View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by TUIS Academic Repository

自動プログラミング私見

大林久人*

ソフトウエア（Softwares）という語は、古い英語ではシルクロードを経てヨーロッパに運ばれてきた絹織物をさしていたようである。近代ではビロード、モスリンなどの手ざわりの柔らかい毛または綿織物をさすことが1933年版のオックスフォード英語辞典(The Oxford English Dictionary）の中に出てくる。コンピュータ用語としてのソフトウエアが、権威ある同辞典の中に採用されたのは1986年発行の補遺版（Supplement）からである。

ここでの説明は、次のようになっている。

a．コンピュータに特定の仕事（task）を処理させることができるようにするために必要なプログラム[(1)]及び処理手続き

b．特定のコンピュータを動かすためにメーカから使用者に提供されるコンパイラ[(2)]（プログラム言語の翻訳プログラム）やライブラリ・ルーチン[(3)]を含むシステム・プログラム[(4)]

さて、21世紀を迎える時期に、日本では100万人近いソフトウエア技術者が不足するといわれている。

もちろん、コンピュータ・メーカでシステム・プログラムの開発をする技術者が100万人不足するということではない。それらを含めて、ソフトウエアに関係する技術者（正確には技術者と呼ぶのが正しいかどうかわからないが、他に適切な用語がないので技術者としておく）が足りないということである。

ところが、ソフトウエアという語自身がプログラムをさすのか、処理手続きを含めたより広範囲のとらえ方をするのかが不明確なままでは、どの程度の知識、技能、経験を持つ技術者が不足するのかよくわからない。

そこで、現在、ソフトウエアという語の対象について、どのように理解されているかということから考えてみることにしたい。

狭義のソフトウエア技術者

ソフトウエアがなければコンピュータはただの箱と呼ぶ人もいる。ソフトウエアとプログラムを同義と解すれば、たしかにそのとおりということができる。ほとんどのオフィスで利用されるようになったといってよいワードプロセサ（パーソナル・コンピュータをワープロ用ソフトで使用する場合も含めて）も、内臓されているマイクロ・コンピュータを動かすプログラムとそのプログラムで使用する辞書や表示用の文字パターン[(5)]などがなければ、ワードプロセッサとして機能することはできない。

それどころか電卓程度の簡単な加減乗除の演算でさえプログラムがなければ実行できないこともよく知られた事実である。

さて、そのプログラムであるが、コンピュータを使用する（あるいは購入する）ユーザの立場から見たとき、次のように分類することができる。

a．ユーザが自分で作るプログラム。主として特定の業務に関する情報処理に必要なプログラムと特定の問題を解くために必要な計算処理をする技術計算プログラムである。

b．コンピュータを運用する部門が、コンピュ

---

* 昭和64年度東京情報大学教授就任予定

ータの運用状況や障害の発生状況、磁気記録媒体の使用状況などを知るために自分で作るプログラム。

c．メーカから有償、無償を問わず提供するシステム・プログラム。

d．メーカまたはソフトウエア専門会社から有償で提供する特定の業務や特定の分野の技術計算、各種の統計手法を用いる計算処理などに利用できるプログラム・パッケージ。[6]

パーソナル・コンピュータで使えるワープロ用ソフトなどもこの範ちゅうに入る。

ソフトウエアという語を最も狭い範囲でとらえたとき、ソフトウエアはプログラムと同義ということになる。それではソフトウエア技術者とはこれらのプログラム言語で書く作業（コーディングという）や書いた結果を翻訳してから正しい結果が得られるかどうかを検証する作業（テストという）に従事する人たちと考えて良いのであろうか。答えはもちろん正しくない（ノー）である。

どの種類のプログラムを作る場合にも、コーディングにとりかかる前に、そのプログラムでどういう内容の情報をデータとして扱い、どういう条件のときはどういう処理または計算をして、結果をどういう形式で取り出すかということをあらかじめきめておく必要がある。プログラムの仕様を作るわけで、正式に文書化されるかどうかに関わりなく、この過程を省いて上記の種類のプログラムを作成することは皆無といってよい。

また、プログラムの仕様書の中には、それぞれプログラムとしての全体の構成法やテストすべき条件、テストに使うデータの指定などが含まれている必要があることも当然である。

現在、プログラマと呼ばれる人たちの一部は仕様書作りの作業から手をつけるが、大部分はコーディングとテストの作業を受け持つ人たちであるのが実態であろう。

そしてプログラムの仕様書作りまでの作業を主として受け持つ人たちを日本では一般にシステム・エンジニアと呼びならわしている。システム・エンジニアも、プログラム仕様を作るという意味で、当然、狭義のソフトウエア技術者の範囲に入ってきてよいことになる。

広義のソフトウエアと情報処理技術

それではシステム・エンジニアの受け持つべき仕事はなにか。プログラムの仕様書を作るまでには、さらにさかのぼった作業が必要になることはいうまでもない。

コンピュータで処理するユーザー分野の仕事は、大別して、各種の業務の処理と技術的な計算処理の分野に分かれる。これらはコンピュータの利用という意味で、ビジネス・アプリケーションとサイエンティフィック・アプリケーションと呼ばれてきた。

また、各種の手法を応用して行なうデータの統計処理は、いずれの分野にも必要な処理といえる。

サイエンティフィック・アプリケーションの分野では、適用例の多いものや特定の計算法を用いるものについて、メーカ及びソフトウエアの開発、販売を専門とするソフトウエア会社（ユーザ自身が開発して販売する会社も含む）がプログラム・パッケージを提供することが多い。

ユーザー側では、設計部門の技術者などコンピュータ担当部門に所属しない現場のエンジニアが自分で利用したいパッケージを選択し、必要な計算精度をチエックしたうえで利用するケースが一般的である。

ただし、利用できるパッケージがない場合は、設計業務の中で計算を必要とする要素を選び、要素ごとの計算に適用できる手法をきめ、精度や計算時間を含めた優劣の判断を行なうなど業務上のノウハウ、コンピュータ使用上のノウハウをあわせた高度の知識を持つエンジニアが、プログラム仕様を決定するまでの段階に必要となる。

ビジネス・アプリケーションでは対象とする業務についての処理方法や内容を理解したうえ

で、コンピュータによる処理システムを設計できる能力が、システム・エンジニアに求められるのがふつうである。

このとき、手作業では基本的に配慮されていた内部牽制制度や管理上のポイントなどを知っていて、システムに反映させる必要があることはいうまでもない。しかし、手作業を行なってきた担当部門の人には、コンピュータによる処理に移行したときに生じる問題を推察できないことが多い。したがって、あらかじめ問題点を検討し、必要な手段を講じるようにシステムを設計していく能力は、システム・エンジニアの側に主として求められてきた。

またビジネス・アプリケーションでは、大量のデータを処理したり、長期間保存しておくケースが多いので、磁気記録媒体を使用するファイル[7]やデータベース[8]の設計とデータコード[9]の設定のしかたの良否によって、最終的にプログラム開発本数や難度の増加、将来のシステム拡張の阻害などの問題を生じることもある。

さらにコンピュータを利用するビジネス・アプリケーションでは「ごみを入れたらごみが出てくる」(Garbage in, Garbage out) ということわざもできているように、不正確または間違ったデータが1件でも入力されたら、結果はすべて不正確または誤ったものになる。そのために、入力データの正確性を保証するコンピュータ・チェックの方法を、理論的かつ経験的に熟知しておくことも、システム・エンジニアに求められる能力ということになる。

また、ビジネス・アプリケーションでは、正確な記録の保存という観点から、コンピュータの運用管理、防災対策、システムの監査などの問題も付随して発生してくる。これらに対応できる能力も、システム・エンジニアの能力として要求されてくる。

システム・プログラムの開発の場合は、ユーザの立場でなく、メーカの立場からの開発ということになるが、システム・エンジニアに求められる能力が、プログラマのそれと比較して格段の差があることは同じといってよい。

システム・プログラムに関しては、コンピュータ機器自身の特性や制約などを熟知していないと関与できない基本的なプログラムがある。また、それらのプログラムを使って作ることができるプログラム言語の翻訳用プログラムなどの違いもある。しかし、いずれの場合も、システム・エンジニアには、ユーザが予想できないような使い方、誤った使い方をすることを経験的に予見できることが求められる。

それなりの対応が行なわれていないとシステム全体がダウンしたり、ユーザ側に誤まりを指摘することもできないからである。

リアルタイム処理システム[10]の場合の機密保護対策、防災対策などとくにこの種の予見あるいは経験的に得られているノウハウを、次のシステムに生かすことが必要といえる。

また、コンピュータの性能の飛躍的な向上とともに、各種の業務の統合化とリアルタイム処理への移行など、ビジネス・アプリケーションの分野では開発するシステムの規模が拡大してきた。それとともに、開発チームの管理、開発スケジュールのフォローアップ、標準化、文書化など、開発を担当する部門の内部管理が重要な課題となってきた。そして、システム・エンジニアには、担当部門の中堅管理職としての能力もあわせて求められているのが実情であろう。

システム・プログラムや通信系のシステムの場合も、このような開発チームの管理能力がシステム・エンジニアに求められることは同様といえる。

広義にソフトウエアを解釈する場合、システム・エンジニアに能力として求められる範囲のものは、すべて包含されると考えてよいとすれば、ソフトウエア技術者の理想像も自ら明らかになるといえよう。

情報処理技術、コンピュータ利用技術という語が手段としてコンピュータを使用することを前提とするかぎり、広義のソフトウエアに含められる範囲の能力が、それに携わる人びとにとって、裏付けとして必要になることはいうまでもない。

かつての“ソフトウエア危機”

いまから20年ほど前、1960年代の末期に、アメリカから“ソフトウエアの危機”(Software crisis)という言葉が伝えられてきたとき、日本で実際に危機感を持った人はほとんどいなかったといってよいのではなかろうか。当時のアメリカでの危機感は、コンピュータとその周辺機器（ハードウエア）の費用に比べてソフトウエアの費用（システムの開発、プログラムの作成、テスト、運用後のプログラム修正を含む）の比率が急上昇しはじめたことに気付いて生じたものであった。

ソフトウエアの費用は、テスト中に使用するコンピュータの使用料（ハードウエアの費用、動力費など）も含むが大部分は人件費である。

当時の日本では1ドル360円の為替レートであったから、ハードウエアの費用は関税も含めてアメリカよりも割高、人件費は同レベルのシステム・エンジニアやプログラマの月収を比べると数分の1という程度であったから、実感として理解しにくかったのもむりはない。しかし20年の間に為替レートは1ドル130円台にまで上昇し、逆にハードウエアの費用は1桁以上の単位の下降線をたどってきたわけであるから、同じような危機感を持たなければならない条件がそろってきたといえよう。

さて、ソフトウエアの費用の上昇ぶりは、当時のアメリカでロケット打上げ並みと称されるほどであったが、そうなった背景には次のような点があげられる。

a. コンピュータが第2世代[11]から第3世代に移りつつある時期で、性能も向上し、大量の業務を処理できるようになった。そのため対象とする業務システムの範囲や計算処理の種類も増加し、システムの開発にかかる期間や投入する人員も急カーブで増加してきた。

b. 第3世代になってコンピュータの故障率も急激に下がり、ハードウエアの寿命がのびるとともにそれを使用して処理する業務システムの寿命（ライフサイクル）ものびはじめてきた。システムの寿命がのびると、周囲の条件の変化によるシステムの修正とプログラムの追加、変更（プログラムの保守という）の機会も増加してくる。

問題を大きく提起したのは、bのプログラムの保守にかかる費用の急上昇であったといえよう。常識的には僅か数行の命令の追加と変更程度と考えていたにもかかわらず、実際に作業が終了するまでにかかった期間は、当初の開発にかかった期間と同じか、それをはるかに上回わるケースが多く、人件費を増加させる主要な原因となった。

ほんの数時間のうちに終了するケースは、よほど簡単な修正か、開発してからの経過期間が短かいうちに修正が生じ、もとのプログラムの作成者が自分で修正作業を行なうような、運の良いケースであることもはっきりしてきた。

事態を最も深刻に受けとめた組織のひとつはアメリカの国防省であったに違いない。軍用機、戦斗用車両などに搭載している制御用コンピュータは各種あるが、衝撃や温湿度などの環境条件のテストの関係から、1970年代に開発されたものは、当時に製造されたコンピュータをそのまま搭載して現在も使用しているのがふつうである。ビジネス用のコンピュータであれば、システムはそのままで、新しい小型、高性能、低価格のコンピュータに置換する（リプレース）ことが可能であるが、軍事用では使用環境がシビアなことから、それが許されない。

1960代の末期に、どの程度のコンピュータが軍用機などの装置に組込まれていたか正確にはわからないが、1980年ごろには、国防省で使われているプログラム言語の種類が2000とも3000とも数えられていたことからも、当時ですら容易ならぬ事態となりつつあったことは、簡単に想像がつく。

さて、プログラムの保守をそれほど困難にする原因を列挙してみよう。

(1) 古いコンピュータではコンパイラ（注2参照）を使えないので、ほとんどアセンブラ言

語[12]で書かれている。そのため経験を積んだベテランでないと、簡単に解読したり修正できないのがふつうである。

(2) プログラムの作成者と保守をするときの要員が同じでないことが多い。そのため、プログラムの内容を仕様書、プログラム流れ図[13]、プログラムリスト[14]などをたよりに解読する。しかし、作成者のくせなど、文書化されていない部分を理解できるようになるまで作成時の数倍の時間がかかる。

(3) もともと修正を考えてプログラムが作られていることが少なく、将来、どの部分を修正すればよいというような親切な仕様書が書かれていることはめったにない。

(4) 修正が数回行なわれるうちに記録が逸脱し仕様書、プログラム流れ図も現在使用中のプログラムの内容を正しく伝える文書ではなくなっているため、リストの解読だけがたよりとなる。

(5) 変更された条件にともなう処理、追加される処理などをプログラムの末尾の部分に書き足すようなことが多かったため、3000行くらいの命令の書かれているプログラムでは、1ページ66行のリストを30ページ程度めくり返しながら解読作業を行なうことがふつうで、解読作業の効率を下げる一因となった。

パッケージ・ソフトウエアは使えるか

メーカやソフトウエア会社は別として、一般のユーザがコンピュータを使用する場合、どうすればソフトウエア費用を低く抑えることができるかを考えてみることにしよう。

プログラムがないとコンピュータはなにも処理してくれないことはすでに述べたとおりである。新しい業務システムを開発しコンピュータに処理させようとすれば、当然のことながらプログラムが必要になる。そしてプログラムをいったん開発したら、システムのライフサイクルが終るまで保守によるコストがかかり続ける。

“プログラミングは必要悪”という考え方が生まれてきたゆえんともいえる。

ところで、ユーザにとってソフトウエア費用をいちばん低く抑える方法は、なんといってもプログラムを自分で作らないことである。

同じ業務を処理できるアプリケーション・パッケージを購入して使う方法で、アメリカでは多くのケースが見られたが、日本ではパーソナル・コンピュータが普及するまで、ビジネス分野での応用例はほとんど見当らなかったといえよう。

アプリケーション・パッケージはある企業でうまく使えたとしても、他の企業でうまく使えるかどうかはわからない。取扱えるデータの量や処理時間のかかり過ぎなど、基本的な制約条件に触れてくることもあるし、ユーザ固有の要求条件を完全にみたすための修正（カマトマイズ[15]という）を加え難いという問題もある。

ある企業でのカマトマイズ要求の中には、他の一般的な企業から見れば常識はずれといえるような条件もあるので、パッケージの製作者が最初からそのような条件を組入れていることは期待できない。

また、パッケージの質の問題としてデータの完全なチエックを行ない難い点もあげられる。

パーソナル・コンピュータ用のパッケージにはほとんどチエックなしですませているものが多いが、ごみ（Garbage）の混入ということが現実のデータ処理には大きい問題点となってくる。

そのほか、一定の構造のファイルは扱えてもデータベースを扱えないといった制約も、後日のシステムの拡張を阻害する要因になりかねないことになる。

自分で作るかわりにソフトウエア専門の会社に発注して作らせるというアイディアは論外である。保守も同様にソフトウエア会社に発注するわけであるから、費用がかかることは同じと考えてもよい。ソフトウエア技術者をユーザ自身がかかえたとして、将来の昇進ルートをどう設定したらよいかわからない場合には好ましい措置であっても、コスト低下の要因とはならな

い。最近、話題を呼んだ、データが持つ情報(機能を示すコード)だけでプログラムなしに処理のできるコンピュータについての詳細は不明であるが、アプリケーション・パッケージの持つ問題点をどこまで解決したうえで提供されるか、いまのところわからない。

いずれにしても、いまの段階では多くのユーザ(大企業から中小企業、官公庁を含めて)はプログラムを自分で作るか、パッケージを購入してカストマイズするかの途を選ぶしかないといってよかろう。

"芸術的"プログラミングの罪科

ユーザが自分でプログラムを作ることを前提としたとき、将来のソフトウエア費用の増加をできるだけ抑えられるようなプログラムの開発方法をとることが、アメリカで生じたような"ソフトウエアの危機"を未然に防ぐ途ということができる。

プログラムを作るという立場から考えると、ユーザ向けのアプリケーション・パッケージや表計算ソフト[16]をはじめとする各種のビジネス・アプリケーション用ソフトウエアなどのメーカ(ソフトウエア会社を含む)にとっても、合理的な開発方法をとることが望ましいのはいうまでもない。

さて、従来からユーザが使うプログラムの開発費、保守費をできるだけ低く抑えるためにとられてきた方法をあげてみよう。

a．新しいプログラムを作るとき、以前に作ったプログラムの中に似たものがあればそれを複写して、違う点だけを修正する。

b　以前に作ったプログラムから同じ処理をするルーチン[17]があったらその部分だけを複写して使う。

c．プログラム・ゼネレータ[18]のような、コンピュータを使ってプログラムを作る用具(ソフトウエア・ツール)を使う。

d．プログラム中に定数として持つ科目コードと名称などをファイル化して、コードの変更があってもプログラムを修正するのでなく、コードのファイルの内容だけを修正するようにプログラムを作る。

e．ごく定形的なリポートの作成など、条件をデータ(パラメータ)[19]として与えるだけですむような汎用的なプログラムを作って、それを使わせる。

f．保守作業のことを考えて、要員を業務単位に固定し、同じ人が開発と保守を受け持つようにする。

以上のようなことが、どこでも簡単に行なわれてきたわけではない。アメリカの雑誌の論評の中によく見かけるが、初期のプログラムは"芸術的"とはいえても、けっして"製品"とは呼べないしろものであったし、その後も、こうした傾向は受け継がれてきた。日本でも同じ傾向が見られ、"名人芸"ということばが、多くのコンピュータ室で聞かれたものであった。

したがって、以前に作ったプログラムの再利用といっても、本人が作ったものという条件がなければ、まず困難であったといってよい。他人の作ったプログラムは、たとえ仕様書などの文書がそろっていたとしても、文書化されていないコーディングのテクニックや個人的なくせなどを知らないと、解読に手数のかかることが多いため敬遠されるわけである。こうした場合、結果的には他人の作ったものを使わずに、自分のわかる方法で新しく作ることになる。たとえ、一部のルーチンであっても、自分によく理解できないものは不安で使えない。

開発の時期ならまだしも、保守の段階になってもこのような事態がよく生じたことは事実である。数行の修正程度とわかっていても、パズルのような"芸術的"プログラムを解読するには、開発するよりも時間がかかるという理由から、書き直し(実は改めて開発するのと同じ)を行なってしまうことも多かった。

書き直しも含めて、保守の場合は、追加、変更したい条件はわずかでも、その修正によって全体のプログラムが従来どおりの正しい処理を実行できるかどうかテストする必要が生じる。

この種のテストを退行テストと呼ぶが、“芸術的”なプログラムの中には、追加、変更された処理は正しいが、以前正しかった処理が正しくなくなる。しかも、どこで誤まりを生じたかが解読しきれないというものもあった。（筆者自身も経験したことがあるし、また、筆者が3日程度で開発し、テストを終らせることのできたプログラムを、他人が手がけたら３ヶ月たっても作れないということもよくあった。）

保守の期間がかかりすぎる主な原因はこれで、結局、期間に相当する人件費が膨張するし、プログラマ１人当りの生産性も低下する。アメリカではよく、アセンブラでは１日平均２行くらい、コンパイラ言語でも10行程度ということがいわれていた。（筆者自身は信じ難いが、詳細な文書化作業なども含めると、１人当りの平均はそのくらいになるのかもしれない。）

コーディング作業にコンピュータを利用することはＲＰＧ[20]に代表されるプログラム・ゼネレータによって試みられ、現在でもオフィス・コンピュータ用言語の多くはこの流れをくんでいるといってよい。

コンピュータを利用して自動的にコーディングが行なわれるのであるから、すべてのユーザが使えばよさそうに見えるが、実はそれほど使われてはこなかった。その理由は、入出力に使用するファイルの数の制限、ある条件の処理はできても、それを少し修正するような条件の処理はできないといった制約があるうえ、機能を変更すると同じゼネレータが使えないというような制約事項が多すぎたことである。

ファイル数の制限があると、ひとつのプログラムで処理できる工程を、いくつかに分けて実行するように、プログラムを作らなければならない。

結局、腕の良いプログラマなら、コンパイラ言語を使って自分で書いたほうが早いということになるし、大量のデータがあるときは、１回の処理をわざわざ数回に分けて、無駄な処理時間を稼ぐのも意味がない。

ゼネレータがあっても使えないのは、大型コンピュータのユーザの実感といったところであったろう。

ただし、プログラム保守の点からは、個人差を発揮する局面が少なく、他人の作ったものでも少しは理解しやすい長所もあった。

ｄにあげた、定数をファイル化してプログラム中に記述しないようにする方法は、磁気ディスク装置が出現する以前はあまり用いられなかったが、現在は広く使われる手法である。

また、ｅ．にあげた汎用のリポート作成プログラムは一部のユーザ（筆者も日本放送協会で作った）内で以前から使われていた。ただし、リポート形式の定型化、標準化など事務管理的な発想が受け入れられ易い組織でないと定着しにくい欠点がある。

なお、保守効率を重視するあまり、開発担当者をいつまでも（少なくともシステムのライフサイクルがつきるまで）コンピュータ部門に配置しておくということは、まずむりな話しである。

アメリカのように流動性の高い国なら人が動くのは当然であるし、スカウトもだまって見逃しはしない。日本でもソフトウエア会社の要員がけっこう動くことも事実である。一般の企業に所属する人はそれほど流動性は高くないが、いつまでも同じ部門にいると昇進の道が閉ざされてしまうこともあり、ローテーションをはかる企業もけっこう多い。

構造化プログラミングの歩み

それでは開発と保守の効率を向上させることを前提として、自からプログラムを開発するとすればどういう方向をめざすべきか。

それについて従来、どう考えられてきたかをふりかえってみよう。

もちろん、人は動くもの、そして、システム開発に必要な人員が自分の企業内で不足したときは、ソフトウエア会社への委託などで補なうことを前提とした話しである。

つまり、開発と保守は必ずしも同じ人があたる

のでなく、他人が保守をすることを前提とするわけである。

さて、効率向上の基本は、なんといってもプログラムの再利用とコンピュータ自身をプログラムの開発と保守に活用することにある。

他人の作ったプログラムが解読困難の理由で再利用できなかったのであれば、はじめから再利用し易いように作れば良い。

プログラム・ゼネレータが制約が多くて使いにくいとすれば、使えるものを開発すれば良いということになる。

そのためにどうすればよいのかの手がかりとして提唱されたのが、構造化プログラミングという概念であった。“構造化プログラミング”という語を1965年にオランダのダイクストラ博士[(21)]が造り出したことはよく知られている。

ただし、この時期には具体的にどうすれば良いかは示されなかったし、また、ユーザのほうも実際に使われ始めたばかりのCOBOLコンパイラなどの使い勝手がよく理解できず、悪戦苦斗を繰返していた時期だったので、ほとんどだれも顧みるものはなかったという。

1966年にベームとジャコピーニが共同執筆した論文[(22)]の中に流れ図で取扱うプログラムの論理構造は、3つの形式の組合わせですべて表現できるという記述があって注目を浴びた。

その3つの形式は図1のとおりであるが、たしかに、どのようなプログラムでも、詳細にわたって整理すれば、この3つの形式の構造を組合せるだけで十分に作りうると理解された。

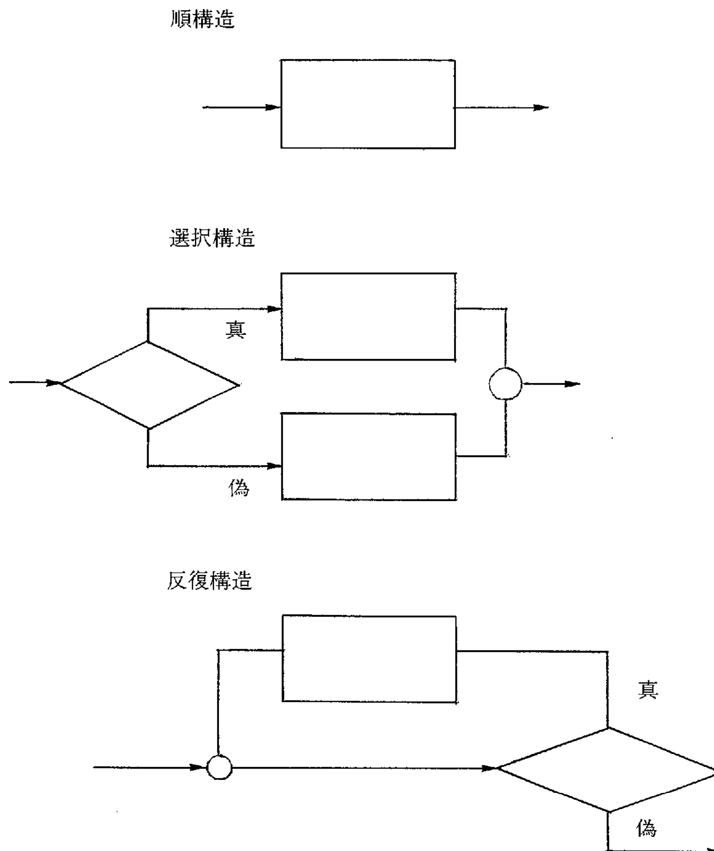

図1　3つの構造

図の反復構造はある条件が成立している間、ブラックボックスの中の処理を繰返すという表示であるが、このように命令語を使ってコーディングしていれば、入口と出口にはさまれた繰返し（ループ）の外にとび出すことはない。

また、選択構造は、ある条件が成立したら成立側の処理、不成立なら不成立側の処理をして次のブロックへ進むという表示である。この図のとおりコーディングできれば、成立、不成立の場合のそれぞれの処理が続けて書かれているので、一方がどこかへとび出すようなことにはならない。

実際のプログラムを解読しにくくする原因のひとつは、プログラムの流れの中から外へとび出す命令（Go To 命令という）の多用である。長いプログラムでは、条件が成立したらどの処理の部分へとび出せという命令が書かれているので、探してみたらリストを30ページもめくっていたということがおきる。

たしかにとび出す命令は、簡単に使えて便利なので多用されてきたが、その弊害は解読を困難にすることであった。

1968年、同じダイクストラ博士が“Go To 命令有害論”[23]という論文を書き、Go To 命令は廃止すべきであると提言した。この提言には反論が生じ、Go To 命令を使わないとかえって解読しにくい例をいくつもあげての論争がしばらくの間続いた。

1970年代に入ると論争も一段落し、Go To 命令の誤った使い方は有害であり、解読し易く、修正し易いプログラムを作るためには、誤用を避けるべきであるという考え方が大方のコンセンサスを得たといってよい。

1973年以降、構造化プログラミングをとり入れた各種の開発支援ソフトウエアやプログラム設計に関する理論、手法などが次々と発表されて今日に至っている。

なお、Go To 論争の発生した当時のコンパイラ言語のFORTRANでは、Go To 命令のない選択構造になる処理を書くことはできなかったし、反復構造を忠実に記述できる命令語はFORTRANにもCOBOLにも含まれてはいなかった。

そのため、原始プログラムを Go To 命令なしに書いて、それを現用の FORTRAN 又は COBOL の命令語に置換えるプリプロセサ（翻訳前に処理するプログラム）なども作られてきた。

しかし、ソフトウエアの生産性をあげ、開発保守にかかる費用を低減させるには、コーディングの方法を３つの形式にあわせ、Go To 命令の多用を避けるというだけでは不十分であった。また、プログラムをコンピュータに作らせる、いわゆる自動プログラミングへの道も、直ちに開けるというわけではなかった。

この時期、構造化プログラミングについて、識者がどのような定義ずけを行なっていたか、主なものを拾ってみよう。

大規模でかつ複雑に入りくんでいる流れ図を分解し、基本的な反復構造と選択構造に組み換える理論である。（ミルズ）

容易に解読できかつ修正しうるプログラムを組立てコーディングするためのマナーとでもいうべきものである。（ドナルドソン）

基本概念は正確さを証明することである。（カープ）

Go To 命令がないことが特徴ではなく、構造がそこに存在することである。（ミルズ）

プログラムに含まれる命令文や演算の対象とするデータ項目を階層的に整理し、選択処理は入口と出口を１つずつ持つネスト構造になるように組立てる技法である。（ビルト）

“構造化”の意味するところはなにか

構造化プログラミングの有用性がようやく認識された時期、実際にプログラミングに当っているシステムエンジニアやプログラマの多くは“構造化”とは Go To 命令を使わない、または多用しないプログラムを書くことが、すべてと考えていたようである。

たしかに、プログラムの先頭から順に読み下

していけることは、とび出す命令で上に戻ったり、離れた場所を探すよりも解読作業を楽にしてくれる。

初期のコンピュータ、最近では出回りはじめたばかりのマイクロ・コンピュータで、記憶容量の不足のために、単純な処理でも複雑な論理を組立てて、ゆきつ戻りつするようなプログラムを作らないと、記憶装置に入りきらない時代があった。しかし、記憶容量が次第に増加しはじめたうえ、記憶容量をこえるサイズのプログラムでも実行可能な仮想記憶[24]という技法がとり入れられはじめた1970年代の初期には、一部のリアルタイム処理のような大規模のアプリケーションを除いて、ほとんどのユーザの一般的なプログラムであれば、むりに短かくしなくてもすむ環境になっていた。

ところで、Go To 命令を使わないとそれほど、プログラムは解読しやすくなるか。

実のところ、それだけではないのである。次の例を見ていただきたい。

Ｆ11　×　Ｔ１　×　Ｒ２

という算式が命令の中に出てきたとしよう。これだけみてもなんの計算をしているのか、プログラムを書いた人以外には、理解するまでに時間がかかる。Ｆ11、Ｔ１、Ｒ２がそれぞれどういう情報を持つ項目かを調べてみないとわからないからである。

もし、1時間当り単価×超過勤務時間数×割増率のような語が命令の中に書いてあれば、なんの計算か一読して理解できる。

プログラム中のところどころにつけておく手続き名(ラベル)は、もともと悪評高いGo To 命令でとび込む入口を示すためにつけられたものである。この手続き名も、ＲＴＮ１、ＲＴＮ２、ＲＴＮ３のような暗号めいたものよりも、超過勤務手当の計算、宿日直手当の計算、減額計算のように書かれていると、その部分（ルーチン）がどの種の計算をする命令の書かれている場所か、一読してわかるというものである。

ただし、コンパイラ言語で、データ項目につける名称や手続き名の長さを30字以内のように余裕を持たせているものは、COBOL 、PL/I くらいで、FORTRAN ではデータ項目は６桁以内の英数字、手続き名は５桁以内の文番号というわけであった。現在もこの点はほとんど改善されていない。

結局、FORTAN は構造化には無縁であり、自分で作って自分で使うための言語ということを前提として作られたものと割り切ればよいのであろう。

さて、Go To 命令を減らし、名前をわかり易くしたあと、さらにプログラムの可読性、保守性を高めるには、プログラムを部分的に切り分けたモジュール構造にすることが提唱されてきた。それによって、修正作業をするとき、対象となる部分が簡単につかめるからである。

プログラムをモジュール単位に分けて、それぞれのモジュールが分担する機能、モジュール相互の関係などを一覧できる文書を作ればいっそうわかり易い。

1965年ごろの日本放送協会がリアルタイムシステムの設計のために使っていた手法がこれであり、その後、1970年代になってＩＢＭ社から発表されたHIPO[25]という手法もほとんど同じであった。

HIPO と流れ図無用論

HIPO という技法が、一時期ユーザに与えたインパクトのひとつは、流れ図不要ということであった。従来、流れ図は次のような役割りを持っており、規則で書くことを義務ずけていたところも多かった。

ａ．プログラムを作成する前に、処理の順序を考え、論理を組立てる過程で作るメモ書きの役割りを果す。

ｂ．プログラムのテスト中、論理的に処理が誤ったとき、その原因となっている箇所を探す手がかりとして使う。

ｃ．完成したプログラムの内容を説明する文書のひとつとしての役割りを果たす。次回の修正のときの手がかりとなる。

実際にプログラムを作成する人が使うのはaとbで、コンピュータ部門の管理者がほしいのはcの流れ図というところに、もともと無理があることは否定できない。

現実のコンピュータ部門では、最初にプログラムを開発した時点の流れ図を提出させていたとしても、その後に行なわれる数回の修正作業のとき、流れ図のほうの修正が欠けてしまうことが多かった。結果的にたよれるのはプログラムのリストだけとなるわけである。

また、完成後にプログラムの作成者に書かせた流れ図は、記号の中に実際に使われている命令文をそのまま書いたようなものになることが多く、肝心のどういう種類の処理をする部分かの説明は全くない文書なので、修正作業の手がかりといっても、解読を助ける資料としてはあまり役に立たないのがふつうであった。

したがって、流れ図の管理をもてあましていたところも多く、流れ図不要論が大きいインパクトとなったわけである。

HIPOで書く図式目次中の箱の関係は、上部の箱（モジュール）が下部に並ぶ箱（モジュール）を順次実行するように制御するだけの機能を持つものとする。つまり、上部の箱の中に書かれる命令は、下部に並ぶ箱を順番に呼び出す命令だけという約束にしておく。こうすると、最終的にデータを扱って演算処理をするのは下部に従属する箱を持たない最下位のモジュールである。なにをどう演算し、結果をどこに格納しているかの詳細は、そのモジュールの機能の説明書に記述する。

結局、図式目次は、全体としてどのモジュールがどのモジュールをどういう順序で実行させていくかという流れを説明し、実際の処理を受け持つモジュールの内容は、機能の説明書に詳細に書かれる。それで従来のような流れ図は書かなくても、全体と細部の処理内容がわかるというのが、流れ図不要論の根拠であった。

実は、図式目次のような階層化したモジュール構成は、リアルタイム処理のためのプログラム

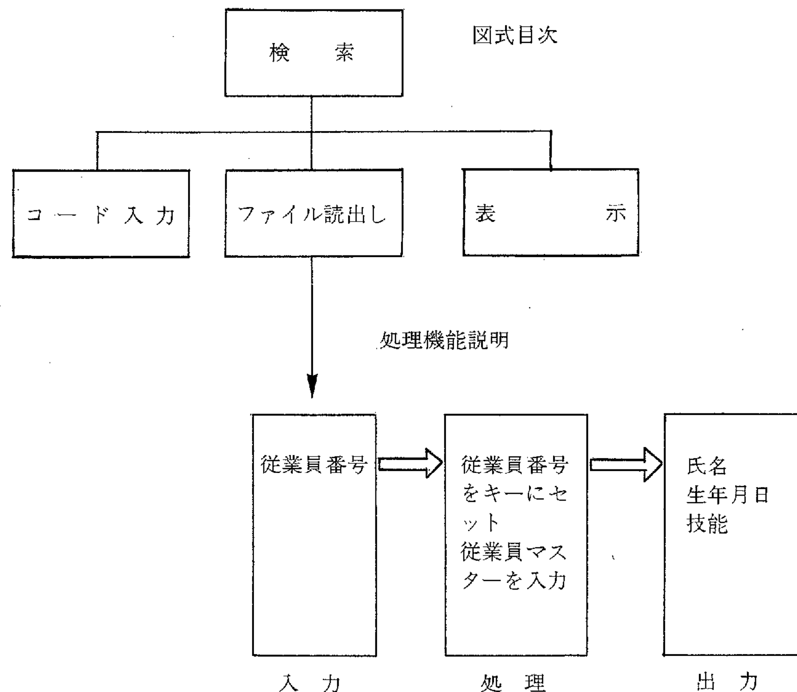

図2　HIPOの図式

の構造と各部の機能の説明には、非常に適したものである。リアルタイム処理をするプログラムは、1件のデータを端末から受信する部分、受信したデータの種類（銀行なら、預入、払戻し、記帳、残高照会など多数の種類に分かれる）に対応して、必要な処理機能を持つモジュールを選んで実行させる部分、処理の終ったデータについて結果をもとの端末に返送する部分とに大きく分けられる。もちろん、短かい時間のうちに多数の端末から同時にデータが送られ、多数のデータが同時併行的に処理されて、結果も各端末にほとんど同時に送り返すようになってはいるが、1件のデータの処理を論理的に考えれば、上のように3つの部分に分けられるわけである。

リアルタイム処理に対して、ファイルされているデータを1件ずつ読み出して処理をし、結果を書き出したあとで、再び次のデータの読み出しに戻るような処理をすることをバッチ処理[26]という。ファイルに蓄積されているデータの数だけ反復処理をするので、実行効率は良いかわり、データが発生するたびに1件ずつ処理をすることはできない。

さて、バッチ処理の場合は、データの読み出しと処理、結果の出力をなん回も繰り返すが、それらをグループ単位にまとめて、グループごとの集計結果をつけ加えることもある。また、1ページの用紙に数人分を並べてリストする場合のように、なん回かの読み出しと処理を反復したあと、まとめて結果を書き出すようなこともある。これらの反復処理の制御は、データの種類ごとに必要な処理とその実行順序というものとは、基本的に性格の異なるものである。

また、従来からその制御のしかたはプログラム作成者のくせや好みに左右されており、似かよった内容の処理が、全く違うパターンの制御のしかたで実行されている例も多かった。

つまり、HIPO図であらわすリアルタイム処理のプログラムは、1件のデータを種類ごとに処理する場合であり、バッチ処理のプログラムはその上にかぶせて、反復処理を制御するだけの論理の流れを示す部分が必要になる。それに着目せず、HIPO図を書けば流れ図不要ということに短絡して、バッチ処理のプロクラムにHIPOの図法とり入れようとしたユーザもあった。その中には、大きな混乱を招いて失敗したケースも多かったようである。

バッチ処理プログラムの制御構造

流れ図を書く、書かないということと、そこに流れが存在するということは、別の次元の話しでなければならない。

バッチ処理のプログラムには全体の反復処理を制御するという流れがあり、その中に個別のデータを処理するために必要な機能とその実行順という別の流れが存在すると考えてよい。

この異質の流れを、同一次元でとらえようとすると、なにを基準にモジュールの機能を分割するかという点で混乱を生じてしまう。個別のデータ処理をする部分だけであれば、計算をさせる機能と、その結果を印字したり、表示したりする機能とは分離するという基準を簡単に作ることができる。

例えば、リアルタイム処理で預金の利息を計算するとしよう。利息計算は当然のことながら、個別のデータを種類別に処理する部分で実行される。普通預金や定期預金の解約にともなう処理の一部である。

ところが、処理した結果の印字は、通帳、伝票、ジャーナル・シート[27]など、印字する相手の用紙に応じて別々に処理できるよう、機能を分けてモジュールを用意しておく必要がある。この機能は結果を返送する部分に含まれる。実際は、端末側の装置に、これらの印字編集機能を分担させることも多いが、そこまでを含めて返送部分と考えても別に問題はない。通信回線を通じて送る前に印字用の編集を行なうか、送ってから行なうかの順序が違ってくるだけである。

厳密に機能を分割するとすれば、リアルタイムにデータを処理するときのモジュール構成を念頭に置いて基準を作ればよいといえる。

この種の機能に対し、バッチ処理のプログラムに必要な全体の反復処理を制御する機能は、まったく別の次元でとらえる必要がある。

具体的には全体としての流れを反復前の処理、反復させる処理、反復後に行なわせる処理というような機能に分割し、その順に実行させるという基準を設けることになろう。

つまり、開始前の処理、反復処理（主処理）、終了時の処理というような区分を最上位に設けるとともに、開始前に行なうべき処理、終了時の処理の中に含めるべき処理などを基準としてきめておくわけである。

常識的に考えて、総合計はファイルに蓄積されていたデータを最後まで処理した後でないと出力できない。ということであれば、総合計の出力という機能は、終了時の処理の中に置くべきであって、つごうで反復処理の終りの部分に含めておく機能ではないように割り切ることができる。

これに対してグループごとにとる合計は、ファイルされているデータをグループ単位にまとめる処理であるから、当然、反復処理の中に従属する処理と考える。

全体の反復処理の制御パターンと同じようにグループごとの前処理、同一グループのときの処理、グループが変ったときの後処理に分け、全体の反復処理（主処理）の下部にそれが展開されるように基準を設けておく。

前処理、後処理などに含める内容も、同様に基準としてきめておくことができる。グループ合計の出力は、当然のことながら後処理に含めるべきものであり、そこで出力したグループ合計をゼロに戻す（カウンタの帰零）処理も同じ後処理に含めるのがよいといえる。

前処理には、これから処理をするグループの見出しを印字用にセットしておく処理など、いくつかの処理をきめておくことができる。

ここまでに述べてきた反復処理を制御する機能構造を、図3に示してみよう。この図法はHIPO図法と別に変わるところはないが、筆者が

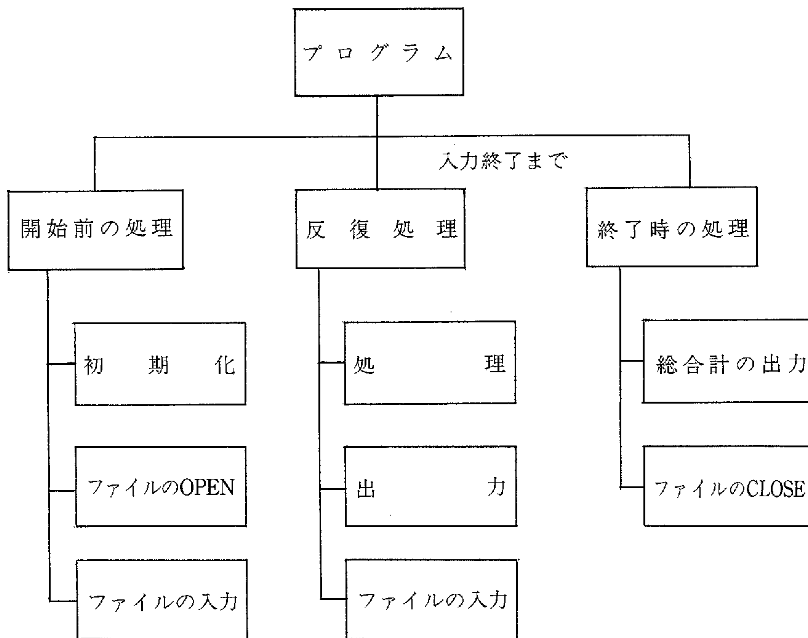

図3 階層化フローチャート

1976年に階層化フローチャート[28]（ＨＯＦ）法と名付けて発表したものである。

実際は図の表現法が問題ではなく、制御の流れを、基準を設けて類型化することをテーマとして発表したものであることをおことわりしておこう。

固定した制御構造は実用性に乏しい

現在、構造化プログラミングをベースとするプログラミングレベルでの開発技法は、演算処理の対象とするデータの持つ項目ごとの処理を中心とするものが多い。フランスのワルニエを祖として、イギリスのジャクソンをはじめとする研究者たちが行なってきた研究の成果は、それなりの意義を持っている。いずれも、最終的には、基本的な反復構造と選択構造の組合わせになったプログラムを組立てるもので、結果的には似たものができ上ることになる。

ベルティーニとタリノーの著作があるプログラムの木[29]（ツリー）も、基本的な反復構造と選択構造を組合わせてプログラムを作成するときの考え方や図の書き方などを説明したものであるが、いまひとつ、具体的なケースにどう対応すればよいかが、実務に携わっているプログラマには理解できなかったようである。

データそのものの処理でなく、ファイルに蓄積されているデータをバッチ処理することを前提として、その反復処理を制御するパターンを類型化するアプローチは以前から存在した。

古くは、コンピュータにプログラムを作らせるリポート・プログラム・ゼネレータである。

この種のゼネレータは簡易な表現から命令語を作り出す自動プログラミングのはしりともいえるわけで、今日的な意義も大きいのにかかわらず、多くのユーザで忘れられた存在のようになっているのは残念である。

ただし、リポート・プログラム・ゼネレータは、生成するプログラムの入力、出力に使えるファイルの数とファイルの編成法[30]が限られていることや、反復処理を制御するパターンが一定なので、そのパターンからはずれた処理をすることができない点が泣きどころといえた。しかも利用者は、簡易な表現でデータ項目と演算方法を書くことはできても、パターンからはずれた処理をするための命令を書きこむことができないので、結果としてユーザーの要求をみたせないものが多かった。

同種のゼネレータを開発していたメーカが、ユーザの注文を受入れて専用のゼネレータを作っていくうちに、同じような処理、例えば集計表を印字するプログラムだけでも数10種類でき上ってしまい、どれを使ったらよいか選択に困るようになったという笑い話も残っている。

COBOL 言語に含まれる報告書機能というものも、ゼネレータと似た一定の制御パターンのもとでプログラムを作り出すが、この場合は小計を出力する前とか、総合計を出力する前のように場所を指定して生成されない命令や特殊な処理を書きだせるようになっている。

しかし、単純な合計は自動的にとれても、平均値を算出するような場合、そのつど命令を書き足すことになるので、プログラム作成者にとっては、わずらわしさが感じられるばかりで、この機能を実際に使っているユーザはあまりないようである。

パーソナルコンピュータ[31]用に開発されている LEVEL II COBOL などのコンパイラでは、開発の手数ばかりかかるわりに利用度の低い機能ということで、賢明にも報告書機能をカットしているが、使用上、支障を感じる人はほとんどいない。

とはいうものの、せっかくコンピュータがあり、その性能が向上しているのであるから、それを利用した自動プログラミングを実行しないのも考えものである。

しかも現実のユーザには、ファイルに蓄積されているデータが事実として存在している、それらをデータフローの概念にしたがって、項目ごとにどのような処理を必要とするかといった解析からやり直すよりも、手っとり早く集計して結果を出したい。忙しく働くユーザのソフト

ウエア技術者の本音といえよう。

自動プログラミングの前提

それでは、リポート・プログラム・ゼネレータなどの持つ欠陥を排除するにはどういう点に着目すべきか。その1は入力、出力に使用するファイルの数やその編成法を実務的に必要とする範囲まで拡張することである。

磁気ディスク装置に数億字の記録ができるようになってからは、プログラムの中に単価表を書き込んでおいたり、命令の中でいちいち単価を定数[32]で書くようなことをするかわりに、単価表のファイルを磁気ディスク装置[33]に記憶させておいて、いつでも必要なときに読み出せるようにする方法がとられるようになってきた。

単価が変更されてもプログラムを修正することなく、単価表のファイルにあるデータだけを変更すればよいからである。そうした理由から、ひとつのバッチ処理をするプログラムに必要な入力用のファイルの数は、磁気ディスク装置の開発以前には信じられなかったような数に増加している。

ただし、この種のファイルは記憶しうる容量に無理がない限り、全体の処理を制御する流れの中では、開始前の処理の部分ですべてのデータを入力し、表（単価表）の中に値を移すようにすればよいから、反復処理の制御のしかたを変更する要因とはならない。

そして入力するデータのどの項目を、表のどの項目に移せばよいかが明確になっていることが多いので、それらの処理を自動的にプログラムとして生成するのも、さほど難かしいことではない。

ファイルの編成法さえわかれば、それに対応する入力の命令も作り出せるからである。

全体の処理を制御する流れに影響を与えるのは、反復処理の中で順次入力してくる必要のあるファイルの数である。出力するファイルの数やその編成のしかたは全く影響することはないと考えてよい。プログラム・ゼネレータの欠点であった出力ファイルの数や媒体、編成法などに関する制約は、本来とり払われていてよかったものといえる。

実例を考えてみよう。個人ごとの国語、英語、数学、理科の得点がデータとして入力されてきたとき、その合計点を計算したあと、リストに印字して出力しようが、ファイルに合計点を持つデータとして出力しようが、反復する処理を制御する流れに影響を与えるわけではない。用紙への出力か、ファイルへの出力かの違いがあるだけのことである。

たしかに、用紙への出力の場合、1ページ当りの印字行数が用紙の寸法からきまってくるので、その範囲内に収まるようにページがえを制御することを必要とする。未熟なプログラマは、その制御の方法でさえよくわからずに誤まりをくり返すが、定型化した手法を用いればけっして難かしい処理ではない。

それよりも、演算処理をする機能、結果を出力する機能と把握すれば、ファイルへの出力も用紙への出力も、機能的には同等のものとして扱えることに着目していただきたい。

結局、反復処理の制御のしかたに影響を与えるのは、順次入力して処理する必要のあるファイルの数が1つか、2つか、3つ以上かということが、まず第1レベルの要因となり、次にグループ単位に分けて扱うか、個々に扱うか、前後関係をチェックするかが2次要因となる。

入力するファイルが3つあっても、2つを直接読出し[34]で処理するなら、順次入力するファイルは1つと数えてよい。直接読出しなら、入力したデータ中にある項目の値を使って、相手のファイルからほしい記録だけを読出してくることができる。したがって、個々のデータの演算処理の一部に含まれる処理として扱えばよいことになる。

これに対し、順次入力するファイルが2つのときは、相互のファイルを参照する項目の値を比べあうのが、基本的な制御のしかたに変わる。大小を比べあうので、その結果は一方が大きいか、小さいか、等しいという3つの比較結果に

分かれる。

学生番号と科目コード、得点を持つデータと学生番号、氏名を持つ学生名簿のファイルとを参照して、学生番号、氏名、科目コード、得点を持つデータを作るような処理も、このようにして行なう。等しいときは一致する記録が名簿の中にあったということであり、データの方の学生番号が大きいときは、名簿にある人の得点がデータとして入力されていないと解釈する。また学生番号が小さいときは、一致する番号を持つ人が名簿にないということになる。これを不一致と呼ぶ。不一致の生じる原因は、学生番号の誤りが多いが、名簿へ未登録のままになっていたケースでも生じる。

３つ以上のファイルを順次入力して参照しあうときも、基本的には２つずつの相互参照を組み合わせていくような制御の流れになる。未熟な人にはかなり難かしいが、類型を知っていればそれほどではない。ただし、４つ、５つまで増えると複雑な条件になるので、実務的ではなくなってくる。

グループごとに分けて扱う処理は、順次入力するファイルが１つの場合にも、２つの場合にも組合わせることができる。

順次入力が１つの場合は、前に記述したグループ単位の前処理、同一グループの処理、後処理のセット、全体の反復処理の下部に展開することになるし、２つのファイルを参照するときは、指定されるファイルからのデータの読み出し処理が、グループ単位の処理のセットに置換えられると考えてよかろう。

こうした基準によって展開される制御の流れは、同じ基準にもとづく限り、同一のパターンをとらせることができる。

バッチ処理のプログラムを自動的に生成しようとするときは、条件に応じて、まず全体の処理の流れを一定の基準に従って作り出す必要がある。

個別のデータについて、その種類ごとに必要とする処理は、それぞれ明確に定義することができたとしても、それらをどの部分に置くかが全体の制御の流れしだいで変わってしまう。しかも制御の方法によっては、一連の処理をいくつかに分けてはめこむようなことも生じる。個別の処理の定義から、プログラムを自動的に生成できたとしても、うまく分割することは非常に難かしい。

そのために、類型的な制御の流れの生成の途を講じることが、自動プログラミングの完成への前提条件と考えるのである。

あとがき

構造化プログラミングが提唱されて以来、今日で20年を経過した。

その間にコンピュータを利用する業務処理システムは、コンピュータの低価格化と性能の向上によって、ユーザも飛躍的に増加し、大規模ユーザでは、業務処理の統合化とリアルタイム化が進んでいる。そして、システムの規模の拡大は、管理資料を得るためのバッチ処理を同時に増加させる。

コンピュータがプログラムなしには動かない限り、プログラミングの作業量はますます増え、システムのライフサイクルの延長は保守の機会を増加する。

今日までに、コンピュータ自身をシステム設計、分析の段階から、文書（記録）を作成し、保守し、その中の要求仕様から必要なプログラムの自動的な生成を試みるいくつかのプロジェクトが生まれ、いくつものツールが開発されてきた。

しかし、まだまだ小回わりのきくバッチ処理用の自動プログラミング・ツールなどの需要は広い範囲のユーザに潜在していると考えてよいわけであり、すぐれたツールの開発を試みたいと考えている。

注記

(1) コンピュータに対して動作を指示する命令を、目的とする処理の内容にあわせて順序よく並べて作ったものをプログラムと呼ぶ。コンピュータを実際に動かすプログラムは、0と1の組み合わせで表現されている目的プログラムであるが、人間が書くときは、記号と10進数の組み合わせか、英単語、10進数などの組み合わせで書く。これを原始プログラムと呼ぶ。

(2) 英単語、数式などを用いて書いた原始プログラムを目的プログラムに変換（翻訳）するプログラム。COBOL 、FORTRAN などのプログラム言語の種類ごとに専用のコンパイラが作られている。

(3) コンピュータの記憶装置と入力装置との間でデータをやりとりするための部分的なプログラムや計算式の中に出てくる開平（平方根を得る）の処理をするプログラムなど、どのプログラムでも使えるものをあらかじめ作って登録してある。それらを一括してライブラリ・ルーチンと呼ぶ。

(4) コンパイラやライブラリ・ルーチンはユーザのプログラムの翻訳に必要なものである。コンピュータを実際に使うときは、ユーザからのプログラムの翻訳と実行の要求を受付け、実行し、終了したら別のユーザの要求の実行に切り換えたり、実行状況を記録しておくような処理をするプログラムが必要になる。その他、一般によく実行される処理をするための標準的なプログラムなども用意しておく。これらを総称してシステム・プログラムと呼ぶ。

(5) 漢字を表示（印字）するとき、縦横を24個ずつの点に分解し、どの点からどの点までをつないで光らせる（またはインクを塗布する）かを1文字ずつについてきめておき、それを0と1で表わすデータにしたものを文字パターンという。1文字について24×24＝576の0と1が並んだデータとなる。縦横の点の数をさらに大きくとれば、よりきれいな漢字が表示できるようになる。

(6) 複数のプログラムを順次、または選択して実行することにより目的とする業務の処理や構築物の強度の計算などが実行できるところからパッケージと呼ぶ。
ビジネス分野のもの、技術計算（サイエンティフィック）分野のものなどがある。
アプリケーション・パッケージとも呼ばれている。

(7) 台帳や名簿などの基本的な情報、発生した取引の内容などをそれぞれ、ひとつの単位で扱う記録（レコード）として、同種の記録をまとめたものをファイルという。例えば会計取引の内容をまとめた会計データのファイルは、伝票の内容を連続記録する仕訳帳（ジャーナル）と同様の内容になる。

(8) 記録（レコード）をひとつの単位として各種の記録を相互に関連づけながら蓄積して、いろいろな処理に共通に仕様できるようにまとめたもの。ひとつの大きいファイルと考えてもよい。

(9) 従業員番号、社会保険の被保険者番号など個人を識別するためにつける番号、口座番号のように預金口座を識別する番号、都道府県コードや商品分類コードのように行政区画や商品などを分類するコードの総称。ファイルから 特定の記録（レコード）を参照するときに使うコードをレコード・キーと呼ぶこともある。

(10) ＣＤカードを使って預金を引出すように、データを入力したら即時（リアルタイム）に処理をして、結果を出力する（預金の払出しと伝票の出力または通帳への印字）のようなシステムをさす。座席予約をはじめ各種の直接顧客の要求に対応するシステムがあるが、企業内での処理をするシステムも多数開発され運用されている。

(11) コンピュータの記憶や換算をするための材料（素子）になにが使われたかにより次のような世代の区分がある。
第１世代（1946～ ）真空管
第２世代（1956～ ）磁気コア、トランジスタ、ダイオード
第３世代（1964～ ）ワイヤメモリなど
第3.5 世代（1970～ ）半導体メモリ、集積回路（IC 、LSI 、VLSI ）

(12) 記号と記憶場所を示す数字（番地）またはそれにつけた短かい名前（ラベル）の組合わせでプログラムを書く言語。コンパイラを使うにはそれなりに大きい記憶容量が必要なので、初期の小さいコンピュータでは、ほとんどアセンブラ言語しか使えなかった。

(13) 命令を与えていく順序を考えるため、記号の中に処理内容を記入して図式で表わす文書。上から下、左から右のように流れを追っていけるので流れ図と呼ぶ。

(14) 原始プログラムの内容（翻訳された結果を並べることもある。）を１行ずつ印字したリスト。プログラムの内容が一覧できる。

(15) ユーザ固有の条件にあうように汎用のプログラムの内容を修正すること。勘定科目名を変更したり、組織名を実際のユーザのものに設定したりする。これらの点は簡単に修正または初期設定できるように作られているものもある。

(16) パーソナル・コンピュータでよく使われているソフトウエア。１ページ数千字ときめておいて、そのページを縦横に区切り、横方向に、見出しをつけ、縦方向に実際のデータを入力していって計算させるところから表計算ソフトと呼ばれる。

(17) プログラムとほぼ同じ意味に使う。ふつう部分的な処理だけをするプログラム、つまり完成したプログラムの一部分を構成するプログラムである。

(18) 命令語でなく、加算する項目名だけを並べて書いたような処理仕様書からスログラムを生成するプログラム。代表的なものに、RPG（リポート・プログラム・ゼネレータ）がある。入力に使うファイル、出力するファイルまたはリストの処理方法がきめられていて、その条件内で処理をするプログラムしか生成できないのが難点となっていた。

(19) 見出しに印字する内容、明細行に印字する項目、加算する項目などをデータとして与える。それらの内容をパラメータという。

(20) リポート・プログラム・ゼネレータ。注(18)を参照

(21) エイントホーヘン大学教授。プログラムの正確さを証明することを主眼に構造化という語を使ったという。
(22) Flow Diagrams, Turing Machines and Languages with only Two Formation Rules (1966, 5) ACMコミュニケーションズ誌
(23) GO TO Statement Considered Harmful (1968, 3) 同誌
(24) 割当てられている記憶容量に入りきらないプログラムがあるとき、はみ出す部分を磁気ディスク装置に残しておいて、プログラムの実行中、はみ出した部分が必要になったときは、記憶している内容を自動的に入れ換える技法。
これによって、プログラムを書く人が自分で容量の制約からプログラムを区分し、自分で入れ換えをするオーバレイ (over lay) という手法を用いなくてよいことになった。
(25) Hierarchy Input Process Out put の頭文字をとった技法。全体の構成を階層的に示す図式目次と各構成要素の機能を入力、処理、出力の形式で記述する機能説明書から成る。(図2参照)
(26) 生産方法のひとつ。多数の部品をまとめて同一の加工を施し、次の段階に進めていく。BATCH という語はもともと1組、1束のような意味を持っていた。
(27) ロール状になったシート。金銭登録機などに使用されているものに同じ。
(28) 日本経営協会主催の公開セミナーで講演。
その後、「構造化プログラム、フローチャート」を1980年9月に刊行(日本経営出版会)各種のパターンを公開した。
(29) "Le COBOL Structur'e-unmode'le de programmation" (1974) 「構造的コボル教則本」の題名で訳書が出版されている。(TBS出版会, 1977)
(30) ファイルの作り方のこと。先頭から順に記録し、読み出すときも先頭から順に読み出すしか処理の方法がない順編成とデータの記録されている場所がわかるようになっていて、直接ほしいデータの読み出しができる索引編成、直接編成と呼ばれる編成法がある。一般的に1回処理したらよい取引情報 (トランザクションと呼ぶ) のデータファイルは順編成であり、名簿や台帳、原簿の磁気記録化されたファイル (マスタファイルと呼ぶ) は、3つの編成法のいずれかをとる。
(31) MICRO Focus 社の開発したコンパイラ。MS/DOS というシステムの使えるパーソナル・コンピュータではほとんど例外なしに使用可能である。汎用の大型コンピュータで使っている COBOL のプログラムに僅かの修正を加えるだけで処理できるほど優れた性能を持っている。
(32) 命令文の中に乗率1.5のような数値を書く方法。1.5がなんの乗率がわかりにくいため、誤まりの原因、保守作業の効率の低下の原因になり易い。
(33) パーソナル・コンピュータで使うフロッピーディスクも含めて、磁気ディスク系の装置は、ひとつの装置に多数のファイルを記録し、ファイル単位にいつでも使用できるようになっている。
(34) ほしい記録がどこになあるかがわかるファイルの編成法 (直接編成または索引編成) では、データの持つ参照キーの値 (ほしい記録がどれかを示す値) を使って、それだけを読出すことができる。