贵州电子计算机应用通讯
（第二次系列软件工作会议文集）

2

贵州省计算技术协作组
贵州省科学技术委员会
一九七七年二月

目　　录

程序语言的发展近况

计算所 唐 稚 松

近年来，由于程序语言应用领域的拓展以及有关理论研究的深入，而更可能是由于“结构程序设计”的提出使人们对程序语言的一些基本概念和成分产生疑问。有关程序语言设计原理的讨论十分活跃，国外组织过许多次专门讨论这方面问题的会议以及有关的专业组活动。面对这样的情况，事实上存在着两种相反的态度：一种是积极支持和推动这样一种发展[1][2][3]，最有代表性的是Knuth的下述意见[4]：

“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。……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……已有征兆表明这样的一种语言正在慢慢地形成之中……”。

另一种则是相反的意见。他们对发展新型语言表示怀疑，比如有人（Ralston）主张只要对人们已经习惯了的Fortran进行某些必要的修改就可以了，不必另行设计。更引人深思的是Naur[5]中所指出的一个事实：即世界语当年的争论情况。世界语发展到一定程度以后，许多语言学家提出与原来的Esparanto语言方案不同的新语言，虽然后面这些新的语言方案从语言学角度看比原来的方案要好，但因人们已习惯了原来的方案，由此引起争论而造成分裂，最后导致世界语运动的失败。Naur认为程序语言当前的情况与此颇为类似。

本文作者不想对这两种意见作出判断。但是本文将要介绍的内容是与提出语言成分的新概念相联系的。

Wasserman[6]与Schwartz[7]曾对程序语言研究的新进展作过简明的概括。文[8]对本文将要介绍的内容作了较详的说明，可供参考。

以下分三个方面介绍程序语言的新概念：

一、语言的宏观结构。

此处所谓宏观结构包括语言的总体结构及模块结构方面的一些问题。比如，语言所描述的对象应分为程序与子程序两级还是应增加更高的系统程序一级？各级软件应以什么形式的结构组成模块？各模块之间的关系应是一种什么样的关系？各模块之间信息应以什么样的方式进行交换？与宏观结构密切相关的是一项较大的软件工作应按照什么样的结构对参加工作的人员进行组织？等等。

结构程序设计的思想使人们对程序语言的宏观结构更为注意。特别是系统程序语言及模拟语言在这方面提出了一些新的问题和要求：比如：

(i) 有许多系统程序采用分层结构，其外层模块对较内层的任一模块原则上均可引用，可以说是一种线性全序结构。这样一种结构不但Fortran那样的块结构语言不能描述，就是

ALGOL60或PASCAL那样的半序，树形结构语言也不适于描述。这方面Hansen在Concurrent PASCAL[9]中提出了一种描述系统程序结构的方法。White与Presser的JOSSLE[10]虽在程序与子程序之上提出了更高一级的"联系区"概念与系统程序相应，但各级的结构仍然是半序树形结构的形式。

(ii) 当程序大，层次多的情况下，一模块中出现的公用量不易看出其出处，且易发生各种副作用。不论是 Algol 型的嵌套结构及 Fortran 型的块结构均有这方面的问题。为此，Walf 与 Shaw[11] 曾对"公用量"概念提出过异议。针对这一情况，White 与 Presser 在JOSSLE的各级模块的开始处规定将所引用的外层量予以列出，在高层则应重新说明，低层则只需列出其名字即可。

(iii) 在系统程序及模拟程序中，两模块间的关系常常不限于"调用"这样一种从属关系，有时需要建立一种作用或相互作用的关系。比如，模块A引用模块B进行打印，A不必等待B打印完以后转返就能继续进行，A、B应可以并发地工作，甚至A在B之前结束亦无不可。

与此相联系，还有以下几个方面的问题：

(iv)量的生成（即存贮分配）应可动态地进行（比如LIST结构即需要如此生成）；

(V)某些模块在执行结束后其中某些量的值还应保留在以后用；

(vi)数据结构的意义事实上是由定义在其上的运算来决定的。某些数据结构只有限于某些运算才有意义。比如，"栈"这样一种结构虽然也是一种一维向量，但它只应对"上托"（Pop）"下推"（Push）等某几种运算才有意义，它不能像一维场一样允许对其中任一分量送信息。与此相联系，系统程序中常需对某些公用的数据结构进行保护，不允许对它施行一定范围以外的运算。

为了表示出以上(iii)—(vi)这些方面的功能，在 SIMULA67[12] 中提出了 CLASS的概念。其特征是将一模块的定义，生成，与其中所示的运算的执行分开来。其定义与过程定义的形式很相似。不过在定义了一CLASS后，还应定义若干指向该CLASS的指针；在算法执行时可通过NEW运算动态地生成一CLASS，此时，实际上即动态地分配此 CLASS 的局部量，生成了一CLASS后即将所分配的局部量地址头送入一指向此CLASS的指针，此时还有可能再执行若干对此局部量的初始预备性运算(比如清除送初值等)，但不执行任何其它的算法。至于CLASS的执行，即执行其中所定义的运算。比如CLASS A中有运算 Oper(m)，X为具有类型A的一个对象，则X·Oper(n)即表示执行X中的运算Oper(n)，此处n为参数。此外，CLASS中还可出现"CALL"，"Resume"等运算来实现模块间的交互作用的关系。而且在CLASS之内或之上还可定义CLASS，由此实现CLASS的重叠关系。

Hansen在Concurrent PASCAL中，将CLASS概念加以扩充，提出monitor的概念，其主要特点是在 CLASS 之上引进排队概念。他区别两种排队：一种称为短期排队，这是由系统安排的，比如同一运算同时被不同模块所执行时则系统自动予以排队；另一种称为长期排队；即是由程序模块书写者所安排的排队，比如对一片场，一方面输入信息一方面输出信息，当信息输满时，则应等待暂停输入，信息输空时则应等待暂停输出。对于这种情况，系统提供排队岗queue以及delay与Continue两运算，用户运用这些手段即不难写出长期排队的算法。

CLASS中所实现的由运算决定数据结构意义的思想事实上一定程度上体现了多年来程序设计中"一一致指引"的要求：即如果数据结构是由运算所决定，则程序算法中只需写出其有关运算，而不必直接写出其数据结构。因此当对一程序的数据结构进行修改时，只需修改其数据结构的定义部分而不必改变其算法部分。这当然是大型程序设计的一种理想境界。Liskov与Zilles[18]中的 CLuster 概念即是从这一方面对 CLASS 概念的改进。主要是使其中所定义的数据结构作更为周详的遮掩，以使数据结构的改动更不影响程序算法。

本文作者认为上面所提出的有关宏观结构及模块概念的问题都是值得注意的问题。而上述各种解决的方案，虽然取得了重要的成功，但从我国的现实情况来衡量，似仍有可商榷之处：

(a) JOSSLE在程序之上再提出联系区一级来反映系统程序结构，这是可取的，然而其联系区仍按半序树形结构来组织，则与一般操作系统的全序线性结构不符。本文作者认为参考JOSSLE的作法，但将其总体结构改为如下形式较妥：

〈软件〉::=〔〈不分层部分〉，〈分层部分〉〕

〈不分层部分〉::=〔〈模块列〉〕

〈分层部分〉::=〈n层结构〉

〈0层结构〉::= 'LEVO' 〔〈公用量信息〉〈模块列〉〕

〈i+1层结构〉::='LEV' 'i+1' 〔〈公用量信息〉〈模块列〉〈i层结构〉〕

此处〈模块列〉即由模块组成的序列，模块中除包括通常的程序，子程序作为模块外还可以包括另一些系统程序所需要的模块概念，如过程等。

此外，JOSSLE 中对外部量的引用在各层予以注明也有其优越性，不过，似可表示得简明一些，且似不必规定得太死。

(b) CLASS，Monitor，或Cluster等概念虽有其优越性，但在通常的语言模块概念外另设一新的模块概念亦增加用户掌握的困难，似不如在常用的语言成分之上增加一些新的规定来分别实现其功能为宜。比如，为了限制一数据结构上所能施加的运算，即可在通常的类型之上增加一限制部分，例如

```
Stack (m, 'int') =*{m, 'int'}
                  → 〔Pop, Push, erastop〕
```

此处除了定义Stack为一长度为m的一维场外，还限制其上所能施加的运算为Pop，Push等，这些运算则作为子程序在子程序部分另予定义。

又如动态分配存贮的数据结构，即可参用PASCAL的 New运算。设先已定义P为指向某类型的指针，则执行语句New(P)后，即按P所指类型分配一片存贮然后将其头地址送入P。

至于存贮保护及排队问题还是采用常见之P，V操作以及常见的PCB表的技术较为灵活，易与常见的操作系统概念协调。

二、语言的控制结构

这方面主要是讨论语言的语句形式。从结构程序设计的观点出发，对常见的语句形式提出了如下的一些问题：

(i) Bohm与Jacopini[24]证明，全部程序框图可以由串联（即复合语句），选择（即条

此为试读，需要完整PDF请访问：www.ertongbook.com

件语句），重复（即While型循环或Repeat型循环）来构造。有人认为以这些语句为基本语句形式适于进行程序正确性的证明[15]。人们甚至认为这些基本形式类似于开关线路的"与"、"或"、"非"门，在这基础上，有可能建立程序领域的Huffman方法[18]。

Dahl将While型循环与Repeat型循环合并成如下的一种形式：

'Loop'：S； 'While' B：T； 'repeat'

其中，当S为空时相当于While型；T为空时相当于'repeat'型。

(ii)自1967年Dijkstra[17]提出GOTO语句的危害性以来，有关这方面的意见很多，总的讲，一般都认为，无节制地使用GOTO语句对程序的结构破坏性很大，可是，完全删去GOTO语句对于有些情况又难以表达。Knuth[4]中曾举出一些使用GOTO语句使算法优化的例子。而最为常见的情况即多出口的循环。为了能表示这样的语句，Zahn[19]提出了如下的一种语句形式：

'until' EV_1 'or' EV_2 'or' …… 'or' EV_n 'do' So

'then' 'case'

EV_1:S_1

……

EV_n:S_n

Knuth[4]曾对此语句形式极力赞扬。引起人们的重视。但Halaas[10]认为此语句形式在嵌套层次较多时仍不够清晰，提出另一修改形式：即保留'until'…… 'do'部分，删去'Then'以下的部分，而当So中出现EVi时，即写成EVi{S_i}，用以表示在遇到EV_i时，即转去执行S_i而后跳去。但本文作者认为，这一修改形式也同样具有嵌套时不够清晰的问题，可是它却不如Zahn原来形式醒目。

(iii)近来有人对"'IF'— 'Then' — 'ELSE'"提出异议[20]认为这一语句形式在嵌套多层以后极不醒目。不少人提出对这语句形式的修改方案，大多从'Case'语句加以改变而来，其特点是：用它表示分情形算法时便于减少嵌套的层次，从而加强清晰性。本文作者认为，在这些新型'Case'语句中，Embley—Hansen的KAILSELECTOR[21]是较有特色的，且可将它与Zahn语句合并成一统一的形式，值得考虑。

三、语言的数据结构

近年来有关数据结构的讨论很多，最值得注意的是以下两方面的问题：

(i)'Hoare'的数据结构理论及PASCAL语言

Hoare-Wirth近年提出的数据结构理论[22]并由之实现的PASCAL语言[23]都是较为重要的。他们提出了一些关于设计数据结构的颇有意义的原则，值得注意。但由于他们往往将这些原则推向极端由此提出的一些新型数据结构形式却往往是颇为牵强的，大可商榷。

(a)近年不少人从数学的观点出发分析数据概念，比如有人将人脑中的"信息"（information），书写形式的"数据"（Data）和机器中"存贮"（Storage）三个概念分开，并将由信息到数据的映射称为信息结构（information），由数据到存贮的映射称为数据结构。这样来分析"表示"与"数据"的关系是颇有意义的，有助于人们对数据概念的理解。Hoare[18]也试图对数据结构进行数学分析、他把"类型"（type）即归结为"值的集合"。在这种思想指导下，他不但将常见的基本类型如"整型"、"实型"，"符号型"均看成是

值的有穷集，而且还引进两种新的基本类型：一种是纯量型，即用户自定义的名字集，《如（Math，Physics，Chemstry）》；另一种即所谓子范围（如1‥31）。应该承认，这两种新类型是颇为有用的，用来表示某些常见的表格颇为方便。但是将“值的集合”通通理解为类型则有些流于牵强。比如场的界对，虽然是一种子范围，但如将之看成是类型即很不自然。事实上虽然计算机语言中的类型概念不应与其在某具体机器上的表示方式混为一谈，但如完全脱离表示（至少是在抽象机上的表示）去谈类型则往往要丢失掉类型的一些基本含义。比如计算机语言中的整型，实型毕竟与数学中整数，实数不同、数学中的整数集是实数集的一个子集，凡能施之于后者运算亦应可施之于前者，而在计算机中则不然、原因即在于在计算机中整型与实型的“表示“完全不同。本文作者认为，用户自定义的名字集及子范围虽不宜作为“类型”，但仍可以其它的方式引入到程序语言之中。比如：以“X：‘Int’”表示X为整型量，以“X＜＝＞3”表示X为整数3的名字。前者使X取值范围最广，后者使X取值范围最窄，处于这二者之间可引入“X=(1‥31)”，它表示X取1至31范围内任一整数为值。这样即可既引入这些概念，而又避免对“类型”概念作牵强的解释。

(b)设置类型应能便于查错，对于编译时不易查错的类型应予避免。

这一原则本身应该说是可取的。特别是系统程序往往要求编译时查出尽可能多的错。可是PASCAL为了贯彻这一点而引入Record的Vaviant Part则令人感到很不自然。

类型折取的确是一种常易引起难查错误的类型。比如“X：‘Union’（‘int’，‘Real’）”，当对X赋值时，很难确定X当时究竟应为‘int’，还是‘Real’。为了解决这一问题，事实上只要规定类型折取的各分量均应带一分量名字，在X参加运算时，亦应带其分量名字即可。比如上例，即应写成：“X:Union (i: ‘int’，r: ‘real’)”，当X参加运算时，如为整型，即应写成X·i，否则，即应写成X·r。此时，正确与否编译时即可检查了。但PASCAL还不满足于此，还要求检查X·i或X·r是否正确，因此，它规定将X定义为：

```
X: 'Record'  'Case'  tag=(A, B)  'of'
        A=(i= 'int');
        B=(r= 'real')
        'end'
```

X·i，（或X·r）是否正确当且仅当tag=A（或tag=B）。这样，的确进一步地加强了其查错功能。可是这样一种‘Record’的‘Variant Part’的书写方式实在非常令人不易理解。文本作者认为用这样的方式来贯彻查错原则是有些过分的。这样一种令人费解的书写形式实不值得提倡。

(C)设置类型时应考虑其实现的效率。这也的确是一条很重要的原则。经验证明，过去某些语言在设计时完全不考虑如何实现，待实现时发现困难重重，这是一种不应重蹈的复辙。但Wirth等在PASCAL中却将这一原则推向极端，凡是实现效率较低的类型一律不允许出现，甚至为了提高其实现效率，不惜使它徒具空名而完全改掉其实质性的含义。最典型的例子即集合型（Set）。本来，这是近年来较受人注意的非常高级的类型、用它表示某些检索算法是很方便的。它的标准形式应该是按条件组成集合、即“{S|P(S)}”的形式。不过，这样形式的集合型实现时效率是较低的。PASCAL为了贯彻效率原则曾多次改动这类型的含义、最后改成为Powerset，即如A是一有穷集（例如{1，3，5}），则“X：‘Set’ ‘of’

A"，即表示X是A的一个子集（即X为A的所有子集的集合中的一个元素）。为了提高效率，使一个机器字的"位串"能表示出这类型，PASCAL还要求上述A中元素的个数不能超过一机器字长。这是多么不自然的一种集合型概念！它的实用价值显然远不如上述按条件组成集合的集合型概念。本来，一高级语言中不包含集合型也是可以的，PASCAL为了贯彻效率原则竟不惜偷换概念，用这样一种"子集"的概念取代集合型概念，徒然造成概念混乱，实在是一种不值得提倡的作法。

(d)高级语言中的类型应独立于机器概念。这是多年来为人们普遍接受的原则。PASCAL中定义file型时，将这原则贯彻得极不老实，他实质上是以一种假装的独立于机器的词汇去陈述与机器有关的内容。他说："每一filef的说明自动地引入一缓冲变量(buffer Variabel)，它表示为f个，具有该file分量的类型。它可被看作是一个'窗口'，通过它人们可以观察（读）当前存在的分量，以及附加（写入）新的分量"。如果读者原来没有外存带以及内存缓冲存贮区等机器概念，他将如何去理解file及其缓冲变量的定义呢？这样一种冒充的独立于机器的概念也是难为人们所接受的。

(e)Hoare在文[24]中曾指出一颇有意义的事实，即程序语言中控制结构与数据结构在逻辑上的一致性。比如：表示不同型对象的排列，在控制中为复合语句，在数据中为纪录；表示相同型对象的重复，在控制中为循环，在数据中为向量；表示不同情形的选取，在控制中为Case语句或条件语句，在数据中为类型折取；在控制及数据中又各有一打乱原排顺序的成分、控制中为GOTO语句，数据中为指针。二者如果滥用，均可能产生有害后果，但又都是为使用灵活所必需。且加以适当限制，也都是可允许的。此外，在控制和数据中亦均有一不讲排列顺序之成分，在数据中为集合型，而在控制中则为Dijksfra新近提出的不确定语句。Hoare指出这种对应对理解程序语言的内在逻辑结构是很有意义的，曾有人提出一种程序设计的新方法[25]即先根据问题的数据结构确定其算法的轮廓，然后再逐步求精。其途径即根据数据结构的基本逻辑结构选出其相应的控制结构，比如其数据结构为向量，则其算法部分必为一循环语句。如此等等。

Hoare虽指出了上述事实，但是包括PASCAL语言在内的现存的高级语言中，其控制结构与相应的数据结构表示形式上差别很大（比如当型循环为"While……do……"，而数组说明为"array……of……"）。这样一种差异极大地掩盖了它们之间逻辑上的一数性，使人们看不清这种内在的联系。为了突出这种一数性，本文作者认为相应的两种结构最好具有相似的表示形式。比如循环表示成"*〔……〕"，向量表示为"*{……}"等等。

(ii)面向机器与面向算法问题。

面向机器与面向算法这两个矛盾的方面如何适当处理是程序语言的老问题。这个问题在系统程序等大型程序中显得更为突出。一方面这类程序要求正确性高，应便于阅读，故强调面向算法，而另一方面这类程序因需经常重复执行，故亦强调效率，因此，又要求具有面向机器功能。对于这矛盾应如何安排，有以下几种方案：

(a)强调一面的极端意见。

一种意见是强调面向算法，故主张采用高级语言成分，至于效率问题，交给编译系统去解决[26]。

另一相反的意见是强调面向机器，故主张采用汇编语言，至于算法阅读性问题，认为汇

编语言亦能写出好读的程序[27]。

对于这样一种极端的意见，我想不必详论，其片面性是显见的。

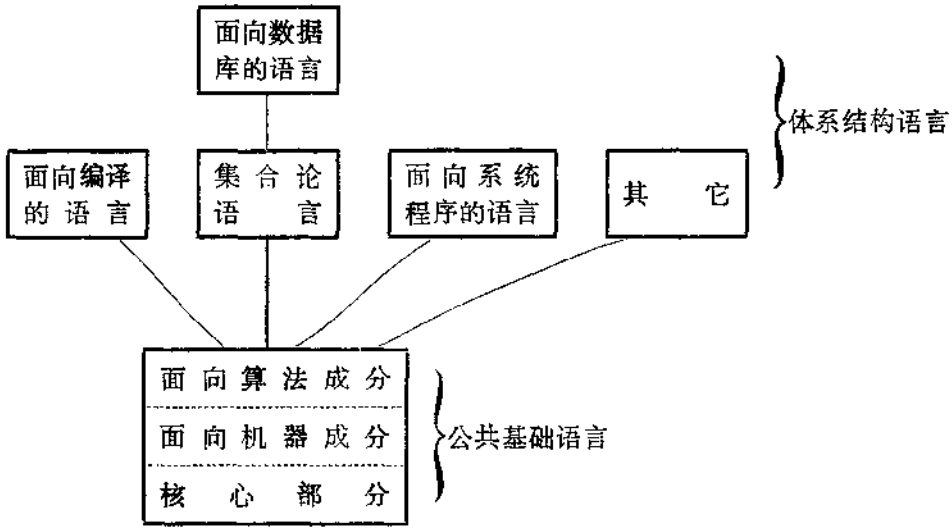
(b)主张用面向机器的数据结构与面向算法的控制结构结合起来组成面向机器的高级语言。其代表为BLISS，PL/360等。

这的确是解决矛盾的可行的方案之一。不过认为面向机器的数据结构不影响可阅读性则是不尽符事实的。比如“纪录”形式的数据结构显然对程序的可阅读性是关系很大的。

(c)主张构造分层的语言系统，其低层为面向机器的语言成分，其高层则包含面向算法的高级成分，这样即可用高层成分先写出便于阅读的程序，而后再用低层的成分改写其算法以提高其效率。这方面的代表如WSL[27]。

这样的方案显然比(b)的方案前进了一步，不过，就面向算法而论，各不同的应用领域中所包含的专用的非常高级的词汇很不相同，（比如面向编译的语言应包含CODE语句，而在其它专用语言中则无需这样的成分不可能在同一分层系统中包含所有这些不同的非常高级的词汇。所以分层系统还不能彻底满足面向算法的要求、本文作者认为，较理想的作法是构造语言族。

(d)分层的语言族。这一族语言包含两个组成部分，一是全族所共同的公共基础语言，另一部分则是满足各不同专用要求所扩充出的各种体系结构语言。公共基础语言本身可以是一分层系统，其最底层是最小的核心部分，在此之上再加进各种面向机器的语言成分成为第二层，然后，再在此基础上扩充出一些一般性的面向算法的语言成分，构成整个公共基础语言。至于进一步面向算法，则应加进各专用的非常高级的语言成分（或行话）而构成各专用的体系结构语言，整个语言族结构如下图所示。



XYZ语言族结构

在语言族方面目前国际上较有名的是SIMPLE族[28]。本文作者正在从事的XYZ语言族即属于这一类。

参 考 文 献

〔1〕 Wirth.D.IFIp Congress74.

〔2〕 Hoare.C.A.R.Proc.British Comp.Soc.Conf.on H.L.Prog.Lang.1973

〔3〕 Neely.P.M.Software Practice and Experience Vol5.No1.1975

〔4〕 Knuth.D.E.PB233507, 1974

〔5〕 Naur.P.CACM Vol18No2 1975

〔6〕 Wasserman.A.I.AFIps Proc.1975 Vol44

〔7〕 Schwartz.J.T.ACM.1975 Proc.

〔8〕 唐稚松，结构程序设计与结构程序语言（中），（即出）

〔9〕 Hansen.P.B.IEEE Trans.on Software Engineering VolsE-1 No2.1975

〔10〕 Presser.L and White.J AFIps Proc.Vol43 1974

〔11〕 Wulf.W.and Shaw.M.SIGPLAN Notices Vol8 No2, 1973

〔12〕 Dahl.O—J and Hoare.C.A.R. "Structured Programming" (ed.ty Dahl et.al) 1972

〔13〕 Llskor.B.and Zilles.S.SIGPLAN Notices Vol9.No4.1974

〔14〕 Bohm.C.and Jacopini.G CACM Vol9 1966

〔15〕 Mirth.N. "Systematic Programming" Pretice Hall Inc.1973

〔16〕 Mills.H.R "Structured Programming.A Tutorial" ComP.Con75

〔17〕 Dijkstra.E.D.CACM.Vol10 1967

〔18〕 Zahn.C.T.jr.Lect.Not.on ComP.Scie. Vol19

〔19〕 Halaas.A.BIT ou15.3.1975

〔20〕 Weinberg.G.M.et.al SIGPLAN NOTICES Vol10 No8.1975

〔21〕 Embley.D.W.and Hansen.W.I.SIGPLAN NOTICES Vol11.No1.1976

〔22〕 Hoare C.A.R. "Structured Programming" (ed.Dahl et.al)

〔23〕 Jensen.k.and Wirth.N.Lect.Not.on ComP.Scie.Vol18

〔24〕 Hoare.C.A.R.Proc.INTL.Conf.Reliable Soft 1975

〔25〕 Jackson.M.A.Principles of Program Design 1975 (Aplc Studies).

〔26〕 Wells.M.B. "Machine Oriented High.Level Languages" (ed.Poel) 1974

〔27〕 Fletcher.J.G.and Kalan.J.E.ACM75 Proc

〔28〕 Geiselbrechtinger.F.et.al. "Machine.Oriented High Level Languages" (ed.Poel) 1974

〔29〕 Basili.V.R.and Baker T "Structured Programming: A Tutorial" Comp.con75

软 件 移 植 概 况

数学所 计算站 周 龙 骧

一 引 言

实现软件（如汇编程序、解释程序、编译程序以至操作系统等等）是一件浩繁的工作。实现一套软件系统过去通常所用的方法需要经过以下几个步骤：

1.调查研究，准备资料（包括对一些新的思想、新的方法的学习）；

2.交流讨论确定方案；

3.人员组织、确定分工；

4.分别完成各自的具体方案，画出框图，彼此不断进行交流讨论，协同作出调整；

5.各自完成程序的编制，进行分调，并进行交流；

6.进入总调，进行再调整；

7.试用；

8.正式交付使用。

一个真正有效的软件它所需要的人力和时间是很多的，给计算机配一套语言即使是比较简单的语言它的代价也都以“人年”来计。例如国产DJS—21机的宏汇编语言是一个面向机器的低级语言，可以说是一种起码的软件系统，而它的实现也花费了约一个人年的代价。DJS—21机的BASIC系统花费的代价是三个人年。复杂一些的软件系统花费的代价那就更多了。国外大的操作系统，比较成功的一般需花费45人年—105人年，如PDP—10的操作系统TOPS—10（Total Operating Syetem—10）就用了七，八年时间[1]。IBM公司为了使其生产的计算机能为用户所接受，为其计算机所配的软件在1953年为支持650机提供了10000行代码，1962年为支持7090机提供了100000行代码，增加到十倍。1964年为支持360系列提供了1000000行代码，两年就再翻到十倍。1968年则提供了8000000行代码以上[2]。IBM 360 的操作系统的花费是 2 亿5千万美元。可见计算机软件代价的惊人。

实现软件耗费巨大的原因在于它完全是一种“手工”劳动。由于各种计算机之间的指令代码，存贮组织结构等很不相同，因此每生产一台新计算机这种耗费巨大的手工劳动就得重复一遍，许多训练有素的软件工作者被迫束缚在这种重复的、繁重的劳动上，这实在是一种惊人的浪费，不符合多快好省的精神。此外由于生产或研制软件系统的这种手工劳动性质，使产品在很大程度上依赖于软件工作者的手艺和技巧，除了极大影响研制或生产的时间周期之外还极其严重地影响软件系统的质量。软件系统从它初步交付使用到它真正“好用”要拖一个很长很长的尾巴，个别庞大的系统或质量差的系统简直是千疮百孔补不胜补，只好一直拖

此为试读，需要完整PDF请访问：www.ertongbook.com

下去，甚至作废了事。例如国外的花费了几千人年的大型程序设计系统就是这样[8]。一些正常使用的系统为了求得相当的质量，在调整、维护、扩充上也是代价很大的。国产DJS—21机的ALGOL60系统在正式交付使用之后还断续地由不同的单位花了几年时间来逐步完善，机器时间在几百小时以上。

软件系统的这种耗时，耗力，质差的状况国外有把它叫做“软件危机”的[4],[6]，各国都在想方设法致力于摆脱这种困境。目前考虑到的趋势，一种是改进写软件的工具，即研制面向机器的较高级系统程序设计语言，如BLISS,PL360,PL/S, PLOB[7]等。由于是“较高级”语言，因此写出的软件质量高，错误少，易写易读易维护，代价较小。由于又“面向机器”，因此目标程序质量好，效率高。另一种趋势是将软件的研制，生产科学化，工程化，即国外所谓的“软件工程”[6]，把软件系统从“工艺品”变成“工业产品”，以此来求得质量高，代价低的软件。研制可移植的软件是软件工程的重要内容。软件具有了可移植性，则当为新机器配软件时，只需花费较少的代价将已有的软件移植上去就可以了，这样一来将解放大批比较熟练的软件工作者，使他们集中力量于研制新软件，把计算机科学提到一个新的高度。

二　可移植性

1.一般讨论

与可移植性（Portability）概念密切相关的是独立于机器（Machine independence）的概念，但它们又是有区别的。关于这两个概念的联系和区分有许多讨论[8],[9]，比较简明通俗的还是G.Goos[10]叙述的。所谓独立于机器是指一个程序的这样一些性质，它使得该程序不依赖于计算机结构的细节如字长，编址办法，寄存器的种类和数目等等。而可移植性则还加上一些要求，使得该程序不依赖于操作系统的特殊的性质，即它能适应大多数计算机所提供的环境，例如比较通常的I/O（输入/输出）设备。

这两种性质用高级语言写的程序都能近似地达到。用高级语言写的程序其依赖于机器的部分有：实算术运算的精度，算术运算值的范围，字符集等。例如I.Dahlstrand举的一个例子：[8]考虑如下一个程序

```
'begin' 'integer'i;
        i:=4000000*4000000;
        Print("i=", i)
'end'
```

它显然和机器无关的。但在一个计算站的四台不同的机器上计算时却给出四种不同的结果。第一台给出正确结果，第二台给出错误结果，第三台给出运行时错误的信息，第四台给出编译时错误的信息。即实质上上述程序是依赖于机器的，更具体地说即和机器的字长有关。诚然上述困难可以在高级语言中象商用语言COBOL那样用图象(Picture)来克服[8]，但其代价是很大的。关于字符集的困难可以通过规定采用简单的字符集（例如48字符集）来克服。

在系统程序设计领域中的可移植问题是本文要考虑的主要问题，一般所谓的可移植性即指这方面来说。经过许多年的探索、实验、研究，迄今已取得了相当的进展。

一个软件系统要可移植即它应适应大多数计算机所提供的环境。这些环境大致是指这样一些内容[9]：

a.机器寄存器的组织结构；

b.机器内存的组织结构；

c.数据集合（aggregate）的寻址机构；

d.I/O设备。

当然在考虑同操作系统的接口关系时还需考虑作业控制语言（Job control language）的不同等[8]。

a.关于寄存器

有无寄存器的，指令从内存取运算对象，结果也放内存（IBM1400系列，IBM1620）。

有单个算术运算寄存器的（DJS—21，IBM7090，CDC3000系列）。

有多个算术运算寄存器的，指令可从寄存器或内存取运算对象（200系列，013，IBM360系列）。

有多个算术运算寄存器，指令的运算对象必须取自寄存器（CDC6000，7000）。

有使用栈（stack）的，算术运算指令的运算对象在栈中（ICL KDF9，BURROUGHS 5000，5500）。

上述寄存器组织结构的不同其影响在于对单算术寄存器所有中间结果必须显式地存贮起来，对栈则不需要，对多寄存器则仅当同时产生的中间结果太多时才需显式存贮。

b.关于内存

最小的地址编码单位可以是字（word）也可以是字节（byte）。字长则有各种规定，如16位，24位，32位，36位，42位，48位，64位等等。

地址有最通常的线性编址，内存由一列连续编号的单元组成（DJS—21，IBM360系列，CDC6000，7000）。有分块线性编址，内存由多个独立的线性编址的模块组成（BURROUGHS 5500）。有分层内存，它有几种独立编址的内存，各具不同的速度，可由程序员显式地控制各层间数据的转换（带扩内的CDC6000）。

c.关于数据集合的寻址机构

通常对数据集合（例如数组）的分量的访问其实在的地址的计算有由程序完成的，也有由硬件完成的（变址修改）。

对于过程的调用包括两步动作，即参数过渡（parameter passing）和状态保护。实现参数过渡与存取数据集合是有关的。对状态保护则有各种不同。当执行一个转向子例程指令时，有通过硬件将有关状态放入一个栈的，有放入寄存器的，也有提供单独的指令以保护某些状态的。

d.关于I/O

I/O是内存和外部设备之间的通讯。它涉及各种字符集，各种外部设备（例如光笔等）。但至少应包括读，写（打印）等。

在考虑可移植的软件时必须使其设计适应上述的各种不同的环境。

2.实现可移植性所涉及的一些方法技术

本节分别介绍实现可移植性时经常用到的一些方法技术。

a.自展（bootstrapping）技术

自展技术是许多系统经常采用的，它的大致的意思是这样：设有一台计算机，我们首先为它设计一个最小最简单的汇编语言B_1，然后为这个B_1写一个汇编程序，用两种方法来写，一是直接用机器指令来写，一是用该汇编语言本身来写，则将后者作为汇编源程序让前者来加工，其加工的结果应同前者相同，这样我们得到一个正确的一级汇编程序AB_1。

其次对B_1加一些适当的设备进行某种扩充，这样得到的语言记为B_2。它的汇编程序应当是AB_1的某种扩充，记为AB_2。AB_2用B_1来写，让AB_1加工，这样得到能运行的AB_2。于是淘汰AB_1，以后的自展以AB_2为基础来进行。

再将B_2加上一些精细的结构成为B_3，它的汇编程序由比AB_1强有力的汇编程序AB_2来加工，得到更强有力的能运行的汇编程序AB_3。

依此做下去，逐级加上一些精细设备，逐级加工，最后获得一个足够有力的合乎需要的汇编语言及其汇编程序。

对自编译语言的编译程序也是通过自展技术来取得的。首先对该语言取它一个子集，子集大小不多不少正好使得子集中所含的数据类型和基本运算够写出该子集的编译程序（例如子集中应包含字符处理，简单的整数算术运算，但不包含实算术运算，过程等）。

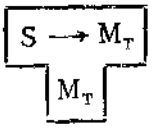
然后用此子集语言写出子集的编译程序，并用手编翻成可运行的机器指令形式（或通过汇编一级再运行）。将前者作为源程序让后者来加工，则编译生成的结果应与后者相同，这也是检验子集编译程序正确性的一个办法。

其次对子集语言加些设备，仿照汇编程序的自展办法一级级扩展上去，最后生成自编译语言的编译程序。

通过自展技术进行自编译的最有名的例子是NELIAC[11]。类似的做法有清华大学周寿昌等的系统，数学所陈冬岳的实验系统TR_0，TR_1，南京大学为200系列机配的自编译系统，以及计算所俞嘉惠的已正式投入运行的111机的自编译系统（第一级）。计算所董蕴美用它在111机上建立了PLOB的编译程序。

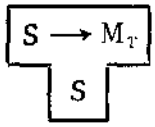
以上介绍的是通过自展将一个简单的原始语言扩充成一个具有高级设备的语言并建立了相应的编译程序。自展还可以用来在不同的机器之间移植编译程序。例如设有自编译语言S，它的自编译编译程序已在宿主计算机H上实现，现在想将该编译程序移植到目标计算机T上。H机的机器语言我们表为M_H，T机的机器语言表为M_T。用Koster的记法[5]，在H机上已实现的（即用M_H写的）将S翻成M_H的编译程序是：

现在要在T机上实现将S翻成M_T的编译程序：

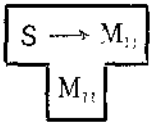
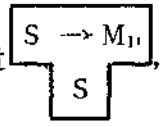


（△）

首先要写一个编译程序：

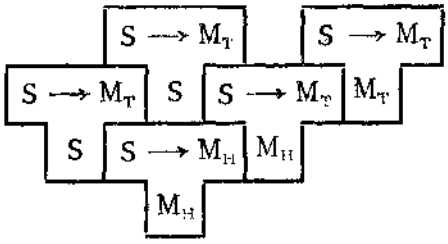


（*）

这比较简单，因当初实现  的过程中曾写过  ，其改动是不大的，

词法分析，语法及语义分析部分基本不动，要改的仅是代码产生部分。

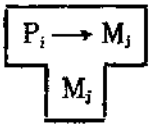
然后在H机上运行它，生成能在宿主机H上运行的将S翻成目标机机器语言 M_T 的编译程序，将(*)再让这个编译程序加工一次即得到所需要的在目标机上运行的编译程序(△)。整个过程要在宿主机H上运行两次，图示如下：



采用上述办法进行移植的一个成功的例子是将瑞士的一个宿主机上的编译程序移往北爱尔兰的一个目标机上，所花的代价是两个人周[12]。这种办法的缺点是产品的效率较低。

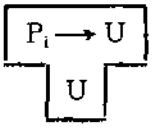
b.UNCOL 问题

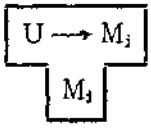
UNCOL是一个很老的问题了，它是面向通用计算机的语言（Universal Computer Oriented Language）的缩写。设我们有m种语言 P_i，i=1，…，m要配在 n 种机器上 M_j，j=1，…，n。则需实现m×n个编译程序：



i=1,…, m
j=1,…, n

如果存在一种中间语言UNCOL（简记为U），使m种语言都先翻成U，再将U翻成各个机器的语言，则编译程序可以减少到m+n个：





i=1,…,m
j=1,…,n

当m，n较大时这种节省是很大的，因为省下一个编译程序就省下了几个人年。但是由于U担当的任务太繁重了，它既要是各种语言 P_i 的目标语言（各种 P_i 都能翻成它，即它应能反映各种语言的特色），又是能描述编译程序的具有系统程序设计功能的语言，而且还能在各种机器上都能以相当的效率来实现。这几乎是不可能的！因此UNCOL虽早在1954年就有人分别

讨论过，1958年就正式提了出来[18]，但一直进展很慢。直到最近几年许多作者对UNCOL加了种种限制以后，使UNCOL不要太大，太复杂，那么“万能”，而是“专”一些，从而取得了许多显著的进展[5]，[8]，[14]。

c.抽象机器和描述语言

对UNCOL进行修正的思想由许多作者提出，考虑的中心思想是不搞万能的UNCOL而是针对不同的具体需要分别设计不同的中间语言。

考虑一个编译程序C，将它划分成独立于机器的部分或仅依赖于语言的部分以及依赖于机器的部分，这种划分可以是各种各样的，大致说如果将C分成词法分析部分，语法和语义分析部分和代码生成部分的话，则第一，二部分可以归到独立于机器的部分，而第三部分可归到依赖于机器的部分。对独立于机器的部分要进行描述则需要决定一些数据类型和相应的基本运算，这些数据类型和基本运算合在一起构成一个中间语言I，有的作者叫它做抽象机器[4]，即它相当于一台假想的计算机的汇编语言。也有的作者叫它描述语言[15]，[5]。不论叫法怎样其大致的意义是相同的。I的设计的特点是它只用于描述C的第一，二部分，决不更多。如果需要描述别的需要移植的软件则按具体需要另行设计I，完全是“量体裁衣”，决不搞象上述UNCOL那样的全能的U。从而使I的效率很高。在具体对某个要移去的目标机实现I时实现的效率也很高。

d.macro技术

macro最初产生于汇编语言，即大家熟知的宏指令和宏调用。它的特点是用于推广汇编语言（被它扩充的汇编语言叫基语言），用种种新的语法成分将基语言扩充得更高级更有力。例如可以对汇编语言定义macro语句：IF ＊＝＊THEN＊ELSE＊。对macro的处理一般由宏加工程序来完成（macro Processor）。基语言一般是汇编语言，后来也推广到了各种高级语言如ALGOL，FORTRAN，PL/1等。对每个基语言定义的macro设备都必须相应地建立宏加工程序，通常它是作为基语言的编译程序的予加工程序来动作的。这种只针对单一的基语言的宏加工程序叫专用宏加工程序。另一类宏加工程序不依赖于基语言，它可以对几个不同的基语言的macro进行加工，它输入一段原文（text）处理后输出一段原文，对基语言的语法形式不加限制。这一种叫做通用宏加工程序[2]，[18]，[17]。

使用macro来描述可移植的软件，移植时对目标机实现这些macro是常用的方法，最早的一个例子是WISP编译程序的移植[20]。在下一节中我们将通过一些已运行的系统来详细介绍这种方法

3.实现可移植性的具体途径

不同的作者对他们自己的可移植系统的描述是各种各样的，他们实现可移植性的具体途径也是有差别的，但从本质来看他们的中心思想是很少不同的。

以编译程序为例，上面提到过它可大致划分为独立于机器的部分（词法，语法及语义分析）和依赖于机器的部分（代码生成）。将不依赖于机器的部分用一个中间语言I来描述，然后再将I对目标机实现。一般可以有三种办法[8]，一种是对之写一个解释程序[18]，一种是对之写一个代码生成程序[19]，一种是通过宏加工程序[15]，[9]。

移植软件比较困难的是对数据类型的处理，因在不同的机器中它们的表示是差别很大的。又要能面对各类机器，又要实现的效率高，这个界限是很难掌握好的。G.Goos[10]，[8]

提出似乎最好的办法应该是将软件分成数据描述部分和算法部分。算法应尽可能独立于机器。数据描述部分为适应各类不同的机器可作适当的修正。这种划分是逻辑上的而不是物理上的。属于数据描述部分的"说明"可以散扦到算法部分当中去。数学所陆汝钤搞的已投入运行的可移植的专用于描述软件的系统汇编语言XHY是体现了这种思想的一个成功的例子。

三　一些已经运行的可移植系统

1.ML/I

ML/I (Macro Language/I) 是P.J.Brown[16]发展的一种通用宏加工程序。其目的是用来作为机器上已配的各种语言的汇编程序或编译程序的予加工程序，以扩充这些已配的语言。扩充是完全面向用户的[21]，例如对于汇编语言可以作些扩充：

```
IF arg1=arg2 THEN arg3 ELSE arg4;
MOVE arg1 TO arg2;
DO arg1 TIMES arg2 REPEAT
JUMP arg1 nl
STOP
MCGO arg1 IF arg2=arg3;
```

其中IF, THEN, ELSE, =, ; , 等是定义符，rg1, arg2, 等是变元。这些扩充了的macro语句形式象高级语言一样有力，易读，方便。

对ALGOL60的扩充可有象下形的macro语句：

'cycle' 〈变量〉:=〈表达式〉 'to' 〈表达式〉 'do' 〈语句〉;

调用时例如 'cycle' i:=1 'to' m 'do' A[i]:=0;

可被ML/I展开成（当然用户应相应定义好宏定义）

```
'begin' i:=1;
    101: 'if' i≤m 'then'  'begin' A[i]:=0;
                                  i:=i+1;
                                  'goto' 101
                          'end'
'end'
```

又如对FORTRAN的扩充：

WHEN〈逻辑表达式〉THEN〈语句〉ELSE〈语句〉

则宏调用

```
WHEN A.EQ.B THEN X=D ELSE READ (2, 100) X
```

可被ML/I展开成（用户已定义了对应的宏定义）

```
                IF (A.EQ.B) GO TO10026
                READ (2,100) X
                GOTO20026
10026  X=D
20026  CONTINUE
```


此为试读，需要完整PDF请访问：www.ertongbook.com