北海道大学工学部研究報告
第85号（昭和52年）

Bulletin of the Faculty of Engineering,
Hokkaido University, No. 85 (1977)

ブートストラッピング技法による FACOM 230-75 用 Pascal コンパイラの作成

宮本衛市* 上原憲二*

（昭和52年3月28日受理）

Implementation of a Pascal Compiler for the Computer FACOM 230-75 by the Bootstrapping Technique

Eiichi MIYAMOTO Kenji UEHARA

(Received March 28, 1977)

Abstract

Pascal, which is an ALGOL-like general purpose programming language, was implemented for the computer FACOM 230-75. Generally Pascal is characterized by its powerful facilities of user defined data structures, together with its compact grammar.

This paper deals with the implementation process by the bootstrapping technique, the compiling algorithms, particularly analysis regarding expressions, and run-time data structures. The implemented compiler is written in Pascal itself as a consequence of the bootstrapping technique, so that the compiler is highly flexible for the maintenance and extension of its specifications.

1. まえがき

プログラミング言語 Pascal[1),2),3)] は，N. Wirth によって提唱された ALGOL 系の汎用言語であり，ALGOL 60 を基礎にして言語設計されているが，データ構造記述能力が格段に強化されているのが，この言語の最も大きな特徴である。また，Pascal は系統的なプログラミング教育用として使え，かつ通常の計算機で効率よく作成できることを意図して，簡潔な言語仕様にまとめられている。このように，Pascal は言語仕様上の簡潔さと豊富なデータ構造記述能力を兼ねそなえているバランスのとれた言語であるため，通常の計算用としてばかりでなく，オペレーティングシステム記述用[4)] として，あるいはプログラミング研究のモデル言語[5)] として，世界各地および国内においても普及が進んでいる言語である。

著者らは，北海道大学大型計算機センターに設置されている FACOM 230-75 用 Pascal コンパイラの開発を進めてきた。開発目標としては，標準仕様を満足する Pascal コンパイラとすることはもちろんであるが，さらに実用に耐えうるコンパイラとするため，エラー検出を厳密に行うばかりでなく，ALGOL 系言語の弱点の一つであるエラー回復処置を重点的に行い，できるだけソースプログラム中に存在するエラーを正当に検出することであった。また，今後 Pascal

* 情報工学専攻■情報システム工学講座

をソフトウェア研究のための言語あるいはモデルとして利用できるように，ファイルの取扱いが容易であり，他言語で書かれたプログラムとも結合できること，さらには将来起りうる言語仕様の拡張要求に対しても容易に対処しうることなどが開発に当って考慮された。

今回開発した FACOM 230-75 用 Pascal コンパイラの特徴をあげると次のようになる。

（a） Pascal の標準仕様は完全に満足されている。

（b） エラー検出および回復処置が強力に施されている。

（c） 1語中に構造化データをつくる機能を有している。

（d） FORTRAN 用ライブラリおよびアセンブラプログラムと結合することができる。

（e） ファイルを容易に扱うことができる。

（f） いくつかのコンパイラオプションを有している。

このコンパイラは ALGOL を出発点としてブートストラッピング技法により作成されており，最終的に生成されたコンパイラは Pascal 自身で記述されているのが大きな特徴である。そのため，コンパイラのデバッグ，あるいは必要に応じての言語仕様の拡張などが Pascal プログラム上で行うことができるので，非常に明瞭かつ安全に作業を行うことができる。

2. ブートストラッピング・プロセス

Pascal コンパイラを作成する方法として，いくつかの方法が考えられる。1つはアセンブラレベルの言語でコンパイラを記述して作成する方法であり，Pascal 程度のコンパイラになると，コーディング，デバッギングあるいは拡張にさいしての柔軟性に大きな困難さを伴う。次に考えられるのが高級言語を用いてコンパイラを記述する方法である。Pascal がコンパイラ記述にも適していることから，Pascal コンパイラを Pascal 自身で簡潔に記述することができるが，当然そのコンパイラは目的とする計算機の機械語にまで一度コンパイルされなければならない。そのためには，人手で直接コンパイルする方法，すでに Pascal コンパイラが働いている他の計算機でコンパイルしてもらう方法，あるいは仮想的な計算機で働くように作成された Pascal 'P' コンパイラでインタプリータを介してコンパイルする方法[7] などが考えられている。しかし，人手で間違いなくコンパイルを行うのは大変難しく，また他の計算機でコンパイルしてもらうためには手近にそのような計算機がなければならず，'P' コンパイラではファイルを扱えないというような難点がある。

一方，コンパイラ作成過程から考えると，最初から目的のコンパイラを一度に作成する方法と，何段階かに分けてコンパイラを成長させてゆくブートスラッピング技法とが考えられる。手近に Pascal コンパイラが働いている計算機がない場合，Pascal コンパイラをアセンブラ言語はもちろん，高級言語でも Pascal ほどの記述能力がないもので始めから記述するのでは大変な手間を要す。そこでブートストラッピング技法では，図1に示すように，目標とする言語 L_N を作るために，まずそのサブセットである言語 L_1 のコンパイラを他の言語 L_0 で記述して作成することから始め第 i 段階 ($i \geq 2$) では言語 L_i のコンパイラを L_{i-1} で記述する。したがって，各段階で対象とする言語は段

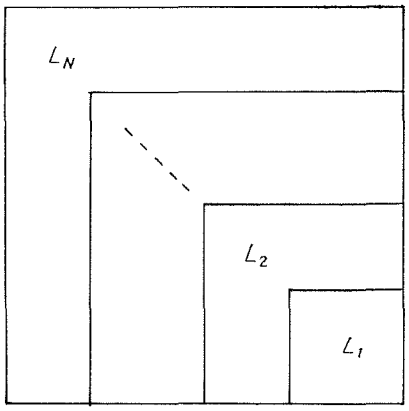

図1 ブートストラッピング過程における言語仕様の拡大

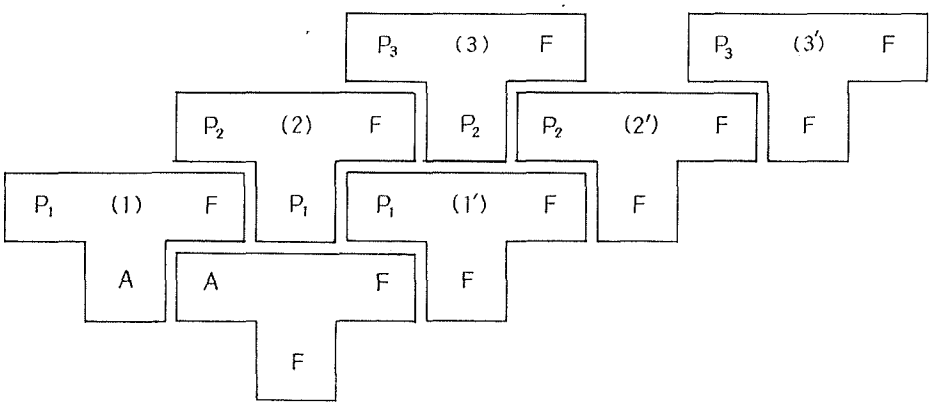

図2　T-ダイアグラムによるブートストラッピング過程の表示
P_1: $Pascal_1$,　P_2: $Pascal_2$,　P_3: $Pascal_3$
F: FACOM 230-75 コード,　A: ALGOL

階的に大きくなってきているので，前段階よりも言語仕様上拡大された部分を主として考慮すればよく，しかも前回記述した言語よりも，より大きな言語で記述することができる。

筆者らはブートストラッピングプロセスを3段階設けて Pascal コンパイラを作成した[8]。その過程を T-ダイアグラムで示すと，図2のようになる。第1段階のみは Pascal 以外の言語を用いなければならず，Pascal と類似した ALGOL を用いて記述している。しかし，FACOM 230-75 用 ALGOL はデータ構造記述能力が乏しく，かつ procedure の recursive call も仕様から除外されていることから，$Pascal_1$ ではその言語仕様を小さくおさえ，コンパイラの負担を軽くしている。それゆえ，$Pascal_1$ の仕様からは array 構造を除きデータ構造記述に関する部分はほとんど省き，エラー回復機能も含めていないが，procedure の recursive call ができるプログラム構造を有している。第2段階では，$Pascal_1$ を用いて $Pascal_2$ コンパイラを記述しており，その言語仕様はほとんど標準仕様を満足しているが，$Pascal_2$ は $Pascal_3$ の記述用として使うので，そのためには不要な部分を省略してある。この段階でのコンパイラは，正常なプログラムをコンパイルすることを目的としているため，エラー回復機能は組込まれていない。第3段階では $Pascal_3$ コンパイラを $Pascal_2$ で記述しており，$Pascal_3$ が Pascal 標準仕様を満足しているほか，エラー回復機能などが組込まれ，実用に供しうるコンパイラとなっている。

各段階で作成した $Pascal_i$ $(i=1, 2, 3)$ は，記述に用いた $Pascal_{i-1}$ のスーパセットになっている。しかし，各コンパイラのプログラム構造およびデータ構造は，基本的には同一の構造図式を有しており，したがって前段階での結果をそのまま取入れることができるように設計されている。

3.　コンパイラのプログラム構造とデータ構造

$Pascal_3$ コンパイラは，それ以前のコンパイラの内容をすべて包含しているので，以下では $Pascal_3$ について述べることにする。

語句解析では，1語句および1文字先読みを基本とし，ソースプログラムから特殊記号，識別子，数および文字列を抽出する。

構文解析では，文法エラーの厳密な検出と，検出後の回復処置を入念に行っており，そのアルゴリズムの基本パターンは次のとおりである。すなわち，コンパイラは構文規則に従って順次構文の解析をするべく語句解析からの語句を待ちうけているが，到来した語句があらかじめ決め

られている語句セットの要素であることを確認する。この語句セットは，構文の先頭となりうる語句のセットであり，したがってソースプログラムから読出された語句がそのセットの要素でなければエラーとして検出される。エラー検出後は，上述の語句セットか，あるいは次の構文解析を再開することのできる語句セットのいずれかに含まれる語句が現われるまでソースプログラムを読みとばす。前者は，文法上冗長な語句が挿入されていることへの対策であり，後者は，当面の構文に関するそれ以降の解析を放棄することを意味する。そこで再び，該当した語句がその構文に関する語句セットにある要素であるかどうかを確め，その要素であればその構文に関する構文解析に入るが，そうでなければ，次の構文解析に制御を移す。さらに，構文解析の出口においても次の構文解析を始めるための語句が現われていることを確認して，当面の構文に関する処理を終了する。

上述のアルゴリズムを次式で定義される *constant* の構文を例にとって説明する。

$$\langle constant\rangle ::= \langle unsigned\ number\rangle \mid \langle sign\rangle\ \langle unsigned\ number\rangle \mid \langle constant\ identifier\rangle \mid \langle sign\rangle\ \langle constant\ identifier\rangle \qquad (1)$$

これより，*constant* の解析を始められる語句セットは次の語句からなる。

$$[plus\_sym,\ minus\_sym,\ number\_sym,\ string\_sym,\ identifier\_sym] \qquad (2)$$

ここで，(2)式の各要素は語句を記号化し，Pascal におけるスカラ量として扱っている。

エラー検出および回復処置のみに着目して，*constant* の処理内容を示すとプログラム1のようになる。ここで，*restart_sym_set* は procedure *constant* を呼んだ procedure で決る次の構文解析の再開となる語句のセットである。他の識別子については，識別子自身がその意味も表わしているものとする。なお，識別子は説明上，下線でつながれた英数字で表わしている。

コンパイラのデータ構造で基本となるのは，識別子に関するもの，データ構造の記述に関するもの，ディスプレイ構造に関するもの，および数式解析上の属性に関するものである。

識別子に関する *identifier_record* は，新しい識別子が現われるたびごとに1レコードが生成され，主として計算機のハードウェア上への写像を示す。このレコードは，対応するレベルのディスプレイレコードを根としたバイナリトリー構造に組立てられる（付録，プログラム6参照）。

プログラム1 *constant* におけるエラー処理アルゴリズム

```
procedure constant(restart_sym_set: symbol_set);
begin
  if not (symbol in constant_head_set + restart_sym_set) then
    begin  error; skip_to(constant_head_set + restart_sym_set) end;
  if symbol in constant_head_set then
    process_constant
  else error;
  if not (symbol in restart_sym_set) then
    begin  error; skip_to(restart_sym_set) end
end;
  ..................
constant_head_set := [plus_sym, minus_sym, number_sym, string_sym,
                      identifier_sym];
```

データ構造の記述に関する *structure_record* は，タイプ宣言で与えられるデータ構造を内部的に表現するものであり，一般に *identifier_record* と共にトリー構造を作ってデータ構造の意味記述を行う（付録，プログラム7参照)。

ディスプレイ構造に関する *display_record* は，ブロック構造および with statement 内での識別子の有効範囲の設定を行うためのもので，各レベルにおいて，*display_record* 内の *first_name* を根とする *identifier_record* のバイナリトリー構造が引用可能な識別子であることを示す。このレコードは，procedure または function 定義で新しいブロック構造に入るたびごとに，および with statent で record variable が現われるたびごとにレベルを1つあげてディスプレイ構造に追加され，ブロックから解放されるとき，および with statement を終了したときに，該当するレコードが削除される（付録，プログラム8参照)。

属性に関する *attribute_record* は，数式解析に必要なオペランドのタイプ属性およびアドレス情報などを与えるもので，1オペランドにつき1レコードが割当てられる（付録，プログラム9参照)。

上述の各レコードは，必要に応じて standard procedure *new* を用いて生成され，その有効範囲が切れるブロックの出口で割当てられた領域を返却する。

4. 数式解析のアルゴリズム

Pascal では，数式中の演算子の優先順位は次の4つに簡素化されている。

（a） 第1順位： NOT（論理演算における否定）
（b） 第2順位： 乗算演算子（スカラ量の乗除算，AND 演算，積集合）
（c） 第3順位： 加算演算子（スカラ量の加減算，OR 演算，和集合，差集合）
（d） 第4順位： 関係演算子（データタイプに応じた同値，比較演算）

この優先順位に従った数式解析は，recursive call を含めた procedure call のスタックと，*attribute_record* のスタックを用い，プログラム2に示すように，きわめて簡潔なアルゴリズムで示すことができる。なお，同プログラムではエラー検出および回復に関するアルゴリズムはすべて省略してある。

属性レコードはすべてポインタ変数を介して指示されているが，これは演算結果をできるだけアキュムレータに保存しておき，むだな中間結果の転送命令を省くためである。そのため，プログラム9（付録参照）に示すように，オペランドの種別として，定数および変数の場合のほかにアキュムレータ上にある場合と作業領域に待避した場合を設けてある。演算後はすべて中間結果の属性を第1オペランドである *formal_attr_p*↑ に集約して，その種別を *accumulator* としておく。引続く演算でアキュムレータ上にある中間結果がオペランドになっているときは，そのまま演算を続行するようにコードを生成すればよいが，アキュムレータ上の結果が次の演算のオペランドでないときには，次の演算に入る前に作業領域に待避させる。以上のアルゴリズムを乗算演算子の処理を例にとって示すと，プログラム3のようになる。

Pascal ではポインタ変数が使えるために，ポインタ変数と構造化データタイプとの組合わせで複雑な変数指定を行うことができる。そこで，オペランドが変数の場合のアドレス指定として次の3つの場合を設け，間接引用の多重度を1以下におさえるため，2多重以上の間接引用があれば，そのアドレス情報を一旦作業領域に待避させるようなコードを生成している。

（a） 直接引用（DA)： 実効アドレスは *const_address*
（b） 間接引用（IA)： 実効アドレスは (*var-address*)+*const_address*, *var-level* は任意

（c） 作業領域へ待避 (WA): 実効アドレスは (*var_address*)+*const_address*, *var_level* は current level

変数処理のアルゴリズムをプログラム4に示す。 with statement で指定されるレコード変数は，そのアドレス情報をそのままディスプレイレコードに移され，そのレコードのフィールド

プログラム2 数式解析アルゴリズム

```
procedure expression(formal_attr_p: attr_p);
  var local_attr_p: attr_p;
  procedure simple_expression(formal_attr_p: attr_p);
    var local_attr_p: attr_p; save_symbol: symbol_type;
    procedure term(formal_attr_p: attr_p);
      var local_attr_p: attr_p; save_symbol: symbol_type;
      procedure factor(formal_attr_p: attr_p);
      begin
        if symbol=not_sym then
          begin input_symbol; factor(formal_attr_p);
            not_operation(formal_attr_p)
          end
        else
          if symbol=left_parenthesis then
            begin input_symbol; expression(formal_attr_p); input_symbol end
          else selector(formal_attr_p)
      end;   (* factor end *)
    begin  factor(formal_attr_p); new(local_attr_p);
      while symbol in mul_sym_set do
        begin  save_symbol := symbol; input_symbol; factor(local_attr_p);
          mul_operation(formal_attr_p, local_attr_p, save_symbol)
        end
    end;   (* term end *)
  begin  term(formal_attr_p); new(local_attr_p);
    while symbol in add_sym_set do
      begin save_symbol := symbol; input_symbol; term(local_attr_p);
        add_operation(formal_attr_p, local_attr_p, save_symbol)
      end
  end;   (* simple_expression end *)
begin  simple_expression(formal_attr_p); new(local_attr_p);
  if symbol in rel_sym_set then
    begin save_symbol := symbol; input_symbol; simple_expression(local_attr_p);
      rel_operation(formal_attr_p, local_attr_p, save_symbol)
    end
end;   (* expression end *)
```

プログラム3　データ転送アルゴリズム

```
procedure load(formal_attr_p: attr_p);
begin
  if formal_attr_p<>acc_attr_p then
    begin  if acc_attr_p<>nil then save_accumulator(acc_attr_p);
      load_accumulator(formal_attr_p)
    end
end;
  ..................
procedure mul_operation(formal_attr_p1, formal_attr_p2: attr_p;
                        symbol: symbol_type);
begin  load(formal_attr_p1); mul_code(formal_attr_p2, symbol);
  acc_attr_p := formal_attr_p1; formal_attr_p1↑.attribute_kind := accmulator
end;
```

プログラム4　変数解析アルゴリズム

```
procedure selector(formal_attr_p: attr_p);
begin  base_address(formal_attr_p);
  while symbol in selector_sym_set do
    if symbol=left_bracket_sym then index_code(formal_attr_p)
    else
      if symbol=arrow_sym then indirect_code(formal_attr_p)
      else const_displacement(formal_attr_p)
end;
```

が現われると，そのフィールドのアドレス情報として複写する。

5.　実行時のデータ領域とその構造

プログラムには，実行に先立ってデータ領域が割当てられる。この領域は，図3に示すようにスタック領域とヒープ領域に2分され，スタック領域の先頭アドレスがベースレジスタ4に設定される。スタック領域では，procedure あるいは function が呼ばれるたびごとに，図4に示すようなデータブロックがスタックされる。ダイナミックリンクは1つ前のデータブロックとリンクしてスタック構造をつくり，スタティックリンクはレベルの1つ低いデータブロックとリンクして，ブロック構造をなす各変数へのアクセスを可能にする。図5にデータブロックが結合されている様子を例で示す。

FACOM 230-75 には，プログラムで自由に使えるインデックスレジスタ (XR) が7個ある。そのうち，XR 1 をスタック領域の最新のデータブロックの基準番地を保持するために用い，XR 6 と XR 7 をアドレス計算のための作業用として使っている。残りの4個のレジスタを任意のデータブロックの基準番地を示すために使い，一度セットされると計算の流れが変らない間はその値を保持しているものとして扱う。そのため，各レジスタがどのデータブロックの基準番地を保持しているかを示す表を用意しておき，レベル i のデータブロックへアクセスがあったとき，この表をみて該当するレジスタがなければ，レベル i よりも高く，かつ最も近いレベルのデータブロックの基準番地を保持しているレジスタを探し，そのブロックのスタティックリンクからさかの

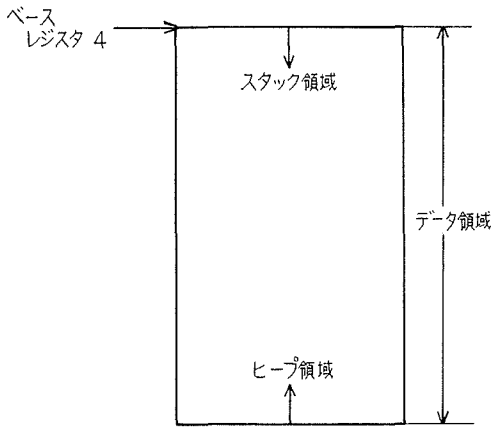

図3 データ領域の設定

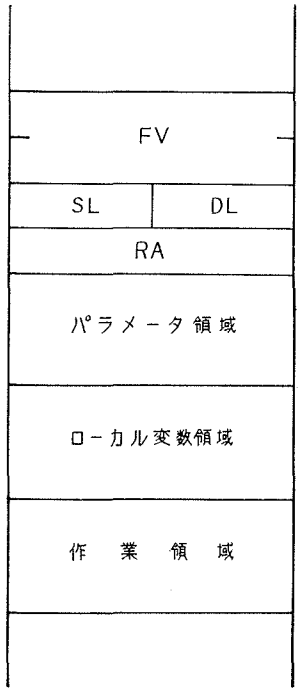

図4 データブロック
FV: 関数値, SL: スタティックリンク
DL: ダイナミックリンク, RA: 戻り番地

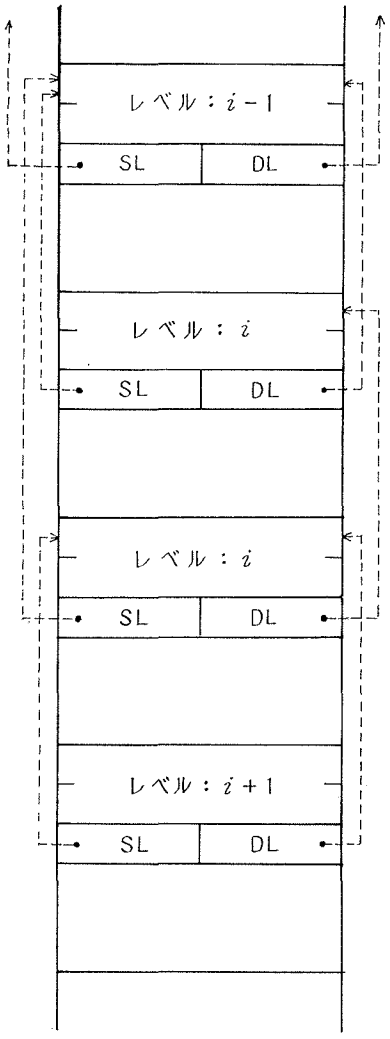

図5 スタック領域におけるデータブロックの結合例

ぼって，レベル i の基準番地を求めるようなコードを生成する。そのアルゴリズムをプログラム5に示す。

一方，ヒープ領域は standard procedure *new* により生成される変数にメモリを割当てるための領域で，アドレスが小さくなる方向に向って割当てる。*new* で割当てられたメモリは standard procedure *dispose* あるいは *release* により返却することができるが，これら返却されたメモリはフリーリストに組込まれ，再び *new* でメモリ要求があったときにそなえている。

プログラム5　基準番地設定アルゴリズム

```
var index_table: array[1..7] of level_range;
  ..................
function base_index(level: level_range): index_range;
  var index_number: index_range; lev: level_range;
begin
  if index_table[level] <> 0 then base_index := index_table[level]
  else
    begin  lev := level + 1;
      while index_table[lev]=0 do lev := lev + 1;
      request_index(index_number);
      while lev <> level do
        begin  load_static_link(index_number, lev); lev := lev - 1 end;
      base_index := index_number
    end
end;
```

さて，呼ばれた procedure あるいは function から戻るときには，関数値の入っている先頭の2語を残してデータブロックを返却する。したがって，function を呼んだときには，一旦 function のためのデータブロックがスタックされるが，戻ってきた後では関数値の入っている2語が作業領域に追加されたことになる。

6. オブジェクトプログラムの形式

Pascal コンパイラは，Pascal 文法に従って書かれたソースプログラムを FACOM 230-75 のコードに翻訳し，RB (relocatable binary) 形式のオブジェクトプログラムに直接変換する。もちろん，recursive call を許す他の高級言語，例えば PL/I などに変換することも可能であるが，コンパイル効率および実行効率をあげ，実用的な言語とするため，直接機械語におとしている。procedure および function は分割型順編成ファイル[9]の1つのファイルエレメントとして格納され，実行時に必要なライブラリ，および RB 形式に変換されている Pascal 以外の言語で書かれたプログラムと結合され，1つの論理的に完結したプログラムとなる。

7. あ と が き

筆者らが開発した Pascal コンパイラはブートストラッピング技法により作成されているので，コンパイラ自身が Pascal で書かれており，そのためコンパイラの保守性，拡張性にすぐれており，さらに以下に述べるような他計算機へのコンパイラの移植に利用することができる。

一般にコンパイラは，特定の計算機に依存しない部分と依存する部分に分けて考えることができ，前者がソースプログラムを文法に従って構文解析する部分に相当し，後者が前者の解析に基づいて特定の計算機のコードに変換する部分に相当する。そこで，ある特定の計算機のために書かれたコンパイラから，その計算機に依存する部分を削除し，その代り他の計算機向けのものに置換えるだけで，他の計算機向きのコンパイラを作ることができる。通常は，構文解析をする部分はコンパイラ全体にまたがった大局的な論理構造を構成するのに対し，コード生成部分は比較的，局部的な論理構造からなっている。したがって，上述のように計算機依存部分のみの手直

しで新しいコンパイラを作成できれば，その作業量を大幅に軽減することができる。現在，この方法によりFACOM 230-75 用 Pascal コンパイラから ECLIPSE 用 Pascal コンパイラへの移植を行っている。

現在の Pascal コンパイラでは，生成コードの最適化あるいは実行時のデータ検査は局部的にしか行っていないが，Pascal のように構造化された言語では大局的な最適化，あるいはデータ検査を組織的に行うことができる。そのアルゴリズムの開発とコンパイラへの組込みは今後の課題である。

文　献

1) Wirth, N.: Acta Informatica 1 (1971), p. 35-63.
2) Wirth, N.: Berichte der Fachgruppe Computer-Wissenschaften, ETH Zürich, 5 (1973).
3) Jensen, K. and Wirth, N.: A User Manual for Pascal (1974), ETH Zürich.
4) Hansen, P. B.: Software, 6 (1967), p. 141-200.
5) Suzuki, N.: Automatic Program Verification II (1974), Stanford AIM-255.
6) 宮本：北大大型計算機センターニュース，9 (昭52)，2.
7) Nori. K. V. *et al.*: Berichte des Instituts für Informatik, ETH Zürich, 10 (1975).
8) 宮本，上原：情報処理学会第17回全国大会講演論文集 (昭51).
9) 富士通：システムマクロ文法書 II.

付　録

プログラム6　識別子に関するデータ構造

```
identifier_record_pointer = ↑identifier_record;
identifier_record =
  record  identifier_name: packed array[1..8] of char;
    left_link, right_link, next_link: identifier_record_pointer;
    case identifier_kind: identifier_type of
      type_id: (forward_declared_type: Boolean);
      const_id: (value: value_type);
      var_id: (var_kind: direct_indirect; var_level: level_range;
               var_address: address_range);
      field_id: (field_address: address_range;
                 case packed_field: Boolean of
                   true: (bit_position: bit_range));
      procedure_id,
      function_id: (case p_f_declared_kind: p_f_declared_type of
                      standard: (p_f_key: p_f_key_type);
                      declared: (p_f_level: level_range;
                                 case p_f_kind: p_f_type of
                                   formal: (p_f_address: address_range);
                                   actual: (p_f_class: p_f_class_type;
                                            p_f_count: p_f_range;
                                            save_data_address: address_range)))
  end;
```

プログラム7 データ構造記述のためのデータ構造

```
structure_record_pointer = ↑structure_record;
structure_record =
  record  size: word_bit_size;
    case structure_form: form_type of
      scalar: (case scalar_kind: scalar_declared_type of
                 declared_scalar: (first_const: identifier_record_pointer));
      subrange: (range_type: structure_record_pointer;
                 lower_bound,upper_bound: value_type);
      pointer: (refered_type_id: identifier_record_pointer);
      power_set: (element_set: structure_record_pointer;
                  packed_set: Boolean);
      array_st: (element_type, index_type: structure_record_pointer;
                 case packed_array: Boolean of
                   true: (elements_per_word: bit_range));
      record_st: (first_field_id, field_id: identifier_record_pointer;
                  record_variant: structure_record_pointer;
                  packed_record: Boolean);
      file_st: (element_of_file: structure_record_pointer;
                segmented_file: Boolean;
                buffer_size, window_size: address_range);
      tagfield: (tagfield_pointer: identifier_record_pointer;
                 first_variant: structure_record_pointer);
      variant: (first_variant_field: identifier_record_pointer;
                next_variant, subsequent_variant: structure_record_pointer;
                variant_value: value_type)
  end;
```

プログラム8 ディスプレイ構造記述のためのデータ構造

```
display_record =
  record  first_name: identifier_record_pointer;
    case block_or_record: occur_type of
      record_dp: (record_access: direct_indirect; record_level: level_range;
                  var_displacement, const_displacement: address_range;
                  case packed_rec: Boolean of
                    true: (var_bit_displacement: address_range;
                           const_bit_displacement: bit_range))
  end;
```

プログラム9 オペランド属性に関するデータ構造

```
attribute_record =
  record  type_pointer: structure_record_pointer;
    case attribute_kind: attribute_type of
      const_at: (const_value: value_type);
      variable_at: (var_access: access_kind; var_level: level_range;
                    var_address, const_address: address_range;
                    case packed_var: Boolean of
                      true: (var_bit: address_range; const_bit: bit_range));
      accumulator: ( );
      work_stack: (work_address: address_range)
  end;
```