Vol. 21 No. 6　　情報処理学会論文誌　　Nov. 1980

名前置換えプログラム†

和田英一††　久保田稔†††

プログラムで使用する名前 (identifier) の選び方は，プログラムのよみやすさ，わかりやすさに大きく影響する．一旦完成したプログラムでも，質を改良するために，名前をさらに適切なもので置き換えたくなることがある．名前の置換えは，テキストエディタを使用しても原理的には可能だが，実用的なプログラムの名前の置換えを人手で制御するのは決して実際的ではない．これを代行するプログラムは，プログラム言語の構文，特に名前の参照の規則，名前の有効範囲の規則をわきまえ，名前置換えの結果，プログラムの意味のかわることがないようにしなければいけない．本論文はそのような名前置換えプログラムについて，必要性，機能，処理法を考察し，実際にインプリメントした過程をのべる．

1. はじめに

信頼性のあるプログラム (reliable program) を作ろうという動きが盛んである．それには

i) 充分にテストを行う

ii) 正しいことを検証する

などの方法が考えられるが，i) はダイクストラ (Dijkstra) がいうように[1]「虫のいることは証明できるが，虫のいないことは証明できない.」ii) はラッカム (Lackham) がいうように[2]「現状では，検証することは，きわめて困難にみえる.」そうだとすると信頼性のあるプログラムを作るには，もうひとつ

iii) 計算機の助けを借りて人力で調べる

を加えなければならない．もちろんi) やii) もある意味では計算機の助けを借りた人力だが，iii) はプログラマがプログラムを解読し，その機能を理解し，プログラムを修正して信頼性をもたせるのであり，その解読と理解を容易にすべく，計算機の助けを借りるというのである．

プログラムの解読と理解を容易にするために従来奨励されてきたのは「注釈をつける」ことであった．なるほど注釈つけは奨励されてはきたが，プログラムを書く立場からいえば，これはつまらぬ仕事である．自分にはわかりすぎるくらいよくわかっていることを，ただ単にもう一度書き直さなければならないし，だからといってあまり考えずに注釈をつければ，他人にはやはりわかり難く文句をいわれる．上手に注釈をつけるのは，時にはプログラムを書くより大変である．プログラムを書きながら注釈をつけるのはプログラムを書き直すと無駄になるので，一通り完成してから注釈をつける流儀の人もいる．注釈挿入用エディタも作られている．いずれにしろ注釈をつけたところで，プログラムを作り直す必要が生じると，プログラム部分は当然書き直すが，注釈をつけ直すのにまた相当の努力がいるので，新版は注釈をはぶくか，書き直しを諦めるか，よい結果は得られない．

注釈はプログラムの読み手にも問題をもっている．はたしてこの注釈はプログラムと一致しているだろうか，プログラムの書き直しのとき注釈もつけ直されたであろうか．また注釈が多すぎて，どこが肝心のプログラムかわからないのも困る．書いてはあるが意味の全然通じないのもある．「プログラム書法[3]」の翻訳では注釈部分の訳が難しかったそうである．

プログラムを解読しやすく，理解しやすくするのに注釈以外の方法はないかといえば，それはある．プログラムで使う名前の選び方とプログラムのわりつけ方がそれである．この点に関して数年前にトロント大学で実験が行われた[4]．それは同一のプログラムに対して

i) 注釈　ii) 構造を示すわりつけ　iii) 意味をもつ名前

の要因の各々を，採用する，採用しない，で $2^3=8$ 通りの版を用意し，大勢の学生にそのいずれかを渡して解読と理解の難易の比較を試みた．その結果，各要因は独立には採用しないより採用した方が，解読と理解が容易であった．しかし構造を示すわりつけや意味をもつ名前と注釈に関しては，注釈はむしろ邪魔になる

† Identifier Replacing Program by EIITI WADA and MINORU KUBOTA (Department of Mathematical Engineering, University of Tokyo).

†† 東京大学工学部計数工学科

††† 現在は日本電信電話公社武蔵野電気通信研究所

ことがわかったという．この結果はますます筆者らをして注釈に対する不信の念を募らしめ，注釈なしで，構造を示すわりつけと意味をもつ名前だけを採用するのが，プログラムを解読，理解しやすくし，さらにプログラムに信頼性をもたせるに至ると確信させるようになった．

その後，カーニハンとプローガーの「プログラム書法」が出版され，つづいて翻訳された．この本でも嬉しいのは規則の中に

ⅰ）プログラムの割付けをくふうして，理解しやすくしよう．(162 ページ)

ⅱ）意味のある変数名を使おう．意味のある名札を使おう．(160, 161 ページ)

のあることであった．そして注釈については，その使用を控えさせるような規則があった．(169 ページ)

以上のような次第で，筆者らは信頼性のあるプログラムを作る道具として

ⅰ）わりつけを行うためのプリティプリンタ

ⅱ）一旦完成したプログラム中の名前を別の名前で置き換えるためのプログラム

が必要不可欠だと考え[5]，それらの開発を心がけてきた．このうちプリティプリンタについてはよく知られているし，最近多少文献も発表されるようになったし[6]，筆者らも別のところで意見をのべたりしているので[7]，今回は以下の章において，名前置換えプログラム，それも特にプログラム言語 Pascal に対するものについてのべることにする．

2. 名前置換えプログラムの必要性

JIS の Algol の規定をみると，名前の意味の項(4.5.1)に「その選び方は任意とする」と書いてあるが，その選び方がプログラムの解読，理解しやすさに影響すること甚大なのは前述のとおりである．ところでプログラムを書くに際しては，名前は一応前もって決めてから書きはじめるのであるが，

ⅰ）できあがってみると名前の意味が初め考えていたのとちがってしまった．

ⅱ）名前の構成規則が統一的でないことに気がついた．

ⅲ）入れ子の関係を変えようとしたら名前の衝突がおきそうになった．

ⅳ）プログラムの入力の手間のせいで短い名前にしておいたのを戻したくなった．

そのほかいろいろな理由であとから名前を置き換えたくなるのが現実である．しかし一通り動くのだから，置換えが大変面倒だったらそのまま完成とするわけだが，もし計算機の助けを借りて，比較的簡単に置き換えられるのなら，机帳面な人なら誰しも気にいった名前に置き換えてから完成としたくなるであろう．

名前の置換えにテキストエディタを使うのはどうかといえば，それは不可能ではないかもしれない．置き換えるべき名前をいちいち見つけだしては文字列置換えのコマンドで置き換えるのである．しかし置換えの箇所が多くなるとこれは実用的でなく，誤りのもとである．もしこのテキストエディタに，指定した範囲内の指定した文字列をすべて置き換えるコマンドがあったとしても，通常のテキストエディタは，ここで考えているような名前の置換えには適さない．たとえば，変数 x を y で置き換えようとすると，通常のテキストエディタでは，指定した範囲内のすべての x を，それが変数であれ，文字列定数内や注釈内であれ，綴りの記号内であれ，ひとつ残らず y で置き換えてしまう．仮に名前 x だけを見つけるテキストエディタがあったとしたら，問題は解決するかというと，決してそうではない．Pascal のような有効範囲の概念を有するプログラム言語では，あるレベルの名前 x を y で置き換える場合，その内部で局所的に x が宣言されていれば，その有効範囲内では置換えを一時中断しなければならない．テキストエディタでは，有効範囲を調べながら置換えの採否を決めるのは不可能であろう．そこでどうしても Pascal なら Pascal の文法，それも特に名前の有効範囲の規則を知っていて，それに基づいて名前の置換えをするプログラムが必要になる．

3. 名前置換えプログラムの機能

このプログラムは次のように働く．まず初めに Pascal の（構文的に正しい）プログラムがあるとする（これを初めのプログラムという．）その中で置き換えたい名前（複数個あってよい）については，それぞれの定義の場所 (defining occurence) の直前に，両端を記号 # で囲んだ新しい名前をテキストエディタを使って挿入する．それからそのように修正されたプログラムをこの名前置換えプログラムで処理すると，定義の場所と使用の場所 (applied occurence) にあった初めの名前がすべて新しい名前で置き換えられたプログラム（新しいプログラムという）が得られる．

定義の場所とは，Pascal の文法でいうと

ⅰ）定数定義　定数名＝定数　の定数名

ii) 数えあげ型（スカラ型）
（定数名 {, 定数名}）の定数名
iii) 型定義　型名＝型　の型名
iv) レコード部　フィールド名 {, フィールド名}：型　のフィールド名
v) 変数宣言　変数名 {, 変数名}：型　の変数名
vi) 手続き頭部　**procedure** 手続き名 [(仮パラメタ部 {; 仮パラメタ部})]; の手続き名
vii) パラメタ組
変数名 {, 変数名}：型名　の変数名
viii) 関数パラメタ組
関数名 {, 関数名}：結果型　の関数名
ix) 仮パラメタ部 **procedure** 手続き名 {, 手続き名} の手続き名
x) 関数頭部 **function** 関数名 [(仮パラメタ部 {; 仮パラメタ部})]：結果型；の関数名

を指す.（標準の定数名，型名，変数名，手続き名，関数名は定義の場所がプログラム内にないので置換えはできない．なお標準のフィールド名はない.）

次に簡単な例を示す.

図1で（a）は初めのプログラムである．このプログラムの変数名xをyで置き換えたいとする．定義の場所は変数宣言の変数名，つまり（2行目）**var** x: *real* の x である．そこでテキストエディタを使って(b)のようにその x の直前に # y # を挿入する．名前置換えプログラムの処理の結果が（c）で，変数宣言のxはもとより，使用の場所のx，つまり（3行目）x:=3.14 と *writeln* (x) の x も y で置き換えられた新しいプログラムが得られた.

図2は入れ子のある場合で，(a)はすでにテキストエディタで新しい名前を挿入した状態を示す．このプログラムは外側のブロック（2行目）で変数xが宣言されており，内側のブロック（3行目）でも変数仮パラメタxが宣言されている．使用の場所のxは内側の宣言に対するものが(4行目) x:=3.14 のそれ，そして外側の宣言に対するものは（5行目）$p(x)$ と *writeln* (x) の x である．いまは外側の x だけを y で置き換えようとしている．その結果(b)のように外側のxに対するものだけがyで置き換えられ，内側のxに対するものはそのまま残っているようなプログラムが得られた.

Pascalのようにレコード型が使える言語では，レコードの定義やレコード変数，**with** 文の中でフィールド名に対する有効範囲が開かれるので，名前置換えプログラムはこのことも考慮しなければならない．**図3**はレコードのある場合で，(a)ではレコード型の型名 $r1$ の型定義のレコード部のふたつのフィールド名(2行目) $f1$, $f2$ をそれぞれ $g1$, $g2$で置き換えようとしている．変数宣言（3行目）の$f1$, $f2$ はレコード定義の有効範囲の外だから名前の衝突はおきない．名前置換えプログラムの処理の結果，(b)のように（2行目）レコード型の定義の $f1$, $f2$と，(4行目）レコード変数の中のフィールド名 $f1$, $f2$と，(5行目）**with** 文の中のフィールド名の $f1$, $f2$ がすべて $g1$, $g2$で置き換えられ，レコード型変数名の

```
program example 1 (output);
  var x: real;
  begin x:=3.14; writln (x) end.
```
(a)
```
program example 1 (output);
  var # y # x: real;
  begin x:=3.14; writeln (x) end.
```
(b)
```
program example 1 (output);
  var y: real;
  begin y:=3.14; writeln (y) end.
```
(c)

図 1　名前置換えの例1

Fig. 1　Example of identifier replacement 1.

```
program example 2 (output);
  var # y # x: real;
  procedure p (var x: real);
    begin x:=3.14 end;
  begin p(x); writeln (x) end.
```
(a)
```
program example 2 (output);
  var y: real;
  procedure p (var x: real);
    begin x:=3.14 end;
  begin p(y); writeln (y) end.
```
(b)

図 2　名前置換えの例2

Fig. 2　Example of identifier replacement 2.

```
program example 3 (output);
  type r1=record # g1 # f1, # g2 # f2: real end;
  var f1, f2: r1;
  begin f1. f1:=3.14; f1. f2:=2.71;
  with f1 do writeln (f1, f2) end.
```
(a)
```
program example 3 (output);
  type r1=record g1, g2: real end;
  var f1, f2: r1;
  begin f1. g1:=3.14; f1. g2:=2.71;
  with f1 do writeln (g1, g2) end.
```
(b)

図 3　名前置換えの例3

Fig. 3　Example of identifier replacement 3.

(4行目と5行目) $f1$ は $f1$ のまま残っているようなプログラムが得られた.

図4は入れ子のブロックの内でも外でも名前を置き換える場合である. (a)のように外側のブロックに定義の場所をもつ (2行目) x を y で, 内側のブロックに定義の場所をもつ (4行目) y を x で置き換えようとしている. x と y の使用の場所はともに (5行目) 内側のブロックの実行文部にある. その結果 (b) のように x と y が交換されたプログラムが得られた.

```
program example 4 (output);
  var # y # x: real;
  procedure p;
    var # x # y: real;
    begin y:=3.14; x:=2.71 end;
  begin writeln (x) end.
```

(a)

```
program example 4 (output);
  var y: real;
  procedure p;
    var x: real;
    begin x:=3.14; y:=2.71 end;
  begin writeln (y) end.
```

(b)

図4 名前置換えの例4

Fig. 4 Example identifier replacement 4.

4. 名前置換えプログラムの検出すべきエラー

名前置換えプログラムに入力されるプログラムは, 前述のように一応完成した (少なくとも構文的には正しい) ものとする. 折角完成したプログラムが, 名前置換えの結果, 構文的に正しくなくなったり, 初めのプログラムと違う意味のプログラムになったりしては困るので, その点に注意して名前を置き換えなければならない. 構文的に正しくなくなるのは, 得られたプログラムをコンパイルすれば検出できるけれども, 意味を変える方はコンパイラでも検出できない場合があるから, 名前置換えプログラムで面倒をみる必要がある.

名前の置換えで生じそうなエラーは

i) 新しい名前の構文エラー

ii) 新しい名前の挿入位置エラー

iii) 名前の二重定義

iv) 対応関係のくずれ

である. i) はたとえば *begin* を新しい名前にしようというような, 名前の構文に合わないものを # で囲むものである. ii) は定義の場所でないところに新しい名前を挿入するものである. これらは名前置換えプログラムの誤用である. iii) は名前を置き換えた結果, 同一ブロックの先頭やレコード型の定義の内部で名前の衝突が新しくおきることである. たとえば図3のレコード型の定義で

(例1) **record** # $f2$ # $f1$, $f2$: *real* **end**;

とすると第1フィールドの新しい名前と第2フィールドの初めの名前と衝突する.

(例2) **record** # $g1$ # $f1$, # $g1$ # $f2$: *real* **end**;

とすると第1フィールドと第2フィールドの新しい名前が衝突する.

(例3) **record** # $f2$ # $f1$, # $f1$ # $f2$: *real* **end**;

とすると名前が交換されるだけで衝突しない. 衝突がすなわち二重定義であり, 名前置換えプログラムは二重定義エラーを表示する (double define error).

iv) の対応関係のくずれはその結果プログラムの意味を変えることになる. 図4の例では外側と内側のブロックで同時に名前を置き換えたが, これがどちらか一方だけの置換え指定だったとしよう.

(例4) 外側の x は y で置き換え, 内側の y はそのままとする.

この場合は内側のブロックのふたつの代入文の変数はともに y になり, あとの代入文の変数の対応が外側から内側のブロックで宣言されたものへ変る.

(例5) 外側の x はそのままとし, 内側の y を x で置き換える.

この場合は内側のブロックのふたつの代入文の変数はともに x になり, まえの代入文の変数の対応が外側から内側のブロックで宣言されたものへ変る. いずれの場合も初めのプログラムと新しいプログラムは違ったものになる. 図4のように両変数を同時に置き換える場合は対応関係は変らない. 対応関係が変った場合, 名前置換えプログラムは対応関係エラーを表示する (misidentification error).

5. 名前置換えプログラムの処理

前々章の機能をもち, 前章のエラーを表示するためには, 名前置換えプログラムは次のように処理すればよい. ブロック構造およびレコード型の有効範囲の規則に従って名前の対応関係をとるには, スタック状に構成された名前表 (*symbol table*) を使うことがよく知られている. 名前置換えプログラムも基本的にはこの手法を使うことになる. 一般のコンパイラでは, 定義の場所に現れた名前に対しては名前の登録 (*enter*

id) を行い，使用の場所に現れた名前に対しては，名前の探索 (*search id*) を行う．ブロックに入る (*enter block*) たびに，表にブロックの境界を明示し，その内部で定義された名前は，登録の手続きによりこの境界の上に積む．ブロックから出る (*leave block*) ときは，このブロックに入るときにつけた境界の上に積まれた部分を除去する．探索に際しては，名前を，最後に積んだ上の方から最初に積んだ下の方へ順々に，初めて一致するものがでてくるまで探す．レコード型の定義があれば，フィールド名を積んだ，ブロックに相当する表の部分を用意し，このレコード型の変数が現れると，この型に対応する表の部分を名前表の一番上に一時的に積んで，フィールド名を探索する．レコード型変数なら，一番上の部分だけを探索するが，**with** 文の場合は，**with** 文を構成する文の処理が終るまで，**with** 文のレコード変数に対応する表の部分を積んだままにしておき，普通の名前のようにずっと下の方まで探索する．

ところで名前置換えプログラムの場合は，この名前表の扱いを次のように変更する．名前表には名前の欄を初めの名前 (*oldname*) のためのものと，新しい名前 (*newname*) のためのものとふたつ用意する．ブロックの出入り，レコード型の定義での表の扱いはコンパイラの場合と同様であるが，

i) 名前の登録では，初めの名前と新しい名前を値パラメタとしてもってゆく．これは記号読み込みルーチン (*insymbol*) で，名前置換え指定があれば，初めの名前と新しい名前をそのそれぞれに，指定がないと初めの名前を両方に入れるようにしている．さて両方の名前をそれぞれに登録したあと，新しい名前について，このブロック，レコード内で二重定義になっていないかどうかを調べる．

ii) 名前の探索では，初めの名前を値パラメタでもってゆき，新しい名前を変数パラメタでとりだす．まず初めの名前で名前表を探索する．初めて一致するものがでてきたら，それに対応する新しい名前を読みだす．次に新しい名前で名前表を探索する．初めて一致するものがでてきたら，それが先程の場所かどうか，つまり対応関係がくずれていないかどうかを調べる．

以上のようなプログラムを用意しておき，名前以外の記号はそのまま出力し，名前に関しては常に新しい名前を出力するようにすれば名前置換えプログラムは完成である．**図5**はこの処理に関係した部分の Pascal によるプログラム，**図6**はそれに使うデータの説明である．表 *disp* は *display* のつもりで，各レベルのブロックの名前の部分の先頭の位置を記憶している．表 *symtab* は *symbol table* のつもりで，前述のように *oldname* と *newname* からなるレコードの配列である．それぞれの表は次に使用可能な位置を示すポインタ，*dispp* と *symtabp* をもっている．この両ポインタはそれぞれ1に初期設定してある．表のオーバフローは説明を簡単にするため，ないものとしている．

insymbol も普通の *insymbol* を次のように修正す

```
var disp: array [1..dispmax] of integer;
    symtab: array [1..symtabmax] of
        record oldname, newname: alfa end;
    dispp, symtabp: integer;
procedure enterblock;
  begin disp [dispp]:=symtabp;
    dispp:=dispp+1 end;
procedure leaveblock;
  begin dispp:=dispp-1;
    symtabp:=disp [dispp] end;
procedure enterid (oldname, newname: alfa);
  var symtabp1: integer;
  begin symtab [symtabp]. oldname:=oldname;
    symtab [symtabp]. newname:=newname;
    symtabp1:=disp [dispp-1];
    while symtab [symtabp1]. newname <> newname
      do symtabp1:=symtabp1+1;
    if symtabp1 <> symtabp then
        double-define-error;
    symtabp:=symtabp+1 end;
procedure searchid (oldname: alfa; var newname: alfa);
  var symtabp1, symtabp2: integer;
  begin symtabp1:=symtabp-1;
    symtabp2:=symtabp1;
    while symtab [symtabp1]. oldname <> oldname
      do symtabp1:=symtabp1-1;
      newname:=symtab [symtabp1]. newname;
      while symtab [symtabp2]. newname <> newname
      do symtabp2:=symtabp2-1;
    if symtabp1 <> symtabp2 then
      mis identification-error end;
```

図 5 名前置換えプログラムの主要部分

Fig. 5 Main part of the identifier replacing program.

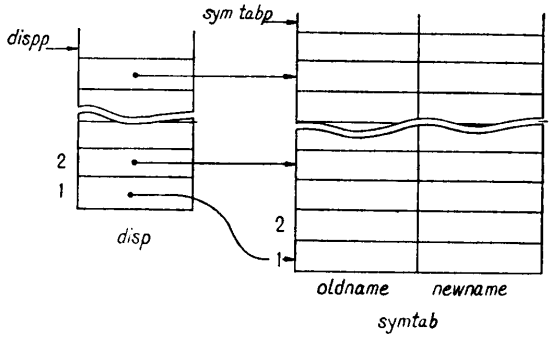

図 6 *disp* と *symtab*

Fig. 6 *disp* and *symtab*.

る程度で使える．まず空白を読みとばす．次の文字が英字であれば，名前の構文の部分を読み込む．綴りの記号でなければ置換え指定のない名前であるから，記号の種類を‘名前’とし，*oldname* にも *newname* にもその名前の綴りを入れて帰る．次の文字が＃であれば，置換え指定の新しい名前がくる筈だから，つづいて名前の構文の部分を読み込み，綴りの記号でないことを確かめた上でその綴りを *newname* に入れる．さらに次の文字が＃であることを確かめ，それにつづく名前の構文を読み込み，その綴りを *oldname* に入れ，記号の種類を‘名前’として帰る．

6. 実際のインプリメンテーション

前述の処理を実際に行うには，構文を解析し，必要な情報を名前表に追加して覚えておき，あとでそれも利用しながら名前表を扱わなければならない．これらは Pascal のコンパイラでもやっていることであるから，名前置換えプログラムをインプリメントするには，コンパイラの部品を前述のように修正し利用するのがよい．すなわち名前表に *newname* の欄を追加する．*enterid*, *searchid*, *insymbol* を前章のように修正する．*insymbol* は名前以外のものは，読み込んだものの出力もする．*enterid*, *searchid* は *newname* の出力もする．もちろん標準の名前の登録もこの *enterid* を使うように修正しなければならない．

Trunk Pascal から作ったコンパイラは名前表が二進木になっているので，前章のとおりにはインプリメントできないが，原理的には同じことを行えばよい．とにかく Pascal のコンパイラの Pascal で書いたものをもっているので，それの修正からはじめて，約3日でインプリメントすることができた．オブジェクトコードを発生するところを取り除いてないので，現在の版では名前を置き換える処理のたびに副作用でコンパイルもできるが，このプログラムの実績をみるまではスピードアップには着手しないでいる．

7. 補足すべき事項

以上で名前置換えのプログラムの一応の説明を終えるが，なお多少補足しておくことがある．まずこのプログラムではできないことについて．

i）名札 (label) の置換えは行わない．Fortran のように文番号の沢山でてくるプログラム言語だと，文番号を探すのが便利なように定義場所の順に大きくなるように文番号を置き換えたりもするようだが，Pascal では名札ないし **goto** 文はあまり使用しないので，このプログラムでは面倒をみず，もし置き換えたいなら，それは人力でやってもらうことにする．

ii）8字より長い新しい名前での置換えは行わない．Pascal の処理系はほとんどが先頭の8字しか識別しないが，新しい名前は，この8字がほかと異なればどんなに長い名前でも構わない筈である．また長い名前も使えた方が，解読，理解しやすい名前を選ぶのに便利かもしれない．しかし今はとりあえず本質的な部分の実験を行うのが目的であったので，新しい名前は長く書いても8字しか出力されないようになっている．将来は長い名前も扱えるようにしたいと考えている．それには名前表の *newname* は8字の *alfa* 型とするが，新しい名前の8字より長い分はどこか別の場所に記憶しておき，*newname* を読みだすと，長い分もつづけて読みだせればよい．このための修正はそんなに大変ではないと思っている．

次に定義の場所より前に使用の場所のくる場合について．Pascal ではポインタ型とレコード型の参照のし合いのとき，お互いに呼び合う関数と手続きに対する *forward* 宣言のときは定義の場所以前に使用することができる．(*forward* の場合は定義の場所が2回あるとみるべきか.) このふたつの場合では，名前表には定義の場所ではなくて，1回目に現われたときに登録しなければならず，もし名前の置換えを指定するなら，そのときに指定しなければならない．このことは，名前置換えプログラムの機能の説明で，置換えの指定を定義の場所でなく，1回目にでてきたときに行うようにするだけでよい．プログラムの方はコンパイラを修正したものであれば，特別の手当てをしなくても，このままでうまくゆくようである．**図7**は *forward* 宣言のある場合の置換え，**図8**はポインタ型とレコード型の参照のし合いのある場合のそれである．

```
procedure # s # q; forward;
procedure # r # p;
  begin ··· q ··· end;
procedure q;
  begin ··· p ··· end;
```

(a)

```
procedure s; forward;
procedure r;
  begin ··· s ··· end;
procedure s;
  begin ··· r ··· end;
```

(b)

図 7 *forward* 宣言のある場合
Fig. 7 *forward* declaration.

```
type ＊q＊p=↑＊s＊r;
   r=record d: t; n: p end;
```
(a)

```
type q=↑s;
   s=record d: t; n: q end;
```
(b)

図 8 ポインタ型とレコード型の相互参照の場合

Fig. 8 Cross reference between pointer and record types.

8. おわりに

名前置換えプログラムの必要性，Pascal用に設計したものの機能，その実現法についてのべた．特に既存の Pascal コンパイラを改造すれば，ごく短時間で実現できることもわかった．新しいプログラムの行が初めのものより長くなることもありうるが，それは出力をプリティプリンタにかけることで簡単に解決する．ソフトウェア危機をのりきるためには，多くの有効な道具を用意しなければならないが，それもこの程度の手間で用意できれば，大いにやりがいがあると思われる．

追記，本稿を書き終ってから最近似たような趣旨のプログラム ID 2 ID (id to id と読む)が発表された[8]が，これは有効範囲も考慮せず，ただ新旧両名前の対のファイルを初めのプログラムのファイルとともに入力して，名前の置き換えられたプログラムを得るもので，さほど便利なものとはいえないように思われる．

参考文献

1) ダイクストラ他 (野下他訳)：構造化プログラミング，p. 249，サイエンス社 (1975)．
2) Luckham, D. C.: Program Verification and Verification Oriented Programming, Information Processing 77, pp. 783-793, North Holland (1977).
3) カーニハン，プローガー (木村 泉訳)：プログラム書法，p. 198，共立出版 (1976)．
4) Weissman, L.: Psychological Complexity of Computer Programs: An Initial Experiment, Technical Report CSRG-26, p. 82, Computer Systems Research Group, Univ. of Toronto (1973).
5) Wada, E.: Abstracts in Software Engineering, Software Engineering Notes, Vol. 1, No. 2 (1976).
6) Purdon, P. W.: Automatic Program Indentation, BIT, Vol. 18, No. 2, pp. 211-218 (1978).
7) 鶴田陽和，武市正人，和田英一：Syntax Directed Pretty Printer，第17回プログラミング・シンポジウム報告集，pp. 155-166，情報処理学会プログラミング・シンポジウム委員会(1976)．
8) Mickel, A.: Recording a Pascal Program Using ID 2 ID, Pascal News # 15, pp. 31-34 (1979).

(昭和55年4月11日受付)
(昭和55年6月19日採録)