総　論

1. 自動プログラミング

1.3 知的プログラミング環境†

—プログラム理解を中心に—

上　野　晴　樹††

1. はじめに

プログラミング環境は，プログラマがコンピュータを用いてプログラミング作業を行うときの対話型環境を意味し，ヒューマンインタフェースとしての操作性の改善だけでなく，プログラム生産性の向上という重要な目的をもっている．また，人工知能においても，人間にとって親和性の高いヒューマンインタフェースあるいは知的環境は，きわめて興味深い研究対象である．したがって，これはソフトウェア工学と人工知能の両方における重要な研究課題であると考えられる．ソフトウェア工学の分野では，この問題は，TSS システムが普及し始めたころから意識されるようになったと思われるが，当初はソフトウェア開発に必要な各種ツールの整備に焦点が置かれていた．環境として取り扱われるようになったのは，80年代になって，パソコンの出現によりプログラミング作業がいわゆるペーパーワークから対話型作業へ変わり，さらに高機能ワークステーションの利用へと進歩するにつれて，その重要性がますます増してきたことにより，単にツールの整備という概念では不十分になってきたからであると考えられる．

一般に，プログラミング環境と呼ばれるものは，エディタ，コンパイラ，トレーサなどからなるプログラム開発用各種ツールと，これらの使い方を支援するための電子化マニュアルおよび HELP, GUIDE, BROWSER などの情報提示機構，およびビットマップやマルチウィンドとマウスを組み合わせた使い勝手の良い対話型端末装置などを統合化したシステムを指している．この種のプログラミング環境の出現によりプログラミング作業は非常に改善されたが，AI 技術の進歩は，より一層の改善の可能性をもたらした．ごく簡単にいえば，いわゆる従来型プログラミング環境はプログラマの意識的操作によって補助的に働くに止まっていたが，AI の応用により，より積極的にプログラマを支援できるという可能性が明らかとなってきた．各種の専門知識とこれを利用する推論とによってこのことが可能となる．プログラマの勘違いを指摘し，必要な情報を教え，プログラムの論理的ミスを検出し，プログラミングをスムーズに行えるようにする．さらにドキュメント管理やプログラムの保守の支援もある程度できつつある．ただし，いわゆる“知的”という言葉の定義は厳密には与えられていない．したがって，単に知識ベースを用いただけのものから，非常に高度な推論あるいは理解能力をもつ（ことを目標とした）ものまでさまざまである．単に知識ベースを用いているだけのものは知的と呼ぶには相応しくないが，それでも，原理的に発展の可能性を秘めているとみることはできる．ここでは，なんらかの意味で AI 技術を応用したもの，もしくは専門知識に基づく推論機能をもつプログラミング環境を，知的プログラミング環境と呼ぶことにしよう．

さて，プログラム開発への AI 応用には2つの対照的なアプローチがあると考えられる．プログラムの自動生成（合成）とプログラミング支援である．知的プログラミング環境の研究は後者に属する．前者は，プログラムを合成型問題としてとらえ，仕様を入力してプログラムを出力するシステムの実現を目標とした研究である[22),23)]．プログラミングの自動化はプログラミングの研究の歴史を通じてもっとも挑戦的テーマであ

† Knowledge Based Intelligent Programming Environment by UENO Haruki (Department of Systems Engineeering, College of Science and Engineering, Tokyo Denki University).

†† 東京電気大学理工学部経営工学科

ったし,今後もあり続けるであろう．ただし,非常に魅力的ではあるが，同時にきわめて困難な課題でもあるので,実用に役立つシステムの実現は当分,簡単かつ限定された応用を除き，期待できないように思われる．一方，後者は，プログラマの作業を支援する仕事に徹しており，プログラム開発そのものはあくまでもプログラマの仕事であるという考え方を採っている．この点で自動生成に比べて地味であり魅力に乏しく迫力に欠けるように思われるかも知れないが，決してそうではない．プログラミングに関する各種知識とその利用，プログラムの意味理解に基づく論理チェックや学習，プログラマの認知モデル，知的 CAI の機能など，AI およびソフトウェア工学の両方にとって重要かつ魅力的な課題を多く含んでおり，自動合成とも関連のある要素が少なくない．また，思考実験的色彩の強いソフトウェア開発や，自動化しにくい多くのソフトウェア開発および保守管理においては，後者が重要であり続けるはずである．

本論では，まず AI とプログラミングとの係わり合いについて述べ，次に知的プログラミング環境に関する諸問題，すなわちプログラミングにおける知識，プログラムの意味と理解，知的プログラミング環境へのアプローチなどについて論じ，続いて研究動向を概説し，最後に今後の展望を述べる．特に，知識とそれを用いたプログラム理解に焦点をあてることにする．ただし，著者の理解がいまだ未熟でありかつサーベイ不足による突っ込みの浅さを免れないので，十分に意図するところを表現できないもどかしさがある．その点はご容赦願いたい．

2. プログラミングと AI との係わり合い

AI (人工知能) は大きく2つの側面をもっている．1つは，人間の知能の原理の究明を試みるという認知科学的側面であり，もう1つはコンピュータをより有用なツールにすることを目標とする工学的側面である．前者としての AI 研究ではコンピュータは知能の(認知科学的) モデルに関するアイデアの実験用ツールとして用いられ，副産物としてさまざまな実用性のあるソフトウェア技法やツール，すなわち記号処理や記号推論技法さらには知識ベースシステムやエキスパート・システム，などが開発された．

後者は，これらの研究成果を応用することによって，これまでコンピュータが苦手としてきた種類の情報処理である情報の意味的理解や高度な専門的判断などを行うこと，つまりコンピュータをより有用なツールとすることが目的である．情報処理においてこの種の仕事は，従来人間が行う役割とされていた．この種のツールを使えば知的システムの実現の可能性があるということが重要な点である．ただし，いまだこの種の問題をどうコンピュータで処理するかに関するわれわれの理解と経験が乏しいことが理由で，実用的な知識システムの開発例は少ない．

ソフトウェア開発は，AI の観点からみると後者に属する問題分野であるといえよう．つまり，エキスパート・システムとしての色彩が強い問題である．ただし学問的研究として解決されなければならない問題がいまだ多く残されており，実用化も従って今後の課題である．そこで，これらのことを念頭に置いて AI とソフトウェア開発との係わり合いについて若干の考察を行ってみよう．これには，以下に述べるように3つの形態がある．

第1は，AI のアプローチをソフトウェアとしての新しいプログラミング・パラダイムとしてみるものであり，AI システムを実現するための知識表現言語を新しいタイプのプログラミング言語としてとらえるものである．第2は，問題解決の対象であるプログラム開発を合成型エキスパート・システムとしてとらえる見方であり，そして第3がプログラマを支援するための知的プログラミング環境を実現することを目標とするもの，すなわち本論のテーマである．以下にそれぞれについて簡単に議論しよう．

第1のプログラミング・パラダイムについては，次のようなことがいえる．ソフトウェア工学はソフトウェアの生産性と信頼性の向上を目的としており，これを実現する手段としてプログラミング言語の占める比重はきわめて大きい．なんとなれば，人間のアルゴリズムをコンピュータのアルゴリズムに書き換えるための言語がプログラミング言語であると考えてよく，よい言語を用いればこの変換作業がスムーズに行え，作業能率だけでなく信頼性も向上するからである．当然，逆のこともいえる．また，問題の性質に応じて適切な言語が決まる．この点で，従来型汎用プログラミング言語の特徴は手続き型言語であることである．この種の言語は，技術計算や事務データ処理の分野に適している．一方，人間がもつ知識およびそれに基づく推論は手続き的よりも多分に宣言的であると考えられている[1),2),25)]．手続きはローカルに存在するようである．したがって，これを従来型言語で記述するには，

一度手続き型のアルゴリズムに変換しなければならないので，容易なことではない．これに対して，知識表現言語は宣言的であるので，いわゆる人間のアルゴリズム（知識）を表現しやすいわけである．当然，手続きが向いている部分に対しては従来型言語を使うべきである．これは，プログラミングにおける新しいパラダイムとみなすことができる．つまり，従来の手続き型プログラミング言語によるプログラミングでは，“データ構造＋アルゴリズム＝プログラム”というパラダイムであったが，知識表現言語によるプログラミングにおいては，これに代わって，“知識ベース＋推論機構＝プログラム”というパラダイムが提供されたといえよう．しかも，一般に推論機構の部分はシステムがもっている場合が多いので，“知識ベース＝プログラム”であるとみなせる．つまり，知識表現言語は知識ベースを定義するための宣言用プログラミング言語である．いずれにせよ，知識表現言語の出現はプログラミング言語に対する考え方に変革をもたらしたばかりでなく，コンピュータの応用範囲を質的に拡大したといえる．

第2の合成型エキスパート・システムであるという点は，以下のようなことである．プログラム開発は合成型問題である．これまでに開発されたエキスパート・システムの大部分は分析型であり，合成型問題は本質的に困難な課題を含んでいる[2]．それでも，3次元 CAD，LSI 設計や比較的簡単なソフトウェア自動合成などへの応用の試みをとおして，合成型エキスパート・システムの輪郭が少しずつ明らかとなってきている．合成型問題は，与えられた要求を満たすシステムを，一定の拘束条件の下に，部品の組み合わせとして，最適のものを合成する問題である．部品の種類は少なくても，可能な組み合わせの数が膨大となるので，問題空間が非常に大きくなり，解決が困難であるというのが特徴である．探索空間を狭める働きをするのが拘束条件であり，これが強ければ強いほど解決が容易となる．また，解の候補が複数個あるときは評価関数を用いて最適のものを選択する必要がある．このように，合成は部分的にみると選択である．1つの選択は，次のステップへの拘束条件を生成し，処理が進むに従って拘束条件も強くなっていく．拘束条件が強いほど自動設計は容易となり，設計の手順が明確になっているほど自動合成システムの開発は容易となる．部品合成によるプログラム開発はこの点で可能性の高いアプローチであると考えられる[22),23)]．ただし，モジュール化に関する標準化が実現できていない現時点においては，いまだ実用システムの達成は当分困難であろうと思われる．

第3のプログラミング支援エキスパート・システムとは，次に述べるようなことである．すなわち，プログラム自動生成ではなく，知識ベースを応用してプログラミング環境を知的なものにし，プログラマの作業を支援することによってプログラムの開発能率や信頼性を向上させるというアプローチも重要なものである．自動合成に比べて環境の改善の方は，現在すでに利用されているプログラミング環境の改善であるから，いろいろなレベルでそれなりの効果が期待できるはずである．プログラミング言語，プログラミング技法やアルゴリズムに関する各種の専門知識を利用して，プログラムのエディットをガイドし，関連情報を提示し，作成されたプログラムを意味理解に基づく論理チェックなどを行うことによって，間接的にプログラム開発や保守作業の能率や信頼性を向上させることを狙ったものである．また，知的プログラミング環境の延長として，プログラム開発における各段階，すなわち仕様作成，プログラミング，プログラム更新，ドキュメント管理を統一的に支援するシステムの実現も長期的には可能であろう．この段階になると，プログラム自動合成と知的プログラミング環境とが統合化されるべきものである．

3. 知的プログラミング環境の考え方

プログラムはエディタをとおしてコンピュータに入力されるから，エディタはプログラミング環境において中心的機能をもつ．しかしながら，現在の TSS 環境下における通常のエディタは，プログラミング環境として問題がある．ここではこの点について考察し，知的プログラミング環境を知的エディタを中心に置いて実現する考え方について検討してみよう．

現在広く利用されているエディタは，テキスト・エディタと構造エディタである．前者は，入力されたプログラム文を単なる文字列として操作するだけであるので，ユーザは自分自身で構文を管理しなければならない．反面，汎用性が高いという利点がある．構造エディタ（構文エディタ）は，特定の言語の構造にあわせて設計されているので，その言語特有のいろいろ便利な機能を含み，プログラミング環境としては，より優れている．ただし，特定の言語にしか利用できないという制限は当然ある．たとえば PASCAL を対象と

した構造エディタはPASCALの構文木をエディタのGUIDEおよび構文チェック機構として有効に活用しており，プログラマの負担はその分だけ軽減される．ただし，これらはいずれもプログラミング環境として十分なものではない．そこでより機能の高いエディタとはなにかを明らかにするために，プログラマが対話型エディタを用いてプログラミングを行うときの問題点のいくつかを示そう．なお，これは大学における学生や研究者が，エディタを用いてプログラミングを行う状況を想定している．

まず，エディタを用いてプログラミングを行うときに遭遇する問題点としては，1) エディタやプログラミング言語に関する予備の勉強を強いられる，2) プログラミング中にど忘れが起こる，3) 言語仕様を確認したい，4) プログラムの書き方を確認したい，5) 次になにをすべきか分からない，6) ほかのプログラムを利用したい，7) プログラムのエラー箇所とその理由を知りたい，8) なにが分からないかが分からない，などがあげられる．これらはいずれも，エディタそのものの問題ではなく，エディタ環境の問題である．

一方，これらの問題あるいは疑問が生じたときに参照する資料類は，1) エディタ・マニュアル，2) プログラム言語入門書，3) プログラム入門書，4) プログラム例題集，5) プログラム言語仕様書，6) BNF記法による言語定義，7) プログラム集，などであり，いずれも印刷された資料である．

エディタを用いてプログラミング中に前述のいずれかの問題に出会ったら，一時作業を中断させて，これらの資料の中から必要な情報をさがさなければならない．資料は年々増加しつつあるので，必要な情報を，もれなく，適切にさがし出すことは困難である．しかも，次になにをするべきか分からないなどという疑問には，単にテキスト情報の提示だけでは応えられない．ここで，もし，相談者，助手，もしくは先生がかたわらにいれば，特に初心者プログラマは大変助かるであろう．専門プログラマでも類似の問題は常に生ずるので，上記のマニュアル類は欠かせない．したがって，専門家にとっても助手的な存在は有効である．

知的エディタの重要な機能の1つは，人間に代わってこれらの支援業務を代行するシステムにある．チュータはプログラマの疑問や混乱を適切に理解し，どの情報を提示すべきかを判断し，実行する．あるいは助言する．そのためには，プログラマの能力を理解する能力も必要となる．したがって，知的エディタ環境のイメージは「人間（プログラマ）はディスプレイ・ターミナルをとおしてコンピュータに対して仕事を伝え，コンピュータはディスプレイ・ターミナルをとおして人間の作業を観察し，彼がなにを考えているのかを理解しながら助けようと努力する」，というものであろう．そこで，なにが知的エディタであり，そのためにはどんな能力をもつ必要があるのかについて，PASCAL言語を例に論じてみよう．

第1に，コンピュータがプログラミング言語を知っていることが必要である．つまり，たとえば，PASCALの文法，PASCALの使い方，PASCALによるプログラム例を知っている必要がある．第2に，コンピュータがPASCALを理解できることが必要である．レベルの低い方から高い方へとならべると次のようになる．単一のPASCAL文を理解できる．複数のPASCAL文を理解できる．ブロック構造を理解できる．プログラム構造を理解できる．そして，プログラムの意味を理解できる．知的な支援の能力を高めるには，プログラムの意味を取り扱う機能が不可欠である．第3に，コンピュータが仕事を理解し，プログラマを支援できることが必要である．プログラマが開発しようとする応用プログラムの対象である仕事を理解できれば，支援の形態は飛躍的に向上する．たとえば，連立一次方程式を解くプログラムを作成しようとしていることが分かれば，助手の仕事はより概念的かつ高水準のものになることができるのと同様である．そのためには，トピックを理解できること，それに関する対話ができること，プログラムの方法もしくはプロトタイプ構造を知っていること，などが必要となる．またこのためにはプログラムの意味理解能力が前提となる．

第4は，コンピュータがプログラマを理解できることが必要である．これは前述のような機能である．良いチュータは生徒（プログラマ）を適切に理解できる．システムがこの能力をもつには，いわゆる“プログラマ・モデル”を内部に構築できることが必要となる．これはCAIとも共通のテーマである．もしこの能力をもたせることができれば，無駄な情報を呈示し過ぎることのわずわらしさをさけ，かつ，欲しい情報を出してくれないもどかしさを解消できよう．

以上は知的エディタを中心に置いた知的プログラミング環境のイメージをスケッチしたものであるが，以後プログラミングにおける知識，意味，意味の理解および理解に基づく論理チェックについて議論しよう．

4. プログラミングにおける知識

プログラミングに関する知識は，プログラムの作成，理解，テスト，変更などにおいて必要不可欠であり，したがって知的プログラミング環境もこれをもつ必要があることはいうまでもない．逆にいえば，この知識の質と量が，環境の知的能力を左右するといえる．現在までのところ，このような問題に関する研究の歴史は非常に浅く，研究例が少ないために，まだ十分な考察はできない．特に，我が国ではこの種の研究の重要性は正しく理解されていないようであり，そのためにプログラム合成に比べると研究がはなはだ低調であり，いまだほとんど行われていないといえよう．したがって，ここではまず，プログラミング知識に関して著者の個人的な考察を行い，次に3つの代表的アプローチを紹介するというやり方を採ることにしよう．

4.1 プログラミング知識について

初級プログラマに比べて上級プログラマは多くのプログラミング知識をもっていると考えられる．この知識をも用いて，上級プログラマは，与えられた問題を分析し，アルゴリズムを作り，プログラムを設計し，プログラミング言語でコーディングし，書かれたプログラムを理解し，論理的ミスを検出し，プログラムミスを修正し，プログラムの変更を行うなどのことができる．そこで，プログラミング知識とはなにであるか，初級プログラマと上級プログラマとはプログラミング知識に関してどう違うか，認知科学的にみてプログラミングという問題解決はどうモデル化できそうか，などの問題を議論してみよう．

まず，大規模で複雑な実用のデータ処理システムをチームで開発する場合と，小規模な問題を解く場合もしくは個人が実験システムとしてプログラムを作成する場合とは，重要な役割を担う知識の種類が異なっているであろうことは，比較的容易に理解できよう．前者では，ユーザから適切な要求を引き出し，それに基づいてシステム全体の枠組みを設計することが最も重要であり，特定の問題分野での永年の開発経験をもついわゆるシステムエンジニアの専門知識がきわめて有用である．また，大規模な問題は各々が典型的な小規模な問題の合成で置き換えられ，これに基づいて具体的設計を行い，次に特定のプログラミング言語でコーディングする作業へと移る．このようなケースでは，全く新しい問題を除いては，大抵分野ごとに定型化しやすいことが多く，このためのノウハウを十分にもっている専門家がシステムエンジニアであるといえる．これに対して，小規模な問題では要求の分析や枠組みの設計もさることながら，プログラムの開発作業はプログラミング環境を用いて対話的にある程度の試行錯誤によって行われる部分が多い．したがって，具体的アルゴリズムの設計，コーディング，エディット，デバッグ，テストおよび変更などの作業が主体であり，これに必要なさまざまな知識がプログラマに要求される．本論では，システムエンジニアが行っているような大規模な業務およびそれに必要な専門知識については，議論の対象からは外すことにし，対話型プログラムエディタを中心とするプログラミング環境と直接の関連をもつ範囲に絞って議論することにする．これは，大規模なデータ処理システムの開発においては，後半の段階すなわち具体的なプログラムの作成において必要となる事柄である．（ただし，将来は要求分析，仕様作成，全体設計なども支援するための環境が実現するであろう．）なお，これらに関連した知識を以後プログラミング知識と呼ぶことにして議論しよう．

プログラミング知識は，まず，プログラミング技法に関する知識とプログラミング言語に関する知識とに分類できそうである．両者は関係があるが，お互いに独立してもいると考えられる．これは，次のような事実から推察できる．たとえば，複数のソーティング技法と複数のプログラミング言語を知っている人は，組み合わせの数だけの専門知識を羅列的にもっているのではなく，特定の技法と特定の言語が指定された時点で表現法（コーディング）を考えるはずである．また，新しい言語を教わると，その言語を使って指定された技法のプログラムを書くことができる．そこで，この両者の知識構造や性質などについて簡単に考察してみよう．

プログラミング技法に関する知識は，細かな技法的知識から抽象的な概念的知識までが，いくつかのレベルに階層化されているものと思われる[5),20)]．具体的な細かい技法に関する知識が最下位にあり，抽象的概念知識が最上位にある．これらを最下位から最上位に向けて，基本操作要素技法レベル，基本データ処理技法レベル，抽象データ処理アルゴリズム・レベル，問題解決概念レベル，とでも呼ぶことにしよう．これらの各レベルの知識は以下に述べるようなものである．最下位の知識つまり基本操作要素技法の知識とは，変数への値の代入，2つの値の比較，条件分岐などの基本的

データ操作や制御に関する知識であり，どんな複雑なアルゴリズムでも結局はこれらの基本的操作の組み合わせで実現できる．次のレベルの基本データ処理技法の知識とは，基本的データ処理技法に関する知識，すなわち2つの変数間の値の交換法，個数の計数法，配列の中の最大値の求め方，などが含まれる．経験を経たプログラマは，これらの手法の組み合わせでプログラミングすると思われるから，プログラムが比較的整然としており理解しやすい．その上位の抽象データ処理アルゴリズム・レベルの知識とは，配列の積を求める手続き，各種の整列化の手続きなどのような，代表的データ処理アルゴリズムに関する実現手続きの知識をいう．このレベルの知識は，個々の知識単位が大きいので，大まかな枠組みだけが記憶されており，詳細はテキストなどから引用するように，付加情報が付けられている．最上位の，問題解決概念レベルの知識とは，配列の積や整列化のような代表的問題解決技法の概念やその抽象的アルゴリズムの情報である．また，このレベルの知識の特徴は，宣言的であり，またある概念を内部に含むようなより大きい単位の概念も存在し，それらの間の関係が属性情報として管理されていることである．あるレベルの技法でその上位レベルの技法が表現でき，逆にそのレベルの技法はすぐ下位のレベルの技法によって表現できる．このようにして，記憶すべき知識の情報量がおさえられている．

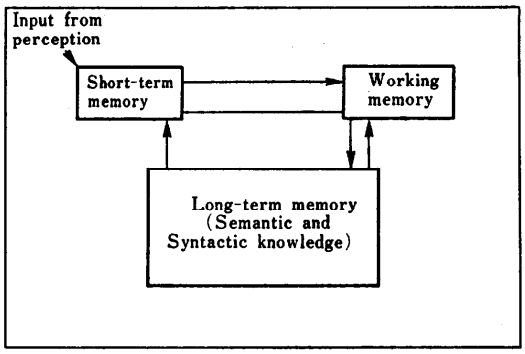
一方，プログラミング言語に関する知識は，2段の階層構造をしていると思われる．すなわち，いろいろなプログラミング言語に共通の言語要素である基本的プログラム文に関する概念的知識が下位にあり，上位には個別のプログラミング言語に関する知識が存在する．基本的プログラム文に関する概念的知識とは，VAR, TYPE, ARRAY, INTEGER などの変数宣言文，代入，算術演算，比較演算，論理演算，インデキシング，型変換などのデータ操作文，および IF 文，WHILE 文，UNTIL 文，CASE 文，FOR 文，PROCEDURE 文，などの制御文などに関する概念的知識である．これらは各プログラミング言語から独立の，手続き型プログラミング言語に共通の基本言語要素であり，多少形を変えて各プログラミング言語の中に採り入れられている．これらの情報と関連した情報，すなわち，構文規則，意味規則，および使い方や使用例などのヒューリスティックスも関連知識として記憶されている．1つのプログラミング言語を修得すれば，2番目以降の言語の修得が容易になること，複数の言語を知っているプログラマは時々表記上の混乱を起こすことがある，経験豊富なプログラマは短時間で洗練されたプログラムを書くことができるなどの事実から，このような記憶構造が推察できる．なお，PROLOG や LISP, SMALLTALK などのように言語的性格の異なるプログラミング言語に関しては，それぞれ類似性の高い言語グループを形成して，同様な知識構造で管理されていると考えられる．ただし，以上の知識構造モデルについては，今後認知科学的実験によって検証する必要がある．

4.2 プログラマの認知モデル

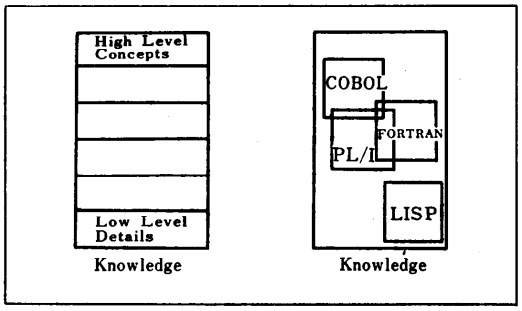
知識や理解は本来認知科学がベースとなる．そこで，正面からこの問題に取り組んでいる例として，プログラマのプログラミングという問題解決を認知科学モデルとしてモデル化する試みを行っている研究例を紹介しよう．Shneiderman と Mayer[5] は，プログラミングおよびプログラム理解におけるプログラマの認知科学的振る舞いのモデル化を行っており，このモデルと関連づけてプログラミング知識やプログラミング作業を把握することを試みている．以下がその概要である．

Shneiderman らは，プログラミングにおけるプログラマの問題解決の5つの基本タスク，すなわち，プログラムの作成，与えられたプログラムの理解，論理エラーの検出およびプログラミング知識の学習を，問題解決に関する人間の情報処理モデルと，Polya の一般的問題解決における4ステージのモデルとを組み合わせて表すモデルを提案し，実験をとおしてさまざまな角度から考察している．図-1 に，プログラミング知識を長期記憶 (long-term memory) にもつ人間の問題解決情報処理モデルを示す．図で，プログラマが注目している問題部分が外部からの情報として，容量の小さな短期記憶 (short-term memory) に一時的に入力され前処理が行われた後，問題解決処理のために作業記憶 (working memory) へ移される．ここで，長期記憶から必要な知識が参照され，上記に述べたプログラミングに関するタスクを処理する．その結果，新しい知識は長期記憶へ学習される．

彼らの，プログラミング知識の構造は，前節で述べた考え方に近いものである．すなわち，プログラマがもっているプログラミング知識はプログラミング技法に関する知識とプログラミング言語に関する知識とに分かれて長期記憶に記憶されるというものである．前者を意味論的知識 [semantic konowledge] と呼び，



(a)



(b) Long-term memory

図-1 プログラミング知識の長期記憶モデル(b)と，一般的記憶モデル(a).

後者の統語論的知識 [syntactic knowledge] とは独立なものとして構造化され，記憶される．ここでは，意味論的知識の方はプログラミングの経験や教科書をとおして獲得され，これらが抽象化され，ほぼ前節で述べたような概念構造に階層化されて記憶される．一方，統語論的知識の方は，類似性の高い言語の知識が図のように部分的に重複したフラットな構造で記憶される．ここで注目すべき点は，彼らは，意味論的知識は，知的要求や意味のある学習をとおして概念化され，すでに存在する意味論的知識と統合化されるが，統語論的知識の方は機械的なものとして記憶され，意味論的知識を管理する機構の中にはうまく統合化されないと指摘していることである．このことは，統語論的知識は単に追加されるだけであるので，新しく修得した言語の知識とすでに修得している言語の知識とを混同することが多い事実が示している．たとえば，すでに PASCAL を修得している学生に FORTRAN を教えると，間違って代入記号にコロンを付けたり，文の最後にセミコロンを付けたりすることが多い．

このモデルは具体的な記憶構造まではいまだ言及していない．なお，このモデルに基づくプログラム理解に関するモデルについては後述する．

4.3 プ ラ ン

知的プログラミング支援やプログラム理解に関する研究においては，プログラミング知識に対して，プラン [plan] という概念がよく使われる[7)-10)]．この概念は，認知科学における問題解決の基本モデルをプログラミングにおける問題解決モデルに応用したものと思われる．しかしながら，同じプランという用語を使いながら，それが意味する内容は研究者によってかなり異なっている．これは，プログラミングにおける問題解決のモデル化，特にプログラミング知識に関する考え方の違いによるものであるようである．ただし，プランという概念を用いている場合には，これがシステムの中で最も重要な役割を担っている点については共通である．そこで，プランの概念や，この概念に基づくプログラミング知識について概説することにしよう．

プランという概念は，日常生活の中での一般的問題解決において使われる計画や案とほぼ同一のものであると考えてよいが，プログラミング知識のモデル化においてはこれらとは多少ニュアンスが異なるので，ここでは専門用語としてこのまま使うことにする．さて，Polya[26)] は，問題解決が次の4つの段階から構成されていると提案している．それらは，

1) 問題の理解
2) プランの考案
3) プランの遂行
4) 結果の評価

である．問題解決者は，まず第1の段階である問題の理解において，与えられた問題はなんであるか（初期状態）ということと達成すべき目標はなんであるか（目標状態）を決定する．これに基づいて，次の段階では，解決のための一般的戦略をプランとして発見することが必要となる．このプランは，次の第3段階で，実行のための具体的な特定の行為過程に翻訳される．そして，最後の段階では，このプランが正しく機能するかどうかの検証が行われる．

この枠組みで考えると，プログラム作成とはプログラミングという手段によって与えられた問題を解決するためのプランを作成する仕事であると見なすことができる．ここで，プランとは抽象レベルでのプログラムであると考えてよい．さらに，プランはプランの断片を合成して構成されたものであると考えることもで

きる．これは，DENDRAL における分子構造が，分子構造の断片の合成によって構成されているのと同様な考え方である．また，プランを考案するにはプランの知識が必要である．この知識をプランと呼ぶこともできよう．つまり，プランはプログラムの抽象表現であり，あるいはそれを構成するための断片であり，かつまた知識でもある．プログラム作成もしくはプログラミング知識という観点からプランという概念をとらえると，このように幾つかの解釈が成り立つ．これが，プランという概念の使い方の違いを生じさせた理由ではないかと思われる．

プランをプログラムの抽象表現として位置づけている例が Rich らの PA (Programmer's Apprentice)[6)~8)] におけるプランであり，プログラミングの断片知識として捉えている例が Soloway らの PROUST[9),10)] におけるプランである．PA のプランは，データ・フローとコントロール・フローを統合化した，言語独立な抽象プログラム表現であり，外部からエディタを通して入力されたソース・プログラムはプラン表現に変換されて管理される一方，プランをソース・コードに変換して表示させたり，プログラム作成にプランを活用することや，プランそのもののエディットなどもできるようになっている．なお，ここではサブルーチンがプランの表現単位であると考えてよいようである．PA は，このような機構によってプログラマのプログラミング作業を支援する助手として働く知的プログラミング環境であり，そこでプランが中心的役割を演じていることは開発者自身が強調しているところでもある．ただし，このシステムではプランをプログラムの抽象表現として位置づけてはいるが，認知モデルとして位置づけているわけではないので，私自身はそれほど興味をもてない．また，開発者の主張と異なり，PA のプラン表現は人間にとって理解しやすいとも思えない．むしろ，Soloway らの PROUST におけるプランの方が AI の観点からははるかに意義があるように思われる．

PROUST は，プログラム理解によって初心者によるプログラミングを支援することを目的とした知的プログラミング支援システムである．そのために，正しいプログラムを理解するばかりでなく，間違ったプログラムをも理解できることに特徴がある．つまり，初心者が侵しやすいミスや勘違いとその心理学的原因をも推定し，ミスの種類，原因に加えて，修正法を理由を添えて提示できるように工夫されている．このためには，プログラマが入力したプログラムから彼の意図を理解できることがポイントとなる．このようなことを可能とするために，Soloway らは上級プログラマと初級者との違いを認知科学的実験手法で分析している[9)]．その成果が，彼らのいうプランとそれに基づくプログラム理解に結実したものである．以下に PROUST におけるプランの概要を示す．

彼らは，“プログラミング・プラン（以下単にプランと略す）とは，プログラムの断片でありプログラミングにおいて典型的なアクションのシーケンスを表現するものである”と位置づけている．図-2 に，入力された数値データの平均値を求める PASCAL プログラムとその中で用いられている4つのプランを示す．この例では，データの数をカウントするためのカウンタ・プラン，カウントに同期させて合計値を求めるためのランニング・トータル変数プランとこれを制御するためのランニング・トータル・ループ・プラン，および正常な終了条件でループから抜け出すためのバリッド・リザルト・スキップ・ガードなどと呼ばれるプランが使われている．最初の2つが変数プラン，後の2つが制御プランの例である．

この例に示されるように，プランはプログラミング知識の一種であり，コーディングにおいてプログラムの意図を実現するための手続きあるいは戦略として利

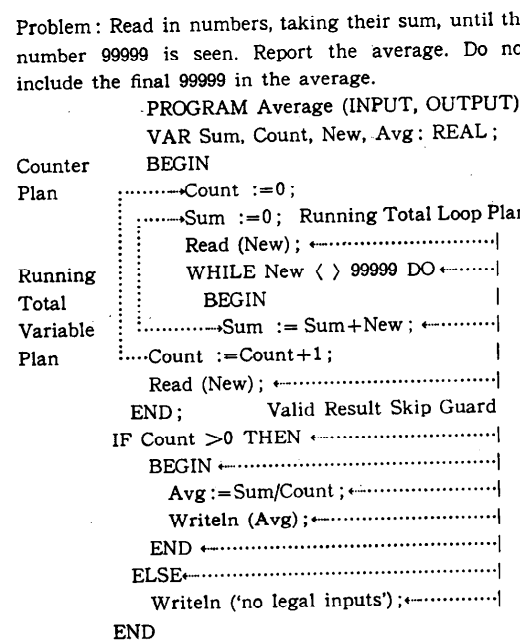


図-2 PROUST のプログラミング・プランの例

用される．エキスパート・プログラマは，いろいろな場面ごとに有用となるこのようなプランをもっており，プログラミングに際して積極的に活用しようとしていると考えられる．したがって，エキスパート・プログラマが書いたプログラムは一定の形式をしており，ミスが少なく，理解しやすいわけである．これに対して，初級者はこのようなプランを知識としてもっていないので，必要な都度プログラミング言語によって表現を工夫しているわけである．したがって，表現に一貫性がなく，ミスが生じやすく，理解しにくいものとなる．後の節で述べるが，PROUST においてはプランを認識することがプログラム理解のベースとなっている．

Soloway らのプランは，認知科学的視点でモデル化されたプログラミング知識であり，自然言語理解におけるスキーマやスクリプト[18]と呼ばれる概念に類似のものである．同様にプログラム理解も認知科学的な立場に立って研究されている．

4.4 プログラムのグラフ・モデル

アルゴリズムの知識をグラフで表現するアプローチも考えられる．特に，プログラムの手続きをグラフ表現する方法が理解しやすい．従来から広く用いられているフローチャートもこの範疇に入るが，これは制御フローに焦点が当てられているので，プログラム理解には応用しにくい．したがってここでは議論の対象から外すことにする．

プログラムのグラフ・モデルとしては，Adam らの初心者プログラマのためのデバッグ・システム LAURA において採用されているグラフによるプログラム・モデル[24]，上野らの知的プログラミング環境 INTELLITUTOR のプログラミング・ガイドおよび論理チェック用のプログラミング知識として採用されている手続きグラフ (procedure graph)[4]，や Soloway らの初心者プログラムの論理チェック・システム PROUST で用いられている戦略グラフ (strategy graph)[10] などがある．

プログラム・モデルは，プログラム手続きを言語独立に表現した有向グラフであり，図-3 に示されるようにノードでオペレーションをアークでシーケンスを表した，プログラムの抽象表現である．ソース・コードからグラフへの変換機構だけでなく，特定のプログラミング言語による表現上の違いや，プログラミングにおける細部の表現上の違いを吸収するための，標準グラフおよびそれへの変換機構をもっている．LAURA は，後述するようにこの知識とプログラマが入力したプログラムとのマッチング処理によって，複数の論理ミスを漏れなく検出することができる．

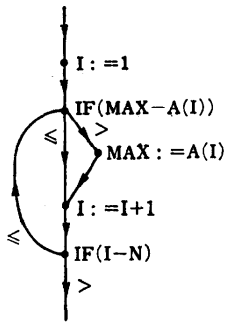


図-3 LAURA のプログラム・モデルの例

INTELLITUTOR が採用している手続きグラフは，図-4 に示されるように，プログラム手続きを階層的に表現したグラフであり，上位レベルが処理手順の概要を，中間レベルで詳細な手続きを，そして最下位レベルで特定のプログラミング言語に対する手続き表現を表しているという特徴がある．最下位を除くと言語独立である．手続きグラフは，プログラムのエディット時には，ガイデッド・エディタとしてトップダウン的にプログラム詳細化の概念に基づくプログラミング支援に利用され，論理チェックにおいてはボトムアップ的にソース・プログラム中の複数の論理ミスを検出するための知識として用いられる．論理チェックに必要な，変数の役割の同定，表現上の差異を吸収する機構などは，手続きグラフの属性情報として管理される．

戦略グラフは，特定の問題解決処理に対して複数存在する処理手続きをそれぞれについてプランの組み合わせとして表現したグラフである．さらに，各戦略グラフに対して，その下位に複数の具体的プログラム手続きが，正しいプログラムおよび誤りのプログラムとして知識化されている．誤りのプログラムは，論理ミスの原因の推定および改善の助言に有効となる．

プログラム・モデルは理論性が高く，手続きグラフはより実践的である．認知科学的観点からは，手続きグラフの方が自然であるように感じられる．なんとなれば，プログラム・モデルの方はフラットなグラフであるのに対して，手続きグラフの方は抽象-具体の階層構造をもち，大まかな手続きと詳細な手続きとが融合しており，必要に応じて任意のレベルの知識が利用できるからである．また，前節で述べたプランと手続

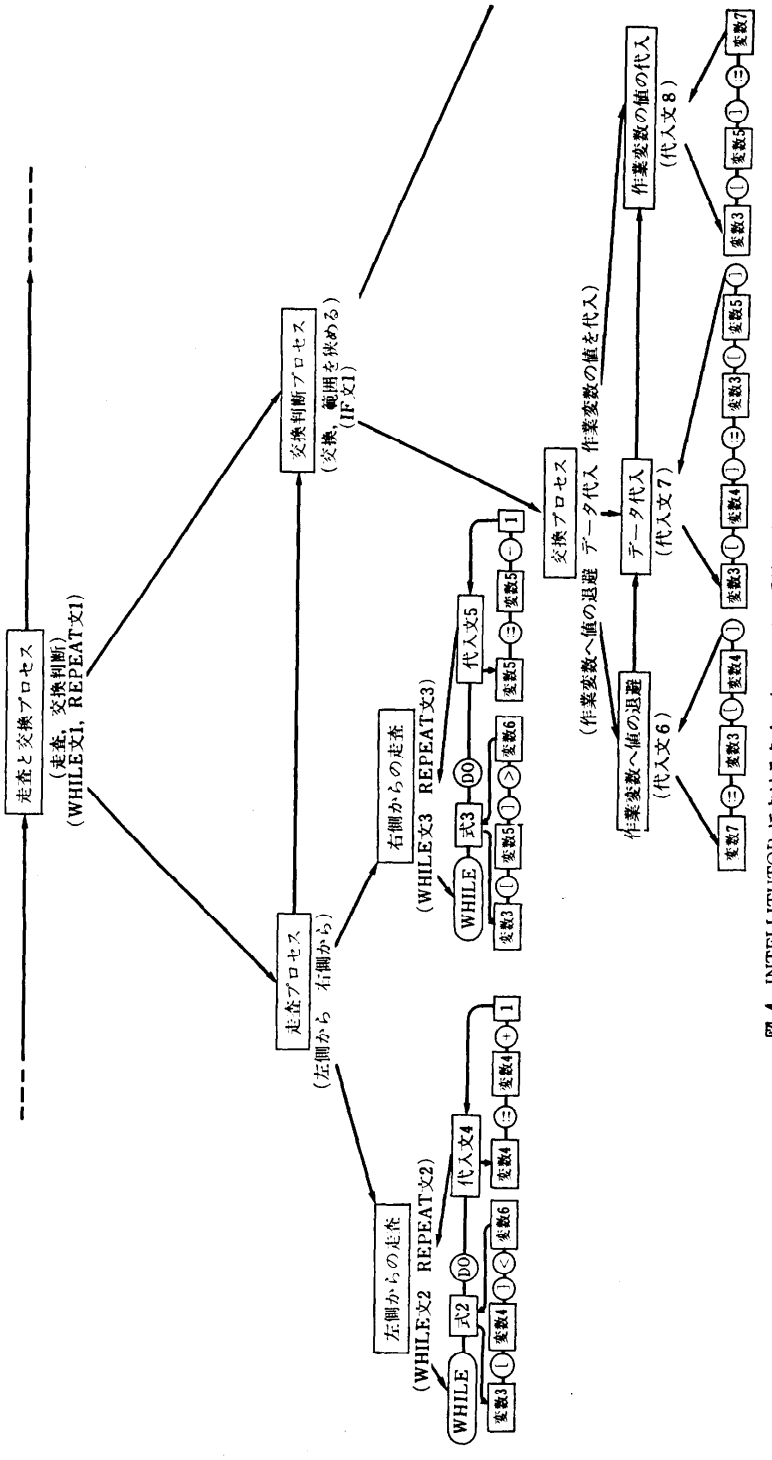


図-4　INTELLITUTOR におけるクイック・ソートの手続きグラフの一部

きグラフを比較してみると，手続きグラフは処理アルゴリズムの単位で記憶されるのに対し，プランはプログラミングの断片的知識として記憶される．したがって，両者は性格が異なる．プランはむしろプログラミングに関する普遍的コーディング・テクニックの知識であるといえる．それゆえに，対応する手続きグラフがなければそのプログラムの理解ができないが，プランの利用の場合にはアルゴリズムを知らなくてもより細かなレベルでのプログラム理解と論理チェックができる．ただし，戦略グラフは（現在の）手続きグラフよりも認知モデルという点でより自然であり，かつ論理ミスの検出能力および助言能力において強力であると思われる．

5. プログラムの意味について

ここで，プログラム理解に関する議論に先だって，プログラムにおける意味とはなにかを考察し，それと理解との関係を簡単に議論しよう．また，自然言語文における意味理解と対比してプログラムにおける意味と理解を考察しよう．なんとなれば，自然言語文もプログラム文も特定の言語で書かれた文である点は同じであり，われわれは

プログラムを読むとき自然言語文を読むようにして読んでいるとも思えるからである[19]．そこで，われわれに馴染みがありしかも研究実績の比較的豊富な自然言語文について考察し，これとプログラムにおける場合とを対比してみよう．

さて，意味と理解の問題は，AIにおける基本的課題の1つである．知的プログラミング環境においても同様に重要かつ不可欠である．すなわち，コンピュータがもしプログラマが入力したプログラムを理解することができれば，プログラマの仕事の意図を推察し，プログラムの論理的エラーを検出し，適切に助言することが可能となるからである．逆に理解能力がなければ，これらのことは実現できない．別の表現をとれば，あたかもエキスパート・プログラマがかたわらにいて個人的にプログラミングを支援してくれるようにコンピュータが支援してくれる環境の実現が，知的プログラミング環境の理想的目標であるといえよう．

まず，意味と理解および知識の関係について考えてみる[20]．一般に，意味 (meaning, semantics) を推論する処理が理解 (understanding, comprehension) である．理解処理の第1の段階が，構文則 (syntax) に基づく文の解釈すなわち形態素解析および構文解析であり，この処理によって意味要素の抽出が可能となり，この結果に問題分野の知識を適用して文章全体の意味あるいは意図 (intention) を推論することができる．コンピュータで意味理解のための推論処理をするには，意味情報がコンピュータで操作できるように特定のデータ構造で表現されている必要がある．また，意味理解においては問題分野の知識が利用されるが，関連する知識とのマッチングをとる処理が基本となる．したがって，意味表現のためのデータ構造と知識表現のためのそれとは，おのずと類似のものとなる．たとえば，スクリプトは自然言語理解における知識表現かつ意味表現モデルであり，前節で述べたプランや手続きグラフはプログラム理解のための知識であると同時に，意味表現モデルでもあるということができる．なお，自然言語文においては各種のあいまいさのために1つの文に対して複数の解釈が成立するので，意味処理を構文解釈にフィードバックすることが不可欠であるが，プログラム文における解釈は一通りしか存在しないのでこの点は楽である．その代わりに，別の困難がある．そこで，この両者を簡単に比較してみよう．

図-5 は自然言語の文章とプログラムを意味理解の観点から比較してみたものである．比較項目は，意味理解の手掛かりになると思われる，構文，語，主題，目的，形態を取り上げてみた．まず，構文は前述のように，解釈の重要な手掛かりを与える．この点については，自然言語よりもプログラムの方がはるかに単純である．自然言語文では構文があいまいで，次に述べる語の意味や全体の意味を考慮に入れないと構文の不完全さやあいまいさを正しく処理できない．この点が，自然言語処理の重大な問題点となっていることは良く知られているとおりである．一方，プログラムの方は構文が単純かつ明瞭である．1つの構文的表現については解釈は一通りであるので，この点についての問題はない．

語は文の理解にきわめて重要な役割をもっている．たとえば，自然言語文においては動詞を手掛かりとして構文および意味の処理を行っているばかりでなく，名詞や形容詞がもつ意味が意味理解にとって重要である．一方，プログラムにおいては，予約語が最も重要な役割をもつ．これを手掛かりにして文の構造はきわめて容易に解釈することができる．しかし，意味理解については様子が大きく異なる．すなわち，自然言語文においては文中の語がもつ実世界の意味の果たす役割が重要な働きをするが，プログラムの方ではそれは単なる記号としての意味しかもたないからである．したがって，プログラムの意味理解はこの点からきわめて困難であることが分かる．

主題は自然言語文における最も重要な意味といえる．すなわち，主題のない自然言語文は文とはいえない．つまり，自然言語文においては主題が明確であり，これを中心にしてあらすじが構成される．一方，プログラムの方は主題が不明瞭であることが多い．代わりに，機能あるいは役割が明瞭でなければならないが，これもあいまいなことが多い．つまり，設計者あるいはプログラマの能力や方針によって非常にまちまちであり，一定の思想や形式が存在しないのが現状である．このことがプログラムの理解を困難にしている．

自然言語文の目的は説明であるが，プログラムの目

	自然言語	プログラム
構文	複雑	単純
語	意味をもつ	単なる記号
主題	明確	不明確
目的	説明	操作
形態	多様	多様

図-5 自然言語とプログラムの比較

的は操作であるといえる．ただし，知的問題解決という観点からみると，両者はかなり類似性が高い．つまり，人工知能型プログラムにおいては，HOW ではなく WHAT を与えるだけでよいようにすることを目標に置いているからであり，知識の表現は多分に状況の説明的色彩がある．(ただし，現状では WHAT 型にはいまだ遠く，HOW 型であるが) いずれにしても，自然言語文とプログラムはこの点において性格が異なるものとなる．また，形態については，自然言語文もプログラムも多様である．つまり，同一の内容もしくは処理について，多様な表現が可能であり，これが意味理解をきわめて困難にしている．

さて，意味理解の立場からみると，あいまいさや多様さを減らすことが重要であるので，この点からプログラムの意味処理について簡単に考察してみよう．まず，構文に関しては全く問題はない．次に，語は意味理解においてきわめて重要な役割をもつので，実世界つまりそのプログラムの問題解決の対象である領域の用語を用いることが大切である．すなわち，IやJより NUMBER やNAME がはるかに望ましい．この点はすでにプログラムの読みやすさの観点から推奨されていることがらである．つまり，人間の理解モデルを用いてコンピュータ化するのが人工知能の基本的アプローチであるから，意味理解をコンピュータにやらせる場合でも，まず人間にとって理解しやすいことが重要である．

次に，主題を明確にすることは可能であろう．機能単位にモジュール化し，かつ適切なコメントを付ければよい．表現形態に関する統一は少しやっかいであると思われる．1つのやり方は，なるべく部品化された標準的プログラム・モジュールを準備し，これを使ってプログラムを書くようにすることと，同じく標準的プランによってコーディングすること，さらにはコーディングをコンピュータが支援して表現の多様度を極力少なくするようにガイドすることであろう．このやり方の最終目標は，機械によって生成されたプログラムが最も機械によって理解されやすいという考え方である．

以上に加えて，自然言語による文章の理解とプログラム理解との間には，重大な違いがある．自然言語文はあいまいではあるが，このことが逆に表現と解釈の自由度を大きくしている．つまり，少々の表現ミスや省略があってもデフォールトなどで補って意図を理解できるが，プログラムにおいては表現ミスや省略は致命的である．このことはしかし，自然言語文は冗長も多く，すべての文要素が厳密な役割をもっているわけではないことも理解を比較的容易にしている理由となっている．これに対してプログラムは，厳密な論理のもとにすべてのプログラム文および変数が一定の役割をもっており，特定のデータ処理目的に関して完全に働くシステムでなければならない．したがって，理解の目的そのものが異なるともいえる．たとえば，自然言語においてはミスの検出は重要な問題ではないが，プログラムにおいては，これは知的プログラミング環境にとってきわめて重要な問題であり，しかも次節で述べるように，非常に困難である．また，プログラム特有の問題として，データ構造に関する問題がある．解くべき問題が複雑化あるいは大規模化すればするほど，プログラムにおいてはデータ構造がもつ意味が大きくなるが，アルゴリズムがどちらかといえば静的であるのに対してデータ構造は動的であり，かつ特定の問題解決と特定のデータ構造とが対応するほどに標準化できにくい．したがって，後述するように，データ構造から特定の意味を抽出することは現時点では一般に困難である．同様に，データ構造の方に比重の大きいプログラムに対するアルゴリズムの理解も困難であるといえる．

次に，意味のレベルについて考えてみよう．自然言語による文の意味は，語の意味，文の意味，文脈の意味，および文全体が表す意味，つまり要旨，というように，複数のレベルがある．同じように，プログラムにおいても意味には幾つかのレベルがあるが，これはプログラムの抽象度のレベルに対応していると考えられる．抽象度が高いほど問題解決の目的，意図や概念に近づき，低いほど具体的操作に近づく．また，プログラミング言語における意味と，その言語で書かれたプログラムの意味とは，お互いに関係はあるが異なっている．(手続き型) プログラミング言語は具体的操作をコンピュータ向きに記述するための言語に過ぎないからである．その意味で，プログラミング言語においては，構文規則に対して解釈を適用して抽出された情報が，プログラムの意味レベルの最下位に対応すると考えてよい．さて，**4.1** でプログラミング知識を4つのレベルに階層化したが，意味もこれに対応して4つのレベルで階層化できると思われる．すなわち，まず最下位が基本データ操作レベル，その上がデータ処理技法レベルで，その上が抽象データ処理アルゴリズム・レベルであり，そして最上位が問題解決概念レベ

ルである．

これらよりさらに下位に位置づけられるプログラミング言語における意味とは，通常，その命令文に対応する行為（actions）あるいは効果をいう．たとえば，WRITE 文（命令）の意味は主記憶装置内の情報を外部媒体へ移すという行為である．また，宣言文は処理系への命令であり，特定のデータ構造を造り出すための行為を引き起こす．つまり，このレベルでは，構文と一対一に対応する効果がプログラムの意味である．なお，基本データ操作レベルから上のプログラムにおける意味は，プログラミング知識と対応しており，観点によって多少異なるかもしれないが，知識を意味と読み変えることができる．つまり，知識はプログラムの意味に対応するものであり，知識表現と意味表現はほぼ同一の構造である．

6. プログラム理解と論理チェック

与えられたプログラムから意味を推論することがプログラム理解である．前述のように意味には複数のレベルがあるから，理解にもそれに対応して複数のレベルがあり，かつ目的や方法が異なる．理解処理に用いられる知識もそれに応じて異なる．まずプログラム理解について議論し，次にこれに基づくプログラムの論理チェックについて述べる．

その前に，プログラムの意味理解の困難さについて簡単に考察する．プログラムは“データ構造＋アルゴリズム”であり，さらにアルゴリズムは“ロジック＋コントロール”であることは良く知られている概念である．つまり，プログラムの意味を理解するにはこれらのことを認識できなければならないと考えてよい．この中で，アルゴリズムはデータに対する操作であるので手続きとして明示的に書かれており，構文を手掛かりにして理解できそうである．しかし，前にも述べたようにデータ構造の方はこれよりはるかに困難である．複雑なプログラムほどデータ構造の役割が重要となる．データ構造とは対象となる実世界のモデル（a model of the world）化であると考えることができる．データ構造を中心に考えると，アルゴリズムとはデータ構造に対して操作を行い初期状態から目標状態へ変換するための手続きであると考えられる．したがって，実世界のモデル化が重要な鍵となり，特定の問題解決に対して特定の典型的モデル（データ構造）と操作列が対応するはずであるが，いまだそこまで概念や方法に形式化が進んでいない．このような理由で，現時点では，データ構造から意味を引き出すことはアルゴリズムの理解よりはるかに困難であると考えられる．これはわれわれ人間にとっても理解が困難であることが多い．現在のところは，多くのシステムがアルゴリズムに焦点をあてている．しかし，複雑な問題に関しては，データ構造の設計が，プログラム設計の少なくとも8割を占める重要な部分であることは良く知られている重要な事実である．したがって将来の問題としては，データ構造からの意味理解はプログラムの意味理解において不可欠である．そのためにも，データ構造に関するモデル論の研究がもっと活発になるべきである．

6.1 プログラム理解

物事の理解について論ずるときは，暗黙のうちに人間による理解を前提としている．この点で Shneiderman らの認知モデルは参考となる．彼らによると，プログラマのふるまいに関する認知モデルに基づくプログラム理解は，与えられたプログラムに対して2種類の知識すなわち意味論的知識と統語論的知識を使って，意味論的モデルを構築する仕事である[5]．彼らは考え方だけを提案しており，具体的なモデルや構造については論じていないが，実験を通してこの概念の妥当性を検証している．そこで，この考え方とできるだけ矛盾しないように，少し具体的にプログラム理解とその方法を考察してみよう．

最も抽象度が高い理解は，そのプログラムの目的（ゴール）を理解する仕事である．このレベルでの理解は，自然言語における概要理解に対応する．次のレベルの理解は，アルゴリズムの理解である．これは自然言語における筋書きの理解に対応する．アルゴリズムの理解においては，与えられたプログラムのアルゴリズムを抽出して知識としてもっている特定のアルゴリズムとのマッチングを取ることによって行うことができる．プログラム・モデル，手続きグラフあるいは戦略グラフとの詳細な比較ができるので，局所的な論理ミスの検出が可能である．この下位に位置する理解は，プログラミングに関する断片的知識すなわちプランなどを使ってコーディングに関する論理構造を理解する推論である．この処理結果からアルゴリズムの理解はできないが，ある種の論理的ミスの検出はできる．これより下位のレベルでの理解は，構文則に関する知識の適用であり，通常のコンパイラやインタプリタが行っていることと同じであるので省略する．

プログラムの理解において重要なことは，与えられ

たソース・プログラムの中に現れる変数，文および文の組み合わせ（プラン，ブロックなど）がそのプログラムの中でどんな役割を担っているかを同定することである．これはプログラムに関する諸拘束から可能である．つまり，使われている言語やプログラム表現上の細かな差異はあっても，キーとなる変数，変数に対する基本的操作，操作の順序は一定の枠組みの中で現れなければならないからである．特定の問題解決を行うためのプログラムにおいては，このような基本構成要素と呼べるものに対して一定の拘束条件が存在し，それが理解の手掛かりを与える．

まず，概要理解について考えてみよう．自然言語文における概要理解は，誰が，いつ，どこで，なにを，なんのために，どんな方法で行い，その結果，なにが，どう変わった，…などの項目に特定の値をあてはめる作業であると定義してよいであろう．一方プログラムについては，たとえば，“このプログラムの目的は，100 個の数値を，クイック・ソートの方法で並べ替えることである”というような文章が，与えられたソース・プログラムの分析から生成できたら，“プログラムの意味（概要）が理解できた”といってよいであろう．どうしてクイック・ソート法と分かるのか，どうして並べ替える作業と分かるのか，などがここでのポイントである．前もって知っている知識を使って，文脈から抽出された情報の断片を組み合わせて，知っている概念や手法に関する属性情報とのマッチングを取ることによって理解できる．この処理では，アルゴリズムの理解もある程度必要ではあっても，細部の理解やミスの検出は必要ではない．この点が次に述べるプログラム理解に基づく論理チェックとは本質的に異なる点である．このレベルでのプログラム理解は，自然言語文の理解におけるプランやスクリプト[18]などの利用と類似性が高い．

アルゴリズムの理解においては，与えられたソース・プログラムが，問題解決の目的を達成するための正しい手続きを与えているか否かを判定できることが，重要である．このためには，そのプログラムからアルゴリズムを抽出し，知識としてもっているアルゴリズムのテンプレートとのマッチングを取る操作が必要である．当然のことながら，テンプレートをもたない問題に対しては理解ができない．これは，“知っているから理解できる”ことの実現であり，“知らなくても理解できる”ことを実現するのではない．（知らないことの理解は，類推や学習との関係が強く，今後必要となる研究課題であろう．）またマッチング処理においては，変数や文の役割の同定が不可欠であり，柔軟性をどこまで高められるかが1つの課題である．

プランを用いれば，アルゴリズムをもたなくても与えられたプログラムの正当性をある程度理解できる．たとえば，プログラム中に “SUM＝SUM＋1” という代入文があれば，合計変数プランを仮定し，この文の前でかつループの外側の代入文 “SUM＝0” が，初期値設定のための代入文であることを理解できる．もしこの初期値設定の代入文がなければ，論理ミスの存在を疑うこととなる．

6.2 プログラム理解に基づく論理チェック

与えられたプログラムが与えられた仕様を正しく満たしているか否かを判定する問題は，プログラムの検証と呼ばれ，従来から研究されてきた重要な課題である．論理チェックはこれと関連しているが，異なった問題でもある．プログラムの検証が，操作的意味論，表示的意味論および公理的意味論などと，形式的意味論として数学的に証明しようと試みているのに対し[21]，論理チェックでは認知科学的モデルとしてプログラムを理解し，その過程でバグを検出しようとしている．また，プログラム検証では与えられたプログラムが正しいか否かを二者択一で結論づけることに焦点を当てているのに対し，論理チェックにおいては複数のバグをもれなく検出しようとしている．数学的取り扱いは最も客観的で美しいが，現実の複雑なプログラムを対象とするほどに強力ではない[24]．バグの検出能力がないことに加えて，この点も致命的である．

認知科学的アプローチにおいては，人間のエキスパートが論理チェックをほぼ完全に実行できるという事実から出発し，彼らのやり方（ふるまい）を観察し分析し，これに基づいてコンピュータ・モデル化することを試みる．研究者によって視点や趣向が異なるので，異なったモデルやシステムができ，かつ必ずしも論理的に完全ではない．しかし，人間はいつでも論理的には完全ではないが，すばらしい能力を発揮している．認知科学的アプローチは，いわば人間のこの能力を発見する手掛かりをつかむための試みであるともいえる．このようなわけで，プログラム理解に基づく論理チェックの研究は AI 研究としても重要である．

論理チェックで重要なことは，まず与えられたプログラムが正しいか否かを正確に判断できること，そしてもし正しくない場合にはすべてのバグを検出できること，さらには各バグの原因を指摘し，できれば訂正

のための助言を与えられること，などである．この4つが完全にできれば理想的であるが，現在のところではいまだそのようなシステムは実現されていない（ようである）．

プログラム中の複数のバグをもれなく検出するためには，ローカルなチェックができる必要がある．プログラムのグラフ・モデルを用いた方法がこの点で優れているといえる．LAURA のプログラム・モデル，INTELLITUTOR の手続きグラフ，PROUST の戦略グラフなどがこの類である．これらはそれぞれ異なった特徴をもっている．プログラム・モデルはアルゴリズムを抽象的に表現したグラフであり，多様なプログラム表現を1つの標準的表現に変換する高度な機構をもっているが，フラットな表現であるのでバグの意味づけや原因推定において限界があると思われる．手続きグラフはグラフそのものが抽象-具体の階層構造をもっているので，アルゴリズムのどの段階でどんなミスを侵したかを説明する能力をもつ．したがって，修正に関する助言がある程度できる．しかし，プログラム・モデルほどには客観性がない．戦略グラフは，アルゴリズム上のバグだけでなく，プログラム表現技法としてのプランに関するバグの検出ができ，さらにこれに加えて，ミスの原因を引き起こしたプログラマの勘違いを指摘し修正に関する助言もできる．これは，PROUST が正しいプログラミング知識に加えて間違ったプログラミング知識をもモデル化し，戦略グラフの中に含ませているからである．逆にいえば，プランを基本要素とする戦略グラフはこのような柔軟性をもっているといえる．したがって，これらの中では，戦略グラフが最も優れていると考えられる．

報告によると[10]，PROUST は読み取った数値データの平均値を求めるという Rainfall Problem 問題について，206 本の初心者の作成した PASCAL プログラム中，161 本（79%）のプログラムに関して完全なバグ・レポートを作成した．これらは，合計 570 のバクを含んでいた．他については，検出もれや，間違ってバグとしたものを含んでいたが，全くバグ・レポートが出せなかったプログラムは9本（4%）だけであった．なお，LAURA についても実験報告があるが[24]，INTELLITUTOR に関してはまだない．

これらはそれぞれ優れたプログラム理解能力をもっているといえるが，問題もある．たとえば，PROUST に関していえば，いろいろな問題解決アルゴリズムの完全な戦略グラフを用意することは，現実には決して容易なことではない．これは，他の2つについても同様である．しかしながら，アルゴリズムにかかわらず，自然言語においても意味理解システムを実現しようとすると，関連する問題分野に関する意味理解のための知識構造をプランやスクリプトの形式で準備しなければならず，この点が意味処理における限界の1つである．これはしかし，長期的には，少しずつこの種の知識を積み上げていくことによって，可能性がおのずと開かれる類の問題でもあり，悲観するに当たらない．人間でも，専門知識の修得には長時間を要している．人間の知識が個人専用であるのに対して，知識ベースの知識はコンピュータを使ってシェアできる点は大きな利点である．

グラフ・モデルによるプログラミング知識は，いわゆる深い知識の1種とみなすことができる．IF-THEN ルールの組み合わせでもプログラムの論理チェックは可能であるが，グラフ・モデルの方が抽象度が高く，客観的でもある．したがって，将来はプログラムからの自動知識獲得や，対話的知識獲得支援の可能性が開かれよう．このようなことが可能となると，プログラム理解システムの実用の可能性は高くなる．

7. 関連の研究

次に，幾つかの関連研究を紹介しよう．

知的エディタに関する大規模かつ長期的プロジェクトに Shapiro らの IPE (Intelligent Program Editor) がある[17]．これはプログラム・メインテナンスもサポートする計画におけるサブシステムであり，米国防省が年間10億ドルも費やしているプログラム・メインテナンスの経費の大幅な減少を意図している．ただし，いまだ計画の初期の段階にすぎない．

知的という表現はとっていないが，CMU の Habermann らの GANDALF プロジェクト[13]は，同様にプログラム開発および保守にかかわるあらゆる面の支援を意図しているという点で驚嘆のかぎりである．特に興味深いのは，GANDALF システムがいろいろなプログラミング言語に対する GANDALF 環境を生成するためのシステムであるという点である．日本的感覚では，とても大学が試みられる規模のプロジェクトではない．米国の大学の研究プロジェクトは，プロジェクト自身が研究員を雇用しているという点で，我が国の大学とは異質である．しかも大学であるから可能となるリスキーな計画を実行できる．米国における AI 研究は，このような環境で育ってきた．ただ

し，このプロジェクトが当初の目標を達成するまで続くとは，他の多くの米国の大規模プロジェクトがそうであったように，とても思えない．

Ruth の知的プログラム分析システム[16]は，プログラミング教育の支援を対象とするシステムである．このシステムは，PGM (Program Generation Model) というアルゴリズムの表現モデルを使って，ユーザのプログラムを理解し，論理ミスを検出することを試みている．プログラムは特別に定義された基本的な ACTION のリストとして表現することができ，特定の問題解決のための ACTION リストが PGM である．与えられたプログラムの分析（理解）は ACTION リストとのマッチングを取ることによって行われるが，彼はこの処理を逆生成 (reverse synthesis) と呼んでいる．このシステムも複数の論理的バグの検出能力をもっている．

福永の PROMPTER[15]は，アセンブリ言語で書かれたプログラムに対し，自動的にコメント文を生成するシステムである．オペランド部によって命令の意味が異なってくる点に着目してプログラム理解を行っている．ただし，アルゴリズムやプログラム技法に関する知識はもっていない．

山本らの Soft DA (Sofoware Design Automation)[14]は，プログラムの再利用によって新しいプログラムの作成を対話的に支援することを試みた知的プログラミング環境である．段階的詳細化の概念でユーザのプログラミングを支援するエディタと R/D ネットワークと呼ばれるプログラムの構造木とが有機的に働き合い，プログラム作成，変更，再利用などを支援する．ただし，プログラム理解による論理ミス検出の機構はもっていない．

8. おわりに

プログラミング知識とその知識を利用したプログラム理解を中心にして，知的プログラム環境の概念や方法を議論した．この種の研究の歴史は浅く，我が国ではプログラム自動生成に興味が集中し過ぎているために，この分野の研究者が極端に少なく，問題の所在さえも知らない人が大部分であると思われる．外国でも研究者や研究例が少なく，PA などごく限られた研究例が知られているに過ぎない．

知的プログラミング環境に関する研究は，プログラミング知識，プログラムの意味，プログラム理解，プログラムのデバッグ，プログラム・ガイド，プログラミング教育などの課題を含み，認知科学，人工知能，知識工学からソフトウェア工学にわたる研究分野であり，かつプログラム生成とも強い関係をもっている．著者の未熟さのために問題の重要さや面白さが適切に伝えられたかどうか心許ないが，この論文がこの分野の研究が活発化する一助となることを期待する．

参考文献

1) Newell, A.: The Knowledge Level, AI Magagine, pp. 1-20 (Summer 1981).
2) 上野晴樹：エキスパート・システム—研究動向と技術的課題—，人工知能学会誌，Vol. 1, No. 1, pp. 48-56 (1986).
3) 上野晴樹：知的プログラミング支援システム INTELLITUTOR について—背景と開発思想—，情報処理学会知識工学と人工知能 37-5 (1984).
4) 上野晴樹，中島慶人：プログラム理解による論理チェックシステム，昭和62年度人工知能学会全国大会論文集，pp. 449-452 (1987).
5) Shneiderman, B. and Mayer, R.: Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results, Inter. J. of Computer and Inf. Sciences, Vol. 8, No. 3 pp. 219-238 (1979).
6) Rich, C. and Shurobe, H. E.: Initial Report on a Lisp Programmers Apprentice, IEEE Trans. on Soft Eng, Vol. SE-4, No. 6, pp. 456-467 (1978).
7) Rich, C.: A Formal Representation for Plans in the Programmer's Apprentice, Proc. 7 th IJCAI, pp. 1044-1052 (1981).
8) Waters, R. C.: The Programmer's Apprentice: Knowledge Based Program Editing, IEEE Trans. on Soft. Eng., Vol. SE-8, No. 1, pp. 1-12 (1982).
9) Soloway, E. and Ehrich, K.: Empilical Studies of Programming Knowledge, IEEE Trans. on Soft. Eng., Vol. SE-10, No. 5, pp. 595-609 (1984).
10) Johnson, W. E. and Soloway, E: PROUST: Knowledge-Based Program Understanding, IEEE Trans. on Soft. Eng., Vol. SE-11, No. 3, pp. 11-19 (1985. 3).
11) Johonson, W. L.: Intension-Based Diagnosis of Novice Programming Errors, Morgan Kaufman Publishers (1986).
12) Shapiro, D. G. and McCune, B. P.: The Intelligent Program Editor, IEEE,. pp. 226-232 (1983).
13) Habermann, A. N. and Notkin, D. S.: The Gandalf Software Development Environment, Carnegie-Mellon University (1982).

14) Yamamoto, S. and Isoda, S.: SOFTDA: A Reuse-Oriented Software Design System, Proc. IEEE COMPAC 86, pp. 28-290 (1986).
15) Fukunaga, K.: PROMPTER: A Knowledge Based Support Tool for Code Understanding, Proc. IC on Software Engineering (1985).
16) Ruth, G. R.: Intelligent Program Analysis, Artificial Intelligence 7, pp. 431-441 (1976).
17) Shapiro, D. and McCune, B. P.: The Intelligent Program Editor—A Knowledge Based System for Supporting Program and Documentation Maintenance, Proc. IEEE, pp. 226-232 (1983).
18) Schank, R. C. and Abelson, R.: Scripts, Plans, Goals and Understanding, Lawrence Erlbaum Associats, Hillsdale, NJ (1977).
19) Soloway, E., Bonar, J. and Gold, E.: Reading a Program is Like Reading a Story (well, almost), Proc. Cognitive Science Conf. (1983).
20) 上野晴樹：プログラムに於ける意味について，科研報告「統合化ソフトウエアツールに基づくソフトウエアデータベース構成法の研究」(代表者：国井)，pp. 205-209 (1985).
21) 水野幸男：ソフトウエア生産技術，電子情報通信学会誌，Vol. 70, No. 7, pp. 693-703 (1987).
22) 原田 実：部品合成によるプログラム自動生成へのアプローチ，研究報告：583018，電力中研 (1984).
23) 古宮誠一：部品合成によるプログラム自動合成システム PAPS—知識工学的アプローチを用いたその実現方式について—，電子情報通信学会研究会資料，SS 87-2, pp. 5-12 (1987).
24) Adam, A. and Laurent, J. P.: LAURA: A System to Debug Student Programs, Artificial Intelligence, 15, pp. 72-122 (1980).
25) 大須賀節雄：知識表現に関する一考察，人工知能学会誌，Vol. 1, No. 1, pp. 20-29 (1986).
26) Polya, G.: How to Solve It, Doubleday (1957).