知識工学と人工知能 40 － 3
（１９８５．５．７）

# 学習機能を持つプログラム翻訳システム

沼尾正行　　志村正道
（東京工業大学工学部）

## １．まえがき

現在、各種のアプリケーションに応じるため、数多くのプログラム言語が存在している。これらは本来、人間の労力を軽減するために作成されたものだが、いったん一つのプログラム言語上でプログラムを作成すると、これを他のプログラミング言語に移すことは、厄介で人手を要する作業となっている。ここでは、このようなプログラム翻訳を自動化する方法を考察する。

プログラム翻訳の自動化は、一見コンパイラの作業と似た面を持つが、次のような違いがある。

1) プログラム翻訳システムによって出力されたプログラムは、人間にとって分りやすいものでなければならない。
2) コンパイラの場合は、高レベル言語から低レベル言語への翻訳となるが、プログラム翻訳システムの場合には、必ずしもこのような翻訳ではなく、低レベル言語から高レベル言語の場合もあるし、普通は同程度のレベルの言語で機能が異なる言語間の翻訳となる。
3) コンパイラの場合は、プログラムのディバッグ時にコンパイルを繰り返すので、コンパイル速度が高速であることが要求されるが、プログラム翻訳は完成されたプログラムを翻訳するので、翻訳の速さはそれほど要求されない。

プログラム翻訳システムに多くの知識が必要とされることが、(1)と(2)から予想される。すなわち、プログラム言語の性質についての知識や、そのプログラム言語を用いて記述したプログラムのスタイルについての知識などが、要求される。(3)の特徴は、プログラム翻訳システムが人手による翻訳を代行するシステムであることから生じるものであり、時間をかけてもよいから(1)や(2)を満たすべきなのを示している。

これらの特徴を持つシステムを得るため、学習によって翻訳システムを構築する方法について検討している。本論文では、学習すべき内容、すなわち、翻訳のしかたを発見するための方法について述べる。翻訳のしかたを見つけるというのは、翻訳の手順を探索することに他ならない。この観点から、主として目的プログラムの探索法について述べることにする。

## ２．プログラム翻訳システムの概要

### 2.1 翻訳の対象言語と翻訳の方法

翻訳の対象となるプログラム言語としては、Lispの二つの方言を考えることにした。これは、(1) Lispではプログラムがリスト構造で表わされているため、取扱いが簡便なことと、(2) プログラム翻訳システムをLispで記述するため、対象言語をLispとしておいた方が実験の便宜上都合がよいことによっている。他のプログラミング言語、例えば Pascal, Fortran, Prolog などを対象とするのであれば、簡単なテキストの入力リーダを作成して、プログラムをS式の形に変換してから取り扱うことになる。

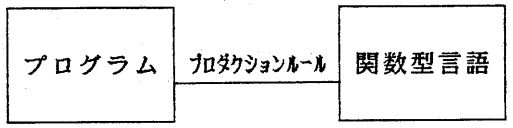



図1

プログラムの翻訳においては、プログラムの意味を変えないことが重要である。これを保証するため、原言語および目的言語の意味を独立にプロダクションルールの形で与えることにした。これらのプロダクションルールは各プログラム言語の表示的意味[Stoy 77][Gordon 79][中島82]に基づいたものとなっており、図1に示されるように各プログラム言語と関数型言語との間を相互に変換する。

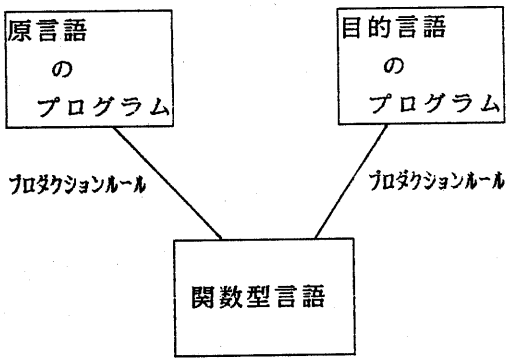



図2

翻訳は図2に示されるように、関数型言語を中間言語としてプロダクションルールによって行なわれる。複数の言語の翻訳ルールが関数型言語を中心として独立に与えられるので、このプログラム翻訳法は、自然言語の機械翻訳の用語を借りれば、「ピボット」方式の翻訳法となる。このようにして関数型言語を中間言語とすると、次のような利点および欠点が生じる。

i) 原言語から中間言語への変換ルール、および中間言語から目的言語への変換ルールの正しさが各言語の表示的意味によって保証される。したがって、翻訳全体の正しさも保証されることになる。

ii) 一般のプログラミング言語のような手続き型言語に比べて、関数型言語は変形が容易である[Burstall 77]。したがって、中間言語に変換した段階で各種の変形を施して、よりよいプログラムを生成することができる。

iii) 原言語のプログラム全体をそのまま関数型言語に翻訳すると、膨大なものになってしまう。

iv) 原言語や目的言語と中間言語の対応は一意に定まらない。すなわち、あるプログラムに対し適用できるプロダクションルールが多数存在するので、目的言語の

〈ルール〉 ::= ( 〈左辺〉 = 〈右辺〉
　　{ RESTRICTS: 〈マッチ条件リスト〉 }
　　{ "〈コメント〉" }　　)

〈左辺〉 ::= 〈パターン〉
〈右辺〉 ::= 〈パターン〉
( 〈パターン〉 において、＊ のついた変数はメタ変数として扱われる。)

〈マッチ条件リスト〉 ::=
　　( (〈メタ変数〉 〈Lisp関数〉) (〈メタ変数〉 〈Lisp関数〉) ・・・ ) .

図３　プロダクションルールの形式

プログラムを得るにはかなりの探索が必要となる。

(iii)の欠点は、原言語、中間言語および目的言語を一つの表現中に混在させて、翻訳中の部分だけを中間言語で表わすことにより解決できる。(iv)の欠点は、逆にこの方法が人工知能の各種技法を適用できるものであることを示している。例えば、プロダクションルール選択のためにヒューリステックを使用することや、学習機能により、より強力なプロダクションルールを合成し、探索を減らしてゆくことなどが考えられる。すなわち、初期状態においては翻訳の正しさを保証する基本ルールだけが存在しており、基本ルールをもとに小さな訓練例で学習を繰り返すことにより、システムが構築される。

## 2.2 プロダクションシステムによる翻訳

コンパイラのような従来のプログラム翻訳システムでは、翻訳されるプログラムを先頭から読み込みながら翻訳結果を出力するようになっている。このため、プログラムを何回読み込むかにより、１パス、２パス、多パスなどの翻訳システムが作られた。このような処理法によれば、小さなメモリで高速の処理を行なうことができる反面、一時にプログラムの一点だけを参照しながら処理しているので、全体構造を考慮にいれた処理が難しくなる。また、処理が必ずプログラムの先頭から後尾に向かって行なわれるので、この順序にそぐわない処理が困難である。本システムでは、これらの問題を避けてより高度な翻訳を行なうため、ワーキングメモリ(Working Memory, 以下 ＷＭ と呼ぶ)にプログラム全体を格納し、プロダクションシステム(ＰＳ)により翻訳を行なう。

各プロダクションルールの形式を、図３に示す。ルールは左辺(LHS)と右辺(RHS)が等しいことを表わしている。すなわち、左辺のパターンとマッチする表現を右辺のパターンに書き換えること、および右辺のパターンとマッチする表現を左辺のパターン

```
<中間言語> ::= <式>

<式> ::= nil | t | <数字> | '<S式> | <名前>
        | (cond   ( <式>  <式> )··· )
        | ( <関数>   <式>···  )

<関数> ::= (lambda ( <名前>··· ) <式> )  | <式>

<S式> ::= <原言語> | <目的言語>
```

図４

に書き換えることが可能なことを表わす。パターン中の * のついた変数、例えば *x , *y 等で、ルール用のメタ変数を表わす。メタ変数がどのような表現とマッチするべきかを示すのが<マッチ条件リスト>で、メタ変数とLisp関数の組により各メタ変数の満たすべき条件を与える。

プログラムは、WMに格納されて翻訳される。翻訳の中途ではプログラムの各部分は異なった翻訳段階にあるので、各翻訳段階、すなわち原言語、中間言語および目的言語のプログラムを一つの表現中に混在できる必要がある。このため、WM中では図４に示されるように、中間言語に原言語および目的言語がデータとして埋め込まれた形でプログラムが表現される。例えば、翻訳の開始時にはWMの内容は次のようになっている。

```
(s:p: '<原言語のプログラム> :input-data:)
```

s:p:は、原言語の意味関数である。いろいろな翻訳段階にあるプログラムの混在した表現を少しづつ変形することにより、翻訳が進められる。

このような表現法は、システムに学習機能をもたせる場合に適した表現法でもある。原言語、中間言語および目的言語が一つの表現で表わされるので、原言語から中間言語への翻訳ルールと中間言語から目的言語への翻訳ルールを混在させて、一元的に管理できる。したがって、学習によってこれら既存のルールを合成して、原言語から直接目的言語を生成するルールを作り出すこともできる。

## ３．目的言語プログラムの探索

### 3.1 バックトラック法とグラフ探索法

プログラムに対し適用できるプロダクションルールは多数存在するので、目的言語のプログラムを得るには探索が必要となる。プロダクションシステムにおいて探索を行なう方法としては、バックトラック法とグラフ探索法がある[Nilsson 80]。バックトラック法では、あるルールを適用しそれが解を導かなければ、そのステップは忘れてその代りに別なルールを適用する。これに対しグラフ探索法では、適用されたルールとそのときのWMの内容を探索グラフに記録する。探索グラフの各ノードはWMの内容を示しており、アークはルールの適用を表わしている。終了条件を満足するノー

ドが作られるまで、このようなグラフが成長してゆく。

これらの探索法で問題なのは、適当なヒューリスティックが得られない場合に探索の範囲が極めて大きくなってしまうことである。バックトラック法およびグラフ探索法どちらの場合でも、探索グラフの各ノードを調べねばならず、探索の範囲がプロダクションルールの数に対し組み合わせ的に増大してしまう。ここではこの問題を解決するため、バックトラックを行なったり探索グラフを作ったりせずに、探索を行なう方法を述べる。

## 3.2 統合表現

探索の範囲が組み合わせ的に増大してしまうのを避けるために、ＷＭのデータ表現を工夫することにする。探索の範囲が大きくなるのは、表現のごく一部が異なっているだけで、表現全体を別の表現とみなしているからである。これらの別々の表現を統合して、一つの表現で表わすことができれば、探索の範囲を小さくすることができる。これは、プログラム中に新しいオペレータ " == " を挿入して、表現の異なっている部分 〈表現１〉〈表現２〉 ･･･ を次のように並べることで実現される。

(== 〈表現１〉 〈表現２〉 ･･･ )

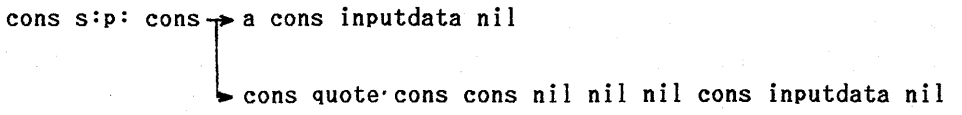
このようにしてできた表現を、「統合表現」と呼ぶことにする。

例えば、(A B C) という表現の部分表現Ｂにルールを適用して新しい部分表現Ｄを得たとする。従来の方法では、ＷＭを (A D C) と書き換えると共に、前のＷＭの内容 (A B C) をバックトラックのために保存する必要があった。これに対し、統合表現では、これらを一つの表現 (A (== B D) C) で表わすことができる。しかも、この表現は二つの表現 (A B C) と (A D C) を同時に表わしているので、仮にこのルールの適用が不適当であった場合にも、バックトラックの手続きは不要である。単に変形を続行すればよい。

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、(s:p: '(nil) inputdata) は次のように変換される。

```
cons s:p: cons cons quote cons cons nil nil nil cons inputdata 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、(s:p: (== a '(nil)) inputdata) の場合には、ネットを次のようにたどればよい。




" == " 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。

### 3.3 インプリメンテーション

以上述べたような統合表現を使用して、SORD M685 の Utilisp 上にシステムを実現した。プログラムは、ルールを合わせてほぼ 1200行である。本システムでは、原言語および目的言語の意味をプロダクションルールとして与えるが、[岡 83]で記述されたLispの表示的意味をもとにして、このプロダクションルールを作成した。現在は、小さなプログラムを翻訳しながら、これらの基本プロダクションルールの検討を行なっている。

プロダクションシステムは高速になるように工夫している。プロダクションシステム用の高速アルゴリズムとしては、Reteアルゴリズム[Forgy 82]がよく知られている。このアルゴリズムの基本的考え方として、次の二点が挙げられる。

i) 全プロダクションルールのパターンをネットワークにコンパイルする。

ii) WM中のデータのうち、一回のルールの適用によって変化するのはごく一部である。そこで、WM中の変化のあった部分だけパターンマッチングをやり直す。

このうち(i)については、前述したようにパターンを弁別ネットにコンパイルしている。(ii)については、本システムではWMのデータ表現が異なるため、Reteアルゴリズムと同じ方法を使用することができないが、同じ方針でインプリメントを工夫した。

試験用のごく簡単な例による実験結果を、表１に示す。使用したルールは、原言語の意味記述用が２８個、目的言語の意味記述用が２２個、その他のルールが１０個である。一番目の翻訳は、原言語の NIL というプログラムが目的言語に変換されると同じ NIL になることを確認するためのものである。二番目の例は、原言語のCATCH関数と目的言語のCATCH関数の仕様を変えた場合の翻訳を示している。CATCH関数は、THROW関数と組み合わせて大域脱出を行なわせるための関数である。原言語においては、 CATCH の第一引数がタグとなっており、第一引数が評価された結果がタグとして使用される。これに対して、目的言語では CATCH の第二引数がタグとなっており、評価されずにそのまま使用される。このような仕様で原言語と目的言語の意味を独立に与えておけば、表に示したような翻訳が行なわれる。 三番目は、原言語中の

表１　実験結果

| 翻訳内容 | ヒューリスティック有 | | | ヒューリスティック無 | | |
|---|---|---|---|---|---|---|
| | ステップ数 | 探索範囲 | 処理時間 | ステップ数 | 探索範囲 | 処理時間 |
| NIL -> NIL | 10 | 11 | 28.3秒 | 15 | 194 | 71.4秒 |
| (CATCH 'L EXPR) -> (CATCH EXPR L) | 15 | 16 | 81.8秒 | 21 | 1714 | 260.6秒 |
| (CATCH 'L (THROW 'L NIL)) -> NIL | 15 | 21 | 113.4秒 | 21 | 9538 | 316.9秒 |

処理時間：SORD M685(CPU 68000 12.5MHz，主記憶 4MB) の CPU時間
ステップ数：統合表現に対するルール適用の回数
探索範囲：統合表現を探索グラフに換算した場合のノード数

CATCH と THROW が消去される場合を示している。このように、翻訳の過程でプログラムの簡約化を行なうこともできる。実験結果の各数値のうち、ステップ数は、統合表現を使用した場合のＷＭに対するルールの適用回数を示している。このステップで探索された範囲をグラフ探索法におけるノード数に換算して示したのが、探索範囲である。処理時間は、SORD M685 上のＣで書かれた UtiLisp で測定した。また、プロダクションルールの競合解消用として、「なるべく大きな範囲のプログラムとマッチするルールを優先する。」という簡単なヒューリスティックを設け、これを用いた場合とそうでない場合を比較した。このヒューリスティックは簡単なものであるが、この程度のプログラムではかなり効果的である。

ヒューリスティックを用いた場合には、探索範囲はそれほど大きくならず、統合表現を採用したことによるステップ数の減少は４０％程度にとどまる。これに対し、ヒューリスティックを用いなかった場合には、探索範囲は、組み合わせ的に増大する。しかしながら、統合表現を用いたためにステップ数はそれほど増大しない。このことから、ヒューリスティックが当てにならないときに統合表現を用いれば、特に有効であることがわかる。

## ４．あとがき

学習によりプログラム翻訳システムを構築するための前段階として、目的プログラ

ムを探索によって見つけだす方法について述べた。プロダクションシステムにおいて探索を行なう方法としては、バックトラック法とグラフ探索法が一般的であるが、プログラムの変形に対してこれらの探索法を適用すると、探索の範囲が極めて大きくなってしまう。本論文ではこの問題を解決するため統合表現を提案し、その効果を簡単な例を用いた実験によって確かめた。今後は、基本ルールによる翻訳能力を調べるための実験を続けるとともに、発見した翻訳手順をもとに、翻訳ルールを合成する機能をインプリメントしてゆきたいと考えている。

## 謝辞

本研究で使用した M685 C言語上のLispを提供して下さった、東京大学の和田英一教授に感謝する。志村研究室の桜井成一朗氏はこの Lispを UtiLispコンパティブルな Lispに拡張してくれた。また、宮田俊介氏には弁別ネットのプログラムの初期バージョンの移植を行なってもらった。ここに記して礼を述べる。

なお、本研究は文部省科学研究費特定研究(1) 課題番号59118003によるものである。

## 参考文献

[Burstall 77] Burstall, R.M. and Darlington, J.: A Transformation System for Developing Recursive Programs, J.ACM, Vol.24, No.1, pp.44-67(1977).

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

[Forgy 82] Forgy, C.L.: Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, Artificial Intelligence, Vol.19, pp.17-37 (1982).

[Gordon 79] Gordon, M.J.C.: The Denotational Description of Programming Languages, Springer-Verlag, New York(1979).

[中島 82] 中島玲二 : 数理情報学入門, 朝倉書店, 東京(1982).

[Nilsson 80] Nilsson, N.J.: Principles of Artificial Intelligence, Tioga, Palo Alto(1980).
邦訳: 白井, 辻井, 佐藤 訳 : 人工知能の原理, 日本コンピュータ協会, 東京(1983).

[岡 83] Lispの表示的意味について, 情処学会ソフトウェア基礎論研究会4-2(1983).

[Stoy 77] Stoy, J.E.: Denotational Semantics: The Scott-Strachey approach to programming language theory, MIT Press, Cambridge(1977).