Programming System の発展* (2)

戸島 潤

目　次

3. FŌRTRAN の出現

IBM 701 は信頼性の低い cathode ray tube による記憶方式をとっていた。磁気 core が利用できる様になってからは，701 に磁気 core をつけた 701 M がつくられるという噂があったが，704 という新しい computer がつくられることになって，701 M の idea はすぐ立ち消えになった。704 は過去に programming system がつくり上げてきた種々の便宜を hardware として実現しようとしていた。自動浮動小数点演算装置や index register は Speed Code の様な interpretive system を不要のものにしてしまった。704 の 1 号機は 1956 年 1 月に出現して，いよいよ超高速計算機の時代が開幕したのである。

ところで，704 の hardware の開発と平行して，John Backus を長とする group によって 704 を使って科学＝技術計算を行なうことを容易にするための新しい compiler を開発するという project が進められていた。この project こそが FŌRTRAN をつくり上げることになったのである。この FORTRAN 開発 group は IBM から Backus など 9 名，MIT, California 大学，Radiation Laboratory, United Aircraft Corporation から各 1 名，計 13 名からなっていた。programmer が数学の表現方法の様な明快な言語を使って数値計算の手続を示し，その手続を遂行する能率的な 704 の

* 原稿受領 1970 年 1 月 13 日

program を自動的に作成する system をつくることが目的である FŌRTRAN project は 1954 年の夏から開始された。FŌRTRAN 語の specification はほとんど project の初期にきまってしまっていた。その当時は，ほとんどの programmer は symbolic machine instruction のみを使っており（1部のものは octal ないしは decimal machine instruction を使っていた），誰もが機械による coding は programmer が現にもち，そしてもつことが必要な融通に富んだ器用さがつくり出す program に全くかなう筈がないと信じていた。それゆえ，compiler がつくり出す program は人間が coding したものにくらべると非常に能率が悪いと考えられていた。そして，このことを裏付ける様な確固たる証拠があったのである。例えば，ある compiler は

$A=B+C$

に対応する machine instruction として次の様なものをつくり出した。

```
LŌAD   B
STŌRE  WS1
LŌAD   C
STŌRE  WS2
LŌAD   WS1
ADD    WS2
STŌRE  WS1
LŌAD   WS1
STŌRE  A
```

実際にはこれらの instruction の間にはさらにいくつかの NŌ-ŌP が加わるのである。FŌRTRAN 語の設計が開始されて完了したのはこの様な雰囲気の中においてであった。そして，この system は 704 **のみ**に対して適用するものと考えられた。当時は各種の machine にかかる machine-independent source language というものにはなかなか思い至らなかったのである。FŌRTRAN group の関心はもっぱら programmer がつくり出す program に匹敵する program をつくる automatic coding system は可能であるかどうかということにあった。答はかなり悲観的であった。このことは苦心してつくり上げた translator program を動かしてみた時にたしかめられることになった。translator がつくり出した program は同じ問題に対して人間がつくった program の半分の能率しかもたなかったのである。そのために FŌRTRAN automatic coding

system は一般にうけ入れられるには至らないと考えられた。

そこで，computer がつくり出した code のどこが inefficient であるかということが問題にされた。まず，算術式を計算する部分についての inefficiency はいくつかの簡単な工夫でとりのぞかれた。そして，能率の悪さの最大の原因は配列の要素の番地を求めるためにいくつかの乗算を行なわなければならない address calculation にあることがわかってきた。もし配列の要素が順番に使われるなら，人間のくむ program では前の要素の address に適当な定数を加えることによって次の要素の address を計算するであろう。一般に，A_{ij} の address は

$$(i-1)+d_1(j-1)=-(1+d_1)+i+d_1j$$
$$\equiv k_1+i+k_2j$$

を計算して，これを A_{11} の address に加えることによって求めることができる。ここで DIMENSION $A(d_1, d_2)$ と宣言されているものとする。そこで，毎回上の様な計算を行なわないためには，translator は i, j が毎回定った大きさだけ変ることを確めたうえ，A_{ij} の address をどこかに保存しておき，A_{ij+d} に対しては A_{ij} の address に k_2d を加える code をつくり出す様になっていなければならない。そこで，配列の要素の reference を 2 つに区別して考える必要がおこってくる。ひとつは，前に reference が行なわれた配列の要素の番地に一定数を加えて増加させる場合であり，他は始めから address calculation を行なう場合である。ここで，前の計算過程をかえりみて，どこで i 又は j に定数を加えることによって modify を行なっているかを知ることはかなり厄介であって，その点で A_{ij} の reference を能率よく扱うことには困難さがつきまとう。従って，すべての場合を能率よく扱うことはできないことが明らかになった。例えば，$A(I, J)$ への reference と $J=J+F(K)$ という statement を含んでいる loop では，次の場合にのみ $A(I, J)$ の番地は定数を加えることによって求めることができる。すなわち，K が loop の中で constant である時，F が K の 1 次式である時，K が loop の中で 1 次式であらわされる変動をする時などである。そこで，うえの様な loop であれば，translator はまず loop の中の K の変動をみて，それが 1 次の変動であるならば，適当な増加分を計算すればよいということになる。ただし，$A(I, J)$ への reference がひとつの loop に属するばかりでなく，I, J に関して他の取扱いをしている様な loop に同時に属していては困るので，そうなっていないことをたしかめなければならない。以上の様な考察の帰結として，配列の要素の address

calculation がどの様な場合に能率よく行なわれて，どの様な場合に能率よく行なわれないかを user に知らせる必要があると考えられた。そして，DŌ statement がすでに user の便宜のためにつくられていたので，この DŌ loop の中で index を添字としてもっている配列の要素の番地計算は定数を加えることによって行ない，そうでない配列の要素の番地については始めから address calculation を行なう様にすることにきめられた。この様なやり方は DŌ loop が nest になっていても同じく適用できた。さらに，index の値の test, modification, initialization はできるだけ能率よく行なう様にされた。こうして能率性のために language に対していくつかの制限が課されることになったのである。

(1) DŌ loop の index variable は一定値だけしか変ることができない。しかも，704 の制約からその値は positive に限られた。

(2) subscript expression は index variable に関して linear でなければならない。

(3) subscript の次元は 3 次元をこえてはならない。

(4) DŌ loop の中へ外から control を移すことはできない。

この他に，translation process を speed up するために，例えば，名前は 6 文字をこえてはならないという様な minor な制限がおかれた。FŌRTRAN group はこの様にして translator から generate される object program からあらゆる能率の悪さを取除くために非常な努力を払ったのである。あらゆる好奇の目がこの点に向ってそそがれた。例えば，translator は program の flow がどの様になっているかという分析を行ない，Monte Carlo 法で running の simulation をするという様なきわめて複雑な方法によって index register allocation の問題の解決を計っていたのである。この様なことの結果として compilation に要する時間はしばしばきわめて長いものになった。しかし，その代り，object program の能率のよさは申し分がない様になった。ことに，DŌ loop が使われ，source program 自身も能率的につくられている時は，十分人間がつくった program に比肩しうる program がえられた。このため，FŌRTRAN は compilation の遅さにもかかわらず，これがつくられた当時の programming の負担を非常に軽減することになり 704 を多くの人が使える様にしたのである。computer の user も object program を調べたうえで充分使いものになることを認め，次第に FŌRTRAN system をうけ入れる様になっていったのである。この FŌRTRAN project は 1957 年 1 月に完了している。その後の FŌRTRAN language とその

compiler の発展は改めて述べるまでもないであろう。compilation が遅いという点も1958年に FŌRTRAN II が完成して克服された。

FŌRTRAN II のそれ以前の FŌRTRAN に対する違いは，ひとつの job を行なう program が，独立に test し debug できるいくつかの subprogram の集りからなってもよい様になった点である。さらに，symbolic machine code で subprogram をかくこともできる様に拡張された。その他，alphanumeric information の入出力ができる様になった点も最初の FŌRTRAN と違う点である。ところで，FŌRTRAN II の compiler は最初の FŌRTRAN の compiler をただ単に拡張したものではない。compiling の speed up のために最初の FŌRTRAN compiler で使われた idea のうちのいくつかは余儀なくすてさられた。例えば，index register allocation technique のある部分は FŌRTRAN II compiler には含まれていない。しかし，全体として FŌRTRAN II は最初の FŌRTRAN の思想をうけついでおり，従って generate する object program の能率性は，のちの FŌRTRAN IV compiler が generate する program よりもよいといわれている。IBM 7090 (1959年11月) に対して FŌRTRAN II compiler がつくられたのは，もっとも早いもので1959年1月であるが，この頃から FŌRTRAN II は非常な勢で普及し始めている。そして，いくつかの点で単純化された内部構造をもち，programming language がさらに拡張された**新しい** compiler が次第に user によってのぞまれる様になってきた。FŌRTRAN IV はこうした背景の下に7090に対して1962年12月に完成している。従って，FŌRTRAN IV は単に programming language が拡張されているばかりでなく，compiler の内部設計の点でも FŌRTRAN II とはかなり異なっているということに注意しなければならない。まず，FŌRTRAN IVの language に追加された主要な機能は次の通りである。

(1) CŌMMŌN 宣言ができる様になった。

(2) 2倍長演算と複素数演算が可能になった。

(3) 論理変数と論理演算子が加わった。そのため logical IF がつけ加わった。

(4) DATA statement が加わった。

(5) TYPE 宣言ができる様になった。

(6) I/Ō statement から TAPE, PUNCH に refer しているものは除かれた。また，QUŌTIENT ŌVERFLŌW を判定する機能なども除かれた。

compiler 作成の指導原理も最初につくられた FŌRTRAN compiler の当時からみる

と全く変ってしまっている。すなわち，compiler は generate される object program の能率性よりも compilation の能率性に重点をおく様になった。これは source program を debug したり修正したりする間にくりかえして compiling が行なわれることを考慮して，そのことに最大の便宜を与えようとするものである。結局，programming language (problem-oriented language) を使う最大の利点は試行錯誤的に program を修正して行くことを容易にする所にあると考えられるのである。最近の compiler はますますこうした色彩を強めている。

さて，FŌRTRAN II による FŌRTRAN 人口の増大の事実は 50 年代にはそれほど知られていなかった。そのため，もし，そのことがもっとよく認識されていたならば，ALGŌL project なども現在知られているものとは違った形になったであろうといわれている。

参考までに，ここで，IBM の FŌRTRAN project が最初につくり上げた FŌRTRAN compiler の構造の大綱を述べておこう。

compiler 全体は次の様な連続した 6 つの section からなっている。

Section 1 : statement をよんで分類する。算術式ならば object instruction をつくりだす。算術式でなければ部分的に compile して，残りの情報を table に記録する。ここで compile されたすべての instruction は CŌMPAIL file に入れられる。

Section 2 : DŌ statement や subscripted variable から結果する indexing に関係した instruction をつくりだす。これらの instruction は CŌMPDŌ file に入れられる。

Section 3 : CŌMPAIL file と CŌMPDŌ file を統合して 1 つの file をつくる。一方，section 1 でやりかけた算術式でない statement の compilation を完了する。ここで，object program ができ上るが，しかし，この program は index register が無数にある computer を想定している。

Section 4 : object program の flow を分析し，section 5 の利用にそなえる。

Section 5 : object program を index register が 3 つしかない 704 に合う様に変換する。

Section 6 : object program を assemble し，すぐ実行にとりかかれる様な relocatable な binary program をつくりだす。しかし，指定によっては SAP 語の object program をつくりだすこともできる。

compiler の中で情報は 2 つの形をとって，section から section へ移る。1 つは compiled instruction としてであり，もう 1 つは table としてである。CŌMPAIL file と CŌMPDŌ file (これは後には 1 つになる) に入っているひとつの compiled instruction は，その instruction が格納される symbolic location，3 文字からなる operation code，相対番地表現もゆるす symbolic address (例えば，LŌC+3)，symbolic tag (index modify の指定をする)，absolute decrement などからなり，これは 4 word にわたって格納される。これらは section 6 までもちこされる。その順序は section 1 で statement が現われる毎につけられる symbolic location (internal statement number) によって保存されている。一方，table には compiled instruction に入り切らない情報が貯えられる。このために source program は section 1 で 1 回 scan されるだけでよいのである。

この様に compiler の基本的構成は簡単であるが，それがもっている複雑さの大部分は人間が組んだ program に比肩する様な能率のよい object program をつくりだそうということから生じている。個々の section の動作にそうした複雑さのいくつかがみられるが，system 全体としては，能率のよい program を生みだすのに必要な分析がなされるまで compilation を先にのばそうとすることから，しばしば厄介な compiled instruction と table の相互作用が生ずることになっている。

4. ALGŌL 小史

ごく最近まで大規模な computer といえば合衆国に独占されており，中規模以下の computer が世界的に普及していた。Europe の computer 学会である GAMM (Gesellschaft für Angewante Mathematik und Mechanik) の member の間に 1957 年に型の異なる種々の computer に対して algebraic compiler を設計しようという機運がもり上ってきた。その様な共通な computer independent でかつ problem oriented な言語が存在するということは明らかに利点である。そこで，言語が真に国際的であるためには合衆国からの参加が必要であることは当然であるので，GAMM の会長は ACM の会長である John Carr 博士に，ACM と GAMM の代表が computing procedure を記述しうる国際語を定めるために会合をもつことを提案した。当時，ACM は computer に関する種々の問題の討議の場所にはなっていたけれど，実際に言語設計を行なったことはなかった。GAMM の提案に対して Carr 博士は Perlis 博

士を programming language 委員会の委員長に指名した。この委員会は GAMM との共同の会合に提出される国際語に関する合衆国案の作成を始めた。委員会は主要な computer maker, いくつかの大学及び compiler に関して業績のある研究所の代表から構成されていた。これらの member の中で IBM の John Backus は最も活動的であった。彼はその地位からして言語設計の問題にかかり切ることのできたただ1人の member であったと思われる。合衆国案の大部分は彼の仕事に基礎をおいている。

ACM の委員会の小委員会は言語の種々の部分を検討して委員会に報告したが，それに対して賛否両論がたたかわされた。しかし，大綱に関しては意見の一致をみた。言語はできる限り数学の記法に近い形をとることにきめられたが，実際の implementation の困難さを考えれば妥協せざるをえない点もあったのである。こうした委員会の会合の1つに，GAMM の提案の推進者の1人である Bauer 博士が Europe 案の報告を提出した。それによると，begin, end, for, do などの英語の key word が一般的基準として提案されていた。これは ACM の委員会では考えてもいなかったことであるが，まず妥当なものと思われた。実際，用語のいくつかは異っていたけれど GAMM の提案と ACM の委員会の考えているものには基本的な不一致はそれほどなかったのである。しかし，ACM の委員会内部の基本線についての意見の一致を別にすれば，言語をどの範囲まで考えるかとか，より進んだ概念をどこまで言語に入れるかなどという点についてはなかなか一致をみなかった。記号列の manipulation, vector と matrix の直接の取扱い，倍長演算，問題の segmentation，記憶装置のわりあてと分割などに対しては議論がつづいた。そこで，ACM は小委員会を2つつくることにしたのである。1つは広い一致がえられると期待される言語を specify する委員会で，1つは将来のことを考えもっと進んだ考えを反映させた言語を specify する委員会である。GAMM との共同の会合には Perlis 博士が派遣された。この会合は1958年の春に Zürich で開かれ，その結果は Preliminary Report on International Algebraic Language として結実した。これは簡単に ALGŌL 58 という名前でよばれている。

ところで，1958年といえば，すでに述べた様に，FŌRTRAN II もできて FŌRTRAN の基礎が確立しその普及が緒についた頃であるから，なぜ ACM の委員会は国際語として FŌRTRAN をさらに拡張した言語ないし FŌRTRAN と両立しうる言語を提案しなかったのかという疑問がおこるかもしれない。それには種々の理由があるが，まず，もっとも明らかなのは，FŌRTRAN 語自体の制限性から由来するということで

ある。FŌRTRAN 語は完全に computer independent な言語ではなく，それは **704 の** compiler 言語として設計されたものである。例えば，DŌ statement や IF statement にみられる様に，704 の hardware の制約を反映した statement が存在する。また，前述の様に object program の optimization のために statement のかき方を相当簡略化もしているのである。さらに，もっと重大な理由は，ACM の委員会が ALGŌL ができつつあった 1957 年から 58 年の間の大規模 computer の分野における IBM の一頭地をぬいた地位に FŌRTRAN が関係していたことを無視したということである。現在以上に IBM には競争相手がいなかった。data 処理分野で Univac II が 705 と競争する様になったのはかなり遅れてからであったし，RCA の Bizmac, Honeywell の Datamatic 1000 も到底大刀打ちできなかった。科学計算の分野では Univac の 1103/1103a/1105 の series は IBM 701/704/709 に勝ると感じている人もいたが，Univac の納期の遅れと service の悪さは 704 に軍配をあげさせた。1103 a に対して maker が開発した最初の algebraic compiler は Unicode で，これはなかなか興味ある特徴をもっていたが，1960 年にいたるまで完成しなかったので，computer の方が先に陳腐化してしまったのである。ACM の幹部の中には FŌRTRAN はこの様な IBM 帝国を象徴しており，FŌRTRAN を国際語の基準として採用するならば，科学計算の大規模 computer の分野における IBM の独占体制をますます固めることになると感じていたものもいた。

ALGŌL 58 が発表された 1958 年はまた transistor 化された大規模 high speed computer が出現した年で，philco 2000 などが市場にでるという様な予告が行なわれていた。こうした状況の下で algebraic compiler の必要性は高まってきた。しかし，FŌRTRAN 以外のものは考えられなかったのである。例えば，2000 の最初の契約では 704 FŌRTRAN の source deck が何らの変更もなしにかかる compiler を備えていることが要求された。この様な事情は Honeywell, Control Data, Bendix でも同じであった。ACM の委員会の態度にかかわらず，FŌRTRAN は標準的な科学計算語になっていったのである。こうして皮肉なことに FŌRTRAN が standard になったために，科学計算の computer の分野の競争状態は妨げられるよりも，むしろ助長される様になったといえるのである。

さて，ALGŌL の普及について述べることにしよう。1958 年から 59 年にかけて数多くの新しい compiler が出現した。従って，新しい言語の実験の機は熟していたのであ

る。どんな algebraic language にも共通の要素が多数あったので，これらの年に新しい言語を開発したものは誰でもそれを ALGŌL か又は ALGŌL の方言とよんだ。こうしたこころみは多様な形でそれらの言語を普及することになった。Burroughs の鋭敏な若い programmer は Burroughs の新しい computer である 220 に対して非常に早い one pass の compiler をつくったが，これは Balgol としてしられている。ALGŌ という compiler が Bendix G 15 に対してつくられた。1958年6月，California の Santa Monica にある System Development Corporation は procedure-oriented programming language の研究に着手した。そして，同じ年に公表された ALGŌL 58 と目的において類似する予備的結果がえられた。これは基準化と便宜のために，ALGŌL の記法の大部分を採用することになった。1959年に完成したこの言語は CLIP (Compiler Language for Information Processing) という名前がつけられた。CLIP の実用性が確かめられてから，同様な procedure-oriented language が，Corporation が開発しつつあった軍用の command and control system の1つに対してつくられることになった。この言語は JŌVIAL (Jules Schwartz's own Version of the International Algebraic Languages) とよばれ，CLIP と同様に ALGŌL に準拠していた。そして，SAGE Air Defense System の program の開発からえられた経験によって必要であることがわかったいくつかの工夫を採用して，JŌVIAL を computer を使用する command and control system に適合する様につくりあげた。その後，最初につくられた language が成功したことと，その広い潜在的可能性が次第にみとめられる様になったことにより，JŌVIAL を関連した procedure-oriented programming language の基準とすることにきまった。JŌVIAL はかなり computer independent な programming language で，IBM 7090, CDC 1604, philco 2000, Burroughs D 825, SAGE AN/FSQ-7, AN/FSQ-31 に対してつくられた JŌVIAL compiler は1964年当時でも稼動していた。これらは monitor の制御の下で動く compiler としても，computer の中に常駐して単に compilation を行なうにすぎない compiler としても動く様になっている。こうした融通性は1つには JŌVIAL compiler が computer independent な JŌVIAL 自身でかかれているということと，もう1つは JŌVIAL が autonomous modules からつくられていることに由来する。JŌVIAL compiler の動作を簡単に述べると次の様になる。compiler は2つの部分からなり，おのおのは別々の transformation を行なう様になっている。第1の部分は

data の記述を整理し JŌVIAL の source program の statement で記述された計算を行なう基本的動作の順序を定める。この最初の transformation は JŌVIAL の中の Generator が担当するが，これは形式も機能も全く機械から独立に行なわれる。Generator は output としてやはり機械から独立な IL (Intermediate Language) をつくりだす。これは制限されてはいるが Universal Computer Oriented Language の 1 種である。IL は Translator による第 2 の transformation によって machine -oriented assembly language に翻訳される。Translator の JŌVIAL の中の形は machine independent であるが，機能は machine dependent になっている。どの computer に対する JŌVIAL compiler も固有な Translator をもつが，machine independent な Generator と IL はすべての compiler に共通である。これは，放置していれば方言が次第に生まれてくる傾向に抗して，JŌVIAL の文法を統一しておくことを可能にした。さらに，Generator が共通ということは新しく compiler をつくる苦労をかなり軽減する。例えば次の様な条件の下で新しく JŌVIAL compiler を製作するのに 6 人で 9 カ月を要した。

1. Translator は JŌVIAL でかかれる。
2. JŌVIAL compiler が利用でき，しかも，それがかかる computer がある。
3. compiler を開発している computer が作業の後半で利用できる。
4. 少くても半数の人間が熟練した JŌVIAL compiler writer であり，残りは compiler を開発している computer に関して熟練した programmer である。

ところで，この様な条件の下でどの様な手続で新しい compiler がつくられるのだろうか。それは次の様にして行なわれる。いま，A, B 2 つの computer があって，A にはすでに JŌVIAL compiler ができており，B に対して新しい JŌVIAL compiler をつくろうとしているものとする。まず，A の JŌVIAL compiler の Translator を A の machine code ではなく，B の machine code をつくりだす様に改めた compiler を JŌVIAL でかく。これを A の JŌVIAL compiler と A を使って compile すると，A の machine code で表現された JŌVIAL から B の machine code を generate する compiler がえられる。これを A に入れてもう 1 度うえの translator を改めた JŌVIAL 表現の compiler を Aで compiler すると，JŌVIAL から B の machine code を generate する compiler でかつ B の machine code で表現されたものがえられることになる。これが B の JŌVIAL compiler であることはいうまでもないで

あろう。

San Diego の Naval Electronic Laboratory は Sperry Rand の新しい computer Countess を設置した。この研究所は特に compiler に自分の言語で自分自身を表現させて，ひとつの computer の compiler から別の computer の compiler を容易につくりだすということに重きをおいて考えると共に，また，必要があれば object code の running time よりも compiling time が短いことに重点をおいた。1959年5月にでき上った compiler は NELIAC とよばれ，これも ALGŌL の方言のひとつである。NELIAC compiler も JŌVIAL と同様に多くの computer に対してつくられている。Michigan 大学は FŌRTRAN の compilation の遅さに困っていたので（ことに学生の短い program に対してその様にいえた），MAD (Michigan Algebraic Decoder) を1960年2月に開発した。MAD は元来704用に開発されたが，1961年3月に7090用にも改変された。これも ALGŌL 58 に基いている。

ACM-GAMM の委員会はその後も作業をつづけ，その結果を1960年に Report on the Algorithmic Language ALGŌL 60 として発表した。これは簡単には ALGŌL 60 といわれる。ALGŌL 60 は予想に反してかなり普及した。ALGŌL 60 は必ずしも意見の一致をみた部分に限られていない。recursive subroutine, dynamic storage allocation, block structure, own variable and array の概念が導入されているが，これらの処理はかなり複雑で，いろいろと議論のある所である。ALGŌL は programming language の syntax の厳密な定義の仕方において一つの先例をつくっている。すなわち，Backus Notation がそれである。ALGŌL は FŌRTRAN と比べてかなり制限のゆるい言語である。例えば，loop, conditional statement などを考えてみればこのことはあきらかであろう。しかし，ALGŌL の最も大きい欠点は入出力に関しての規定がないことであった。入出力に関する service は compiler が user に提供すべき最も重要なものであるから，これが欠けていたのでは general purpose algebraic compiler language ということはできない。そのため，1964年に IFIP 委員会と ACM 委員会からそれぞれ ALGŌL の入出力に関する提案が出されている。これより先，ALGŌL 60 はいくつかの誤りとあいまいな点を含んでいたので，1962年4月の Rome の Conference でそれらの点が改められて，Revised Report of Algorithmic Language ALGŌL 60 という改訂版が出されている。これは revised ALGŌL 60 といわれている。1964年5月の ISŌ/TC-97 の New York の会議では revised ALGŌL 60 とそ

の subset (1964年の IFIP 総会で承認されたもの) を国際標準言語とすることがきめられた。そのさい，入出力の規定としては ACM のものと IFIP のものを ALGŌL 文法に対する補足として採用することになった。なお，ACM の I/Ō 手続は IFIP のものを包含する様に改訂されている。revised ALGŌL 60 も必ずしも完全なものでなく，まだいくつかの問題点が残っていることが指摘されている。IFIP の下部機構の Working Group 21 という委員会がその後 ALGŌL の種々の問題 (例えば，revised report の前文にそのいくつかがあげられている。

1. function の副作用.
2. call by name の概念.
3. own : static or dynamic.
4. for statement : static or dynamic.
5. specification と declaration の矛盾.)

を検討していたが，そこでもっと進歩した ALGŌL が必要であるという意見がだされ，Holland の Wijngaarden はどこまでも範囲をひろげることができる様な ALGŌL X を提案した。この案は委員会の審議過程で種々の議論をよび更に新しい機能がつけ加えられて非常に複雑なものになった。そして1968年12月 Wijngaarden 案が票決に附せられた結果 IFIP の公式資料として公表されることになった。ALGŌL compiler は数多くつくられているが，その普及は Europe の方が合衆国よりも優勢の様である。合衆国では依然として FŌRTRAN が根強い。

5. Data-Processing Compiler について

Data-Processing compiler として最初のものは Univac I, II に対して開発された1956年の FLŌW-MATIC であろう。これは英語を言語として使い，procedure とは独立に data を記述することができ，機械の register とは独立な field-name を使うことができた。FLŌW-MATIC はのちに Univac 1105, IBM 705 に対してもつくられている。このほか初期の Data-Processing compiler として注目に値するものは Washington の Hanford で 702-705 のために開発された Hanford generator がある。この system は generated record handling, sorting, report writing などの機能をもっていた。これは SHARE/IBM がつくった 709 の 9-PAC などにうけつがれている。また，704用の SURGE data processing system も SHARE が1958年から

59年にかけて開発したものである。これらとは別に1956年から IBM は Commercial Translator の開発にとりかかっていたが，その進展は遅々たるものであった。tentative manual が1959年春，1959年夏，1960年夏に出された。そして，7070，7080，7090などに対して implementation が進められていたが，1961年にそのうちのいくつかが使用できる状態になったのにすぎなかったのである。

さて，1959年5月に Department of Defense の Charles Phillips は Pentagon で会議を開いた。この会議の目的は computer を使って data-processing をするさいの共通言語をつくる必要性と可能性を検討するためのものであった。疑いもなく，data processing のための装置の最大の user は合衆国政府である。合衆国政府は法律上特定の maker の computer のみを使用するわけには行かないので，各機種の間に共通性がないことに非常な不便を感じていたのである。この会議には user, government installation, computer maker, その他の代表が集った。彼らは共通語をつくるという project を開始することには全く異論がなかった。これが CŌDASYL (Committee on Data System Language) の始まりである。CŌDASYL の中には短期，中期，長期の3つの委員会が設けられた。そして，全体の調整は Executive Committee がとることになった。短期委員会はその時までの経験を基礎にできるだけ早く共通語をつくるべきであるという立場を代表していた。これに反して中期委員会は data-processing というものがどの様なものであるかをよく検討してから共通語を考えるべきであるという立場を代表するものであった。短期委員会は，maker から Burroughs, IBM, Minneapolis-Honeywell, RAC, Sperry-Rand, Sylvania Electric Product の代表，政府関係から Air Material Command, U.S.A.F., David Taylor Model Basin, Dept. of Navy の代表からなり，議長は National Bureau of Standards からでた。この委員会は1959年6月23日に最初の会合を開いた。そして，2つの working group がつくられた。1つは Data Description に対してであり，1つは Procedual Statement に対してである。これらの group は会合を重ね，委員会に提出する検討資料を作成した。短期委員会は1959年8月18日から21日までと，8月24日から25日までと2回開かれ，Charles phillips が議長である Executive Committee への報告書を準備した。この報告書はあまり満足のいくものではなかったので，短期委員会は1959年12月1日までにもっと完全な形のものに仕上げる様に要請された。そして，短期委員会は implementation にも立ち会うために，1959年12月1日以降も存続することに9月4

日付けをもってきめられた。委員会は9月18日から10月21日まで何回か会合を開き，問題点の検討を重ねて，COBOL (Common Business Oriented Language) という名前を採用した。以後この委員会は COBOL 委員会といわれる様になった。10月26日から11月7日まで小委員会は COBOL system をつくり上げる作業を行なったが，その結果は11月16日から20日の間に COBOL 委員会（短期委員会）によって検討され承認された。次の2週間に小委員会は最終稿を仕上げた。そして，1959年12月17日に Executive Committee にそれを提出した。それに伴って，Sylvania, RCA, Univac は MOBIDIC, 501, Univac II などに対して implementation を開始した。

ところで，1959年春から FACT の開発が始っていた。これは，1958年に Datamatic 1000 の失敗から一時 computer 関係の事業を中止した Honeywell が Honeywell 800 という比較的安価な computer で市場に喰い込もうとして Computer Science Corporation と契約してつくりつつあった Data-Processing Compiler であった。FACT の Commercial Translator に対する関係はいくらか Haydn に対する Mozart に似ているといわれている。FACT は FLOW-MATIC, SURGE, GE Hanford, 9PAC, Commercial Translator などの先行者の影響をうけており，とくに Commercial Translator の影響は著しい。しかし，FACT の設計者はいち早くそれらを脱して新しい technique をつくり出し，input-editing, sorting, report writing を言語の中に結合した。ところで，1959年10月に使える様になったこれらの開発の結果は，1959年夏の Commercial Translator の manual と1960年6月のそれとを比較すればわかる様に，Commercial Translator の一層の開発に少なからぬ影響を与えている。この様に FACT と Commercial Translator の関係は循環的である。FACT は1961年12月にでき上った。全体は3-address の命令を250,000個もち，4K の記憶装置と4本の MT で動く様につくられている。

さて，COBOL の最初の specification が明らかになってから，有名な委員会の闘争がおこった。会合を重ねていた中期委員会は初期の COBOL specification は不満足なものだという評価を下したのである。そして上述の Honeywell の FACT の最初の specification の方が COBOL よりも Common Business Oriented Language としてより適当であるという見解を示した。しかし，COBOL 委員会としては COBOL をすててしまう気持は毛頭なかった。こうした事態も，結局は Executive Committee が1960年1月に COBOL を承認することによって，COBOL が COBOL になるとい

うことで落着をみることになった。COBOL は1960年4月までに更に編集をうけ，小さな誤りを正されて，6月に Government Printing Office から出版された。これが COBOL 60 である。1960年2月に Exective Committee は2つの group（1つは user のみからなる10名の group，1つは maker のみからなる9名—短期委員会から6名，CDC, GE, NCR から各1名—の group）が COBOL の current な maintenance を行なう様にきめた。5月までに両方の group は活動を開始した。以後，今日まで，FACT, Commercial Translator の影響をうけながら，COBOL 61, COBOL 61 extended, COBOL 63, COBOL 65, COBOL JOD 1 と改訂版がつづいて出されている。

最初のうちは，いくつかの maker は COBOL の implementation に対して余り熱心ではなかった。とくに上述の様に IBM と Honeywell は Commercial Translator と FACT を開発しつつあった。これらはもし COBOL が目的の通り一般化すれば当然陳腐化する運命にあったのである。ところで，1960年に合衆国政府は，今後は COBOL が使えるか又は使える見込のある computer 以外は使用しないという声明を出した。これによって，COBOL に対する不熱心な態度は消えうせた。以来，多くの COBOL compiler がかかれている。例えば，1962年12月に開発された7090用の COBOL などはその代表的なものである。

COBOL を使うことはかなり厄介である。例えば，SYNCRONIZED, COMPUTATIONAL などという長い語を頻繁にかかなければならない。しかし，多くの programmer は，そうしなければなされえない重要な機能を compiler が果すので不便さを我慢しているのである。