ソフトウェア工学 46－10
(1986. 2. 6)

手続きの抽象化・蓄積・再利用

大石 東作
(電子技術総合研究所)

1、はじめに

現在の、計算機ハードウェア技術の急速な進展と普及は、かえって必要とされるソフトウェアの高度化・巨大化・多量化をもたらした。このような状況で、ソフトウェア・クライシス[1]は回避されず、さらにソフトウェア・バックログが日々増加している。これはソフトウェア作成の生産性が、ハードウェアの作成の生産性ほど向上していないことに起因している。ソフトウェア工学の成果に一定程度支えられているとはいえ、従来のソフトウェア作成では、プログラムを一品一品はじめから設計・生産している。とすると、世の中では似たようなソフトウェアが多数かつ独立に生産されることになる。これでは生産性の高度な向上は期待できないし、また人類全体の知的能力が不必要に浪費されことにもなる。この状況を解決するために、様々な方法が提案あるいは試行されている。第４世代言語、第５世代（論理型）言語、表操作言語、プロトタイプ技法、プログラム部品を用いるソフトウェア生産等がそれらである。

本論文では、上記のようなソフトウェア生産の問題点を解決することを目標として、多数の手続きを抽象アルゴリズムという形式で抽象化し知識として蓄積する。これらを検索して所定の機能・性能を持ったものを選択し、さらに具体的な細部仕様を付加して現実的なプログラムへと変換する方法について述べる。なおこの方法は、プログラム部品を用いる技法に親近性を持っている。

2、ソフトウェア生産における知識

知識情報処理における「知識」というものを考えてみると、表１のように２種類の形態がある。つまり分析的知識と総合的（合成的）知識である。

形式	支援する側	支援を受ける側	柔軟性	例
分析的	知識システム	人間	大	DENDRAL
総合的	人間	知識システム	小	FURNACE

表１、知識情報システムの２形式

従来多く開発されてきた知識情報システムはいわゆるエキスパート・システムと呼ばれるもので、ＭＹＣＩＮ（医学的診断）、ＤＥＮＤＲＡＬ（分子構造解析）等が著名である。これらは分析な知識を用いており、特定の分野の問題点において人間の判断や意志決定を支援している。内部的にはその分野に関する知識ベースに対して推論規則を適用し、ある結論を得ている。このようなエキスパート・システムにおいて、知識は事実とルールの二たつに大別され、内部的には１階の述語論理式、プロダクション・ルール、フレーム、意味ネットワーク等で表現されている。

このようなシステムであれば、最終的な意志決定をするのは人間であるから知識システムのほうで異常な判断を出してきたとしても、その異常に人間が気付けばシステムになぜそのような判断をしたのかを問うことも出来る。場合によってはシステムから出された判断を無視することもできるし、システムの異常さをバグとして、それらを無くすように知識システムを再構成することも可能である。

総合的（合成的）な知識システムは、ソフトウェア生産をも含めた工業生産に役立つものでありね知的ＣＡＤ、知的ＣＡＭ等の実現が期待されるのであるが、知識表現も定評のあるものが確立されてない。開発例もいまだ非常に少ないが、一例として、後述するように我々が開発しているプログラム合成システムＦＵＲＮＡＣＥがある。

この総合的な知識の場合、その質と形態が重要となる。人間が日常に操る知識であればね人間がバックグラウンドに持っている知識や常識というものを基礎にして、新しく追加された知識の多様性やその非形式性もなんとか処理できるだろう。しかし、計算機システム上に表現された知識には、人間の場合のように融通がきくことは期待できない。

特に総合的な知識の場合、それ自信で完結していて矛盾がなく、他からの支援なしに工業的生産に役立つことが要求される。この点にも総合的な知識システムを開発することの困難さがある。

さらにソフトウェア生産に知識情報を応用する場合について検討してみよう。

ソフトウェア生産は、以下のように大きく二つのフェーズに分けられる。

（１）対象とするシステム（ソフトウェア）の要求分析、仕様定義、概念設計

（２）同上システムの詳細設計、プログラム作成とデバック、完成ソフトウェアの保守管理

二つのフェーズを、前記の分析的と総合的知識に対応させるなら、第一フェーズは分析的フェーズに主導され、第二フェーズは総合（合成）的知識に主導される。

第一フェーズにおいては、システムが対象にする応用分野や問題自体に固有な知識情報が重要である。前に似たようなシステムを作ったことがあり、そのときの経験を整理して蓄積してあればこれを知識として活用できる。

したがって、このフェーズにおいて主要なのは問題や応用分野に固有な分析的な知識情報であるが、システムの概念設計においては、第二フェーズのプログラム作成における合成的な知識もある程度必要となる。

第二フェーズの主たる課題は、プログラム作成であるから、主たる知識情報はプログラミングに関する総合的な知識となる。もちろん、システムの詳細設計においては、問題や応用分野に固有な要求や仕様を反映させせなければならないから、分析的な知識も必要となる。総合的な知識においても、問題自身の性質に起因する要求や仕様の具体性（外部仕様）、プログラム実現に関する具体性（内部仕様）プログラム作成・稼働環境の具体性に適合できることが要求される。

3、なぜ手続きか？

プログラムやソフトウェアに関する知識には、次のようなものが考えられる。

（１）ソフトウェア・システム作成方法

（２）ソフトウェア生産の工程・品質管理法

（３）プログラム作成技法

（４）プログラムそのものの集合

（５）プログラム（ソフトウェア）の説明・検索法

（６）プログラムの正当性検証法

（７）プログラムの試験・検査・虫取り法

（８）ソフトウェア維持・管理・改造法

以下では、前章のソフトウェア生産の第二フェーズの知識情報処理を実現するために、上に列挙した「知識」うちでプログラムあるいは手続きを抽象化し、蓄積・利用する方法につき考察する。

従来のソフトウェア作成において生産性が低いのは、システムがいかに（Ｈｏｗ）動作するかを手続き型プログラム言語で記述しているためであり、これをシステムがなに（Ｗｈａｔ）をするべきかを仕様言語・宣言型プログラム言語で記述するように変更すれば、記述の量も大幅に減少し、生産性も向上するとされている。すべてのトップダウン的なソフトウェア生産技法は、上記の視点を根底に持っているといっても過言でない。

しかし問題は、単純でない。様々なトップダウン技法でも、問題点が現実に解決されていないことを見ればそれは明らかである。その原因は、システムのWhatとHowの間にある大きな意味的ギャップにあり、また今日のVonn Neumann型計算機がその根幹においてＨｏｗしか解釈しないからである。

ソーティングの例でみてみよう。論理型（宣言型）言語であるPrologで最も簡潔に記述すれば次のようになる。

```
sort(X,Y):-permutation(X,Y),ordered(Y).  ・・・・・ (1)
```

これを左から右へと馬鹿正直に評価(直接実行:generate and test法によって計算）すると、その計算量は莫大なものになり全く現実的でない。高度の並列アーキテクチャ上で並列計算したとしても状況は変らない。ハードウェア増強による効果がほとんど期待できないのである。またこの仕様を、効率のよいアルゴリズムに変換する一般的な戦略（規則）は存在しない。したがって論理型のProlog言語においても、置換（permutation）と順序比較（Ordered）を融合させたソーティングのアルゴリズムを人間が工夫する必要が出てくる。一般に簡潔な仕様を効率よく実行することが要求されるなら、それを実現する手続き（アルゴリズム）をその場で開発するか、前もって準備しておかなければならない。ここに我々が手続きを重視する根拠がある。

前もって手続きを準備しておくのであれば、一旦作成した手続きを保存しておき、これを何度も再利用すればよい

ことになり、ソフトウェア作成の生産性は向上するはづである。この観点から、プログラム部品によるソフトウェア生産技法が開発されている。すなわち、一旦作成したプログラムで汎用性の高いと思われるものを部品として蓄積・登録しておき、これを再利用するのである。ハードウェア生産における部品の有効利用という現実とのアナロジーで考えれば、ソフトウェア部品による方法は決め手としかいいようがない。

ところが、このようなプログラム部品登録システムを作成し、これを利用したとすると、なかなか旨く行かない。とかくプログラム部品というものは、”帯に短かし襷に長し”である。なぜそうなるか？ プログラムというものはある特定の状況を想定して作成されるのが通常であるから汎用性を持たせることが困難なのである。もともと汎用性がないものをあえて利用するとすれば、利用のための手数が膨大なものとなり非現実的になる。また従来のソート・パッケージのような多くの機能を持ち嵩張るものを汎用的な部品とするのも問題が残る。このような部品を用いると、ソフトウェアの最終製品に余計な機能を持った部品がそのまま残ることになる。それを十分余裕を持ったシステムで実行すれば問題はないが、現実のシステムでは様々な制約があるのが普通であるから空間的・時間的に効率の悪いソフトウェア・システムは望ましくない。

以上のような観点から我々は、手続きを抽象化することによりその汎用性を高め、その再利用を図ることにする。

4、なにを抽象化するか

プログラム作成時における様々な具体性を背負っているプログラム部品では、その再利用時に内包した具体性から逃れられないという桎梏から解放されるためには、手続きを各種の具体性から抽象化することが不可欠である。

従来手続きの抽象化は、データ抽象化あるいは抽象データ型の研究において実施されている。これらの研究では、プログラム作成上の細部（実現法）からの抽象化、あるいは実現された内部データを外部から直接的に参照・更新させない（カプセル化）に重点が置かれた。しかし後述するような十分な抽象化と具体化への方法を欠くために、様々な具体性に適合させるには、内部データへの直接参照や内部手続きの変更・追加が不可避だった。ではいかなる抽象化がさらに必要であろうか。そこで抽象化の対局となる、捨象さるべき具体性について見てみよう。

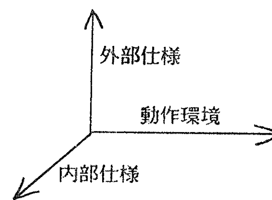
一般にプログラムの作成は、様々な具体性を配慮しながら実施されており、この具体性がゆえにそのプログラムは有用な機能を果すのである。これらの具体性を、分類すれば次のような３種類になろう。

（１）外部仕様 ： 特に与えられた問題の細部具体仕様

（２）内部仕様 ： 手続きの実現方法（時空間上の効率に大きな影響を与える）

（３）動作環境 ： 実際のプログラムを作成・実行するときのソフトウェア・ハードウェア環境

通常のプログラム作成では考慮すべき具体性が３種類であることは明確には意識されない。しかし手続きを抽象化して蓄積し、その再利用を図るとなると、３種類の具体性が個々に把握されかつ明確に捨象されねばならない。



第１図、３種類の具体性

さらにこの抽象化された手続きを再利用するには、抽象化とは逆の操作が必要になる。

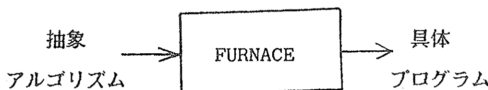
これらの具体性を捨象して得られた手続きの核あるいはエッセンスというべき表現を、抽象アルゴリズム[2]とよぶことにする。

抽象アルゴリズムは、具体的な細部仕様をあたえられ、機械的に処理を受けて現実の具体プログラムになる。したがって、人手によるプログラム作成と異なり、３種類の具体性が明確に処置されて具体プログラムへと変換する機構が工夫されねばならない。それは、実際に動作するプログラムが要求されており、かつ具体性の種類によって具体化の方法および時期が異なるからである。

5、抽象アルゴリズム

抽象アルゴリズムは、Algol流の言語Metalで記述されている。同アルゴリズムは、通常のコンパイラで処理できるような具体性を持っていない。なぜなら目的プログラムの実行時に必要な内部データ構造や、解かんとしている問題によって決定される各変数やそのデータ型等の情報を一切保持しておらず、手続きの本質的な部分のみが記述されているからである。この点でも通常のプログラムとは、一線を画している。

抽象アルゴリズムは、様々な具体性を付与され、第２図にしめすようＦｕｒｎａｃｅシステムによって、具体プログラムへと変換される。具体プログラムの記述言語としては、現在Pascalを用いているが、Ada等の言語への適応も容易である。



第２図　Ｆｕｒｎａｃｅシステムの概要

QuickSortを抽象アルゴリズムで記述した例を第３図にしめす。このアルゴリズムを見て、従来のアルゴリズムとの差がほとんどないと思われるかもしれないが、その差は次に詳しく述べられる。

抽象アルゴリズムに、４章で述べた３種類の具体性に付与するのにあたり、手続き本体に大幅に手を入れる必要があるのでは、アルゴリズムの発明者と同等の理解度が、同アルゴリズムの再利用者に要求されることになり、利用者の負担が大きすぎることになる。そこで具体性の付与にあたっては、抽象アルゴリズムの非手続き的なデータ定義部への追加が大部分で、手続き部への追加が最小限であるようにするため、以下のような構造をMetal言語に導入した。

（１）抽象単位データ（関係構造と属性構造の区別）
（２）代入文を関係と属性において区別
（３）拡張単位データ操作
（４）述語の関数引数

抽象単位データは、抽象アルゴリズムが操作する抽象データ構造を構成する基本要素の定義となっており、同上アルゴリズムの要の一つである。抽象単位データの例として、QuickSortに用いられるものを次にしめす。

```
algorithm QuickSort ( input*, output ) ;

 type
 BT = dunit [ ? ; ?? ; 1 ]
       ( SUC, PRE : BT )
       INDEX : integer ; VAL : key end ;

  procedure QUICKSORT
   ( function Order ( BT ; BT )
    : boolean; LEFT, RIGHT : BT );
   var LWORK, RWORK, KEY : BT ;
   function MEAN ( L, R : BT ) : BT ;
    var J,K : integer ; W : BT ;
    begin
     W <- L; J := ( R.INDEX + L.INDEX ) div 2;
     while W.INDEX <> J do W <- W.SUC ;
     MEAN <- W
    end ;
   begin
    if LEFT.INDEX < RIGHT.INDEX then
     begin (* size test *)
      KEY <- MEAN ( LEFT, RIGHT );(* sample *)
      MOVEPBT ( TEMPBT1, KEY ) ;
      RWORK <- RIGHT ; LWORK <- LEFT ;
      repeat
       while Order ( TEMPBT1, RWORK ) do
        RWORK <- RWORK.PRE ; (* right down*)
       while Order ( LWORK, TEMPBT1 ) do
        LWORK <- LWORK.SUC ; (* left up *)
       if LWORK.INDEX <= RWORK.INDEX then
        begin
         SWAPPBT ( LWORK, RWORK ) ;
         LWORK <- LWORK.SUC ;
         RWORK <- RWORK.PRE
        end ;
      until ( LWORK.INDEX > RWORK.INDEX ) or
       NULLBT( LWORK ) or NULLBT ( RWORK ) ;
      if not NULLBT( RWORK ) then   (* next *)
       QUICKSORT ( Order, LEFT , RWORK );
      if not NULLBT( LWORK ) then
       QUICKSORT ( Order, LWORK, RIGHT )
     end
    end ;
 begin  end .
```

第３図、QuickSort抽象アルゴリズム

```
type
BT = dunit [ ? ; ?? ; 1 ]
      ( SUC, PRE : BT )
      INDEX : integer ; VAL : key end ;
```

ＢＴは、抽象単位データをdunit文を用いて定義する。?は、具体プログラムにおいてこの単位データを実現する具体データ構造のスロット、??は同様に単位データの実現値の総数をしめすスロットである。これらは、問題やプログラム環境の具体性であり、それが現在未定であることをしめしている。??の右にある１は、QuickSortアルゴリズムで使用される一時変数の個数である。(　)内のSUC,PREは、単位データの２つの実現値間の”関係”を表現し、この関係を使って抽象的な両方向リストが形成される。（しかし、このことは具体プログラムにおいて、両方向リストが

使用されることを必ずしも意味しない。なぜなら、抽象的な両方向リストを実現する具体的なデータ構造は多数あるからである。）

（ ）の後部は単位データの"属性"を表現する。INDEXは、実現値からなる単位データの集合の要素間に全順序関係をあたえる。（このINDEXは属性の中でも特別の扱いをうける） VALは、分類さるべきデータ名である。そのデータ型はkeyであるが、実体は現在未定である。これが遅延(deferred)データ型であり、問題の具体性への一つの適合法となっている。再利用時に問題の仕様に応じて、VALの名称を変更することや属性部に新たなデータ項目を追加することは自由である。

抽象アルゴリズムは、通常のプログラム制御文(if then else,while do,repeat until等)や、抽象単位データの実現値への参照・更新、式の状態判定述語等を用いて記述される。

しかし、他のプログラム言語と異なるのは、Metal言語における代入文の扱いである。Metalでは、"関係"に対する代入文(<-記号を用いる)と、"属性"に対する代入文(:=を用いる)が区別されている。この区別により、抽象アルゴリズムの理解が容易になる。第３図のQuickSortにおいても、関係に対する代入文が多く、属性に対する代入が少ないことが明らかに読み取れる。これは、Sortのようなアルゴリズムでは、関係への操作が重要な役割を果たしているからである。また属性に対する代入が少ないのは、次に述べる拡張操作が使用されているためでもある。

拡張単位データ操作は、抽象単位データの実現値の関係や属性の複数個のフィールドを、一括して参照・更新・交換するもので、各種のものが用意されている。

第３図後半にある SWAPPBT (LWORK, RWORK); はその一例であり、LWORKとRWORKで指定される単位データの二つの実現値の属性部を一括して交換する。(ただし、前述のようにINDEXは特別で、このデータは交換されない)

この拡張操作を用いると、問題の細部仕様に応じた具体性を付与するときに、抽象単位データの定義にかなり自由な追加・修正を加えることが可能となる。かつ逆に手続き部には手を入れる必要がなくなる。したがって、この操作は問題の具体性への強力な適合法となっている。一方、抽象アルゴリズムの開発者の立場からすると、拡張操作を利用することにより、問題の細部仕様の具体性に一切気を配る必要がなく、自由にアルゴリズムを記述できる。これもまた、手続きの抽象化の観点からすると非常に望ましい性質である。

述語の関数引数もまた有効な具体性への適合法として利用できる。たとえば、ソーティング（分類）のアルゴリズムを再利用する際には、実行時に分類の昇順・降順の指定、属性部に定義されている任意のデータ項目を分類Keyとして指定が可能でなければならない。また分類Keyのデータ型への対処も必要である。これを実現するのが関数引数である。

第３図の例で、procedure QUICKSORTの引数部にある function Order (BT ; BT) : booleanがそれである。実行時に、この関数引数に分類のKeyと昇順・降順の指定を述語として乗せるのである。これにより、問題の細部仕様に対応する記述が必要かつ十分にでき、また実行時には空間的・時間的に最小限のオーバヘッドで具体化が実現できる。これは、具体化のための可能なかぎりの多様性に対する対処を背負いこんでいて嵩張るソート・パッケージを使用するのと対照的である。

6、抽象アルゴリズムの蓄積と検索

抽象アルゴリズムを利用して、各種の応用ソフトウェアが合成可能であるためには、様々の分野の抽象アルゴリズムが抽象アルゴリズム・バンクに多数蓄積されていなければならない。そこで、我々はKnuth,D.E.[3],[4],Aho,V.A.[5],Gotlieb,C.C.[6]の教科書等にある各種のアルゴリズムを採り上げ、これらを人手でMetal言語による抽象アルゴリズムに変換し、抽象アルゴリズム・バンクに登録・蓄積している。なおKnuthによるアルゴリズムは、"構造的プログラミング"以前の著作であるためgoto文を多用しており、これを除去した抽象アルゴリズムに変換するのに相当の手間を必要とした。同上バンクには、抽象アルゴリズム本体の他、同アルゴリズムに関する書誌的情報も登録している。

書誌的情報には、抽象アルゴリズム名、同発明者・改良者名、同発明ないし発表年月日、出典、使用法の説明、使用上の制約条件、計算量、特記事項、検索のためのキーワード等がある。

これらの書誌的情報を、通常の文献検索システムを改良したもので検索し、所定の抽象アルゴリズムが得られるようにしている。

7、再利用（具体的なプログラムへの変換）

抽象アルゴリズムの再利用（具体化）にあたっては、前述のように３種類の具体性に対処しなければならない。

通常の高水準言語によるプログラム作成においては、上記のような具体化はプログラムの作成自体とコンパイル時に集中して行なわれる。最新の言語Ａｄａを用いてもこれは変わらない。この方法では、プログラムの記述に特定の言語の仕様が前提となり、言語上の制約から脱することができない。特にデータ型が言語に組み込まれたものに制約されるのでアルゴリズムの抽象化は困難となる。また実行時に異なるデータ型につき同じような操作をするときも、実行時には各データごとに手続きを必要とし、時空間の効率は悪くなる。さらにプログラムの作成に当たって有効な「種」になるものはなく、工程数は大きくなる。

我々の方法では、Metal言語で記述された抽象アルゴリズムを「知識」として最大限に活用する。抽象アルゴリズムは通常のプログラム言語から独立しており、それらの制約を受けることはない。この抽象アルゴリズムは具体性を付加され、FURNACEシステムによりPascalプログラムへと変換され、通常のコンパイラ処理を経て実行される。したがってこの方法での具体化は、ＦＵＲＮＡＣＥ処理時、コンパイル時、実行時の３段階において自然に実施される。

ＦＵＲＮＡＣＥ処理時における具体化は、ホストとなるハードウェア・ソフトウェア環境の選択、問題に応じた抽象アルゴリズムの選択およびデータ定義、実行効率を考慮した具体データ構造の指定および要素数の決定である。コンパイル時には、各変数についての場所割付け、各データ型に応じた演算が準備され、目的コードが生成される。実行時における具体化は、抽象アルゴリズムから生成された同一のプログラムを用いて、与えられた問題に応じた異なるデータ型や演算を処理することである。

具体プログラムへの変換の一例として、簡単な住所録の処理を採り上げる。住所録は、次のようなデータ項目の並び（５００人分）からなるものとする。

氏名、電話番号、住所（都市名）

この住所録を、先程のQuickSortの抽象アルゴリズムを用いて処理する。この場合、遅延データ型と抽象単位データの定義は次のように具体化（修正）される。（なお具体データ構造としてはPascal言語のrecordデータ型を使用する）

```
type
 key = alfa ; key2 = integer ; key3 = alfa ;
 BT = dunit [ rcd ; 1..500 ; 1 ]
      ( SUC, PRE : BT )
      INDEX      : integer ;
      Name : key ; TeleN : key2; CityNa : key3 end ;
(*    name        telephone #   city_name     *)
```

抽象アルゴリズムの汎用性をしめすために、一つのアルゴリズムを用いて、住所録を電話番号で昇順に、名前を辞書式順序で降順に、住所を昇順にソートする。このために、次の３種類の述語関数を準備する。（なお、これらの述語関数は、現在抽象アルゴリズムの利用者があたえているが、これを抽象単位データの定義から自動生成することはごく容易である）

```
function NameRorder ( LOWER, UPPER : BT ) : boolean ;
  (* descending order on Name *)
  begin NameRorder := LOWER.Name > UPPER.Name end ;

function TeleNorder ( LOWER, UPPER : BT ) : boolean ;
  (* ascending order on TeleN *)
  begin TeleNorder := LOWER.TeleN < UPPER.TeleN end ;

function CityNaorder ( LOWER, UPPER : BT ) : boolean ;
  (* ascending order on CityNa *)
  begin
   CityNaorder := LOWER.CityNa < UPPER.CityNa
  end ;
```

住所録を、３通りにソートするには、次のように述語関数名を実引数にしてQuickSortを抽象アルゴリズムの本体部から必要に応じて適宜呼出せばよい。

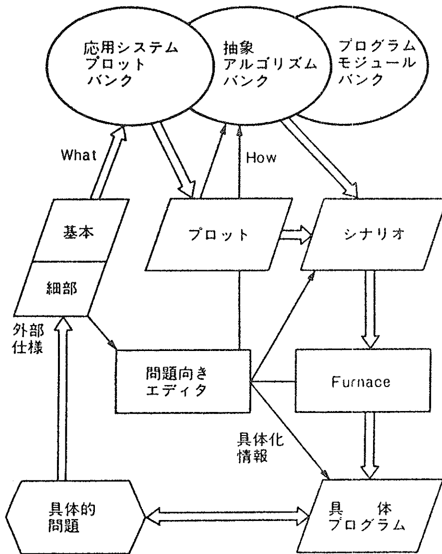
```
QUICKSORT ( TeleNorder,  TOP, LAST );
QUICKSORT ( NameRorder,  TOP, LAST );
QUICKSORT ( CityNaorder, TOP, LAST );
```

入出力部を付加して、全てを具体化（詳細化）し終ったQuickSort抽象アルゴリズムを付録１にしめした。この具体化にあたっては、QuickSortの手続き本体に手を入れる必要は全くなく、当初の目的が達成された。

8、手続きの統合的再利用

現在「知識」システムとして開発している汎用問題解決システム（Ｍｅｔａ－８８）では、応用ソフトウェア作成

用として、3種類の知識ベースを持っている。これらの知識を活用することにより、問題の具体的な要求仕様にあった応用ソフトウェアを短期間にかつ確実に作成して、ソフトウェア開発の大幅な生産性向上を目的としている。また、システム開発中に得られた有用な知識や経験は「知識」ベースに蓄積され強化される。第4図にその概要を示す。



第4図、汎用問題解決システム（Meta-88）の概要

3種類の知識ベースは、応用システム・プロット・バンク、抽象アルゴリズム・バンク、プログラム・モジュール・バンクである。応用システム・プロット・バンクには、システムに関するプロット（ｐｌｏｔ）つまり筋書きが多数蓄積されている。各プロットはシステムに関する基本外部仕様と基本手続きが対になって記述されている。基本手続きは、要素となる基本機能群とそれらの実行順序および基本データ定義により構成される。同バンクの利用者は、システムの基本外部仕様を鍵として同バンクを検索し特定のプロットを得る。この基本手続き部に細部仕様を反映させて具体的なシステムを生成する。この生成法には後述のように2種類ある。抽象アルゴリズム・バンクには汎用性のあるアルゴリズムが抽象化された形式で蓄積されている。利用者は必要な機能（名）を鍵として検索し、所定の要求・性能を持った抽象アルゴリズムを選択する。これに問題や実現に関する細部仕様を追加して具体プログラムを自動的に生成する。この生成を任うのが前にも述べたFURNACEシステムである。プログラム・モジュール・バンクには、通常のプログラムやサブルーチンに相当するものが蓄積されている。所定のものを即実行できるのが最大の利点である反面、問題や実現上の仕様変更に対応するのは非常に困難である。

9、おわりに

手続きを抽象化し、知識として蓄積し、これを統合的に再利用する方法について述べた。今後は、現在開発しているシステムを、形式的仕様技法と結合することによりWhatから効率のよいHow（手続き）を引出すことを可能とする予定である。

末筆ながら、日ごろ御指導いただく棟上ソフトウェア部長、佐藤情報システム研究室長、討論して頂いた情報システム研究室・言語処理研究室の皆さんに感謝する。

参考文献

[1]川合他”ソフトウェア・クライシス”情報処理，Vol.26,No.9
[2]大石”抽象アルゴリズムの記述と具体プログラムへの変換”情報処理,Vol.19,No.11
[3],[4]Knuth,D.E.”The art of computer programming” Vol.1,Vol.3,Adison-Wesley,1968,1973
[5]Aho,V.E.et al”The design and analysis of computer algorithms”,Adison-Wesley,1974
[6]Gotlieb,C.C.et al”Data types and structures”, Prentice-Hall,1978

付録1　具体化されたQuickSort抽象アルゴリズム

```
algorithm QuickSort ( input*, output ) ;
 const Ubound = 500 ;

 type
  key = alfa ; key2 = integer ; key3 = alfa ;

  BT = dunit [ rcd ; 1..500 ; 1 ]
       ( SUC, PRE : BT )
       INDEX      : integer ;
       Name : key ; TeleN : key2; CityNa : key3 end ;
 (*    name          telephone #    city_name     *)

 var I: integer ;
  TOP, LAST, WORKBT, PRE1 : BT ;
  PName : key ; Telnum : key2 ; City : key3 ;
  Dindex : integer;

 function NameRorder ( LOWER, UPPER : BT ) : boolean ;
   (* descending order on Name *)
   begin NameRorder := LOWER.Name > UPPER.Name end ;

 function TeleNorder ( LOWER, UPPER : BT ) : boolean ;
   (* ascending order on TeleN *)
   begin TeleNorder := LOWER.TeleN < UPPER.TeleN end ;

 function CityNaorder ( LOWER, UPPER : BT ) : boolean ;
   (* ascending order on CityNa *)
   begin
    CityNaorder := LOWER.CityNa < UPPER.CityNa
   end ;
```

次ページに続く

```
 procedure QUICKSORT
  ( function Order ( BT ; BT ) : boolean; LEFT, RIGHT : BT );
  var LWORK, RWORK, KEY : BT ;
  function MEAN ( L, R : BT ) : BT ;
   var J,K : integer ; W : BT ;
   begin
    W <- L; J := ( R.INDEX + L.INDEX ) div 2;
    while W.INDEX <> J do W <- W.SUC ; MEAN <- W
   end ;
  begin
   if LEFT.INDEX < RIGHT.INDEX then
    begin (* size test *)
     KEY <- MEAN ( LEFT, RIGHT );(* sample *)
     MOVEPBT ( TEMPBT1, KEY ) ;
     RWORK <- RIGHT ; LWORK <- LEFT ;
     repeat
      while Order ( TEMPBT1, RWORK ) do
       RWORK <- RWORK.PRE ; (* right down*)
      while Order ( LWORK, TEMPBT1 ) do
       LWORK <- LWORK.SUC ; (* left up *)
      if LWORK.INDEX <= RWORK.INDEX then
       begin
        SWAPPBT ( LWORK, RWORK ) ; LWORK <- LWORK.SUC ; RWORK <- RWORK.PRE
       end ;
     until ( LWORK.INDEX > RWORK.INDEX ) or
      NULLBT( LWORK ) or NULLBT ( RWORK ) ;
     if not NULLBT( RWORK ) then QUICKSORT ( Order, LEFT , RWORK ); (* next *)
     if not NULLBT( LWORK ) then QUICKSORT ( Order, LWORK, RIGHT )
    end
   end ;

procedure InputData ;
 begin
  PRE1 <- NILBT ;
  for I := 1 to Ubound do
   begin
    readln ( input, PName, Telnum, City ) ;
    LAST <- CREBT ( NILBT,PRE1, 0, PName, Telnum, City ) ;
    if I = 1 then TOP <- LAST else PRE1.SUC <- LAST ; PRE1 <- LAST ;
   end (* of input *) ;

 end ;

procedure OutputData ;
 begin
  writeln(output);
  writeln(output, 'Name      telephone number    City '  ); writeln(output);
  WORKBT <- TOP ;
  for I := 1 to Ubound do
   begin
    GETPBT ( Dindex, PName, Telnum, City, WORKBT ) ;
    writeln ( output, PName,'      ', Telnum: 6,'        ', City );
    WORKBT <- WORKBT.SUC ;
   end ;
  writeln(output); writeln(output);
 end ;

begin
 InputData ;
 writeln(output,'Input data'  ); OutputData ;

 QUICKSORT ( TeleNorder, TOP, LAST );
 writeln(output,
  'Sorted by Telephone number  '     ); OutputData ;

 QUICKSORT ( NameRorder, TOP, LAST );
 writeln(output,
  'Sorted by Name in reverse order'  ); OutputData ;

 QUICKSORT ( CityNaorder, TOP, LAST );
 writeln(output,
  'Sorted by City Name in normal order'); OutputData ;
end .
```