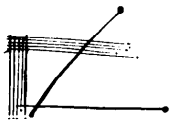
展　望



自動プログラミング†
——ソフトウェア工学と人工知能をつなぐ——

間野　暢興††

1. はじめに

ソフトウェア工学と人工知能学とは従来お互いに関係のない独立な道を歩んできた．ソフトウェア工学は，ソフトウェアの計画・設計・製作を行い，システムを操作し維持する過程において必要とする体系的知識および方法論と定義されている[1]．その工学としての発展の段階は，Jackson 法[2),3)]などシステムプログラムの設計原理が幾つか提唱されている段階にあり，自動化ツール (automated tool) がプログラムの変換などに多少用いられるようになってはきたものの，プログラミングの自動化はまだこれからの課題である．これに対し，人工知能における自動プログラミングの研究は，形式的な推論による問題解決のアプローチから知識工学的アプローチへの移行がみられるとはいえ，扱う問題の規模は toy problem のレベルを脱したとはまだいい切れない状況にある．

しかし，最近ソフトウェア工学と人工知能の分野の間で，互いの利用価値が多少認識されてきた兆しが見える．自動プログラミングという遠大な目標に向かって一歩でも研究を推し進めるには，両者の連携をさらに強化することが必要である．本展望では，ソフトウェア工学において蓄積された知識および方法論を，人工知能における形式化，表現，問題解決メカニズムを用いて自動化するアプローチについて述べる．

ただし，自動プログラミングの実用化は，理論の綺麗ごとだけでは解決できず，対象問題領域の特性や人間の行っているプログラミング方式などと結びついたある種の泥臭さを避けることはできない．本誌編集委員会からの要望もあり，この展望では理論よりも実用面に焦点をあて，しかも（全）自動という言葉にあまりとらわれずに話を進めることにしたい．

† Trends in Automatic Programming——Bridging the Gap between Software Engineering and Artificial Intelligence—— by Nobuoki MANO (Electrotechnical Laboratory).

†† 電子技術総合研究所ソフトウェア部言語処理研究室

1.1 「自動プログラミング」とは？

ひとくちにプログラミングといっても次のような広い範囲の段階がある．

① 問題の提示
② 要求仕様の作成
③ 抽象データ・オブジェクト (abstract data object)，プロセス，それらの間の論理的インタフェースから表わされるシステム設計
④ プログラムの論理モデルの作成
⑤ 実現データ構造・物理的インタフェースの指定による具体的プログラム・コードの作成
⑥ コードの検査
⑦ コンパイル，実行
⑧ 保守

本展望において「自動プログラミング」という言葉が対象とする範囲は上の③～⑤である．

「自動プログラミング」という言葉は，最近では主に人工知能の分野で用いられており，演繹・推論・帰納などの知的機能に基づいて計算機が自動的にプログラムを合成することをさしている．人工知能における種々のアプローチについては，文献 4)～7) を参照されたい．

これに対して，ソフトウェア工学においては，超高級言語 (very high level language) や非手続き型言語 (non-procedural language) にさらに多少の知的なプログラミング支援ツールを具備したシステムを用いて，人間の指導性を生かした半自動的なプログラミングの形態をとっている．こういうわけで，「自動プログラミング」ではなく「自動生成」(automatic program generation) という言葉が使われる．ここでは，「自動プログラミング」という言葉を広義にとって全自動・半自動の両方を含むものとする．

1.2 自動プログラミングの必要性

プログラミングの生産性を高めることは，昔から計算機科学における重要課題であった．その目的を達成するため，高級言語やソフトウェア工学が提唱されて

きたが，これからはさらに進んで自動プログラミングの研究が大切なことを述べたい．

ここで合成の対象であるプログラムの性質を分類してみると，大まかに次の2種類に分けられる．

i）いわゆるアルゴリズムを記述したプログラム

ii）応用問題の事実 (facts) に依存したプログラム

i）は，Knuth の Fundamental Algorithms の本にのっているような，ひらめきと高度の論理的思考を要するアルゴリズムであり，システムを構成するモジュールとしては比較的まとまった閉じた性格をもつ．この種のプログラムは人工知能の分野におけるプログラム合成の対象として選ばれることが多かったが，現在のところ論理的に深い推論を計算機にやらせることは困難であり，部品としてコード・ライブラリに貯え，必要に応じて修正や具体化を行って使用することが考えられる．

ii）はシステムの目的を実現するために働くモジュールであり，現実のシステムプログラム中では圧倒的に大きな割合を占める．その設計は自由度が大きいが，システムの最適化の見地からなされることが多い．これについては，十分大きなコード・ライブラリが開発されればそれを使えば良いというものではなく，大規模なコードの再使用は現実にはない．Alford が指摘しているように[8]，ソフトウェア・モジュールはさまざまな目的に奉仕する異なったタイプの処理の混合物であって，コード・ライブラリからいくつかのモジュールを引き出してきてそれらをつなげればシステムができ上がるというわけにはいかない．それゆえその自動作成が可能となれば，たとえばオフィス・オートメーション[9),10)]の場などで，その効果はすこぶる大きなものがあろう．

1.3 自動プログラミングの困難性

自動プログラミングが実現に至るまでには多くの困難が横たわっている．それらを次に列挙する．

- プログラミングにおける概念が十分分析抽出されておらず，形式化・体系化もなされていない．
- 少し複雑なシステムになるとその仕様を完全に与えるのが困難である．
- 問題の性質としてオープンなところがあり，多様な情報の利用が考えられ，合成過程も複雑となり，制御構造が明確にしにくい．
- 人工知能の定理証明によるプログラム合成のアプローチでは，ヒューリスティクスを用いないと公理の組合せの爆発が起り，合成に必要な計算時間が指数関数的に増大し，実用上無意味となる．

これに対する研究の進め方としては，

- 対象問題の領域をしぼる．
- Floyd[11)] のいうように，計算のパラダイムに関心を払い，人間のもつプログラミングに関する知識の形式化と蓄積をはかる．
- 知的プログラミング支援システムをまず考え，その延長線上で自動化を実現する．
- ソフトウェア工学と人工知能の研究成果を積極的に利用し，両者の結合に努める．

などの方針が考えられる．

1.4 ソフトウェア工学と人工知能の結合

図-1 に示すように，これからの自動プログラミングは，ソフトウェア工学の知識や方法論を形式化して人工知能が提供する機構の上に具現化をはかる，という方式がとられるものと思われる．このようなシステムにおいては，システムの設計者とユーザの間に知的なコミュニケーションが成り立つ．このように自動プログラミングの研究を推し進めることにより，自動化という当然のメリットのほかに，analysis by synthesis によるプログラミング過程の解明が，人間プログラマにもプログラミングについての新たな指針を与えてくれるものと期待できる．

2. 半自動および全自動プログラミングシステムの例

半自動・全自動プログラミングシステムがどんなものであるかを理解するために，二，三の具体例をあげる．同時に，それらが含んでいる考え方のうち，今後の自動プログラミングシステムにとって重要な役割を果すと思われるものに注目する．

2.1 半自動プログラミングシステムの例

ここでは，MODEL-II[12)] および PMB[13)] (Program

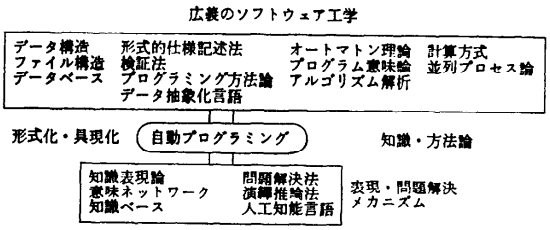


図-1 ソフトウェア工学と人工知能の結合による自動プログラミング

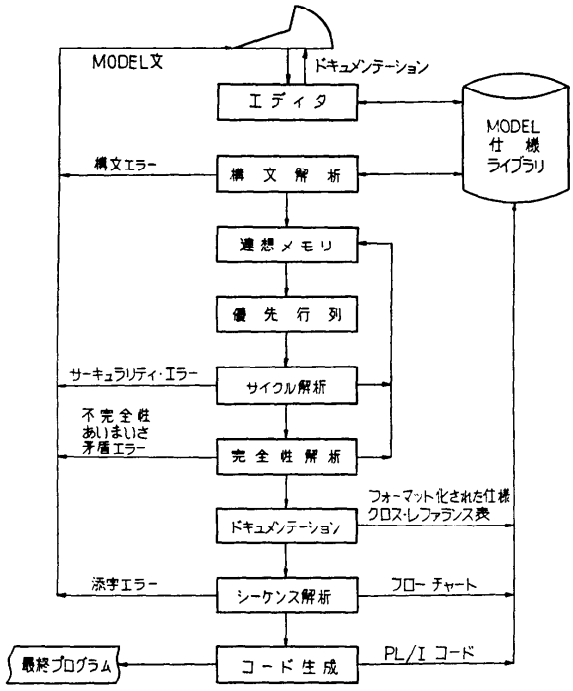


図-2 MODEL-II 処理系の構成

```
DISK IS MEDIA
  SALE IS FILE (DISK, SEQUENTIAL)
    SALEREC IS RECORD (SALE, (*))
      CUST# IS FIELD (SALEREC, PIC (9(7)))
      ITEM IS GROUP (SALEREC, (NITEM))
        QUANT IS FIELD (ITEM, PIC (229))
        STOCK# IS FIELD (ITEM, PIC (9(7))
CUSTOM IS FILE (DISK, ISAM, KEY=CUST#)
          .....
INVEN IS FILE (DISK, ISAM, KEY=STOCK#)
          .....
INVOICE IS FILE (DISK, SEQUENTIAL)
          .....
  (a) データ文
POINTER. STOCKREC. STOCK#=ITEM. STOCK#
  (FOR EACH_ITEM)..... a 5
  (b) ポインタタイプの表明文
IF SOURCE. BAL+TOTAL>LIM THEN SELL=TRUE
                        ELSE SELL=FALSE... a 6
IF SELL THEN MSG='CREDIT LIMIT EXCEEDED' ... a 7
  (c) ビジネス決定の表明文
EXT (FOR_EACH_ITEM)=QUANT (FOR_EACH_ITEM)*
                    PRICE (FOR_EACH_ITEM)... a 8
SUBT=SUM (EXT (FOR_EACH_ITEM)) ..... a 9
TAX=SUBT*0.6..... a 10
TOTAL=SUBT+TAX ..... a 11
  (d) 会計規則の表明文
```

図-3 MODEL-II の非手続き言語によるプログラムの一部

Model Builder の略) を取り上げる．これらは専用のプログラム言語によるユーザからの（断片的）プログラム入力をもとに，漸増的にプログラムのモデルを内部に組み立てていくシステムで，そのモデルの（構文的）完全性・無矛盾性などをチェックするプログラミング支援機能をもつ半自動プログラミングシステムである．MODEL-II は，人工知能的なアプローチを排除した自動プログラム生成システムであり，PMB は人工知能の考え方を取り入れた知的エディタとでもいうべきものである．

2.1.1 MODEL-II[12]

ビジネスデータ処理プログラム作成用に Prywes らが開発した MODEL-II システムは，ユーザの非手続き言語によるプログラムから PL/I のコードを生成するシステムである (図-2 参照)．データの構造およびデータフローに基づいてプログラムの組立てを行う点と，ユーザプログラムを解析してその不完全さ・あいまいさ・矛盾，そのほかを検出してユーザに知らせ修正させるプログラミング支援機構を備えたシステムである点に特色を有する．

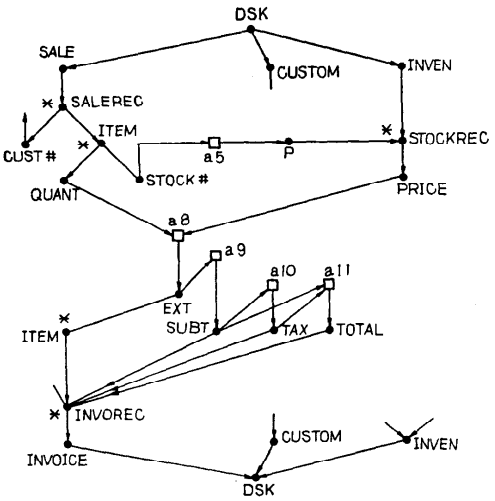
ユーザプログラムは，図-3 に示すように，(a) のようなデータ構造記述文と，(b)，(c)，(d) のような表明文で記述される．(b) のポインタタイプの表明文は，2つのファイルの繰返し集団間のつながりを示すものである．(c) のビジネス決定は，選択を表わす変数を適当な条件に関連づけて決定を表現する．

データ文に基づく入出力データの構造のレベル関係と，表明文のデータフローを有向グラフとして表現したものが図-4 である (一部略)．有向グラフの一部をトポロジカル・ソートにより全順序に順序づける手続きと，同じ繰返しループの範囲に属する文を集める手続きの相互再帰呼び出しにより，有向グラフはフローチャートの形式に変換され，それからコードが生成される．

データの構造を巧みにプログラム生成に利用しているところが面白いが，データ文は高レベル・低レベルの表現が混在している感じがある．

2.1.2 PMB[13]

集合操作・リスト処理・探索・分類・単純なデータ記憶と検索などを対象問題領域としたプログラミング支援システム PMB が，McCune により PSI プロジェクト[14]の中心的構成要素として開発された．PMB



●データ名；□表明ノード；*繰返しノード；Pポインタ変数

図-4 図-3 中の文の有向グラフ

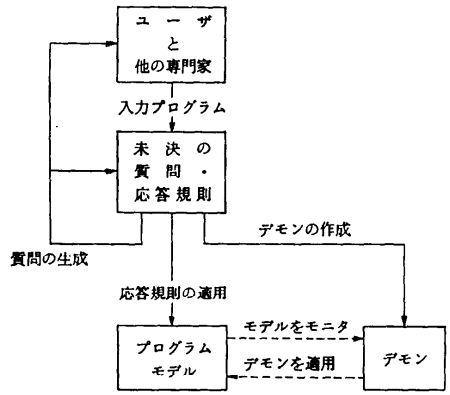


図-5 PMB の制御構造

はこれら対象領域のプログラミングに関する構文的知識や意味論的知識の一部を前向き推論の応答規則やデモン（**3.1.2** 参照）として内蔵し，インクリメンタルに与えられる自然言語などの非形式的ユーザ言語の内部表現（プログラム・フラグメント言語）をもとに，プログラムのモデルを半自動的に構成していくシステムである（**図-5** 参照）．プログラムのモデルは木構造状のゴール—サブゴールの展開により表現される．

PMB の動作のトレースの一部を**図-6**に示す．図の（a）の状態にあるとき，（b）のように classify_loop のフラグメントを与える．これは “exit condition” とよばれるブール表現が真になるまで “loop body”（未定義）を繰り返し，真になると “exit” へ出るループを定義している．loop を処理する応答ルールは “exit_condition” が loop body 中に含まれていることを保証しなければならないから，“exit condition” の位置

```
(a)
Current program model:
  program classify;
    begin
      concept←input (concept_prototype,user,concept_poromp);
      classify_loop: ???
    end
Current demons active:
    None
(b)
Inputting fragment:
  classify_loop:
  until exit (exit_condition)
    repeat loop_body
    finally exit:
  end loop
(c)
Current program model:
  program classify;
    begin
      concept←input (concept_prototype, user, concept_prompt);
      until exit
        repeat
          loop_body: ???
        finally
          exit:
    endloop
    end
    exit_condition: ???
  Current demons active:
    Demon 1: awaiting control structure containing
      "exit_condition"
    Demon 2: awaiting control structure containing
      "exit_condition"
(d)
Inputting fragment:
  loop_body:
    begin
      loop_input;
      exit_condition;
      classification;
      output_classification
    end
(e)
Current program model:
  program classify,
    begin
      concept←input (concept_prototype, user, concept_prompt);
      until exit
        repeat
          begin
            loop_input: ???;
            if exit_condition: ???
              then assert_exit_condition (exit);
            classification: ???;
            output_classification: ???
          end
        finally
          exit:
      endloop
    end
  Current demons active: None
```

図-6 PMB の動作のトレース例

が決められるまで待つように Demon 1 を作り，同様に “exit condition” の位置が定まったとき，それを assert_exit_condition をもつテスト内に置く Demon 2 を作る．このときプログラム・モデルは(c)のようになる．ここで(d)のように loop body を4つの名づけられたステップからなる複合体として与えてやると，PMB は “exit condition” がどこに起るかを知る．Demon 1 はそれが “loop body” 内にあり，その親が loop であるのを見出して成功裡に消え失せる．Demon 2 も，“exit condition” をテストの判定述語とする新しいテンプレートを作り出す．この時点でのプログラムモデルは(e)のようになる．

このシステムは，ユーザの指導権とシステムの知的な書記的役割が協調している例として面白く，プログラムのフラグメントを与える順は任意であるなどの融通性を備えている．

2.2 全自動プログラミングシステムの例

従来の定理証明などの形式的推論によるプログラム合成システムはこの例であるが，ここでは筆者の私見から，2階のヒューリスティクスを用いる Duran の合成システム[15),16)]と，カーネギー・メロン大学における PQCC (Production Quality Compiler Compiler) プロジェクト中のコード・ジェネレータ・ジェネレータ[17),18)]を例としてとりあげる．ただし合成されるプログラムの規模は大きなものではないことをあらかじめお断りしておく．

2.2.1 Duran の合成システム[15),16)]

1階の論理による形式的定理証明法を使わず，公理や推論を可能な限りシステム構造に組込みにし，知識工学的にまとめたヒューリスティックな2階のプログラミングの知識によって合成を行うシステムとして Duran のシステムがある．

プログラムの合成の問題は一般に次式で与えられる入出力関係を満たすプログラムPを見出すことである．

$$\exists P\,[I(v_{in}) \longrightarrow R(v_{in}, Fp(v_{in}))] \qquad (1)$$

すなわち，入力表明Iを満たす入力変数のベクトル v_{in} を，出力表明Rを満たす出力へ変換する関数 Fp であるようなプログラムPを見出すことである．(1)式は関数 Fp の存在の証明を必要とするので2階の問題である．ループをもつプログラムの合成には，さらに問題に応じた数学的帰納法を与えるか，ループ不変量Qとループ制御述語Bを与えなければならない．

Duran のシステムでは，ユーザから与えられた {I, R, Q, B} の情報から，初期設定を表わす述語 δ_I，ループ制御変数更新を表わす述語 δ_B，そのほかのループ変数の更新を表わす述語 δ_Q を求め，それらから初期設定とループ本体のコードが生成され，[⟨initialization⟩ WHILE B DO ⟨loop body⟩ OD] の形のループセグメントが組み立てられる．

このシステムでは，

(1) ループ制御変数の初期化
(2) ループ不変量に含まれていない変数の初期化
(3) δ_B の生成
(4) δ_Q の生成

の各フェーズを行うためのヒューリスティックな知識を condition-action のプロダクションルールの集合としてまとめて与えており，(1)から(4)へと順に進んで δ を見出すようになっている．後のフェーズで前のフェーズにおける選択が悪かったことが見出されると，前のフェーズへ後戻り (backtrack) し別の選択が行われる．表-1 にその知識の一部を示す．

整数の積を計算するプログラムを合成する場合の {I, R, Q, B} と結果のフローチャートを図-7に示す．ここで，フェーズ(1)では，v_B は c が相当し，表-1の(b)のルールの条件に該当するので，そのルールの action が実行され，$c=y$ がループ制御変数の初期化に対応する δ_I 部分を生成する．

このシステムではこのほか，整数の平方根の求解，配列の要素の探索などが合成例としてあげられており，toy problem の範囲を出ていないが，その2階の知識と推論過程は人間のそれと類似していると思われ，興味深い．ただし，フェーズ(1)から(4)へとい

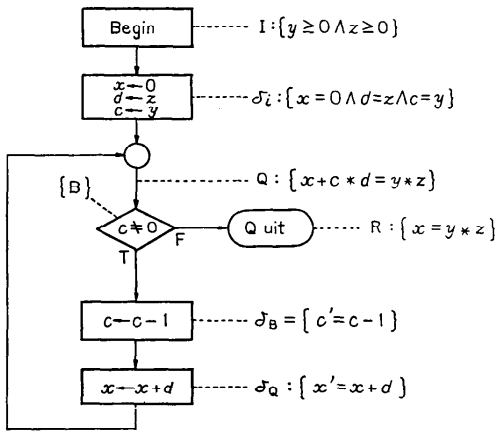


図-7 整数の積を計算するプログラムのフローチャートと表明

表-1 ループ制御変数の初期化用ヒューリスティクス

¬B atom form	action rule
(a) v_B=integer≒(0 or 1) or input variable	v_B=(0 or 1)
(b) v_B=(0 or 1)	v_B=(select next candidate from input variable list)
(c) v_B<integer≒(0 or 1) or input variable	v_B=(select next candidate from input variable list)
(d) v_B>integer≒(0 or 1) or input variable	v_B=(0 or 1) or (less likely), perform v_B=(select next candidate from input variable list)
(e) contains no local or output variables (subscripts ignored)	null δ_i contribution
⋮	⋮

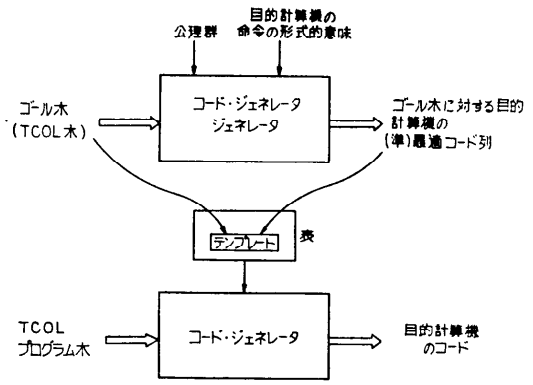


図-8 PQCC におけるコード・ジェネレータ・ジェネレータ

う推論過程の制御構造が適切かどうかに疑問が残る.

2.2.2 PQCC におけるコード・ジェネレータ・ジェネレータ[17),18)]

コンパイラ・コンパイラの自動化のうちで，目的計算機の特性に合せたコード生成機の自動作成は一番遅れている部分である．PQCC[18)]においては，目的計算機の形式的記述からパース木の機械コード翻訳への表現を導出するのに，形式的な方法による機械独立なコード・ジェネレータ・ジェネレータを利用する研究を行っている（図-8 参照）.

たとえば，PDP-11 の機械命令の意味は，次のように Hoare の形式化[19)]による記述で表現される.

{MOV src, dst} dst=src∧N=(src<0)∧Z=(src=0)

{BIC src, dst} dst=(dst′∧¬src)∧Z=(dst=0)

また，このシステムが使用する算術的・論理学的公理としては，たとえば次のものがある.

$$B \equiv \neg\neg B \qquad (2)$$

いま，ゴール木として次式を与えて対応するコードシーケンスを求めるものとする.

$a := b$ and c

この式の事後条件は，$a=b \wedge c$ となり，means-ends analysis により，右辺の主要演算子∧を事後条件にもつ意味的に近い機械命令は BIC である．しかし，BIC は割り付け文の左側と∧の左側の被演算子を同一にすることを要求するので，システムはそのためのサブゴールを次式のように設定する.

$a := b$

もうひとつの問題は，BIC がその source 被演算子の補数をとることである．これとゴール木との差異を埋めるには，¬を右辺の主要演算子として持つような，$P_1 \Rightarrow P_2$ の形の公理を捜すと（2）式が見出される．システムはこれを∧の右側の引数に適用し，

$t := \text{not } b$

というサブゴールを設定する．結局もとのゴールを作り出すコード・シーケンスは，

MOV c, t

CMP t

MOV b, a

BIC t, a

となる．有限時間内で複数個の解が見出されたときは，そのうちで最適なものが選ばれる．ゴール木とコードシーケンスのペアはテンプレートとして図-8 の表に登録され，コンパイラのコード・ジェネレータに与えられる.

ここでとられているアプローチは，ヒューリスティクスの働かせにくい一般的な定理証明法を用いるのでなく，問題領域に応じた単純化を行い，meansends analysis の手法を適用して探索の能率化をはかっている点が注目される.

3. 自動プログラミング・システムの基本概念

この章では，これからの自動プログラミング・システムを想定して，その基本概念と関連事項を検討する.

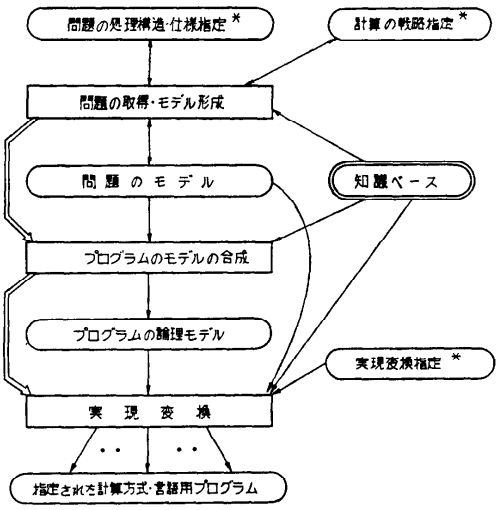
3.1 自動プログラミング・システムの構造と機能

3.1.1 システムの構造と問題解決機能

ソフトウェア・モジュールを作成する場合に，それが満たすべき仕様として，次のものがある.

① プロセスやデータ・オブジェクトのモジュールなどと，それらの間のデータフロー・コントロールフローからなるシステムの処理構造の定義

② モジュールの機能の定義



← データフロー ⇐ コントロールフロー * ユーザの与える情報

図-9 自動プログラミングの過程

③ モジュール間で受渡しされるデータのタイプと領域，データの構造の定義

このほか，論理レベルから物理レベルへのインタフェースの変換指定などがあり，ソフトウェア・モジュールはこれらの混合物として実現される．Jackson 法は，これらを統一的に扱い，入出力列の定義から変換を実行するモジュール列を導出するという原理に基づくプログラム作成法である．もちろん，Jackson 法は人間プログラマ向きに提案された原理であるが，それをモデルとした自動プログラミング・システムを考えることもできよう．そのようなシステムの問題解決過程は**図-9** のようになる．

図から分るように，これからの（自動）プログラミング・システムは次の特徴を持つ．

① マクロな水準からミクロな水準までの多段階性

② 知識を利用してプログラム合成を行う，いわゆる知識工学的アプローチ

この特徴のゆえに，システムが持つべき問題解決の性質は次のようになる．

①′ 上位水準の仕様や設計決定が，context として下位水準で用いる知識の範囲を限定し，判断に影響を及ぼす．

②′ 対象問題領域から帰納的に抽出されたプログラミングの知識を，推論用規則の集合の集まりとして整理し，それらを用いてプログラム合成の問題解決を行う．この場合従来の形式的推論による問題解決法と比較して，前向き推論がより中心的な役割を演ずるようになる（ただし，逆向き推論も検索やローカルな証明にもちろん大切なものである）．また，意味的なヒューリスティクスとよばれていたものが，ここではむしろ問題解決を進める主体となって働いていることに注意すべきである．このほか，類似した解決ずみの問題を利用して新しい問題を解く場合に，データの構造の対比を利用するなど，類比の機能も重要なものとなる．

なお，規則で表わされる知識の増大につれて，知識の modularity の保証と，制御構造の導入による処理能率の向上，という相反する要求があり，これは未解決の大きな問題である．

3.1.2 システムが内蔵する知識

「知識」とは何をさすのかの厳密な定義はなされていないようである．ソフトウェア工学の方法論などは人間の所有している知識であるが，ここでいう知識とは，計算機の内にあり，メタレベルにおける問題解決（プログラム合成など）において使用されるもので，問題領域に関しての形式化されたデータ・規則・手続きなどをさす．知識は断片的なものであって，ある目的を達成するためには，推論機関 (inference engine) などにより組み合わせて用いられて初めて意味のある利用がなされる．

自動プログラミングの世界では，次にあげるものは知識に属する．

（1） オブジェクトおよびセグメント

自動プログラミング・システムや知的プログラミング・システムにおいては，計算のモデルを導入してそれを用いて対象問題のプログラムや知識（構造）を表現することが多い[20]．モデルの記述には，人工知能の知識表現で使われる frame あるいは script 表現が用いられる．その場合，SIMULA の class や CLU の抽象データタイプあるいは代数的仕様記述におけるのと同様に，データ・オブジェクトとそれに関連する操作手続き（セグメント）の集合が定義され[21]，その instance がモデルを記述するのに用いられる[20]．データ・オブジェクトはそのデータの内部構造についての記述を持つ．

プログラムあるいはそのコードの一部分の，機能的にあるまとまった処理単位のモデルをセグメントとよぶ．セグメントは，入出力に関する振舞の仕様記述（入力・出力・実行の前に満たされるべき事前条件 (precondition)，実行後の状態変化を表わす事後条件 (postcondition)，エラー条件）と，その処理手続き本体を表わす記述をもつ(**図-10**[22]および **3.3.2** 参照)．

```
ProcessScript  MoveMessage;

  Input    mn:MessageNumber. fn:FileName;
  Output   nmn:MessageNumber;
  DataStructuresAccessed   CurrentMessageFile;
  Preconditions   MessageFileOpen = true;
  SideEffects   none;
  Undoability   derived;
  ErrorConditions     MessageNotFound,
          DestinationFileNotFound,
          NoWritePrivilegeForDestinationFile,
          CopiedButNotDeleted-NoDeletePrivilege;
  Body
    begin
      nmn : = CopyMessage(mn,fn);
        errorcase
          MessageNotFound: fail;
          DestinationFileNotFound: fail;
          NoWritePrivilegeForDestinationFile: fail;
      DeleteMessage(mn);
        errorcase
          MessageNotFound: fail;
          NoDeletePrivilegeForOpenFile: fail with
            CopiedButNotDeleted-NoDeletePrivilege;
    end;
```

図-10 Script 形式によるセグメントの記述例[23]

(2) 規　則

前向き推論用と逆向き推論用の2種の規則がある．

前向き推論用規則は，

〈conditions〉⇒〈actions〉

の形式で，左辺の条件が満たされると右辺の action (データの書き替えなどの副作用のある動作) が実行される．プロダクション・システムのルールはこの形式をとる．また，データベースの見張り役として，その不完全性や矛盾のあるなしをチェックするデモンもこの一種である．PMB におけるトップダウンのデータ駆動型応答規則やデモン，Duran のシステムにおけるプロダクション・ルールを参照されたい．

逆向き推論用規則は，次の形式をとる1階述語論理式

〈antecedents〉⇒〈consequent〉

を，consequent をゴール，antecedents をサブゴールの集合として，含意を逆の方向に使用するものである．逆向き推論の連鎖により，あるゴールが成立するかどうかの証明や，それらを満たすものの検索を行う場合などに用いられる．

(3) 知 識 構 造

オブジェクト，セグメント，関係・属性などのおのおのにおいて，上位下位関係(一般化—特殊化の関係)を主体とした分類構造を表わす．下位のものは，その水準における固有の特性のほかに，例外についての記述がなければ上位のものの特性を受け継ぐ．この構造を利用して，最も意味的に近いものの取り出しなどが行える．またこの構造上に(2)の規則を重畳して表現しておくと，その取り出し利用や知識の獲得，整理などに好都合である．図-11[22] に知識構造の例を示す．太い線は上位概念の関係 (a-kind-of) を示す．

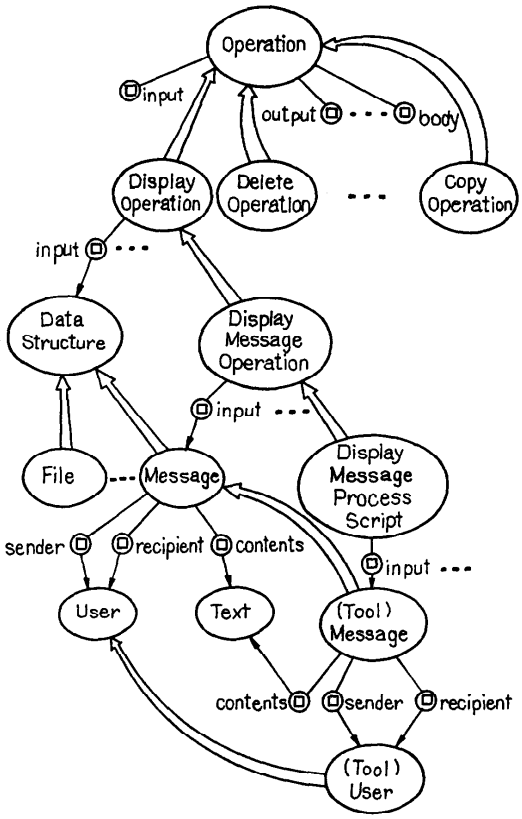


図-11 知識構造の例[22]

(4) 公　理

プログラムのモデルの正当性の証明や，定理証明などのゴール指向型の問題解決において，論理のつながりをつけるために必要な役割を果すものである．論理的なもの・数学的なものと，問題に特有のものとがある．

(5) 専門家としての手続き

MODEL-II における各種解析ルーチンや，PQCC においてさまざまな側面からオブジェクト・コードの最適化を行う専門家的な手続きルーチン群も，知識と称されているようである．

(6) パラダイムあるいはスキーマ

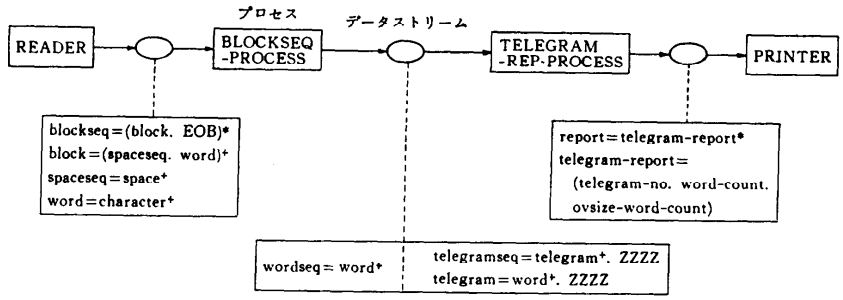


図-12 Jackson の電信文の問題

同じような目的を達成するアルゴリズムどうしは，共通する計算のパターンをもつものである．これをパラダイム[11]あるいはスキーマ[23]という．パラダイムはかなり抽象度の高いものであって，問題解決法の generate-and-test などがその例である（ちなみに，**3.3.2** で述べるプログラム合成や Duran のシステムはこのパラダイムに属する）．スキーマはプログラムの構造が決っていて，その個々の具体的なサブセグメントが未定のものである．

3.2 問題の定義・仕様

ⅰ）処理構造の定義

Jackson は，Dijkstra 流の構造的プログラミングが問題生来の並列性を殺し過度な逐次化を強制する点を批判して，問題の論理的構造に基づくプログラム設計を提唱している[3]が，これは妥当な考え方と思われる．オフィス・オートメーションのシステム[10]やオペレーティング・システム[24]などに代表されるように，一般にシステムは，複数個のプロセスと共有データ構造，およびそれらを結ぶデータフローとコントロールフローによる階層的ネットワークとして表現される．またこの表現を基本として，この上にプロセスの入出力仕様，抽象データ・オブジェクトとしての共有データ構造や操作の仕様，データストリーム構造の記述などを重畳して，システムを複数個のモジュールに分割して表現することができる．図-12 に Jackson の電信文の問題における処理構造とその各データストリームの構造を表わす文法を示す．Jackson 法における構造不一致 (structure clash) の解決法も処理構造の定義と関係している．

人工知能における自動プログラミングの研究では，処理構造の考えが出てくるものはまだ僅かしかないが，問題を論理レベルと物理レベルとに分けて考える上で重要な概念であり，マイクロプロセッサのネットワークというこれからの処理系ともよく整合するものである．

ⅱ）データの構造定義

データベースの重要性が広く認識され，またデータの構造に基づくプログラミングが提唱されるようになって以来，データ構造の論理レベルにおける構造記述が仕様の一部として重要性を帯びてきた．データの構造は，全体から部分へとさまざまの水準で記述を与えることができる．

① 複合構造

実際に用いられるデータ構造は，②で述べる基本構造の複合構造であって，複数個の構造を共有するものと，構造は共有せず値のみ共有するものがある．前者の比較的単純なものとして階層結合があり，後者は直積結合とよばれる．**2.1.1** に紹介した MODEL-II のポインタ文は，直積結合のファイルのレコード間の結びつきを示したものである．

② 基本構造

集合，関係，シーケンスなどのタイプがこれにあたる．各タイプに対し，そのドメインや制約条件の記述を与える．

③ 繰返しあるいは再帰データ構造

Jackson 法における Jackson 木は有限オートマトンの正規表現で表わされる[25]．また再帰データ構造は文脈自由文法の形式で記述される[26]．述語論理を用いてデータの構造を記述する方式もある[27]．データストリームもデータ構造と同じものと考えて構造を記述できる．

④ 構造の単位の記述

データ構造を論理レベルでとらえた場合，構造の単位どうしの関係と単位の持つ属性を表わす[28),31)]．

ⅲ）入出力仕様記述

プログラムの入力に関する条件（事前条件あるいは

入力表明）と入出力間の満たすべき条件（事後条件あるいは出力表明）を与えるもので，特に人工知能における定理証明によるプログラム合成法では，これが問題の主要仕様である．その場合，**2.2.1** の（1）式は（3）式のように1階の問題として表現され，述語論理による証明過程をたどって，その入出力仕様を満たすプログラムが抽出される．

$$(\forall v_{in})\,[I(v_{in})\rightarrow(\exists v_{out})\,R(v_{in}, v_{out})] \quad (3)$$

ここで v_{out} は出力変数のベクトルを表わす．

プログラムの規模が少し大きくなると一般に，述語論理表現による入出力仕様記述は困難になる．Jackson 法では，入力と出力のデータの構造をそれぞれの Jackson 木で表現し，両者の部分間の対応をプログラマがつける．データを扱うプログラムの場合，このような仕様法の方がはるかに自然で容易なことが多い．ただし，自動プログラミング・システムに与える仕様形式としては，これだけでは不十分であり，たとえば，Jackson の電信文の問題における ovsize-word-count のような未定義な語（図-12 参照）は，システムが理解できるところまで rephrase してやる必要がある．その場合はやはり，述語や集合による表現，あるいは MODEL-II における表明文のような表現形式がとられることになる．

iv）計算の戦略指定

知識工学的アプローチの自動プログラミングにおいては，設計決定が後々のプログラム合成過程に context として影響を及ぼし，適用可能なプロダクション・ルールを限定する．この設計決定としては種々のものが考えられるが，たとえば合成するプログラムが関数型のものか副作用のあるものかの指定，集合データの分割方式の指定（すなわち tail-recursion 型か否か，など）などが考えられる．

論理レベルから物理レベルへプログラム・モデルの変換を行う際の指定については **3.3.3** で述べる．

3.3 プログラム合成の過程

3.3.1 問題の取得とモデル形成

i）知的インタフェース

前節で述べた問題の定義と仕様は自動プログラミング・システムのユーザが与える記述である．システムは，これをモデル化した内部表現に変え，内蔵する知識構造や問題領域に関する概念の中での位置づけをして初めて，その問題を正しく扱えるようになる．この意味でユーザとシステムとの間で，対象問題についての正しい共通の理解が樹立されなければならない．そのためには，システムは不明な点・不完全な点をユーザに質問したり，推論を働かせてその結果をユーザに確認してもらう必要がある．たとえば，システムは図-12 を問題の記述として与えられた場合，word と ZZZZ は単語の下位概念であり，この単語列は電報を構成するシーケンスの一種であって，ZZZZ は delimiter の役割を果している，などの理解が必要である．システム内蔵の知識を用いて協調的にシステム開発を支援することが目的のシステム (CONSUL[22] など）では，ユーザの問題概念の知識構造への挿入・分類機能により，システム側からの理解と支援を受けることが可能になる．このようにこれからのシステムではシステム側が積極的にマンマシン・インタフェースにおいて活躍すべきものであろう．

ii）問題のモデルの形成

入力データの構造記述文法の繰返し表現（＋や＊），あるいは再帰表現の式ごとに，セグメントを導入して対応をつける．このセグメントひとつひとつが(副)問題を表わす．問題は単純化された副問題の集まりとそれらの間の調整部分とに分割される．入力と出力との対応から，入力の何をフィルタし，何を残すか，計算で求めるべきものは入力の何であるか，などの（意味的）解析を経て，各セグメントの外部仕様が明確化される．

3.3.2 プログラムのモデルとその合成

i）プログラムのモデル

プログラムのモデルは次のいずれかの表現形式をとる．

① 有向グラフ

上位セグメントの内部論理構造を下位セグメントの集まりとそれらの間のデータフロー，コントロールフローで表現するものである．MODEL-II の有向グラフ（図-4 参照）や**図-13** のプログラムの論理構造モデル[29]がこれにあたる．後者は有限オートマトンの状態図表現と関連があり，データの構造記述とデータフローを通して対応がつく．このモデルはシーケンシャルなプログラムだけでなく，データフロー言語や PROLOG のような論理プログラムにも変換可能である．

② palm tree

トップダウンにゴール，サブゴールを展開する表現形式 (PMB 参照).

③ リスト構造の計算木

PQCC の TCOL 木がこの例で，変換などの処理に

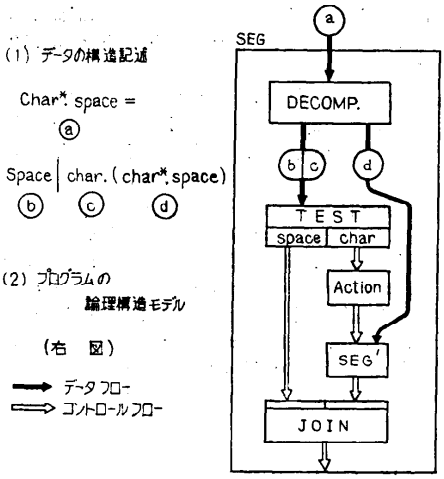


図-13 データの構造とプログラムのモデルの対応

好都合である.

①,②は incremental なプログラム合成に向いているといえる.

ii) プログラムのモデルの合成

セグメントひとつひとつのプログラムの論理モデルを合成する. このとき, 副作用のない関数型プログラムとして考えることができれば合成は容易となる.

セグメントの論理モデルは, 大略次の2つが融合してできたものと考えられる.

① 入力データ構造のトラバース部分

入力データの構造を記述している文法の解析に基づいて, モデルの制御構造の部分が作られる. 論理構造モデルでいえば, これは有限オートマトンの状態図の遷移表現に対応する. 解析結果に従って判定・セレクタ・再帰などのサブセグメントが導入され, データフローやコントロールフローの一部がつけられる.

② Action 部分の作成

状態図の遷移における action を与えるものであり, 入力と出力の対応から作られた問題のモデルを解析して作成される. 論理構造モデルの場合, action を表わすサブセグメント, および定数サブセグメントが挿入され, フローが追加される. この場合データフローは, MODEL-II におけるように, 因果関係を完全化するのに活用できる.

組み立てたモデルが矛盾のない完全なものかどうかは一般に論理的な検証が必要である. このためには, 同時に, セグメントの事前条件, 事後条件を作成し, サブセグメントのそれらと公理を用いて検証を行う. 公理の導入も上記の過程で同時に行えることが多い.

3.3.3 実現変換

i) プロセス間インタフェースの変換

論理レベルでプロセス・ネットワークとして設計された処理構造を, 実現変換指定に応じて物理レベルのプログラムにおとす. たとえば, プロセッサ資源の可用性に応じてプロセスの統合をはかるためにプログラム変換 (program inversion) を行ったり, プロセス間のデータの受渡しを, 呼び出しのパラメータによる (CSP[30], コルーチンなどの利用) か, queue などの共有データ構造によるかを定めて変換する.

ii) データ構造の具体化

抽象データタイプを用いれば, その実現のための具体データ構造と独立にアルゴリズムを定められることはよく知られている. 実現データ構造を定めて抽象レベルのプログラムを具体レベルのプログラムに精製するには, ある条件が満たされたときにパターンの inline 置き換えを行う自動化ツールがある[21]. また最適な実現データ構造の選択を行うシステムも発表されている[31]. 精製 (refinement) の自動化は困難な問題であり, これからの研究課題である.

4. おわりに

3. において, 問題解決法として知識工学的手法による方法をとりあげたが, もうひとつの代表的アプローチとして, 従来人工知能の分野でとられている形式的意味論に基づく方法がある. これに属するものとしては次のような方法がある.

- 定理証明による方法
  - 導出原理を用いる方法
  - 自然演繹法による方法[27]
  - Hoare の形式化に基づくプログラム論理[19]による方法[32]
- 変換による方法[33),34]

定理証明による方法のうちで, 導出原理を用いる方法は人間の思考法とはなじまない点があり, 自然演繹法による方法はより自然な推論形式であるが, 実際に行うには人間の発見的能力による案内が必要である. Hoare の形式化に基づくプログラム論理による方法の変形として, 定理証明法でなく means-ends analysis の手段を用いて, ヒューリスティックにプログラム合成を行っているシステムの例は 2.2.2 にあげた.

変換による方法は, プログラムの達成すべきゴール (事後条件) を, 場合分け, サブゴールへの分割, 帰

納の使用による再帰呼び出しの導入，プリミティブ・ステップの検出，計算のステップを飛び越してのゴールの passback などの手段を用いて達成する方式で，再帰プログラムの合成に用いられる．この方式の例として，Fellott による passing back pair の概念を用いたプログラム合成システム PROSYN がある[34]．これは，副作用のあるプログラムの場合に，Dijkstra の predicate transformer の考え方を利用した Waldinger の並立ゴール達成法[33]をさらに拡張したものである．このシステムで合成されたプログラムの例としては，配列中の要素を上下に動かすプログラムや線型連立方程式の解をガウス消去法で見出すプログラムなどがある．並立ゴールの問題は，副作用のあるプログラムで修正を施して新しいプログラムを得る場合など日常のプログラミングにおいてもよく起る問題である．

一般的にいって，定理証明による方法は，数学的厳密さはあるものの，ヒューリスティクスを利用しないと実用は困難であり，またその利用は容易でない．またこれらの方法が適用される対象の問題は，入出力仕様でゴールが与えられる小規模な問題が多く，ソフトウェア工学が対象とするような規模の問題に使えるかどうかは疑問がある．

おそらく，実際に役立つシステムは，理論的な美しいものよりも，計算機科学の進歩とともに蓄積されてきたプログラミングの原理や知識を取り込んだ泥臭いものでなければならず，しかも泥臭さに弱れることのない深い洞察力によって体系的に整理されたものでなければならない．

これからの自動プログラミングの中心的テーマとしては，抽象データタイプの処理[35]とその精製があげられる．ソフトウェア工学と人工知能を結ぶ努力[36]によって，自動プログラミングへの道が少しずつ開けつつあることを述べた．80年代にはこの方面の研究が大いに進展することを期待したい．

おわりに，日頃お世話頂く石井治部長，棟上昭男博士，本展望の書き方についてご教示頂いた鳥居宏次博士に感謝致します．

参 考 文 献

1) 田中幸吉：情報処理に関する学問体系，情報処理，Vol. 21, No. 5, pp. 539-549 (1980).
2) Jackson, M. A.: Principles of Program Design, Academic Press (1975), (鳥居宏次訳：構造的プログラム設計の原理，日本コンピュータ協会 (1980)).
3) Jackson, M. A.: Information Systems: Modelling, Sequencing and Transformations, 3rd Int. Conf. on Software Engineering, pp. 72-81 (1978).
4) Biermann, A. W.: Approaches to Automatic Programming, Advances in Computers, Vol. 15, Rubinoff, M. and Yovits, M. (Eds.), Academic Press, pp. 1-63 (1976).
5) 伊藤貴康：プログラムの自動作成，情報処理，Vol. 19, No. 10, pp. 993-1002 (1978).
6) Elschlager, R. and Phillips, J.: Automatic Programming (a section of the Handbook of Artificial Intelligence, Barr, A. and Feigenbaum, E. A. (eds.)), Stanford Univ. HPP-79-24 (1979).
7) Tyugu, E. H.: Towards Practical Synthesis of Programs, Information Processing 80, Lavington, S. H. (ed.), North-Holland, pp. 207-219 (1980).
8) Alford, M.: Software Requirements in the 80's; from Alchemy to Science, ACM Annual Conf., pp. 342-349 (1980).
9) Hammer, M., Howe, W. G., Kruskal, V. J. and Wladawsky, I.: A Very High Level Programming Language for Data Processing Applications, CACM, Vol. 20, No. 11, pp. 832-840 (1977).
10) Ellis, C. A. and Nutt, G. J.: Office Information Systems and Computer Science, Computing Surveys, Vol. 12, No. 1, pp. 27-60 (1980).
11) Floyd, R. W.: The Paradigm of Programming, CACM, Vol. 22, No. 8, pp. 455-460 (1979).
12) Prywes, N. S.: Automatic Generation of Computer Programs, Advances in Computers, Vol. 16, Rubinoff, M. and Yovits, M. (Eds.), Academic Press, pp. 57-125 (1977).
13) McCune, B. P.: Building Program Models Incrementally from Informal Descriptions, Stanford Univ., AIM-333 (1979).
14) Green, C.: The Design of the PSI Program Synthesis System, 2nd Int. Conf. on Software Engineering, pp. 4-18 (1976).
15) Duran, J. W.: Heuristics for Program Synthesis Using Loop Invariants, ACM Annual Conf., pp. 891-899 (1978).
16) Duran, J. W.: Automatic Program Synthesis via Synthesis of Loop-free Segments, NCC, pp. 1059-1062 (1979).
17) Cattell, R. G. G.: Automatic Derivation of Code Generators from Machine Descriptions, ACM Trans. on Prog. Lang. and Systems, Vol. 2, No. 2, pp. 173-190 (1980).
18) Wulf, W. A.: PQCC: A Machine-relative Compiler Technology, 4th COMPSAC, pp. 24-36 (1980).
19) Hoare, C. A. R.: An Axiomatic Basis for Computer Programming, CACM, Vol. 12, No.

10, pp. 576-580, 583 (1969).
20) Rich, C. and Shrobe, H. E.: Initial Report on LISP Programmer's Apprentice, IEEE Trans. on Software Engineering, Vol. 4, No. 6, pp. 456-467 (1978).
21) Cheatham, T. E., Townley, J. A. and Holloway, G. H.: A System for Program Refinement, 4th Int. Conf. on Software Engineering, pp. 53-62 (1979).
22) Lingard, R. W.: A Software Methodology for Building Interactive Tools, 5th Int. Conf. on Software Engineering, pp. 394-399 (1981).
23) Gerhart, S.: Knowledge about Programs: a Model and a Case Study, Int. Conf. on Reliable Software, pp. 88-95 (1975).
24) Hansen, P. B.: The Architecture of Concurrent Programs, Prentice-Hall (1977).
25) Hughes, J. H.: A Formalization and Explication of the Michael Jackson Method of Program Design, Software-Practice and Experience, Vol. 9, pp. 191-202 (1979).
26) Burge, W. H.: Recursive Programming Techniques, Addison-Wesley (1975).
27) Hansson, A. and Tarnlund, S.: A Natural Programming Calculus, 6th-IJCAI, pp. 348-355 (1979).
28) Claybrook, B. G. and Wyckoff, M. P.: Module: An Encapsulation Mechanism for Specifying and Implementing Abstract Data Type, ACM Annual Conf. pp. 225-235 (1980).
29) 間野暢興：プロセス・ネットワーク・モデルに基づくプログラム合成について，情報処理学会第22回全国大会 3F-6 (1981).
30) Hoare, C. A. R.: Communicating Sequential Processes, CACM, Vol. 21, No. 8, pp. 666-677 (1978).
31) Rowe, L. A. and Tonge, F. M.: Automating the Selection of Implementation Structures, IEEE Trans. on Software Engineering, Vol. 4, No. 6, pp. 494-506 (1978).
32) Buchannan, J. R. and Luckham, D. C.: On Automating the Construction of Programs, Stanford Univ. AIM-236 (1974).
33) Manna, Z. and Waldinger, R.: Synthesis: Dreams⇒Programs, IEEE Trans. on Software Engineering, Vol. SE-5, No. 4, pp. 294-328 (1979).
34) Fellott, R.: Synthesizing Recursive Functions with Side Effects, Artificial Intelligence, Vol. 13, pp. 175-200 (1980).
35) 鳥居宏次，二木厚吉，真野芳久：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).
36) Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling, ACM SIGART Newsletter No. 74-SIGMOD Record, Vol. 11, No. 2-SIGPLAN Notices, Vol. 16, No. 1 (1981).

(昭和56年6月11日受付)