報　告

パネル討論会
構造的プログラミング
昭和50年度第16回大会*報告

パネリスト
木　村　　泉（東京工業大学）
原　田　賢　一（慶応義塾大学）
藤　林　信　也（日本電気(株)）
（司会）和　田　英　一（東京大学）

和田　構造的プログラミングというのはプログラムの作り方の反省だと思うが，そうではないという意見もある．本会では高橋先生が会長のときソフトウェア危機を打開するにはどうすればよいか，という懸賞論文を募集した．それがどうなったかは知らないが，これは世界的な課題で，ひとつの答えが，ダイクストラのgoto文有害論（goto文の数とプログラムの悪さとは比例する）であった．次に彼は“Notes on Structured Programming”を書いてそれがサーキュレートしていた．それがダール，ホアーと一緒の本になった．その後，世の中には2種類の文献が多くなった．そのひとつは「なになにによる構造的プログラミング」“Structured programming with goto statement”あるいは“Structured programming in Fortran”．一方では「なになに有害論」グローバル変数，if then else．ポインター変数，ことしの春頃は構造的プログラミング有害論．N. Y. 大のワインバーグは，有害論有害論というのを書いている．プログラミング・シンポジウムではことしの夏に「構造的プログラミングとその経験」というシンポジウムを開いた．

私は構造的プログラミングはさっきあげた3冊の本にあるものと思っていたが，夏のシンポジウムにいってみたらもっといろいろあった（**T. P. 1**）．その報告集によれば，IBM流SPがある．ダイクストラの構造的プログラミングとは，質のちがうものである．SP, IPTはそれぞれStructured Programming, Improved Programming Techniqueの略である．もうひとつはダイク流で，更にこの方は構造的「に」と構造的「な」のふたつがある．これはダイクストラの本をよんでもどちらかわからないことがある．プログラムは構造にわけて調べないとロジックはよくわからないと書いたところもあり，作り方のこともいっているので，ダイクストラも「な」と「に」の両方の話をしているようだ．そこでこう書いたが，ダイクストラのはどちらかに寄ったり両方だったりするようだ．

T. P. 1

構造的プログラミング
- IBM流SP (IPT)
- ダイク流
  - 作り方　に
  - 完成品　な

T. P. 2

構造的プログラミング

意味，背景，哲学	木村
データの型の抽象化	藤林
構造的…用言語	原田
好き，嫌い，その他	全員

Machine Oriented Higher Level Languageの報告集をみると，そこでホーニングが「な」の方は，千匹のサルがキーパンチをたたいてもできるかもしれず，構造的に作っても，構造のみえないものができるかもしれない，だからそのふたつはちがうんだといっているが，始終混乱している．きょうは大体は「に」，たまには「な」，そしてたまにはSPかと思う．パネリストの分担は（**T. P. 2**），意味とか背景とか哲学という点を，木村さん．データの型の抽象化を藤村さん．原田さんには構造的プログラミング用言語．最後にそれがすきだとかきらいだとかは全員．

* 日時　昭和50年11月21日（金）15：30～17：30
場所　慶応義塾大学日吉第6校舎〔全体会場〕

ダイクストラ，ダール，ホアーの本は最初にダイクストラが意味，背景，哲学を書いている．次に構造的データのノートというホアーの部分があり，藤村さんのがそれに当る．最後にダールが Simula 67 のことを書いている．あそこはクラスの話があって，そこはデータの型の抽象化にも入るが，この部分は Simula 67 が主だから言語の話だと思うと，あの本のみっつの部分が，きょうの話題になっているがここにいるのは大変流暢にしゃべるダイクストラ，大変かっ幅のよいホアー，髪の毛の黒いダールである．

木村 高まいな話をということだが，うまく話せるかどうかわからない．高まいな話であるのは事実で，先に結論をいうことになるが，これは要するに，プログラミングというのはむずかしいから気をつけてやろうよということだと了解している．

高まいな哲学を，15 分でしゃべろうというのは土台むりだから，むしろななめからながめたお話しをしてみたい．

この話は充分よく理解されないうちにムードだけがバァーッとひろがった感があって，いろんなところで誤解を招いている．さっき和田さんの言及された，いわゆる I. P. T. との関係に関して，その点が特に目立つ．その中に何項目かあり (**T. P. 3**)，ここに 6 項目あげたのは 1975 年 4 月の CACM に IBM が広告をだしていてそこに，こうあがっているのだが，このほかにもたとえば Composite design などを加えたりもするようだ．ともかくその広告には，この頃 IBM の内外で開発され，使われはじめている新しいプログラミングの手法を紹介する，とある．3 番目の Structured programming が誤解のもとで，これはこの頃は Structured coding ともいっているようである．一応上から簡単に紹介すると，HIPO documentation というのはドキュメントをかくのに大きな紙をもってきて，モジュールの入力，処理，出力を簡単な絵でかき，下の方には，他とのインターフェースをかく．大きな紙に人間にわかり易い絵をかこうとするものである．

T. P. 3

Improved Programming Technologies

1. HIPO documentation
2. Top-down development
3. Structured programming
4. Team operations
5. Structured walkthroughs
6. Development support library

etc.

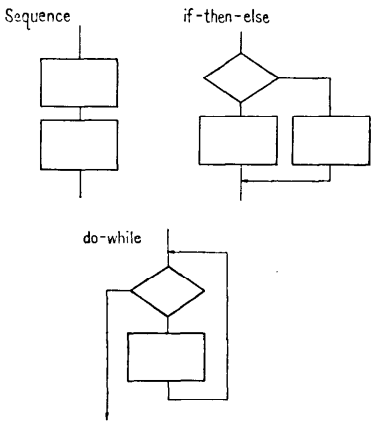
CACM, 1975. 4

HIPO とは Hierarchy＋Input-Process-Output の頭文字である．Top-down development は，さっきのダールたちの本に例題があるやつだが，問題を先につかまえて，それを段々分解してゆく過程でプログラムを設計してゆこうという考え方である．3 番目はひとまず措いて，4 番目の Team operation にゆくと，これはいわゆる主任プログラマ制という，あの話で，一人のすぐれた人をチーフにし，他の人はその人に雑用をおっつけないように，その人の雑用を全部片づけてあげるようにしようという話である．ちょうど手術の際，外科医が「メス」というメスが渡され，マスイはマスイ医がみているというふうに一人の能力をうんと発展させようという考えである．実はニューヨークタイムズのデータベースを作る際に大成功を収めたので，ご存知の方も多いであろう．Structured walkthroughs というのは，要するに技術系の人間だけが集って相談する口頭説明会である．マネージメント関係の人がいないことが大事なのだそうだ．Development support library というのは，これも例のニューヨークタイムズのデータベースでやりだしたもので，非常に優秀な秘書がいて，(これはものすごく優秀でないといけないのだが)，プログラムをキーパンチして計算機にかけるというような仕事を全部ひきうける．チームの長はプログラムは書くのだが自分では計算機には絶対にさわらない．という仕掛けになっている．この 6 項目全体で非常に能率が上がるんだという説明がされている．

問題は "Structured programming" で，これは，こういう絵 (**T. P. 4**) をどこかでご覧になったかと思

T. P. 4

3 構造



うが，この絵にあるような制御構造しか使わないという話である．この3種類に限っているといろいろ問題も生じるので，ある程度柔軟に例えばCASE構文(式の値いかんによってたくさんの文のうちからひとつを選んで実行する構文)をつけ加えたりするが，ともかくこの，構文を限定するという考えは，IPTの他の項目に密接な関係をもつ．主任プログラマというのは大変えらいわけで，上述の流儀によるとそういうえらい人がパァッと設計してしまうわけだ．そうするとあとはウゾームゾーがプログラムを書ける状態になる．それでは，というのでウゾームゾーに書かせるとすると，goto文がからみあって後で読めないようなものを作られるとかなわないので，お前達この3種しか使っちゃいかんと足枷をはめてしまうことが必要になるわけだ．もちろん，do-whileの中にdo-whileがあってもよいし，if-then-elseの中にsequenceが沢山あるということもあるわけで，階層構造でものを作っていくというわけである．これをStructured programmingといったのは大変な誤解だと思う．Structured programming (構造的プログラミング) という言葉は，正確なとこは知らないが1968年にガルミッシュの会議で，ダイクストラがいいだしたとかきいている．それをハーランミルズがいち早くとり入れていろいろ開発をしたということになっている．そうやっていち早くとり入れたとき，もとの言葉の意味がまちがって理解されてしまったのだと思う．ダイクストラのその話は精神論みたいな，プログラムは気をつけて書かなきゃだめにきまってるよという話であったように思われる．それがIBM流では，全然ちがったものの名前になってしまった．

さて次のスライド (**T.P. 5**) は，この方面で私が特に大事だと思う文献を書き並べてみたものである．まず，ダイクストラの ① Structure of the THE multiprogramming systemだが，これは非常に重要な論文だと思う．THEというのはアインドホーヘン工科大学の頭文字で，THE systemとはそこで作った紙テープベースの小さなマルチプログラミングシステムである．はじめて読んだ時は正直いって全然わからなかったが，これはもの凄く大変なことをいっている．6人で非常に短時間に，しかも大変立派なシステムを作った．ほとんど虫がいない．その時の経験が当然バックになっていると思うのだが，②がNotes on Structured Programming. それからその少しうら側を楽しめる形で話したのが，③ The Humble Programmer,

T.P. 5 (1)

① Dijkstra, The structure of the "THE"-multiprogramming system, CACM **11**: 5 (68) 341-346.
② Dijkstra, Notes on structured programming, In Structured Programming, Academic, 1972, 1-82.
③ Dijkstra, The humble programmer, CACM **15**: 10 (72) 859-866.
④ Baker, System quality through structured programming, Proc. 1972 FJCC. AFIPS **41**, I, 339-343.
⑤ Stevens et al., Structured design, IBM Systems Journal **13**: 2 (74) 115-139.

T.P. 5 (2)

⑥ Parnas, Some conclusions from an experiment in software engineering tecnhiques, Proc. 1972 FJCC, AFIPS **41** I, 325-329.
⑦ Weinberg, The psychology of computer programming, Van Nostrand Reinhold, 1971.
⑧ Mills, On the development of men and machines, Springer Lecture Notes on Computer Science **23** (75) 1-10.

この3つのものは非常に重要な文献だと思う．さき程のニューヨークタイムズのデータベースについては，④のベイカーのFJCCの論文に実質的なデータがでている．そのほかIBM System Journalにいい論文がでている．IBMのIPTの一部としてあげられることもあるComposite designのことを書いたある意味で常識的で，よくまとまった論文としてスティーブンスの論文⑤をあげておきたい．⑥にパルナスの論文，これは気をつけてものを作るときの技術を開発した話で，これはあとで藤村さんから話があるかと思う．

全然方面が違うのだが，ワインバーグの本⑦は好著である．また，ミルズがこれからやろうと思っていることを書いたというものが，あげてあるが⑧，こんなところが，是非みておくべきものだと思う．

最後に議論のタネになるかと思って書いてきたのだが，スライドをもう1枚お目にかける (**T.P. 6**). OS/360はどうやって作ったかというと人海戦術で作った．OS/360のためのドキュメントははじめはちゃんと紙に印刷したものをメンバーに配って皆それをみながら仕事をしていた．ところがあるひとつのビルの中で働いているプログラマに配られるドキュメントが，つみあげるとそのビルより高くなるぐらいになってしまい，それでマイクロフィッシュに切り換え，そ

T.P. 6

American style-OS/360
⋮
European style-Dijkstra
⋮
New American style-IPT
?? Japanese style ¿¿

T.P. 7

Languages for Structured Program (ming)

- SIMPL
- PASCAL
- BLISS
- ALPHARD
- PEARL
- CLU
- PL/I, FORTRAN, COBOL

のマイクロフィッシュを，毎日のように配って仕事をした．大変な人海戦術をやったわけだが，それはいわばアメリカンスタイルだと思う．アメリカンスタイルでワアッとやる．そうやって大変苦しんでやっと OS/360 ができ上った．デリバーされた直後は虫のかたまりだったといわれている．

それに対してダイクストラがいっているのはヨーロピアンスタイルだと思う．きちんと机に坐ってずーっとこうはじめは設計をして，どうかすると部屋の中を熊のように歩きまわって，いろいろブツブツいい乍ら考えて，きちんと書いて最後はパンチして，一発でパッと通ってもう虫はいない．そういうプログラミングをすべきだ，という発想があると思う．その辺がとてもヨーロッパ的だと感じる．では IPT はなにかというと，これはニュー・アメリカンスタイルだと思う．アメリカンスタイルとヨーロピアンスタイルが，正，反ときて，合としてでてきたニューアメリカンスタイルが IPT なのだと思う．これが充分にヨーロピアンスタイルを消化しているかどうかは私にはわからない．そこでどうしたって今度我々がしなきゃならないのはジャパニーズスタイルを開発することだと思う．これはもちろん現時点では笑い事として申しあげるが，やっぱりこれは，これから長い期間の間には一生懸命考えてゆかなければならない問題ではないかと思う．

和田 原田さんは3年前にメリーランドにいらして，あすこの Simpl という言語をご存知だから，その他の言語もついでに話して頂きたいとお願いしてある．

原田 構造的プログラミング用言語ということで調べるとずい分あって (**T.P. 7**)，Sinmla 67 は抜けているが，ここに代表的な言語を並べてみた．

データの抽象化に関係した言語については藤林さんが話して下さるので，Simpl, Pascal, Bliss, Pearl その他ここで言語の説明をする．

最初に Simpl. 移殖可能な処理系記述言語だが (**T.P. 8**)，これはメリーランドで開発された，言語の処理系を書くために使った言語である．言語仕様をきめた精神訓話はひとこともなく，ただ構造的プログラミング用といっている．設計者の意図はわからないが，Pascal にくらべて簡単になっている．procedure or function の定義はネスティングが許されない．つまりブロックを並列に並べたという感じになる．最初に global declaration があってつぎに procedure or function の定義が並ぶ．最後に START に続けて main-procedure-name を書いてしめくくる．

T.P. 8

SIMPL
Transportable Compiler-Writing Language
global declaration list
PROC or FUNC $p_1(a_1, a_2, \cdots, a_n)$
local declaration list
statemedt list
⋮
PROC or FUNC $p_k(b_1, b_2, \cdots, b_m)$
local declaration list
statement list
START main-procedure-name

SIMPL プログラムの一般形

Simpl は言語のファミリーの名前でファミリーのメンバーは Simpl 何々と名づけられている．今使っているのに Simpl R と Simpl C があって，メリーランドでは Simpl D を作っている．D はデータ定義の機能をとり入れたもので最終目標は人工知能用言語とかグラフ処理用の言語をつくることにあり，それに向って下から築きあげている．私の話は言語の文ということになるが Simpl は7つの文があるだけである (**T.P. 9**)．階乗計算プログラムはこのようになる (**T.P. 10**) (次頁参照)．上半分がひとつの procedure，下の方が recursive integer function である．Simpl の式には logical の型がなく，integer 型で 0 は false それ以外は true になっている．n≠0 は if n と書けばよく，n=0 のときは false になる．if 文や while 文では，if then や while do の次は文のリストがく

T.P. 9

SIMPL の文

1. $v := e$
2. IF e THEN S_1
   [ELSE S_2]
   END
3. [\I\] WHILE e DO S END
4. CASE e OF
   \a_1\ S_1
   \a_2\ S_2
   ⋮
   \a_n\ S_n
   [ELSE S_{n+1}]
   END
5. CALL $p(b_1, b_2, \cdots, b_m)$
6. EXIT [(*l*)]
7. RETURN [(*e*)]

T.P. 10

```
PROC P
INT I
I:=0
WHILE I<=10
   DO
      WRITE (SKIP,I,FACT(I))
      I:=I+1
   END
REC INT FUNC (INT N)
IF N THEN RETURN (N*FACT (N-1))
      ELSE RETURN (1)
      END
START P
```

T.P. 11

PASCAL

- Systematic Programming
- Algol like
- Data Structuring
- Practical

T.P. 12

if *e* **then** S_1
if *e* **then** S_1 **else** S_2
while *e* **do** *S*
repeat S_1; S_2; ⋯; S_n **until** *e*
for *v*:=e_1 **to** e_2 **do** *S*
for *v*:=e_1 **downto** e_2 **do** *S*
goto *l*
case *e* **of**
C_1: S_1;
C_2: S_2
⋮
C_n: S_n
end

ることを仮定しているから，必ず end で区切るというのが見かけ上の特徴である．

次は Pascal だが (**T.P. 11**)，特徴というと Wirth の書いた系統的プログラミングを指向し，step wise refinement の実用言語とみることができる．言語の特徴は Algol-like，これから紹介するのはみんな Algol-like だが，特徴はデータの構造化が7種類くらいできること．かなり実際に使えそうだ．言語仕様がよく整理されているのでうまく処理系ができる．教育実習にも使える．文はふつうにみかけるものが全部揃っていて (**T.P. 12**)，goto もある．

Pascal の説明書の一節によると goto というのはアブノーマルの状態になったときだけ使用せよとなっている．もうひとつ goto が入れてあるのは今まで他の言語で goto をさんざん使ってなれた人がいるからだと思う．

データの型の定義だけれども (**T.P. 13**)，はじめが scalar 型，次が array 型で memory という配列は integer 型の要素からなることを宣言している．次は，

T.P. 13

- **Scalar type**
  type color=(white, red, blue, orange, black);
  type letter 'a'..'z';
- **Array type**
  var memory: **array** [0..1000]
  of integer;
  var M: **array** [a..b] **of**
  array [c..d] **of** T;
- **Record type**
  type date=**record**
  mo: (jan, feb, ⋯, dec);
  day: 1..31;
  year: integer;
  end

T.P. 14

- **Set type**
  type primary=(red, yellow, blue),
  color=**set of** primary;
  var C1: color;
  C1:=[red]; C1:=C1+[yellow]
- **File type**
  var date: **file of** integer;
- **Pointer type**
  type link=↑person;
  person=**record**
  ⋯
  next: link;
  ⋯
  end

T.P. 15

BLISS: A Language for Systems Programming

- Machine Oriented Higher Level Lang.
- Algol/PL/I like
  block structure, conditional &
  looping structure
- Interpretation of Names
- Expression Oriented
- Accessing Algorithm-structure, map
- Typeless Data

Cobol にある階層構造をもったデータ型に相当する record 型である．date という型は日にちと月と年からなることを定義している．month(mo) というのは January から December までというようなことでそのフィールドの実際の型を示している．

あと他ではあまりみられないのが set 型で (**T.P. 14**)，primary は red, yellow, blue のいずれかを値とするデータ型とする．このとき，color は primary を base とする power set を定義したことになる．[ ] で書いてあるのは，要素を color の set に加えることを意味している．

このほかに file, pointer などがある．

次は Bliss 11 (**T.P. 15**)．特徴は machine oriented higher level language．機械にデペンドした言語で，見かけは Algol または PL/I like である．名前の前

T.P. 16 (1)

Control Expressions

1. Conditional Expr's
   IF E_1 THEN E_2 ELSE E_3
   IF E_1 THEN E_2
2. Loop Expr's
   WHILE E_1 DO E_2
   UNTIL E_2 DO E_2
   DO E_2 WHILE E_1
   DO E_2 UNTIL E_3
   INCR name FROM E_1 TO E_2 BY E_3
   DO E_4
   DECR name FROM E_1 TO E_2 BY E_3
   DO E_4

T.P. 16 (2)

3. Escape Expr's
   LEAVE label WITH E
   EXITLOOP E
   RETURN E
   LEAVE label
   SIGNAL E
4. Choice Expr's
   CASE E OF
   SET E_1; E_2; ···; E_n TES
   SELECT E OF
   NSET E_1: E_1'; ···; E_n: E_n' TESN

に content operator というドットをつけるのが，みたときの大きな特徴だ．あと素のデータはビットの集りがあるだけで，とくに型というものはない．そのために浮動小数点用の演算子やビット処理用の演算子が用意されている．丁度 Bliss でプログラムを書くときはアセンブラで書くときのようなことを頭の中で考えていることになる．

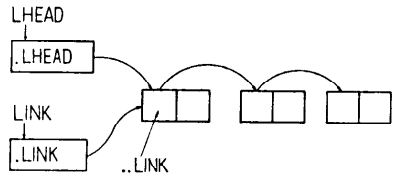
プログラムが expression によって構成されるというのも特徴である (**T.P. 16**)．通常の while, until の他に，繰り返しの終了条件をひっくりかえしたものもある．また increment, decrement の繰り返し文がある．これが goto 論争の escape expression で，いろいろな条件によって脱出するものが用意されてある．あとはふつうの case 文である．

content operater の例をあげると (**T.P. 17**)，ここに線形リストがあったときに，Lhead がリストの最初の要素をさしているとして，最後の要素をどうしてみつけるかというアルゴリズムはこのように書ける．name はストーレージに対するポインターと解釈できる．普通は代入文の右辺に名前をかくと右辺の名前は「内容が」ということになるが，内容をドットであらわす．ドット，ドットは間接番地指定になる．

配列は Structure なる定義を用いて (**T.P. 18**) あるアクセス・アルゴリズムを特定の記憶域にマップすることによって実現する．添字式の計算のアルゴリズ

T.P. 17

```
LINK=.LHEAD;
WHILE..LINK NEQ 0 DO
   BEGIN
      LINK=..LINK
   END;
```



T.P. 18

```
BEGIN
   STRUCTURE ARY2 [I,J]=
      (.ARY2+(.I-1)*10+(.J-1));
   OWN X[100], Y[100];
   MAP ARY2 X; Y;
      ⋮
   X[.A,.B]=.Y[.B,.A];
      ⋮
END;
```

Structure, Map の例

T.P. 19

PEARL Snowdon 1971

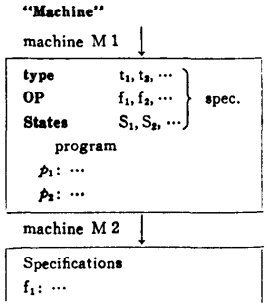
Program Elaboration And Refinement Language

- To provide an environment for the writing of correct programs
- Facilities for the construction and filing of structured programs
- Teceniques for the inclusion of assertions involving abstract op's and data types

ムをこのように Structure として定義する．Own X は単に X として 100 個のデータ，100 話の場所をとってくれというので，配列ではない．そこでアクセス・アルゴリズムと名前を Map によって結びつける．使う方は x[.a,.b] と書くと X はアクセス・アルゴリズムに結合されているから，定義通りにそれらが分解されて配列要素の取扱い可能となる．

次は Pearl (**T.P. 19**)，ダイクストラの本に真珠の首飾りという話がある．それにちなんで Pearl という名がつけられたと思うが，Pearl はプログラミング言語とみるより，構造的プログラミングを支えるためのシステムとしてみた方が正しいかもしれない．これはまだ試作中で，完成されたものではない．目的としてはヴァーチャルマシンを作って，あるモジュールを作り，そこに用いた命令を実現するヴァーチャルマシンを次のレベルで組み立て，真珠の首飾りをつくろうとするものである．実際にはどんなことになるかというと，ひとつのモジュールというのはデータ型，オペレ

T. P. 20



ータ，states の宣言とプログラムからなる (**T. P. 20**). 枠内がひとつのヴァーチャルマシンで，その中の下の方がプログラム．プログラムの部分では上で宣言した型とかオペレーターを使って記述をする．次のレベルでマシン M_2 を考え，前のレベルで定義したオペレーター f_1 は実はこれというわけで，上位のものから下位のものへと，計算機と会話型で作業をしながら，段々とこまかくしていってプログラムを完成させる．ということでダイクストラの考え方をそのままとり入れたシステムである．

和田　最近のプログラム言語はどれも構造的プログラミング用言語と称しはじめたので，きりがない．これがどうして構造的プログラミング用言語かしらと思うのがずい分あったが，そういう批判はあとで伺う．

今度は藤林さんにデータタイプアブストラクションの話をおねがいしたい．藤林さんは，カーネギーメロンのウルフのところにいらっしゃって，ウルフが，構造的プログラム用の Alphard という言語を設計している．そんなのを話してほしい．Alphard というのは，星座 Hydra の主星の名前なのだそうだ．

藤林　抽象化というのは構造的プログラミングの議論の中でかなり重要な考え方のひとつだということがいわれているが，プログラミングの世界で抽象化というのはどんなことか，簡単にいえば，ある問題をある時点でといているときに，本当に必要な詳細だけがわかればよい．その時点で，そのレベルでくわしすぎるような，不適切な詳細事項はみせないように，知らなくてよいようにしよう．そういう考え方ではないかと思う (**T. P. 21**). 例えば，あるものを使うとき適切な詳細として，それはなにができるのか，どのように使うのかがわかればよく，それがどんなふうにできているのか，というのは使おうとする側ではいらないわけだ．むしろ知りすぎて害になることがある．そういう

T. P. 21

プログラミングにおける**抽象化 (Abstraction)** とは，
適切な詳細（何をやるか）を表現，不適切な詳細（いかに実現するか）を隠す．

抽象化のツール

- **機能，手続きの抽象**
  サブルーチン（手続き），関数，マクロ
- **データの抽象**
  ?

考えがダイクストラ達のいっている構造的プログラミングの思想の中にあると思う．適切な詳細をうまく表現して，不適切なことは隠すということがプログラミングの世界でどんなふうに行われているか考えてみたい．

ひとつは我々が今までしばしば行っていた機能，手続きというものの抽象化である．これはすでに大抵の言語でサブルーチン，手続き，あるいは関数，マクロ，といった形でサポートされている．ところが問題を解くときは，機能，あるいは手続きの抽象化と同時にもうひとつの非常に大きな道具，考えをあらわすものとしてデータがあるが，そのデータをどんなふうに扱っているだろうか．機能あるいは手続きの抽象化は割とやられているようだが，その中で使われているデータはプログラム言語で表現されるような配列だとか，integer, real といったレベルでいきなりでてきて，表現レベルのアンバランスがおこる．元来，問題を解く上では，問題領域に現われる概念に近い表現で，機能の抽象化と同様に同レベルのデータの抽象化を行うことが，このアンバランスを避け，自然な無理のない展開に大事なわけだ．そういったデータの抽象化をプログラム言語でどうサポートしているか見る前に，まず抽象的なデータの型のとらえ方を考えてみたい．

従来考えられているのは値，value というものの集りであるデータ構造として扱うことである (**T. P. 22**). データがどういう集りぐあいになっているかを表現することをサポートするのが多かった．ところがデータというのは集めたときに何故集めたのか，集めた理由が大事なのである．また構造があってもその構造にな

T. P. 22

抽象データタイプのとらえ方

- 値の集まり⟶データ構造
- 作用 (operation) の集まり

例:

ALGOL 60
- **boolean**
  - 値の集まり　{**false, true**}
  - 作用の集まり　{**and, or, not**}

stack の抽象
- — 配列やリスト構造で表現
- — push, pop, top

T. P. 23

抽象データタイプの扱い方の規準

① データタイプ定義は，そのタイプの object に関連するすべての情報を含むこと．
② 抽象データタイプの使用者には，そのタイプの object の表現（実現）方法を意識させないこと．
③ 使用者はそのタイプの operation を通してのみ，object を操作できる．（直接その記憶域表現を操作しない）

にか操作あるいは作用するといったことが当然でてくる．これらは，結局扱おうとしたデータの性質を表わしている．そういうことで値の集まりということだけではなく，作用の集りもデータに対する大切な特性である．例えば Algol 60 の boolean という型は false と true という値の集まりからできる性質を表わしている．また boolean に対しては and, or, not という作用が，与えられている．これらがそれぞれ言語で別別に定義されている．つぎに例えばスタックというような抽象化されたデータ，あるいはデータ型を問題を解くときに使いたい．この場合例えばスタックとは，配列とかリスト構造とかで作られるであろう．それはスタックというもののデータの集り，構造の実現方法を表現しているのだが，こういうスタックに対する性格づけと同時に例えば push, pop だとかスタックに対して何らかのオペレーションをもってスタックに対するデータの抽象とするわけである．そのとき抽象的データ型をどうプログラム言語で扱うべきなのか，考え方，扱い方の規準を少しあげてみる (**T. P. 23**)．①データ型の定義はその型のオブジェクトに対するすべての情報を含むこと，ここでいうオブジェクトはひとつなにか抽象的な型ができたとき，それが実際に実現されたものをよぶとする．いま抽象化しようとしているデータをあらわすすべての性質，例えば先程いったような値の集りとしてのデータ構造，表現もそうだし，いろいろなオペレーションも集める．これらをひとつのデータの抽象という形でみんな集めてしまうことが大事である．つまり，例えばデータ構造だけを指定して作ってやる，それに対して演算はどこか別の場所で定義する．そういうことをバラバラにやるとその間のいろいろなインターフェースがちらばることになる．これは，プログラムのモジュール化ではインターフェースができるだけ少ない方が考え方が楽になるということに反するわけだ．つぎに②抽象データ型の使用者，つまり抽象的なデータ型が作られたときに，それを使う人にはその型のオブジェクトの表現または実現がどんなになっているかを意識させるなということである．あまり細かいことを知りすぎるとかえってそれが余分な情報になって本来の正しい使い方かどうかわからなくなる．あるいは細いことが分りすぎて，それをチョコチョコとさわってしまって大事なデータの概念をこわすということがおこる．ユーザにはどういうことができるかだけ知らせ，中がどう作られているかを意識させない．このようにするともうひとついいのは実現の方法が使われ方と独立に型を効率よく実現するよう工夫できるわけで，またある条件，例えばコンパイラがかわって，実現の方法がかわったとして，使っている方の状態は影響をうけないということが可能になってくる．それから③それと関連するが，使用者は型のオペレーションを通してのみデータの型のオブジェクトに操作できる，実現を直接みてそれを使うということをしない．実現と機能を切り離すのが大事な考え方になるのではないかということである．

つぎにこのデータの抽象化を既存の言語でどのようにサポートしているか見てみたい．既存言語の中で特にデータの型を，例えばデータの構造を型として定義できるような言語に限って話をする．そういった既存の言語としては Algol 68, Pascal, Simula 67 などが代表的だと思う (**T. P. 24**)．Pascal は言語のところで紹介があったが (**T. P. 25**) 簡単にいうと，**type** で person という型が定義でき，person は **record**…**end** の構造をもつレコードと定義できる．この型に対して person という名前がつけられ，それ自身には記憶場所がなく，オブジェクトとして存在していない．これ

T. P. 24

データタイプ定義のできる既存言語

- ALGOL 68
- PASCAL
- SIMULA 67

新しいアイデアの定義

- CLU の cluster Liskov (MIT)
- ALPHARD の form Wulf (C-MU)
- Concurrent PASCAL の process, monitor, class Brinch Hansen (CIT)

T. P. 25

```
PASCAL
    type person=record name: string;
                       age: integer;
                       sex: (male, female)
                  end;
    var John: person;
    John. sex:=male;
ALGOL 68
    mode person=struct (string name,
                        int age,
                        bool male);
    person John;
    male of John:=true;
```

をデータとして定義するには **var** という宣言のなかで，John が person のデータ型だよと結合する．このように integer とか real とかの通常の型同様に使える．John・sex は宣言されたデータの sex というセレクターで要素をとりだす使い方である．Algol 68 も同じようで，例えば **mode** という機能で person というのをひとつの型として名前をつけ，いまのレコードと同じ構造を **struct** なる構成要素を使って定義する．

既存の言語でもうひとつ代表的なものに Simula 67 があるが (**T.P. 26**)，Simula 67 の **class** という言語構成要素を使うと，今と同様に person というクラスとしてデータの構造が定義できる．これらの言語は主としてデータ構造を型として定義できるが，最後の Simula 67 は，データ型としてのデータ構造の定義のほかに，データ型の考え方でいったオペレーションをあわせて定義できるといういいところがあって，先にのべた考え方の規準に近くなっている．例えば stack という抽象的なデータ型を考えたときに (**T.P. 27**) この中で stk というのが物理的に実現している．また push, pop といった手続きとしてそれに対するオペレーションが定義できる．もうひとついいことにはこの **class** stack というデータ型が，オブジェクトとして作られるときに top:=0 のような初期化の処理を **class** の中で記述できる．しかし **class** で囲った中の内部データが外側の世界にみえてしまって，それが直接触れることになり，オペレーションを通してのみ触らなければいけないという規準からみるとうまくない．そういう問題点があるので，最近新しいアイディアがでてきた (**T.P. 24** (前頁参照))．

M.I.T. のリスコフは Clu という言語を提案し，その中でデータ型を定義する cluster という機構を実現しようとしている．それから C.M.U. のウルフは Alphard という言語の中で form という cluster に似た機構でデータ型に限らず，そういう抽象化を記述できるものを考えている．またカリフォルニア工科大学のブリンク・ハンセンは Concurrent Pascal という言語で，これは Pascal をパラレル処理の扱える言語に拡げようとしているのだが，この中で process, monitor, class という概念の機構を使って同じようなデータ型を実現しようとしている．あまり時間がないのでどんな好恰をしているか雰囲気だけをちらっとおみせしたいと思う．

Clu という言語の cluster では (**T.P. 28**)，例えば stack というのは cluster の構造の中で **is** (**T.P. 29**) というインターフェースで，どういうオペレーションが使えるか，push, pop, empty とか……を書く．**rep** では cluster の中で stack がどんなデータ構造で実現されているかを定義している．**create** (**T.P. 30**) (次頁参照) では stack がオブジェクトとして作られるときに，初期処理をおこなうコードがかかれている．あとはオペレーション pop, push など定義されたオペレーションを実現するプログラムがかかれている．これ

T.P. 26

```
SIMULA 67
    class person;
        begin text name;
              integer age;
              boolean male;
        end;
    ref (person) John;
    John:-new person;
    John. male:=true;
```

T.P. 27

```
SIMULA 67 の class
  class   stack (size); integer size;
  begin   array stk [1: size];
          integer top;
          procedure push (X); real X;
            begin   top:=top+1;
                    if top≦size then stk [top]←X
            end;
          procedure pop (X); real X;
            begin…
            end;
          top: =0
  end of stack;
  ref (stack) S1;
  S1:-new stack (100);
  S1. push (2.5);
```

T.P. 28

```
Cluster の定義
stack:  cluster (…)
          is push, …; インターフェース記述
        rep (…)=…;  object 記述
        create      object 生成定義
          …
          end
        push: operation (…); operation i の定義
          …
          end                    ⋮
          ⋮
        end stack
```

T.P. 29

```
1.  インターフェースの記述
    stack: cluster (element-type: type)
           is push, pop, top, erasetop, empty;
2.  Object 表現の記述
    rep (type-param: type)=(tp: integer;
                            e-type: type;
                            stk: array [1..]
                              of type-param);
```

T. P. 30

```
3. Object 生成の定義
   create
      S: rep (elemement-type);
      S.tp:=0;
      S.e-type:=elemeet-type;
      return S;
      end
4. Operation の定義
   push: operation (S: rep, V: S. e-type);
      S.tp:=S.tp+1;
      S.stk [S.tp]:=V;
      return;
      end
```

T. P. 31

```
form の例
   form stack (y: form)=
      {decl unique top: int, stk: vec (100, y);
       initu:: top←0;
       fcn push (x: y)=
          begin top←top+1;
                if top≤100 then…
          end;
       fcn pop: y=…;
       export push, pop;
       sequence:: [{push; pop}]
      }
```

T. P. 32

```
form complex=
   {decl r: real, i: real;
          ⋮
    export r⟨read⟩, i⟨read⟩
   }
```

らのオペレーションはここで実現された **rep** の構造を直接使うことができるが，外側から cluster のデータを直接触ることはできない．ということで，使用者と中の実現の世界をきりはなした．

Alphard に form という機構があるが (**T. P. 31**), form で同じように stack を書くとする．**decl** で cluster の **rep** と同じように内部で実現するためのデータを定義する．**initu** でデータに対する初期化を記述する．それから function をいろいろ書く，Alphard の中のひとつの重要な言語の哲学として，プロテクションの重要視がある．いかにして外にデータを見せなくするか．見せる場合どのように許可するか．**export** は外部からアクセス可能なものを定義している．function 名だけでなく，よく使われる変数を外部からアクセス可能とする場合も export で指定する (**T. P. 32**)．例えば変数を外部に使わせる場合画一的にグローバルにしてしまうと非常に問題になるのである程度制限をつけようという考え方だと思う．**sequence** はまだアイディアだけで，言語の構文がきまっているわけではないが，ここで定義していることは，オペレーションの正しく使われる順序である．

T. P. 33

```
Monitor の例
type diskbuffer=
   monitor (consoleaccess, …: resource; …);
      var disk: virtualdisk; …
          sender, receiver: queue;
             …
      procedure entry send (block: page);
          begin
             …
          end;
      procedure entry receive (…);
          begin
             …
          end;
      begin "initial statement"
          init disk (…);
             …
      end
```

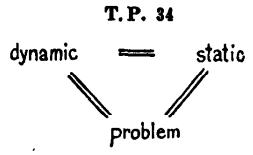
最後に Concurrent Pascal. これには process, monitor, class の一種のデータ型を記述する要素があるがその中で monitor (**T. P. 33**) の例を示す．考え方としては Simula 67 の class と同じようにデータ型を記憶場所とそれに対するオペレーションで書く．monitor には並列処理をするプロセスの同期がうまくとれるような，シェアされるデータがあってそれを中心にオペレーションが考えられていて，そのオペレーションの並列処理の同期をとろうということを考えている．

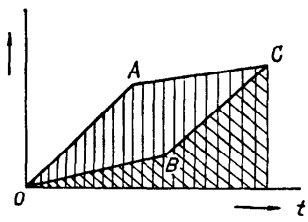
monitor がアクセスできるものが最初に書かれて，monitor を実現するシェアード変数，sender receiver が定義され disk buffer という monitor に対しては send, receive というオペレーションがある．monitor が実現されて動きだしたとき初期処理するプログラムが init に書ける．

新しいアイディアとしてできているものは，データの集りとして構造を拡張できるとか，データの型として定義できるというばかりでなくて，そういった構造に対するオペレーション，作用というものを含めて定義してやらなくてはならない．またひとつのまとまったものとして定義してやることによってインターフェースを少なくし，使用者に不必要な情報をかくすことができるというかなり徹底した考え方があると思う．

和田 いま藤林さんは話さなかったがこしの日米コンピュータ会議の紫合さん，下村さん（日電）などのお話でグループというのがあった．それもこの種の話だ．

構造的プログラミングの考え方は動的な構造とプログラムの静的な構造とをなるべく合わせるのだという





のである．つまり動的と静的な構造をなるべく合わせるようにするわけで，goto 文があるとこれが狂うからいかんというのが，ひとつの根拠だったと思うが，私の思うには，これに対して問題の構造ともこれを合わせなければいけない (**T.P. 34**)．結局プログラミングとは問題の構造をみつけることではないか，読めば読むほど，プログラムを書けば書くほどそう思う．またもうひとつ，プログラムはなるべく速く作れということとなるべくむずかしいところからやれということを申しあげたい．速くやれという理由はダイクストラのものを読むと構造的プログラミングというのは人間の頭蓋骨の中にちょうど入る程度の情報の中でプログラムを書かなきゃいかん．それより大きいものはどうしても二次記憶を読んでくるのだから，うまくいかないというわけで，それで小さく作れというのだが，時間が長くなるとやはり頭蓋骨のなかに入り切らないのだからなるべく速く頭蓋骨のなかからなくならないうちにプログラムを書かなければならないと思う．それでむずかしいところから作るという方は横軸が時間，縦軸が覚えていなければならないものの量である．プログラムというのは書きながらどんどん忘れてゆけるかというと，そうではなく，構造的プログラミングというのは少しだけ覚えていればいいようなものの実はいろんなことを覚えていなければいけない．覚えてなければいけないものが段々ふえてくる．最後にプログラムができあがるとストンとおちるが途中は図のようにふえてゆく，そこで時間がたつと図の面積を覚えていなければいけない．つまり一時期はこれだけだが，あるプログラムをかくのに対して頭から熱が発散する量が図のハッチの面積である．点 A で急にむずかしいところになったらどうなるかというと，ここから急にプログラムが進まなくなり覚えなければならない量のふえる速度もへってくる．場合によると平になるし，場合によると段々忘れてゆくので下ったりする．しかしいずれにしてもどこかでむずかしくなると平になって，こういう形で，頭がカッカとしてくる．大体トップダウンに作ると，最初は **begin** initialize; compute; finalize **end** と書けばいいので O の辺はすごく簡単に書けるのだが，そんなことやっているうちににっちもさっちもいかなくなる．一方むずかしい所を最初にやると，最初は $O \rightarrow B$ とおそいスピードであがる．その辺が全部できると易しくなるから $B \rightarrow C$ と速いスピードであがりだす．覚えなければならないものは最後まで同じ，そうするとこの方針ではカッカとしている量は $O \rightarrow B \rightarrow C$ だけである．$OABC$ だけエネルギーがセーブできたではないか，簡単にいうとそういうことになるのだ．そういう調子でプログラムはむずかしいところから作ろう．例えば Pearl の話のときに，最初の方にデータがあり，下の方にマシンの命令があって，原田さんがダイクストラの考えと逆だといった．ダイクストラは最初にデータをおきあとから命令をおくのは馬の前に馬車をつけるようなものだと書いていた．それに対して私がザコパネのシンポジウムで反論したのはミニコンのプログラムを作るときには，私はデータの方を先にきめる．なぜかというとミニコンではデータをつめなければならずその部分はむずかしい．それで，先にうまくつまるように考える．あるいはコンパイラーを作るときはオブジェクトコードをどう作るかの方がいまではむずかしい．構文解析はもう易しくなっている．だからオブジェクトコードを先に設計する．そういうことをやるのではないかと思う．私の構造的プログラミングの解釈だが，「な」の方，どういうプログラムを作るかというと構造が問題と合っているように，どういうふう「に」作るかといえば，むずかしい方から先に作る．そういうものと思っている．

五十嵐(京大)　そもそも構造的プログラミングの目的はなにかである．ダイクストラが最近いっている公理主義という面はどう扱われているか，木村さんに伺いたい．藤林さんのいわれたことには最初の抽象的な説明の部分では同感だが，後半になったら本質的な部分はとんでしまって，ただ昔と同じレベルの言語仕様に終始されたと思う．

木村　構造的プログラミングの目的はなにかという質

問だった．

ダイクストラかがその点について何を考えているかは知らないのでお答えできない．お前自身の目的はどうかといわれれば，それはこういうことだと思う．とにかくプログラミング活動が自分でもすきだし，いろんなソフトがほしいので，他人と協力して大いにやろうと思うのだが，やってみると非常にフラストレーションを感ずることが多い．よく他の部門の偉い学者の方で，ソフトなんて頭がよければできるので，ソフトの奴らは頭がわるいからできないんだ．などとおっしゃる方があるが，実はとてもそんなものではない．自分でやっていない人にはどうもそう見えるものらしいが，やってみるとうまくいかない．確かに頭のわるいせいもあるとは思うが，本質的な何かがそれ以外にもありそうで，そこのところを何とかしたい．それが私の発想であり，そこを何とかするのが目的だと申しあげておきたい．

藤林 データの抽象化についてだが，実際の言語の例を話した段になって非常に抽象化でないということだが，私の考えでは，いままでやっていた方法はただデータの構造を配列からリスト構造とか構造体とか構造体のくみあわせというハイヤーレベルのデータ構造まで表現できるようになったということしかいってなくて，実際にはそういうものと，そういうものがどう操作されるかということが，わかってデータという姿がでてくると思う．そういうものがいままでの言語では，データの構造と手続きとして独立になっていた．これはデータの抽象化でなく，プログラムの中で構造を操作しているだけである．そういう形ではひとつのデータとしてまとまって考えられないとか，フィジカルな構造が表にでてくることで不必要な詳細を知りすぎて間違ったことがおこり易いことがでてくる．それが情報ハイディングとしていわれている．問題をといてゆくときに当然でてくるデータは，静的，動的の構造を合わせるという話と同様に，問題の構造に対応させようと和田さんの発言があったが，そういうものと同じことで，問題のレベルでのデータが直接扱える方がよく，それはただの構造だけでは表現できない．オペレーションがひとつのまとまったものとして定義されはじめてできる．そういう意味でデータの抽象化はかなり新しいアイディアとしてサポートされていることを話した．ひとつ残る問題はそういうデータの抽象化をするのに，いかにスペシフィケーションを与えるか．オペレーションや構造を作って書いてもどんな性質のものなのかという仕様を記述する段になって仲々うまく記述できない．研究はやられているが，そういうことは正当性の証明などと同様仲々手がまわっていないという感じだ．実際にプログラムを作っている感覚からすると，今日話したデータの抽象化は，例えばモジュールを考えたりするときにただ機能をわけてゆくというふうなことの中でデータ，リソース，デバイスなどに注目してそういうものの構造とその振舞い，オペレーションを対応づけてモジュールとしてゆくとか，ひとつのモジュールの中に沢山の手続きで実現できるのではないかという感覚が強い．インターフェースを少なくして複雑になる要素をへらすことでかなり実際的と思っている．

和田 きょうここにおいでの方で，構造的プログラミングは役に立ちそうだと思う方は手をあげてほしい．次にこれはだめだと思う方は，手をあげてほしい．だめだという側の国鉄の稲田さん，理由をいってほしい．

稲田(国鉄) 実は私両方手をあげた．それは皆さんの話をきいているとなるほど立派だと思うし，こうやらなければいけないと思う，という話になってくるが，私3年ばかり泥くさい仕事をしていて，新幹線のコンピュータ・プロジェクト2年ばかりの間に700kステップのものを100人ばかりで，…それでそれを3月10日に終ってさてということで若い人とずい分苦労したが最近構造的プログラミングというものがあるというので反省してみようといって見だしたが，さきほどいくつかの点は和田さんからも木村さんからも出たが，一番大きな問題はやはり動的構造と静的構造を一緒にしたいということ．たしかにそうだが，オンラインのシステムは，一致しないという点がでてくる．ひとつは現在のコンピュータの一番の特徴であるといわれている割込みに原因があるのではないかと思う．これを和田さんにおききしたい．むずかしいところを先にやるという話で実際にいろいろやってみるとオンラインシステムはある容量の範囲に収めなければならない，ある種のレスポンス時間に収めなければならないということになると，データレコードにしても例えば一番簡単なのはオブジェクトに対してアトリビュートをまとめておくのだが，そうするとどうしてもある処理形態からいってデータが全部コアに入らないとかで構造になりにくいのではないか，そういう意味で，私の意見は両方手をあげたのは，そういう形になる部分もあるし，もう少しコンピュータの本質的，ハードウ

ェア的なところを直さないと，あるいは考え方をもう少し整理しないとならない部分があるのではないかと感じる．

和田　わずかに答えらしいことを考えているのだが，ここに静的，動的と書いたけれど，これ必ずしも静的，動的ときまっているとも思えない．見方によっては静的なものも動的だし，動的なものも静的である．例えば新幹線の中に坐っていると，静的なような気がするがコーヒーでも飲もうと思うと，動的だなァという気がする．Concurrent Pascal は並行に走っているものを何とかしてできるだけ静的にしようとしているのだと思う．monitor, critical region だとかああいう種類のものは，そうだと思う．でやはりそこは考え方ではないかと思う．それと，いろいろなところで書いたり話したりしているのだが，構造的プログラミングというのは，私はさっき司会だったから手をあげなかったけれど，なにに役にたつと思うかというと，むずかしいプログラムを書くのに役に立つと思う．易しいプログラムは SP でもやっていると書けるだろうが，むずかしいプログラムはよくよく考えないとできない．場合によると，構造的プログラミングでできたプログラムを見ると，非常に能率がわるいのではないかという気もするが，それはまだ構造的プログラミングが最後までいっていないのだと思う．むずかしいプログラムを書くのが構造的プログラミングで，オンラインのむずかしいのをなるべくわかり易く考えて，あるリスポンスタイムの中で片づくように考えるのが，構造的プログラミングではないかと思う．それで構造的プログラミングは，3月10日までに博多開業とか，3月15日までにひかりは西へとか，そういうのを先にきめるとむずかしいことになる．

木村　それでもプログラムは書かなければならないという話になるので，いろいろ議論が起るのだと思う．それはさておき，割込みの扱いというのはやはりむずかしいわけで，その原因のひとつは，どうやって扱ったらいいかがまだ充分よくわかっていないところにあると思う．よく和田さんが引き合いに出される話だが，コンパイラ作りは 10 何年前にはものすごくむずかしかった．Fortran I のコンパイラは，作るのに 50 人年かかったといわれている．いま Fortran を 50 人年かけて作ったりしたら多分ソフトウェア会社はつぶれるだろうと思う．コンパイラ作りの技術はもうみんなの財産になっていて，教科書を読むと書いてあって，その通りにやるとできて，少し上手な人がやるとすごくいいのができてという世の中になっている．学部の学生でさえ，まがりなりにも動くという程度のものなら作れる時代が来ている．割込みについてもいつかそういうふうにみんなの考え方が整理される日がくると思いたい．そういう日を来させるためには，さしあたっては現実の問題に則して一生懸命考えてゆかなければいけないのだと思っている．アーキテクチャをかえなければしょうがないのではないかという話には私は賛成で，そのうちどこかでハード屋さんにソフト屋さんの気持ちをわかってもらうにはどうしたらよいかという問題を考えるシンポジウムをやりたいと思っている．どなたか御賛成の方はないか．

和田　割込みというのは僕の考え方では，非常に静的でよく整理された考え方だと思う．割込みができる前はひとつのプロセスがしょっちゅういろいろ調べて，あっちへとんだりこっちへとんだりしていたが完全に割込みで走るプロセスと割込まれるプロセスにわけて，割込みを無色透明にしてしまったと思う．そこの考え方を整理すれば割込みはうまく制御できると思っている．それはそのくらいにしよう．

後藤(東大)　割込みもそうだが，エラーの処理，ディバッグ中に生じるエラーではなく，藤林さんのスタックの沢山の言語の例題で，あの簡単な例でいつでも問題になるのは push してはみだしたらどうするのだということ，それが簡単な例で，プログラム言語ではそういうことの記述が不充分ではないかと思う．ミンスキーはデモンといって，デモンはああいうところには書かないけれどみている．非常事態のときにでてくるように構造を作るべきだという考え方である．スタックという簡単なものでも異常事態，あるいはプログラム言語で書いたときも異常状態を考え忘れているのはないか，ディバッグでやると，異常状態をすべて発生させてみるというのではなくて，やはりもっと機械によってベリフィケーションとちがうと思うが考えおとしたチェックすることを考えるべきだと思う．

和田　今度は役に立っていると手をあげた方に伺おう．

佐々木(日電)　構造的プログラミングは個人的な方法ではなくて，これからは組織的レベルでまとめる方向でなければならないと思うが，その場合，構造的なプログラミングをする方向として言語で規制するのがひとつの方法と思う．構造的にプログラミングするには，組織的レベルでどういう規制をしたらよいか，どういうことがやられているか，というようなことをお

ききしたい．

木村　直接規制するのはむずかしいと思うがたとえばIBM の中では，大きいことをやろうとして当然やっていると思う．和田さんがいっているのは，ものすごい話で，少なくとも現時点ではあのくらいの頭蓋骨でないとできないかもしれないと思っているが，しかし頭蓋骨の大きさはとにかくとして，つきつめた考え方は 10 年たつと当り前になっていることが多いので，あっちの方向も大事と思う．もちろんおっしゃるようにオーガニゼーションをよく考えるというのも大事であろう．ただ，大きな問題の大きい理由というのは沢山あって例えば，これは夏のシンポジウムであった話だが，客先の要求が予測できず，システムを作る方は恐しくてたまらないから，これでもかこれでもかと機能をいっぱい入れてしまう，そのために大きくなるという要素がある．OS/360 が大きくなったについては，かなりそういう面があったと思う．システムをお鍋のなかみにたとえると，ブクブクと泡がたっていて，ずいぶん沢山入っているようだが，火をとめればずーッと下って行ってしまうというようなかたむきがないだろうか．こんなに大きいから，うんと大きい箱に入れないと入り切らないと思って，大きな大きな箱に入れている，というような面がある，と思う．もうひとつの要因は，大きいものを作るために大勢で仕事をする，というところにあると思う．その場合，人間同志のインターフェースが問題になる．そのインターフェースを，安全をみてか重く作るので，全体がますます重くなる．それでは人間をへらせないかと思うと主任プログラマチームは明らかにそれで，これはわりに肝心なところを一人のところにおしこみ，肝心でないことはないが肉体労働に類するところを他の人にもやれるようにして，人と人とのインターフェースを少なくしようという考え方であるといえる．

それからもうひとつはプログラミングの道具，例えば非常にすぐれたテキストエディタができてきたとき，能率があがるということがある．希望的観測にすぎるかもしれないが，最初のお鍋は火をとめてずっと小さくなる．道具をよくしてまた半分にする，チームをうまく作ってさらに半分にしていくということをしていくと，まァ案外うまくいく面もあるのではないか．もちろん本質的に大きい問題もあるわけで，だから常にそうだといえないが，いろんなことを考えなければいけないのだと私は思う．

藤林　作られるプログラムにはいろいろなものがある．一回動かせばそれでいいというのと，国鉄の例というのもある，だからあるひとつの方法論なり作り方がすべてに適用できると思うのはおかしい．構造的プログラミングは非常にむずかしい，あるいははじめから問題をほぐして考えてゆかなければならない，そういう問題にアタックするのによいのではないかと思う．

原田　まとめになるかどうかわからないが，プログラムを作るには目的，規模によっていろいろ作り方があって，ひとつのやり方ですべてのプログラム，例えば大学関係だと教材としての数ステートメントくらいのものから，いわゆるメーカで作る何万ステップのものも一様にできればよいが，そうはいかない．私自身はいざプログラムを作ろうとなると，必ずしも世の中の教科書通りにやっているわけでなく，あるところは構造もへったくれもなくて，どうでもいいから書いてしまうということもある．時間的，精神的にゆとりがあればちゃんと構造的プログラミングで，ダイクストラの本のどの辺をやっているんだとプログラミングしながら思っていることもある．トップダウン的考え方は少なくともそれを意識してのプログラム作りでそれほど害はないというふうに考えている．

和田　原田さんのいわれた構造的プログラミングということは知っているが，そうしないこともしばしばある，というのは割合いと私にとっても真実だが，これは東京女子大の水谷先生のいわれたことだが人間の言語知識と言語活動とはちがうことがある．という．言語はこうなっているということを説明する先生でもちがうような話し方をされることもあるらしい．言語というのは案外そんなものかとも思う．それで構造的プログラミングでプログラムを書いたから，プログラムがよく書けたという人もいるのだが，一説によるとそれは構造的プログラミングを知らなくても上手に書ける人なんだという．つまりダイクストラはああいうことをいう前からプログラムはうまかった．またダイクストラのああいう講義をきいた学生でも，そう上手に書けないのがいる．だから構造的プログラミングはそれを知っていればプログラムがうまく書けるというお守りでもなんでもない．従って構造的プログラミングというのは，うまくプログラムを書く努力をしなければならない．というしつけみたいなもので，一生懸命プログラムを書くより仕方がない．プログラムというのはことしの5月の NCC にプログラム言語をどう設計するかというセッションがあって，ワッサーマン

がいっていたことだが最近わかってきたのはプログラムは非常にむずかしくプリンシプルとスキルとアートと一緒にアプライしなければならないむずかしいものだとかいていたが，そういうものだと思っている．結局申しあげたいのは構造的プログラミングはそういう意味では直接は役にたたない．どちらかというと上手な人はプログラムをどう書いているかということが，構造的プログラミングではないかという気さえする．むしろうまいプログラムはどうなっているかというのをしょっちゅうみて反省する，なるべくそういうふうに書こうと努力するのが構造的プログラミングでないかと思っている．

(昭和51年8月30日受付)