TR-I-0376

ASURA 英語生成処理機構の機能評価
Performance of the Generation Module for ASURA

菊井 玄一郎　　渡辺 学
Gen-ichiro KIKUI　Manabu WATANABE

1993.3

梗概

本報告書では ASURA の生成処理機構に対して加えられて来た種々の効率化機構の効果、および、生成処理機構の解析処理への応用の試みとその結果について述べる。
まず、各効率化機構の有無に応じて処理速度、消費メモリなどを測定し、期待された効果が得られているかどうか検討する。次に、この生成処理系のために記述された言語知識を用いた解析処理とその結果、および、問題点について示す。

ATR 自動翻訳電話研究所
ATR Interpreting Telephony Research Laboratories

目次

第 1 章

まえがき

データ処理研究室では自動翻訳電話の実現可能性を確認するため、音声認識、言語処理などの要素技術を組み合わせた音声言語翻訳実験システム ASURA を作成した。このシステムの言語翻訳処理の大きな特徴は、言語現象に関わる知識を制約として素性構造の形式で表現し、これを単一化 (unification) 演算によって処理する方式としたことである。

素性構造の単一化を用いた生成処理システムとしては、従来、言語処理研究室で開発されていたタイプ付き素性構造による文生成モジュールがある [2]。これは、素性構造のタイプに対する制約を記述した主辞駆動型の文法規則を用いてトップダウンに生成処理を行なうものである。

この方式は適用可能な規則のタイプを制約することによって処理の効率化を図ったものである。しかし、基本的な枠組がトップダウン方式であるため、語彙情報を優先的に使うことが難しく、主辞駆動文法 (Head-driven Phrase Structure Grammar)[1] などの語彙情報が処理を制御する文法では効率が悪いという問題があった。

これに対して ASURA の生成処理系は必要な語彙情報を早期に利用できる「意味主辞駆動型」[3] と呼ばれるアルゴリズムを基本的なアルゴリズムとし、これを ASURA の基本的なデータ表現である素性構造が適切に扱えるように改良したものである。さらに、この処理系は素性付き木付加文法 (Feature Structure-based Tree Adjoining Grammar)[4] も扱うことができるように拡張されている。

本報告書では、まず、この処理系をさらに効率的かつ汎用的にするために我々の行なってきた幾つかの改良点の効果について述べる。次に、この生成処理系のために記述された言語知識を解析処理に応用する実験について述べる。

第 2 章

生成処理の効率の評価

2.1 実験条件

本処理系のために考案された機構の有効性を評価するために、GS92-V3.5.4 に対して Table 2.1に示すファイルを用いて生成実験を行った。

2.2 構造共有 (中間結果の再利用)

生成過程で中間多義が発生するとそれぞれの仮説木に対して独立に生成処理が繰り返されるが、これらの展開はお互いに良く似た部分木を含む事が予想される。そこで中間多義以下では展開節点の素性構造と展開に用いた PD 連鎖を保存しておき、他の仮説木の展開で同じ意味構造を持ち素性構造が包摂される節点を展開する時には、その PD 連鎖を再利用するようにした。この再利用のことを歴史的経緯により「構造共有」と呼ぶ。
構造共有は PD 連鎖を再利用できるだけでなく、展開に失敗する部分木も共有する事で無駄な展開を減らす事ができる。
しかし一方で、素性構造と PD 連鎖の保存および素性構造の包摂性検査などのオーバヘッドがあり、実際の効果は文法やコーパスに依存する。

2.2.1 実験結果

独語コーパスと英語コーパスに対する実験結果を Table 2.2および Figure 2.1、Figure 2.2に示す。
表はコーパス全体の平均値を表している。
図には構造共有しない場合の展開節点の数に対する処理時間の比（構造共有なし／構造共有あり）をプロットしてある。

2.2.2 独語生成

Figure 2.1からは展開節点が多いものほど速度比も高い事が分かる。
横軸の値が小さい所では速度比がばらついているが、これは比をとっているために展開節点が少ない所 (処理時間も小さい) では誤差が拡大されてしまうためである。
Figure 2.1の結果は、構造共有が多義を多く含む (展開節点も多い) 場合に有効である事を定性的に裏付けている。
ところで懸念された処理時間に対するオーバヘッドはほとんど見られない。これは、実際に中間

独語	ファイル名
コーパス	ref-a10.921117.lisp
PD 規則	xx-09.lisp
MG 規則	921002.lisp
MG ネット	921113.lisp
形態素辞書	921113.dict

英語	ファイル名
コーパス	mset1026.loop.loop
PD 規則	pd.smp.mset.oct.30
MG 規則	defrule.lisp
MG ネット	defnet.921109.lisp
形態素辞書	atr_made.dict5.1104

表 2.1: 生成実験に用いたファイル一覧

独語	構造共有なし	構造共有あり	比
CPU(sec)	4.08	2.09	2.0
消費セル (KB)	2,129	640	3.3
展開節点数	69.3	40.2	1.7
探索 PD 列数	127.7	50.6	2.5
活性化 PD 数	234.7	95.6	2.5
適用 PD 数	14.8	14.8	1.0
中間多義数	6.8	3.0	2.3
部分木数	40.1	19.7	2.0

英語	構造共有なし	構造共有あり	比
CPU(sec)	1.00	0.98	1.0
消費セル (KB)	234	229	1.0
展開節点数	9.23	9.13	1.0
探索 PD 列数	14.38	12.28	1.2
活性化 PD 数	66.21	52.57	1.3
適用 PD 数	10.93	10.93	1.0
中間多義数	0.91	0.89	1.0
部分木数	3.41	3.30	1.0

表 2.2: 構造共有の実験結果

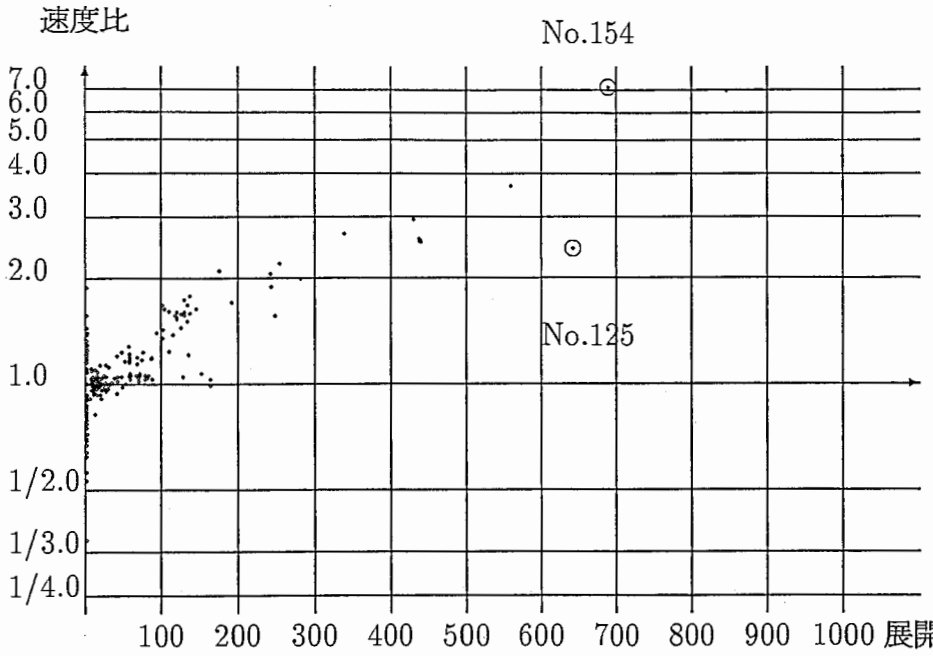

図 2.1: 独語コーパスに対する構造共有の実験結果

No.170(2256,21.2) は 1 点だけ飛び抜けており図示していない。

二重丸の点は Table 2.3で詳細を示す。

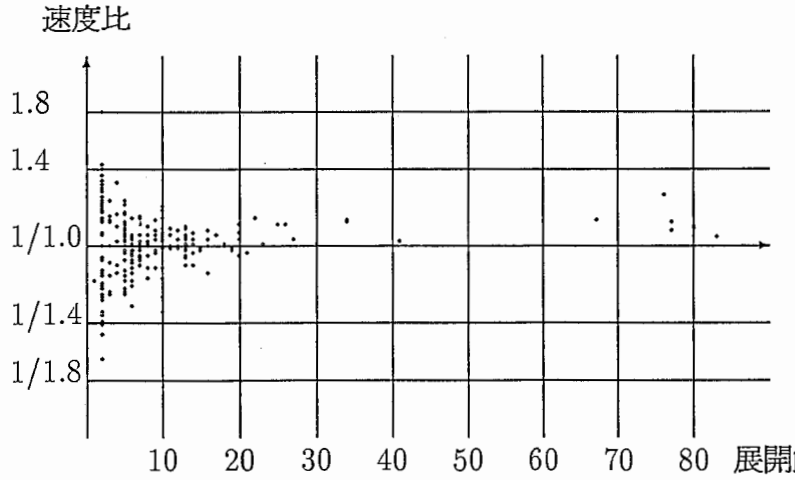

図 2.2: 英語コーパスに対する構造共有の実験結果

No.125	構造共有なし	構造共有あり	比
CPU(sec)	26.23	10.79	2.4
展開節点	642	239	2.7
多義節点	13	4	3.3
仮説木	300	76	3.9
共有 PD 連鎖	—	133(56%)	—
PD 再活性化	—	213(89%)	—

No.154	構造共有なし	構造共有あり	比
CPU(sec)	37.23	5.24	7.1
展開節点	688	183	3.8
多義節点	65	4	16.3
仮説木	332	54	6.1
共有 PD 連鎖	—	41(22%)	—
PD 再活性化	—	152(83%)	—

No.170	構造共有なし	構造共有あり	比
CPU(sec)	251.53	11.87	21.2
展開節点	2256	445	5.1
多義節点	256	25	10.24
仮説木	1107	118	9.4
共有 PD 連鎖	—	124(28%)	—
PD 再活性化	—	401(90%)	—

表 2.3: 独語コーパスに対する構造共有実験の個別例

多義ができてから初めて構造共有の仕掛が働くため、オーバヘッドを押さえることができたためと考えられる。
次に、実験結果の中から 2 、 3 の例を選んで個別に調べる (Table 2.3)。

Table 2.3から、 No.170 や No.154 の生成処理では単に PD 連鎖を再利用するだけでなく、展開に失敗した部分木を共有した結果、展開節点そのものを減らせた事が高速化に大きく寄与していることが分かる。一方で No.125 の生成では、 PD 連鎖の共有はかなり (56%) 起きているにも拘らず、他と比べてそれほど高速化されているとはいえない。
処理速度が何倍にもなるような著しい高速化は、主に失敗する部分木の共有によって達成されていると考えられる。

2.2.3 英語生成

Figure 2.2では展開節点の数と速度比の間には特に相関は認められない。
横軸の値が小さい所では速度比がばらついているが、独語と同じ理由でこのばらつきには意味がない。
構造共有は独語生成では劇的に効果があったが、英語生成ではほとんど効果が見られない。 Table 2.2を見ると、独語に比べて英語では中間多義と仮説部分木の数が非常に少ない。
構造共有は多義を多く含む場合に有効な手法であるから、英語生成に用いた PD のように中間多義をほとんど作らない規則には効果がないと考えられる。

独語	TopDown	BottomUp	比
CPU(sec)	49.90	1.03	48.4
消費セル (KB)	8,251	306	27.0
活性化 PD 数	58.21	58.65	1.0
適用 PD 数	9.59	9.59	1.0
中間多義数	1.82	1.40	1.3
部分木数	7.82	7.55	1.0

英語	TopDown	BottomUp	比
CPU(sec)	0.94	0.98	1.0
消費セル (KB)	212	229	0.9
活性化 PD 数	52.17	52.57	1.0
適用 PD 数	10.93	10.93	1.0
中間多義数	1.41	0.89	1.6
部分木数	3.76	3.30	1.1

表 2.4: PD 列探索方向の実験結果

2.3 PD 列探索方向

非終端節点を展開するために PD 列探索を行う際には、最大投射からヘッドに向かって探索する戦略と逆向きの戦略の 2 つが有り得る。2 つの戦略のどちらが有利かは PD 規則の書き方に依存する。
ヘッドに全ての制約を書いておく場合は、ヘッドから最大投射に向かって PD 列を探索した方が PD 列の失敗を早期に発見でき効率がよい (BottomUP)。制約がヘッドに限らず全ての PD に分散して記述されている場合は、最大投射からヘッドに向かって PD 列を探索した方が、展開節点に記述されている制約を利用しながら探索できる分有利になる (TopDown)。
独語生成用の PD は前者の例であり、英語生成用の PD は後者の例である。本処理系はどちらの方向もサポートしているが、特に BottomUp 方向の生成では 1 つの PD 列の中に同じ PD を 2 個以上含む事ができる (「PD の再帰」)。
独語生成用の PD は「PD の再帰」を利用しているため、独語コーパスの中には TopDown モードでは生成できないものが含まれている。

2.3.1 実験結果

独語コーパス (262 文中 154 文) および英語コーパス (280 文) に対する実験結果を Table 2.4および Figure 2.3、 Figure 2.4に示す。
表は全体の平均値を表している。
Figure 2.3は独語コーパスに対する実験結果を BottomUp 生成の処理時間を横軸にし TopDown と BottomUp の処理時間の比を縦軸にとって各文に対して点をプロットした。
一方、 Figure 2.4では英語コーパスに対する実験結果を BottomUp 生成の処理時間を横軸にし TopDown の処理時間を縦軸にして各文に対する点をプロットした。

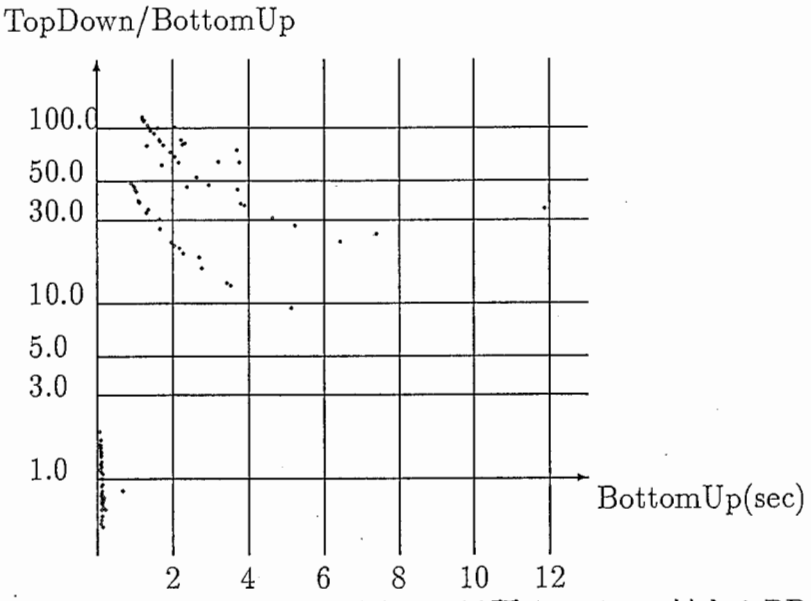

図 2.3: 独語コーパスに対する PD 列探索方向の実験結果

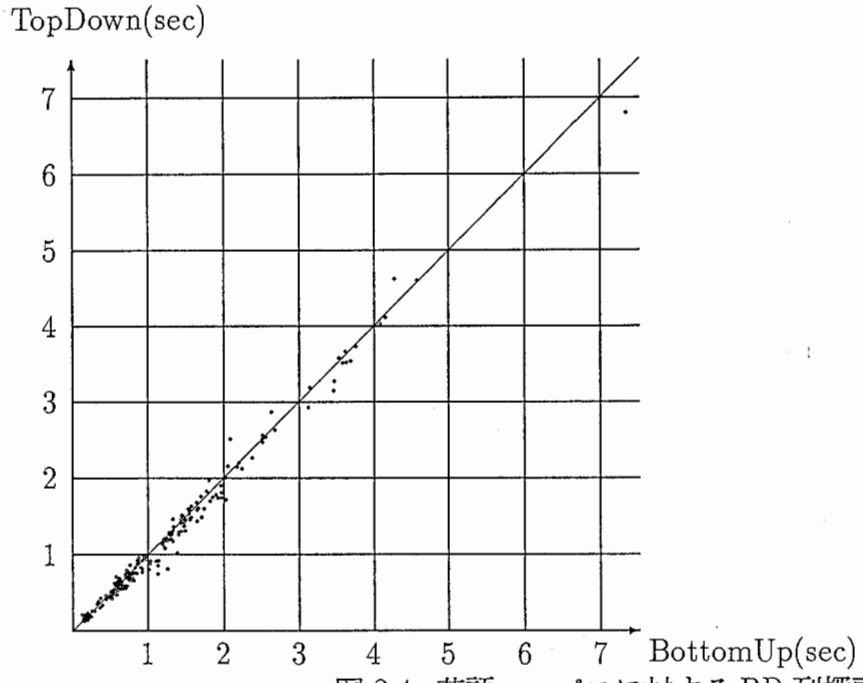

図 2.4: 英語コーパスに対する PD 列探索方向の実験結果

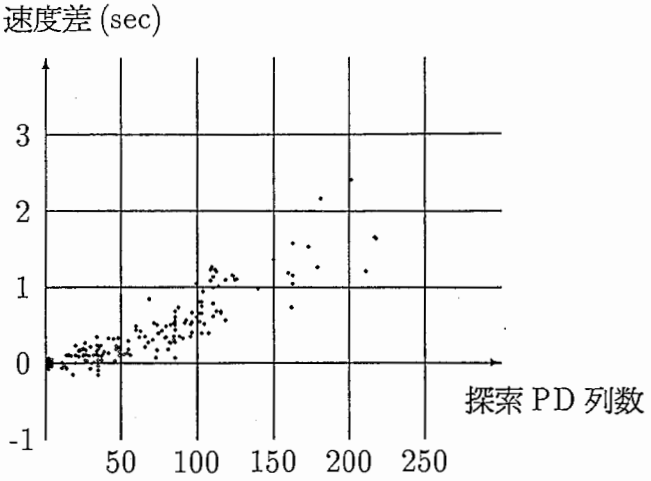

図 2.5: 独語コーパスに対する遅延コピーの実験結果

2.3.2 独語生成

Figure 2.3から１秒以上かかるような文は、TopDownで生成すると１０倍から１００倍も遅くなる事が分かる。
これはPDに書かれた制約が有効に働かず、PD列の探索空間が非常に大きくなるためである。

2.3.3 英語生成

Figure 2.4を見るとグラフ上の点はほとんど対角線上に乗っており、２つに戦略の間のパフォーマンスの差はほとんどない。
BottomUp生成に対する枝刈りの仕組みが充分に働いたためと考えられる。

2.4 大域的遅延コピー

PD連鎖を探索するときは、PD列の伸長に従って素性構造の単一化が多数回発生するが、単一化操作では素性構造のコピーに大きなコストがかかる。探索に成功したPD連鎖に対するコピーは避けられないが、途中で失敗したPD列の場合には素性構造のコピーは全く無駄になる。
そこで探索途中ではできる限りテンポラリフォワードを用いてコピーをせず、探索に成功したPD連鎖だけ改めてコピーを作るようにした。
生成処理を通じて探索に失敗するPD列が多ければ、無駄なコピーの分コストを削減できる。本方式は?? の考え方をPD列の生成というタスクに応用したものである。

2.4.1 実験結果

独語コーパスと英語コーパスに対する実験結果をTable 2.5およびFigure 2.5、Figure 2.6に示す。
表は全体の平均値を表している。
図は遅延コピーをしない場合の探索PD列の数を横軸にとり、処理時間の差（遅延コピーなし－遅延コピー有り）を縦軸にとって１文毎にプロットしてある。

独語	遅延コピーなし	遅延コピーあり	比
CPU(sec)	2.38	2.09	1.1
消費セル (KB)	1,108	640	1.7
展開節点数	40.2	40.2	1.0
探索 PD 列数	47.6	50.6	0.9
活性化 PD 数	95.6	95.6	1.0
適用 PD 数	14.8	14.8	1.0
中間多義数	3.0	3.0	1.0
部分木数	19.7	19.7	1.0

英語	遅延コピーなし	遅延コピーあり	比
CPU(sec)	1.09	0.98	1.1
消費セル (KB)	319	229	1.4
展開節点数	9.13	9.13	1.0
探索 PD 列数	12.28	12.28	1.0
活性化 PD 数	52.57	52.57	1.0
適用 PD 数	10.93	10.93	1.0
中間多義数	0.89	0.89	1.0
部分木数	3.30	3.30	1.0

表 2.5: 遅延コピーの実験結果

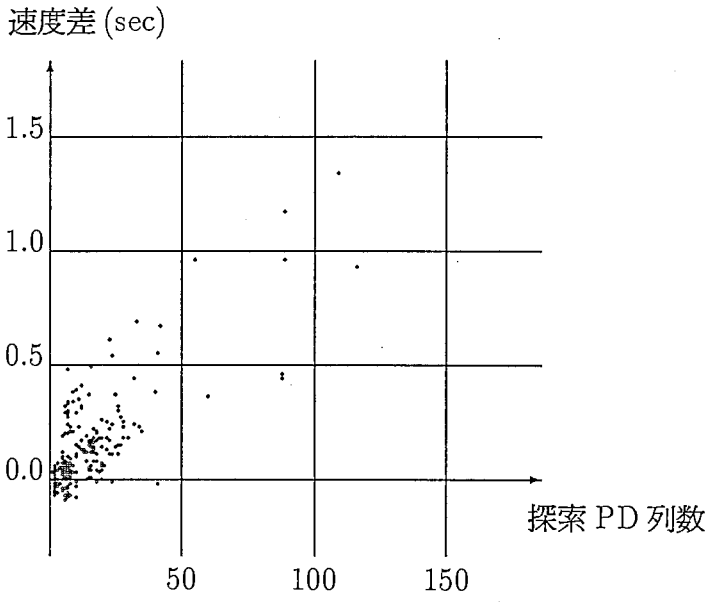

図 2.6: 英語コーパスに対する遅延コピーの実験結果

2.4.2 独語生成

Figure 2.5を見ると、探索PD列数と共に処理速度の差は増加する傾向にあり、失敗する探索PD列の数は探索PD列全体の数に比例すると考えれば、定性的には予想通りの結果といえる。
遅延コピーをしない場合の方が、丈夫な素性構造を使って一段深い探索PD列の絞り込みが行えるため、遅延コピーをする場合より探索PD列が少なくなっている。
しかし、オーバヘッドはあっても遅延コピーをした方が全体で約12%高速化されている (Table 2.5)。

2.4.3 英語生成

独語生成の場合同様、探索PD列の数にほぼ比例して処理時間が短縮され (Figure 2.6)、全体の平均では約10%高速化されている (Table 2.5)。

2.5 制約伝搬

非終端節点を展開するためにPD連鎖を作る時に、展開節点とPD列の素性構造を制約にしてPD列に対する枝刈りを行っているが、HPSG風の文法記述の場合は主要な制約はヘッドにかかれているために、制約の適用が遅れて枝刈りが有効に働かない。
そこで、展開節点、および、展開に関与する可能性がある各規則の素性構造から一定の条件を満たす素性を選び、これらを伝搬させることによって大域的な単一化可能性を検査し、探索空間を縮小する仕掛を導入した (制約伝搬)。
PD列探索の前処理としてPDの2項関係をネットワークで表現するが、この時に制約に基づいて無駄なアークを取り除き、PD列の探索回数を減らす。
生成処理を通じて探索されるPD列を減少できれば、これに応じて処理時間の短縮が行えるはずである。

2.5.1 実験結果

独語コーパスと英語コーパスに対する実験結果をTable 2.6およびFigure 2.7、 Figure 2.8に示す。
表はコーパス全体の平均値を表している。
図は制約伝搬をしない場合の探索PD列の数を横軸にとり、処理時間の差（制約伝搬なし − 制約伝搬有り）を縦軸にとって、1文毎にプロットしてある。

2.5.2 独語生成

Figure 2.7では、探索PD列数と処理時間差はだいたい比例関係にあり、予想通りであると言える。
平均ではPD列の探索数が44%削減され、その結果、処理時間は31%短縮された (Table 2.6)。

2.5.3 英語生成

Figure 2.8を見ると制約伝搬がほとんど効いていない。
英語のPDの場合主辞駆動型文法ではないので、この結果はある程度は予想された事である。

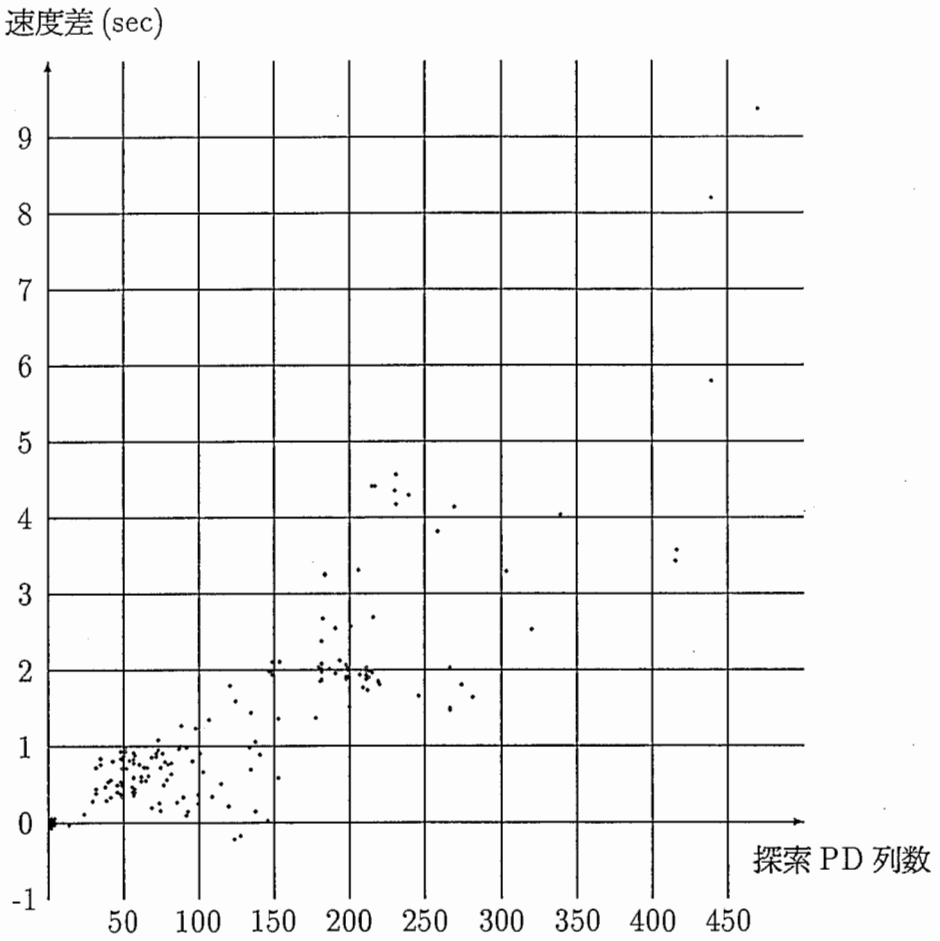

図 2.7: 独語コーパスに対する制約伝搬の実験結果

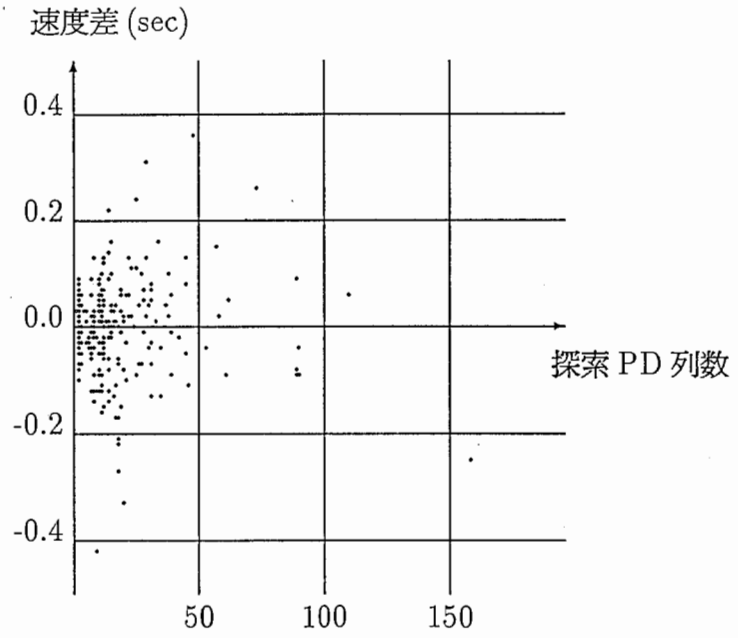

図 2.8: 英語コーパスに対する制約伝搬の実験結果

独語	制約伝搬なし	制約伝搬あり	比
CPU(sec)	3.04	2.09	1.5
消費セル (KB)	826	640	1.3
展開節点数	40.2	40.2	1.0
探索 PD 列数	89.8	50.6	1.8
活性化 PD 数	95.6	95.6	1.0
適用 PD 数	14.8	14.8	1.0
中間多義数	3.0	3.0	1.0
部分木数	19.7	19.7	1.0

英語	制約伝搬なし	制約伝搬あり	比
CPU(sec)	0.98	0.98	1.0
消費セル (KB)	221	229	1.0
展開節点数	9.13	9.13	1.0
探索 PD 列数	15.78	12.28	1.3
活性化 PD 数	52.57	52.57	1.0
適用 PD 数	10.93	10.93	1.0
中間多義数	0.89	0.89	1.0
部分木数	3.30	3.30	1.0

表 2.6: 制約伝搬の実験結果

2.6 PD の単一化可能性学習

. PD 連鎖を探索するとき PD 同士の単一化可能性に基づいて 2 項関係を作る。この単一化可能性の計算はダイナミックに行っているが、実際には静的な情報であり実行時にわざわざ計算する必然性はない。ただし、単純に全ての組み合わせを計算しておこうとすると、およそ (PD 数)^2 回の単一化計算と結果を保持するための記憶域が必要となる。これは不可能ではないにせよ実際的とは言えず、適当な戦略による可能性の絞り込みが必要である。
そこで実行時に計算した結果を覚えておき、その結果を再利用する事にした。つまり PD 間の単一化可能性をダイナミックに学習する訳である。もしも、2 項関係を実際に計算しなければならない PD の組み合わせが限られているならば、これは良い妥協になるはずである。
実験では次の 2 点を調べた。

- 処理性能への効果。
  予めコーパス全体に対する学習を行った後、学習結果を使う場合と使わない場合とで生成処理の性能の比較を行った。

- 学習の飽和性。
  コーパス全体に対して生成処理を行うときに学習された 2 項関係の数をモニターし、その積算値が適当な上限に収束するか否かを調べた。

2.6.1 実験結果

独語コーパスと英語コーパスに対する実験結果を Table 2.7および Figure 2.9、 Figure 2.10、 Figure 2.11、 Figure 2.12に示す。
表は全体の平均値を表している。

独語	学習モードなし	学習モードあり	比
CPU(sec)	2.54	2.09	1.2
消費セル (KB)	720	640	1.1
展開節点数	40.2	40.2	1.0
探索 PD 列数	50.6	50.6	1.0
活性化 PD 数	95.6	95.6	1.0
適用 PD 数	14.8	14.8	1.0
中間多義数	3.0	3.0	1.0
部分木数	19.7	19.7	1.0

英語	学習モードなし	学習モードあり	比
CPU(sec)	1.13	0.98	1.2
消費セル (KB)	257	229	1.1
展開節点数	9.13	9.13	1.0
探索 PD 列数	12.28	12.28	1.0
活性化 PD 数	52.57	52.57	1.0
適用 PD 数	10.93	10.93	1.0
中間多義数	0.89	0.89	1.0
部分木数	3.30	3.30	1.0

表 2.7: 単一化可能性学習モードの実験結果

Figure 2.9と Figure 2.10は探索 PD 列の数を横軸にとり、処理時間の差（非学習モード － 学習モード）を縦軸にとって 1 文毎にプロットしてある。
Figure 2.11と Figure 2.12はコーパス中の文 ID を横軸にとり、学習された PD 間の 2 項関係の個数の積算値を縦軸にしてプロットしてある。
Figure 2.13と Figure 2.14は、コーパス中の文 ID を横軸にとり、PD 間の接続可能性を調べるために行った単一化可能性の計算回数の積算値を縦軸にしてプロットしてある。

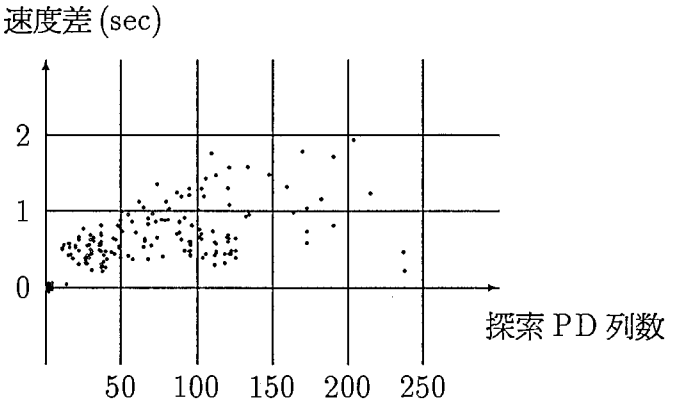

図 2.9: 独語コーパスに対する単一化可能性学習の実験結果

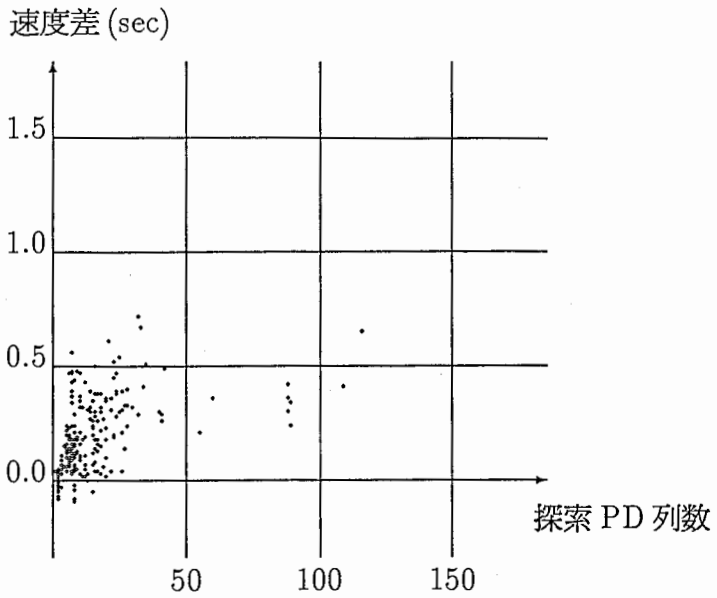

図 2.10: 英語コーパスに対する単一化可能性学習の実験結果

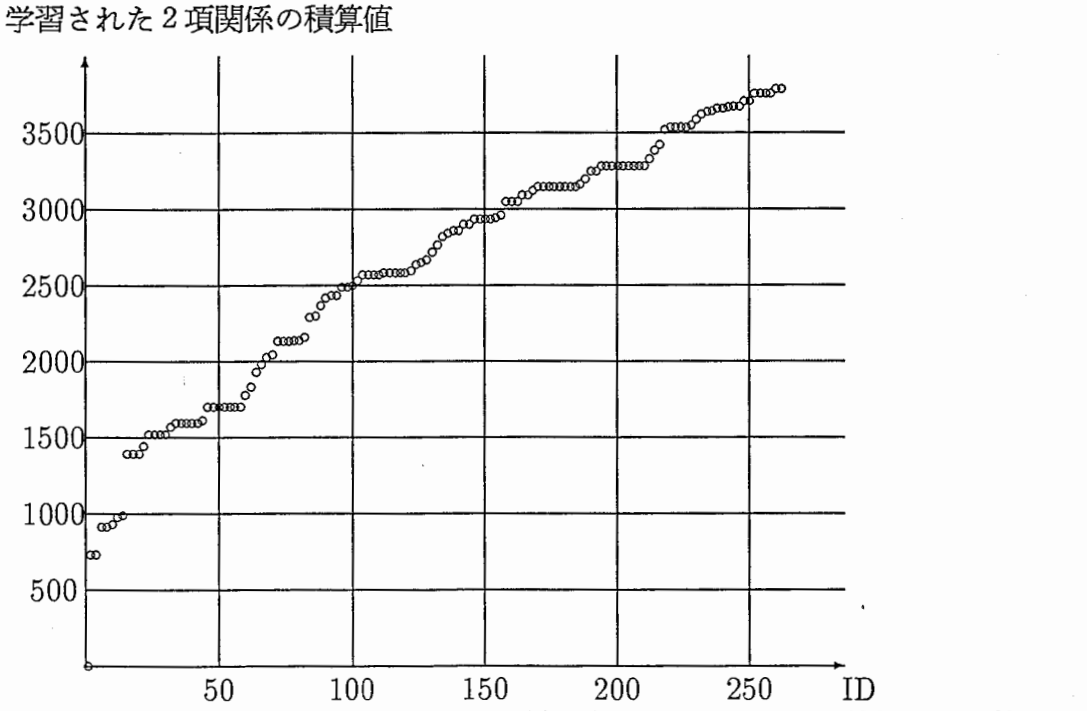

図 2.11: 独語生成における単一化可能性学習の飽和性

学習された2項関係の積算値

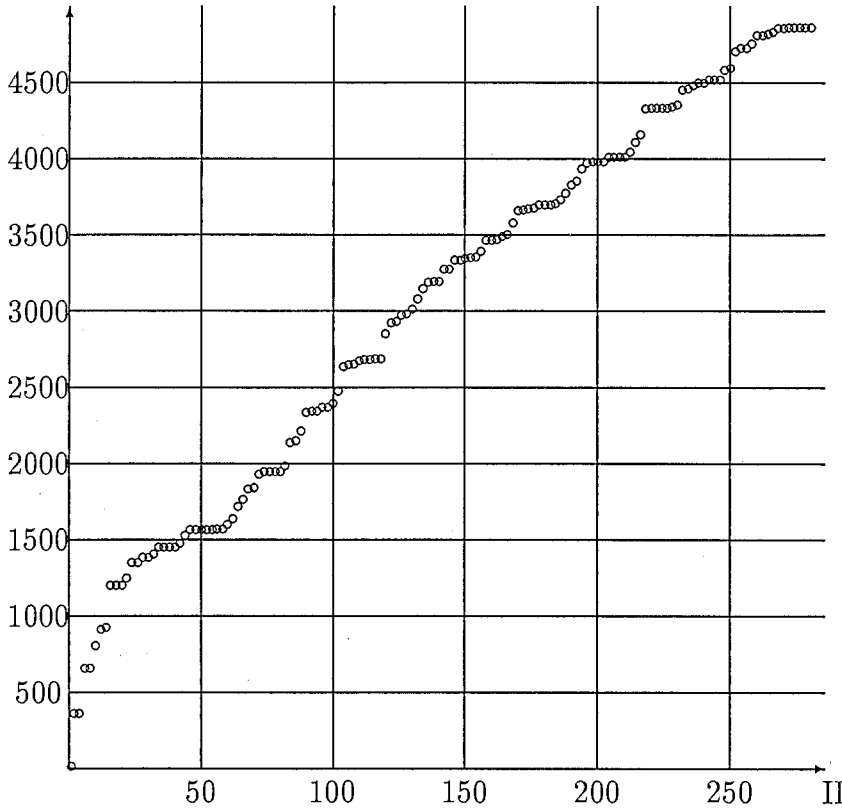

図 2.12: 英語生成における単一化可能性学習の飽和性

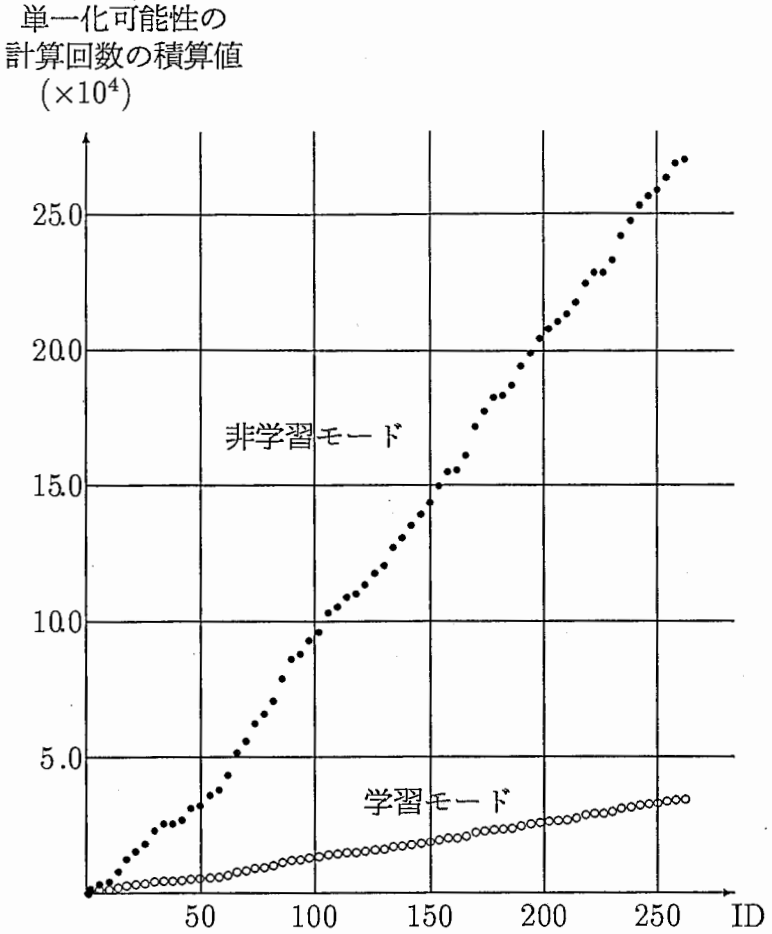

図 2.13: PD 間の接続可能性検査の負荷 (独語)

2.6.2 処理速度への効果

探索 PD 列が多ければそれだけ PD 間の単一化可能性計算の負荷が高いと考えられるが、独語生成でも英語生成でも、短縮された処理時間と探索 PD 列数との間には明瞭な関係は見いだせなかった (Figure 2.9および Figure 2.10)。しかし、独語生成でも英語生成においても、PD 間の接続可能性の検査のために行われる単一化可能性の計算は、学習を導入した事でかなり減っており (Figure 2.13、 Figure 2.14)、全体では単一化可能性の学習によって独語で約 18%、英語で約 13% の高速化が達成された (Table 2.7)。

2.6.3 学習の飽和性

最終的に独語で 262 文に対して 3,781 個、英語で 280 文に対して 4,859 個の 2 項関係が得られているが、上限値に収束しているとは言えない (Figure 2.11および Figure 2.12)。学習が飽和するためにはもっと大きなコーパスが必要なのであろう。
しかし生成に用いた PD 規則は独語 557 個、英語 780 個であり、単純には 309,692 個と 607,620 個の 2 項関係が有り得る。(自分自身との接続は許していないが、接続には順序があるので組み合わせではなく順列になる。)
つまりどちらも 1% 前後の計算量で学習が済んでおり、ダイナミックな学習機構を選択した効果は充分にあったと言って良い。PD が大規模になればこの差はもっと重要になると考えられる。

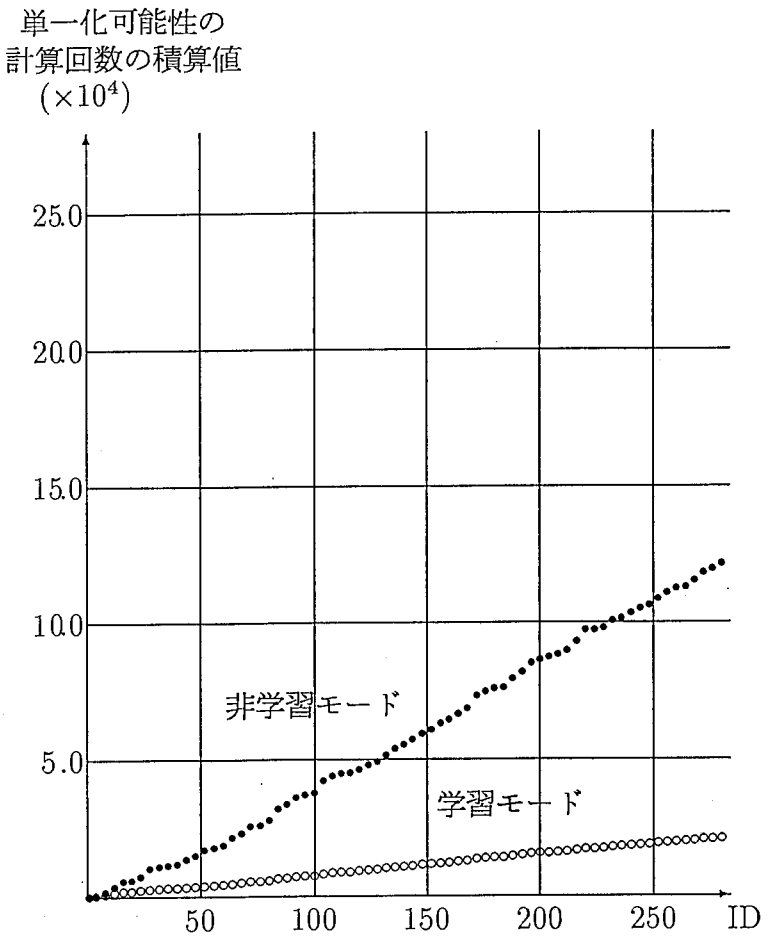

図 2.14: PD 間の接続可能性検査の負荷 (英語)

2.7 PD の 2 次記憶化

本処理系では PD(Phrasal Description) と呼ばれる生成知識に基づいて統語構造の生成を行っているが、個別の語彙に関する辞書情報も PD で記述されるので、コーパスの大規模化につれて PD を全て主記憶上に保持しておく事は難しくなる。
コーパスの大規模化に対応するために、PD にインデックスを付けて本体は 2 次記憶上に置く仕掛を実装した。頻繁にディスクから PD を読み出していたのでは処理速度が著しく低下するので、キャッシュ用のスタックを設けて LRU 制御を行うようにした。
PD の 2 次記憶化による処理性能への影響としては、

- ディスク I／O による速度の低下、
- 動的に PD を作るために余分に消費される CPU とセル、

などの不利な側面と

- 常時確保されるメモリの軽減による GC の減少、
- 充分なキャッシュサイズによるディスク I／O の低減

などの有利な側面が考えられる。
実験では次の 2 点を調べた。

- 処理速度への効果。
  キャッシュ用スタックのサイズを 100 に取り、主記憶版と 2 次記憶化版とで処理時間を比べた。

独語	2次記憶化	主記憶	比
CPU(sec)	2.70	2.69	1.0
消費セル(KB)	657	636	1.0
PDアクセス回数	589.4	70.3	–
キャッシュフォルト	5.7	—	–
活性化PD数	95.5	95.9	1.0
適用PD数	14.8	14.8	1.0
中間多義数	3.1	3.0	1.0
部分木数	20.0	19.7	1.0

英語	2次記憶化	主記憶	比
CPU(sec)	1.23	1.40	0.9
消費セル(KB)	240	225	1.1
PDアクセス回数	76.86	19.89	–
キャッシュフォルト	4.26	—	–
活性化PD数	52.59	52.55	1.0
適用PD数	10.93	10.93	1.0
中間多義数	0.88	0.88	1.0
部分木数	3.23	3.23	1.0

表 2.8: PD 2次記憶化の実験結果

- メモリ使用量。
  主記憶版と2次記憶化版のそれぞれに対して、一文生成毎に(room)関数で使用中のDynamic Storageを調べて、その変化を追いかけた。

2.7.1 実験結果

独語コーパスと英語コーパスに対して行った実験結果をTable 2.8およびFigure 2.15、Figure 2.16、Figure 2.17、Figure 2.18に示す。
表は全体の平均値を表している。
この実験内で条件を揃えるために、主記憶版の実験条件が他の項目での実験と異なっており、結果の平均値も他の項目での実験結果と多少ずれている。
主記憶版と2次記憶化版とで活性化PDや中間多義の数が異なるのは、PD連鎖が適用される順序がPDの活性化の順序に依存し、PD連鎖の適用順序が変わると(構造共有のために)展開節点の数も変わっていしまうためである。

Figure 2.15とFigure 2.16は主記憶版の処理時間を横軸にとり、2次記憶化版の処理時間を縦軸にとって1文毎にプロットしてある。
またFigure 2.17とFigure 2.18はコーパス中の文IDを横軸にとり、(room)関数で得られる使用メモリー量を縦軸にとって一文毎にプロットした。

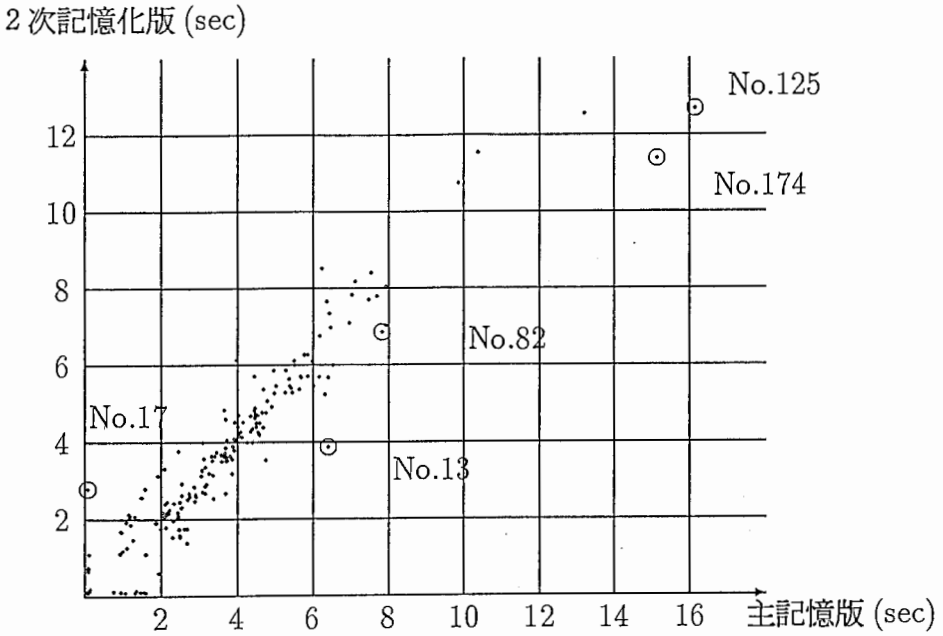

図 2.15: 独語コーパスに対する PD 2次記憶化の実験結果
2重丸の点はその文の実行中に GC が起きた事を示している。

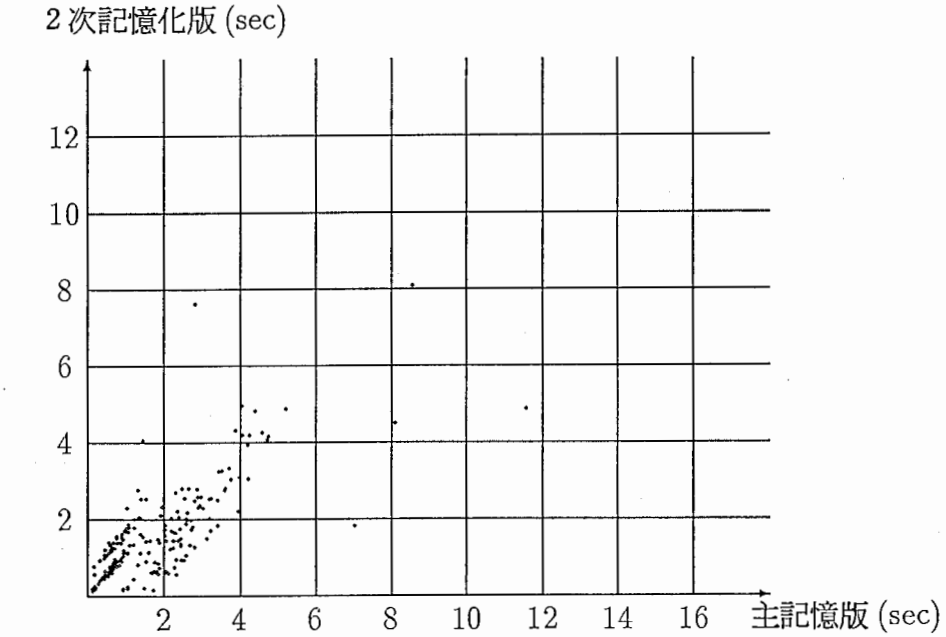

図 2.16: 英語コーパスに対する PD 2次記憶化の実験結果

2.7.2 処理速度への影響

独語の場合、Figure 2.15上の点はほぼ対角線上に乗っており、2次記憶化による処理速度の低下は見られない。No.13、No.125、No.174では主記憶版の実行中にGCが発生し、対応する点が2～4秒右にシフトしている。逆に、No.17では2次記憶化版でGCが発生している。No.82ではどちらもGCを起こした。
このGCによる処理時間の揺れはLISPのメモリマネージメントに依存しており、完全にはコントロールできなかったが、傾向は正しく反映していると思われる。

英語では独語のように明快な関係は見いだせなかったが(Figure 2.16)、平均値ではむしろ2次記憶化版の方が速くなっている(Table 2.8)。

2.7.3 使用メモリ量への効果

2次記憶化の効果としてFigure 2.17とFigure 2.18から顕著に分かる事は、2次記憶化によってメモリ使用量の絶対値、増加率共に半分またはそれ以下に軽減された点である。
即ち、PD 2次記憶化によって

- PDの大規模化に対応できるばかりでなく、
- 一度に処理できるコーパスを大きくする

事ができる。

2.7.4 メモリ不足を起こす原因に関する考察

Figure 2.17とFigure 2.18のどちらにおいても生成処理を繰り返すにつれて使用中のメモリ量が増えているが、本来ならば適当なコンスタントの周りを前後するはずである。この原因としては次の2つの事が考えられる。

- PDネット(PD検索用の多重索引)
  PD検索が起こる度に、PDネットのいくつかのノードに入力素性構造の部分構造がセットされる。この部分素性構造はPD検索終了時には不要になるが、実際には無効を示すフラグが立つだけで部分素性構造そのものはノード内に残ったままである。PDネットの全ノードに部分素性構造がセットされるまでノード内に捕らえられた部分素性構造は増加して行く。

- PD自身の素性構造
  PD自身の素性構造は、生成処理過程で一時的に展開節点の素性構造と単一化される。この時に素性構造のノード内には、一時的フォワードや補完アークなどがセットされる。これらは生成木の展開後には解消されるが、実際には単一化カウンタによって論理的に無効になるだけで、フォワード先の構造などはノード内に残されたままになっている。

この2つの内もしPDネットが原因ならば、解放されずに増大するメモリ量は主記憶版も2次記憶化版も同じ様に増え、同じように飽和するはずである。一方PD自身の素性構造が原因ならば、2次記憶化版ではPDそのものが解放されてしまえば、そこに捕らえられていた素性構造

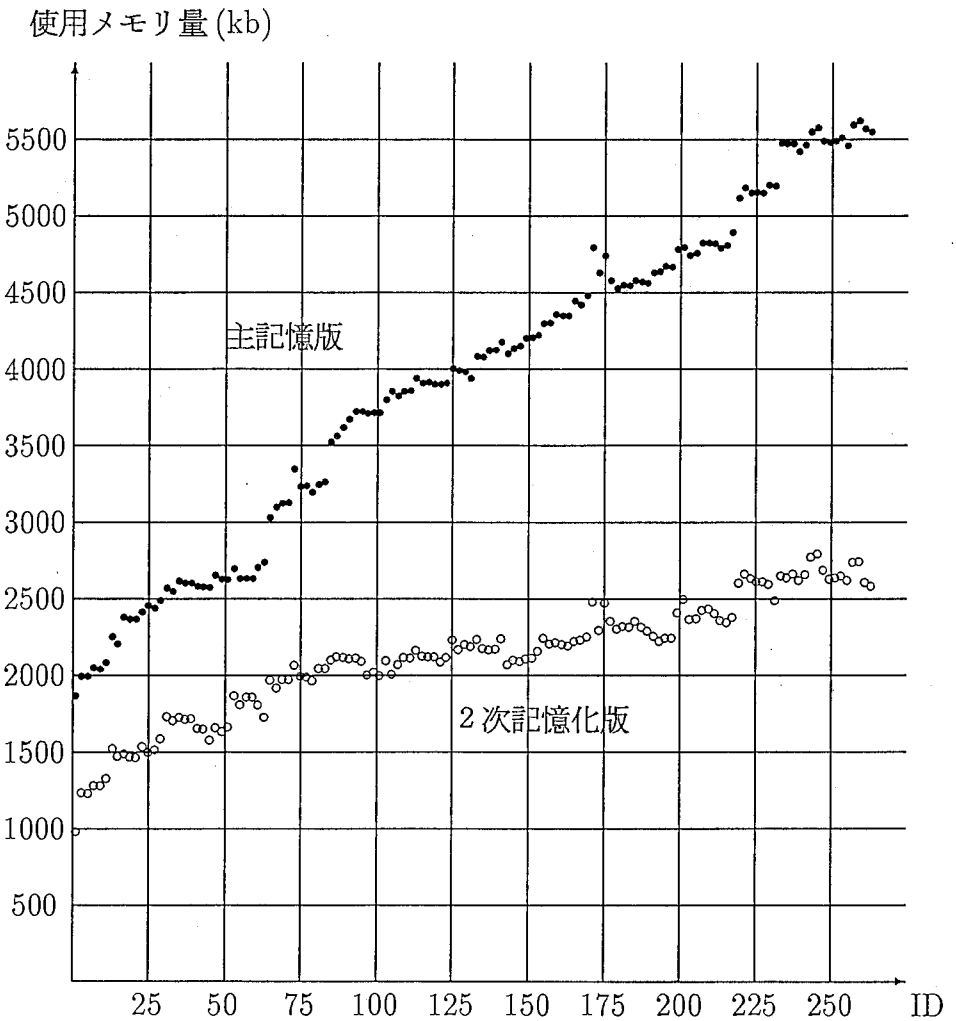

図 2.17: 独語コーパスに対する PD 2次記憶化のメモリ使用量への効果

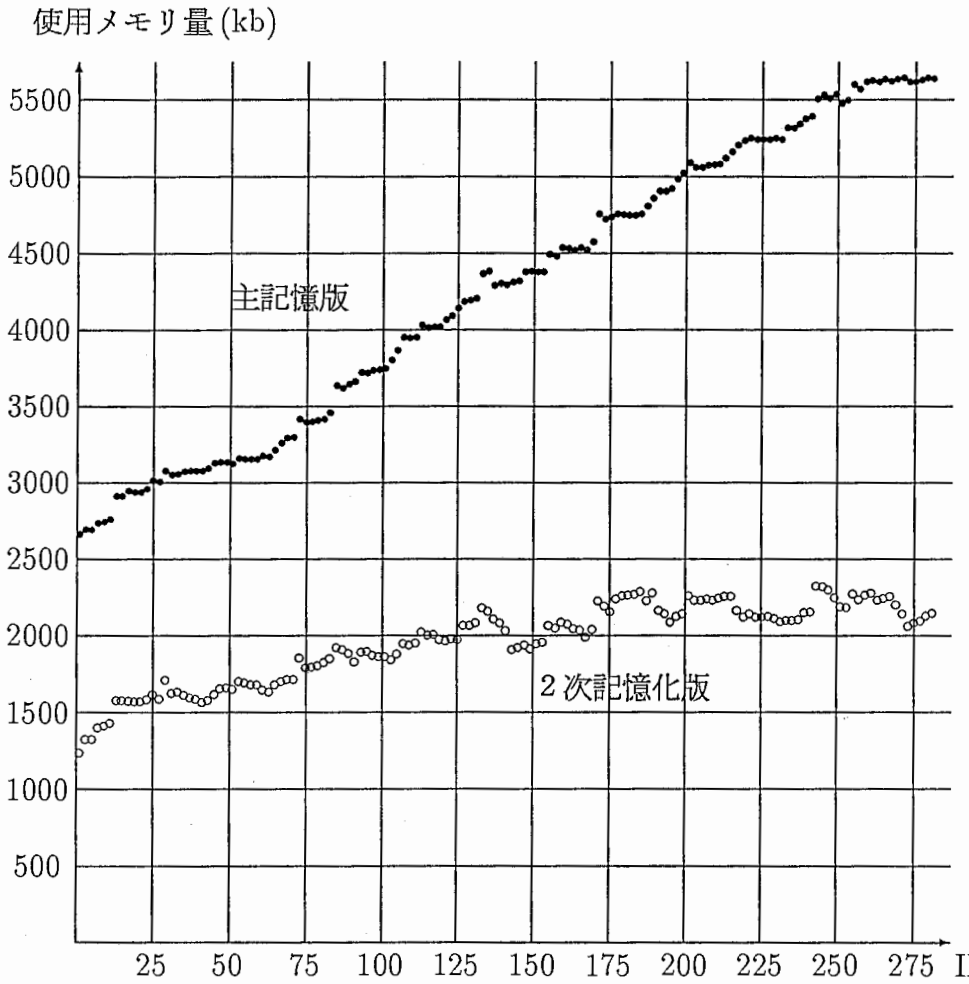

図 2.18: 英語コーパスに対する PD 2次記憶化のメモリ使用量への効果

	ファイル名	コメント
コーパス	eset12-15.loop	569 個 (93.1.21 版)
PD 規則	pd.smp.eset	1550 個 (93.1.22 版)
MG 規則	defrule.lisp	
MG ネット	defnet.lisp	(93.1.22 版)
形態素辞書	atr_made.dict7	(93.1.21 版)

表 2.9: 大規模文法実験に用いたファイルの一覧

も回収可能になるはずである。もちろん PD キャッシュ等の分は残るが、キャッシュが小さければすぐに飽和する。
独語でも英語でも主記憶版の方がメモリ使用量の増加率が大きい。従って PD 自身の素性構造に捕らえられている素性構造が原因である可能性が大きいが、今のところ確認できていない。

ここで候補として挙げた内容は、何れも不要になった部分素性構造を消す場合に、実際にスロットを空にする代わりに大域変数をインクリメントして論理的に無効にするテクニックが原因になっている。これは簡単かつ非常に高速に動作するため、このテクニックを直ちに捨て去る事はできない。
これはある意味では、速度とメモリ量の間のトレードオフという一般的な問題の一つと考える事もできる。

2.8 大規模文法への適用可能性

トイシステムで非常にうまくいくプログラムが、実用システムに移行しようとする時に直面する本質的な問題の一つは、語彙や文法の大規模化に伴うパフォーマンスの低下である。語彙数や文法規則数が増えると、必要になる記憶域や検索に対する負荷がリニアに増大するだけでなく、それらの組み合わせの可能性を調べるための処理量が普通は指数関数的に増大する。
本処理系ではこのような問題に対して、PD 規則の 2 次記憶化や PD 検索の多重索引化、構造共有などの対策をとっている。

2.8.1 実験の方法と条件

語彙や文法を拡大した場合の本処理系の有効性を調べるために、E セットコーパス (英語) と E セットコーパス用の PD 規則を使って実験を行った。
E セットコーパスには 569 個の変換結果素性構造があり、それに対して作成された PD 規則の数は 1550 個で、実際に生成処理の過程に適用された PD は 1044 個である。実験に使ったデータファイルを Table 2.9に示す。

この実験は PD 規則を順次増加させた時に生成処理系の性能がどう低下するかを見るのが目的だが、新たな PD 規則を追加するのは難しいので逆に既存の PD 規則を減らして処理系の性能の向上を調べる事にした。
既存の PD 規則を減らすために E セットコーパスを分割した。分割された各コーパスに対して必要な PD 規則は、元のコーパスの生成に必要な PD 規則に比べて少なくなる。分割しても結局は全コーパスに対して生成処理を行うので、コーパス中に特異な文が含まれていたとしてもそ

の効果は相殺され、PD 規則数に対する依存性だけが抽出できると期待できる。

実験の手順はおよそ以下に示す通りである。

1. コーパスを N 等分に分割する。

2. 分割されたコーパス毎に必要な PD を集めて、別々の小さな PD 規則ファイルを作る。

3. 分割されたコーパスそれぞれに対して、小さな PD 規則ファイルで生成処理を行う。

4. 分割されたコーパスに対して行った生成処理結果を集計する。

この手順で分割数 N を変えて実験を繰り返し、分割数毎に集計された生成処理結果から性能の変化を調べる。

実際には分割数を 1、2、4、8 と 4 通りに変えて実験を行った。
また各コーパスの生成に必要な PD とは、生成処理過程で適用された PD の事であるとした。つまり、最終結果に残らなくても木構造の生成に関与した PD は必要な PD であるとし、活性化はされても中間節点の展開に寄与しなかった PD は無視する事にした。

2.8.2 実験結果

E セットコーパスに対する実験結果を Table 2.10および Figure 2.19に示す。
Figure 2.19では、分割数毎に PD 規則数の平均値を求めて横軸にとり、生成処理にかかった処理時間の合計が縦軸にとってある。さらに分割数毎に最初の 71 個についての記憶域使用量の平均を求め、記憶域使用量として同じグラフにプロットした。(Figure 2.19の記憶域使用量と Table 2.10のセル消費量は別のものである。)

Table 2.10を見ると、569 個の入力に対する生成処理時間は 2000 秒前後である。一個あたり平均は 3.5 秒前後で、M セットコーパスの実験結果 (約 1 秒) に比べてかなり大きい。この差はおそらく PD 規則の適応度の違いから来ていると思われる。つまり、実験に使った M セット用の PD 規則は既に充分にチューンされていたが、E セット用の PD 規則は M セット用の PD 規則ほどにはチューンされていなかったと考えられる。

これを確かめるために生成多義が起きる頻度を調べてみた。M セットコーパス (280 個) と E セットコーパスで、中間多義の数、仮説木の数、生成結果の個数を比較したのが Table 2.11である。Table 2.11を見ると、全ての指標において E セットコーパスの方が多義の頻度が高い。もし、この多義性が PD 規則開発の進展につれて解消されるならば、E セットコーパスの生成でも高い性能が得られるようになると考えられる。しかし、M セット用の PD 規則が M セットコーパスに過剰に適応していたとも考えられ、独語生成実験の結果 (平均約 2 秒) も参考にすると、平均 2 秒前後が期待すべき性能であるかも知れない。

PD 規則の増加に対する処理系の性能の低下は非常に緩やかである。Figure 2.19を見ると、平均の PD 規則数が 299 から 1044 まで 3 倍以上増えているのに対して、処理時間は 1.15 倍、記憶域使用量は 1.09 倍しか増えていない。
さらに PD 規則数のオーダーを上げた場合の予測は難しいが、現在のオーダー (数千個) の範囲内ならば、本システムで充分対応できると考えられる。

	入力素性構造数	PD規則数	処理時間(sec)	セル消費量(KB)	出力英文数	中間多義数	仮説木個数
1分割							
001-569	569	1044	2273.76	450275	1155	2286	7164
合計	569	1044	2273.76	450275	1155	2286	7164
2分割							
001-284	284	747	1114.78	229566	605	1169	3627
285-569	285	673	1067.63	222932	550	1122	3537
合計	569	1440	2182.41	452498	1155	2291	7164
4分割							
001-142	142	476	549.65	114240	335	574	1659
143-284	142	507	506.90	111851	270	595	1964
285-426	142	436	525.94	112899	303	540	1759
427-569	143	450	505.61	110243	247	577	1770
合計	569	1869	2088.10	449232	1155	2286	7152
8分割							
001-071	71	320	208.33	44372	127	240	725
072-142	71	289	322.29	67536	208	331	920
143-213	71	332	273.40	61741	145	334	1172
214-284	71	312	201.34	45263	125	264	798
285-355	71	299	242.31	52234	141	271	980
356-426	71	255	254.45	55411	162	269	780
427-497	71	279	264.99	59243	133	306	1029
498-569	72	309	215.09	46298	114	273	746
合計	569	2395	1982.20	432099	1155	2288	7150

表 2.10: 大規模文法実験結果

コーパス	入力素性構造の個数	中間多義数の平均値	仮説木数の平均値	生成結果数の平均値
Mセット	280	0.89	3.30	1.15
Eセット	569	4.02	12.59	2.03

表 2.11: 生成多義の生起頻度

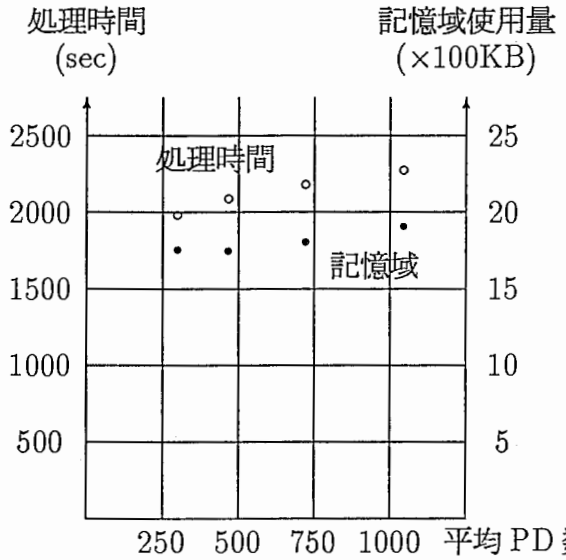

図 2.19: PD 規則数に対する生成処理系の性能の依存性

第 3 章

解析への適用可能性の評価

3.1 PD を使った解析手法

PD を CFG 規則に変換して CFG パーザで解析しようというのが、PD を用いて解析を行うために採用した基本的なアプローチである。

つまり PD の木構造のうち根節点と葉節点だけに着目し、それぞれの節点の素性構造を (syn lex) または (syn cat) の素性値で代表させれば、PD を CFG 規則に変換できる。この CFG 規則を使って解析した結果に基づいて PD を木構造に組み立て、それらを単一化すれば入力文に対応する素性構造が得られると考えた。

この解析処理系に対して、予想された問題や実装過程で明らかになった困難には次のようなものがある。

- (syn cat) が:ATOM 型以外の素性値を持つ節点がある。
- 空範疇をどう扱うか。
- PD の葉節点以外に埋め込まれている表層形をどうするか。
- 形態素解析。
- 循環 PD の扱い。
- 単節点 PD の扱い。

以下で、これらの問題に対して実施した対策と未解決のまま残った課題および現在のパフォーマンスについて述べる。

3.1.1 :ATOM 型以外の品詞

PD を CFG 規則に読み替えるために、節点の素性構造を (syn cat) の素性値で代表する事にした (ただし (syn lex) があればそちらを使う)。実際の PD 規則の中では (syn cat) のほとんどは:ATOM 型の素性値を持つが、中には:SET 型の素性値を持つ節点も幾つか存在する。さらに仕様上は (syn cat) の素性値が不定 (:NOT 型や:VAR 型) の節点も考えられる。このような節点を含む PD をどう扱うかに関しては、単純に考えて 2 つの立場がある。

- 全ての組み合わせを求める。
  即ち:NOT や:VAR は:SET に読み換え、:SET は更に:ATOM の組み合わせに変換する。こうすれば解析はできるが、CFG 規則数が増大して多義が増える。

- :ATOM 以外の品詞を持つ PD は無視する。
  CFG 規則数は最初の PD 規則数以下に押さえられるが、このような PD を必要とする文については解析に失敗する。

どのような立場を選ぶかは、PD 規則の書き方に対する制限にも依り理論的には決まらないが、今回の実装では以下のような中間的な立場を選んだ。

- :SET 型の素性値は:ATOM 型の組み合わせに変換する。
- :NOT 型と:VAR 型および素性パス (syn cat) および (syn lex) がない場合は無視する。

ただし:NOT 型や:VAR 型も:SET 型に書き換える事ができるようにするスイッチは付けた。

3.1.2 空範疇

生成処理系では空範疇を扱うために、スラッシュ終端と NP 制御を導入した。スラッシュ終端は 2 つの痕跡 (wh- 痕跡と NP 痕跡) を作るためであり、NP 制御は代名詞照応形 (PRO) のために用いられている。

解析処理系では、これらの空範疇を取り込むためにメタ規則を導入した。さらにメタ規則は外付けとし、スラッシュ終端条件や NP 制御条件と同様にメタ規則適用条件を文法記述者が記述できるようにした。

実験では取り合えず英語コーパスの痕跡 (trace) を扱うために、以下のようなメタ規則適用条件を与えた。

```
(def_slash_ana
  ((:node slash)
   "[[slash [[sem ?sl_sem][syn ?sl_syn]]]
     [sem ?sl_sem]
     [syn ?sl_syn]]"))
```

この条件は、「葉節点が素性パス (slash) を持ち、かつ上に示した素性構造と単一化可能であれば、その葉節点を取り除いた PD 規則を新たに合成せよ。」という意味である。

ところで英語の文法記述はスラッシュ制約が緩く、この条件だけでは本来スラッシュ終端しない葉節点にも適用されてしまう。メタ規則が大量に適用されると、解析多義が増えると同時に処理系への負担が非常に重くなる。

メタ規則の無意味な適用を避けるために、スラッシュ終端するはずがないと思われる節点を以下の条件で判別し、そのスラッシュ素性を無効にする機構を追加した。

- PD のスラッシュ素性の内、:VAR 型で相互参照されていないものがあれば、そのスラッシュ素性に:ATOM 型の素性値 '-' を与える。
- 移動変形の障壁のような節点を指定できるようにする。
  具体的には、PD の根節点の素性構造が def_barrier 宣言で指定された素性構造に包摂される場合、その根節点のスラッシュ素性に:ATOM 型の素性値 '-' を与える。

英語の解析実験で用いた障壁の定義を次に示す。

```
(def_barrier [[syn [[cat NP]]]])
```

この障壁は「NP を越えた移動は起きない」という主張を意味している。

3.1.3 PD の葉節点以外に埋め込まれている表層形

PD から CFG 規則を作る時には、原則として根節点と葉節点の (syn lex) および (syn cat) 以外の素性は関係がない。ところが PD 規則の中には、葉節点の (syn lex) 以外の素性から (他の素性構造との単一化を経由して) 葉節点の表層形を供給する PD が存在する。

例えば、英語 M セット用の PD 規則における
(syn signlex) (syn pp-lex) (syn pass-pp) (syn instprep) (syn loctprep)
などの素性パスがそうである。これらの素性パスは文法記述者によって特別に意味付けされたものであり、処理系が予めその意味を想定する事はできない。このような表層形を CFG 規則に反映するために、文法記述者が特別に指定した素性パスから表層形を集めて辞書型の PD を合成する機能を付けた。

3.1.4 形態素解析

PD の素性構造に記述されている表層形は形態素生成前の文字列であり、実際に解析処理系に入力されてくる屈折変化などを受けた文字列とは異なっている。このため解析処理系に入力されてくる文字列を、屈折変化や縮約などを受ける前の形に戻す処理を行う必要がある。

このような解析前処理 (形態素解析) を行うためには形態素生成規則や形態素生成辞書を解析知識として用いるのが本来であるが、今回の実装では取り合えず生成処理結果を辞書として利用する事にした。この目的のために、生成処理系に形態素のリストを出力するオプションを追加した。

また形態素解析は屈折前の文字列以外に時制などの情報も上位モジュール (解析処理系) に渡すのが普通であるが、現状では LR パーザとの I/F がないためにそれらの情報は単に捨てられている。

3.1.5 循環 PD

葉節点を 1 つしか持たない PD で、根節点と葉節点の品詞が同じものを循環 PD と呼ぶ事にする。

循環 PD を単純に CFG 規則に変換すると CFG パーザが停止しない規則ができる。そこで循環 PD の葉節点と非循環 PD の根節点とを単一化して、単一化に成功したら新たな PD を合成して循環 PD を消す様にした。

循環 PD 消去の手順の概要を次に示す。

```
foreach 循環PD {
    foreach 非循環PD {
        循環PDの葉節点を非循環PDの根節点と単一化する
        if 単一化成功 {
            単一化された素性構造に基づいて新しい非循環PDを合成する
        }
    }
}
新しい非循環PDに元々の非循環PDを加えて新たなPD規則ファイルを作る
```

3.1.6 単節点 PD

PD の中には節点が 1 つだけものがあるが、これを単節点 PD と呼ぶ事にする。単節点 PD にも非伝搬型 PD と伝搬型 PD がある。

	ファイル名	コメント
コーパス	mset12-24.loop	最初の 50 個 (93.1.20 版)
PD 規則	pd.smp.eset	(93.1.22) 版
MG 規則	defrule.lisp	
MG ネット	defnet.lisp	(92.11.4 版)
形態素辞書	atr_made.dict6	(92.12.17 版)

表 3.1: 英語解析実験に用いたファイルの一覧

- 非伝搬型単節点 PD
  この型の PD は個別の語彙に関する辞書がほとんどなので、その (syn cat) と (syn lex) を前終端記号と終端記号に割当てて通常の CFG 規則に変換する事ができる。

- 伝搬型単節点 PD
  伝搬型の単節点 PD は法などの特定の意味構造を被覆するために使われているが、今の方針では CFG 規則に変換できない。

伝搬型単節点 PD に相当する CFG 規則はなくても取り合えず解析可能なので今回は無視したが、そのような PD から供給されるべき意味素性を解析結果に反映する事はできなかった。

ただし伝搬型単節点 PD は生成処理系の要求を満たすためだけに存在している面があり、本質的な規則か否かには疑問がある。一方このような PD 規則が作られる背景には、変換結果の素性構造の各部分に必ずしも対応すべき表層表現があるわけではないという事実もあり、整合性を持った取扱いは簡単には決められない。

3.2 性能評価

3.2.1 評価対象

PD を使った解析処理の可能性を評価するために、英語の M セットコーパスを対象に解析実験を行った。英語の M セットコーパスは全体では生成結果が 322 文あるが、今の解析系の能力では M セットコーパス全体をカバーするような大きな文法は扱えない。そこで取り合えず M セット生成結果の最初の 75 文を対象にして評価実験を行う事にし、文法も対象となる 75 文に関与するものだけに絞った。

実験に使ったファイルなどを Table3.1に示す。

3.2.2 実験方法

解析処理系に対する評価項目として、次の 2 点を調べた。

- 解析失敗する文の個数とその原因。

- PD 規則の増加に伴う性能低下の程度。

PD 規則の増加に伴う性能低下を調べるために、「2.8 大規模文法への適用可能性」の時と同じようにコーパスを分割して実験を行った。解析実験では、コーパスの分割は 1 分割、2 分割、4 分割の 3 通りとした。

	PD 規則の個数	CFG 規則の個数	入力文数	出力個数	失敗文数	Elapsed Real Time (sec)	User Run Time (sec)
1 分割							
001-050	166	150	75	263	16	681.69	587.37
合計	166	150	75	263	16	681.69	587.37
2 分割							
001-025	133	122	36	122	8	309.37	263.93
026-050	131	123	39	66	8	166.23	119.93
合計	264	245	75	188	16	475.60	383.86
4 分割							
001-012	86	83	18	16	5	42.50	19.85
013-025	100	89	18	54	3	151.24	127.80
026-038	86	79	18	35	0	80.78	59.32
039-050	83	82	21	13	8	41.09	16.14
合計	355	333	75	118	16	315.61	223.11

表 3.2: 英語解析実験結果

3.2.3 実験結果

英語解析実験の結果を Table3.2および Figure3.1に示す。Table3.2において User Run Time の項目は LISP の TIME 関数で計測した値である。したがって大ざっぱに言って User Run Time は LISP 側の処理時間を示し、Elapsed Real Time と User Run Time との差が LR パーザの処理時間を示していると考えられる。

失敗文数には、入力文の内で解析結果を得られなかったものの個数を示した。この実験では解析結果の内容に関する吟味は行っておらず、厳密な意味では解析失敗はこれより多くなると考えられる。

解析失敗文

解析結果が得られなかった文は、入力 75 文中に 16 文であった。
以下に原因別に解析失敗文を示す。

- NP 障壁
  メタ規則の過剰適用を抑制するために導入した NP の障壁が、逆に必要なメタ規則の適用も抑制してしまった例が 3 例あった。

```
U-ID=No.23 :  What should I do?
U-ID=No.44 :  What kind of procedure should I make?
U-ID=No.44 :  What kind of procedure should I follow?
```

- 代名詞照応形 (PRO)
  代名詞照応形の空範疇を扱うためのメタ規則は、今回の実験では導入しなかったが、この現象を含む例が延べ 10 例あった。

```
U-ID=No.5  :  I want to apply for a conference.
U-ID=No.5  :  I want to apply for the conference.
U-ID=No.5  :  I'd like to apply for a conference.
U-ID=No.5  :  I'd like to apply for the conference.
U-ID=No.22 :  I want to attend the conference.
```

```
U-ID=No.22 :  I'd like to attend the conference.
U-ID=No.43 :  I want to apply for a conference.
U-ID=No.43 :  I want to apply for the conference.
U-ID=No.43 :  I'd like to apply for a conference.
U-ID=No.43 :  I'd like to apply for the conference.
```

- AdjunctionPD
  生成処理系には通常の PD の他に付加操作のための PD(AdjunctionPD) が存在するが、現在のところ AdjunctionPD を解析用の規則に変換する機能はない。この AdjunctionPD が足りないために解析できなかった例が延べ 3 例あった。

```
U-ID=No.2  :  Is it the conference office?
U-ID=No.39 :  Is it the conference office?
U-ID=No.42 :  What kind of business is it?
```

処理時間の PD 規則数に対する依存性

Figure3.1からは、PD 規則の増加に伴って処理時間が急激に増大している事が分かる。出力される解析結果の個数も同様に増えており、規則の増加に伴う多義性が性能低下の大きな原因になっているらしい。

多義性の原因としては次のような事が考えられる。

1. 形態素解析で得られる時制などの情報を取り込むための I/F がない。
   時制や代名詞の格、名詞の単複などが不定なまま解析を行うために、中間多義ばかりでなく解析結果の多義も増大する。

2. 空範疇のためにメタ規則を導入した。
   メタ規則によって合成される PD 規則が中間多義の原因になるのは勿論であるが、合成された PD 規則の持つ制約が緩ければ解析結果の多義の原因にもなる。

3. LR パーザでは素性構造の (syn cat) 以外の制約が利用できない。
   単一化できないような PD の組み合わせでも、LR パーザの範囲では品詞の整合性さえあれば生き残ってしまう。このような多義は単一化の過程で解消されるが、中間多義が大量にできれば性能低下の原因となる。

今回の実験でプロットした点はわずかに 3 点だけであり、PD 規則をさらに増やした時に処理系がどのように振る舞うかは分からないが、多義性を抑制するための機構を導入する必要性は明らかである。

どうやって多義性を抑制し処理系の性能低下を防ぐかという点に関してはまだ本格的な検討はなされていないが、

- ASURA の解析系を流用する。
- 素性構造を取り込めるように LR パーザを拡張する。

などのアイディアがある。

PD 規則は単一化に基づく句構造文法を記述するために設計されており、理論的には容易に ASURA の解析系に処理させる事ができるはずだが、ASURA の解析系を流用する場合の難点は素性構造の仕様が異なる点にある。ASURA 解析系は終端記号をストリングではなく LISP のアトムとして扱うので、アトム値として使えない終端記号があると問題である。生成系で導入した特別な素性値の扱いも問題になる (:SPECIFIED :NULL など)。

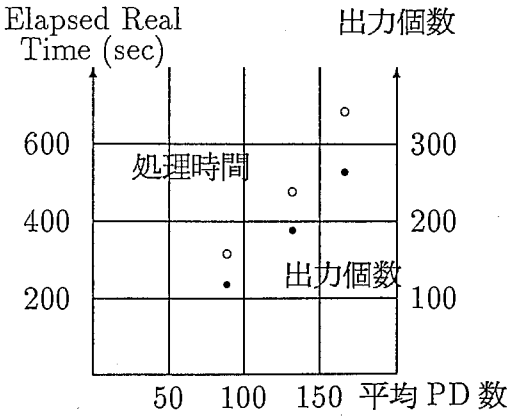

図 3.1: PD 規則数の増加に対する解析処理系の性能低下

また性能に関する予測は、ASURA 解析系と実際の PD 規則の詳細に依存するため難しい。

LR パーザを拡張して素性構造を取り込めるようにする場合、LR パーザ側 (C 言語) と単一化ルーチン側 (LISP) との間の構造変換が頻繁に起こる。LR パーザと単一化ルーチンとの融合の程度にも依るがオーバーヘッドは避けられない。

この拡張を行った場合は形態素解析から得られる時制や単複の情報が多義の解消に使える他に、多品詞語に起因する多義も部分的に解消できる可能性がある。また LR パーザから単一化を呼び出すようにすれば、PD の素性構造に記述されている制約が早い段階で利用できるために、中間多義が減って性能の向上も期待できる。

さらに現在はメタ規則で対応している痕跡など空範疇についても、空範疇を束縛するものが必ず空範疇より前に現れるならば、生成処理系と同じメカニズムで扱える可能性がある。

3.3 まとめ

LR パーザに生成処理系の単一化機構を組み合わせて、英語生成用に作成された生成規則 (PD) を用いた解析実験を行った。本実験は生成と解析の双方向に適用可能な生成規則を、PD の枠組みの中で開発できる可能性を示したものである。

しかし一方で、伝搬型単節点 PD の問題や形態素解析での時制情報の欠落などの難点も残した。また PD に記述されている制約が LR パーザで活用できないために起こる中間多義の増大も問題である。

これらの問題を解決するための手段として、次の 2 つのアイディアを提案した。

- ASURA の解析系を流用する。
- 素性構造を持てるように LR パーザを拡張する

ASURA 解析系の流用は理論的にはスジの良いアイディアであるが、難点としては素性構造の仕様の違いが挙げられる。

LR パーザを拡張する場合は、LR パーザから単一化ルーチンを呼び出す事で PD の制約がフルに利用できるが、LR パーザと単一化ルーチンとの間のデータ交換のオーバヘッドが予想される。

何れにせよ、これらのアイディアはまだ充分に検討されたものではなく、今後の課題として残されている。

参考文献

[1] Pollard,C. et al., "Information-based Syntax and Semantics Volume 1 Fundamentals", CSLI, 1987

[2] Ueda, Y., "タイプ付き素性構造主導生成",ATR technical report, TR-I-0206

[3] Shieber, S.M. et al., "A Semantic-Head-Driven Generation Algorithm", In Proceedings of 27th ACL, 1989

[4] Vijay-Shanker, K. et al., "Feature Structure Based Tree Adjoining Grammars", in Proceedings of COLING'88, 1988

[5] Kikui, G.,"Feature Structure Based Semantic Head Driven Generation", In Proceedings of COLING'92, 1992

[6] Tomabechi, H., "Quasi-Destructive Graph Unification", In Proceedings of 29th ACL, 1991.