TR-I-0093

素性構造書き換えシステムマニュアル
The Feature Structure Rewriting System Manual

長谷川　敏郎
Toshiro HASEGAWA

1989.8

概要

自動翻訳電話システムにおける意味構造変換を行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行う。本書き換えシステムの特徴は、書き換え規則の適用をパラメータ(書き換えタイプと書き換え方向を表現する)を用いて制御するところにある。本報告では、書き換えシステムにおける規則の記述方法や適用方法について述べる。

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

目次

はじめに

素性構造を入力とし、素性構造を出力する書き換えシステムを作成した。本報告では、作成した書き換えシステムの機能について説明する。

特徴

(1)パタンマッチングに基づく素性構造の書き換えシステムである
(2)素性構造の書き換えは書き換え規則を適用することによって行われ、書き換え規則は特定のタイプと書き換え方向が規定される
(3)書き換え規則間の結合は疎であり、モジュラリテイーが高い
(4)全ての可能な書き換え結果を出力する

以下に上記の特徴に関する説明を行う。

(1)パタンマッチングを基にした素性構造の書き換えシステム

機械翻訳システムにおける変換フェーズとして書き換えシステムを見た場合、大量の書き換え規則(文法)を記述する必要がある。したがって、入力素性構造の条件の記述や書き換えるべき構造の記述の容易性や可読性が文法の開発効率的に大きく影響する。また、個々の規則における柔軟な処理能力が要求される。

素性構造を書き換える規則の表現方法として、規則を記述的に表現する方法と、手続的に表現する方法が考えられる。規則を記述的に表現すると記述性や可読性は高まるが規則内における柔軟な処理や制御は困難になる。一方、手続的な表現では表現能力は十分であるが記述の容易性や可読性は低下するおそれがある。そこで、本書き換えシステムではパタンマッチングを書き換え規則の表現の基礎とし、パタンマッチング1の機能を手続的な表現に埋めこむという表現形式を取った。

(2)パラメータによる規則適用制約

一般にある素性構造に対して、種々の書き換えタイプや書き換え方向が存在する。これらの書き換え規則を一様に取り扱うことは、書き換え規則適用制御の面でも規則を管理する面でも好ましくない。本書き換えシステムでは、ある書き換え規則がどのようなタイプの書き換えを行い、入力の素性構造をどのように書き換えるかを、書き換え規則に対して定義する。ここでは、書き換えタイプをパラメータ、書き換え方向をパラメータ値として表現した。そして、書き換え規則とは独立して、入力素性構造に対して環境(パラメータとパタメータ値の集合)を定義し、規則の書き換えタイプと書き換え方向の定義(パラメータ定義)が環境において満たされている規則が入力素性構造に適用される。したがって、規則のパラメータ定義は、規則適用の制約として働く。

文脈情報情報などをパラメータとパラメータ値として環境として表現することにより、文脈に依存した素性構造の書き換えを行うことができる。また、書き換え規則の適用は、書き換え

1.入力素性構造の検査や生成パタンを生成するためにはパタンマッチングの他に、ユニフィケーションをもちいたメカニズムが考えられる。しかし、入力の制約記述である素性構造から入力の素性構造への素性の供給の問題により、今回はユニフィケーションのメカニズムは採用しなかった。

タイプ別に行われるように設計したので、書き換えタイプを一つのフェーズとして考えると、書き換えフェーズの設定や変更を容易に行うことができる。

(3)書き換え規則のモジュラリテイー

規則数が大量になると、規則の管理の問題が生じる。本システムでは、規則間の結合は疎であり、書き換え規則は特定の語彙に対して書き換えタイプ別に定義され、管理が比較的容易に行える。書き換え規則中から部分素性構造の書き換え呼び出しができるように設計したが、書き換え呼出しの際、全く制限なしに書き換え呼び出しを行うのでは一般的すぎるので、何らかの形で制限を加えたいといった要求が生じる。部分素性構造の書き換え呼び出しを行う際、特定の書き換えを行うために、部分素性構造に適用する規則(あるいは規則の集合)を陽に記述すると、呼び出す規則と呼び出される規則間に依存関係が生じ、規則の独立性は薄れる。本システムでは、部分素性構造の書き換え呼び出しを行う際、適用する規則を陽に指定することなく、どのように書き換えるべきかを、書き換えタイプと書き換え方向を指定することにより、特定の方向に書き換える。これによって、書き換え規則からの部分素性構造の呼び出しにおいても、規則間の独立性を保つことができる。

(4)曖昧性の扱い

素性構造の書き換えを行うにあたって、曖昧性を取り扱う方法として、バックトラックを行って処理する方法とパラレルに処理する方法とが考えられる。バックトラックを用いた制御方式では、全解探索を行うよりも計算量の面で効率的であるが、部分素性構造の書き換え呼び出しの記述の際、規則中にバックトラックのための手続を明示的に記述しなければならず、規則の記述が煩雑になる。一方、全解探索を行うメカニズムでは、書き換え呼び出しの際の記述は単純になるが、効率面での低下が予想される。ここでは、曖昧性が生じた時(複数の適用可能な規則が存在する時)、それらを並列に計算することを基本とし、ヒューリスティックスを用いて書き換え結果にプレファレンスを与えたり、規則の優先順位付けを行うようにする。

報告の範囲

本報告では、素性構造の書き換えシステムの機能と、書き換え規則の機能とシンタックス、および、書き換え規則の定義方法について述べる。本書き換えシステムは自動翻訳システムの意味構造変換モジュールを目指して作成したが、書き換え規則(変換規則)の構築方法やどのような書き換えタイプ(パラメータ)を用意すべきか等についてまでは言及しない。

本報告は次のような構成になっている。1章では、書き換えシステムで取り扱われるデータ構造について説明し、2章で、規則がどのような形式で記述され、どのように適用されるかを中心に、簡単にシステムの概略を述べる。3章で、書き換え規則の定義方法や、検索方ほ、適用方法について述べる。4章で、書き換えシステムにおける基本機能(primitive function)について述べ、5章で、書き換え規則のシンタックスや規則内における制御の詳細について述べる。

1.書き換えシステムの概要

素性構造の書き換えは定義された書き換え規則を素性構造に適用することにより行われる。ここでは、書き換え規則がどのような形式で定義され、適用されるかということを中心に、システムの概略を説明する。

典型的な書き換え規則は、以下の形式をしている。

(1.1) 書き換え規則の例

```
on 登録用紙
   in= [[reln 登録用紙]
        [obje []]]
   out= [[reln registration-form]
         [obje []]
end 登録用紙
```

書き換え規則(1.1)は、「登録用紙」に関する書き換え規則である。書き換え規則は"on"で始まり、次に何に関する書き換え規則であるかを示す規則名が続き、"end"と規則名で終了する。規則の本体はこれらの間に記述される。書き換え規則は素性構造のRELatioN素性の値(関係名)に対して定義され、書き換え規則(1.1)ではRELatioN素性の値が「登録用紙」である素性構造に対して適用される。

書き換え規則(1.1)の本体は、二つの部分からなっている。一つは、素性構造を検査する部分であり、キーワード"in="に続く素性構造のパタンと規則が適用された素性構造とが一致するれば、規則の残りの部分が実行される。素性構造とパタンが一致しないときには、規則適用は失敗となり、規則を終了する。もう一つの部分は、素性構造の書き換えパタンであり、素性構造がキーワード"out="に続くパタンで、規則が適用された素性構造がパタンン書き換えられる。したがって、素性構造(1.2)に、規則(1.1)が適用された結果、素性構造(1.3)が得られる。

(1.2)

```
[[reln 登録用紙]
 [obje []]
```

(1.3)

```
[[reln registration-form]
 [obje []]
```

システムは、入力素性構造をトップダウンに辿りながら、各部分素性構造の関係名を検索キーにして書き換え規則を検索し、規則が見つかれば部分素性構造に対して規則を適用する。以下、入力素性構造(1.4)に対して書き換え規則(1.5)、(1.6)、(1.7)が定義されている時、どのように書き換えが進んでゆくかを説明する。

素性構造(1.4)は、「分からない点があれば聞いて下さい」という発話の解析結果における意

(1.4)　入力素性構造

```
[[reln request]
 [agen !sp [[label *speaker*]]]
 [recp !hr [[label *hearer*]]]
 [obje [[reln 聞く-1]
        [expr !sp]
        [obje [[reln ある-1]
               [obje {「分からない点」の意味記述}]]]]]]
```

(1.4')　「分からない点」の意味記述

```
[[parm !x1[[parm !x2[]]
           [restr [[reln 点-1]
                   [obje !x2]]]]]
 [restr [[reln negate]
         [obje [[reln 分かる]
                [expr !hr]
                [obje !x1]]]]]]
```

味表現部分である。書き換え規則(1.5)、(1.6)、(1.7)は、それぞれ「聞く」、「ある」、「点」を関係名とする素性構造の書き換え規則であり、日本語の意味構造(素性構造)を対応する英語の意味構造(素性構造)に書き換える。

(1.5)　「聞く」の書き換え規則

```
on 聞く-1
   in= [[reln 聞く-1]
        [expr ?expriencer]
        [obje ?object]
   out= [[reln ask]
         [expr ?expriencer]
         [obje ?object]]
end 聞く-1
```

(1.6)　「ある」の書き換え規則

```
on ある
   in= [[reln ある-1]
        [obje ?object]
   out= [[reln be-1]
         [obje ?object]]
end ある
```

書き換えシステムが入力素性構造をトプダウンに辿ると、最初に出会う関係名は「request」である。ここでは、関係名「request」に定義されている規則はないので書き換えはおこらない。この素性構造はRELatioN素性以外に三つの素性を持っており、システムは各素性値を再帰的に処理する。AGENt素性およびRECPient素性の値である素性構造は、RELatiN素性を持たないので規則の検索は行われない。OBJEct素性の値である部分素性構造では、関係名「聞く」に関する書き換え規則(1.5)が定義されている。書き換え規則(1.5)中のマッチングパタンおよび生成(書き換え)パタンには二つの変数"?experiencer"と"?object"が記述されている。パタン

(1.7)　「点」に関する書き換え規則

```
on 点-1
   in= (restr obje) @分からない点
       [[parm !x1[[parm !x2[]]
                  [restr [[reln 点-1]
                          [obje !x2]]]]]
        [restr [[reln negate]
                [obje [[reln 分かる]
                       [expr ?expr]
                       [obje !x1]]]]]]

   @分からない点 = [[parm !X[]]
                    [restr [[reln question-1]
                            [obje !x]]]]
   return @分からない点
end 点-1
```

マッチングにおいて、変数は対応する任意の素性構造と一致することができ、パタンマッチングが成功すると、対応する部分素性構造が変数に代入される。また、生成パタンでは、変数に代入されている素性構造で変数が置き換えられる。

書き換え規則(1.5)を部分素性構造に適用することにより、入力素性構造(1.4)は(1.8)のように書き換えられる。

(1.8)　書き換え規則(1.5)の適用結果

```
[[reln request]
 [agen !sp [[label *speaker*]]]
 [recp !hr [[label *hearer*]]]
 [obje [[reln ask-1]
        [expr !sp]
        [obje [[reln ある-1]
               [obje {「分からない点」の意味記述}]]]]]]
```

次に、書き換えられた部分素性構造(関係名が「ask」である素性構造)の各素性値が再帰的に処理される。EXPeRiencer素性値は、関係名が「request]である素性構造のAGENt素性の値と共有されており、この部分素性構造は書き換えが既に試みられたので、ここでは、書き換え対象とはならない。OBJEct素性の値である部分素性構造には書き換え規則(1.6)が定義されており、書き換え規則(1.6)を関係名が「ある-1」である部分素性構造に適用することにより、素性構造(1.9)が得られる。

(1.9)　書き換え規則(1.6)の適用結果

```
[[reln request]
 [agen !sp [[label *speaker*]]]
 [recp !hr [[label *hearer*]]]
 [obje [[reln ask-1]
        [expr !sp]
        [obje [[reln be-1]
               [obje {「分からない点」の意味記述}]]]]]]
```

さらに、書き換えが続けられ、関係名が「点」である部分素性構造に辿りつく。この部分素性構造には、書き換え規則(1.7)が定義されており、適用が試みられる。この規則(1.7)は新らな記述をいくつか含んでいる。“in＝”に続く“(parm restr)”は修飾子であり、パタンマッチングが起動されると、素性構造を根の方向にむかってパスを辿り、パスの先の素性構造とマッチングパタンとのパタンマッチングが行われる。“@分からない点”はパタンマッチングが成功した時に、“@分からない点”という名前で修飾子で指定されたパスの先の素性構造にラベリングが行なわれ、後で、この名前で素性構造を参照することができる。現在着目しているのは関係名が「点」である部分素性構造であるから、そこからrestr、parmというパスを根の方向に辿った素性構造とマッチングパタンとののパタンマッチングは成功し、パタンマッチングの対象となった素性構造(「分からない点」の意味表現に対応する素性構造)に対してラベリングされる。書き換え規則中の次の文、“@分からない点 = ...”ではラベリングされた素性構造に対して書き換えを行っている。その結果、「分からない点」の意味表現に対応する素性構造は、「question」の意味表現に対応する素性構造(意味表現)に書き換えられる。最後の文では、規則の書き換え結果として、「question」の意味表現に対応する素性構造を返している(ラベリングされた素性構造はすでに書き換えられている)。この規則を適用した結果、素性構造(1.9)は(1.10)のようになる。

(1.10) 書き換え規則(1.7)の適用結果

```
[[reln request]
 [agen !sp [[label *speaker*]]]
 [recp !hr [[label *hearer*]]]
 [obje [[reln ask-1]
        [expr !sp]
        [obje [[reln be-1]
               [obje [[parm !x[]]
                      [restr [[reln question]
                              [obje !x]]]]]]]]]]
```

書き換え規則(1.7)には二つの操作が含まれている。一つは、「分からない点」という複雑な構造を持った名詞句を、単純な構造にまとめ上げる処理であり、もう一つは、単純化された日本語の意味構造を対応する英語の関係名に置き換える処理である。これらの処理を、一つの規則にまとめず、二つの書き換え規則(1.11)、(1.12)で記述することができる。書き換え規則(1.11)は、「分からない点」に対応する意味構造を「質問」に対応する意味構造に縮退させる書き換える規則であり、書き換え規則(1.12)は「質問」に対応する英語の意味構造「question」に書き換える規則である。書き換え規則(1.11)は、日本語の素性構造から日本語の素性構造への書き換えを行っており、明らかに他の書き換え規則とは性格が異なる。また、書き換え規則(1.11)は、書き換え規則(1.12)よりも先に適用されなければならない。

書き換え規則(1.11)が他の規則とタイプの異なる書き換え規則であることを明示するために、書き換え規則に書き換えタイプと書き換え方向を定義し、他の書き換え規則と区別する。書き換えタイプと書き換え方向の定義は書き換え規則(1.13)のように規則の先頭に、接頭辞“in”

(1.11) 「点」に関する書き換え規則

```
on 点-1
   in= (parm restr) @分からない点
       [[parm !x1[[parm !x2[]]
                  [restr [[reln 点-1]
                          [obje !x2]]]]]
        [restr [[reln negate]
                [obje [[reln 分かる]
                       [expr ?expr]
                       [obje !x1]]]]]]

   @分からない点 = [[parm !X[]]
                    [restr [[reln 質問-1]
                            [obje !x]]]]
   return @分からない点
end 点-1
```

(1.12) 「質問」に関する書き換え規則

```
on 質問-1
   in= [[parm !X[]]
        [restr [[reln 質問-1]
                [obje !x]]]]

   out= [[parm !X[]]
         [restr [[reln question-1]
                 [obje !x]]]]
end 質問-1
```

に続けて記述する。書き換えタイプは書き換え規則を適用することにより、どのようなタイプの書き換えが行われるかを、書き換え方向は規則適用により、あるタイプにおいてどの方向に素性構造を書き換えるかを示す。

書き換え規則(1.13)は「点」に関して、「こと(イベント)」的な表現から「もの(オブジェクト)」的な表現に日本語内で書き換える規則であると定義される。基本的にはすべての書き換え

(1.13) 「点」に関する書き換え規則

```
on 点-1 in :event-or-object :object :language :Japanese
   in= (parm restr) @分からない点
       [[parm !x1[[parm !x2[]]
                  [restr [[reln 点-1]
                          [obje !x2]]]]]
        [restr [[reln negate]
                [obje [[reln 分かる]
                       [expr ?expr]
                       [obje !x1]]]]]]

   @分からない点 = [[parm !X[]]
                    [restr [[reln 質問-1]
                            [obje !x]]]]
   return @分からない点
end 点-1
```

規則は、書き換えタイプと書き換え方向が定義されなくてはならないが、未定義の場合は、書

き換えタイプが言語(:LANGUAGE)、書き換え方向が英語(:English)とみなされる。したがって、書き換え規則(1.12)は、(1.12')の定義と同等である。

(1.12')「質問-1」に関する書き換え規則

```
on 質問-1 in :language :English
   in= [[parm !X[]]
        [restr [[reln 質問-1]
                [obje !x]]]]

   out= [[parm !X[]]
         [restr [[reln question-1]
                 [obje !x]]]]
end 質問-1
```

書き換えは書き換えタイプごとに独立して行われる。入力素性構造に対して、書き換えタイプの順序が設定[1]され、各書き換えタイプに応じて個別の環境が設定される。ここでは、まず日本語内のパラフレーズを行ってから、日本語から英語への書き換え(変換)を行うよう書き換えタイプの順序を設定する。

環境とは書き換えタイプと書き換え方向の対の集合であり、入力素性構造に対してどのようなタイプの書き換えが可能であり、各々の書き換えタイプにおいてどの方向に書き換えるべきかを示している。

日本語内での名詞句に関するパラフレーズは、:event-or-objectという書き換えタイプを定義し、環境を(:event-or-object :object, :language :Japanese)に設定することにより行うことができる。ここでは、書き換えタイプが:event-or-objectであり、かつ、書き換え方向が環境で設定されている値(:object)と等しい値が定義された規則が、入力素性構造に適用される。このタイプによる書き換えで、素性構造(1.4)に書き換え規則(1.13)が適用され、素性構造(1.14)が得られる。

(1.14) 書き換え規則(1.13)の適用結果

```
[[reln request]
 [agen !sp [[label *speaker*]]]
 [recp !hr [[label *hearer*]]]
 [obje [[reln 聞く-1]
        [expr !sp]
        [obje [[reln ある-1]
               [obje [[parm !X[]]
                      [restr [[reln 質問]
                              [obje !x]]]]]]]]]]
```

次に、書き換え対象を:languageに、環境を(:language :English)に設定することにより、書き換えタイプが言語である書き換えが行われる。この時の、素性構造(1.14)が書き換えの対象となる。このフェーズでは、書き換え規則(1.5)、(1.6)、(1.13)が適用され、最終的に素性構造(1.10)が得られる。

1 現在、書き換え規則から環境を変更するメカニズムは存在するが、書き換えタイプの順序を指定したり、書き換えタイプに応じて環境を変更するための、記述枠組みはまだ用意していない。したがって、現在は、どの入力に対してもデフォールトとして予めシステムで用意された書き換えタイプの順序と環境が与えられている。

2. データ構造

ここでは、書き換え規則内で扱われるデータ構造のシンタックスと、それらがどのように用いられるかについて説明する。

2.1 シンボル(symbol)

シンボルはアルファベットで始まり任意の文字列が続く。ただし、以下の文字を含んではならない。

space tab newline [ ] ? ( ) ! @ : , ; $ { } . < >

シンボルは、書き換え規則中では、規則名や変数名、素性名として用いられる。

2.2 文字列(string)

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。ダブルクォートで囲まれた文字列はダブルクォート以外の任意の文字列が許されている。書き換え規則では、書き換え規則のドキュメンテーションとして用いられる。

(注)現在、文字列中のダブルクォートのエスケープはない。

2.3 数(number)

0から9までの任意個の文字列の並び。現在、書き換え規則中では用いられていない。

2.4 素性構造パタン(FS-pattern)

素性構造パタンは、書き換え規則の適用対象である素性構造(これを対象素性構造と呼ぶ)とのマッチングパタンの記述や、書き換えパタンの記述などに用いられる。素性構造パタンには、COMPLEX、ATOMIC、LEAF、VARIABLEの四つのタイプが存在し、各々のタイプに応じて記述が異なる。素性構造パタン中には、タグ、修飾子の記述が可能である(詳しくは2.6 タグ、4.1.2 修飾子を参照)。

2.4.1 complexタイプ素性構造パタン

complexタイプの素性構造パタンは、接頭辞'['で始まり、接尾辞'] 'で終了する。complexタイプの素性構造は、素性とその値(素性構造パタン)の対を一つ以上含んでいる。素性と素性構造の対も、接頭辞'['と接尾辞']'で囲んで記述する。

(2.1) complexタイプ素性構造パタンの例1

```
[[reln 持つ]
 [agen [[label *speaker*]]]
 [recp [[label *hearer*]]]
 [obje 本]]
```

2.4.2 atomicタイプ素性構造パタン

atomicタイプの素性構造パタンは、頭辞'$'で始まり、シンボルがそれに続く。ただし、

complexタイプの素性構造パタンの記述中に現れる場合には、接頭辞'$'を省略することができる。

(2.2) complexタイプ素性構造パタンの例2

```
[[reln 持つ]
 [agen ?agent]
 [recp ?recepient]
 [obje ?object]]
```

(2.3) atomicタイプ素性構造パタンの例

$登録用紙

2.4.3 leafタイプ素性構造パタン

complexタイプの素性構造パタンで、素性を持たないものを特にleafタイプ素性構造と呼ぶ。

2.4.4 variableタイプ素性構造

variableタイプの素性構造パタンは、接頭辞'?'で始まりシンボルが次に続く。varibleタイプの素性構造パタンは変数の一種でもある。variableタイプ素性構造は値の定まっていない素性構造パタンであり、素性構造とのパタンマッチングの際に、任意の素性構造と一致することができる(2.5変数を参照)。

素性構造パタンはBNFでは、以下のように記述することができる。

```
<素性構造パタン>      ::= '[' <FSP-CONTENT> ']'
                      ::= '[' ']'
                      ::= <ATOMIC-TYPE-FS>
                      ::= <VARIABLE>
<FSP-CONTENT>         ::= <FV-PAIR>* [<REST-VARIABLE>]
                      ::= <RESTR-VARIABLE>
<FV-PAIR>             ::= '[' <FEATURE> <素性構造パタン> ']'
<FEATURE>             ::= <SYMBOL>
<ATOMIC-TYPE-FS>      ::= '$' <SYMBOL>
<VARIABLE>            ::= '?' <SYMBOL>
<REST-VARIABLE>       ::= '?' <SYMBOL>
```

2.5 変数(varibale)

変数にはシステムであらかじめ設定されているシステム変数とユーザの定義によるユーザ変数の二種類が存在する。ユーザ変数はvariableタイプの素性構造でもあり、接頭辞'?'で始まり、シンボルが続く。

2.5.1 システム変数

システム変数は、システムであらかじめ定義されており、システムによって、値が初期化されたり代入されたりする。システム変数には、inputとitの二つが存在する。

(1) input

書き換え規則適用時に、書き換え規則の適用対象である対象素性構造に初期化される。

(2) it

書き換え規則実行中に、一時的に情報を格納し、参照するのに使用される。システムによって書き換え規則内の特定の処理結果が代入される。したがって、変数"it"の値は、書き換え規則内の文脈に依存する。以下の文あるいは式の評価結果が変数"it"に値が設定される。

(1)部分素性構造の書き換え
(2)条件式 (素性構造と素性構造パタンのパタンマッチング、素性構造に対する素性の有無の検査)
(3)パス指定

2.5.2 ユーザ変数

ユーザ変数は、ユーザが自由に定義、代入することのできる変数である。素性構造パタン中に現れる変数もユーザ変数であり、それには、regular variableとrest variableの二種類が存在する。

(1) regular variable

パタンマッチングにおいて、任意の素性構造と一致が可能な変数である。

(2) rest varibale

パタンマッチングにおいて、素性と値の対の集合と一致が可能である変数である。

regular variableとrest variableの表現形式は同様であり、素性構造パタン中の変数が現れる位置によって区別される。regular variableは、通常の素性構造が現れる位置に記述される変数であり、パタンマッチングにおいて任意の素性構造と一致する。一方、rest variableは、素性が記述される位置に現れる変数であり、パタンマッチングにおいて任意の素性-素性構造対の集合と一致する。ただし、rest variableを同じレベルに複数個記述することは許されない。以下に、regular variableとrest variableの例を示す。

(2.4) ユーザ変数の例1

```
?variable
```

?variableはregular variableである。

(2.5) ユーザ変数の例2

```
[[reln  s-request]
 [obje  [[reln  informref]
         [obje  ?object]
         ?rest1]]
 ?rest2]
```

?object は regular variable、?rest1, ?rest2 は rest variable である。

2.6 タグ(tag)

タグは二つの素性構造がトークンとして同一(token identical)であることを表現するための手段として用いられる。タグはスコープおよび定義(ラベリング)の違いによってlocal tag と global tag の二種類に分類される。

(1) local tag

[syntax]

!＜シンボル＞

local tagは接頭辞'!'で始まり、アトミックなオブジェクトがタグ名として続く。local tagは、素性構造パタン中でタグに続く素性構造に対してラベリングされる。local tagの参照スコープは、定義された素性構造パタン内で有効である。素性構造パタン(2.6)はグラフでは図2.1

(2.6) local tag を含んだ素性構造パタンの例

```
[[a !x [[b c]
        [d e]]]
 [f !x]]
```

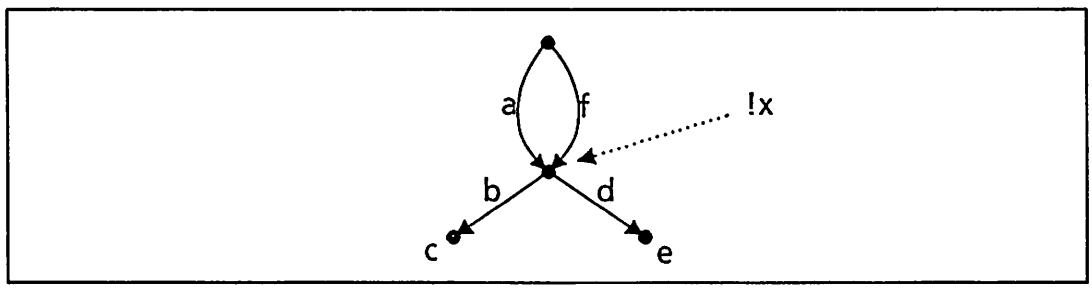

図2.1 素性構造の共有と local tag

のように表現できる。

(2) global tag

[syntax]

@＜シンボル＞

global tagは、接頭辞'@'で始まり、アトミックなオブジェクトがタグ名として続く。global tagは素性構造パタン中に記述され、パタンマッチングの際に、対象素性構造の対応する部分素性構造に対してラベリングされる。global tagの参照スコープは書き換え規則中で有効である。global tagでラベリングされる素性構造は、変数と同様にパタンマッチングの際に決定されるが、タグで参照される素性構造の再定義はできないという点で変数と異なる(4.1.3スローバルタグ参照)。

(2.7)global tagを含んだ素性構造パタンの例

```
[[reln 持つ]
 [agen @agen [[label *speaker*]]]
 [obje @obje ?obje]]]
```

2.7 パス(PATH)

素性構造パタン、グローバルタグ、あるいは、既に素性構造が代入されている変数(これらを総称して素性構造オブジェクトと呼ぶ)に対して、素性のパスを指定することにより、指定されたパスの先にある部分素性構造にアクセスすることができる。パスの指定は素性構造オブジェクトに続けて、ドット('.')で始まり、パスを構成する素性間をドットで区切って記述される。

(2.8) パスの記述例

```
input.obje.obje.agen
```

例(2.8)において、変数inputの値が(2.9)で表現される素性構造であるとき、パスで参照される値は、[[label *hearer*]]となる。

(2.9)

```
[[reln  s-request]
 [agen  !sp [[label  *speaker*]]]
 [recp  !hr [[label  *hearer*]]]
 [obje  [[reln  informif]
         [agen  !hr]
         [recp  !sp]
         [obje  [[reln  持つ]
                 [agen  !hr]
                 [obje  登録用紙]]]]]]
```

2.8 定数(constant)

システムで定義されている定数には、以下の二つがある。

TRUE
FALSE

FALSEは論理演算における偽を表し、TRUEは真を表す。パタンマッチングにおいてFALSE以外の任意の対象はTRUEと一致する。

2.9 タイプ(TYPE)

素性構造のタイプであり、complex、atomic、leaf、variableの四つのタイプが定義されている。

[syntax]

type of <素性構造オブジェクト>

素性構造のタイプは上記のように演算子"type of"で参照することができる。

2.10 パラメータ

[syntax]

:<シンボル>

パラメータおよびパラメータ値は、接頭辞':'に続けてシンボルを記述することにより表現される。書き換えシステムにおいて、パラメータは書き換え可能な素性構造間の関係のタイプ(書き換えタイプ)を表す。パラメータ値は、パラメータで規定された書き換えのタイプにおいて、どの方向に書き換えをおこなうかを規定する。したがって、書き換え規則のパラメータ定義における各パラメータは、書き換え規則がどのようなタイプの書き換えをおこなう規則であるかを規定し、パラメータ値により、その書き換え規則によってどの方向に書き換えがおこなわれるかを規定する。パラメータおよびパラメータ値から構成されるものには、以下の四つのオブジェクトがある。

(1) パラメータ定義
(2) パラメータ環境
(3) パラメータシーケンス
(4) 書き換え対象パラメータ

2.10.1 パラメータ定義

パラメータ-パラメータ値対の集合であり、書き換え規則に個別に定義される。書き換え規則がどのようなタイプの書き換えに属し、そのタイプの書き換えにおいて、どの方向に書き換えおこなうかをパラメータ定義で、定義する。書き換え規則は、規則のパラメータ定義がパラメータ環境で満たされている時に適用される。

2.10.2 パラメータ環境

パラメータ環境はパラメータ-パラメータ値対の集合であり、入力素性構造あるいは部分素性構造に対して定義される。パラメータ環境は入力素性構造の書き換え可能なタイプと方向を示したものである。

2.10.3 パラメータシーケンス

素性構造の書き換えは、一度に一つのパラメータに関して行われる。したがって、いくつかのタイプの書き換えが存在する場合、各々のタイプの書き換えが順に行われる。パラメータ

シーケンスは、パラメータの並びであり、入力素性構造に対して書き換えタイプの順序を定義する。

2.10.4 書き換え対象パラメータ

対象素性構造に対してどのタイプの書き換えを行うかを表すパラメータである。書き換え対象パラメータは、パラメータシーケンスのなかの一つのパラメータであり、ある時点で、パラメータシーケンスのどのパラメータに関して書き換えをおこなっているかを示す。書き換え対象パラメータがパラメータ定義に含まれている書き換え規則が適用可能となる。

3. 書き換え規則の定義と検索、適用

ここでは、書き換え規則の定義方法およびその検索方法と素性構造への適用方法について述べる。

3.1 書き換え規則の定義

書き換え規則はある素性が持つ特定の素性構造に対して、特定のパラメータ定義のもとで定義される。書き換え規則は以下の構造をしている。

(3.1) 書き換え規則の基本構造

```
on <規則名> <パラメータ定義>
   <書き換え規則本体>
end <規則名>
```

書き換え規則はon<規則名>で始まり、end<規則名>で終了する。二つの<規則名>は同じでなければならない。書き換え規則は、atomicタイプの素性構造に対して定義され、書き換え規則が定義されたatomicタイプの素性構造を特定の素性(キー素性と呼ぶ)に持つ素性構造に対して適用される。

キー素性のデフォルト値[1]は"reln"素性および"iden"素性となっている。これは、言い替えると、書き換え規則は素性構造の関係名Rに対して定義され、関係名Rを持つ素性構造に対して、適用される。一方、素性構造の書き換えシステムを翻訳システムの変換モジュールの立場から見ると、書き換え規則は、特定の語彙に対して定義されており、語彙駆動型の書き換えシステムであると言える。

(3.2) 書き換え規則の例

```
on 登録用紙
      in=  [[reln 登録用紙]
            [obje []]
      out= [[reln registration-form]
            [obje []]
   end 登録用紙
```

書き換え規則(3.2)は、規則名が「登録用紙」であり、atomicタイプな素性構造「登録用紙」に対して定義される。この書き換え規則は(3.3)のように、素性"reln"の値が「登録用紙」であるような素性構造が入力された時に検索、適用される。

(3.3)

```
[[reln  登録用紙]
 [obje  []]
```

<パラメータ定義>は規則名の後に、接頭辞"in"に続けてパラメータとパラメータ値の対として、任意個記述される。

1.規則の検索の際に参照される素性構造を示すキー素性は、素性のリストとして大域変数 *rw-rule-fetch-key* に設定されている。

[syntax]

in *parameter1 value1 ... parameterN valueN*

パラメータ定義は、その書き換え規則が(1)どのようなタイプの書き換えをおこなう規則であるかということと、(2)書き換え規則がどのようなパラメータ環境において適用されるかを示している。

(3.4) 書き換え規則の例

```
on 登録用紙 in :language :English
   in=  [[reln 登録用紙]
         [obje []]
   out= [[reln registration-form]
         [obje []]
end 登録用紙
```

書き換え規則(3.4)においてパラメータ定義は、書き換えタイプが":language"であり、書き換えの方向は":english"である。書き換え規則を定義する際、パラメータ定義の記述を、省略[2]することができ、その時はデフォルトとしてパラメータ:language、パラメータ値:englishが設定される。したがって、書き換え規則(3.4)の定義は、書き換え規則(3.2)の定義と同等である。

パラメータ定義において、パラメータ値に:unspecifiedというワイルドカードを用いることができる。パラメータ定義で、パラメータPの値をワイルドカード:unspecifiedに設定することにより、パラメータ環境でパラメータPが設定されていれば、その値が何であろうとパラメータPの制約は満たされる。これは、書き換え規則の定義の際にどのタイプの書き換えであるかだけを宣言し、特に値に関する制約を設けたくないときに、有用である。

3.2 書き換え規則の検索

システムは以下の条件を満たした素性構造に対して、書き換え規則の検索を行う。

(1)素性構造がcomplexタイプの素性構造である

(2)素性構造がキー素性を持っており、キー素性の値がatomicタイプの素性構造である

次に、検索された書き換え規則の各々のパラメータ定義の検査が行われる。パラメータ定義に関して以下の二つの条件が満たされた規則が素性構造に適用される。

(1)書き換え規則のパラメータ定義中に書き換え対象パラメータが存在する

(2)書き換え規則のパラメータ定義がパラメータ環境において満たされている

したがって、書き換え規則(3.4)は、以下の制限が満たされたときに適用される。

2. デフォルトのパラメータは大域変数*default-rule-parameter*に設定されているパメータとパラメータ値が用いられる。

(1)適用対象である素性構造の素性"reln"の値が「登録用紙」である

(2)書き換え対象パラメータが:languageである

(3)パラメータ環境中にパラメータ:languageが存在し、その値が:englishである

3.3 書き換え規則の適用

対象素性構造において制約を満たした書き換え規則が見つかると、それらの書き換え規則が対象素性構造に対して適用される。適用可能な書き換え規則が一つ以上存在する場合、各々の書き換え規則は対象素性構造に対して独立に適用される。例えば、対象素性構造Fに対して、n個の適用可能な書き換え規則が存在し、全ての書き換え規則が書き換えに成功したなら、結果としてn個の素性構造が返される。

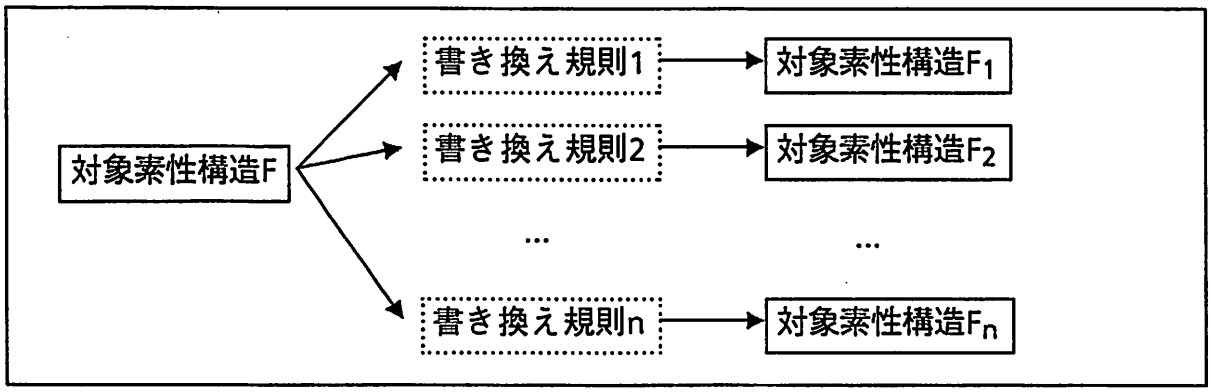

図1　複数の書き換え規則の適用

現在は、全ての適用可能な書き換え規則はまったく同等に扱っているが、機械翻訳システムでの変換モジュールへの適用を考えると、書き換え規則適用に何等かのヒューリスティックスなどを用いて、適用の優先順位をつけたり、複数の書き換え結果に対して何等かの評価関数をもうけて、プリファレンスを与えたりする必要がある。

[implementation memo]

現在のバージョンでは書き換え規則の数だけ入力素性構造をコピーし(書き換え規則中で部分素性構造の書き換え呼び出しとして書き換えが行われる時には、部分素性構造を含む入力素性構造と規則のその時点における環境もコピーされる)、コピーされた素性構造に対して書き換え規則を適用している。ある入力素性構造の中に、書き換えの対象となる対象素性構造がm個存在し、各対象素性構造に関する適用可能な書き換え規則がn_i (i=1, ..., m)個あるとすると入力素性構造のコピーは

$$\prod_{i=0}^{m} n_i$$

回おこなわれることになる。書き換え規則数が増えることにより計算量の増加が予想される。これに対する対処方法としては、構造をシェアリングする方法が考えられる。

現在のインプリメントでは、素性構造の書き換えは、元の構造をそのまま残し、書き換えられた部分だけ新たな構造を作りだし、元の構造から新たな構造へforwardingすることによって

部分構造の書き換えを実現している。素性構造を、素性構造をノード、素性をアークとするグラフで表現すると素性構造(3.6)は図2のように表現できる。

(3.6)　書き換え対象の素性構造

```
[[a [[b 1]
     [c 2]]]
```

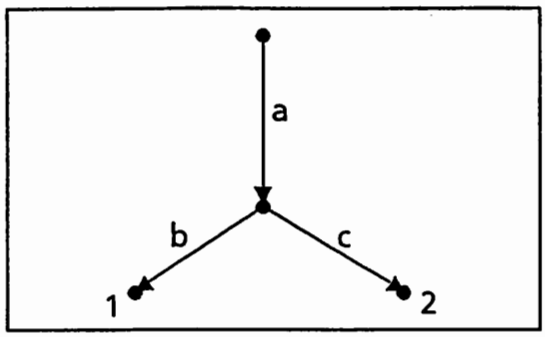

図2　素性構造の表現

部分構造の書き換えは、元のノードから新たな構造へforward linkを張ることによって行われる。素性構造(3.6)から素性構造(3.7)への書き換えをおこなうと、その構造はグラフでは図3のようになる。

(3.7)　書き換えられた素性構造

```
[[a !x[[bb 1]
       [cc 2]]]]
```

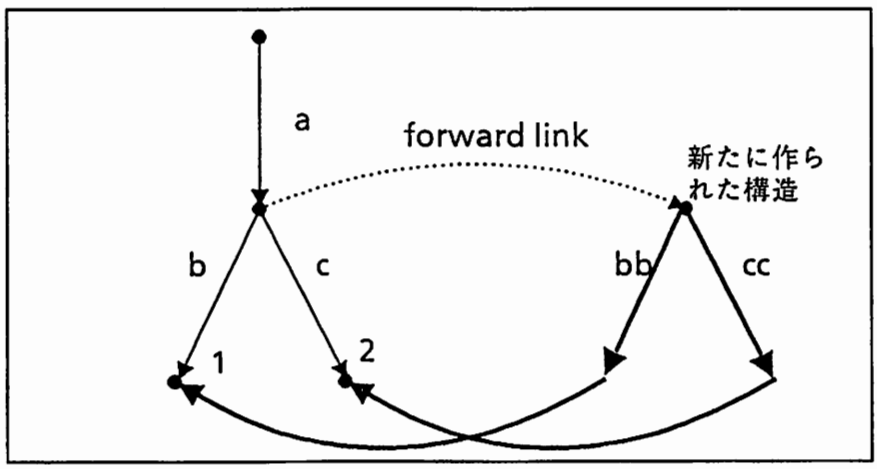

図3　素性構造の書き換え

現在は、forward linkは一つノードに一つしか存在しえないが、複数個のforward linkが存在できるよう拡張し、各々の書き換え規則に個別のforward linkを対応させることによって、素性構造のコピーは必要なくなる。

4. 書き換え規則の基本機能

書き換え規則における基本的な機能は、素性構造とのパタンマッチング、代入、素性構造の生成、素性構造の書き換え、素性構造の書き換え呼び出し、素性構造への素性-素性構造対の集合の追加、パラメタの設定の七つである。ここでは、これらの基本機能について述べる。

4.1 素性構造と素性構造パタンマッチング

書き換え規則中で、対象素性構造の検査や、目的の部分構造を取り出すために、素性構造と素性構造パタンとのパタンマッチングが行われる。パタンマッチングの対象は素性構造であり、構造の共有、循環を許す。

二つの素性構造のパタンマッチングは、二項演算子"=?"、"=!"で行う。

[syntax]

＜素性構造＞ =? ＜素性構造パタン＞

＜素性構造＞ =! ＜素性構造パタン＞

素性構造と素性構造パタンとのパタンマッチングを行うと、二つの構造が一致したか否かによって、値TRUEあるいはFALSEが返る。演算子"=?"はパタンマッチングが成功したときにはTRUEを返し、演算子"=!"はパタンマッチングが失敗したときにTRUEを返す。パタンマッチングは＜式＞(expression)であり、if文などの条件式として用いることができる。また、システム変数"it"を用いて結果の値を参照することもできる。正確には、パタンマッチングは、素性構造以外に、素性構造オベジュクトを対象とするしたがって、パタンマッチングのシンタックスは以下のように書ける。

素性構造パターン中には、変数、修飾子、タグを記述できる。以下に、変数、修飾子、タグの詳細を説明する。

4.1.1 変数

素性構造と素性構造パターンとのマッチングにおいて、値の代入されていない変数は、構造上対応の取れる任意の素性構造と一致することができる。マッチングが成功した場合、副作用として、対応する素性構造が変数に代入される。変数と素性構造とのパタンマッチングの際に、注意しなければならないのは、まだ値が代入されていない変数とのマッチングでは任意の素性構造とのマッチングがおこなわれるが、既に値が代入されている変数では、対応する対象素性構造と、変数の値とのマッチングが試みられる。

変数にはregular variableとrest variableの二種類がある。二つの種類は、素性構造パターン中の現れる場所によって決定される。

(1)regular variable

regular variableは素性構造パターン中で素性構造が現れる位置に記述され、パタンマッチングの際、regular variableは任意の素性構造と一致することができる。

また、<素性構造パターン>中に同じ名前で複数個の変数が現れる場合、それらに対応する素性構造は、トークンとしての同一性が保たれていなければならない。したがって、素性構造(4.1)は、パタンマッチングにおいて、素性構造パターン(4.3)および(4.4)と一致する。

```
(4.1)          (4.2)        (4.3)        (4.4)
[[a !tag A]    [[a A]       [[a ?X]      [[a ?Z]
 [b !tag]]      [b B]]       [b ?Y]]      [b ?Z]]
```

素性構造(4.1)と素性構造パタン(4.3)および(4.4)とのパタンマッチングは共に成功し、各変数は以下のようになる。

```
?X = ?Y = A
?Z = A
```

一方、素性構造(4.2)と素性構造パターン(4.3)とのパタンマッチングは成功するが、素性構造(4.2)と素性構造パタン(4.4)とのマッチングは、変数?Zに対応する素性構造が一致しないので失敗する。

(2)rest variabale

素性構造パターンの素性の位置に現れた変数は rest variable とみなされ、マッチングが成功した時、素性構造パタンに記述されていない残りの素性-素性値対の集合が代入される。rest variable は素性構造パターン中の同じレベルに一つだけ記述できるという制限がある。

```
(4.5)          (4.6)
  [[a A]         [[a A]
   ?rest]         [b B]
                  [c C]]
```

素性構造パターン(4.5)と対象素性構造(4.6)とのパタンマッチングは成功し、その結果、rest variableには、

```
?rest = { [b B] [c C] }
```

の素性-素性値対の集合が代入される。

4.1.2 修飾子

素性構造パターン、あるいは、その部分構造に修飾子を付けることができる。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記述であり、修飾子の付けられた素性構造パターンと素性構造のマッチングが行われる際に評価される。修飾子を用いることにより、パタンマッチングによって、再帰的な素性構造から目的とする部分構造を取り出したり、対象素性構造を含む素性構造(上位の素性構造)にアクセスすることができる。素性構造と修飾子が付けられた素性構造パターンのマッチングは、まず、対象素性構造を修飾子の記述にもとづいて素

性のパスを探索する。修飾子の記述と一致するパスが対象素性構造中に存在すれば、パスの先にある素性構造と修飾子が付けられた素性構造パタンとのマッチングga行われる。修飾子として記述された素性のパスが対象素性構造中に存在しなければ、マッチングは失敗する。修飾子はパスの探索の方向によって二種類存在する。

(1)downward modifier
対象素性構造の末端に向かってパスを探索する修飾子
(2)upward modifier
対象素性構造の根に向かってパスを探索する修飾子

(1)downward modifier

downward modifierは、部分素性構造パタンに先だって、接頭辞'<'で始まり、次に素性のパスのパタンが空白文字で区切られて記述され、最後に接尾辞'>'で終了する。パスの表現は正規表現のように、素性の選言(OR)、および、繰り返しが記述できる。素性の選言は、一つ以上の素性を'('と')'で囲むことにより表現される。素性あるいは素性の選言の繰り返しには、0回以上と一回以上の繰り返しがあり、0回以上の繰り返しは、繰り返しの対象となる素性に続けて接尾辞'*'で表現され、一回以上の繰り返しは接尾辞'+'で表現される。

(,)	素性の選言
+	素性あるは素性の選言の1回以上の繰り返し
*	素性あるは素性の選言の0回以上の繰り返し

以下に修飾子を含んだ素性構造パタンの例を挙げる。

(4.6) 修飾子を含んだ素性構造パタン

```
[[reln  持つ]
 [agen  ?agent]
 [obje  <parm* restr>[[reln 登録用紙]
                     [obje []]]]]
```

素性構造パターン(4.6)は素性構造(4.7)とも(4.8)とも一致することができる。

(4.7) 対象素性構造の例1

```
[[reln  持つ]
 [agen  [[label *speaker*]]]
 [obje  [[parm  !x1[]]
         [restr [[reln  登録用紙]
                 [obje  !x1]]]]]]
```

素性構造(4.7)と素性構造パタン(4.6)のパタンマッチングは成功し、その際、修飾子<parm* restr>はパスrestrに展開される。

(4.8)　対象素性構造の例2

```
[[reln  持つ]
 [agen  [[label *speaker*]]]
 [obje  [[parm  !x0  [[parm  !x1[]]
                      [restr [[reln  登録用紙]
                              [obje  !x1]]]]]]
        [restr [[reln 赤い]
                [obje !x0]]]]]
```

素性構造(4.8)と素性構造パタン(4.6)のパタンマッチングは成功し、その際、修飾子<restr* parm>はパスはparm.restrに展開される。

修飾子は(4.9)のように素性構造だけでなく変数(ただしregular varibaleのみ)に付けることもできる。素性構造パターン(4.9)と素性構造(4.7)、および、(4.8)とのパタンマッチングは共に成功する。

(4.9)　修飾子を含んだ素性構造パタン

```
[[reln  持つ]
 [obje  <parm* restr reln> ?object-core]
 ?rest]
```

素性構造パタン(4.9)と素性構造(4.7)、および、(4.8)とのパタンマッチングの結果、変数には以下のように同様の部分素性構造、および、素性-素性構造対の集合が代入される。

```
?obje-core = 登録用紙
?rest      = { [agen [[label *speaker*]]] }
```

このように、修飾子を用いることによって、名詞句のような再帰的な構造から特定の部分構造を取り出したり、再帰的な構造とのマッチングを容易に記述することができる。

downward modifierの計算について

素性のORや繰り返しを素性のパス記述に認めると、パスの記述に対応するパスがいくつか存在する可能性がある。

(A)ループの例

構造上のループを含む素性構造(4.10)と素性構造パタン(4.11)をマッチングさせると、素性構造パタン中の修飾子に対応するパスは無限個存在し、どのパスを通ったかによって、変数に対応する素性構造が異なる。

(4.10)対象素性構造

```
!x [[a  [[a !x]
```

```
            [b   val1]]]
      [c   val2]]
```

(4.11) 素性の繰り返しの修飾子を持つ素性構造パタン

```
<a*>?x
```

(解1) パス == NULL, a.a, a.a.a.a, ...

```
?X    == !x [[a [[a !x]
                [b val1]]]
          [c   val2]]
```

(解2) パス == a, a.a.a, a.a.a.a.a, ...

```
?x    ==  !x [[a [[a !x]
                   [c val2]]]
             [b   val1]]
```

(B)ORの例

素性構造(4.12)と素性構造パタン(4.13)をマッチングさせると、ORで記述されているどちらの素性も対象素性構中に存在するので、修飾子に対応するパス(素性)が複数個存在し、どちらを選択したかによって、変数?Xの値が異なる。

(4.12) 対象素性構造

```
[[a val1]
 [b val2]]
```

(4.13) 素性構造パタン

```
<(a b)>?X
```

(解1) パス == a

```
?X    == val1
```

(解2) パス == b

```
?X    == val2
```

(C)繰り返しの例

素性構造(4.14)と素性構造パタン(4.15)のマッチングを行うと、修飾子中の各素性との対

(4.14)対象素性構造

```
[[a   [[a v1]
       [b v2]]]
 [b v3]]
```

(4.15)素性構造パタン

```
<a* b*>?X
```

応を調整することによって、表1に示すように変数?Xの値が異なる。

表1　修飾子の展開

		解1	解2	解3	解4
パス	a*	a	a.a	∅	∅
	b*	∅	∅	b	∅
値	?X	[[a v1] [b v2]]	v1	v3	[[a [[a v1] [b v2]]] [b v3]]

修飾子を評価する際、修飾子に対応するパスが複数個存在するような場合に、解を一意に決定するために、以下の規則を設ける。

(1) 素性のパス並びにおいて、先頭にある素性を優先する
(2) ORは記述中の素性の並びにおいて、先頭にある素性を優先する
(3) 繰り返しは最長パスを優先する
(4) 繰り返しに循環を認めない

したがって、接尾辞'*'の付いた素性のパスの探索は以下のように行われる。接尾辞'*'の前の素性を可能な限り繰り返す。ただし、繰り返しの中では循環が起こった時点で、繰り返しを停止しパスを一つ戻す。次に残りの部分をマッチさせる。これが失敗した場合にはバックトラックが行われる。つまり、接尾辞'*'によって繰り返さたパスの構成要素のいくつかを捨てて、修飾子の残りの部分がマッチするように試みる。

素性構造(4.10)と素性構造パタン(1.11)のパタンマッチングでは、修飾子a*に対応する素性を辿ってゆくと、パスが'a.a'となった時点で、パスに循環が生じる。したがって、修飾子にマッチしたパスおよび変数の値は以下のようになる。

```
パス   <a*> == a
変数   ?X == !x [[a   [[a   !x]
                      [c val2]]]
                [b   val1]]
```

素性構造(4.12)と素性構造パタン(4.13)のマッチングでは、ORの先頭の要素が優先されるので以下のようになる。

```
パス   <(a b)> == a
変数   ?x == val1
```

(4.14)と(4.15)のマッチングでは、規則(1)によってa*に一致するパスがb*に一致するパスより優先されるので(解3)、(解4)が却下され、規則(3)によってa*に一致するパスa.aがaより優先されるので(解1)が却下される。したがって、(解2)が得られる。

修飾子が付けられている部分素性構造パタンが、変数で複数のパスが存在する時には、(1)から(3)の優先規則にそってパスを決定するが、修飾子が付けられているのが素性構造であり、パスが一意に決定できる場合には、パスを調節しながらパタンマッチングを行う。素性構造パターン(4.14)と、表2の修飾子が付けられた各素性構造パタンとのパタンマッチングはすべて成功し、各パタンマッチングの結果は表2のようになる。

(4.14)

```
[[a  [[a v1]
      [b v2]]]
 [b v3]]
```

表2　downward modifierを含んだ素性構造パタンのマッチング

	パタン1	パタン2	パタン3	パタン4	パタン5
パタン	<a* b*>V1	<a* b*>V2	<a* b*>V3	<b* a*>V1	<b* a*>V3
パス	a* = a b* = b	a* = a b* = b	a* = ∅ b* = b	b* = ∅ a* = a.a	b* = b a* = ∅

(2)upward modifier

upward modifierは、入力素性構造を根の方向に向かってパスを探索する。upward modifier

(4.16)　入力素性構造

```
[[reln  ある]
 [agen  !hr  [[label *hearer*]]]
 [obje       [[parm !x1 [[parm  !x2[]]
                        [restr [[reln 点]
                                [obje !x2]]]]]
             [restr [[reln  ない]
                     [obje  [[reln  分かる]
                             [expr !hr]
                             [obje !x1]]]]]]]
```

(対象素性構造)

(マッチングの対象となる素性構造)

を使うことによって、対象素性構造を含んだより大きな素性構造を参照することができる。修飾子は素性構造パタンに先立ってパスを'('と')'で囲んで記述する。upward modifierは、downward modifierと異なり繰り返し、および、選言を用いることはできない。

(4.17) 素性構造パタン

```
(restr obje) [[parm  !x1  [[parm  !x2[]]
                           [restr [[reln  点]
                                   [obje !x2]]]]]
             [restr [[reln  ない]
                     [obje  [[reln  分かる]
                             [expr ?hr]
                             [obje !x1]]]]]
```

入力素性構造(4.16)において、素性relnの値が「分かる」である素性構造と素性構造パターン(4.17)とのマッチングを行うと、素性のパスrestr、objeを根の方向に辿った素性構造と一致する。

4.1.3 グローバルタグ

グローバルタグはパタンマッチングの際に、素性構造パタン中のグローバルタグの位置と構造上対応する素性構造に対してラベリングされる。グローバルタグは、接頭辞'@'で始まり、シンボルが続く。

[sytax]

@<シンボル>

グローバルタグとローカルタグの違いは以下の二点である。

(1)スコープ

グローバルタグの参照スコープは規則内であり、ローカルタグは素性構造パタン内である

(2)ラベリング

ローカルタグはタグに続く素性構造をラベリングするが、グローバルタグはパタンマッチングの際に、構造上対応する素性構造をラベリングする

グローバルタグは、パタンマッチングの際、対応する素性構造が決定され、ばたんまっちんぐ終了後規則中で参照することができるという点で変数と似ているが、その再定義(代入)が許されないという点で異なる。

素性構造パタン中で修飾子を用いることによって、より柔軟な素性構造パタンの記述がおこなえる。しかし、upward modifierによるパタンマッチングでは、素性構造パタンが変数以外の場合、パスの先の素性構造をパタンマッチングの後でアクセスすることができない。また、downward modifierによるパタンマッチングでは、パスの元の素性構造にアクセスすることができない。グローバルタグを修飾子と併用することにより、パスの元あるいはパスの先の素性構造を参照することができる。グローバルタグのラベリングのされかたは、ぐろーばるたぐと併用される修飾子の種類によって異なる。

(1)グローバルタグに続けてupward modifierが記述されている場合には、パスを辿った先の素性構造に対してラベリングされる。
(2)グローバルタグに続けてdownward modifierが記述されている場合には、パスの元の素性構造に対してラベリングされる。
(3)グローバルタグに続く素性構造パタンに修飾子が付けられてない場合には、直接対応する素性構造に対してラベリングされる。

(4.20) 入力素性構造

```
[[reln  ある]
 [agen  !hr  [[label *hearer*]]]
 [obje       [[parm !x1 [[parm  !x2[]]
                        [restr [[reln 点]
                                [obje !x2]]]]]
             [restr [[reln  ない]
                    [obje  [[reln 分かる]
                            [expr !hr]
                            [obje !x1]]]]]]]
```

(4.21) 素性構造パタン

```
@tag1 (restr obje)
[[[reln ない]
 [obje [reln 分かる]
       [expr ?expr]
       [obje @tag2<parm* restr reln>?OBJECT]]
```

素性構造(4.20)と素性構造パタン(4.21)のパタンマッチングをおこなうと、グローバルタグ、変数には図1に示すように部分素性構造が代入される。

4.2 変数への代入

変数への代入は素性構造のマッチングの副作用として行われることは既に述べたが、規則中で明示的に変数への代入を行うこともできる。

[syntax]

```
set <変数> to <素性構造オブジェクト>
```

4.3 素性構造の生成

素性構造は、素性構造パターンによって生成することができる。素性構造パターンは書き換え規則の実行時にダイナミックに生成される。素性構造パターン中に変数が含まれているときには、それらの値は未確定であってもよい。しかし、規則が終了するまでには、値の代入が行われなくてはならない。

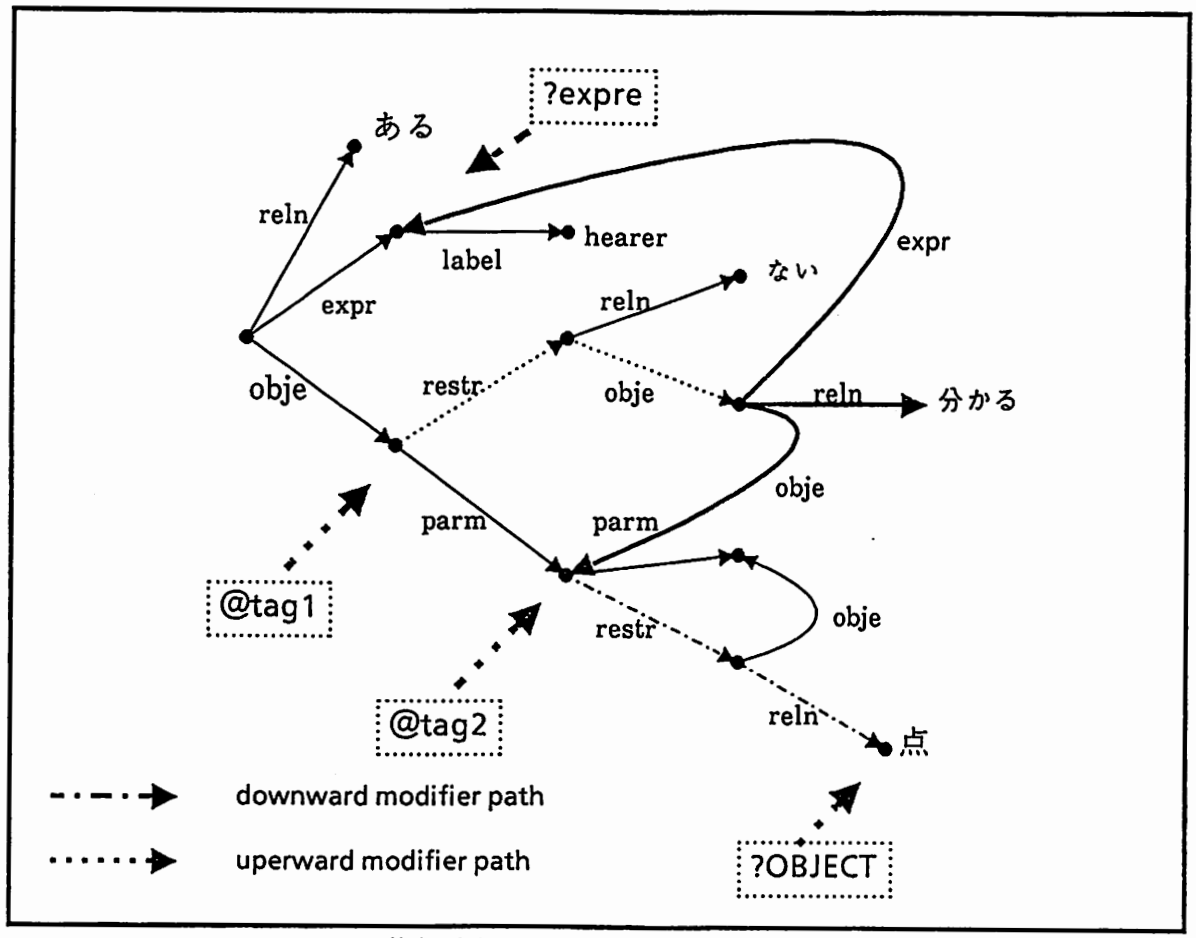

図1　修飾子とグローバルタグのラベリング

4.4 素性構造の書き換え

素性構造の書き換えは、二項演算子'='あるいは"replace"をもちいておこなうことができる。

[syntax]

<素性構造オブジェクト> = <素性構造オブジェクト>

replace <素性構造オブジェクト> with <素性構造オブジェクト>

(4.22)　素性構造の書き換え記述例1

```
input = [[reln have]
         [agen ?agent]
         [obje ?object]
         ?rest]
```

(4.23)　素性構造の書き換え記述例2

```
?agent = ?speaker
```

二項演算子'='は左辺の素性構造オブジェクトを右辺の素性構造オブジェクトで書き換える。

書き換えられる(二項演算子の左辺の)素性構造オブジェクトが変数、あるいは、グローバルタグ単体である場合には、それらには値が代入されていなくてはならない。

4.5 素性構造の書き換え呼び出し

書き換え規則中で部分素性構造に対して、書き換え呼び出しを行うことができる。書き換えの呼び出しはオペレータ"->"あるいは、"=>"を用いて行う。

[syntax]

-> <素性構造オブジェクト> [with <parmeter-set>]

=> <素性構造オブジェクト> [with <parmeter-set>]

オペレータ"->"は、素性構造オブジェクトのトップレベルだけを書き換える。一方、オペレータ"=>"は、素性構造オブジェクトに対して再帰的に書き換えをおこなう。部分構造の書き換え呼び出しの際、キーワード"with"に続けてパラメータセットを記述することができる。書き換え対象パラメータがwithに続くパラメタセットの最初のパラメータに設定され、パラメータ環境が局所的(部分素性構造の書き換えがおこなわれている間だけ有効)に、パラメータセットに設定される。つまり、書き換え呼び出しが評価される時に書き換え対象パラメータおよびパラメータ環境にそれぞれパラメータ、パラメータセットが設定され、評価終了時に元に戻される。

書き換え呼び出しの結果として、TRUEとFALSEのどちらかの値が返る。

(1)TRUE

書き換え規則が適用され、素性構造の書き換えが行われた時の状態

(2)FALSE

適用可能な規則が存在しなかった、あるいは、適用可能な規則は存在したが、適用が失敗に終わった等の理由により、書き換えが行われなかった時の状態

書き換え呼び出しが成功した場合には、書き換えられた素性構造を、システム変数"it"によって参照することができる。また、書き換え呼出しの対象が、変数に代入されている場合は、その変数によって書き換えられた素性構造を参照することができる。

素性構造の書き換え呼び出しの結果、複数個の素性構造が返されることがある。この時、システムは書き換え結果の数だけプロセスを生成(フォーク)し、各々の書き換え結果に対応するプロセスにおいて、書き換え呼び出し以降の手続を独立に処理する。各プロセスは、システム変数"it"(あるいは書き換え呼び出しの引数となった変数)の値には、各書き換え結果が代入される。例えば、(4.24)のような手続きにおいては、部分素性構造(ここでは対象素構造input)の書き換え呼び出しが行われる。変数inputの値がFSであり、書き換え呼び出しの結果、FS1、FS2、FS3の三つの素性構造が返されたとき、図2の様に三つのプロセスが生成される。

書き換えが失敗した場合には、変数"it"の値はFALSEであり、"it"の値がFALSEである一つのプロセスが生成され、残りの手続が処理される。

(4.24)　部分素性構造の書き換え呼び出し

```
...
-> input
it
...
```

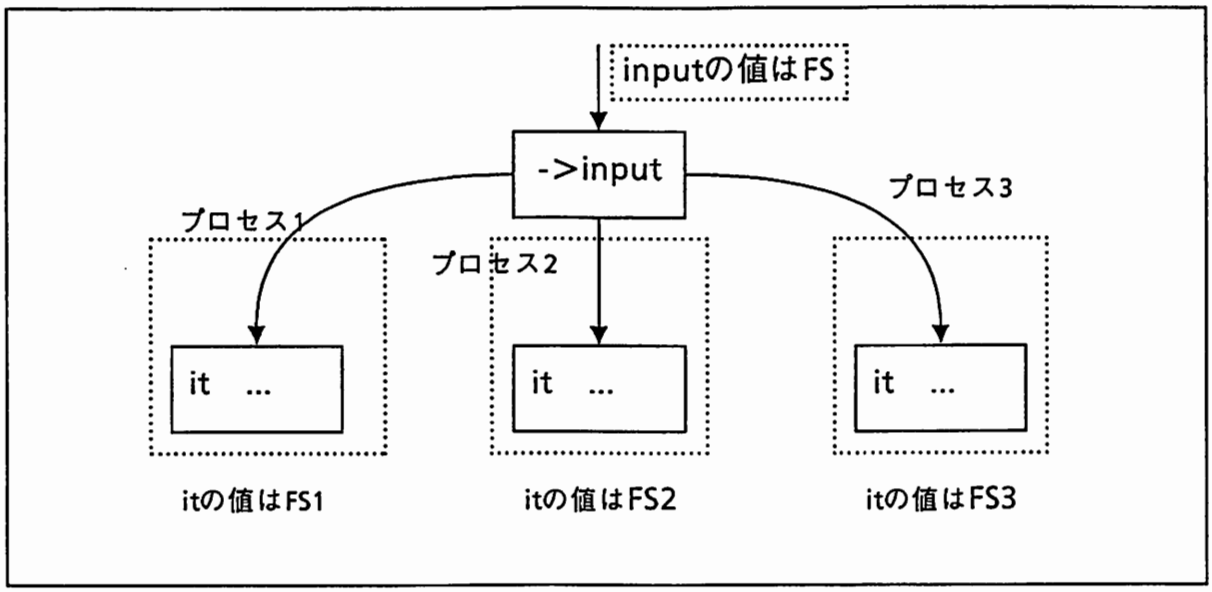

図2　部分素性構造の書き換えによるプロセスのフォーク

書き換え規則(4.25)は「する」のOBJEct素性である部分素性構造を述語的な表現に書き換

(4.25) 書き換え呼び出しを含んだ書き換え呼び出し

```
on する in :event-or-object :unspecified
   in= [[reln する]
        [obje <parm* restr> ?object]]
   -> ?object with :event-or-object :event
   if it is TRUE
   then
      out= ?object
   endif
end する
```

え、書き換え結果を対象素性構造の書き換え結果として出力する。この書き換え規則では、まず、対象素性構造の検査を行い、次に、部分素性構造の書き換え呼び出しを行っている。この書き換え呼び出しは、パラメタの局所設定が指示されており、書き換え対象パラメタに:event-or-objectを設定し、さらに、パラメータ環境の:object-or-eventパラメータの値を:eventに局所的に設定する。最後に、書き換え呼び出しが成功したか否かの判断があり、書き換え呼び出しが成功したなら、対象素性構造を書き換え結果で置き換える。書き換えが失敗した場合の処理はここでは記述されていないが、その場合、対象素性構造は何等影響を及ぼされない。

4.6 素性-素性値対の集合の追加

rest variableを含んだ素性構造パタンのマッチングによって、rest variableには素性-素性構造対の集合が代入される。この素性-素性構造対の集合を素性構造に追加することができる。

[syntax]

add <素性-素性構造対集合> to <素性構造オブジェクト>

<素性-素性構造対集合>中の素性が、<素性構造オブジェクト>に既に含まれている場合はエラーになる。

4.7 パラメータの設定

部分素性構造の書き換えでも、パラメータ環境の設定を行うことができるが、パラメータの設定は一時的なものであり、部分素性構造の書き換えが終了した時点で、パラメータ環境の設定はもとに戻される。それに対して、set parameterおよびunset parameterはパラメータ環境の恒久的な設定が行われる。

[syntax]

SET PARMETER *parameter1 value1 ... parameterN valueN*

UNSET PARAMETER *parameter1 ... parameterN*

set parameterはパラメータとパラメータ値の対を複数個取る。set parameterはパラメータ環境に対して、パラメータ*parmaeter1*を追加し、その値を*value1*に設定する。もしパラメータ*parameter1*がすでにパラメータ環境に設定されていれば、その値を*value1*に変更する。
unset parameterは引数であるパラメータをパラメータ環境から削除する。

5. 書き換え規則における制御

書き換え規則内での制御に関する文のいくつかについては既に述べたが、ここでは、それらを含めた制御文のより厳密な説明を行う。書き換え規則はいくつかの文(statement)の並びからなる。文の並びには特に制限はないが、単純な書き換え規則では、対象素性構造の検査部と対象素性構造の書き換え部とからなるのが一般的である。

5.1 対象素性構造の検査文

[syntax]
in= <素性構造パターン>

単項演算子"in="は、対象素性構造と素性構造パターンとが一致できるかを検査(パタンマッチング)し、もし一致すれば、書き換え規則を続行する。対象素性構造と素性構造パターンとが一致しなければ、書き換え規則の適用は失敗したとして規則を終了する。したがって、"in="は以下の文と同等の効果をもつ。

```
if input =! <素性構造パターン>
then
    fail
endif
```

5.2 対象素性構造の書き換えと書き換え規則の終了

[syntax]
return <素性構造オブジェクト>
out= <素性構造オブジェクト>
fail

書き換え書き換え規則の終了には二つの状態が存在する。(1)書き換え規則を正常に終了する場合で、このとき、書き換え規則の適用は成功(あるいは正常終了)したと言う。(2)対象素性構造が書き換えをおこなう条件を満たしていない等の理由で、書き換えをおこなわずに終了する場合で、書き換え規則の適用は失敗(あるいは異常終了)したと言う。return文、および、out=文は、書き換え規則適用の成功を意味し、fail文は書き換え規則適用の失敗を意味する。return文が実行されると、即座に書き換え規則を終了し、returnに続く素性構造パターンは、その書き換え規則における対象素性構造の書き換え結果が、素性構造パターンであるを書き換えシステムに伝える。out=文は、対象素性構造を素性構造パターンで書き換え、規則を終了する。これは、以下の文と同等の効果をもつ。

1. 書き換え規則のシンタクチックな構造において、GRADEのrewriting ruleのように、matching instruction part, matching condition part, creation partといった規則の構成要素をあらかじめ定義し、その並びを規定することにより定型的構文構造を規則に持たることができる。規則に予め一定の構造を持たせておくことにより、記述上での不必要な混乱を抑制したり、規則の可読性や保守性を増すというメリットが存在するが、言語としての自由度は薄れる。ここでは、ベースとなる規則記述言語においては、特に構造を規定せず、自由度を持たせておき、規則の蓄積の際に一定のパターンが定まった時点で、マクロ機能等を用いて定型的な構造を規則に導入したい。

```
input= <素性構造オブジェクト>
return <素性構造オブジェクト>
```

fail文は、書き換え規則の異常終了を意味する。fail文は、評価された時点で直ちに書き換え規則を終了し、書き換え規則の適用が失敗したことをシステムに伝える。fail文が評価されると、その書き換え規則で行われた処理、例えば部分構造の書き換えなど副作用を持った処理なども、すべて無効となる。

5.3 条件文

[syntax]

```
if <条件式> then <文1> [else <文2>] endif
```

一般のプログラミング言語のif文と同様であり、条件式の真偽値によって分岐する。分岐の論理式(条件式)が真となったときはthenに続く文が実行され、偽のときにはelseに続く文が実行される。分岐の一つであるelse-文は省略することができるが、then-文は省略することはできない。分岐先は各々一つ以上の文が記述されなくてはならない。if~then~else構文のネスティングを許している。

5.4 switch文

[syntax]

```
switch <オブジェクト>
    case <オブジェクト> <文>
    ...
    [default <文>]
endswitch
```

switch文はオブジェクトの値に従って、いくつかの文のうちの一つに制御を移すためのものである。switch文が実行されると、先頭のオブジェクト(switchオブジェクト)の値が計算され、各caseのオブジェクトと順に比較される。オブジェクトの比較には、パターンマッチングが呼び出される。もしcaseオブジェクトと(switchオブジュクト)が一致すれば、制御はその一致したcase接頭辞に続く文に渡される。オブジェクトの比較はcaseオブジェクトの並びの順に比較され、switchオブジュクトが複数のcaseオブジェクトと一致する場合、最初に一致したcaseオブジェクトの文だけが実行される。switchオブジェクトがどのcaseオブジェクトとも一致しなかった時、defaultがある場合には、制御はその接頭辞のついた文に渡される。defaultがない場合には、switch文中のどの文も実行されない。

5.5 演算子

これまでに、パタンマッチング等いくつかの演算子について議論した。ここでは、書き換え規則中で使用できるすべての演算子について、まとめておく。

[syntax]

(1) <素性構造オブジェクト> =? <素性構造オブジェクト>

(2) <素性構造オブジェクト> =! <素性構造オブジェクト>

(3) <素性構造オブジェクト> has <素性>

(4) <素性構造オブジェクト> has not <素性>

(5) <素性構造オブジェクト> . <素性>

(6) not <オブジェクト>

(7) <オブジェクト> or <オブジェクト>

(9) <オブジェクト> and <オブジェクト>

(10) =><素性構造オブジェクト>

(11) -><素性構造オブジェクト>

(12) <素性構造オブジェクト> = <素性構造オブジェクト>

式(1)、(2)は、素性構造オブジェクトのパタンマッチをおこなう。式(1)は、二つの素性構造のパタンが一致すると、真(定数TRUE)を、一致しないときには、偽(定数FALSE)を返す。一方、式(2)はパタンが一致すると偽を、一致しないときには偽を返す。式(3)、(4)は素性構造オブジェクトがある素性を持っているか否かを判定する式である。式(3)は素性を持っていれば真を、式(4)は素性を持っていなければ真を返す。式(5)は、素性構造オブジェクトが素性を持っているとき、その値(部分素性構造)を返す。素性構造オブジェクトが素性を持っていないときには、偽を返す。(6)、(7)、(8)は論理演算子であり、and演算子は二つの論理式を評価し、両方とも真のときに真を、どちらか一方が偽のときには偽を返す。or演算子はどちらか一方が真のときに真を返す。not演算子は、一つの論理式をとり、論理式の逆を返す。(10)、(11)は素性構造オブジェクトの書き換え呼び出しを行う。(12)は素性構造オブジェクトの書き換えを行う。

以下に、これらの演算子の優先度と結合規則をまとめておく。優先順位は表の上にあるものが下にあるものより高い。

演算子	結合規則
.	左から右
type of	
=? =! has	左から右
and	左から右
or	左から右
not	
= set....add	左から右
-> =>	

6. おわりに

本報告では、素性構造を対象とした書き換え規則による書き換えシステムの機能と、書き換え規則のシンタックスについて述べた。書き換えシステムの作成にあたっては、汎用性を重視し、より一般的なシステムを目指してたが、ここで、残っている幾つかの問題点と今後の課題を整理しておく。

問題点と今後の課題

(1) 書き換え規則の局所的な制御と書き換え結果のプリファレンス

書き換え規則の大局的な制御は書き換えタイプを制御することによって行うことができる。しかし、ある書き換えタイプの書き換えにおいて、複数個の書き換え規則があったとき、それらは、すべて対等に扱われる。より効率的に書き換えを行うために、規則に優先順位をつける方法が考えられる。また、書き換え規則の適用により結果が複数個得られた場合、ヒューリスティックなどを用いて書き換え結果にプリファレンスを与え、優先的に処理するようなメカニズムの導入も検討してゆく。

(2) パラメータ制御言語の作成

書き換え規則の大局的な制御は書き換えタイプを表すパラメータによってを制御することができる。しかし、現在は、デフォールトという形でパラメータの並びと、パラメータ環境を設定している。より柔軟な書き換えを行うには、入力や文脈に応じて、パラメータの順序とパラメータ環境の設定を行うべきであろう。入力素性構造をどのように書き換えるべきかという情報をパラメータとパラメータ値という形式に還元することにより、書き換えタイプの順序と各タイプにおける書き換え方向を決定できる。例えば、入力素性構造全体を最初にざっと調べて、書き換えの計画を立案することにより、パラメータの順序やパラメータ環境を設定したり、文脈情報から入力素性構造の書き換え可能性(制約)をパラメータ環境として表現することなどが考えられる。

(3) マクロプロセッサの導入

頻繁に現れる定形的な表現や処理をマクロ定義することにより、書き換え規則の記述量を軽減したり、不注意による誤りを削減することができる。

(4) 素性構造のコピーの問題

書き換え規則を適用する際、前もって入力素性構造のコピーを行っている。コピーは適用可能な規則の数だけ行われるので、規則の数の増大により素性構造のコピーのオーバーヘッドによる効率の低下が予想される。弧の問題に関しては、本文3.3節で述べたforwad linkを多重化する方法により対処してゆく。

(5) 並列処理

前述したように各書き換え規則の依存関係は少なく独立性は高い。また、複数の適用可能な書き換え規則が存在する場合、各規則は独立に素性構造に適用される。現在は、複

数の適用可能な規則が存在するとき、各々の規則の適用を逐次的に処理しているが、書き換えを並列に計算することは非常に容易であり、これにより、より効率的な処理が期待できる。今後、規則適用のメカニズムを実際に並列計算機で実現こと も考えて行きたい。

参考文献

[1] 長谷川,"日英対話翻訳における意味構造変換手法,"第38回全国大会, 情報処理学会, 1988.

[2] 西田,"変換処理過程の基本設計," 自然言語処理研究会, vol. 38, 1983.

[3] Emele, M. and R. Zajac, "RETIF: A Rewriting System for Typed Feature Structures," ATR Technical Report, TR-I-0071, 1989.

[4] Nagao, M. and J. Tsujii, "The Transfer Phase of the Mu Machine Translation System," COLING-86, 1986.

[5] Nakamura, J., J. Tsujii, and M. Nagao, "Grammar Writing System (GRADE) of Mu-Machine Translation Project and its Characteristics," COLING-84, 1984.

[6] Tsujii, J. and M. Nagao, "Future Direction of Machine Translation," COLING-86, pp. 655-667, 1988.

[7] Zajac, R., "Operations on Typed Feature Structures: Motivationsand Definitions," ATR Technical Report, TR-I-0045,1988

A1 書き換え規則のシンタックス

```
rewriting-rule          := rewriting-rule ON rule-name IN parameter-set document
                           statements END rule-name
                        := rewriting-rule ON rule-name document statements END
                           rule-name

rule-name               := symbol

document                := ε
                        := string

parameter-set           := parameter-value-pair
                        := parameter-set parameter-value-pair

parameter-value-pair    := key symbol
                        := key key
                        := key number

parameter-list          := key
                        := parameter-list key

statements              := rewriting-call
                        := rewriting-call statements
                        := statement
                        := statement statements

statement               := control
                        := assignment
                        := fetch
                        := object
                        := expression

rewriting-call          := -> fs-object WITH parameter-set
                        := -> fs-object
                        := => fs-object WITH parameter-set
                        := => fs-object

control                 := FAIL
                        := IN= fs-pattern
                        := OUT= fs-object
                        := RETURN fs-object
                        := IF expression THEN statements ENDIF
                        := IF expression THEN statements ELSE statements ENDIF
                        := SWITCH object casebody ENDSWITCH

assignment              := SET PARAMETER parameter-set
                        := UNSET PARAMETER parameter-list
```

```
                := SET variable TO object
                := SET symbol OF fs-object TO object
                := PUT object INTO variable
                := PUT object INTO symbol OF fs-object
                := ADD object TO object
                := REPLACE fs-object WITH fs-object
                := fs-object = fs-object

fetch           := GET symbol OF fs-object

expression      := condition
                := ( expression )
                := expression AND expression
                := expression OR expression
                := NOT expression

condition       := object IS object
                := object IS NOT object
                := object =? object
                := object =! object
                := fs-object HAS object
                := fs-object HAS NOT object

casebody        := case statements
                := casebody case statements

case            := CASE object
                := DEFAULT
                := case CASE object

object          := symbol
                := number
                := constant
                := fs-object
                := funcall
                := TYPE OF fs-object

fs-object       := fs-patternkey
                := symbol of fs-object
                := fs-object . symbol
                := variable

fs-pattern      := fs
                := tag fs
                := tag
```

constant	:= TRUE
	:= FALSE
variable	:= ? symbol
	:= IT
	:= INPUT
string	:= ".*"
symbol	:= *[a-zA-Z]**
tag	:= !symbol
key	:= :symbol

A2 書き換え規則の例


```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Base: 10 -*-

;;;
;;;
;;;                          Sample transfer rules
;;;
;;;

on S-REQUEST in :PREVIOUS-QUESTION-TYPE :UNSPECIFIED :LANGUAGE :ENGLISH
   in= [[RELN S-REQUEST]
        [OBJE [[RELN INFORMIF]
               [OBJE ?OBJECT]
               ?REST1]]
        ?REST2]
   if RELN of ?OBJECT is $NEGATE
   then
       set parameter :PREVIOUS-QUESTION-TYPE :NEGATIVE
   else
       set parameter :PREVIOUS-QUESTION-TYPE :AFFIRMATIVE
   endif
end S-REQUEST


on INFORMIF
   in= [[RELN INFORMIF]
        ?OTHERS]
   out= [[RELN INFORMIF]
         ?OTHERS]
end INFORMIF


on NO_MATTER_WH
   in= @NO-MATTER-WH(RESTR)
       [[PARM ! X1[[PARM ! X2[]]
                  [RESTR [[RELN 時-TIME]
                          [OBJE ! X2]]]]]
        [RESTR [[RELN NO_MATTER_WH]
                [OBJE ! X1]]]]
   @NO-MATTER-WH = [[PARM ! X[]]
                    [RESTR [[RELN AT-ANY-TIME]
                            [OBJE ! X]]]]
   return @NO-MATTER-WH
end NO_MATTER_WH

;;;
;;; あ [A]
;;;
on ありがとう-THANKING
   in= [[RELN  ありがとう-THANKING]
        ?REST]
   out= [[RELN THANK]
         ?REST]
end ありがとう-THANKING


on ある IN :STATE-OR-ACTION :ACTION
   in= [[RELN ある]
        [OBJE ?OBJE]
        ?OTHERS]
   out= [[RELN 持つ]
         [OBJE ?OBJE]
         ?OTHERS]
end ある


on ある
    in= [[RELN ある]
         [OBJE ?OBJE]
         ?OTHERS]
    out= [[RELN EXIST-BE]
          [OBJE ?OBJE]
          ?OTHERS]
end ある


on ある-1
```

```
    in= [[RELN ある-1]
         [OBJE ?OBJE]
         ?OTHERS]
    out= [[RELN EXIST-BE]
          [OBJE ?OBJE]
          ?OTHERS]
end ある-1

;;;
;;; い [I]
;;;

on いいえ-NEGATIVE in :LANGUAGE :ENGLISH :PREVIOUS-QUESTION-TYPE :NEGATIVE
   in= [[RELN いいえ-NEGATIVE]
        ?REST]
   unset parameter :PREVIOUS-QUESTION-TYPE
   set parameter :ANSWER-TYPE :NEGATIVE
   out= [[RELN YES]
         ?REST]
end いいえ-NEGATIVE


on いいえ-NEGATIVE in :LANGUAGE :ENGLISH :PREVIOUS-QUESTION-TYPE :AFFIRMATIVE
   in= [[RELN いいえ-NEGATIVE]
        ?REST]
   unset parameter :PREVIOUS-QUESTION-TYPE
   set parameter :ANSWER-TYPE :NEGATIVE
   out= [[RELN NO]
         ?REST]
end いいえ-NEGATIVE


on 要る-1
   "AGENT格が埋まっている時は WANT に, それ以外は NECESSARY に訳す"
   in= [[RELN 要る-1]
        ?REST]
   if INPUT has AGEN and AGEN of INPUT is not []
   then
      out= [[RELN NEED-1]
            ?REST]
   else
      out= [[RELN NECESSARY-1]
            ?REST]
   endif
end 要る-1


;;;
;;; う [U]
;;;

on 受け取る
   in= [[RELN 受け取る]
        [AGEN ?AGEN]
        [ORGN ?ORGN]
        [OBJE ?OBJE]
        ?REST]
   out= [[RELN GET]
         [AGEN ?AGEN]
         [ORGN ?ORGN]
         [OBJE ?OBJE]
         ?REST]
end 受け取る

;;;
;;; お [O]
;;;


on 送る-1
   in= [[RELN 送る-1]
        [OBJE @OBJECT<RESTR* PARM>?OBJE-CORE]
        [AGEN ?AGEN]
        [RECP ?RECP]
        ?REST]
   switch semf of ?OBJE-CORE
```

```
   case $TIME  ;; 晩年を送った
        set ?RELATION to $SPEND
   case $HUMAN ;; 見送る
        set ?RELATION to $SEE-OFF
   default
           if input has INSTR ;; instrumental
           then
               INSTR of INPUT =? <PARM* RESTR>[[RELN ?INSTR]
                                                ?REST3]
               switch ?INSTR
               case $銀行振り込み
                   set ?RELATION to $PAY
               endswitch
           else
               set ?RELATION to $SEND
           endif
   endswitch
   out= [[RELN ?RELATION]
         [AGEN ?AGEN]
         [RECP ?RECP]
         [OBJE @OBJECT]
         ?REST]
end 送る-1


on 送る-1 IN :VIEWPOINT :REVERSE
   in= [[RELN 送る-1]
        [AGEN ?AGEN]
        [RECP ?RECP]
        [OBJE ?OBJE]
        ?REST]
   out= [[RELN 受け取る]
         [AGEN ?RECP]
         [ORGN ?AGEN]
         [OBJE ?OBJE]
         ?REST]
end 送る-1

;;;
;;; か [KA]
;;;

on が-MODERATE
   in= [[RELN  が-MODERATE]
        [OBJE ?OBJE]
        ?REST]
   ;; ?OBJE を丁寧な表現で書き換える.
   -> ?OBJE with :POLITENESS :+
   if it is true then
        out= it
   else
        out= [[RELN MODERATE]
              [OBJE ?OBJE]
              ?REST]
   endif
end が-MODERATE

on 会議-1
   in= [[RELN  会議-1]
        ?REST]
   out= [[RELN CONFERNCE]
         ?REST]
end 会議-1

on 会議事務局-1
   in= [[IDEN 会議事務局-1]
        ?REST]
   out= [[IDEN THE-OFFICE-OF-THE-CONFERENCE]
         ?REST]
end 会議事務局-1

;;;
;;; き [KI]
;;;

on 聞く-1 in :VIEWPOINT :REVERSE
```

```
   in= [[RELN 聞く-1]
        [AGEN ?AGEN]
        [ORIG ?ORIG]
        [OBJE ?OBJE]
        ?REST]
   out= [[RELN 伝える]
         [AGEN ?ORIG]
         [RECP ?AGEN]
         [OBJE ?OBJE]
         ?REST]
end 聞く-1


on 聞く-1
   in= [[RELN 聞く-1]
        ?REST]
   out= [[RELN ASK]
         ?REST]
end 聞く-1

;;;
;;; さ [SA]
;;;

on させる-PERMISSIVE in :VIEWPOINT :UNSPECIFIED
   in= [[RELN させる-PERMISSIVE]
        [OBJE ?OBJE]
        ?REST]
   -> ?OBJE with :VIEWPOINT :REVERSE
   if it is true
   then
        out= it
   endif
end させる-PERMISSIVE

on させる-PERMISSIVE
   in= [[RELN させる-PERMISSIVE]
        ?REST]
   out= [[RELN LET]
         ?REST]
end させる-PERMISSIVE

on 参加する-1
   in= [[RELN 参加する-1]
        [AGEN ?AGENT]
        [SLOC ?OBJECT]
        ?REST]
   out= [[RELN TAKE-PART-IN]
         [AGEN ?AGENT]
         [SLOC ?OBJECT]
         ?REST]
end 参加する-1

on 参加料-1
   in= [[RELN 参加料-1]
        ?REST]
   out= [[RELN ATTENDANCE-FEE-1]
          ?REST]
end 参加料-1

;;;
;;; し [SI]
;;;

on 至急に-1
   in= [[RELN 至急に-1]
        ?REST]
   out= [[RELN IMMEDIATELY]
         ?REST]
end 至急に-1

on 質問
   in= [[RELN 質問]
        ?REST]
   out= [[RELN QUESTION]
         ?REST]
```

```
end 質問

on 失礼する-CLOSE_DIALOGUE
   in= [[RELN 失礼する-CLOSE_DIALOGUE]
        [AGEN ?AGEN]
        [RECP ?RECP]
        ?ARCS]
   out= [[RELN GOODBY]
         [AGEN ?AGEN]
         [RECP ?RECP]
         ?ARCS]
end 失礼する-CLOSE_DIALOGUE


on 住所-1
   in= [[RELN 住所-1]
        ?REST]
   out= [[RELN ADDRESS]
         ?REST]
end 住所-1

;;;
;;; す [SU]
;;;

on 既に-1
   in= [[RELN  既に-1]
        ?REST]
   out= [[RELN ALREADY]
         ?REST]
end 既に-1


on する-1
   in= [[RELN する-1]
        ?OTHERS]
   out= [[RELN DO]
         ?OTHERS]
end する-1


on する-1 IN :EVENT-OR-OBJECT :UNSPECIFIED
   in= [[RELN する-1]
        [OBJE <PARM* RESTR> ?OBJE]
        ?OTHERS]
   -> ?OBJE with :EVENT-OR-OBJECT :EVENT
   if it is true
   then
        add ?OTHERS to it
        out= it
   endif
end する-1

;;;
;;; そ [SO]
;;;

on それでは-1
   in=  [[RELN それでは-1]
         [OBJE ?OBJE]]
   out= [[RELN THEN]
         [OBJE ?OBJE]]
end それでは-1

;;;
;;; た [TA]
;;;

on だ-IDENTICAL
   in= [[RELN  だ-IDENTICAL]
        [OBJE ?OBJE]
        [IDEN ?IDEN]]
   out= [[RELN  BE-IDENTICAL]
         [OBJE ?OBJE]
         [IDEN ?IDEN]]
end だ-IDENTICAL
```

```
on だ-STATEMENT
   in= [[RELN だ-STATEMENT]
        [OBJE [[PARM ! X[]]
               [RESTR [[RELN そう-1]
                       [OBJE ! X]]]]]
        ?REST]
   out= [[RELN RIGHT]
         [OBJE [[PARM ! X1[]]
                [RESTR [[RELN THAT]
                        [OBJE ! X1]]]]]
         ?REST]
end だ-STATEMENT


on だ-STATEMENT
   in= [[RELN だ-STATEMENT]
        ?REST]
   out= [[RELN COPULA-BE]
         ?REST]
end だ-STATEMENT


on 登録費-1
   in= [[RELN 登録費-1]
        ?REST]
   out= [[RELN REGISTRATION-FEE-1]
         ?REST]
end 登録費-1


on たい-DESIRE IN :POLITENESS :+
   in= [[RELN たい-DESIRE]
        ?REST]
   out= [[RELN WOULD-LIKE]
         ?REST]
end たい-DESIRE


on たら-CONDITIONAL
   in= [[RELN たら-CONDITIONAL]
        ?REST]
   out= [[RELN IF-CONDITIONAL]
         ?REST]
end たら-CONDITIONAL

;;;
;;; つ [TU]
;;;
on 伝える
   in= [[RELN 伝える]
        ?REST]
   out= [[RELN TELL]
         ?REST]
end 伝える


ON 伝える IN :VIEWPOINT :REVERSE
   in= [[RELN 伝える]
        ?REST]
   out= [[RELN 聞く]
         ?REST]
end 伝える

;;;
;;; て [TE]
;;;

on 手続-1 IN :EVENT-OR-OBJECT :EVENT
   in= [[RELN 手続-1]
        ?REST]
   out= [[RELN 手続きする]]
end 手続-1


on 手続き-1
   in= [[RELN 手続き-1]
```

```
        ?REST]
   out= [[RELN PROCEDURE]
         ?REST]
end 手続き-1


on 手続きする
   in= [[RELN 手続きする]
        ?REST]
   out= [[RELN PROCCED]
         ?REST]
end 手続きする


ON 手続きする
   in= [[RELN 手続きする]
        [INST @INST<PARM* RESTR RELN>?INSTRMENT]
        ?REST]
   switch ?INSTRMENT
   case $登録用紙-1
        out= [[RELN SUBMIT]
              [OBJE @INST]
              ?REST]
   endswitch
end 手続きする


ON てもらう-RECEIVE_FAVOR
   in= [[RELN  てもらう-RECEIVE_FAVOR]
        ?REST]
   out= [[RELN RECEIVE-FAVOR]
         ?REST]
end てもらう-RECEIVE_FAVOR


on では-1
   in= [[RELN では-1]
        ?REST]
   out= [[RELN THEN-1]
         ?REST]
end では-1

;;;
;;; と [TO]
;;;
on と-COORDINATE
   in= [[RELN と-COORDINATE]
        ?REST]
   out= [[RELN AND]
         ?REST]
end と-COORDINATE

on 登録用紙-1
   in= [[RELN 登録用紙-1]
        ?REST]
   out= [[RELN REGISTRATION-FORM]
         ?REST]
end 登録用紙-1


on どう-1
   in= [[RELN どう-1]
        ?REST]
   out= [[RELN HOW]
         ?REST]
end どう-1


on どうも-1
   in= [[RELN どうも-1]
        ?REST]
   out= [[RELN FEELING-EMPHASIS]
         ?REST]
end どうも-1

on 時-TIME
```

```
   in= [[RELN 時-TIME]
        ?REST]
   out= [[RELN WHEN]
         ?REST]
end 時-TIME

;;;
;;; な [NA]
;;;

on なくてはならない
   in= [[RELN なくてはならない]
        [OBJE ?OBJE]]
   out= [[RELN MUST]
         [OBJE ?OBJE]]
end なくてはならない

ON 名前-1
   in= [[RELN 名前-1]
        ?REST]
   out= [[RELN NAME]
         ?REST]
end 名前-1

;;;
;;; ね [NE]
;;;

on 願う-REQUEST
   in= [[RELN 願う-REQUEST]
        ?REST]
   out= [[RELN REQUEST]
         ?REST]
end 願う-REQUEST

;;;
;;; は [HA]
;;;


on はい IN :LANGUAGE :ENGLISH :PREVIOUS-QUESTION-TYPE :NEGATIVE
   in= [[RELN  はい]
        ?REST]
   unset parameter :PREVIOUS-QUESTION-TYPE
   out= [[RELN NO]
         ?REST]
end はい


ON はい IN :LANGUAGE :ENGLISH :PREVIOUS-QUESTION-TYPE :AFFIRMATIVE
   in= [[RELN はい]
        ?REST]
   unset parameter :PREVIOUS-QUESTION-TYPE
   out= [[RELN YES]
         ?REST]
end はい

ON ばよい-SHOULD
   in= [[RELN ばよい-SHOULD]
        ?REST]
   out= [[RELN SHOULD]
         ?REST]
end ばよい-SHOULD


on ばよい-SHOULD
   "どうすればよい -> WHAT SHOULD I DO"
   in= [[RELN  ばよい-SHOULD]
        [AGEN ?AGENT]
        [OBJE @する[[RELN  する-1]
                    [AGEN ?AGNET]
                    [MANN @どう[[PARM ! X03[]]
                                [RESTR [[RELN どう-1]
                                        [OBJE ! X03]]]]]
                [OBJE []]]]]
   @どう = [[PARM ! X03[]]
```

```
            [RESTR [[RELN WHAT-1]
                    [OBJE ! X03]]]]
   @する = [[RELN  する-1]
            [AGEN ?AGNET]
            [OBJE @どう]]
   out= [[RELN SHOULD]
         [AGEN ?AGENT]
         [OBJE @する]]
end ばよい-SHOULD

;;;
;;; ひ [HI]

on 必要だ-1
   in= [[RELN 必要だ-1]
        ?REST]
   out= [[RELN NECESSARY-1]
         ?REST]
end 必要だ-1

;;;
;;; ま [MA]
;;;

on 先ず-1
   in= [[RELN 先ず-1]
        ?REST]
   out= [[RELN FIRST]
         ?REST]
end 先ず-1

on まだ-1 IN :LANGUAGE :ENGLISH :ANSWER-TYPE :NEGATIVE
   in= [[RELN まだ-1]
        ?REST]
   if input =? @BE (OBJE RESTR)[[RELN COPULA-BE]
                                [OBJE [[PARM ! X1[]]
                                       [RESTR [[RELN まだ-1]
                                               [OBJE ! X1]]]]]]
   then
       unset parameter :ANSWER-TYPE
       REPLACE @BE WITH [[RELN NOT-YET]]
   endif
end まだ-1

on まだ-1 IN :LANGUAGE :ENGLISH :ANSWER-TYPE :AFFIRMATIVE
   in= [[RELN まだ-1]
        ?REST]
   unset parameter :ANSWER-TYPE
   out= [[RELN STILL]
         ?REST]
end まだ-1

;;;
;;; も [MO]
;;;

on もう-1
   in= [[RELN もう-1]
        ?REST]
   out= [[RELN ALREADY-1]
         ?REST]
end もう-1

on もしもし-OPEN_DIALOGUE
   in= [[RELN もしもし-OPEN_DIALOGUE]
        ?REST]
   out= [[RELN HELLO-OPEN_DIALOGUE]
         ?REST]
end もしもし-OPEN_DIALOGUE


on 申込む-1
   "SELECT WORD BY OBJE CASE FILLER."
   in= [[RELN 申込む-1]
        [OBJE @OBJECT<PARM* RESTR RELN>?OBJE-RELN]
        ?REST]
```

```
   if input has OBJE
   then
      switch ?OBJE-RELN
      case $試合                                    ; 試合
           out= [[RELN CHALLENGE]
                 [OBJE @OBJE]
                 ?REST]
      case $結婚                                    ; 結婚
           out= [[RELN PROPOSE]
                 [OBJE @OBJE]
                 ?REST]
      default
           out= [[RELN APPLY]
                 ?REST]
      endswitch
   endif
end 申込む-1

on 申込む-1
"SELECT WORD BY SLOC CASE FILLER."
   in= [[RELN 申込む-1]
        [SLOC @SLOC<PARM* RESTR>[[RELN ?SLOC-RELN]
                                 ?SLOC-REST]]
        ?REST]
   if input has OBJE then
      fail
   endif
   switch ?SLOC-RELN
   case $会議-1                                  ; 会議
        out= [[RELN MAKE]
              [OBJE [[PARM ! X[]]
                     [RESTR [[RELN REGISTRATION]
                             [OBJE ! X]]]]]
              [SLOC @SLOC]
              ?REST]
  endswitch
end 申込む-1

on 持つ-1
   in= [[RELN 持つ-1]
        ?REST]
   out= [[RELN HAVE]
         ?REST]
end 持つ-1


on 持つ-1 IN :STATE-OR-ACTION :STATE
  in= [[RELN 持つ-1]
       [AGEN []]
       [OBJE ?OBJECET]
       ?REST]
  out= [[RELN ある]
        [OBJE ?OBJECT]
        ?REST]
end 持つ-1

;;;
;;; よ [YO]
;;;

on よう-GUESS
   in= [[RELN よう-GUESS]
        ?REST]
   out= [[RELN GUESS]
         ?REST]
end よう-GUESS


on 用紙-1
   in= [[RELN 用紙-1]
        ?REST]
   out= [[RELN FORM-1]
         ?REST]
end 用紙-1

;;;
```

```
;;; わ [WA]
;;;

on 分かる-CONFIRMATION
   in= [[RELN 分かる]                    ;; 分かる
        [EXPR ?EXPR]
        ?REST]
   out= [[RELN SEE]                      ;; I SEE.
         [EXPT ?EXPR]
         ?REST]
end 分かる-CONFIRMATION


on 分かる-1
   in= [[RELN 分かる-1]
        [EXPR ?EXPR]
        ?REST]
   out= [[RELN UNDERSTAND]
         [EXPR ?EXPR]
         ?REST]
end 分かる-1

on 分かる-1 IN :EVENT-OR-OBJECT :OBJECT
   in= @SITUMON (RESTR OBJE)
            [[PARM ! X1[[PARM ! X2[]]
                       [RESTR [[RELN 点-1]
                               [OBJE ! X2]]]]]
             [RESTR [[RELN NEGATE]
                     [OBJE [[RELN 分かる-1]
                            [EXPR []]
                            [OBJE ! X1]]]]]]
    @SITUMON = [[PARM ! X[]]
                [RESTR [[RELN 質問]
                        [OBJE ! X]]]]
   return @SITUMON
end 分かる-1

on 分かった-CONFIRMATION
   in= [[RELN 分かった-CONFIRMATION]
        ?REST]
   out= [[RELN I-SEE-CONFIRMATION]
         ?REST]
end 分かった-CONFIRMATION
```

索引