22. Pコードを中間コードとするPascal処理系のimplementation

東京大学 工学部 計数工学科 金田 泰
Yasusi Kanada

従来の，Trunk からつくった1パスのPascalコンパイラには，Trunkが改訂されたときの再移植がむずかしく，また最適化がむずかしいという欠点があった。

この問題を解決するためには，適当な中間コードをえらんで，コンパイラを多パスにすればよいが，そのようなコンパイラをPascal-Pを拡張してつくった。この処理系（の機械依存の部分）はdisplay をもちいる方式をとったが，これは，display の内容をこわした手続きが，その実行終了時にその修復をする方式であり，非局所変数のないときには，わずかの最適化をすることによって，もともと非局所変数をもたない言語のばあいとおなじ効率を達成できる。

1. はじめに

あらたなPascalコンパイラをつくるにあたって目標としたことは，つぎのような問題点を解決することであった。

従来の，機械語を出力するPascalコンパイラ（たとえば〔Ammann77〕,〔疋田76〕）の大部分は，1パスであるため，針穴的 (peep-hole) でない最適化をおこなうことは困難であり，また，針穴的最適化もコンパイラのよみやすさをおとし，虫の原因となりやすかった。

また，いわゆるTrunk からつくったコンパイラは，Trunk に虫があったばあい，あるいは標準の言語仕様が変更されたばあいに，それに対応する（すなわち再移植〔3節参照〕する）のがはじめての移植とおなじくらい困難である。

一方，Pascal-P〔NOri78〕は インタプリタ・コードを出力するので移植性は高いが，これはPascalがつかえない機械のうえにはじめて移植をおこなうときのものであり，インタプリタで実行するため実行速度がおそいので，実用にはならなかった。（また，Pascal-P はPascalの不充分なサブセットであることも，実用にならないひとつの理由である）。

この，効率，最適化，および再移植の困難という問題を解決するかんたんな方法はつぎのとおりである。コンパイラは2パスにする。パス1は機械およびimplementationに独立にし，すべての処理系にたいして同一のものを用いる。パス2は各機械にひとつずつ作り，これは機械語を出力する（図1A）。中間コード体系は，パス1に多少の変更があっても影響をうけないように設計する。最適化が必要なときは，この2つのパスのあいだに 独立のパスとしてoptimizerを走らせる（図1B）。それもさらに，機械独立な最適化，機械依存の最適化という2つのパスにわけることができる（図1C）。パス1の出力は機械独立な中間コードであるが，機械依存のoptimizerの出力は，その中間コードを機械依存に拡張したものである。（並行プロセス またはコルティンと分割コンパイルができる言語の処理系が使えれば，多パスにする必要はないが，今のところそのようなものはつかえない）。

あたらしい処理系をPascal-EPとよぶ。Pascal-EPは，この方法にもとづいて，中間コードとしてはPascal-Pの中間コードであるPコードを多少変更したものをもちいてつくったが，それは，Pコンパイラがすでに上のパス1の条件をかなりみたしていて，わずかの変更だけで目的にかなうであろうとかんがえたからであり，上の目標にとってそれがもっともよいからというわけではない。

第21回プログラミング
シンポジウム 1980.1

2節では，再移植性についてよりくわしくのべ，3節では，Pコードと，おもに実行効率をよくするためにそれにくわえた変更についてのべる。そして，Pascal-P をもとにしたために，最適化の問題がどのようになったかについて，4節でのべる。また，パス2はMelcom Cosmo 700/900 のためにつくったが，このimplemcntationで採用したdisplay, static link などの実行時環境について5節でのべ，6節では他のこまかい問題についてのべる。

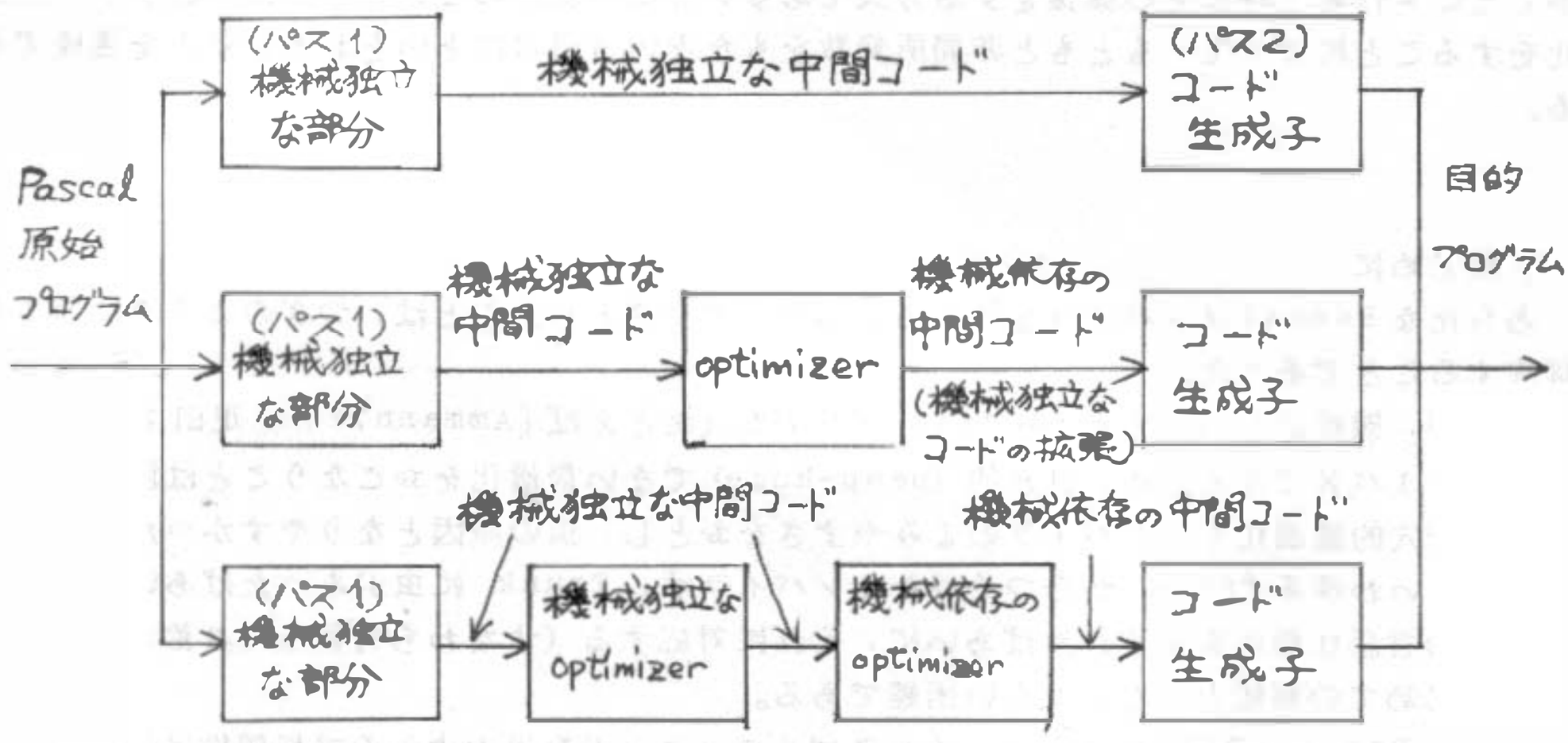


図1. 最適化，再移植性をかんがえたコンパイラ構成

2 再移植性

初期移植の目的で開発されたPascalコンパイラには，Pascal-P〔Nori76〕やM1-Pascal〔河村78〕がある。Pascal-Pのコンパイラは，仮想スタック計算機のためのコード（Pコード）を出力し，これをインタプリタで実行する。単にPascalコンパイラがうごけばよい，というときには，Pコードという比較的単純な機械語のインタプリタをかくだけでうごかすことのできるPascal-Pを使うのが最もたやすいが，これはインタプリタをつかうので実行速度がおそい。M1-Pascalは，コンパイラの出力するM1コードをマクロ展開して機械語を生成するという方法をとることによって，implementerの仕事をあまりふやすことなく，この問題を解決している。おなじ目的でつくられた他の言語のコンパイラとしてはBCPL〔Richards69〕，Alg●168C などがある〔Elsworth79〕。

上の2つのPascalコンパイラに共通なのは，文法上かなりの制約があるため，初期移植にもちいられるだけで，いずれは移植性のないコンパイラにとってかわられる運命にあるということである。初期移植性をよくすることと，機能を充実させることはあいいれない要求である。そこで Pascal-EPでは，初期移植には多少てまがかかったとしても，再移植にはてまがかからず，標準Pascalの機能をほぼ完全にもったものをつくることを目標とした。再移植とは，

もとのコンパイラに虫があったり，言語仕様が変更されたりしたときに，もとのコンパイラに手をくわえ，それをつかってふたたび移植をおこなうことである。
（初期）移植性をよくするためには，1）コンパイラの機械依存の部分を極力へらし，2）機械独立の部分と機械依存の部分とを分離することが必要である。これにたいし，再移植性だけを問題とするときは，2）を中心にかんがえればよい。1）は機能の充実と相反する要求だが，2）はそうでない。したがって，再移植性と機能の充実は両立させることができる。

Trunkは，1パス・コンパイラの中にちらばった骨組だけのコード生成子にimplementerが肉をつけるようにつくられており，初期移植性に関してはPascal-EPと同程度とかんがえられるが，再移植のためには，まったくあらたにコード生成子をかきなおすほかはない。それよりはparserをかきなおす方が楽なので，虫の駆除や言語仕様の改訂による変更があったとき，各implementerが独自にparserにつぎをあてていた。そのために，コンパイラの内容を充分理解し虫とりに熱心なひとがいるところではよいが，そうでない計算センターでは いつまでも虫がとれないということになったし，世界中で虫とりについやされた人月はばかにならないものであっただろう。したがって，初期移植性か，機能の充実か，という二者択一で後者を選択したばあいにとるべき道は，Trunkのようないきかたではなく，Pascal-EP のようないきかただとかんがえられる。

Pascal-EPでは，Pascal-Pをもとにし，Pコードを拡張することによって再移植性と機能の充実を両立させ，Pコードを機械語に展開するという方法によって，M1-Pascal同様，実行効率の問題を解決した。1節では，パス1を完全に機械独立にするようにかいたが，これはPascalのばあいむずかしい。実数，集合などがしめる記憶語数や，その境界条件（倍語境界など）は機械によってことなるので，完全に機械独立にするためには，記憶わりあてをのちのパスでおこなうようにしなければならないが，そうするとパス1からパス2へわたすべき情報が急激にふえ，のちのパスの仕事が複雑になる。Pascalのような比較的単純な言語のばあい，機械依存の部分があまり複雑になったのでは，それをつくるのに要するてまが，あらたに1パスのコンパイラをつくったのとたいしてかわらなくなってしまう。したがって，記憶わりあてはパス1でおこなうべきである。Pascal-P では，パス1にいくつかのパラメタをもうけて（定数定義の形であたえられている），それを機械におうじて設定することによって，パス1での記憶わりあてを可能にしている。たとえば，real型の変数がしめる記憶単位数をrealsize，記憶単位数であらわした境界条件をrealalという定数であらわしている（記憶単位が「語」なら，realal＝2で倍語境界をあらわす）。Pascal-EPでもこの方法を継承している。したがって，パス1は完全に機械独立ではないが，パラメタの設定はきわめてたやすい仕事であるから，機械独立性は高い。

3　コード体系

次節でPascal-EPの目標のひとつであった最適化についてのべるが，そのまえにそのコード体系についてのべよう。まず，Pコードについてのべ，次に，それにどのような変更をくわえたかについてのべる。

3.1.　Pコード

10の階乗を求めるプログラム（図2）と，それをコンパイルしたPコード（図3）をしめす。

3.2.　Pコードの設計変更とコードならびの限定

Pコードは，このimplementationにとって不都合な点をいくつかもっていた。それをつ

```
 1          {0 PROGRAM FACTORIAL(OUTPUT);
 2         10 FUNCTION FACTORIAL(X : INTEGER) : INTEGER;
 3          0 BEGIN
 4          3   IF X = 0 THEN FACTORIAL := 1
 5          9   ELSE FACTORIAL := X * FACTORIAL(X - 1);
 6         00 END;
 7         21 BEGIN
 8         21   WRITE(FACTORIAL(10));
 9         29 END.
```

図2. 10の階乗を求めるプログラム

ぎのようにして解決した。改訂されたコードをＥＰ　コードとよぶ。BCPL[Richards69]では，中間コードとしてOCODE をもちいているが，ＥＰコードは　これによくにている。

3.2.1.　スタック計算機とレジスタ計算機のひらき

Ｐコードはスタック計算機のための機械語であり，IBM360/370 に代表されるレジスタ計算機のための中間コードとしてもちいるにはいくつか不都合があった。それらはいずれも，スタック計算機ではスタックに値をのせるたびにスタック・ポインタをふやし（またはへらし），値をとりさるたびにスタック・ポインタをへらす（ふやす）のにたいし，レジスタ計算機でそれを模倣すると非効率になるため，スタック・ポインタの増減をおこなうタイミングをずらして，まとめて増減する必要があることから生じた。

最初は手続き（関数）よびだしの直後のスタック・ポインタの位置にかんする問題である。もとのＰコードでは手続きよびだしと関数よびだしを区別していない。そのかわり，もどりは関数からのもどりと手続きからのもどりを区別している。関数よびだしのばあいには値をかえすためスタックをひとつのばさなければならないが，それをもどり命令でおこなうように設計されているのである。しかし，レジスタ計算機にとっては，これをよびだす側で処理するようにしないと都合がわるい。そこで，よびだしの命令に，よびだされるべきものが手続きか関数かをしめすタグをつけくわえた。

この事情を説明しよう。いま，手続き（または関数）ｐのなかから関数ｆをよんだとしよう。ｆをよぶ直前のスタックの状態は図１Ａのようになっている（スタックは上から下へのびるとする）。実際のスタック・ポインタは⟶の位置をさしているが，コンパイラは⇢の位置がスタックの頂上だということを知っており，ここを仮想的なスタック・ポインタがさしているとかんがえることができる。ｆをよんで（図４Ｂ），もどってくると，スタックの状態は図４Ｃのようにならなければならない。仮想のスタック・ポインタはよびだしのまえよりひとつさき(ｆの値をふくんでいる)をさしている。もし，これが値をかえさない手続きよびだしだとすると，仮想スタック・ポインタの位置ははじめとかわらない。

インタプリタで実行するときには，図４Ａの状態から関数本体を実行したのち，関数からのもどり命令を実行するので，そこでスタック・ポインタを調整すればよいが，機械語を生成するときには図４Ａの状態から直接図４Ｃの状態へうつらなければならないので，よびだし命令にスタックをふやすかどうかの情報がふくまれていなければならない。それを知らせる最も容易な方法は，もどり命令とおなじ形のタグをつけくわえることである。

次は，コード生成はＰコードの解釈実行とはちがって最初から最後まで逐次的におこなわれるため，式スタックが空でない状態での無条件分岐があると生成が困難になる(たとえば分岐の直後は式スタックを空にして，とびさきに達したら，とぶまえのスタック・ポインタの値をおもいだすというようなことをしなければならないから)。これはコードじたいの問題というより，

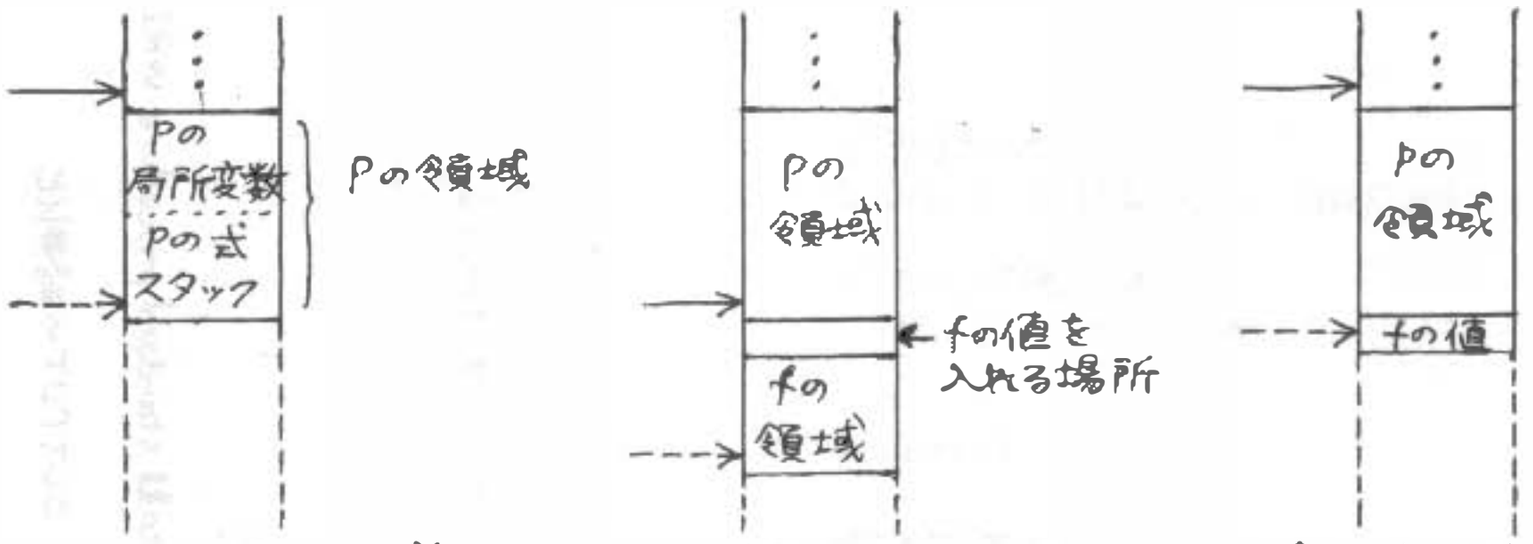
```
L   3
  ENT          1    L    4      -- FACTORIALの入口
  ENT          2    L    5      -- 局所変数をとる（その大きさはL4で定義）
  LODI         0              6 -- load X:INTEGER
  LDCI           0               -- load constant 0 (INTEGER)
  EQUI                           -- FACTORIAL内のスタック・ポインタの最大値(変位の)を示す.
  FJP        L   6               -- False jump to L6
  LDCI           1
I   10                           -- この点での命令カウンタの値をしめす.
  STRI         0               3 -- store FACTORIAL: INTEGER
  UJP        L   7               -- unconditional jump to L7
L   6
  LODI         0               6
  MST          1                 -- mark stack
  LODI         0               6
  LDCI           1
  SBI                            -- subtract INTEGER
  CUP          2    L    3       -- call user program
  MPI                            -- multiply INTEGER
  STRI         0               0
L   7
I   20
  RETI                           -- return with INTEGER value
L   4=           8
L   5=           9
L   8
  ENT          1    L    9
  ENT          2    L   10
  MST            0
  LDCI           10
  CUP          2    L    3
  LDCI           10
  LDA            0             7 -- load address OUTPUT
  CSP      WRI                   -- call standard procedure WRI
  RETP                           -- return from procedure
L   9=           10
L  10=           9
0
I   0
  MST            0
  CUP          0    L    3
  STP
0
```

図3. FACTORIALのPコード"

```
N     1      -- この点での原始プログラムの行をしめす.
N     2      --          〃
N     3      --          〃
N     4      --          〃
```

```
L   3
  ENT          2    L    4      -- レベル2で局所変数の大きさがL4
  LODI         2              4
  LDCI           0
  EQUI
  FJP        L   5
N     5
  LDCI           1
  STRI         2              0
I   10
  UJP        L   6
L   5
  LODI         2              4
  MST            1
  LODI         2              4
  LDCI           1
  SBI
  CUPI         1    L    3
  MPI
  STRI         2              0
L   6
H     6
  RETI         2    L    4
L   4=           6
M     7
N     8
L   7
I   20
  ENT          1    L    8
  LAO         42
  LAOF'OUTPUT  *                 } -- OUTPUTの初期化
  LDCI        85
  CSPT        85
  LAO         42                 } -- rewrite(OUTPUT)
  CSP         82
  MST            1
  LDCI        16
  CUPI         1    L    3
I   30
  LDCI        11
  LDA          1             42
  CSP            3                -- call standard procedure WRI
N     9
  LAO         42                 } -- OUTPUTの終期化
  CSPT        86
  RETP         1    L    0
L   8=         156
0
0
```

図6. FACTORIALのEPコード"



A. pからfをよぶ前の状態　　B. fの実行中　　C. fからもどったあとの状態

図4. 関数よびだしの前後のスタックの状態

コードならびの問題である。このようなコードならびを生成する必要は，Pascalという言語にとってはもちろん，コンパイラにとってもほとんどないため，ほとんど問題はおこらなかったが，ただ1か所，case文の選択式の評価のあと，その値をスタックにのせたまま無条件分岐をおこなっていたので，このコードならびをあらためる必要があった。

3.2.2　例

Pコードの例とおなじプログラムをEPコードにコンパイルした結果をしめす（図6）。

4.　最適化について

プログラミング教育での使用を目的とするコンパイラは，あまり最適化に時間をかけず，その分コンパイル時間を短縮した方が得策である。従って，Pascal-EPのばあい，コード生成子（通常のパス2）は必要最小限の最適化だけをするようにし，他はすべて，必要なときにのみつかわれるoptimizerにまかせるのがよい。必要最小限の最適化とは，optimizerをつかったときには満足のいく目的コードを出力することができる程度の最適化であり，そのうちの重要なものについて4.2でのべる。optimizerはまだ実際にはつくっていないが，4.1ではその可能性と限界についてのべる。

4.1.　optimizer

optimizerは，コンパイラ（パス1）の出力したEPコードを入力して，機械依存に拡張されたEPコードを出力する。EPコードを自由なアセンブリ言語とかんがえると，最適化の余地はわずかになってしまうが，そのつかいかたに制限をつければ，ある程度データのながれを解析し，最適化をおこなうことができる。たとえば，

```
a := x[1];      x[j] := 0;      b := x[1]
```

のようなプログラムで，j≠1ならば最後のx[1]をaでおきかえることができるが，j≠1であるかどうかがわからないときには，おきかえることはできない。ここでもしコンパイラの出力するEPコードの上で，x[1]が単純変数とおなじかたちでアクセスされているとすれば，optimizerはx[1]とx[j]がおなじ変数にたいするアクセスだとは認識できず，上のx[1]を単純変数でおきかえた形のときも，上のような最適化をすることはできないが，配列はつねに（基底番地）＋（変位）のかたちでアクセスすることにすれば，たしかにことなる変数にたいするアクセスだと認識でき，最適化をおこなうことができる。

しかしながら，Pコードのうえでは，原始プログラムがふくんでいた型に関する情報のおおくはうしなわれているので，ある種の最適化は非常に困難になる。たとえば，

```
procedure  p(var  a：T1; var  b：T2)
```

において，T1≠T2なら a, bはaliasses（おなじ変数にことなる名前がついたもの）ではありえないが，型がわからなくなると，そのことがわからなくなる。ポインタ変数についても変数パラメタとおなじことがおこる。この問題は，EPコードのかわりに，optimizerが必要とするだけの型にかんする情報をふくんだコードをつかうことによって解決可能である。

4.2. コード生成時の最適化

コード生成部はそれほど最適化をする必要はないが，最適化をおこなったEPコードが入力としてあたえられたとき，充分よいコードがだせるようにつくっておかなければならない。たとえば，

```
LODI  1  d1   -- 1はこのコードをふくむ手続きのレベル
LODI  1  d2
EQU
FJP   Ln
```

にたいしては

```
LW, r   d1, SP  -- LW = load word,
CW, r   d2, SP  -- CW = compare word
BNE     Ln      -- BNE = branch if not equal
```

を生成したい（rは適当なレジスタ，SPはスタック・ポインタ）。ほかの2項演算についてもほぼ同様である。

5. 実行時環境の管理法

Pascal-EPのMelcomへのimplementationは，実行時環境（スタック）の管理のしかたにいくつかの特徴がある。したがって，それらについてのべよう。

5.1. display, static link

従来のPascalコンパイラは，1) displayとstatic linkを併用し，displayはそれをこわす手続きをよぶ側で修復する方法，または2) displayをつかわず，static linkだけもちいる方法（Pascal-Pはこの方法をもちいることを前提としている）のうちいずれかを用いていた。1)の特徴は，レベルの高い手続き（すなわち，多重の入れ子の内側の手続き）からレベルの低い手続きをよんだあとの処理にレベル差に比例するてまがかかるが，よぶまえのてまは一定であること，レベルの高い手続きでレベルの低い手続きに属する変数をアクセスするのに要するてま，および手続きのそとへgotoするときのてま（非局所的gotoのてま）がレベル差に依存しないことである。2）の特徴は，レベルの高い手続きからレベルの低い手続きをよぶまえの処理にレベル差に比例するてまがかかるが，よんだあとのてまは一定であること，変数のアクセスおよび非局所的gotoのてまがレベル差に比例することである。

これらにたいし，第3の方法が存在する。それはdisplayをこわした手続きがその実行を終了するときにこわした部分を修復する方法である。この方法によれば，手続きよびだしおよび変数のアクセスのてまはほぼ一定の時間でできるが，そのかわり，手続きのそとへのgotoはプログラムの実行の履歴の長さおよび手続きのレベルの最大値に依存する（従って2）の方法より更に長い）時間がかかる。Pascal-EPでは，この方法をためしてみた。この方法で手続きよびだし，手続きの外へのgoto，手続き（関数）パラメタがどのようにimplementされるかをしめそう（ただし，いまのところ，手続き（関数）パラメタは実際にはimplementしていない）。ほかの方法については〔バルマン79〕などを参照されたい。

5. 1. 1.　手続きよびだし

手続きがよばれたときは，まず，その手続きのレベルのdisplayの内容をスタックに格納し，（したがって，これが通常のstatic linkにとってかわる）その手続きの局所変数などの領域をさすスタック・ポインタの値をdisplayへ入れる。手続き からもどる直前には ，逆に，スタックに格納してあった古いdisplayの値をdisplayにもどす。これにより，ある手続きの変数やラベルが見える範囲に制御があるときには，対応するdisplayをとおして，その手続きの領域をaccessすることができる。たとえば，　　手続きq から手続きpをよぶ場合，スタックおよびdisplayの様子は図7のようになる。

5. 1. 2.　非局所的gotoのばあい

手続きのそとへgotoでぬけだすときは，とびさきの手続き（または，主譜）を実行していたときの環境になるまでdisplayの状態をもどさなければならないが，スタックのbase address（displayでさされる番地）は新しくとられた領域ほどおおきい（スタックののび方が逆だとちいさい）ことを利用すると，つぎのようにすればできる。

とびさきの手続きのレベル1のdisplayには，期待される値がはいっている。したがって，これはそのままにしておいて，ほかのdisplayの値は，レベル1のdispayの値よりちいさく（スタックののびかたが逆のときは，おおきく）なるまでたぐっていく。すなわち，その display がさす領域　　に格納されているdisplayの値をdisplay にのせるという操作をくりかえす。この操作は，それまで実行していた手続きより高いレベルのdisplay（がもしあればそれ）についてもおこなわなければならない。（しかし，1いかのレベルについてはおこなう必要はない）したがって，すでにのべたように，手続きのレベルの最大値に依存する時間がかかる。

たとえば，図8Aのプログラムで　手続き q2からr へとびだすときのスタックおよびdisplayの状態は図8Bのようになる。

5. 1. 3　手続き・関数パラメタ

かんがえうる方法はいくつかある。もっともかんたんなのは，手続き実パラメタは，その手

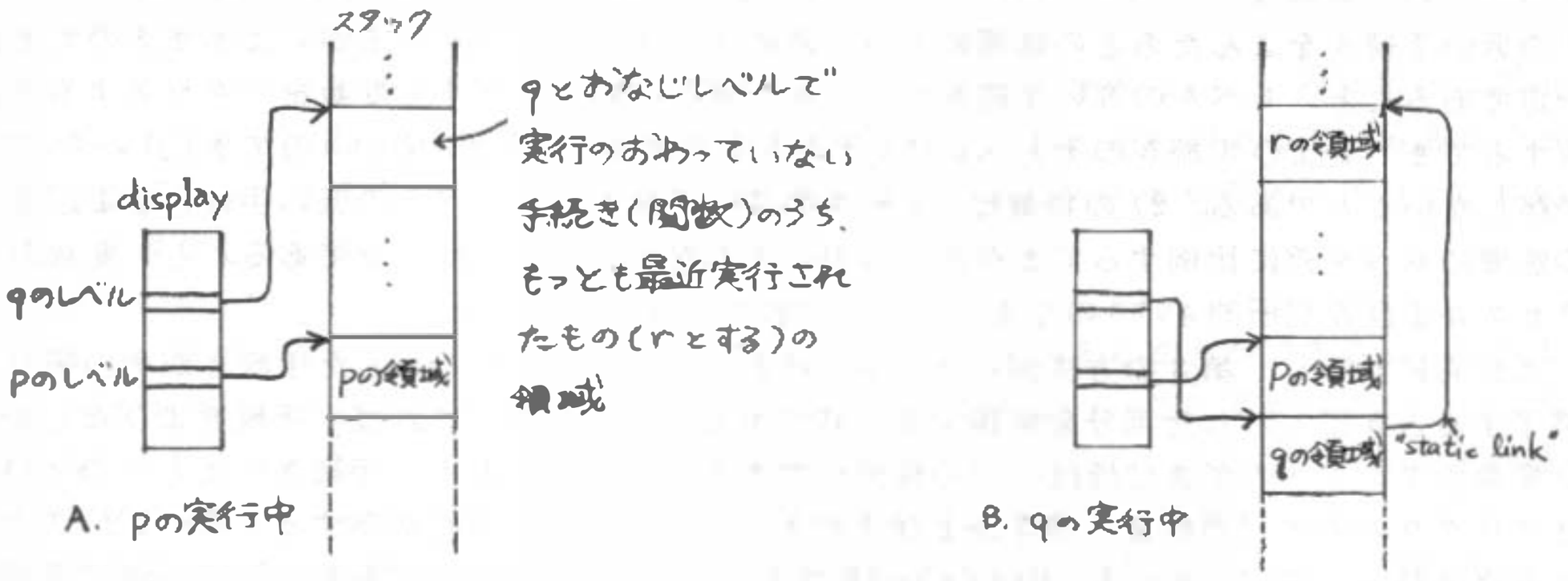


図7.　手続きpから手続きqをよぶときのスタックおよびdisplayの様子.
qのレベルがpのレベルより（ひとつ）高いとき（qがpの局所手続きのとき）は，displayの順序がことなるだけで，スタックの様子はかわらない。

```
procedure r;
  label 1;
  procedure q1;
    procedure q2;
    begin ··· goto 1; ··· end;
  begin ··· q2; ··· end;
  procedure p1;
    procedure p2;
    begin ··· q1; ··· end;
  begin ··· p2; ··· end;
begin ··· p1; ···
  1: ···
end
```

A.

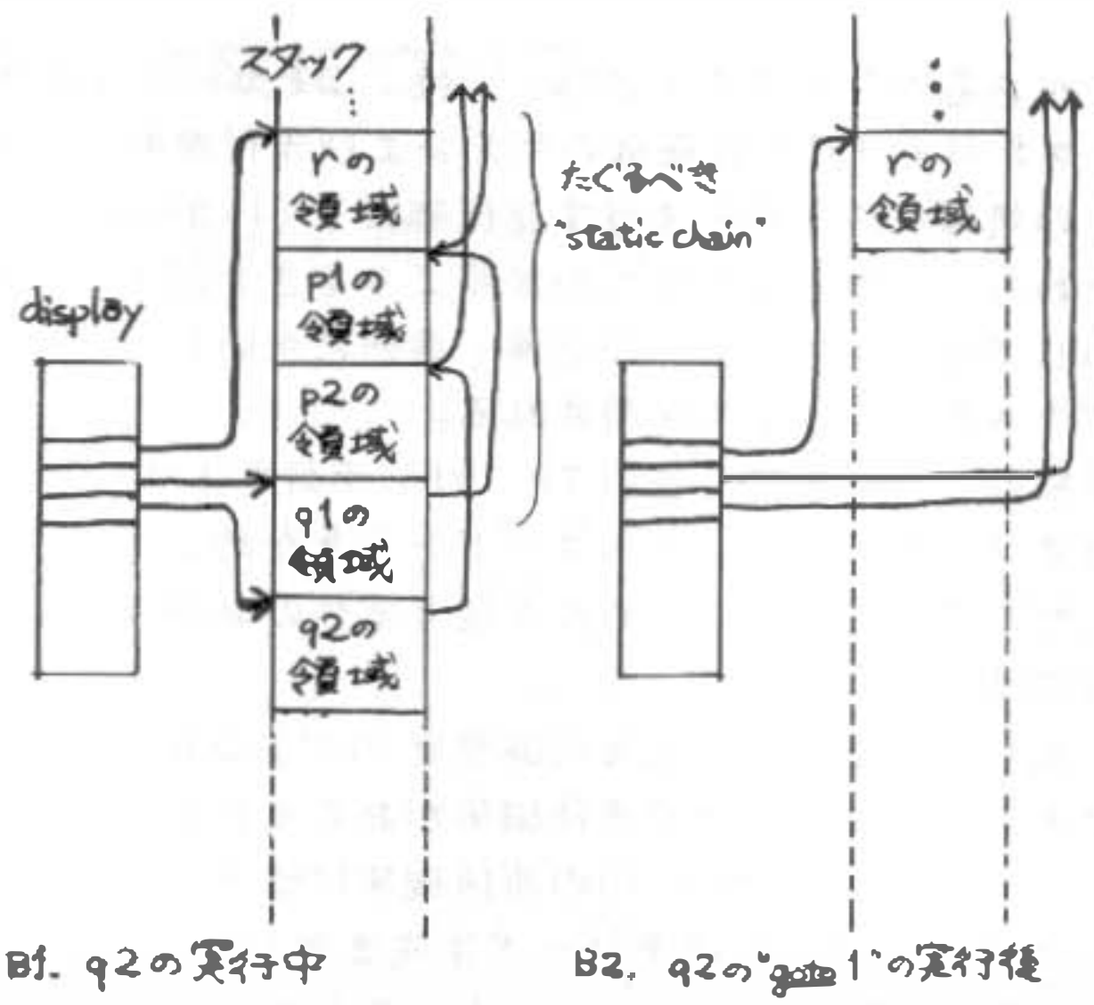


図8. 非局所的gotoのimplementation

続きの入口番地と実行時環境，すなわちdisplay（全体）からなるものとし，その手続きを実行するときには，よびだす側でそのときのdisplayをすべてスタック上に退避し，パラメタとしてわたされたdisplayのコピーをdisplayにコピーし，帰ってきてから退避したdisplayの値をもとにもどす方法である。実パラメタとしてdisplayの全体をもっていく必要はかならずしもない（そのうちひとつだけをもっていき，あとはgotoのばあいと同様にしてもとめることができる）が，その手続きを実行するさいにdisplayを退避することは省略できないであろう。詳細は，興味をもたれた読者にまかせたい。

5.1.4. 最適化

これまでのところでは，「第3の方法」にあまり利点はないようにみえる。しかし，この方法によれば，手続きpがその局所的変数をほかの手続きによって参照されず，かつほかの手続きから非局所的gotoでそのなかにとんでこられることもないときには，5.1.1.の操作をおこなう必要がない（ただし，実行中の手続きの領域の番地は，スタック・ポインタとして別にもっているとする）。そのばあいには手続きの実行の効率はもともと非局所変数をもたない（たとえばConcurrent Pascal〔Brinch Hansen 75〕やBCPL〔Richards 69〕のような）言語とひとしくなる。従来の方法をもちいたばあいでも最適化の余地はあるが，この方法にくらべてむずかしい。

5.2. dynamic linkの削除

Pascalでは，各手続きの領域のおおきさはコンパイル時にきまるので，手続きの実行前にスタック・ポインタをふやし（へらし），実行後におなじだけへらす（ふやす）命令を実行させれば，dynamic linkは不要である。すなわち，dynamic linkとして保存すべき情報を，あらかじめプログラムのなかにうめこむことができる。こうすることによって，手続きの実行にかかるコストを，時間，場所ともに，多少節約することができる。

6. ラベルの処理

EPコンパイラと，そのMelcomのためのコード生成子にかんしていくつかちいさなエピソ

ードがあるが，そのうちのひとつは，EPコード上のラベルの処理にかんすることである。EPコンパイラは分岐命令の行先および先行参照のためにラベルを生成するが，コード生成部はその処理をほとんどすべて連係編集子（linkage editor）の先行参照機能にまかせている。すなわち，ラベルがそれへの参照よりもさきにあらわれるばあいもふくめて，ラベルの定義にたいしては「先行」参照の定義，参照にたいしては「先行参照」を出力する。先行参照は，目的プログラムのおわりまで保持される。

従来のコンパイラ〔金田79〕は，手続き本体（宣言につづくbeginから対応するendまで）単位でコード・バッファへコードをかきため，できる限りの先行参照を解決したうえで外部へかきだしていた。そのために手続き本体のおおきさには制限があったが，それはあたらしい処理系ではなくなった。

しかし，連係編集子は先行参照を$O(n^2)$の算法で処理しているらしく，ちいさなプログラムのばあいには充分高速で連係編集がおこなわれるが，たとえば原始プログラムで約4800行あるEPコンパイラ（パス1）の連係編集には全CPU時間で約7.8分かかる(Melcom Cosmo 700で)。自己コンパイル（EPコード生成まで）に約1.3分，そのEPコードから目的プログラムを生成するのに約0.9分かかること，先行参照がほとんどない同程度の目的プログラムの連係編集が0.3～0.4分でできることをかんがえると，相当におそいことがわかるだろう。

最後に

Pascal-EPを再移植と最適化のしやすい処理系にするという目標は一応達せられたとおもう。しかし，まだ移植はしていないし，optimizerもできていないので，決定的な評価をくだすことはできない。しかし，従来のコンパイラ〔金田79〕と比較すると，目的プログラムはoptimizerをとおさなくてもすでにコードのながさで約20%，実行時間でも10～20%減少した。ただし，目的プログラムをつくるまでの時間は，みじかいプログラムのばあいでも30%増くらい，EPコンパイラにいたっては6節でのべた理由により5倍以上かかる。

参考文献

[Ammann,U.77] On the code generation in a Pascal compiler,Software practice and experience,7,391-423

〔バルマン，デビッド，M. 79〕 スタック計算機入門，bit，11:5-11:7（訳：井田昌之），from Computer,10:5,14-28[1977]

[Brinch Hansen,P. 75] The programming language Concurrent Pascal,IEEE Trans. on Software Engineering,SE1:2,199-207

[Elsworth,E.F. 79] Compilation via an intermediate language,Computer Journal, 22:3,226-233

〔正田潤雄76〕 コンパイラのキットを用いたPascalの移植，日経エレクトロニクス，1976.12.3，100-131

〔金田　泰79〕 東京大学教育用計算機センターにおけるPascalについて，教育用計算機センター報告 No.13. 3-25

〔河村知行78〕 Pascal compilerのportabilityの向上をめざして，筑波大学数学研究科修士論文

[Nori,K.V. et al. 76] The Pascal <P> compiler : Implementation notes revised edition,Institut fuer Informatik,Eidigenuessische Technische Hochschule

[Richards,M. 69] BCPL : A tool for compiler writing and system programming, Proc. AFIPS SJCC