TR-IT-0048

ASURAにおける英語構文生成処理系解説書
- 付：英語構文生成知識記述マニュアル -
The Syntactic Generation Module of ASURA (User's Manual)

菊井 玄一郎
Gen-ichiro KIKUI

1994.3

梗概

ASURAの生成処理は意味素性構造を入力としてこれに対応する統語構造を生成する処理である。この処理では生成知識としてあらかじめ用意された素性構造付きの木を意味構造に応じて組み合わせることによって統語構造を作成する。本資料では生成知識の記述者を対象として、生成処理の考え方、および、生成知識を記述する形式について解説する。また、システムを導入し起動する方法についても述べる。

Abstract

The generation module of ASURA creates syntactic structures from a given semantic feature structure by referring to linguistic knowledge, which is represented as a set of trees with feature structures (like feature-structure-based tree-adjoining grammars).
This report introduces the basic concept of the module and gives the syntax of the linguistic knowledge used. It also describes how to install and use the module.

ATR 音声翻訳通信研究所
ATR Interpreting Telecommunications Research Laboratories

For those who don't read Japanese:

- For those not familiar with unification-based grammar formalisms, an introductory book such as [1] is recommended.
- The second chapter introduces the generation algorithm. If you are interested in the algorithm, Appendix A provides almost the same information (but in more detail).
- The 3rd, 4th and 5th chapters are reference manuals. Please try to guess the contents from examples and from brief descriptions in English (English descriptions are italicized and marked with ◇ in the margin.).

もくじ

第 1 章

はじめに (Introduction)

生成処理とは文法・辞書などから成る生成知識を用いて、与えられた内部表現 (意味表現と呼ぼう) に対応する統語構造を作り出す処理である。本処理系では生成知識として素性構造付き木構造 (各節点に素性構造を持つ木構造) の集合を用いる。従って、本処理系において、生成処理とは生成知識中の木を組み合わせて入力意味表現に対応する統語構造 (木) を作成する処理のことである。

複数の木を組み合わせる方法として本処理系では「代入 (substitution)」と「付加 (adjunction)」とを用意している。これらの方法によって素性構造の付いた木を組み合わせて統語構造を規定する方法は、「単一化に基づく木結合文法 (Feature structure-based Tree Adjoining Grammar:FTAG)[2]」と呼ばれる文法記述の方法と同等なものであり、「単一化に基づく文脈自由文法」を含む[1]強力な記述力を持つ。

本資料はこの処理系のための生成知識を作成しようとする人を対象として生成知識、および、生成処理の概要 (2 章)、生成知識を記述するためのシンタックス (3 章) について解説する。また、生成処理系の利用法 (4 章) 生成知識記述のデバッグツール (5 章) についても説明する。

> 本資料は素性構造の単一化に基づく文法記述についての基礎的な知識を前提としている ([3][1] など)。さらに、主辞駆動型句構造文法に関する知識あればなお望ましい (たとえば [4])。

[1] 上位コンパチ

第 2 章

生成処理の概要[1]

2.1 木の代入と付加

第 1 章で述べたように、生成処理とは生成知識として記述されている複数の木を組み合わせて、入力意味構造に対応する木構造を作成する処理である。生成知識として記述されている各木のことを 要素木 (elementary tree) という。ある要素木を別の木と組み合わせる方法は「代入」または「付加」のどちらかであり、いずれの方法が使えるかは要素木ごとに決められている。

- 代入 (substitution)

  代入とはある木 (T1) の葉節点 (n1) に別の木 (T2) （の根節点 (n2)) を接続する操作である （図 2.1) 。代入に与る二つの節点 (n1,n2) の素性構造は単一化される。従って、接続されてできる節点 (n3) の素性構造は、n1,n2 の素性構造を単一化したものである。なお、他の木の (葉) 節点に代入できる要素木を初期木 (initial tree) と呼ぶ。

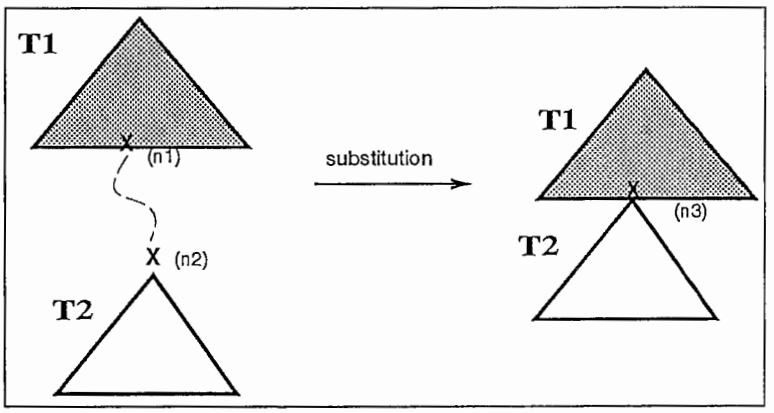

図 2.1: 代入 (Substitution)

- 付加 (Adjunction)

  付加とはある木構造 (T1) のある節点 (n1) に別の木構造 (T2) をはめ込む操作である （図 2.2) 。他の木の付加を許すような節点 (n1 のような節点) を 付加節点 (adjoining node) と呼ぶ。付加節点には上側

[1] 処理の詳細については A も参照されたい。

素性 (top feature)、下側素性 (bottom feature) という二つの素性構造を与える。他の木に付加できる木 (T2 のような木) を付加木 (auxiliary tree) と呼ぶ。付加木の葉節点のうち付加先の構造と接続すべき節点を foot 節点と呼ぶ。T2 を T1 の付加節点 (n1) に付加する時には、n1 の上側素性と T2 の根節点の素性構造を単一化し、n1 の下側素性と T2 の foot 節点の素性構造を単一化する。従って、n4 の素性構造は n1 の上側素性と n2 の素性構造を単一化したものであり、n5 の素性構造は n1 の下側素性と n3 の素性構造とを単一化したものとなる。

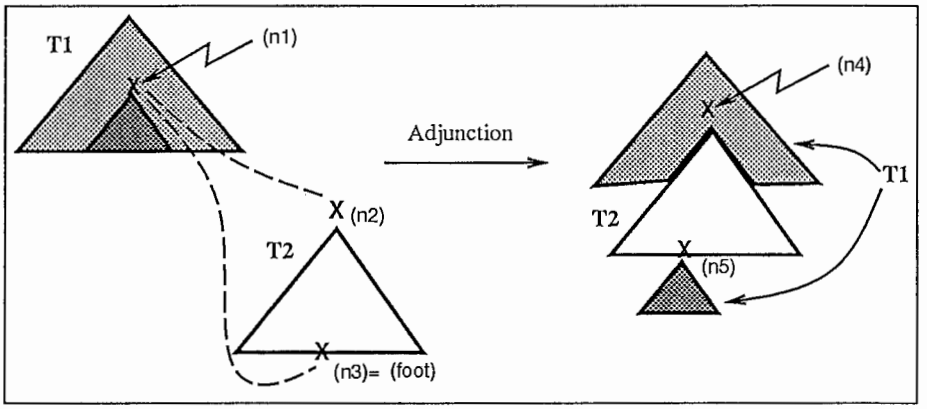

図 2.2: 付加 (Adjunction)

代入や付加で二つの木構造を組み合わせる際には、その時に行なわれる素性構造の単一化に成功しなければならない。もし失敗した場合、そのような組み合わせ方は不適切なものとみなされ排除される。すなわち、どのような木構造の組み合わせが可能であるかは、木構造の各節点の持つ素性構造に依存する。生成知識が適切であるためには、文法的に正しい構文構造の組み合わせのみが許されるような (素性構造付き) 木構造の集合が定義されていなければならない。

2.2 代入のみを用いる生成

ここでは、まず、代入のみを用いる生成について説明する。付加を含んだ生成については次節で説明する。

2.2.1 代入のみを用いる生成知識

代入のみの許された生成知識記述は HPSG に代表される「単一化に基づく文脈自由文法」と同等の記述形式となる。このような生成知識の例を図 2.3に示す。

図 2.3の生成知識は Tree1 から Tree10 までの 10 個の要素木からなっている。各要素木は親節点と子節点のみからなる木に限られているので、親節点を左辺、子節点の並びを右辺とするような (素性構造付きの) 文脈自由文法の書き換え規則とみなしても良い。各木の節点の横の点線に囲まれた素性構造はその節点に付与されている素性構造である。素性構造中の "?" で始まる記号は変数 (あるいは tag) であり、一つの木の中で同一の変数は同一の素性構造を表す。また、図 2.3の各木構造の節点には品詞記号が与えられているが (たとえば Tree1 では、S,XP,VP)、これは読み易さのためにその節点の素性構造の syn,cat 素性の値を記したものであり、各節点の素性構造と独立に定義されているわけではない[2]。なお、syn,cat 素性の一意に定まっているもの (Tree1 の S や VP など) は素性の値を記述しているが、素性構造内で品詞が規定されていない場合 (たとえば、Tree1 の XP) には仮に XP と記している。

[2]PATR 記法 [1] とは異なる

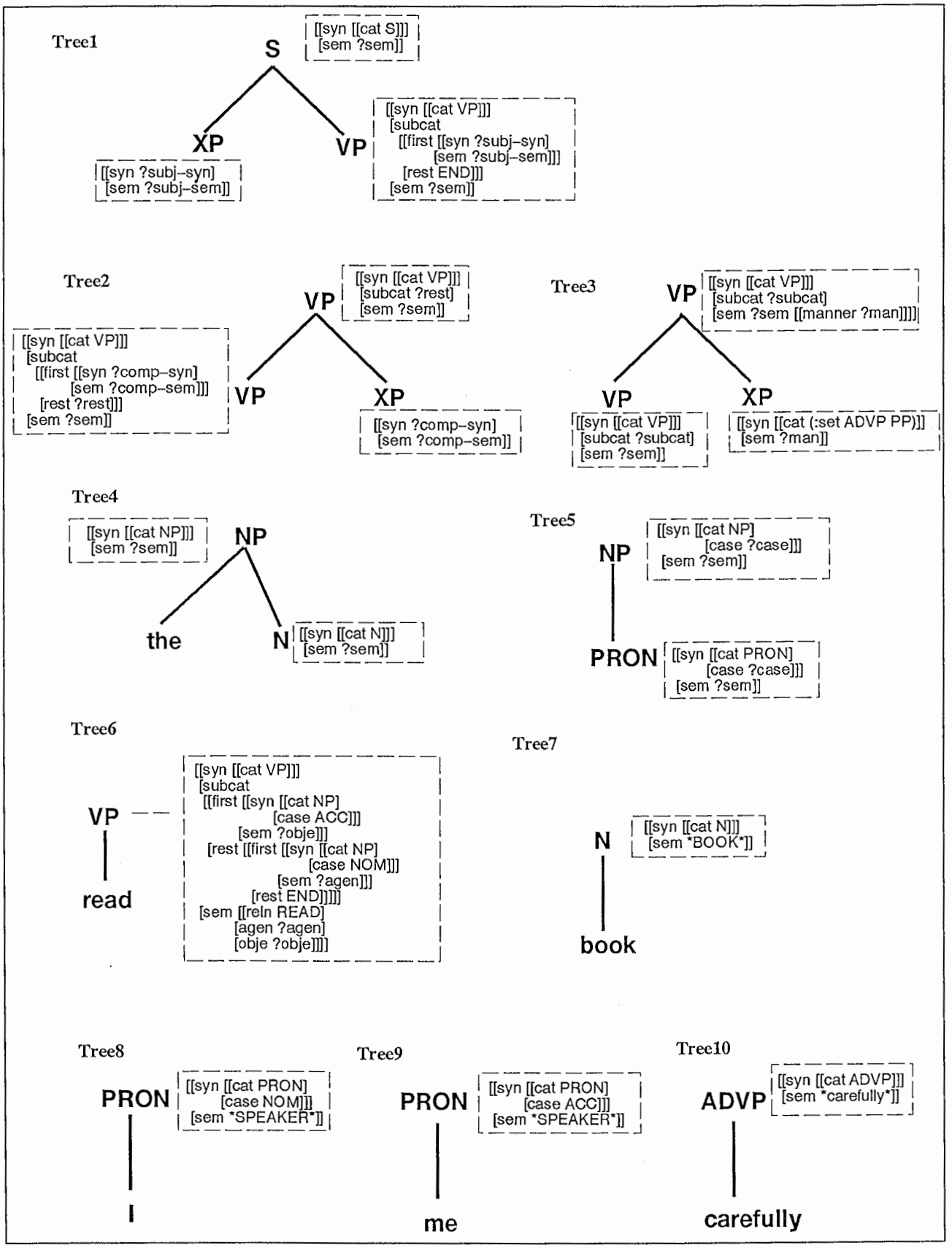

図 2.3: 規則の例 (an example of a rule set)

以下、それぞれの木について簡単に説明する。

Tree1 は親節点の品詞が S であり二つの子節点を持つ木構造である。左側の子節点は統語素性、意味素性ともに、右側の子節点の subcat 素性の first 要素の統語素性、および、意味素性と同一である。右側の子節点は品詞が VP であり、意味素性が親節点の意味素性と同一であるような素性を持つ。

Tree2 も親節点と二つの子節点とからなり、左側の子節点は品詞として VP、意味素性は親節点の意味素性と同じである。この節点の subcat 素性は first と rest とからなっており、first 素性はこの規則の右側節点の素性構造と同一であり、rest 素性は親節点の subcat 素性と同じである。すなわち、VP 節点に与えられるの subcat 素性のうち first の部分が右側節点の統語素性、意味素性と同一であり、rest の部分が親節点の subcat 素性と同一である。

Tree3 は動詞句に対する付加語 (副詞的修飾句) の構造を表す木である。親節点と二つの子節点とからなり、左側の子節点は品詞として VP、意味素性 subcat 素性、統語素性は親節点のそれぞれの素性と同じである。右側節点の品詞は副詞句または前置詞句、意味素性は親節点の manner 素性の値である。

Tree4, Tree5 はそれぞれ名詞、代名詞と名詞句とを関係づける規則である。

Tree6 は動詞 “read” の持つ統語・意味的性質を表す木構造であり、子節点に単語文字列 “read”、親節点にこの単語の素性構造が与えられている。素性構造の内容は、品詞が VP、意味構造が「?agen が?obje を読む」という情報に対応する。?agen や?obje などの変数は後の処理の過程でバインドされる (埋められる)。subcat 素性はこの動詞に対する必須要素 (補語) に関する情報を表す。この素性は first,rest を再帰的に用いたリスト構造となっており、第一要素が「品詞が名詞句で、意味が?obje の内容であるような言語表現」に対応し、第二要素が「品詞が名詞句で意味が?agen であるような言語表現」に対応する。sem 内の agen,obje 値は空であるが、これらの値が適切な値に定まれば subcat 内の各要素の sem の値も定まることに注意しよう。

Tree7,Tree8,Tree9 は名詞に関する単純な要素木であり、Tree10 は副詞 carefully に関する要素木である。

ここで、木構造の代入について例を用いて復習しておこう。Tree5 の葉節点に Tree8 を代入すると、図 2.4のような構造になる。

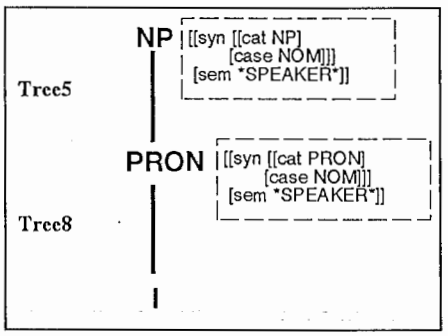

図 2.4: 木構造の組合せ

さて、生成処理は生成知識に含まれる要素木を代入操作によって組み合わせて次の条件を満たす木構造を作成する。

1. 木構造全体の意味構造 (根節点の意味構造) が入力意味構造と同一である
2. 葉節点がすべて語彙である (語彙には空語彙[3]も含まれる)

たとえば、図 2.5のような入力素性構造が与えられたとき、図 2.6のような組み合わせがこの条件を満たす構造である。

[3]関係節化などによって表層上欠損している要素

```
[[reln READ]
 [agen *speaker*]
 [obje *book]
 [manner *carefully*]]
```

図 2.5: 入力意味素性構造 1

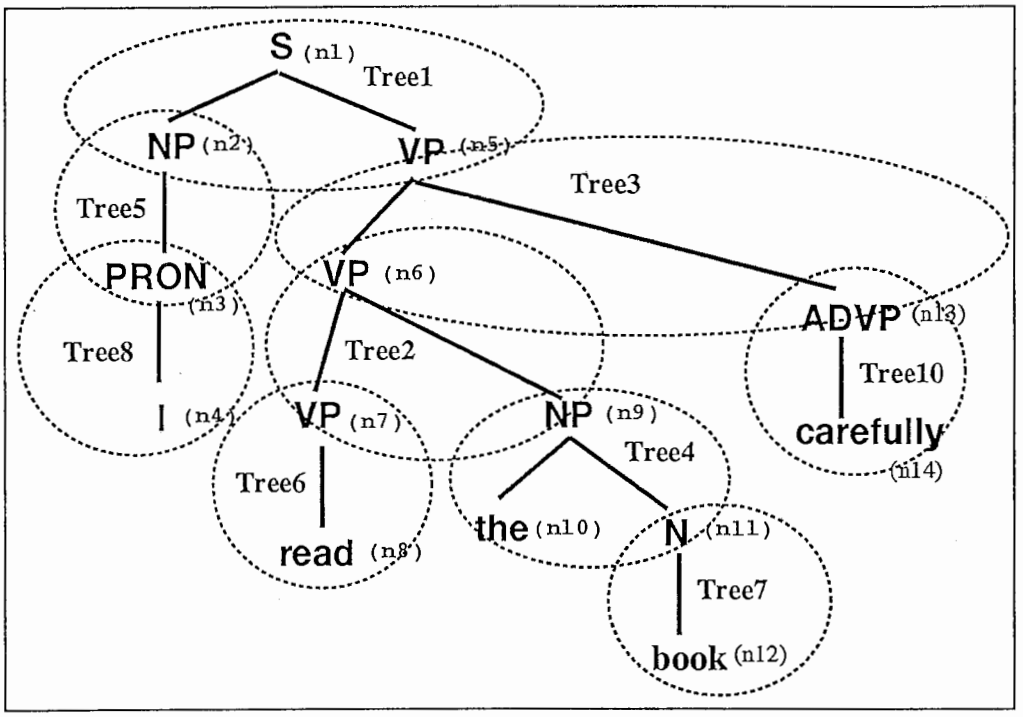

図 2.6: 最終的な構造

2.2.2 代入のみを用いる生成処理

本システムの生成処理は前節で述べた二つの条件を満たすような木構造の組み合わせを効率的に求めるものである。この処理は意味主辞駆動生成とよばれるアルゴリズムに基づいている（処理効率を改善するためオリジナルのアルゴリズムを若干変更してある [5]）。

この処理を考える上での基本的な概念が鎖構造（chained structure）である。鎖構造とは、ある要素木を起点としてその根節点に別の要素木を接続する操作を繰り返してできる木構造であり、起点となる要素木の根節点から木構造全体の根節点に至る経路上の全ての節点が同一の意味構造を共有するような構造である。

例えば、図 2.6において、n1,n5,n6,n7 は同一の意味構造を持つ[4]ので、Tree1,Tree3,Tree2, Tree6 を図 2.7のように組み合わせた構造は鎖構造である。同様に、Tree5,Tree8 の組み合わせ (図 2.4)、Tree4,Tree7 の組み合わせもそれぞれ鎖構造である。

処理系は、まず与えられた意味素性と、あらかじめ規定されている統語制約とを合わせた素性構造を持つ節点を作成し、この節点 (展開節点と呼ぼう) を最上位節点とする鎖構造を作成する。作成された鎖構造に語彙でない葉節点が含まれていればこれらの節点を新たな展開節点としてこれを最上位節点とする鎖構造を再帰的に作成する[5]。なお、鎖構造の作成は単一化を用いて木の組み合わせを求める探索であるため負荷が大きいので、

[4] 図 2.3より、Tree1, Tree3, Tree2 それぞれの根節点と VP 節点は同一の意味素性であるからこれらの木を単一化した結果である n1,n5,n6,n7 の意味構造はすべて同一になる。

[5] この方法はオリジナルの意味主辞駆動生成と異なる。この点に対する議論は 2.5を参照のこと。

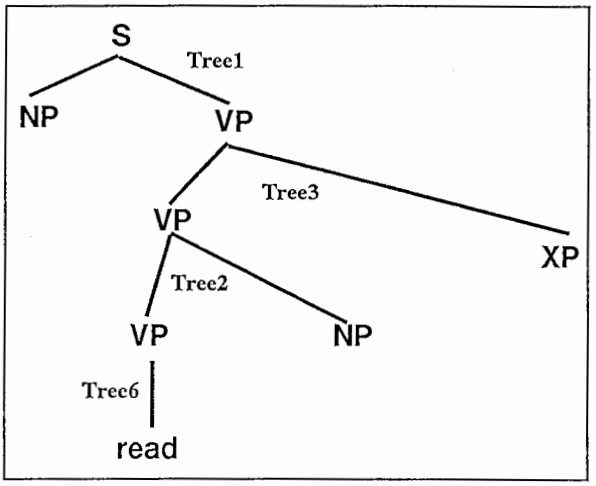

図 2.7: 鎖構造の例

処理系は展開節点の持つ素性構造に基づいてあらかじめ連鎖に関係する木を選択する処理を行ない探索空間の削減を図る。

すなわち、生成処理は次の 3 つの処理ステップを再帰的に繰り返すことにより実行される。

1. 規則の絞り込み (予備選択)
2. 鎖構造の作成
3. 語彙化されていない葉節点に対する処理の再帰的な適用

以下では、これらの処理ステップについて説明する。

1. 規則の絞り込み

   (a) 意味的な観点からの絞り込み

   展開節点の意味素性構造を包摂する (subsume)[6] ような意味素性構造を根節点に持つ木構造を選択する。

   図 2.5の意味構造に対して選択される木は (Tree1,Tree2,Tree3,Tree4,Tree5,Tree6 の 6 つである)。

   (b) 統語的な観点からの絞り込み

   link 述語 [6] を用いた統語的な制約によってさらに木を絞り込む。詳しくは文献 [6] を参照のこと。いまの例では (Tree1,Tree2,Tree3,Tree6) という木が残る。

2. 鎖構造の作成

   準備として、木構造を鎖木 (chain tree) と非鎖木 (non-chain tree の二種類に分類する。鎖木とは親節点の意味構造と同一の意味構造を持つ葉節点 —意味主辞節点 (semantic head node) と呼ぶ — の存在する木構造である。鎖木以外の木は全て非鎖木である。図 2.3では Tree1,Tree2, Tree3, Tree4, Tree5 が鎖木で残りの木が非鎖規則である。意味主辞節点は Tree1,Tree2,Tree3 が VP、Tree4 が N、Tree5 が PRON である。

   鎖構造は非鎖木を最下位構造とし、この構造の根節点から上方に 0 個以上の鎖木を接続した構造で、かつ最上位節点が展開節点と単一化可能であるようなものである。接続は、(鎖木の) 意味主辞節点と下位構造の根節点とを単一化することで行なう。

[6] 直観的には素性構造 A が素性構造 B より「小さい」とき A が B を包摂する [1]。

この例では Tree5 を最下位として Tree2,Tree3,Tree1 という順序で鎖構造が作成される (図 2.7)。

なお、ある展開節点に対して複数の鎖構造が作成可能である場合、(中間) 多義が発生する。この場合、以降の処理は多義すべてについて並行的に行なわれる。

3. 生成処理の再帰的な適用

図 2.7の二つの NP 節点、および、XP 節点は語彙項目ではない葉節点であるから、これらを新たな展開節点として再帰的に生成処理を実行する。

2.3 代入と付加を用いる生成処理

2.3.1 代入と付加を用いる生成知識

ここでは説明のために先の例と等価な生成知識を木結合文法を用いて記述する (図 2.8)。

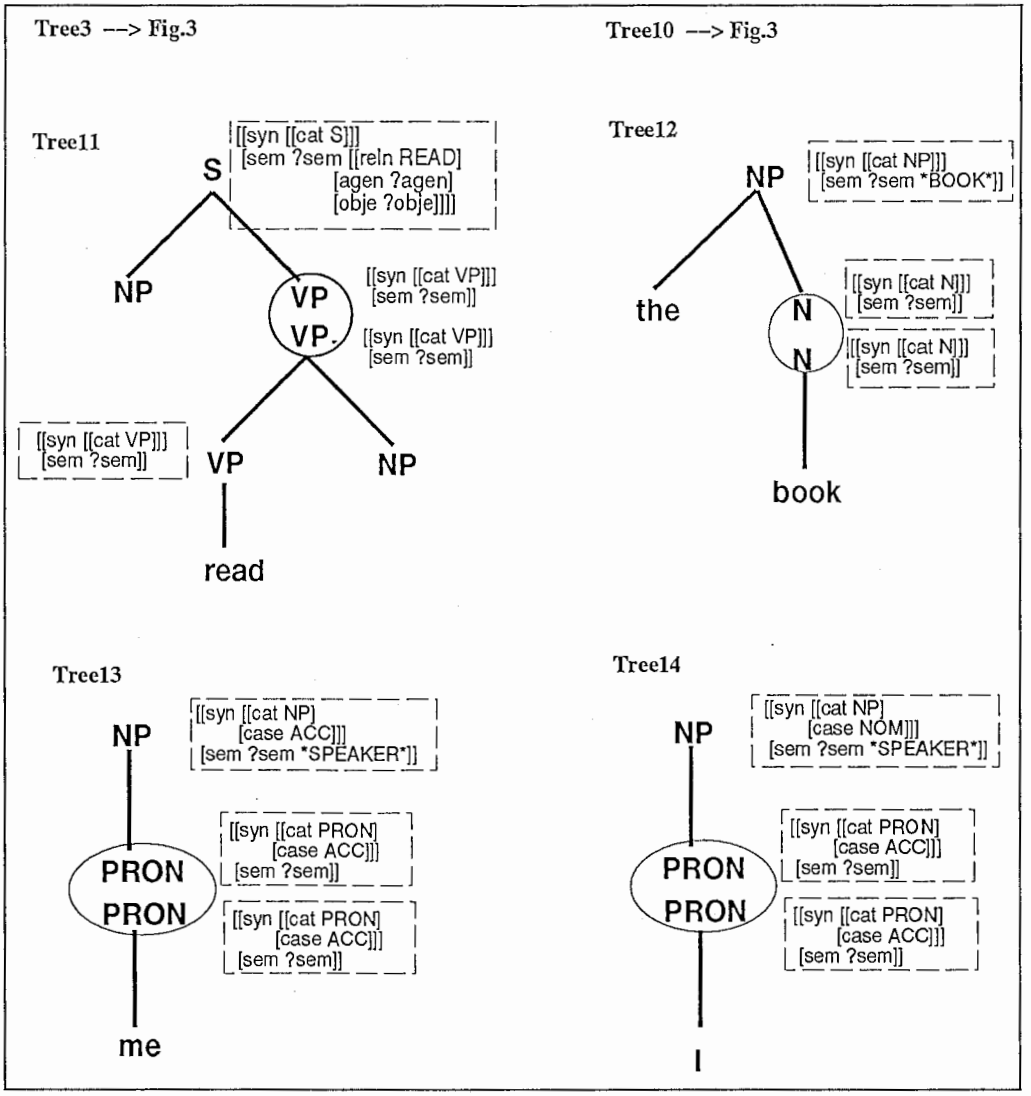

図 2.8: 付加を含む生成知識

ここで Tree3,Tree10 は図 2.3のものと同じである。先の例と異なるのは他の木の結合を許す節点である付加

節点を持つ木があることである。図中丸で囲まれた節点が付加節点である。付加節点は二つの品詞記号 (節点[7]) を含んでいる (Tree11 の VP 付加節点) が、この二つの節点でそれぞれ上側素性、下側素性を表す。

本システムにおいて付加節点は次のような性質を持つ必要がある。

付加節点の上側素性と下側素性で意味素性構造が同一である (統語素性は異なっていても良い)。

また、本システムにおいては付加木は鎖木に限定されており (modification adjunction)、footnode は意味主辞節点と等しい。このことを利用して付加木か初期木かは自動的に決定されるので、文法記述者が与える必要はない[8]。

図 2.8の規則を用いると、代入のみの例と同様に、図 2.5の入力に対して図 2.6のような木が作成される。

2.3.2 代入と付加を用いる生成処理

生成知識が付加を含む場合の生成処理は次のようになる。

1. 木の選択
    1. 意味的な観点からの絞り込み
    2. 木の分割 (1)
    3. 統語的な観点からの絞り込み
2. 鎖構造の作成
3. 木の分割 (2)
4. 語彙化されていない葉節点に対する処理の再帰的な適用

「木の分割」という処理が 2 箇所入っている以外は基本的に代入のみの処理と同様である。なお、作成中の木構造の各節点には「必須木スロット」と呼ばれるスロットを設ける。このスロットの初期値は空であり、「木の分割 (2)」においてセットされる。

1. 木の絞り込み

    (a) 意味的な観点からの選択

    代入のみの生成と同様。但し展開しようとする節点の必須木スロットに木構造が存在した場合は、これを必ず選択する (必須木)。

    図 2.5の意味構造に対しては Tree3,Tree4, Tree10 を選択する (必須木スロットは空である)。

    (b) 木の分割 (1)

    付加節点のうちその節点の属する木の根節点と意味構造を共有するものがあれば、その節点で木を分割する。

    Tree10 の付加節点 (VP) がこの節点に該当する。これを図 2.9のように上下二つの木に分割する。

    (c) 統語的な観点からの絞り込み

    代入のみの生成と同様の処理を行なう。但し、分割された木の一部が排除された場合、残りの部分木もすべて排除する。なお必須木は必ず含まれていなければならない。

    いまの例では選択された規則に変化はない。

---

[7] quasi-node とも呼ばれる [7]

[8] 初期木と付加木の区別が統語的な観点から本質的に必要かどうかは現在の重要な研究テーマである [7]。この区別は統語的なものではなく、意味的な観点からなされるべきだという議論もある。

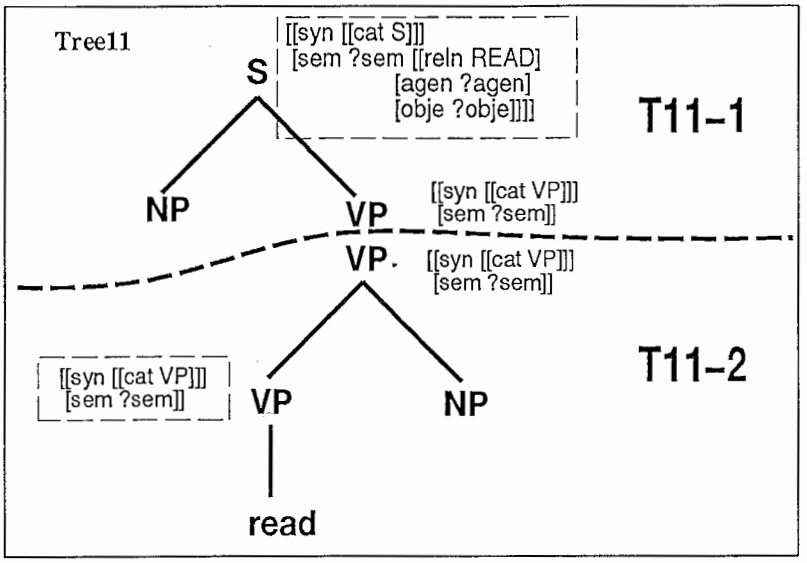

図 2.9: 要素木の分割

2. 鎖構造の作成

   基本的に前節でのべたのと同様の処理を行なう。分割 (1) によって分割された部分木は別々の木として扱う。但し、分割前の部分木の上下 (支配,domination) 関係は保持する。今の例の場合、部分木 (T11-1) は必ず部分木 (T11-2) より上方に位置しなければならない。生成される鎖構造は先の例と同じく図 2.7のようになる。

3. 木の分割 (2)

   鎖構造に属する木の付加節点の中で鎖構造の根節点と意味構造を共有しないものについてその節点で木を分割し、鎖構造から切り離された部分をその節点の必須木スロットに保存する。

   図 2.8ではこのような木は存在しない[9]。

4. 語彙化されていない葉節点に対する処理の再帰的な適用

   代入のみによる生成と同様である。

以上の処理の結果やはり図 2.6のような木が生成される。

2.4 長距離依存について

疑問詞疑問文、関係節などにおいて、統語的に深く関係する二つ (以上) の要素を統語上離れた位置に生成しなければならない場合がある。

例えば、

the book which I read

という名詞句において、book は動詞 read の目的語に対応しているが本来の目的語の位置から離れた所に現れている。もちろん、動詞 read に対して、通常の位置 (動詞の後ろ) に目的語を生成する知識と関係節用に主語と関係代名詞の前に目的語を生成する知識を別々に定義することも可能であるが、このように明らかに規則性のある現象を個別に扱うのは無駄である。

[9] A にこの例がある

さらに

the book which I would like to read

のようにこれらの間はさらに離れる場合があることを考えると、このような個別的な扱いは破綻する。

上記のような現象は長距離依存 (long distance dependency)、あるいは、非有界依存 (unbounded dependency) と呼ばれており、(英語) 文法理論を考える上での一つの試金石である。この現象に対する考え方は、単一化に基づく句構造文法 (「代入のみによる生成」に相当する) と木結合文法とで異なる。

2.4.1 単一化に基づく句構造文法における長距離依存の扱い

単一化に基づく句構造文法は句構造規則の集合からなっている。各句構造規則が直接表現できるのは木構造に対する局所的な — 親節点と子節点の間の — 制約である。一方、扱おうとする言語現象は親子関係を越えた位置にある 2 つの節点が関係する現象である。

句構造文法による長距離依存の扱いには色々なバージョンがあるが、いずれも基本的な木構造の組にわずかの変更を加えるだけで関係節や疑問詞疑問文などを宣言的な枠組で扱うことができるように工夫されている。ここでは、関係節を例にとって「スラッシュ素性による扱い」を紹介する。なお、以下では説明のためいくつかの簡略化を行なっている。

1. 木構造の各節点にスラッシュ素性と呼ばれる素性を用意する (統語素性、意味素性とは独立の素性とする)。スラッシュ素性の取り得る値は任意の統語・意味素性構造 (今まで「節点の素性構造」と称していたもの) である。

2. スラッシュの導入

   関係節修飾を表す木 (図 2.10) において、関係節に対応する葉節点のスラッシュ素性の統語部分を図のように名詞とする。スラッシュ素性の意味素性部分は被修飾名詞の意味素性と同一である。

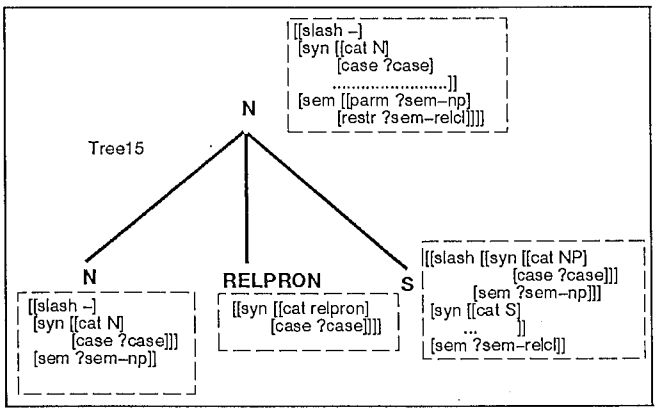

図 2.10: スラッシュの導入

3. スラッシュの終端

   ある節点のスラッシュ素性と構文カテゴリ素性とが単一化可能であれば、その節点を空範疇[10]とするような規則あるいは処理を設ける (図 2.11)。

[10] その節点の支配する文字列が空列であるようなカテゴリ

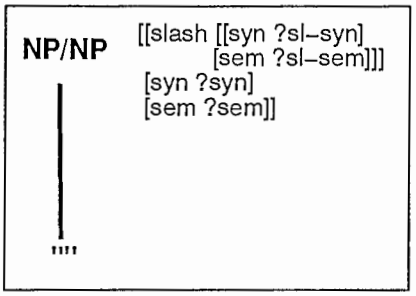

図 2.11: スラッシュの終端

4. スラッシュの伝搬

上記以外の各要素木において、根節点と葉節点のスラッシュ素性は原則として同一であるとする (同一のラベルを与える)。但し、例えば親節点が名詞句であるような木ではスラッシュ素性を空にするなど、幾つかの言語に依存した扱いがある。このスラッシュの同一化によって、要素木の根節点の情報と葉節点の情報が相互に関連づけられることになる。このような要素木を複数接続してできる木構造においてもこの関係で結ばれた根節点と葉節点のスラッシュ素性が同一となることに注意せよ。

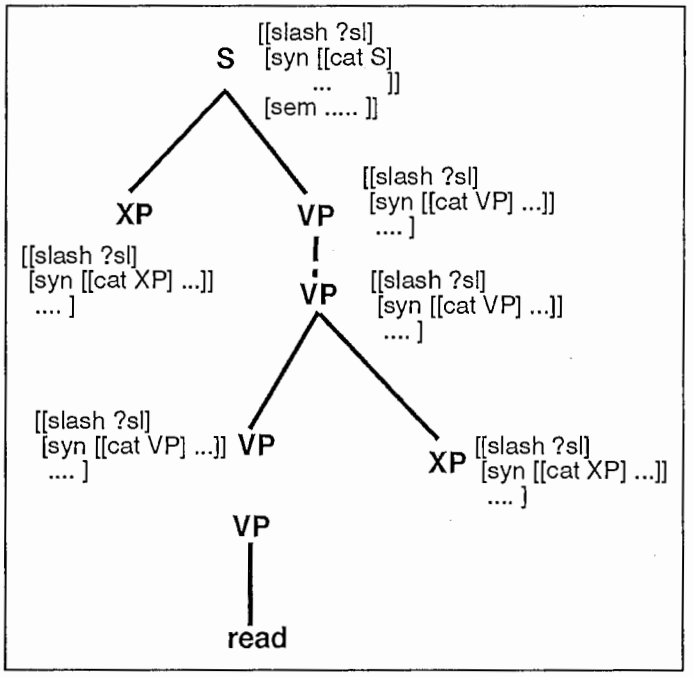

図 2.12: スラッシュ素性の伝搬

長距離依存の例

以上の仕掛けでどのように関係節の構造が規定されるか考えてみよう。ここで、先行詞の意味構造が関係節の表す動作 (この例では read) の対象と同一であるとしよう。名詞句全体の意味構造は図 2.13のようになる。この素性構造において、parm 素性が被修飾名詞 (主名詞) の意味構造を表し、restr 素性が修飾節 (句) の構造を表す。被修飾名詞が「読む動作の対象」であることは修飾節の obje 素性と parm 素性とが同一の素性構造を共有していることで表されている。

```
[[parm ?parm *book*]
 [restr [[reln READ]
         [agen *speaker*]
         [obje ?parm]]]]
```

図 2.13: 意味素性構造 2 (fs2)

この素性構造に対しては Tree15(図 2.10) が適用される。適用後の木構造の一番右の葉節点 (品詞 S の節点) の素性構造は図 2.14のようになる。

```
[[slash [syn [[cat NP]...]]
        [sem ?sem-np *book*]]
 [syn [[cat S] ... ]]
 [sem [[reln READ]
       [agen *speaker*]
       [obje ?sem-np]]]]
```

図 2.14: 適用後の Tree15 の S 節点 (The feature structure on the S node of Tree15 after application of tree15 to fs2)

この節点に対しては図 2.15のような鎖構造が生成される (2.2 節と同様の手続きによる)。ここで、目的語の部分 (n1) に着目しよう。この節点において、スラッシュ素性と統語・意味素性とは単一化可能であるから、図 2.11が適用されて空範疇となる。主語の部分はこのような条件が成立しないので、通常の生成処理が適用され、結局 "book which I read" という文字列に対応する構造が作成される。

2.4.2 木結合文法における長距離依存の扱い

木結合文法は長距離依存を扱うための特別な機構は必要としない。というのは、この文法では依存関係のある節点を同一の要素木に収まるように記述するからである。例えば、先の関係節の例では図 2.16のような要素木を用いる。この要素木には VP の位置、S の位置に様々な構造を付加することができる。すなわち、長距離依存を捉えている。

この手法の欠点は動詞毎、関係代名詞化される要素ごとに要素木を定義しなければならないことである。この問題に対処するには、素性構造を導入して規則を抽象化したり、要素木の定義を階層化したりすることが考えられ、現在さかんに研究が行なわれている。

2.5 補足 (意味主辞駆動生成との相違点について)

ここで述べた生成手法は、与えられた節点を根節点とする鎖構造をこの鎖構造に支配される構造が全て出来上がる前に作成するという所に特徴がある。このことによって、根節点の素性構造情報は鎖構造のすべての部分に反映され、鎖構造の部分構造を再帰的に生成する際の探索空間を削減する。また「再入可能な素性構造を用いた生成処理における部分結果の再利用」を実現する上でも有効であった[11]。但し、この手法が適用できる文法 (生成知識) は、意味主辞の持つ統語制約と根節点の統語制約によって必要な鎖構造が生成できるようなものでなければならない。ASURA 用の英語、および、ドイツ語の文法において、これらの制約は特に問題にならなかったが、今後、適用範囲を広げる場合には問題が生じるかも知れない。

[11] 詳細は本資料では述べない

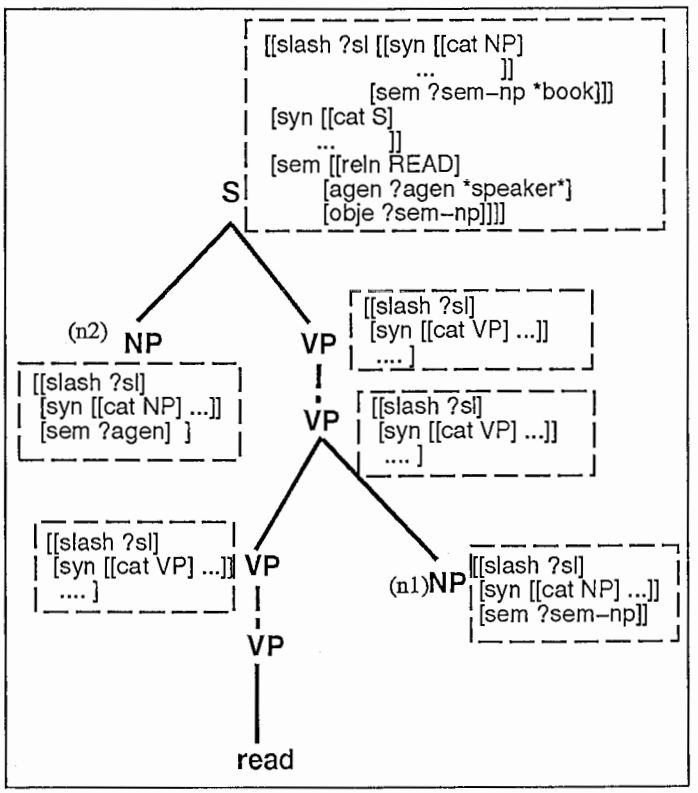

図 2.15: スラッシュを含んだ鎖構造 (A chained structure with a slash feature)

一方、Shieber らによるオリジナルの手法は鎖構造を完全にボトムアップに作成する。このため、一般に利用可能な文法の範囲は本手法より広がるが、ボトムアップに作成された部分木構造の妥当性は全ての構造が完成した後確定するので処理効率が悪くなる。様々な枝刈りや部分結果の再利用などの手法を採り入れることが考えられる [8]。

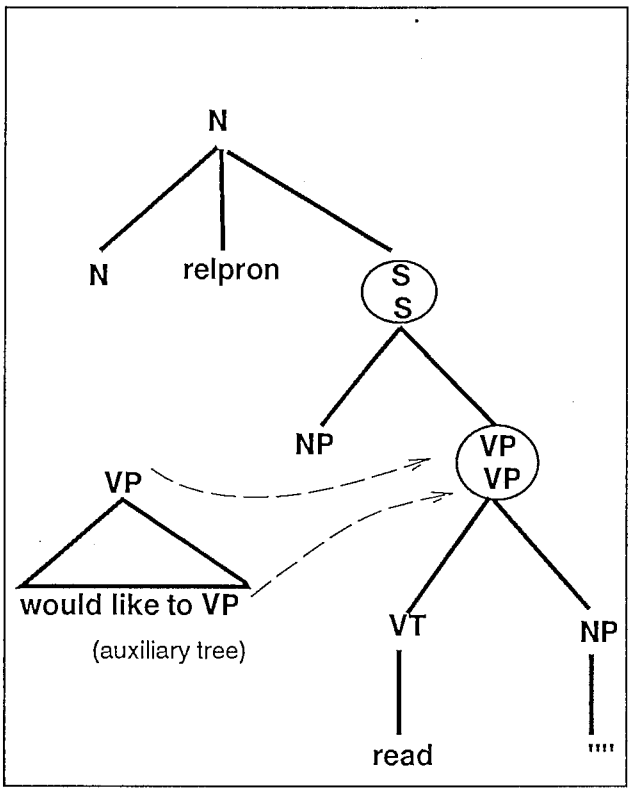

図 2.16: 木結合文法における関係節用要素木

第 3 章

生成知識記述マニュアル (Describing Generation Knowledge)

3.1 生成知識の構成 (Contents of generation knowledge)

概要編において生成知識は要素木の集合であることを述べた。実際の処理系ではこれにいくつかの補助的な情報を加えたものを生成知識と呼んでいる。すなわち、生成知識は次のものから成る。

1. 各種パスの宣言 (feature paths)
2. スラッシュ素性に関する記述 (the slash feature)
3. 初期節点に対する制約 (constraints on the root node)
4. PD(要素木) の定義 (definition of PD's)

なお、1) と 2) は 4) に先だって処理系に与える必要がある。

以下では、まず、本システムで用いる素性構造について説明する。その後、各種パスの宣言について述べ、初期節点に対する制約、PD の定義、伝搬素性に関する宣言の順に説明する。

3.2 素性構造 (Feature Structure)

3.2.1 素性構造のシンタックス (Syntax)

素性構造は、素性名と素性値の組の集合である。素性値は、**素性構造** (空素性 '[]' も含む)、**アトム**、**アトムの集合** (選言:SET)、**アトムの集合の否定** (NOT) のいずれかである。図 3.2.1に素性構造のシンタックスを表す。

3.2.2 素性構造の単一化 (Unification of feature structures)

2 つの素性構造の単一化は通常の素性構造の単一化の方法に従う [1]。空素性以外の素性構造とアトム型、SET 型、NOT 型の素性値との単一化はそれぞれ常に失敗する。アトム型、SET 型、NOT 型の素性値相互の単一化は表 3.1の規則に従う。この表において A,B はそれぞれ素性値を表し、' は SET 型、NOT 型に含まれる要素の集合を表す (たとえば、
A=(:SET X Y Z)のとき、A'= { X Y Z })。この規則で NOT と NOT を単一化する部分は便宜的なものである (全体集合が与えられないので完全な定義は不可能である)。

単一化の例を示す。

[素性構造のシンタックス]
注意！ * はスペースで区切られた 0 個以上の並びを言う
(X^* means a sequence of 0 or more X's)
素性構造 (*featurestructure*) ::= [素性*]
素性 (*fs_element*) ::= [素性名 素性値]
素性値 (*value*) ::= 素性構造 | アトム型素性値 | SET 型素性値 | NOT 型素性値
アトム型素性値 (*atom*) ::= シンボル (*symbol*) | 文字列 (*string*)
SET 型素性値 (*set(or disjunction)*) ::= (:SET アトム型素性値*)
NOT 型素性値 (*not(or negation)*) ::= (:NOT アトム型素性値*)

図 3.1: 素性構造のシンタックス (the syntax of a featrure structure)

表 3.1: 構造を持たない値の単一化規則 (unification rules for non-structural values, A and B)

A のタイプ (the type of A)	B のタイプ (the type of B)	単一化が成功するための条件 (Unification Condition)	単一化結果 (Result)
ATOM	ATOM	A = B	A or B
ATOM	SET	A ∈ B'	A
ATOM	NOT	$\neg A \in B'$	A (type :ATOM)
SET	SET	$A' \cap B' \neq \{\phi\}$	A' ∩ (:SET B')
SET	NOT	$A' - B' \neq \{\phi\}$	(:SET A' - B')
NOT	NOT	always true	(: $NOT A' \cup B'$)

例)
(:SET ADVP PP) と PP の単一化結果 = PP

(:SET ADVP PP) と (:SET PP NP) の単一化結果 = (:SET PP)

3.2.3 素性ラベル (The label, or tag, of a feature value)

素性記述にはラベルを用いることができる。ラベルは"?" で始まるシンボルであり、素性名の直後、すなわち、素性値の直前に記述して素性構造を指示する。なお、ラベルの直後に素性構造が記述されていない場合は、空素性 ([]) が記述されているものとみなす。次の例で 1) と 2) は等価な記述である。*1) and 2) are equevalent.* ◇

例)
1) [[cat ?cat []]]

2) [[cat ?cat]]

一般の素性構造ではラベルのスコープはそのラベルの出現する素性構造内であるが、本処理系ではこれを拡張して、ラベルの出現する PD(要素木の定義) 内とする。スコープ内で同一のラベルが複数箇所に出現した場合、ラベルの表す素性構造は、各々の箇所で規定されている素性構造を単一化したものとなる。

```
---------------------------------------
  (NP  [[SYN  ?syn [[person 3][number sing]]]])
---------------------------------------
  ?syn の内容は [[PERSON 3][number sing]]
```

下記の素性記述が同一 PD 内に出現したとき *When both of the following feature structures are defined in*
◇ *one PD,*

```
-------------------------------------
  (NP  [[SYN  ?syn [[person 3]]]])
  (VP  [[SYN  ?syn [number singular]]])
-------------------------------------
```

?syn の内容は [[person 3][number sing]]
◇ *the value of ?syn is [[person 3][number sing]]*

3.3 各種パスの宣言 (Paths)

意味情報や統語情報をどのような素性構造で表すかは、原則として生成知識の記述者の自由である。たとえば、概要編で用いた要素木では意味素性構造を素性名 sem の素性値としたが、たとえば、素性名sem_infoの素性値として表しても構わない。

◇ *The system requires feature paths for accessing certain features important for generation.*

そのかわり、処理系が必要な情報 (意味素性構造は生成処理において重要な役割を演ずる) を正しくアクセスできるようにき味素性や語彙項目を表す素性など生成処理において特別な意味を持つ素性が各節点の素性構造中のどの位置に存在するかを処理系に与えておく必要がある。

「パスの宣言」とは、このような素性の位置を素性構造の外側 (根) からこの素性に至る素性名のリストで表現するものである。

3.3.1 意味構造へのパス定義 (The sem feature)

ここで定義されたパスでアクセスできる素性構造を意味構造とみなす。この部分の素性構造と入力の素性構造の照合を行なう。

[意味構造へのパス定義]
(def_sem_feature_path パスを表すリスト) ;; Macro

例)
概要編でのべたような素性構造

```
[[syn ....]
 [sem 意味構造]]
```

を用いるとき、

```
(def_sem_feature_path (sem))
```

3.3.2 主品詞へのパス定義 (Syntactic category)

品詞は伝搬素性の自動生成、処理の効率化などのために用いる。

[主品詞へのパス定義]
(def_cat_feature_path パスを表すリスト) ;; Macro

例)
素性構造が

```
[[syn [[cat NOUN]]]
 [sem ....]]
```

のとき、

```
(def_cat_feature_path (syn cat))
```

3.3.3 見出し (語彙素性) へのパス定義 (Lexical string)

このパスでアクセスできる部分に素性値が存在すれば、語彙見出しとみなす。見出し素性は原則として文字列である。文字列でない場合は強制的に文字列に変換する。

[見出し素性へのパス定義]
(def_lex_feature_path パスを表すリスト) ;; Macro

例)
素性構造が

```
[[syn [[cat NOUN]]]
 [sem ....]]
```

のとき、

```
(def_lex_feature_path (syn lex))
```

3.4 初期節点に対する制約 (Constraints on the root node)

生成すべき構造の根節点に対する制約を素性構造で記述する。 *The root node of the structure to be generated should be constrained using a feature structure.* ◇

[初期節点制約のシンタックス]
(def_root_feature 初期節点制約)

例)
生成すべき構造の根節点の品詞を S(文) に制約する場合、

```
(def_root_feature ([syn [[cat s]]]]))
```

3.5 PD(要素木) の記述 (Definition of PD's)

3.5.1 PD 記述の概要

まず、PD を記述する形式の概要を摑むために記述の例を見てみよう。要素木 1 は概要編の要素木 Tree1 に対応する PD 記述である。 *The following is a sample PD corresponding to tree1 (Fig.2.3).* ◇

要素木 1 (G_PD
```
:name S-XP-VP
:internal_structure (S XP VP)
:annotation
((S [[syn [[cat S]]]
     [sem ?sem]])
 (XP [[syn ?subj-syn [[syn [[cat (:set NP ADVP)]]]]]
      [sem ?subj-sem]])
 (VP [[syn [[cat VP]]]
      [sem ?sem]
      [subcat [[first [[syn ?subj-syn]
                       [sem ?subj-sem]]]
               [rest :NIL]]]])))
```

一つの要素木は(G_PDで始まるリストで表す。:name に続く項目はこの要素木の名前であり、:internal_structure(構造記述) は要素木の構造 (節点の親子関係) を表す。:annotation(素性記述) は構造記述に現れる各節点に対する素性構造を表す。なお、概要編に示した Tree1 において XP 節点には何の統語制約もなかったがこの図では品詞に関する制約を付加してある。

3.5.2 構造記述 (Internal structure)

[構造記述のシンタックス]

$< tree >$ ::= $< symbol >$ | ($< mother >< tree >^*$)
$< mother >$ ::= $< symbol >$ | ;; non-adjoining node
($< symbol >< symbol >$) ;; adjoining node

統語木構造をリストの形式で記述する。リストの第一要素が親節点を表し、第二要素以降が子の構造を表す。第二要素以下にリストを記述することでネストした (親子より広い範囲の) 構造を表すことができる。第一要素親節点はシンボルか 2 要素から成るリストである。もし前者であれば非付加節点、後者であれば付加節点である。構造記述に含まれるシンボルは次に述べる素性記述との対応を取るポインタとして用いられる。従って、一つの構造記述に含まれるシンボルはすべて異なっていなければならない。

> [注意！]
> 構造記述中のシンボルはその要素木の定義内でのみ意味をもつポインタである。NP,N などの品詞のような記号を用いているのは単に規則を読み易くするためであり、言語的な意味は持たないことに注意されたい。

図 3.2に木構造とこれを表すリストを示す。図の右側の木には NP,VP がそれぞれ複数個出現しているので NP1,NP2 など数字で区別している。

3.5.3 素性記述

[素性記述のシンタックス]
$< annotation >$ ::= ($< symbol - feature - pair >^*$)
$< symbol - feature - pair >$ ::= ($< symbol >< feature\ structure >$)

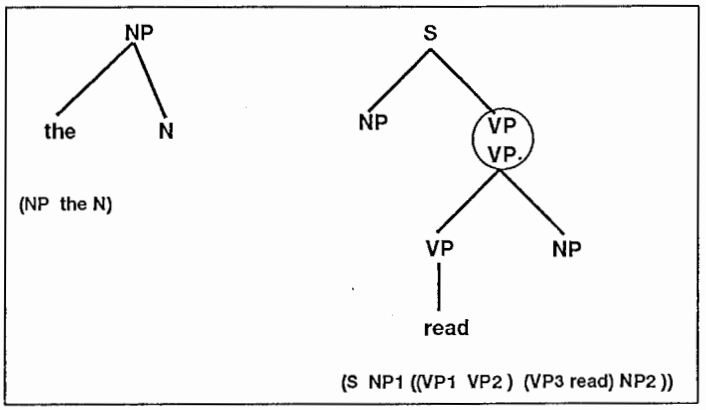

図 3.2: 木構造のリスト表現 (Internal structures and their graphical representations)

素性記述は構造記述に出現するシンボルとこのシンボルに対する素性構造の対を括弧で囲んで表現する。たとえば、シンボルSが [[syn[[cat S]]]] なる素性構造を持つ時は、

(S [[syn [[cat S]]]])

のようになる。

3.6 デフォールト意味素性 (The default sem feature)

入力素性構造中に生成処理に必要な情報が欠損している場合、その意味構造を入力とした生成処理は失敗する。特に機械翻訳においては、目的言語の生成のために必要となる情報が原言語から全て得られるという保証はない。

そこで、邪道ではあるが、統語生成上必要な意味構造が欠損している場合に、生成処理において意味構造を補う処理を用意した。

要素木の (非意味主辞) 葉節点は、defsem 素性という素性を持つことができる。生成処理系は意味構造の与えられている節点を優先的に展開していく。もし、未展開節点全ての意味構造が空である場合には、そのうち一つの節点の defsem 素性の値をその節点の sem 素性と単一化する。このことにより、defsem 素性の情報が sem 素性に供給される。

A non-semantic-head node of an elementary tree can have a 'defsem' feature. The value of the defsem feature on a node is unified with (i.e. provided to) the sem feature if the input semantic structure can not provide any semantic structure to the node. In English generation of ASURA, this is used to fill zero-pronominalized arguments with default expressions. ◇

defsem 素性の素性パスは (defsem) である。

例 ex)

```
[[syn [[cat NP][case acc]]]
 [sem ?obje]
 [defsem [[label *3-pers-sing*]]]]
```

3.7 スラッシュ素性に関する記述 (Handling slash)

第2章で述べたように、長距離依存現象は適切に要素木を記述することで、前節までの記述シンタックスの枠内で扱うことができる。ここでは、生成処理や生成知識記述をより効率的に行なうために用意された機構の

利用法について述べる。

本節では以上 2 つの伝搬素性に関する機能について述べる。

3.7.1 スラッシュ終端記述 (Slash termination rule)

スラッシュ素性終端記述はその名の通り、スラッシュの終端を制御するための記述である。ある節点を空範疇とする (下位構造を作成しない) かどうかは、2 章で述べたようにその節点のスラッシュ素性、統語・意味素性の間の単一化可能性で検査する。しかし、意味構造の設計によっては、単一化可能性だけでは弱過ぎるため、素性構造の間の同値関係を考慮した方が効率的となる場合がある。そこで、任意のスラッシュ終端条件が記述できるような枠組を用意した[1]。

[素性記述のシンタックス]

(def_slash_scheme < *condition* >< *feature − structure* >)

< *condition* > ::= (:and < *condition* >*) | (:or < *condition* >*)

< *condition* > ::= (Unifiable < *path* >< *path* >) |

(Eq < *path* >< *path* >) | < *s − expr* >

展開節点が condition 部の条件を満たす場合、この節点の素性構造に feature-structure 部の素性構造が単一化されその節点の展開を中止する。注意！ この記述は本処理系の処理アルゴリズムに依存する手続き的な性質をもつ。 *If the expanding node satisfies* condition, *then the node is unified with* feature-structure *and is not expanded further.* ◇

3.7.2 スラッシュ素性付与記述 (Slash feature generation)

要素木に対するスラッシュ素性の付与を正確かつ効率的に行なうために、スラッシュ素性を自動的に付与する処理を用意した。

[スラッシュ素性を付与すべき節点の品詞]

(def_binding_nodes < *category names*(*a list of symbols*) >)

[スラッシュ素性の名前の宣言]

(def_binding_feature_name < *slash feature name*(*symbol*) >)

要素木のある節点の品詞が def_binding_nodes[2] に規定されている品詞リストに含まれている場合、その節点に対して def_binding_feature_name[3]で規定されている素性名のスラッシュ素性を付与する。付与されるスラッシュ素性の値は空であるが、同一のラベル (変数) を与えることで要素木内のスラッシュ素性値の同一性を保証する。 *If the syntactic category on a node is one of the category names defined by def_binding_nodes*[4], *then the system generates a feature whose name is defined according to def_binding_feature_name. Note that the value of the feature is empty but all generated slash features contain the same label.* ◇

---

[1] 歴史的事情によりこのような関数名となっている

[2] 歴史的事情によりこの名前である

[3] 直前の注に同じ

[4] this rather odd name has an historical origin

3.8 要素木の階層的な定義 (PD テンプレート)(Hierarchical definition of PD's)

本処理系ではインスタンスレベルで (一定のまとまりを持った表現ごとに) 要素木を記述することを許している。このため、複数の要素木が共通の部分を持つことがある。このような共通部分は括り出して一箇所に記述することとことが生成知識の作成・管理上で有効である。また、このような括り出しを階層的に行なうことができればさらに望ましい。

本節では、この階層的な要素木の定義のための簡易的な実現であるテンプレートと呼ばれる仕組みについて説明する。

テンプレートとは良く似た要素木を多数記述する場合にその共通部分を括りだして一箇所で定義し、個々のPD記述はこの共通部分の定義を利用して行なうためのものである。本システムでは、共通部分の定義も個々のPD記述も、概要で述べたPD記述の形式に、:type、および:usingという二つの項目を設けることで行なう。 *A PD can contain :type and :using slots to enable use of the template system.* ◇

:type :typeの値は:templateか:instanceである。:instanceの場合のそのPDは要素木として生成処理に直接用いられる。: templateの場合、そのPDは他のPDを記述するために用いられるだけで、直接生成には利用されない。なお、:typeの項目を省略した場合は:instanceとみなされる。 *The value of the :type slot is either :template or :instance(default).* ◇

:using この項目には他のPDの名前一つ、あるいは複数の名前のリストを記述する。ある要素木(PD1)の:using欄にPD名が存在する場合、処理系はPD1の記述と:using欄の各PDの記述とを重ね合わせたPDを:using欄のPD毎に生成する。(生成されたPDの名前は自分の名前と重ね合わせたPDの名前とを"!"でつないだものとする。) *The value of the :using slot is a PD name or a list of PD names.* ◇

```
要素木 2 (g_pd
          :name VP-VP-ADJUNCT
          :type :template
          :internal_structure (VP1 VP2 XP)
          :annotation
         ((VP1 [[syn ?syn [[cat VP][agr ?agr]]]
                [sem ?sem]
        [subcat ?subcat]])
          (VP2 [[syn ?syn]
                [sem ?sem]
        [subcat ?subcat]])
          (XP []))))

要素木 3 (g_pd
          :name VP-ADV-MANNER
          :using VP-VP-ADJUNCT
          :internal_structure (VP1 VP2 XP)
          :annotation
         ((VP1 [[sem ?sem [[manner ?man]]]])
          (VP2 [])
          (XP  [[syn [[cat (:set ADVP PP)]]]
                [sem ?man]])))
```

これら2つの要素木の定義は下記の要素木の定義と等価である (この例だとtemplateを用いた方が記述量が多いが、実際は要素木3のようなものが多数存在するので記述量は少なくなる)。 *The set of two PD's above is equivalent to the following PD.* ◇

要素木 4 (g_pd
```
  :name VP-ADV-MANNER!VP-VP-ADJUNCT
  :internal_structure (VP1 VP2 XP)
  :annotation
 ((VP1 [[syn ?syn [[cat VP][agr ?agr]]]
        [sem ?sem [[manner ?man]]]
[subcat ?subcat]])
  (VP2 [[syn ?syn]
        [sem ?sem]
[subcat ?subcat]])
  (XP  [[syn [[cat (:set ADVP PP)]]]
        [sem ?man]])))
```

3.9 生成知識の例

生成知識の例は生成システムの rules サブディレクトリに存在する。なお、また資料として [9][10][11] がある。*Sample PD files are in the "rules" sub-directory of the generation system. Technical reports [9][10]*
◇ *[11] are available. See 4.*

また、形態素生成処理についての資料として [12], [13] がある。*About morphological generation rules, re-*
◇ *fer to [12] and [13]*

第 4 章

生成処理系利用の手引き (Using the Generation System)

4.1 動作環境 (Hardware requirements)

本処理系は以下の環境において動作が確認されている。
これ以外の組み合わせであっても CommonLisp 処理系であれば動く可能性はあるが、その場合は多少プログラムを修正する必要がある。

- Sun

  ハードウェア Sun SparcStation2

  OS SunOS 4.1.1

  言語 Sun CommonLisp 4.0.0 (4.1.0)

- HP

  ハードウェア HP700 Series

  OS HPUX 8.0.7

  言語 Lucid CommonLisp 4.0.0

4.2 処理系の導入 (Installation)

4.2.1 処理系のロードとコンパイル

配布されたメディアからプログラムを落とし、コンパイルするまでの手順を説明する。

なお、ATR においてはコンパイル済みの処理系が存在する[1]。 At ATR, the system has already been installed and compiled.

1. インストール先のマシン（以下ではローカルマシンと呼ぶ）に本処理系用にディレクトリを作成する。
   Make a directory. ◇

2. テープ（または配布されたメディア）から、本処理系をローカルマシンに落とす。
   この段階で本処理系のディレクトリ構成は図 4.1のようになっている。 *Restore files from the distribution tape. The files are organized as shown in Fig. 4.1* ◇

[1] 1994.3.10 日現在 "as26:/usr/project/asura/develop/translation/generation/release/generation" (or /DB/oldproject/asura/develop/trans

3. 処理系のトップレベルのディレクトリである generation ディレクトリに移動する。 *Change directory to the top ("generation") directory of the system.*

   これ以降はこのディレクトリで作業するものとして説明する。

4. 次に本処理系のディレクトリ名をローカルマシンの構成に合わせる。
   *Edit underlined parts of ./site.lisp. Fig.4.2.*適当なエディタを使って、./site.lisp の中のホームディレクトリ名（図 4.2のアンダーライン部分）をローカルマシンのディレクトリに合わせて編集する。

   (a) *GEN-DEFAULT-HOME-DIRECTORY* は生成処理系のルートディレクトリの位置を示す絶対パスを表す。
   **GEN-DEFAULT-HOME-DIRECTORY* specifies the absolute path to the root directory of generation system files.*

   (b) *GEN-RULE-DIRECTORY* はサンプル規則、辞書の位置を表す。必須ではない。 **GEN-RULE-DIRECTORY* specifies the absolute path to sample rules and dictionaries. This is optional.*

5. Lisp 処理系を立ち上げる。 *Start lisp.*

6. Lisp 処理系が立ち上がったら、 *Evaluate*

   (load "defsystem.lisp")

   と入力して、 defsystem のためのツールを読み込む。

7. 続いて、 *Evaluate*

   (load "site.lisp")

8. mini-rws のコンパイル *Evaluate*

   (operate-on-system 'mini-rws 'compile :force :all)

9. generation system のコンパイル *To compile the generation system, evaluate*

   (operate-on-system 'generation 'compile :force :all)

ここまでで、プログラムのインストールは終了する。
このまま作業を続ける事もできるが、次回以降の生成処理系の立ち上げの練習も兼ねて次節で改めて立ち上げる事にして、いったん Lisp を終わる事にしよう。

4.2.2 他のホストからの利用 (Using the system from foreign machines)

リモートマウントにより一つの生成処理系ファイルを複数のマシンで利用する場合、ルートディレクトリの絶対パス名がマシンによって異なることがある。 *If you would like to run the generation system on a foreign machine by remote mounting, the absolute path of the generation system on your machine may differ from the one defined in site.lisp file.* もし、 site.lisp にある絶対パス名と実際の絶対パス名が異なっている場合には、 site.lisp をローカルマシンにコピーして正しいパス名に修正することが必要である。 *In this case, you should make your own copy of the site.lisp file and modify it to reflect your own directory organization. (See the item 4 in 4.2.1)*

4.3 実行環境の設定 (Setting environment parameters)

生成処理を行なうためには処理系に生成知識 (PD や辞書) を与えなければならない。また、デバッグレベルや生成に失敗した場合の処理の方法など生成処理実行の際のパラメータを設定する必要がある。これらをまとめて「実行環境」と呼ぼう。

生成処理の実行環境の設定は、環境設定用の各種関数を呼び出す（評価する）ことによって行なわれる。必要な設定 (関数呼び出し) はまとめて一つのファイル (以降 **初期化ファイル** (`initialization file`) と呼ぶ) に記述するのが便利である。

ここでは句構造生成に関係するパス設定について説明する。なお、形態素生成については *Parameters for morphological generation are explained in TR-I-0361)* ◇

ATR 自動翻訳電話研究所テクニカルレポート「形態素生成処理解説書 (TR-I-0361)[12]」

生成処理の実行時パラメータに関しては *For other parameres, see Appendix B* ◇

付録の関数一覧 (付録 B)

を参照して欲しい。

4.3.1 PD ファイルのパス名の設定 (Pathnames of PD files)

本処理系においては次の 2 種類の PD の管理法がある。各管理法ごとに、PD 記述ファイルのパス名を設定する。 *PD's in a PD file are maintained on main memory or in a secondary storage. The user should define which PD file is loaded on memory, and which PD file is maintained on the disk.* ◇

1. オンメモリモード (on memory)

   PD の内容を実行に先だってメモリに読み込む。

2. 2 次記憶モード (in a secondary storage) （マスターおよびユーザー）

   原則として、必要な PD をその都度ディスク上のファイルから読み込む。もちろん、一度読み込まれた PD はキャッシュされるが、LRU 制御によってアクセスされないものから順に捨てられる。

   2 次記憶を用いた管理では、どの PD が必要かを判断してファイルから高速に検索するために専用のインデックスとこれに対応した形式の PD ファイルを用いる。そういうわけで、これらを作成ために、実行に先だって PD のコンパイルという操作が必要である。

   2 次記憶モードでは「マスターファイル」「ユーザーファイル」の 2 つの PD ファイルが管理できる。アクセスの優先度がマスターよりユーザーが高い他は同等である。

PD のコンパイル (Compilation of PD's) について *Compilation is necessary for a PD file maintained in the secondary storage.* ◇

2 次記憶化 PD を使用する場合は、PD のコンパイルが必要である。

この場合の コンパイル とは、PD ファイルの形式を変換して検索用の索引を付ける事を意味する。

PD のコンパイルするには、 *To compile a PD file, evaluate* ◇

(compile_pd "PD_definition_file")

と入力する。

これで、

- "PD_definition_file.data"

- "PD_definition_file.index"

◇ という 2 つのファイルが作成される。 *These files will be created.*
2 次記憶化 PD のパスを指定する際はコンパイルによって作成されるファイルではなく、ソースファイルのパス名を指定する。

4.3.2 入力ファイル (変換結果ファイル)

ファイルに記述されている素性構造を入力として生成処理を実行する場合のファイルのパス名を指定する。

4.3.3 初期化ファイルの例 (An example of the initializaiton file)

初期化ファイルの例と各行の意味を図 4.3に示す。このファイルを適当に編集して、個別の実行環境を作成する。

なお、rules/ に英語、および、ドイツ語の実行可能な規則、辞書、初期化ファイルがある。 *"rules/" contains PD files, dictionaries, and initialization files for English and German generation. See the README* ◇ *file for details.*

4.4 生成処理系の立ち上げ (Setting up the generation system)

(4.2節で導入した) 生成処理系を立ち上げる。

◇ 1. Lisp 処理系を立ち上げる。 *Start lisp.*

◇ 2. 次の S 式を評価する。 *Evaluate:*

(load "defsystem.lisp")

3. 次の S 式を評価する。もし、各自の site.lisp ファイルあればそちらをロードする。 *If you have your own* ◇ *site.lisp file, then load it; else, evaluate:*

(load "site.lisp")

このとき、 *gen-default-home-directory* の値が表示される。 *The value of *gen-default-home-directory** ◇ *is displayed.*

◇ 4. 次の S 式を評価する。 *Evaluate:*

(gen_setup :init "初期化ファイル "[2])

と入力して、生成処理系を立ち上げる[3]。

([:init "初期化ファイル "] を省略すると、既定値である"~/.genrc"を初期化ファイルとして評価する。また、:init の値を nil とすると初期化ファイルを読み込まない)

5. 初期化ファイルが適切にロードされていることを確認する。 *Confirm that the intialization file including* ◇ *PD's and dictionaries is properly loaded.*

◇ (gs) *This function displays the current settings (PD files etc.).*

[2] initialization file(genrc)

[3] バイナリファイルを読み込む際にエラーが出る場合、リスプのバージョンなどを確かめる

6. もし、適切にロードされていない場合は、初期化ファイル、規則ファイルなどを修正し、再び初期化ファイルをロードする。 *If you found errors, correct the initialization file, PD files, etc. Then, load the initialization file again.* ◇

これで生成処理の実行が可能となる。 *The generation system is now set up.* ◇

4.5 生成処理の実行 (Execution of generation)

1. 生成処理が立ち上がり初期化ファイルが適切にロードされていることを確認する。 *Confirm that the generation system is property set up.* ◇

   (gs) *This function displays the current settings (PD files etc.).* ◇

2. 辞書をオープンする。 *Open dictionaries.* ◇

   (gb)

3. 生成処理を実行する。 *Generate sentences from feature structures in the transout file.* ◇

   (gg 開始文番号 (sentence number start) 終了文番号 (sentence number end)

   or

   (gg 文番号 (sentence number))

4.6 生成処理の終了

1. 次のS式を評価する (辞書をクローズする)。 *Evaluate:* ◇

   (gq)

2. lisp から抜ける。 *Quit lisp* ◇

```
generation                          生成処理系全体
        |-- load.lisp                       生成プログラム全体の読込み
        |-- version.lisp                    生成プログラムの一覧
        |-- tools.lisp                      PD 開発用ツール類 (結果比較等)
        |-- generate.lisp                   ユーザ I/F
        |-- mgen                      形態素生成処理系
        |       |-- load.lisp               形態素生成プログラムの読込み
        |       |-- mgen.lisp               外部 I/F
        |       |-- mgnet             MG ネットエンジン
        |       |       |-- load.lisp       MG ネット実行プログラムの読込み
        |       |       |-- mgmain.lisp     形態素生成メインルーチン
        |       |       |-- mgrule.lisp     屈折変化表アクセス用 I/F
        |       |       |-- mgnet.lisp      MG ネットアクセス用 I/F
        |       |       |-- net-macro.lisp  MG ネット読込み用マクロ
        |       |-- mgdict            形態素辞書検索系
        |       |       |-- load.lisp       形態素辞書検索プログラム読込み
        |       |       |-- mgdict.lisp     形態素辞書アクセス用 I/F
        |       |       |-- emd.lisp        形態素辞書検索関数群
        |       |       |-- trie.lisp       辞書検索用トライ索引関数群
        |       |       |-- dict.lisp       辞書検索関数群
        |-- pgen                      基本構造生成処理系
        |       |-- load.lisp               基本構造プログラムの読込み
        |       |-- pd2               基本構造生成エンジン
        |       |       |-- load.lisp       生成エンジンプログラム読込み
        |       |       |-- constants.lisp  定数定義
        |       |       |-- defstruct.lisp  構造体定義
        |       |       |-- glib.lisp       ライブラリ関数群
        |       |       |-- interface.lisp  変換結果アクセス用 I/F
        |       |       |-- main.lisp       メイン処理
        |       |       |-- slash_patch.lisp スラッシュ終端
        |       |       |-- npcontrol.lisp  NP 制御
        |       |       |-- search.lisp     PD 活性化
        |       |       |-- preorder.lisp   PD 列作成
        |       |       |-- graph.lisp      PD 列探索
        |       |       |-- rule-macro.lisp PD 定義の読込み
        |       |       |-- pd-dic.lisp     PD 2 次記憶化
        |       |       |-- index.lisp      PD 多重索引
        |       |       |-- regulate.lisp   生成木正規化
        |       |       |-- tpr.lisp        木構造プリンタ
        |       |       |-- tpr_int.lisp    木構造プリンタ
        |       |       |-- tools.lisp      PD 開発用ツール類 (実行時)
        |       |       |-- debug.lisp      デバッグ用プリンタ (実行時)
        |       |-- u_parser          単一化エンジン
        |       |       |-- load.lisp       単一化プログラムの読込み
        |       |       |-- defstruct.lisp  構造体定義
        |       |       |-- mk-dag2.lisp    素性構造作成
        |       |       |-- unify_t.lisp    単一化関数群
```

図 4.1: 処理系のディレクトリ構成

```
;;; -*- Mode: Lisp; Syntax: Common-Lisp; Base: 10; Package: USER -*-
;;;*****************************************************************************
;;;
;;;    SITE-SPECIFIC SETTINGS FOR PD-BASED GENERATION
;;;
;;;*****************************************************************************
(in-package "USER")
(proclaim '(optimize (speed 3) (safety 1) (compilation-speed 0)))

(defvar *g_version* "v405")
;;;
;;;  生成処理のホームディレクトリの設定(generation engine)
;;;
(defvar *gen-default-home-directory*
      "/usr/project/asura/develop/translation/generation/")
;;;
;;;  生成規則  generation rules
;;;
(defvar *gen-rule-directory*
  "/usr/project/asura/develop/translation/generation/rules/")
;;;-----------------------------------------------------------------
(defvar *gen_dir* "generation/")
(when (boundp '*default-home-directory*)
    (setq *gen-default-home-directory*
  (concatenate  *default-home-directory* *gen_dir*)))
;;;
;;;
(format t
"~%;*GEN-DEFAULT-HOME-DIRECTORY* =>  ~a"  *gen-default-home-directory*)
;;;
```

図 4.2: ./site.lisp ファイルの編集

```
(in-package USER)

;;; デバッグレベル
(g-set-debug nil)
;;; 変換結果ファイル名 the transout (=input) file
(set-transout_path "/home1/nadine/generation/fsdata/sample.input")
;;; 変換結果ファイルの読み込み loading the transout file
(load_transout)

;;; 屈折変化表ファイル the inflection rules file
(mg-set-rule "/home1/nadine/generation/mgen/mg-rule/defrule.lisp")
;;; 屈折変化表のローディング
(mg-load-rule)
;;; MGネットファイル名
(mg-set-net "/home1/nadine/generation/mgen/mg-net/defnet.lisp")
;;; MGネットのローディング
(mg-load-net)

;;; マスター辞書ファイル名 use the master morph. dict or not
(mg-set-m-dict "/home1/nadine/generation/mgen/dict/master.dict")
;;; ユーザ辞書使用の可否 use the user morph. dict or not
(mg-use-u-dict nil)
;;; ユーザ辞書ファイル名
;;;(mg-set-u-dict "~ /user.dict")

;;; 使用する PD の組み合わせの指定 valid PD access methods
(set_pd o m)
;;; オンメモリー PD 用 PD 規則ファイルのパス名 the on-memory-PD file
(set-o_pd_path "/home1/nadine/generation/pgen/pd-rule/sample.pd")
;;; 2次記憶化 PD 用 PD 規則ファイルのパス名 PD in a secondary storage (user)
;;;(set-u_pd_path "~ /user.pd")
;;; 2次記憶化 PD 用 PD 規則ファイルのパス名 PD in a secondary storage (master)
(set-m_pd_path "/home1/nadine/generation/pgen/pd-rule/sample.pd")

;;; PD 規則ファイルのオープン open secondary strage PD files
(format t "~ %(open-pd)")
(open-pd)
;;; PD 規則ファイルのロード load the on-memory PD file
;;;(format t " %(load_pd)")
;;;(load_pd)



;;; end of file ;;;
```

図 4.3: 初期化ファイルの例

第 5 章

文法のデバッグの手引き (Debugging the generation knowledge)

5.1 はじめに (Introduction)

システムは、PD 規則 (要素木) 読み込み時にある程度のシンタックスチェックを行ないメッセージを出力する。従って、構造定義と素性定義のミスマッチなどはこれによってある程度のデバッグが可能である。

しかし、文法の開発においては、何と言っても実際に生成処理を行ないながらのデバッグが有効でありかつ避けられない。このようなデバッグに関しては次のようなツールが提供されている。

1. トレーサー (Tracer)
2. ブレークハンドラー (Break handler)
3. 鎖構造チェッカー (Tree Chain Verifier)
4. emacs+ilisp とのインターフェース (Interface with emacs)

ここではこれらの機能について説明する。

5.2 トレーサー (The tracer)

生成処理の状況を表示する。資料 (関数一覧) では「デバッグ機能」と呼ばれているもので関数 set_g_dbg によって表示させるべき情報を選択する。 *The system outputs various trace information. The sort of trace information can be controlled by the set_g_dbg function.* ◇

5.3 ブレークハンドラー (The break Handler)

生成の途中で処理を中断し、read-eval-print ループに入る。この状態で

1. 木構造を表示させたり、この木構造の各節点の素性構造を表示させたりできる。
2. また、任意の葉節点に対して指定した規則の適用を試みることができる。

なお、利用例は 5.6章を参照のこと。

5.3.1 ブレークポイントの設定と解除 (Setting break points)

ブレークポイントとは処理を中断すべき位置のことである。手続き型のプログラムでは、サブルーチンの名前や行番号によってブレークポイントを指定するが、本処理系においてプログラムに相当する「文法」は、いわゆる「宣言的な」文法であるから、このようなやり方は使えない。そこで、生成処理アルゴリズム (これは手続き型のプログラムである) 中の処理の単位である「節点の展開」に注目し、「現在展開しようとしている節点の素性構造がブレークポイントとして与えられた素性構造と単一化可能な場合にその時点 (実際は展開の前後) で処理を中断させる」という方略を取る。

```
(gen_break_before  fs)
(gen_break_after   fs)
```

(gen_break_before fs) は展開節点の素性構造が fs と等しいとき、その節点の展開を行なう直前で処理を中断する。処理の中断はリスプ処理系の提供しているブレーク関数によって実現されている。従って、中断からの復帰はリスプ処理系で定義されているコマンド (continue コマンド) によって行なう。

Each time when a new expanding node is selected, the system tries to unify *fs* with the feature structure on the expanding node. If they are unifiable, then the system suspends the generation process before expaning the node for *(gen_break_before fs)*.

(gen_break_after fs) は展開節点の素性構造が fs と等しいとき、その節点の展開を行なった直後で処理を中断する。これ以外は前者と同様である。

The system suspends the generation process after expaning the node for *(gen_break_after fs)*.

5.3.2 ブレーク中に利用できる関数 (Functions available during break)

- 木構造の表示 (Print the current tree)

  ```
  (tpr)
  ```

  木構造表示で表示させる情報は次のような関数によって制御できる。

  1. 要素木の列 (鎖) に関する情報
     節点が展開節点であればその節点から下方に伸びる要素木の列 (鎖) に関する情報を表示する。

     (set_tpr_mode &optional *mode*)

     mode:

:number	規則列を構成する PD 規則 (要素木) の数
:names (default)	規則列 (= PD 規則 (要素木) のリスト)

  2. 素性値
     節点の持つ素性構造の値を木構造の節点に表示する (詳しくは「関数一覧」を参照のこと)。

- 節点の持つ素性構造の表示 (Print the fs on a node)

  (show_node *node_number*)

  節点番号は「木構造の表示」によって与えられる番号 *Choose one of the node_numbers on the printed tree.*

- 展開節点アジェンダの表示

  (show_agenda)

- 規則列の適用可能性検査

  指定したPD列が指定した展開節点に接続可能かどうかを検査する (後述)。

  (s_checker *&key node pds*)

  node 節点番号。省略すると現在の展開節点。

  pds PDのリスト。先頭が最上位のPD

5.4 鎖構造チェッカー

注目する節点に対してある鎖構造が適用可能かどうかを検査する。適用不可能な場合はその理由を表示する。ブレーク中に利用する。利用例を5.7章に示す。

(s_checker *&key node pds*)

node 節点番号。省略すると現在の展開節点。

pds PDの連鎖を表すリスト。先頭が最上位のPD

5.5 emacs+ilisp上とのインタフェース

emacs+ilisp環境で生成処理系を作動させている場合、この環境を用いたツールが提供されている。

5.5.1 利用法

1. generation/の下にあるenv.elをemacs起動時にロードする *Load "*`generation/env.el`*" into emacs lisp when you invoke emacs. (Modify " /.emacs".* ◇

2. (n)emacs or mule + ilisp 環境で(lisp +)生成処理を起動する。 *Setup the generation system within ilisp environment* ◇

5.5.2 機能

1. 節点の持つ素性構造の表示 (Print fs on a node)

   ブレークポイントで利用できる。 *Available during a break* ◇

   (a) ブレークポイントで木構造を表示させる (tpr) *(tpr)* ◇

   (b) その木構造の節点番号 (例 #452) の所にカーソルを持っていき、 *Move the cursor to the number of the node (of the printed tree) you would like to inspect.* ◇

   (c) control-c n とタイプする。 *Hit control-c n* ◇

2. PD規則 (要素木) の表示 (Print the definition of a PD)

   (a) 規則名の所 (例 S-NP-VP) にカーソルを持っていく *Move the cursor onto the PD name.* ◇

   (b) control-p とタイプする。 *Hit control-c p* ◇

5.6 ブレーク機能の利用例

```
$>(gg 1)                              ;;; 生成処理の実行

No.1 ::U-ID=S01
 -(8)-------------------------------
I read the book
 ---------------------------------
 =========================================
 "I read the book."
 =========================================
T
$> (gen_break_before [[syn [[cat NP][case Acc]]]]) ;; ブレークポイント設定
(#S<F_NODE :ID 463 :FW - :TYPE :COMPLEX>)
$> (show_break)                              ;; ブレークポイント表示

------- break BEFORE conditions -------
<<#1>>
 [[SYN [[CAT NP]
        [CASE ACC]]]]

NIL
$> (gg 1)

No.1 ::U-ID=S01
>>>>> broke before expanding [#27 ACTIVE  (0)]  ;; 処理の中断
>>Break: GEN-BREAK

BREAK-BEFORE_EXPANSION:
   Required arg 0 (S_NODE): #<S_NODE_U FEATURES #S<F_NODE :ID 694 :FW - :TYPE :COMPLEX> LEX NIL CAT NP>
:C  0: Return from Break
:A  1: Abort to Lisp Top Level

-> (tpr)                                     ;; 木構造の表示
S [#22 EXPANDED  (4)]
|--NP [#23 ACTIVE  (0)]
|--VP [#24 EXPANDED  (0)]
   |--VP [#25 EXPANDED  (0)]
   |  |--VERB [#26 LEXIFIED "read"  (0)]
   |--NP [#27 ACTIVE  (0)]

#<S_NODE_R FEATURES #S<F_NODE :ID 648 :FW - :TYPE :COMPLEX> LEX NIL CAT S>
-> (show-node 23)                        ;; 節点番号 23 の内容の表示
 [[SYN [[CAT NP]
        [AGR []]
        [CASE NOM]]]
```

```
  [SEM [[LABEL *SPEAKER*]]]
  [SLASH []]]
#<S_NODE_U FEATURES #S<F_NODE :ID 664 :FW - :TYPE :COMPLEX> LEX NIL CAT NP>
-> 0                                    ;; 処理の続行
Return from Break
 -(8)-------------------------------
I read the book
 ---------------------------------
 =========================================
 "I read the book."
 =========================================
T
```

5.7 鎖構造チェッカーの利用例

```
$>(gg 1)          ;; 生成実行

No.1 ::U-ID=S01
>>>>> broke before expanding [#44 ACTIVE  (0)]>>Break: GEN-BREAK ;; ブレーク (節点 #44 で中断)
BREAK-BEFORE_EXPANSION:
   Required arg 0 (S_NODE): #<S_NODE_U FEATURES #S<F_NODE :ID 923 :FW - :TYPE :COMPLEX> LEX NIL CAT NP>
:C  0: Return from Break
:A  1: Abort to Lisp Top Level

-> (tpr)                ;; 木構造の印刷
S [#39 EXPANDED  (4)]
|--NP [#40 ACTIVE  (0)]
|--VP [#41 EXPANDED  (0)]
   |--VP [#42 EXPANDED  (0)]
   |  |--VERB [#43 LEXIFIED "read"  (0)]
   |--NP [#44 ACTIVE  (0)]

#<S_NODE_R FEATURES #S<F_NODE :ID 877 :FW - :TYPE :COMPLEX> LEX NIL CAT S>
-> (s_checker :pds '(np-def book))  ;; np-def book という木の列が
                                    ;;  現在の展開節点 (#44) に接続
                                    ;;  可能かどうか調べる
 EXP_NODE NP-DEF BOOK            ;; exp-node 展開節点
*--- PDs are globally unifiable !  ;; 単一化可能
Checking top level features ...
 Expanding node              :PARM RESTR   ;; 展開節点のもつ素性
 NP-DEF                      :             ;; np-def  のもつ素性 (空)
 BOOK                        :PARM RESTR   ;; book のもつ素性 (parm restr)
*---All top features are covered   ;; 展開節点のすべての素性が PD によって覆われている
Checking activation                ;; 枝刈りがうまく適用されている
 Activated PDs (after filtering)..
 NP-DEF  NP-PROPN  NP-PRON  BOOK
```

```
*---All the PDs are active with respect to the expanding node.
NIL

-> (s_checker :pds '(np-pron speaker))
EXP_NODE  ** NP-PRON SPEAKER     ;; 展開節点と NP-PRON をつなぐ時単一化に失敗した
 !!!! FAIL !!!!
*INCONSISTENT SUB STRUCTURES*  ;; 失敗の原因
<PATH> (M SYN CASE)            ;; 素性構造のどのパスで矛盾を生じたか
++ Upper Part ++
 ACC                           ;; 上側 (展開節点側) の上記パスの素性値
++ Lower Part ++
 NOM                           ;; 下側 (規則列側) の上記パスの素性値
=== Upper FS ===
 [[M [[SYN [[CAT NP]
            [CASE ACC]]]
       [SUBCAT []]
       [SEM [[PARM !X1[]]
            [RESTR [[RELN BOOK]
                    [ENTITIY !X1]]]]]]]]
=== Lower FS ===
 [[M [[SYN [[CAT NP]
            [CASE !X3 NOM]
            [AGR !X2[[PERSON 1]
                     [NUMBER SING]]]]]
       [SEM !X1[[LABEL *SPEAKER*]]]
       [SLASH []]]]
   [D1 [[M [[SYN [[CAT PRON]
                  [CASE !X3]
                  [LEX "I"]
                  [AGR !X2]]]
           [SEM !X1]]]]]]

Checking top level features ...
 Expanding node               :PARM RESTR   ;; 展開節点のもつ素性
 NP-PRON                      :
 SPEAKER                      :LABEL
 ===> (LABEL) is not in the expanding node ;; LABEL 素性が展開節点にない
 ===> (PARM RESTR) is uncovered            ;; PARM RESTR 素性がどの規則にもない
Checking activation
 Activated PDs (after filtering)..
 NP-DEF  NP-PROPN  NP-PRON  BOOK
 (SPEAKER) is not active              ;; SPEAKER という PD は選択されない
NIL
```

謝辞

処理系のコーディングにおいて多大な貢献を果たした渡辺 学氏 (東洋情報システム) に感謝する。また、多くの議論をして頂いた株式会社 ATR 自動翻訳電話研究所データ処理研究室の諸氏に感謝する。さらに、本稿の英語の改善に関して助言を頂いた Mark Seligman 氏に感謝する。

付録 A

Semantic-Head-driven Generation for Feature-Structure-based Tree-adjoining Grammars

1

Abstract

This appendix describes a natural language generation system developed for a spoken language translation system. Our system employs Feature-Structure-based Tree Adjoining Grammar (FTAG) for generation knowledge representation. Each elementary tree of our grammar is paired with a semantic feature-structure.

Feature-structures attached to nodes of elementary trees are unrestricted. Thus our formalism allows HPSG style phrase structure description as well as TAG style description. The advantage of our generation knowledge representation is the ability to incorporate both HPSG style "core" grammar and TAG-style case-based grammar.

The system generates a syntactic tree by combining elementary trees so as to satisfy an input semantic structure. The generation algorithm is an application of a semantic head-driven generation. To carry out an adjoining operation, an elementary tree with an adjunction node is dynamically split at the adjunction node during generation.

Key Words : tree adjoining grammars, unification-based grammars, semantic head-driven generation

A.1 Introduction

Natural language generation (NLG) creates well-formed sentences or utterances from their semantic representations. NLG is required for man-machine interface modules of AI-systems and for natural language systems such as machine translation and text abstraction systems.

An NLG task can be divided into two subtasks: the what-to-say subtask and the how-to-say subtask. The former determines the content of an utterance, and the latter determines syntactic realizations of the content. This paper focuses on the latter issue. More specifically, it looks at a mechanism for mapping from a semantic structure onto a syntactic structure using linguistic knowledge represented in a version of *Tree Adjoining Grammars.*

[1] A part of this paper was written at DFKI (Deutsches Forschungszentrum für Künstliche Intelligenz). The author would like to express his gratitude to Wolfgang Wahlster and members of the incremental language generation group for their constant support and helpful discussions.

There are several frameworks for representing the linguistic knowledge that defines well-formed structures in a language. Some of them (e.g. HPSG or LFG) also provide semantic information for each well-formed structure, which is indispensable for language generation from semantic information.

In addition, the linguistic knowledge should deal with both general 'grammatical' constructions and idiomatic expressions [14] [15]. This is because idioms and fixed collocations of words play an important role in human conversations [14].

Tree adjoining grammar (TAG) is an attractive framework for representing idiomatic linguistic phenomena because of its *extended domain of locality*, as exemplified by [16]. TAG annotated with feature-structures, called Feature-Structure-based TAGs (FTAGs) [2], enables us to attach semantic information to each elementary tree which is necessary for generating a sentence from its semantic representation. TAG can also handle non-idiomatic linguistic phenomena with theoretically elegant factorization of recursion and local dependency. However, the representation of non-idiomatic expressions tends to contain many replications of phrase structures [17], and causes inefficiency for practical processing.

This inefficiency, however, is improved if we use Head-driven Phrase Structure Grammar (HPSG) [4] for non-idiomatic constructions. HPSG consists of a small set of highly generalized rules and rich lexical entries with specific information. Therefore, generation knowledge representations are compact and, moreover, efficiently improved simply by adding new lexical entries.

The generation system described here uses generation knowledge that combines TAG and HPSG in an FTAG framework. It consists of a set of trees. Each tree contains feature-structures for representing its syntactic and semantic properties. With our formalism, general grammatical constructions are described with small trees specifying just immediate dominance relations, whereas idiomatic constructions are described with larger trees.

The generation system combines elementary trees of the grammar so as to satisfy the input semantic structure. The combination algorithm is a version of semantic head-driven generation [18] extended to handle *adjoining* as well as *substitution* so as to handle FTAGs.

Section A.2 describes our generation knowledge representation. Then, Section A.3 introduces an algorithm using such linguistic knowledge.

A.2 Generation Knowledge Representation

Generation knowledge for a language (e.g. English) is represented as a set of trees, called *elementary trees*. Each elementary tree represents a part or parts of a well-formed structure of the language together with semantic information. The syntactic and semantic information of an elementary tree is stored in feature-structures that are associated with the tree basically in the same way as in the FTAG (Feature-Structure-based Tree Adjoining Grammar) framework.

In this section, we briefly introduce the FTAG framework, then introduce our representation syntax, and finally present a sample grammar which contains both HPSG and TAG.

A.2.1 Feature-Structure-based Tree Adjoining Grammar (FTAG)

FTAG is a unification-based extension of Tree Adjoining Grammars. FTAG consists of a set of *elementary trees* and a feature structure called the *root feature*. A well-formed structure is created by *combining* one or more elementary trees so as to satisfy certain *well-formedness conditions*. The structure of an

elementary tree, possible combination operations, and well-formedness conditions are defined as follows.

- Elementary trees

  Each node of an elementary tree has a feature structure called the *top feature* and another called the *bottom feature*. The recent version of FTAG assigns no label on each node. This means the property of a node is totally defined by the feature-structures on the node. In addition, this version regards a node consisting of two *quasi nodes* linked by a *domination link*. A quasi node contain either the top-feature or the bottom-feature. Elementary trees are classified into two categories: *initial trees* and *auxiliary trees*. An auxiliary tree has a special leaf node called the *foot node*.

- Combination operations

  Two operations are used to combine elementary trees. In the following definitions, "tree" means either an initial tree or a tree created by the following operations. Thus operations are iteratively applicable.

  1. Adjunction

     Adjunction adjoins an auxiliary tree to a tree.

     Suppose an auxiliary tree t_1 adjoins a tree t_2 at a node n_1. The top feature of n_1 and the top feature of the root of t_1 are unified, and the bottom feature of n_1 and the top feature of the foot of t_2 are unified. The resulting tree is shown in Figure A.1. In the figure, the '+' sign is a unification operator.

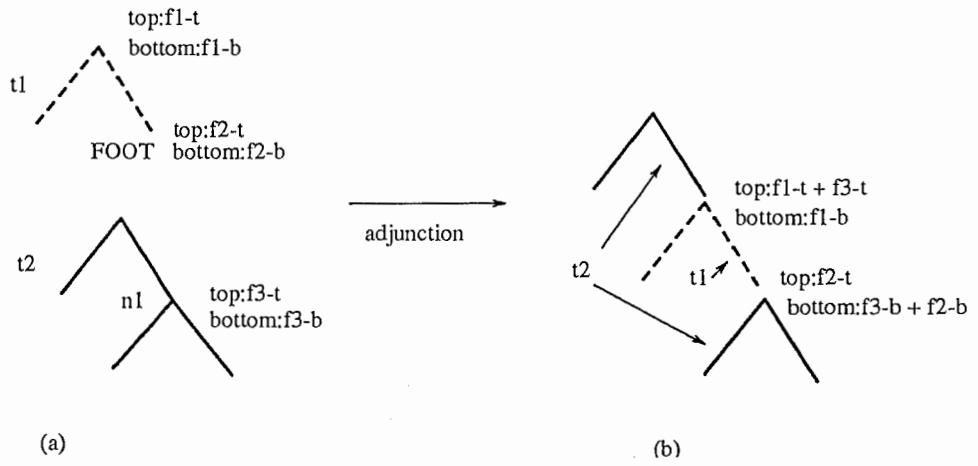

図 A.1: Adjunction of trees

  2. Substitution

     Substitution substitutes an initial tree for a leaf node of a tree. This operation requires the concept of quasi nodes.

     Suppose an initial tree t_1 is substituted for a leaf node n_1 of a tree t_2. The bottom feature of n_1 (the lower quasi node for n_1), is unified with the top feature of the root of t_1 (the upper quasi node for the root of t_1). Figure A.2 shows this operation. Quasi nodes are written as small circles.

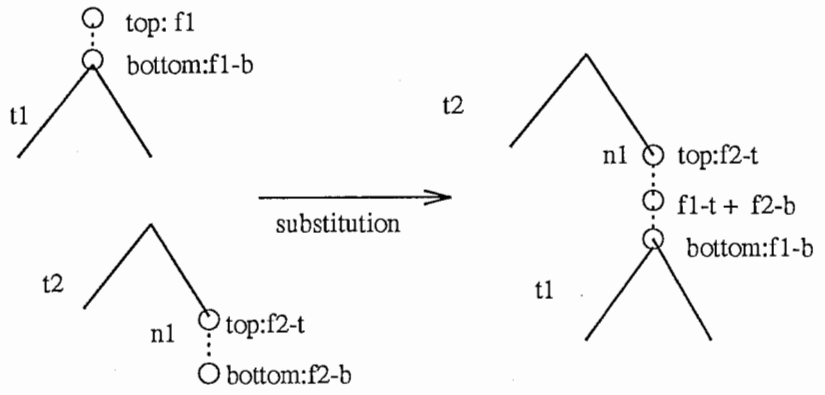

図 A.2: Substitution of trees

- well-formedness conditions

  A tree is well-formed, if and only if it satisfies the following conditions.

  1. The top-feature of the root node of the created tree is unifiable with the given root feature.
  2. All unification operations in combining trees are successfully carried out.
  3. For each node that is not adjoined with an auxiliary trees, the top feature and the bottom feature on the node are successfully unified.
  4. Every leaf node is marked as a terminal category and has a lexical feature (*an empty string* is regarded as a lexical item).

A.2.2 Our Framework

To create well-formed structures from semantic representations, each elementary tree contains semantic information. The semantic information has to be coded within the FTAG framework.

This subsection first introduces notational extensions peculiar to our framework, and then, describes constraints on semantic descriptions.

Representation Syntax

- Elementary trees.

  The structure of an elementary tree is basically the same as that of an elementary tree in FTAGs.

  To reduce redundancy of description and to improve processing efficiency, we classified nodes into two categories: *adjunction nodes* and *non-adjunction nodes*. An adjunction node can be adjoined with an auxiliary tree, but a non-adjunction node can not be. An adjunction node has top and bottom feature-structures just like a node in FTAG. A non-adjunction node, on the contrary, has only one feature-structure. On a non-adjunction node, both *the top feature* and *the bottom feature* are used to refer to the feature-structure.

  An elementary tree may be a single node when it is used as a lexical entry.

- Combination operations

  We use both *substitution* and *adjunction* to combine trees. But the adjunction operation is extended as follows:

1. Multiple adjunction

   Multiple adjunction adjoins *a sequence of auxiliary trees* to one adjoining node.

   Suppose a sequence of trees, $t1, ..., tn$ are adjoined to an adjoining node $n1$ of t. The top feature on $n1$ is unified with the top feature of the root of t1. The bottom feature on the foot node of Ti and the top feature on the root node of Ti+1 are unified, where $2 < i < n-1$. The bottom feature on the foot node of Tn is unified with the bottom feature on $n1$ (Figure A.3).

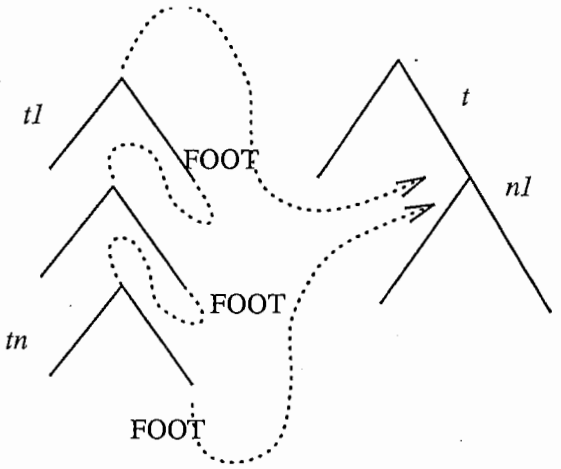

図 A.3: An example of multiple adjoining

   Multiple adjunction can be simulated by auxiliary trees whose roots, or feet, are adjunction nodes.

- well-formedness conditions

  The well-formedness conditions are identical to those of FTAG explained above.

Semantic Features

Each elementary tree has semantic feature-structures, called *SEM features*, which represent semantic information associated with the tree. The following restrictions on semantic features are introduced to properly generate sentences:

1. The (top) feature on each root node should contain a SEM feature. The value of the semantic feature can be null as long as it is properly co-referred to by other features. This SEM feature on the root of an elementary tree is called *the semantic feature of the elementary tree.*
2. The top and the bottom features on an adjunction node should have SEM features and these SEM features should co-refer to the same feature-structure.
3. Every non-terminal leaf node of an elementary tree should have a SEM feature in its bottom feature.
4. The SEM feature on the root node and the SEM feature on the foot node are identical.

A.2.3 Tree-based Generation Knowledge

Using the above framework, we constructed generation knowledge consisting of general grammar trees and specific expression patterns.

General Grammar Trees

The general grammar trees are responsible for expressions where semantic compositionality holds.

The grammar trees are based on HPSG. Every rule of HPSG is transformed into a tree. Each lexical entry is transformed into a node rather than a tree. Trees for major linguistic phenomena are described as follows.

- Head-complement Structure (projection of lexical information)

  The lexical entry of a certain category (e.g. verb) has a *subcat frame* that constrains possible complements. Figure A.4 shows the feature structure of the lexcal entry for "hit". It is represented as a special tree with one node containing this feature structure (cf. Section 2.2.1). T1 is the identifier of this entry . The SEM feature on the entry means that the entry is used to express the HIT relation, which takes an agent and an object as its arguments. "?agent" in the SEM feature appears in the second element of the subcat frame. This means the agent of the HITting should be realized as the subject of the verb "hit". T2, T3 and T4 are again identifiers of the trees.

```
[[syn [[cat V][root ''hit'']]]
 [sem [[reln hit]
       [agen ?agen]
       [obje ?obje]]]
 [subcat  [[first [[syn [[cat NP][case ACC]]]
                   [sem ?obje]]]
          [rest [[first [[syn [[cat NP][case NOM]]]
                         [sem ?agen]]]
                [rest :NIL]]]]]]
```

図 A.4: A one-node tree (T1) for lexical entry

  Elements in the subcat frame of a lexicon are realized as appropriate phrases by trees shown in Figure A.5[2].

- Adjunct-head Structure

  An elementary tree for an adjunct is shown in Figure A.6. T5 is for a spatial location, and T6 is for degree of certain state. Note that the syntactic head (VP) of this tree is also a semantic head. This is different from the popular DCG notation where the semantic head is not the syntactic head but the adjunct. This issue will be discussed in Section 4.

- Long Distance Dependency

  Long distance dependency phenomena, such as wh-movement and NP-movement, are handled by *slash features* in the same way as in HPSG.

Specific Patterns

Specific patterns deal with phrase structures which are memorized and accessed as a whole. Some specific patterns are frequently-used expressions considered to be in long-term memory, like idioms or canned phrases. Others are phrases recently used, which are considered to be in short-term memory.

[2]These three trees can be reduced to one tree if we use general VP with additional features instead of S, VP and V.

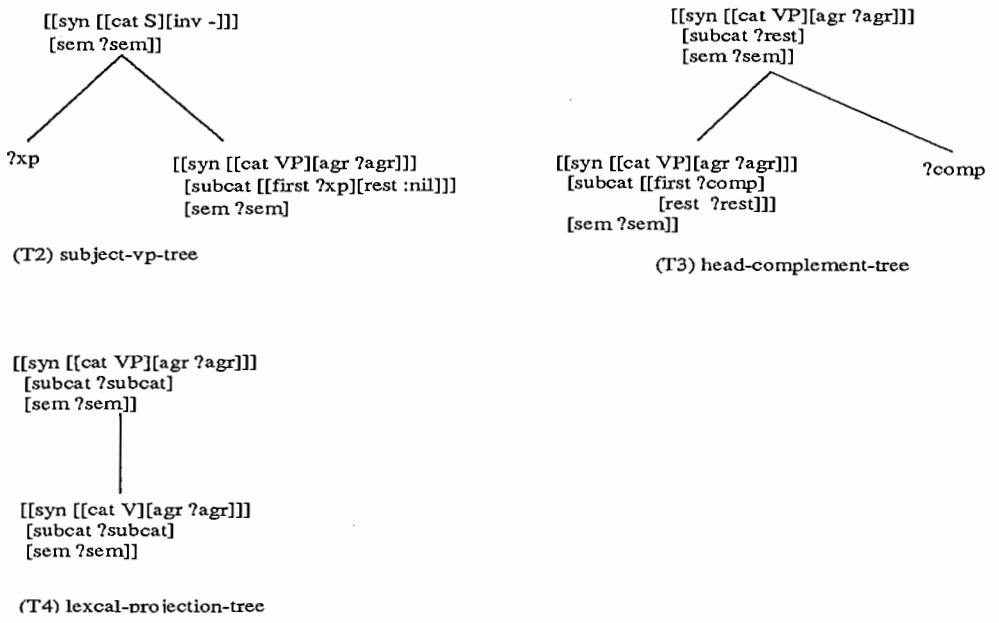

図 A.5: Trees for head-complement structures

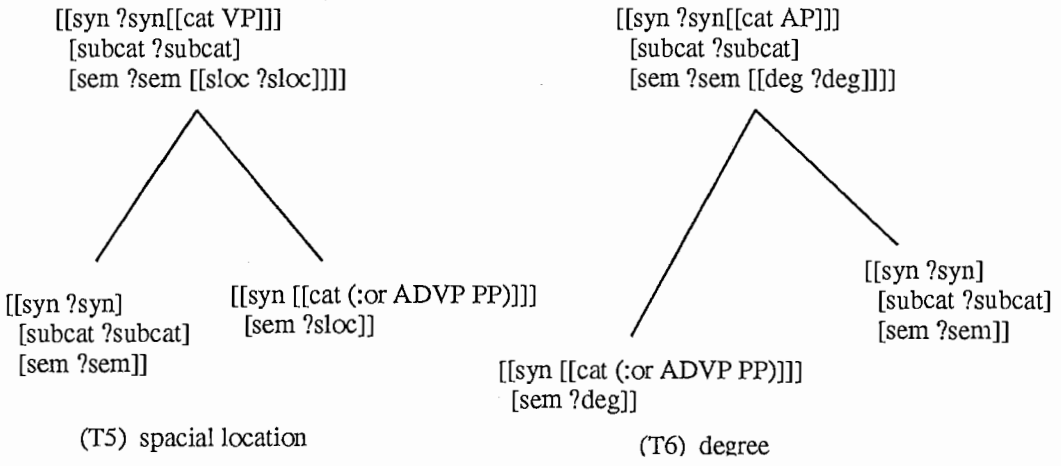

図 A.6: Trees for adjunct-head structures

Specific patterns are indispensable for phrase structures incapable of being created by general patterns in a compositional way. But they are not restricted to non-compositional structures.

A tree for a specific pattern can contain substitution nodes and adjunction nodes. feature-structures on these nodes control possible trees applied to the nodes.

Figure A.7 shows a pattern for a canned phrase. This is almost the same as a parse tree of the expression. Note that subcat frames of adjunction nodes (AP and VP) set to nil. They block inappropriate trees like head-complement trees.

To compare the TAG style with the HPSG style, we show a TAG tree for a declarative sentence with "hit" in Figure A.8.

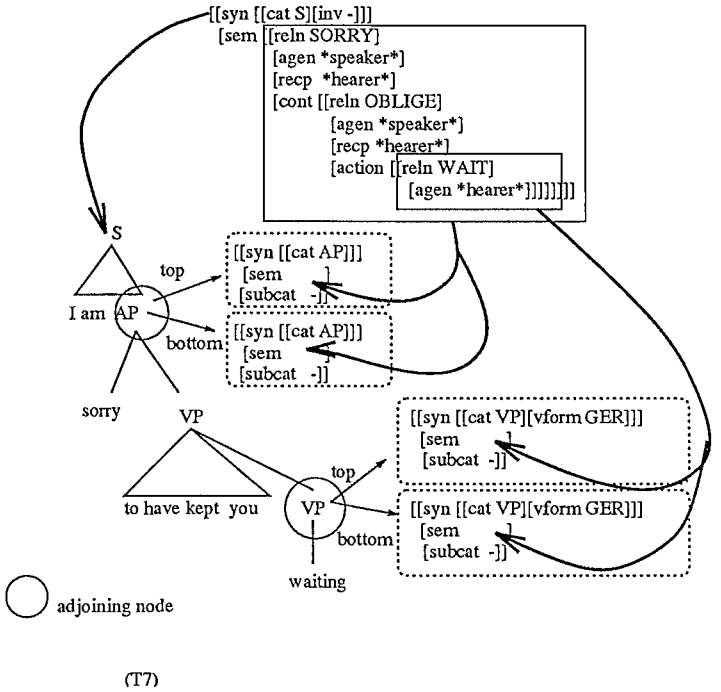

図 A.7: A tree for a canned phrase (with adjunction)

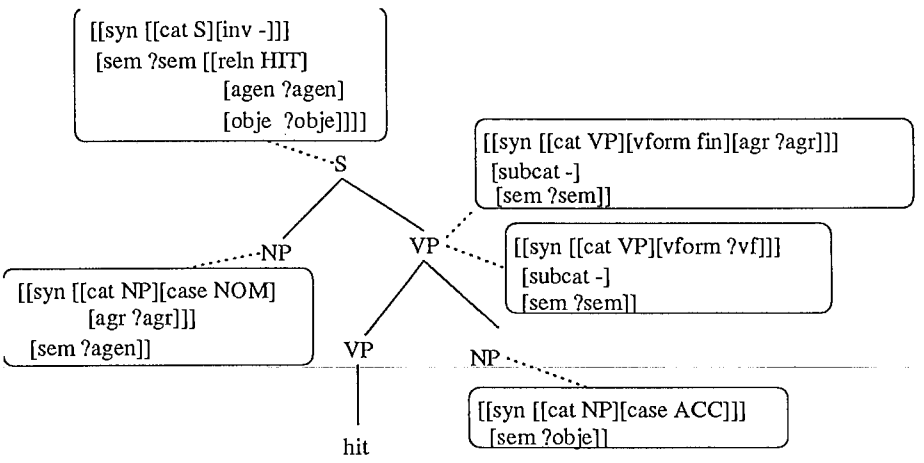

図 A.8: A tree for a declarative sentence with "hit"

A.3 Generation Algorithm

Generation is a process of combining elementary trees so as to satisfy given semantic and syntactic constraints. Our generation system carries out this combination process using a semantic head-driven strategy [18].

Input to the generation is a feature structure containing syntactic and semantic constraints. This feature structure is used as the root feature (RF).

We classify trees into two types: chain-type elementary trees and non-chain-type elementary trees. A *chain-type elementary tree* has a leaf node whose semantic feature structure is identical with that of the root node. A *non-chain-type* elementary tree does not have this characteristic. This classification of elementary trees is a version of SHDG's distinction between *chain rules* and *non-chain rules*, adapted for use with trees.

A *semantic spine node* is a node whose semantic feature is identical with that of the root.

A.3.1 Generation with Substitution Only

First, we explain our version of semantic head-driven generation when only substitution is needed. This is applicable when elementary trees in generation knowledge have no adjunction nodes. This means that the set of trees is equivalent to a unification-based context-free grammar. Therefore, we can use an existing algorithm.

The main routine of the generation process is as follows. This routine takes a node with a root feature, which is given as an input.

1. Create a chain for the given node.
2. Recursively call the generation process for every non-terminal node in the chain.

A chain for a node, n_{top}, is created in the following steps:

- **Preselection Step:**

  Collect all the elementary trees, each of which satisfies the following conditions:

  1. The semantic feature on the root node of the elementary tree subsumes the semantic feature of n_{top}.
  2. The syntactic category on the root node of the elementary tree is a member of the derivable categories for n_{top} (Derivable categories are obtained by a *link* table pre-compiled from the set of elementary trees [6]).

- **Top-down step:**

  1. Choose a non-chain-type tree from the preselected trees.
  2. Let the chosen tree be the initial value of the chain t_{chain}.

- **Bottom-up step:**

  1. If the root node of t_{chain} is unifiable with the top node, n_{top}, return t_{chain}; else, choose a chain-type elementary tree whose semantic head leaf node is unifiable with the root of t_{chain}.

2. Extend t_{chain} by unifying the semantic head leaf node of the chosen chain-type tree with the root of t_{chain}.
3. Go to 1 (of the bottom-up step).

Note that a chain is created non-deterministically, because the top-down step and the bottom-up step involve non-deterministic choice points.

Example with Substitution Only

A small example will help to clarify the above algorithm. We use the trees (T1) through (T7) presented in A.2.

Suppose the root node has the feature structure shown in Figure A.9, which means "Mary hit John at the station". A symbol with '*' is used here in stead of a complex feature-structure.

```
[[syn [[cat S][inv -]]]
 [sem [[reln hit]
       [agen *Mary]
       [recp *John]
       [sloc *Station]]]]
```

図 A.9: Root feature

First, relevant trees for the root node are selected. In this case, T1, T2, T3, T4 and T5 are selected because the semantic structure of each tree subsumes the SEM feature on the root node. Note that the "sloc" feature on the root of T5 is taken into account for selecting this tree, although the value of this feature is undefined. T1 is a non-chain-type, and the remaining trees are chain-type trees.

The top-down step chooses T1 because it is a non-chain type. Finally, a chain is created by iteratively and non-deterministically attaching one of the preselected chain-type trees until the top node of the chain is unifiable with the root node. Figure A.10 shows the created chain. The order of attachment is T1-T4-T3-T5-T2.

A.3.2 Extension to Adjoining Operation

This section describes an extension of the above algorithm so that it may handle adjoining operations.

In the extended algorithm, each node of the tree has a special slot, called an *obligatory tree slot*, for storing a tree. In the initial state, the obligatory tree slot of each node is empty.

The main routine is the same as that of the previous algorithm. Only the chaining steps are affected. The new chaining steps consist of four steps.

1. **Tree Preselection Step:**
   (a) Collect elementary trees in the same way as with the previous algorithm.
   (b) If the top node t_{top} contains a tree in the obligatory slot, add this tree to the set of collected trees.
2. **Tree Splitting Step1:**
   If a preselected elementary tree has an adjunction node that is also a semantic spine node, split the tree at this node, called a *split node*, into two elementary trees, say *the upper tree* and *the lower*

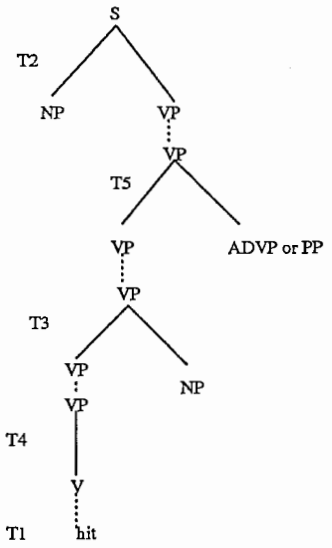

図 A.10: An example of a chain

tree. Note that the split node is split into a leaf node for the upper tree and the root node for the lower tree. The split node in the upper tree contain the top feature of the original split node and the split node in the lower tree has the bottom feature of the original split node.

Each split tree should still maintain the feature-structure-sharing relations of the original tree.

3. **Top-down Step:**

   (a) If the obligatory tree is of a non-chain type, then choose it; else, choose one non-chain-type tree from the preselected trees.

   (b) Let the chosen tree be the initial value of the chain t_{chain}.

4. **Bottom-up Step:**

   Extend t_{chain} in the same way as in the bottom-up step of the previous algorithm. If there is an obligatory tree in the preselected trees and it is not involved in t_{chain}, then discard t_{chain} and retry the extension of t_{chain}.

5. **Tree Splitting Step2:**

   If t_{chain} contains an adjunction node, say n_a, that is not a semantic spine node, then split t_{chain} at n_a. Again we call the split structures *the upper tree* and *the lower tree*. The top feature and the bottom feature on the original n_a are distributed to the upper tree and the lower tree, respectively. The lower structure is stored in the obligatory tree slot on n_a in the upper structure.

Example with Adjunction

To explain the algorithm, we use an artificial example. Suppose the root node has the feature structure shown in Figure A.11. It means "I'm very sorry to have kept you waiting at the station."

The preselection step selects all trees except T1 and T5 because these trees are not semantically relevant to the given semantic structure. Then, T7 is split at the AP node because this is an adjunction node

```
[[syn [[cat S]]]
 [sem [[reln SORRY]
       [agen *speaker*]
       [recp *hearer*]
       [deg  VERY]
       [cont [[reln KEEP]
              [agen *speaker*]
              [recp *hearer*]
              [action [[reln WAIT]
                       [agen *hearer*]
                       [sloc *Station]]]]]]]]
```

図 A.11: An input feature-structure

and it shares the SEM feature with the root node of the tree (i.e. a semantic spine node). The resulting trees are shown in Figure A.12

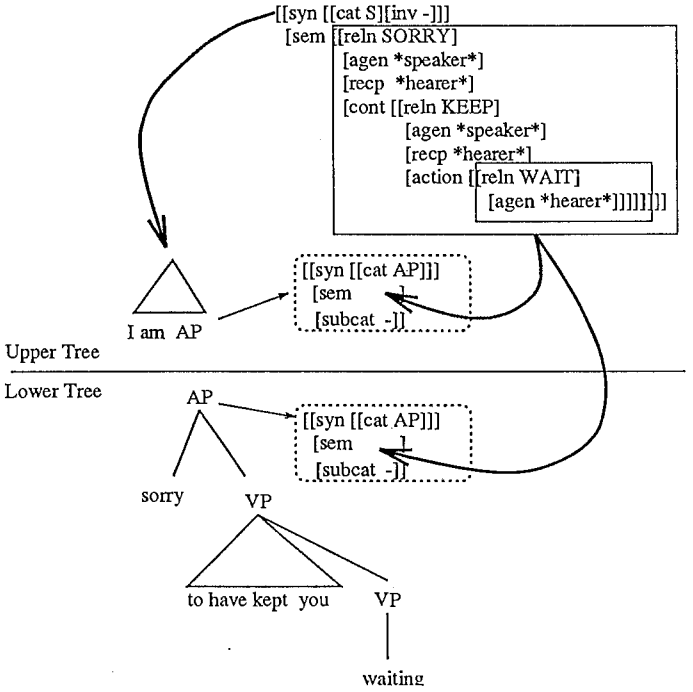

図 A.12: Splitting T7 at the semantic-spine node.

These split trees and other preselected trees are used to create a chain. Figure A.13 shows the creation of the chain.

The final step of chain creation is again tree splitting for non-semantic spine nodes. The chain contains one adjoining node which does not share its SEM feature with the root. Therefore the tree copied from the chain is split at this node into two parts, and the lower part is stored in the obligatory tree slot in the split node (Figure A.14).

Then system goes into the second iteration, where the lowest VP (vp2) is expanded to a subtree.

For this node, T2, T3, T4, T5 are preselected. Then, in the same way as for the first iteration, the chain shown in Figure A.15 is created. This chain is instantiated and substituted for *vp*2 in Figure A.14.

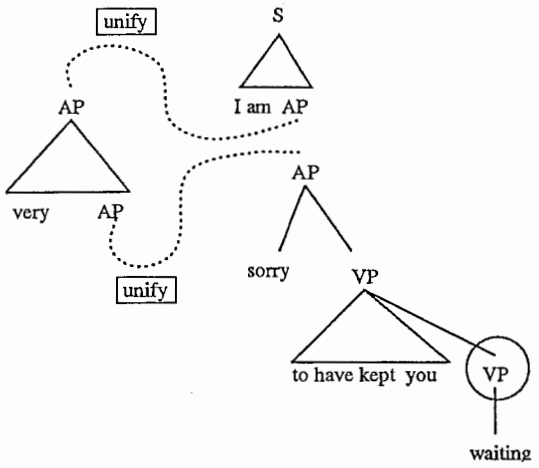

図 A.13: Adjoining an auxiliary tree to T7

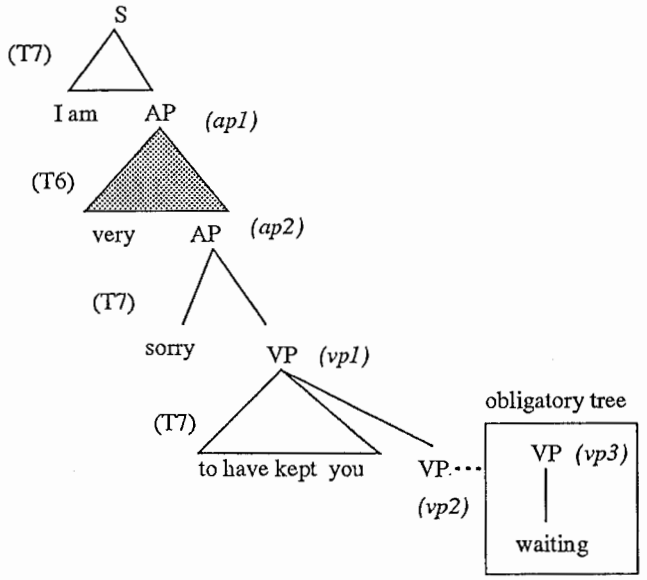

図 A.14: Splitting T7 at the non-semantic-spine node

A.3.3 Ordering Generated Structures

The generation algorithm described in the previous sections generates all possible syntactic structures. The current implementation orders generated structures according to the number of the elementary trees used for each structure and outputs the structure with the smallest number of elementary trees. This heuristic reflects the accepted obsavation that the structure created from specific knowledge is preferred.

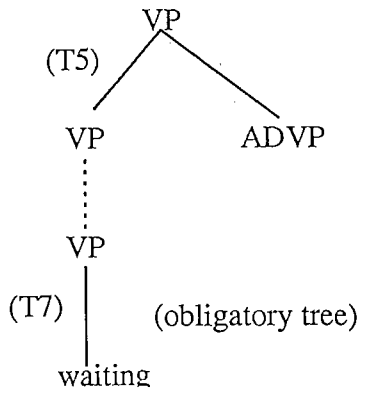

図 A.15: A chain for VP with "waiting"

付録 B

生成処理関数一覧 (V4.0)

B.1 生成処理の実行

(gb)

辞書のオープンなどを行ない、生成処理を実行可能にする。成功するとプロンプトが “$>” になる。

(gq)

辞書のクローズなどを行なう。

(gx)

英文生成を実行する。文番号の範囲を聞いてくる。

(gg [m [n]])

文番号 m から n まで英文生成を実行する。もし、m,n を省略した場合には入力ファイルの全ての文番号が生成処理の対象となる。

(gi id1 id2...)

指定されたＩＤと一致する変換結果に対して生成処理を実行する。ＩＤはストリングではなく英数字列のシンボルで指定する。

(generate0 rws-node)

意味素性構造 (データ構造は変換で定義されている素性構造用構造体 [rws::node]) を引数として生成処理を実行する。

(generate0_internal f-node)

意味素性構造 (データ構造は生成で定義されている素性構造用構造体 [user:f_node]) を引数として生成処理を実行する。

(gen_loop4 m [n])

変換結果ファイルのm番目からn番目までの変換結果構造に対して生成処理をする。nを省略するとmから最後までを実行する。

(gen_loop3)

プロンプト"%"を出し、以下のコマンドを受け付ける。

コマンド	機能
x(eXecute generation)	生成処理モードに入る。
q(Quit session)	プログラムを終了する。
e(Evalate S-expression)	リスプのS式を評価する。
s(show Status)	デバッグモードなどパラメータの状態を表示する。
l(List sentences)	処理できる文の一覧を表示する。
?(help)	コマンドの一覧とその動作を表示する。

(g?)

生成処理の実行にかかわる関数の説明を表示する。

B.2 規則ファイル、実行パラメータなどの表示

(gs)

現在のシステムの設定値全てを表示する。

(gs0)

処理系のバージョン番号を表示する。

(gs1)

デバッグ情報のレベルを表示する。

(gs2)

システムの動作モードなどを表示する。

(gs3)

現在、参照している辞書や規則ファイルを表示する。

(gs4)

辞書や規則ファイルのデフォルトのパス名を表示する。

(gs5)

バックアップファイルを作るときの拡張子を表示する。

B.3 生成処理モニター

(set_g_monitor [t/nil])

生成処理の各種の数値のモニターを開始 (t) または解除 (nil) する。引数を省略すると現在の状態を表示する。

(print_g_monitor)

モニターした情報を表示する。

(reset_g_monitor)

それまでモニターした値をクリアする（=全ての値を 0 にする）。

(print_g_monitor2 [:items items :ids ids :from from :to to])

モニターした情報を一文毎に表示する。

:items 表示する情報のリスト

:ID	文のＩＤ
:TIME	ユーザプロセスが消費したＣＰＵ時間
:BYTES	消費したセルのバイト数
:ABCCESS	参照されたＰＤのリスト
:CACHE	ＰＤキャッシュがヒットしなかった回数
:ACTIVATES	活性化されたＰＤのリスト
:APPLIED	適用されたＰＤのリスト
:P_NODE_MATRIX	Ｐノードマトリックスの更新回数
:AMBIGUITIES	多義接点の個数
:SUBTREES	仮説木の個数
:OUTPUTS	生成文字列
:TREE	生成結果の木構造

デフォルトは :items '(:ID :APPLIED :OUTPUTS)

:ids 表示する文ＩＤのリスト

:from, :to 番号による範囲指定

(print_g_monitor3 [:mode mode])

引数 :mode は :APPLIED または :UNAPPLIED をとる。:APPLIED の場合、ＰＤ規則に対して適用された文ＩＤのリストを表示する。:UNAPPLIED の場合、適用されなかった規則名の一覧を表示する。デフォルトは:APPLIED。

B.3.1 変換結果ファイル

(g_set_input_mode [:INDEX/:ON-MEMORY/:NAIVE])

変換結果ファイルの読み込みモードを切り換える。デフォルトは :index 。

:INDEX 生成の際にディスクからインデックスアクセスにより読み込む

:ON-MEMORY 生成に先だってディスクからメモリに読み込んでおく。生成の際にはディスクをアクセスしない。

(set_transout_path [file])

変換結果ファイルを指定する。file を省略すると現在の設定を表示する。

(load_transout [file])

読み込みモードが:on-memory の場合、file で指定された変換結果ファイルをロードする。読み込みモードが:index の場合、file で指定された変換結果ファイル上の各素性構造に対するインデックスを作成する。

(set_transout_index [file])

file で指定された変換結果ファイルから検索インデックスを生成する。file を省略すると前回指定されたファイルを対象にする。読み込みモードが:index の場合の load_transout と同等な動作を行なう。

B.4 要素木 (PD) の管理

B.4.1 メモリーＰＤ規則アクセス関数

(load_pd [file][:init t/nil][pd1 pd2...])

ＰＤ規則ファイルをメモリーＰＤとして主記憶上にロードする。

(clear_pd)

メモリーＰＤを主記憶から開放する。

(set_o_pd_path [file])

メモリーＰＤ規則ファイルのデフォルトを指定する。file を省略すると現在の設定を表示する。

B.4.2 PD 規則のコンパイル

(compile_pd [file])

PD 規則を 2 次記憶化アクセスのための形式に変換する。file を省略すると、前回コンパイルしたファイルがあればそれをもう一度コンパイルし、なければ何もしない。

B.4.3 2 次記憶化 PD 規則アクセス関数

(open_pd)

ＰＤ規則ファイルをオープンする。

(close_pd)

ＰＤ規則ファイルをクローズする。

(set_o_pd [t/nil])

メモリーＰＤを使用するか否かを指定する。t/nil を省略すると現在の設定を表示する。

(set_m_pd [t/nil])

マスターＰＤ規則を使用するか否かを指定する。t/nil を省略すると現在の設定を表示する。

(set_u_pd [t/nil])

ユーザーＰＤ規則を使用するか否かを指定する。t/nil を省略すると現在の設定を表示する。

(set_pd [o u m])

オンメモリーＰＤ (=o)、ユーザＰＤ (=u)、マスターＰＤ (=m) のうち、使用するＰＤ（1～3 個）を指定する。引数を省略すると現在の設定値を表示する。

(set_m_pd_path [file])

デフォルトのマスターＰＤ規則ファイルのファイル名を指定する。file を省略すると現在の設定を表示する。

(set_u_pd_path [file])

デフォルトのユーザーＰＤ規則ファイルのファイル名を指定する。file を省略すると現在の設定を表示する。

(open_m_pd [file])

file をマスターＰＤ規則ファイルとしてオープンする。file を省略するとデフォルトのマスターＰＤ規則をオープンする。

(open_u_pd [file])

file をユーザーＰＤ規則ファイルとしてオープンする。file を省略するとデフォルトのユーザーＰＤ規則をオープンする。

(close_m_pd)

マスターＰＤ規則だけをクローズする。

(close_u_pd)

ユーザーＰＤ規則だけをクローズする。

(set_cache [number])

キャッシュに記憶するＰＤ規則数の上限値を指定する。number を省略すると現在の設定値を表示する。（デフォルト値は４０）

B.4.4 ＰＤの削除と復活

(kill_pd pd1 pd2...)

pd1 pd2... の様に指定されたＰＤを無効にする。

(delete_pd pd1 pd2...)

上と同じ。

(revive_pd pd1 pd2...)

(kill_pd) によって無効にされたＰＤの内で pd1 pd2... の様に指定されたＰＤを有効にする。

(undelete_pd pd1 pd2...)

上と同じ。

B.5 デバッグ機能

B.5.1 トレース

(g_set_degug [levels])

表示すべきトレース情報を設定する。levels を省略すると現在の設定値を表示する。

(set_g_dbg [levels])

デバッグレベルを設定する。

(g_open_output [file])

デバッグ情報を出力するファイルをオープンする。file を省略するとデフォルトのファイルをオープンする。

(g_close_output)

デバッグ情報を出力するファイルをクローズする。

※デバッグレベルの説明

デバッグモード	表示する項目
1	展開節点と品詞。 活性化されたＰＤ（非伝搬型ＰＤは＜＞で囲んで表示）。 適用されるＰＤ連鎖。 スラッシュターミネイト。 多義解消（展開失敗）。
2	生成中の木構造。
3	展開節点の素性構造。
4	スラッシュターミネイトした節点の素性構造。
5	生成結果の素性構造表示。
6	適用されたＰＤ名。
7	活性化されたＰＤの間の接続関係。 作成中のＰＤ列（非伝搬型ＰＤは＜＞で囲む）。 失敗した場合はその理由。
２１	一文毎の処理時間。
２２	メモリ消費量。
0	全部表示（１２３４６７２１２２と同じ）。

２１２	形態素生成結果の後にギャップのリストを出力する。
３１２	ＬＲパーザが返す句構造からＰＤ列を作る過程。
３１３	ＰＤ列を単一化して素性構造を作る過程。
３１０	３１２、３１３両方の出力。

B.5.2 ＰＤ連鎖検査用ツール

`(s_checker [:node node] :pds '(pd1 pd2...))`

ノード node とＰＤ列 (pd1 pd2...) が単一化可能か否かを調べる。node を省略すると現在の展開節点との単一化可能性をチェックする。node を nil にするとＰＤ列だけの単一化可能性をチェックする。

B.5.3 木構造プリンタ

`(tpr [tree])`

生成木構造を表示する。tree を指定しなければ現在の生成木 (*current_tree*) が対象になる。

`(set_tpr_mode [:number :names nil])`

木構造の節点に［ＰＤの個数］と［ＰＤ名］のどちらか（または両方）の表示を指示する。引数を省略すると現在の設定値を返す。

`(add_tpr_features` $< path >< path > ....$`)`

$< path >$ で表されるパスの素性構造を木構造の節点で表示させる情報に加える。

`(remove_tpr_features` $< path >< path > ....$ `)`

$< path >$ で表されるパスの素性構造を木構造の節点で表示させる情報から削除する。パスの指定がないときは表示すべき素性を全て削除する。

B.5.4 ブレーク条件の設定

`(gen_break_before fs)`

展開節点が fs と単一化可能である時に、その節点を展開する前にブレークする。

`(gen_break_after fs)`

展開節点が fs と単一化可能である時に、その節点を展開した後でブレークする。

`(gen_break_endproc )`

最終的な木構造が作成された時点でブレークする。

`(gen_unbreak [mode])`

ブレーク条件を解除する。mode には以下のものが指定できる。mode が指定されないときは [mode t] と解釈される。

引数の値	動作
t (default)	全てのブレーク条件を削除する。
fs	fs と単一化可能なブレーク条件を解除する
endproc	「最終的な木構造ができた時ブレークする指定」を解除する。
nil	全てのブレーク条件を表示して、番号でユーザが選択する。

B.5.5 ブレーク中に利用可能な関数

(show_agenda [agenda])

アジェンダにある節点の素性構造を表示する。agenda を指定しなければ現在のアジェンダ (*current_agenda*) が対象になる。

(show_node number [tree])

生成木構造の各節点に付与される節点番号をキーとしてその節点の素性構造を表示する。tree を指定しなければ現在の生成木 (*current_tree*) が対象になる。

(show_break)

現在設定されているブレーク条件の一覧を表示する。

(show_pd name)

ＰＤ名をキーとしてその内容をＰＤ定義フォーマットで表示する。

B.6 動作モード

B.6.1 デフォルトの意味構造

(use_defsem [t/nil])

defsem 素性を有効／無効にする。

B.6.2 環境制約

(dev_env_feature_path path)

入力素性構造の環境制約への素性パスを指定する。デフォルトは '(PRAG) 。

B.6.3 ＰＤ接続可能行列

(set_connectable [t/nil])

接続可能行列の使用を変更する。

B.6.4 処理の順序

(set_expand_mode [:para/:fast/:depth])

ある節点に複数の鎖構造が適用できる場合の処理の順序を設定する。

引数の値	展開の仕方
:para	幅優先展開モード（全解）
:fast	深さ優先、ファーストヒットモード（単一解）
:depth	深さ優先モード（全解）

:para/:fast/:depth を省略すると現在の設定値を表示する。

B.7 定数素性値の伝搬によるグラフの刈り込み

(generalized_filtering [t/nil])

枝刈りの使用の可否を指定する。引数を省略すると現在値を表示する。デフォルトは t

(generalized_filtering_paths [paths])

展開節点の素性構造の部分構造を指定する素性パスのリストを指定する。引数を省略すると現在値を表示する。デフォルトは '((SYN))。この素性パスで指定される部分素性構造の中にある定数素性値 (:ATOM, :SET, :NOT) を枝刈りに使う。

(generalized_filtering_ignore_paths [paths])

展開節点の素性構造の部分構造を指定する素性パスのリストを指定する。引数を省略すると現在値を表示する。デフォルトは '((SYN CAT))。この素性パスで指定される部 分素性構造は枝刈りには使用しない。

B.7.1 その他の実行パラメータ

以下のパラメータは特に変更不要である。

(set_bu_mode [:copy/:ncopy])

ＰＤの組み合わせを作るときに、途中までの素性構造をコピーしながら残すモードと、コピーせずＰＤの組み合わせ毎に作り直すモードとを選ぶ。引数を省略すると現在の設定値を表示する。デフォルトは :copy。

(enable_recursive [t/nil])

ＰＤ再帰の可否を決めるスイッチ。ボトムアップモードの時だけ有効。

(dynamic_filtering [t/nil])

活性化されたＰＤだけで動的に接続可能行列を作るためのスイッチ。

(set_cat_node [t/nil])

ＰＤの品詞を予めスロットにセットしておくか否かを指定する。

(structure_sharing [t/nil])

構造共有を行うか否かを指定する。

(p_node_matrix [t/nil])

Ｐノードの接続関係を学習するか否かを指定する。

(preload_empty_pds [t/nil])

無意味型ＰＤを予め活性化しておくか否かを指定する。

(pd_net [t/nil])

ＰＤ検索に多重化索引を使用するか否かを指定する。

(fs_print_order arcs)

素性構造プリンタに新たな表示の優先順序を指定する。arcs は優先順に並べたアークラベルのリストで、順序の指定されていない素性の表示は後回しにされる。

B.8 生成結果ファイルの比較プログラム

(g_comp file1 file2 &key (infl nil)

２つの生成結果ファイル (file1 と file2) を比べる。:infl nil （デフォルト）の場合は、形態素生成前の結果を比較し、:infl t とすると形態素生成結果を比較する。

B.9 評価用ツール

(print_qualified_pd data_file out_file conditions)

デバッグプリント付きの生成結果ファイルをしらべ、そのファイルの中で生成に使われたＰＤのＰＤ定義をファイル出力する。

data_file 生成結果ファイル（デバッグレベル１、６）

out_file 出力先ファイル名

conditions ＰＤを選択する条件

引数の値	動作
:contributed	生成結果に寄与があったＰＤを選ぶ
:applied	生成中にとにかく適用されたＰＤ
:activated	活性化されたＰＤ

条件の前に “-” を付けると、その条件を満たすＰＤを除く事が出来る

`(print_unspecific_pd &rest types)`

根節点か葉節点で品詞が特定されていないノードを含むＰＤがあれば、そのＰＤ定義を表示する。types は特定されない品詞のタイプで、デフォルト値は (:not :var)。

`(print_1_node_pd )`

単節点ＰＤがあれば、そのＰＤ定義を表示する。

付録 A

木の融合操作の試み (Merging trees)

付加語を効率的に扱うために、木と木を組み合わせる操作として新たに融合 (merging) という操作を導入する。

A.1 融合 (The Merging Operation)

融合操作とはある木T1の根節点n1を他の木T2の根節点n2に重ね合わせる操作である。融合操作において、T1の子節点とT2の子節点との間の順序関係は未定義である。従って、処理系はT1とT2を組み合わせる段階では図A.1の(1)または(2)のいずれかの構造を暫定的に生成する。全ての要素木の結合が終了した後、融合操作の行なわれた節点に対してその節点の子節点の順序に関する制約が適用される。

The merging operation attaches the root of a tree, T1, onto the root of another tree, T2. This operation does not specify the order of childen nodes of the merged trees. Instead, the order is determined by linear ordering constraints explained in the next subsection.

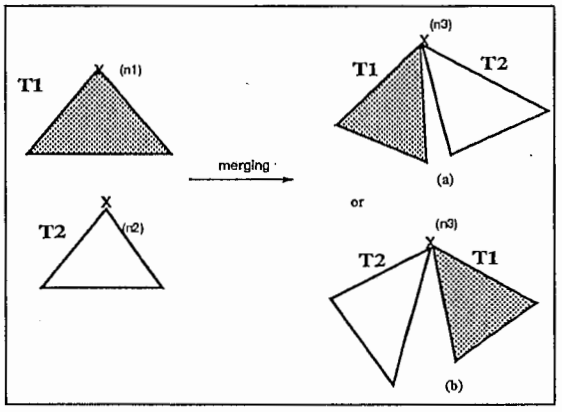

図 A.1: 融合 (merging)

他の木に融合可能な木を融合木 merging tree と呼ぶ。融合木の記述シンタックスは構造記述の部分を除いて基本的に他の要素木と同様である。融合木の構造記述は一つの根節点シンボルと一つ以上の子節点および必ず一つの "*" を子節点として含む (図A.2)。The syntax of a tree which can be merged to another tree is almost the same as that of an elementary tree described in Chapter 3 except the *:internal_structure* part(Fig.A.2.

:internal_structure (VP * PP)

図 A.2: 融合木の構造記述の例 (An example of the internal structure of a merging tree)

A.2 順序制約の適用 (Linear Order of Merged Trees)

全ての要素木の結合が終了した結果の木に対して、順序制約記述の適用を行なう。順序制約記述の形式は次の通りである。

もし融合木の結合している節点のうちで、$< mother - fs >$ の素性構造と単一化可能な節点があれば、その子節点の素性構造を並べた時に:c_order で規定されている素性構造の前後関係と矛盾しないように再配列する。

(cog

:name *name*

:mother *feature structure*

:c_order *a list of feature structures*)

なお、現在のバージョンではこの再配列の処理は融合された木に対してのみ有効である[1]。

[1] 理由は特にない

参考文献

[1] Shieber, S. M.: *An Introduction to Unification-Based Approaches to Grammar*, CSLI (1985).

[2] Vijay-Shanker, K. and Joshi, A. K.: Feature Structure Based Tree Adjoining Grammars, in *Proceedings of ACL-88* (1988).

[3] 郡司隆男：自然言語の文法理論, 産業図書 (1987).

[4] Pollard, C. and Sag, I. A.: *Information-based Syntax and Semantics*, CSLI (1987).

[5] Kikui, G.: Feature Structure-based Semantic Head-driven Generation, in *Proceedings of Coling-92* (1992).

[6] Matsumoto, Y., Tanaka, H., Hirakawa, H., Miyoshi, H. and Yasukawa, H.: BUP, *New Generation Computing* (1983).

[7] Vijay-Shanker, K.: Using Descriptions of Trees in a Tree Adjoining Grammar, *Computational Linguistics*, Vol. 18, No. 4 (1992).

[8] 春野雅彦, 伝康晴, 松本裕治, 長尾真：ボトムアップチャート法に基づく並列文生成, 情報処理学会自然言語処理研究会資料 (92-NL-88) (1992).

[9] 菊井玄一郎, 関倫彦：英語構文生成規則解説書 –ASURA における英語構文生成知識 –, Technical Report TR-I-0359, ATR 自動翻訳電話研究所 (1993).

[10] Tropf, H. S.: The German Grammar for ASURA, Technical Report TR-I-0289, ATR 自動翻訳電話研究所 (1993).

[11] Tomokiyo, M. and Uratani, N.: Japanese generation within ASURA framework, Technical Report TR-I-0349, ATR 自動翻訳電話研究所 (1993).

[12] 菊井玄一郎, 渡辺学, 関倫彦：形態素生成処理解説書 –ASURA における形態素生成処理 –, Technical Report TR-I-0361, ATR 自動翻訳電話研究所 (1993).

[13] Seligman, M.: Morphological Facilities for German Generation in ASURA, Technical Report TR-I-0362, ATR 自動翻訳電話研究所 (1993).

[14] Hovy, E. H.: *Generating Natural Language Under Pragmatic Constraints*, PhD thesis, Yale University (1987).

[15] Jacobs, P. S.: A generator for natural language interfaces, in al, et D. D. M. ed., *Natural Language Generation Systems*, chapter 7, Springer-Verlag (1988).

[16] Abeillé, A. and Schabes, Y.: Parsing Idioms in Lexicalized TAGs, in *Proceedings of ACL-89* (1989).

[17] Vijay-Shanker, K. and Schabes, Y.: Structure Sharing in Lexicalized Tree Adjoining Grammars, in *Proceedings of Coling-92* (1992).

[18] Shieber, S. M. and Schabes, Y.: Synchronous Tree-Adjoining Grammars, in *Proceedings of Coling-90* (1990).

さくいん