结构程序设计与结构程序语言

唐 稚 松

中 国 科 学 院 计 算 技 术 研 究 所

1977.7.

结构程序设计与结构程序语言

唐 稚 松

中 国 科 学 院 计 算 技 术 研 究 所

1977.7.

目　　录

§1 前　言

《Software Practice and Experience》杂志 1975 年第一期社论[1]中有如下的一段话：

“一场革命正在程序实践所依据的概念中发生，慢慢地，一种新的程序设计原理正在出现——其目的即在于能写出正确的(没有错的)程序并且知道它们是正确的。人们日益明白，现有的程序语言中许多成分造成写正确程序的困难……设计一种或几种新型的程序语言正变成迫切的要求”。

这里所说的“革命”就是指结构程序设计。这里所说的新型程序语言也就是指结构程序语言。Knuth 在[2]中也有类似的两段话：

“一场革命正在我们如何写程序及教程序设计的方式方面发生。因为我们正开始更为深入地理解与之有关的思维过程。一个人不可能读了那本关于结构程序设计的新书之后而不在自己的生活中引起变化。这场革命的意义及其将来的前景，Dijkstra 已在 1972 年 Turing 报告“The humble programmer”中恰当地描述过了……”

“在当前，我感到我们正处在即将发现程序语言究竟应该是个什么样子的时刻。我展望到在今后不多几年内将在语言设计方面会有许多认真的试验。我的梦想是在 1984 年前后，我们将看到一种共同发展出的真正好的程序语言(或者更可能是一族协调的语言)……现在，我们虽离此目标尚远，但已有征兆表明这样的一种语言正在慢慢地形成之中……”

上面引述的两段话的内容竟如此地相近似，使人不能不感到这样一种观点具有相当大的代表性。

在 1968 年前后 NATO 在 Garmisch 召开的计算机科学家会议上第一次提出“软件工程”的概念；人们认为[5]，“软件工程的目的就在于得到廉价的软件，它是可靠的而且能在实际机器上有效地工作，……故软件工程即旨在建立和应用牢固的工程准则以达到这一目的。”软件工程所研究的课题，一般包含：结构程序设计，可移植性与可适应性问题，软件工作的组织与管理，软件的存档与维护等等……。而结构程序设计则是其中的核心问题。

1968 年 E. W. Dijkstra 写给 CACM 的著名的信[4]，虽然所提的问题是针对 Goto 语句的，但实际上是一个关于程序结构的更为深刻的问题。这件事已被公认是提出结构程序设计概念的开始。

软件工程及结构程序设计思想的出现不是偶然的。其客观背景是由于系统程序（特别是操作系统）等大型的程序系统日益发展，使计算机上配置的程序系统不断增大，造价不断增高，一方面使出错率增加，系统的正确性越来越难以验证和保证；另一方面，程序系统对不同的环境和用户往往需要进行修改，而系统越大又越难以修改。据说有些大的软件计划因此而不得不中途仃顿。从而产生所谓“软件的危机”。软件工程与结构程序设计概念就是在这样一种情况下产生出来的。

从这些概念提出以后，经过几年的争论、探索和实践。果然在应用中取得了实效，据 Datamation[6]报导，由于采用了结构程序设计的方法以及与此相联系的一些措施，曾使程序出错率降低到每 10000 行中仅含一个错，维修费减少 50%，程序生产率增长 50%。因此，

近年来，结构程序设计的讨论不但充盈欧美有关软件的杂志和学术会议，而且已引起生产软件的厂家的重视。正如文[7]中所述："在现在这个时刻，结构程序设计思想已经得到广泛的接受，不仅在学术界，而且在程序生产机构中亦如此。象 Datamation 这样一个面向商业数据处理的在美国居领导地位的杂志，亦有一期中包含几篇讨论这个问题的文章，SHARE 与 GUIDE 这两个重要的计算机用户组织的集会中已经举行了日益增多的以结构程序为议题的讨论集，在 IBM 公司系统科学研究所正在开设这方面的课程和举办这类学习班，而且已有几本关于这方面的专著在付印。"

不少人认为，结构程序设计方法的推行其意义还不止于当前的实用价值，更深远的意义在于它使程序设计由一种技术变成一套系统的方法[8],[9],[10]，使程序设计从依赖设计者手艺(Craftsmanship)的活动变成一门科学。[6][11][12]

对于这样一种重要的科技新动向，我认为有必要作些介绍，同时有不少我国软件工作者也迫切希望了解这方面的情况。

本文拟将近几年来关于这方面调查和研究的结果作一次总结*。尽量以第一手材料为依据，对其主要问题做些分析。为了能面向更为广泛的读者，文中回避陈述太复杂的理论推导和其它技术细节。

本文准备讨论三个问题：(1) 结构程序设计，(2) 结构程序语言，(3) 结构化的编译系统。

本文作者对许孔时、仲萃豪、程虎等同志及数学所计算站的同志在编写过程中给予的支持和邦助表示深切的谢意。

§2 结构程序设计

什么是"结构程序设计"？到现在止，还没有一普遍接受的定义。Dijkstra 虽然从 Goto 语句的问题开始提出这方面的思想，但他从未把删除 Goto 语句包含在结构程序设计的定义之中。他在提到结构程序设计概念时的确常涉及程序正确性的证明，但根据Knuth[2]的解释："他(Dijkstra)所谓正确性证明并非指从公理出发的形式推导，他是指任何一种'足够有说服力'的证明(形式的也好或非形式的也好)；一个证明实际上是指一种理解"。Baker 曾称：结构程序设计是"按照一组能提高程序的可阅读性与易维护性的规则而进行程序设计的方法"。[13],[62]。此外，Denning[15], Faulk[16], Lucena-Berry[17]都以专文对这个名词的定义进行过讨论。Gries[160]列举了对结构程序设计的十多种不同的解释，我们将在§5中再予介绍。我们在此不想就这概念的定义多予讨论。为了便于陈述，估且就"结构程序设计"，"结构化程序"，及"结构程序语言"三个概念予以区别说明如下：

结构程序设计是为了使程序具有一合理结构以便于保证和验证其正确性而规定的一套应如何进行程序设计的准则。按此设计出的程序称为结构化程序。其语言成分反映结构程序设计的要求和限制，从而便于书写出结构化程序，使用它写出的程序易于保证其正确性的程序语言称为结构程序语言。

更具体地说，本文作者认为，归纳起来，结构程序设计通常包含下述几个方面的内容：

---

* 本文曾以(上)，(中)，(下)三篇的形式印出，现将内容作了较大的修改，合成一册，特此说明。

(A) Goto 语句问题

事实上，这是关于程序的合理结构问题，因为Goto语句是对程序结构起有害影响最大的语言成分。自Dijkstra 在[4]中提出这个问题前后事实上已有不少人对这一语言成分提出异议，(文[2]中有一段详细的介绍)，主张删去 Goto 语句的人，其主要理由有以下两点：

(i) 一程序的正确性表现在其执行的结果。而这是一动态的过程。如程序中不加限制地使用 Goto 语句，则其静态结构与动态执行情况差异甚大，这样即使得程序难以阅读和理解，故容易出错亦难以查错。因此，删去 Goto 语句后即可增加其静态结构与动态执行情况的一致性，使程序结构较为合理。

(ii)一程序中如不包含这种不加限制的 GOTO 即可用对循环施行归纳以证明其正确性。故删去 GOTO 语句后，即可使证明程序正确性较易实现。

事实上，从理论上讲，GOTO 语句也并不是必不可少的。因许多重要的能行性理论如正规算法论，递归函数论以及 Lambda 演算中都没有与 GOTO 语句相应的成分。在 1966 年 Böhm 与 Jacopini[18]进一步证明，只要有图 1 中 (a)，(b)，(c)[或(a)，(b)，(d)]那样的一组控制结构(即串联，选择，重复(While 型循环或 Repeat 型循环)即可以构造出全部程序框图。

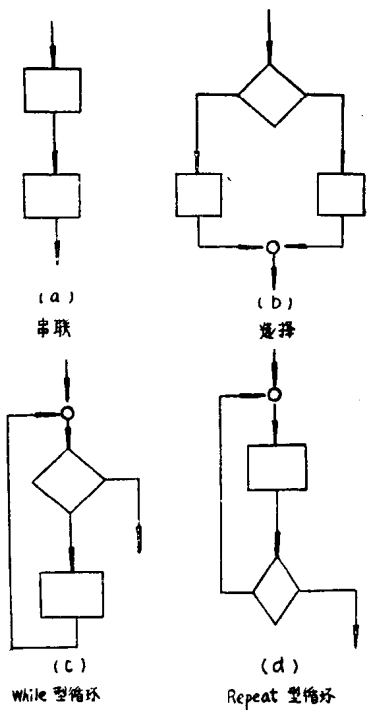

图 1

以上这几种控制结构的一个重要特点即都是只含一个入口一个出口。由单入口单出口的基本单元组成的程序即可为程序提供一清晰的微观结构。便于阅读，亦便于分划。但是包含多重出口的循环，如果要用图 1 所示的控制结构来表示，只有允许增加辅助变量，增加附加的计算，或改变执行的顺序时才能做到(见[12])。

在[18]之前，文[19]也已对整数函数讨论过类似的程序结构问题。并曾证明：(1)在传送，基本算术运算的指令之外，只要再加上与While型循环相应的指令，(代替与条件转移及GOTO语句相应的指令) 即可计算出一切能行可计算函数；如将此 While 型循环指令换成与步长型语句相应的指令，则所计算的函数类较小，为原始递归函数类。但这函数类中亦包括了所有通常数值计算问题中所遇到的函数。这些事实，也就表明GOTO语句并非必不可少的，并指明了可用什么样的指令来代替。(2)GOTO 语句事实上有两种：一种是限于只能往前跳不能往回跳的 GOTO，一种是不加这些限制的 GOTO。前一种 GOTO 并不能构成循环，因此，更不能构成图 2 所示那种有害的循环结构。下面我们称这种 GOTO 为受限制的 GOTO，因此，应将删去 GOTO 与限制 GOTO 区别开来。

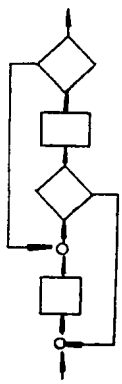

图 2

删除或限制 GOTO 语句所带来的好处。事实上还不止上述(i)，(ii)两点。近年来经过实践，还使人认识到有以下几方面的好处：

(a) 使程序便于修改，即使程序修改后所引起的付作用局部化。文 [20]

指出："当一不含 GOTO 的系统被修改时，其付作用最可能出现的位置是它所在的模块，或被它调用的下一层模块，或调用它的上一层模块…因结构程序设计使模块之间的联系局部化，以致一次改动所引起的付作用只是以所在模块为中心按同心圆的方式播散"。Dijkstra 在文[21]§10 中也指出程序结构与修改的关系，他说："程序结构应该能予见其可适应性与可修改性。我们的程序不仅应该(在结构上)反映我们对它的理解，而且应该从其结构上清楚地表明能方便地提供什么样的可适应性。谢天谢地，幸好这两方面的要求是一致的。"

(b) 能使全局优化的算法简化。Wulf[22]指出："删除 GOTO 的另一好处——这一好处在 BLISS 设计之前其作者是没有充分认识到的——即对代码优化带来的好处。在一带有动态存贮分配功能的子程序结构语言中出现的 GOTO 语句要增多运行时跳出子程序或过程的指令。删除 GOTO 就使这一堆指令也省掉了。而更重要的是使控制环境的辖域能静态地确定…由于流向分析是全局优化的先决条件，删除 GOTO 在这方面带来的好处不可低估"。文[23]也曾指出，由 GOTO 回跳所构成的循环中随循环而改变其值的循环参数很难查找，从而给全局优化带来困难。故删除 GOTO 使全局优化简化。文[24]具体讨论了用 SIMPL 这一结构程序语言写的结构化程序如何优化的问题。

有一些关于 GOTO 语句的研究，多着重于找到一种一般性的方法能用递归过程[27]，While 型循环[25]或分裂结点的办法[31](如下图 3) 去代替在程序中出现的任意的 GOTO 语句。这样的研究容易给人一种错觉，以为只要用这样的办法去代替掉程序中出现的 GOTO 语句，也就得到一符合结构程序设计要求的结构化程序了。事实恰好相反。虽然，这些方法中也提供了许多很好的技巧，但一般地讲用这样的方法替换出的程序不但往往效率很低，而且更为严重的是往往很缺乏直观性，比原来包含 GOTO 语句更难阅读，更难理解，恰好违反了结构程序设计的要求。因为，这样一些一般的方法，事实上仍然只能从理论上论证删除GOTO 语句的可能性。而结构程序设计的真正目的，正如 Knuth[2]中指出的："是将程序组织得易于被人读懂"。而不止是简单地删除 GOTO 语句了事。

删除 GOTO 语句后一种最常被人认为难以处理的情况，即图 4(a) 这种循环非正常出口的情形。而这种情形却是查表算法中经常用到的[27,26]。

例：场A中A[1]…，A[m]存以不同的值，依次查找其中是否存有值 x，如果没有则将值 x 存入新的一项中，无妨设一场B，其中 B[i]存查到A[i]的次数。

这一查表问题最常用的算法是：

```
for i:=1 step 1 until m do
if A[i]=x then goto found fi;
not found: i:=m+1; m:=i; A[i]:=x; B[i]:=0; found: B[i]:=B[i]+1;
```

这样的用 GOTO 表示的循环非正常出口，用 Wulf[22]中所设想的一般方法 [用图 4(b) 代替(a)]，是不能完全解决问题的，解决这个问题的办法之一是采用只允许往前跳而不许往回跳的受限制 GOTO 语句以代替任意的 GOTO 语句。但也随之发生一个问题，即程序的基本结构中将出现多重出口的情形。有人认为这样的结构是可以接受的[27]。

Kunth[2]中指出："关于 GOTO 语句问题人们最可能犯的错误是假定结构程序设计就是让人们还象通常一样写程序，然后再将其中 GOTO 删掉…我们真正需要的是那样去考虑程序，使得我们甚至很少想到 GOTO 语句，因而几乎不感到需要"。故问题的关键是语言中应包含足以取代 GOTO 语句的控制成分。图 1 所示四种结构显然是不够用的，故应再增加

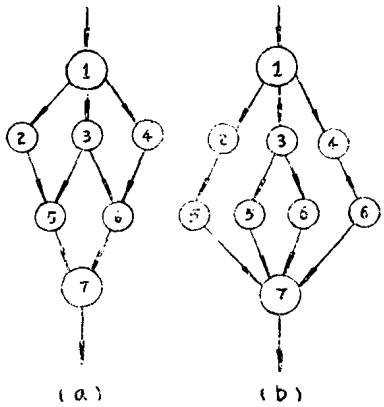

图 3

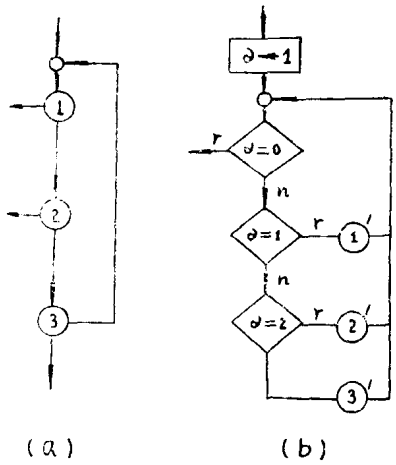

图 4

控制成分。比如步长型循环，if-then 型条件语句等都是常被引入的。此外，还有分情形语句，Wirth-Hoare[28]中开始提出这语句时，形式为

caseE:Ai,…,An end

后来因为由E算出之整数值 i 去找 Ai 时，在后面 $A_1,\cdots,An$ 较复杂的情况下不易看清楚其中那一个为 Ai，乃在文[29]中将此种语句改成如下的形式：

cases E of

L_1: A_1

…

Ln: An

end

此处 $L_1,\cdots,Ln$ 为标号，E即取其一为值。这样的改变自与原来形式并无本质上的改进。

大量增加控制成分以提高书写方便性的典型例子为 BLISS[30]。为了解决各种语句的非正常出口，对应于各种控制语句 BLISS 中引进了 8 种跳出语句，即：exitblock E,exitcompound E,exitloop E,exitset E,exitcase E,exitselect E,exit E,return E. 此处E即跳出时所带之值。这样一种处理，显然令人感到累赘，而且每次只许跳出一层更觉烦琐。后来 Wulf 对此作了修改[31]，在语句前容许出现标号，然后引进如下形式的一种跳离语句以代替上述跳出语句：

leave L with E

其含义是带着值E跳离标号L所示的语句，这样做不但形式划一而且所跳出的层数也不受限制，显然是一大改进。

此外，BLISS 中引进 6 种形式的循环语句：

while E_1 do E

do E while E_1

until E_1 do E

do E until E_1

incr〈名字〉from E_1 to E_2 by E_3 do E

decr〈名字〉from E_1 to E_2 by E_3 do E

并在前述的分情形语句外，还另增加了一种变型，Select 语句：

select e ofset E_0:E_1; E_2:E_3;…E_{2n}:

E_{2n+1} tsen

关于 BLISS 中以上这些控制语句的含义，文[32]已有解释请参阅此处不赘。

用大量增加控制成分以提高书写方便性的语言，在 BLISS 之前还有 BCPL[33]也具有这样一种风格。

在 BCPL 中包含以下一些循环语句：

unless E do C

until E do C

while E doC

C repeat until E

C repeat while E

C repeat

for NAME=E_1 to E_2 do C

以及如下二种条件语句

test E then C_1 or C_2

if E do C

还有如下形式的分情形语句

switchon E into

§ case con_1: C; endcase

…

§ case con_k: C; endcase

default: C §

以及如下的一种在执行C后求 E 值的指令：

valof C; rosultis E §

此外，还包含一个跳出一层循环的跳出指令 break 以及转返指令 return，在上面所述各式中，E表示表达式，C表示指令，也就是通常的语句。Evans 文[34]中指出，循环的多重出口以及过程的非正常出口，都可以在不用 GOTO 与标号的情况下，分别用 BCPL 中的 valof 及 switchon 两类语句表示出来。

增多控制语句的类型虽然可以在一定意义下为删去 GOTO 后书写程序带来一定的方便。但同时也增加了用户为掌握数量过多的控制语句带来不便。Dijkstra 将这样一种做法称为“baroque monstrosities”(畸形的丑八怪)。许多人认为这并不是一种值得效法的语言风格。近年来在语言设计的研究中出现了一种新的动向：即试图设计一些新的语言控制成分，一方面其功能足够的强有力而且灵活，用少数这种成分即可代替一大类控制语句；另一方面在结构上它又能满足结构程序设计的要求为程序提供一种合理的结构。

如C.T.Zahn,Jr[35]中提出如下的一种控制成分，称为事件推动的分情形语句(Event driven case statement)(以下简称事件语句)：

until EV_1 or EV_2 or…or EV_n do S_0

then case

EV_1: S_1

EV_2: S_2

…

$EV_n:S_n$

用图形表示如图 5。

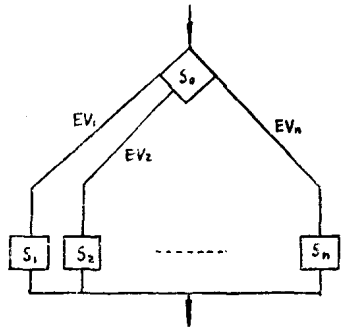

图 5

其含义是：重复执行 S_0，直到 $Ev_1,\cdots,Ev_n$ 这 n 种情形中有一种出现为止，当 Ev_i 出现时，则转去执行 S_i。

这样一种控制成分不但能代替许多其它控制成分，而且在处理查表，走树等算法时都较为方便。

Zahn 这种形式和 Clint and Hoare[40] 中提出的一种思想相近似，文[40]中主张，标号和过程一样，包含一分程序作为其"标号体"，还和过程一样，在一过程或分程序的说明部分予以说明。当执行到 GOTO 语句时，即象过程调用一样转去执行此标号说明，不过，在执行完毕后不再返回到原来位置而是跳出整个包含此 GOTO 语句的分程序。显然，这些 GOTO 语句，即相当于 Zahn 的事件语句中 S_0 内的事件，而标号说明，即相当于 Zahn 的事件语句后面带标号的语句，所不同的是事件语句中出现的这种语句位于语句的循环部分的后面而不是在分程序的说明部分。显然，Zahn 这样处理比 Clint-Hoare 方式更为自然。

Zahn 的事件语句这种控制成分，事实上是 G. V. Bochmann 文[36]中提出的如下形式"多端跳出循环语句"(Loopwith multiple exits)的一种变型：

〈简单循环语句〉

〈标号〉$_1$：〈语句〉$_1$

…

〈标号〉$_n$：〈语句〉$_n$

ended：〈语句〉$_0$。

其含义是：如该循环中有非正常出口，则在简单循环语句中安置如下形式的跳出语句：

exit〈标号〉i (i = 1, …, n)

当出现非正常出口时，即应执行此跳出语句，从而跳到相应的〈标号〉i，然后执行该标号后的〈语句〉i，执行完毕，则离开此语句；如执行该简单循环语句时不发生非正常出口的情形，则执行完毕此简单循环以后，跳去执行 ended 后面的〈语句〉$_0$。然后离开此语句。这样一种形式的循环语句一般即可较方便地表示出图 4 那种非正常出口，而且又避免了多重出口的问题。

Bochmann 并将这种形式的控制，推广到处理过程的非正常出口，即在过程调用时，写成如下的形式：

〈过程名字〉〈参数表〉exits

〈标号〉$_1$：〈语句〉$_1$

…

〈标号〉n：〈语句〉n

ended：〈语句〉$_0$。

而在过程说明的过程体中，出现非正常出口处写一跳出语句 exit〈标号〉$_i$。当执行到此跳出语句时，即跳出该过程，而转回到调用时所书写的〈标号〉$_i$ 处，执行其后面的〈语句〉$_i$，执行完毕，则离开此调用语句；如执行过程时不遇到非正常出口，按正常情况返回时，则返回到 ended 后面执行〈语句〉$_0$，执行完毕，再离开此调用语句。

与此相近，C. W. Barth[33]也提出了一种推广分情形语句的方案。用它可代替许多种控制语句：其形式是：

```
docase X of
case τ1
  SL1
esac
case τ2
  SL2
esac
  ...
case τn
  SLn
esac
esacoj
```

其中X如为取整值的量，则每一τ_i可以是一列整数i_1，i_2，$\cdots i_k$，当然所有$\tau_1,\cdots,\tau_n$中所含整数应不相同。最后τ_n如写成misc即表示剩下的情形；其中X也可以为空，此时每一τ_i为一条件，表示当这些条件依次从上而下执行时，遇到成立即执行其相应的语句，执行后转出整个语句，不成立则看下一条件。这样一种控制功能，显然颇似LISP中的条件式。

Knuth[2]中对前面所述Zehn的事件语句最为赞许。他说"我所知道的最好的一种这样的语言成份是新近由C. T. Zahn提出的。"他给这种形式作了小小的变动，将之分为两类，一类是以loop…repeat为括号的，一类是以Begin…end为括号的，即

```
loop until⟨event⟩1, or…or⟨event⟩n
        ⟨Statement lists⟩0;
repeat
then⟨event⟩1⟹⟨Statement list⟩1;
                  .....
    ⟨event⟩n⟹⟨Statement list⟩n;
fi
```

及

```
begin until⟨event⟩1 or…or⟨event⟩n
     ⟨Statement list⟩0
end
then ⟨event⟩1⟹⟨Statement list⟩1;
          ...
     ⟨event⟩n⟹⟨Statement list⟩n;
fi
```

并设一种⟨event⟩语句，即前面⟨event⟩$_i$的名字，当S_0执行时遇到这种语句时，即跳去执行后面相应的语句。

上面两种语句之不同在于loop开始的语句循环不已，直到S_0中执行到一⟨event⟩语句时

才跳出，而 begin 开始的语句只执行一次 S_0，执行到遇到一〈event〉语句时，往后跳去执行该〈event〉后的语句。

这类语句还可进一步增加功能，比如〈event〉还可以带参数，等等。

Knuth 认为虽然这样二种语句形式已足以定义出其它各种语句。但为了便于阅读，亦为了使编译效率提高，他主张再引进以下几种语句形式。

(a) Ole-Johnan Dahl 语句　形式为

loop: S; while B; T; repeat

此处 S，T 均为语句序列，亦可为空。当 B 成立时则跳出循环。并约定，S 或 T 为空时，则删去其前面之帽号，易见，当 S 为空时，相当于 While 型循环。当 T 为空时，相当于 repeat 循环。

(b) For 语句

loop for $1 \leqslant i \leqslant n$; S repeat

(c) if-then-else 型条件语句。

Knuth 认为，虽然这样一种控制形式已足够方便地表示各种算法。但是，为了使一程序得到更优的结果指令，可在写出这样的程序之后，将某些关键部分加以改进，要进行这种改进，对于用循环替换递归过程，或避免布尔变量以及处理 Coroutine 等类问题，在某些特殊情况下，最好再重新引入 GOTO 语句，Knuth[2]中以例子说明，一方面这种情况下引入的GOTO似乎很难用其它成分所代替，另一方面在这种特定场合引进 GOTO 似乎对程序结构有害影响并不大。因此，他主张在整个语言控制成分中仍保留GOTO，且功能方面并不加限制。但限制其使用范围，对一般的用户不必开放。这似乎是对待 GOTO 的一种较慎重的态度。已有的语言中 PASCAL，即如此对待 GOTO 语句。

本文作者认为以 Zahn 的事件语句为代表的这样一种设计一般控制形式的趋势值得重视。它不但可以在不用 GOTO 语句情况下仍能为书写程序提供方便，而且又能使语言的结构简洁。对于我国用户来说还有一个更特殊的优点，即这样功能灵活的控制成分中，所出现的自然语言文字如 loop，do case 等等，实际上不过一个符号而已，与实际的自然语言中文字的含义相距已很远。因此，无妨即用符号表示，这样可以使我国程序语言避免用外文作分隔符，而且更为简洁。

(B)　逐步求精的设计方法

这是程序设计的方法问题。程序的合理结构的取得与如何进行设计的方法是紧密相关的。现在通常所谓的程序设计技巧，是在计算机内存较小，希望编制紧凑的程序以有效利用内存这样一种要求之下提出的[16]，它与不加节制地使用 GOTO 语句亦有密切的关系。这样一种长期流行的风尚，一方面导致程序难于阅读，容易出错，另一方面使程序设计很大程度上依赖于设计者的手艺，很难使编制程序的方法系统化，自更难于考虑自动综合的问题。这种情况与 Huffman 方法尚未出现之前开关线路的情况有些相似。高级程序语言的出现，事实上是从描述工具的角度对这种情况的一种挑战，(这有点类似于 Shannon 等将布尔代数用于描述开关网络)。机器语言逐步不被使用即在表明这样一种变化。然而这个变化还没有从方法论上彻底解决问题。要使程序设计成为一门科学，必须从方法论上进行更大的变革。由顶向下(Top Down) 设计方法的提出，即是这方面的一种尝试。而逐步求精则是一种较为系统地被解释

过的由顶向下方法[8][9][81]。对这个方法，Wirth[8]中作了如下的说明：

“我们对付复杂性的最重要的工具是抽象，所以，一个复杂的问题不应马上即用计算机指令，数位与“逻辑字”来表示，而宁可用对问题本身较为自然的，在某种合适的意义下进行了抽象的词句与对象来表示，在这个过程中，一个抽象的程序出现了，它是对抽象的数据进行某些特定的运算并且是用某些合适的记号(极可能就是自然语言)来表示的，这些运算即看成是这程序的组成成分，对这程序应作进一步的分解(decomposition)而进入下一层的抽象，这样的精细化(refinement)的过程一直进行下去，一直到这程序能被计算机所“理解”为止。此时，此程序也可能是用某一高级语言如 FORTRAN 所书写的，也可能即是用机器指令书写的。”

下面先看一看例子。

例 1 矩阵A与向量X相乘送到向量Y中[41]

第 1 步：即直接写出此问题的抽象程序：

```
Y:=A*X
```

第 2 步：如此机器上没有直接实现矩阵运算的高级语言(如 APL)，而只有象 ALGOL60 那样的高级语言，则应将此程序中的运算用循环语句来实现。假定已对A，X，Y进行了说明，并且系统中已配有求向量长度的函数 length，则可将上述程序精细化成如下的程序：

```
for i=1 step 1 until length(Y)do
begin
    Y[i]:=0
    forj=1 step 1 until length(X)do
    Y[i]:=Y[i]+A[i,j]*X[j]
end
```

在一般情形下，如果这机器上有 ALGOL60 这样的语言，精细化到这一步也就可以上机了。但如果我们现在考虑的矩阵是一对称矩阵，为了节约存贮，只要一个长为 n(n+1)/2(此处 n 为A的长度)的向量也就够了。为此，则应将上述程序进一步精细化。

第 3 步：将上面程序中的A[i,j]改为

```
A1[if i<j then j+n*(i-1)-i*(i-1)/2
    else i+n*(j-1)-j*(j-1)/2]
```

此处 A1 即上述对称矩阵所存之一维场。

上面这个例子，当然精细化的过程不必就是如此三步，也还可以分解得更细密一些。不过以上三步亦可看出精细化过程的梗概。

下面再看一个 Dijkstra 文[21]及 Wirth[10]中曾举过的例子，

例 2 问题：打印出前一千个素数

第 0 步：即

```
begin“打印出前一千个素数”end
```

第 1 步：分析出“存表”与“打印”两个部分：

```
begin variable “table p”                    [1.1]
  “将前一千个素数存入 table P”              [1.2]
  end “print table p”                       [1.3]
```

第 2 步：将 table p 表示成整数场，并对“前一千个”进行分析：

```
begin integer array p[1:1000];                    [2.1]
  K从1到1000
     p[k]:="第k个素数"                            [2.2]
  K从1到1000
     print p[k]                                   [2.3]
```

下面对[2.2]进行分析，至于[2.3]的分析暂略。

第3步：将[2.2]改写成当型循环语句的形式，即

```
begin integer k,j; k:=1;j:=1
  while k<1000 do
    begin"j为下一素数"
      K:=K+1; p[K]:=j
    end
end
```

下面对"j为下一素数"进行分析，先定义一布尔量prim，故"j为下一素数"可表示成

```
while ¬prim do
   begin j:=j+1;
      prim:="j为一素数"。
   end
```

由于从2以上的素数均是奇数，故从2起j可按奇数递增以增加速度，即得

第4步：

```
begin integer k,j; boolean prim;
     k:=1; p[1]:=2; j:=1; prim:=false;
  while k<1000 do
    begin while ¬prim do
      begin j:=j+2;
         prim:="j为一素数"
      end;
    k:=k+1; p[k]:=j
     end
end
```

下面来表示"j为一素数"，显然，这可表示为"j不能被小于j的非1正整数所整除"。但事实上不必从1试到j-1，因根据数论，设p_i为第i个整数，则有$p_{i+1}<p_i^2$，故只要从1试到$\sqrt{j}$即够了。同时，我们可将"j被y整除"表示为"(j mod y)=0"

由此即得

第5步：[2.2]可精细化成如下的语句：

```
begin integer i,k,j,lim;boolean prim;
 K:=1, p[1]:=2;j:=1; lim:=1;
    prim:=false;
 while k<1000 do
```

```
begin while¬prim do
    begin:=j+2;
        if sqr(p[lim])≤j thenlim:=lim+1;
      i:=2; prim:=true;
          while prim∧(i<lim)do
          begin prim:=(j modp[j]≠0); i:=i+1
          end
    end
      k:=k+1; p[k]:=j;
  end
end
```

我们假定所用来书写算法的语言中能实现 mod 运算，故求精过程到此即可终止；如果此语言中无 mod 运算，则尚需进一步写出实现 mod 的程序。我们即不再细述了。

将此处得到由[2.2]精细化所得的程序段与[2.1]，[2.3]合在一块即得到所给问题的程序。

如果在所给的机器中没有实现上述各步中书写算法的语言，则还应以机器上所已实现的语言(比如汇编语言)来改写上述算法。

从以上两个例子，可以看出逐步求精过程的基本步骤。对这样一种程序设计方法，有以下几点值得注意：

(i) 逐步求精方法事实上是一种由顶向下的设计方法。即从最能直接反映问题的体系结构的概念出发，逐步精细化，具体化，逐步补充细节。直到成为一可以在机器上执行的程序。但是，这样一"由顶向下"的过程不能理解得太绝对化，因为，有时按某一方式精细化之后，在以下的步骤中发现原来那样一种精细化的设想并不好，或者对以后进一步精细化不便，或者有错，或者使算法不够有效，或者某些部分的算法可以合并等等。此时，即必须"由底向上"对原来已定下的某些步骤进行修改。没有这样一种修改，事实上等于要求上层作的每一步决定都是正确的而且是最优的，这是不可能达到的，这样去要求必使得每一步的决定都难以进行。因此，逐步求精过程应该理解为一种不断地为由底向上的修正所补充的由顶向下的设计方法。

(ii) 逐步求精的过程势必需要一种语言对它进行描述。这样的语言应该具有什么样的特征？在上面两个例子中，我们都是用自然语言表示接近用户直观的成分，而用类似 ALGOL 60 那样的高级语言表示程序细节部分。这样做自然比用框图描述更能反映逐步求精过程的要求的程序结构。但仍有不足之处：一方面这部分自然语言没有形式化，其含义没有精确的规定，另一方面像 ALGOL60 这样的高级语言不适于表示机器的细节，因而当精细化过程愈接近机器一端，则其算法愈难以用这样的语言来表示。欲补救这两方面的不足，最好的办法是建立两类结构程序语言，虽都具有表示合理的程序结构的特点。但表述对象的抽象程度不一样。一类是面向问题的专用的体系设计语言(Architecture Design Language)[20]它专用于表示所给问题的体系结构。如编译程序的体系设计语言，整机体系设计用的体系设计语言(见[43])，或解某类科学计算问题的体系设计语言等等。这种语言的特点是它能非常直接地表示

出一类问题的体系结构而舍弃与此无关的细节。如前面例1的第一步所用以表示矩阵与向量相乘的运算即非常接近通常数学语言的要求，故这类语言中必包含非常高级语言成分。它甚至不必直接在机器上实现，故不必考虑实现效率问题。另一类是公共基础语言(Common Base Language)其中一方面包含某些通用的非常高级的数据结构及控制成分，另一方面又包含某些能表示机器细节及程序细节的语言成分。这种语言作为中间语言，它一端与体系设计语言相衔接，体系设计语言中的"非常高级"的概念应可以在这中间语言中表达得出来；它的另一端可与机器语言或汇编语言相衔接，因此，这语言中应包含许多面向机器的程序语言的特征。故在逐步求精过程中可用这语言写出高效的解题程序。正如Knuth[2]中指出的："我们将看到分层的语言；用其最高层，我们能写抽象程序，而其最低层，我们将能表示存储控制，寄存器分配，下标范围检查压缩等等"。通常所谓算法语言与程序语言的矛盾应在这语言之中统一起来。显见，这样一分层的语言系统中，各层语言的差异主要在数据结构方面，其控制结构可以基本上相同。正如Goos[56]中所述："在分层新语言中所主要关心的是引进与新的抽象机相应的新的运算，数据类型及数据结构，没有理由认为相应于不同两层语言应具有不同的控制结构。"我认为，对结构程序语言的这样一种设想，是逐步求精方法所导致的必然结果。

(iii)当逐步求精过程的各级算法均用形式语言来表示以后，必然发生如下的问题：(1)如何过渡？(2)如何保证过渡的正确性？前一问题即是问由设计者手工过渡还是自动过渡，后一问题即是问由设计者保证过渡的正确还是自动证明其等价性。对这两个问题，事实上目前存在三种不同的处理办法：

(a) 要求自动过渡(人可能给少数信息)，而且自动验证等价性。这就是所谓自动程序设计的要求[42]。目前尚难以实现。

(b) 手工实现过渡，但机器上配有许多辅助手段，能用以检验每一步过渡的正确性。这方面目前已有一些试验性的成果，如文[41]中所述的情况。

(c) 两方面均由设计者负责。目前大多数仍只能做到这一步。

(iv)结构程序设计方法与框图设计的关系。

框图设计过程虽然也常表现为由粗框到细框的演变过程，但框图设计与逐步求精方法有本质的不同。逐步求精方法可以说是对设计者思路的一种约束，要求设计者先考虑好全局的结构再考虑局部的细节，而且，逐步求精方法要求设计者对各步中每一概念在引入细节加以精细化之前，均作为一整体概念加以标明，在未作这种说明之前不能直接即写成由细节所表示的算法。而框图设计中则没有这样一种约束，它很容易使设计者在全局结构尚未筹划清楚的时候即去直接写出大量的细节，同时，又可能将那些充满细节的框，用没有限制的GOTO联结起来，以致使算法的全局结构变得十分混杂难以理解。Koster在[44]中指出："框图正好是用来显示荒唐的细节的，它把人们引去将大量微小步骤写在一些框框之中……故框图本性上即不是进行逐步求精的最好的工具"。而且"框图鼓励你不去说明一框的含义是什么，而让你将一大堆细节塞入一框之中"。除了以上这一缺点外，Koster与Ross(均见[44])还分别指出：(1)步长型循环"在框图中没有一合式的记号来表示"，(2)在结构程序设计需要标明的成分如数据结构的定义以及程序模块在几处被调用的关系等，在框图中都难以合适的表示。由于有以上的不足之处，故主张采用结构程序设计方法的人，多不主张用框图进行设计。有人说：[44]，他已多年没有用过框图；也有人指出，他的学生曾先用框图写算法，原来希望框图设计好以后再将它改写成高级语言的程序，后来发现这样做很困难，只好从头开始。

不过，框图设计也毕竟有其优点，即直观性强，可以平面表示各部分算法的关系，故便于阅读，特别便于初学。故结构程序设计方法提出后，即已有人提出符合结构程序设计要求的框图格式[45][46]，下面是 Chapin[46]中的方案：

Chapin 框图的基本单元是长方形框，它只能有一出口一入口，其中用横线隔开表示依次执行由横线分隔出的子算法，各子算法之间亦假定只一入口一出口。(如图 6(a)，(b))。

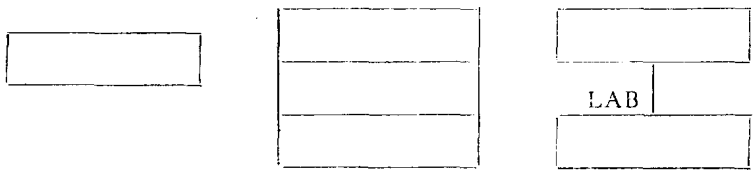

图 6

如一子算法需用标号标出，则将它与其前面之子算法分开，用一直线将它们相连，并将此标号写在该子算法所对应的框的左上方，如图 6(c)。

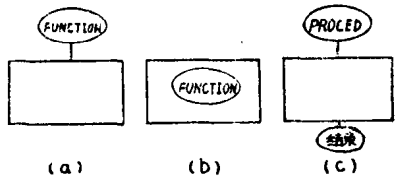

图 7

椭元型框用于表示函数(或过程)名字。将它置于一组长方框之上以直线相连，以示此组长方框即为该函数的内容(如图 7(a)；如此椭元型框出现在一长方框内即表示此椭元型所示之函数(或过程)在此长方框所示之算法内被调用(如图 7(b))

一函数(或过程)的结尾有两种，一种是它不被调用而结束，其结尾为一用直线相联的椭元框(如图 7(c))，如它是被调用的，即自动返回，此时，在长方框底下不再表示任何出口(如图 7(a))。

条件语句表示如图 8(a)，其中上面子框的中间写条件，用斜线划出的部分表示条件成立与否的出口，下面子框中用直线分成两部分分别对应于其上方之两出口，以表示条件成立与否所对应的两语句。

条件语句加以推广即可表示分情形语句(如图 8(b))。

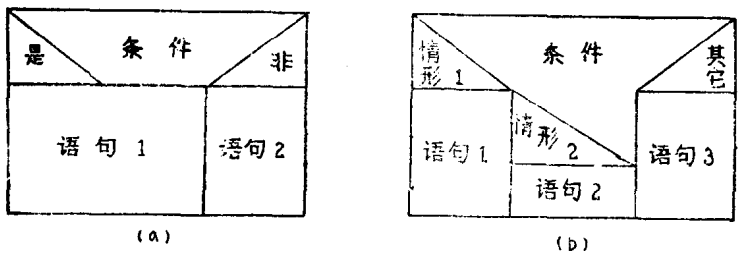

图 8

循环语句表示(如图 9)其中与画斜线部分相通的横框表示循环所依赖的条件，与画斜线的长方框相邻的部分表示循环体。由循环所依赖的条件是在循环体之上或下而区别 While 型与 Repeat 型两种循环(如图 9(a)，(b))。