ICOT Technical Report: TR-362

TR-362

# 論理型形態素解析LAX

杉村領一、赤坂宏二、久保幸弘
松本裕治、佐野　洋

March, 1988



ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

# 論理型形態素解析 *LAX*

杉村領一、赤坂宏二、久保幸弘
(財)新世代コンピュータ技術開発機構
松本裕治　　佐野洋
電子技術総合研究所　　(株)東芝

## ABSTRACT

本報告では、並列実行可能な日本語の形態素解析手法について述べる。本手法では、形態素辞書を決定的に動作する論理型プログラムへ変換し、これにより形態素解析を行なう。形態素文法は辞書に記述される。文法のクラスは正規言語である。辞書はTRIE構造になっており、論理型言語のインデクスサーチメカニズムを形態素検索に効果的に用いている。解析には解探索空間を陽に表現するレイヤードストリームを応用している。解析終了時には、全ての解析結果が縮退された形で出力される。解析結果は、本方式とは別途開発された並列実行可能な構文解析プログラムSAXへ直接入力できる。本方式は未登録語や未登録記号をリニアオーダで処理できる。また、逐次論理型言語上での本方式の性能評価についても報告する。

## 1.はじめに

並列実行可能な日本語の形態素解析手法について述べる。本手法では、形態素辞書を①ESP、Prolog等の逐次論理型言語、または、②GHC[1]等の並列論理型言語で記述された、決定的に動作するプログラムへ変換し、変換されたプログラムによって、形態素解析を行なう[2]。

形態素文法は、形態素辞書の各エントリ毎に記述される。文法は非決定性有限オートマトン[3]で示され、解探索にはレイヤードストリーム[4]を応用し、逐次及び並列論理型言語上での効率的な解析手法を実現する。なお、文法の非決定性により解析データに曖昧さが出る場合には、これを縮退させる。

2では、まず形態素辞書について述べる。形態素辞書には、各形態素の①表層、②形態素カテゴリー、並びに、形態素文法として、形態素間の連接可能性を調べるための連接素性(③前方連接素性と④後方連接素性)を記述する。連接素性は、非決定性有限オートマトンの遷移状態として用いられる。

辞書はTRIE構造[5]になっており、論理型言語のインデクスサーチメカニズムを形態素検索に効果的に用いている。

3では、本方式の基本的なアルゴリズムについて述べる。これは、基本的に2型文法の解析に用いられるEarlyのアルゴリズム[6]や、上昇型チャートパージング[7]アルゴリズムで3型の文法を解析するのに等しい。

4では、基本アルゴリズムの3型文法への最適化を図った本方式の実現方式について述べる。解析時のプロセス数の最適化と解析データの縮退が実現される。一つの入力文に対して、複数の解析結果が得られた場合には、解析終了時に全ての候補が縮退された形で出力される。また、未登録語や未登録記号の処理が実現される。解析結果は、本方式とは別途開発された並列実行可能な構文解析プログラムSAX[8]へ入力される[9]。

5では、逐次言語上へのインプリメントのための最適化について述べ、6で、この性能評価について報告する。

## 2.形態素辞書

### 2.1.形態素辞書概要

形態素辞書には、一つの形態素について、4種類の情報を記述する。

まず、①「かな」もしくは「漢字」で示される表層、②各表層の持つ形態素カテゴリを記述する。

解析文が「かな」列または「ローマ字」列の場合には、解釈の曖昧さを絞りこむために、「文節数最小法」「最長一致法」[10]等のヒューリスティクスを用いるべきである。しかし、本方式は「漢字かな混じり文」の形態素解析を主眼に置く。よって、これらのヒューリスティクスは用いず、形態素間の連接規則[11]のみを形態素文法として用いて、曖昧さを減少させる。そこで、形態素辞書には、前記①②に加えて、形態素間の連接規則を用いるための③前方連接素性、④後方連接素性を記述する。以降、関数Zを形態素からその前方連接素性への関数、関数Kを形態素からその後方連接素性への関数とする。

形態素解析LAXと構文解析SAXを同時に用いれば、構文解析時のトップダウン予測を形態素解析時

---

Title : Logic based Lexical Analyzer LAX
Authors : Ryoichi Sugimura, Koji Akasaka, Yukihiro Kubo
Institute for New Generation Computer Technology (ICOT)
Yuji Matsumoto
Electrotechnical Laboratory (ETL)
Hiroshi Sano
Toshiba Corporation

の曖昧を減らすために用いる事が可能であるが、本論では触れない。

## 2.2 連接規則

以下、形態素解析文法として用いられる形態素間の連接規則について述べる。

基本的に連接規則は、連接する形態素の連接可能性を示す述語で表される。辞書引きにより入力文に見つけられた、隣接する形態素nと形態素n＋1の連接規則を以下に示す。

連接規則(形態素n,形態素n＋1)
＝(K(形態素n),Z(形態素n＋1))∈連接可能集合
＝{T,F}
T:形態素nと形態素n＋1は連接可能
F:形態素nと形態素n＋1は連接不可能

連接規則の実現方法としては、関数Kと関数Zに異なる値域R(K)、R(Z)を割り当て、連接可能な素性の組(X,Y) where X∈R(K),Y∈R(Z) を要素とする連接可能集合を表として持つ方式[11]がある。

現在LAXでは、関数K、関数Zを修正し(以下K'、Z'とする)、以下のように連接規則を定義している。まず、K'、Z'を以下のように定義する。Aは形態素である。

$$Z'(A) = X$$
$$K'(A) = \{Y_1, Y_2, \ldots, Y_n\}\ (n \geqq 1)$$
where $X, Y_1, Y_2, \ldots, Y_n \in$ 連接素性集合

上記の関数K', Z' を用いて連接規則を以下のように定義する。

連接規則(形態素n,形態素n＋1)
＝Z'(形態素n＋1)∈K'(形態素n)
＝{T,F}

現在LAXでは、例えば、図1の例のように、複数の状態遷移表を用いる事が可能でり、連接素性は、状態遷移表名とこの表における状態名の組として以下のように示す。

状態遷移表名([状態名]) ＝ 連接素性

複数の状態遷移表を許す理由は、日本語文法を、構造主義言語学的なアプローチに従って記述する際に、有用であるためである[12]。具体的に、どのような状態遷移表を用いるかは、別途[13]に詳しいので本論では触れない。

図1の例では、文節「痛めるな」の複数の状態遷移表を用いた解析過程を示している。

まず、語基「痛」が『文節先頭』状態で受け付けられ、次状態が『派生表-動詞派生接辞め』(動詞派生)や『派生表-イ系ア』(形容詞派生)などになる。「め」は『派生表-動詞派生接辞め』状態で受け付けられ、次状態は『派生表-ウ系弱変化』(動詞派生)と『終了状態』(文節資格を持ち構文解析を呼び出す)などになる。「る」は、『派生表-ウ系弱変化』状態で受け付けられ、次状態は『派生表-現在形』や『陳述表-終助詞な』などになる。「な」は『陳述表-終助詞な』状態で受け付けられ、『終了状態』になり文節として認識される。

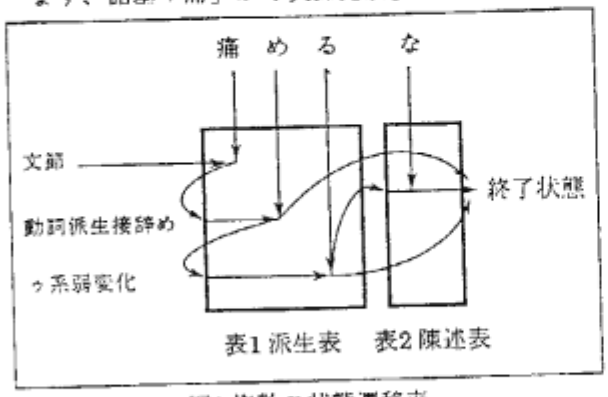


図1 複数の状態遷移表

## 2.3 辞書記述形式

形態素辞書の記述形式の概略は、以下の通りである。

```
begin(カテゴリ名)
表層 ::  遷移表名a([状態名,..,]),                    ㋑
         &&  [遷移表名b([状態名,..,]),..]            ㋺
         $$  [[in(I),共通処理],                       ㋩
              遷移表名b(状態名,                       ㋥
                  [固有処理,send(O)],                 ㋭
                  [固有処理,call-sax(O2)]),..].       ㋬
end(カテゴリ名).
```

まず、同一の形態素カテゴリーを持つ形態素の定義は、beginとendの間にまとめる。

㋑で、形態素の表層とその前方連接素性を書く。::は区切り記号である。㋺では区切り記号&&に続いて後方連接素性を書く。

区切り記号$$以降は意味処理を記述する。形態素意味処理の詳細は別途報告したいので、ここでは概略だけを紹介する。㋩では区切り記号$$に続いて、後方に連接する形態素に依存しない意味処理を記述する。述語in(I)は前方に連接する形態素から送られてくる意味情報をIとして受け取るために用いる。㋥㋭㋬には、後方に連接する形態素に依存し、かつ、後方の形態素では処理できない処理を記述する。㋥には、後方連接素性を記述する。㋭には後方へ送る意味情報処理を記述する。述語send(O)は後方に連接する形態素へ意味情報Oを送るために用いる。㋬には構文解析呼び出しをcall-sax($O_2$)により記述する。

図2に図1の例に対応した形態素辞書の実際の記述例を示す。図中各行の右側の記号は、上記記述形式の概略で用いた記号と同じ意味を持つ。また、下線で示した部分が、図1の状態遷移に直接関係している。図中の述語tmerge(A,B,C)は、Aを変化させずにBへAの構造をマージし、結果をCとして返す。意味処理部分にはCIL[14]を用いている。

## 3.基本アルゴリズム

まず基本的な実現方式を示す。基本アルゴリズムは、2型の文法により構文解析を行うEarleyのアルゴリズムや、上昇型チャートパージングアルゴリズムで3型の文法を解析するのに等しい。具体的には、図3に示すようなテーブル状(well-formed substring table)に配置された多重プロセスを用いて処理が進む。

図3では、形態素解析すべき文字列は「いためるな」である。後の説明での都合上、ひらがな入力を例とする。文字列には説明の便宜上先頭から1、2の順に番号を打っておく。

'p12~p67' は形態素の検索プロセスを示す。'F1~F6'はテーブルの行を示す。

```
begin(形容語基).
痛   ::   end([文節])
     &&[派生([イ系ア,ウ系強変化マ,
           動詞派生接辞め,...])
     $$ [[L0 = {表層/痛}],
        派生(イ系ア,
            [tmerge(L0,{格/{agt/X, comp/Y}}, L1),
             send(L1)],[]),
        派生(動詞派生接辞め,
            [tmerge(L0,格/{agt/X, obj/Y}}, L2),
             send(L2)],[])].
end(形容語基).
begin(動詞派生接辞).
め   ::   派生([動詞派生接辞め])
     &&[派生([ウ系弱変化,...]),end([文節])]
     $$ [[in(I)],
        派生(ウ系弱変化,[send(I)],[]),
        end(__,[],
            [tmerge(I,{連用型/連用形,品詞/動詞},L),
             call-sax(L)]).
end(動詞派生接辞).
begin(ウ系弱変化活用助辞).
る   ::   派生([ウ系弱変化,...])
     &&[派生([現在形]),隣述([終助詞な,...]),
        end([文節,。])]
     $$ [[in(I),tmerge(I,{時制/現在,丁寧/no},L0)],
        派生(現在形,[send(I)],[]),
        隣述(終助詞な,[send(I)],[]),
        end(文節,[],
            [tmerge(I,{ムード/主張},L),
             call-sax(L)]).
end(ウ系弱変化活用助辞).
begin(終助詞).
な   ::   隣述([終助詞な])
     &&[隣述([終接辞よ,...]),end([文節,。])]
     $$ [[in(I),tmerge(I,{ムード/禁止},L)],
         隣述(終接辞よ,[send(L)],[]),
         end(__,[],[call-sax(L)])].
end(終助詞).
begin(用言語基).
傷   ::   end([文節])
     &&[派生([ウ系強変化マ,動詞派生接辞め,...])
     $$ [[L0 = {表層/傷}],
        派生(ウ系強変化マ,
            [tmerge(L0,{格/{agt/X}}, L1),
             send(L1)],[]),
        派生(動詞派生接辞め,
            [tmerge(L0,格/{agt/X, obj/Y}}, L2),
             send(L2)],[])].
炒め   ::   end([文節])
     &&[派生([ウ系弱変化マ,...])
     $$ [[L0 ={表層/炒め}],
        派生(ウ系弱変化マ,
            [tmerge(L0,格/{agt/X, obj/Y}}, L2),
             send(L2)],[])].
end(用言語基).
begin(体言語基).
板目   ::   end([文節])
     &&[派生([体言語基,...]),
         隣述([終接辞よ,...]),end([文節,。])]
     $$ [[L = {表層/板目}],
        派生(体言語基,
            [send(L)],[]),
         隣述(終接辞よ,[send(L)],[]),
         end(__,[],[call-sax(L)])].
end(体言語基).
```

図2 形態素辞書記述例

検索プロセス'pRC' は番号Rに対応する文字から番号Cに対応する文字の手前までの部分文字列に対応する表層を持った形態素を形態素辞書から検索する。例えば、p23 は2の位置の「た」に対応する表層を持った形態素を形態素辞書から検索する。 検索は'p12~p67'各々によって並列に行う事が可能である。

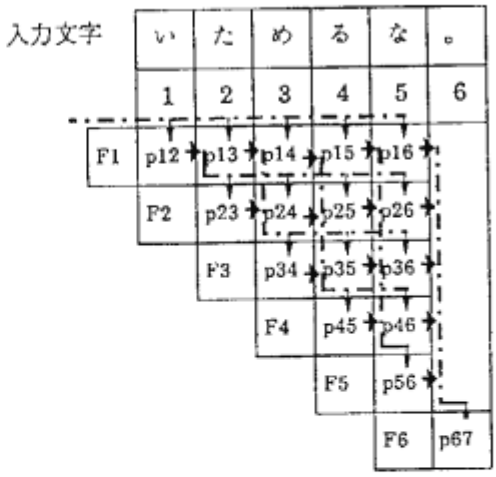


図3 基本アルゴリズムの処理の流れ

検索プロセスは検索結果を流すストリームで互いに連結されている。第C列の'p?C' で示されたプロセスの出力はマージされて第C行に示された各プロセスの入力となる。例えば、第4列 p14, p24, p34, の出力は、マージされて第4行のプロセスp45, p46 の入力となる。p14, p24, p34は3の位置の「め」で終わる文字列を検索し、p45, p46 は4の位置の「る」で始まる文字列を検索する。そこで、これらのプロセスをストリームで連結することにより、連続した文字列の形態素解析が可能になる。

アルゴリズムをまとめると以下の様になる。

1)プロセスの駆動制御

① 入力文字(漢字かな文字の長さN+句読点等1文字)に対応して、((N+1)*N/2) +1個のプロセスを(並列に)駆動する。

((N+1)*N/2) 個のプロセスは、入力漢字かな文字の全ての部分文字列に対応する。また文端の句読点1文字に対応して1個のプロセスを駆動する。

駆動する各プロセスは、図3の一点鎖線で示されたように、ストリームで連結しておく。つまり、前述の様に第C列の'p?C' で示されたプロセスの出力はマージされて第C行に示されたプロセス'pC?'各々の入力となる。

② 第1行のプロセスp1? には文頭であることを示す後方連接素性をストリームを用いて流す。

③ p(N+1)(N+2)の出力を解析結果とする。

2)駆動されたプロセスの動き

1の①で駆動されたプロセスは以下の様に処理を行なう。

① 入力ストリームから検索データDinを受け取る。検索データが無い場合にはデータが来るまで待つ。

② 受け取ったデータがストリームの終端を示す[]ならば、[]を出力して処理を終わる。[]でない場合には③へ進む。

③ 割り当てられた文字列Drf(pNK の場合にはNからK-1までの部分文字列)を検索する。

Drfがあれば④へ処理を進める。なければ、処理を終わる。

④ DinのDrfへの連接可能性をDinの後方連接素性とDrfの前方連接素性を基に決定する。
連接可能な場合には、DinにDrfを連接させた文字列Dfnを作り、Dfnの後方連接素性をDrfの後方連接素性として出力する。続いて①へ。
連接不可能な場合にも①へ。

以上の実現方法により並列形態素解析は行えるが、本方式には以下に述べるように実用に供するには問題点がある。

【問題点】

1) プロセス駆動の段階で(N+1)*N/2 + 1個のプロセスを入力文字列の長さに応じて駆動する必要がある。ところが、駆動されたプロセスの内、割り当てられた部分に対応する文字列を見付けることのできるプロセス数は小さい事が容易に予想される。つまり、不要なプロセスを、大量に生み出し処理効率が悪い。
2) 本アルゴリズムは未登録語の検出が考慮されていない。
3) 解析途中で解析データを共用して問題が無い場合でも、全ての解を全く別別に求めて出力する。また、以下の問題がある。
例えば、プロセスpXYで、前方から来る連接可能なデータがM個だとし、pXYで見つかった形態素がL個とすると、最悪の場合、後方へはM*L個のデータが流れる。各プロセスにおいて以上の処理を行うと、データストリームの最後尾では最悪の場合Π(プロセスの見つけた形態素数)個のデータが流れる。計算量もこれに比例する。これに対処するには、各プロセスが、前方からくるデータを、そのプロセスで見つかった形態素L個各々についてまとめあげ、後方へは、最高でもL個のデータを送るようにする必要がある。

以上3点の問題点に鑑み、1) 最小限のプロセスだけを動的に駆動し、2)未登録語の処理を少ないコストで行ない、3) 解析データの共用を行ない結果を縮退した形で得る事の出来るよう改良された実現方式について、4章で述べる。

## 4.改良アルゴリズム

改良アルゴリズムについて、以下、4.1.プロセスの効率的な駆動方法、4.2.プロセス間の通信方法、の順に例を用いて説明し、4.3でまとめを行なう。

### 4.1.プロセスの効率的な駆動方法

ここでは、第3節で述べた基本アルゴリズムの問題点の1)を解決する、プロセスの効率的な駆動方法について述べる。

解析開始時には、各文字に一つのプロセスを対応させて、これを駆動する。以下図4を例に説明を行なう。文法は図2を用いるが、説明の都合上辞書の表層はひらがなで与えられているとする。

図4において、'dict'は初期状態で駆動するプロセスである。文字列が終端記号を除いてNの長さの場合、N+1個の'dict'を駆動する(例では6個の'dict'を駆動する)。'dict'には、それの置かれた位置以降の文字列を検索する機能を持たせる。第1行(F1)に置かれた'dict'は、「い」で始まる文字列を検索する。形態素辞書に「いた」(図2の例では「痛」「傷」)が有る場合には、前記基本アルゴリズムの中で駆動されたプロセス'p13'に相当するサブプロセスを'dict'が駆動する。形態素辞書に「いため」(図2の例では「炒め」「板目」)がある場合には、前記と同様に'p14'に相当するプロセスを'dict'が駆動する。形態素辞書に「いためる」という表層文字列が無い場合には'p15'に対応するプロセスは駆動されない。

| 入力文字 | い | た | め | る | な | 。 |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| F1 | dict | p13 | p14 | | | |
| F2 | | dict | | | | |
| F3 | | | dict | p35 | | |
| F4 | | | | dict | | |
| F5 | | | | | dict | |
| F6 | | | | | | dict |

図4 改良アルゴリズムで駆動されるプロセス

'dict'とこれに駆動されるサブプロセスは予めプログラムとして形態素辞書からコンパイルしておくことが可能である。例えば、図2の形態素辞書に加えて、体言語基「い(胃)」、サ変語基「いたく(委託)」を持つ形態素辞書図5が与えられているとする。

| [表層] | [カテゴリ] |
|---|---|
| い | 体言語基(胃) |
| いた | 形容語基(痛) |
| いた | 用言語基(傷) |
| いため | 用言語基(炒め) |
| いため | 体言語基(板目) |
| いたく | サ変語基(委託) |

図5 形態素辞書例

上記の辞書はTRIE構造を用いて例えば図6の様に表現出来る。

```
[い   体言語基(胃)
    [た   形容語基(痛), 用言語基(傷)
        [め   用言語基(炒め), 体言語基(板目)]
        [く   サ変語基(委託)]
    ]
]
```

図6 辞書のTRIE構造

更に図6の構造を図7のような'dict'の辞書検索基本アルゴリズムのプログラムに変換することは容易である。但し、説明の簡略化のために、図7に示したプログラムは辞書検索のみを行う部分だけを切り出している。本アルゴリズムはPrologで記述されているが、これは本質的ではなく他の言語に置き換えることは可能である。

下記の'dict'は、それの置かれた位置以降の文字列

を論理変数Rにより参照することが可能である。検索用の文字列は、'dict'の第一引数となっている。'dict'の第3引数Iには、辞書検索の結果が返る。

```
dict(い,R,I)  :- R=[H|RR],
                 I=[[体言語基(胃)]|Ir],
                 い(H,RR,Ir).
い(た,R,I)    :- R=[H|RR],
                 I=[[形容語基(痛)],
                    [用言語基(傷)]|Ir],
                 いた(H,RR,Ir).
いた(め,R,I) :- I=[[用言語基(炒め)],
                    [体言語基(板目)]].
いた(く,R,I) :- I=[[サ変語基(委託)]].
```

図7 TRIE構造による辞書検索プログラム

図7で例えば'dict'に、[い,ほ,め,る]という文字列が渡っていると、dict(い,[ほ,め,る])のコールに成功することで、「い(体言語基-胃)」の検索は成功するが、く(ほ,[き])のコールには失敗して、以降、「いほ」「いほめ」「いほめる」を検索する不要なプロセスは駆動されない。

ここで注意しておきたいのは、Prolog等の論理型言語はコンパイルされた場合Clauseの呼出しがハッシングを用いて行われる事である。辞書引きがPrologの埋めこみ機能により無理なく多段ハッシュで実行され、これにより高速な解析速度を本アルゴリズムは持つことになる。

以上の様に形態素辞書を予めコンパイルし、検索に用いる事により、必要最小限のプロセス駆動で処理を進める事ができる。

## 4.2.プロセス間の通信

ここでは、まずデータの'dict'間の受け渡し方法を示し、続いて、第3節で述べた基本アルゴリズムの欠点の3)を解決する、検索データの縮退について述べる。

前記4.1で述べたプロセス'dict'は駆動時に検索データを流すストリームで図8の様に結合されている。

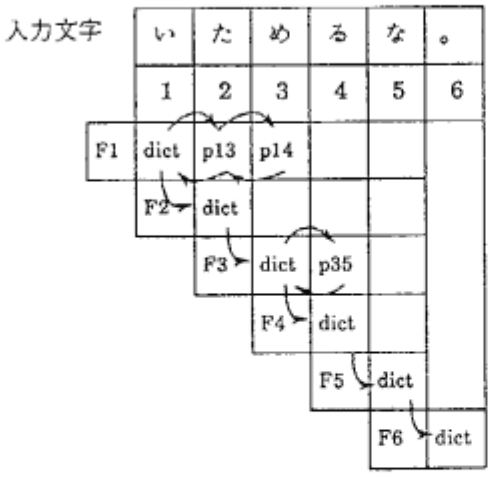


図8 データストリームの流れ

データストリームは図9のような構造を持っている。これは、有限状態オートマトンの動作履歴を陽に表現したものになっており、解探索空間を陽に表現するレイヤードストリームの応用となる。

```
データストリーム  ::= '[' データ並び ']'
データ並び        ::= データ ',' データ並び
データ並び        ::= データ
データ            ::= '[' 検索開始位置番号 ','
                      後方連接素性並び ','
                      形態素解析結果 ']'
検索開始位置番号  ::= N(1≦N≦入力文字数)
後方連接素性並び  ::= [後方連接素性,..]
形態素解析結果    ::= '[' 形態素表層 ','
                      形態素カテゴリー ','
                      形態素解析結果並び ']'
形態素解析結果並び ::= []
形態素解析結果並び ::= 形態素解析結果,
                      形態素解析結果並び
```

図9 データストリームの構造

改良アルゴリズムでは、プロセス'dict'に入力されるデータは必ずしもそれが処理すべき検索データにはならない。例えば図8で'p14'の検索結果は第4行の'dict'に渡るまでに第2行と第3行の'dict'を経由する。このため、改良アルゴリズムでは、データは、それが処理されるべきプロセスを示す[検索開始位置番号]を陽に持っている。

ここで、注意したいのは、レイヤードストリームを用い、2型の文法を扱うSAXでは、文法ルールの間に解探索点を示す識別子を持たせたが、3型の文法を扱うSAXでは文法を直接終端記号の位置に記述するだけでよいため、文法ルール間の識別子が不要になる事である。これにより、3型の文法への最適化が図られる。

上記に呼応して、プロセス'dict'には各々駆動時点で、位置付けられた場所に対応した番号を割り付ける。例えば第1行(F1)の'dict'には番号1を、第3行(F3)に位置付けられた'dict'には3を駆動時に渡す。第1行(F1)の'dict'にはオートマトンの初期状態として特別に以下の構造を渡す。end([文頭,文節])は、文頭連接素性である。

```
[1,end([文頭,文節]),[]]
```

行F1の'dict'が文字「い」の検索に成功すると、当該「い」の前方連接素性と文頭連接素性により「い」が文頭に連接可能かどうかを'dict'は調べる。連接可能なときには、'dict'は「い」の次に検索すべき文字の位置を示す番号2を「い」に添えて以下のようにストリームに出力する。

```
[2,「胃」の後方連接素性,[胃,体言語基,[]]]
```

'p13'が文字「いた」の検索に成功すると「いた」に番号3を添えて以下のデータをストリームに出力する。

```
[3,[派生([イ系ア,ウ系強変化マ,動詞派生接辞め])],
        [痛,形容語基,[]]]
[3,[派生([ウ系強変化マ,動詞派生接辞め])],
        [傷,用言語基,[]]]
```

'p14'は以下のデータをストリームに出力する。

```
[4,[派生([ウ系弱変化マ])],[炒め,用言語基,[]]]
[4,[派生([体言語基]),陳述([終接辞よ]),
        end([文節,。])],
        [板目,体言語基,[]]]
```

第1行(F1)の'dict'及び'p13','p14'の検索結果は、マージされて、第1行(F1)の'dict'から第2行(F2)の'dict'へ渡される。

第2行(F2)の'dict'は、入力ストリームから検索開始位置番号が2のデータを取り出し、自分が検索したデータ(例では「た」)と連接させ、出力ストリームに流す(例では「胃」に「た」が連接しないので何も流れない)。検索開始位置番号が3以上のデータ(例では「痛」「傷」「炒め」「板目」)をF1の'dict'から受け取った場合には、これを直接出力ストリームに流し、第3行の'dict'に渡す。

以上のようにして、データがストリームを流れる。次にデータの縮退について述べる。例えば図10のような、検索位置3におけるデータの流れと連接可能性を考える。

```
[入力データストリーム]
[
[3,[派生([イ系ア,ウ系強変化マ,動詞派生接辞め])],
                    [痛,形容語基,[]]]
[3,[派生([ウ系強変化マ,動詞派生接辞め])],
                    [傷,用言語基,[]]]
[4,[派生([ウ系弱変化マ])],[炒め,用言語基,[]]]
[4,[派生([体言語基]),陳述([終接辞よ]),
                    end([文節,。])],
                    [板目,体言語基,[]]]
]
[位置2の形態素]
め :: 派生([動詞派生接辞め])
   && [派生([ウ系弱変化]),end([文節])]

[出力データストリーム]
[
[4,[派生([ウ系弱変化]),end([文節])],
      [め,動詞派生接辞,
        [[痛,形容語基,[]],[傷,用言語基,[]]]]
[4,[派生([ウ系弱変化マ])],[炒め,用言語基,[]]]
[4,[派生([体言語基]),陳述([終接辞よ]),
      end([文節,。])],
      [板目,体言語基,[]]]
]
```

-図10-

位置2の形態素「め」には、入力データストリームにあるデータの内で「痛」と「傷」が連接する。この場合、出力データストリームを図10の様にまとめ上げる。この処理は4.3.の2)に示したプロセスの④で行われる。

## 4.3.改良アルゴリズムのまとめ

以上、4.1,4.2の説明を踏まえ、改良アルゴリズムをまとめると以下の様になる。

### 1)プロセスの駆動制御

入力文字列(漢字かな文字の長さN+句読点等1文字)に対応して、N+1個のプロセス'dict'を並列に駆動する。'dict'にそのサブプロセスを駆動させる。

N個のプロセスは、入力漢字かな文字個々に対応する。また文頭の句読点1文字に対応して1個のプロセスが駆動される。'dict'には対応する文字の位置を示す番号を渡しておく。

隣接する'dict'をストリームで連結しておく。ストリームの中をデータは左から右へ流れる。

② F1行のプロセス'dict'には文頭であることを示すデータをストリームを用いて流す。

③ 最終文字に対応して駆動された'dict'の出力を解析結果とする。

### 2)駆動されたプロセス'dict'の動き

1の①で駆動されたプロセス'dict'は以下の様に処理を行なう。'dict'の処理を図11に示す。図中のプロセスに付けられた番号は、以下の説明で用いる番号に対応している。実線で示されたリンクは各処理の間をつなぐデータの流れである。これら全てのプロセスは並列に処理を進める事ができる。これは、並列論理型言語GHCで実現可能である。

① 入力ストリームから検索データDinを受け取る。検索データが無い場合にはデータが来るまで待つ。

データを受け取れば、プロセス③へ渡す。

ストリームの終端を受け取ればプロセス④へ渡して処理を終了する。

② Dinの検索開始位置番号が割り当てられた文字位置番号と同じならば、プロセス③④へDinを渡す。(以降DinをDInと表記する)

検索開始位置番号が文字位置番号より小さい時は、このデータを捨てる。

検索開始位置番号が文字位置番号より大きい時は、このデータをプロセス⑤へ出力する。

ストリームの終端を受け取ればプロセス③④⑤へ渡して処理を終了する。

③ 'dict'に、既にコンパイルされて記述されている個々の形態素情報DrfとDInの連接可能性を調べる。

連接可能な時は、DInにDrfを連接させた文字列Dfnを作りDfnの後方連接素性をDrfのものにしてプロセス⑤へ渡す。

連接不可能な場合には、このデータを捨てる。

ストリームの終端を受け取れば処理を終了する。

④ [サブプロセスpXYの動作]

④-1 入力ストリームから検索データDinを受け取る。検索データが無い場合にはデータが来るまで待つ。

データを受け取れば、プロセス④-2とサブプロセス内のサブプロセスへ渡す。(ここでサブプロセスの定義が再帰的になっている)

ストリームの終端を受け取ればプロセス④-2へ渡して処理を終了する。

④-2既にコンパイルされて記述されている個々の形態素情報DrfとDInの連接可能性を調べる。

連接可能な時は、DInにDrfを連接させた文字列Dfnを作りDfnの後方連接素性をDrfのものにして親プロセスへ返す。

連接不可能な場合には、このデータを捨てる。

ストリームの終端を受け取れば処理を終了する。

⑤ 最初に受け取ったデータがストリームの終端であれば、'dict'に割り当てられた文字を未登録検索

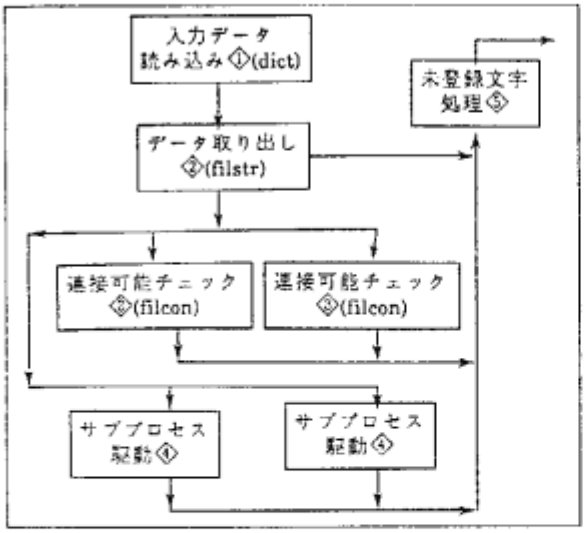


図11 'dict'の処理

データとし、入力データを全てに連接した形で出力ストリームへ出力する。これにより、4節の問題点2)で述べた改良点を実現している。

上記以外の何らかのデータを最初に受け取った場合には、受け取るデータを順次出力ストリームへ出力する。ストリームの終端を受け取れば処理を終了する。

3)計算オーダについて

逐次アルゴリズムの場合には計算量と計算時間が比例するが、並列アルゴリズムの場合にはそうならない。そこで、並列アルゴリズムの計算オーダを、全てのプロセスが計算を終了するまでの時間として、以降の考察を行う。

LAXのアルゴリズムでは計算の対象が非決定性有限状態オートマトンで受理される。非決定性有限状態オートマトンを決定的な有限状態オートマトンに書き換える事は行っていない。(逐次処理の場合、決定的な有限状態オートマトンの計算量はNを文字列の文字数とするとO(N)になる)。

まず、ある文字位置以降に文の終端まで均等に形態素があると仮定した際の計算量を考える。

この場合、ストリームを用いた基本アルゴリズムのオーダは、全ての形態素の可能な組み合わせ(文字のあらゆる可能な区切り方の量は、文字長さNの場合N-1個の文字の間で区切るかどうかの組み合わせで、$2^{n-1}$)を受理する唯一のプロセス(図3ではp67)の計算オーダとなり、これは、たとえ並列に他のプロセスが処理を行っても、指数関数的になる。しかし、ストリームの縮退を並列に行なう本アルゴリズムでは、前方からくるデータの一番多いプロセス'dict'(図4では第6行の'dict')のデータ読みこみ処理量(読みこみは線形に図11の①で行われる)となり、データ量は、前方に位置する'dict'の数*定数(定数は形態素の曖昧さの最高値)なので、並列処理の場合、O(N)で処理は行なえる。

以上が、ある文字位置以降に文の終端まで均等に形態素があると仮定した際の計算オーダである。

しかし、漢字かな混じり列を入力とする場合には、上記の仮定は厳しすぎる。漢字かな混じり列の場合、ある文字位置以降の形態素は、高々1個でかつ、形態的な曖昧さは、語基から形態素を解析する我々の方法を用いても高々3,4個である。前記の仮定による探索空間と現実の探索空間の相違を図12に示す。

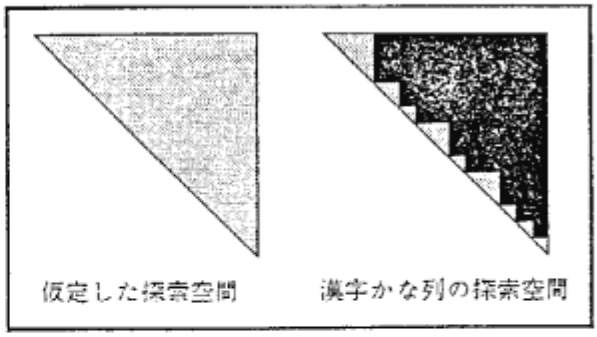


図12 探索空間例

図中薄い背景で示した部分が探索空間である。つまり、漢字かな混じり列を入力とする場合には、探索空間自体が殆ど文字列の長さに比例して広がる。このため、逐次処理の場合、計算量はO(N)になる。並列の場合でも、文字の先頭に対応した'dict'から後方の'dict'へ、ストリームによりデータが流れるための待ち合わせがあるため、やはり、逐次言語よりも辞書引きが並列に行われる分だけ処理は格段に速くなると予想されるが、基本的には線形時間O(N)で処理が進むと考えられる。未登録語は曖昧さのない長さ1の文字の連接として受理されるので、リニアオーダで処理できる。

## 5.逐次型言語でのアルゴリズム

LAXアルゴリズムの逐次型言語による実現について述べる。実現に用いた言語はESPで、PSIおよびPSI-II上で動作する。

レイヤードストリームは、GHCなどの並列型言語で、プロセスが独立して動作することを可能にするが、ESPやPrologのような逐次型言語では、レイヤードストリームで示される検索データを、シェアードメモリを用いて解析に用いる方が、メモリ効率と実行速度の点で有利である。

そこで、TRIE構造はそのまま用いて検索を行い、レイヤードストリーム上のデータが持つ検索開始位置番号毎にテーブル(ESPのスタックベクタ)を設け、各検索プロセスでこれを共有して解析を実現した。

このアルゴリズムは二つのステップからなる。

ステップ①は、入力文の各文字位置から始まる形態素の候補を辞書引きしてテーブルに並べる作業で、前出のアルゴリズム同様TRIE構造をコンパイルしたクローズに対するインデックスサーチにより検索を行う。

ステップ②では、ステップ1で検索した形態素間の連接チェックと未連接時の処理を行う。

アルゴリズム:

① 入力文字数+2のサイズのスタックベクタ(テーブル)を作る。テーブルの0および$n+1$番目にそれぞれ文頭と文末の連接素性データを代入する。

次に$i(1=<i=<n)$番目のテーブルエントリに対して以下の処理を行う。

i番目の文字から始まり、それ以降の文字が一致する形態素をすべてその連接情報を含めて検索し、テーブルにリストの形で代入す

る。各形態素は差分リストの形をとる。その語で始まる形態素が一つも無ければ未登録語としてテーブルに代入する。

② i(0=<i=<n)番目のテーブルエントリに対して以下の処理を行う。

i番目の形態素(長さをlとする)の後方連接素性と、i+l-1番目のエントリの各形態素の前方連接素性を調べ、一致すれば前方の形態素を後方の形態素のテイルにユニファイする。i番目のエントリで連接できるものが一つもなかった場合でかつ、i番目以前の形態素がi番目以降の形態素と繋がっていないときに、前方から繋がっている形態素を後方の形態素の一つに連接させる。前方から繋がっているものが一つもなければ何もしない。

以上のアルゴリズム実現するインタプリタを約100ステップのESPプログラムで記述した。形態素辞書については、1クラスに入る形態素数が限られているので、複数のクラスで構成するようにして扱える形態素数を増やしている。

## 6.逐次アルゴリズムの評価

逐次アルゴリズムの性能評価について報告する。付録1に解析に用いたテキストを、付録2にテキスト各文の文字数、解釈数、および、解析時間を示す。

評価マシンはPSI-IIを用いた。OSはSIMPOS 3.1-II版である。評価テキストは光村図書の小学6年テキスト「自然を守る」の冒頭30文である。

形態素解析時間は、30文全てを解析するのに1004msecを要した。これは、1文あたりに直すと33.5msecと十分高速である。解釈数が多い場合(文例29など)でも解析時間への影響は少ない。

現在の形態素の登録数は語基が500、助辞および接辞が300である[11]。今後、汎用性を高めるには、語基を更に登録する必要があるが、辞書のTRIE化により、解析速度への影響は殆ど無いと予想される。

以上のように、逐次言語上の本実現方式は、十分実用的な速度を持ち、満足の行くものである。

## 7.おわりに

日本語漢字かな混じり列の場合には、形態素探索空間がほぼ、O(N)で増える。このため、逐次版の実現方式は実用上、満足できる処理効率を示している。逐次版は、ICOTで開発中の日本語処理ツールLTB(Language Tool Box)の形態素解析エンジンとして、更に強力なものにしてゆきたい。

レイヤードストリームは、動的に駆動されるプロセス間でのメモリ共有問題を上手く解決している。しかし、オーダが並列処理においても、O(N)である(但し、逐次処理に比べ定数項は小さい)ことには不満を感じている。今後、PIMの立ち上がりを待って、現方式の評価を行うと共に、O(log(N))のアルゴリズムに付いても研究を行なって行きたい。

[謝辞]

LAXの研究開発においては、東京工業大学情報工学科、田中研究室で研究開発中のLanbLABから学ぶ点が多くありました。形態素文法研究開発には、上智大学国語学科の川田亮一氏に御協力を頂いております。KDD研究所の瀧塚孝志氏からは、計算量について貴重な御意見を頂きました。関連メーカの方々、ならびに向井主任研究員をはじめ、ICOT同僚諸氏からは、LAX研究開発において暖かいご理解とご支援を頂戴しております。紙面をお借りし、お礼申し上げます。

最後に、本研究の機会を下さいましたICOT渕所長、内田第2研究室長に感謝いたします。

[参考文献]

[1] Ueda Kazunori: Guarded Horn Clauses, ICOT Technical Report, No.103, 1985.

[2] 杉村領一,赤坂宏二,松本裕治: 並列形態素解析システムLAXの実現, 情報処理学会第35回全国大会, pp1323-1324,1987.

[3] 水谷静夫: 国文法素描, 文法と意味I, 朝倉日本語講座3, 朝倉書店, 1983.

[4] 奥村晃, 松本裕治: レイヤードストリームを用いた並列プログラミング, Proceedings of the Logic Programming Conference 87, pp223-232, 1987.

[5] 上脇三, 田中穂積: 辞書のTRIE構造化と熟語処理, Proceedings of the Logic Programming Conference 85', pp 329-340, 1985.

[6] Earley, J.: An Efficient Context-Free Parsing Algorithm, Comm.ACM, Vol13, No.2, pp.94-102., 1970

[7] Kay, M.: Algorithm Schemata and Data Structures in Syntactic Processing, Technical Report CSL-80-12, XEROX PARC, Oct.1980

[8] 松本裕治, 杉村領一: 論理型言語に基づく構文解析システムSAX, ソフトウェア科学会論文誌, 1986.

[9] 赤坂宏二, 杉村領一, 松本裕治: 逐次処理系上でのLAXの問題点とその解決法, 情報処理学会第35回全国大会, pp1355-1356, 1987.

[10] 吉村賢治, 日高達, 吉田将: 日本語文の形態素解析における最長一致法と文節数最小法について, 情報処理学会自然言語処理研究会資料30-7, 1982.

[11] 相沢輝昭, 江原暉将: 計算機によるカナ漢字変換, NHK技術研究, 25巻, 5号, 1973.

[12] 森岡健二: 語彙の形成, 現代語研究シリーズ1, 明治書院, 1987.

[13] 佐野洋, 赤坂宏二, 久保幸弘, 杉村領一: 語構成に基づく形態素解析, 情報処理学会第36回全国大会(発表予定), 1988.

[14] Mukai, K., A System of Logic Programming for Linguistic Analysis Based on Situation Semantics, Proceedings of the Workshop on Semantic Issues in Human and Computer Languages, Half Moon Bay, CSLI, 1987

付録1

解析用テキスト(/は文節の区切り、-は形態素の区切り、下線部分には曖昧さがある)

(1) 人間-が/、/この/地球-の/上-で/生-き/続け-て/い-く/ため-に-は/、/どうしても/、/自然-の/恵み-に/頼ら-なければならな-い/。

(2) 私-たち-が/毎日/食べ-る/物-も/、/住ん-で/い-る/家-も/、/着-る/服-も/、/もとはといえば/、/みな/自然界-から/手-に/入れ-た/物-で/あ-る/。

(3) 人間-は/、/そ-の/優れ-た/技術-を/使-って/、/自然-から/得-た/物-を/巧み-に/加工-し/、/自分-たち-の/生活-を/豊か-に/して/いる/。

(4) 人間-に/と-って/、/自然-は/、/限り無-い/資源-の/宝庫-な-の-だ/。

(5) また/、/人間-は/、/宅地-を/造-る/ため-に/、/山-を/切り/崩-して/平地-に/したり/、/交通-を/便利-に/する/ため-に/、/森-を/切-り/開い-て/道路-を/造-ったり-して/いる/。

(6) あるいは/、/電気-を/起こ-す/ため-に/、/川-の/流れ-を/せき止め-て/ダム-を/建設-したり/、/工業地帯-に/する/ため-に/、/海-を/埋め/立て-て/陸地-に/変え-たり/して/い-る/。

(7) つまり/、/人間-は/、/いろいろ-な/方法-で/自然-に/手-を/加え-て/い-る/の/で/あ-る/。

(8) このように/、/人間-は/、/自然-が/生-み/出-す/物-を/資源-と/して/利用-する/一方/、/自分-たち-に/都合-の/良-い-よう-に/自然-の/姿-を/変え-て/生活-して/い-る/。

(9) こ-の/地球/上-で/、/人間-だけ-が/、/自然-の/資源-を/思う存分/利用-したり/、/自然-を/改造-したり/する/知恵-と/力-と-を/備え-た/生物-な-の-で/あ-る/。

(10) しかし/、/そ-れ-だ-から-と/いって/、/人間-が/思いのまま-に/自然-の/姿-を/変-え/、/そ-の/資源-を/手当たり次第-に/自分-たち-の/物-に/して/しまって-も/いい-の-だろう-か/。

(11) こ-こ-で/、/考え-て/み-なければならな-い-の-は/、/自然界-に-は/、/さまざま-な/種類-の/生物-たち-が/、/それぞれ-の/環境-に/応じ-て/生き-て/い-る-と/い-う/こと-で/あ-る/。

(12) そして/、/こ-れ-ら-の/生物-たち-は/、/互い-に/影響-を/与-え/合-い/、/複雑-に/絡み/合-った/関係-を/保-ち-ながら/、/生活-して/い-る-と/い-う/こと-で/あ-る/。

(13) 森-と/い-う/環境-を/例-に/と-って/み-よう/。

(14) 森-に-は/、/いろいろ-な/動物-が/生-んで/いる/。

(15) 獣-も/鳥-も/虫-も/、/もっと/小さ-な/微生物-も/い-る/。

(16) 彼-ら-は/、/そ-こ-に/森-が/あ-る-から/生活-して/い-る-と/い-って/いい/。

(17) 森-の/植物-は/、/動物-たち-に/食物-を/提供-して/い-る/。

(18) 昆虫-たち-は/、/木-や/草-の/葉-を/食べ-たり/、/花-の/蜜-や/、/木-の/幹-から/出-る/樹液-を/吸-ったり/して/生活-して/いる/。

(19) 小鳥-たち-の/あ-る/もの-は/、/木-の/実-や/新芽-を/盛ん-に/ついば-む/。

(20) シカ-や/サル-は/、/木-の/葉-や/実-を/好ん-で/餌-に/する/。

(21) また/、/動物/同士-の/関係-を/見-て-も/、/昆虫-を/食べ-る/鳥-が/い-る-し/、/鳥-を/付-け/狙-う/イタチ-のよう-な/獣-も/い-る/。

(22) 虫-を/餌-に/し-て/い-る/カエル-は/、/ヘビ-に/一飲み-に-され-て/しま-う/。

(23) そ-の/ヘビ-も/、/タカ-のよう-な/大き-な/鳥-に-は/かな-わ-な-い/。

(24) 動物-たち-が/死-んだり/、/落-ち/葉-が/積もって/土-に/混じ-ったり/する-と/、/土-の/中-の/微生物-が/働き/始-め-る/。

(25) カビ-とか/バクテリア-のよう-な/微生物-は/、/動物-や/植物-の/死骸-を/腐-ら-せ/、/分解-して/しま-う/。

(26) 分解-の/結果-でき-た/物-は/、/やがて/、/植物-の/養分-と-して/根-から/吸-い/取-ら-れ-る/。

(27) そうして/、/植物-は/成長-し/、/動物-たち-に/、/再び-新し-い/食物-を/提供-する/こと-に/な-る-の-だ/。

(28) こうして/見-て/くる-と/、/森-の/生物-は/、/互い-に/食べ-たり/食べ-られ-たり/し-ながら/、/全体-と-して-は/、/うま-い/具合-に/釣合い-を/保-って/い-る/こと-が/分か-る/。

(29) 彼-ら-は/、/それぞれ/でたらめ-の/生-き/方-を/して/い-る-の-で-は/な-く/、/生物/同士-が/、/いわば/、/目-に/見え/な-い/鎖-のよう-な/もの-で/つなが-り/合-って/い-る-と-も/い-え-る/。

(30) だから/、/もし/何-か-の/原因-で/こ-の/釣合い-が/破れ/、/生物/同士-を/結-んで/い-た/鎖-が/断-ち/切-られ-る-と/、/思いがけな-い/出来事-が/起こ-る/こと-も/あ-る/。

付録2:逐次アルゴリズム解析データ

| 文番号 | 文字数 | 解釈数 | 解析時間 (msec) |
|---|---|---|---|
| 1 | 46 | 4 | 40 |
| 2 | 49 | 4 | 39 |
| 3 | 47 | 4 | 31 |
| 4 | 23 | 1 | 17 |
| 5 | 61 | 8 | 42 |
| 6 | 64 | 8 | 44 |
| 7 | 30 | 8 | 28 |
| 8 | 59 | 8 | 47 |
| 9 | 56 | 8 | 42 |
| 10 | 66 | 8 | 54 |
| 11 | 65 | 16 | 61 |
| 12 | 57 | 32 | 49 |
| 13 | 16 | 4 | 17 |
| 14 | 18 | 2 | 19 |
| 15 | 20 | 1 | 12 |
| 16 | 26 | 4 | 37 |
| 17 | 21 | 2 | 15 |
| 18 | 45 | 4 | 35 |
| 19 | 26 | 2 | 16 |
| 20 | 21 | 2 | 11 |
| 21 | 45 | 1 | 29 |
| 22 | 27 | 8 | 23 |
| 23 | 24 | 1 | 21 |
| 24 | 43 | 8 | 27 |
| 25 | 39 | 2 | 28 |
| 26 | 33 | 1 | 30 |
| 27 | 38 | 1 | 24 |
| 28 | 63 | 8 | 54 |
| 29 | 66 | 32 | 68 |
| 30 | 60 | 2 | 44 |

付録3：以下に4節でのべたアルゴリズムを実現するPrologのプログラム例を示す。文法は図2のものを用いた。主な述語名は図11のものに対応させた。

```
%dict 機動部
lex_P2([C|W],Result):-
    dict(C,W,0,[[0,end([文節,文頭]),nil]],Out),
    mawaru(W,1,Out,Result).
mawaru([],NO,IN,IN).
mawaru([W|R],NO,IN,Out):-
    dict(W,R,NO,IN,O1),NO1 is NO + 1,
    mawaru(R,NO1,O1,Out).

%'dict'群
dict(い,C_Words,ID_NO,Instr,OStr0):-
    IDNew is ID_NO + 1,
    filstr(Instr,ID_NO,OStr0,OStr1,StObj),
    filcon(StObj,
        [胃前方,IDNew,胃(体言語基),
            胃後方],OStr1,OStr2),
    (C_Words = [C|Words],
        い(C,Words,ID_NO,StObj,OStr2,OStr3)
        OStr2 = OStr3),
    sweep(OStr0,OStr3,[[IDNew,1,未登録語(い)|StObj]]).
dict(め,C_Words,ID_NO,Instr,OStr0):-
    IDNew is ID_NO + 1,
    filstr(Instr,ID_NO,OStr0,OStr1,StObj),
    filcon(StObj,
        [め前方,IDNew,め(動詞派生接辞),
            め後方],OStr1,OStr2),
    (C_Words = [C|Words],
        め(C,Words,ID_NO,StObj,OStr2,OStr3)
        OStr2 = OStr3),
    sweep(OStr0,OStr3,[[IDNew,1,未登録語(め)|StObj]]).
dict(る,C_Words,ID_NO,Instr,OStr0):-
    IDNew is ID_NO + 1,
    filstr(Instr,ID_NO,OStr0,OStr1,StObj),
    filcon(StObj,
        [る前方,IDNew,る(ウ系弱変化活用助辞),
            る後方],OStr1,OStr2),
    (C_Words = [C|Words],
        る(C,Words,ID_NO,StObj,OStr2,OStr3)
        OStr2 = OStr3),
    sweep(OStr0,OStr3,[[IDNew,1,未登録語(る)|StObj]]).
dict(な,C_Words,ID_NO,Instr,OStr0):-
    IDNew is ID_NO + 1,
    filstr(Instr,ID_NO,OStr0,OStr1,StObj),
    filcon(StObj,
        [な前方,IDNew,な(終助詞),
            な後方],OStr1,OStr2),
    (C_Words = [C|Words],
        な(C,Words,ID_NO,StObj,OStr2,OStr3)
        OStr2 = OStr3),
    sweep(OStr0,OStr3,[[IDNew,1,未登録語(る)|StObj]]).

%未登録文字処理部
dict(C,RR,ID_NO,Instr,OStr0):-
    IDNew is ID_NO + 1,
    filstr(Instr,ID_NO,OStr0,OStr1,StObj),
    sweep(OStr0,OStr1,[[IDNew,1,未登録文字(C)|StObj]]).

%サブプロセス部
いた(C_Words,ID_NO,StObj,OStr0,OStr3):-
    IDNew is ID_NO + 2,
    filcon(StObj,
        [痛前方,IDNew,痛(形容語基),
            痛後方],OStr0,OStr1),
    filcon(StObj,
        [傷前方,IDNew,傷(用言語基),
            傷後方],OStr1,OStr2),
    (C_Words = = [],
        OStr2 = OStr3
        C_Words = [C|Words],
        いた(C,Words,ID_NO,StObj,OStr2,OStr3)).
い(_,_,_,_,X,X).
いた(め,C_Words,ID_NO,StObj,OStr0,OStr2):-
    IDNew is ID_NO + 3,
    filcon(StObj,
        [炒め前方,IDNew,炒め,炒め後方],OStr0,OStr1),
    filcon(StObj,
        [板目前方,IDNew,板目,板目後方],OStr1,OStr2).
いた(_,_,_,_,X,X).
め(_,_,_,_,X,X).
る(_,_,_,_,X,X).
な(_,_,_,_,X,X).

filstr([],_,X,X,[]):-!.
filstr([[ID|R]|RR],ID,Str0,Str1,[R|R1]):-!,
    filstr(RR,ID,Str0,Str1,R1).
filstr([[ID|R]|RR],IDN,[[ID|R]|Str0],Str1,R1):-
    ID > IDN,!,
    filstr(RR,IDN,Str0,Str1,R1).
filstr([[ID|R]|RR],IDN,Str0,Str1,R1):-
    ID < IDN,!,
        filstr(RR,IDN,Str0,Str1,R1).
filcon([],_,X,X):-!.
filcon([[KS|W]|Data],[CMS,Id,HC,US],
        [[Id,US,HC,W|RO]|Strtail],Strtail):-
    check(KS,CMS),!,
    filcon1(Data,CMS,RO).
filcon([_|Data],CMSHC,RO,Stt):-!,
    filcon(Data,CMSHC,RO,Stt).
filcon1([],_,[]):-!.
filcon1([[KS|W]|Data],CMS,[W|R]):-
    check(KS,CMS),!,
    filcon1(Data,CMS,R).
filcon1([_|Data],X,Y):-!,
    filcon1(Data,X,Y).

sweep(X,Y,Z):-    X = = Y,  X = Z.
sweep(X,Y,Z):-    nonvar(X),!.

/*接続可能性判定部(実際には、2.2で示したように書くべきだが、紙面の都合上簡略化して示した)*/
check(end([文節,文頭]),胃前方).
check(end([文節,文頭]),痛前方).
check(end([文節,文頭]),傷前方).
check(end([文節,文頭]),炒め前方).
check(end([文節,文頭]),板目前方).
check(痛後方,め前方).
check(傷後方,め前方).
check(め後方,る前方).
check(炒め後方,る前方).
check(る後方,な前方).
```