人工智能与软件工程的交叉

C. Rich and R. C. Waters

人工智能和软件工程的交叉多年来一直是个十分活跃的研究领域。

人工智能在软件工程上的应用至少在两个方面是令人感兴趣的。一是，人工智能对软件工程具有实际的重要性。人工智能技术可以提高程序员的生产率和程序的可靠性达一个数量级。

二是，软件工程已被证明是一个推动人工智能研究的领域。试图把人工智能技巧应用到程序设计任务方面已促进了知识表示和自动推理的进一步发展。

一、自动程序设计

人工智能应用于软件工程的最终目标是自动程序设计。在这种极端情况下，通过自动程序设计，用户只要说明想要什么，就会得到完全自动产生的程序。然而这种性能看来不可能在近期实现。因此完全的自动程序设计有时被嘲弄为幻想的自动程序设计。

从某种意义上说，具有一般水平的自动程序设计已问世很长时间了。ACM通讯杂志1958年四月№.4第8页上有一张图表，列出了当时各种自动程序设计系统的可用性。所列的大多数系统现在称为汇编程序，一部分(特别是IBM704的Fortran)是编译程序。显然那时使用的术语自动程序设计似乎不太妥当，然而，实际上是很合理的。比起用绝对的十六进制进行程序设计，编译程序是很有力的。

为了对何谓自动程序设计有一个清楚的理解，进一步考虑短语"用户只要说明想要什么"。特别是提问：谁是用户？用户提出什么？用户的说明与编制一个程序有何区别？根据对这些问题的不同回答，可以预见不同种类的自动程序设计系统。

假定一个大公司的财会部门需要一个程序，图1说明在产生所需程序过程中涉及的代理人。现在所有这些代理人都是人，最终其中某些代理人将由自动程序设计系统所取代。

注意图1并不意味着赞同程序设计过程的任何特殊观点，而主要目的是强调在程序设计过程中不同阶段所涉及的不同种类知识。

这一层次结构的顶层是一个好象只懂得基本财会知识的管理者。他的工作是体会需要什么并提供一个简短的、含糊的要求，从而开始整个过程。这里"含糊"一词用来强调要求是不完全、有岐义和(或)不一致的。

下一个代理人是财会专家。他的工作是根据管理者"含糊"的要求产生一个稍稍完全的、无岐义的、一致的详细要求。这个要求的主要特征是用财会的基本词汇来表达并试图由其它财会人员来评价。因此在顶层附近财会知识起了主要作用。

第三层代理人是系统分析员／设计员。分析员／设计员的工作是设计程序的基本框架并将要求转换为详细规格说明。这个规格说明的关键特征是用程序设计词汇而不是财会词汇表达的。系统分析员／设计员既应懂得很多程序设计知识，又对财会有基本理解，以便在层次的上一部分和下一部分之间建立

接口。

底层的代理人是程序员，他的工作是根据详细的规格说明用高级语言产生代码。不期望程序员懂得财会知识。底层主要涉及程序设计知识。

图1也明显地定义了自动程序设计可以应用的四个不同层次。在最底层，程序员是自动程序设计系统（编译程序）的一个使用者，他们使用该系统可将高级代码自动转换成机器可执行代码。在这种情况下使用者需

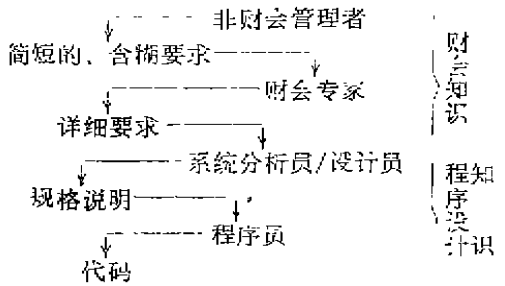

图1 财会系统产生中代理人的层次结构

要的显然是一个程序。

在上面一层，分析员/设计员可以使用以规格说明作为输入的自动程序设计系统。使用这个系统，分析员/设计员用某种形式化的规格说明语言进行表达。这种语言可能类似于程序设计语言，不同之处在于它能直接支持象量词、集合之类的抽象概念，而不强调分析员/设计员说明它们是如何实现的。

再上面一层，财会专家可以使用一个直接从要求产生程序的自动程序设计系统。财会专家既可以用英语也可以用形式化的要求分析语言来表达。在这两种情况下，得对自动程序设计系统输入一些明显不同于程序的信息。自动程序设计系统不仅取代程序员也取代分析员/设计员。因此，该系统不仅需要具有相当的财会知识，也需要具有很强的程序设计知识。

在最高一层，管理者将直接用英语同自动程序设计系统交谈。如上所述，这是自动程序设计的最终目标。注意在这一层大多数困难问题与程序设计无关。该系统将取代财会专家，而且自己将成为一个财会专家。这个系统的主要活动将帮助管理者准确描述在财会范围内他想得到什么。

二、问题分解

完全的自动程序设计是一个太大的问题，不可能一步解决。为了提供可以解决的途径，研究者按不同的方式将问题分解。不同的方法导致了不同的中间产品和结果。

解决问题的一个途径是从图1的底部开始研究。例如，大量的研究工作集中在从规格说明自动地产生代码。这个领域的一个重要研究方法是开发所谓的甚高级语言。

解决问题的另一个途径是从图1的顶部往下着手。这方面的研究工作集中于将自由形式的英语转换为形式化的要求以及要求分析中的知识表示。虽然迄今沿这个方向所做的努力不太多，但这个方法却证明特别重要，因为在程序开发过程早期所犯错误其代价是昂贵的。另外，虽然对于程序实现有许多特殊的工具如编辑器，但帮助要求获取的工具目前却少得可怜。因此能输出可靠要求文档的自动系统有很大价值，即使软件开发过程的其它方面是手工的。

还有一种分解自动程序设计问题的途径，是从图1中纵向地选取一小部分。假定一个领域足够窄，人们就可以构造一个完全的自动程序设计系统。其做法是设计一种专用的面向问题的语言并实现一个编译该语言的专用程序生成器。用户不必管程序的产生过程就可以创造并修改这些程序。在商业和科学应用中程序产生器的例子很多。这是目前自动程序设计的第二种意义。

目前程序产生器的主要缺点是其不灵活。它们在专门领域里工作得很漂亮，但哪怕超出了这个领域一点点就不行。欲使程序产生器逾越其预定的工作范围，其唯一方法是靠人工修改由产生器产生的代码。然而出于以下两个原因这样做是不可能的。首先，大多数产生器产生的代码旨在编译，几乎不可

读。修改这类代码极易出错。其次，程序产生器的主要好处是当要求变化时，只需修改高级语言输入，并再运行一遍产生器，就可修改输出程序。不幸，一旦产生器的输出由人工修改，那么每当再运行产生器时就必须再修改它一次。

研究程序产生器方法的主要目标是覆盖更广的领域且其边界上具有更大的伸缩性。Draco系统和Φnix 系统代表了这个方向的进展。

分解自动程序设计问题的第二种"纵向选取一小部分"的方法是对一些层次提供部分自动化。这样的系统起码应完成内务及其它一些实际任务。超过这个起点可以对不同的程序设计任务提供不同程度的自动化，比如实现、测试或要求分析。设计员/检验员助手、程序员学徒、基于知识的软件助手都是这种方法的例子。这种助手式方法的一个吸引人的特点是可以立即采取办法逐步改进不同种类的自动化。

三、定理证明技术

自动（或帮助）定理证明技术是人工智能研究中较早、较活跃的一个领域。随着Flogd和其它人为程序设计语言建立了坚实的数学基础，人们就开始试图对程序设计任务运用自动定理证明技术。这主要表现为两个形式：程序演绎综合和程序验证。

演绎综合 程序演绎综合基于如下的看法：因为构造性证明的每一步都可以翻译为一步计算，所以构造性证明等价于程序。例如在构造性证明中，无论何时使用实例分析，都必须提供一种显式计算方法，使得实例在任何特殊的情形中都成立。与此相反，非构造性证明只须说明覆盖所有可能情形的实例并将其放在一块。

如下所述，一个构造性的定理证明系统可用来从规格说明导出程序。假设将被导出的程序有输入x和输出y。进一步假定程序的规格说明是：输入x满足前置条件P(x)，且输出将满足后置条件Q(x,y)。这个规格说明转换为下面定理证明系统接收的定理

$$\forall x \exists y [P(x) \Rightarrow Q(x, y)]$$

构造性证明的过程实际上是找到一种方法，用以对任意给定的x找出相应的y。实现这种方法的程序既可以在证明过程中逐步构造，也可以作为一个独立的后阶段从证明中提取。人们可以看到这种方法已将自动程序设计这个困难问题变为自动定理证明这个困难问题。

有许多研究演绎综合的例子。然而当前的发展水平只局限于产生那些具有易于用某种逻辑语言形式表述的规格说明的小程序。因为大程序对应于大的证明，所以综合只局限于小程序。现在的定理证明系统不能找到大的证明，因为在可能的演绎空间中它不能有效地控制搜索证明的过程。

与演绎综合密切相关的研究领域是自动算法设计。在普通的程序综合和算法设计之间几乎相差一个等级。普通程序综合实际上一般含有或多或少的标准算法。相反，算法设计好象只发现完全新的算法。

算法设计的大部分工作只建立演绎综合的一般性框架。它与普通的演绎综合的主要差别在于系统的理解深度。为了支持实际的发现，自动算法设计系统需要详细表达有关的数据对象的数学性质。

算法设计的另一种途径是试图复制由设计人员使用的方法。这导致了模拟设计人员的研究和分析与符号执行在设计过程中的作用之研究。

程序验证 给定一个程序及其形式规格说明，自动定理证明系统(构造性的或非构造性的)可用来验证程序是否满足规格证明。这可以使用两种不同的方法来完成。在第一种方法中，将前置条件正向传给程序，产生一个描述程序净效果的逻辑公式。即一次产生一个语句，办法是前提条件和描述该语句行为的公理相结合。每步产生一个新的逻辑公式，该公式包括该语句之后满足的所有计算。一旦获得描述整个程序与前置条件交互的逻辑公式，就用自动定理证明系统来验证后置条件由此逻辑公式得出。第二种方法采用相反的方式，将后置条件反向传给程序，然后证明得到的逻辑公式被前置条件所隐含。

象演绎综合法一样，自动程序验证也有局限性，因为目前的自动定理证明系统不能证明那些需要庞大证明过程的定理。有两种途径可用来增加可自动验证的程序的规模。首先，可给与定理证明系统以关于特殊程序设计领域的专用知识（即引理）来帮助验证这个领域的程序。第二，人机交互可用来指导定理证明系统。在这种限制下，这种方法将定理证明系统降低为机器证明检验系统。

因为目前验证一个程序代价很高，且很费劲，所以迄今程序验证的主要应用是一些小的、要求高的程序，比如通讯协议和安全性算法。另一个重要应用领域是可重用数据类型实现的验证。

四、变换方法

目前自动程序设计研究中最活跃的领域是用变换支持程序构造。一个程序变换取出程序的一部分并用另一个新的（变换过的）部分取代之。特别是，程序变换是保正确性的，因为变换中新的部分象老的部分一样计算同样的事情。变换的目的是使得这部分程序在一定意义上更好（比如更有效或更具体）。比如一个变换可能用"X*X"取代X**2"。

通常实现的程序变换有三部分。它有一个与程序的某部分相匹配的模式，由此决定在哪儿应用这个变换。（被变换的程序通常是用一种宽谱语言写成的，该语言包含一个传统结构和类规格说明结构的混合。）它有一组变换可应用条件，进一步限制了变换的应用范围。（在许多变换系统中，可应用变换的条件是同逻辑语言表达的，并用定理证明来判断它们在给定的情形下是否满足。）最后，一个变换有一个（通常是过程的）动作，以便产生一个新的程序部分来取代模式所匹配的部分。

有两种主要的变换：纵向变换和横向变换。纵向变换是指某抽象级的一个表达式用更低抽象级的表达式来定义——例如，定义如何用循环计算某向量元素的个数。横向变换说明同一抽象级的两个表达式之间的等价性——例如，说明加法的交换性。横向变换主要用于提高效率，同时为纵向变换的应用提供一些合适的条件。

最初使用变换基于下列想法，在大多数程序中，效率和清晰度不能两全其美。通常可能的是，编写程序采用直接反映其意图的方式，从而该程序的正确性是自明的。不幸的是，这样一个程序经常是十分低效的。横向变换可以用来将一个清晰但低效的程序转换成高效的程序。

Burstall和Darlington的变换系统使用不大的一组固定的强功能通用横向变换来改进程序的效率。他们的系统已经证明允许程序员获得清晰度的好处而不必付出程序效率低的代价的可行性。

变换的最一般应用是作为变换实现的基础。这些系统通过将用一种易读、非常高级的广谱语言编制的程序转换为一个高效率的、可直接执行的程序，从不同方向探讨程序正确性自明的概念。为了做到这一点，反复运用一组纵向和横向变换规则直到在最初程序中所有类规格说明的结构由普通结构取代（实现）为止。与Burstall和Darlington的系统不同，这些系统具有一个大型专用变换库。

在很多方面，变换实现系统可视为一种特殊的程序产生器。变换实现系统和程序产生器之间的主要差别是：用一个变换库而不是用一个过程来表示如何实现程序的知识。这使得变换实现系统较容易扩充和修改。然后这带来了一个新问题：如何控制变换的应用。

控制变换应用的主要困难是，在特殊场合下，有许多不同的变换可供使用，选择不同的变换，将得到不同的结果。例如，假如两个变换都可施用于程序的同一部分，那么程序应用的其中一个变换就可以对程序进行修改而另一个变换可以不再应用。决定应用一个而不是另一个变换相当于对实现做决策。

对于纵向变换，因为大多数情形下只有一、两个变换可以应用于一个给定的抽象表达式，所以控制问题就不太严重。然而横向变换的选择一般是很困难的，不得不依靠人的控制。

现有的变换实现系统可以分为两类：一类能力十分有限但不需要用户控制，另一类能完成非常复杂的实现但只能在用户控制下工作。TAMPR和PDS对可以实施变换类

使用简单的控制策略和限制，以便免除用户的控制。PDS非常有趣，因为它强调让用户定义新的抽象术语和变换作为编程过程的一部分。

PSI系统是对复杂编程使用变换实现模块的最早系统之一。PSI变换模块不靠指导进行操作，可以产生所有可能的低级(语言)程序。它假定另一组成部分指导变换的使用。研究复杂的变换实现系统的工作在南加里福尼亚大学的信息科学学院和克斯托学院开展得十分活跃。他们的工作主要集中于减少选择变换过程中用户的指导。

程序变换研究的另一重要方面集中于变换的保正确性性质和变换的知识表示方面。保正确性方法强调变换的形式逻辑基础，使验证成为可能。知识表示方法强调将变换用作整理编程知识的工具。

五、规格说明技术

如上所述，自动程序设计的一个主要问题是用户采用哪种语言来描述所需要的程序。许多研究集中于三种用户界面：自然语言、甚高级语言和例示程序设计。

自然语言规格说明 这方面早期工作的主要教训是，自然语言规格说明的本质困难不是自然语言的语法或语义，而是人们在通讯时对任务使用了非形式表示。例如，本质问题不是理解the inputs to program are与the program's inputs are意思相同，而是理解当用户如下描述时

The RATS transmission times are entered into the schedule.

其真正含义是

Each RATS clock transmission time and transmission length is made a component of a new transmission entry which is entered into the transmission schedule.

根据这些教训，现在大多数自动程序设计研究者试图将自然语言处理问题与非形式化问题分开。当前自动程序设计研究的焦点已经转向设计一种新的形式化语言，而其中语法处理是直接的，但允许语义的非形式化。许多新的形式表示体系正在甚高级语言的指导下开发。

甚高级语言 甚高级语言的原始模型是SETL，SETL支持任何类Algol程序设计语言的大部分标准结构。另外，它还支持两种普通的数据结构：元组和集合。例如，一个映射可视作一个2-元组的集合。SETL也支持程序中全称量词和存在量词的使用。例如，下面是对集合S的每个元素进行某一计算的SETL语句

$(\forall x \in S)$ …end $\forall$;

提供这种表述设施其目的是解放程序员，不必考虑数据结构的详细设计。SETL编译程序将决定数据结构如何实现。这种减轻程序员的担心是使用甚高级语言获得成效的关键。

由于V语言最近的演化，它在许多方面与SETL类似。GIST语言是甚高级语言研究在更宏大方向上的代表。GIST的目标是不仅采用形式化的语法和语义而且提供自然语言的表达。GIST提供不同于类SETL语言能力的两个例子是，参考历史（参考过去过程状态的能力）和限制（限制可接受的全局说明形式的系统行为）。

毫不奇怪，你放在一个甚高级语言中的高级特征越多，就越难以编译。SETL和V的编译程序至少已部分实现。尽管GIST是个十分活跃的研究领域，但其编译程序还没有建成。

当编译象SETL和V这样一些语言时，首要问题是决定如何实现程序中的抽象数据结构。不仅表示数据需要作决定，而且也要决定如何使用循环来实现程序中的量词。一些研究者已经集中于从任何特定的甚高级语言的细节中抽出来的数据结构的实现问题。

SETL编译程序的工作方式有些类似于传统的优化编译程序。目前它在产生有效的代码方面只获得了一定程度的成功。为了选择实现产生更有效的运行时的性能，SETL编译程序将必须对输入程序作深入分析。完成这些分析是当前SETL研究的重点。同时，提供一种描述式语言，程序员可用来告诉SETL编译程序如何实现特殊的集合。虽然这种逐步增自动化的思想是一种较有趣的折衷方案，但实际上仍不满足。问题是假如SETL的所有表达能力都用在写一个程序上，那么程序员作出建议是困难的。

目前编译甚高级语言所热衷的技巧是使用程序变换系统来排除甚高级的结构。例如V语言就使用这个方法。如上所述，这样的变换系统要求对在什么地方使用什么变换提供建议。很不幸，象显式说明的情形一样，程序员及时提供必要的建议是很

困难的。

甚高级语言的另一个重要的发展趋向是建立特殊应用领域的专用语言。象类似于商业数据处理的应用，使用一些合理的直接了当的技术可以成功地编译已开发的一些十分高级的语言。

例示程序设计 与以上讨论有较大差别的一个有吸引力的规格说明设计思想，是通过程序行为的一些例子来描述程序。这种方法的吸引力在于，象自然语言一样，甚至那些没有程序设计知识的人也熟悉一些描述技巧的例子。而且，单个的例子易于理解和修改。

例示程序设计的困难是，尽管例子本身易于理解，但产生的程序应比给定的例子所直接隐含的程序更为一般。重复所给的例子而不做其它事情，这样构造一个程序是十分烦锁的。所期望的是，一个程序应以一种类似于例子但比例子更一般化的方式对所输入的一类数据进行操作。这种概括的困难在于总有多种方式来概括任何一组例子。

因此，任何例示程序设计系统的主要努力都倾向于寻找适当的概括程度。例如，考虑由如下输入输出对所说明的程序，

(ABCD) ⇒ (DDCCBBAA)

很明显，这个例子的输入是一个表，输出中这个表的每一项重复一次并且颠倒。然而，假如将重复和颠倒施用于输入表的元素（而元素本身又是表），那么这种概括也许过头了。另一方面，假定输入的长度必须为四，那么这种概括可能又不够。

例示程序设计的早期研究是即兴的。Summers的工作使之改观，他为这个领域建立了理论基础。多数后期工作基于Summers的方法。

例示程序设计的大部分工作使用Lisp作为目标语言。之所以如此，不在于为任何技巧所固有，而是因为Lisp程序设计环境更易于编写那些操作程序。

尽管例示程序设计最初产生了许多激动人心的结果，但现在似乎只有理论意义。问题是这种技术难以推广到具有现实复杂性的程序。很不幸，随着输入输出对变得越来越复杂，对它们进行适当概括的问题也变得复杂得不得了。

只有两种基本途径可以缓解概括问题，达到可处理的地步。首先，可以提供更多的例子以减少歧义性，包括中间计算步的一些例子。很不幸，当例子的数量变得太大时，比起其它形式化技术来，这变成了一种低效的规格说明方法。另一方面，就可能产生的程序类，可以许一些假设插入例示程序设计系统中。例如，从例子综合机器人程序。Hedrick所描述的系统假定综合程序是一个基于规则的产生式系统。很不幸，当结构化的假定变得这么强时，例示程序设计系统在本质上变成了一类特殊的程序产生器。

使用例子的另一种方法由Tinker系统所说明。Tinker不试图自动地概括例子。而是提供一种程序开发环境，帮助用户完成概括过程。在这种情形，输入输出例子或许可认为是测试实例。

六、智能助手

以上描述的大部分系统企图使程序设计任务完全自动化。另一种途径是帮助程序员而不是取代程序员。这种途径的一个优点是，即使没有达到自动程序设计的最终目标，也可以构造一些有用的系统。

智能助手是程序设计过程的积极参与者。从最近看，机器就象程序员的小伙伴和批评者，保持跟踪程序设计过程中的细节并在日常工作方面予以帮助，让程序员关心更重要的部分。然而，随着人工智能研究的进步，可能越来越多的任务从程序员转移到机器。

智能助手的研究有两个主要问题。第一个问题是区分程序员和助手的劳动；机器和程序员将如何进行成功地合作和通讯？这导致第二个主要问题。依据最初的原则程序员向助手解释每条指令和判定不可能是冗长的。进而，程序员必需依赖于一个共享知识体以便方便地通讯。

智能助手起码应该懂得程序设计的技术词汇，包括搜索、堆栈、副作用等。正如下面将进一步讨论的，自动程序设计研究的一个重要部分一般涉及到识别和整理机器为参加特殊的程序设计任务所需要的具体知识。

一系列有影响的"思想"性论文进一步标志着智能助手研究的发展过程，它们提出了各种程序设计助手的体系结构。

智能程序设计助手的原型实现包括Designer/Verifier助手，它强调机器的定理证明能力，以及KBEmacs强调在程序构造中使

用标准程序形式的知识库。在程序测试和项目管理领域中也构造了实验性助手。

程序设计导师 一类特殊的智能程序设计助手是程序设计导师。在这一应用中，助手帮助一个程序员新手学习程序设计，而非帮助一个老练的程序员去构造程序。与上述相比，这一应用的一个有趣方面是期望助手比用户具有更多的关于程序设计的知识。

程序设计导师的主要功能是检查由程序员新手写的程序，找出其中的错误，并向学生解释程序，以帮助他学习。为了识别出程序中的错误，需要一些关于可能发生什么错误的理论知识。已经构造（或提出了）各种程序设计导师，而其主要区别在于以什么方式表达关于错误的知识，以及以什么方式识别错误。

第一个可以从学生程序中找出大量错误的系统是Ruth的分析器。这个系统依赖一种文法以说明学生程序中期望哪些算法。这种文法用来分析学生程序的语法。文法规定和程序之间的任何差别都作为错误报告。这种方法的一个关健问题是尽管一个程序事实上是正确的，但也容易不与文法匹配。

最近，Zelinka提出使用一种类似的方法，不同之处在于文法和分析不是应用于程序正文而是应用于一种更抽象而且类似于图形的程序表示。这大大地增强了系统识别什么时候两个程序等价的能力。

发现错误的另一种方法是以显式地表示什么是错的，而不是表示什么是对的。例如MENO-Ⅱ系统包含一个有具体错误模式的预编译库。Sniffer系统使用一个错误识别过程库.在Adam和Laurent所描述的系统中关于错误的知识以图形变换的形式编码。"错误库"方法的一个问题是假如程序包含错误，而这些错误又不在库中，那么系统将显示这个程序是正确的。

检测错误的第三种方法是将程序的执行情况与其规格说明相比较。这既可以通过符号求值也可以通过比较行为的逻辑描述来完成。Proust系统将识别、关于规格说明的推理和关于特定错误的知识组合起来。

对智能导师的需要还引起了最后一个研究领域：即对问题求解中目标和规划的作用的人工智能基础研究。

七、知识表示

在整个人工智能和软件工程领域重复的题目是，什么是程序员知道的知识？在一个自动系统中如何有效地表示知识？程序员的知识自然地分为两部分，程序设计知识和领域知识。

程序设计知识 下面的事实说明从基本的简单的到特殊的复杂的程序设计知识：

(1) 一个堆栈可以用一个数组和一个指针来实现…

(2) 尾递归可以由…转换为一个迭代。

(3) 点集的凸闭包可以由…有效地确定。

这些知识与计算机科学的传统内容（如控制结构、数组、图形）以及与之有关的算法、实现关系有关。这是在计算机科学教科书中可以找到的知识。

知识表示已成为人工智能研究的中心内容。程序设计知识表示的工作已经能够充分利用其它任务中开发出的知识表示技巧。程序设计也是开展知识表示方面新的工作的主要动力。特别是，程序设计知识的表示特别需要下面的内容：

·表示 表示应能够捕获大范围内的程序设计知识，包括数据抽象和过程抽象。

·便于组合 知识单元的组合方法应易于实现，组合的性质应从各部分的性质得到证明。

·语义正确性 表示应基于允许表述正确条件的数学基础。

·机器可操作性 能够使用计算机工具高效地处理表示。

·程序设计语言的独立性 形式化将不依赖于任何特定的程序设计语言的语法。

有许多不同的表示可用于程序设计知识。一个常见的方法是使用程序设计语言图解。很不幸，这个方法在考虑表示、便于组合和语言独立性时有一些严重的缺陷。

其它一些研究者使用流程图，这种方式有极强的语言独立性，但表示能力十分有限，只在一定程度上改进了组合的性质。

现在表示程序设计知识的主要方法是使用程序变换。因为变换通常用特定的程序设计语言的语法表达，所以它不是与语言无关的。也因为变换具有过程成分，所以变换也有语义正确性方面的问题。然而，比起图解方式变换更好表达，更易组合。

Rich的工作使用一个结合程序变换和流程图的殊征与谓词演算的混合表示，这方法具有以上描

述的所有优点。

程序设计知识表示的两个更进一步的问题是大量的程序设计知识如何识别并在库中它们如何关联。Darshowitz的工作利用类比回答这些问题，Rich为程序设计知识库定义了一些有用的分类结构。

领域知识 自动程序设计系统所要求的第二类知识是领域知识——关于目标软件运行的现实世界的知识。下面的事实说明财会领域的知识：

(1) 实际工资等于总工资减去扣除部分。

(2) 如果预计全年收入低于 $500，那么不要求作扣除。

(3) 一年有十二个月。

因为领域知识是表述和理解要求的固有部分，所以它是任何自动程序设计系统的本质内容。因此，在自动程序设计研究中领域知识最近开始得到更多的注意。一个重要的问题是在程序综合处理中领域知识和程序设计知识如何交互。

目前还没有一个以要求作为输入的通用自动程序设计系统。然而有两个非常有趣的专用系统至少在与要求相近的层次上进行工作。这两个系统都含有较强的领域知识。

Draco系统是一个程序产生器的产生器。Draco使用一个易于定义专用程序产生器的变换框架。当使用Draco对某领域定义一个程序产生器时，"领域设计员"遵从传统的程序产生器方法，首先设计一个可以方便描述该领域程序的面向问题的语言，然后说明如何产生用面向问题的语言写的程序。Draco的贡献是提供一组设施，它可以规定以说明为主要形式的程序产生过程。BNF用来定义面向问题语言的语法。横向和纵向变换用来定义它的语义和领域知识。过程只用作最后手段。当一个程序由基于Draco的程序产生器实现时，期望用户对使用哪些变换提供指导。因此最后的程序产生器并不是完全自动的。

Φnix系统具有大量与石油测井的计算有关的知识。比起现在任何其它的自动程序设计系统来，它可能包含更多（或更详细的）领域知识。这类知识以程序变换的形式表达，有些是用过程方式嵌入的。

八、人工智能程序设计

以上的讨论集中于制造更强的嵌入有人工智能的程序设计工具。然而，还有其它的人工智能研究帮助程序员（可能不断提供帮助）的途径。人工智能研究者本身是程序员。从计算机科学的最早时起，人工智能研究者就开发了供自己使用的程序设计工具，这些工具已证明对一般的程序员是非常有用的。

更近些，值得注意的是，现在所熟悉的许多界面特征(比如窗口、鼠标器和菜单)在人工智能程序设计环境中成熟以后才得到普遍使用。同样的情况发生在面向对象的程序设计、逻辑程序设计和面向约束的程序设计中。

来自人工智能程序设计思想的副产物的增长近年来正在加速。其中的重要原因是程序设计行业日益强调快速原型技术和构造大型的、复杂的、前沿的程序。构造这样的程序是一个探索性的课题，而不只是问题，并且不遵从传统的软件生存期模型。其原因是在探索式程序设计中，规格说明和实现不可避免地交织在一起。人工智能程序设计界总是面临着探索式程序设计，因此开始开发适当的语言、环境和硬件设施。

〔**林金明**译自C. Rich and R. C. Waters, Reading in Artificial Intelligence and Software Engineering, Morgen Kaufmann Publishers, Inc., 1986，**仲 源、声 钟**校〕