Prolog 等価変換エディタと変換戦略†

中川裕志†† 中村直人††

我々は，与えられた Prolog プログラムを仕様とみなして，効率のよいプログラムへの変換を検討している．その基本的手法として，unfold, fold といった等価変換を採用している．そこでまず，unfold, fold を基本コマンドにもつプログラム変換エディタ“TRANS”を開発した．さらに，機械的判断によって適用しうる変換の幾つかを，評価関数をもった半自動コマンドとして備えている．この TRANS を利用することによってプログラムの等価変換が簡単かつ確実に行えるようになった．次に我々は，TRANS 上での変換を解析し，効率化への変換パターンの典型例を見いだした．このような変換パターンは個々の述語に依存したものではないので，等価変換における戦略的知識と考えることができる．そこで，変換戦略を記述した言語により TRANS を制御して自動変換を行うインタプリタ“PAROTS”を製作した．先に得た各種変換パターンの知識を，これを利用して記述することにより，自動変換規則としての能力を試すことができた．

1. まえがき

Robinson の提案した resolution により述語論理表現を計算可能なプログラムとして扱えることが明らかになって以来，述語論理で表現された仕様から演えき操作によりプログラムを合成しようとする Hogger[1], Bible[2] らの研究の流れがある．一方，Burstall, Darlington らは，関数型プログラミングにおいて，unfold, fold を中心とする方法によるプログラムの効率化変換の研究[3),4)]を進めている．佐藤，玉木らは，unfold, fold を述語論理型言語 Prolog に適用し，プログラムの変換，合成の研究[5)-7)]をすすめている．

我々は，以上のような研究の流れを踏まえて，Prolog プログラム効率化の観点での検討を行ってきた．すなわち，与えられた Prolog プログラムを仕様とみなして，unfold, fold といった等価変換を基本とした手法での，効率のよいプログラムへの変換を検討してきた．変換においては，プログラムの等価性の保存，述語に依存した急進的な過程の排除に注意を払っている．その結果，幾つかの変換パターン例を得ることができた．

本稿では，まず pure Prolog を対象としたプログラム変換エディタ TRANS[8)] を紹介する．そして TRANS 上での変換例を通じて，変換パターンを解析し，効率化への着眼点を示す．最後には，変換戦略を記述した言語によって TRANS を制御して自動変換を行うインタプリタ PAROTS (Prolog Automatic Ratifier Of Transformation Strategy) を紹介する．

2. プログラム変換エディタ TRANS

2.1 エディタの環境

このエディタは，Prolog インタプリタとともに Lisp 上に記述されている．文献 4) に紹介されているシステムに似たインタラクティブな操作により，unfold, fold 等の変換を簡単かつ確実に行うことができる．

ただし，エディタは副作用のある述語は対象としていない．

2.2 基本コマンド

エディタは変換しようとする述語名をもって起動される．コマンドは述語の定義に対する変換命令と考えられる．次に示す主なコマンドは，等価変換のプリミティブとなるものである．

なお，( ) の内にコマンド名を示す．

- Unfold (U) Procedure を，指定した goal について unfold する．いわゆる探索木の展開である．指定した goal にユニファイできる head をもつ clause がなければ，閉世界仮説の下での deletion となる．
- Fold (F) Procedure に現れる指定した goal の並びを，fold 条件[5)] を満たす述語定義により置換する．
- Goalの統合 (SG) 指定した二つの goal がユニファイできれば統合する．
- 等価則の適用 (SE) いわゆる Laws である．エ

† Prolog Transformation Editor and Strategies for Transformation by HIROSHI NAKAGAWA and NAOHITO NAKAMURA (Department of Computer Engineering, Faculty of Engineering, Yokohama National University).

†† 横浜国立大学工学部情報工学科

ディタ外部にもっている等価則データベース・ファイルを用いて，知識による変換を行う．本稿では，append の結合則が蓄えられているとする．等価則は fold と同様な条件[9]だけチェックして適用される．それ以外の等価性は，個々の規則を与えたユーザの責任となる．つまり，条件だけ満たす Eureka を意図的に加えることもできる．

●Goal の並べ変え (UG) 指定した goal の clause 内での位置を変える．goal が副作用をもたない場合，procedure 中の clause の実行順序は問題とならないが，clause 中の goal の実行順序はプログラムの停止性に係わることがある（例を 3.1 節に示す）．

2.3 semi-automatic コマンド

上述の基本コマンドを，適当な評価の下で自動的に実行する semi-automatic コマンドがある．ここでいう自動とは，適用する部分の自動選択，および，適用可能性の自動判定の二つの意味である．これらのコマンドには，プログラム変換における基礎的なヒューリスティックスがインプリメントされている．そのため，変換の総ステップ数の減少や，機械的な判断の一部自動化が達成されている．

●一意展開 (AU) unfolding で clause が一意に選択されて条件分岐が生じないものを，「一意展開」と呼んでいる．このコマンドは，一意展開できる goal を procedure 中から見いだし unfold するものである．一意展開可能な場合は，procedure としての goal の実行順序に関係なく行われることに注意されたい．

●実行順序の評価 (AT) Prolog の goal は通常 head に近いものから順に実行されるが，その実行順序によってバックトラックの回数が異なる場合がある．そこで mode 宣言を利用して，goal の実行と各変数の instantiate との関係を解析する．この情報を用いて goal の実行順序を評価する．そして，変数がすべて instantiate された（テスト的に振る舞う）goal や，入力変数がすべて instantiate された（解を発生する条件が整った）goal を必要があれば前方へ移動する．

●等価 goal の統合 (AM) mode 宣言によって goal を関数とみなして，入力が等しい goal を見いだし統合する．厳密な等価変換では入出力の区別なく同一な goal だけ統合できたが，ここでは制約を入力変数の同一性だけに緩めてある．よって，多価関数に適用された場合，等価性が損なわれることもある．

●自動 folding (AF) 自分自身の（または他の述語の goal 並びより作成した新述語の場合，その「親」の）定義節と fold できる goal 並びを探して，もしあれば fold する．

●新述語作成 (MFC) 新述語を，変換している述語に現れる goal の並びから作り出す．そして，変換の注目を新述語へ移す．情報が十分であれば新述語の mode 宣言も作り出せる．

●等価則の自動適用 (AE) 適用可能な等価則が，データベース・ファイル上にあれば利用する．

3. TRANS の使用例

2章で示したコマンドのほかに，変換セッションの便宜を図るコマンドがあるので略記する．

RP : MFC での「親」へ戻る．
ARP : MFC での「親」へ戻り，新述語は捨てる．
FRD : MFC で作った新述語によって，「親」を fold する．多くの変換は，MFC, 新述語の効率化，そして FRD と経過するので，このコマンドは MFC, RP とともに用いられる．
D : 述語の定義節に戻り，変換をやり直す．
OB : 最新の一コマンドを取り消す．
SB, B : 変換の途中状態を保存し (SB)，そこからやり直す (B)．

3.1 semi-automatic コマンドの利用

簡単な例を用いて semi-automatic コマンドの評価関数とその効果を示す．

●実行順序の評価 mode 宣言を利用した test-goal の優先による効率化については 3.2 節で述べるので，ここでは generator-goal の適切な配置について示す．リストから二要素を得る述語 pickup 2 を考える．(3.1) のプログラムでは，変数 LIS のみ値を定めて pickup 2 を呼ぶと正しい幾つかの解を発生した後，止まらなくなってしまう．しかし，mode 宣言を調べれば pickup 2 が呼ばれた時点では LIS だけ値が決定しているので，2番目の tail の方が優先すべき generator であることが分かる．よって，(3.2) が得られる．ユーザは，tail をリストを分解に用いていることだけを知っていればよい．

```
pickup 2 (LIS, FST, SND)←
    tail (REST, SND, DUST), tail (LIS, FST,
    REST).
tail ([ITEM|TAIL], ITEM, TAIL).
tail ([CAR|CDR], ITEM, TAIL)← tail (CDR,
```

```
ITEM, TAIL).
mode pickup 2 (+, −, −)  mode tail (+, −, −)   (3.1)
```

AT 実行後

```
pickup 2 (LIS, FST, SND)←
    tail (LIS, FST, REST), tail (REST, SND, DUST).   (3.2)
```

●一意展開 階層的に記述されている述語などで，パターン的に一意に実行される部分を同一レベルに表現することができる．(3.3) の例では，階層的記述や組込述語（ただし，pure Prolog でも記述できるもの）によって表現されているが，これを(3.4)へ展開することができる．

```
W_occur (LIST, FST)←
    pickup 2 (LIST, FST, SND), FST=SND.   (3.3)
```

AU 実行後

```
W_occur (LIST, FST)←
    tail (LIST, FST, REST), tail (REST, FST, DUST).   (3.4)
```

3.2 コマンドを組み合わせた効率化

3.1 節では，semi-automatic コマンドが直接適用可能な例を示したが，ここではコマンドを組み合わせたより実際的な例を示す．(3.5)の generate and test 型のソートプログラムの変換において，semi-automatic コマンドが変換時の思考レベルにいかに対応しているかに注目されたい．

```
sort (IN, OUT)←permutate (IN, OUT), sorted (OUT).
mode sort (+, −)   (3.5)
```

なお，permutate と sorted は以下のように定義されている．

```
permutate ([ ], [ ]).
permutate (LIS, [CAR|CDR])←
    into (LIS, CAR, REST), permutate (REST, CDR).
into ([AIM|REST], AIM, REST).
into ([CAR|CDR], AIM, [CAR|REST])←
    into (CDR, AIM, REST).
sorted ([ ]).
sorted ([ANY]).
sorted ([FST, SND|REST])←
    FST≦SND, sorted ([SND|REST]).
mode permutate (+, −)  mode into(+, −, −)
mode sorted (+)  mode ≦(+, +)   (3.6)
```

以上の述語を Prolog のインタプリタ上で定義した後，述語 sort について TRANS を起動する．以下が実際の TRANS のセッションであり，イタリック体の部分がユーザによる入力である．

(変換)

```
command : U          ; 第一 goal を unfold
n-th goal : 1
sort ([ ], [ ])←sorted ([ ]).
sort (IN, [CAR|CDR])←
    into (IN, CAR, REST), permutate (REST, CDR), sorted ([CAR|CDR]).
command : AU         ; 一意展開
sort ([ ], [ ]).
sort (IN, [CAR|REST])←
    into (IN, CAR, REST), permutate (REST, CDR), sorted ([CAR|CDR]).
command : MFC        ; 新述語作成
n-th clause : 2 from n-th goal : 2
how many goals : 2 predname : sort#1
mode declaration is sort#1 (+, −, +)
sort#1 (REST, CDR, CAR)←
    permutate (REST, CDR), sorted ([CAR|CDR]).
command : U n-th goal : 1   ; 展開
sort#1 ([ ], [], CAR)←sorted ([CAR]).
sort#1 (REST, [CAR|CDR], A)←
    into (REST, CAR, B), permutate (B, CDR), sorted ([A, CAR|CDR]).
command : AU         ; 一意展開
sort#1 ([ ], [ ], CAR).
sort#1 (REST, [CAR|CDR], A)←
    into (REST, CAR, B), permutate (B, CDR), A≦CAR, sorted ([CAR|CDR]).
command : AT         ; test (≦) の優先
sort#1 ([ ], [ ], CAR).
sort#1 (REST, [CAR|CDR], A)←
    into (REST, CAR, B), A≦CAR, permutate (B, CDR), sorted ([CAR|CDR]).
command : AF         ; 自動 fold
sort#1 ([ ], [ ], CAR).
sort#1 (REST, [CAR|CDR], A)←
    into (REST, CAR, B), A≦CAR, sort#1 (B, CDR, CAR).
```

```
command : RP                ; sort の変換へ戻る
command : FRD               ; 新述語で fold
sort ([ ], [ ]).
sort (IN, [CAR|CDR])←
    into (IN, CAR, REST), sort # 1
    (REST, CDR, CAR).
```

4. 効率化に役立つコマンド列の解析

自動的な効率化変換の準備として，幾つかのプログラムに対する TRANS による効率化の例を見る．

【魔方陣】

```
magic 3 (IN, OUT)←
    permutate (IN, OUT), magic 3 test (OUT).
magic 3 test (OUT)←
    getsum (OUT, SUM), alleq (SUM).
getsum (OUT, SUM)←
    linesum (OUT, LS), columsum (OUT, CS),
    crosssum (OUT, XS), append (LS, CS, LSC),
    append (LCS, XS, SUM).
linesum ([ ], [ ]).
linesum ([FST, SND, TRD|REST], [ST|SR])←
    add 3 (FST, SND, TRD, ST), linesum
    (REST, SR).
columsum ([UL, UM, UR, ML, MM, MR, DL,
DM, DR], [FST, SND, TRD])←
    add 3 (UL, ML, DL, FST), add 3 (UM, MM,
    DM, SND), add 3 (UR, MR, DR, TRD).
crosssum ([UL, UM, UR, ML, MM, MR, DL,
DM, DR], [LR, RL])←
    add 3 (UL, MM, DR, LR), add 3 (UR, MM,
    DL, RL).
alleq ([ANY]).
alleq ([FST, SND|REST])←
    FST=SND, alleq ([SND|REST]).
mode magic 3 (+, −) mode add 3 (+, +, +, −)
                                        (4.1)
```

コマンド列=AU, AT

(結果)

```
magic 3 (IN, [UL, UM, UR, ML, MM, MR, DL,
DM, DR])←
    into (IN, UL, REST), into (REST, UM, A),
    into (A, UR, B), into (B, ML, C),
    add 3 (UL, UM, UR, ST), into (C, MM, D),
    into (D, MR, E), add 3 (ML, MM, MR, ST),
    into (E, DL, F), add 3 (UL, ML, DL, ST),
    add 3 (UR, MM, DL, ST), into (F, DM,
    [DR]),
    add 3 (DL, DM, DR, ST), add 3 (UM, MM,
    DM, ST),
    add 3 (UR, MR, DR, ST), add 3 (UL, MM,
    DR, ST).                            (4.2)
```

一意展開と実行順序の評価それぞれが効率化である．

【sort】

変換は，3.2 節を参照のこと．

コマンド列=U, AU, MFC , FRD 新述語 : , U, AU, AT, AF

新述語における実行順序の評価が効率化である．効率化に至るまでに，sort での unfold される goal，および，新述語を作る goal 並びの決定が探索空間となる．また，新述語では，効率化の後，再帰定義になることも必要であろう．

【reverse】

```
reverse ([ ], [ ]).
reverse ([CAR|CDR], ANS)←
    reverse (CDR, R), append (R, [CAR],
    ANS).                               (4.3)
```

コマンド列=MFC , FRD 新述語 : , U, AU, AE, AU, AF

(結果)

```
reverse ([ ], [ ]).
reverse ([CAR|CDR], ANS)←
    reverse # 2 (CDR, [CAR], ANS).
reverse # 2 ([ ], A, A).
reverse # 2 ([CAR|CDR], A, ANS)←
    reverse # 2 (CDR, [CAR|A], ANS).
                                        (4.4)
```

端的に言ってこの場合の効率化は，スタック式の reverse プログラム生成によるものである．しかし，このようなセマンティクスを変換に持ち込むと一般性がまったくなくなってしまう．そこでコマンド列の特徴としては，新述語で U, AU（展開を尽くした）の後 AE（等価則の利用）を行い，その結果，AU が行えた（展開が進んだ）と捉えることになる．

5. 変換戦略表現システム PAROTS

前章で述べた変換例を参考にして，一般的な変換戦略について考える．その準備として，変換コマンド列を表現する言語（「PAROTS 言語」と呼ぶ）を定義する．この言語は TRANS のコマンド列を S 式をベースにして表現し，制御を表現する IF THEN ELSE と TRY そして END を追加したものである．IF THEN ELSE は IF の条件にあるコマンドが作用できたか，できなかったかによって THEN 部，あるいは ELSE 部に制御を移す．たとえば，実行順序を評価して goal が並べ変えられたかで制御を変えるなら，

(IF AT) (THEN…) (ELSE…)

のような形になる．TRY は unfold や新述語の切り出し MFC をどの部分に適用するかに関する探索を，while 文型の制御で実行するものである．また IF 判断のために SINGLE-CLAUSE（述語が単一の clause であるかどうかをチェックする）などを導入する．

たとえば定義節に対して unfold が探索の対象であれば，それ以後の変換を含めた変換戦略は，PAROTS 言語では次のように表される．

```
(IF SINGLE-CLAUSE) (THEN
  (TRY (U)…END を含んだ unfold 後の変換
  …))                                    (5.1)
```

unfold 後の変換で効率化されたなら，END を行うことにより変換はその場で終了する．END が実行されない場合は，述語が自動的に元の状態に戻され，別の goal の unfold が試される．さらに，unfold の選択肢をすべて尽くしてしまうと，実行する前の状態で TRY を終了する．

変換時に unfold などの探索が多重になる複雑な場合は，必要なだけ TRY をネストさせて記述することができる．

一方，サブルーチンを許し，再帰呼び出し的な方法も考えられるが，

- 異なった変換戦略を表現するのであるから，各レベルごとにカスタマイズされた個々の戦略を記述するのが基本的立場であろう．
- 既知の変換には，任意の深さの TRY のネストを必要とする例がなくすべて固定階で記述できている．
- 任意の深さの処理を許した場合，自動変換の探索空間が無秩序に大きくなる可能性があり，それを管理する知識と手段が必要になる．

といった理由からサブルーチンを採用していない．

5.1 変換パターンの記述法

4 章で述べた幾つかの変換例および掲載しなかったその他の変換から，初歩的な自動変換システムを PAROTS 上に記述することができる．

- 一意展開による効率化

```
(IF AU) (THEN END)                       (5.2)
```

- 実行順序の評価による効率化

```
(IF AT) (THEN END)                       (5.3)
```

- 定義節の再帰化

```
(IF SINGLE-CLAUSE) (THEN
  (TRY (U) AU
  (IF AF) (THEN END)))                   (5.4)
```

- 新述語＋等価則による効率化

```
(TRY (MFC)
  (TRY (U) AU AE
    (IF AU) (THEN
      (IF AF) (THEN RP FRD END))))
                                         (5.5)
```

- 新述語＋等価 goal 統合による効率化

```
(TRY (MFC)
  (TRY (U) AU
    (IF AM) (THEN
      (IF AF) (THEN RP FRD END))))
                                         (5.6)
```

これらが行う探索には共通の部分（たとえば，(5.5) と (5.6) では，ともに新述語を作成したのち展開している）があるので，実際には集約してから用いるべきである．例を**付録 1** に示す．

このようにして PAROTS を利用する目的は，既知の変換パターンを自動的に実行することである．それは，研究的立場からは変換パターンの有効性確認であり，実用的立場からは新しい Prolog プログラムの効率化の手始めである．とくに後者の目的からは，効率化が行われる限り相異なる変換戦略を繰り返して適用するといった使い方がされるであろう．

よって，ユーザの便宜を図ることがあっても絶対に余分な手間を掛けさせないために，効率化に至らない変換戦略によって適用範囲外の述語が不必要に書き換えられて出力されないよう注意する必要がある．

5.2 探索に goal の実行順序も加えた変換戦略

5.1 節で紹介した変換は，探索として新述語と unfold とを用いていた．このほかの探索としては，文献 10) で利用されている goal の並べ変えがある．

その例が次の

append を用いたクイックソートの変換である.

```
qsort ([ ], [ ]).
qsort ([CAR|CDR], ANS)←
    partition (CAR, CDR, A, B), qsort (A, SA),
    qsort (B, SB), append (SA, [CAR|SB],
    ANS).
mode qsort(+, −) mode partition (+, +, −, −)
mode append (+, +, −)                    (5.7)
```

(結果)

```
qsort ([ ], [ ]).
qsort ([CAR|CDR], ANS)←
    partition (CAR, CDR, A, B), qsort (B, SB),
    qa (A, [CAR|SB], ANS).
qa ([ ], B, B).
qa ([CAR|CDR], B, ANS)←
    partition (CAR, CDR, A, C), qa (C, B, L),
    qa (A, [CAR|L], ANS).                (5.8)
```

この変換は Tarnlund の研究[11] においては data structure mapping を用いて行われていたが, TRANS 上では次のコマンド列になる.

```
コマンド列=UG, MFC                    , FRD
  新述語:  , U, AU, AE, AU, UG, AF, UG, AF
```

上の例のように goal を並べ変えた結果として, 等価則が適用可能になり, 再帰定義に変換される場合がある. しかしながら, 実行順序の評価が効率化に結びつくことから分かるように, goal を無作為に並べ変えてもかえって効率を悪化させたプログラムを生じないとも限らない. さらに, それの変換を行っても無意味であるばかりか, 探索空間をいたずらに広げることになる. これを PAROTS 上で解決するには,

```
(IF HAVE-MODE) (THEN
  (TRY (UG)
    (IF AT) (THEN CONTINUE)
    …効率化への探索…))                  (5.9)
```

の形式で探索を記述すればよい. ただし, (5.9) において, HAVE-MODE は注目している述語内では mode 宣言が既知であるとき真となり, また, CONTINUE はいちばん内側の TRY においてただちに選択肢を次に進めるものとする. このプログラムでは, AT を実行順序の評価による効率化ではなく, goal を並べ変えて作った body の妥当性判定に用いている. つまり, UG を施すことにより AT が真となるような述語定義ができた場合, それは上述の探索しても効率化の望めない状態なのである. 結局, (5.9) によって, goal 一つを移動することで生成されるすべての有意義な body について探索できることになる.

以上の制御の元で, 先のクイックソートのコマンド列が記述できる. 次のプログラムを, (5.9) の …効率化への探索… に代入することで実現される.

```
(TRY (MFC)
  (TRY (U) AU AE
    (IF AU) (THEN
      (TRY (UG)
        (IF AT) (THEN CONTINUE)
        (IF AF) (THEN
          (TRY (UG)
            (IF AT) (THEN CONTINUE)
            (IF AF)
              (THEN RP FRD END)))))))
                                         (5.10)
```

この方法で,「append の第一引数が他のゴールとの共有変数になっていたら, そのゴールと append を組み合わせた新しい述語を定義せよ」という基本原理で作られた玉木氏らの Append オプティマイザ[12] と同じ程度のことができる. 文献 12) で紹介されている変換例を自動変換できる PAROTS プログラムを**付録 2** に示す.

6. むすび

unfold, fold を中心としたプログラム変換エディタ TRANS を開発し, プログラムを短いコマンド列で変換できるようになった. その上に変換戦略を表現する言語によって自動変換を行うインタプリタ, PAROTS の構築を試みた. これを利用すれば, 各種変換パターンの知識を PAROTS 言語で記述することによって, 各知識の自動変換規則として能力を試すことができる.

我々の開発した TRANS は, 基本的な変換能力については十分であるが, さらに使い心地を良くするためには, マルチウィンドウの導入により変換前後の述語を同時に見れるようにするなどが望まれる.

今後は, PAROTS 言語による各種変換パターンの記述や, PAROTS 自身の拡張を通じて, より広い領域のプログラムを自動変換できるシステムを目指す.

謝辞 本研究は文部省科学研究費 (特定研究—多次元知識情報処理) に予算のバックグランドを得てい

る.

さらに，変換に関する貴重な資料を提供して下さった電子技術総合研究所の佐藤泰介氏に感謝いたします.

参 考 文 献

1) Hogger, C. J.: Derivation of Logic Programs, *J. ACM*, Vol. 28, No. 2, pp. 372-392 (1981).
2) Bible, W.: *Syntax-Directed, Semantic-Supported Program Synthesis, Artificial Intelligence 14*, North-Holland (1980).
3) Burstall, R. M. and Darlington, J.: A Transformation System for Developing Recursive Programs, *J. ACM*, Vol. 24, No. 1, pp. 44-67 (1977).
4) Darlington, J.: An Experimental Program Transformation and Synthesis System, *Artif. Intell.*, Vol. 16, No. 1, pp. 1-46 (1981).
5) 佐藤，玉木：Prolog に於けるプログラム変換，ICOT, The Logic Programming Conference '83, 6.1 (1983).
6) H. Tamaki and T. Sato: Unfold/Fold Transformation of Logic Programs, 2nd Logic Programming Conference (1984).
7) Sato, T.: Transformational Logic Program Synthesis, Proc. of FGCS '84 (1984).
8) 中村，中川：Prolog プログラムの等価変換エディタ，第29回情処学会全国大会，4p-1 (1984).
9) 松方他：Prolog のプログラム変換について，情報処理学会第27回全国大会，1N-1 (1983).
10) 玉木，佐藤：Prolog における Append プログラミング，情報処理学会第28回全国大会，4H-8 (1984).
11) Tarnlund, S.-A. and Hansson, A.: *Program Transformation by Data Structure Mapping, LOGIC PROGRAMMING*, Academic-Press (1982).
12) 玉木，佐藤：Append オプティマイザについて，ICOT, The Logic Programming Conference '84, 9-1 (1984).

付録 1 初歩的な自動変換システム記述例

```
(IF AU) (THEN END)
(IF AT) (THEN END)
(IF SINGLE-CLAUSE) (THEN
  (TRY (U) AU
    (IF AF) (THEN END)
    (TRY (MFC)
      (TRY (U) AU
        (IF AT) (THEN
                (IF AF) (THEN RP FRD END))))))
  (ELSE
    (TRY (MFC)
      (TRY (U) AU SB AE
      (IF AU) (THEN
        (IF AF) (THEN RP FRD END))
      B
      (IF AM) (THEN
        (IF AF) (THEN RP FRD END)))))
```

付録 2 PAROTS 言語による文献 12) の変換例記述

```
(IF SINGLE-CLAUSE) (THEN
  (TRY (U) AU (IF AF) (THEN END)) END)
(TRY (MFC)
  (TRY (U)
    (IF AU) (THEN SB AE
      (IF AU) (THEN
      (IF AF) (THEN RP FRD END))
    B
    (IF AF) (THEN AE
      (IF AF) (THEN RP FRD END)))))
(IF HAVE-MODE) (THEN
  (TRY (UG)
    (IF AT) (THEN CONTINUE)
    (TRY (MFC)
      (TRY (U) AU
        (IF AF) (THEN OB AE
          (IF AU) (THEN
            (TRY (UG)
              (IF AT) (THEN CONTINUE)
                (IF AF) (THEN
                  (TRY (UG)
                    (IF AT)
                      (THEN CONTINUE)
                    (IF AF) (THEN
                      RP FRD END))))))
          (ELSE
            (TRY (MFC)
              (TRY (U) AU
                (IF AF) (THEN OB
                  (IF AE) (THEN
                    (TRY (UG)
                      (IF AT) (THEN
```

```
  CONTINUE)
(IF AF) (THEN
  (TRY (UG)
    (IF AT) (THEN
      CONTINUE)
    (IF AF) (THEN
        RP FRD RP FRD END))))))))))))
```

(昭和59年12月25日受付)
(昭和60年 2 月21日採録)