第一部分导论

程序设计语言及系统

程序设计是计算领域的基础。程序描述由物理的和概念的计算机装置所执行的计算步骤，因而它们是广泛的机器自动控制的基础。在一个更高的层次上，程序被用来描述和交流所有类型的算法，以组织和管理复杂性，并用来减弱和舒缓人类的许多繁琐和重复性的任务。

程序设计语言是表达几乎所有这些计算的手段，它们起着双重作用。一方面它们是使思维得以流畅表达的记号（见第二部分中肯尼思·埃弗森 1979 年的演讲），另一方面它们又作为一个抽象计算机器的指示，由之通过自动翻译得到物理机器的指令。因此，毫不奇怪，近乎一半的图灵奖都是授予对程序设计语言、程序设计方法学以及程序设计系统的贡献的。

这部分中的 10 篇演讲是在过去 18 年间准备的。一些作者给出了对其技术贡献的回顾性小结，使他们受到激励的想法以及这些想法所属的范畴。其他人介绍了在他们获奖期间所从事的研究。总而言之，这些演讲把在一个伟大活动时期里这个领域中取得的许多重要进展记入了史册。

读者将会发现，这些演讲中有许多其他图灵奖获得者的成果的摘录。组织这个讨论的一个可能方式将是把相关的内容集中在一起。我已经选择以年代顺序来选择而不是讨论这些论文，以便提供某些历史的方面。

阿兰·珀利斯在 1966 年发表他的头一个图灵奖演讲时，大多数程序设计还是使用编码纸和穿孔卡片进行的。珀利斯展望未来，在承认图灵的计算模型作为对我们理解计算的重要影响的价值，以及 ALGOL 在影响我们思维的重要性之后，珀利斯讨论在程序设计语言和系统方面需要的进一步的进展，他描述了程序员需要有能力来定义更丰富的数据类型和数据结构及其相关操作。这种需要随后通过对抽象数据类型的研究来表达，并且以继续进行中的对于类型系统的思想的观察来表达。因此，这些思想已经成为 LISP 和 SMALLTALK 系统的重要组成部分，也成为程序环境的设计中的重要问题。尽管珀利斯提出的一些问题在随后的研究中已经令人满意地获得解决了，但是其他问题仍然存在。

1972 年的图灵奖获得者是埃德斯盖尔·狄伊克斯特拉，他由于写给《ACM 通信》编辑的那封把使用 goto 语句嘲弄为“把自己的程序弄成一团糟的一个邀请”，而成为当时最有名的人物。狄伊克斯特拉在回顾中提出了他的成就中某些

需要全本请访问 www.ertongbook.com

重要课题。在提及对于 EDSAC 系统，以及对于 FORTRAN、ALGOL 和 LISP 的贡献，并把完全的 PL/I 表征为一个潜在的严重疫病之后，他专注于对可靠软件的创建上。狄伊克斯特拉提出重要的思想，即生产可靠软件的关键是避免引进错误而不是在事后去消灭错误。他说，无错误的程序设计既是经济上重要的，也是技术上可行的。可行性将来自于把我们自己限定于知识上可以管理的程序，现在经济上的论证已是众所周知的了。

在 1974 年唐纳德·克努特发表演讲时，他已经对程序设计和程序设计系统做出了许多贡献，包括他的三卷集《计算机程序设计艺术》。克努特已经从事开发用于高质量排版的极其成功的 TEX 系统。克努特有着学者对于现代思想的历史基础的热爱，以及程序员对于细节的高度关注。在他的演讲中，他探索了艺术和科学的历史思想，并且论证程序设计就是一种艺术形式。好的程序可以而且应当具有优美性和优雅性。在一种真正的“迷”的精神下，克努特断言，程序设计应该是可欣赏的——完全可以仅仅为了好玩而写程序。最后，他敦促为程序设计艺术家们创建“漂亮的”工具，以及鼓励好的风格的语言。

达纳·斯科特和迈克尔·拉宾由于他们在自动机理论方面早期的共同工作而荣获 1976 年的图灵奖。由于后来他们在研究领域上分道扬镳了，因此他们选定对于他们当前的工作分别给出演讲。斯科特当前从事的是程序设计语义学的数学基础，并且建立了指称语义学的理论。在他的演讲中，他专注于奠定其研究成果的基础的重要思想。他概述了影响他在语义学研究中使用的方法的个人专业经验的年代顺序，而后转过来谈成果本身，并且说明了重要的启示以及从中获得的主要结果。斯科特的工作仍然保持着重要性，而且已经繁殖出大量的进一步的研究。

尽管 1977 年图灵奖的主要目的是承认约翰·巴克斯在 FORTRAN 和语言描述方面的贡献，但 FORTRAN 在那之后已经成为计算界熟知的语言，因而巴克斯早已经把他的注意力转移到其他研究课题上。因此，巴克斯利用这个机会，向程序设计的冯·诺依曼风格的基础方法提出挑战。他论证说，传统的程序设计风格要做极大努力来致力于清晰说明数据从主存的来回移动，以及有时来推敲支持这个风格的命名和确定作用域的规则。因此他提出了代替这种风格的变量的函数程序设计风格。这种风格不仅有威力和优美，而且有数学上坚实的语义。

巴克斯的论文对他的演讲做了进一步的补充，而且是既面向专家也面向普通的计算听众的。非专家们首先可能发现吓人的记号的数量，但他们的努力将是有酬报的，即使他们仅仅浏览最形式的部分（而不跳过末尾的小结）。

罗伯特·弗洛伊德 1978 年演讲的主题是程序设计范例。弗洛伊德提出了许多例子，其中包括结构程序设计、递归的共行程序、动态程序设计、基于规则系统以及状态转换机制，并且说明在他自己的研究中他是如何使用其中的某些范例的。

他论证说，重要的是要对使用这些范例保持自觉性，向初学者讲授这些范例，并且为表达用于程序设计的重要范例提供程序设计语言支持。尽管所提供的例子可能会因时间而改变，但是弗洛伊德论文的主题就如同它被写成时那样，对于今天仍然新鲜。

我清楚地记得，在 1980 年 10 月聆听霍尔的图灵奖演讲时，正是 Ada[1] 出现一年左右的时间。当霍尔对他的事业进行平淡叙述时，听众仍然为他介绍的他对于系统和语言的贡献而深深打动了。在详细叙述他早年作为一名软件管理人员的经验之后，叙述了参与导致 ALGOL 68 和 PL/1 问世的设计语言的努力。在列举了一系列的例子说明在语言和系统设计中简单性相对于复杂性的好处之后，他把自己的经验同 Ada 的设计以及他对结果的意见关联起来。读者不应错过霍尔所小结的故事。

在肯·汤普森和丹尼斯·里奇荣获 1983 年图灵奖时，UNIX[2] 从研究实验室推向市场。在他的演讲中，里奇考虑了导致 UNIX 成功的因素，包括它漫长的酝酿时期以及在除商业压力外其他的有关事物不存在的环境中对它的创建。里奇还引述了其他的研究努力，其中同样的培育过程是重要的，而且他还表达了对以下情况的关心，即过多的相关联性可能会抑制创新性，也会抑制自由交流。

肯·汤普森给出了一个分开的但是补充性的演讲。和他之前的狄伊克斯特拉、霍尔、肯努特以及弗洛伊德一样，汤普森也自我介绍为一名程序员，他编写了为克努特在他的图灵奖演讲用做开场白的一个程序，并且说明在弗洛伊德的论文意义下的范例，尽管它是弗洛伊德可能不会选其作为向学生讲授的一个范例。里奇和汤普森两人都对程序设计系统今天被发展的背景提供了轻微的社会评述。

在这一部分中最后一个图灵奖演讲是 1984 年由尼克劳斯·沃思发表的。也许由于作为 Pascal 的发明者而闻名于世，沃思作为程序设计语言的创建者有着显要的记录。在他迄今为止所做出的重要的程序语言设计努力的历史年表中，沃思强调了简单性的重要性。由于他总是注视着实现问题，他也讨论了相互补充的硬件和软件的需要，而且把设计 Lilith 机器中的经验作为设计 Modula-2 的一个参考，沃思指出传授经验的好处以及选择好的工具的价值。

我深感荣幸的是同在这里选辑的这些演讲的所有作者都相识，而且还听过其中四位的课。对于我而言，重新阅读这些演讲的乐趣之一是来看看已经对他们的工作做出重大贡献的这些特别的人们的反思，能以他们作为良师，作为作者，作为知识的导师和作为朋友，我们确实深为荣幸。

苏珊·L·格拉罕
伯克利，加利福尼亚

1 Ada 是美国国防部 Ada 联合设计办公室的一个商标。
2 UNIX 是 AT&T 贝尔实验室的商标。

1966 年图灵奖演讲

算法系统的综合

阿兰 · J · 珀利斯
卡内基理工学院
匹兹堡，宾夕法尼亚

引论

知识和智慧两者都扩展了人类的触及范围。知识导致了计算机，而智慧导致了筷子。遗憾的是，我们的协会过度地陷入前者，后者将需要等候更庄严的一天。

图灵的名望基于什么，而且这一名望将继续存在吗？他证明了一个定理，即对于一个一般的计算装置——后来称为图灵机——是否存在它不能计算的函数？我对此持怀疑态度。更为可能的是，它是基于他所发明和应用的模型的，即他的形式化机制。

这个模型已经引发了想像力，也引发了一代科学家的思索，他证明了导致这些理论的论证基础，他的模型已证明是如此有用，它生成的活动不仅在数学中分布，而且还传遍了很多技术部门。所采用的论证不总是形式化的，而且随之而来的建树不总是抽象的。确实，图灵机最丰硕的成果是可计算函数，即在计算和程序设计下可计算的函数的创建、研究以及计算。这并不奇怪，因为计算机所能计算的比我们还不知道如何确定的要多得多。

我确信，我们大家都同意，这个模型极有价值。历史将会饶恕我没有在我的这个演讲中去使人们注意图灵对于通用数字计算机的发展所做出的影响。数字计算机已经进一步加速了我们对计算理论和实践的涉入。

当然，由于图灵模型的出现，从而导致一些人开始关心计算机问题并使我们从他们的研究中获益。然而，我想只有一样东西能有和图灵的东西同样巨大的影响，这就是称为 ALGOL 的形式机制。许多人会立即表示反对，并指出我们中知道它和使用它的人太少了。虽然令人不快的情况曾经是这样，但问题并

1966年 8月在 ACM 第 21 次全国会议上报告。
作者现地址：Computer Science Department, Yale University, P.O. Box 2158, Yale Station, New Haven, CT 06520-2158。

不在于此。ALGOL 对于计算机科学研究发展的刺激是要紧的，而拥护者的数目无关紧要。ALGOL 还激发我们的思维并且为我们提供论证的基础。

长期以来，我一直捉摸不透，为什么 ALGOL 是我们领域中这样一个有用的模型。也许个中原因是：

（a）它的国际上的支持；

（b）其语法在印刷中的描述的清晰性；

（c）它把汇编和子程序程序设计的重要程序设计特征组合在一起的自然方式；

（d）语言可自然地分解这一事实，使得人们可能会建议并且定义稍微广泛地对语言的一些部分修改，而不破坏其结构和记号的令人印象深刻的和谐性。短语“类似 ALGOL”是有应予承认的意义的。它经常用于程序设计、语言学计算的论证中。ALGOL看起来是一个可持久的模型，甚至在手术之下仍然繁荣昌盛——因为它是可探索的、有弹性的或者是可截肢的；

（e）事实上，它不适合于我们希望来编程的许多任务。

有一件事我是确信的，那就是 ALGOL 并不把它的魅力归功于它诞生的过程，即通过委员会起草。最近，当阐明对 ALGOL 的令人印象深刻的改进的同时，仅仅带来来自我们集体想像的一个缺陷。这些可能是对于 ALGOL 的改进，但它们作为模型并不成功。

自然地，我们应当进行和做出利用他们所提供的改进来纠正 ALGOL 的弱点，也应当想想为什么它们不能刺激我们创造性的能源。我们应当问为什么计算机科学的研究，乃至计算机的实践、工作，没有在它们的影响下大踏步前进？我不会装着我知道整个答案。但是我深信，它们的痴呆性的一个重要部分来自于我们专注于 ALGOL 的错误弱点上。

语言和数据结构的综合

我们知道，设计一种语言的目的是，用于简化通过重要的一类问题建立起来的无数个算法的表达问题。这个设计只有在经过某些改进后，在计算机上运行一段时间后，以及使用现存语言的程序员花费了相当数量的编制时间之后，当施加或者有可能施加对于这类问题的算法之后，才应当被实施。而后，这种语言必须减少一组事务处理的费用来支付它的设计、维护和改进的费用。

后继的语言来自于各种各样的过程：

（a）对于在一种语言中的错误、疏忽或过剩东西的改正，揭示了一个自然的重新设计，它产生一种优良的语言。

（b）对于在一种语言中的错误、疏忽或过剩东西的改正，要求重新设计一种有用的语言。

需要全本请访问 www.ertongbook.com

（c）由任何两种现存的语言通常可以建立起第三种语言，它（i）仍以集成的形式包含来自两者的设施，以及（ii）要求一个与集合的方法以及两者的评价规划相比不那么复杂的方法和评价规则。

有了上述想法，那么从何处开始综合一个后继模型，它将不仅通过机器来改进商业，而且还将把我们的注意力集中在计算本身的重要问题上面呢？

我相信，自然的出发点必须是对数据的组织和分类。这就是说，最低程度，如果不知道它的数据的本性，要建立一个算法是困难的。当我们试图以一种程序设计语言来表示一个算法时，在可以指望来做一个有用的计算之前，必须知道在该语言中算法对于数据的表示。

由于我们的后继语言是一个通用的程序设计语言，它应具有通用的数据结构。依赖于你是如何来考查这个问题的，它既不比你可能想像的更难也不更容易。这个东西应当如何来安排呢？让我们看看在已有的语言中我们做了些什么。其方法如下：

（a）一些“原始”的数据结构，例如整数、实数、在类型上同簇的数组、表、串和文件，被定义到语言中。

（b）在这些结构上，提供了一个“充分”的运算集合。例如算术的、逻辑的、摘取的、赋值的和组合的运算。

（c）任何其他的数据结构都被认为是非原始的，因而必须借助于原始的形式来表示。在非原始的结构中，固有的组织由在原始数据上的运算明确地提供，例如通过实数算术运算来表达一个复数的实部和虚部之间的关系。

（d）这些非原始数据结构的“充分”的运算集合被组织成为过程。

这个扩充过程不可以出错，每种程序设计语言必须允许它方便地使用，因为最终这总是被要求的。然而，如果这个扩充过程使用得太广泛，那么算法经常不能显示它们实际上具有的结构的清晰性。更糟的是，它们的执行就趋向于比需要的还慢。前一种弱点是由于对于这个算法而言，语言的设计是以错误方式进行的。后者的存在是由于在数据上的过分组织以及在执行期间要求管理，而它本可在算法执行之前就完成了。在两种情况下，变量都已由语法和计算规则在错误的时间中做了约束。

我想，大家都知道，我们的语言还没有足够的数据类型。可以肯定地说，在我们的后继模型中，不应当通过添加多一些例如有限分数的新类型和通用的“聚宝箱”结构来修补这个缺点。

在定义函数时的经验应当已经使我们知道该做什么：不要在通用的层次上来专注于完整的定义函数的集合，而应该在这种语言内提供结构，并提供在其有效的定义中以及在程序内使用函数时将遵循的控制。

因此，在后继模型中，应把注意力集中在提供定义数据结构的手段上。但仅这一点本身是不足够的。相伴随的运算的“充分”集合，在为之而描述数据结构的程序中，还必须给出这些运算出现的上下文以及它们的计算规则。

对于数据结构必须提供的一些能力的清单将包括：

（a）结构的定义；
（b）把一个结构赋值给一个标识符，即给标识符以信息单元；
（c）给定结构后，对于一些部分的命名规则；
（d）对于附加到一个标识符的一些单元的赋值；
（e）访问标识符所附属的单元的规则；
（f）组合、复写和删除结构以及单元内容的规则。

在大多数语言中，现在肯定已经以有限的形式提供了这些能力，但是在它们的语法和计算规则内通常是以过于固定的方式来提供的。

我们知道，一种语言的设计者不可能把将驻留于结构中的信息量以及结构内所实施的数据量固定下来。必须允许每个程序自然选择来实现时间和存储之间的一个合意的平衡。我们知道，不存在表示数组或表结构，或串或文件或它们的组合的惟一方式。选择依赖于：

（a）存取的频率；
（b）结构变化中嵌入给定数据的频率，即把新记录结构附加到一个文件中或对数组设边界的频率；
（c）在计算机存储要求中不必要的大块的费用；
（d）在存取数据中不必要的时间的费用；
（e）一个能有序增长的算法表示的重要性，使得结构的清晰性总是存在。

这些选择对于程序员来说是难以做出的,他们肯定不可能在设计层次中做出。

数据结构不可能凭空地创建出来。确实，我们习惯上使用的方法是使用具有固定原始数据结构的背景机器的方法。这些结构是通过实际计算机标识的那些，尽管就定义数据结构来说，背景机器可能是更抽象的。一旦选定了背景机器，由我们的定义所要求的额外结构就必须表示为数据，即作为一个结构的名称或指针。并非所有指针都访问同类型的结构，因为一个程序的段本身是结构，例如“(*x*) 的过程标识符内容”这样的指针确立一类变量，这类变量的值都是过程名。

常数和变量

确实，一种语言的灵活性是由允许程序员在组成中或在执行中对它改变的程度来衡量的。语言中变化的系统发展在程序设计中是一个核心问题，因此在

我们后继者的设计中也是核心问题。经验总是为我们提供从中可确立新变量定义的特殊情况。每个新经验都把我们的注意力集中在对于更大通用性的需要上。分时是我们的新经验之一，它有可能成为一个习惯。分时把我们的注意力集中到系统的管理上，以及由程序员在执行之前、执行之中和执行之后对文本的管理上。同程序的交互将逐渐变得更为灵活，因而我们的后继者不会使这难以实现。我们对会话程序设计的视野比快速围绕时间的转变以及方便的调试辅助包括更多：我们最感兴趣的程序决不会出错且决不是最终的。作为程序员，在希望提供会话式程序设计的一个适当模型之前，必须把新的程序设计同会话的程序设计分开。我坚持认为，新的要求是使语言中的变量成为以前当做固定的东西。现在我不指新的数据类，而是指其值是程序或程序的一部分、语法或语法的一部分以及控制方式的变量。

我们的大多数注意力都用在管理文件的系统发展上，它们改进了整个系统的管理。而在改进计算的管理方面的专注相对较少。前者可以在我们用来写程序的语言之外进行，而对于后者，必须在用来解决问题的程序设计语言内，改进对于可变化性的控制。

在一个程序的处理过程中，一段文本的出现可能是一次，但其执行可能多于一次。这导致了对标识出常量性和可变化性两者的需要。一般地使之具有变量的形式，而通过一个初始化过程使之成为常量，而且经常允许这个过程本身变成为受复制支配的。这个初始化过程是基本的，因而我们的后继者必须有处理它的方法性途径。

我们来考虑某些初始化的实例以及在 ALGOL 中的可变性：

（a）一个分程序的入口。在进入一个分程序入口时，声明进行初始化，但仅仅是对于标识符的某些性质。因此 **integer** *x* 初始化作为一个整数的性质，但不可能用来初始化在这个分程序范围内将改变的某些值。说明 **procedure** *P*(. . .); . . . ;着重地初始化标识符 *P*，但不可能在分程序中改变它。**array** *A* [1 : *n*, 1 : *m*]被赋予一个初始化的结构，但不可能用来初始化其单元的值，或者用来改变附加到标识符 *A* 的结构。

（b）*for* 语句。这些表达式，我将称它们为步进和直到成分，不可能被初始化。

（c）过程说明。这是过程标识符的初始化。在一个过程调用中，它的形式参数像过程标识符那样被初始化，而且它们甚至可以被初始化为值。然而，不同的调用建立对于形参标识符不同的初始化，而不是对于值的不同的初始化模式。

在 ALGOL 中，曾经被认为，对于形式的约束和对于标识符值的约束允许的选择都是适当的。然而，如果我们考查对于形式赋值的运算、形式的计算以及初始化作为在一种语言中合理地确定的重要功能，就可以发现 ALGOL 是有局限性的，甚至在其可利用的选择方面是变幻莫测的。我们应当预期后继语言

不会那样任意和有局限性。

举一个普通的例子。在 *for* 语句中，使用一个如 value E 的结构，其中 E 是一个表达式，作为一个步进成分，将表征表达式 E 的初始化。Value 是一类运算符，用以控制把值约束为一个形式。这个运算符的每一个应用附加有一个自然作用域。

我已经提到过，过程标识符通过说明而被初始化，于是把过程附加到标识符可以通过赋值而改变。我已提到借助于指针如何来实现这一点。当然，还有其他办法。最简单的办法是全然不改变标识符，而是有一个选择下标，它从一个集合中挑出一个过程来。初始化现在定义一个数组形式，例如， procedure array $P[1:k](f_1, f_2, \ldots, f_j)$; . . . begin . . . end ; . . . ; begin. . .end ；调用 $P[j](a_1, a_2, \ldots, a_j)$ 将选择第 i 个过程体来执行。或者，人们可以定义一个 procedure switch $P := A, B, C$ ，以及过程指定的表达式，使得上述调用将选择第 i 个过程指定表达式来执行。上述方法对于某些应用来说过于静态，因此它们缺乏赋值的重要性质：即当一个赋值形式不再可存取时有能力来确定，使得它能另作别用；这种过程的一个可能应用，即被动态地赋值，是一个生成元。假设我们有一个过程，它用于计算 (a) $\sum_{k=0}^{N} C_k(N) X^k$ 作为确定了整数 N 时某个函数 (b) $f(x) = \sum_{k=0}^{\infty} C_k X^k$的一个近似值。现在一旦找到了 $C_k(N)$，则仅仅对于不同的 x 值计算（a）感兴趣。于是我们可能希望来定义一个过程，它从（b）准备（a）。在其初始的执行中，这个过程或者对它本身赋值，或者对某个其他标识符赋值，计算（a）的过程赋值。随后对于该标识符的调用将仅仅产生这个生成的计算。这样一个动态赋值引起了一些有趣的可能性：

（a）这个程序的某个存储可以作为第二个赋值的结果而被释放。

（b）数据存储可以被赋值为其说明或定义被创建的过程标识符的 own。

（c）初始调用能够修改得到的定义。例如，在初始调用中对于一个形式参数换名调用或赋值调用，都将影响所得到的定义的类型。

容易看出，我所论述的这一点对于对初始化，以及对于附加给标识符的形式和值的变化，是有必要附加统一方法的。这是计算过程的一个要求。因而，我们的后继语言对于指挥初始化的动作以及对于它的标识类的变化，必须具有一般的方法。

在会话式程序设计语言中，我们希望实现的动作之一是能对数据值和正文进行系统的或有控制的修改，以便与调试中出现的非系统的修改相区别。这种动作的实现显然意味着正文的某些部分被理解为变量。我们再次通过说明、初始化和赋值实现这一点。因此，在一个分程序的头部，我们可以写声明

real x, s;
arithmetic expression t, u;

在伴随的正文中，$s := x + t;$ 的出现使得算术表达式的值被赋给 $t;$ 例如通过被加到 x 的值和把结果赋作为 s 的值。我们发现，t 可以被输入和存储为一个形式。操作 $+$ 仅在已经应用适当的转换函数之后才能完成，表达式的部分翻译是在经典的“翻译时间”完成的这个事实不应吓住我们。这恰好是以系统的方法来开始面对部分翻译的问题的时候。可以作为变量的正文的自然部分就是由语言的语法单位所标识的。

对于程序未经事先考虑的变化的安排会稍微困难一些。这里的主要问题是原始正文中要加以改变的部分的标识，以及实际上被计算的正文在翻译过程中如何找出它的对应部分。说起来容易，解释地执行原来的正文即可。但是只有通过处于翻译和解释之间的中间解，才能找到令人满意的费用的平衡。我将在下一部分中表达一个观点，当每个程序要求它时，它对实现这个平衡可能会提供某些启示。

数据结构和语法

尽管表结构和递归控制在我们的后继语言中将不起中心作用，但对于 LISP 却是起很大作用的。这种语言在程序员当中引起了争论，且经常出于同一特征而受到谴责和赞扬。在这里我只想说一点，和我所知道的其他语言相比，它的描述以更多的清晰性简洁地揭示出了语言定义的适当成分。 LISP 的描述不仅包括它的语法，还包括它的语法表示为语言的数据结构，而且也把环境数据结构表示为语言的数据结构。实际上该描述对后来的描述设置了障碍，但不是以任何基本的方式。从以前的描述中有可能给出对计算过程的描述，因为一个 LISP 程序使用一些原始函数。而对于其他语言，这个描述的完备性是可能的，但一般不认为它是定义描述的一部分。

对 ALGOL 的考查表明，对于表示 ALGOL 的正文说来，它的数据结构是不适当的，至少不是以适合于语言的计算方案的描述方式。关于描述 ALGOL 程序的环境数据结构，它的不适当性，我们也可以给出同样的评述。

我们的后继语言实现适合于表示语法和环境的处理数据结构的平衡，我把这一点当成是关键性的，以便计算过程可以在语言中明确地表述。

为什么给出这样一个描述是重要的呢？是不是仅仅为了给语言附加“封闭性”的优雅性质，才使得自举可被组织？绝对不是。它是有能力进行会话或计算的程序设计系统的系统构造的关键。

程序设计语言具有语法和一组计算规则，通过把程序表示为计算规则可对之应用的数据而把它们联系起来。这个数据结构是内部的或针对语言语法的计算。为了人们的通信，我们以外部语法来编写程序。我们把该外部语法固定下来。一般假定内部语法与翻译程序及机器密切关，因而它在文献中从未被描述。通常存在一个翻译过程，它把正文从一个外部语法表示转换成一个内部语法表

示。实际上，内部描述中的变化与执行计算规则的机器相比，更为基本地同计算规则本身相关联。计算规则的选择以关键性方式同语言变量的约束时间相关。

这指出了在正文改变情况下有用地组织计算的方法。由于内部数据结构反映了被处理正文的可变化性，让翻译过程选择语法的适当内部表示，而且让一个通用的计算程序以选定的语法结构为基础选择特定的计算规则。因此，必须在外部语法中给出指出变量的提示。例如，**arithmetic expression** *t*; **real** *u*,*v*; 以及语句 *u* := *v*/3***t*; 指出对于 *v*/3 和 *t* 的值不同内部语法的可能性。应当指出，*t* 的特性非常像一个 ALGOL 的形式参数。然而，对于赋值的要求较少。我想，这仅仅指出，形式-实际的赋值同封闭的子程序的概念无关，而且它们已经以确定初始化的范围的方式在过程构造中被统一起来。

在事先未考虑到的变化情况下，内部语法结构的知识使得正文变化时，最少数量的重新翻译和计算规则的改变成为可能。

由于人们要在某种语言中完全考察和构造数据结构以及计算规则，因而它在源语言本身中是合理的。人们可以把其字符串是在源语言允许的那些子集的一个内部语法定义为翻译的目标。这样一个语法，如果选择为接近机器代码，则可通过非常像一个机器的规则来计算。

虽然我已随便地谈了附加于语言的标识符的可变化性，但对于控制的变化性，我还什么都没说。实际上我们没有一个描述控制的方法，所以不能说明它的体制。应当期望我们的后继语言有像 ALGOL 那样多的控制类型，甚至还多些，并行操作是正在被研究的一类控制。另一种是刚刚在语言中出现的分布式控制。我将称之为监控。进程 A 连续地监控进程 B，使得 B 达到一个状态时，A 通过干扰来控制这个进程未来的活动。在 A 内的控制可以写为 **when** *P* **then** *S*; *P* 是在测试之下，在某个定义的范围内总是存在的一个谓词。每当 *P* 为真时，在监视之下的计算被中断，而且执行 *S*。我们希望在一个动作已被执行且它可能使 *P* 为真时，通过测试 *P* 来机械化这个控制，反之却不然。因此我们在定义语言、环境以及计算规则时，包括了在执行期间可加以监控的状态。从这些原始状态开始，其他的东西可以通过程序设计来构造，通过这些原始状态的一点知识，连接在可能点处的测试的安排甚至可以在定义一个程序中的谓词之前就完成了。那样，无须干扰程序本身，就能对程序查错。

语法的变化

在单个语言的范围内，可能会有惊人数量的变化性。而且经验告诉我们，变化的需要将对语言本身的变化施加逐渐增加的压力。设计者不可能预先处理这些变化的精确性质，因为它们是对于尚未解决的问题有待写出程序的结果。具有讽刺性的是，正是最有用和最成功的语言，最多地受到这个变化压力的支

需要全本请访问 www.ertongbook.com

配。幸而，预期较早一类的改变是稍微可预测的。因此，在科学计算中，数的表示和算术是变化的，但表达式的性质不变，除非通过操作数和操作符而变来自于这些源的语法中的改变十分容易处置事实上，在语言中，算术表达式的语法和计算规则还未定义。取而代之的是，在语言中提供语法和计算规则，以便对算术表达式的定义进行程序设计，并且来设置这样的定义的作用域。

在这种持续一个晚上的语言游戏中，惟一的实际困难是计算规则的详细说明它们必须小心给出。例如，在引进矩阵的算术运算中，矩阵表达式的求值应当小心使用临时存储单元而不要实施不必要的迭代

在使用定义时所应用的一种自然技术以语言 X 开始，它将定义考虑为把语法扩大成语言 X' 的语法并且把计算规则给定为一个归约过程，这个过程把 X' 中的任何正文归约为 X 中等价的东西。

还应说明，语法的变化要求语法的一个表示，宜于作为 X 本身的一个数据结构 、

结论

程序设计语言是围绕着变量来构造的——它的运算、控制以及数据结构。由于这些概念对于所有程序设计都是共同的，因此通用语言必须专注于它们有序的发展虽然我们归功于图灵的抽样模型，这个模型也专注于重要的概念，但我们应毫不犹豫地对更复杂的机器和数据进行运算程序员决不应当满足于那样的语言，它允许他们对每一件事情进行程序设计，但却很容易编写毫无兴趣的东西我们的进步是通过在有效性和通用性之间实现的平衡来测定的。由于涉及的计算变化——它确实在变——的本质，即语言变化的适当描述，我们的重点在转移我感到，我们的后继者模型将显示这样的一个变化。计算机科学是一个不安定的婴儿，它的进步既在很大程度上依赖于观点的转变，也在很大程度上依赖于我们当前概念的有序发展。

这里介绍的概念无一是新的；但它们却往往被忘却

我愿感谢协会为我提供第一个图灵奖演讲的特权在结束这个演讲时有没有比说下列话更好的方式呢？即如果图灵今天在我们中间，那么他将在有不同的命名的演讲中阐述不同的观点

分类和主题描述

D.3.1 [软件] 形式定义和理论；D.3.2 [软件] 语言分类一 ALGOL；D.3.3 [软件] 语言构造－数据类型和构造；D.4.3 [软件]：文件系统管理一存取方法

一般术语

语言，算法

后　记

阿兰·J·珀利斯
耶鲁大学计算机科学系

在一个像计算这样动态的活动中，人们不应该指望一篇 20 年前的见解性的论文仍保持为有先见之明的。因此，我高兴和惊讶其内容的明显的解释，既同已经发生过的事情相吻合，而且对于程序设计语言来说仍然在发生着

我仍旧对于一种语言表达的计算模拟赋予巨大的重要性。激起我们想像力的大多数较新的语言对这些模型提供了语法的糖分和语义的丰足，使得我们被诱使来做以前我们不愿做的雄心勃勃的实验。考虑四个模式：流水线（ APL ）、分布式程序设计（如同 Smalltalk 所例证的，更普通地称为面向对向程序设计）、归约程序设计（如同为 LISP、EP 或 ML 所例证的，函数程序设计）以及非过程程序设计(如同由 PROLOG 的逻辑程序设计所验证的）。这些模型很像 25 年前的 ALGOL 那样激起我们的想像力我们没有理由相信，这些是将驱使我们企图做更巨大综合的最后模型。

我的演讲也专注于程序结构中数据结构的重要性。语言发展所沿的轴线之一是数据结构定义和控制的逐渐增加的复杂性虽然 ALGOL 和它的直接“后裔”已经选择数据结构中可定义的可变化性，并导致记录和类型的定义，但上面提到的模型通过它们对于单个复合数据结构的依赖而获得支持，例如表或数组。当然，随着它们的使用变得越来越广泛，数据结构的可变性将变成这些模型更为关心的东西。

工作站、个人计算机以及网络还没有变为程序设计语言系统要集成于其中的普通工具编辑程序是原始的，因而决不可看做是通过它人们就可进入计算的魔门。然而这个演讲确实指出会话式计算以及支持它的语言的重要性同时还指出，把程序表示为数据对于这样的程序设计是很关键的。列表对于这样的表示证实要比数组更好，因为语法是一组嵌套和替换限制，因而表结构修改规则也是这样 APL 和流水线由于数组和列表的不公平性已经受到冷落。 APL 的新版本正在试图重新修整这种不公平性

程序设计正在变为一种无所不在的活动，并且正在做巨大努力来标准化一些语言（也包括模型）迄今为止，我们已经抵抗了这样一个局限性，而且这样做是明智的。新的结构和问题领域肯定支持将驱动我们想像力的新的计算模型，从这些模型和我们现有的模型开始，将会出现像 Ada 这样有作用的下一种语言。和通常一样，我们应当继续避免图灵的陷阱：被迫使用这样的语言，即每一件事情都是可能的，但毫无兴趣的东西很容易出现。

1972 年图灵奖演讲

谦恭的程序员

埃德斯盖尔 · W · 狄伊克斯特拉

[摘自 ACM 图灵奖委员会主席 M · D · 麦希尔洛伊(M. D. McIlroy) 宣读的图灵奖授奖声明，1972 年 8 月 14 日在波士顿的 ACM 年会上发表演讲时]

各处的程序员的工作词汇都散布着由狄伊克斯特拉所创建和强有力地传播的词汇——显示、死死地拥抱、信号灯、较少 goto 的程序设计、结构程序设计。但他对程序设计的影响要比任何词汇所能表示的更为深入。这个图灵奖要感谢的珍贵礼物是狄伊克斯特拉的风格：他把程序设计作为一个高度智力挑战的方法；他雄辩地坚持和实际地论证，应当正确地编写程序，而不仅仅把程序调试成正确的；以及他在程序设计的基础上对于问题的有启发的感知他已发表了大约十余篇论文，既有技术性的，也有思考性的，其中特别要指出的是他在国际信息处理联合会 (IFIP) 上的哲学演讲 [1]，他关于合作顺序进程的已成为经典的论文 [2],以及他关于 goto 语句的令人难忘的控告 [3]。狄伊克斯特拉所写的有影响的信件最近作为关于编写程序的艺术的漂亮专著出版 [4]

我们像评价好的著作一样来评价好的程序，而在这一运动的中心，狄伊克斯特拉坚持创建和思考既有用又不乏美丽的模式

1952 年的头一个春天的清晨，在经历了长时间的巧合之后，我正式地进入程序设计这个专业。而且就我的追忆而言，我是荷兰的头一个程序员就我的记忆所及，至少在我所处的那个世界，最有趣的事情是程序设计专业出现的缓慢性。这种缓慢性是现在难以相信的。但我决无任何怀疑地对于建立这个缓慢

作者现地址：Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712.

1 Some meditations on advanced programming, Proceedings of the IFIP Congress 1962, 535-538; Programming considered as a human activity, Proceedings of the IFIP Congress 1965, 213-217.

2 Solution of a problem in concurrent programming, control, CACM 8 (Sept. 1965), 569; The structure of the "THE" multiprogramming system, CACM 11 (May 1968), 341-346.

3 Go to statement considered harmful, CACM 11 (Mar. 1968), 147-148.

4 A short introduction to the art of computer programming. Technische Hogeschool, Eindhoven, 1971.

性的历史时期缅怀感激，这是由两个生动的回忆所致。

在从事程序设计大约三年后，我和范·温加尔登（van Wijngaarden）有一次面谈，他后来成了我在阿姆斯特丹数学中心的上司——这是一次令我永远感激的面谈。谈话的要点是让我同时在莱登（Leiden）大学学习理论物理，而由于我发现这两项活动越来越难以组合在一起，我必须下决心，要么停止程序设计，努力成为一个受人尊敬的理论物理学家；要么做我的物理研究直到形式上完成为止，而且只须花费极少的努力来成为……成为什么呢？变成一名程序员吗？这是一个受人尊敬的行业吗？终究，什么是程序设计？支持它作为知识上可受人尊敬的领域的健全知识体在哪里？我仍十分清楚地记得，当问及我的硬件同事们关于他们专业的竞争性时，我对他们何等羡慕，因为至少有一点，即他们知道关于真空管、放大器及其他每一件事，而我感到，当面对这些问题时，我将是空手的。当我敲范·温加尔登的办公室的门时，我充满疑虑，我问他可不可以和他说几句？离开他办公室之后几个小时，我就变成另一个人了。在耐心地听了我的问题之后，他同意，直到那时为止，还没有多少程序设计学科，然后他平静地说明，自动计算机已经就在我们身边，我们还处于开始阶段，因而为什么我不可以是在未来岁月中使程序设计成为可尊敬的学科的人之一呢？这在我的一生中是一个转折点，因而我尽可能迅速并正式地完成我的物理学习。当然，上述故事的教益是，当我们向年轻人提出忠告时必须非常小心，有时他们是遵循它的！

两年后，1957 年，我结婚了。荷兰人结婚的习俗是你要说明你的专业，因此我声明我是一名程序员。但是阿姆斯特丹市的市政当局不接受它，因为没有这样的专业。因此，信不信由你，在我的结婚记录中，我的“职业”一栏荒唐地写着“理论物理学家”！

关于在我的国家里程序设计专业出现的缓慢性就谈这些。自那以后，我看到更多的世界，在其他国家里，我总的印象是，除了可能的日期的改变外，其成长的模式都和这差不多。

让我再尝试来更详细地介绍旧时的一些情况，以期对今天的情况有更好的了解虽然我们在进行分析，但应当看到，关于程序设计的真正本质有多少普通的误解却可追溯到遥远的过去。

最初的自动电子计算机都是独一无二的、单台的机器，而且它们全都可在具有令人兴奋的风味的一个实验室环境中找到——旦有了自动计算机的视野，它的实现对于可利用的电子技术就成为一个巨大的挑战，而且有一件事就成为肯定的：我们不能拒绝决定来尝试构造这样一个奇妙装置的人群的勇气。对于这种奇妙的装置它们是：回想中，人们只能惊讶，这些最初的机器全都能行。至少在一段时间里，压倒一切的问题是使得这个机器处于和维持在工作状态中。

对于自动计算的物理方面的偏见仍然反映在这个领域中较老的科学学会的名称中；例如计算机器协会或英国计算机学会，这些名字明显地提及物理装置。

可怜的程序员如何呢？好，诚实地说，他很难被注意。因为，最初的机器都是庞然大物，你几乎动不了它。而且除此之外，它要求如此广泛的维护，因此十分自然的，在人们试图使用这些机器的地方，那就是研制这个机器的地方。其次，程序员的无论怎么说都看不见的工作是没有任何基础的，你可以向参观者展示机器，并且其数量级要比几张代码更惊人。但在这一切当中最为重要的是，程序员对于他自己的工作有一个非常谦虚的观点，他的工作因美妙机器的存在而重要。由于这是独一无二的机器，他深知，他的程序只具有局部重要性，而且还因为它一清二楚地显示，这台机器将只有有限的生命。因此他知道，他的工作只有很少一点可以有长久价值。最后，还有另一个环境，是对程序员对自己的工作态度有长远影响的：除不可靠外，一方面，他的机器通常太慢，而内存通常太小，即他面对的是难穿的鞋。而另一方面，它稍微古怪的代码将迎合大多数非预期的构造。在那些日子里，许多机灵的程序员由狡猾的技巧导出无限知力的满足。借助于它，他设法把不可能挤压成为他的装置的限制。

关于程序员有两个意见是从那些日子开始的。我现在来把它们提出来，过一会我还会再谈及它们。一个意见是，真正有竞争力的程序员应该是喜欢解难题并非常喜欢机敏的技巧的；另一个意见是，程序设计不外乎是在某个方向来优化计算过程的效率。

后一条意见是下列经常情况的结果，即，确实，那时可利用的装备是痛苦的、扎脚的鞋，在那些岁月中，人们经常碰到天真的期望，即一旦有了更强有力的机器，程序设计就将不再是一个问题了。因为把机器推向其极限的奋斗也将没有必要了，而这正是程序设计之全部任务。不是吗？但在之后的几十年中完全不同的事情发生了，更强有力的机器出来了，不仅仅是更强有力一个数量级，而是更强有力好多个数量级。但是代替发现我们自己处于为所有程序设计问题都被解决的狂喜状态，我们发现自己深陷于软件危机之中！怎么回事？

有一个极小的原因：在一两个方面，现代机器基本上来说比旧的机器更难于处理。首先，我们有输入输出中断，它们在不可预料、不可再现的时刻出现；与装着是完全确定的自动机的旧串行机器相比，这是一个巨大的变化，而且许多系统程序员的白发证实了下列事实，关于由这个特性所建立的逻辑问题，我们不宜轻松地去谈论之。其次，我们已经有了以多级存储装备起来的机器，这提出了管理策略的问题，即，尽管关于这个课题有大量的参考文献，但它们仍然是模糊的。关于由于实际机器结构变化引起的额外的复杂性，就谈这些。

但我把这称为一个小的原因：重大的原因是……机器已经变成更强有力好多个数量级。直截了当地说，只要没有机器，程序设计就根本没有问题；当我

们有一些弱的计算机时，程序设计变成一个温和的问题，而现在我们有了庞大的计算机，那么程序设计也变成为同样庞大的问题。在这个意义下，电子工业还没有解决一个问题，它仅仅去创建它们——它创建了使用它的产品的问题。再从另一方面来说，由于可利用的机器的威力增长达 1000 倍以上，应用这些机器的社会雄心也按比例地增长，而可怜的程序员发现自己的工作处于极端和平均之间的紧张区域之中。硬件增长的威力，连同它在可靠性方面或许更巨大的增长，使得问题的解成为更可行。在一些年前，程序员对它们连想都不敢想，而现在，一些年之后，他已经敢去梦想它们，甚至更糟的是，他已经把这样的幻想转换成现实！我们发现我们处于软件危机之中，这是一个奇迹吗？不，肯定不是！而且正如你可能猜测的那样，事先就预测到它；但当然对于有小的预测的麻烦是，仅仅过了 5 年之后，你才真正知道它们是对的。

然后，在(20 世纪)60 年代中期，某些麻烦出现了，所谓第三代计算机登台亮相。正式的文献告诉我们，它们的性价比是主要设计目标之一。但如果把机器各种部件的责任周期记做性能，那么阻止你结束一个设计的可能性很小。在这个设计中，你的性能目标的主要部分是通过必要性可怀疑的内部簿记活动达到的。如果你对于价格的定义是对硬件要付的价格，那么防止你去结束一个这样的设计的可能性很小，这个设计极难以编程。例如，命令代码可能是对程序员，或者对系统的强制，早期的约束决定代表了实际上解决不了的冲突。在更大范围内说，这些不愉快的可能性似乎都变成了现实。

当这些机器被宣告且它们的功能说明书也被公之于众后，我们中许多人必定变得极不愉快，至少我是这样。仅有的合理预期是，这样的机器将在计算机界泛滥，因此最为重要的是它们的设计应当尽可能地好。但是这个设计体现了严重的缺陷，这就是，我感到，通过一个打击，计算科学的进步已被延缓至少十年；正是在那时，在我整个专业生涯中，我度过了最黑暗的一周。或许现在最可悲的事情是，即使在所有这些年令人感伤的经验之后，仍然有这么多的人们真诚地相信，某个自然规则告诉我们，机器应该是这样子。他们通过观察有多少这样的机器被卖出而对他们的怀疑保持沉默，而从这些观察导出错误的安全感，毕竟，这个设计不可能是那么坏的。但在更仔细的观察之后，防线已经和论证有相同的令人信服的强度，即抽烟必定是有益健康的，因为那么多人都在抽。

正是在这样的联系中，我感到遗憾的是，计算领域中的科学杂志不习惯于以像我们评论科学出版物那样来评论最新发布的计算机：至少对机器进行评论将是同样重要的。在这里我要承认，(20 世纪)60 年代初期我曾写过这样的评论，并且有意投到 *Communications* 去，但尽管有这样的事实，我把稿件寄给少数同事征求意见，他们鼓励我这样做。但我不敢，怕对我本人造成困难，或者让编辑部遭受到这样的困难，而它将证明这种困难是极大的。这种忍让在我这

需要全本请访问 www.ertongbook.com