法政大学学術機関リポジトリ
HOSEI UNIVERSITY REPOSITORY

PDF issue: 2025-02-06

構造化プログラミング言語「PASCAL」のインプリメンテーションと応用

FUKUMA, T. / 駒木, 悠二 / 福馬, 敏子 / 中里, 勝 / 蜂谷, 博 / KOMAKI, Y. / NAKAZATO, M. / HACHIYA, H.

(出版者 / Publisher)
法政大学工学部
(雑誌名 / Journal or Publication Title)
Bulletin of the Technical College of Hosei University / 法政大学工学部研究集報
(巻 / Volume)
15
(開始ページ / Start Page)
73
(終了ページ / End Page)
89
(発行年 / Year)
1979-03
(URL)
https://doi.org/10.15002/00004153

構造化プログラミング言語「PASCAL」のインプリメンテーションと応用

駒木悠二*・福馬敏子*
中里　勝*・蜂谷　博**

Implementation of Structured Programming Language PASCAL and its Application

Y. KOMAKI, T. FUKUMA, M. NAKAZATO and H. HACHIYA

preface

Carring on the development of software system, exalts the quality, the raliavility and the productivity of program, and after it facilitates conversative works.

For producing program the thechnique and the concepts of "structured programing" have been recognized as the important Problem for the production management on softwareside.

On the developing of the large scale Software, the concepts of "Structured Programming" is a effective tools particularly.

We have implemented the "PASCAL" by N. Wirth as "Structured Pro graming Language" to FACOM 230-45s of Hosei UNIV. Computer Center.

In this paper, how to consider the S. P is described and gives an outline of Programming language "PASCAL".

§0　はじめに

ソフトウェアの開発をより系統的に行うことは，プログラムの品質と信頼性およぴ生産性を高め，後日の保守作業を容易にする。

プログラムの作成における構造化プラグラミングの手法と考え方は，ソフトウェアの生産管理において重要な問題と認識されるようになった。特に大規模なソフトウェアの開発においては，構造化プログラミングの考え方は，有効な方法である。

我々は構造化プログラミング言語として，N. Wirth が開発した「PASCAL」を本計算センタの FACOM 230-45S にインプリメントした。

---

* 経営工学科
** 法政大学計算センター

ここでは，構造化プログラミングの考え方と，プログラミング言語「PASCAL」の概要を述べる。

§1. 構造化プログラミングの背景

コンピュータ・プログラミングは，コンピュータの発達とそのコスト・パフォーマンスの向上にともなって，初期の機械語から，アセンブラ言語へ，さらにコンパイラ言語へと発展した。第三世代の計算機ではコンパイラ言語によるプログラミングがごく当然のものとユーザに理解されている。

事実今日ではFORTRANやCOBOL，PL/1などのコンパイラ言語を使えば，初心者であってもわずかなトレーニングを受けるだけで目的とする計算や処理を行うプログラムを作ることが，困難ではなくなった。特に理工学系の研究者，エンジニアにとっては，計算機はもはや特殊な分野ではなく，ごくあたりまえの道具として電卓などのように使いこなせる時代となった。本学計算センタでも利用者の大半はアセンブラや機械語ではなくFORTRANを使って数値計算を行なっている。又年々法政大学の計算機教育の充実によって計算機人口は着実に増加していることはこの現象を裏付けている。

このように計算機の利用は昔にくらべ容易にはなったが，ある面からみるとそれでもまだ依然としてプログラムを作ることは大変な作業である。そしてまた一度作られたプログラムは何年もの長期間使用するためには改良・修正・保守が必要であるが，これは一般にはかなり困難で労力のかかる作業である。

一方，プログラミングが昔とくらべ容易になったとはいえ，数千ステップ以上の大規模なソフトウェアを作ることは，やはりそう簡単なことではない。大規模なソフトウェアはもちろん，千ステップ程度のプログラムでさえ，作成者から別の人にその維持，管理が譲り渡された後の修正・変更・機能追加などの保守作業は容易ではない。保守作業の容易さは，開発時にそのプログラムの保守の必要性を十分認識し，保守性（maintenancebility）を考慮してあるかどうかに依存している。

従来計算機教育は，ややもするとプログラムの内容面が重視され，プログラミングの方法論は軽視されがちであった。本学計算センターでも，その利用形態が数値計算が多いこともあって，プログラム相談室に持ち込まれる多くのプログラムは，方法論的な工夫はあまり考慮されておらず，ちょうど無計画に増設を重ねた建物のようにそのロジックが入り組み，錯綜していて，それがトラブルの主な原因になっている。

ソフトウェアの品質，信頼性，生産性，保守性，移植性を高め，寿命の長いプログラムを作るためには，プログラミングの方法を検討し改善する必要があるという主張は，システム・プログラムの開発担当者やコンピュータ・サイエンスの研究者の間では，今日重要な議論の一つになっている。その結果HIPO，ストラクチュア・チャート，IPTといったシステム設計の技術論と，

プログラミングの方法としてSP（Structured Programming）などの提案が数多く発表されている。

構造化プログラミングの最初の議論は，Dahl [4] やDijkstra [31]，Jacobini，Bohn [30]，Mills [32] によって提案された。

これらの議論と提案は大別すると四種類に整理される。

(1) 既存のプログラミング言語（FORTRAN，COBOLなど）を使用して良い構造（well structured）で高品質のプログラムを作る方法論。これはプログラミングの標準化を主張し，その具体的な提案をする。[17] [18] [21] [22] [23] [24] [28]

(2) 既存の言語では理想的な高品質のプログラミングのを行なうことは期待できないとし，新たに構造化プログラミングを実現しやすい言語を設計し開発する。

（例） PASCAL [5] [6] [7] [8] [9] [10] [14] [15] [16] MODULER，SIMPL [19]

(3) 既存の言語では構造化プログラミングは実現しにくいので，新たな言語を設計し開発する。そしてpre-processorによって既存の言語に変換する。

（例） RATFOR [27]

(4) 既存の言語に構造化プログラミングを実現しやすいような機能を付加・拡張する。[20] [22]

以下の章では(2)を中心として構造化プログラミングと，その強力な言語PASCALについて述べる。

§2. プログラムの品質性

一般の生産工場では製品の品質管理は生産コストを左右し，商品の競争力を決める上で，重要な要因となっているが，計算機のプログラムについても品質管理はゆるがせにできない問題である。プログラムの品質とは，信頼性，保守性，拡張可能性（変更容易性），汎用性，移植容易性，使いやすさ，およびオブジェクトプログラムの実行効率が考えられる。

これらの7要素はそれぞれ矛盾した作用をし合う。例えば汎用性の高いプログラムを作ろうとすれば，分析—設計—開発—テストの全工程にわたって単能型のものより大作業になり，前述の生産性はある意味では犠牲にせざるを得ないしまた実行効率も下がる。また信頼性や拡張可能性を高めようとすれば，やはり実行効率はその配慮のされていないのよりは悪くなる。実行効率とは，オブジェクトプログラムの実行速度とプログラムの大きさ（実行に必要とする記憶空間の大きさ）などである。

したがってすべての要件を満足させるような解決策はあり得ない。どの要件を重視すべきかは，その作ろうとするソフトウェアの目的と運用，見つもった寿命（いつごろまで使用するか）などに依存する。

信頼性とは，プログラム中のロジカルなミスやエラーがどれだけ存在しないかという事に帰着する。これは一見当然の事ではあるが，実際に1000ステップ以上のプログラムを開発する場合には，すべてのロジックをテストすることは困難であり，完全を期すことは不可能である。「プログラムの正当性（correctness of programing）」の問題は，プログラミングにおける最も重要で，本質的でかつ避けえない問題の1つである。正当性を確保する方法は現在のところプログラムテストしかないが，……それは高価で時間がかかり結局すべての起こり得るケースをテストすることはできないであろう。さらに「プログラムの実験によるテストは，誤りの存在を示すのには使用できるが，誤りがないということを証明するのに使用することはできない。」とWirthは指摘している。〔5, p 20〕実際，かなり深く検討して設計されたプログラムでも，一般にエンドユーザの作ったプログラムは，設計者の仮定した条件と使用者の意図とが異った時に十分そのギャップに甚えられるだけのチェック機能をもった，“強い”ソフトウェアは少い。このことは，そのソフトウェアを拡張しようとするときや，別のコンピュータ・システムに移植しようとする場合に，移植作業を行うのが当の設計者であるとしても，最初に仮定した計算機による制約が異っているために，簡単にはゆかない。このように，一般にプログラムの正当性を証明することは極めて困難であり，我々エンド・ユーザが作ったプログラムは，設計者の仮定した条件のもとでたまたまうまく動いていると言っても過言ではない。

保守性，拡張可能性，使いやすさなどは，それを評価する人の主観によってかなり差異があり，一般論として議論をしたり，何らかの尺度を設けることは困難である。抽象的な表現にならざるを得ないが，これらはそのソフトウェアが製作されてからかなり時間がたった後に，作製者又は別の人がそのプログラムをメンテナンスするのに，容易であるか否かという尺度によって評価される。そしてプログラムの品質を向上させるために，「構造化」が強力な方法として注目されている。

§3. ソフトウェアの生産性と構造化プログラミング

計算機に関する技術革新は著しいものであるが，ハードウェアの進歩にくらべソフトウェアはやや異った状況にある。特に最近では計算機の開発コストはソフトウェアのコストがハードウェアのそれを上回り，「ソフトウェア危機」とさえ言われるようになった。数百万ステップのシステム・プログラムを開発し，商品化する計算機メーカにとっては，このソフトウェアの開発には厖大なコストやマンパワーが必要で，ハードウェアの開発によって新シリーズが発表されるたびに，新しく作りなおし，あるいは改良・更新を行うことは実際には不可能に近くなっているので，危機感をもって認識されることは当然と言えよう。

何年もにわたって開発し，蓄積されたユーザのソフトウエアも計算機のリプレイスなどにも対処して寿命を長く使ってゆくためには，かなりの時間や人手をかけて保守・更新してゆかねばな

らない。さらに又ハードウェアのイノベーションによって計算機自体はより高性能により安価になったが，ソフトウェアを開発してゆく要員はそのテンポよりはるかに低い率でしか補充されていない。もちろん，ハードウェア価格の低下，コスト/パフォーマンスの向上に反して，人件費は常に上昇傾向にある。そこで従来の要員でいかにしてソフトの生産性を高めるかという問題も発生した。

これらの問題を解決するために，HIP O[29]，ストラクチュア・チャート，IPT[29] などのソフトウェア生産技術や，ストラクチュア・プログラミングなどのプログラミングの方法論が提唱されたのである。ここではプログラミングの方法論としてのストラクチャード・プログラミングについて我々とのかかわりを検討する。

限られた要員で限られた期日内にプログラムを作り上げる。本学計算センタの利用者のプログラムを，プログラム相談室に持ち込まれる問題をながめてみると，従来のプログラミングの方法では，利用者は大半の時間とエネルギーを「ディバグ」作業に費していると言える。一般にFORTRAN で問題を解決しようとする場合には，問題の解析に１～２日（５%），設計に１～２日（５%），コーディングに１～２日（５%）カード穿孔作業に１～２日（５%），最初のコンパイルから一応の完成までのディバッグに２～４週間（75%），最終的なテストと本データの処理に１～２日（５%）といった日程配分はよく見られる。

これはもちろん最初から計画的にそうなったというよりは，結果としてこのような傾向が多いので，計画的にプログラム開発を行なわない習慣がこの原因となっている。

一般にメーカーのソフトウェア開発部門でも，設計に25％開発に25％，そしてテストとディバグに50％もの時間が費されているとみられ，このディバグ作業にかかる工数を見なおす動きがあらわれ，これが前記のソフトウエア生産性技術やストラクチャード・プログラミング論の起因になっている。

プログラミングの生産性を高めるための最も効果的な対策は，ディバッグとテストのプロセスを短縮することである。ここで開発に使う言語プロセッサに構造的な機能があれば，ディバッグの能率はかなり上げる事が可能である。

プログラミングとは，ある問題に対して計算機を使って一定の言語仕様に従って，解決をしてゆくプロセスである。人間の問題解決に対するアプローチは一般に次のプロセスで行われる。

①問題の認識　②問題の分析　③問題の本質の理解　④評価基準の設定　⑤解決策を考え出す　⑥解決策の検討　⑦解決策の実行　⑧評価

一方人間ののアプローチのしかたは常に具体一抽象一具体というパターンをとっている。①問題の認識は常に具体的なもの，現象から出発しており，③本質の理解と確認はその抽象化である。

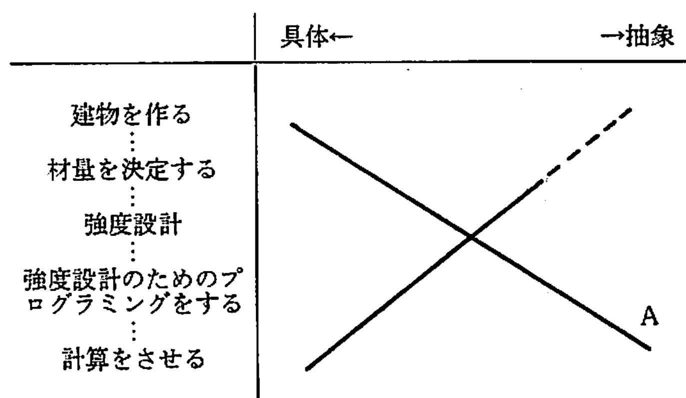

図1. プログラム作成の抽象―具体

そして次に，⑤解決策は常に具体的なものであり，このように具体化に還元される。

また，抽象化具体化はそれを見る立場によって相対的な関係として表現できるのであって，絶対的な抽象化とか具体化はありえない。例えば，ある高層建築を作ろうとする問題が存在したとする。そこでは，まずある建物を作るという事が，現実的な，具体的な問題である。そして次にそのために，どんな工法を採用するか，どんな構造にするか，どの位の強度が必要かといった，ある建物を作るという現実の問題から考えれば，相対的に抽象化された分析と認識・検討が必要となる。そしてそれはさらに例えば構造上の強度を求ゆるために計算機で“強度計算”をするという。建物を作るという観点からは抽象的な作業・手順が必要となる。そしてその後，

このプロセスを，次のように整理する。

問題――本質・目的の理解・把握――→分析――→プログラミング
（具体）（抽象化）　　　　　　　　　　　　（抽象化）

さらにプログラム作成のプロセスは，目的，方法（使用する言語，解法，アルゴリズム，テクニックなど）条件（使用できるコンピュータ・リソース，精度，速度，容量……）インプット・アウトプット・データの仕様設計などのサブプロセスに分解できる。ここで最初のステップは目的を明確にすることであり，それはプログラミングという作業からみると最も抽象的な作業である。例えばある建築物の強度計算をするといった，計算の目的は，計算機のプログラミングからみれば，どんなアルゴリズムを採用するかとか，入出力データ仕様をどのようにするか，どのarrayをどんな手順でどんな計算をするかを選択する作業にくらべ，抽象的である。

我々はここで１つのプログラムを，この抽象化された目的に対応させ，「１つの目的を解決するための手続き」と認識することにする。この手続き（procedure）は，そのプログラムの内で何がおこなわれていようと，ユーザは，入力仕様に従って計算させたいデータを入力し，出力仕様にそった計算結果を入手することができる。その意味で，この手続きは，ある目的にそって，入力データを加工（計算）し，出力データを戻してくれるブラック・ボックスと考える事ができる。

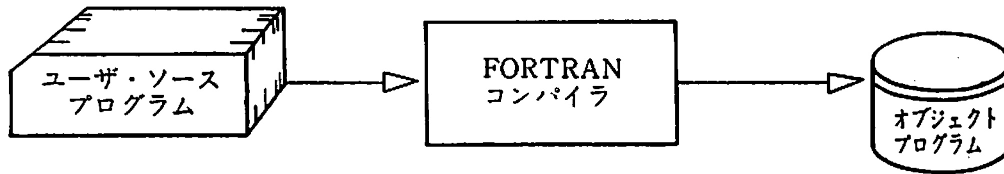

図2. コンパイラと入出力データ

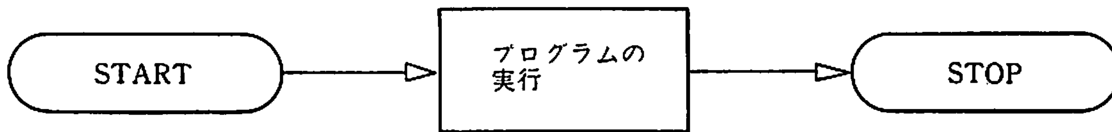

図3. プログラムの制御の基本的な流れ

我々はふつう，無意識に計算機のソフトウエアをブラックボックスとみなしている。例えばFORTRANコンパイラは，FORTRAN言語の仕様で書かれたユーザのソース・プログラムを入力データとし，それを加工してオブジェクト・プログラムを出力する。このとき，コンパイラのなかの作業や，どのようなアルゴルズムが使われているかなどをエンドユーザは通常は問題にはしないし，むしろ知ろうともしない。エンドユーザのニーズは自分の入力したソース・プログラムから，意図どうりの演算ができるオブジェクト・プログラムが作られることである。

さらに，手順についてみると，1つのプログラムは，決ずstart→procedure→stopという順序で実行される。どんなプログラムも，この形で完結していなければならない。(完結しているという表現をもう少し厳密に区別すると，そのprocedureが正常に終了した場合と，procedureの実行中に作成者が予期しない異常が発生し，オペレーティング・システムによって強制終了される場合とがある。)

このようにプログラムを「1つの目的（処理）を行う完結した手続き」としてとらえることを，プログラミングの基本的な概念として導入する。

プログラムの構造化はここから出発し，基本的にこのブラックボックスとしてのプロセジュアの構造を保ちながら，より下位の入れ子（nest）構造のプロセジュアに分解してゆく。「ある処理を行う」という大目標に対して，より具体的なアルゴリズム決定し，下位のプロセジュアを決定し，プログラムをより細部のモジュールに書きおろしてゆく。この過程ではプログラミングを抽象から具体化という作業をtop-downでおこなう。

§4. プログラムの基本構造

以上の考え方を基礎に構造化プログラミングでは，プログラムを4.2で示す4つのパターンに分類する。

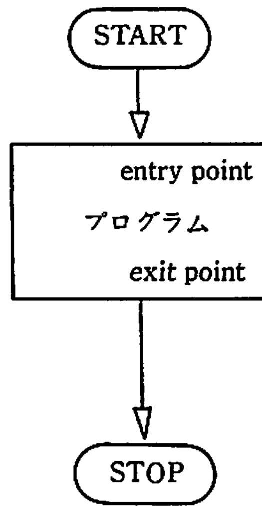

図4. 基本型

4.1 基本型

Proper Program（正しいプログラム）は1つの入口と1つの出口を持つ完結したモジュールである。

これについては，[1] で，その証明のスケッチが述べられている。

そのプログラムのどの部分をとっても，入口（entry point）から入ってこの部分を通り，出口（exit point）に出てゆく径路が必ずとれるもの一すなわちプログラムのどの部分も孤立していないものとなっている。

4.2 モジュール化

(1) 線形シーケンス（Concatenation）

複雑な手順も，全体をいくつかの部分（モジュール）としてとらえることができる。その分解の第1のステップは，単順な手順の連続化である。（procedure がシリアルに処理される。）ここでは連続する手順が1つ1つの命令と考えてもよいし，あるまとまった手順（即ちモジュール）と考えてもよい。

(2) 選択パターン（条件制御，Selection）

次に条件付の手続きを，それ自体が1つの閉じた手続きの中で実現し，モジュールとして完結させる。それによって，この条件による分岐もそれ自体は基本形に準じた「単純な流れ」を構成する。

このために，一般に次の二つの文法が提供される。

if 〈C〉 **then** 〈S_1〉 **else** 〈S_2〉　　図5. b1

it 〈C〉 **then** 〈S_1〉　　図5. b2

if 〈C〉 **then next sentence else** 〈S_2〉　　図5. b3

さらに，この条件分岐は nest 構造を作ることができる。

if 〈C_1〉

　then

　　if 〈C_2〉 **then** 〈S_{21}〉 **else** 〈S_{22}〉

　else

　　if 〈C_3〉 **then** 〈S_{31}〉 **elte** 〈S_{32}〉;　　図5. b3

これらの条件分岐は，**then** 及び **else** に属する処理が実行された後，図5 b1，b2，5 b1，b3 に示されるように，分岐に対応した1つの exit point にコントロールが集約される。その結果，条件分岐 if ステートメントも，常に1つの入口と1つの出口とを持つ構造をしている。

また入れ子構造をしている複合 **if** 文は，図5. b3 に示すように分岐条件を節（node）とする木

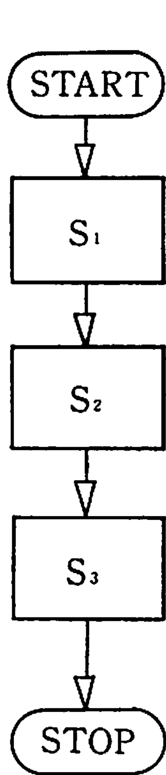

図5. a
モジュールの連続

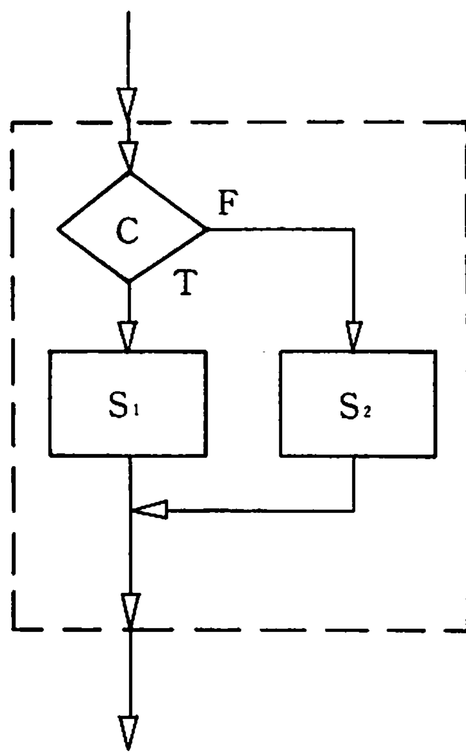

図5. b 1
if 〈C〉 then 〈S_1〉 else 〈S_2〉

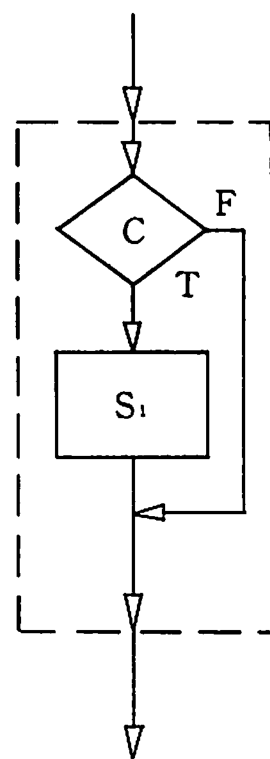

図5. b 2
if 〈C〉 then 〈S_1〉

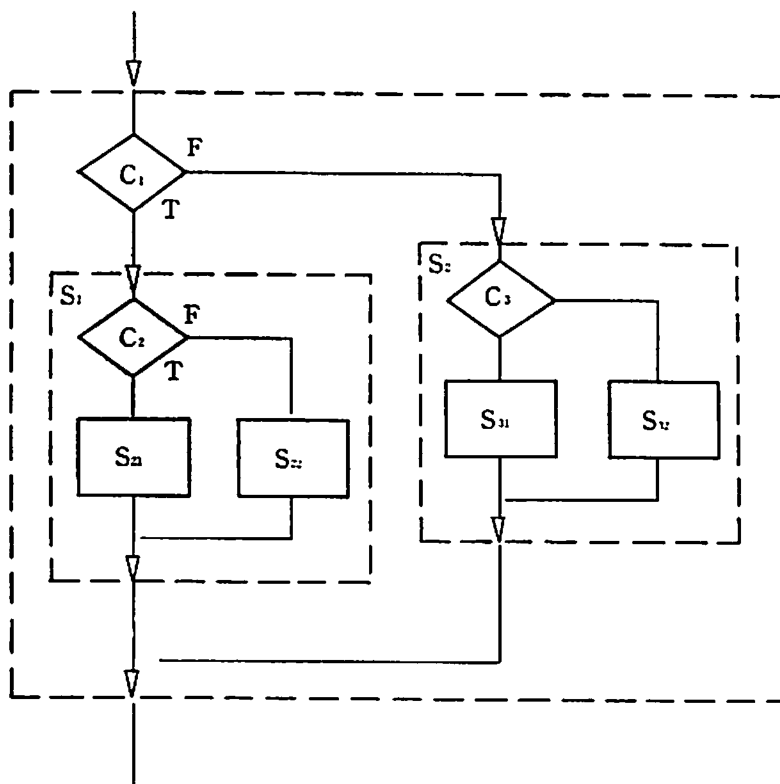

図5. b 3

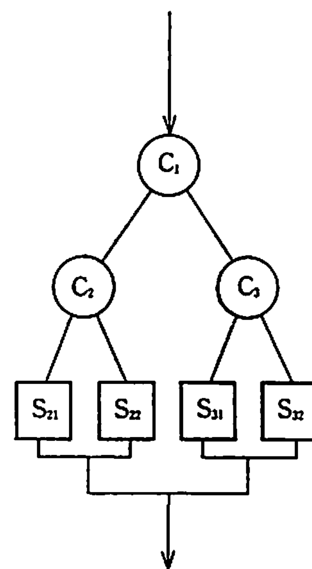

図5. b 4

構造と考える事ができる。このように分岐された先の実行命令も，すべて入れ子型の複合化された命令（ステートメント）を記述することが許され，それによってプログラムは記述上は多層の入れ子をなすが，それをブロックとみなすことができ，その時には制御の流れは常に start から stop へ一意に決ってゆく。

b 2，b 3 の構造は，本質的には b 1 と同じもので，b 1 の変形である。b 4は，2つの **if** 文が，上位の1つの **if** 文の下位に入れ子として置かれている。点線のボックスで示された S_1 は，b 1 のそれに対応し，ここでは S_1 が **if** 文になっているだけで，**if** 文も1つの入口と1つの出口からなる手続きという点では，単純なモジュールと同じに扱うことでできる。このように，入れ子型を作ることによって，どんな複雑な分岐も常に単純な流れに変換して表現できる。

(3)　多分岐条件制御

if 〈C〉 then 〈S_1〉 else 〈S_2〉 の特殊なケースとして，選択される文が多数あるような構造は，case 文で表現する。

case 文では選択子（selector）の値が，ケース・ラベルに属する命令〈Sn〉が実行される。小規模なテーブル・ルックアップはこれで十分対処できる。

(4)　ループの制御パターン（repetation）

プログラミングには，ある処理を繰り返えす「ループ」が必要である。**if** 文のような条件分岐命令だけでも，ループを作り出す事は不可能ではない。しかし **if** 文を使ってループを作るためには，本来プログラムのアルゴリズムには無意味なラベル（プログラム上のある“ところ”につけた名前）を設置し，**go to** のような無条件ジャンプ命令が必要となる。この2つのわずらわしさからプログラマを開放するため，構造化プログラミングは3つの基本的なループ制御命令を

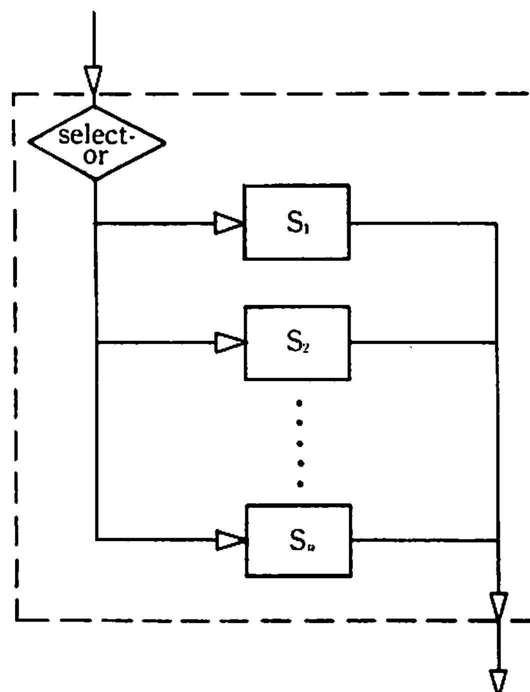

図6. 多分岐条件文

case 〈selector〉 **of**
〈case label 1〉 : 〈S_1〉
〈case label 2〉 : 〈S_2〉
⋮　　　　⋮
〈case label n〉 : 〈S_n〉
end

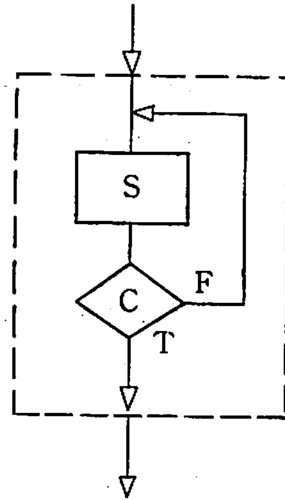

図7. a
repeat 〈S〉 until 〈C〉

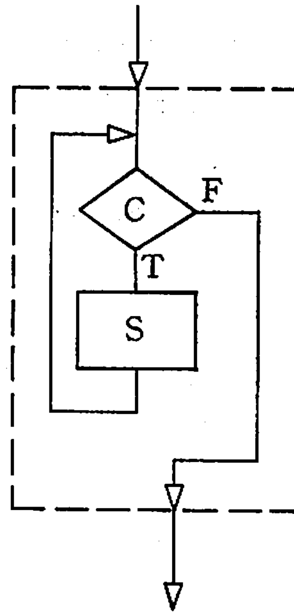

図7. b
while 〈C〉 do 〈S〉

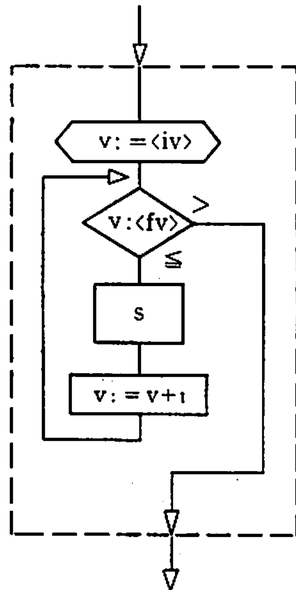

図7. C1
for 〈v〉:=〈iv〉 to 〈fv〉 do 〈S〉

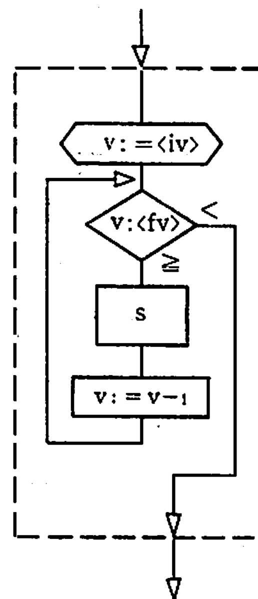

図7. C2
for 〈v〉:=〈iv〉 downto 〈fv〉 do 〈S〉

提供する。

(1) **repeat** 〈S〉 **until** 〈C〉　　図7. a

(2) **while** 〈C〉 **do** 〈S〉　　図7. b

(3) **for** 〈v〉:=〈iv〉 **to** 〈fv〉 **do** 〈S〉　　図7. c1

(4) **for** 〈v〉:=〈iv〉 **downto** 〈fv〉 **do** 〈S〉　図7. c2

repeat

repeat 文は条件〈C〉が成立する（**until** 〈C〉）まで命令〈S〉を繰り返えす。

while 文

while 文は，**repeat** 文と対応する構造をもち，ある条件〈C〉が成立するあいだ（while），命令〈S〉を繰り返えす。**repeat** が，〈S〉を実行した後に，条件〈C〉を評価するのに対し，**while** では先に条件〈C〉を評価し，**false** である間〈S〉を繰り返し実行する。

repeat，**while** とも，〈S〉の実行中にいつか〈C〉が変化する手続が，〈S〉に含まれていなければならない。

for 文

for 文は制御変数 v を持ち，初期値（initial value）から終端値（final value）までの間を v が 1 ずつ増加（to）又は減少（down to）し，その間，命令〈S〉を繰り返えす。

これらの繰り返し制御のための命令はプログラムのループを作る際に強力な機能を持つ。繰り返し制御は，プログラミング時の不注意による，エンドレス・ループを避けることが重要な目的で，従来の言語にくらべるとかなり改良されてはいるが，これだけでエンドレス・ループを完全になくす事はできない。例えば制御条件〈C〉が変化しないようなプログラムでは，エンドレス・ループを防ぐ事はできないからである。

4.3 構造化プログラミング論の整理

以上の議論は以下のようにまとめられる。

(1) プログラムは，start から end まで常に 1 つの方向に制御が流れる。ここでは 1 つの入口と 1 つの出口からなる閉じた論理構造をなし，それはすなわち入れ子型をした，あるいは単純に連続したブラックボックスと考えることができる。ここでそれぞれのレベルで閉じたブラックボックスを，モジュールと考えることができる。

(2) プログラムはモジュールの連続したシーケンスで制御できる。

(3) 閉じたモジュールのなかで，条件によって分岐する機能を持つ。

(4) 閉じたモジュールのなかで，ループを構成する制御構造を持つ。

§5. 思考の一次元性と表現の二次元性

次にプログラミングと人間の思考との対応をながめる。人間の思考は，一般に 1 次元的であり，シリアルに考えを展開させる。

問題が複雑になるほど人間は思考の過程では 1 次元的になり，条件分岐が多くなると，そのいくつかの分岐条件と状況を一時保留しながら考えを先にすすめることが困難になる。分岐が発生すると一つの条件と状況をプッシュダウン・スタックに入れておき，残った分岐先の処理を解決した後に，スタックに入れておいた分岐先の処理を行うのは，人間でも計算機でも類似している。ただ人間の場合はこのスタックの深さ，すなわち分岐の入れ子の深さはあまり多くない。

そこでフローチャートという2次元的な表現を，プログラミングの補助的な手段として使用ことが多い。これは，2次元的な平面の上に記述するドキュメントであるので，条件分岐や，複雑な入れ子構造が極めて明快に整理できる。

そのため，プログラムのドキュメンテーションとして，あるいは又ディバッグの手段として強力な手法である。

しかし，プログラム自体は，コーディングも，オブジェクトモジュールの制御の流れも，1次元的であるため，フローチャートと，プログラミングの間には，フローチャート上の表現をそのままコーディングできないというギャップが存在する。フローチャートは人間の思考を整理するのには極めて有効な1つの手法ではあるが，それをプログラムにコーディングする際には，もう一度人間の思考でフィルターをかけ，1次元的な表現に変換する作業が不可欠である。

さらに，通常完成したプログラムと，それを作る時に使ったフローチャートは，必ずしも対応してはいない。何故ならフローチャートは思考の補助的な手段でしかなく，フローチャート上でのロジックの完成をテストする方法はないに等しいので，フローチャートに従ってプログラムをコーディングした後，計算機にかけてロジックのテスト（すなわちディバッグ）を繰り返した後に，一応完成したプログラムができ上る。このディパッグの過程で発見されたロジックのミスやエラーは，プログラム上では修正されるが，普通フローチャートまでは修正されない。プログラムが完成された際に，改ためてフローチャートを書きなおせばよいのであるが，通常はその時はすぐ次の仕事が控えていて，フローチャートの書きなおしがされる事はほとんどない。

又，完成されたプログラムと，それに対応するフローチャートがあったとしても，プログラムは性能を向上させるためとか，別の計算機にも使えるようにするため，あるいは発注者のニーズが変更されるために，修正・更新が必要となる。このようなプログラム・メンテナンスの際に，そのフローチャートまでも修正・更新してゆくことは，かなり大変な作業であり，通常ほとんどなされてはいない。

完成したプログラムから，*xy* プロッタやグラフィック・ディスプレイを使ってフローチャートを描き出し，それをプログラムの保管・管理と連動すれば，フローチャートとプログラムの一対一の対応は可能ではある。

このように，フローチャートは，完成したプログラムのドキュメントとしては，あまり活用されてはおらず，プログラム設計時の思考を整理する手段としてしか機能していないと言ってもよい。

このように，フローチャートはプログラミング及びプログラムのメンテナンスには必ずしも有効な手段ではなくなっている。そして，もしそうであるなら，フローチャートに頼らないプログラミングを実現する方が，プログラミング時の不要な作業を除去し，プログラム作成者の負担を軽減する事になる。構造化プログラミングでは，これらの点でも，プログラム自体のドキュメン

ト性を高め，フローチャートなどの表現手段の助けを貸りなくても，プログラム・リストを読めばその内容が理解できることも重要な目的とする。さらに，1次元的思考を，2次元的表現になおし，そして又1次元的なプログラムに変換するというわずらわしさも，解放されるという長所を持っている。

これらの長所を持つ構造的な概念を導入することは，プログラミングの生産性と品質を高め，保守を容易にし，プログラムの寿命を長くするために，強力な方法である。

§6. PASCAL概要

構造化プロケラミング言語「PASCAL」はN. Wirthによって1971年に発表された。Wirthはその少し前にALGOL-Wと呼ばれる言語を提案しているが，PASCALはALGOL系の言語仕様を基礎にFORTRAN, PL/1, COBOLの機能を導入している。

PASCALは最初CDC-6600にインプリントされ，その後各種の計算機システムにインプリントする作業が多くの研究者によって試みられた。[11] [12] [13] [16]

PASCALの特徴は次の点に集約される。

(1)　構造的プログラミングを意図したプログラム言語である。

(2)　プログラム構造

(2.1)　宣言部と手続き部を積極的に分離した。

(declaration と Procedure)

プログラムの入れ子構造

データのスコープ（データの有効範囲の明確化）

(2.3)　ブロック構造によって，プログラムをブロックにまとめ入れ子構造を明確にした。

(begin-end)

(2.4)　制御構造をより明確に示すステートメントを提供した。

if—then—else—, case…,

repeat—until—, while—do—, for—to—do—

(3)　豊富なデータ構造

(3.1)　基本型（unstructared type）

boolean, integer, real, character, scaler, snbrange

(3.2)　構造型データ（structured type）

array, recorde, set, file, pointer

(3.3)　既に定義してあるデータ型を組み合せて，ユーザが任意に新しい構造のデータ型を定義できる機能。

(4)　明快な文法と明快なコーディング記法

(4.1) BNF による文法記述によって言語仕様が，明確にされている。

(4.2) 構文図式 (Syntax diagrams) による文法記述

(4.3) 従来の言語のあいまいな文法，不統一な仕様を改善し，言語として，又コーディングにおいても仕様を統一しあいまいさを排除した。

—assignment operator (代入演算子：＝) と relational operator (関係演算子＝) の明確な区別。

—名前 (identifire) と予約語 (reserved word)

—コーディング上の自由さ

桁位置やフィールドの制限のない自由な記述

コメント文の任意な位置への記述

—標準入出力書式の提供

(5) 再帰呼出機能 (recursive call)

(6) TSS で使用することを目的に開発されたが，バッチで使っても不自由はない。(Batch—TSS の互換性)

(7) 用途

(7.1) プログラミング教育に好適

(7.2) 大規模ソフトウェアの開発に強力な TOOL となり得る。

(7.3) システム・プログラムの開発に強力な機能をもつ

§7. PASCAL のインプリメンテーション

PASCAL の文法は，参考文献 [6] [7] [8] [9] [10] [14] [15] [16] に詳しく述べられているので，ここでは述べない。我々が法政大学計算センターのシステムにインプリメントした PASCAL は，東大大型計算センターで稼動しているシステムと一応互換性のある言語仕様である。制限される点は，**goto** 文，および，recursion の nest が浅い点，文字の内部表現などである。

我々がインプリメントした PASCAL の問題点を次に述べる。

(1) コンパイラがまだ熟成していないため，エラーメッセージは十分親切であるとは言えない。ユーザのプログラムを構造的に記述するという主目的に重点がおかれていて，言語プロセッサの副次的な機能やサービスについては，既存のメーカ提供のコンパイラに比べると，十分親切ではない。

(2) プログラムのオブジェクト保管の考え方が弱い。

プログラミング教育に主眼を置いたためと考えられるが，サブプログラムもすべてソースのまま保管する。そのため大規模なソフトウェア開発の TOOL としては，ややわずわしい。将来の版ではオブジェクト・モジュールのリンク機能は追加されるであろう。

(3) 入出力機能が，まだ不十分である。

入出力書式 (FORMATING) 及び，ファイル・アクセスも，改良の余地がある。

(4) 文字セット，特に小文字

元来 TSS で使用する目的で作られているため，Wirth の提案の版では小文字の使用が可能であるが，Batch で使用するためには，入出力機器の制限から小文字が使用できない。

§8. おわりに

大規模ソフトウェアの開発及びその保守は，従来の無方針な手工業的なプログラミングから，構造的な手法の導入によって一定の効果を上げることができる。構造的な手法は，システムの設計からプログラミングまで，各レベルでそれぞれ定式化され，体系づけられるべきである。我々は，構造化プログラミングの考え方と PASCAL の言語仕様と機能を参考に，他言語の記述の標準化案を作ることを次の課題と考えている。

本学計算センターの FACOM 230—45 S で稼動している PASCAL の使用方法は，センター内資料「ジョブ制御マクロ PASCAL」などに記されている。

最後に PASCAL のインプリメンテーションにあたって，全面にわたる支援をいただいた電気通信大学武市正人氏と，貴重なノウハウを提供して下さった東京電機大学藤斉剛氏，有効なアドバイスを下さった法政大学計算センター前副所長山下清明先生および作業に援助して下さった法政大学計算センター諸氏に謝辞を述べる。

参考文献

[1] 高信頼性ソフトウェアー複合設計 Glenford J. Myers 著 日本 IBM 久保未沙，国友義久，共訳 近代科学社

[2] ソフトウェアの信頼性 ——ソフトウェア・エンジニアリング概説 G. J. Myers 著 有沢誠訳 近代科学社

[3] 階層システム論 M. D. Mesarovic, D. Macko, Y. Takahara 共著 研野和人訳 共立出版株式会社

[4] 構造化プログラミング O. J. Dahl, F. W. Dijkstra, C. A. R. Hoare 野下浩平，川合慧，武市正人共訳 サイエンス社

[5] 系統的プログラミング入門 (原題 Systematic Programming : An Introduction) N. Wirth 著野下浩平，箕捷彦，武市正人共訳 近代科学社

[6] PASCAL USER MANUAL AND REPORT K. Jensen. N. Wirth Spring-Verlag

[7] Algorithms+Data Structures=Programming Niklaus Wirth. Prentice-hall

[8] The Programming Language Pascal N. Wirth Acta Informatica 1. 35—63 (1971) © by Spring-Verlag 1971.

[9] An axiomatic detinition of the Programming language PASCAL C. A. R. Hoare and N. Wirth Acta Informatica 2. 335—355 (1973)

[10] Critical Comments on the Programming language PASCAL A. N. Haberman PB 224 777 NTIS National Technical Information Service

[11] A PASCAL Compler for CDC 6000 Computer and its Tranterring to IBM 360/370 Com-

puters M. Iglewsky, A. Krepsky, M. Missala EIK11 (1975) 4—6, 352—359

[12] Implementation of a PASCAL Compiler for the IBM/360 David L. Russell and Jeffrey Y. Sue Digitall System lab. stanford. UNiV

[13] The PASCAL ⟨P⟩ Compiler: Implementation Notes K. V. Non, U. Ammann K. Jensen, H. H. Nägeli

[14] プログラミング言語 PASCAL の文法 N. Wirth 和田英一訳 bit vol. 8 No. 4 (1976)

[15] プログラム言語 PASCAL 和田英一 bit Vol 10 No 1. 2. 3. 4. 5. 6. 7. 8. 9. 10.

[16] PASCAL 入門 ——PASCAL 8000 に沿って 疋田輝雄，安村通晃，石畑清 東京大学大型計算機センター

[17] プログラム書法 Brian W. Kernighan and P. J. Planger 木村泉訳 共立出版

[18] FORTARAN はいつ亡びるか 木村泉 bit vol. 7. No 3. No 4. No. 5. No. 6. (1977)

[19] SIMPL の構文法とコンパイラの概要，コンパイラの作り方 中西正和，大野義夫 bit Vol. 9. No. 1. No. 2. No. 3. Vol. 4. 5. 6. 7. 8. 9. 10. 11. 12 Vol. 10. No. 1. 2. 3. 4.

[20] Mark Ⅲ FORTRAN 77 要説 bit Vol. 9, No. 14

[21] modular prgraming in COBOL Russell M. Armstrong 著 大日方真訳 TBS 出版会

[22] COBOL によるストラクチュアード・プログラミング H. P. スティーブンソン編 日本経営協会訳

[23] 構造的コボル教則本 マリー・テレーズ・ベルティン，イヴ・タリノー共 山崎利治訳 TBS 出版会

[24] 木構造によるプログラミング 鈴木芳雄 NOHMA

[25] 新しい世代のプログラマのあり方 ソフトウェア工学的視点からの反省 米口肇 bit Vol 10. No. 5

[26] クタバレ「あみもの的」軟件製造法 石田康勝 bit Vol 10. No. 4, No. 5

[27] A RATFOR-FORTRAN TRANSLATOR in SOFTWARE TOOLS chp. 9 Brian W. Kernigham, P. J. Planger Addison-Wesley Publishing Company

[28] 整構造 COBOL ——ストラクチャード・プログラミングをめざして 真野芳久，植村俊亮 bit Vol 10. No. 9. 10. 11……

[29] 効果的プログラム開発技法 [1]-[6] (IBM 出版物) ACCESS 1977 13, 15, 16, 17, 18

[30] Flow Diagrams, Turing Machines and Languages with Ohly Two Formation Rules C. Bohn, G. Jacopini C. ACM Vol. 9 No. 3 (1966)

[31] GO TO Statement Considered Harmful E. W. Dijkstra letter to Editor C. ACM Vol. 11, No. 3 (1968)

[32] Mathematical Foundation for structured Programming H. D. Mills IBM Corp. FSC72—6012 (1975)

[33] A Genealogy Contral Structures Henny F. Ledgard, Michael Marcotty UNIV. of Macsachusetts C. ACM Vol. 18, No. 11 (1975)