TR-I-0330

言語変換処理系解説書

-素性構造書き換えシステム改良版-

Synopsis of Language Transfer Engine for ASURA

- Sytem and Users Manual for Feature Structure Rewriting System V2 -

鈴木 雅実　　古崎 博久*
Masami SUZUKI　Hirohisa KOSAKI

1993年3月

概要

素性構造書き換えシステムを用いた言語変換処理が最初にATRの音声言語翻訳実験システムに搭載されて以来、書き換えシステムには数々の改良が加えられた。そこで、従来の「素性構造書き換えシステムマニュアル」を更新するとともに(本書のPART I)、日英変換処理の実例を含めた利用者マニュアル(本書のPART II)を新たに作成して、利用者の便宜を図った。

ATR自動翻訳電話研究所
ATR Interpretin Telephony Research Laboratories
* 東洋情報システム株式会社
Toyo Information Systems Co., Ltd.

PART I

素性構造書き換えシステム改良版 システムマニュアル

Feature Structure Rewriting System V2 System Reference Manual

要旨

このPART Iは、TR-I-0187 素性構造書き換えシステムマニュアル(改定版)が作成された1991年11月以降に、システムに加えられた改良項目を反映させるため、記載内容を全面に見直したものである。TR-I-0187 の構成に従っているが、書き換えシステムにおける規則の記述方法（シンタックス）や、規則の適用方法について種々の変更が生じている。従って、本書のPART II 素性構造書き換えシステムのユーザーズマニュアルを合わせて参照されたい。

ATR自動翻訳電話研究所

ATR Interpretin Telephony Reserach Laboratories

もくじ

1 はじめに

本マニュアルは、素性構造の書き換えシステムについて述べたものである。書き換えシステムは素性構造を定義された規則にしたがって別の素性構造に書き換える。本マニュアルでは書き換え規則の定義、記述方法、シンタックス、適用方法について適宜書き換え規則の例を用いながら説明する。また、規則はコンパイルされてLISPインタプリタで実行されるが、書き換えシステム内で定義されている関数および書き換え規則コンパイラが出力するLISPコードについても簡単に述べる。

以下、簡単に各章の内容を説明する。2章では規則の例を用いて書き換え規則とはどのようなものか簡単に解説する。3章では書き換え規則の定義について説明する。4章および5章で書き換え規則の適用制御に用いられる規則適用制約と書き換え環境について、6章では各書き換え規則の検索方法と素性構造への適用について述べる。7章、8章、9章では、書き換え規則の記述(シンタックス)に関係するデータ構造、演算子、書き換え規則内での制御機構について説明する。完全なシンタックスは付録Aに記載されている。さらに10章では、タイプシステムに関して説明する。また、システムの実行方法については本書PART IIのユーザーズマニュアルに詳細を記述した。なお参考文献は、まとめて巻末に付した。

2 書き換え規則の構造

入力素性構造を検査し、別の素性構造に書き換える典型的な書き換え規則(以下、規則とも言う)は以下のような形式をしている。

規則1

```
on <reln> 送る
   in=  [[reln 送る]
         [agen ?agent]
         [recp ?recipient]
         [obje ?object]]
   out= [[reln send]
         [agen ?agent]
         [recp ?recipient]
         [obje ?object]]
end
```

書き換え規則はonで始まり、endで終了する。規則は規則定義部と規則本体に大きく分けられる。規則1において、規則定義部は“<reln> 送る”であり、規則本体部はin=に続く部分とout=に続く部分とからなる。

書き換え規則は素性構造の素性パスに対して定義される。書き換え規則は定義された素性を持つ素性構造に対して適用される。規則1は素性relnに対して定義された書き換え規則である。規則を定義する素性は<reln>のように“<”と“>”で囲んで記述する。書き換え規則はreln以外の任意の素性に対して定義できる。また、単純な一つの素性でなく素性パス(feature path)(複数の素性を連結したもの)に対して定義できる。規則は定義された素性パスを持つ素性構造に対して適用される。規則1の素性パスrelnに続く記述「送る」は、素性パスrelnの先の素性値の指定である。素性値にはatomicタイプの素性構造(シンボル)を記述する。ある入力素性構造が与えられた時、規則の定義部に記述された素性パス、素性値の指定を満足した規則が、入力素性構造に対して適用される。したがって、規則1は、入力素性構造がreln素性を持ち、reln素性の素性値が「送る」であるとき、規則1の本体が実行される。素性構造1はこの条件を満足するので、規則1は素性構造1に適用される。

素性構造 1

```
[[reln 送る]
 [agen *speaker*]
 [recp *hearer*]
 [obje *登録用紙*]]
```

素性構造 2

```
[[reln send]
 [agen *speaker*]
 [recp *hearer*]
 [obje *登録用紙*]]
```

規則本体のin= [...]は入力の素性構造のパターン記述であり、入力素性構造パターンと呼ぶ。入力素性構造パターンは入力素性構造とパターンマッチングによって照合され、パターンマッチングが成功すれば書き換えシステムは規則の入力素性構造パターン以降の定義を実行する。入力が入力素性構造パターンに一致しなければ規則の適用はその時点で失敗に終る。素性構造パターン中の“?”で始まるシンボルは変数である。変数は任意の素性構造と一致することができる。規則1が素性構造1に適用されると、入力素性構造と素性構造パターンのパターンマッチングが行なわれる。これにより、入力素性構造パターン中の変数?agent, ?recipient, ?objectには、それぞれ入力の部分素性構造*speaker*, *hearer*, *登録用紙*が一致する。この時、変数には構造的に対応する入力の部分素性構造が代入される。out= [...]は規則が生成する素性構造のパターン(出力素性構造パターン)である。規則1は入力素性構造をこの素性構造パターンに書き換える。出力素性構造パターンから素性構造を生成する際には、変数は値(素性構造)で置き換えられる。規則1を素性構造1に適用すると、素性構造1は素性構造2に書き換えられる。

3 書き換え規則の定義

ここでは、書き換え規則の定義について述べる。書き換え規則は規則定義部(definition)と規則本体(body)から構成される。

---

on 規則定義部
　　規則本体
end

書き換え規則の記述はonで始まりendで終る。onとendの間に規則定義部と規則本体が記述される。

書き換え規則は素性のパスに対して定義され、素性パスの定義は規則定義部に記述される。規則定義部のシンタックスを以下に示す。

---

on < *FeaturePath* > *AtomicFeatureStructure* [*ApplicationConstraints*]
　　body
end

規則定義部は、素性パス(*FeaturePath*), *AtomicFeatureStructure*, *ApplicationConstraints*からなる。*FeaturePath*には一つ以上の素性が記述される。書き換え規則の多くは、概念間の関係を表現するreln素性に定義されている。*FeaturePath*に二つ以上からなるの素性のパスを指定する時には、素性パスを構成する素性を空白で区切って記述する。

*AtomicFeatureStructure*は、*FeaturePath*で指定した素性パスの値(素性値)を指定する。*AtomicFeatureStructure*は、atomicタイプの素性構造でなくてはならない。*AtomicFeatureStructure*は省略することはできないが、:unspecifiedを指定することにより、任意の素性構造を素性パスの素性値として許す。

ApplicationConstraints は省略可能であり、既定値はLISP大域変数*default-rule-parameter*の値が代入される。詳しくは4参照。

書き換え規則は素性のパスに対して定義されるので、atomicタイプの素性構造には規則を定義できない。したがって、この書き換えシステムではcomplexタイプの

素性構造を任意の素性構造に書き換えることはできるが、atomicタイプの素性構造単体を書き換えることはできない。

素性構造 3

```
[[FIRST A]
 [REST [[FIRST B]
        [REST [[FIRST C]
               [REST []]]]]]]
```

規則 2

```
on <first> a
   in= [[first a]
        ?rest]
   out= [[first a+]
         ?rest]
end
```

規則 3

```
on <rest first> b
   in=  [[first a]
         [rest [[first b]
                ?rest]
   out= [[first a++]
         [rest [[first b++]
                ?rest]
end
```

規則2は素性“first”に対して定義されており、first素性の値が“a”であることを指定している。素性構造3は素性のパス“first”が存在し、その素性値は“a”であるので、規則2は素性構造3に適用可能である。規則2の適用の対象(入力素性構造)は、素性構造3全体である。

また、素性構造3は、素性のパスrest, firstが存在し[1]、素性のパス rest first の値はatomicな素性構造Bである。したがって、素性構造3は規則3の規則定義部の条件を満たしており、規則3も素性構造3に適用可能である。規則3のように素性パスに二段階の素性を記述したときも、書き換え対象である入力素性構造は規則2と同様に素性構造3全体である(素性構造3において素性パスを辿った部分構造ではない)。規則3の素性構造3への適用結果は素性構造4になる。

素性構造 4

```
[[FIRST A++]
 [REST [[FIRST B++]
        [REST [[FIRST C]
               [REST []]]]]]]
```

[1]素性構造3は素性構造のトップから素性を辿った時、rest素性を持ち、その素性値である部分素性構造にはfirst素性が存在し、その素性値はbである。

このようにネストした素性パスに規則の定義が可能なので、書き換え規則で注目しているatomicタイプの素性構造fを含むより大きな素性構造を書き換え対象にすることができる。例えば、サ変名詞を含む動詞句の意味構造を英語の意味構造に書き換える規則は、素性構造5のように記述される。規則4では、注目している(あるいは規則の定義対象)語彙は「登録」であるが、ネストした(一段以上の)素性のパスを記述することにより「登録」を含むより大きな素性構造(動詞句に対応する素性構造)が規則の入力素性構造となる。規則4を素性構造5に適用すると、規則4の入力素性構造は素性構造5全体になる。規則4の適用の結果、素性構造5は素性構造6に書き換えられる。

規則 4

```
on <obje restr reln> 登録
  in=  [[reln する]
        [obje [[parm !x []]
               [restr [[reln 登録]
                       [entity !x]]]]]
        ?rest]
  out= [[reln register]
        ?rest]
```

素性構造 5

```
[[reln する]
 [agen *speaker*]
 [obje [[parm !x[]]
        [restr [[reln 登録]
                [entity !x]]]]]]
```

素性構造 6

```
[[reln register]
 [agen *speaker*]]
```

4 規則適用制約

規則の適用条件として、前述した素性パス、素性値、入力素性構造パターンの他に規則適用制約(Application Constraints)がある。規則適用制約は規則の適用時に書き換え環境(rewriting environment)と照合され、規則適用制約の記述が書き換え環境で満たされていればその規則は適用される。書き換え環境は書き換えシステムの一つのモジュールであり、特定の状態を保持している。書き換え環境の状態、および、規則適用制約はそれぞれ属性リストの形式で記述される。この規則適用制約と書き換え環境の充足機構により、書き換え環境の状態を変更することにより適用可能な規則の集合を動的に決定でき、規則の適用を書き換え環境の状態設定によって制御できる。

---

in *AttributeValueList*

規則適用制約は規則定義部の*AtomicFeatureStruture*の後に、inに続けて記述する。規則適用制約*AttributeValueList*は属性-値の対の形式であり、

attribute1 value1 ... attributeN valueN

の形式をしている。*AttributeValueList*として記述された規則適用制約が書き換え環境で満たされた時、その規則は適用可能になる[2]。

書き換え規則の定義で規則適用制約は省略可能である。規則適用制約が省略されると、大域変数*default-rule-parameter*の値が書き換えシステムへの規則ロード時に規則適用制約に設定される。*default-rule-parameter*の値は:Phase :J-E :Type :Generalに初期化されている。

規則5では:Phase :J-E :type :Generalが規則適用制約である。規則適用制約には規則がどのような働きをするか、あるいは、どのクラスの属しているかを記述する。規則5の規則適用制約は、二つの属性(:Phase, :Type)を持ち、各々の属性値は:J-E, :Generalである。この規則適用制約の記述により、規則5が日英の言語間で語義変換をし、タイプは一般規則であることを示している。書き換え環境

[2]規則定義部の三つの制約素性パス、素性値、規則適用制約をすべて入力素性構造が満たした時、規則本体が実行される。

が:Phase属性と:Type属性を持ち、その値が各々:J-E、:Generalの時、規則5の規則適用制約が満たされ、規則5の適用が可能になる。

規則 5

```
on <reln> 送る-1 in :Phase :J-E :Type :General
   in= [[reln 送る-1]
        [AGEN ?AGEN]
        [RECP ?RECP]
        [OBJE ?OBJE]
?rest]
   out= [[reln send-VT-1]
         [AGEN ?AGEN]
         [RECP ?RECP]
         [OBJE ?OBJE]
 ?rest]
end
```

5 書き換え環境

5.1 書き換え環境の設定

書き換え環境の状態は、書き換え規則の二つの演算子set parameter、unset parameterおよび書き換え呼び出し(9.3章参照)によって変更できる。

---

set parameter *AttributeValueList*
unset parameter *attribute1 attribute2 ...*

---

set parameterは、属性リスト(*attribute value*のリスト)を書き換え環境に設定する。set parameterによる書き換え環境の変更では、*AttributeValueList*の記述にない属性は保持される。つまり、*AttributeValueList*の記述にない属性を書き換え環境が持っている時には、その属性は保持される。*AttributeValueList*の属性が、既に書き換え環境に存在する時は、値の変更が行なわれ、書き換え環境に存在しない時には書き換え環境への属性の追加が行なわれる。unset parameterは、*attribute*で指定された属性のリストを書き換え環境から削除する。

set parmaeterあるいはunset parameterを使って書き換え環境の状態を変更することにより、書き換え規則の適用をダイナミックに変更することができる。

例えば、アルファベットのaからeのソートされたリストと、aからeをそれぞれaaからeeに書き換える規則があるとする。このとき、a, b, c だけをaa, bb, cc, に変更したい時、書き換え環境の状態を変更し規則適用を制御することにより、実現できる。

```
規則 6 on <first> a
          in= [[first a]
               [rest ?rest]]
          ==> input with :rewrite :double
          out= input
       end
規則 7
       on <first> a in :rewrite :double
          in= [[first a]
               [rest ?rest]]
          out= [[first aa]
                [rest ?rest]]
```

規則 8

```
on <first> b in :rewrite :double
   in= [[first b]
        [rest ?rest]]
   out= [[first bb]
         [rest ?rest]]
```

規則 9

```
on <first> c in :rewrite :double
   in= [[first c]
        [rest ?rest]]
   unset parameter :rewrite
   out= [[first cc]
         [rest ?rest]]
```

規則 10

```
on <first> d in :rewrite :double
   in= [[first d]
        [rest ?rest]]
   out= [[first dd]
         [rest ?rest]]
```

規則 11

```
on <first> e in :rewrite :double
   in= [[first e]
        [rest ?rest]]
   out= [[first ee]
         [rest ?rest]]
```

素性構造 7

```
[[first a]
 [rest [[first b]
        [rest [[first c]
               [rest [[first d]
                      [rest [[first e]
                             [rest []]]]]]]]]]
```

```
(rws::transfer fs)
;;; ------------ Transfer Input ------------
 [[FIRST A]
  [REST [[FIRST B]
         [REST [[FIRST C]
                [REST [[FIRST D]
                       [REST [[FIRST E]
                              [REST []]]]]]]]]]


Parameter                    Value
---------                    -----
:PHASE                       :J-E
:TYPE                        :GENERAL


Parameter                    Value
---------                    -----
:PHASE                       :J-E
:TYPE                        :GENERAL
;;; | 1[1]:> A-203
;;;  1 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
;;; CALLED recursive subFS rewriting call on A.
;;; Set local parameter.

Parameter                    Value
---------                    -----
:PHASE                       :J-E
:TYPE                        :GENERAL
:REWRITE                     :DOUBLE
;;; |  | 2[1]:> A-204
;;; |  | 2[1]:< A-204 Success
;;; |  | 2[1]:> B-205
;;; |  | 2[1]:< B-205 Success
;;; |  | 2[1]:> C-206
;;; |  | 2[1]:< C-206 Success
;;; Set local parameter.

Parameter                    Value
---------                    -----
:PHASE                       :J-E
:TYPE                        :GENERAL
;;;  1 <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
;;; Terminate subFS Recusive Rewriting CALL on A.
;; Reset local parameter.
```

```
Parameter                    Value
---------                    -----
:PHASE                       :J-E
:TYPE                        :GENERAL
;;; | 1[1]:< A-203 Success

;;;=========== Transfer Result ============
 [[FIRST AA]
  [REST [[FIRST BB]
         [REST [[FIRST CC]
                [REST [[FIRST D]
                       [REST [[FIRST E]
                              [REST []]]]]]]]]]
;;; ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

6 書き換え規則の検索と適用

ここでは、書き換え規則の検索方法および適用方法について述べる。

6.1 書き換え規則の検索方法

図1に書き換え規則が格納されているルールベースの構造を示す。規則は以下の手順で検索される。

1. 素性パスの検索
   入力素性構造中に規則の定義されている素性パスが存在するかを検索する(図1中のFeature Pathテーブルを検索する)。規則はシステムにロードされると規則の素性パスがFeature Pathテーブルに登録され、コンパイルされた規則のコードがHash Tableに*AtomicFeatureStructure*をキーとして登録される。

2. 素性値の検査
   入力素性構造中に規則が定義されている素性パスが存在すれば(Feature Pathテーブルのエントリに一致する素性のパスが入力素性構造に存在すれば)、素性のパスにしたがって入力素性構造を辿り部分素性構造(素性値)を取り出す。部分素性構造がatomicタイプの素性構造であれば部分素性構造[3]を検索キーとしてハッシュテーブルを検索する。

3. 規則適用制約の検査
   atomicタイプの素性構造をキーとしてハッシュテーブルを検索した結果、定義されている規則があれば、それらの規則の規則適用制約が書き換え環境で満たされているかをチェックする。

4. 規則の適用
   規則適用制約の条件が書き換え環境で満たされれば、入力素性構造を対象に規則本体を実行する。

[3]正確にはnode構造体のvalue値

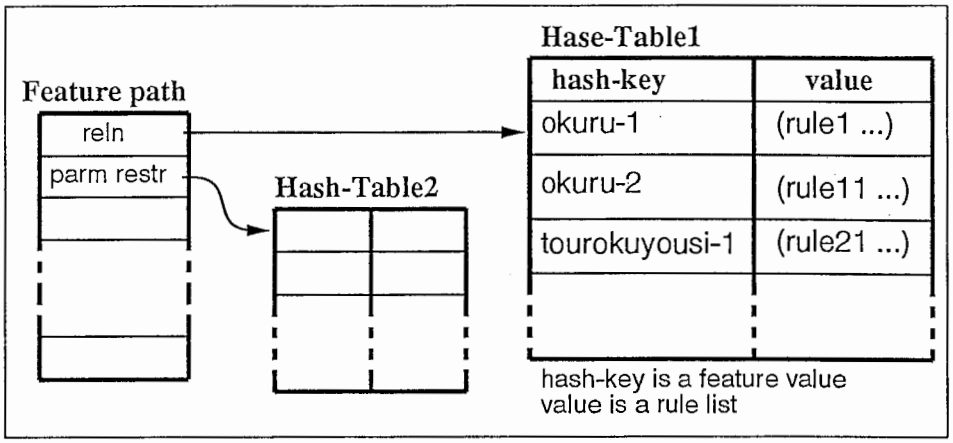

図 1: ルールベースの構造

6.2 書き換え規則の適用制御

書き換え規則の素性構造への適用は、基本的には書き換え呼び出し (rewriting call) によっておこなわれる。書き換えシステムでは、いくつかのタイプの書き換え呼び出しが用意されているが、ここで、その一つである rewrite 書き換え呼び出しについて説明する (その他の書き換え呼び出しについては9.3章参照)。

規則の検索の結果、適用可能な規則が複数個見つかったときには、それらの規則は並列に実行される。規則適用の結果、n 個の規則が適用に成功したとすると (ただし、各々の規則は書き換え結果として一つの素性構造を返すとする)、n 個の結果が返される (9.3参照)。

`rewrite` *fs* `with` *AttributeValueList* `by` *control*

素性構造*fs*に対して書き換え規則の適用を試みる。*fs*には、通常入力素性構造が代入されている変数rootあるいはinputが指定される。また、素性構造の局所的な部分に書き換え規則の適用を試みる場合は、変数に素性パス (7.7参照) の記述を行い、対象になる素性構造パターンを切り出すことができる。

AttributeValueList は書き換え環境の状態記述であり、書き換え環境を *AttributeValueList* に設定し、*fs*に対し規則の検索、適用(書き換え呼び出し)を試みる。書き換え呼出時の書き換え環境の設定は、演算子 set parameter

と異なり、書き換え環境の状態を完全に*AttributeValueList*で置き換える。*AttributeValueList*は属性-値の対のリスト形式であり任意個の属性が記述できる。

*control*は規則適用の制御記述であり、素性構造の辿りかた、および、規則適用の終了条件を指定する。*control*の値には:RECURSIVEおよび:LOOPを指定できる。*control*に:RECURSIVEが指定されると、complexタイプの素性構造*fs*をルートから再帰的に辿り、各々の部分素性構造に対して規則の検索および適用が試みられる。すべての部分素性構造に対して規則の適用が終了すれば、最終的な書き換え結果が返される。*control*に:RECURSIVEの指定がなければ、素性構造を辿らず素性構造*fs*のルートにだけ規則の検索と適用が行なわれ、その結果が返される。

*control*に:LOOPが指定さると、適用できる規則がなくなるまで、規則の検索と適用が繰り返し行なわれる。つまり、素性構造に対して適用が成功した規則があれば、その適用結果(書き換え結果)である素性構造に対し、再度、規則の検索と適用が行なわれる。この処理を、規則が見つからなくなるか、あるいは、すべての規則の適用に失敗するまで、繰り返し行なう。規則の適用を一度しか行わない場合は*control*に:ONCEを指定する。また、*control*の指定がない時には、素性構造のトップに対し規則が一度だけ適用される。

規則の適用制御に:LOOPを指定する時には、規則の適用が無限ループに陥らないよう注意が必要である。素性構造に素性を単に追加するだけの書き換え規則では、規則の入力素性構造パターンの条件記述で入力素性構造に追加すべき素性がないことを適用条件として与えないと、規則が無限に適用されてしまう。例えば、以下の規則を*control*が:LOOPのモードで適用されると、規則の適用が失敗しないので無限ループに陥る。

```
規則 12 on <a> b
          in=  [[a b]
                ?rest]
          out= [[a b]
                [c d]
                ?rest]
        end
```

以下に、rewriteオペレータを用いた規則例を示す[4]。

[4]書き換え規則13はメタな書き換え規則:mainである。規則:mainは変換過程のサブプロセス(各フェイズ)を制御するための規則である。規則:mainはLISP関数transにより呼び出される。

規則 13

```
on <> :main
   set ?SP to input.prag.speaker
   set ?HR to input.prag.hearer
   in= [[sem ?sem]                                    (1)
?rest]
   input= [[sem [[reln UNKNOWN-IFT]                   (2)
                [agen ?sp]
                [recp ?hr]
                [obje ?sem]]]
          ?rest]
   rewrite input.sem with :IF :REDUCE :TYPE :GENERAL by :LOOP:RECURSIVE (3)
   rewrite input.sem with :IF :REDUCE :TYPE :DEFAULT  by :LOOP :RECURSIVE (4)
   rewrite input.prag with :PHASE :JAPANESE :PRSP :INIT by :RECURSIVE (5)
   rewrite input.sem with :PHASE :JAPANESE  by :LOOP :RECURSIVE (6)
   rewrite input.sem with :PHASE :J-E :TYPE :IDIOM  by :LOOP :RECURSIVE (7)
   rewrite input.sem with :PHASE :J-E :TYPE :GENERAL  by :LOOP :RECURSIVE (8)
   rewrite input.sem with :PHASE :J-E :TYPE :DEFAULT  by :LOOP :RECURSIVE (9)
   rewrite input.prag with :PHASE :ELLIPSIS-RESOLUTION  by :RECURSIVE (10)
   rewrite input.sem with :PHASE :ENGLISH  by :RECURSIVE (11)
   rewrite input.sem with :PHASE :ASPECT-INIT by :RECURSIVE (12)
   rewrite input.sem with :PHASE :ASPECT by :RECURSIVE (13)
   return input
end
```

書き換え規則13は、日英の言語変換処理で用いられるトップレベルの変換規則の例である。この規則では、入力素性構造の検査(1)をまず行ない、入力素性構造を書き換えた(2)後、書き換え環境の状態を変更しながら11回rewriteの書き換え呼び出しを行なっている。この書き換え環境の設定をともなった書き換え呼び出しは、変換過程での各サブプロセス(フェーズ)に対応している。例えば、最初の書き換え呼び出しは、発話のタイプの決定を行なうサブプロセスであり、Generalな規則をまず適用(3)してから、defaultの規則を適用(4)している。(5)および(12)は初期化の処理であり、次に続く処理のために、素性構造に素性を追加する規則を適用するため、書き換えられた素性構造(素性の追加された素性構造)に再度同じような規則を適用する必要がないので、*control*部に:LOOPの指定はない。

6.3 規則の適用結果

書き換え規則が適用されると適用結果として書き換えられた素性構造[5]が返される。規則の適用は、次に示す事態が起これば失敗とみなされる。規則の適用が失敗すると規則内で行なわれたすべての処理は無効(素性構造はもとの状態に戻される)になる。また、書き換え呼び出しで、適用可能な規則が検索されなかった、あるいは、検索されたすべての規則の適用が失敗した時、定数emptyが返る。この時も、書き換え呼び出しの対象であった素性構造は保存される。

- エラー
- 入力素性構造パターンのパターンマッチングの失敗(9.1章参照)
- 出力素性構造パターンの生成の失敗(9.2章参照)
- failの実行

書き換え規則中で演算子out=あるいはreturn (9.2章参照)を用いて明示的に規則の戻り値を記述しないと、書き換え規則内で行なわれた処理は無効になる。例えば、規則中で入力素性構造に対してdeleteやadd、=を用いて素性の削除や追加、書き換えの操作を行うと、規則内では一連の操作は素性構造に対して反映される。しかし、演算子out=やreturnを用いて明示的に規則の適用結果を記述し規則を終了しないと、規則内部で行なわれた処理はすべて無効になり、規則の適用が終了した後、規則の入力素性構造には何の変化も起こさない。規則中で入力素性構造に対する操作を規則適用結果として反映させるためには、必ずreturnあるいはout=を用いて、規則が素性構造を返すことを明示的に記述しなければならない。

規則 14
```
on <a> b
    delete a from input
end
```

規則 15
```
on <a> b
    in= [[a b]
         [c d]]
    input.e = f
end
```

[5]正確には素性構造のリストが返される。これは、書き換え規則中から書き換え呼び出しが行なわれ、書き換え呼び出しの結果、素性構造がかえされることがあるからである。しかし、規則適用の結果複数の素性構造が返された場合、システムは自動的に各々の素性構造に対して並列に処理を進めるので、ユーザは規則が複数個の素性構造が返るか否かを気にする必要はない。

規則 16

```
on <a> b
   in= [[a b]
        [c ?x]]
   -> ?x
   fail
end
```

書き換え規則14、15、16では、各種演算子を用いて素性構造に操作を行なっているが、returnあるいはout=による明示的な返却値の指定がないので、規則を適用しても入力素性構造に変化は起こさない。

7 データ構造

7.1 素性構造パターン

素性構造パターン(feature structure pattern)は書き換え規則の入力である素性構造とのパターンマッチングや書き換えるべき素性構造の生成に用いられる。素性構造パターンは、構造上の違いにより、complexタイプ、atomicタイプ、leafタイプに分類される。また、素性構造パターンは変数を含む。

$symbol	atomicタイプの素性構造
[]	leaf タイプの素性構造
?symbol	変数

atomicタイプ素性構造の単体を表現するときは、素性構造シンボルに接頭辞$をつけて記述する。atomicタイプの素性構造パターンが、complexタイプの素性構造中に現れる時には接頭辞$は省略する。

leafタイプの素性構造は、[]で記述する。

変数は変数名(*symbol*)に接頭辞?をつけて記述する。

complexタイプの素性構造パターンは"["で始まり、"]"で終る。complexタイプの素性構造パターンの記述中には、atomicタイプおよびleafタイプの素性構造、変数が含まれる。以下に、complexタイプの素性構造パターンの例を示す。

素性構造 8

```
(complex タイプの素性構造パターンの例)
[[reln 下さい-request]
 [agen ?speaker]
 [recp ?hearer]
 [obje ?object]]
```

素性構造パターン8は変数を含んだ素性構造パターンであり、agen、recp、obje素性の値がそれぞれ変数である。変数は値を持つことができ、その有効範囲は一つの書き換え規則中でのみ有効である。変数が書き換え規則中で新たに生成された時は、値を持っていない。値を持たない変数はパターンマッチングにおいて任意の素性構造と一致することができる。素性構造パターン中の変数が値を持っている時

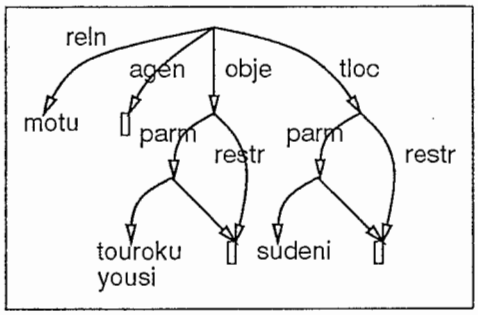

図 2: 素性構造のグラフ表示

には、素性構造パターンから素性構造を生成する際に[6]、変数はその値で置き換えられる。変数への素性構造の代入および変数の参照に関しては7.2章参照。

素性構造パターンには変数の他に、素性の同一性を表現するためにタグがある。

素性構造 9

```
(complex タイプの素性構造パターンの例)
[[reln  持つ-1]
 [agen []]
 [obje [[parm !x[]]
        [restr [[reln  登録用紙-1]
                [entity !x]]]]]
 [tloc [[parm !y[]]
        [restr [[reln  既に-1]
                [entity !y]]]]]]
```

素性構造パターン9の記述中の!x、!yは、**タグ**(tag)である。タグは素性構造の同一性を表現するために用いられる。タグに続いて素性構造が記述されると、局所的にその素性構造をタグでラベリングする。タグが単体で現れると、タグはラベリングされている素性構造で置き換えられる。素性構造9をグラフで表現すると図2になる(タグの詳細については7.5参照)。

素性構造は構造上complex, atomic, leaf [7], variableの四つのタイプに分類される。これらのタイプの他に、タイプシステムで定義されたより詳細なタイプを素性構

[6]素性構造パターンから素性構造の生成は、規則の適用時に動的に行なわれる。

[7]素性構造の単一化では、leafタイプ素性構造は任意の素性構造と単一化可能なので変数であるが、書き換えシステムのパターンマッチングでは、leafタイプ素性構造はleafタイプ素性構造としか一致しない。

造に与えることができる。タイプ定義、記述方法については10章参照。

7.2 変数

変数はユーザ変数(user variable)とシステム変数(system variable)に分類される。ユーザ変数はユーザが任意に定義できる変数であり、おもに素性構造パタン中で用いられる。システム変数は書き換えシステム内で予め定義されている変数であり、システムにより素性構造あるいは定数が自動的に代入される。

7.2.1 ユーザ変数

ユーザ変数は接頭辞"?"で始り、itおよびinput, root以外の任意のシンボルが使用できる。素性構造パターン中に記述された変数もユーザ変数である。規則中でユーザ変数が最初に現れた時には、変数は値を持っていない。

変数と素性構造のパターンマッチング

素性構造と素性構造パターンのマッチングにおいて、値の代入されていない変数は構造上対応する任意の素性構造と一致することができる。パターンマッチングが成功した場合、副作用として変数には対応する部分素性構造が代入される。パターンマッチングにおいて、既に値が代入されている変数が素性構造パタン中に現れた時には、変数の値と構造上対応するの部分素性構造がトークンとして同一か否かが[8]が検査される。トークンとして同一でなければ、パターンマッチングは失敗する。素性構造パターン中に同じ変数が二度以上現れた時は、パターンマッチングではその変数に対応する部分素性構造はトークンとして同一でなければならない。

素性構造 10
```
[[a ?var]
 [c ?var]]
```

素性構造 11
```
[[a !x b]
 [c !x]]
```

素性構造 12
```
[[a b]
 [c b]]
```

素性構造パターン10と素性構造11のパターンマッチングでは変数?varに対応する素性構造bはトークンとして同一であるので、パターンマッチングは成功する。パ

[8]構造上等しいのではなく、同じ素性構造をポイントしているか否かが検査される。

ターンマッチングの結果、変数?varには部分素性構造bが代入される。一方、素性構造パターン10と素性構造12のパターンマッチングでは、同じ素性構造bではあるがトークンとして同一ではないので、パターンマッチングは失敗する。パターンマッチングが失敗した時は、変数への値の代入は起こらない。

regular variableとrest variable

ユーザ変数はregular variableとrest variableに分類される。regular variableは任意の素性構造とマッチングがとられる変数である。rest variableは任意の素性と素性構造の対のリスト(fvlist)とマッチングがとられる変数である。regular variableとrest varibaleとは素性構造パターン中の現れる位置によって区別される。[9]

規則 17

```
on <reln> 持つ
   in= [[reln 持つ]
        [agen ?agent]
        [obje ?object]
        ?rest]
   out= [[reln have]
         [agen ?agen]
         [obje ?obje]
         ?rest]
end
```

規則 18

```
on <reln> 持つ
   in= [[reln 持つ]
        [agen ?agent]
        [obje ?object]]
   out= [[reln have]
         [agen ?agen]
         [obje ?obje]]
end
```

素性構造 13

```
[[reln 持つ]
 [agen *speaker*]
 [obje *登録用紙*]
 [tloc *昨日*]]
```

素性構造 14

```
[[reln have]
 [agen *speaker*]
 [obje *登録用紙*]
 [tloc *昨日*]]
```

規則17において、変数?restがrest variableであり、?agent、?objectがregular variableである。規則17が素性構造13に適用されると、規則中のin=に続く入力素性構造パターンと素性構造13のパターンマッチングは成功する。パターンマッチングの結果、変数?restには素性tlocと素性構造 *昨日*の対のリストが代入される。変数?agent、objectには*speaker*、*登録用紙*が代入される。

[9]現状の意味表現形式では、任意格は必須格と構造上同じレベルに記述される。したがって、動詞句に対応する意味表現である素性構造の素性のセットは不定である。rest variableは、素性のセットが不定の素性構造と素性構造パターンとのマッチングをとるためによく使用される。

```
?agen = *speaker*
?object = *登録用紙*
?rest = { [tloc *昨日*] }
```

規則17のout=に続く素性構造パターンの生成では、rest variableに代入されている素性と素性構造の対のリストが展開され素性構造14が生成される。

一方、規則18が素性構造13に適用されると、規則18の入力素性構造パターンの素性のセットと素性構造13の素性のセットが異なる(規則18の入力素性構造パターンには素性tlocがない)ため、パターンマッチングは失敗する。

素性のコンフリクト

rest variableを含む素性構造パターンによる素性構造の生成では、素性の重複を生じる場合がある。

規則 19

```
on <a> b
   in=  [[a b]
         ?rest]
   out= [[a b]
         [c d]
         ?rest]
```

規則19は、入力素性構造に素性cを追加する規則である。既に素性cをもった素性構造に規則19が適用されると、out=での素性構造パターンから素性構造生成時に、素性cにおいてコンフリクトが生じる。このように素性構造の生成において素性のコンフリクトが生じた時には、入力素性構造パターンに記述された素性が優先され、素性cの値はdになる。

```
[[a b]          ⇒        [[a b]
 [c x]]   規則19を適用    [c d]]
```

7.2.2 システム変数

システムで定義されているシステム変数には以下のものがある。

```
input
?input
root
?it
```

システム変数root、inputは、規則適用開始時にシステムによって値が初期化される。システム変数はユーザが値を変更することもできる。

システム変数rootには、書き換え対象である素性構造のルート[10]が代入される。

システム変数inputあるいは?inputには、規則適用対象である素性構造が、規則適用時に初期値として代入されている。

システム変数?itは、規則の記述内容に依存した変数であり、書き換え呼び出しの結果、および、条件式の評価結果が代入される。書き換え呼び出しが成功に終った時には、書き換え呼び出しによって書き換えられた素性構造が変数?itに代入される。書き換え呼び出しで適用可能な規則が見つからなかったり、すべての規則が適用に失敗に終った時には、変数?itには定数emptyが代入される。条件式の評価の後では、条件式の評価結果により、?itにはtrueあるいはfalseが代入される。変数?itおよび?inputは素性構造パターン中に記述されると代入されている値に展開されるが、変数input、rootは展開されない。

7.2.3 パス修飾子

素性構造パターンにパス修飾子path modifierを付けることにより、素性構造パターンのパターンマッチングの際に、素性のパスを辿ることができる。パス修飾子は素性構造パターン中に記述される一種の素性のパスである。パス修飾子を用いることにより、再帰的な部分構造を持った素性構造パターンを記述できる。パス修飾子は、downward modifierとupward modifierに分類される。パス修飾子は素性構造パターンに先だって記述される。

[10]素性構造の書き換えを行なうLISP関数に引数として渡された素性構造が変数rootに代入される。

1. downward modifier

   downward modifierは“<”で始まり、素性のパスが記述され、“>”で終る。素性のパスを素性構造の葉の方向に向かって辿る。パスの表現には、選言、繰り返しが指定できる。以下に、パス修飾子に記述できるオペレータを挙げる。

   (,) 素性の選言

   + 素性の一回以上の繰り返し

   * 素性の0回以上の繰り返し

2. upward modifier

   upward modifierは“(”で始まり、素性のパスが記述され、“)”で終る。素性のパスを素性構造の根の方向に向かって辿る。upward modifierには、素性の繰り返しや選言は記述できない。

素性構造 15

```
[[parm !A[[parm !B[[parm !C[[parm !D[]]
                              [restr [[reln 登録用紙]
                                      [entity !D]]]]]
                    [restr [[reln 赤い]
                            [obje !C]]]]]
          [restr [[reln 小さい]
                  [obje !B]]]]]
 [restr [[reln 送る]
         [agen [[label *speaker*]]]
         [obje !A]]]]
```

素性構造 16

```
[[parm !X[]]
 [restr [[reln 登録用紙]
         [entity !X]]]]
```

素性構造パターン

```
<parm* restr> [[reln 登録用紙]
               [entity []]]
```

はパターンマッチングにおいて、素性構造15および素性構造16と一致する。

7.2.4 素性名変数

書き換え規則適用中で格ラベルが決定されていない（特に任意格）可能性がある。この場合、あらかじめ定義した任意の変数で素性名を代用する。ただし、変数を用いた素性名に対応する素性値は変数を許可しない。なぜなら素性名、素性値がどちらも変数であれば、どのような素性パターンにもマッチするからである。
以下に素性名変数を用いた書き換え規則の例を示す。

規則 20

```
on <obje obje restr reln> そちら-2 in :PHASE :J-E :TYPE :IDIOM
  in= [[RELN  QUESTIONIF]
       [OBJE [[RELN  だ-IDENTICAL]
              [OBJE [[PARM !x[]]
                     [RESTR [[RELN  そちら-2]
                             [ENTITY !x]]]]]
              [IDEN [[PARM !y[]]
                     [RESTR [[reln.arg  NAMED]
                             [ENTITY !y]
                             [IDEN  ?iden]]]]]
               ?rest1]]
        ?rest2]
 out= [[RELN  QUESTIONIF]
       [OBJE [[RELN  だ-IDENTICAL]
              [OBJE [[PARM !x[]]
                     [RESTR [[RELN  this-PRON-2]
                             [ENTITY !x]]]]]
              [IDEN [[PARM !y[]]
                     [RESTR [[reln  NAMED]
                             [ENTITY !y]
                             [IDEN  ?iden]]]]]
               ?rest1]]
        ?rest2]
end
```

また、変数名をあらかじめ定義する形式を以下に示す。素性構造書き換えシステムの実行時における素性名変数の処理は大域変数*DEFNAME-LIST*を参照して行う。

```
(in-package 'USER)
```

```
(defvar *DEFNAME-LIST* nil)
(if (not (member 'reln.arg \underline{*DEFNAME-LIST*}))
(push 'reln.arg \underline{*DEFNAME-LIST*}))
```

7.3 外部変数

書き換え規則中にはこれまでに示した変数以外は利用不可であるが、一方で、11章で説明するようにLISP式が書き換え規則中に記述可能であるため、LISP式内では他の変数が利用できる。本システムの機能拡張で前文参照などを利用した文脈処理には外部変数（素性構造書き換えシステム中の変数）の利用が必要であり、基本機能として以下の項目がある。

1. 規則中の変数への外部変数のSET
2. 規則中の変数への外部変数からのGET

外部変数の書き換え規則内での参照は一旦、規則内で定義した局所変数に外部変数の内容をセットし、その変数で規則内の処理を実現する。

7.4 fvlist

fvlist は素性と素性値(素性構造)のリストである。素性構造と素性構造パターンのパターンマッチングにおいてrest variableに代入される対象はfvlist である。また、fvlist は“{”,“}”で囲んで記述することにより生成することもできる。

素性構造 17 { [a b] [c d] }

fvlist には、fvlistを素性構造へ追加する演算子(add)がある。

7.5 タグ

タグは素性構造パターン記述中で、素性構造がトークンとして同一であることを表現するために用いられる。タグは、タグに続く素性構造をタグ名でラベリングし、後で同じタグが単独で現れた時は、ラベリングされた素性構造が参照される。タグは、スコープによってlocal tagとglobal tagに分類される。

!*symbol*	(local tag)
@*symbol*	(global tag)

素性構造パターンにおいてタグに続けて素性構造が記述された時には、タグの定義とみなされる。このとき素性構造は*symbol*名でラベリングされる。タグの再定義はできない。タグが単独で現れた時には、ラベリングされた素性構造で置き換えられる。

local tag と global tag はタグを参照できるスコープが異なる。local tagは素性構造パターン中で定義され、参照範囲は素性構造パターン中に限定される。global tag は素性構造パターン中での振舞いはlocal tagと全く同じである。しかし、global tag のスコープは素性構造パターンの外でも有効であり、glocal tag によってラベリングされた素性構造を素性構造パタンの外でglobal tag を用いて参照できる。

global tag は素性構造パターン中ではlocal tagと同様に素性構造パターン中に現れる素性構造がトークンとして同一であることを示している。しかし、global tag を含む素性構造パターンと素性構造のパターンマッチングが行われると、global tag は入力素性構造の対応する部分構造をラベリングする。つまり、global tag は素性構造のパターンマッチングにおいて、制約が与えられたの変数のように振舞う。以下にglobal tagを含んだ素性構造パターンの例を挙げる。local tagの例については、7.1章参照。

規則 21
```
on <a> b
   in=  [[a b]
         [c @tag [[d e]
                 [f g]]]]
   out= [[a b+]
         [c @tag]]
end
```

規則 22
```
on <a> b
   in=  [[a b]
         [c ?var]
   out= [[a b+]
         [c ?var]]
end
```

素性構造 18
```
[[a b]
 [c [[d e]
     [f g]]]]
```

素性構造 19
```
[[a b+]
 [c [[d e]
     [f g]]]]
```

規則21を素性構造18に適用すると、global tag を含んだ素性構造パターンと入力素性構造18とのパターンマッチングが行なわれる。global tag“@tag”には、素性構造パターンと構造上対応する入力素性構造の部分構造が代入される。規則21の適用の結果、素性構造19が得られる。また、規則22を素性構造18に適用すると、変数?varにはglobal tag @tagと同じ入力素性構造の部分素性構造が代入される。したがって、規則21の書き換えの効果(処理内容)は規則22と等価であるが、規則21の方が規則22より制約が強い。

7.6 type

---

type of *FeatureStructure*

演算子type ofは素性構造*FeatureStructure*のタイプを返す。

7.7 素性パス

FeatureStructure . *feature*
path *feature* of *FeatureStructure*

演算子“.”(ドット)は素性構造*FeatureStructure*の*feature*素性をたどり、*feature*素性の値である部分素性構造を返す。素性パスは素性構造を対象とする演算子の引数に用いることができる。ただ、注意が必要なのは、素性構造の書き換えを行なう演算子=の引数に用いられた時にである。素性パスの記述が書き換え演算子=の左辺(書き換え対象)に現れた時と、右辺(素性構造の参照)に現れた時では、素性パスの扱いが異なる。素性構造の書き換え対象として素性パスが現れた時には、指定されたパスが素性構造*FeatureStructure*にないときには新たに素性が生成(追加)される。一方、素性構造の参照としてパス記述が用いられた時では、素性構造*fs*が*feature*素性を持っていなければ定数emptyが返る。

演算子“.”をつないで記述することにより、素性構造に対してネストして素性のパスをたどり、素性のパスの先の部分素性構造にアクセスできる。例えば、変数inputに素性構造20が代入されているとすると、パス記述

```
input.a.b
```

の値は素性構造cになる。また、パス記述

```
input.a.d
```

の値は定数emptyになる。書き換え対象としての生成パスの詳しい記述については8章を参照。

素性構造 20
```
[[a [[b c]]]]
```

7.8 文字列

" {'"'(ダブルクォート)以外の任意の文字列} "

文字列は、ダブルクォートで始まりダブルクォートで終わる。文字列は書き換え規則のドキュメンテーションに用いられる。

7.9 数

```
[0-9]*
```

0から9までの文字列の任意の並び。現在、書き換え規則では用いられていない。

7.10 定数

```
true
false
empty
```

定数true, falseは条件式の評価結果の値として用いられる。条件式の評価結果が真の時には、定数tureが返る。条件式の評価結果が偽の時には、定数falseが返る。定数emptyは素性構造の操作の結果として用いられる。書き換え呼び出しが失敗に終わったとき、あるいは、素性構造の素性のパスにおいて該当する素性がなかったとき、定数emptyが返る。

7.11 wildcard

書き換え規則中の素性構造パターンの中には表層に近い位置の素性と深い位置の素性構造を検査することがある。この場合、

1. 語義の存在有無を検査する。

2. 語義および素性パスを含めた構造を検査する。

の2通りが考えられる。特に1の場合で素性構造パターン内の複数の部分パターンのみを検査すればよい場合では、書き換え規則の記述を簡略化する機能が有効になる。そこで、素性パスおよび素性構造パターンに任意のパターンにマッチするwildcardの記述機能を提供する。

[FeatureName.arg ConstantFeatureValue]

規則 23

```
on <obje obje restr #1 reln> ばよい-SHOULD2
                        in :PHASE :ELLIPSIS-RESOLUTION
   S̈-REQUEST + INFORMREF + ばよい-SHOULD --> 二人称主語¨
   in= [[reln S-REQUEST]
        [obje [[reln INFORMREF]
               [obje [[restr [[#1 [[reln ばよい-SHOULD2]
                                   ?rest]]
                              ?rest1]]
                     ?rest2]]
              ?rest3]]
        ?rest4]
   if input.obje.obje.restr.obje.agen =? []
   then
     input.obje.obje.restr.obje.agen = root.prag.speaker
   endif
   RETURN INPUT
end
```

7.12 素性構造の終端要素

[FeatureStructureName Symbol.end]

素性のパスで指定した箇所が終端であるかないかを判定するような条件を記述可能にする。

規則 24

```
on <obje obje restr reln> そちら-2 in :PHASE :J-E :TYPE :IDIOM
   in= [[RELN  QUESTIONIF]
        [OBJE [[RELN  だ-IDENTICAL]
               [OBJE [[PARM !x[]]
                      [RESTR [[RELN  そちら-2]
                              [ENTITY !x]]]]]
               [IDEN [[PARM !y[]]
                      [RESTR [[RELN  ?name]
                              [ENTITY !y]
                              [IDEN  ?iden.end]]]]]
                ?rest1]]
         ?rest2]
  out= [[RELN  QUESTIONIF]
        [OBJE [[RELN  だ-IDENTICAL]
               [OBJE [[PARM !x[]]
                      [RESTR [[RELN  this-PRON-2]
                              [ENTITY !x]]]]]
               [IDEN [[PARM !y[]]
                      [RESTR [[RELN  ?name]
                              [ENTITY !y]
                              [IDEN  ?iden]]]]]
                ?rest1]]
         ?rest2]
end
```

7.13 並列パス修飾子

```
set variable to FeaturePath::Path==RelnValue
```

変数に値を代入する場合、指定した素性パスに対応する素性構造を取り出すとともにその素性構造にある素性名relnの値をチェックする。

規則 25

```
on <reln> 参加する-2  in :PHASE :J-E :TYPE :default
   in= [[reln 参加する-2]
        [agen ?AGEN]
        [LOCT ?LOCT]
        ?rest]

 set ?xxx to input.loct.parm::restr==会議-1

   out= [[reln ?xxx]
         [agen ?agen]
         [LOCT ?LOCT]
         ?rest]
end
```

8 演算子

8.1 基本演算子

FeatureStructure1 = *FeatureStructure2*
add *FeatureValueList* to *FeatureStructure*
delete *feature* from *FeatureStructure*
type of *FeatureStructure*
set *variable* to *FeatureStructure*

素性構造の操作を行なう演算子は、引数である*FeatureStructure*に、素性パスによる部分素性構造の参照記述や変数など素性構造を返す記述を取ることができる。

演算子=は素性構造*FeatureStructure1*を素性構造*FeatureStructure2*で書き換える。*FeatureStructure1*, *FeatureStructure2*は"."(ドット)によるパスの記述でも良い。*FeatureStructure1*が素性パスによる部分素性構造の参照のときには、素性パスで指定された一連の素性が*FeatureStructure1*に存在すれば、素性のパスの先の部分素性構造が*FeatureStructure2*で書き換えられる。素性パスで指定された素性が*FeatureStructure1*に存在しなければ、新たに素性が*FeatureStructure1*に追加され、その素性の値は*FeatureStructure2*となる。演算子=による書き換え結果は*FeatureStructure1*に保持される。演算子=の右辺にパス記述などによる素性構造の参照を行なった時、その値がemptyの時にはエラーを起こす。

演算子addは、素性構造*FeatureStructure*と、素性と素性値の対のリスト*FeatureValueList*をとり、*FeatureStructure*に*FeatureValueList*を追加する。*FeatureStructure*と*FeatureValueList*とで素性のコンフリクトが生じた時には、*FeatureValueList*の素性が優先される。

演算子deleteは、*FeatureStructure*から素性*Feature*を削除する。

演算子type ofは*FeatureStructure*のタイプを返す。

演算子setは変数*variable*に*FeatureStructure*を代入する。

```
input.a.b.c = input.e.f.g
```

素性構造21
```
[[e [[f [[g h]]]]]]
```

素性構造22
```
[[a [[b [[c !X h]]]]]
 [e [[f [[g !X]]]]]]
```

上記の=演算子による素性構造の書き換え処理では、変数inputに素性のパスa.b.cがない場合新たに素性が生成される。例えば変数inputの値が素性構造21であるとき、上記の書き換え処理を行なうと変数inputの値は素性構造22になる。

規則 26

```
on <reln> 行う-1 in :phase :japanese
   "「行う」のobject格要素を動詞に派生させる"
   in= [[reln 行う-1]
        [obje ?obje]
        ?rest]
   -> ?obje with :phase :japanese :event-or-object :event
   add ?rest to ?obje
   out= ?obje
end
```

規則 27

```
on <reln> UNKNOWN-IFT in :IF :REDUCE :TYPE :Default
   in= [[reln UNKNOWN-IFT]
        [agen ?agen]
        [recp ?recp]
        [obje ?obje]]
   set ?output to [[reln inform]
                   [agen ?agen]
                   [recp ?recp]
                   [obje ?obje]]
   if ?obje.infmann then
        ?output.infmann = ?obje.infmann
        delete infmann from ?obje
   endif
   if ?obje.conect then
        ?output.conect = ?obje.conect
        delete conect from ?obje
   endif
   set parameter :IFT :INFORM
   out= ?output
end
```

規則26では、書き換え呼び出しの結果の素性構造に対して、任意格のセット(fvlist)を追加している。規則27では、=演算子を用いた素性の追加とdelete演算子を用いた素性の削除によって、infmanおよびconect素性を、素性構造?objectから素性構造?outputに移動している。

8.2 条件式

FeatureStructure is *FeatureStructure*
FeatureStructure is not *FeatureStructure*
FeatureStructure =? *FeatureStructure*
FeatureStructure =! *FeatureStructure*
FeatureStructure has *feature*
FeatureStructure has not *feature*
predicate and *predicate*
predicate or *predicate*
not *predicate*

---

isは、二つの素性構造が構造的に等しいか検査する。等しければ真を意味する定数trueを、等しくなければ偽を意味する定数falseを返す。is notは、isの否定であり、二つの素性構造が構造的に等しいかパターンマッチングをおこなって検査し、等しければ定数falseを、等しくなければ定数trueを返す。

=?はisの別名であり、isと同様である。=!はis notの別名である、is notと同様である。

hasは素性構造*FeatureStructure*が素性*feature*を持っているかを検査する。*FeatureStructure*が*feature*素性を持っていれば、定数trueを返す。持っていなければ、定数falseを返す。has notは、hasの否定であり、素性構造*FeatureStructure*が素性*feature*を持っていなければ定数trueを返し、持っていれば定数falseを返す。演算子hasによる条件記述は“.”(ドット)によるパス記述と等価である。

条件式には素性構造、変数、素性パスを伴った素性構造の記述も条件式として記述することができる(付録A参照)。

変数、素性パスも単体で条件式として記述できる。変数は値が代入されていれば真である。素性パスを伴った素性構造の記述では、素性のパスが素性構造に存在すれば真になる。

条件式が一つの文として現れた時には、[11]条件式の評価結果(定数trueあるいはfalse)はシステム変数?itに代入される[12]。論理演算子not、and、orを用いて条件式を組み合わせることにより、より複雑な条件式を表現できる。andとorは右から左に条件式を評価してゆく。条件式を“(”と“)”で囲むことにより、評価の順序を変更できる。

[11]一つの独立した文として記述された時。付録Aのシンタックスの記述中の*stmt*に対応。

[12]if文などの条件式に現れた時には変数?itには、代入されない。

9 書き換え規則内での制御機構

9.1 入力素性構造の検査

---

in= *FeatueStructure*

---

in=は入力素性構造が素性構造パターン*FeatueStructure*と構造的に等しいかパターンマッチングを行なって検査する。構造的に等しければ書き換え規則中の次の式が引続き処理される。構造的に等しくなければ規則の適用はin=*FeatureStructure*を評価した時点で失敗に終る。

in= *FeatueStructure*は、次のif文と等価である。

```
if input =!  FeatureStructure
then
   fail
endif
```

9.2 出力素性構造の生成と規則の終了

---

out= *FeatueStructure*
return *FeatureStructure*

---

out= は入力素性構造を素性構造*FeatueStructure*で置き換え(書き換え)、*FeatureStructure*を適用結果として返す。returnは素性構造*FeatureStructure*を規則適用の結果として返す。out=あるいはreturnが評価されると、規則を強制的に終了する。書き換え規則の適用において、out=あるいはreturnを用いて書き換え規則が明示的に素性構造を返すようにしないと、書き換え規則内の処理は無効になる。

素性構造パターン*FeatureStructure*に含まれる変数は評価され、変数は変数に代入されている値で置き換えられる。素性構造パターンをもちいた素性構造の生成では、値を持たない変数が素性構造パターンに含まれていると、エラーを起こし規則適用はその時点で失敗に終わる。

9.3 書き換え呼び出し

-> *FeatureStructure* [with *AttributeValueList*]
--> *FeatureStructure* [with *AttibuteValueList*]
=> *FeatureStructure* [with *AttributeValueList*]
==> *FeatureStructure* [with *AttributeValueList*]

これら四つの書き換え呼び出し演算子は、素性構造*FeatureStructure*に対して書き換え呼び出しを行なう。with *AttibuteValueList*は書き換え環境の局所的状態設定である。書き換え環境の局所的状態設定は、オプショナルな記述である。状態設定が指定されると書き換え呼び出しを行なう前に、書き換え環境の状態が局所的に*AttibuteValueList*に変更される。この記述が省略された場合は、規則適用時の書き換え環境の状態で、書き換え呼び出しが行なわれる。

書き換え呼び出し演算子は、書き換え対象の素性構造の辿り方と、書き換え呼び出しの終了状態による処理の制御方法の違いにより、四つ用意されている。

--> と ==> は素性構造の素性構造の書き換え呼び出しが失敗に終った時(適用可能な規則がない、あるいは、すべての規則の適用が失敗した時)、規則中の続く式を実行せず規則を強制的に終了する。この時、書き換え呼び出しを行なった規則も失敗となる。

-> と => は、書換えか呼び出しが失敗に終っても規則の強制終了せず、引続き処理が続行される。

-> と --> は、書き換え呼び出しにおいて、素性構造*FeatureStructure*のトップレベルだけを対象に(再帰的に構造を辿らないで)規則の検索、適用をおこなう。

=> と ==> は、書き換え呼び出しにおいて、*FeatureStructure*を再帰的に構造を辿り、*FeatureStructure*を構成するcomplexタイプの部分素性構造に対し規則の検索、適用をおこなう。*FeatureStructure*のすべての部分構造に対し規則の検索、適用が終了したら、その書き換え結果を書き換え呼び出しの結果として返す。

書き換え呼び出しが終了すると、書き換え結果がシステム変数?itに代入される。書き換え呼び出しが成功した場合には、変数?itには、書き換え結果の素性構造が代入され、失敗した場合には定数emptyが代入される。

書き換え結果が複数返された時は、規則内で書き換え呼び出しに続く処理が、各々の書き換え結果に対して並列に実行される。

素性構造 23
```
[[x y]
 [z [[a b]]]]
```

規則 28
```
on <x> y
   in=  [[x y]
         [z ?z]]
   --> ?z
   out= ?z
end
```

規則 29
```
on <a> b
  in=  [[a b]]
  out= [[a b1]]
end
```

規則 30
```
on <a> b
  in=  [[a b]]
  out= [[a b2]]
end
```

規則 31
```
on <a> b
  in=  [[a b]]
  out= [[a b3]]
end
```

上記の四つの規則があった時に、素性構造23に規則28を適用したとする。規則28では、まず入力素性構造のパタンマッチングによる検査がおこなわれる。次に、変数?zに代入された部分素性構造に対し書き換え呼び出しが行われると、規則29、規則30、規則31の三つの書き換え規則が部分素性構造に適用される。その結果、三つの素性構造が書き換え呼び出しの結果として返される。この時、各々の書き換え結果の数だけサブプロセスが生成され、次に続く処理が並列に実行される[13]。書き換え結果に伴って生成された各サブプロセスでは、それぞれ異なった書き換え結果が与えられ、各サブプロセスで規則記述中の次の処理(ここではout= ?z)を並列に実行する(図 3参照)。

以下に再帰的な書き換え呼び出し(=>,==>)と、素性構造を再帰的に辿らないフラットな書き換え呼び出しに関して、規則適用のトレース結果を示す。

まず、再帰的な書き換え呼び出しを行なっている規則を適用した時のトレース結果を示す。規則J-1420は演算子==>を用いて部分素性構造に再帰的書き換え呼び出しを行なっている。その際、書き換え環境の局所的な変更を行なっているので、規則F-1422と規則H-1421が適用可能になる。

[13]サブプロセスの生成は見かけ上はUNIXのforkに近いイメージである。またここで言うサブプロセスとはUNIXのプロセスではなく処理の単位を言う。実際には、規則本体中の書き換え呼び出しに続く処理をシーケンシャルに実行する。

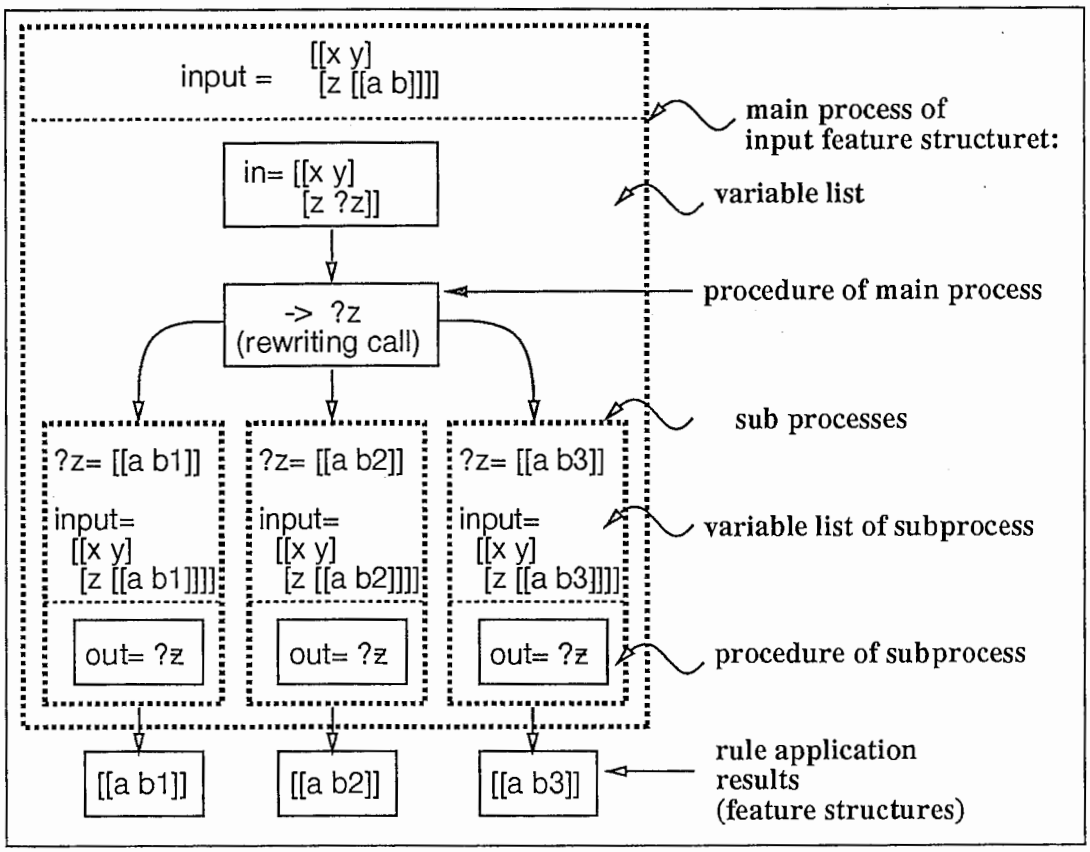

図 3: 書き換え呼び出しとサブプロセスの生成

```
> (rws::pprint-fs fs)

 [[A [[B [[C D]
          [E F]]]
      [G H]]]
  [I J]]
NIL
> (rws::print-all-rw-rules :print-text t )

J:
----------
J-1420
Parameter                    Value
---------                    -----
:PHASE                       :J-E
:TYPE                        :GENERAL

on <i> j
   in= [[i j]
        [a ?var]]
   ==> ?var with :X :Y .....................部分素性構造の再帰的な書き換え呼び出し
   out= ?it
end

F:
----------
F-1422
Parameter                    Value
---------                    -----
:X                           :Y

on <e> f in :x :y
  in= [[e f]
       [c d]]
  out= [[e f+]
        [c d+]]
end

H:
----------
H-1421
Parameter                    Value
---------                    -----
:X                           :Y

on <g> h in :x :y
```

```
   in= [[g  h]
          [b ?b]]
   out= [[g h+]
           [b ?b]]
end

NIL
> (rws:debug-transfer)
Print Input Feature Structure [RWS::YES](Y or N): y
Print Rewritten Results [RWS::YES](Y or N): y
Trace Applying Rules to FS and Result FSs [RWS::YES](Yes, No or Verbose) v
Trace Rewriting Call [RWS::YES](Yes, No or Verbose) v
Trace Pattern Matching[RWS::NO](Y or N): n
Trace Parameter Setting [RWS::NO](Yes, No or Verbose) n
Do you set parm-env for Rewrite FS (Y or N): n
Do You Execute Preference Check ? (Y or N): n
(T T RWS::VERBOSE RWS::VERBOSE NIL NIL)
> (rws::transfer fs) ........................................素性構造の書き換えを実行
;;; ------------ Transfer Input ------------
 [[A [[B [[C D]
          [E F]]]
      [G H]]]
  [I J]]
;;; | 1[1]:> J-1420 ....................................................規則 J-420の適用
;;; Apply rewriting rule ID:J-1420.
;;; Input Feature structure is following.
 [[A [[B [[C D]
          [E F]]]
      [G H]]]
  [I J]]
;;;   1 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> .....書き換え呼び出し ==> ?z ...の実行
;;; CALLED recursive subFS rewriting call on J.
;;; SubFS is following. ...................................................変数 ?zの値
 [[B [[C D]
      [E F]]]
  [G H]]
;;; |   | 2[1]:> H-1421 ................................................規則 H-1421の適用
;;; Apply rewriting rule ID:H-1421.
;;; Input Feature structure is following.
 [[B [[C D]
      [E F]]]
  [G H]]
;;; |   | 2[1]:< H-1421 Success ...................................規則 H-1421の適用終了
;;; Success to apply rewriting rule ID:H-1421
;;; There is one result.
;;; The first rewrited Feature Structure is following ............規則 H-1421の適用結果
```

```
 [[B [[C D]
      [E F]]]
  [G H+]]
;;; |   | 2[1]:> F-1422 ..............................................規則 F-1422の適用
;;; Apply rewriting rule ID:F-1422.
;;; Input Feature structure is following. ...............再帰的に素性構造をたどっている
 [[C D]
  [E F]]
;;; |   | 2[1]:< F-1422 Success .....................................規則 F-1422の適用終了
;;; Success to apply rewriting rule ID:F-1422
;;; There is one result.
;;; The first rewrited Feature Structure is following ............規則 F-1422の適用結果
 [[C D+]
  [E F+]]
;;;  1 <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< .................書き換え呼び出しの終了
;;; Terminate subFS Recusive Rewriting CALL on J.
;;; After Recursive Rewriting Call, there is one result
;;; The first result is following ...............................書き換え呼び出しの結果
 [[B [[C D+]
      [E F+]]]
  [G H+]]
;;; | 1[1]:< J-1420 Success ........................................規則 J-1420の適用終了
;;; Success to apply rewriting rule ID:J-1420
;;; There is one result.
;;; The first rewrited Feature Structure is following ............規則 J-1420の適用結果
 [[B [[C D+]
      [E F+]]]
  [G H+]]

;;;=========== Transfer Result ============
 [[B [[C D+]
      [E F+]]]
  [G H+]]
;;; ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

(#<Structure RWS::NODE C66F26>)
>  (rws:debug-transfer)
```

次のトレースは素性構造のフラットな書き換え呼び出しの例である。素性構造は変数fsに代入されている。三つの書き換え規則が定義されており、素性iに定義されている規則は書き換え環境の変更をともなった書き換え呼び出しを行なう。ここではフラットな書き換え呼び出しが行なわれているので、前述の例とは異なり規則F-1422の適用は行なわれないことに注意して欲しい。

```
> (rws:pprint-fs fs)
```

```
 [[A [[B [[C D]
          [E F]]]
      [G H]]]
  [I J]]
NIL
> (rws::print-all-rw-rules :print-text t)

J:
----------
J-1438
Parameter                      Value
---------                      -----
:PHASE                         :J-E
:TYPE                          :GENERAL

on <i> j
   in= [[i j]
        [a ?var]]
   --> ?var with :x :y .....................部分素性構造のフラットな書き換え呼び出し
   out= ?it
end
F:
----------
F-1422
Parameter                      Value
---------                      -----
:X                             :Y

on <e> f in :x :y
  in= [[e f]
       [c d]]
  out= [[e f+]
        [c d+]]
end

H:
----------
H-1421
Parameter                      Value
---------                      -----
:X                             :Y

on <g> h in :x :y
   in= [[g  h]
        [b ?b]]
```

```
   out= [[g h+]
         [b ?b]]
end

NIL
> (rws::transfer fs)
;;; ------------ Transfer Input ------------
 [[A [[B [[C D]
          [E F]]]
      [G H]]]
  [I J]]


Parameter                     Value
---------                     -----
:PHASE                        :J-E
:TYPE                         :GENERAL
;;; | 1[1]:> J-1438
;;; Apply rewriting rule ID:J-1438. ..................................書き換え規則の適用
;;; Input Feature structure is following.
 [[A [[B [[C D]
          [E F]]]
      [G H]]]
  [I J]]
;;;  1 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>  ............書き換え呼び出し -->の実行
;;; CALLED subFS Rewriting call on J.
;;; SubFS is following.
;;;
 [[B [[C D]
      [E F]]]
  [G H]]
;; Set local parameter. ....................................書き換え環境の局所的状態設定

Parameter                     Value
---------                     -----
:X                            :Y
;;; |   | 2[1]:> H-1421
;;; Apply rewriting rule ID:H-1421. ..........................書き換え規則 H-1421の適用
;;; Input Feature structure is following.
 [[B [[C D]
      [E F]]]
  [G H]]
;;; |   | 2[1]:< H-1421 Success ..........................................規則の適用は成功
;;; Success to apply rewriting rule ID:H-1421
;;; There is one result.
;;; The first rewrited Feature Structure is following .....................書き換え結果
```

```
 [[B [[C D]
      [E F]]]
  [G H+]]
;;;  1 <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< .................書き換え呼び出しの終了
;;; Terminate subFS Rewriting CALL on J.
;;; After Rewriting Call, there is one result
;;; The first result is following ...............................書き換え呼び出しの結果
 [[B [[C D]
      [E F]]]
  [G H+]]
;; Reset local parameter. ........................................書き換え環境のリセット

Parameter                     Value
---------                     -----
:PHASE                        :J-E
:TYPE                         :GENERAL
;;; | 1[1]:< J-1438 Success .........................................書き換え規則の終了
;;; Success to apply rewriting rule ID:J-1438
;;; There is one result.
;;; The first rewrited Feature Structure is following ......................書き換え結果
 [[B [[C D]
      [E F]]]
  [G H+]]

;;;=========== Transfer Result ============
 [[B [[C D]
      [E F]]]
  [G H+]]
;;; ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

(#<Structure RWS::NODE C6CF56>)
>
```

9.4 if文

---

if *test* then *statement1* endif
if *test* then *statement1* else *statement2* endif

---

if文は多くのプログラミング言語で見られるような、if-then-else構文要素に対応している。最初に条件式*test*が評価される。もし、その結果(値)が定数falseでなければ、式*statement1*が実行される。そうでなければ式*statement2*が実行される。elseがない場合、*statement2*はもちろん実行されない。

以下にif文を用いた簡単な例を示す。この例では、入力素性構造がconncet素性を持っていれば、その素性を書き換えのルートの素性構造に移動している。例えば、部分素性構造が持っているconncet素性をルートの素性構造に移動するような処理に使われる。

```
if input has conncet then
   root.conncet = input.connect
   delete conncet from input
endif
```

9.5 switch文

---

switch *key*
    case *FetureStructure1 statement1*
    ...
    default *DefaultStatement*
endswitch

---

switch文は、素性構造*key*とcaseの項目に挙がっている素性構造*FeatureStructure*をパターンマッチングによって構造的に比較することによって実行すべき一つの節を選択するような条件つき実行である。素性構造*key*と選択子であるcaseに続く素性構造*FeatureStructure*との構造的な比較は先頭から並びの順に行なわれる。素性構造の比較はパターンマッチングによって行なわれる。すべての

case項目の比較に失敗した場合defaultに続く*DefaultStatement*が実行される。switch文で*statement*を省略することはできない。

以下にswitch文を用いた簡単な例を示す。この例では、入力素性構造のreln素性によって処理の流れ(フェイズの設定)を変更している。

```
switch input.reln
    case $phatic
         rewrite input with :phase :J-E :type :idiom
 return input
    case $response
         rewrite input with :phase :J-E :type :idiom
 return input
    case $expressive
         rewrite input with :phase :Japanese :type :idiom
         rewrite input with :phase :Japanese :type :general
         rewrite input with :phase :J-E :type :idiom
         rewrite input with :phase :J-E :type :general
         rewrite input with :phase :J-E :type :default
         rewrite input with :phase :Aspect :type :default
         return input
    default
endswitch
```

9.6 fail

---

fail

---

failが実行されると書き換え規則は直ちに失敗して終了する。failが実行されると、その書き換え規則内でfail以前で実行された書き換え処理等はすべて無効になる(もとの状態に戻される)。

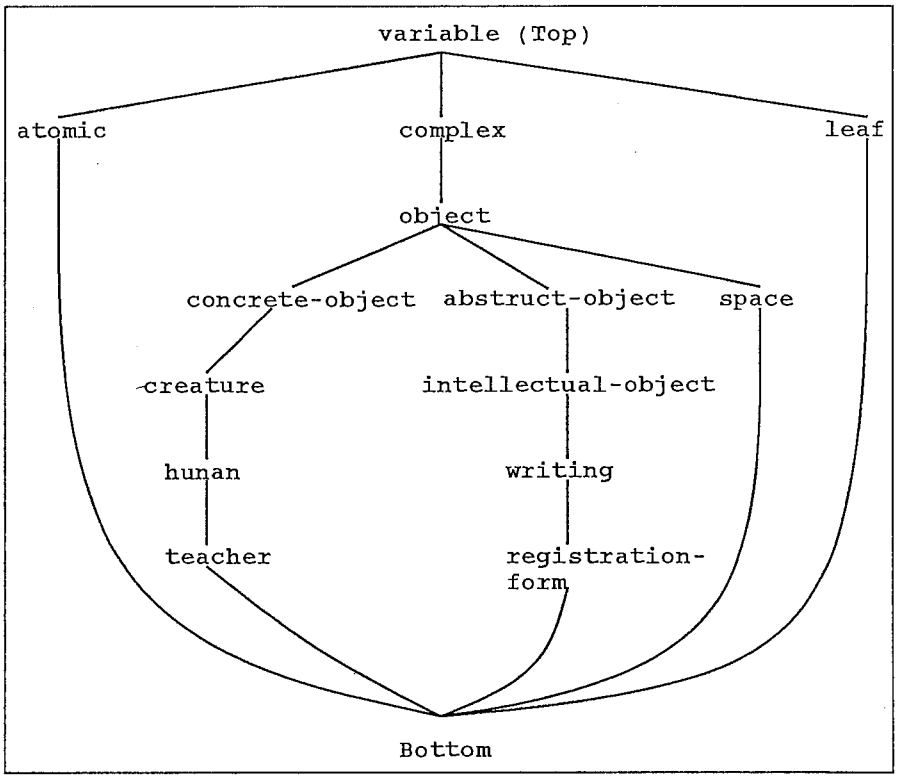

図 4: シソーラス体系

10 タイプシステム

10.1 タイプシステム

素性構造書き換えシステムでは、deffstypeによってタイプシステム中のタイプシンボル間の階層関係を定義する。[14]

---

deffstype *type subtypes*

---

deffstypeはタイプシンボル*type*を新たに生成し、*type*とタイプ*subtypes*の階層関係を定義する。タイプ*type*は*subtypes*の上位タイプ($type \geq subtype$)として定義される。サブタイプが明示的に定義されていないタイプのサブタイプは、bottom($\perp$)になる。*subtype*は複数のタイプシンボルが記述できる。上位タイ

[14]タイプシンボルの定義はLISP関数によっておこなわれる。他のタイプシステムに関する操作は本書PART IIのユーザーズマニュアルを参照。

プが明示的に定義されていないタイプの上位タイプはtop(⊤)になる。

図4に図示されるタイプシステムは、deffstypeを用いて以下のように定義される。[15]

```
(deffstype complex things sentences affairs)

(deffstype things object)

(deffstype object concrete-object abstruct-object space)

(deffstype conctete-object creature)

(deffstype creature human)

(deffstype human teacher)

(deffstype abstruct-object intellectual-object)

(deffstype intellectual-object writing)

(deffstype writing regstration-form)
```

10.2 タイプ付き素性構造の記述方法

素性構造のタイプは、素性構造パタンの直前にタイプシンボルを記述することによって宣言される。タイプシンボルの記述は接頭辞":"で始まる。complex, atomic, leafの四つのタイプシンボルは素性構造の構造に依存して書き換えシステムが素性構造に自動的に与える(陽にタイプが与えられていない素性構造に対しては、complex, atomic, leafのいずれかのタイプのみが与えられる)。素性構造24では、:human, :writingがタイプシンボルの記述であり、「先生」、「登録用紙」に対応する部分素性構造がそれぞれタイプシステム(知識ベース, シソーラス体系)中の:human, :writingのエントリに位置付けられていることを表わしている。

[15]このタイプシステムの記述ではcomplexタイプのサブタイプとして概念体系（シソーラス体系）を表現している。これにより、素性構造24では、complexタイプの素性構造には、さらに意味分類による詳細なタイプが付与されている。しかし、意味分類に関するタイプを、どのサブタイプとして位置付けるか任意であり、例えば、atomicタイプのサブタイプとしてもよい。但し、本システムのデフォルトで用意されているタイプはcomplexタイプの下位タイプとして位置付けている。

素性構造 24

```
[[reln 送る]
 [agen :human[[label *speaker*]]]
 [recp :human[[parm !x[]]
              [restr [[reln 先生]
                      [entity !x]]]]]
 [obje :writing[[parm !y[]]
                [restr [[reln 登録用紙]
                        [entity !y[]]]]]]]
```

atomicタイプ素性構造および変数に対してより詳細なタイプを与える時も、complexタイプと同様に、タイプ名を素性構造パターンの直前に記述する。タイプによる制限が与えられた変数は、変数に与えられたタイプのサブタイプを持つ素性構造に制限される。

素性構造 25

```
:writing $登録用紙
```

素性構造 26

```
:writing ?object
```

atomicタイプが単独で(complexタイプの部分素性構造ではなく)記述される時にも、素性構造25のようにタイプ名をatomicタイプ素性構造の直前に記述する。

素性構造26は、変数?objectにパターンマッチングにおいて一致できる素性構造のタイプをwritingに制限している。タイプによる制限付の変数には、変数名とタイプシンボルの記述を合わせた略記法がある。

素性構造 27

```
(1) ?:writing
(2) :writing ?writing
```

素性構造27では、変数名がwritingであり、この変数に一致できる素性構造はタイプwritingのサブタイプに制限される。(2)の意味は(1)と完全に同一である。

変数にタイプの制限を与える時には、選言が記述できる。タイプの選言には“|”を用いて記述する。

素性構造 28

```
:%(human|animal) ?variable
```

素性構造パターン28は、タイプにより制限つき素性構造であるが、変数?variableに一致できる素性構造は、humanタイプあるいはanimalタイプのサブタイプの素性構造に制限される。

10.3 タイプつき素性構造のパターンマッチング

タイプ付き素性構造パターンと素性構造のマッチングでは、まず、素性構造パターンに記述されているタイプ値が定義されているかのチェックを行う。次に、素性構造パターンと対象となる素性構造のチェックが行なわれる。タイプ値のチェックでは、素性構造パターンと入力素性構造のタイプ値を比較する。二つのタイプ値が同一の場合はタイプ制約を満たしているとする。また、素性構造中のタイプ値が書き換え規則の素性構造パターンに記述されているタイプ値に対して下位に定義されている場合も上位概念を継承していると考えられるので、同じくタイプ制約を満たしているものとする。パターンマッチングにおいて素性構造のタイプ制約が満たされると、さらに、二つの素性構造が構造的に等しいか否かが検査される。

規則 32

```
on <reln> 送る
   in=  [[reln 送る]
         [agen ?:human]
         [recp ?recipient]
         [obje ?:writing]
         ?rest]
   out= [[reln send]
         [agen ?human]
         [recp ?recipient]
         [obje ?writing]
         ?rest]
end
```

書き換え規則32の?:human, ?:writingはタイプ制限付き変数であり、変数?:human, ?:writingに一致できる素性構造は、それぞれhumanタイプ, writingタイプの素性構造かそのサブタイプの素性構造である。規則32の入力素性構造パターンは素性構造24と一致し、素性構造29に書き換える。

素性構造 29

```
[[reln send]
 [agen :human[[label *speaker*]]]
 [recp :human[[parm !x[]]
              [restr [[reln 先生]
                      [entity !x]]]]]
 [obje :writing[[parm !y[]]
                [restr [[reln 登録用紙]
                        [entity !y[]]]]]]]
```

11 LISP式

% 任意のLISP式の記述

LISP式は書き換え規則中の定義(on <*FeaturePath*> *fs*)の後の任意の場所に記述することができる。LISP式はプレフィックス%で始まり、改行で終る。%は任意の場所に記述可能であり、改行までがLISP式としてみなされる。改行がLISP式の終りであるので、規則記述に改行が記述できない書き換え規則定義関数defrwruleでは、LISP式を記述してはならない。%で始まる一つのLISP式は必ずしも完全なS式でなくてもよく、規則の記述中でS式として完結していればよい。

以下にLISP式を含んだ書き換え規則を例示する。

規則 33

```
on <reln> copula in :PHASE :English
   in= [[reln copula]
[obje ?obje]
[iden ?iden]
?rest]

% (rws::fetch-global-fs ?previous rws::*previous-transout*)
   if ?previous.prag.context.focus.restr.reln is ?obje.restr.reln
     then set ?pred to ?previous.prag.context.main-pred
     else set ?pred to be-VI-5
   endif

   out= [[reln ?pred]
 [obje ?obje]
 [iden ?iden]
?rest]
 end
```

規則33では、外部変数rws::*previous-transout* の内容を変数?previous にセットする。そして、その変数の素性パスで示した終端の素性：relnの素性値と素性構造パターンの変数?objectにセットされている素性パスの終端の素性：relnの素性値を検査し、その結果により書き換える素性構造のrelnの素性値を選択している。

規則 34

```
on <a> :unspecified
  in= [[a ?x]]
%  (format t "variable ?X = ")
%  (rws::pprint-fs
            ?X
%             t :terpri nil)
  out= [[a ?x]
        [c d]]
end
```

また、規則 34では、変数?Xを規則中で評価した結果(素性構造)を、素性構造を表示する関数pprint-fsの引数として与えている。書き換え規則の実行時に変数?Xの値を評価したいので、LISP 関数pprint-fsの引数?XをLISP式の中に入てはならない。規則??が適用されると、入力素性構造と素性構造パターンのパターンマッチングが行なわれ、パターンマッチングが成功すると変数?Xに代入される。つぎに、LISP式によって変数?Xの値である素性構造が画面に表示される。以下に規則34の実行したトレース結果を示す。

```
> (rws::print-rw-rule :unspecified nil :print-text t)        ;;規則の表示
:UNSPECIFIED:
----------
UNSPECIFIED-9
Parameter                      Value
---------                      -----
:PHASE                         :J-E
:TYPE                          :GENERAL

on <a> :unspecified
  in= [[a ?x]]
%  (format t "variable ?X = ")
%  (rws::pprint-fs
            ?X
%             t :terpri nil)
  out= [[a ?x]
        [c d]]
end
T

> (setq fs (rws::read-fs [[a b]])) .....................入力素性構造の生成
#<Structure RWS::NODE CB1D66>

> (rws::transfer fs) ..........................................規則の適用
;;; ------------ Transfer Input ------------
 [[A B]]
variable ?X =  B .....................LISP式によって出力されたメッセージ
```

```
;;;=========== Transfer Result ============
 [[A B]
  [C D]]
;;; ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
(#<Structure RWS::NODE CBEA4E>)
```

12 ドキュメンテーション

書き換え規則のドキュメントを規則の定義部(on <*feature-path¿*> atomc-fs)の後に文字列(string)で記述することができる。書き換え規則のドキュメンテーションはオプションである。

13 コメント

;

書き換え規則中で";"に続く任意の改行までの文字列はコメントとみなされ無視される。

A シンタックス

rwrule : on *def* in *AttributeValueList documentation instrns* end
| on *def documentation instrns* end

def : < symbol* > symbol
| < symbol* > key

documentation :
| string

AttributeValueList : *AttributeValue*
| *AttributeValueList AttributeValue*

AttributeValue : key symbol
| key key
| key NUMBER

AttributeList : key
| *AttributeList* key

instrns : *RewritingCall*
| LISP-Expression
| *stmt*
| *RewritingCall instrns*
| LISP-Expression *instrns*
| *stmt instrns*

stmt : *control*
| *assign*
| *conds*

rewriting : -> *fs* with *AttributeValueList*
| -> *fs*
| => *fs* with *AttributeValueList*
| => *fs*
| --> *fs* with *AttributeValueList*
| --> *fs*
| ==> *fs* with *AttributeValueList*
| ==> *fs*
| rewrite *fs* with *AttributeValueList* by *AttributeList*

control : fail
| in= FS
| out= *fs*
| return *fs*
| if *conds* then *instrns* endif
| if *conds* then *instrns* else *instrns* endif
| switch *object casebody* endswitch

assign : set parameter *AttributeValueList*
| unset parameter *AttributeList*
| set *variable* to *object*
| add *fvlist* to *fs*
| *fs* == *fs*
| *fs* = *fs*
| delete *path* from *fs*

conds : *condition*
| *(conds)*
| *conds* and *conds*
| *conds* or *conds*
| not *conds*

condition : *fs* `is` *fs*
| *fs* `is true`
| *fs* `is false`
| *fs* `is not` *fs*
| *fs* `is not true`
| *fs* `is not false`
| *fs* `=?` *fs*
| *fs* `=?` `true`
| *fs* `=?` `false`
| *fs* `=!` *fs*
| *fs* `=!` `true`
| *fs* `=!` `false`
| *fs* `has` symbol
| *fs* `has not` symbol
| *fs*

casebody : *case instrns*
| *casebody case instrns*

case : `case` *object*
| `default`
| *case* `case` *object*

object : symbol
| `number`
| *constant*
| *fs*
| `type of` *fs*

fs : `fs`
| *variable*
| `fs` . *path*
| *variable* . *path*

path : symbol
| *path* . symbol

constant : `empty`
| `false`

variable : ?symbol
| `?it`
| `input`
| `root`

LISP-Expression : [A-Za-z0-9]+

symbol : [A-Za-z0-9]+

string : "'"'を除く任意の文字列"

FS : *tag* 素性構造の記述

tag : !symbol
| @symbol

さくいん

PART II

素性構造書き換えシステム
ユーザ利用マニュアル
Feature Structure Rewriting System V2
User Reference Manual

要旨

このPART IIは、PART Iのシステムマニュアルには十分に記載されていない、書き換えシステムの実際の使用方法を、できるだけ分かり易く解説することを目標に作成された、利用者向けのマニュアルである。PART Iの冒頭にも述べた通り、素性構造書き換えシステムは種々の機能向上が実行され、改良版(V2)となっている。ただし、旧版で定義された関数をそのまま残してあるため新版で仕様を変えた関数名は最後に"2"が付されていることが多い。使用に際しては、この点に注意されるようお願いする。

ATR自動翻訳電話研究所

ATR Interpretin Telephony Reserach Laboratories

もくじ

1 はじめに

本マニュアルは、素性構造の書き換えシステムを利用して日英変換処理を行う方法および実験に必要な機能について述べたものである。日英変換処理は主にLISPの実行環境下で行われるが、書き換え規則のコンパイル処理にC言語で作成された別のモジュールも利用する。各章での説明は実際にユーザが入力する内容に準拠して示すが、一部のメッセージなどを省略する場合もある。

以下、簡単に各章の内容を説明する。2章では日英変換処理を行うための書き換えシステムの起動方法を説明する。また、複数のユーザがリソースを共有して利用するための環境もあわせて示す。3章では日英変換処理を実行するための方法を中心に説明する。4章では書き換え規則の管理モジュールの機能について説明する。6章では文脈情報を提供する談話構造解析モジュールとの結合について説明する。また、7章では実験で得られた素性構造結果をファイル単位で比較する機能について説明する。最後に8章では書き換え規則のコンパイラについての利用方法を説明する。特に3.2章以降の説明では、具体的な関数の使用方法を中心にして行い、本システムが提供する開発環境を含めた機能を有効に利用できるように配慮した。

なお、実際に言語変換処理用の規則を記述する際には、種々の技法に習熟する必要があるので、巻末の参考文献[4] [6]等を参考にされたい。

2 書き換えシステムの起動

素性構造書き換えシステムはＬＩＳＰの環境下で実行可能です。そのため、

1. ＬＩＳＰを立ち上げる。

2. 初期化に必要なファイルをロードする。

が必要です。以下に初期化を行うロードファイルの例を示し、それぞれの処理に対して簡単なコメントがつけてあります。
また、素性構造書き換えシステムはＲＷＳというパッケージの下で実行されるように関数などが定義されています。そのため、本システムの利用および一部の機能の利用においてもロードファイルの先頭に(in-package 'RWS)の実行が必要です。
以下の初期化のためのロードファイルの例では次のようなdirectory構成のファイル群を利用すると仮定しています。

```
transfer-----rws-v2----engine
           |          |
           |          |--compiler
           |          |
           |--rules   |--type
           |          |
           |          |--context
           |          |
           |--fs      |--engine-v2
```

```
(setf *rws-trans-directory* (real-module-name))
  → ユーザのカレントパスに対して変換実行モジュールのパスをセット
(setf *RWS-HOME-DIRECTORY*
  (concatenate 'string *rws-trans-directory* "rws-v2/"))
  → 変換実行処理系のパスをセット
(setf *RWS-RULE-DIRECTORY*
  (concatenate 'string *rws-trans-directory* "rules/mset-fix/"))
```

```
  → 変換書き換え規則のパスをセット
(setf *RWS-RULE-FILE*
  (concatenate 'string *rws-rule-directory* "mset.data"))
  → 二次記憶化規則本体データファイルのセット

;;;;  ここまでは大域変数のセット

(load (merge-pathnames "engine/load.lisp" *rws-home-directory*))
  → 変換処理系本体のロード
(when (yes-or-no-p "Do you load V2-engine ?")
★ (load (merge-pathnames "engine-v2/load.lisp" *rws-home-directory*)))
  → 変換処理系の拡張版を利用する場合にロードする
(load (merge-pathnames "compiler/load.lisp" *rws-home-directory*))
  → 書き換え規則のコンパイラモジュールのロード
(load (merge-pathnames "type/load.lisp" *rws-home-directory*))
  → タイプシステム利用の処理機能ロード
(load (merge-pathnames "context/context-load" *rws-home-directory*))
  → 前文参照などの機能ロード
(setf *RWS-RULE-LOAD-FLG* t)
(in-package 'user)
(when (yes-or-no-p "Do you load Input FS?")
  (setq dec (rws::push-fs-from-file1
      (merge-pathnames "fs/mset-demo.fix" rws::*rws-trans-directory*)))
  → 入力素性構造のロード
(setq rws::*extend-flg* nil)
(when (yes-or-no-p "Do you load Context-Rule ?")
  (setq rws::*extend-flg* t)
  (load (merge-pathnames
        "rules/context/load-grammar" *rws-trans-directory*))
  → 前文参照などの書き換え規則のロード
```

上記で示した実行でload.lispを利用している部分は該当のパスにあるファイルの内容を参照して下さい。特にコンパイラに関する初期化はCのライブラリを利用するので、注意が必要です。
★の部分は本解説書PART Iのシステムマニュアル7章にある

1. 素性構造の変数化
2. wildcard の記述
3. 素性構造の終端要素
4. 並列パス修飾子

の記述を使用する場合には必要となるファイル群です。

3 実行環境

ここでは、書き換えシステムを利用した日英変換処理の実行に関する機能について説明します。

3.1 書き換え規則のロード

書き換え規則のロードには以下の機能があります。

1. 初期化状態で、すべての書き換え規則をロードする。
2. 二次記憶ファイル用のインデックスをロードする。
3. 指定した規則名の書き換え規則のみをロードする。

1による方式は書き換え規則が定義されているファイルをＬＩＳＰのＬＯＡＤ関数を使って行います。

```
(load "ファイル名")
```

通常、書き換え規則は書き換え環境制約を基本単位にして、ファイルに定義されているので、上記のロードを必要なすべてのファイルについて行う必要があります。ただし、それらをまとめて以下のような定義を作成し、ロードすることも可能です。

```
(in-package :user)
(dolist (file '(
        "main.schema"
        "ift.euc"
        "misc.euc"
        "japanese.euc"
        "idiom.euc"
        "general.euc"
        "defnoun.euc"
        "defverb.euc"
        "aspect2.euc"
        "prag2.euc"
        "ellipses.euc"))
    (load (merge-pathnames file
        (concatenate 'string "/mset-fix/foo" *rws-rule-directory*)))
```

2はあらかじめ利用するすべての書き換え規則に対し二次記憶化のインデックスおよび本体ファイルが作成されている時、インデックスファイルのみをロードする方法です。
ロードの方法は1と同じで、指定するファイルはインデックスファイルです。

```
(load (merge-pathnames "/mset-fix/mset.index" *rws-rule-directory*))
```

本体の書き換え規則は規則の適用時にインデックスの情報に基づき本体ファイルからロードします。3はすでに1あるいは2の方法で書き換え規則がロードされている時、一部の書き換え規則のみを更新する場合に利用します。この方法は以下のように行います。

(rws::load-rule '規則名 規則登録ファイル名)

上記の実行では該当規則がある場合はその件数（更新された件数）が表示されます。

3.2 入力素性構造のロード

入力素性構造は解析処理で得られた素性構造（ファイル出力形式）をメモリ上にロードして、利用します。指定する入力ファイルの内部形式は以下の2種類に対応しています。

1. 入力対象のキーと素性構造のみが保存されている。
2. 上記に対象の日本語入力文が追加されている。

上記のそれぞれに対応する実行方法及びその結果を以下に示します。

---

push-fs-from-file *file* &optional *only-fs*
push-fs-from-file1 *file* &optional *only-fs*
read-fs-from-string *string*

---

push-fs-from-fileは*file*を読み、*file*の素性構造記述から素性構造を生成しリストにして返します。push-fs-from-fileおよびpush-fs-from-file1はあらかじめ入力対象となる素性構造ファイルから入力素性構造を読み込み、

1. 入力素性構造データのリスト (push-fs-from-file)
2. 日本語入力文と素性構造のドットリストのリスト (push-fs-from-file1)

の結果を作成します。push-fs-from-file1 の入力ファイルの仕様は次の通りです。

```
;>XXX          ← カウンターのチェック用(XXX は番号等)
日本語入力文    ← quotation 等をつけないテキスト
素性構造        ← quotation 等をつけない一個の素性構造
```

この組合せが一つのまとまりで、これを任意の回数繰り返した後、ファイルの最後に終了マークとして @eof を挿入します。

また、素性構造がストリングとして存在している場合は、以下のようにして素性構造を読み込むことができます。

```
> (setq fs
    (rws::read-fs-from-string
       "[[a [[b [[c d]] [e f]] [g h]] [i j]]]"))
```

3.3 出力ファイルの設定

日英変換処理を実行した結果、得られた素性構造は通常、標準出力に出力されます。しかし、後で述べる7章で利用したり、一括して対象文を実行するために、ファイルに出力する機能が必要になります。
実行結果をファイルに出力するため、次の機能を提供します。ただし、ファイルに出力される内容は一部のエラーメッセージが除かれています。

```
(open_output ファイル名)
```

ファイル名はデフォルトのパスが指定されてまする。このパスは*out-pathname*に設定されています。
また、出力が不必要になった場合は

```
(close_output)
```

の実行により通常状態に戻ります。

3.4 日英変換処理の実行

変換処理の実行は以下の2関数を使って実行します。

---

trans *fs*

素性構造*fs*にメタ規則:mainあるいは指定したメタ規則を適用し、素性構造のリストを返す。transでは、5章で実現するタイプシステムで利用する素性構造内のタイプ値を削除する。本システムで扱うタイプシステムおよびタイプ値は日英変換処理に利用する書き換え規則の意味的な制約を実現するために導入したものであり、解析処理および生成処理と連動していないためである。また、6章の前文参照機能に対応して、書き換え結果を大域変数:*previous_transout*にセットする。

---

transfer *fs* &key :parameter-environment :control-rule-application :raw-mode

---

素性構造*fs*に書き換え規則を適用し素性構造のリストを返す。キーワードパラメータparameter-environmentは書き換え環境の指定である。parameter-environmentで指定した書き換え環境のもとで、書き換え規則の検索が行なわれる。書き換え環境の既定値は大域変数*default-rule-parameter*の値((:phase . :j-e) (:type . :general))である。規則の適用制御方法はキーワードパラメータcontrol-rule-applicationで指定する。control-rule-applicationの指定は書き換え規則の演算子rewriteの引数*control*に対応している。control-rule-applicationの既定値は大域変数*control-apply-rule*の値(:loop :recursive)である。この指定では、ルートから素性構造を再帰的に辿り(:RECURSIVE)、かつ、規則適用の方法は適用できる規則がなくなるまで(:LOOP)規則の適用が試みられる。素性構造の書き換えは、システム内部では元の素性構造から書き換えるべき素性構造へforwardingすることによって行なわれる(8.3章参照)。:raw-modeに*nil*が指定されると、すべての書き換えが終了した時点で、書き換えられた素性構造からforwardingにしたがって新たに素性構造のコピーを作り、コピーされた素性構造を返す。:raw-modeに*nil*以外が指

定されると、コピーを生成されずforward linkを含んだ元の(引数として与えられた)素性構造*fs*を返す。

3.5 処理範囲の指定

---

trans-all

入力素性構造のリストに対して処理範囲の指定を行う。関数を起動すると、以下のメッセージが出力され、質問に答える形式で設定を行う。

```
> (rws::trans-all 入力素性構造リスト)
Do you need process time (Y or N):
Do you have exec-range (Y or N):
Input start position =>
Input end position =>
```

- Do you need process time (Y or N):
  yと答えると、1文単位で消費された時間の情報を出力する。
- Do you have exec-range (Y or N):
  yと答えると、
  1. 処理の開始位置;Input start position =>
  2. 終了位置;Input end position =>

  を指定する。nと答えると、入力素性構造リストすべての素性構造について処理を行う。

3.6 規則適用トレースの設定

---

debug-transfer

トレースモードの設定をおこなう。関数を起動すると、以下のメッセージが出力され、質問に答える形式でどの情報をトレースするかを指定する。これらの質問にはy、n、VあるいRETURNを入力する。RETURNは[ ]内で表示されて

いるデフォルト値がある場合はその値が選択される。ただし、素性構造の途中結果の出力を指定する項目（下線部分）のみyの入力により、ユーザが書き換え環境を入力する。メッセージはストリーム *debug-output*に出力される。

```
> (rws:debug-transfer)
Print Input Feature Structure [RWS::YES](Y or N):
Print Rewritten Results [RWS::YES](Y or N):
Trace Applying Rules to FS and Result FSs [RWS::NO](Yes, No or Verbose)
Trace Rewriting Call [RWS::NO](Yes, No or Verbose)
Trace Pattern Matching[RWS::NO](Y or N):
Trace Parameter Setting [RWS::NO](Yes, No or Verbose)
Do you set parm-env for Rewrite FS (Y or N):
Do You Execute Preference Check ? (Y or N):
```

- Print Input Feature Structure[YES](Y or N):
  yと答えると、書き換え規則適用関数transferあるいはtransを呼び出した時、入力の素性構造(引数)を画面に表示する。
- Print Rewritten Results[YES](Y or N)
  書き換え規則適用関数transferあるいはtransを呼び出した時、出力の素性構造(関数値)を画面に表示する。
- Trace Applying Rules to FS and Result FSs [NO](Yes, No or Verbose)

  規則の適用をトレースする。yが指定されると、書き換え規則の呼び出し、および、規則適用の終了の状態(successあるいはfail)、規則適用が成功した場合には規則適用によって何個素性構造が返されたかが表示される。vが指定されると、上記メッセージに加え規則の入力素性構造と書き換え結果の素性構造を表示する。
- Trace Rewriting Call [NO](Yes, No or Verbose)
  書き換え呼び出しをトレースする。yが指定されると、書き換え規則から書き換え呼び出しが行なわれた時呼び出しを行なった規則名と書き換え呼び出しによって生成されたプロセスの終了状態を表示する。vが指定されると、上記メッセージに加えて書き換え呼び出しの対象となる素性構造と、書き換え呼び出しによって得られた素性構造を表示する。

- Trace Pattern Matching[NO](Y or N):
  書き換え規則内で行なわれるパターンマッチングをトレースする。yが指定されると、パターンマッチングが失敗した時、何が原因で失敗したかをレポートする。

- Trace Parameter Setting [NO](Yes, No or Verbose)
  書き換え環境の設定をトレースする。yが指定されると、書き換え環境の状態が変更されたとき書き換え環境の状態を表示する。vが指定されると、上記メッセージに加え、書き換え環境の変更が行なわれた時の素性構造も表示する。各フェイズでどのように素性構造が書き換えられたかをモニターする。

- Do you set parm-env for Rewrite FS (Y or N):
  書き換え処理途中の素性構造を出力する。yが指定されると、次のメッセージが出力される。Please input set-environ.

  ここで以下のように設定すると、書き換え環境の設定の確認の問い合わせを行う。そして、j-j構造変換終了時での書き換えられた素性構造を表示する。(:phase :japanese)
  You set fs-environ::(:PHASE :JAPANESE). OK (Y or N):

- Do You Execute Preference Check ? (Y or N):
  ある書き換え制約下で複数の結果が得られた場合、適用された書き換え規則の記述内容によって結果を絞り込む機能を付加する。例えば、素性構造パターンのより詳細な記述がある規則を優先するなどがある。

3.7 大域変数

3.7.1 実行環境パス

- *rws-trans-directory*
  任意のユーザが素性構造書き換えシステムを利用する場合のパスをセットする。この変数の設定方法は9.1を参照のこと。

- *rws-home-directory*
  素性構造書き換えシステムの処理系を利用するための相対パスに対応させるためのパスをセットする。

- *rws-rule-directory*
  素性構造書き換えシステムに使う書き換え規則を利用するための相対パスに対応させるためのパスをセットする。

- *rws-rule-file*
  書き換え規則の二次記憶化ファイルを作成した場合、規則本体が存在するパスをセットする。この変数によりインデックスファイルでは相対パスで本体ファイルが指定されている箇所に対処するためである。

3.7.2 書き換え規則

- *default-rule-parameter*
  規則の定義時に規則適用制約が省略された時、この変数に代入されている値を規則適用制約とする。既定値は((:PHASE . :J-E) (:TYPE . :GENERAL))である。

- *rw-rule-load-verbose*
  nil以外の値が設定されていると書き換え規則のロード時にメッセージを出力する。

- *rule-memory-fault-message*
  nil以外の値が設定されていると書き換え規則がメモリ上にない時に、メッセージを出力する。既定値はnilである。

- *pathbase*
  書き換え規則がメモリに読み込まれると、各書き換え規則の素性パスおよび対応するハッシュテーブルをセットされる。この時、素性パスが同一の規則はグループ化される。

4 書き換え規則の管理モジュール

4.1 書き換え規則の定義

---

defrwrule &body *form*
defrwschema2 *id type sort-key body*

書き換え規則を定義する。規則はコンパイルされルールベースに登録される。defrwruleは、引数*body*に書き換え規則の定義の並びを取る。defrwschema2で定義した規則は、*id*を*rws-IDDB*に対応するハッシュテーブルに登録し、さらに*rw-rule-schema-type-database*にも登録される。規則の管理モジュールにおける検索は*id*をキーにして*rws-IDDB*を通して行う。規則の定義は*body*にストリングで記述する。*type*および*sort-key*は規則をファイルに書き出すsave-rw-schemaのための引数である。

4.2 書き換え規則の更新

---

remove-all-rw-rules2
remove-rw-rule2 *id*

4.3 書き換え規則の削除

remove-all-rw-rules2は定義されているすべての規則をルールベースから削除する。具体的には*rws-IDDB*に対応するハッシュテーブルをクリアすることで実現する。remove-rw-rule2は*id*で指定された規則名の書き換え規則をルールベースから削除する。

4.4 書き換え規則の表示

---

print-rw-rule2 *id*
print-all-rw-rules2 &key :print-document :print-text

ルールベースに定義されている書き換え規則を表示する。*id*は規則の固有名を指定する。print-rw-rule2は、*id*に対応する書き換え規則を表示する。キーワード:print-documentにtを指定すると、規則のドキュメンテーションが表示される。キーワード:print-textにtを指定すると、規則のテキスト(規則の定義)が表示される。

```
(rws::print-rw-rule2 'defn001)

登録用紙:
----------
defn001.....................................................規則固有名
Parameter                    Value
---------                    -----
:PHASE                       :J-E ..........................規則適用制約
:TYPE                        :GENERAL

on <reln> 登録用紙 .....................................規則の定義内容
  in=  [[reln 登録用紙]
        [agen ?agen]
        [obje ?obje]
        ?rest]
  out= [[reln registration_form]
        [agen ?agen]
        [obje ?obje]
        ?rest]
end
```

4.5 書き換え規則の保存

---

save-rw-schemas *filename type*
save-index-and-rules2 *index-filename rule-filename* &optional **extend-flg**

---

save-index-and-rules2は書き換えシステム中に定義されているすべての書き換え規則をファイルに書き出す。*index-filename*は書き換え規則のインデックスを出力するファイル名の指定である。インデックスとは、本解説書PART Iの6章にある図1中のfeature path tableとhash tableを指す。*rules-filename*はコ

ンパイルされた書き換え規則を出力するファイル名の指定である。書き換えシステムを再起動した後、あるいは、すべての規則を削除した後、*index-filename*をLISP関数LOADでロードすることにより新たに、書き換え規則のインデックスが作られる。規則のインデックスがロードされていると、規則の参照(検索や表示)が行なわれた時にメモリ中に規則の定義がなければ、*rules-filename*から自動的にロードされる。規則本体のロードは、ファイル*rule-filename*をopenし規則定義のcompiled codeを読み込む。save-index-and-rules2の実行時に、&optionalが指定されている場合は書き換え規則をファイルに出力する際、素性構造のパターン部分を内部構造に展開して出力する。また、インデックスを用いたロード時には相対パスで指定されたファイルをオープンするが、変数*RWS-RULE-DIRECTORY*をあらかじめ本体ファイルへのパスをセットすることにより、システムを実行しているcurrent working directoryに関係なく参照可能となる。

以下に、規則のセーブの例を示す。

```
> (rws::defrwschema2 defv001 t t " .....................................規則の定義
on <reln> 送る
   in=  [[reln 送る]
         [agen ?agen]
         [obje ?obje]
         ?rest]
   out= [[reln send]
         [agen ?agen]
         [obje ?obje]
         ?rest]
end
")

送る
#<Structure RWS::RW-RULE C75FEE>
> (rws::print-all-rw-rules2) ..........................................規則の表示

送る:
```

```
defv001
Parameter                    Value
---------                    -----
:PHASE                       :J-E
:TYPE                        :GENERAL

NIL
> (rws::save-index-and-rules2 "index-file" "rules-file") .....規則のセーブ

0
NIL
> (rws::remove-all-rw-rules2) .....規則の削除
NIL
> (rws::print-all-rw-rules2) .....規則の表示
NIL .....定義されている規則はない
> (setq rws::*rule-memory-fault-message* t)
T
> (load "index-file") .....インデックスファイルのロード
;;; Loading source file "index-file"
#P"/home3/nadine/transfer/rules/index-file"
> (rws::print-all-rw-rules2) .....規則の表示

;;; loadind rewrite rule 送る .....規則の本体をロードしている
送る: .....規則がロードされ表示された
NIL
Parameter                    Value
---------                    -----
:PHASE                       :J-E
:TYPE                        :GENERAL

NIL
>
```

4.6 大域変数

- *save-index-name-and-address*
  二次記憶化のインデックスファイルに書き換え規則名と本体ファイル内のアドレス値のペアリストを作成し、セットする。この変数はメモリ上に本体が存在しない書き換え規則のメモリロードに利用する。

5 タイプシステム

タイプシステムを利用する場合、書き換え規則に記述するタイプ値は規則の適用を行う前に登録する必要があります。また、既に定義されているタイプ値については

- タイプ値の更新
- タイプ値の削除
- タイプ値の表示

などの管理機能が必要になります。本システムではタイプ値に関するこれらの機能を提供しています。

5.1 タイプ値の定義

---

deffstype *type subtype*

関数deffstypeは現在のタイプ値定義にタイプ値*type*およびそのタイプ値の下位分類に相当するタイプ値を定義する。*subtype*は複数が定義可能であり、これらは*type*の直下に対応づけられる。
deffstypeで定義されたタイプ値は関数type-level-set-mainの実行により、メモリ上の内部構造に展開される。

5.2 タイプ値の更新

---

update-fstype *type subtype*

関数 update-fstypeはタイプ値の関係づけを更新する。*type*の直下に新たに関係づけられるタイプ値を*subtype*に記述する。*subtype*は複数指定しても構わない。

5.3 タイプ値の削除

---

delete-deftype *type*

関数 delete-deftypeは*type*で指定したタイプ値を削除する。*type*の直下に関係づけられているタイプ値は*type*の上位で直接、支配されているタイプ値に結合を変更する。

5.4 タイプ値の表示

---

print-subtype *type*
print-subtypes *type*

関数 print-subtypeは、タイプ値*type*の直下に定義されたサブタイプを表示する。関数 print-subtypesはタイプ値*type*のサブタイプとして定義されたすべてのタイプを表示する。さらに、各タイプ値のレベルを表示する。

6 談話処理モジュールとの結合

現在の素性構造書き換えシステムを用いた日英変換処理では

1. 格の省略補完
2. 発話タイプの決定
3. 英語意味構造の生成

に大きく分類した区分けで素性構造の書き換え処理が行われます。談話処理モジュールが要求する素性構造は格の省略補完後の素性構造です。関数transの実行による日英変換処理ではメタ規則の途中（素性構造の書き換え途中）に談話処理モジュールを呼び出す機能をサポートしていなため、談話処理モジュールを結合する場合は以下の関数を利用して日英変換処理を行います。

---

trans-ellipses *fs* &optional *mrule*
trans-main *fs* &optional *mrule*

---

trans-ellipsesは書き換え規則中の省略補完に関する規則を適用する。trans-mainは談話処理モジュールから得られた素性構造に対して英語の素性構造を生成する。生成された素性構造は大域変数*previous_transout*にもセットされ、次の文の日英変換処理で前文情報として参照される。

7 処理結果の比較

素性構造書き換えシステムを利用して得られた素性構造結果はファイルに出力可能です。また、大量の処理を一括して実行可能な機能を用意しているため、この機能を利用して書き換え結果をファイルに出力することができます。書き換え規則の改良／拡充に伴い、大量実験する処理結果は毎回、その内容をチェックする必要がありますが、そのツールとしてファイル単位で処理結果（素性構造）を比較する機能を提供します。この利用方法を以下に示します。

「動作環境の設定」

- 変換処理の実行環境が用意されている
- 比較プログラムのソースファイル（fs-comp.lisp）がある
- 変換処理結果ファイルが２つある

上記の前提で通常、以下の実行を行う。

1. 比較プログラム（fs-comp.lisp）をロードする

```
> (load (merge-pathnames "engine/fs-comp" *rws-home-directory*))
```

2. 比較プログラムを実行する

```
> (rws::fs-comp "結果ファイル１" "結果ファイル２")
```

上記の実行で出力されるメッセージには以下のものがあります。

1. 変換プログラムのエラーに関するメッセージ

   (a) 変換結果ファイルに『＝＝＝Transfer Result＝＝＝』の部分がない。

＜出力例＞

```
> (rws::hikaku "ke1" "ke7")
 file1 = #P"/home/nadine/henkan/ke1"
 file2 = #P"/home/nadine/henkan/ke7"
 変換結果ファイル "ke7" に Transfer Result がない！！
 ********** ID =  ">No. da-5" **********
 file2 の 素性構造がない
```

(b) File1あるいはFile2の素性構造がない

＜出力例＞

```
> (rws::hikaku "file1" "file2")
file1 = #P"/home/nadine/henkan/file1"
file2 = #P"/home/nadine/henkan/file2"
********** ID = " >No. da-5" **********
file2 の 素性構造がない
```

(c) File1あるいはFile2の適用規則数か書き換え規則数がない

```
> (rws::hikaku "ke1" "ke12")
 file1 = #P"/home/nadine/henkan/ke1"
 file2 = #P"/home/nadine/henkan/ke12"
 ********** ID =  ">No. da-5" **********
 file2 の 適用規則数がない
 適用規則数が違う : file1="17 " ,file2=NIL          ←それぞれの規
則数
 file2 の 書き換え規則数がない
 書き換え規則数が違う : file1=" 7 " ,file2=NIL      ←それぞれの規
則数
```

(d) File1のＩＤあるいは文ＩＤがない

2. 対応する素性構造の違いに関するメッセージ

(a) 変換結果の個数の違い

i. File1とFile2で一致する文IDの変換結果がない

＜出力例＞

```
> (rws::hikaku "ke1" "ke13")
 file1 = #P"/home/nadine/henkan/ke1"
 file2 = #P"/home/nadine/henkan/ke13"
 file2 に ID の一致する変換結果がない
   足りない結果の ID =  ">No. da-5"                ←file2に
ないＩＤ
```

(b) 規則数の違い

i. 適用規則数が違う

＜出力例＞

```
> (rws::hikaku "ke1" "ke8-1")
file1 = #P"/home/nadine/henkan/ke1"
file2 = #P"/home/nadine/henkan/ke8-1"
********** ID = " >No. da-5" **********
適用規則数が違う : file1="17 " ,file2="117 "        ←それぞれ
の規則数
```

ii. 書き換え規則数が違う

＜出力例＞

```
> (rws::hikaku "ke1" "ke8-2")
        file1 = #P"/home/nadine/henkan/ke1"
file2 = #P"/home/nadine/henkan/ke8-2"
********** ID = " >No. da-5" **********
書き換え規則数が違う : file1=" 7 " ,file2=" 107 "  ←それぞれ
の規則数
```

(c) 素性構造の違い

i. ラベルによる参照が違う

＜出力例＞

```
>(rws::hikaku "ke1" "ke2-1")
file1 = #P"/home/nadine/henkan/ke1"
file2 = #P"/home/nadine/henkan/ke2-1"
********** ID = " >No. da-5" **********
ラベルによる参照が違う
  fs_pass : (SEM OBJE AGEN)
```

ii. 素性値の（タイプ）種類が違う

＜出力例＞

```
> (rws::hikaku "ke1" "ke4")
file1 = #P"/home/nadine/henkan/ke1"
file2 = #P"/home/nadine/henkan/ke4"
********** ID = " >No. da-5" **********
fs のタイプが違う
        fs_pass : (PRAG TOPIC)
        type1 : :COMPLEX   ,  type2 : :LEAF          ←それぞ
れの素性値の種類
```

iii. f s の値（素性値）が違う

＜出力例＞

```
> (rws::hikaku "ke1" "ke9")
file1 = #P"/home/nadine/henkan/ke1"
file2 = #P"/home/nadine/henkan/ke9"
********** ID = " >No. da-5" **********
fs の値が違う
         fs_pass : (SEM OBJE TENSE PRESENT)
 value1 : PRESENT , value2 : PRPRESENT          ←それぞれの
素性値
```

iv. File1 と File2で一部の素性構造が異なる

＜出力例＞

```
> (rws::hikaku "ke1" "ke5-1")
file1 = #P"/home/nadine/henkan/ke1"
file2 = #P"/home/nadine/henkan/ke5-1"
********** ID = " >No. da-5" **********
file2 に対応する素性パスがない
        fs_pass = (SEM OBJE RECP)                  ←file2
にない素性パス
        fs_pass = (SEM OBJE RECP AGEN)
fs_pass = (SEM OBJE RECP AGEN TENSE)
```

8 書き換え規則コンパイラの対応

8.1 規則マクロプロセッサ

書き換え規則の中には、名詞（特に固有名詞）などのように定型的な素性構造パターン素性構造パターン記述になる規則が多くなる可能性があります。そのため、予め書き換え規則の定義パターンを用意し、書き換え規則毎に変更される情報のみを他のファイルで定義しておき、書き換え規則を自動的に生成する機能を用意します。
例えば、サンプルテンプレートに対して

- @JAPANESELEX@ 日本語語義
- @ENGLISHLEX@ 英語語義
- @YOMI@ 読み
- @ABSONO@ スキーマ番号

の変数部分を代入することにより、書き換え規則が定義できます。

```
(rws:defrwschema2 defn@ABSONO@ N @YOMI@
"on <restr reln> @JAPANESELEX@-1 in :PHASE :J-E :TYPE :default
   in= [[PARM !X[]]
        [RESTR [[RELN @JAPANESELEX@-1]
                [ENTITY !X]]]]
   out= [[PARM !X[]]
         [RESTR [[RELN @ENGLISHLEX@-1]
                 [ENTITY !X]]]
         [INDEX [[NUMBER SING]
                 [GENDER NEUT]
                 [DEFINT NULL]
                 [PERSON THIRD]]]]

end")
```

入力データは先の変数に対応する値を 1 レコードに対して第 1 フィールドから第 4 フィールドまでに記述します。例えば、以下のデータを登録し、書き換え規則作成のシェル規則作成シェルを実行すれば、テンプレート中の変数部分を置き換えた書き換え規則が作成できます。

アカンパニーパーソン accompanying_person-IDIOM あかんぱにーぱーそん 1
秋 autumn-N あき 1

以下にここで用いた例を作成するためのシェル記述を示します。

```
PATH=/bin:/usr/ucb:/usr/bin; export PATH
AWK=/bin/awk
tmprulef=/usr/tmp/ruletmp.$$
tmpldir=/user1/asura/transfer/rws-v2/compiler/samples/macro-processor/tmplate
baseno=1
verbose=no

if [ x$1 = x-b ]; then
baseno=$2
shift; shift
fi
if [ x$1 = x-v ]; then
verbose=yes
shift
fi
if [ x$1 = x-t ]; then
   tmpldir=$2
   shift; shift
fi

lpcnt=1
for filen in $*; do
if [ ! -f $filen.smpl ]; then
echo "$filen.smpl: No such file" >&2
goto USAGE
echo "Usage: `basename $0` [ -b basenumber ] [ -v ]
                [ -t template-dirname ] basefilename basefilename ..." >&2
echo "Filenames are those which are got rid of .smpl postfix." >&2
exit 8
fi
cp /dev/null $filen.source
while read lined; do
if [ $verbose = yes ]; then
echo No.$lpcnt $lined >&2
fi
ruleno=`expr $baseno + $lpcnt - 1`
errchk=`echo $lined | $AWK ' print NF '`
if [ $errchk != 4 ]; then
echo "line No.$lpcnt error." >&2
continue
fi
japanese=`echo $lined | $AWK ' print $1 '`
english=`echo $lined | $AWK ' print $2 '`
yomi=`echo $lined | $AWK ' print $3 '`
tmplname=`echo $lined | $AWK ' print $4 '`
echo "s/@RULENO@/$ruleno/" > $tmprulef
echo "s/@ABSONO@/$lpcnt/" >> $tmprulef
echo "s/@JAPANESELEX@/$japanese/" >> $tmprulef
echo "s/@ENGLISHLEX@/$english/" >> $tmprulef
echo "s/@YOMI@/$yomi/" >> $tmprulef
```

```
echo "s/@FILENAME@/`basename $filen`.smpl/" >> $tmprulef
sed -f $tmprulef $tmpldir.$tmplname >> $filen.source
rm $tmprulef
lpcnt=`expr $lpcnt + 1`
done < $filen.smpl
done
exit 0
```

8.2 コンパイル機能の対応

書き換え規則のコンパイラは書き換え規則からLISPプログラムを生成します。コンパイラは書き換え規則1を以下のLISPプログラムに変換します。

規則 1

```
on <reln> 送る in :phase :J-E
   in=  [[reln 送る]
         [agen ?agent]
         [obje ?object]
         ?rest]
   out= [[reln send]
         [agen ?agent]
         [obje ?object]
         ?rest]
end
```

```
(RWS::PUT-RW-RULE
 "送る->send"
 '(lambda (input object)
    (block general-block0
           (let ((pairlis nil))
             (setq pairlis (RWS::INSTALL-KEY-VALUE '?input input pairlis))
             (setq pairlis (RWS::INSTALL-KEY-VALUE '?ROOT
                                                   (RWS::OBJECT-FS object)
                                                   pairlis))
             (unless ( RWS::FS-MATCH ( RWS::FETCH-VALUE '?input pairlis)
                                     ( RWS::MAKE-FS '( [[reln 送る]
                                                       [agen ?agent]
                                                       [obje ?object]
                                                       ?rest] )
                                                    pairlis)
                                     object pairlis 'okuru)
                     (return-from general-block0 nil))
             (return-from general-block0
                          (RWS::MAKE-FS-OBJECT
                           (RWS::FS-FORWARD
                            (RWS::FETCH-VALUE '?input pairlis)
                            (RWS::FS-RECONSTRUCT (RWS::MAKE-FS '([[reln send]
                                                                  [agen ?agent]
                                                                  [obje ?object]
```

```
                                                          ?rest])
                                                       pairlis)
                                                 pairlis))
                                object)))))
 "on <reln> okuru
in :phase :J-E
in= [[reln okuru]
         [agen ?agent]
         [obje ?object]
         ?rest]
   out= [[reln send]
         [agen ?agent]
         [obje ?object]
         ?rest]
end"
 '((reln) 送る)
 :phase :J-E)
```

以下コンパイルされたコードに含まれるLISP関数について説明します。

---

put-rw-rule *documentation S-exp text definition* &rest *ApplicationConstraints*

関数put-rw-ruleは書き換え規則をルールベースに登録する。*document*は書き換え規則のドキュメンテーションである。*S-exp*は書き換え規則のLISPコードへのコンパイル結果である。*text*は書き換え規則のソースコードである。*definition*はLISTであり、LISTの第一要素は規則が定義された素性パスであり、第二要素は素性パスの素性値である。*ApplicationConstraints*は規則適用制約である。

コンパイラは書き換え規則をlambda式に変換する。変換されたlambda式は二つの引数をとる。第一引数 input は入力素性構造であるnode構造体が、第二引数objectは書き換えシステム内のプロセスに相当するobject構造体がバインドされる。object構造体は以下のスロットを持つ。

FS
parameter-environment
FS-history
forward-list

FSは素性構造のルートを保持する。parameter-environmentは書き換え環境である。FS-historyはプロセス内で辿った素性構造の履歴(どの素性構造を既

に辿ったか)が記録され、辿った素性構造のリストが入る。これは対象の構造がDAGであることから必要性が生じる。forward-listでは、プロセス内でforwardingしたforward linkのリストを管理する。つまりこのプロセス内で行なわれた書き換えの履歴をforward-listスロットで記録している。

lambda式は一つのブロックから構成される。このブロックは、failやreturn、out=による書き換え規則の強制終了に対応するLISP特殊形式return-fromがブロックから出るために定義されている。次にletで生成される局所変数pairlisは、規則内の変数を管理するペアリストである。規則内で参照される変数はすべてparlisに登録される。次の(setq pairlis ...)はシステム変数のpairlisへの登録である。システム変数root、inputを初期化している。(unless (RWS::FS-MATCH ...))は素性構造のパターンマッチングである。素性構造パターンは規則実行時にマクロ定義MAKE-FSで動的に生成される。マクロ定義MAKE-FSは素性構造生成時にparlisを参照している。素性構造パタン中に現れた変数が既に値を持っていれば、変数を値で置き換える。変数が値を持っていなければ、変数を生成する。入力素性構造と素性構造パターンとのパターンマッチングは、マクロ定義FS-MATCHにより行なわれる。FS-MATCHがパターンマッチングが成功すると、変数と一致した素性構造を変数の値としてpairlisに登録する。FS-MATCHはパタンマッチングに成功するとnil以外の値を返す。失敗した時nilを返す。

式(return-from ...)では入力素性構造を素性構造パターンから生成した素性構造に書き換える(関数fs-forward)。このlambda式の適用によって、素性構造とオブジェクトのCONSのリストが返される(関数make-fs-object)。

以下にコンパイラが生成するLISPコードで参照される関数の一覧を示します。

---

fs-match *fs1 fs2 pairlis* &optional *rule-name*

素性構造*fs1*と*fs2*とのパタンマッチングをおこなう。素性構造に変数が含まれている時には、パタンマッチングが成功したら*pairlis*に変数と値を登録する。

---

object-fs *object*

---

object構造体のfsスロットの値を返す。つまり、ルートの素性構造である。

---

fs-rewrite-main *fs object AttributeValueList control*
fs-rewrite-subfs *rule-name result fs object pairlis* &rest *AttributeValueList*
fs-rewrite-subfs-recursively *rule-name result fs object pairlis* &rest *AttributeValueList*

関数fs-rewrite-mainは書き換え呼び出しrewriteに対応している。関数fs-rewrite-subfsは書き換え呼び出し=>, ->に対応している。関数fs-rewrite-subfs-recursivelyは書き換え呼び出し==>, -->に対応している。*rule-name*は書き換え規則の規則定義部の*AtomicFeatureStructure*である。書き換え結果は*result*をキーとして*pairlis*に登録される。*result*にはシステム変数?itが与えられる。*fs*は書き換え対象の素性構造である。書き換え呼び出しの際、書き換え環境は*AttributeValueList*に設定される。

---

map-progn *block-name (input object pairlis) rewrite-call* &body *body*
map-progn2 *body block (object pairlis) rewrite-call* &body *body*

マクロmap-progn, map-progn2は書き換え呼び出し*rewriting-call*を評価する。一般に書き換え呼び出しをおこなうと複数の素性構造が返る。map-progn, map-progn2は各々の書き換え結果に対して*body*を実行する。

---

make-fs-object *fs object*

素性構造とオブジェクトのconsのリストを返す。

---

make-fs *fs object*
fs-reconstruct *fs pairlis*
fs-get-subfs *fs path*
tr-fs-feature-list *fs*
fs-type *fs*
fs-append-fv-pair-list-with-copy *fs fvlist*
fs-put-feature-value-with-copy *fs value path*
fs-delete-feature-with-copy *fs path*

---

make-fsは素性構造パタンの記述から素性構造を生成する。

関数fs-get-subfsは素性構造*fs*において素性のパス*path*を辿り値を返す。

関数tr-fs-feature-listは素性構造*fs*の素性のリストを返す。

関数fs-typeは素性構造*fs*のタイプを返す。

関数fs-append-fv-pair-list-with-copyは素性構造*fs*に素性と値の対のリスト*fvlist*を追加する。fs-append-fv-pair-list-with-copyはfsのルート(一つのノード構造体)をコピーし元の素性構造からコピーへのforwardingを行なった後、ノードのコピーに対してfvlistを追加する。

関数fs-put-feature-value-with-copyは素性構造*fs*に素性のパス*path*にしたがって再帰的にパスを辿り、素性パスの値として素性構造*value*を与える。素性のパスを辿る際に素性構造*fs*に素性パス*path*の素性がない時には、パスが追加される。パスの追加では対象の素性構造をコピーし、対象素性構造をコピーへforwardingした後コピーに対して素性を追加する(図?? 参照)。

関数fs-delete-feature-with-copy素性構造*fs*から素性*path*を削除する。削除は上述の関数と同様コピーされた素性構造が操作対象である。

---

fs-forward *fs1 fs2*

素性構造*fs1*を*fs2*にフォアーディングする。

---

install-key-value *variable value pairlis* &optional *test*
fetch-value *variable pairlis* &optional *test*

マクロinstall-key-valueは変数*variable*に値*value*のペアを変数を管理しているペアリスト*pairlis*に登録する。この登録の操作は変数への値の代入操作に相当する。fetch-valueは変数*variable*を*pairlis*から検索し、変数の値を返す。

---

set-parameter-environment-2 *object* &rest *AttributeValueList*
remove-parameter-environment-2 *object* &rest *AttributeList*

---

set-parameter-environment-2は書き換え環境の設定を行なう。書き換え環境の設定は*AttributeValueList*を*object*のスロットparameter-environmentにセットすることにより行なわれる。remove-parameter-environment-2は*object*のparameter-environmentに格納されているペアリスト*AttributeValueList*から*AttributeList*で指定された属性(ペア)を削除する。

8.3 素性構造の内部構造と書き換え処理

node structure

value
type
rest-var
tr-forward
modifier
modifier-backward
nameplate
id

forward-link

forward-id
feature-structure

valueは素性構造が:atomicタイプのときにはatomic素性構造の値が入る。また、:leafタイプの時はnilである。それ以外のタイプのときには素性と素性値の対のリストが入る。typeは素性構造のタイプが入る。値は:complex, :atomic, variable, :leafのいずれかである。rest-varは素性構造パタンがrest variableを持っているとき、このスロットにrest variableが入る。tr-forwardにはポインタforward linkが入る。素性構造の書き換えはforward linkをもちいて元の素性構造から書き換えるべき新しい素性構造にフォワードすることによって実現されている。forward linkには、どのプロセス（書き換え規則）でフォワードされたものかが識別できるよう識別名が付けられている。各プロセスは自分のプロセスで有効なforward linkの識別名を保持している。nameplateには、素性構造パタンにglobal tagが付けられた時にglobal tagの名前が入る。idは各ノードのidである。素性構造のトークンとして同一性はidによって判断する。

規則2を素性構造1に適用すると規則2内の書き換え呼び出しによって規則3および規則4が適用される。規則2により最終的に素性構造1は二つの素性構造2, 3が得られる。書き換えシステム内で行なわれたフォワーディングをグラフで表現すると図1になる。図1においてforward link*p1*でフォワードされた素性構造は素性構造2、*p2*でフォワードされた素性構造は素性構造3である。

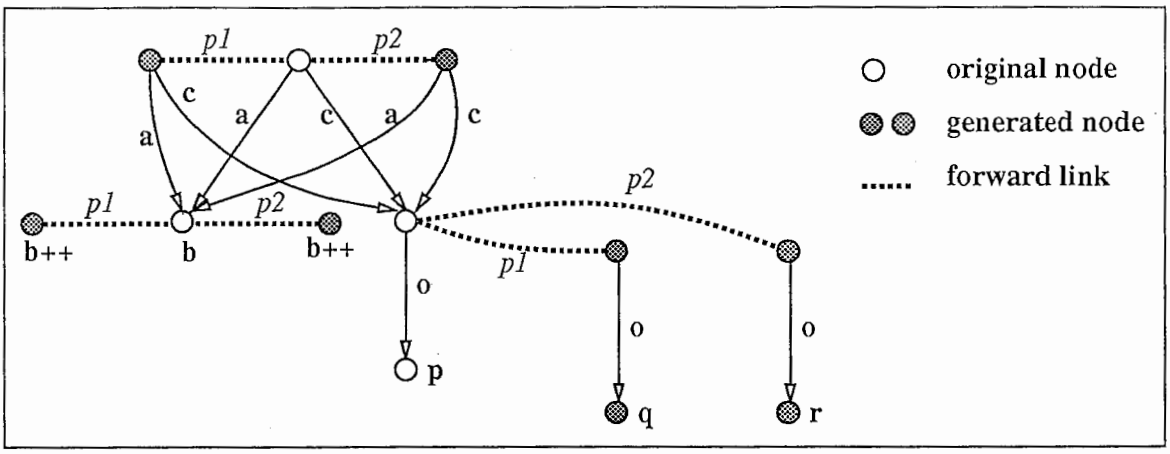

図 1: forwardingによる素性構造の書き換え

規則 2

```
on <a> b
   in=  [[a b]
         [c ?X]]
   -> ?X with :att :val
   out= [[a b++]
         [c ?X]]
end
```

規則 3

```
on <o> p in :att :val
   in=  [[o p]]
   out= [[o q]]
end
```

規則 4

```
on <o> p in :att :val
   in=  [[o p]]
   out= [[o r]]
end
```

素性構造 1

```
[[a b]
 [c [[o p]]]]
```

素性構造 2

```
[[a b++]
 [c [[o q]]]]
```

素性構造 3

```
[[a b++]
 [c [[o r]]]]
```

9 素性構造書き換えシステムの利用方法

素性構造書き換えシステムを複数のユーザが利用する方法を説明します。
ユーザが上記システムを利用する場合、ユーザ毎にworking directoryが異なり、利用するマシンが異なる場合もあります。
書き換えシステムを一元管理し、任意のworking directoryから利用するために書き換えシステムのロード機能（初期化）に改良を加えました。

9.1 ユーザ毎の設定

書き換えシステムを利用するユーザは次のファイルをworking directoryに保持する。

libc.o libc.oはユーザが利用しているマシンのホスト名を得る機能を提供するモジュール。

load.lisp load.lispは書き換えシステムのロードファイル。

これらのファイルは2章の*rws-home-directory*の下位に存在しています。まず、自分のworking directoryにコピーして下さい。
上記のload.lispにあらかじめホスト名と本システムを利用するためのパスの対応を記述します。記述方法は以下のように指定して下さい。

```
(defvar *HOST_LIST* nil)
(defvar *HOST_Mclaren* (cons 'Mclaren "/usr/asura/translation/transfer/"))
(defvar *HOST_as12*
  (cons 'as12 "/mnt/usr/asura/translation/transfer/"))
(defvar *HOST_as17*
  (cons 'as17 "/NFS/usr/asura/translation/transfer/"))

(eval `(setq *HOST_LIST* '(,*HOST_Mclaren* ,*HOST_as12* ,*HOST_as17*)))
```

上記の例はホスト名：McLarenにある素性構造書き換えシステムを他のマシンからリモートで利用する場合の指定方法です。

また、ホスト名を得るためにロードファイルに以下の記述を追加して下さい。

```
;;;    define C function
(define-c-function (c_gethostname "_c_gethostname") (buf)
 :result-type :string)
;;;    loading C function
(setf c-object "libc.o")
(lucid::load-foreign-files (list c-object))
;;; LISP function Calling compiler
(defun real-module-name()
 (setq pathname "                                    ")
 (setq hostname (c_gethostname pathname))
        (let ((s-hostname (read-from-string hostname)))
   (if (assoc s-hostname *HOST_list*)
       (cdr (assoc s-hostname *HOST_LIST*))
     (loop
      (format t "Please Input Path ==> ")
      (terpri)
      (setq s-hostname (read-line))
      (when  (stringp s-hostname)
     (return s-hostname))))))

(setf *rws-trans-directory* (real-module-name))
```

→ ここでホスト名に対応するパスをセットする

9.2 計算機環境による設定の相違

素性構造書き換えシステムは書き換え規則のコンパイルにC言語で作成したオブジェクトファイルを利用しています。このファイルは異なる計算機ではオブジェクトとしての互換性がない（例えば、ＳＵＮとＨＰ）ため、利用環境に応じて、必要なオブジェクトファイルを作成する必要があります。具体的には書き換え規則のコンパイラに利用するモジュールを作成するmakefileを修正することになります。makefileは2章の*rws-home-directory*の下位にあるcompiler/MakefileおよびMakefile.hpを参照して下さい。さらに、同じディレクトリにあるcompiler.lispも修正する必要があります。(compiler.hp.lisp)を参照のこと。

10 実行履歴

本システムの開発に利用したサンプル文を例に、日英変換処理の実行履歴を示します。ただし、規則の適用に失敗した履歴は一部、省略しました。また、適用した書き換え規則の一部の定義として、規則5は発話タイプの決定、規則6は語義：登録用紙のJ-E変換、規則7は語義：送るのJ-E変換示しています。

規則 5

```
IFT009          てもらう-RECEIVE_FAVOR   (OBJE RELN)
on <obje reln> てもらう-RECEIVE_FAVOR in :iF :rEDUCE :type :genera
   "させる-PERMISSIVE + てもらう-RECEIVE_FAVOR --> PROMISE"
   in= [[reln UNKNOWN-IFT]
        [agen ?sp]
        [recp ?hr]
        [obje [[reln てもらう-RECEIVE_FAVOR]
               [agen ?AGEN]
               [recp ?recp]
               [obje [[reln させる-PERMISSIVE]
                      [agen ?recp]
                      [recp ?recp2]
                      [obje ?obje]
                      ?rest1]]
               ?rest2]]]

   set ?obje2 to ?obje
   add ?rest2 to ?obje2

  out= [[reln PROMISE]
        [agen ?sp]
        [recp ?hr]
        [obje ?obje2]]
end
```

規則 6

```
DEFN284        登録用紙-1                    (RESTR RELN)
on <restr reln> 登録用紙-1 in :pHASE :j-E :tYPE :default
   in= [[PARM !x[]]
        [RESTR [[RELN 登録用紙-1]
                [ENTITY !x]]]]
   out= [[PARM !x[]]
         [RESTR [[RELN registration_form-IDIOM-1]
                 [ENTITY !x]]]
         [INDEX [[NUMBER SING]
                 [GENDER NEUT]
                 [DEFINT NULL]
                 [PERSON THIRD]]]]

end
```

規則 7

```
DEFV046        送る-1                        (RELN)
on <reln> 送る-1 in :pHASE :j-E :type :default
   in= [[reln 送る-1]
        [AGEN ?AGEN]
        [RECP ?RECP]
        [OBJE ?OBJE]
        ?rest]

   out= [[reln send-V-1]
         [AGEN ?AGEN]
         [RECP ?RECP]
         [OBJE ?OBJE]
         ?rest]
end
```

以下に規則のトレース結果を示す。

```
;;; 入力文：登録用紙は至急送らせていただきます。
;;; ------------ Transfer Input ------------
 [[SEM [[RELN てもらう-RECEIVE_FAVOR]
        [ASPT UNRL]
        [AGEN !X8[]]
        [RECP !X9[]]
        [OBJE [[RELN させる-PERMISSIVE]
               [AGEN !X9]
               [RECP !X10[]]
               [OBJE !X11[[RELN 送る-1]
                          [AGEN !X10]
                          [RECP []]
                          [OBJE !X7[[PARM !X6[]]
                                    [RESTR [[RELN 登録用紙-1]
                                            [ENTITY !X6]]]]]
                          [MANN [[PARM !X5[]]
                                 [RESTR [[RELN 至急-1]
                                         [ENTITY !X5]]]]]]]]]]]
  [PRAG [[RESTR [[IN [[FIRST [[RELN POLITE]
                              [AGEN !X2[[LABEL *SPEAKER*]]]
                              [RECP !X1[[LABEL *HEARER*]]]]]
                     [REST [[FIRST [[RELN EMPATHY-DEGREE]
                                    [LESS !X9]
                                    [MORE !X8]]]
                            [REST [[FIRST [[RELN POLITE]
                                           [AGEN !X2]
                                           [RECP !X1]]]
                                   [REST !X3[]]]]]]]]
                 [OUT !X3]]]
         [ASPE [[IN !X4[]]
                [OUT !X4]]]
         [HEARER !X1]
         [PRSP-TERMS [[IN []]
                      [OUT []]]]
         [SPEAKER !X2]
         [TOPIC [[IN [[FIRST [[FOCUS !X7]
                              [SCOPE !X11]
                              [TOPIC-MOD WA]]]
                      [REST []]]]
                 [OUT []]]]]]

;;; | 1[1]:> MAINRULE
;;; Apply rewriting rule ID:MAINRULE.
```

```
;;; Input Feature structure is following.
 [[SEM [[RELN てもらう-RECEIVE_FAVOR]
        [ASPT UNRL]
        [AGEN !X8[]]
        [RECP !X9[]]
        [OBJE [[RELN させる-PERMISSIVE]
               [AGEN !X9]
               [RECP !X10[]]
               [OBJE !X11[[RELN 送る-1]
                          [AGEN !X10]
                          [RECP []]
                          [OBJE !X7[[PARM !X6[]]
                                    [RESTR [[RELN 登録用紙-1]
                                            [ENTITY !X6]]]]]
                          [MANN [[PARM !X5[]]
                                 [RESTR [[RELN 至急-1]
                                         [ENTITY !X5]]]]]]]]]]]]
  [PRAG [[RESTR [[IN [[FIRST [[RELN POLITE]
                              [AGEN !X2[[LABEL *SPEAKER*]]]
                              [RECP !X1[[LABEL *HEARER*]]]]]
                     [REST [[FIRST [[RELN EMPATHY-DEGREE]
                                    [LESS !X9]
                                    [MORE !X8]]]
                            [REST [[FIRST [[RELN POLITE]
                                           [AGEN !X2]
                                           [RECP !X1]]]
                                   [REST !X3[]]]]]]]]
                 [OUT !X3]]]
         [ASPE [[IN !X4[]]
                [OUT !X4]]]
         [HEARER !X1]
         [PRSP-TERMS [[IN []]
                      [OUT []]]]
         [SPEAKER !X2]
         [TOPIC [[IN [[FIRST [[FOCUS !X7]
                              [SCOPE !X11]
                              [TOPIC-MOD WA]]]
                      [REST []]]]
                 [OUT []]]]]]

             ;;;          ;;;
             ;;;   中略   ;;;
             ;;;          ;;;
```

```
;;; |   | 2[1]:> IFT009
;;; Apply rewriting rule ID:IFT009.
;;; Input Feature structure is following.
 [[RELN UNKNOWN-IFT]
  [AGEN !X2[[LABEL *SPEAKER*]]]
  [RECP !X1[[LABEL *HEARER*]]]
  [OBJE [[RELN てもらう-RECEIVE_FAVOR]
         [ASPT UNRL]
         [AGEN !X2]
         [RECP !X1]
         [OBJE [[RELN させる-PERMISSIVE]
                [AGEN !X1]
                [RECP !X2]
                [OBJE [[RELN 送る-1]
                       [AGEN !X2]
                       [RECP !X1]
                       [OBJE [[PARM !X4[]]
                              [RESTR [[RELN 登録用紙-1]
                                      [ENTITY !X4]]]]]
                       [MANN [[PARM !X3[]]
                              [RESTR [[RELN 至急-1]
                                      [ENTITY !X3]]]]]]]]]]]

;;; |   | 2[1]:< IFT009 Success
;;; Success to apply rewriting rule ID:IFT009
;;; There is one result.
;;; The first rewrited Feature Structure is following
 [[RELN PROMISE]
  [AGEN !X4[[LABEL *SPEAKER*]]]
  [RECP !X3[[LABEL *HEARER*]]]
  [OBJE [[RELN 送る-1]
         [ASPT UNRL]
         [AGEN !X4]
         [RECP !X3]
         [OBJE [[PARM !X2[]]
                [RESTR [[RELN 登録用紙-1]
                        [ENTITY !X2]]]]]
         [MANN [[PARM !X1[]]
                [RESTR [[RELN 至急-1]
                        [ENTITY !X1]]]]]]]]


             ;;;          ;;;
             ;;;   中略   ;;;
```

```
              ;;;          ;;;

;;; |   | 2[1]:> DEFV046
;;; Apply rewriting rule ID:DEFV046.
;;; Input Feature structure is following.
 [[RELN 送る-1]
  [ASPT UNRL]
  [AGEN [[LABEL *SPEAKER*]]]
  [RECP [[LABEL *HEARER*]]]
  [OBJE [[PARM !X2[]]
         [RESTR [[RELN 登録用紙-1]
                 [ENTITY !X2]]]]]
  [MANN [[PARM !X1[]]
         [RESTR [[RELN 至急-1]
                 [ENTITY !X1]]]]]]

;;; |   | 2[1]:< DEFV046 Success
;;; Success to apply rewriting rule ID:DEFV046
;;; There is one result.
;;; The first rewrited Feature Structure is following
 [[RELN SEND-V-1]
  [ASPT UNRL]
  [AGEN [[LABEL *SPEAKER*]]]
  [RECP [[LABEL *HEARER*]]]
  [OBJE [[PARM !X2[]]
         [RESTR [[RELN 登録用紙-1]
                 [ENTITY !X2]]]]]
  [MANN [[PARM !X1[]]
         [RESTR [[RELN 至急-1]
                 [ENTITY !X1]]]]]]
;;; |   | 2[1]:> DEFN284
;;; Apply rewriting rule ID:DEFN284.
;;; Input Feature structure is following.
 [[PARM !X1[]]
  [RESTR [[RELN 登録用紙-1]
          [ENTITY !X1]]]]

;;; |   | 2[1]:< DEFN284 Success
;;; Success to apply rewriting rule ID:DEFN284
;;; There is one result.
;;; The first rewrited Feature Structure is following
 [[PARM !X1[]]
  [RESTR [[RELN REGISTRATION_FORM-IDIOM-1]
          [ENTITY !X1]]]
```

```
  [INDEX [[DEFINT NULL]
          [GENDER NEUT]
          [NUMBER SING]
          [PERSON THIRD]]]]

;;; | 1[1]:< MAINRULE Success
;;; Success to apply rewriting rule ID:MAINRULE
;;; There is one result.
;;; The first rewrited Feature Structure is following
 [[SEM [[RELN PROMISE]
        [AGEN !X3[[LABEL *SPEAKER*]]]
        [RECP !X2[[LABEL *HEARER*]]]
        [OBJE [[RELN SEND-V-1]
               [ASPT UNRL]
               [TENSE FUTURE]
               [AGEN !X3]
               [RECP !X2]
               [OBJE [[PARM !X5[]]
                      [RESTR [[RELN REGISTRATION_FORM-IDIOM-1]
                              [ENTITY !X5]]]]]
               [MANN [[PARM !X4[]]
                      [RESTR [[RELN 至急-1]
                              [ENTITY !X4]]]]]
               [SEM-ASPE UNREAL]]]]]
  [PRAG [[RESTR [[IN [[FIRST [[RELN POLITE]]]
                     [REST [[FIRST [[RELN EMPATHY-DEGREE]
                                    [LESS !X2]
                                    [MORE !X3]]]
                            [REST [[FIRST [[RELN POLITE]]]
                                   [REST !X1[]]]]]]]]]
                [OUT !X1]]]
         [BENEFIT HEARER-SIDE]
         [HEARER !X2]
         [INTENTION OFFER]
         [PRSP-TERMS [[PRSP-MOD NULL]]]
         [SPEAKER !X3]
         [TOPIC [[TOPIC-MOD WA]]]]]]

;;;=========== Transfer Result ===========
 [[SEM [[RELN PROMISE]
        [AGEN !X1[[LABEL *SPEAKER*]]]
        [RECP !X2[[LABEL *HEARER*]]]
        [OBJE [[RELN SEND-V-1]
               [ASPT UNRL]
```

```
                    [TENSE FUTURE]
                    [AGEN !X1]
                    [RECP !X2]
                    [OBJE [[PARM !X3[]]
                           [RESTR [[RELN REGISTRATION_FORM-IDIOM-1]
                                   [ENTITY !X3]]]]]
                    [MANN [[PARM !X4[]]
                           [RESTR [[RELN 至急-1]
                                   [ENTITY !X4]]]]]
                    [SEM-ASPE UNREAL]]]]]
  [PRAG [[RESTR [[IN [[FIRST [[RELN POLITE]]]
                          [REST [[FIRST [[RELN EMPATHY-DEGREE]
                                         [LESS !X2]
                                         [MORE !X1]]]
                                 [REST [[FIRST [[RELN POLITE]]]
                                        [REST !X5[]]]]]]]]]
                     [OUT !X5]]]
          [BENEFIT HEARER-SIDE]
          [HEARER !X2]
          [INTENTION OFFER]
          [PRSP-TERMS [[PRSP-MOD NULL]]]
          [SPEAKER !X1]
          [TOPIC [[TOPIC-MOD WA]]]]]]

;;; ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

(#<Structure RWS::NODE C0AA8E>)
```

さくいん

参考文献

[1] 長谷川敏郎：素性構造書き換えシステムマニュアル, *ATR Technical Report*, TR-I-0093, 1989.

[2] 長谷川敏郎：素性構造書き換えシステムマニュアル(改訂版), *ATR Technical Report*, TR-I-0187, 1990.

[3] 長谷川敏郎：素性構造書き換えシステムのSL-TRANS変換過程への適用, 人工知能学会全国大会, 1990.

[4] 鈴木雅実・関倫彦：日英変換処理規則解説書, ATR Technical Report, TR-I-0329, 1993.

[5] 鈴木雅実・関倫彦：ATR音声言語翻訳実験システムASURAにおける日英変換処理の現状と課題, 電子情報通信学会技術研究報告, NLC92-58, 1993.

[6] Mark Seligman: A Japanese-German Transfer Component for ASURA, *ATR Technical Report*, TR-I-0365, 1993.

[7] 古崎博久・鈴木雅実：素性構造書き換えシステムを利用した日英変換処理の高速化, 情報処理学会第46回全国大会 5B-6, 1993.