ソフトウェア工学 46－7
（１９８６．２．６）

述語の動作例を用いてリスト処理プログラムを合成する手法について

仲瀬明彦　　門倉敏夫　　深沢良彰
（早稲田大学理工学部電気工学科）　（相模工業大学工学部情報工学科）

Ｐｒｏｌｏｇプログラム合成の一手法として、プログラムの具体的な動作を記述した述語の例からＰｒｏｌｏｇのリスト処理プログラムを合成する手法と、それを実現するシステムについて述べる。

１．本システムの概要

帰納的推論[1]とは、幾つかの具体的な例からより一般的な法則を推論することをいう。プログラム合成の分野では、プログラムの動作を記述した具体的な入出力の例を与えて、その例の示している動作をより一般的に行なうプログラムを合成する作業が帰納的推論の一例と考えられる。

プログラムの具体的な入出力例からプログラムを合成する手法は、リスト処理の分野では、対象言語をＬｉｓｐに設定して数多くの研究がなされてきた[2]-[4]。また近年、知識情報処理技術の研究の発達等により、Ｌｉｓｐで作成できるプログラムもＰｒｏｌｏｇを用いて作成する要求、利点も論議されてきた[5]-[7]。このような背景により、Ｐｒｏｌｏｇプログラムを述語の具体的な動作例から合成するシステムもいくつか研究されている[6],[8]。

Ｐｒｏｌｏｇのプログラムを具体的な動作例から合成する研究には大別して２つのアプローチがある。１つは、述語内の引数を入力リストと出力リストのペアとして考えた、複数の述語の動作例を与え、それらの入力リスト間の再帰性と出力リスト間の再帰性を発見し、プログラムを合成するもの[8]である。もう１つは、与えられた述語の動作例から、先ずモデルとなる大まかなプログラムを推論し、次にユーザーとそのプログラムに対しての対話を行いながらプログラムを修正し、ユーザーの望むプログラムを推論してゆくもの[6]である。

以上で述べた研究では、単一の関数や述語を合成することに関してはかなり良い結果が示されている。

以上に示したような帰納的推論アルゴリズムを用いてプログラムを合成する場合、１回の合成で作成できるプログラムはごく小さなもの（小さなサブルーチン、関数、述語等）で、大きなプログラムを合成することは出来ない。そこで、それらの手法を用いて大きなプログラムを作成する場合は、プログラムの下位部品を作成する支援システムとして使用されることが効果的であると思われる。しかし、実際に使われているプログラムの下位部品の作成支援を行うためには、帰納的推論で合成できるプログラムの対象範囲を広くしなければならず、そのことによる合成に必要な時間の増大や、合成に必要な動作例の増大は避けられない。

本システムでは、単一の述語の動作例の中で引数として与えられているデータのリストを変形することにより、その変形過程の再帰性を発見し、プログラムを合成する手法をとっている。この手法をとることにより、リストの中のアトムの意味の違いや、入出力、数値データに対する扱いも簡単になるので、幅広い分野のプログラムが合成できる。また、データの変形や再帰性発見においては、完全な探索やマッチングは行わず、統計的に実際のＰｒｏｌｏｇプログラム中で頻繁に使われているパターンのみについての探索やマッチングを行う。これにより、合成の完全性は無くなるが、適用分野の広さに比べて合成に必要な時間(計算量)や、合成に必要な述語の動作例が少なくて済む。

以下に、本システムの用いているプログラム合成のアルゴリズム、実際のプログラム合成の例、合成されたプログラムを用いたより大きなプログラムの作成支援の例について述べる。

2．再帰性とプログラム合成について

再帰的なプログラムは帰納的推論を用いて合成しやすいので、多くのシステムがプログラムの再帰性を発見して再帰的プログラムを合成する手法をとっている[2], [8]。本システムでもこの手法を用いることにし、いかにして再帰的なプログラムを合成するかを以下に述べる。

再帰的なＰｒｏｌｏｇプログラムの実行中においては、入力した定数データの動作も又、再帰的な動きをしている。

（例）２つのリストを結合するプログラム。

```
append([], X, X).
append([A|X], Y, [A|Z]):-append(X, Y, Z).
```

において、質問

```
?-append([a, b], [c, d, e], [a, b, c, d, e]).
```

を入力し、プログラムの振舞いをトレースすると、以下のようになる。

```
append([a|[b]], [c, d, e], [a|[b, c, d, e]])
      :-appned([b], [c, d, e], [b, c, d, e]).        (1)
append([b|[]], [c, d, e], [b|[c, d, e]])
      :-append([], [c, d, e], [c, d, e]).            (2)
append([], [c, d, e], [c, d, e]).                    (3)
```

ここでデータのみに注目すると、以下のようになる。

	第一引数	第二引数	第三引数
(1)	[a|[b]]	[c, d, e]	[a|[b, c, d, e]]
(1)	[b]	[c, d, e]	[b, c, d, e]
(2)	[b|[]]	[c, d, e]	[b|[c, d, e]]
(2)	[]	[c, d, e]	[c, d, e]
(3)	[]	[c, d, e]	[c, d, e]

上の例からも明らかなように、データのみに注目しただけでも再帰的な規則性が発見できる。

上の例では、最初からプログラムが存在しており、これにデータを入力し実際にプログラムを実行させることにより、データの規則的な変形の履歴を得ている。ここで、この操作を逆に辿ってプログラムを得ることを考える。つまり、データとデータの規則的な変形の履歴を最初に与えておき、それらよりプログラムを合成するのである。

従って、本システムがサポートしているデータ変形規則を用いても定数データが変形できなかったり、データ変形の履歴から再帰性が発見できなかった場合は、プログラムの合成は失敗する。

3．システム構成

図１にシステムの構成を示し、各部の詳細を以下に示す。

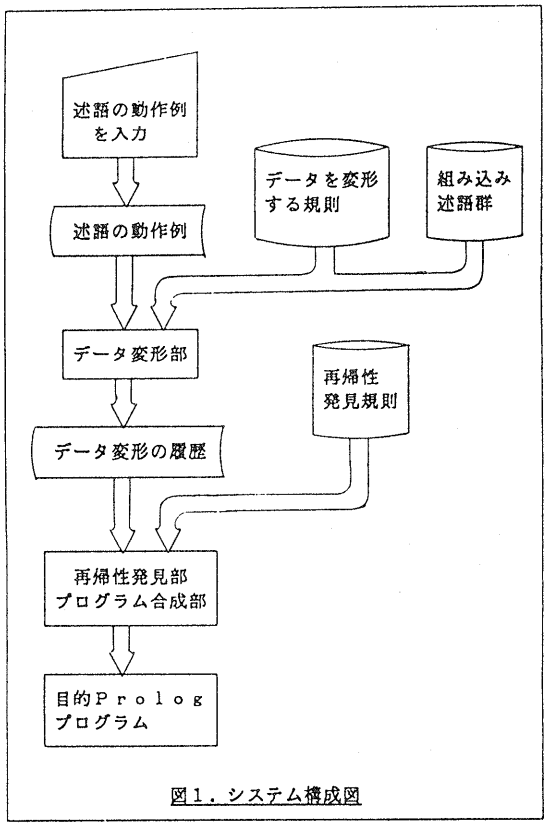


図１．システム構成図

（１）述語の動作例の入力

合成したい述語の具体的な動作を記述したものを入力する。述語内の引数（データ）は、アトム、リスト又は数値でなければならない。但し、引数の数に制限はない。

例えば、リストの反転を取るプログラムを合成したい場合は、

```
reverse([a,b,c,d,e,f,g],[g,f,e,d,c,b,a]).
```

などという動作例の入力が考えられる。

また、以下に本システムにおいて拡張された、述語の動作例の表現方法を示す。

(a) リスト内の要素の意味の違いを扱う場合。

入力文字列のなかである記号が発見された時にその次の文字を削除するプログラムを合成したい場合次の例が考えられる。

```
del([a,b,c,d,e,f,*,g,h,i],
        [a,b,c,d,e,f,*,h,i]).
```

ここでは、"*" のあとの文字を削除する述語を具体的な例で示している。この例では、リスト中の"a,b,c,・・・"はリストの構造を記述するためのアトムとして使われているが、"*" は前者の意味に加えて次のアトムを削除する特殊文字としての意味も持たされている。このようにリスト内のいくつかの要素に特殊な意味を持たせたい場合は、その要素を"{}"で囲んだ動作例を入力する。つまりこの場合は、

```
del([a,b,c,d,e,f,{*},g,h,i],
        [a,b,c,d,e,f,{*},h,i]).
```

という動作例を入力する。

1つのリスト中でこのような指定を複数回行ないたい場合は { } の後に添え字をつけて識別する。

例えば、＊の次の一文字と＃の前の一文字を削除するプログラムを合成したい場合、

```
del2([a,b,c,d,{*}1,e,f,g,{#}2,h,i,j,k,l],
        [a,b,c,d,{*}1,f,{#}2,h,i,j,k,l]).
```

と入力する。

(b)　プログラムの実行中に入出力を行ないたい場合

入出力は、シーケンシャルに行われると考え、目的のプログラムが呼出されてからの入出力にもシーケンシャルな通し番号を付け、その番号をｎとする。

入力は、inp(n)(入力ストリームからの入力データ)、出力は、outp(n)(出力ストリームへの出力データ)，で示される。入出力はこの通し番号の順番で行なわれる。入出力ストリームをプログラムの実行中に変化させる事は、現在はサポートしておらず、それらはキーボードからの入力とディスプレイへの出力に設定されている。

例えばリストが与えられており、次に数字ｎを読み込み、リスト内の要素でリストの最初からｎ番目にある要素を書き出すプログラムに対しては、

```
writeatom([a,b,c,d,e,f,g,h,i]),
        inp(1)(3),outp(2)(c).
```

という動作例を入力する。

（２）データを変形する規則とその適用方法

本システムでは、付録１のデータ変形規則に従って述語の動作例内の引数データを変形する。付録１のデータ変形規則は必ずしも必要十分なものではないが、Ｐｒｏｌｏｇのプログラムの例が多く記載されている文献中[9],[10]で瀕出しているＰｒｏｌｏｇの事実（Fact）や規則（Rule）を優先的に抽出し、それと等価に働くデータ変形規則に置き換えたものである。以下の図２は、リスト処理のＰｒｏｌｏｇプログラム中で使われている最下位の述語７０個をサンプリングした結果を示している。

(1)	述語内の引数が全て空リストになると終了	7
(2)	述語内の引数の中で、空リストでない変数が１つになると終了	4
(3)	述語内の引数の中で、リストの頭部又は尾部が等しければ取り除く	１３
(4)	述語内のいくつかのリストを、頭部と尾部に分ける	9
(5)	(1)-(4)の操作を行い、且つ入出力を行う	1
(6)	(1)-(4)の操作を行い、且つ数値を１減らす	2
(7)	数値が０になると終了	2
(8)	本システムの組み込み述語を適用して終了	１１
(9)	その他	２１

図２．Ｐｒｏｌｏｇプログラム中で瀕出するパターン

付録１のデータ変形規則を総あたりで述語の引数データに適用すれば、必ず一つは目的のデータ変形履歴が得られるが、なるべく高速にデータ変形履歴を得るために付録２に示すアルゴリズムでデータ変形規則を適用する。

また図３は、付録１のデータ変形規則中で使われている本システムの組み込み述語を示したものである。

equal(X,X,・・・).	引数の値が同じ
car([X|Y],X).	片方の引数の頭部が他の引数と同じ
cdr([X|Y],Y).	片方の引数の尾部が他の引数と同じ
cons(X,Y,[X|Y]).	１つの引数の頭部と尾部が他の２つの引数と同じ

図３．本システムが用いている組み込み述語

（３）データ変形履歴からのプログラムの合成

プログラムの合成は、（２）で作成されたデータ変形履歴を用いて以下のステップに基づいて行われる。（以下で現れる"述語記号K"とは、最初に述語の動作例として入力した述語の述語記号に数字K を付けたものをいう。）

〈1〉 データ変形履歴の述語形式への変換。

このステップは、以下の手順で行われる。

(1) "述語記号1
(〈規則が適用出来なくなった時のデータ〉)."
を登録する。

(2) データ変形履歴に最後に登録されたものから順番に、規則を適用した各段階ごとに、
"述語記号N+1
(〈規則を適用する前のデータ〉):-
述語記号N
(〈規則を適用した後のデータ〉)."
を登録する。

(3) "最初に例として入力した述語:-
述語記号Nmax
(〈例として入力した述語のデータ〉)."
を登録する。

数値を扱っている場合は数値を１減らす述語を、入出力を行う場合はread文やwrite文を、それぞれ":-"の後に挿入する。

〈2〉 定数記号の変数化。

〈1〉で出来たプログラムの各述語（又は述語から述語への規則）の中で、同値のリストや数値が使われている場合は、データを変形した規則を参照しながら、そのリストや数値を変数に置き換える。但し、３．(1).(a)に規定されている特殊な意味をもつ要素に対しては、変数化は行わず定数として扱う。

〈3〉 再帰性発見、プログラム合成

〈3〉のプログラムから再帰性を発見し、入力した述語の動作例の引数データ以外のデータに対しても働くプログラムを作成する。隣り合った節が同じパターンの規則を用いている場合、そこが再帰性をもっていると考える。それら複数個の隣り合った述語の述語記号を固定した述語記号に置き換え、隣接している述語の最上端と最下端のみの述語を残す。つまり、〈2〉で出来たプログラム中で節（P2:-P1,P3:-P2,・・・,Pn+1:-Pn)が同じパターンの述語である時、述語記号の固定された述語をＰとすると、P:-P1,P:-P,Pn+1:-Pとする。この手続きをそれ以上適用出来なくなるまで、繰り返し適用する。

４．　システムの使用法

本システムは、ユーザーの入力した具体的な動作例を１つ読み込み、合成を行う。システムによって合成されたプログラムはユーザーが適当なテストを行うように設計されている。また、データ変形規則からも分かるように、規則(3),(4),(5),(6),(9)が非決定的な規則となっており、複数のデータ変形の履歴が生成される。もし最初のデータ変形履歴から合成されたプログラムがユーザーの意図しないものであれば、システムは次のデータ変形履歴から再度プログラムを作り直す。

(1) リスト中の最初から数えてｎ番目にある要素を取り出す述語　：substring(X, N, A).

入力した述語の動作例。
substring([a,b,c,d,e],3,c).

(2) リスト中のあるアトム又はリストを他のアトム又はリストに置き換える述語　：subst(X, A, B, Y).

入力した述語の動作例。
subst([a, b, c, d, e, f, g], c, x, [a, b, x, d, e, f, g]).

(3) リスト中に"is"というアトムがあった場合そのアトムとその１つ前のアトムを入れ替える述語：quest(X, Y).

入力した述語の動作例。
quest([he, [is], a, nice, boy],
[[is], he, a, nice, boy]).

図４．本システムを用いて合成できた述語の例

５．　システムの評価と応用分野

本システムを用いての小さなプログラムの合成の例を付録３に示す。本システムをプログラムの合成に使用した結果、かなり広い範囲の有用なリスト処理用プログラムが合成できることが分かった（図４）。また、合成を行った例に対しては、ほとんど１回の試行で目的プログラムを合成出来た。

小さなプログラムを合成することに対して有用であることは勿論であるが、大きなシステムを作成する場合の支援システムとしても利用できる。このシステムを用いて簡単な行エディタの作成をした例を付録４に示す。

６．今後の課題

（１）作成されたプログラムのテストは、人手で１つ１つテストデータを入れてテストしなければならないが、システムの方で典型的な述語の入出力データを示してくれるようにする。これにより、テストが容易になる。

（２）現在は単一の述語の動作例の中のリストの要素の再帰性にのみ注目しているが、複数の述語の動作例の間の関係も発見できるようにする。

（３）述語の動作例の入力に於いて、真になる動作例だけでなく、偽になる動作例も入力できるようにする。（２）と（３）により、合成できるプログラムの適用分野が一層広がると思われる。

（４）付録１のデータ変形規則と付録２の再帰性発見アルゴリズムにより多くの規則をもたせれば、合成できる述語の範囲は広がるが、どの程度までそれらを拡張すれば合成に必要な計算時間の増大に比べて効果的な拡張ができるか考察してゆく。

参考文献

［１］ Angluin, D and Smith, C. H. : Inductive Inference Theory and Methods, Computing Surveys Vol. 15, No. 3 (1983).

［２］ Summers, P. D. : A Methodology for LISP Program Construction from Examples, J. ACM Vol. 24, No. 1 (1977).

［３］ Hardy, S. : Synthesis of Lisp Funcitons from examples, IJCAI (1975).

［４］ Shaw, D. E., Swartout, W. R. and Green, C. C. : Inferring Lisp Programs from Examples, IJCAI (1975).

［５］ Computer Today (1984. 1).

［６］ Shapiro, E. Y. : Algorithmic Program Debugging, MIT Press (1982).

［７］ 情報処理 Vol25. No. 12 (1984).

［８］ Yokoi, S. : Inference of Programs from Examples. 情報処理学会29回全国大会

［９］ 安部 : Ｐｒｏｌｏｇプログラミング入門 共立出版 (1985).

［１０］ Clocksin, C. S. and Mellish, C. S. 中村訳 : Ｐｒｏｌｏｇプログラミング (1983).

付録１．述語内のデータを変形する規則の概要

＜再帰の修了条件となるもの＞
(1) 例内のデータが全て空リストになったら、規則の適用を終了する。
（Ｐｒｏｌｏｇでは、pred([],[],･･･,[]). に相当）
(2) 例内のデータに規則を適用できなくなったら、規則の適用を終了する。
（規則が適用できなくなる条件とは、複数個ある最初に与えられた引数としてのデータの中で、空リストでないものが１つとなった場合である。）

＜基本リスト処理手続き＞
(3) 例内のデータ中のｎ個のデータの頭部が同じならそれを取り除く。
（Ｐｒｏｌｏｇでは、pred([A|X],[A|Y],･･･,Z):-pred(X,Y,･･･,Z). に相当）
(4) 例内のデータ中のｎ個のデータの尾部が同じならそれを取り除く。
（Ｐｒｏｌｏｇでは、pred([A|X],[B|X],･･･,K):-pred(A,B,･･･,K). に相当）
(5) 例内のデータ中のあるデータを頭部と尾部に分ける。
（Ｐｒｏｌｏｇでは、pred1([A|B],C,･･･,K):-pred2(A,B,C,･･･,K). に相当）
(6) 例内のデータ中の２つのデータを頭部と尾部に持つデータを作る。
（Ｐｒｏｌｏｇでは、pred1(A,B,･･･,K):-pred2([A|B],･･･,K). に相当）
［注：(5) の規則は、繰り返し適用すると、述語の引数の数が増大する一方で、後に再帰性を発見することの妨げとなる。そこで、ｎ個の引数をもつ述語の例を入力した場合、データの変形の過程では、データの個数はｎ＋１個までしか増大しないような制約を設ける。(5) の規則を繰り返し適用する場合には、(5) と (6)をペアにして使う。即ち、データ中のあるデータを頭部と尾部に分け、以前に規則(5) により分離されたデータを尾部に持ち、規則(6)によって分離されたデータを頭部に持つデータを作成する。Ｐｒｏｌｏｇのプログラムで表わすと、pred(A,[P|B],C,･･･,K):-pred([P|A],B,C,･･･,K). と等価である。］

＜入出力に関するもの＞
(7) 与えられた述語の動作例内にoutp(n)(X)がある時、例内のデータ中のあるデータに対して(3),(4)を行い、かつその操作で分離されたものがoutp(n)(X)のXであれば、Xを出力する。
（Ｐｒｏｌｏｇでは、pred([A|X],･･･,Z):-write(A),pred(X,･･･,Z). に相当）
(8) ある項を入力して例内のあるデータの頭部に結合する。
（Ｐｒｏｌｏｇでは、pred(X,･･･,Z):-read(A),pred([A|X],･･･,Z). に相当）
［注：数値を入出力する場合には、各々述語内の引数の数を１つ増やすか減らすかする。
各々　pred(A,･･･,N)
　　:-read(X),pred1(X,A,･･･,N).
　　pred(A,B,･･･,N)
　　:-write(A),pred1(B,･･･,N).
に相当する。］

＜数値に関するもの＞
(9) 例内のデータ中に数値がある場合、(3),(4),(5),(6)を行い、数値データの値を１減らす。
（Ｐｒｏｌｏｇでは、pred(X,M,･･･,Z):-N is M-1,pred(X,N,･･･,Z). に相当）
［注：この規則は、２つ以上の数値データを含む述語に対しては、数値データの値を同時に１減らすことは行なわず、まず１つの数値データを減らしてゆき０にしようと試みる。］
(10) 例内のデータ中の数値データの値が０であれば、その引数に対する操作を終了する。
（Ｐｒｏｌｏｇでは、pred(A,B,0,･･･):-pred(A,B,0,･･･). に相当）

＜組み込み述語に関するもの＞
(11) 動作例pred(X1,X2,･･･,Xn)に対してＴの値を持つ述語を動作例に適用し、規則の適用を終了する。
［システムがあらかじめ用意している組み込み述語は、Ｌｉｓｐの最も基本的な関数を使用しており、付録２に示してある。］

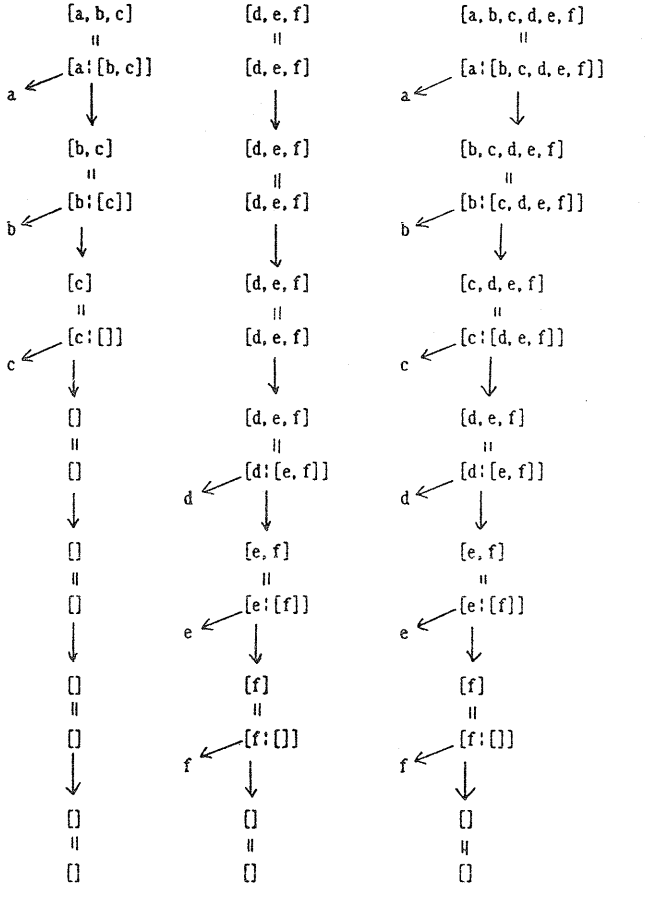
付録２．データ変形規則を適用するアルゴリズム

```
procedure hitory(データ並び);
  if (1)or(2) が成立then
      begin 履歴ファイルに記録;
            return
      end
  else if (11) が成立  then
      begin 履歴ファイルに記録;
            return
      end
  else if データ並びの中にinp(n)(A)
          又はoutp(n)(A)がある  then
      begin nの最も小さいinp又はoutp
              に対して(7) 又は(8)を行なう;
            履歴ファイルに記録;
            新データ並び:-データ並びを(7)又は
                          (8)で変形したもの;
            history (新データ並び)
      end
  else if  データ並びの中に数値がある  then
      begin (10)又は(9) を適用;
            履歴ファイルに記録;
            新データ並び:=データ並びを(10)又は
                          (9)で変形したもの;
            history( 新データ並び)
      end
  else if (3)又は(4) が適用できる  then
      begin (3) 又は(4) を適用;
            履歴ファイルに記録;
            新データ並び:=データ並びを(3)又は
                          (4)で変形したもの;
            history( 新データ並び)
      end
  else if データ並び中の述語の引数の数が、最初
          に入力したデータの引数の数と同じ  then
      begin (5) を適用;
            履歴ファイルに記録;
            新データ並び:=データ並びを
                          (5) で変形したもの;
            history( 新データ並び)
      end
  else
      begin (5) と(6) を同時に適用;
            履歴ファイルに記録;
            新データ並び:=データ並びを(5)と
                          と(6) で変形したもの;
            history (新データ並び)
      end
end history
```

付録３　簡単で基本的な合成例

入力として、" append([a, b, c], [d, e, f], [a, b, c, d, e, f]). " を入れたとする。

[1] 入力データを変形する規則が適用される。



[2] 入力データに対してのみ正しく働くプログラムを合成する。

```
append1([], [], []).
append2([], [f|[]], [f|[]]):-append1([], [], []).
append3([], [e|[f]], [e|[f]]):-append2([], [f], [f]).
append4([], [d|[e, f]], [d|[e, f]]):-append3([], [e, f], [e, f]).
append5([c|[]], [d, e, f], [c|[d, e, f]]):-append4([c], [d, e, f], [c, d, e, f]).
append6([b|[c]], [d, e, f], [b|[c, d, e, f]]):-append5([c], [d, e, f], [b, c, d, e, f]).
append7([a|[b, c]], [d, e, f], [a|[b, c, d, e, f]]):-
                append6([b, c], [d, e, f], [b, c, d, e, f]).
append([a, b, c], [d, e, f], [a, b, c, d, e, f]):-
                append7([a, b, c], [d, e, f], [a, b, c, d, e, f]).
```

[3] 述語内の定数を変数化する。

```
append1([], [], []).                                          (1)
append2(X2, [A2|Y2], [A2|Z2]):-append1(X2, Y2, Z2).           (2)
append3(X3, [A3|Y3], [A3|Z3]):-append2(X3, Y3, Z3).           (3)
append4(X4, [A4|Y4], [A4|Z4]):-append3(X4, Y4, Z4).           (4)
append5([A5|X5], Y5, [A5|Z5]):-append4(X5, Y5, Z5).           (5)
append6([A6|X6], Y6, [A6|Z6]):-append5(X6, Y6, Z6).           (6)
append7([A7|X7], Y7, [A7|Z7]):-append6(X7, Y7, Z7).           (7)
append(X, Y, Z):-append7(X, Y, Z).                            (8)
```

[4] 隣接した規則の間のパターンを調べ、述語記号を固定できるものは行なう。

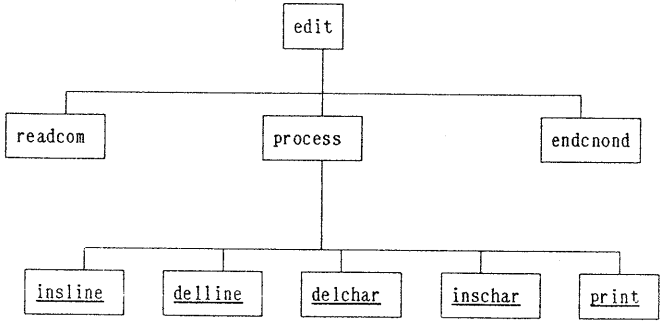
ここでは、(2), (3), (4)で使われている述語を１つの述語appendr_1 とし、(5), (6), (7)で使われている述語を１つの述語appendr_2 とする。プログラムは、以下のようになる。

```
append1([], [], []).
appendr_1(X, Y, Z):-append1(X, Y, Z).
appendr_1(XR1, [AR1|YR1], [AR1|ZR1]):-appendr_1(XR1, YR1, ZR1).
append4(X, Y, Z):-appendr_1(X, Y, Z).
appendr_2(X, Y, Z):-append4(X, Y, Z).
appendr_2([AR2|XR2], YR2, [AR2|ZR2]):-appendr_2(XR2, YR2, ZR2).
append7(X, Y, Z):-appendr_2(X, Y, Z).
append(X, Y, Z):-append7(X, Y, Z).
```

付録４．本システムを使っての簡単な行エディタの作成

作成する行エディタの機能は、以下の通りである。
（１）テキストの入出力
（２）行の削除
（３）行の挿入
（４）文字の削除
（５）文字の挿入
（６）テキストの表示

おおまかなプログラムの構成は以下の通りである。



下線の部分が主に本システムを用いて合成出来た部分である。

また、合成に用いた述語の動作例を以下に示す。

(1) <delline : １行削除 >
```
delline([[a,b,c,d,e],[f,g,h,i],[j,k,l,m,n,o],[p,q,r]],3,
                    [[a,b,c,d,e],[f,g,h,i],[p,q,r]]).
```

(2) <insline : １行挿入 >
```
insline([[a,b,c,d],[e,f,g],[h,i,j,k,l,m],[n],[o,p,q,r]],3,[x,y,z],
          [[a,b,c,d],[e,f,g],[x,y,z],[h,i,j,k,l,m],[n],[o,p,q,r]]).
```

(3) <getline : １行抽出 >
```
getline([[a,b,c],[d,e,f,g],[h,i,j,k,l],[m,n,o]],3,[h,i,j,k,l]).
```

(4) <subst : １文字置き換え >
```
subst([a,b,c,d,e,f,g,h,i],e,x,[a,b,c,d,x,f,g,h,i]).
```

(5) <delchar : １文字削除 >
```
delchar([a,b,c,d,e,f,g],5,[a,b,c,d,f,g]).
```

(6) <inschar : １文字挿入 >
```
inschar([a,b,c,d,e,f,g],4,x,[a,b,c,x,d,e,f,g]).
```

(7) <printl : １行表示 >
```
printl([[a,b,c,d],[e,f,g,h],[i,j,k,l],[m,n,o,p],[q,r,s,t]],4),
                        outp(1)([m,n,o,p]).
```

本システムを用いて合成出来た部分の全体に対する割合を以下に示す。

	大きさ（バイト）	ステップ数（"."の数）
プログラム全体	２５７９	３０
最下位の部分	１４３０	２１
本システムで合成できた部分	９１０	１６

プログラムの大きさで考えると、プログラム全体に対して約３５％、最下位のルーチン中で約６４％のプログラムが本システムを用いて合成できた。
またこれを、プログラムのステップ数に注目して考えると、プログラム全体に対して約５３％、最下位のルーチン中で約７６％のプログラムの合成に成功した。