記 号 処 理　38－1
（1986. 9. 16）

# Nat(Nonalgorithmic translator)の概要

沼尾正行　　志村正道
東京工業大学工学部

ルールの合成と分解による学習の実験用に、プロダクションシステム Nat (Nonalgorithmic translator)を開発した。Natは、問題とルールから答を推論する変換モードと、問題と答から変換法を学習する学習モードを持つ。学習は、ルールの分解による一般化に基づくもので、問題から答を導く最小コストの推論パスを参照して一般的なルールを獲得する。最小コストの推論パスはグラフ探索法により見つけ出されるが、これを効率的に行なうため統合表現を採用し、探索木の状態を書き換えられた部分についてだけ保存するようにした。本論文では、主として Utilisp で記述されたシステムのインプリメンテーションについて述べ、数式の簡単化と上向き構文解析を例として速度等の性能を評価する。

# Nat：Nonalgorithmic translator

Masayuki NUMAO and Masamichi SHIMURA
Faculty of Engineering, Tokyo Institute of Technology
2-12-1, Oh-okayama, Meguro-ku, Tokyo, 152 Japan

The rule based system, Nat(Nonalgorithmic translator), was developed for experiments on learning by rule composition and decomposition. In transform mode, it deduces an answer from a given problem, and in learning mode, rules are learned from problem-answer pairs. In rule decomposition, generalization is guided by a minimum-cost path that derives the given answer. For efficient minimum-cost search, the system preserves only rewritten parts of the working memory by using the representation called Uniform(Unified form). In this paper, we present an implementation on Utilisp and evaluate its efficiency in algebraic simplification and bottom-up parsing.

## 1．まえがき

Ｎａｔ(Nonalgorithmic translator)は、各種言語の変換を行なうプロダクションシステムである。従来、プログラミング言語や自然言語などの構文解析や翻訳など言語の変換には、専用のアルゴリズムが開発され使用されてきた。このような専用アルゴリズムは高速で効率がよいが、柔軟性に欠ける。Ｎａｔは、アルゴリズムが確定しない状態でも、人間のように柔軟に変換を行なうシステムを目指しており、変換モードと学習モードの二つのモードを持ち、新しい問題に対処しながらしだいに効率のよい変換ができるようになっている。

i）変換モード

問題とルールを与えることにより、答を得るモードであり、プロダクションシステムになっている。確定したアルゴリズムを用いれば効率のよい変換を行なえることはもちろんであるが、そうでないときにも試行錯誤的に変換を行なうことができる。

ii）学習モード

問題と答え、すなわち、変換すべきパターンと変換結果を与えることにより、変換法を学習するモードである。このモードでは、変換すべきパターンを左辺に持ち、変換結果を右辺に持つルールがまず生成される。このルールを分解して、一般化することにより、変換法が獲得される。

本稿では、主として Utilisp [Chikayama 81] で記述されたシステムのインプリメンテーションについて述べ、速度等の性能を評価する。

## 2．変換モード

プロダクションルールの形式を図１に示す。ルールは基本的には左辺と右辺が等しいことを表わしている。左辺と右辺は Lisp のＳ式で表わされたパターンであり、

```
<ルール> :=
  (rule  <rule名>
         <左辺> <オペレータ> <右辺>
       { restrict: <マッチ条件リスト> }
       { undefined: <ディフォールト値のリスト> }
       { cost: <ルールの実行コスト> } )

<左辺> := <パターン>

<オペレータ> :=  {!}= | {!}=> | <={!}

<右辺> := <パターン>

<マッチ条件リスト> :=  ( <マッチ条件> ･･･ )
<マッチ条件> := (<パターン> ･･･ <Lisp関数>)

<ディフォールト値のリスト>
                  := ( <ディフォールト値> ･･･ )
<ディフォールト値> := ( <メタ変数> <Lisp関数> )
```

図１　プロダクションルールの形式

| | |
|---|---|
| = | 前向きおよび後向き推論 |
| => | 前向き推論 |
| <= | 後向き推論 |

図２　オペレータ

変数として '*' のついた名前を用いる。図２のようにオペレータにより推論の方向が指定されるが、変換モードでは前向き推論だけが行なわれる。また、オペレータに '!' を含まないときにはルールの適用後も非決定的な推論のために適用前のパターンを保存し、 '!' を含むときには保存を行わない。

ルールには、オプションとして〈マッチ条件リスト〉、〈ディフォールト値のリスト〉、〈ルールの実行コスト〉をつけることができる。〈マッチ条件リスト〉は、変数の拘束条件を与えるもので、変数の満たすべき条件をLisp関数で与える。未定義変数、すなわち、前向き推論において右辺にだけ現れる変数の値を与えるのが〈ディフォールト値のリスト〉である。Ｎａｔは推論コストが最小になる解を探索する。〈ルールの実行コスト〉は、そのルールによる一回の推論コストを指定するもので、省略値は１である。

このようなルール形式を用いると数式の簡単化を行なうルール[Bundy 83]は、次のように表せる。

```
(rule simp1  (*x * 0) = 0 )
(rule simp2  (1 * *x) = *x )
(rule simp3  (*x exp 0) = 1 )
(rule simp4  (*x + 0) = *x )
```

図３に示された自然言語の文法規則[Tomita 85]の左辺と右辺を入れ換えてルール化したのが図４に示すルールである。この規則により直ちに上向き構文解析(bottom up parsing)を行なうことができる。

## ３．学習モード

ａからｂを推論するルール ａ => ｂ は、ａ => ｃ および ｃ => ｂ の合成されたルールとみることができる。このように、ルールを複数のルールに分ける操作をルールの分解と呼ぶ。ａ => ｂ によって行なえる推論は ａ => ｃ と ｃ => ｂ によっても行なうことができるが、一般にその逆は成立しないので、ルールの分解によってルール集合を一般化することができる。ルールの分解を用いて、ルールを獲得する手順は図５のようになる。

ルールの分解を行なうためには、まず、ルールを与えられた基本ルールに分解し、構成グラフを生成する。構成グラフは、そのルールを基本ルールから合成する方法を示しており、〈ルール名〉をノードとする有向グラフとなっている。アークは、結んでいるルールを合成することを示す。

構成グラフは、基本ルールを用いたプロダクションシステムによる探索の結果得られる。すなわち、分解すべきルールの左辺から右辺を導く最小コストの経路を見出し、それに含まれる推論の依存関係を解析することにより得られる。基本ルールによっては次に示すように、前向き推論を行なうと探索範囲が非常に増大してしまうルールがある。

文法

```
S → NP VP
S → S PP
NP → #n
NP → #det #n
NP → NP PP
PP → #prep NP
VP → #v NP
```

辞書

```
#v : saw.
#det : a, the.
#n : I, man, park,
     telescope.
#prep : in, with.
```

図3　文法規則

文法

```
(rule S1  (NP VP . *t) => (S . *t) )
(rule S2  (S PP . *t) => (S, *t) )
(rule NP1 (#n . *t) => (NP . *t) )
(rule NP2 (#det #n . *t) => (NP . *t) )
(rule NP3 (NP PP . *t) => (NP . *t) )
(rule PP1 (#prep NP . *t) => (PP . *t) )
(rule VP1 (#v NP . *t) => (VP . *t) )
```

辞書

```
(rule SAW  saw => #v )
(rule A    a => #det )
(rule THE  the => #det )
(rule I    I => #n )
(rule MAN  man => #n )
(rule PARK park => #n )
(rule TELE telescope => #n )
(rule IN   in => #prep )
(rule WITH with => #prep )
```

図4　構文解析のルール

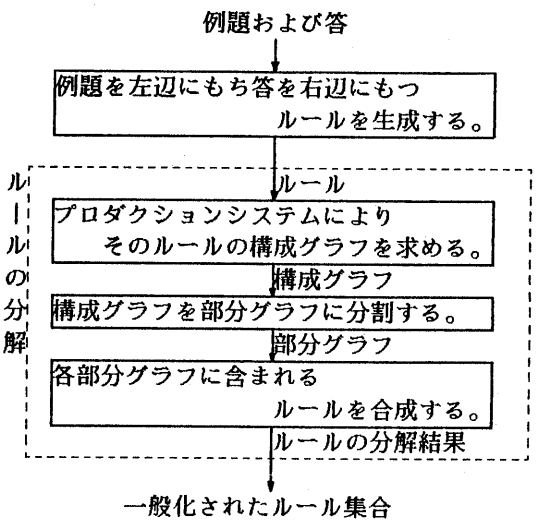


図5　ルールを獲得する手順

(1) ルールから構成グラフを生成し、それぞれを $G_{pi}$ (i=1, ..., n)とする。
(2) 最小コストの経路上にないグラフを一つ選び， $G_N$ とする．$G_N$ がなければ終了する。
(3) $G_N$ 中で最初に適用されたルールを $H_N$ とする．
(4) $H_N$ が $G_{pi}$ に含まれていないならば， $H_N$ を消去する．
(5) $H_N$ が $G_{pi}$ に含まれているならば， $G_{pi}$ 中のそれぞれの $H_N$ の出現について，
(a) $H_N \in G_s \not\subseteq G_N$ ，かつ
(b) $G_s' \subset G_s$ なる別の$G_s'$が存在しない，
ような部分グラフ $G_s$ を一つずつ選び， $G_s$ 中のルールを合成して新しいルールを生成し、 $G_s$ を一つのノードで置き換える．また，$H_N$ を消去する．
(6) (2)へ行く．

図6　ルール分解のアルゴリズム

```
(rule Double-negation
      *x = (not (not *x)) )
```

したがって、このような基本ルールは後向きにしか使えない。このため、各ルールの推論方向を図２に示したオペレータにより指定し、分解するルールの左辺と右辺から双方向に探索を行なう。

こうして、得られた構成グラフを各々$G_{pi}$とする。また、構成グラフを前向きにたどりなおすことにより、左辺から右辺を導く経路上にない推論のグラフを求めて $G_N$ とする。これらのグラフに図６のアルゴリズムを適用することにより、決定的に例題を解くようにルールを分解することができる。

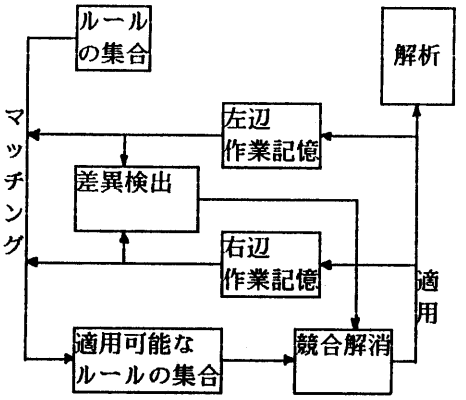


図７　Ｎａｔの構成

## ４．システムの実現

システムは、 Utilisp を用いて 5000行程度で記述されている。その構成を図７に示す。プロダクションシステムの作業記憶としては、左辺作業記憶と右辺作業記憶がある。変換モードでは左辺作業記憶だけを使用し、前向き推論だけを行なう。学習モードでは、左辺作業記憶に分解するルールの左辺、右辺作業記憶に分解するルールの右辺を格納して、双方向探索法により最小コストの経路を探索する。すなわち、左辺作業記憶では前向きの推論が行なわれ、右辺作業記憶では、後向きの推論が行なわれる。差異検出部は、推論の１ステップごとに左辺作業記憶の内容と右辺作業記憶の内容を比較し、一致していれば探索を終了し、一致していなければその差異を検出する。また、各ステップごとに適用可能なルールの集合が更新され、その中から競合解消部により、差異を解消するのに効果的なルールが選ばれ、適用される。

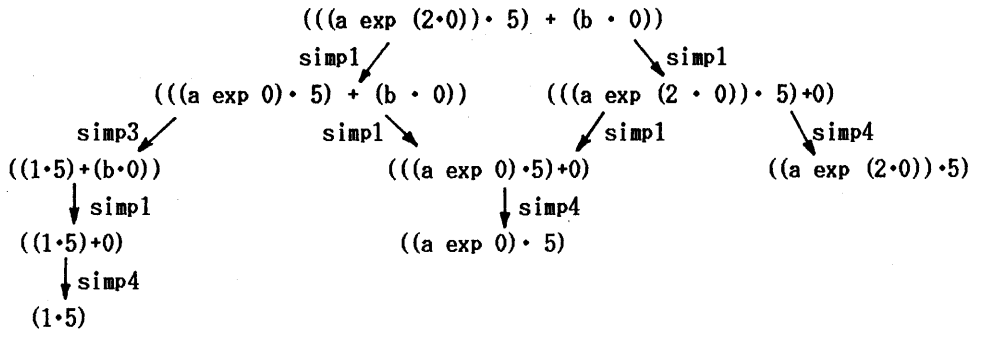


図８　数式の簡単化の探索木

本システムでは、変換モードにおける推論と学習モードにおける構成グラフの生成において、非決定的な推論による解の探索が必要である。次に、これを効率よく実現する方法について述べる。バックトラック法では、最小コストの経路を探索できないので、グラフ探索を用いる。探索木の各状態を保存するには、大量のメモリを必要とし、パターンの各部分に独立にルールが適用されうるので探索空間も組合せ的に増大する。たとえば、'( ((a exp (2 ・ 0)) ・ 5) + (b ・ 0) )' に、前述した数式の簡単化を行なうルールを適用した場合の探索木の一部は図８のようになる。これらの問題は、探索木の状態を書き換えられた部分についてだけ保存することにより解決される。すなわち、〈パターン１〉を〈パターン２〉に書き換えた場合、その部分を '{== 〈パターン１〉〈パターン２〉}' で表現する。すなわち、図８の探索木全体が次のように一つに統合されて表現される。

```
{== ( ({== (a exp {== (2 ・ 0) 0}) 1} ・ 5) + {== (b ・ 0) 0} )
      ({== (a exp {== (2 ・ 0) 0}) 1} ・ 5) }
```

下線の部分はポインタで共有されており、このパターンはルールを４回適用することにより生成することができ、その結果、探索木の９つのノードを探索したことになる。すなわち、記憶領域を節約すると同時に探索範囲も削減することができる。このようなパターン表現法を統合表現と呼ぶ。なお、ルールのオペレータに '!' を付け加えることにより書き換えられた部分のパターンを捨てて、決定的な推論を行なわせることもできる。

統合表現に対しパターンマッチングを行なうのには、弁別ネット(discrimination net)[Charniak 80]を使用する。これにより、ある表現がどのルールのパターンとマッチするのかを効率よく決定することができる。弁別ネットを使用するためには、表現をまずシンボルの列に変換する。例えば、(a exp (2 ・ 0)) は次のように変換される。

```
cons a cons exp cons cons 2 cons ・ cons 0 nil nil
```

弁別ネットのルートからシンボルの列にしたがってネットをたどれば、ルールに到達することができる。表現に " == " オペレータを含むばあいには、シンボルの列を分岐させればよい。すなわち、(a exp {== (2 ・ 0) 0}) の場合には、ネットを次のようにたどればよい。

```
cons a cons exp cons ─→ cons 2 cons ・ cons 0 nil nil
                    │
                    └→ 0 nil
```

" == " オペレータがネストしているばあいには、分岐を同様に繰り返すことになる。このように、弁別ネットを使用すれば統合表現にマッチするパターンを、一度に

すべて求めることができる。

統合表現は、図５で示した自然言語の構文解析においても有効である。すなわち、統合表現の働きにより複数個の部分解析木を効率的に保持することができる。図９は、次の文に図５のルールを適用した場合の推論の依存関係を図示したグラフである。

( I saw a man in the park with a telescope . )

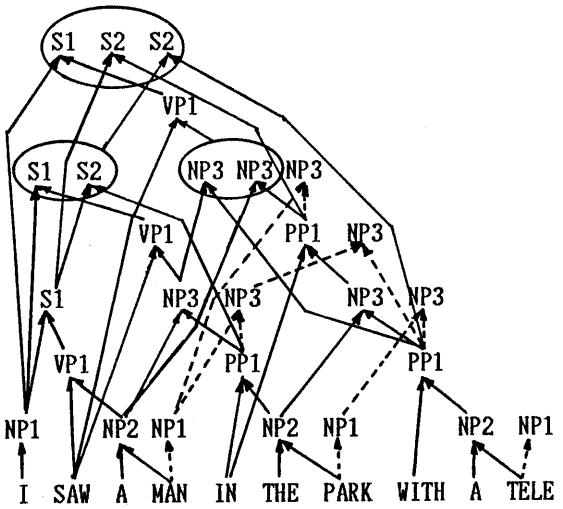


図９　構文解析における適用ルールの依存関係

実線は推論の依存関係を示し、点線は解析結果を導かなかった推論を示す。また、実線で囲まれたノードは、同じ書き換え結果を得た推論である。このように、統合表現によって横形上昇法による構文解析を直ちに行なうことができる。

学習モードにおける構成グラフは、推論の依存関係を解析することにより生成される。このため、作業記憶中のセル、アトムおよびリンクに、それを生成した推論を識別する標識が付けられている。セルおよびアトムの標識により、それらを生成したルールを追跡し、リンクの標識により、リンクの付けかえを行なったルールを追跡する。たとえば、(x y z) に次のようなルールが適用された場合を考える。

```
(rule aaa (x . *y) = (x y) )
```

この場合、(x y z) には、図10のように c1,c2,c3,a1,a2,a3,p1,...,p7の標識がつけられており、ルールの左辺は図10の点線で囲まれた部分のリンクおよびアトムとマッチしている。したがって、c1,p2,a1 の示す推論にこの推論が依存していることがわかる。構成グラフは、左辺作業記憶と右辺作業記憶について、推論の依存関係を追跡することによって得られる。

推論コストは、推論に用いられたルールの 〈ルール実行コスト〉 の和である。これは、推論の依存関係を追跡しながら、〈ルール実行コスト〉 を加算してゆくことにより得られる。

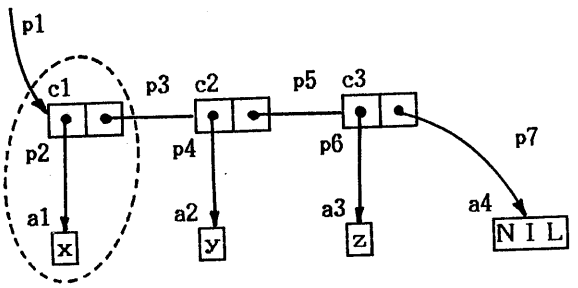


図10　標識

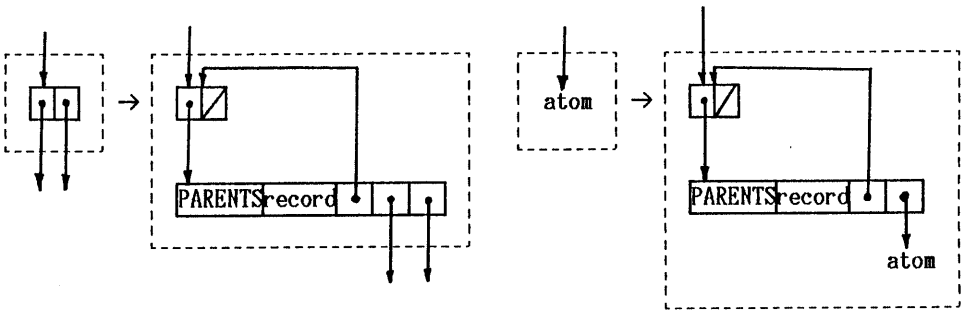


図11　内部構造

統合表現を実現するための内部構造を図11に示す。consセルは長さ５のベクタ(vector)、atomは長さ４のベクタで表現される。'{== ... }'を表現するため、これらのベクタはリストの要素になっており、推論により新たに生成されたパターンはこのリストの要素として追加される。

ベクタの第１要素は、親を指すポインタのリストである。推論の各ステップごとに、書き換え部分を含むパターンのマッチングをやりなおすために用いられる。

ベクタの第２要素は、推論の依存関係を追跡するのに用いられる。セルまたはアトムの標識とポインタの標識が連想リストの形で格納されている。

consセルを表わすベクタでは、第３、４要素がそれぞれ car、 cdr へのポインタになっている。atomを表わすベクタでは、第３要素にそのatomが格納されている。

適用可能なルールと適用部分および変数値をまとめて agenda item と呼ぶ。agenda item は、agenda-list にまとめて保持されており、競合解消ルールにしたがって優先される順序に整列されている。競合解消ルールは Lisp で記述され、次の形で呼び出される。

(priority-function <agenda-item1> <agenda-item2>)

<agenda-item1> が <agenda-item2> より優先順位が高いと t そうでなければ nil が返るようにする。一度適用された agenda item は、ルールごとにルール名のプロパティリストに格納され、同じ推論の繰返しを避けるのに用いられる。

システムは、次の手順により動作する。

ｉ）作業領域にデータを格納するとともに、そのすべての部分表現についてパターンマッチングを行なって、agenda-list を初期化する。
ii）agenda-list の最初の agenda-item を取り出し、適用して結果を得る。
iii）適用結果が作業記憶に存在しているときは ii）へ。
iv）適用結果が作業記憶に存在してない時には、結果を図11におけるリストの要素として付け加え、適用された agenda-item をルール名のプロパティリストに格納する。

| | 最初の解析結果 | | | すべての解析結果 | | |
|---|---|---|---|---|---|---|
| | ステップ数 | CPU時間(ms) | 一秒当たりのステップ数 | ステップ数 | CPU時間(ms) | 一秒当たりのステップ数 |
| M680H | | 139 | 208 | | 183 | 175 |
| FACOMα | 29 | 1465 | 19.8 | 32 | 1975 | 16.2 |
| Ustation/E15 | | 4020 | 7.2 | | 5660 | 5.7 |

表1　Ｎａｔの速度

v）適用結果について、パターンマッチングを行なう。また、図11の PARENTS をたどって適用結果を含むすべての部分表現についてもパターンマッチングをやり直し、これらの結果に基づいて agenda-list を修正する。
vi）ii）～iv）を繰り返し実行する。

前述の構文解析の所要時間を M680H、FACOMαおよびUstation/E15の各 Utilisp 上で測定したものを表１に示す。コスト最小の経路を見つけるためには、競合解消時に推論コストの計算などが必要になる。ここでは、これらのオーバヘッドのない状態で、最初の解析結果が得られるまでの時間とすべての解析結果の得られるまでの時間をそれぞれ示した。

## ５．あとがき

ルールの合成と分解による学習の実験用に開発したプロダクションシステムＮａｔについて、機能と実現法を述べた。また、変換モードの動作例として、数式の簡単化と上向き構文解析の例をあげた。現在、本システムを応用して次のような実験を行なっている。

(1) プログラム翻訳の学習
関数またはマクロの定義と翻訳例を与えることにより、Lispの方言間の翻訳ルールを学習する。
(2) 自然言語翻訳の学習
単語の対訳と翻訳例を与えることにより、翻訳ルールを学習する。
(3) 論理回路の生成の学習
ゲートの仕様および論理表現の簡約化ルールと設計例から、設計ルールを学習する。
(4) 類推
学習モードにおける双方向探索により、二つの知識表現の意味マッチングをとり類比をルールとして学習する。このルールにより類推を行なう。
(5) Lisp tutor
学習モードにおける双方向探索を利用して、Lisp プログラムとして与えられた生

徒の解答と模範解答との意味マッチングをとる。これにより、誤りの指摘等を行なう。

これらについては、別途報告してゆきたい。

参考文献

[Bundy 83] Bundy,A.: The Computer Modelling of Mathematical Reasoning, The Alden Press, Oxford(1983).

[Charniak 80] Charniak, E., Riesbeck, C.K. and McDermott, D.V.: Artificial Intelligence Programming, Chapter11 and 14, Lawrence Erlbaum Associates, Inc, Hillsdale(1980).

[Chikayama 81] 近山隆: UTILISP MANUAL, 東京大学計数工学科 Technical Reports, METR81-6(1981).

[Tomita 85] Tomita,M: An Efficient Context-free Parsing Algorithm For Natural Languages, IJCAI85, pp.756-764(1985).