対しては、二つ以上の文節を {} で囲んで表している。係り受け関係の曖昧性によってスタックは 4 つに分岐するが、そのうちの 2 つに関してはスタックのトップが同じであるので、木構造化することができる。

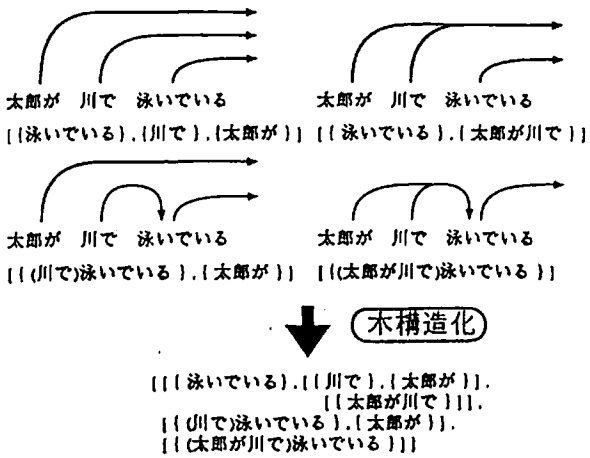

図3：「太郎が川で泳いでいる...」のスタック表現

4.2 スタックの操作

増進的係り受け解析では、アークに対する操作は、文節に入るアークの操作 (IN) と文節から出るアークの操作 (OUT) に大別される。係り受け解析の状態表現をスタックで表現した場合、そのスタックに対する操作は、IN と OUT で次のように分類される (図 4)。

- IN
  - pass アークが文節に係らない。スタックはそのまま OUT に渡される。
  - pop アークが文節に係る。スタックのトップに積まれている要素をポップする。2 回以上の pop が行なわれる場合もある。
- OUT
  - push アークを新たに生成する。スタックのトップに文節 (の表現) をプッシュする。
  - merge すでに存在するアークに、アークを結合して一本にまとめる。文節をスタックのトップに積んでいる要素と融合する。

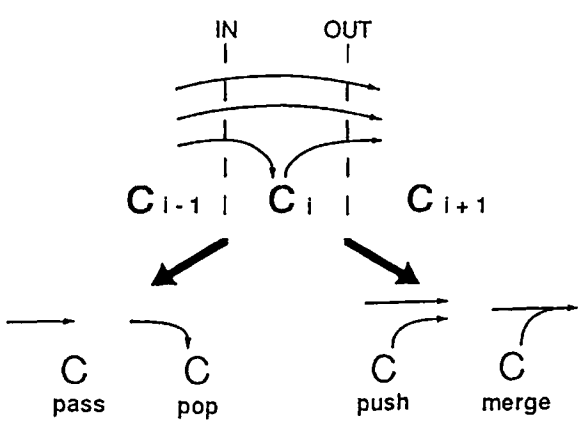

図4：係り受け操作の分類

係り受け解析はGDNを用いて意味主導的に行なわれるが、係り受け関係にある文節間には、その構文的タイプに関して、文法的な制約も存在する。

文節を、それ自体の構文的タイプ(*Type*)と係り先の構文的タイプ(*Mod*)の組(*Type*,*Mod*)によってカテゴリに分類する。タイプはn(体言)とv(用言)の2種類を考える。よって、カテゴリは(n,n), (n,v), (v,n), (v,v)の4種類に分類される(図5)。スタックトップに積まれているアークがINでpopされる(ある文節が別の文節に係る)ためには、アーク(の出発点)のカテゴリの*Mod*と係り先の文節の*Type*が同じでなければならない。また、先に述べたように、同じ動詞に係る文節から出るアークをまとめるため、OUTではアークのカテゴリと文節のカテゴリが共に(n,v)の場合に限りmergeを行なう。

(*Mod*,*Type*)	例	(修飾先)
(n, n)	日本語の	(論理)
(n, v)	日本語を	(話す)
(v, n)	読む	(本)
(v, v)	歩いて	(帰る)

図5：係り受け関係によるカテゴリの分類

4.3 GDNを用いた増進的係り受け解析

4.1節で、増進的係り受け解析では、同じ動詞に係る文節から出るアーク同士を一本にまとめることを述べた。このことによって、増進的係り受け解析では、文節が同じ動詞に係るという仮定を純粋にbottom-upな方法よりも早期に生成する。GDNは、この仮定を意味処理(格フレーム)によって検証し、動詞が現われる前に不適当な解析を排除することができる。

アークを結合して一本にまとめるかどうかの決定は、mergeプロセスによって処理される。mergeプロセスは、現在解析中の文節とスタックのトップに積まれている文節(の集合)のカテゴリが共に(n,v)である場合に呼び出される。解析中の文節の情報を用いて、スタックのトップに積まれている文節(の集合)が表すGDNの状態(ノード)から、GDNを継続してたどれる場合はアークを結合する。すなわち、スタックのトップを新しいGDNの状態に置き変える。

例として、文節「太郎が」に続いて文節「道を」が入力された場合を考える。図1のGDNに対する「太郎が」の解析結果、状態(12,00)がスタックのトップに積まれている。「道を」が入力されると、制約'を/場所'が満足されるので状態(123,000)に遷移することができる。したがって、アークは結合されて、スタックのトップを新しい状態(123,000)に置き変える。

次に、文節「太郎を」に続いて文節「道を」が入力された場合を考える。「太郎を」の解析で、状態(122,010)がスタックのトップに積まれている。「道を」の入力に対して、上の場合と同じく、制約'を/場所'が満足される。しかし、この制約では状態(122,010)から継続してたどることはできない。したがって、アークは結合されない。すなわち、「太郎を」と「道を」は異なる動詞に係るしかないという解析が、動詞が現れる前に早期に行なわれる。

popプロセスに関しても、同様にGDNを用いた意味処理によって不適当な解析を排除することができる。例えば、文節あるいはアークを結合した文節の集合が動詞に係ることができるかどうかの検証には、そのまま継続してGDNをたどれるかどうかを調べればよい。例えば、「川に橋を」(アークは結合している)のGDNの状態は(1212,0100)であるが、この状態から制約'動詞/かける'によってGDNを継続してたどることができるので、アークは動詞「かける」に入ることができる。しかし、制約'動詞/およぐ'ではたどることができず、アークは動詞「およぐ」に入ることができない。

実際には、OUTにおいてmergeプロセスと並行してpushプロセスが実行される。pushプロセスは、埋め込み文が開始されるという仮定を生成する。またINにおいても、popプロセスと並行してpassプロセスが実行される。

popとmergeに関しては、GDNを用いることにより不適当な解析を早期に刈り込むことが期待できる。一方、passとpushに関しては、バックトラックなしに解析が行なわれるため、常にその可能性を考慮しなければならない。ただし、pushプロセスは、スタックの木構造化(4.1節)によって、これまでのすべての解析結果に対して1度だけ呼び出せばよい点に注意していただきたい(詳しくは5.1節)。

また、passやpushが続くと必然的にスタックが深くなり、このことは埋め込みが何重にも行なわれていることを意味する。これはスタックの深さに関するヒューリスティクスを用いたスコア付けを行ない、枝刈りすることで対処できると考える。

以上に述べた増進的係り受け解析は、push操作によって最初にbottom-upに仮定を生成し、その後merge操作によってその仮定を満足させるようにtop-downに予測を用いる点で、左隅構文解析法[Aho,1972]に近い方法である。さらに、潜在的な記憶の限界を考慮し、可能な解析を並列に探索している点は、心理学的にも妥当な方法であると考えられる[Hasida,1990] [Johnson-Laird,1983]。また、この増進的係り受け解析法は、他の多くの方法と同様、入力文字列の長さnに対し、時間計算量が$O(n^3)$、空間計算量が$O(n^2)$である。

5 LR表とGDNによる構文・意味解析の統合

LR表とGDNは、共に計算機で処理しやすい形にコンパイルされたデータ構造である。LR表は、スタックのトップに乗っている「状態」と入力から得られる先読みされた「終端記号」との関係から次の操作を決定し、最終的に次の「状態」に遷移する。一方GDNは、識別子とビットベクトルの組で表される「状態」と入力から得られる「制約」との関係から、次の「状態」に遷移する。この類似性は、LRパーザがLR表を用いた解析を行なうのと同じように、LR表をGDNに置き換えて解析を行なえる可能性を示唆している。

LR表とGDNの本質的な違いは、LR表が制約の入力順序を考慮に入れた状態遷移情報を表しているのに対して、GDNが制約の入力順序を自由にした場合の簡潔な状態遷移情報を表している点にある。文節間解析のために、GDNの代わりにLR表を用いるのは、弁別木のラティス表現(taxonomic lattice [Woods,1978])に相当する。しかし、ネットワークをラティス化すると、同じ情報量を表現するのに冗長な表現形式となり、計算機上で実現した場合に大量の記憶容量が必要に

になる [Okumura,1990]。また、一般化 LR 構文解析アルゴリズムでは、先読み情報が有効に働かない特殊な文法に対しては、計算時間が $O(n^3)$ を越えてしまうことが知られている [Johnson,89]。したがって、語順が自由である文節間解析に LR 表を用いるのは、LR 構文解析の利点を十分に生かしきれない結果となる恐れがある。

我々の日本語解析の手法では、語順が強い制約となる文節内の解析には LR 表を、語順が自由である文節間では GDN を用いて文を解析する。その際、一般化 LR 構文解析アルゴリズムで使用されるスタックと、増進的係り受け解析 (4節) で使用されるスタックとを区別せず、同じ (木構造化) スタックを LR 表と共に GDN でも操作することで、構文解析と意味解析を統合する (図 6)。

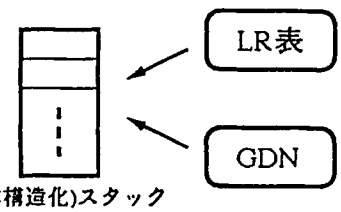

図 6 : LR 表と GDN による処理の統合

解析器の動作について説明する。まず、LR 表が入力された語に対し LR 構文解析を実行し、文節内を解析する。文節の終りまで解析が進むと、reduce プロセスが呼び出されて文節が得られる。文節間の解析も LR 構文解析で行なう手法では、ここで goto プロセスを呼び出し、得られた文節をスタックに積んで解析を進めるが、本手法では得られた文節を GDN に渡す。GDN は、得られた文節とスタックに積まれている以前にたどった GDN の状態 (ノード) を参照して係り受け解析を実行する (4.3節)。

文節内解析と文節間解析のスタックを区別しないことで、処理過程における曖昧性を、文節内と文節間、あるいは構文情報と意味情報で区別することなしに、統一的に扱うことができる。[沼崎,1989] では一般化 LR 構文解析において、解析の曖昧性に応じて生じる複数のスタックに対応するプロセスを並列に実行することで、並列に構文解析を行なう手法を提案している。本稿の手法でも、同様に分岐したスタックを並列に実行することで、日本語の構文・意味解析を並列に実行することができる。

	action				goto	
state	noun	verb	props	$	(n,v)	(v,n)
0	sh1	sh3			5	5
1			sh2			
2	re1	re1		re1		
3	re2	re2	sh4	re2		
4	re3	re3		re3		
5	sh1	sh3			5	5

規則 1: (n,v) → noun props
規則 2: (v,n) → verb
規則 3: (v,n) → verb props

図 7 : LR 表

5.1 解析例

ここで、「太郎が川で泳いでいる人を見た」という文を解析する場合を考えてみる。解析に使用する LR 表と GDN は、それぞれ図 7、図 1 とする。また、4で述べた係り受け関係に関するカテゴリのうち、(n,v) と (v,n) のみを考えることにする。

まず、スタックが空の状態で解析が始まる。スタックには、LR 構文解析で用いる初期状態 0 が積まれている。

[0]

まず文節内の解析が行なわれる。入力された文字列に対し、語の接続が LR 構文解析法によって解析される。

ここでは、入力 (先読み) が「太郎」で品詞は名詞、スタックのトップの状態が 0 である。LR 表を参照すると shift を行ない状態 1 に遷移することがわかる。スタックには、解析結果と新しい状態が積まれる。

入力 (語)	操作	結果 (スタック)
太郎	shift 1	[1,[noun, 太郎],0]

同様に、入力「が」に対して shift が起こり、「川」が入力されると (先読みされると)reduce 1 が生じ、スタックに積まれている [prop, が],[noun, 太郎] がポップされて、文法規則 1 に従い文節 [太郎が] が得られる。また、この文節のカテゴリが (n,v)、すなわち「太郎が」は自分自身が n (体言) で係り先が v(用言) であることがわかる。

が	shift 2	[2,[prop, が],1,[noun, 太郎],0]
川	reduce 1	[0]

文節間の解析もそのまま LR 構文解析を用いる手法では、ここで goto プロセスを呼び、解析結果の [太郎が] をスタックに積んで新しい状態に遷移する。しかし本手法では、ここで GDN を用い、4で述べた増進的係り受け解析でのスタック操作 (pass, pop と push, merge の組合せ) が行なわれる。

現在、スタックは空であるので、IN では何も行なわれず、OUT では push のみが実行される。push プロセスは、得られた文節を制約として、根ノードから弁別ネットワークをたどり、その結果の状態 (たどり着いたノードの識別子とビットベクトル) をスタックに積む (ここでは (12,00))。以下表中では、この弁別ネットワークを途中までたどった状態を、処理された文字列を { } で囲って表すことにする。

入力 (文節)	操作 (IN/OUT)	結果 (スタック)
[(n,v), 太郎が]	-/push	[5,[(n,v),{ 太郎が }],0]

入力「川」「で」に対しても同じように処理が進み、reduce によって文節 [(n,v), 川で] が得られる。

川	shift 1	[1,[noun, 川],5,{ 太郎が },0]
で	shift 2	[2,[prop, で],1,[noun, 川],5,{ 太郎が },0]
泳	reduce 1	[5,{ 太郎が },0]

現在、スタックのトップには、カテゴリ (n,v) の { 太郎が } が積まれている。スタックトップの *Mod* と得られた文節の *Type* が異なるので pop は生じず、pass のみが起こる。しかし OUT では、カテゴリが共に (n,v) なので、merge プロ

入力(文節)	操作(IN/OUT)	結果(スタック)
[(n,v), 太郎が]	-/push	[{ 太郎が }]
[(n,v), 川で]	pass/push	[[{ 川で },{ 太郎が }],
	pass/merge	[{ 太郎が川で }]]
[(v,n), 泳いでいる]	pass/push	[[{ 泳いでいる },[{ 川で },{ 太郎が }],
		[{ 太郎が川で }]],
	pop/push	[{ 川で泳いでいる },{ 太郎が }],
		[{ 太郎が川で泳いでいる }]]
[(n,v), 人を]	pass/push	[[{ 人を },[{ 泳いでいる },[{ 川で },{ 太郎が }],
		[{ 太郎が川で }]],
		[{ 川で泳いでいる },{ 太郎が }],
		[{ 太郎が川で泳いでいる }]],
	pop/push	[{(泳いでいる) 人を },[{ 川で },{ 太郎が }],
		[{ 太郎が川で }]],
		[{(川で泳いでいる) 人を },{ 太郎が }],
	pop/merge	[{(泳いでいる) 人を川で },{ 太郎が }],
		[{ 太郎が (泳いでいる) 人を川で }],
		[{ 太郎が (川で泳いでいる) 人を }]]
[(v,n), 見た]	pass/push	[[{ 見た },[{ 人を },[{ 泳いでいる },[{ 川で },{ 太郎が }],
		[{ 太郎が川で }]],
		[{ 川で泳いでいる },{ 太郎が }],
		[{ 太郎が川で泳いでいる }]],
		[{(泳いでいる) 人を },[{ 川で },{ 太郎が }],
		[{ 太郎が川で }]],
		[{(川で泳いでいる) 人を },{ 太郎が }],
		[{(泳いでいる) 人を川で },{ 太郎が }],
		[{ 太郎が (泳いでいる) 人を川で }],
		[{ 太郎が (川で泳いでいる) 人を }]],
	pop/push	[{ 人を見た },[{ 泳いでいる },[{ 川で },{ 太郎が }],
		[{ 太郎が川で }]],
		[{ 川で泳いでいる },{ 太郎が }],
		[{ 太郎が川で泳いでいる }]],
		[{(泳いでいる) 人を見た },[{ 川で },{ 太郎が }],
		[{ 太郎が川で }]],
		[{(川で泳いでいる) 人を見た },{ 太郎が }],
		[{(泳いでいる) 人を川で見た },{ 太郎が }],
		[{ 太郎が (泳いでいる) 人を川で見た }],
		[{ 太郎が (川で泳いでいる) 人を見た }]]
$	accept	[[{ 太郎が (泳いでいる) 人を川で見た }],
		[{ 太郎が (川で泳いでいる) 人を見た }]]

図 8: 文節間解析のトレース

セスが呼び出される。新しく得られた文節(制約)が、スタックのトップに積まれているGDNの状態から(今の例では、{(12,00)}から)弁別ネットワークをさらにたどれるかどうかがチェックされ、たどれるならば新しい状態をスタックに積む。[(n,v),川で]は弁別ネットワークをたどることに成功し、状態{(123,000),(1211,0010)}が得られる。同時にpushプロセスも呼び出されて、GDNを[(n,v),川で]で新たにたどった状態{(123,010),(1211,0110)}も得られる。ここでコンフリクトが生じるのでスタックが2つに分岐する。すなわち、

[(n,v),川で]	pass/push pass/merge	[[5,{川で},5,{太郎が},0], [5,{太郎が川で}]]

「泳いでいる...」についても文節内の解析はLR解析によって同様に行なわれ、文節間の処理に渡される。文節[(v,n),泳いでいる]が得られ、スタックトップの*Mod*と文節の*Type*が等しいことからINでpopプロセスが呼び出される。スタックは2つに分岐しているので、それぞれのスタックトップに対してpopの可能性が一般化弁別ネットワークによって検査される。今の場合どちらのpopにも成功する。OUTではpushのみが起こる。INでpassされたスタックに対しては、pushプロセスは一度だけ呼び出せばよい。スタックが木構造に表現されるので、その後のプロセスで重複を防ぐことができる。

```
[[5,{太郎が川で泳いでいる},0],
 [5,{川で泳いでいる},5,{太郎が},0],
 [5,{泳いでいる},[5,{太郎が川で},0],
                 [5,{川で},5,{太郎が},0]]]
```

次に得られる文節[(n,v),人を]のINでもpopプロセスが呼び出され、連体修飾の解析が行なわれる。上の図のうち一番上のスタックは、(が格がすでに{太郎}によって埋められているため)ネットワークの遷移に失敗し、刈り込まれる。

以下同様に解析が行なわれる。文節間の解析のトレースを図8に示す。最終結果として、スタックに1つだけ積まれているものだけが受け入れられ、「太郎が川で(泳いでいる)人を見た」「太郎が(川で泳いでいる)人を見た」の2種類の結果が得られる。

6 おわりに

増進的曖昧性解消モデルに基づいた日本語解析の一手法について述べた。本手法では日本語の特徴を考慮して、文節内ではLR表を用いた構文解析を行ない、文節間では意味主導に一般化弁別ネットワークを用いて係り受け解析を行なう。LR表と一般化弁別ネットワークは、共にスタックを操作する知識源として扱うことで同一視でき、処理の統合が行なえる。構文レベルと意味レベルを統合することにより増進的な日本語解析ができることを示した。

関連した研究として、[峯,1990][北,1990]は、文節内と文節間で2つのLR表を用いる2段階LR構文解析法を提案している。しかし3,5節で述べたように、文節間では構文の制約はあまり役に立たず、また自由な語順を扱うにはLR表では冗長な表現となる。我々の手法は文節間のLR表を一般化弁別ネットワークに置き換えたものに相当し、文節間の解析に意味の制約を主導的に使うことで、語順の任意性や省略を扱える枠組となっている。さらに我々の増進的係り受け解析では、アークの統合により、LR法などの純粋にボトムアップな方法よりも早期に係り受けの仮定を生成することができ、またGDNを用いることでその仮定の意味的妥当性を早期に検証することができる。

本稿では、一般化弁別ネットワークで表現する知識として選択制限を用いたが、今後の課題として、これ以外の制約の導入が考えられる。特に連体修飾節の解析は、現状では格要素型の解析しか行なうことができない。[佐藤,1989]では、連体修飾節を幾つかの型に分類して、各型毎に必要な情報と解析方法について述べている。

本研究で述べた増進的曖昧性解消モデルはAIにおける古典的な横型探索モデルに近い。曖昧な解析結果に対して処理を分岐していくことで、木状に処理が展開していく。各レベルでの曖昧性は木の枝の分岐という形で統一的に扱われる。また展開した各枝での処理は並列に実行される。ただし、展開の組合せ的爆発を防ぐために次のような制御を行なう。

1. 同じ計算を繰り返さないために、処理の同じ枝はできるだけ統合する。

2. 増進的曖昧性解消には、早期に利用できる情報を使って解析を行なう一面と、曖昧性解消に役立たない処理は評価を遅延するという一面がある。したがって、ヒューリスティクスを用いて、曖昧性の解消に役立つ情報から展開を行なう。

3. 各枝にスコアを付加し、各レベルでの解析結果を調整するとともに、相対的に低いスコアを持つ枝を刈り込むことで、組合せ的爆発を防ぐ[Carter,1987]。

本稿で述べた手法では、基本データ構造として木構造化スタックを用いて増進的曖昧性解消モデルを実現している。曖昧性に対し複数のスタックに展開している。スタックを木構造化し、無駄な再計算を避けることで1.を実現している。また本手法ではスタックを操作するものとして、文節内と文節間の解析でLR表と一般化弁別ネットワークを使い分けている。すなわち、日本語の特徴を考えて文節内で構文情報を用い、文節間では格フレームを用いて解析を行い、2.を実現している。また、各スタックにスコアを付加することで3.が実現できると考えられる。

我々の目標は、増進的曖昧性解消モデルに基づき、より多くの言語レベルを統合することである。より下のレベルに関しては、[堀内,1990]で未定義語を含んだ辞書引きをバックトラックなしに行なう手法を示しており、本手法に自然に組み込むことができると考える。各レベルで生じる曖昧性を統一的に枝の展開という形で表すことによって、他のレベルの解析も同じように扱えるであろう。どのレベルのどのような情報から展開を行なうかということに関しては、実際の言語現象を調査することで今後考察していきたい。

謝辞

本稿の査読者の方々には、大変有意義なコメントをいただきました。ここに感謝の意を表します。

参考文献

[Aho,1972] Aho,A.V. and Ullman,U.D. "The Theory of Parsing, Translation and Compiling". Prentice-Hall, 1972.

[Carter,1987] David M. Carter. "Control Issues in Anaphor Resolution". SRI International technical reports, 1987.

[Hasida,1990] Hasida, K. "A Constraint-Based Approach to Linguistic Performance". COLING, 1990.

[Johnson,89] Johnson,M. "The Computational Complexity of Tomita's Algorithm". International Parsing Workshop '89, Carnegie-Mellon University, pp.203-208, 1989.

[Johnson-Laird,1983] Johnson-Laird, P.N. "Mental Models — Towards a Cognitive Science of Language, Inference and Consciousness". Cambridge Univ. Press, Cambridge, 1983. (邦訳: AIUEO.「メンタル・モデル — 言語・推論・意識の認知科学」. 産業図書, 1987)

[Okumura,1990] Manabu Okumura, Hozumi Tanaka. "Towards Incremental Disambiguation with a Generalized Discrimination Network". AAAI, 1990.

[Mellish,1985] C. S. Mellish. "Computer Interpretation of Natural Language Descriptions". Ellis Horwood. 1985. (邦訳: 田中穂積.「コンピュータのための自然言語意味理解の基礎」. サイエンス社, 1987)

[Woods,1978] W. A. Woods. "Taxonomic lattice structures for situation recognition". In Theoretical Issues in Natural Language Processing 2, pp.33-41, 1978.

[赤坂,1990] 赤坂宏二.「日本語の並列係り受け解析」情報処理学会第 41 回全国大会,1990.

[奥村,1989] 奥村学, 田中穂積.「自然言語における意味的曖昧性を増進的に解消する計算モデル」. 人工知能学会誌, 4(6), 1989.

[北,1990] 北研二, 竹澤寿幸, 保坂順子, 江原暉将, 森元逞.「2 段階 LR 構文解析法を用いた文認識」. 日本音響学会講演論文集,3-8-17,1990.

[佐藤,1989] 佐藤龍一, 田中穂積.「常識を用いた日本語連体修飾の解析」電子情報通信学会,NLC89-17.

[杉村,1988] 杉村領一, 三吉秀夫, 向井国昭.「コンストレイントに基づく日本語の係り受け解析」情報処理学会第 35 回全国大会,1988.

[沼崎,1989] 沼崎浩明, 田村直良, 田中穂積.「並列論理型言語による一般化 LR 構文解析アルゴリズムの実現」情報処理学会,NL74-5,1989.

[橋田,1988] 橋田浩一.「AI とは何でないか — 情報の部分性について」bit, 20(8): pp.48-60, 1988.

[日高,1989] 日高達.「自然言語理解の基礎-形態論」. 情報処理,vol.30,1989,no.10.

[堀内,1990] 堀内靖雄, 伊藤克亘, 田中穂積.「拡張 LR 構文解析アルゴリズムによる未定義語を含む日本語文の構文解析」情報処理学会第 40 回全国大会,1990.

[峯,1990] 峯恒憲, 谷口倫一郎, 雨宮真人.「日本語の並列形態素解析」情報処理学会第 40 回全国大会,1990.