TR-I-0187

素性構造書き換えシステムマニュアル
(改定版)
Feature Structure Rewriting System Manual
(Revised Version)

長谷川 敏郎
Toshiro HASEGAWA

1990年10月28日

概要

自動翻訳システムにおける意味構造変換を行なうための基本的な機構として素性構造を対象とした書き換えシステムを作成した。本書き換えシステムでは書き換え規則により素性構造の書き換えをおこなう。本報告では、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述べる。

ATR自動翻訳電話研究所
ATR Interpretin Telephony Reserach Laboratories

目次

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の述べたものである。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章では規則の例をもちいて書き換え規則とはどのようなものか簡単に説明する。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説明する。7章、8章、9章で書き換え規則の記述(シンタックス)について説明する。完全なシンタックスは付録Aに、システムの実行方法については付録Bに記述する。

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の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                    素性構造 2
      [[reln 送る]                  [[reln send]
       [agen *speaker*]              [agen *speaker*]
       [recp *hearer*]               [recp *hearer*]
       [obje *登録用紙*]]            [obje *登録用紙*]]
```

規則本体のin= [...]は入力の素性構造のパタン記述であり、入力素性構造パタンと呼ぶ。入力素性構造パタンは入力素性構造とパタンマッチングによって照合さ

れ、パタンマッチングが成功すれば書き換えシステムは規則の本体を実行する。入力が入力素性構造パタ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ンのパタ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ン(出力素性構造パタン)である。規則1は入力素性構造をこの素性構造パタンで書き換える。出力素性構造パタ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いおり、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回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 []]]]]]]
```

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。

---

[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。

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が日英の言語間で変換をし、タイプは一般規則であることが分かる。書き換え環境が: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適用可能になる。

---

[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。

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**書き換え呼びだし**(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。

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

[3]正確にはnode構造体のvalue値

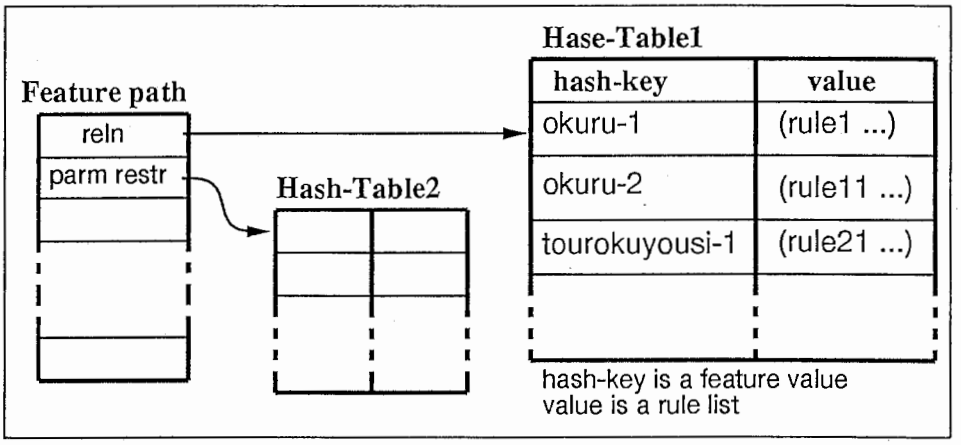

図 1: ルールベースの構造

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、しばしば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。

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。*conntrol*に:LOOPの指定がない時には、規則の検索と適用は一度しか行なわれない。*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]。

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
```

[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)を生成するおおもとの規則である。規則:mainはLISP関数transをにより呼び出される。

書き換え規則13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
- 入力素性構造パタンのパタンマッチングの失敗(9.1章参照)
- 出力素性構造パタンの生成の失敗(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返却値を記述しないと、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

[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。

を終了しないと、規則内部で行なわれた処理はすべての無効になり、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きない。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
```

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きない。

7 データ構造

7.1 素性構造パタン

素性構造パタン(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ンは“[”で始まり、“]”で終る。complexタイプの素性構造パタンの記述中には、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ンの例を示す。

素性構造 8

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

素性構造パタン8は変数を含んだ素性構造パタンであり、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。素性構造パタン中の変数が値を持っている時には、素性構造パタ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。

[6]素性構造パタ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。

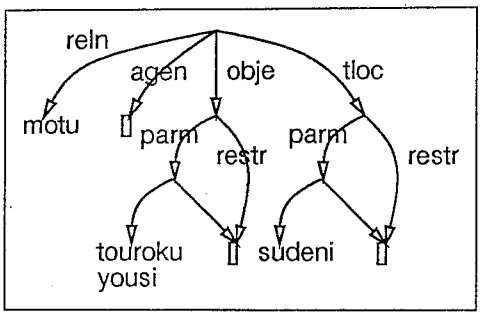

図 2: 素性構造のグラフ表示

素性構造 9

```
(complex タイプの素性構造パタンの例)
[[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7.4参照)。

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、タイプシステムのKBで定義されたより詳細なタイプを素性構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

[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。

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ンマッチング

素性構造と素性構造パタ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、パタンマッチングは失敗する。素性構造パタン中に同じ変数が二度以上現れた時は、パタ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]]
```

素性構造パタン10と素性構造11のパタ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、パタンマッチングは成功する。パタ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、素性構造パタン10と素性構造12のパタンマッチングでは、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、パタンマッチングは失敗する。パタ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

[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。

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ンと素性構造13のパタンマッチングは成功する。パタ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。

```
?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ンには素性tlocがない)ため、パタンマッチングは失敗する。

[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ンとのマッチングをとるためにもっぱら使用される。

素性のコンフリクト

rest variableを含む素性構造パタンによる素性構造の生成では、素性のコンフリクト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、コンテクスト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

[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。

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

素性構造パタンにパス修飾子path modifierを付けることにより、素性構造パタンのパタ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ンに先だって記述される。

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]]]]
```

素性構造パタン

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

はパタ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。[11]

7.3 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ンのパタ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。

7.4 タグ

タグは素性構造パタン記述中で、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は素性構造パタン中で定義され、参照範囲は素性構造パタン中に限定される。global tagは素性構造パタン中での振舞いは 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 をもちいて参照できる。

[11]修飾子の詳細については、TR-I-0093「素性構造書き換えシステムマニュアル」(長谷川)が詳しい。

global tag は素性構造パタン中ではlocal tagと同様に素性構造パタ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ンと素性構造のパタ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ンの例を挙げる。local tagの例については、7.1章参照。

規則 20

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

規則 21

```
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20を素性構造18に適用すると、global tag を含んだ素性構造パタンと入力素性構造18とのパタンマッチングが行なわれる。global tag“@tag”には、素性構造パタ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20の適用の結果、素性構造19が得られる。また、規則21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20の書き換えの効果(処理内容)は規則21と等価であるが、規則20の方が規則21より制約が強い。

7.5 type

---

type of *FeatureStructure*

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

7.6 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]]]]
```

7.7 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。

7.8 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。

7.9 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。

8 演算子

8.1 基本演算子

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

---

[a]Not implemented yet.(1990 Oct.)

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==は素性構造*FeatureStructure1*と*FeatureStructure2*の単一化を行なう。単一化結果はシステム変数?itで参照することができる。単一化に失敗した場合には変数?itの値はemptyである。

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 22

```
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 23

```
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22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23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、infmann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、[12]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[13]。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。

---

[12]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。
[13]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ン *FeatueStructure* と構造的に等しいかパタ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 24
```
on <x> y
   in=  [[x y]
         [z ?z]]
   --> ?z
   out= ?z
end
```

規則 25
```
on <a> b
  in=  [[a b]]
  out= [[a b1]]
end
```

規則 26
```
on <a> b
  in=  [[a b]]
  out= [[a b2]]
end
```

規則 27
```
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24を適用したとする。規則24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25、規則26、規則27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[14]。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。

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

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

J:
```

[14]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。

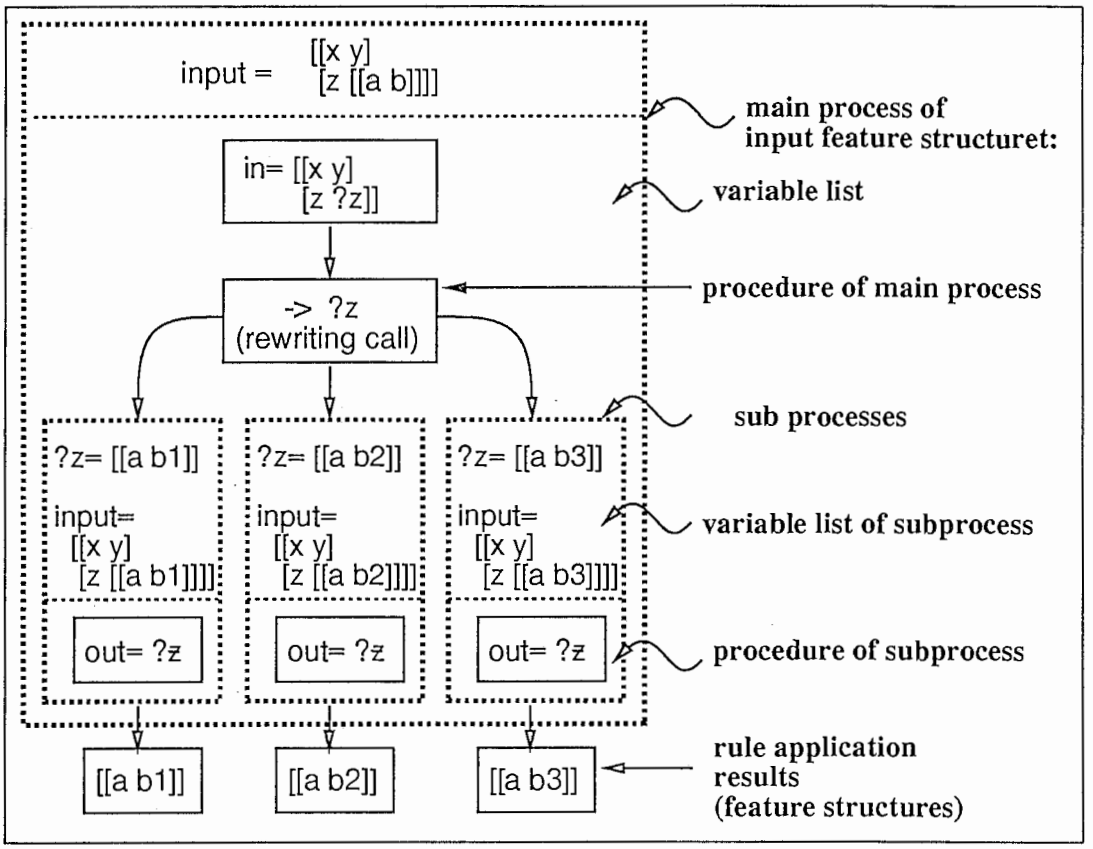

図 3: 書き換え呼び出しとサブプロセスの生成

```
----------
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
(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*
endswicth

---

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)。

10 タイプシステム

15

10.1 タイプシステム

タイプシステムはトップ(⊤)とボトム(⊥)を含むタイプシンボルの半順序集合Pとして定義される。タイプシンボルの半順序集合は、ラティスとして表現される。タイプシンボルの集合Pを仮に

> P = {⊤, ⊥, variable, atomic, complex, leaf, object, concrete-object, creature, human, teacher, abstruct-object, intellectural-object, writing, registration-form, space}

と定義し、タイプ間に適当な順序関係を与えることにより、図4のような階層構造を形成する。図4において、complex, object, human, writing 間には、

> complex ≤ object ≤ human
> object ≤ writing

という順序関係が与えられている。一方、humanとwritingの間には順序関係は存在しない。タイプシステムにおける基本的な演算には、**GLB(Greatest Lower Bound)**と**LUB (Least Upper Bound)**がある。GLBは単一化演算(unification)に相当し、LUBは一般化演算(generalization)に相当する。

GLB: 任意の二つのタイプシンボル(x, y)において、以下の式を満たすタイプシンボルzをGreatest Lower Boundという。

$\exists x \in P, \exists y \in P,\ z \leq x,\ z \leq y$

$\forall u \in P$ iff $u \leq x,\ u \leq y;\ u \leq z$

LUB: 任意の二つのタイプシンボル(x, y)において、以下の式を満たすタイプシンボルzをLeast Upper Boundという。

$\exists x \in P, \exists y \in P,\ z \geq x,\ z \geq y$

$\forall u \in P$ iff $u \geq x,\ u \geq y;\ u \geq z$

---

[15] not implemented yet (1990 Oct).

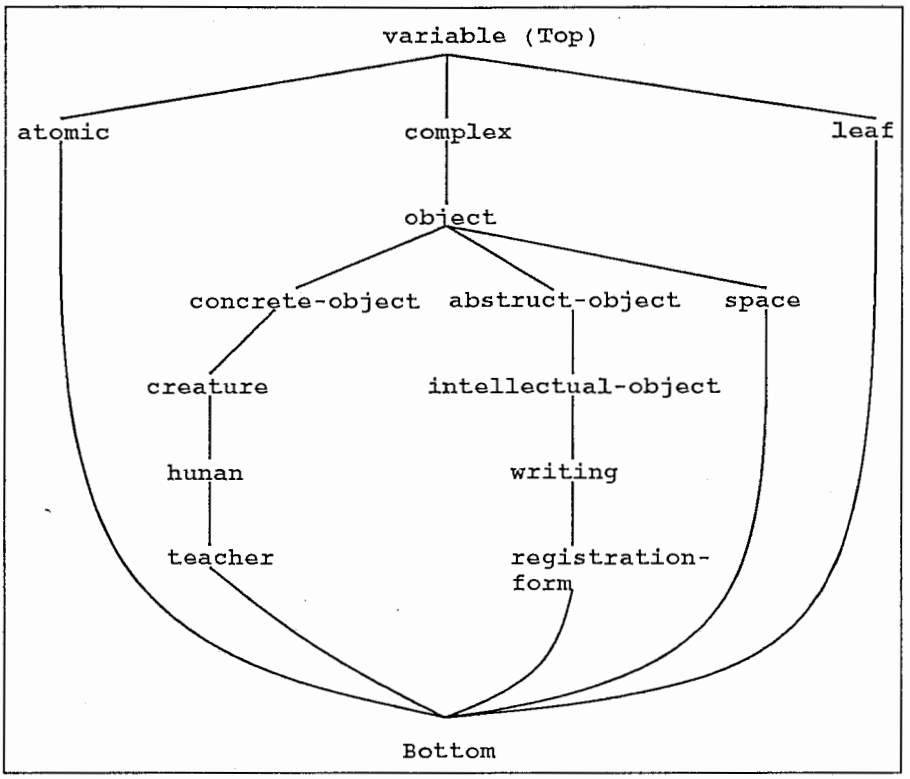
variable (Top)
atomic
complex
leaf
object
concrete-object
abstruct-object
space
creature
intellectual-object
hunan
writing
teacher
registration-
form
Bottom

図 4: 知識ベース

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順序関係を定義する。[16]

---

deffstype *type* &optional *subtypes*

---

deffstypeはタイプシンボル*type*を新たに生成し、*type*とタイプ*subtypes*の順序関係を定義する。タイプ*type*は*subtypes*のスーパータイプ($type \geq subtype$)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。スーパータイプが明示的に定義されていないタイプのスーパータイプはtop($\top$)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。[17]

```
(deffstype variable)

(deffstype variable atomic complex leaf)

(deffstype complex 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,

[16]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付録B.8参照。

[17]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。

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,概念体系)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。

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“(,)”と“|”を用いて記述する。

素性構造 28

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

素性構造パタン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ンマッチング

タイプ付き素性構造パタンと素性構造のマッチングでは、まず、素性構造パタンと入力素性構造とのタイプのチェックが行なわれる。タイプのチェックでは、素性構造パタンと入力素性構造のGLB(Greatest Lower Bound)が計算(meet operation)され、GLBの結果得られたタイプと入力素性構造のタイプが等しければ、入力素性構造は素性構造パタンのタイプ制約を満たしたとする。パタ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 28

```
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28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蔵である。規則28の入力素性構造パタンは素性構造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 29

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

規則29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29が適用されると、入力素性構造と素性構造パタンのパタンマッチングが行なわれ、パタ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29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

B 書き換えシステムの関数

B.1 書き換えシステムのロード

書き換えシステムを使用するには、LISPを立ち上げ、まず、ファイル$RWS/rws.lispをロードする。[18]書き換えシステムの関数はすべてRWSパッケージに入っている。

B.2 書き換え規則の適用

---

trans *fs*

素性構造*fs*にメタ規則:mainを適用し、素性構造のリストを返す。

---

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(C.1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。

B.3 素性構造の入出力

---

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

---

[18]1990年11月2日現在$RWSは/usr/share/src/atr/rwsである。

read-fs は素性構造パタン *body* の記述から素性構造を生成し返す。read-fs-from-string は文字列 *string* を読み、素性構造を生成し返す。read-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す。only-fs に nil 以外を指定すると、ファイルに記述されている素性構造のだけのリストが返える。only-fs に nil を指定すると素性構造以外のオブジェクトも含んだリストが返える。素性構造記述のシンタックスは素性構造パタンの記述に準ずる。

B.4 規則の定義と削除

---

defrwrule &body *form*
defrwschema *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。defrwschema で定義した規則は、さらに *rw-rule-schema-type-database** にも登録される。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。

---

remove-all-rw-rules
remove-rw-rule *atomic-fs* &optional *path name*

---

remove-all-rw-rule 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。remove-rw-rule は *atomic-fs* で指定された規則をルールベースから削除する。*path* は素性パスであり、*path* を指定すると素性 *path* パスに定義してある規則のみを削除する。*path* の指定がないと、*atomic-fs* に定義されているすべての規則を削除する。システムは規則のロード時に固有の名前を規則に与える。*name* は、システムによって与えられた規則名を指定する。*name* を用いることにより、特定の規則を削除できる。システムによって与えられた規則名は、print-rw-rule を参照。

B.5 規則の保存

---

save-rw-schemas *filename type*
save-index-and-rules *index-filename rule-filename*

---

save-rw-schemas は、defrwschema によって *rw-rule-schema-type-database* に登録された規則で、*type* で指定された規則をファイル *filename* に出力する。規則は defrwschema で指定された *sort-key* にしたがってソートして出力する。

save-index-and-rules 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、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-rules の実行時に、*rules-filename* に相対パスが与えられると、インデックスを用いたロード時にも相対パスで指定されたファイルをオープンするので、システムを実行している current working directory がどこであるかを注意する必要がある。

以下に、規則のセーブの例を示す。

```
> (rws::defrwschema 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-rules)  ..........規則の表示

送る:
送る-1442
Parameter                    Value
---------                    -----
:PHASE                       :J-E
:TYPE                        :GENERAL

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

0
NIL
> (rws::remove-all-rw-rules)  ..........規則の削除
NIL
```

```
> (rws::print-all-rw-rules) .....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"/usr/hase/doc/nadine/rws/index-file"
> (rws::print-all-rw-rules) .....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
>
```

B.6 規則の表示

---

print-rw-rule *atomic-fs* &optional *path* &key :print-document :print-text
print-all-rw-rules &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。*atomic-fs*は規則の素性パスの値である素性値を指定する。print-rw-ruleは、*path*で示した素性パスに定義されいる規則のなかで、素性値*atomic-fs*に定義された規則を表示する。*path*を省略すると、*atomic-fs*に定義されているすべての規則を表示する。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。print-rw-ruleおよびprint-all-rw-rulesは、書き換えシステムが規則に与えた固有の名前も表示する。この規則固有の名前は、規則のトレース時に表示されたり、規則の削除(remove-rw-rule)に参照されたりする。

```
(rws::print-rw-rule '登録用紙 '(reln) :print-text t)

登録用紙:
----------
登録用紙-1.....規則に付けられた固有の名前
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
```

B.7 規則のトレース

---

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、以下の七つ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あるいは RETURN を入力する。RETURN は [ ] 内で表示されているデフォルト値が取られる。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)
```

- 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(D参照)。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。

B.8 タイプシステム

---

in-kb *kb-name*
find-kb *kb-name*
find-kbs

関数in-kbは、現在の知識ベースを*kb-name*で名付けられた知識ベースに設定する。現在の知識ベースは大域変数*KB*にセットされる。関数find-kbは*kb-name*に名付けられた知識ベースを返す。関数find-kbsは現在定義されているすべての知識ベースのリストを返す。

---

deffstype *type* &optional *subtype*
alias-type *alias-type-name type*

関数deffstypeは現在の知識ベースにタイプシンボル*type*を定義する。オプションである*subtype*が与えられると、*type*の直下のサブタイプとして*subtype*が

定義される。*subtype*が指定されない時には、*type*はbottom(⊥)以外のサブタイプを持たない。*subtype*が知識ベースに定義されていないときには、新たに*subtype*も定義される。

関数alias-typeは、タイプシンボル*type*に別名*alias-type-name*を与える。

---

print-kb &optional *kb-name stream*
print-subtype *type* &optional *KB*
print-subtypes *type* &optional *KB*

関数print-kbは現在の知識ベースを表示する。*kb-name*が指定されると、*kb-name*の知識ベースを表示する。出力のストリーム*stream*の既定値は*standard-output*である。関数print-subtypeはタイプシンボル*type*の直下に定義されたサブタイプを表示する。関数print-subtypesはタイプシンボル*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。これは、知識ベースの部分構造を表示することになる。

B.9 大域変数

- *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。

C コンパイラ

書き換え規則のコンパイラは書き換え規則からLISPプログラムを生成する。コンパイラは書き換え規則30を以下のLISPプログラムに変換する。

規則 **30**

```
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関数について説明する。

---

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の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 ...))は素性構造のパタンマッチングである。素性構造パタンは規則実行時にマクロmake-fsで動的に生成される。マクロ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ンとのパタンマッチングは、マクロFS-MATCHにより行なわれる。FS-MATCHがパタ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LIPSコードで参照される関数の一覧を示す。

---

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。

---

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* [a]
fs-unify *fs1 fs2* [b]
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*

---

[a] not implemented yet.
[b] Not implemented yet.

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-unifyは素性構造*fs1*と*fs2*との単一化を行なう。

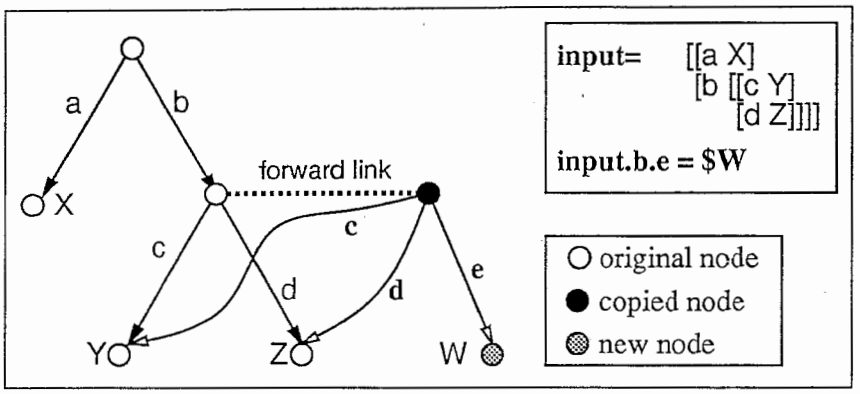

図 5: 素性の追加における素性構造のコピーとforwarding

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5参照)。

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。

C.1 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。:atomicタイプ以外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31を素性構造30に適用すると規則31内の書き換え呼び出しによって規則3233が適用される。規則31により最終的に30は二つの素性構造31, 32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6になる。図6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31、*p2*でフォワードされた素性構造は素性構造32である。

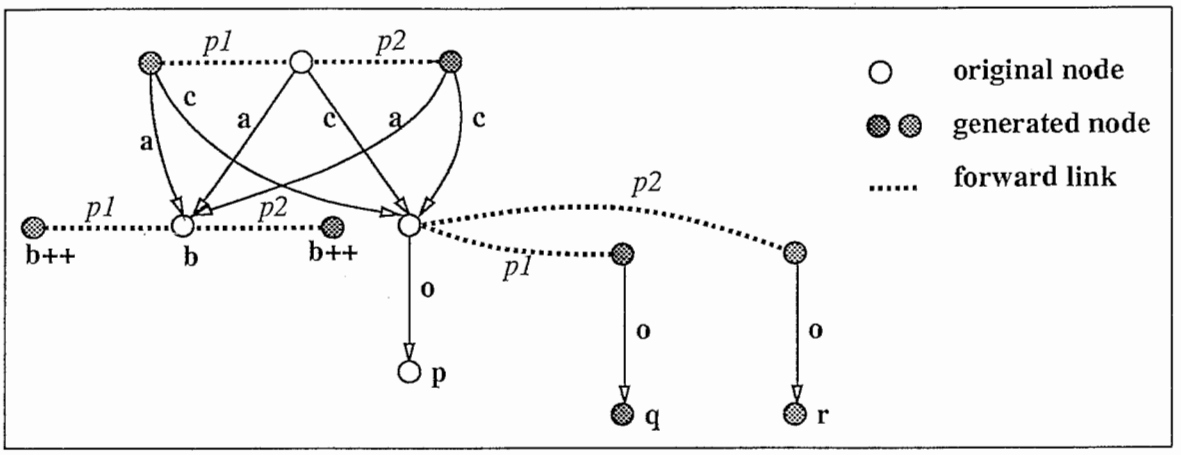

図 6: forwardingによる素性構造の書き換え

規則 31

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

規則 32

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

規則 33

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

素性構造 30

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

素性構造 31

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

素性構造 32

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

D append Example

素性構造で表現された二つのリストのappendをおこなう書き換え規則を例示する。appendは二つの規則からなる。書き換え規則34は第一引数のリストが空だと第二引数のリストを書き換え結果として返す。書き換え規則35は第一引数の先頭要素を取り除いたリストを第二引数のリストとappendした結果に第一引数をappendする。appendはLISPプログラムで記述すると以下のように記述できるが、規則34は(1)の式に相当し規則35は(2)の式に相当する。

```
(defun append (lst1 lst2)
   (cond ((null lst1) lst2)                                    ;;(1)
         ((car lst1)
          (cons (car lst1) (append (cdr lst1) lst2)))          ;;(2)
```

規則 34

```
on <eval> append
   in= [[eval append]
        [arg1 []]
        [arg2 ?arg2]]
   out= ?arg2
end
```

規則 35

```
on <eval> append
   in= [[eval append]
        [arg1 [[first ?first]
               [rest ?rest]]]
        [arg2 ?arg2]]

   -> [[eval append]
       [arg1 ?rest]
       [arg2 ?arg2]]

   out= [[first ?first]
         [rest ?it]]
end
```

以下に規則のトレース結果を示す。

```
> (rws::print-rw-rule 'append '(eval) :print-text t)
----------
APPEND-53
Parameter                    Value
---------                    -----
```

```
:PHASE                          :J-E
:TYPE                           :GENERAL

on <eval> append
   in= [[eval append]
        [arg1 []]
        [arg2 ?arg2]]
   out= ?arg2
end

APPEND:
----------
APPEND-54
Parameter                       Value
---------                       -----
:PHASE                          :J-E
:TYPE                           :GENERAL

on <eval> append
   in= [[eval append]
        [arg1 [[first ?first]
               [rest ?rest]]]
        [arg2 ?arg2]]
   -> [[eval append]
       [arg1 ?rest]
       [arg2 ?arg2]]
   out= [[first ?first]
         [rest ?it]]
end

> (rws::transfer fs)
;;; ------------ Transfer Input ------------
 [[ARG1 [[FIRST A]
         [REST [[FIRST B]
                [REST []]]]]]
  [ARG2 [[FIRST C]
         [REST [[FIRST D]
                [REST []]]]]]
  [EVAL APPEND]]
;;; | 1[1]:> APPEND-54
```

```
;;; Apply rewriting rule ID:APPEND-54.
;;; Input Feature structure is following.
 [[ARG1 [[FIRST A]
         [REST [[FIRST B]
                [REST []]]]]]
  [ARG2 [[FIRST C]
         [REST [[FIRST D]
                [REST []]]]]]
  [EVAL APPEND]]
;;;  1 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
;;; CALLED subFS Rewriting call on APPEND.
;;; |  | 2[2]:> APPEND-54
;;; Apply rewriting rule ID:APPEND-54.
;;; Input Feature structure is following.
 [[ARG1 [[FIRST B]
         [REST []]]]
  [ARG2 [[FIRST C]
         [REST [[FIRST D]
                [REST []]]]]]
  [EVAL APPEND]]
;;;   2 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
;;; CALLED subFS Rewriting call on APPEND.
;;; |  |  | 3[3]:> APPEND-54
;;; Apply rewriting rule ID:APPEND-54.
;;; Input Feature structure is following.
 [[ARG1 []]
  [ARG2 [[FIRST C]
         [REST [[FIRST D]
                [REST []]]]]]
  [EVAL APPEND]]
;;; |  |  | 3[3]:< APPEND-54 Fail
;;; Fail to apply rewriting rule ID:APPEND-54
;;;
;;; |  |  | 3[1]:> APPEND-53
;;; Apply rewriting rule ID:APPEND-53.
;;; Input Feature structure is following.
 [[ARG1 []]
  [ARG2 [[FIRST C]
         [REST [[FIRST D]
                [REST []]]]]]
```

```
  [EVAL APPEND]]
;;; |  |  | 3[1]:< APPEND-53 Success
;;; Success to apply rewriting rule ID:APPEND-53
;;; There is one result.
;;; The first rewrited Feature Structure is following
 [[FIRST C]
  [REST [[FIRST D]
         [REST []]]]]
;;;   2 <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
;;; Terminate subFS Rewriting CALL on APPEND.
;;; |  | 2[2]:< APPEND-54 Success
;;; Success to apply rewriting rule ID:APPEND-54
;;; There is one result.
;;; The first rewrited Feature Structure is following
 [[FIRST B]
  [REST [[FIRST C]
         [REST [[FIRST D]
                [REST []]]]]]
;;; |  | 2[1]:> APPEND-53
;;; Apply rewriting rule ID:APPEND-53.
;;; Input Feature structure is following.
 [[ARG1 [[FIRST B]
         [REST []]]]
  [ARG2 [[FIRST C]
         [REST [[FIRST D]
                [REST []]]]]
  [EVAL APPEND]]
;;; |  | 2[1]:< APPEND-53 Fail
;;; Fail to apply rewriting rule ID:APPEND-53
;;;
;;;  1 <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
;;; Terminate subFS Rewriting CALL on APPEND.
;;; | 1[1]:< APPEND-54 Success
;;; Success to apply rewriting rule ID:APPEND-54
;;; There is one result.
;;; The first rewrited Feature Structure is following
 [[FIRST A]
  [REST [[FIRST B]
         [REST [[FIRST C]
                [REST [[FIRST D]
```

```
                        [REST []]]]]]]]]
;;; | 1[1]:> APPEND-53
;;; Apply rewriting rule ID:APPEND-53.
;;; Input Feature structure is following.
 [[ARG1 [[FIRST A]
         [REST [[FIRST B]
                [REST []]]]]]
  [ARG2 [[FIRST C]
         [REST [[FIRST D]
                [REST []]]]]]
  [EVAL APPEND]]
;;; | 1[1]:< APPEND-53 Fail
;;; Fail to apply rewriting rule ID:APPEND-53
;;;


;;;=========== Transfer Result ============
 [[FIRST A]
  [REST [[FIRST B]
         [REST [[FIRST C]
                [REST [[FIRST D]
                       [REST []]]]]]]]]
;;; ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

(#<Structure RWS::NODE CB8B86>)
```

索引