Pascal 拡張の一手法とモジュール構造をもつ Pascal への応用†

真野 芳久††

Pascal は最もすぐれたプログラミング言語の一つであるが，大規模プログラムの開発やプログラミングスタイルの改善のためのさまざまな拡張もなされてきた．本論文では，Pascal コンパイラが Pascal を拡張するためのすぐれた道具になりうること，とくにモジュール化や抽象化というプログラミング方法論の規範の実現でそうであることを示し，そこで述べた拡張手法に基づいてモジュール化支援機能をもつよう Pascal を拡張した．拡張のため追加・変更された部分はわずかであるが，それによって得られる効果は大きい．

1. はじめに

Pascal[1] はもともと教育用ということで設計・開発されたが，Pascal のもつ多くの長所のため，教育用・実用を含め今日では広く使用されている．広範囲の使用，およびPascal 設計以降のプログラミング方法論の発展に伴い，Pascal に対するさまざまな批判・要求が生まれてきた[2),3)]．さらに，データ抽象や並行プロセス等の基本的概念の教育にも使用したいという要求，大規模プログラミングのための支援機能への要求，等も生じている．

Pascal への批判・要求とともに多くの拡張が提案され，そのいくつかは実際にインプリメントされてきた[4)]．一般に，ある言語を拡張するための実現手法としては，大きく，

（a） 処理系の新規作成，
（b） 既存処理系の一部変更，
（c） プリプロセサの作成，
（d） ライブラリの充実，

の四つがあげられる．これらは，追加できる機能の範囲，処理系の移植性，処理効率，拡張に要する労力，等の面でそれぞれ長短をもつ．Pascal の拡張の場合でも，これらの四つの手法の例を見ることができる（(a)～(d)に対する例としてそれぞれ5),6),7),8)）．

本論文ではまず，Pascal においては(b)の手法による拡張が，Pascal の一般利用者の立場からも十分実現可能であること，そして，新たな計算機構や計算能力をもたらす機能* というよりは，モジュール化や抽象化というプログラミング方法論上の有用な概念の記述を可能とする言語機能の導入には，とくに(b)の手法が適していることを述べる．次に，(b)の手法を応用して実現されたモジュール構造をもつ Pascal について，その実現法をも含めて述べる．

2. 既存処理系の一部変更による Pascal の拡張

初期の Pascal 処理系は，Pascal 設計者周辺で開発された Pascal P** や Trunk Pascal*** からの移植であった．その後開発された Pascal 処理系も，Pascal 自身で書かれ，処理系の構造や記述形態として十分洗練され，広く配布された上記処理系をベースとして，利用者自身を含むさまざまな人の手による変更・改善を受けてできあがってきたものが多い．たとえば，DEC System 10/20 上にある一つの Pascal システムは，Pascal P コンパイラを DEC System 10 のコードを生成するように変更した Hamburg 大学の版から，さらに Rutgers 大学で発展させたものである．

Pascal 処理系が多数の人の手による変更・改善を受けてくることができたのは，コンパイラ全体を含む処理系の多くの部分が Pascal 自身で書かれていたこと，再帰下降型 (recursive descent) 構文解析手法を用いているため構文とコンパイラ中の処理部分との対応がよいこと，等が挙げられる．

Pascal 処理系が理解しやすいものであるとはいえ，現実の処理系には，コード生成等の機械依存部分，エ

† A Technique for Extending Pascal and an Application to Pascal with Module Structures by YOSHIHISA MANO (Electrotechnical Laboratory, Computer Science Division, Langnage Processing Section).

†† 電子技術総合研究所ソフトウェア部言語処理研究室

* コルーチンやバックトラッキングというように，既存の言語機能で模擬はできてもプログラムの構造を大幅に変更しない限り実現できないような機能をここでは指す．

** 仮想スタックマシン用のコードを生成する．

*** コードを生成すべき箇所が，生成すべきコードについてのコメントとなっている．

ラー処理，最適化処理，等が多く含まれている．したがって，処理系の実際の変更には，処理方式や機械に関する十分な知識が要求される．

Pascal に欠けている機能としてしばしば列挙されるものとして，Pascal 設計時以降のプログラミング方法論の発展により明確化された諸概念の実現機能がある．すなわち，モジュール化，データ抽象，情報隠蔽，局所化という言葉で表されるものである[9]．

それらの機能は，しかし，利用者の記述方法ひいては思考方法の変革を求めるものではあるが，Pascal のもつ計算機構や計算能力を増加させるものではない．処理系内部の問題として捉えれば，これらは基本的にはコンパイラが名前の有効範囲をどのように定めるか，という問題に帰着される．

多くの Pascal コンパイラでは，探索木 (search tree) で表現される記号表が有効範囲ごとに存在し，それらは探索木の根へのポインタをもつ一つの配列* によって管理される．そして，名前の登録，探索のための手続きが用意されている．

こうして，上記機能を実現するには，原則的には次の手順に従ってコンパイラを変更することになる．すなわち，新たに追加された構文に対応する再帰下降型の構文解析部分を追加すること，参照できる記号表の範囲や名前を登録する記号表の選択を制御することで名前の有効・無効の決定制御を行うこと，必要なコードを生成すること，である．

構文解析部分の追加・変更は，構文図に対応するプログラム構造にすることで実現される（もちろんそれが可能である構文にしておかなければならない）．記号表を操作する手続きは，コンパイラの中の小さく閉じた部分であり，その部分の仕様を理解することは容易である．コード生成は，目的機械およびコンパイラのコード生成法に依存し，詳細を理解することが困難な部分であるが，前述のように追加機能は計算機構や計算能力の増加を伴わないため，既存のコード生成部分をそのまま利用できることが期待できる．すなわち，新たに追加される構文に対して生成すべきコードと同じコードを生成している箇所を見つけることによって実現でき，コード生成法に関する詳細な知識がなくともよい．

Pascal 処理系を直接変更する方式に対して，プリプロセッサを作ることで実現する方式もまた，実現の容易さ，移植性のよいものを作れること，の点で魅力的である．しかし，次の点では直接変更方式のほうがすぐれている．すなわち，語彙解析部，記号表操作部，等の既存ルーチンを活用でき，新たに重複して作る必要がないこと，コンパイル終了までの処理効率の悪化は無視できるほど小さいこと，プログラムエラーに対してソースプログラム中のエラー箇所とエラー種類とを対応させた指摘等の適切な処理を行えること，である．

また，新たに処理系を作成する方式は，本質的な改善を期待しうるが，他の方式に比べてはるかに大きな労力が必要である．ライブラリの充実という方式では，特定範囲の問題に対する強力な機能は提供しうるが，本論文で議論しているプログラム構造の改善という面にはそれほど貢献しえないであろう．

3章では，2章で述べた主張の実際の応用例として，2種類の Pascal 処理系に対してなされた，モジュール化支援機能の追加とその実現方法について述べる．

3. モジュール構造をもつ拡張 Pascal

モジュール化は，プログラミングにおいて最も基本的でかつ重要な概念の一つである，しかし，Pascal におけるモジュール化支援機能としては，入れ子構造のルーチン* が用意されているのみである．一つのまとまりをもった仕事をするための関連ある要素（ルーチン，変数，データ型，定数，等）を一つのグループにまとめ，そのなかでは互いに密な関係をもち，異なるグループ間ではできる限り互いに疎な関係をもつようにする，という意味でモジュール化を捉えれば，Pascal でそのような構造を作ることは不可能である．

言語仕様というよりはむしろ処理系の立場からモジュール化を支援するための機能として，データ型の一致に関するチェック機能を保存したままコンパイル単位を導入する研究もなされている[10),11)]．ここでは前章の議論の一つの応用として，既存処理系の変更による，言語仕様上のモジュール化支援機能の実現に絞って話を進める．

拡張の対象とされた Pascal 処理系は，最初 Trunk Pascal をもとにして作成された Tosbac 5600 用 Pascal であり，次いで DEC System 20 上の Hamburg 大学版 Pascal に対しても，ほぼ同じ拡張機能が実現された．ともにコンパイラのわずかな変更のみで実現されている．前者はアセンブリプログラムを出力し，後者は機械語を出力するコンパイラであるが，本章で

* DISPLAY と名づけられていることが多い．

* “手続きまたは関数” の代りに “ルーチン” という用語を用いる．

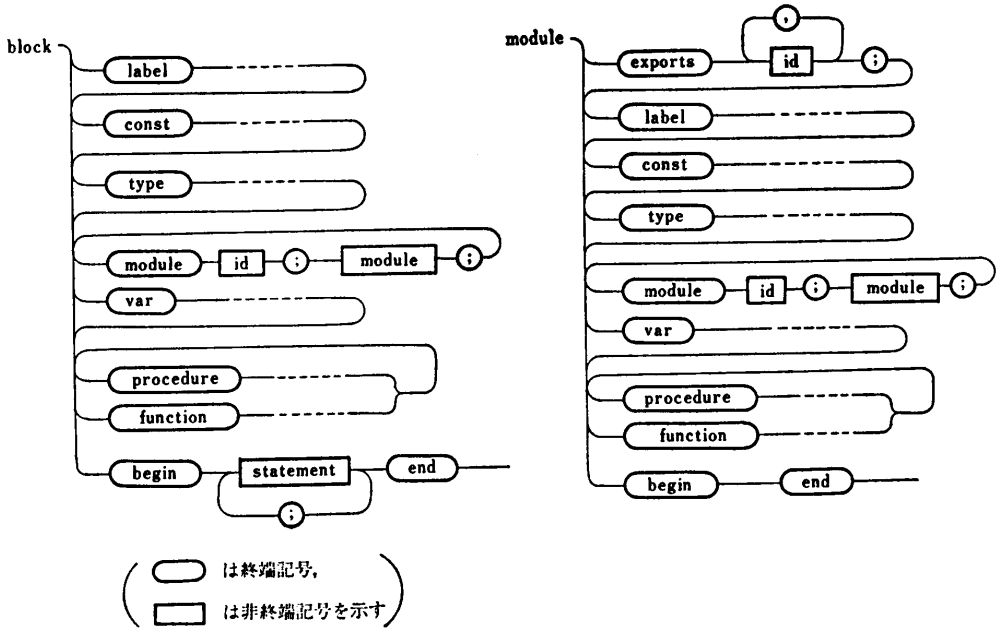


図 1 導入された module 構文のための構文図
Fig. 1 Syntax diagrams for appended modules.

述べる機能の実現にその違いは影響しない.

以下, 導入されたモジュール化支援機能とその実現方法について述べる. 実現方法は, Tosbac 5600 版も DEC System 20 版もほとんど変わらない. また, 2章で述べたように, 多くの Pascal コンパイラが類似の構造をもっているので, 多くの場合にここで述べる手法にわずかな変更を加えるのみで適用できるであろう.

3.1 module 構文の導入

ここで述べる module は, 言語仕様上の追加機能であり, いくつかの定数, データ型, 変数, ルーチンでもって, 一つのまとまりのある仕事をするプログラム部分があったとき, これらを一つの単位として記述することを支援する. それは, 一群の関連ある変数やルーチン等のまわりに, 名前の参照可能性を制限する特別の有効範囲をもつ枠を設ける機能であり, Modula-2[12] の module に似ている.

module の導入に関連して変更された構文部分を図 1 に示す. 図中の……部分, および図に示されていない部分は, 対応する Pascal の構文に一致する. この module は, 外部からの参照を許す名前を列挙する exports 文があること, module 本体の実行部分が空であること*を除いては, block の構文と同じである.

module はその module を含む最小のブロック (以降そのブロックを MinB と呼ぶ) にとってルーチン (およびデータ型) の定義の役割を果たすこと, また MinB に局所的な変数に対する module 内のルーチンの実行による副作用を避けることのため, module の定義は図 1 のように type 定義と var 宣言の間でなされるものとしている.

名前の参照可能性に関しては, module の外から内へは Pascal の通常の有効範囲規則を適用する. module の内から外に対しては, exports 文中で指定された名前のみが MinB で参照可能とする. 任意の種類の名前を参照可能と宣言できるようにすることも可能であるが, module と MinB とのインタフェースを最小にするという立場から, ここではルーチン名に限る. 3.2 節で, 抽象データ型の実現のため, データ型名をも許すように拡張する.

module に局所的な変数は, 制御が MinB に入ったとき生成され, MinB の実行が終了するまで最後に設定された値を保存する.

こうして module は, MinB とのインタフェースが指定されたルーチンの外部仕様のみという制限された形で, 一連の変数やルーチン等を記述する手段を与える. module に局所的な変数に注目すれば, MinB からはそれらの変数を直接参照・更新できないが, そ

* Modula-2 のように, 本体の実行部分を module に局所的な変数の初期設定用コードと解釈することもできる. Tosbac 5600 版の拡張 Pascal では, そのように実現された. ただしその場合はコード生成法に関する知識がいくらか必要となる.

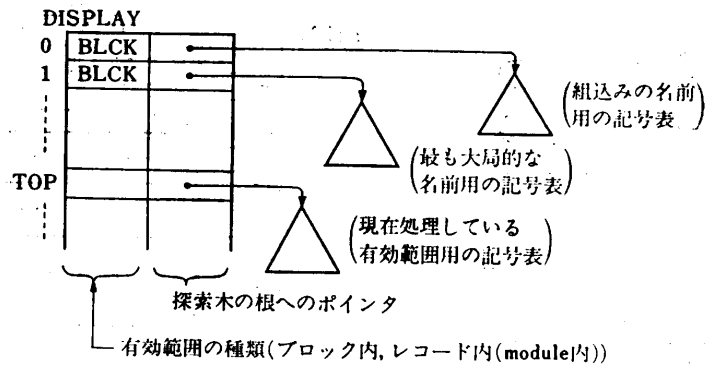


図 2 記号表の構造
Fig. 2 Structure of the symbol tables.

```
  ⋮
procedure ProcedureDeclaration (ProcFuncFlag:
  Boolean);
  begin
      ⋮
            モジュール内に定義されるルーチンで,
            exports 文中に現われていたものであれ
            ば, そのルーチン名を登録する記号表を
            MinB 用の記号表とする;
      ⋮
  end;
  ⋮
procedure ModuleDeclaration;
  procedure ExportDeclaration;
    begin
      ⋮
            export 文を構文解析し, export されて
            いる名前を記憶する
    end;
  begin
      ⋮
        module 名の処理をする;
        OLDTOP←TOP;
        TOP が上限を超えていなければ TOP←TOP
          +1 として新しい記号表を作り, 初期設定す
          る;
        次の記号が exports ならば ExportDeclara-
          tion を呼び出す;
        次の記号が label ならば LabelDeclaration
          を呼び出す;
            ⋮
        空の本体 (begin end) を読み飛ばす;
        exports 文中で指定された名前が module 内
          でルーチンとして定義されたか否かチェック
          する;
        TOP←OLDTOP;
    end;
  ⋮
```

図 3 module 構造導入のための処理概要
Fig. 3 Outline of processes for appended modules.

の値は MinB が活性化されている間保持され, 指定されたルーチンを介して操作できる. すなわち, データ抽象の一つの実現形態となっている.

次に, この module 構造をインプリメントする方法について述べる.

構文解析については, 構文図に従って追加・変更を行うが, その多くはブロック処理部分のコードをそのままコピーして使用すればよく, ほとんど問題はない.

Pascal コンパイラは, 名前の各有効範囲に対して, 探索木で表現される記号表を一つ作る. これらの記号表は, 根を指すポインタ等から成る配列 DISPLAY と現レベルを示す変数 TOP によって管理される (**図 2** 参照). 変数 TOP の増減による有効範囲の生成・消滅, TOP から 0 の方向への名前探索, 等によって, 有効範囲規則が実現されている.

コンパイラが module 構造の処理に入るとき, その module 内の名前専用の記号表を一つ作る. module 中で定義される名前のうち exports 文中で指定された名前は MinB を有効範囲とするため MinB 用の記号表に登録し, 他はその module 専用の記号表に登録する. こうして, module 構造の導入によって追加・変更すべき処理内容は, 概略**図 3** のようにすればよい.

3.2 抽象データ型の実現

抽象レベルの異なる抽象データ型を階層的に構築していくことによるプログラミング法は, 一つのすぐれたモジュール化手法として認識され, その場合の検証法も盛んに研究されている.

3.1 節で module が一種のデータ抽象の機能をもつと述べたが, ここでは module の拡張として抽象データ型を定義する機能の導入について述べる. それには, データ型名も exports 宣言によりモジュールの外から参照できるようにするが, そのデータ構造はまったく隠されてしまうようにすればよい. モジュールの外では, そのデータ型の変数の宣言や, 値全体の

```
module StringModule;
  exports STRING
    ,NEWSTR,STRLEN,EQSTR,CONCAT,STRHD,STRTL,MATCH;
  const AlfaLeng=10;
  type CELLSIZE=0..AlfaLeng; CELLPOS=1..AlfaLeng;
       STRING= ^STR;
       STR=record BODY:alfa;     NEXT:STRING;
                  SPOS:CELLPOS; LENG:CELLSIZE
           end;
  function NEWSTR(A:alfa; L:CELLSIZE): STRING;
    (* returns new string with A *)
    .....
  function STRLEN(S:STRING): integer;
    (* returns string length *)
    .....
  function EQSTR(S1,S2:STRING): Boolean;
    (* comparison of two strings *)
    .....
  function CONCAT(S1,S2:STRING): STRING;
    (*returns concatenated string *)
    .....
  function STRHD(S:STRING): char;
    (* returns a head character *)
    .....
  function STRTL(S:STRING): STRING;
    (* returns a tail string *)
    .....
  function MATCH(S1,S2:STRING): Boolean;
    (* true iff S2 is a substring of S1 *)
    module ArrayRep;
      (* array representation for efficient matching *)
      exports ARRAYMATCH;
      const MAXLENG=1000;
      type ARRAYSTR=array [1..MAXLENG] of char;
      procedure StoA(S:STRING; var A:ARRAYSTR);
        (* local STRING-to-array converter *)
        .....
      function ARRAYMATCH:Boolean;
        (* substring matching using array representation *)
        .....
    begin end; (*module ArrayRep*)
  begin (*function MATCH*)
    if      STRLEN(S1)>STRLEN(S2) then MATCH:=ARRAYMATCH
    else if STRLEN(S1)=STRLEN(S2) then MATCH:=EQSTR(S1,S2)
    else                               MATCH:=false
  end; (*function MATCH*)
begin end; (*module StringModule*)
```

図 4 抽象データ型 STRING を定義する StringModule の骨格
Fig. 4 Skeleton of StringModule for abstract data type STRING.

代入はできるが，構造の一部への参照，変更は不可能となる．こうして，そのデータ型の変数に対しては module 内に定義されるルーチンを介しての操作が余儀なくされる．

構文的には図1のままであるが，exports 文中にデータ型名も書ける点が拡張されている．そのデータ型は module の外の MinB 中では，抽象データ型として振舞う．すなわち，そのデータ型は構造をもたないデータ型で，他のどのデータ型とも一致しない* ものとして扱われる．

図4は，「文字列」のための抽象データ型を定義する module の骨格であり，module の使用例，抽象データ型の定義例を示している．

この拡張は次のようにインプリメントされる．外から参照可能とするデータ型名は，ルーチン名の場合と同様に，MinB 用の記号表に登録される．その module のコンパイル終了時に，コンパイラにより記憶されているそのデータ型の内部表現のうちデータ構造の種類**を「抽象データ型」とする．また，データ型の一致を調べるルーチンを，「抽象データ型」について

* データ型宣言でそのデータ型と等しいとして宣言されたデータ型（すなわちたんに別名をつけられたデータ型）を除く．また，Pascal におけるデータ型の一致には，「構造による一致」と「名前による一致」とがあり，言語仕様上はどちらとも定められていない[9]．ここで対象とした処理系はいずれも構造による一致を採用しているので，ここでもその立場をとる．なお，名前による一致を採用している処理系ではデータ型の構造の比較を含まないのでより容易に実現できるであろう．

** スカラー，配列，ポインタ等の種類を表すもの．

```
  :
  :
type ElementP = ^Element;
     Element =
       record
        Data1: ...; Data2: ...;   } (a) 本来の仕事に直接関連するデータ項目
        RelationP:ElementP; ...   } (b) 構造を表現するためのデータ項目
       end;
  :
module DataManagement;
 exports Init,FindData,Create,Delete,...;
    :
 procedure Init(...);                                  } 構造を管理するモジュール
    :
 function FindData(...): ElementP;
    :
    :
 begin
  (*$H+ (Element)RelationP,...; ... *) (c)
 end; (*end-of-DataManagement*)
  :
begin
 Init(...);
  :
 with FindData(...)^ do
  begin ...:=Data1; Data2:=...; ... end; } (d)
  :
  :
```

図 5 ヒープ領域を使用するプログラム例の骨格
Fig. 5 Skeleton of an example program which uses the heap area.

はたとえ内部構造が同一でも他のどの型とも一致しないように変更する*.

この変更により,「抽象データ型」の内部表現を仮定した操作は不可能となる. なぜなら, 内部表現へのアクセス操作 ([ ], ↑, .) や演算 (+, in 等) は,「抽象データ型」に対しては定義されていず, コンパイラが拒否できるからである.

3.3 ヒープ領域の管理と使用に関する支援

実行時に動的に構造が変化するデータを Pascal で扱う場合, 通常ヒープ領域を利用する. ヒープ領域は一種の大域変数領域であり, 不注意による思わぬエラーが生じやすく, その場合大域的かつ動的なデータであるため, エラーの原因を見つけにくい.

その一つの原因として, 動的データの構造を管理するためのデータ部分と, 与えられた問題の解決に必要なデータ部分とが混在していることを挙げることができる. そこでヒープ領域を扱うプログラムスタイルとして, 構造管理のためのデータ部分と問題解決に直接関連するデータ部分とを分離することが, プログラムの信頼性を高める手法として考慮されるべきである. これら二つのデータ部分を分離し, それぞれを専門に扱うモジュールとして実現することを支援するには, 前節までの module 機能を利用することができる. すなわち両者のインタフェースとして, module 内のルーチンに限り, データの内部表現へのアクセスは管理部分に限る方法である.

ここでは, つねにルーチンを介することによる効率低下を避けたい場合や, さまざまなアクセス法があるためにそれぞれにルーチンを用意することが困難な場合に適用できる一つの方法を提案し, そのための現実的な支援機能を示す*.

その方法による典型的な記述例を図 5 に示す. その記述方針は,

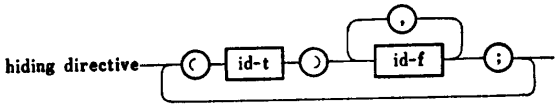
(a) データ構造を維持するための管理部分を他の部分から独立させる,
(b) 個々のデータの内容も, 構造を表現するためのポインタ等の項目と他の項目とを区別する,
(c) データの構造を表現するための項目へは管理部分内からのみアクセスする,
(d) 管理部分以外からのデータへのアクセスは, データへのポインタを値とする関数を介して所望するデータの取出しを行い, データ内の個々の要素へは要素名を指定して直接アクセスする,

である.

これにより, データの構造の実現部分と, 本来の仕事に関連したデータ項目の利用部分とが論理的に分離される. ヒープ領域をとくにデータベース的に使用す

* ただし type id1=id2; という形 (id1, id2 は識別子) に関しては, 処理系の作成手法上この方法で抽象データ化することは容易でない. ここではそのようなデータ型の抽象データ化は許していない.

* ここで述べる機能は Tosbac 5600 版では実現されていない.



ここで id-t：データ型名
id-f：フィールド名

図 6 データ項目名隠蔽指令の構文
Fig. 6 Syntax of directives for hiding data field names.

る場合，そのデータ抽象化の機能としてプログラムの信頼性・理解性等を高めてくれるだろう．

(d)にあるように，データを取り出すためポインタを値とする関数が多用されることになるが，f をそのような関数としたとき，$f(\cdots)\uparrow$ というように，関数呼出しの直後にポインタ値の指すデータへのアクセスをする演算子↑を付ける形は Pascal では許されていない．図5(d)はこれを許すように拡張された機能であり，関数の値を作業用変数に代入する等の冗長な操作を省いており，自然な記述を可能としている．この機能のインプリメントは，ポインタを値とする関数の場合に限り，関数呼出しの処理を with 節中にも含ませることによる．

図5(a)(b)にあるように，(b)の方針に従いデータ中の項目を，本来の仕事に直接関連する項目と構造を表現するための項目とに分けるのであるが，(c)の方針を厳密に守るため処理系によるチェック機能が望ましい．これはデータの構造中の一部要素名が適当な時点でコンパイラにとって隠されてしまうように，コンパイラに指示するという形で導入できる．

分離コンパイル機能がある場合は，幾分トリッキーであるが容易に実現されうる[13]．しかし，図5(c)にあるように module のコンパイル終了直前のコンパイラへの指令による実現方法も困難でない．このデータ型中の項目名隠蔽指令は，module の空の本体の中で構文上は注釈であるコンパイラオプションの形((* $H+ で始まる) で与える．オプション内の構文は図6 に示される．

この機能は，指定されたデータ型中の項目名に対して，それ以降その名前に対する探索が失敗するように，適当な変更（たとえば，名前情報を空白列に変更する）を加えることで実現できる．

なお，この隠蔽指令は，含まれるデータ項目の役割を2種類に区別しやすいヒープ領域中に作られるデータに対してとくに効果をもたらすが，一般のレコード型データ型に対しても適用できる．

4. おわりに

現在 Pascal は標準化の動きが盛んであるが，つねに言語の改善を求める努力は必要であり，標準化が成立した以降も同様である．本稿では容易な実現による利点を重視する立場から，Pascal に対するさまざまな拡張要求のうちプログラミング方法論に関連するある種の拡張とその実現手法について検討した．それらは Pascal 処理系のわずかな変更によって実現できることを示し，実際に2種類の Pascal 処理系に対しモジュール化支援機能を導入した．追加・変更された部分はコンパイラ内に限られ，元のコンパイラの大きさに比べてきわめてわずかである（行数で全体の 1.9～3.8%）* にもかかわらず，それによって得られるプログラム構造上の改善効果は大きい．

Pascal のある種の拡張の容易さが示されたが，この手法をここで述べた拡張機能やその他の機能に応用することによって，

(a) プログラミング教育の場でプログラミング方法論上の種々の概念を実働言語に基づいて教えることが，Ada のような大規模言語を導入しなくとも可能となる，

(b) ある種の機能を言語内に組み込んだことによる効果を実験的に確かめることを可能とする，

(c) 大規模プログラムを，システムによってチェックされるある種の規範に基づいて記述しようとする人々を支援することができる，

等のことが期待できる．

参考文献

1) Jensen, K. and Wirth, N.: Pascal—User Manual and Report, *Lecture Notes in Computer Science*, 18, Springer Verlag (1974).
2) Habermann, A. N.: Critical Comments on the Programming Language Pascal, *Acta Informatica*, Vol. 3, pp. 47-57 (1973).
3) Welsh, J., Sneeringer, W. J. and Hoare, C. A. R.: Ambiguities and Insecurities in Pascal, *Softw. Pract. Exper.* Vol. 7, No. 6, pp. 685-696 (1977).
4) Turba, T. N. and Costello, S. H.: A Survey

* Tosbac 5600 においては，もとの Pascal コンパイラ 4779 行に対し，module 構文・抽象データ型の実現のため追加・変更された行数が 183 (module 本体の実行部分を初期設定部分とみなす機能の実現も含む)，DEC システム 20 においては，もとの Pascal コンパイラ 11939 行に対し，module 構文・抽象データ型の実現のために追加・変更された行数が 177，ヒープ領域に関する支援機能の実現のために追加・変更された行数が 78 である．

of Currently Implemented Pascal Extensions, NCC 81, pp. 217-224 (1981).
5) Overgaard, M.: UCSD Pascal: A Portable Software Environment for Small Computers, NCC80, pp. 747-754 (1980).
6) Tsin, Y. H.: Extending the Power of Pascal's External Procedure Mechanism. *Softw. Pract. Exper.*, Vol. 12, No. 3, pp. 283-292 (1982).
7) Young, S. J.: Improving the Structure of Large Pascal Programs, *Softw. Pract. Exper.* Vol. 11, No. 9, pp. 913-927 (1981).
8) Kriz, J. and Sandmayr, H.: Extension of Pascal by Coroutines and Its Application to Quasi-Parallel Programming and Simulation, *Softw. Pract. Exper.*, Vol. 10, No. 10, pp. 773-789 (1980).
9) 鳥居，二木，真野：プログラミング方法論の展望，情報処理，Vol. 20, No. 1, pp. 22-43 (1979).
10) Kieburtz, R. B., Barabash, W. and Hill, C. R.: A Type Checking Program Linkage System for Pascal, 3rd Conf. on Software Engineering, pp. 23-28 (1978).
11) 鍵政，荒木，都倉：Pascal 内部手続きの分離翻訳，電子通信学会論文誌 (D)，J 63-D, pp. 177-182 (1980).
12) Wirth, N.: MODULA-2, ETH, Institut für Informatik, 36 (1980).
13) 真野：碁局面の表現と Pascal における複雑なデータ構造管理，情報処理学会ソフトウェア工学研究会，20-3 (1981).
14) 真野：プログラミング方法論に基づく拡張 Pascal —その設計と実現—，電子通信学会技術報告，EC 80-30 (1980).

(昭和 58 年 3 月 1 日受付)
(昭和 59 年 5 月 15 日採録)