Programming System の発展* (1)

戸島　潤（とじま　ひろし）

目　次

最初 computer には真空管が使われていたが，1959年までにはじめて transister 化された computer が市場に出てからは，真空管を使った computer は急速に姿を消した。1950年代の初期には研究所の中の装置にすぎなかった magnetic core memory も，1959年頃には価格も安くなっており，次第に，磁気 drum などを主記憶装置の地位から追放しつつあった。1959年以降の computer の発展は hardware, software とも急速でかつ目ざましいので，それらのいわば無統制な発展の跡をたどり，適切な評

原稿受領 1969年12月15日

* この論文は下記の諸論文を参照することによってかかれた。なかでも特に Rosen の論文に依存している。

Backus, J.W. et al., "The FŌRTRAN Automatic Coding System," *Proceedings of the Western Joint Computer Conference,* Vol.11, 1957, pp.188–198.

Backus, J.W. et al., *Revised Report on the Algorithmic Language ALGOL 60,* 1962.

Backus, J.W. and W.P.Heising, "FŌRTRAN," *IEEE Trans. EC–13,* 4(August 1964), pp.382–385.

Bromberg, H., "Survey of Programming Languages and Processors," *Comm. of ACM,* Vol.6, No.3(March 1963), pp.93–99.

Carr, J.W., "Programming and Codings," in Grabbe, E.M., S.Ramo and D.E.Wooldridge(ed. by), *Handbook of Automation, Computation and Control,* Vol.2, Wiley, Los Angels, 1959, pp.2–01ff.

Clippinger, R.F., "FACT–A Business Compiler: Description and Comparison with CŌBŌL and Commercial Translator," in Goodman, R(ed.by), *Annual Review in Automatic Programming 2,* Pergamon Press, New York, 1961, pp.231–292.

Halstead, M.H., "Machine–Independence and Third–Generation Computers," *Proceedings of Fall Joint Computer Conference, AFIPS,* Vol.31, Baltimore, pp.587–592.

Knuth, D.E., "A History of Writing Compilers," *Computers and Automation,* Vol.11, No.12 (December 1962), pp.8–18.

Remington Rand, The A–2 Compiler System, *Univac Form RRU–4,* 1955.

Rosen, S., "Programming Systems and Languages — A Historical Survey," *Proceedings of the Eastern Joint Computer Conference, AFIPS,* Vol.25, London, 1964, pp.1–15.

Sammet, J.E., "A Detailed Description of CŌBŌL," in Goodman, R(ed. by), *Annual Review in Automatic Programming 2,* Pergamon Press, New York, 1961, pp.197–230.

Shaw, C.J., "JŌVIAL–A Programming Language for Real-line Command Systems," in Goodman, R(ed. by), *Annual Review in Automatic Programming 3,* Pergamon Press, New York, 1963, pp.53–119.

岩村　聯「ALGŌL のゆくえ？」*bit* Vol.1, No.1 (1969年3月), pp.74–76.

価を下すことは非常にむずかしい仕事であるが，以下では，それを承知の上で，programming system の発展の概略をふりかえってみることにしよう。

1. 初期の programming system

最初の programming system は subroutine を使用するものであった。IBM と Harvard 大学の Howard Aiken 教授が 1937 年から 1944 年にかけて開発した Harvard Mark I という会計機の機械的部品（例えば，switch, relay, 歯車，ばねなど）を使用した電気機械的 (electromechanical) computer がまさにこれであった。その後，1949 年には Mark II が，1952 年には Mark IV が完成している。1949 年の Cambridge 大学の EDSAC (Electronic-Delay Storage Automatic Calculator) は稼動している stored program 方式の computer では恐らく最初のものであったが，これも subroutine library が programming の基礎であった。

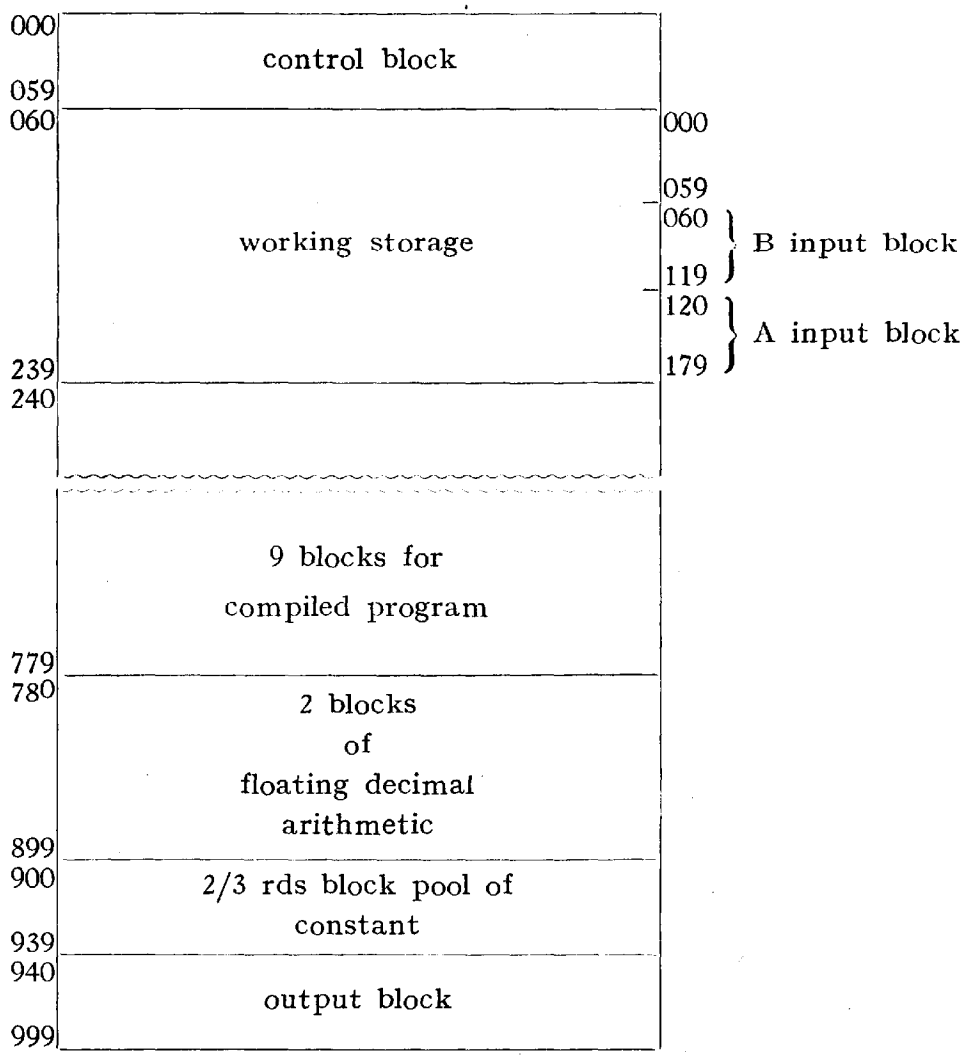
1947 年から 49 年の間に発展した IBM の Card-Programmed Calculator は速くはないが融通性に富んだ装置であった。これは control board に配線をすることによってその動作を制御する。program として 1 組の card に指令と数値を punch して，*CPC* に与えると機械は card を 1 枚づつよみとって，その指令に従って動作をし，最終結果を card に punch して出力する。従って，問題をうまくとくための配線にはかなりの巧妙さが必要であった。また，多くの問題を扱いうるような多目的制御盤 (general purpose board) をつくることも同様であった。*CPC* に対しては，このような一組の多目的制御盤が programming 言語ということになり，多数の言語がつくられて使用された。そして，結局は，科学計算のためには，*CPC* を 3-address で浮動番地演算ができ，平方根，sin, 巾乗などを求める subroutine を組み込んでいる機械として使うことに落着した。この *CPC* に関する経験は来たるべき stored-program computer の programming system に多くの影響を与えることになったのである。

初期の自動 programming の中でもっとも重要なものの 1 つは MIT の Whirlwind project である。1947 年から 51 年の間につくられた Whirlwind は速いけれど簡単な computer であった。この computer は 1 語が 16 bit で，限られた命令と，わずかの高速度記憶装置しかもっていなかった。この機械の machine language を使って program を組むことは非常に困難であったので，programming 言語を開発しようという

強い動機が生まれた。1953年6月の Summer Session Computer は Whirlwind の初期の interpretive system の1つで，これは夏期 course の学生に Whirlwind を使わせるように計画されたものである。このような初期の計画は Whirlwind に対して1952年12月に Comprehensive System という programming system を導くに至った。

商品としての大規模な電子計算機として最初のものは，1951年に市販された Univac I である。このような commercial computer に関する最初の自動 programming group として，Univac をつくった Eckert-Mauchly Computer Corp. (これは現在は Sperry Rand の Univac 部門である。) の Grace Hopper 博士によって始められた group がある。Univac I は10進，alphanumeric で mnemonic instruction をもっていたので computer を machine code で使用することが容易にできた。また，1語の長さは12字で，これは固定小数点の scaling を容易にした。高速度記憶装置は1000語 (12000字) で，平均 access time は 200μs である。ところで，このような computer に対しては，遅いうえに，しかも，あまりうまくつくられていない assembly system や compiler の有利さは必ずしもよく分らないのが普通である。しかし，たびたびの挫折にもかかわらず，Hopper 博士は programming は problem-oriented language でなされるべきであるし，また，なされるであろうという確信をもっていたのである。Hopper 博士の group は Univac に関する一連の language のすべての開発に関与している。それらの中で1953年8月の A2 compiler system は恐らく最もよく使われたものであろう。以下，この system についてやや詳細にのべてみよう。

Univac の入力装置は Unityper とよばれる typewriter で，これは keystroke によって直接 *MT* 上に記録をつくることができるようになっている。他方，出力装置としては high-speed printer があって，これは *MT* 上の記録を自動的に変換して印刷することができるようになっている。このように Univac では computer の入出力はすべて *MT* を通じて行なわれる。そのさい，入出力は60語を一単位として行なわれる。これを block とよんでいる。従って，input tape の記録が 1 block 分にみたない時は 0 を padding しなければならない。program が実行される時の記憶装置の layout は次のようになっている。



control block には system の control instruction (例えば, *MT* から compiled program をよみこめ, という命令など) が入る。これは programer には直接関係しない。working storage の最初の block は中間結果とか計算に必要な定数を格納するために使われる。input block には data が *MT* からよみこまれてくる。block が2つあるから multi-reel からの input が可能である。通常は *A* から使われて行く。compiled program の block に program が入りきらない時は自動的に segmentation が行われる。floating decimal arithmetic の block には, 四則に関する浮動十進演算を行なうのに必要な instruction が格納されている。pool of constant にはよく使われる constant が予め入っている。output block には最終結果が入れられる。この block が一杯になれば自動的に *MT* にかき出される。

ところで, この system で, 例えば, working storage に data をよみこむには, どのように命令をかいたらよいであろうか。それらのために system はいくつかの

pseudo-instruction を用意している。この場合には次の pseudo-iustruction を使う。

1	2	3	4	5	6	7	8	9	10	11	12
G	M	I	$\bar{0}$	(t)	$\binom{A}{B}$	$\bar{0}$	(s)			(m)	

t ..... tape handling unit number を指定する。

$\frac{A}{B}$..... どの input block に data をよみこむかを指定する。

s ..... data が何語 working storage に transfer されるかを指定する。

m ..... data が working storage の何番地（相対番地）から格納されるかを指定する。

例えば

G	M	I	$\bar{0}$	2	A	$\bar{0}$	1	2	0	0	0

のように指定すれば次のようないみになる。tape handler No. 2 から data を input block A に 1 block 分よみこんで，先頭から 12 語を working storage の相対番地 000 から始まる番地に transfer せよ。compiler はこの pseudo-instruction を解釈して subroutine library の中から Input Generator をよび出し，pseudo-instruction をそれにわたす。generator は pseudo-instruction に指定されている parameter を使って 84 個の machine language の instruction (subroutine) を generate する。これは compiled program の 1 部分となる。次に，四則演算の pseudo-instruction の例として加算をあげよう。

A	M	$\bar{0}$	(	A	)	(	B	)	(	C	)

A ..... 被演算数が格納されている working storage の相対番地。

B ..... 演算数が格納されている working storage の相対番地。

C ..... 演算結果を格納する working storage の相対番地。

(例)

A	M	$\bar{0}$	0	0	8	0	0	0	0	1	8

(008) + (000) ⟶ 018

このように，A2 compiler system には4つの基本的な要素がある。それらは

1. subroutine
2. library
3. pseudo-instruction
4. compiler

である。subroutine は数学的または論理的演算を行なうのに必要な一組の instruction からなっている。例えば，四則演算とか sin の計算などである。library は標準的な subroutine を集めて，ある順序に並べたもので，A2 compiler system では *MT* に alphabet 順にかきこまれている。pseudo-instruction は compiler system によってのみ理解される instruction である。compiler は preudo-instruction を machine language に translate する master routine である。translation は program の execution に先立って行なわれる。

ところで，A2 compiler system における compilation は次のように行なわれる。まず，compiling には tape handler が7台使われる。そのおのおのの用途は次に示す通りである。

No. 1　A2 compiler instruction

No. 2～5　working tape

No. 6　A2 library of subroutines

No. 7　pseudo-coded instructions

compilation は4つの phase に分かれている。

Phase I—Translation

compiler の中のひとつの routine である translator は各 pseudo-instruction を調べる。もし pseudo-instruction が何らかの operation を行なうものであるならば，まず，catalog を調べてその operation が library の中にあることをたしかめる。そして，pseudo-instruction は compiler が容易に information をつかみうるように切り離され展開された形にされる。この展開された pseudo-code は Tape 2 にかきこまれる。この翻訳が終了すると，END ŌF TRANSLATIC̄N と printer に印字が行なわれる。

Phase II—Sweep I

この phase では展開された pseudo-code が次々に調べられて何らかの generator が必要かどうか，またはそれは machine 語（これを own-code という）でかかれた subroutine かどうかが判定される。もし generator が必要なら，library が探され，それが記憶装置によみこまれる。generator は pseudo-instruction の parameter に従って subroutine を generate する。つくり出された subroutine は block 単位で Tape 3 にかきこまれ，control は再び compiler にもどる。ついで compiler はのこりの pseudo-code について同じことをくりかえす。もし own-code subroutine に出会ったなら，compiler は Tape 3 にやはり block 単位でかきこむ。結局，この phase では compiler は Tape 3 に sub-libray をつくることになる。

Sweep I ではもう 1 つの operation が行なわれる。これは Segmentation といわれる。program の実行時には program のために 9 blocks (540 語) しか許されていないから，ここで compiler はどれだけの pseudo-code を各 segment にわけるかをきめて，segmenting sentinel を pseudo-code につける。この segment にわけられた pseudo-code は Tape 4 にかきこまれる。この phase が終ると printer に END S. 1 と印字される。

Phase III—Sweep 2

展開され segment に分けられた pseudo-code の 2 回目の sweep は compiler に対して control pseudo-instruction がもっとよくいみをもつようにするために行なわれる。Tape 4 から pseudo-code が computer の中によみこまれると computer は jump 命令を探す。それがみつかれば，どの operation number に control が移るかをみて，Tape 5 の record を探して，その operation number をみつける。record には operation number の次に最終的な compiled program の subroutine の冒頭番地がかきこまれてあるから，この番地がぬき出されて，いま問題にしている pseudo-code にそれをかきこむ。このように修正が施された control instruction を含む pseudo-code は Tape 2 にかきこまれる。この phase が終れば printer に END S. 2 と印字される。

Phase IV—Main Compile

ここで，Tape は次のようになっている。

No. 1 A2 compiler instructions

No. 2 expanded, segmented and control substituted pseudo-code

No. 3 generated sub-library (generated subroutines and own-code)

No. 4 working tape

No. 5 record (not used in main compile)

No. 6 library of subroutines

No. 7 original pseudo-code

Tape 2 にかかれている pseudo-code は今や compiler に対して完全に問題の解法を記述している。この pseudo-code が computer の中によみこまれると，compiler はどんな operation が行なわれるかをみる。もしそれが generated subroutine や own-code subroutine による operation であれば，該当するものを Tape 3 からもってくる。もしその他のものであれば，Tape 6 の library を探す。compiler は一時にひとつの subroutine を computer によみこむ。そして subroutine の中の番地部を状況に応じて適当に修正する。compiler は各 subroutine についてこのことをくりかえし行ないながら，それら全部を link して final program をつくりあげ Tape 4 にかき出す。compilation の終りには printer に END CŌMPILE と印字される。

以上が A2 compiler system の働きの大要である。これで分るように，preudo-instruction は現在の computer の machine 語に非常に近いものである。すなわち，現在の computer は Univac I が software によって処理していたことを hardware によって処理するようになっているといえるのである。さらに，この system の基礎は subroutine にあることはあきらかであって，まだ，ここには数式などを通常の数学的表現のまま与えて，それを machine 語に翻訳するという考えは explicit にはあらわれていない。しかし，1つの pseudo-instruction に一群の machine 語を対応させるという programming language の重要な思想は明瞭に顔を出しており，この点で A2 compiler system は compiler の歴史を語る上でかかすことができないと考えられる。

Math-Matic といわれる 1956年6月の algebraic translator AT3 も Univac の automatic programming system としてつくられ，のちの ALGŌL その他の compiler

に影響を与えたが，残念なことに AT3 ができ上る前に Univac は scientific computer としてはすでにすたれたものになってしまったのである。同時に，Flow-Matic (1956 年 12 月) という B0 compiler は COBOL に大きな影響を与えている。また，1951 年には Univac programming group によって最初の sort generator がつくられているし，また，彼らは symbolic form でかかれた数式の微分演算を行なう symbol manipulation prograrm もつくっているが，これは恐らくこの種の program としては最初のものであろう。Univac のもうひとつの group は今日でいう computer-oriented compiler の分野に関心をもっていた。Anatol Holt と William Turanski は 1957 年 1 月に GP (Generalized Programming) という compiler を開発した。これは非常に一般的な subroutine library の存在を前提にする system である。すべての program はあたかもそれらが library program であるかのようにかかれる。そして，program は compile time に特定の library program やその 1 部分がえらばれたり，修正されたりして assemble される。すなわち，programer のかいた program は parameter や specification であって，それに従って一般的な library program は特定の問題向きに改められるのである。この system は Univac II に対して開発された GPX で一層拡張されている。この group は，また，program の segmenting やそれに関連した storage allocation の問題などに関心をもった初期の group の 1 つである。彼らの最も重要な貢献は恐らく彼らの次のような考えにあると思われる。すなわち，programer が使う computer は単なる hardware ではなく，program の translation と実行に加えてあらゆる種類の service や library maintenance の機能を果たす programming system によって強化された hardware からなる機械である。

1952 年の IBM 701 は binary の大規模 computer として市場に出た最初のものである。701 は 1-address, binary, 固定小数点演算の computer であったが，Speed Code という programming language はこれを 3-address で 10 進浮動小数点演算を行ない，しかも index register をもつ computer として使うことを可能にした。701 に対する PACT という system は RAND, Lockheed などの computer user の PACT group が 1955 年 1 月につくった最初の system という点で 1 つの範例となっている (704 に対する PACT は 1957 年 1 月に完成している)。しかも，これらは system が開発されてみるとすでに computer が陳腐化していたという点においてものちの範

例となっているのである。PACT の idea は SHARE organization の指導の下に行なわれたのちの開発に多くの影響を与えた。

1953年頃から主記憶装置が磁気 drum の中型 computer が市場にあらわれ始め，1955年から56年にかけてこれは非常に重要な computer であった。1954年の IBM 650 (Drum 2–4K) はそのような computer として最も普及したものであった。650は machine 語で program を組むことが容易で，data 処理の分野ではほとんどそのようにして使われた。しかし，650は科学計算用として，のちに改良されたが，hardware による浮動小数点演算の装置を欠いていた。そこで，それを行なうような interpretive system が多数開発されたが，それらの中で1955年9月に Bell Telephone Laboratory でつくられた BELL がよく知られている。これは上でのべた *CPC* の制御盤から 701 の Speed Code につづいている思想をうけつぐもので，3-address floating point system でかつ automatic looping の機能をもち，さらに mathematical subroutine を内蔵していた。interpretive system はその後ほとんど使われないようになったが，最近では micro-programmed computer との関連で再び復活してきている。650は，また，drum の適当な場所に instruction をおくという方法によって program の optimization を可能にした。1955年11月の SŌAP (Symbolic Ōptimizer and Assembly Program) は symbolic assembly とこの自動的 optimization を結合したものである。SŌAP が開発された当時，symbolic assembly system が広くうけ入れられたかどうかは疑問のある所であるが，optimization という特徴は非常に有用なものである。

650の有力な競争相手は同じ1954年の Burroughs の Datatron 205 であった。これは4080語の drum memory をもっていたが，このうちの80語は高速度記憶が可能であるという点に特徴があった。最初はこの computer も hardware で浮動小数点演算ができないため，種々の interpretive system や assembly system が開発され，浮動小数点演算の便宜を与えた。しかし，のちに浮動小数点の hardware が追加されている。Datatron 205 のもうひとつの特徴は市販の computer の中で最初に index register を具え，subroutine の自動的な relocation を可能にした computer であるという所にある。他の computer では当時はこれらは programming system によって行なわれていた。

最初の Datatron computer の1つは Purdue 大学に設置された。ここで Al Perlis

博士に引いられた group によって 1957 年 7 月に Datatron に対して初期の algebraic compiler のひとつが開発された。彼らもまた programming は programming language でなされるべきであるという信念をもっていたのである。ところが，初期の Datatron は alphanumeric input がまったくできなかった。computer は Flexowriter に print される文字を 2 つの数字であらわすけれども，key を 1 回おすだけで同じ一組の数字をつくり出す装置は まったくなかったのである。 のちに そのような 装置が開発されるまで，Purdue compiler に対する入力は source language を人力で 2 つの数字の組にかきかえることによって準備された。 Perlis 博士が Carnegie 工科大学に 移ってからは同様な compiler が 1957 年 2 月に 650 に対してつくられた。これは IT (Internal Translator) と名付けられている。その後，IT はいろいろ改訂された形で，Case Institute (IBM 650, Datatron 220)，Ramo-Wooldridge (Univac 1103A, 1958 年 3 月) North Carolina 大学 (Univac 1107, 1959 年 9 月) Armour Research (Univac 1105, 1961 年 8 月) などでつくられている。

2. IT compiler

IT 語は広く普及した最初の universal programming language である。それゆえ，ここでは IT 語とその compiler にやや詳しくふれておこう。以下では Perlis と Smith によって Carnegie 工科大学で 650 のために開発され，のちに Michigan 大学の Digital Computation Unit でさらに強化された IT に関してのべる。

IT program は IBM 026 Key Punch にある Symbol を使ってかかれる。IT 語は machine 語からは独立でまったく machine 語の形にとらわれていない。Program は statement からなり，statement は文字と 演算記号の列からなる (compiler などによって多少違うが，記号の数は大体 100 位迄は許される)。IT は one pass と two pass の 2 通りに使えるようになっている。one pass として使えば直ちに machine 語が card につくり出される。これは 1 枚の card に 5 word (1 word は 10 進数 10 字) 分づつ punch されていて，記憶装置の どこにでも 格納することができるようになっている。two pass として使えば，SŌAP 語に直された program が作り出される。SŌAP 語の 1 命令は 3 字からなる mnemonic code と 1+1-address からなっている。2 つの address の 1 つは operation の作用番地で，他は次に実行すべき命令の所

在を示す番地である。

2. 1 character

使われる記号は

数字　0から9まで

文字　A から Z まで

特殊記号　( ) + − = · * / ,

などであるが，ある種の記号は翻訳の特定の場所で printer の制約によって英文字であらわされることがある。しかし，英単語の中に あらわれる場合を除いて，英文字は IT の中ではひとつのいみしかもたず，かつそのいみに限られる。

句切記号

記号	表示法
(	(or L
)	) or R
· (小数点)	· or J
← (代入)	= or Z
= (関係演算子)	U
>	V
≧	W
,	, or K
' ' (引用符)	Q
Type	T
Finish	F
Extention identifier	E

変数　I Y C

数字　0から9と B

以下では，k, l, m, n などの小文字は任意の正の integer をあらわす。

演算記号

記号	表示法
+	+ or S
−	− or M
×	* or X
/	/ or D
exp	P　　(例) $Y1\ P\ Y2$ は $Y1^{Y2}$ のいみ
|……| (絶対値)	A
(−…) (unary−)	(−…) or $LM\ldots\ldots R$

2. 2. 変数のかき方

浮動小数点変数

Y_n, YI_n, YII_n　　(例) $Y3$, $YI47$, $YII21$

C_n, CI_n, CII_n　　$C3$, $CI0$, $CII7$

Y, C のうしろについているのは subscript で，このように subscript に subscript をつけることも許される。Y も C も論理的な区別はないが，data によって区別してかく便宜を考えているのである。これらは computer 内部では浮動小数点で表示される。

固定小数点変数

I_n, II_n, III_n　　(例) $I8$, $II36$, $III2$

これらは整数値をとり，例えば index として使われる。

composite 変数

$Y(\ldots\ldots)$　　(例) $Y(I1+6)$

$I(\ldots\ldots)$　　$I(II3\times 19)$

$C(\ldots\ldots)$　　$C(I(I1+2))$

括弧でつつまれるものは必ず固定小数点であらわされる値をとる数式でなければならない。

Matrix（浮動小数点）変数

$YN(\ldots, \ldots)$

$CN(\ldots, \ldots)$

これは matrix の成分が行 wise にそれぞれ $Y0, Y1, \ldots, C0, C1, \ldots$ である matrix の一般的記法である。括弧の中の式は固定小数点で表示される値をもつ数式で，行と列の位置を指定する。row dimension と column dimension は $I1$ と $I2$ にそれぞれ指定する。行も列も0から出発し，各次元よりも1つ少ない数で終る。例えば

$Y0$　$Y1$　$Y2$

$Y3$　$Y4$　$Y5$

$Y6$　$Y7$　$Y8$

$Y9$　$Y10$　$Y11$

ならば，$YN(0, 0)$ は $Y0$，$YN(1, 2)$ は $Y5$ で，$I1$，$I2$ はそれぞれ3，4である。

2. 3. constant のかき方

固定小数点定数（整数）

$n_1n_2\ldots n_k$　$k \leqq 10$　　（例）1066, 129234556　　123. は間違い

浮動小数点定数

a)　$n_1n_2\ldots n_t.\ n_{t+1}\ldots n_k$　$k \leqq 8$　　（例）14.92, .11, 13.

b)　$n_1n_2\ldots n_kBm$　$k \leqq 8$

このいみは $n_1n_2\ldots n_k \times 10^m$ ということである。ここで $n_1\ldots n_k$ は整数でも浮動小数点変数でもよい。m は m_1m_2 かまたは $-m_1m_2$ にかぎられる。ただし，m_1m_2 は整数でなければならない。m が $-m_1m_2$ なら定数全体を括弧でつつむ。

（例）$(1066B-11)$, $(-727B-5)$ これはそれぞれ 1066×10^{-11}，-727×10^{-3} のいみである。

また，statement（後述）の中で使われる浮動小数点定数は上の形でなければならず，決して標準の浮動小数点の形をとってはならない。

(例) 14.92 は 14 J 92 とかく。

2. 4. operand のかき方

a) variable と constant は operand である。

b) $v1$, $v2$ が operand ならば, ($v1 \Delta v2$) も operand である。ここで, Δ は operator である。

c) subroutine (それ自身いくつかの operand の関数である) も operand である。これは "$nE, v1, v2, \ldots vj$" とかかれ, identification number が n (n 番目の extension) の subroutine で, $v1$, $v2$, ..., vj の関数であることを示す。$n \leqq 626$ である。例えば, sin subroutine の番号が 21 であれば, $\sin(Y1+Y2)$ は "$21E, Y1+Y2$" とかく。

d) $v1$, $v2$ が operand であれば, $v1 \Delta v2$ は数式である。ここで, Δ は operator である。compiler はある数式を operand として扱わないが, すべての operand は数式である。数式の norm とは記号の個数のことをいう。ただし, space は除く。programer は誤りを完全に避けるためにすべての statement に括弧をつけなければならない。

e) operand が変数か定数なら, その四則演算は変数か定数同志の演算でなければならない。

f) subroutine を除いて, operand が composite で, その 1 つが浮動小数点の値をとるなら, すべての operand は浮動小数点の値をとらねばならない。

g) subroutine の演算は extension number によって次のように定まっている。

$n < 500$ 浮動小数点, $n \geqq 500$ 固定小数点

2. 5. statement のかき方

各 statement に $k \leqq 626$ である statement number k をつけるが, 実行の順序はこの番号に左右されず, 並んでいる順番に処理される。

substitution statement

k: $v1 \leftarrow v2$

ここで，$v1$ は変数，$v2$ は数式，k は statement number である。これは $v1$ の値が $v2$ の値と等しくおかれるということをいみする。

(例) 7: $vI2 \leftarrow I1+(I3\times(Y3-Y4))$

unconditional linkage statement

k: Gn

k: GI ... In

k: $G(\ldots\ldots)$

k は statement number である。これは無条件 jump 命令である。(……) の中は fixed point expression をかく。どれを用いても同じである。

conditional linkage statement

k: Gn

k: GI ... In $IFv1\gamma v2$

k: $G(\ldots\ldots)$

ここで，$v1$ は operand，$v2$ は数式であり，γ は＝，＞，≧ のいずれかである。これは条件付 jump 命令である。

(例) 4: $GI3$ $IF(Y1+Y2)\geqq 9$

ここで，$(YI+Y2)$ と括弧をつけることが必要である。なければ，$Y2$ は ≧ の operand になる。

halt statement

k: H

input statement

k: READ

これによって停止記号をかくまで数字，英字の data をよみこむ。

output statement

k : $Tv1$ $Tv2$ $Tv3$ $Tv4$ $Tv5$

ここで，v は変数で，list には 5 つまでかける。variable が composite なら subscript を特定して指定する。$YI6$ で $I6$ が 4 なら $Y4$ とかく。結果は card に punch される。

iteration statement

k : j, $v1$, $v2$, $v3$, $v4$

ここで j は integer でなければならない。$v1$ は変数で $v2$, $v3$, $v4$ は X, D, P を含まない数式である。v の norm は 5 をこえてはならない。これは k から j までの間に loop をつくり，$v1$ を $v2$ から $v3$ づつ $v4$ までかえて iteration を行なうことを指示する。

(例) 15 : 19, $I1$, $I3+I4$, $I4$, $I5+1$

21 : $Y5 \leftarrow CI1+2$

19 : $YI1 \leftarrow Y5-7$

$v3$ は負でもよい。$v4-v2$ が $v3$ でわり切れなければ $v1$ が $v4$ をこえる直前にくりかえしはとまる。iteration statement の nesting の深さは 4 重をこえてはならない。iteration statement には整数を使うのが普通であるが，必ずしも，それに限られることはない。しかし，整数以外の場合は慎重を期さなければならない。

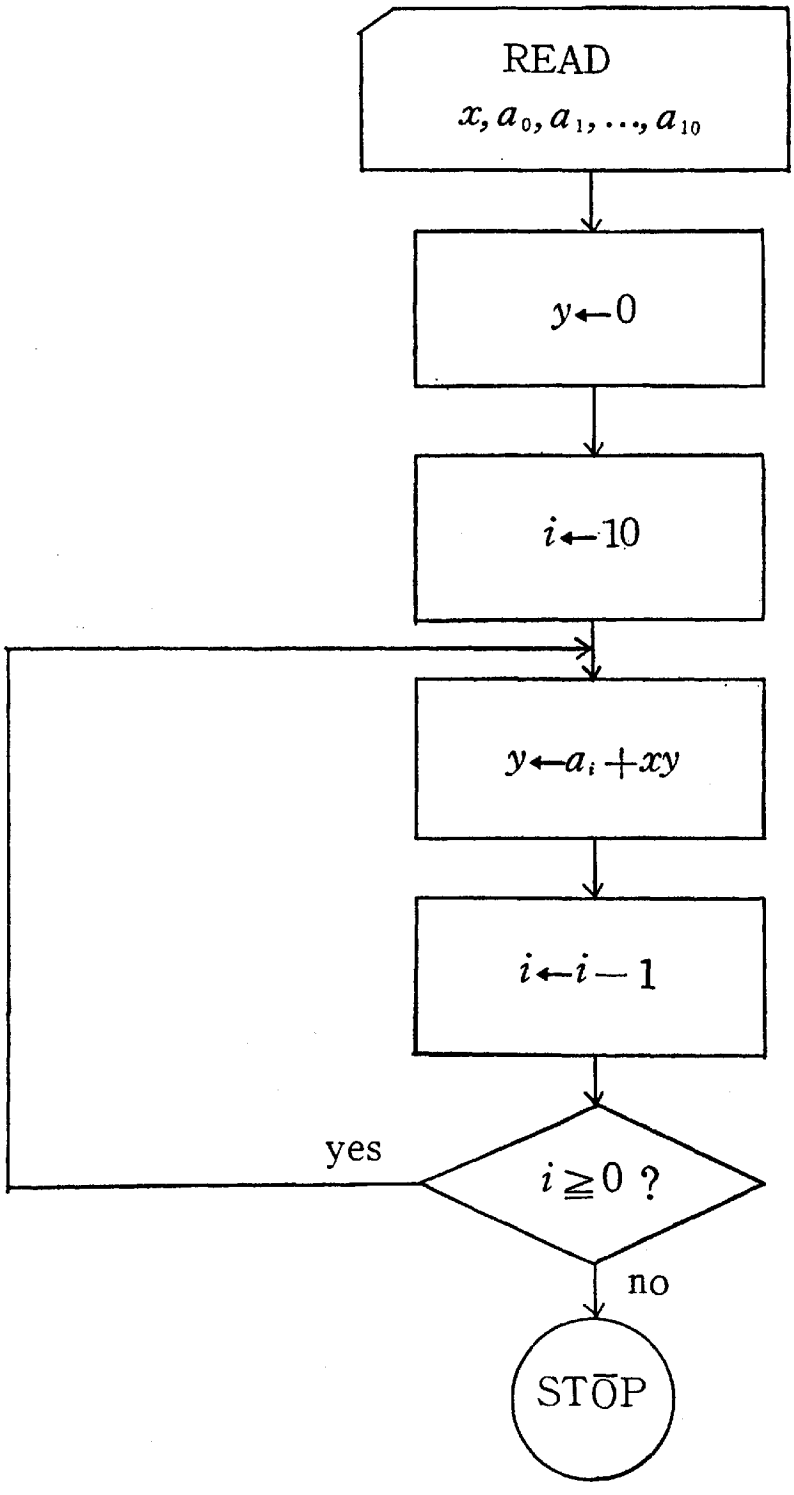
extension statement

k : "nE, ..."

これは subroutine の call statement である。

ここで，IT program の例をあげよう。

問題：$y=\sum_{i=0}^{10} a_i x^i$



$y=\sum_{i=0}^{10} a_i x^i$ の flow chart

IT program (1)

1 : READ	F
2 : $Y2 \leftarrow 0$	F
3 : $I1 \leftarrow 11$	F
4 : $Y2 \leftarrow CI1 + (Y1 \times Y2)$	F
5 : $I1 \leftarrow I1 - 1$	F
6 : $G4\ IF\ I1 \geqq 1$	F
7 : H	FF

IT program (2) ($I5$ に次数を与える)

1 : READ	F
2 : $Y2 \leftarrow 0$	F
3 : 4, $I1$, $I5$, -1, 1	F
4 : $Y2 \leftarrow CI1 + (Y1 \times Y2)$	F
7 : H	FF

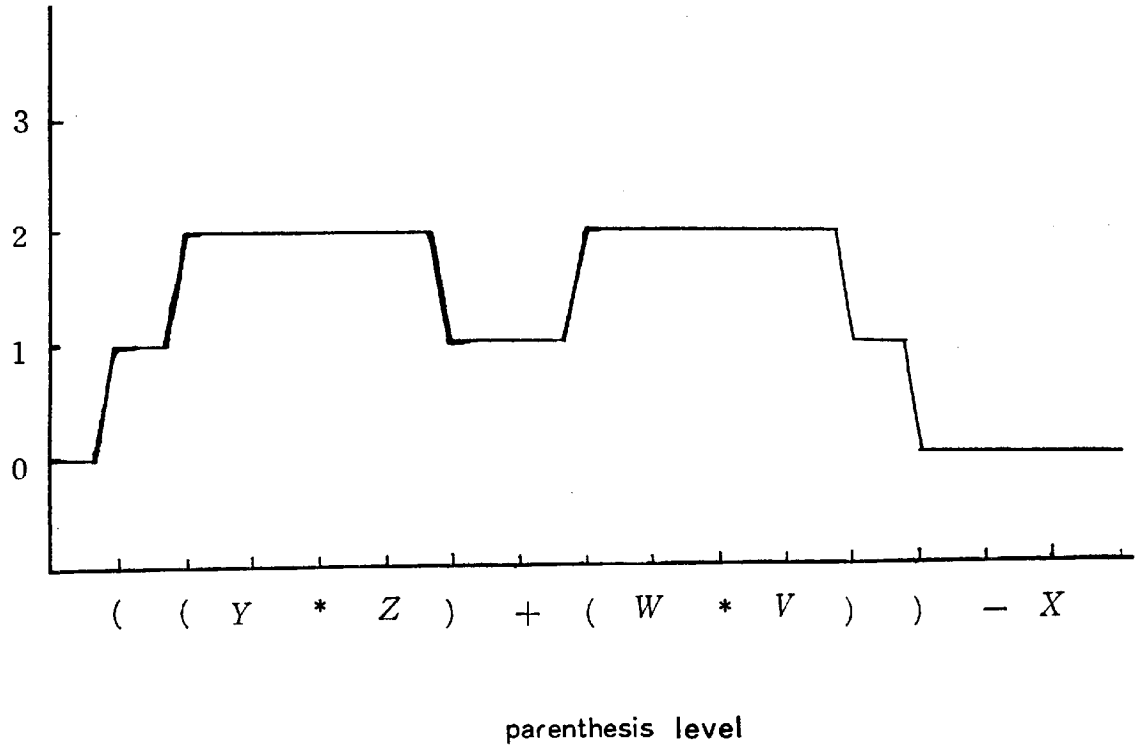
F は statement end mark, FF は program end mark である。

以上の概観にみられる通り，IT は記法こそ違うが文法的には FORTRAN に基本的にはきわめて類似している。従って，このような文法をもつ programming language の構想は 1950 年代後半には合衆国でかなり一般化していたものと考えてよいであろう。

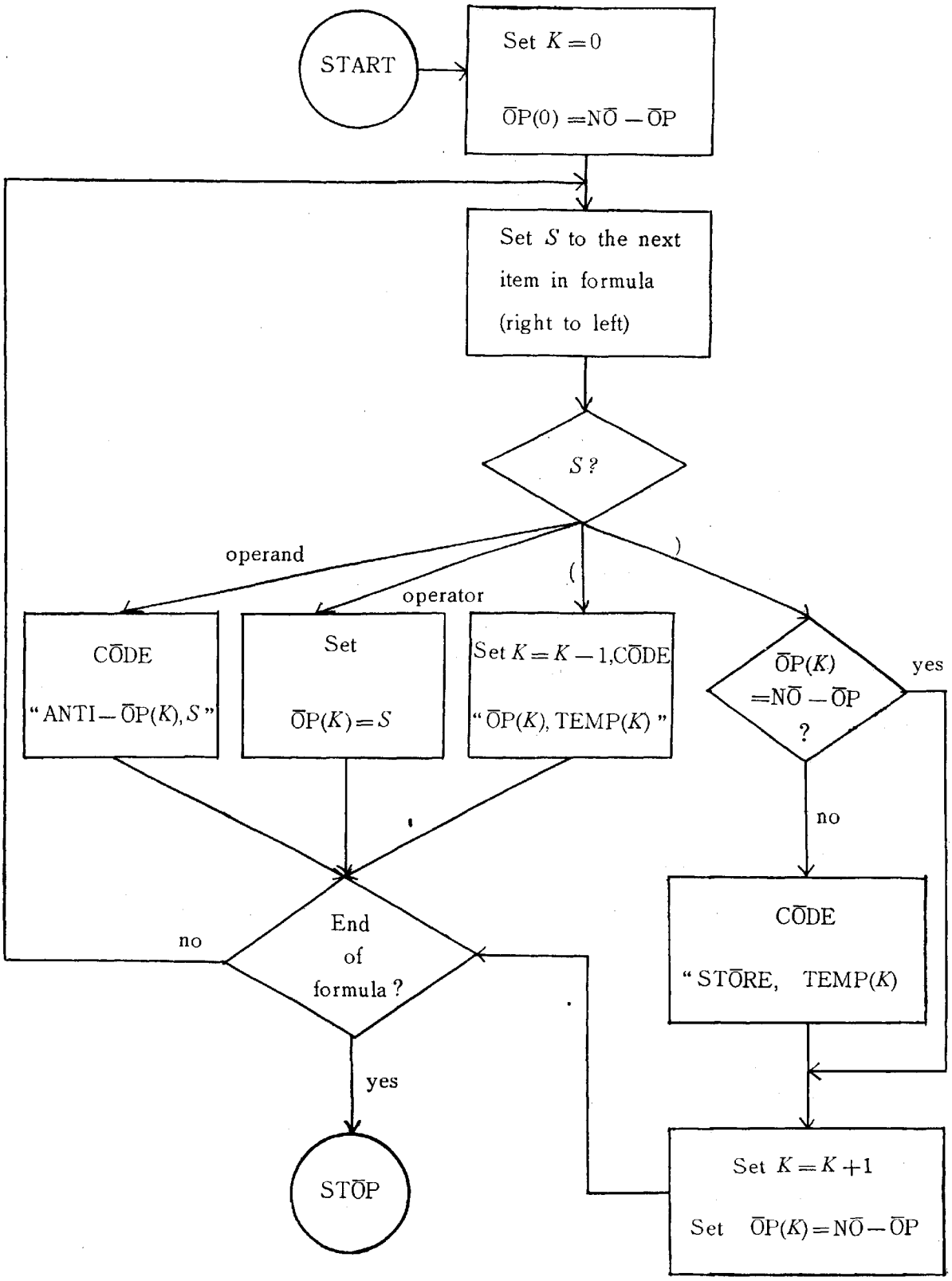
それでは，IT compiler はどのような compilation を行なっているのであろうか。以下では，IT compiler の数式翻訳方法の大要をのべよう。例えば

$$((Y*Z)+(W*V))-X$$

という数式を考えよう。ここでは変数のかき方などは IT の文法には従わないことにする。IT compiler はまず parenthesis level を考える。すなわち，数式に関して等高線を左側の括弧で 1 段上り 右側の括弧で 1 段下るようにしてつくり，その結果えられる「標高」を parenthesis level と名付ける。上の数式ではこれは次のようになる。



IT compiler では maximum parenthesis level として 10 までを許している。次に数式は各 level 毎の処理が行なわれるが，その手続は大体次の flow chart で示されるようなものである。



数式翻訳の flow chart

binary operator に対しては，一般に，次の条件をみたす anti-operator を考えることができる。

A op $B=B$ anti-op A

例えば，$A+B=B+A$ であるから＋はそれ自身が anti-op である。また，/ の anti-op は inverse divide である。ここでは，このような anti-op が computer ですべて利用できるものとする。NŌ-ŌP と LŌAD はたがいに anti-op である。このことは

LŌAD (b) ; $0+b$

NŌ-ŌP (b) ; $b+0$

と考えればあきらかであろう。

さて，compilation は次のように進む。右から左に数式をみて記号を次々に1つづつ取出し，その記号の type によって flow chart が示すように処理が4つにわかれる。

1. operand (variable または constant) ならば

ANTI-ŌP(K), ŌPERAND

という machine code をつくり出す。counter K は parenthesis level を示す。これは最初は0から始める。別に，ŌP-table があって，ŌP(0) から ŌP(9) までで，できている。

2. operator ならば，これを ŌP-table の ŌP(K) の所に save する。

3. (ならば K を1つへらし

ŌP(K), TEMP(K)

という machine code をつくり出す。TEMP(K) は K 番目の temporary storage である。

4.) なら，まず

ŌP(K)=NŌ-ŌP

かどうかをたしかめる。もし違うなら

STŌRE, TEMP(K)

という machine code をつくり出し，K を1だけふやして，ŌP(K) に NŌ-ŌP を save する。同じならば，単に K を1だけふやして，ŌP(K) に NŌ-ŌP を save する。

この手続を数式の最後までくりかえす。上にあげた数式にこれを適用すると次のようになる。

symbol	K	$\bar{O}P$-table			machine code
	0	NŌ-ŌP			
X	0	NŌ-ŌP			LŌAD X
−	0	−			
)	1	−	NŌ-ŌP		STŌRE TEMP(0)
)	2	−	NŌ-ŌP	NŌ-ŌP	
V	2	−	NŌ-ŌP	NŌ-ŌP	LŌAD V
*	2	−	NŌ-ŌP	*	
W	2	−	NŌ-ŌP	*	MULTIPLY W
(	1	−	NŌ-ŌP		NŌ-ŌP TEMP(1)
+	1	−	+		
)	2	−	+	NŌ-ŌP	STŌRE TEMP(1)
Z	2	−	+	NŌ-ŌP	LŌAD Z
*	2	−	+	*	
Y	2	−	+	*	MULTIPLY Y
(	1	−	+		ADD TEMP(1)
(	0	−			SUBTRACT TEMP(0)

このように IT compiler には operator の優先順序という考え方はまだみられない。従って，IT の program では数式に完全に括弧を施して演算の順序を指定しなければならないのである。数式翻訳については以上の他に subscript や subroutine をどのように扱っているかという問題もあるが，ここでは IT compiler はそれらのすべてを考慮に入れているということを指摘するだけに止める。しかし，最後に，IT compiler がつくり出す object program は余り能率のよいものでなかったことを注意しておこう。例えば上例の program に関していえば，最初の 2 命令は不必要であるし，5 番目の NŌ-ŌP 命令もいらない。このような object program の能率の悪さは初期の compiler の共通の特徴であった。

さて，IT は非常に多くの影響をのちの programming language の発展に与えてい

る。IBM と Carnegie 工科大学が 1957 年 10 月に開発した FŌRTRANSIT という compiler は，FŌRTRAN の部分集合から IT に翻訳するものである。翻訳されてできた IT program は次に SŌAP program に翻訳され，最後に SŌAP assembler によって machine code に直される。FŌRTRANSIT はこのように翻訳の手続が面倒であったので余り普及しなかった。