ソフトウェア工学 20－3
（1981. 9. 21）

碁局面の表現とPascalにおける複雑なデータ構造管理

真野 芳久
（電子技術総合研究所）

1. はじめに

我々は，人間の思考方法をモデル化した局面認識に基づき，大局的な戦略とプランを持った碁プログラムの開発を試みている[1]。碁はチェスと比べてもその複雑度ははるかに大きく，チェスにおいてかなりの成果を収めてきた高速ゲーム木探索中心の手法を用いることができない。現時点では，前述のようなアプローチが最善と考えられる。

碁局面をいかに表現しておくかは，このプロジェクトを進める上で発生した最初の問題の1つであり，また，その後のプログラム開発の能率，プログラムの効率・理解性，等に大きな影響を与える決定事項であった。

文献[2]で述べられている局面データ構造は，石のハガシを行なうunplayを可能としているが，碁規則のプログラム化にとどまっている。

Interim.2[3]は現存する最強の碁プログラムと思われる。そこでは，我々の立場と同様に，人間プレイヤーの認識法をモデル化した種々の視覚情報を扱っている。例えば，予め定められているパターンが出現した場合に，関連ある情報とともに作られるレンズ，ある"もの"の周辺を放射状に調べることを目的としたウェブがある。その構造等の詳細は不明である。

本稿ではまず，やはり人間プレイヤーの認識法をモデル化した，簡潔でしかし強力な局面表現方法について述べる。次に，現実的要請を考慮に入れて，Pascalによるその実現について述べる。最後に，プログラムの作り易さ，理解性，信頼性を高めるために導入されたデータ抽象のための1つの手法について述べる。

2. 局面表現のための概念

碁局面を表現する自然な極小集合としては，

- 棋譜，
- 19×19の盤上の石の配置，アゲ石の数，次の手番，劫による禁止点，の組，

のようなものが考えられる。あらゆる局面情報はある極小集合から得られるのであるが，高度な判断（大局観，プラン，等）を行なうには基本的すぎる。人間の場合には，様々なレベルの概念をうまく組み合わせて効率良く的確に局面を認識しているようである。

人間の持つそのような概念を，我々は次のような階層を持つ諸概念とそれらの間の関係へとモデル化した。

点(point)：19×19の盤上の点，
連(string)：縦又は横で直接連結している同色の石の置かれている点の集合，
群(group)：連結しているものとみなすことのできる同色の連の集合，
族(family)：連結する可能性のある同色の群の集合，
結線(linkage)：同色の石の置かれている2点間，又は石の置かれている点と辺上の点との関係で，連結する可能性の十分高いもの，
弱結線(loose linkage)：同色の石の置かれている2点間，又は石の置かれている点と辺上の点との関係で，連結する可能性がある程度あるもの，
連（群，族）間の関係（string (group, family) relation）：異色の2つの連（群，族）の間に存在する敵対関係，又は同色の2つの連（群，族）の間に存在する協力関係．

実際の局面でこれらの諸概念の実現された"もの"を対象物(object)と呼ぶことにする。対象物は局面を表現するための基本要素である。

なお，これらの概念は主として次のような使われ方をされるものと考えている。

連：同時に活き死にする基本単位，
群：死活を考慮するときの単位，
族：プラン構成のための概念，
結線：確定地を構成するための要素，
弱結線：模様を構成するための要素，
連間の関係：連結・包囲のための直接的関係，
群間の関係：連結の可能性，攻め合いの関係，
族間の関係：戦いの発端としての境界領域，死活度の相対的関係によるプラン構成．

3. 局面表現の基本操作

点，連は前節の説明で完全に定義されているが，他の諸概念はそうでない。実際にはプログラムのコードが定義文となり，プログラムの棋力の向上に応じて意味する実体が変化することとなる。これは人間の棋力向上の場合にもあてはまることである。

こうして，各概念に現時点でインプリメント可能な定義を与えるよりは，変更・拡張の可能な形で各概念を操作できるようにしておくことの方が将来の棋力向上のために重要なことである。これらの操作は，複数のプロジェクトメンバ間での共通の用語として，時間の経過による記憶の変化に耐えるため，形式的に定めておくことが望ましい。

各対象物は，階層構造等の構造を示すためのデータや，局面を解析して得られる様々な評価値を，属性として持っているものと考えることができる。これらは図1に示される。ここで，
$\mathbb{C}$：黒側，白側の集合，$\mathbb{F}$：族の集合，

```
C∈ℂ   C = ({F1,...}, {L1,...}, {LL1,...})
        Fi∈𝔽 , Li∈𝕃 , LLi∈𝕃𝕃
      Attr(C) = {Colour: (Black,White);
                 PotentialTerritorySize: integer,
                 TerritorySize: integer,  ....  }
F∈𝔽   F = ({G1,...}, {FR1,...})
        Gi∈𝔾 , FRi∈𝔽ℝ
      Attr(F) = {Colour: (Black,White);
                 TerritorySize: integer,
                 Vitality: real,  .... }
G∈𝔾   G = ({S1,...}, {GR1,...})
        Si∈𝕊 , GRi∈𝔾ℝ
      Attr(G) = {Colour: (Black,White);
                 Vitality: real, .... }
S∈𝕊   S = ({P1,...}, {SR1,...})
        Pi∈ℙ , SRi∈𝕊ℝ
      Attr(S) = {Colour: (Black,White);
                 DameNumber: integer, .... }
P∈ℙ   P = ( )
      Attr(P) = {Coordinate: integer X integer,
                 State: (Black,White,Empty);
                 .... }
L∈𝕃   L = ({P1, P2})
        Pi∈ℙ
      Attr(L) = {Colour: (Black,White);
                 Strength: real, .... }
LL∈𝕃𝕃  LL = ({P1, P2})
        Pi∈ℙ
      Attr(LL) = {Colour: (Black,White);
                  Strength: real, .... }
FR∈𝔽ℝ  FR = ({F1, F2})
        Fi∈𝔽
      Attr(FR) = { ;
                  Contacts: set of ℙ, .... }
GR∈𝔾ℝ  GR = ({G1, G2})
        Gi∈𝔾
      Attr(GR) = { ;
                  Connectivity: real, .... }
SR∈𝕊ℝ  SR = ({S1, S2})
        Si∈𝕊
      Attr(SR) = { ;
                  CommonDame: integer, .... }
```

図1. 対象物の構造と属性

$\mathbb{G}$：群の集合，$\mathbb{S}$：連の集合，$\mathbb{L}$：結線の集合，$\mathbb{LL}$：弱結線の集合，$\mathbb{FR}$（$\mathbb{GR}$，$\mathbb{SR}$）：族（群，連）間の関係，である。また，構造を示すための属性は，理解性を増すため，まとめて組として表現してある。

局面を評価して得られる属性値（図1の $Attr(\cdots)=\{\cdots;\cdots\}$ の；以降）は，棋力に応じて変化する部分で，ここではこれ以上扱わない。

構造を示すための属性を各対象物の内部状態と考えて，対象物の生成・消滅・併合等の操作を内部状態の変化として記述することができる。Parnasの状態機械モデルに基づく仕様記述法[4]を参考にしてそれらの操作を記述したのが図2である。ここで，ϕ：空集合，$X|_i$ は対象物Xを表現しているn組の第i成分（$1\leq i\leq n$）で，$Projection\,of\,X(X)$ というような関数の略記であり，$X|_i'$ は操作後のそ

図2. 状態機械モデルによる局面表現の基本操作群の記述

```
object
 C, Cx are elements of ℂ,
 F, Fx are elements of 𝔽,
 G, Gx are elements of 𝔾,
 S, Sx are elements of 𝕊,
 P, Px are elements of ℙ,
 L, Lx are elements of 𝕃,
 LL, LLx are elements of 𝕃𝕃,
 FR, FRx are elements of 𝔽ℝ,
 GR, GRx are elements of 𝔾ℝ,
 SR, SRx are elements of 𝕊ℝ,


operation Initialize ( ) --->
                        {Cb, Cw, P0, P1, ..., P360 }
 effect
  Cb = NewColour(), Cw = NewColour(),
  Pi = NewPoint()  for each i in 0..360,
  Cb' = (φ, φ, φ), Val'(Cb, Colour) = Black,
  Cw' = (φ, φ, φ), Val'(Cw, Colour) = White,
  Val'(Pi, Coordinate)=(i div 19 + 1,i mod 19 + 1),
  Val'(Pi, State) = Empty  for each i in 0..360.


operation CreateFamily ( C ) ---> F
 effect
  F = NewFamily(),
  F' = (φ,φ),
  C⌋1' = C⌋1 + { F },
  Val'(F, Colour) = Val(C, Colour).

operation Families ( C ) ---> { F1, ... }
 value C⌋1.

operation PurgeFamily ( F ) --->
 let C suchthat C⌋1 ∋ F
 effect
  C⌋1' = C⌋1 - { F }
 assume
  F = (φ,φ).

operation MergeFamily ( F1, F2 ) ---> F
 effect
  F3 = NewFamily(),
  F3' = (F1⌋1 + F2⌋1,
         F1⌋2 U F2⌋2 - {FR | FR⌋1 = {F1, F2}}),
  for each FR ∈ F3⌋2'
    if FR⌋1 = {F1, Fx} then FR⌋1' = {F3, Fx},
    if FR   = {F2, Fx} then FR⌋1' = {F3, Fx},
  C⌋1' = C⌋1 - {F1, F2} + { F3 }.

operation Edge ( P ) ---> b: Boolean
 value
  b = true if P is a point on the edge,
      false otherwise.

operation CreateLinkage ( C, P1, P2 ) ---> L
 effect
  L = NewLinkage(),
  L' = ({P1, P2 }),
  C⌋2' = C⌋2 + { L },
  Val'(L, Colour) = Val(C, Colour)
 assume
  (Val(P1,State)=Val(P2,State)=Val(C,Colour)) or
  (Val(P1,State)=Val(C ,Colour) and Edge(P2)) or
  (Val(P2,State)=Val(C ,Colour) and Edge(P1)).

operation Linkages ( C ) ---> {L1, ... }
 value C⌋2.

operation PurgeLinkage ( L ) --->
 let C suchthat C⌋2 ∋ L
 effect
  C⌋2' = C⌋2 - { L }.

operation CreateFamilyRelation ( F1, F2 ) ---> FR
 effect
  FR = NewFamilyRelation(),
  FR' = ({F1, F2 }),
  F1⌋2' = F1⌋2 + { FR },
  F2⌋2' = F2⌋2 + { FR }.

operation FamilyRelations ( F ) ---> {FR1, ... }
 value F⌋2.

operation PurgeFamilyRelation ( FR ) --->
 let FR⌋1 = {F1, F2}
 effect
  F1⌋2' = F1⌋2 - { FR },
  F2⌋2' = F2⌋2 - { FR }.


operation ExcludeGroup ( G ) --->
 let F suchthat F⌋1 ∋ G
 effect
  F⌋1' = F⌋1 - { G }.

operation ExcludeString ( S ) --->
 let G suchthat G⌋1 ∋ S
 effect
  G⌋1' = G⌋1 - { S }.


operation IncludeGroup ( F, G ) --->
 effect
  F⌋1' = F⌋1 U { G }
 assume
  for each Fi  Fi⌋1 ∌ G,
  Val(F, Colour) = Val(G, Colour).

operation IncludeString ( G, S ) --->
 effect
  G⌋1' = G⌋1 U { S }
 assume
  for each Gi  Gi⌋1 ∌ S,
  Val(G, Colour) = Val(S, Colour).

operation IncludePoint ( S, P ) --->
 effect
  S⌋1' = S⌋1 U { P }
 assume
  Val(S, Colour) = Val(P, State).
```

の値である。$X' = (x_1, \cdots, x_n)$ は、$X⌋_1' = x_1, \cdots, X⌋_n' = x_n$ の略記である。$Val(X, Attr)$ は対象物 X の Attr 属性値で、$Attr\,of\,X\,(X)$ というような関数の略記であり、$Val'(X, Attr)$ は操作後のその値である。assume 部分は操作が適用可能であるための条件を示す。

4. Pascal による局面データ構造の実現

我々は Pascal を用いて碁プログラムを開発している。Pascal を選んだ理由は、第1に、対象物や対象物間の関係の複雑な構造を、その豊富なデータ構造化機構により自然かつ効率良く表現できるであろうこと、第2に、理解性の良いプログラムを作り易いこと、第3に、Pascal を使い馴れ、その処理系の構造もある程度

理解していること，による。

前節で述べた操作群の実現のため，詳細なデータ構造を次の諸点を考慮しながら定めた。

(i) 新たな（局面評価により得られる）属性の追加が容易であること，

(ii) 対象物の動的な生成・消滅に対処できること，

(iii) 大量かつ複雑なデータが必要となろうが，時間的・空間的に効率良いものであること，

(iv) 先読み時の仮想的な局面に対しての(iii)で挙げた点．基本的には，データのコピー（メモリ量，コピー時間の点で問題あり），データの共有（戻る時の逆変換も可能であるようにすること，及び逆変換のための時間の点で問題あり）の2つがある。

(v) Pascal にはないごみ集め（garbage collection）機構を備えること。

これらの要請に基づいて，我々は次のような形でデータ構造を構成した。

(I) 各対象物には固有の識別子を与える，

(II) 対象物の属性はヒープ上にとられるレコード中に置く（(i)(ii)），

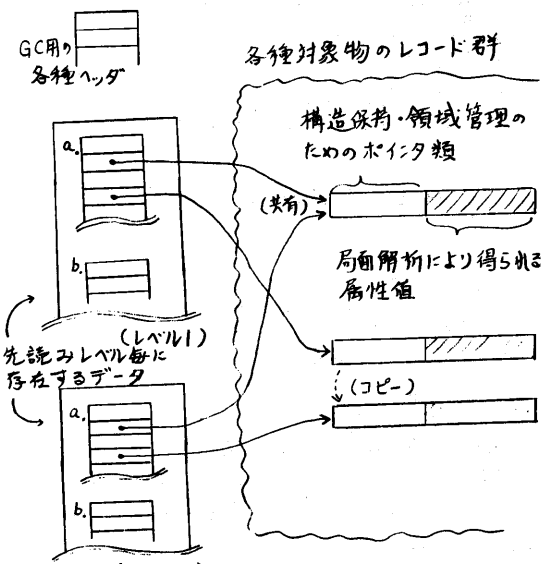
(III) 対象物すべてを管理する配列を用意する（対象物の識別子はその配列への添字として使われる）．その値はヒープ上のレコードへのポインタとする，

(IV) レコード中に置く属性としては，局面を解析して得られる値（(i)），階層等の構造を表現し関連ある対象物を結ぶためのもの（(ii)(iii)），領域の自主管理用のポインタ（(v)），先読み時に作られる仮想局面用のデータを結ぶためのポインタ（(iv)），とする，

(V) 前述の(iv)の問題点を緩和するため，先読み時には対象物管理用及び他の僅かな配列のみをまずコピーし，対象物自体のレコードは可能な限り共有する方式をとった。即ちレコード中のある属性に変更が生じたときのみ，レコードの領域の確保・内容のコピー・配列中の対応するポインタの変更，を行なう。この方式は無駄なメモリ使用，コピー時間が少なく，先読みから戻る時は管理用の配列を1レベル前のものに戻すのみでよい（(iv)）．ただし，この方式を可能とするため，他の対象物への参照を目的とする属性は（領域管理用のポインタを除いて）識別子を経由する間接的な方法としなければならない。

このデータ構造の概略は図3のように図示される。

各対象物に与えられている属性のうち，局面を解析して得られる'属性値'は，プログラムが各対象物をどう捉えているかを表わし，その種類，値はプログラムの棋力に影響を与える．



a.：対象物の識別子とレコードとの対応を示す配列

b.：先読み時の仮想データのためのGC用ヘッダ

図3. データ構造説明図

一方，各対象物に与えられている'ポインタ類'は，着世界を階層的な概念と概念間の関係として捉えるこのプログラムの基本的立場を実現し，領域管理のための補助情報を与えるものである。その構成方法はプログラムの効率に影響を与える。図3には明示されていないが，これらのポインタ類によって作られるリスト構造の集まりは，極めて複雑なものとなっている。

このように異なる2種類のデータ成分が各対象物に与えられ，それぞれは独立にその構成法を決定し変更することができる。しかしある種の対象物の属性値を見ようとする場合，そこへのアクセスのためには様々なポインタ類を参照する必要が生ずる。これは属性値を参照・更新する側から見れば，本質的な操作ではなく，また，効率改善等による構成方法の変更の場合も直ちに影響を受けることになる。

こうして，本質面にのみ着目でき，インプリメント方法による影響から逃れるために，ポインタ類参照部分の抽象化が望まれる。

5. Pascalにおけるデータ抽象

データとそれを扱う手続き（関数も含む，以後も同様）とを同一箇所で定義し，データへのアクセスは手続き群を介する場合に限る，というのがデータ抽象のための基本的手法である。その重要性が認識され，最近の言語では抽象データ型を定義できる機能を持つものがいくつか発表されている[5,6]。

しかしPascalにおいては，その有効範囲規則のために，同一のデータにアクセスが可能な複数個の手続きを定義し，他箇所からはデータへの参照は禁止され手続き群への参照は可能であるような構造を作ることができない。

6. データ抽象のためのPascalシステムの拡張

実用上の理由から分割コンパイル可能なPascal処理系も多いが，その場合，

- 各コンパイル単位で大域的に宣言される変数には固有の領域が割り当てられる，
- 各コンパイル単位は大域的に定義される複数個の手続きを含むことができて，その中のいくつかを他のコンパイル単位から呼び出せるものとして指定できる，

の条件を満たせば，データ抽象を行なうことが可能となる。外部から参照可能な手続き群のみが，他から見た場合，大域的に宣言された変数群にアクセス可能だからである。

このような分割コンパイル機能は，処理系を拡張することで比較的容易に得られることも確かめられている[7]。

我々のプログラムの場合，図3の斜線部（属性値）と斜線部以外（ポインタ類）とは互いに独立であり，属性値を利用する側からはポインタ類の存在・形式に関知しなくてもすむのが望ましい。

そのためにまず，我々のPascal処理系が前述の条件を満たすことを利用して，ポインタ類へのアクセスを必要とする操作（生成・消滅操作，対象物のデータ領域へのポインタを与える操作，等）は，すべて1つのコンパイル単位（管理モジュールと呼ぶ）内に手続き群としてまとめ，他はこれを利用する，ことにする。

一方，属性値へのアクセスは，各属性値の構造が単純であること，及び属性値集合はしばしば変更されるであろうことから，そのレコードへのポインタを用いて直接行なうことが望ましい。

ポインタ類を抽象化し，そこへのアクセスを不必要かつ不可能としたいというのが目標であった。しかしこれはレコードへのポインタを持つということとは相

反する。

この問題は，

管理モジュール内に記されるデータ型定義を他のそれにポインタ類を加えたものとするという，分割コンパイルであることを利用した一種のトリック

によってかなり解決される。即ち，管理モジュールのコンパイル時には，各対象物のデータ型の定義を完全に利用するが，その他のモジュールのコンパイル時には，属性値に関する成分のみを使用するのである。データ型の定義中にないポインタ類成分へのアクセスはコンパイル時にチェックされエラーとなることを利用するのである*。

これを支援するため，共通環境ファイル，及び共通環境ファイル内での選択的コンパイル機能を処理系に持たせることにした。

共通環境ファイルには，全モジュールで共通に使用される定数名，データ型名，外部手続き名の定義・宣言が置かれる。各モジュールのコンパイル時に必要に応じてこのファイルが参照される。

共通環境ファイル内での選択的コンパイル機能は，共通環境ファイルの参照の方法を2種類以上設け，ある方法で参照した場合と他の方法で参照した場合とで異なる見え方をさせるものである。これにより，管理モジュールはデータ型定義の全体を知るが，他のモジュールではそうでないようにできる。具体的な方法は次節で述べる。

このプログラムでは，管理モジュールが局面データベースの役割を果たし，他のモジュールは管理モジュールからレコードへのポインタを受け取り，必要な部分にアクセスする。従って，管理モジュールの多くの手続き・関数がポインタを返し，これらの関数は他のモジュールで多用されることになる。これは1つの問題を発生させた。

fをポインタ型の値を持つ関数としたとき，f(…)↑ という形はPascalでは許されていない。そのため同じポインタ型の局所変数を導入し，関数の値をその局所変数に代入するという表現を強いられることになる。これはしばしばプログラムの自然さ，理解性を損ねることになる。

我々のプログラムのような構造の場合，f(…)↑は変数と考えて何ら差支えがない。また，たとえ同一のコンパイル単位内であってもこの種のデータベースをヒープ領域に構成し，ポインタを返す手続き・関数群によって抽象化を目指すことはむしろ推奨される方法であろう。

我々は，f(…)↑も変数の一表現法として扱えるように処理系を変更し，不必要なコードの挿入なく自然な表現で記述できるようにした。

7. プログラムの構造

前節で述べた機能を持つPascal処理系を用いて碁プログラムを開発しているが，そのプログラム構造の概略は図4のようになる。

コンパイルオプション〝(*$I x*)〟は共通環境ファイルの利用に関するものである。x=+ の場合は，共通環境ファイルの（それまでに読み込んだ部分の直後への）呼出し，x=- の場合は，共通環境ファイルからもとのソースプログラムへの復帰を示す。x=% の場合は，ある種の権限を持っての共通環境ファイルの呼出しで，(*% ---- %*) なる形の注釈を注釈とはみなさず，----- 部分を共通環境の一部とみなしてしまうものである。

* ただし定義されている属性値成分へのアクセスが正常に行なわれることは，処理系の処理方式を調べて確認しておかねばならない。

（主モジュール）

```
program Go;
const (*$J+*)
 :
type  (*$J+*)
 :
var
 :
(*$J+*)
begin
 :
end.
```

（管理モジュール）

```
program MemoryManager,
 CreateString,CreateGroup,...
 FirstString,NextString,DataOfString,...
 ...;
const (*$J%*)
 MaxNest= 20;
 :
type  (*$J%*)
 :
var
 CURNEST: 0..MAXNEST; (*Look-Ahead nesting level*)
 (*Head pointers for free area*)
 SPHEAD: StringPointer;
 :
procedure CreateString(var SP:StringPointer; ...);
 :
function DataOfString(...): StringPointer;
 :
begin end.
```

（共通環境）

```
(*const*)
 BoardSize=19;
  :
 (*$J-*)
(*type*)
 IdType= 1..MaxId;
  StringId= IdType;
   :
 StringPointer=^StringRec;
 StringRec= record
             StoneNum: 0..511;
             Dame    : 0..511;
             Vitality: real;
              :
         (*% NextString: StringId;
             StringRelationHead: StringRelationId;
             Nest: 0..MaxNest;
             GcNextp: StringPointer;
              :                    %*)
            end;
  :
 (*$J-*)
procedure CreateString(...); extern;
 :
 (*$J-*)
```

（その他のモジュール）

```
program DataBaseUpdate,
 StringUpdate,GroupUpdate,...;
const (*$J+*)
 :
type  (*$J+*)
 :
(*$J+*)
procedure StringUpdate(...);
 begin
   :
  with DataOfString(...)↑ do
   begin ... end;                  } (a)
   :
 end;
 :
begin end.
```

図4. プログラム構造概略

図4ではポインタ類に関する定義部分が（*%，%*）によって囲まれており，（*$J% *）によって共通環境ファイルを利用しているのは管理モジュールのみである。こうして共通の共通環境ファイルを用いながら，管理モジュール以外ではポインタ類が隠されている。他のモジュールは，管理モジュール内に用意されている手続き群を利用することで，複雑なデータ構造を間接的に操作する。

なお，属性値へのアクセスの多くは，図4の(a)に見られるような自然な形で行なっている。

8. おわりに

ここではまず，碁プログラムを作成していくための土台となる局面表現を形式的に定めた。階層性を持った概念的には比較的単純なものであるが，この上に様々な属性を追加していくことにより表現力は豊富となる。

我々は実際にこの局面表現をPascalを用いてインプリメントし，その上に碁プログラムを開発しつつある。データ構造は複雑なものであるので，プログラミングの能率，プログラムの信頼性・理解性を高めるため，分割コンパイル機能を利用し，更に敢えてPascal処理系に僅かな変更を加えた。

Pascalは大規模プログラミングに不向きであると言われるが，ここで述べたような方法によりかなり満足できるモジュール化，データ抽象ができることが確かめられた。

終筆ながら討論して頂いた碁研究グループの諸氏に感謝する。

参考文献

[1] N.Sanechika, Y.Mano, H.Ohigashi, Y.Sugawara and K.Torii, "Notes on modelling and implementation of the human player's decision processes in the game of Go", Bul. of ETL, 1981.

[2] S.Soule, "The implementation of a Go board", Information Science 16, 1978.

[3] W.Reitman and B.Wilcox, "The structure and performance of the Interim.2 Go program", 6th IJCAI, 1979.

[4] D.L.Parnas, "A techinque for software specification with examples", CACM 15,5, 1974.

[5] B.H.Liskov and S.N.Zilles, "Programming with abstract data types", SIGPLAN Notices 9,4, 1974.

[6] B.W.Lampson, et al., "Report on the programming language Euclid", SIGPLAN Notices, 12,2, 1977

[7] 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－", 信学技報 EC80-30, 1980.