自動フロー・チャーティング*

前　川　　守**

For the purpose of program debugging and documentation, the automatic converting of instruction sequences into flow-charts is very useful, because such work is usually very time-consuming and cumbersome. The automatic flowcharting process is divided into four parts. The 1 st step is to combine several instructions by two principles. The 2 nd step is to describe flow in the computer. The list-notation is used for this description. The 3 rd and 4 th steps are positioning and lining, respectively.

1. 序

プログラムの表現法としてはフロー・チャートが一般に広く用いられている．フロー・チャートの目的としては，第1にプログラム製作者以外の人々にプログラムを理解してもらうためのドキュメンテーションの一環として書かれる場合が考えられ，第2にプログラム製作者自体のプログラム・チェックに用いられる場合が考えられる．

プログラム作成においては，通常まずフロー・チャートを書きそれをコーディングする．最初に書いたフロー・チャートのままプログラムが完成することは，ごく短いプログラムを除いてまれであり，最初のフロー・チャートと完成後のフロー・チャートは変更されている場合が多い．このようにして完成したプログラムはあとあとの使用のためにプログラムの完全な記録をとっておくことが必要であり，そのための煩わしさは少なくない．特にシステム・プログラムの如き膨大な量にのぼるものでは，完成すると同時に製作者は安堵してしまい，フロー・チャートを書き直すことはなかなか実行されにくいものである．これが計算機で行なわれるならば労力の軽減になると同時に，フロー・チャートの標準化という面でも大きな利益がある．

また，デバッグにおいては，命令シーケンスをフロー・チャートとして容易に見直せることはかなり意義がある．フロー・チャートを書く人間とコーディングする人間とは異なることが多いが，そのような場合フロー・チャートを書く人からコーディングする人に渡されるフロー・チャートは何が行なわれるべきか（どのようにコーディングされるべきか）を規定するものであるのに比し，デバッグに必要なフロー・チャートはコーディングされた命令シーケンスにおいて，何が行なわれているかを明確に規定するものである．これら両者のフロー・チャートを比較することにより，デバッグの目的を達することができる．このためには，命令シーケンスを容易に正確にフロー・チャートとして見られなければならない．これはディスプレイなどの利用により，さらに実用的なものとなる．たとえば我々はよくトレースということを行なうが，トレースがフロー・チャート上で行なわれるならばデバッグする人にとっては朗報であろう．

本論文ではプログラムを表現する手段としてのフロー・チャートについて機能ブロックの結合として，およびコントロールの流れとしての見方からの解析をし，その表現法としてリストを用いた表現法について述べた．この表現法を用いて命令シーケンスをフロー・チャートに書き，図面上に配置するプログラムについて説明した．また，ある場合にはプログラムの短縮化が自動的に見出されるのであるがそれについても述べた．

通常自動フロー・チャーティングにおいてコンパイラ・レベルの言語を扱うのはアセンブラ言語よりも一般的にいってやさしい．それはステートメントをまとめて1個の論理ブロックにすることが計算機にとってはむずかしいことによる．本論文ではアセンブラ言語を扱っているが，その原則はコンパイラ言語にも適用される．自動フロー・チャーティングを行なうプログラム（Automatic Flow Charting Program）を以下AFCP と略称する．

2. 原　　理

2.1 ブロック化

命令シーケンスをブロックに分割することは，なか

* Automatic Flow-Charting, by Mamoru Maekawa (Electronic Computer Engineering Dept., Tokyo Shibaura Electric Co., Ltd.)
** 東京芝浦電気株式会社電子計算機技術部

なかに困難な問題である．人間が見る場合には，それらの論理的つながりを判断し，論理的つながりがある命令を1個のブロックにまとめることは，さほど困難なことではない．しかし計算機がそれを判断するとなると大変むずかしい．そこには思考が要求されるからである．

コンパイラ言語，たとえばFORTRAN語などの場合は個々のステートメントを1個のブロックとしても，フロー・チャートはさほど細かくなりすぎることはないので実用に供せられる．しかしながら，アセンブラ語などの場合は個々のステートメントを1個のブロックにするような方法は実用に供しえない．

フロー・チャートと一言でいってもその程度は千差万別であるが，ここでは次の段階を対象とする．

（i） アドレスがはいったフロー・チャート
（機械に左右される表現である）

（ii） アドレスをなくしたフロー・チャート
（機械に左右されない表現である）

(iii) いくつかのブロックをまとめて大きいブロックとしたフロー・チャート

(i) は命令との対応が最も近く普通機械命令3～5命令が一つのブロックとなる．**Fig. 1** にその例を示す．

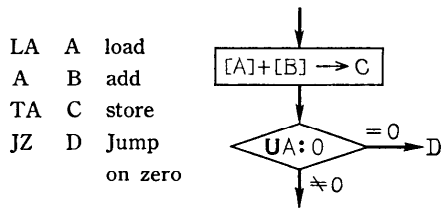


Fig. 1 Instruction sequence

このレベルのフロー・チャートはコーディング直前のフロー・チャートのレベルとちょうど一致する．このレベルのフロー・チャートはプログラムの製作者間の連絡（たとえばフロー・チャートを書く人とコーディングする人，または数人～十数人でコーディングしている場合はその連絡に）およびトレースの際のフロー・チャートとしてはなはだ有効である．これはフロー・チャート上の変更が直ちにコーディングに及ぶことで意味がある．また，このレベルのフロー・チャートからはあまりプログラムの内容を理解することなくコーディングが可能である．したがって，あとあとの使用のためにこのレベルのフロー・チャートを残しておくことは意味がある．

このレベルのフロー・チャートの欠点は第三者が理解しにくいことであり，そのプログラム担当者以外の人間が見て理解するには細かすぎる点である．またアドレスにより記述されているため理解しにくい．

この欠点を無くするためにアドレスをなくしたのが次の（ii）のレベルである．すなわち，この段階では扱われるデータをプログラマに理解しやすいシンボルでもって表現する．この（i）から（ii）のレベルへの移行は計算機では極めて困難であり，コメントに頼るか直接人間の助けを借りなければならない．このレベルのフロー・チャートは担当者以外の者がくわしくプログラムを見るのに適している．最後の（iii）のレベルはいくつかのブロックを結合して一つのブロックとし全体の大きい流れを捕えるのに便利にしたものである．ブロックをまとめることは計算機にとってはなかなかに難しいことである．普通にはコントロールの流れから判断してブロック化を行なう．機能的見地から，より大きな論理ブロックにまとめあげることは，今のところコメントに頼るか人間の助けを借りる以外に方法はなさそうである．

以下 (i) のレベルのフロー・チャートのブロック化を行なう原則について述べる．

原則には二つある．

原則 1． ある一つの機能を果している命令列をまとめて一つのブロックとする．

原則 2． コントロールの流れに着目する．

プログラムをながめる観点はいくつもあると思えるが，ブロック化をはかるに当たっては二つの観点から，とらえるのがよいように思われる．すなわち一つはコントロールの流れとしてとらえる方向であり，これは判断命令にその重きがおかれる．他はデータを処理するものとしての機能面から見た方向である．

過去におけるフロー・チャートの解析を行なった論文は原則2に基づいている．原則1は当然考慮されなければならないものながら，その判断基準を打ち立てることがむずかしいため，あまり考察されていない．

本論文では，原則1に立つブロック化の規準をいくつか設けた．それらの規準の中には理論的背景はないが，実際のフロー・チャートとよく一致する（人間の心理に関係するような）規準もある．

プログラムはせんじつめればある入力情報に従い，何らかの処理をして出力情報を出すことであるから，かようなことが行なわれている命令シーケンスを1個のブロックとすればよいことがわかる．このブロックのレベルとしてはプログラム全体を1個のブロックと

するレベルから個々の命令を1個のブロックとするレベルまである．この両極端のレベルにおいてブロック化をするのは比較的やさしいが，その中間的段階におけるブロック化は大変むずかしい．この中間的段階のブロックを作るには，まず命令シーケンスを最も細かいレベルのブロックに分割し，さらにその上の大きいブロックはこれらを結合させることによって達成する方法と，一個の大きいブロックを主としてコントロールの流れによって分割していく方法がある．

原則2はコントロールの流れとしてとらえる方向である．プログラムをブロック化するのに最も基本的な考え方の一つとしては，分岐命令でもってプログラムを分断し，分岐命令ではさまれた部分を一つのブロック，分岐命令をまた一つのブロックとする方式がある．

これはプログラムのコントロールの流れを解析するには有用な方法である．コントロールの流れを追う場合に問題になることの一つとして，判断命令，飛越命令の行先がプログラムで書き変えられる場合および命令自身が書き変えられる場合には通常の表現では処理できない．本論文ではこれらの場合は扱っていない．

以上二つの原則をあげたが，これに基づきいくつかの判断の基準を設けた．

●原則1に基づく規準

規準 1　あるデータが持ってこられて格納されるまでを一つのブロックとする．

規準 2　スーパバイザ・コールは一つのブロックとする．

規準 3　サブルーチン・コールは一つのブロックとする．

●原則2に基づく規準

規準 4　判断命令は一つのブロックとする．

●原則1および2に基づく規準

規準 5　ネームのついている命令はブロックの区切りとなりその命令の含まれるブロックの一番最初の命令となる．

規準 6　飛越命令は一般にブロックの区切りとする．

規準1は機能ブロックとしてのとらえ方の最も基本となる規準であり，具体的には一般にロード命令で始まり，何らかの演算命令（算術演算のほか論理演算，ビットのシフトなど）がありストア命令で終る．

ロード命令，ストア命令，演算命令が何であるかは個々の計算機により具体的に検討しなければならない．

規準 2，3 は機能ブロックとしてもっとも明確な形をしている場合である．TOSBAC-3400 ではこれらの機能のために特別の命令を有しているので判別は容易である．

規準4はプログラムの流れを追うのにもっとも基本となる規準である．

規準 5，6 は原則1および2に基づくものである．それは人間がプログラムを作る場合にはネームを無意識的に，ある一つの機能を果す命令列の頭につける場合と判断命令により分岐された頭の番地にネームをつける場合とがほとんどの場合であることによる．

これらの規準は理論的裏付けはないが実際に作られたプログラムのブロック化によく符合するので取り入れた．飛越命令は普通ネームのついている命令に飛ぶのが普通であるから，規準 5，6 は相互に一致する．しかし相対番地形式で飛ぶ場合には一致しない．

2.2　フロー分析

2.2 では命令列をブロックにまとめる規準を示した．これらの規準により作られるブロックはフローの最小構成単位であり，もはや分割できないものである．

これらブロックを単位としてフロー分析を行なう．その主眼点はフローをリストでもって表現することにある．Krider, Lee[3] はフローをリストで表現する表記法として *- 記法なるものを導入している．著者はその表記法に計算機処理に適合するように若干の修正を加えると共にプログラムの計算機による簡素化（冗長度を少なくする）をはかるために λ- オペレータおよび γ- オペランドを導入した．以下それらについて述べる．

定義

原ブロック，目的ブロック：ブロックAからブロックBにコントロールが移る場合ブロックAを原ブロック，ブロックBを目的ブロックという．

*- オペレータ：ブロックAからブロックBにコントロールが移ることを示す表現として

*AB

という表現を用いる．ここで＊はオペレータである．ブロックAの目的ブロックがブロック B, C である場合には

*²ABC

という表現を用いる．以下ブロックAの目的ブロックが B_1～B_n の場合には

$*^nAB_1\cdots\cdots B_n$

として表現する.

このようにAからコントロールが移っていく場合，A以後に続く表現（A およびAの前の $*^n$ オペレータを含めて）を“A の *- 記法”と呼ぶ.

λ- オペレータ：

ブロック A, B の目的ブロックがいずれもブロックCである場合には

λ^2ABC

でもって表現する.

γ- オペランド：

ブロック A, B の目的ブロックがいずれもブロックCである場合

$*A\gamma C$ $*B\gamma C$

でもって表現できる.

Ω- オペランド：

プログラムの論理的終了を示すオペランド

ループ：

Aの *- 記法において，Aが目的ブロックとして再び現われる場合それをループと呼び，そのループをAのループと呼ぶ.

最小ループ：

Aのループにおいて二つのAにはさまれるブロックのすべてが最小限2度以上実行される場合そのループを最小ループと呼ぶ.

（注） フローは上記のオペレータ，オペランドのすべてを用いて表現できるが，すべてを用いなくても表現できる.

このように定義するとき

入替ルール

$*^2ABC=*^2ACB$

$\lambda^2ABC=\lambda^2BAC$

削除ルール

$*^2A*B*C\gamma D*E*C\gamma D$

$=*^2A*B\gamma C*E\gamma *CD$

定理 1.

ブロックAの *- 記法においてAが目的ブロックとして現われれば，その間はループを形成し，その一部分が最小ループを形成する.

定理 2.

Aのループにおいて入替ルール，削除ルールを適用して二つのAが最も近くなるようにしたとき，それはAの最小ループである.

3. ブロックの分割および結合

ブロックを結合させるには，**2.1** ブロック化の原則1，2がそのまま適用しうるわけであるが，最小単位としてのブロックを作るための **2.1** ブロック化の規準1～6 とは異なるのは当然である.

ブロックから新しいブロックを作っていく規準を原則1によるものと2によるものに分けて論ずる.

3.1 原則1による規準

最小単位のブロックをいくつかまとめて新しい機能ブロックに作りあげるのは不可能に近く現在のところ人間のコメントにたよる以外に方法はなさそうである.

3.2 原則2による規準

コントロールの流れとしての見方からブロックの結合，分割を行なう規準について述べる.

規準A 列

分岐命令を一つも含まないブロックの結合

Fig. 2 にその例を示す. *- 記法で示せば

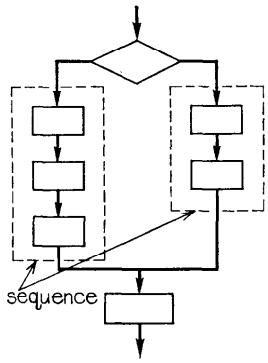


Fig. 2 Sequence

$*A*B*C*\cdots\cdots*P$

の形であるものをまとめる. その間に * の1次のオペレータしか現われない.

規準B 合流

ある1個のブロックから分岐したのち，その分岐した流れすべてが，再びある他のブロックに合流する場合これらすべてのブロックをまとめて1個のブロックとする. **Fig. 3** にその例を示す. *- 記法で示せば

$*^2A*B*C\gamma D$ $E\gamma D$

の形になる.

規準C 逐次分岐

ある一つの変数の値に対して分岐が幾段にもなされる場合これらの分岐をまとめて多分岐の一つのブロックとする. **Fig. 4** にその例を示す. *- 記法で示せば

$*^2AB*^2CD*^2EFG$ の形になる.

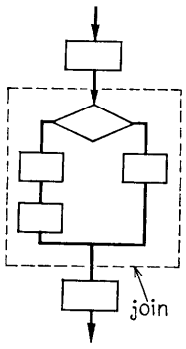


Fig. 3　Join

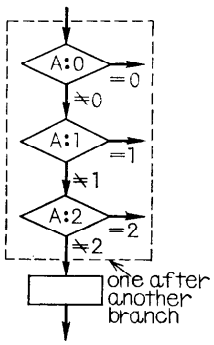


Fig. 4　Ohe after another branch

規準D　ループ

ループはまとめて1個のブロックとする．*- 記法で示せば

*A*B*C*²DAE

の形になる．

図に示された例からもわかるように，これら結合されたブロックを用いてさらにブロックを結合していくことができるし，また逆にブロックを幾段にも分解していくことができる．

4. フロー・チャート化

さて以上によりコーディングされたプログラムからフロー・チャートに書く手順は

（i）　ブロック化の規準 1～6 を適用してブロックにまとめる．もっとも重要かつ負担の多いところである．

（ii）　このようにしてできた各ブロックの結合は *- 記法（計算機の中ではリスト表現）により表わされているわけであるが，これに入替ルール，削除ルールを適用してフロー・チャート化に適するように変える．

（iii）　フロー・チャートのレベルに応じてブロックの結合を行なう．このフェーズでは人間とのコミュニケーションが要求される．

（iv）　最終的な *- 記法によりブロックを紙面上に配置する．

（v）　各ブロックを線で結ぶ．

(iv) について述べれば，このフェーズは千差万別であるが AFCP で約 *- 記法を活かして2次元的配置をする．1枚の用紙の面積を $A \times B$（縦×横）とすると，つぎの条件 1，2 に合致する範囲内のブロックを用紙に配置する．配置の順序はフローの順に用紙の左上隅から右下隅に向って配置する．ここで1ブロックの所要単位面積を $a \times b$（縦×横）とする．

条件 1　ブロックの総数は次の値以下でなければならない．

$[A/a] \times [B \times b]$

条件 2　分岐の総数は次の値以下でなければならない．

$[B/b]$

この方法はもっとも容易な方法といえるが，人間が書くのと同じように巧みに配置するのはなかなかに困難な問題であると同時に，興味のそそられる問題である．

5. 自動フロー・チャーティング・プログラム

AFCP はいくつかのフェーズから成る．

1．プログラムをブロック化する．

これは II の原則，規準に従って行なうわけである．このフェーズでは機能ブロックにプログラムをまとめあげるわけであるが，アセンブラ言語においては多くの困難な問題がある．

- 間接番地指定（特にジャンプ命令における）
- シフト命令の表現
- アセンブラ擬似命令の表現
- サブルーチン

などが問題である．アセンブラ言語では命令の書き換えがよく行なわれるが，その場合の処置はさらに困難である．このフェーズの処理プログラムはアセンブラとトレーサの結合のようなものであるがトレーサとは異なり，総ての場合を追跡しなければならない．このフェーズの入力はプログラムのソース・カードであり，出力はブロック化されたブロックの *- 記法（計算機でのリスト表現）である．また同時に各ブロックと命令との対応表を作り別のファイルとして出力する．この場合に各ブロックにはシーケンス番号が割り振られており，各ブロックの識別はその番号でもって行なう．さらに γ- オペランドの付加されたブロックは他のファイルとして記憶する．これはプログラムの自動簡素化を行なう準備である．

2．γ- オペランドの付加されたブロックの相互関係を調査して，同じ内容のブロックの場合はもっとも若い番号にブロック番号を合わせる．これはプログラムの自動簡素化の一つの方法である．

いま，ブロック E が目的ブロックとなるブロック C, D があればその *- 記法はつぎのようになる．

*A*CγE *B*D*γE

(A, B はそれぞれ C, D の原ブロック)

これら γ- オペランドの付加された C, D ブロックおよびそれの前後関係は目的ブロック別に一つのフェーズで記憶されているので，それをもとにCとDが同じ内容を有するものか検べて，もし同じ場合は

*A*CγE *BγC

に修正する．これをすべての γ- オペランドについて行なう．

3. 入替ルール，削除ルールを適用して早く現われるブロックから順に最小ループを作っていく．

4. 外部からの指定にしたがいブロックの結合，分割を行なう．最も詳細なフロー・チャートはフェーズの1でブロック化されたものをそのまま使うこと．それ以上は **3.** の原則，規準にしたがってまとめる．

5. 新たにまとめられたブロックの *- 記法（リスト表現）にしたがい面積査定によりブロックを定められた用紙に配置していく．

6. 各ブロックを線で結ぶ．

6. あとがき

自動フロー・チャーティングの程度は千差万別であるが，少し高級にするとむずかしさが飛躍的に増大する傾向がある．特に論理的なブロックの構成の方法および定められた用紙上に，いかに人間が見た場合の抵抗感なくブロックを配置するという配置方法が困難な問題である．また自動フロー・チャーティングを行なうプログラムで注意せねばならないのは，その処理に要する記憶容量の膨大さと処理速度に対していかに現実の計算機に合わせるかということである．

参考文献

1) Haibt, Lois M.: A Program to Draw Multi-level Flow Charts, Proceedings of the Western Joint Computer Conference, 1959, pp. 131～137
2) Karp, Richard M.: A Note on the Application of Graph Theory to Digital Computer Programming, Information and Control, Vol. 3, No. 2 (1960), pp. 179～190
3) Krider, Lee.: Flow Analysis Algorithm, Journal of the Association for Computing Machinary, Vol. 11, No. 4 (1964), pp. 429～436
4) Prosser, Resse T.: Application of Boolean Matrices to the Analysis of Flow Diagrams, Proceedings of the Eastern Joint Computer Conference, 1959, pp. 133～138
5) Schurmann, A.: The Application of Graphs to the Analysis of Distribution of Loops in a Program, Information and Control, Vol. 7, No. 3 (1964), pp. 275～282
6) Voorhees, Edward A.: Algebraic Formulation of Flow Diagrams, Communications of the Association for Computing Machinary, Vol. 1, No. 6 (1958). pp. 4～8
7) W.B. Stelwagon: Principles and Procedures for the Automatic Flow Charting Program FLOW 2, Research Department.

付録 1. ブロック化の例

Fig. 5 のようなプログラムは **Fig. 6** のようにブロック化される．Ffg. 6 は計算機によるラインプリンタへの出力である．

ブロック化の規準を示すと

ブロック番号	原 則	規 準
1	1	1, 5
2	1	2, 5
3	1	1, 3
4	1	3
5	2	4
6	2	4
7	1	1, 3
8	1	3
9	1	1, 5
10	1	1, 3, 5
11	1	3
12	1, 2	1, 6
13	1	2, 5

となる．これを *- 記法で表わせば

*1*2*3*4*²5*²6*7*8*9 γ 2*10*11*12 γ 2*13 Ω 14

である．このフロー・チャートでは

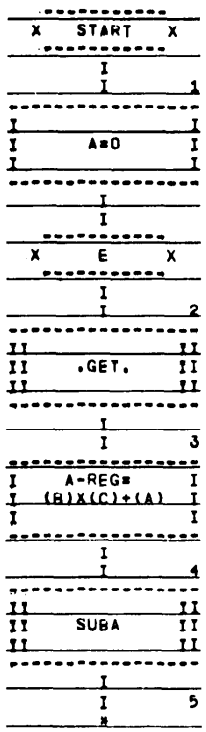
- 変数はそれの記憶されているアドレスで示す．
- サブルーチンは ‖ ‖ でもって示し，中にサブルーチンの名前を書く．
- レーベルが付加されているブロックには

⊂⊃ で示される記号がレーベルと共につく．

⊂⊃ はまた端子記号としても用いられる．

などの特徴がある．

上に示される *- 記法において，γ- オペランドの付加されたブロックを調査することにより，削除ルー




```
      CA              clear
      TA      A       store
E     ESM             enter supervisor mode
      ADR     .GET.   address constant
      LAF     B       load
      MF      C       multiply
      AF      D       add
      JTL     SUBA    jump and store link
                      address
      JZF     L       jump on zero
      JMF     F       jump on minus
      LAF     G
      DF      J       divide
      JTL     SUBB
      JAF     K       store
      J       E       jump
F     LAF     I
      DF      J
      JTL     SUBB
      TAF     K
      J       E
L     ESM
      ADR     .PUT.
      HP              halt and proceed
```

Fig. 5

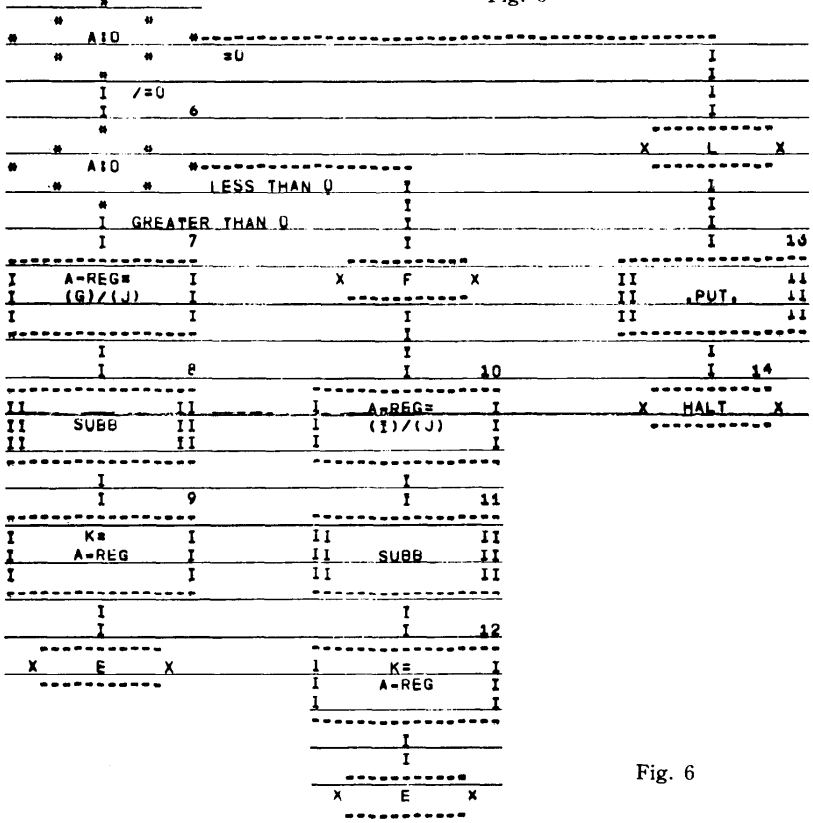


Fig. 6

ルを適用してブロック 12 を9に置きかえてよいことがわかり，さらに反復して 11 を8に置きかえてよいことがわかる．このようにして自動簡素化がなされる．

また，このフロー・チャートをブロックの結合によりまとめていくことができる．たとえば

ブロック 3, 4 を“列”により

5, 6 を“逐次分岐”によりまとめることができる．

付録 2. AFCP（自動フロー・チャーティング・プログラム）

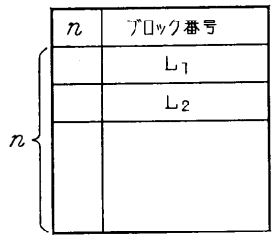
AFCP においてもっとも負担の多い **4.** のフェーズ (i) についてのみ述べる．このフェーズでやることは四つに大別される．

1) プログラムを読込みながら規準 1～6 を適用して命令列をブロックに区切っていく．これら個々のブロックはそれを構成するソース・イメージ（カード・イメージ）のまま1個のブロックが一つのレコードとして外部記憶装置に書き出される．なお，これらブロックには識別のための番号が付加されている．これらの各レコードは極めてひんぱんにランダム・アクセスされるので，ランダム・アクセス・ファイルに書き出す必要がある．

2) ネーム・テーブルを作る．すなわち表われたネームは内部記憶装置に積み上げる．各ネームはそれの含まれるブロックの番号と対応させられている．

3) 各命令のアドレスを定める．すなわち各ブロックのアドレスを定める．

4) 以上 1)～3) の準備が終了すればリストによるフローの表現に入る．最初のブロックからフローの順に追っていく．計算機内部の表現は下記のようにした．



L_i: 次のブロックのポインタ
n: ← 記法における ← オペレータの次数と一致し語数でもある．

リストによる表現によればメモリの節約ができる．たとえば1万語のプログラムでは平均3命令で一つのブロックを成すので，約3,300のブロックができる．

これを計算機内でリストにより表現すると1ブロックが 2～3 語を要するので，結局約1万語を要することとなる．これは通常の計算機でもって処理できる値である．これをもし Boolean Matrics で表現しようとすると，3,300×3,300 で約 1,000 万の情報単位を必要とする．これらをビット単位で表現したとしても，よほど大きな計算機でないと収容しきれない．

さてフェーズ (i) で問題になることとしては

- 間接番地によるアドレスの表現
  （特にジャンプ命令の場合）
- アセンブラ擬似命令の処理
- コメントの利用
- 処理しているプログラムがサブルーチンか否かの判定．

などである．

間接番地による表現では，実効番地がネームにより定められる場合は問題ないが，書き変えが行なわれる場合の処置として，一部トレースを行ない定めると同時に，外部から人間により指定できることとした．

アセンブラ擬似命令の処理では個々の擬似命令について検討しなければならない．

たとえば

CSEC (Identify Control Section)

USING (Use Base Address Register)

などは，この命令に引続くブロックのコメントとして付記する．

コメントの利用では各命令に付加されているコメントは，その命令の含まれるブロックに付加するが，コメントのみのステートメントはブロックからブロックへの線にコメントとして付加する．

サブルーチンの判定は TOSBAC-3400 では比較的容易である．

(昭和 42 年 9 月 16 日受付，昭和 43 年 2 月 5 日再受付)