从逻辑到软件

——唐稚松先生八十寿辰纪念册

二〇〇五年八月

唐先生近照
（２００５年）

作学术报告（１９９５年8月１０日）

何梁何利奖颁奖现场（１９９６年）

唐先生七十寿辰庆祝会
（１９９５年８月）

与Ｐｎｕｅｌｉ等在一起（１９９５年）

春游(右起第６位为ＣＭＵ的Ｊ．Ｍ．Ｗｉｎｇ教授)
（１９８８年）

与夫人童恩健于巴黎枫丹白露宫（１９９７年春）

全家福（２００４年１０月）

与孙子孙女在一起（２００４年１０月）

序

唐稚松先生是我国计算机科学界的一代名师，他的一生都在为科学研究探索真知。早年曾以多带图灵机为计算模型，研究指令系统的递归性，其结论先于国外后来提出并证明的结构程序设计基本定理。他在每一关键时刻发表的具有方向性的论文，对我国计算机科学和软件理论的发展都起着重要的影响。特别是基于时序逻辑的软件工程环境理论和系统方面的研究工作成就，荣获1989年国家自然科学奖一等奖和1996年何梁何利科学技术进步奖。1996年ACM图灵奖得主A. Pnueli教授，在其去美国领奖前夕写给唐的信中说："由于我一贯认为你是时序逻辑的最有力的、最热心的促进者之一，我完全相信，由于使时序逻辑成为一个有深远影响的概念的这一荣誉（指当年图灵奖），你应该分享其中一个有意义的部分"。

唐先生是一位涵养极深且兼具丰富人文哲理情怀的人。他主张立足我国实际的中西文化融合，其学术思想的创新超越就是在他深入分析了北美和欧洲软件工程理论与技术领域长期发展情况的基础上，以中西方哲学相结合的思想为指导而提出的。日本软件工程协会主席岸田孝一先生赞誉为"东方文明对于新的21世纪高新技术发展的一大贡献"。

唐先生是一个专业科学研究工作者，他平常爱引用《中庸》一书中的四句话勉励后学："尊德行而道问学，致广大而尽精微，极高明而道中庸，温故而知新"，体现了唐先生高尚的道德操守和行为风范，实为后人之楷模。

我们非常荣幸能成为唐先生门下的学子。值此唐先生八十寿辰之际，我们收集了先生在不同时期公开发表和未公开发表的部分论著与诗词计20篇，编成此册，以励大家，并表达对先生的崇敬和感激之情。

祝先生健康长寿。

编　者

二〇〇五年八月于北京

目　　录

唐先生简历 …… 1

学术论文选 …… 3

论指令系统的递归性（1965） …… 5
结构程序设计与结构程序语言（1977） …… 24
什么是计算机科学（1978） …… 108
XYZ：A Program Development Environment Based on Temporal Logic（1983） …… 118
程序技术研究三十年（1988） …… 128
XYZ 系统的设计思想（1990） …… 145
A Unified Formal Basis for the CASE Tools System（1993） …… 154
《时序逻辑程序设计与软件工程》（序）（2001） …… 184

其它文章选 …… 195

致中学生的信（1992） …… 197
我国软件产业发展中几个问题（1995） …… 198
XYZ System and It's Philosophical Background（1996） …… 204
XYZ 系统的哲学背景（1998） …… 213

诗词选 …… 233

桃李芬芳 …… 241

唐先生简历

唐稚松，1925年8月7日（旧历）生于湖南长沙，其生父唐子诚，贫病早逝。他自幼过继给寡伯母朱纯轩为子。祖父唐成之，以藏中医书闻名湖南。他是清末民初在长沙最早创办新型学校的人士之一。唐稚松幼年受家庭影响，从小立志献身学术，淡泊功名利禄。

1940—1944年，唐稚松先后在湖南长沙著名的明德中学初中部和省立第一中学高中部学习，文理各科成绩在班中均很突出，尤以诗词受重于师长。1945—1950年在西南联合大学及清华大学哲学系本科学习，1950—1952年在清华大学研究生院学习，专业为数理逻辑。在大学阶段，正遇抗日战争及学生运动时期，名师益友的影响和传统文化的根基使他培养起深厚的民族感情。

1952年8月至1956年10月，唐稚松任中国人民大学数学教研室讲师；1956年10月至1962年10月，任中国科学院数学研究所逻辑室助理研究员；1962年10月至1985年3月，先后任中国科学院计算技术研究所助理研究员、副研究员和研究员；1985年3月起任中国科学院软件研究所研究员。1991年被选为中国科学院学部委员。

从20世纪70年代初开始，唐稚松就已成为我国在计算机科学和软件领域的主要学术带头人，在国内外享有声誉。他在结构程序设计理论、程序语言、形式文法、汉字信息处理等方面都有建树。他在每一关键时刻发表的具有方向性的有关论文，对我国计算机科学和软件理论的发展都起着颇为重要的影响。唐稚松研究工作的最重要贡献是基于时序逻辑的软件工程环境的理论与设计。

唐稚松1956年调入中国科学院数学研究所后，主要从事自动机理论方面的研究工作，发表了关于计算机数学模型方面的一些论文。1965年，唐稚松在一篇关于计算机指令性质的论文中，提出转移指令可用循环替代。文章发表后不久，“文化大革命”开始，文章并未受人注意。不过，从此却开始了唐稚松由数理逻辑研究转入程序设计与理论研究的生涯。

“文化大革命”期间，唐稚松藏身于图书文献之中，首先从事形式文法和编译技术方面的研究，到20世纪70年代中期，他分析总结了国际

上结构程序设计研究方向的大量资料，完成长篇论文“结构程序设计与结构程序语言”。这篇文章对当时我国软件研究工作者影响很大，也为他研究XYZ系统做好了准备。在此基础上，唐稚松设计了一种面向汉字的结构程序语言，他称这种广谱语言为系列化语言族，也就是早期的XYZ系统。

1978年，唐稚松到荷兰参加IFIP（国际信息加工协会）专家组会议。会上，他介绍了XYZ系列化语言族的概念及设计思想，引起与会者的强烈反响。1980年，他当选为来自中国的第一位IFIP专家组成员。

1979年，唐稚松应邀访问了美国斯坦福大学。此后又应邀先后到欧美和日本等国的30多所大学、科研机构讲学访问。他充分利用这些有利条件，分析比较了国际上各种不同流派的情况，一方面博采众长，但又不受其思想的约束，另一方面在自己原来的基础上另辟蹊径，将XYZ系列化语言族改造成适应多种程序设计方式的系列化时序逻辑语言族。在时序逻辑语言及其与软件工程相结合以及多种范型的统一化研究等方面，XYZ系统都属于少数先行者之列。

1983年IFIP巴黎大会上，唐稚松提出了世界上第一个可执行时序逻辑语言——XYZ语言。这一成果被国际著名计算机专家称道为软件工程研究领域中发展可执行时序逻辑的先驱。该语言第一次将状态转换的控制机制引入到逻辑系统之中，又第一次将这种时序逻辑形式化理论与最新软件技术结合起来，它在软件工程这一领域居领先地位。鉴于唐稚松在基于时序逻辑的软件工程环境理论和系统方面的研究工作成就，他先后荣获1989年国家自然科学奖一等奖和1996年何梁何利科学技术进步奖。

唐稚松在国内外发表学术论文逾百篇，专业著作《时序逻辑程序设计与软件工程》上、下册（科学出版社1999、2001）获2003年国家图书奖正式奖和全国优秀科技图书奖一等奖，并有诗词作品《桃蹊诗存》（作家出版社，2005）。

学 术 论 文 选

论指令系统的递归性

（数学学报，15(6)，1965）

本文主要目的之一是：试图应用能行性理论作为工具对数字计算机上的几组指令系统的递归性进行讨论。其中，特别注意到条件转移指令及重复指令的作用。在§2 中讨论的是一组具有原始递归性的指令系统，在§3 中则讨论了几组具有部分递归性的指令系统。

程序控制机器的递归性受其指令系统的递归性所决定。故关于指令系统性能的讨论同时也就是对相应的机器的性能的讨论。本文的另一个目的是介绍一类具有原始递归性的自动机。如所周知，自 Ritchie[4]，Cleave[5] 及 Shepherdson & Sturgis [6] 等文发表之后，差不多对于每一层次的递归性均可找到与之相应的自动机概念，唯独与原始递归性相应的自动机概念尚未见诸文献。我们希望§1 的结果能够补这方面的不足。还不妨指出，我们在这里所介绍的这一机器上的指令系统中包含的指令为传送、送回、相加、重复及停机五条。在§3 的末尾我们进一步证明，如将上述重复指令中一个极为简单的限制取消，则所得的一指令系统即恰好具有部分递归性了。这样即可以清楚地在自动机理论范围内将原始递归性及部分递归性两概念的关系联系起来。

在讨论指令系统时不可避免地要牵涉到这系统所属的机器。当然，可以直接采用 Shepherdson 等 [6] 中 URM 作为基础进行讨论。不过我们没有这样做，其原因是：①我们希望所讨论的机器具有程序内存的特点，这样更接近数字计算机一些；②另一方面我们又希望所讨论的机器与图灵机的概念更为接近。

这里应该提到王浩 [1] 中所定义的图灵机。这一图灵机概念常被认为是最接近数字计算机特征的图灵机概念之一。作者从王浩 [1] 中得到许多启发，但仍感到该文中机器的概念有以下几方面的不够之处：(1) 这机器中的基本运算（[1] 中称为指令），与数字计算机中指令相比较差别太大；(2) 文 [1] 的机器只一条带子，不能表示出现代数字计算机所具有的层次复杂的内部结构。A. Burks 在 [2] 中曾经指出过上面第二点在新的自动机概念中的重要意义，本文作者也有此同感。

要补以上二点之不足我们觉得可以用如下的办法做到：事实上，通常图灵机（包括王浩 [1] 中的机器）的基本运算并不与数字计算机中的指令相当，而只与其微运算相当。正如文 [3] 中所指出的，计算机中的指令是由微运算（通过其控制器的安排）构造出来的。故虽然机器的基本运算是那些微运算，但经过其控制器的构造之后，使得机器所能“识别”的内部语言却是由那些指令排成的程序。作者认为，这一情况同样可以在图灵机上实现。我们将从一通常的多带图灵机出发，用一

种与设计计算机控制器的过程相当相似的办法，构造出一具有数字计算机上许多重要特征的机器。介绍这样一个构造的过程也是本文的一个目的。

本文写作过程中多次与吴允曾同志讨论，得到不少的助益，此外，徐书润、顾毓清两位同志也常与我在一块讨论，并给我指出过一些证明中错处，并此致谢。

§1 具有原始递归性的自动机的构造

我们现在考虑如下的一种多带图灵机：

①带子的条数每次根据所计算的题目的需要而给定，无妨设为 r 条（并约定 $r>6$），给定后在计算过程中不增加，这些带子由上而下排列，并以自然数 i（$i=1, 2, \cdots, r$），作为上起第 i 条带子的地址。并且约定称第 1 至 4 条带为运算带，第 5 条带称为指令寄存带，其余各带均称为内存带。

②每带从右端起向左可任意延伸，带上均以等长分格，假定各带右端是对齐的。

③这图灵机的符号集为 $\{0, 1, *, \Delta\}$，其中以 0 表示空格子。每格中每次必记一个且仅一个符号。

④这机器中包含三个读头，每读头每次注视一带的一个格子。这三个读头中，每一个除了能辨认所注视的格中已存的符号以及向注视格中打印符号集中的符号外，还能作如下的位移操作：

（ⅰ）←：左移一格，

（ⅱ）→：右移一格，如读头已在带的右端，遇到→即原地不动，

（ⅲ）↑跳至上一带的相同格子上。如读头已到了它的活动范围的最上方的带子上，遇到↑即原地不动，此处所谓活动范围将在下面规定，

（ⅳ）↓跳至下一带的相同格子上，如读头已到了它的活动范围的最下方的带子上，遇到↓即原地不动。

我们规定，第 1 读头以 1，…，r 各带为其活动范围，第 2 读头以 1，2，3，4 带为其活动范围，第 3 读头以第 5 带为其活动范围，这些带只能在其相应的活动范围内进行位移操作；同时，我们还规定，第 1，2 两读头以第 1 带的最右一个格子为其开始位置，第 3 读头以第 5 带最右一个格子为其开始位置，除特殊声明的情形外，在一次计算停止后恒让各读头回到其开始位置上。

⑤每一数据以二进位数的形式存于带上，每带只存一数，每次存数时，先将带的右端两个格子空出（留作运算过程中打印标记符号之用），然后在右起第三个格子中存一 *，接下去从右起第四个格子开始存数，由低位向高位依次存放，每格存一位数，“0”对应于表示空格的符号 0，“1”对应于符号 1，在所给之数存完后，在其左边紧接着的一个格子中存一 *。

此外还约定，在第 1 带的最右一个格子中恒存一 *。

⑥每一给定的多带图灵机，恒包含有限个内部状态（其中以足标为 0 的状态符号表示停机）。因共有三个读头，故一给定的这种机器一般由依次排列的若干张的如下形式的过渡表所构成，其中以排在最上方或左方的那一过渡表的第一个内部状

态为其开始状态。

表1

i	0	1	*	Δ
a_{i1}	$\gamma_{i10}b_{i10}a_{i10}$	$\gamma_{i11}b_{i11}a_{i11}$	$\gamma_{i1*}b_{i1*}a_{i1*}$	$\gamma_{i1\Delta}b_{i1\Delta}a_{i1\Delta}$
…	…	…	…	…
a_{in}	$\gamma_{in0}b_{in0}a_{in0}$	$\gamma_{in1}b_{in1}a_{in1}$	$\gamma_{in*}b_{in*}a_{in*}$	$\gamma_{in\Delta}b_{in\Delta}a_{in\Delta}$

此处左上角的$i\in\{1, 2, 3\}$ 以表示这个表是关于第i个读头的，a_{i1}，…，a_{in}表示内部状态，$\gamma_{ijl}\in\{0, 1, *, \Delta\}$，$b_{ijl}\in\{\rightarrow, \leftarrow, \uparrow, \downarrow\}$，$a_{ijl}\in\{a_{i1}, \cdots, a_{in}, a_0\}$。还应指出，下面有时为了使过渡表排得紧凑些，我们采用如下的缩写：即此处b_{ijl}可以是 $\{\rightarrow, \leftarrow, \uparrow, \downarrow\}$ 中元素排成的字，如$\rightarrow^2\uparrow^3$，即表示$\rightarrow\rightarrow\uparrow\uparrow\uparrow$。

下面先构造出一些较简单的多带图灵机：

Y. 其工作情况是：将第1，2两带中所存放的二进位数左端的*按较左的一个*的位置对齐；

这机器的过渡表即

表2

2	0	1	*	Δ
γ_1	$0\leftarrow^3\gamma_2$	$1\leftarrow^3\gamma_2$	$*\leftarrow^3\gamma_2$	任　意
γ_2	$0\downarrow\gamma_3$	$1\downarrow\gamma_3$	$0\downarrow\gamma_4$	任　意
γ_3	$0\leftarrow\uparrow\gamma_2$	$1\leftarrow\uparrow\gamma_2$	$0\uparrow\gamma_7$	任　意
γ_4	$0\leftarrow\gamma_4$	$1\leftarrow\gamma_4$	$*\uparrow\gamma_5$	任　意
γ_5	$*\rightarrow\gamma_6$	$*\rightarrow\gamma_6$	$*\rightarrow\gamma_6$	任　意
γ_6	$0\rightarrow\gamma_6$	$1\rightarrow\gamma_6$	$*\rightarrow^2\gamma_0$	任　意
γ_7	$0\leftarrow\gamma_7$	$1\leftarrow\gamma_7$	$*\downarrow\gamma_8$	任　意
γ_8	$*\uparrow\rightarrow\gamma_9$	$*\uparrow\rightarrow\gamma_9$	$*\uparrow\rightarrow\gamma_9$	任　意
γ_9	$0\rightarrow\gamma_9$	$1\rightarrow\gamma_9$	$*\rightarrow^2\gamma_0$	任　意

A. 其工作情况是：将第1，2两带中所存的左端*已对齐了的二进位数相加，并将其和存在第1带之中，

这机器的过渡表是：

表3

2	0	1	*	Δ
a_1	$0\leftarrow^3\downarrow^2a_2$	$1\leftarrow^3\downarrow^2a_2$	$*\leftarrow^3\downarrow^2a_2$	任　意
a_2	$0\uparrow a_3$	$0\uparrow a_7$	任　意	任　意
a_3	$0\uparrow a_4$	$0\uparrow a_5$	$*\uparrow\rightarrow a_9$	任　意
a_4	$0\leftarrow\downarrow^2a_2$	$1\leftarrow\downarrow^2a_2$	任　意	任　意

续表

2	0	1	*	Δ
a_5	1← ↓²a_2	0← ↓²a_6	任　意	任　意
a_6	1↑ ↓a_2	任　意	任　意	任　意
a_7	0↑a_5	0↑a_8	0↑a_{10}	任　意
a_8	0← ↓²a_6	1← ↓²a_6	任　意	任　意
a_9	0→a_9	1→a_9	*→³a_0	任　意
a_{10}	* ↓a_{11}	* ↓a_{11}	1←a_{10}	任　意
a_{11}	* ↑→a_9	* ↑→a_9	任　意	任　意

J_0. 其工作情况是：将右起第 2 格中记有 1 的带中所存的数码送至第 1 带中，并将右起第 2 格中之 1 消去。

其过渡表是：

表 4

1	0	1	*	Δ
f_{11}	任　意	任　意	*←²f_{12}	任　意
f_{12}	*→²f_{15}	任　意	*←f_{13}	任　意
f_{13}	0←f_{13}	0←f_{13}	0→f_{14}	任　意
f_{14}	0→f_{14}	任　意	*→²f_{15}	任　意
f_{15}	任　意	任　意	*← ↓f_{16}	任　意
f_{16}	0 ↓f_{16}	0←²f_{21}	任　意	任　意
f_{17}	0←f_{22}	1←f_{23}	*→f_{24}	任　意
f_{18}	0→f_{18}	1→f_{18}	*→²f_{19}	任　意
f_{19}	0↑f_{19}	1→f_{19}	*←→f_0	任　意

（a）

2	0	1	*	Δ
f_{21}	任　意	任　意	*←³f_{17}	任　意
f_{22}	0←f_{17}	0←f_{17}	0←f_{17}	任　意
f_{23}	1←f_{17}	1←f_{17}	1←f_{17}	任　意
f_{24}	*→f_{25}	*→f_{25}	*→f_{25}	任　意
f_{25}	0→f_{25}	1→f_{25}	*→²f_{18}	任　意

（b）

以下各较简单的图灵机如其过渡表没有什么特殊之处我们均一概略去。

J_1. 其工作情况是：将第 1 带中所存之数码送至右起第 2 格中记 1 的带中，并将这右起第 2 格中所记之 1 消去。

J_2. 其工作情况是：将右起第 1 格中记 1 的带中所存之数码送至第 5 带中，并将该带右起第 1 格中所记之 1 消去，而在其下一带的右起第 1 格中记 1。

N_1. 其工作情况是：设第 5 带中从右边第 3 格左边第 1 个空格起向左到其左边第一个 * 止这一段格子中所存放的 1 的个数为 α，则在第 α 带的右起第 2 格中记 1。

S. 其工作情况是：首先在右起第 1 格中记有 1 的带中右起第 2 格内记一 *，再设第 5 带中从右边第 3 格左边第一个空格起向左到其左边第一个 * 止这一段格子中所存放的 1 的个数为 α，即在第 α 带的右起第 2 格内记一 Δ。这样一对 * 与 Δ 称为

一对标记，这＊称为与这Δ相应的前标，这Δ称为与这＊相应的后标[①]。

E. 其工作情况是：问右起第 1 格中记有 1 的带中所存之数是否为 0，如是 0，则将此带的下一带中数送至此带，然后将其右起第 1 格中之 1 及右起第 2 格中之＊均消去，再向后找相应的Δ（其找法是，可规定一条带子，开始时是存放自然数 0，在遇到＊后即将其中加 1，以后寻找过程中，遇到Δ减 1，遇到＊再加 1，直到此带中数恢复为 0 时，即找到了所求的相应的Δ）。找到此Δ后，将它亦消去（此时，事实上在这一对标记所在之带的中间各带的右起第 2 格中必不可能出现有标记）。再在此带的下一带的右起第 1 格中记 1，然后停机；如该带中所存之数非 0，则将这带右起第 1 格中之 1 消去，并将此带中所存之数送至下面一带之中，然后在其再下面一带之右起第 1 格中记 1。

B. 其工作情况是：将右起第 1 格中记有 1 的带中数减 1。

D. 其工作情况是：找出与第 1 读头所注视的格中的Δ相应的前标，在存这前标的带中右起第 1 格内记 1。

下面我们还要用到几个带分支的图灵机。对于这种图灵机的 K 个停机状态的足标，我们以 01，02…，0K 来表示，而在表示这种图灵机时，将各种停机状态所可能联结的分支 1，2，…，K，亦记在这机器的符号之右方，比如 $F\begin{cases}1\\2\\3\\4\\5\end{cases}$。

F（亦记成 $F\begin{cases}1\\2\\3\\4\\5\end{cases}$）. 其工作情况是，设第 5 带中从右边第 4 格起向左到其左边第一个空格止这一段格内所存放的 1 的个数为 α，则机器应停机于停机状态 $f_{0\alpha}$。

其过渡表如下：

表 5

3	0	1	*	Δ
f_1	任　意	任　意	$*\leftarrow{}^{4}f_2$	任　意
f_2	$0\rightarrow{}^{4}f_{01}$	$1\leftarrow f_3$	任　意	任　意
f_3	$0\rightarrow{}^{5}f_{02}$	$1\leftarrow f_4$	任　意	任　意
f_4	$0\rightarrow{}^{6}f_{03}$	$1\leftarrow f_5$	任　意	任　意
f_5	$0\rightarrow{}^{7}f_{04}$	$1\rightarrow{}^{7}f_{05}$	任　意	任　意

Z（亦记成 $Z\begin{cases}1\\2\end{cases}$）. 其工作情况是：问右起第 1 格中记 1 的带，上面一带的右起

① 在这种情形下，一般不会要求再向这些记有标记的带中取数或存数，所以不致于要求向这些带的右起第 2 格中记 1。

第 2 格中是否记Δ，如未记Δ，则停机于 z_{D1}，如记有Δ，则先将右起第 1 格记有 1 的带中右起第 1 格中之 1 消去，并让第 1 读头停于其上方一带的右起第 2 格之上，然后停机于 z_{D2}。

我们将要用到上面这些较简的多带图灵机来构成一个较复杂的多带图灵机$\mathscr{G}'$。在构造时，将要用到图灵机的拼及循环。这些运算按文献［7］§68 中所规定，不过，我们在作为循环入口的机器的符号上方所记之点的外面加一对圆括号。

我们将要构造的机器$\mathscr{G}'$应满足如下的条件：任给一有穷长的符号序列，如果：

（ⅰ）其中每一符号是下面规定（1）中所述的$\mathscr{I}'$中的一个指令符号；

（ⅱ）这一符号序列按照下面规定（2）中所述的方式进行编码存入带上；

（ⅲ）初始数据按照下面规定（3）中所述的方式存放；

则机器$\mathscr{G}'$即可按照下面规定（1）的方式执行所给的程序。由于所给的程序是$\mathscr{I}'$中的程序，故在执行有限步以后必停机，此时，所计算的结果即存于规定（3）所指出的带中。

有关$\mathscr{G}'$的三项规定如下：

规定（1），指令系统$\mathscr{I}'$包含下述指令：

传送指令（其指令符号是 $P(1, a)$），含义是：将地址 a 中所存二进位数码送至第 1 带之中，然后执行下一指令。

送回指令（其指令符号是 $P(2, a)$），含义是：将第 1 带中所存二进位数码送至第 a 带之中，然后执行下一指令。

加法指令（其指令符号是 $P(3, a)$），含义是：将第 1，2 两带中数进行相加，其和数存于第 1 及第 a 两带之中，然后执行下一指令。

重复指令（其指令符号是 $P(4, a)$），含义是设本指令符号的编码存于第 i 带之中，则先检查地址 $i+1$ 中所存之二进位数码。如此数为 0，则直接转去执行地址 $a+1$ 中所存之指令，如此数非 0，则将地址 $i+1$ 中原来存之数送至地址 $i+2$ 中存放，然后再执行由地址 $i+3$ 到地址 a 这一段地址中所存之子程序，每执行一次，即将地址 $i+1$ 中所存之数减 1 然后检查此数是否为 0，如果不是 0，则再重复执行由地址 $i+3$ 到 a 中所存之子程序，直到地址 $i+1$ 中所存之数已是 0，再将地址 $i+2$ 中之数送回到地址 $i+1$ 中，然后转去执行地址 $a+1$ 中所存之指令。本指令要求满足以下三条件：（1）$a \geqslant i+3$，（2）一程序中如出现 $P(4, a_1)$，$P(4, a_2)$ 两指令符号，设其所在地址分别为 i_1，i_2，如果 $i_1 < i_2 < a_1$，则要求 $a_2 \leqslant a_1$，（3）对任一 $P(4, a)$，设其所在地址为 i，则在 $i+3$ 到 a 这一段地址中所存之指令，均不以 $i+1$，$i+2$ 为地址码。

停机指令（其指令符号是 $P(5, 0)$），含义即停机。

任给有限个$\mathscr{I}'$中指令符号所排成的一个序列称为一道$\mathscr{I}'$程序，一$\mathscr{I}'$程序中各指令符号即以其在程序中所处的由左向右的顺序为其排列顺序，其中，最左的一个指令符号称为该程序的第一条指令。

规定（2），一程序中的指令符号按如下的方式进行编码存入带中：如该指令符号为 $P(i, a)$，则先将所存带的右起二个格子空出，在第三个格子中存一 *，然后从右起第四个格子起一连 i 个格子中记 1，接着再空出一个格子，然后接着在一连 a

个格子中记 1，最后在其左边紧接着的格子中记一 *，每带存一指令符号的编码。并且，在开始进行计算之前，在程序的第一条指令的编码所存的带中右起第一个格子中先存一个 1。

规定（3），如由$\mathscr{G}'$所计算的函数为一以 x_1，…，x_n 为变元的数论函数 $f(x_1, \cdots, x_n)$，则在开始计算之前，先将第 1 至 5 带空出，然后在第 6 带内存一个自然数 0 的二进位码，在第 7 带内存一个自然数 1 的二进位码，然后在第 8 至 $7+n$ 带内依次存放 x_1，…，x_n 的二进位码，至于其他的常量，或需空出的工作单元则可安排在第 $8+n$ 带以后的各带中，这些数据等均安排好之后，再存放程序中各指令符号的编码，以第 1 条指令的编码存在最前面，然后依次存放，直到所有指令存完为止。并约定，每一程序计算完毕时，必让结果送至第 4 带中存放。

对于满足上述要求的$\mathscr{G}'$，可以直接列出其过渡表，不过，那样做太烦，事实上$\mathscr{G}'$可以用上面已给的各图灵机经过拼与循环而得到。在下面式子中 g_D 表示$\mathscr{G}'$的停机状态：

$$\mathscr{G}' \equiv J_2 \overset{(-)}{F} \begin{cases} N_1 J_0 Z \begin{cases} \dot{J}_2 \\ DBE\dot{J}_2 \end{cases} \\ N_1 \bar{J}_1 Z \begin{cases} \dot{J}_2 \\ DBE\dot{J}_2 \end{cases} \\ \dot{Y}AN_1 \bar{J}_1 Z \begin{cases} \dot{J}_2 \\ DBE\dot{J}_2 \end{cases} \\ SE\dot{J}_2 \\ g_D \end{cases}$$

在下一节中我们即来证明这样具体构造出来的多带图灵机$\mathscr{G}'$是一通用的原始递归自动机。

§2　指令系统$\mathscr{G}'$的原始递归性的证明

如 x 是字母表｛0，1｝中的一个字，则我们以符号 $\underline{x}$ 表示将 x 看成是一个二进位数时相应的十进位码表示式，如 x 是字母表｛0，1，…，9｝中的一个字，则我们以符号 $\bar{x}$ 表示将 x 看成是一个十进位数时相应的二进位码表示式。

如 i 表示一地址，则 $\langle i \rangle$ 即表示这地址 i 内所存的二进位数（存法按§1 中所规定），故 $\underline{\langle i \rangle}$ 即表示地址 i 内所存的数的十进位码。

一数论函数 $f(x_1, \cdots, x_n)$ 称为是$\mathscr{G}'$<u>可计算的</u>，如果存在一$\mathscr{G}'$程序 Γ，使得按规定将任一组 x_1，…，x_n 存入$\mathscr{G}'$中并执行 Γ 至停机以后，有：

$$f(x_1, \cdots, x_n) = \underline{\langle 4 \rangle}。$$

设 Γ_1，Γ_2 均为计算含 n 个变元的数论函数的$\mathscr{G}'$程序，如果对于 n 个变元的任

一种赋值，在执行 Γ_1 以后与执行 Γ_2 以后地址 4 中的二进位数均相同，则称 Γ_1 与 Γ_2 是等价的。

设 Γ 为一 $\mathscr{P}'$ 程序，如果它满足以下二条件：(1) Γ 中恰好出现一次停机指令 $P(5, 0)$，(2) 而且这指令出现在 Γ 的最末端，则我们称 Γ 为一规格化了的程序。

辅理 2.1 每一 $\mathscr{P}'$ 程序必存在一规格化了的程序与之等价。

易证略。

故以下可以假设所考虑的 $\mathscr{P}'$ 程序都是规格化了的（这情况对后面 §3 中仍适用，以后即不再复述了）。下面辅理 2.2，2.3 均易证，故亦略去其证明。

辅理 2.2 递归函数的以下三个模式（Ⅰ），（Ⅱ），（Ⅲ）所规定的初始函数

（Ⅰ）$f(x) = x+1$，

（Ⅱ）$f(x_1, \cdots, x_n) = x_i$，

（Ⅲ）$f(x_1, \cdots, x_n) = q$

都是 $\mathscr{P}'$ 可计算的。

辅理 2.3 设 $h(y_1, \cdots, y_m)$，$g_1(x_1, \cdots, x_n)$，$\cdots$，$g_m(x_1, \cdots, x_n)$ 是 $\mathscr{P}'$ 可计算的递归函数，则按如下的递归函数的模式（Ⅳ）进行代入

（Ⅳ）$f(x_1, \cdots, x_n) = h(g_1(x_1, \cdots, x_n), \cdots, g_m(x_1, \cdots, x_n))$

所得的函数 $f(x_1, \cdots, x_n)$ 也是 $\mathscr{P}'$ 可计算的。

辅理 2.4 设 $g(x_2, \cdots, x_n)$，$h(x_1, x_2, \cdots, x_n, x_{n+1})$ 为已知的 $\mathscr{P}'$ 可计算函数，则由如下的递归模式（Ⅴ）

$$（Ⅴ）\begin{cases} f(0, x_2, \cdots, x_n) = g(x_2, \cdots, x_n) \\ f(x_1+1, x_2, \cdots, x_n) = h(x_1, x_2, \cdots, x_n, f(x_1, \cdots, x_n)) \end{cases}$$

所得的函数 $f(x_1, \cdots, x_n)$ 也是 $\mathscr{P}'$ 可计算的。

证．设 Γ_1，Γ_2 分别为计算 $g(x_2, \cdots, x_n)$ 及 $h(x_1, x_2, \cdots, x_n, x_{n+1})$ 的规格化了的 $\mathscr{P}'$ 程序，其开始地址及结尾地址分别为 r_1，s_1，r_2，s_2，无妨设在 r_1 的左边及 s_1 与 r_2 之间均有足够的空隙可以插入下面所要增加的指令，并设 Γ_1^-，Γ_2^- 为从 Γ_1，Γ_2 中消去末端的停机指令的结果。由此，即可作出计算函数 $f(x_1, \cdots, x_n)$ 的 $\mathscr{P}'$ 程序如下：

表 6

地址	8	$\cdots$	$7+n$	$7+n+1$	$\cdots$	r_1-2n	r_1-2n+1	r_1-2n+2	r_1-2n+3
内容	$\bar{x}_1$	$\cdots$	$\bar{x}_n$		$\cdots$	$p(1, 8)$	$p(2, s_1+2n+3)$	$p(1, 9)$	$p(2, 8)$

$\cdots$	r_1-2	r_1-1	r_1	$\cdots$	s_1-1	s_1	s_1+1	$\cdots$	s_1+2n-4
$\cdots$	$P(1, 7+n)$	$P(2, 7+n+1)$	Γ_1^-			$P(1, 7+n-1)$	$P(2, 7+n)$	$\cdots$	$P(1, 8)$

s_1+2n-3	s_1+2n-2	s_1+2n-1	s_1+2n	s_1+2n+1	s_1+2n+2	s_1+2n+3	s_1+2n+4
$P(2, 9)$	$P(1, 4)$	$P(2, 7+n+1)$	$P(1, 6)$	$P(2, 8)$	$P(4, s_2+5)$	$\bar{0}$	$\bar{0}$

s_1+2n+5	$\cdots$	r_2-1	r_2	$\cdots$	s_2-1	s_2	s_2+1	s_2+2	s_2+3	s_2+4	s_2+5	s_2+6
每格内均存一指令 $P(1, 1)$			Γ_2^-			$P(1, 8)$	$P(2, 2)$	$P(1, 7)$	$P(3, 8)$	$P(1, 4)$	$P(2, 7+n+1)$	$P(5, 0)$

由以上辅理即得。

定理 1 凡原始递归函数都是 $\mathscr{P}'$ 可计算的。

为了证明本定理的逆定理，还需要引进一些概念。一 $\mathscr{P}'$ 程序称为正则的 $\mathscr{P}'$ 程序，如果它满足如下的定义。

(1) 一具有如下形式的 $\mathscr{P}'$ 程序称为 1 层的正则的 $\mathscr{P}'$ 程序

表 7

j	$j+1$	$j+2$	$j+3$
$P(4, j+3)$	$\bar{x}_1$	$\bar{x}_2$	$P(i, a)$

此处 $i \neq 4$，x_1，x_2 为任意自然数。

(2) 设 Γ_1，Γ_2 为二道 m 层的正则的 $\mathscr{P}'$ 程序，则一具有如下形式的 $\mathscr{P}'$ 程序称为 $m+1$ 层的正则的 $\mathscr{P}'$ 程序

表 8

j	$j+1$	$j+2$	$j+3$	$\cdots$	$j+q(m)+2$	$j+q(m)+3$	$\cdots$	$j+2q(m)+2$
$P(4, j+2q(m)+2)$	$\bar{x}_1$	$\bar{x}_2$	Γ_1			Γ_2		

此处 $q(m)$ 的定义如下

$$q(1)=4,$$
$$q(m+1)=3+2q(m)。$$

(3) 只有如上述 (1)，(2) 方式定义的 $\mathscr{P}'$ 程序才是一 m 层的正则的 $\mathscr{P}'$ 程序。

任给 $\mathscr{P}'$ 程序 Γ，如果存在一 m 使得 Γ 是一 m 层的正则的 $\mathscr{P}'$ 程序，则称 Γ 是一正则的 $\mathscr{P}'$ 程序。

任给一 $\mathscr{P}'$ 程序 Γ，如果 Γ 是一规格化了的 $\mathscr{P}'$ 程序，而 Γ^- 是由 Γ 中去掉其结尾停机指令的结果，且 Γ^- 是一正则的 $\mathscr{P}'$ 程序，则称 Γ 是一正规的 $\mathscr{P}'$ 程序；此外，如果 Γ^- 是一 m 层的正则的 $\mathscr{P}'$ 程序，则亦称 Γ 是一 m 层的正规的 $\mathscr{P}'$ 程序。

下面这一事实是显然的，即，对于任一 $\mathscr{P}'$ 程序 Γ，下面这些程序（在有些情形下需要将其中出现的地址码作适当的改变）与 Γ 都是等价的：

表 9

$r-3$	$r-2$	$r-1$	r $\cdots$	s
$P(4, S)$	$\bar{1}$	$\bar{1}$	Γ	

(1)

表 10

$r-3$	$r-2$	$r-1$	r	$\cdots$	s	$s+1$	$s+2$	$s+3$	$s+4$
$P(4,s+4)$	$\bar{1}$	$\bar{1}$	Γ			$P(4,s+4)$	$\bar{1}$	$\bar{1}$	$P(1,1)$

(2)

表 11

$r-7$	$r-6$	$r-5$	$r-4$	$r-3$	$r-2$	$r-1$	r	$\cdots$	s
$P(4,s)$	$\bar{1}$	$\bar{1}$	$P(4,r-1)$	$\bar{1}$	$\bar{1}$	$P(1,1)$	Γ		

(3)

重复以上的事实易证

辅理 2.5 每一$\mathscr{G}'$程序均可化为一与之等价的正则的或正规的$\mathscr{G}'$程序。

故下面考虑定理 1 的逆定理时，即无妨假设所给的程序是正规的$\mathscr{G}'$程序。设 $f(x_1,\cdots,x_n)$ 为一$\mathscr{G}'$可计算的函数，Γ 为计算它的正规的$\mathscr{G}'$程序，设 Γ 的长度为 d，按前面的规定，在计算之前将 $\bar{x}_1,\cdots,\bar{x}_n$ 存于地址 8，⋯，$7+n$ 之中，而地址 $7+n+1$ 到 K 则存其他参数之用。并恒在地址 6，7 中分别以 $\bar{0}$，$\bar{1}$，而程序 Γ 之编码则从地址 $K+1$ 开始存放，因 Γ 长度为 d，故最后一指令存于地址 $K+d$，兹令 $r=K+d$。在任一时刻 t，任选一数 $y(K\leqslant y\leqslant r-1)$，设 $y+i$ 为 t 时刻所执行的指令所存的地址，$g_1,\cdots,g_r$ 分别为 t 时刻地址 1，⋯，r 中所存之二进位数，我们称 $r+1$ 元组（i，g_1，⋯，g_r）为相对于 t，y 的带上情况，并称数

$$p_0^i p_1^{g_1}\cdots p_r^{g_r}$$

为相对于所给 t，y 的带上情况的哥德数。此处 p_j（$j=0,1,\cdots,r$）表示第 $j+1$ 个素数。

再令

$$\omega = p_0^0 p_1^{\bar{0}} p_2^{\bar{0}} p_3^{\bar{0}} p_4^{\bar{0}} p_5^{\bar{0}} p_6^{\bar{0}} p_7^{\bar{1}} p_8^{\bar{x}_1}\cdots p_{7+n}^{\bar{x}_n} p_{7+n+1}^{\langle 7+n+1\rangle}\cdots p_{K+1}^{\langle K+1\rangle}\cdots p_r^{\langle r\rangle},$$

其中，$\langle K+1\rangle$ ⋯ $\langle r\rangle$ 即表示地址 $K+1$，⋯，r 中所存的所给程序 Γ 的指令的编码。此即开始时带上情况的哥德数。

下面，除了要用到一些 Kleene[7] 中常用的原始递归函数，谓词及逻辑联结词将不予说明外，还将用到以下的一些函数和谓词，我们在给出定义的过程中即同时证明这些函数和谓词都是原始递归的。

令

$$l(x) = \max y\leqslant x\left(\left[\frac{x}{10^y}\right]>0\ \&\ \left[\frac{x}{10^{y+1}}\right]=0\right)+1,$$

$$\underline{x} = \sum_{i=0}^{l(x)\dot{-}1}\left(\left[\frac{x}{10^{l(x)\dot{-}(i+1)}}\right]\dot{-}\left[\frac{x}{10^{l(x)\dot{-}i}}\right]\cdot 10\right)\cdot 2^{l(x)\dot{-}(i+1)},$$

$$l'(x) = \max y\leqslant x\left(\left[\frac{x}{2^y}\right]>0\ \&\ \left[\frac{x}{2^{y+1}}\right]=0\right)+1,$$

$$\bar{x} = \sum_{i=0}^{l'(x)\dot{-}1}\left(\left[\frac{x}{2^{l'(x)\dot{-}(i+1)}}\right]\dot{-}\left[\frac{x}{2^{l'(x)\dot{-}i}}\right]\cdot 2\right)\cdot 10^{l'(x)\dot{-}(i+1)}.$$

设 x 表示一指令码，以 $R_1(x)$ 表示 x 中其操作码部分所含 1 的个数，$R_2(x)$ 表示 x 中其地址码部分所含 1 的个数，所以

$$R_1(x) = \sum_{y=0}^{l(x)}\prod_{i=0}^{y}\mathrm{Sg}\left(\left[\frac{x}{10^i}\right]\dot{-}\left[\frac{x}{10^{i+1}}\right]\cdot 10\right),$$

$$R_0(x) = \sum_{i=0}^{l(x)}\mathrm{Sg}\left(\left[\frac{x}{10^i}\right]\dot{-}\left[\frac{x}{10^{i+1}}\right]\cdot 10\right),$$

$$R_2(x) = R_0(x) \dot{-} R_1(x)。$$

令$\in^{|j|}_{|g|}||u|$表示在 u 的素数分解式中以 g 替入 p_j 的指数的结果。

所以$\in^{|j|}_{|g|}||u| = \mu v \leqslant u^{(g+1)}((v)_j = g \ \& \ (x)_{x \leqslant u}(x \neq j \supset (v)_x = (u)_x))$。

然后令

$$\in^{|j_1|, \cdots, |j_{n-1}|}_{|g_1|, \cdots, |g_{n-1}|}||u| = \in^{|j_{n-1}|}_{|g_{n-1}|}|\{\in^{|j_1|, \cdots, |j_n|}_{|g_1|, \cdots, |g_n|}||u|\}。$$

再作以下的原始递归函数和谓词，

$$\tau(z, y, x) \longleftrightarrow R_1((z)_{y+x}) = 4 \ \& \ R_2((z)_{y+x}) = (z)_0 + y,$$

$$\pi_1(z, y) \longleftrightarrow (\exists x)_{x<(z)_0}\tau(z, y, x),$$

$$h(z, y) = \mu x_{<(z)_0}\tau(z, y, x),$$

$$\pi_2(z, y) \longleftrightarrow (\underline{z})_{h(z,y)+y+1} = 0 \ \& \ \pi_1(z, y),$$

$$\pi_3(z, y) \longleftrightarrow \sim \pi_1(z, y)$$

$$\pi_4(z, y) \longleftrightarrow \sim \pi_2(z, y) \& \pi_1(z, y),$$

$$S_1(z, y) = \in^{|0|, \quad |3|}_{|(z)_0+1|, |(z)_{(z)_0-y-1}|}||z|,$$

$$S_1^-(z, y) = \in^{|1|}_{|(z)_{R_2((z)_{(z)_0-y})}|}||S_1(z, y)|,$$

$$S_2^-(z, y) = \in^{|R_2((z)_{(z)_0-y})|}_{|(z)_1|}||S_1(z, y)|,$$

$$S_3^-(z, y) = \in\frac{|1|}{|((\underline{z})_1 + (\underline{z})_2)|}||S_1(z, y)|,$$

$$S_4^-(z, y) = \in^{|h(z,y)+y+1|}_{|(z)_{h(z,y)-y-2}|}||S_1^-(z, y)|,$$

$$S_5^-(z, y) = \in^{|h(z,y)+y+1|}_{|(z)_{h(z,y)-y-2}|}||S_2^-(z, y)|,$$

$$S_6^-(z, y) = \in^{|h(z,y)+y+1|}_{|(z)_{h(z,y)-y-2}|}||S_3^-(z, y)|,$$

$$S_2(z, y) = \in^{|0|, \quad |3|,}_{|h(z,y)+2|, |(z)_{h(z,y)-y-2}|,}\frac{|h(z, y) + y + 1|}{|(\underline{z})_{h(z,y)+y+1} \dot{-} 1|}||z|,$$

$$S_7^-(z, y) = \in^{|1|}_{|(z)_{R_2((z)_{(z)_0-y})}|}||S_2(z, y)|,$$

$$S_8^-(z, y) = \in^{|R_2((z)_{(z)_0-y})|}_{|(z)_1|}||S_2(z, y)|,$$

$$S_9^-(z, y) = \in\frac{|1|}{|((\underline{z})_1 + (\underline{z})_2)|}||S_2(z, y)|,$$

$$S_{10}^-(z, y) = \in^{|0|, \quad |3|}_{|1+R_2((z)_{(z)_0-y})|, |(z)_{R_2((z)_{(z)_0-y})-1}|}||z|,$$

$$S_{11}^-(z, y) = \in^{|0|, \quad |3|,}_{|(z)_0+3|, |(z)_{(z)_0-y-3}|,}\frac{|(z)_0 + y + 1|}{|(\underline{z})_{(z)_0+y+1} \dot{-} 1|}||z|,$$

$$S_{12}^-(z, 1) = \in^{|0|, |1|, |2|, |3|, |3|}_{|0|, |0|, |0|, |0|, |0|}||z|,$$

$$S_{13}^-(z, y) = \in^{|0|, |3|}_{|1|, |(z)_{y-1}|}||z|。$$

下面定义函数$g(\omega, y, t)$，它表示从给定的带上情况的哥德数 ω 出发，以地址 $y+1$ 中所存指令为第 1 步执行的指令在进行了 t 步以后所得带上情况的哥德数，所以

$g(\omega, y, 1) = S_{13}^-(\omega, y)$，

$$g(\omega,\ y,\ t+1)=\begin{cases}
S_1^-(g(\omega,\ y,\ t),\ y), \\
\quad \text{如}\ \pi_3(g(\omega,\ y,\ t),\ y)\ \&\ R_1((g(\omega,\ y,\ t))_{(g(\omega,y,t))_0+y})=1, \\
S_2^-(g(\omega,\ y,\ t),\ y), \\
\quad \text{如}\ \pi_3(g(\omega,\ y,\ t),\ y)\ \&\ R_1((g(\omega,\ y,\ t))_{(g(\omega,y,t))_0+y})=2, \\
S_3^-(g(\omega,\ y,\ t),\ y), \\
\quad \text{如}\ \pi_3(g(\omega,\ y,\ t),\ y)\ \&\ R_1((g(\omega,\ y,\ t))_{(g(\omega,y,t))_0+y})=3, \\
S_4^-(g(\omega,\ y,\ t),\ y), \\
\quad \text{如}\ \pi_2(g(\omega,\ y,\ t),\ y)\ \&\ R_1((g(\omega,\ y,\ t))_{(g(\omega,y,t))_0+y})=1, \\
S_5^-(g(\omega,\ y,\ t),\ y), \\
\quad \text{如}\ \pi_2(g(\omega,\ y,\ t),\ y)\ \&\ R_1((g(\omega,\ y,\ t))_{(g(\omega,y,t))_0+y})=2, \\
S_6^-(g(\omega,\ y,\ t),\ y), \\
\quad \text{如}\ \pi_2(g(\omega,\ y,\ t),\ y)\ \&\ R_1((g(\omega,\ y,\ t))_{(g(\omega,y,t))_0+y})=3, \\
S_7^-(g(\omega,\ y,\ t),\ y), \\
\quad \text{如}\ \pi_4(g(\omega,\ y,\ t),\ y)\ \&\ R_1((g(\omega,\ y,\ t))_{(g(\omega,y,t))_0+y})=1, \\
S_8^-(g(\omega,\ y,\ t),\ y), \\
\quad \text{如}\ \pi_4(g(\omega,\ y,\ t),\ y)\ \&\ R_1((g(\omega,\ y,\ t))_{(g(\omega,y,t))_0+y})=2, \\
S_9^-(g(\omega,\ y,\ t),\ y), \\
\quad \text{如}\ \pi_4(g(\omega,\ y,\ t),\ y)\ \&\ R_1((g(\omega,\ y,\ t))_{(g(\omega,y,t))_0+y})=3, \\
S_{10}^-(g(\omega,\ y,\ t),\ y), \\
\quad \text{如}\ (g(\omega,\ y,\ t))_{(g(\omega,y,t))_0+y+1}=0, \\
\qquad\qquad \&\ R_1((g(\omega,\ y,\ t))_{(g(\omega,y,t))_0+y})=4, \\
S_{11}^-(g(\omega,\ y,\ t),\ y), \\
\quad \text{如}\ (g(\omega,\ y,\ t))_{(g(\omega,y,t))_0+y+1}\neq 0, \\
\qquad\qquad \&\ R_1((g(\omega,\ y,\ t))_{(g(\omega,y,t))_0+y})=4, \\
S_{12}^-(g(\omega,\ y,\ t)),\ \text{其他情形。}
\end{cases}$$

设所给带上情况的哥德数为 v，开始第一步执行的指令所存之地址为 $y+1$，所给程序为一 m 层的正则的 $\mathscr{G}'$ 程序（无妨设其中不出现停机指令）。我们令 $\beta_m(v,\ y,\ j)$ 表示这程序在执行完 j 次最外层的循环时所共进行的步数。

下面施归纳于 m 来证明 $\beta_m(v,\ y,\ j)$ 是一原始递归函数。

$m=1$，即表示所给程序是一 1 层的正则的 $\mathscr{G}'$ 程序。因此

$$\beta_1(v,\ y,\ 1)=2,$$
$$\beta_1(v,\ y,\ j+1)=2+j。$$

所以 $\beta_1(v,\ y,\ j)$ 是一原始递归函数。

设对于任一 m，$\beta_m(v,\ y,\ j)$ 是原始递归函数，下面来证明 $\beta_{m+1}(v,\ y,\ j)$ 也是原始递归的，此时，所给程序是一 $m+1$ 层的正则的 $\mathscr{G}'$ 程序，即具有前面表 8 中

所示的形式。因此

$$\beta_{m+1}(v, y, j) = 1 + \beta_m(v, y+3, (v)_{y+3}) + \beta_m[g(v, y, 1+\beta_m(v, y+3, (v)_{y+3})), y+q(m)+3, (g(v, y, 1+\beta_m(v, y+3, (v)_{y+3})))_{y+q(m)+3}]。$$

为了写起来较清楚计，我们以

$$b(v, y, m, j) = \beta_{m+1}(v, y, j) + \beta_m[g(v, y, \beta_{m+1}(v, y, j)), y+3, (g(v, y, \beta_{m+1}(v, y, j)))_{y+3}],$$

然后

$$\beta_{m+1}(v, y, j+1) = b(v, y, m, j) + \beta_m[g(v, y, b(v, y, m, j)), y+q(m)+3, (g(v, y, b(v, y, m, j)))_{y+q(m)+3}],$$

所以$\beta_{m+1}(v, y, j)$ 也是原始递归函数[3]，因此$\beta_m(v, y, j)$ 是原始递归函数。

前面已设所给开始时带上情况的哥德数为ω，所给程序的第1指令存于地址$K+1$之中，所给计算$f(x_1, \cdots, x_n)$ 的$\mathscr{S}'$程序Γ是一正规的$\mathscr{S}'$程序。至于Γ的层数显然可用如下的原始递归函数来表示

$$S(\omega, K) = \mu x \leq \omega(q(x) + 1 = lh(\omega) \dot{-} K)。$$

再令$\beta(m, v, y, j) = \beta_m(v, y, j)$，故从$\Gamma$的第1条指令开始执行起到最后执行完结尾停机指令止一共进行的步数应为$\beta(S(\omega, K), \omega, K, (\omega)_{K+2}) + 1$。

令

$$\beta^{\Gamma}(\omega, K) = \beta(S(\omega, K), \omega, K, (\omega)_{K+2}) + 1。$$

所以

$$f((\underline{\omega})_2, \cdots, (\underline{\omega})_{1+n}) = (\underline{g(\omega, K, \beta^{\Gamma}(\omega, K))})_4$$

即得。

定理2 凡$\mathscr{S}'$可计算的函数都是原始递归的。

注意．这样得到的$\mathscr{S}'$机器是一通用的计算原始递归函数的机器。在这基础上，也就可以得到与每一给定的$\mathscr{S}'$程序相对应的专用$\mathscr{G}'$机器。其作法是：首先，我们应将$\mathscr{S}'$程序中指令$P(4, a)$ 改为如下的形式$P^*(4, n, b)$。此处b表示一给定的地址，n表示一自然数（称为顺序码），其含义是，将所给程序中从下一指令起到下面第n条指令止这n条指令重复执行，其执行的次数则按照地址b中所存之数而定。每执行一次，将b中数减1，直到b中数为0时，再将b中原来所存之数恢复然后转去执行下面第$n+1$条指令。本指令要求：从本指令以下n条指令中均不以b为地址码；且如$P^*(4, n_1, b_1)$，$P^*(4, n_2, b_2)$ 分别为程序中第i_1，i_2条指令，且$i_1 < i_2 < n_1 + i_1$，则$i_2 + n_2 \leq i_1 + n_1$。设所给的$\mathscr{S}'$程序Γ为

$$P(i_1, a_1) \cdots P(i_K, a_K)。$$

设Γ中所出现的地址码最大者为l，则作一多带图灵机，使之包括l条带子。再作如下的五类子图灵机。

与$P(1, a)(a=1, \cdots, l)$ 相应作子图灵机$\mathscr{P}_1(a)$：其含义是将第a带中数送至第1带。

类似地：与$P(2, a)$，$P(3, a)$，$P(4, n, b)$，$P(5, 0)$ 相应作子图灵机$\mathscr{P}_2(a)$，$\mathscr{P}_3(a)$，$\mathscr{P}_4(n, b)$，$\mathscr{P}_5(0)$。

从而，与所给 $\mathscr{I}^1$ 程序 Γ 相应，所求之专用 $\mathscr{G}^1$ 自动机 C'_Γ（过渡表）即

$$\mathscr{P}_{i_1}(a_1)\cdots\mathscr{P}_{i_K}(a_K)。$$

这样一个由 Γ 作 C'_Γ 的过程事实上即数字计算机中由给定的程序求作专用机控制线路的过程的一种数学描述。

反过来，我们即可将 Γ 的编码看成是 C'_Γ 的一种哥德数，因此，前面所介绍的 $\mathscr{G}^1$ 机器就是一种可模拟任何专用 C'_Γ 机器的通用机。

§3　几种具有部分递归性的指令系统

用 §1 中所介绍的方法，我们再构造一多带图灵机 $\mathscr{G}^2$。

仍从 §1 中所介绍的多带图灵机出发。先作如下的二个多带图灵机（其过渡表略）：

N_2. 其工作情况是：将右起第 1 格中记有 1 的带的右起第 1 格中所记的那个 1 消去，而按第 5 带中指令的地址码，比如 a，在第 a 带的右起第 1 格中记 1。

C（亦写成 $C\begin{cases}1\\2\end{cases}$）. 其工作情况是：比较第 1，2 两带的数，如相等，停机于 C_{D1}，否则停机于 C_{D2}。

然后作多带图灵机 $\mathscr{G}^2$。其约定与 $\mathscr{G}^1$ 在其他方面均相同，只是与 $\mathscr{G}^2$ 对应的指令系统是 $\mathscr{I}^2$。$\mathscr{I}^2$ 与 $\mathscr{I}^1$ 的其他指令均相同。不同只在于将重复指令换成如下的条件转移指令：

条件转移指令（符号是 $P'(4, a)$），其含义是：比较第 1，2 两带中的数，如相等则执行下一指令，否则执行地址 a 中的指令。

故 $\mathscr{G}^2$ 可由已给的多带图灵机组成如下：

$$\mathscr{G}^2 \equiv J_2 \overset{(\cdot)}{F} \begin{cases} N_1 J_0 \dot{J}_2 \\ N_1 \bar{J}_1 \dot{J}_2 \\ YAN_1 \bar{J}_1 \dot{J}_2 \\ YC\begin{cases}\dot{J}_2 \\ N_2 \dot{J}_2\end{cases} \\ \mathscr{E}_D \end{cases}$$

与上节中讨论 $\mathscr{I}^1$ 的情形相似，对于任一指令系统 $\mathscr{I}^K$ 均可定义 $\mathscr{I}^K$ 程序、$\mathscr{I}^K$ 可计算性等概念。下面这定理的证明过程与上面定理 1，2 的证明过程很相似，只是在证明前半部时应增加一关于极小运算（递归模式（VI））的辅理；在证明后半部时关于运算步数的函数可以很简单地用一不受囿的极小运算来定义。还应指出：在 Shepherdson 等［6］中，与定理的前半部相应，证明过一个只指令系统略有不同其余方面均相似的定理，在 Peter［9］中也有过类似的研究，不过［9］中未举出具

体的指令系统。

定理3 凡部分递归函数都是$\mathscr{P}^2$可计算的，反之亦然。

证略。

附带指出，只要将$\mathscr{G}^1$，$\mathscr{G}^2$机器的带长限定为一常数，即可用时序网络来实现。故上面关于$\mathscr{G}^2$的构造可以看成是一种数字计算机的系统设计的数学抽象。

此外，与上一节中谈到的专用$\mathscr{G}^1$机器相似，相对于每一$\mathscr{P}^2$程序亦可构造一实现这程序的专用$\mathscr{G}^2$机器。

下面我们以定理3为基础，讨论几种包含某些特殊的条件指令的指令系统的递归性。

任给两指令系统$\mathscr{P}^j$，$\mathscr{P}^K$，如果对于每一$\mathscr{P}^j$程序存在一$\mathscr{P}^K$程序与之等价，则我们称$\mathscr{P}^j$**包含于**$\mathscr{P}^K$之中，记成$\mathscr{P}^j\subseteq\mathscr{P}^K$；如果$\mathscr{P}^j\subseteq\mathscr{P}^K$而且$\mathscr{P}^K\subseteq\mathscr{P}^j$，则我们称$\mathscr{P}^j$与$\mathscr{P}^K$**等价**，记成$\mathscr{P}^j\equiv\mathscr{P}^K$。如果$\mathscr{P}^j\subseteq\mathscr{P}^K$，而$\mathscr{P}^j\not\equiv\mathscr{P}^K$，则我们称$\mathscr{P}^j$**真包含于**$\mathscr{P}^K$，记成$\mathscr{P}^j\subset\mathscr{P}^K$，显然$\mathscr{P}^1\subset\mathscr{P}^2$。

我们现在考虑如下的指令

加1指令（其符号是$P'(3, a)$）。其含义是：将第1带中所存之二进位数加1，结果存于第1带以及地址a中，然后执行下一指令。

令 $\mathscr{P}^{1\prime}=\{P(1, a), P(2, a), P'(3, a), P(4, a), P(5, 0)\}$，

$\mathscr{P}^{2\prime}=\{P(1, a), P(2, a), P'(3, a), P'(4, a), P(5, 0)\}$。

下面的定理简单故证略。

定理4 $\mathscr{P}^{1\prime}\equiv\mathscr{P}^1$，$\mathscr{P}^{2\prime}\equiv\mathscr{P}^2$。

从逻辑上看$\mathscr{P}^{1\prime}$，$\mathscr{P}^{2\prime}$比$\mathscr{P}^1$，$\mathscr{P}^2$简单些，我们之所以取$\mathscr{P}^1$，$\mathscr{P}^2$作为出发进行讨论是因为$\mathscr{P}^1$，$\mathscr{P}^2$与常见的数字计算机中指令系统相近。

考虑如下的条件指令

下行条件转移指令（其符号是$P(6, a)$）。其含义是：比较第1，2两带中所存的数，如相等则执行下一指令，否则执行地址a中的指令。但要求，如本指令的符号所存地址为i，应有$i<a$。

令$\mathscr{P}^3=\{P(1, a), P(2, a), P(3, a), P(6, a), P(5, 0)\}$。

定理5 $\mathscr{P}^3\subset\mathscr{P}^1\subset\mathscr{P}^2$。

证．用定理2证明过程类似的方法构造函数$g(\omega, K, t)$，$\beta^-(\omega, K)$等，不过此时有以下关系

$$g(\omega, K, t)\leqslant p_{lh(\omega)}^{2^{t\cdot\max((\omega)_0,\cdots,(\omega)_{lh(\omega)})\cdot lh(\omega)}},$$

$$\beta^-(\omega, K)=\mu t\leqslant lh(\omega)((g(\omega, t))_0=r-K),$$

所以，由$\mathscr{P}^3$程序所计算的函数不可能超出初等函数的范围，因此定理得证。

考虑如下的指令。

无条件转移指令（其符号是$P(7, a)$）。其含义是：下一步执行地址a中所存的指令。

再考虑无条件转移指令的一种如下的特殊情形：

返端指令（其符号是$P(7, K+1)$）。其含义是：无条件地转去执行程序的第1条指令。即此处$K+1$表示第1条指令所存的地址。

令 $\mathscr{F}^4=\{P(1,\ a),\ P(2,\ a),\ P(3,\ a),\ P(6,\ a),\ P(7,\ K+1),\ P(5,\ 0)\}$。

定理 6 $\mathscr{F}^4\equiv\mathscr{F}^2$。

证．$\mathscr{F}^4\subseteq\mathscr{F}^2$ 是显然的，故只要证 $\mathscr{F}^2\subseteq\mathscr{F}^4$。为此，事实上只要证对于每一 $\mathscr{F}^2$ 程序，均存在一 $\mathscr{F}^4$ 程序与之等价，而这 $\mathscr{F}^4$ 程序是用一能行的方法从所给 $\mathscr{F}^2$ 程序中将所出现的 $P'(4,\ a)$ 替换掉而得到的。兹设所给的 $\mathscr{F}^2$ 程序为 Γ。

对于 $\mathscr{F}^2$ 程序中所出现的 $P'(4,\ a)$，可分以下两种情形来看，设此指令所存地址为 i，一种情形是 $i<a$ 的，此种情形我们称之为向下转的情形，另一种情形是 $i>a$ 的，则称之为向上转的情形。

对于所给 $\mathscr{F}^2$ 程序中出现的向下转的条件指令，显然即可用 $P(6,\ a)$ 来代替。故只要考虑向上转的条件指令就够了。此时，设 Γ 的第 1 条指令所存的地址为 $K+1$ 而其最后一条指令所占地址为 r。对于向上转的条件指令可用如下的算法来将它替换掉：

先在地址 $r+1$ 中存一指令符号 $P(7,\ K+1)$，设 Γ 中一共出现 d 次向上转的条件指令，即在存储中留下 $2d$ 个空单元（无妨设开始时其中均存放数 $\bar{0}$），设其地址为 $s_1,\ t_1,\ s_2,\ t_2,\ \cdots,\ s_d,\ t_d$。然后按照这些向上转的条件指令在 Γ 中出现的由先而后的顺序，一个一个按下法加以替掉。设前面 $i-1$ 个（$i-1<d$）向上转的条件指令均已替掉了，现在考虑 Γ 中出现的第 i 个向上转的条件指令，设这指令的符号是 $P'(4,\ a_i)$。则对此施行以下的步骤：

（1）将整个程序向后移 4 个地址（程序中出现的有关的地址码均作相应的修改，以后每移动一段程序时各地址码均作如此的修改，不再声明）。在 $K+1$ 到 $K+4$ 这 4 个地址中依次存入如下的指令：$P(1,\ s_i)P(2,\ 2)P(1,\ t_i)P(6,\ a_i+4)$。

（2）从 a_i+4 起再将程序向后移 4 个地址，并在这 4 个空出的地址中存入如下的一段指令：$P(1,\ 6)P(2,\ s_i)P(1,\ 6)P(2,\ t_i)$。

（3）设所考虑的条件指令 $P'(4,\ a_i)$ 此时所占的地址为 j_i，先将 $P'(4,\ a_i)$ 从地址 j_i 中消去，再从地址 j_i 起将程序向后移 6 个地址，然后，在这一段地址中依次存入如下的指令，$P(2,\ s_i)P(1,\ 2)P(2,\ t_i)P(2,\ 2)P(1,\ s_i)P(6,\ r_i)$，此处 r_i 是地址 $r+1$ 经以上各步移动进行修改后所变成的地址。

应用以上步骤将所给程序中出现的向上转的条件指令均消去之后即得到了所求的与 Γ 等价的 $\mathscr{F}^4$ 中的程序。定理证毕。

考虑另一种条件转移指令：

上行条件转移指令（其符号是 $P(8,\ a)$）。其含义是：比较第 1，2 两带中所存的数，如相等则执行下一指令，否则执行地址 a 中的指令，但要求，如本指令的符号所存地址为 i，应有 $i>a$。

令 $\mathscr{F}^3=\{P(1,\ a),\ P(2,\ a),\ P(3,\ a),\ P(8,\ a),\ P(5,\ 0)\}$。

下面为了要证明 $\mathscr{F}^3\equiv\mathscr{F}^2$，先证以下辅理：

辅理 3.1 由辅理 2.2 中（Ⅰ），（Ⅱ），（Ⅲ）三个模式所规定的初始函数都是 $\mathscr{F}^3$ 可计算的；辅理 2.3 中模式（Ⅳ）亦可保证从 $\mathscr{F}^3$ 可计算的函数得到 $\mathscr{F}^3$ 可计算的函数。

证略。

辅理 3.2 设$g(x_2, \cdots, x_n)$，$h(x_1, \cdots, x_{n+1})$ 为$\mathscr{P}^3$可计算函数，则由辅理 2.4 中递归模式（V）所得到的函数$f(x_1, \cdots, x_n)$ 也是$\mathscr{P}^3$可计算的。

证．这是这组辅理中最基本的一个，设Γ_1，Γ_2 分别为计算 $g(x_2, \cdots, x_n)$，$h(x_1, x_2, \cdots, x_n, x_{n+1})$ 的规格化了的$\mathscr{P}^3$程序，令Γ_1^-，Γ_2^- 分别为从Γ_1，Γ_2 中去掉结尾的停机指令并补入$P(1, 4)P(2, 7+n+1)$ 两指令的结果。设Γ_1^-，Γ_2^- 的长度分别为l_1，l_2，并设v，t 这二个地址中一开始均存有数$\bar{0}$，下面为了写得清楚一些我们以q 表示$K+2n+l_1+4$。由此即得所求的计算$f(x_1, \cdots, x_n)$ 的$\mathscr{P}^3$程序如下：

表 12

地址	8	9	…	$7+n$	…	v	…	t	…	$K+1$	$K+2$	$K+3$	$K+4$
内容	$\bar{x}_1$	$\bar{x}_2$	…	$\bar{x}_n$	…	$\bar{0}$	…	$\bar{0}$	…	$P(1, 8)$	$P(2, v)$	$P(2, 2)$	$P(1, 7)$

$K+5$	$K+6$	$K+7$	$K+8$	$K+9$	…	$K+2n+2$	$K+2n+3$
$P(3, t)$	$P(1, 9)$	$P(2, 8)$	$P(1, 10)$	$P(2, 9)$	…	$P(1, 7+n)$	$P(1, 7+n+1)$

$K+2n+4$	…	$K+2n+l_1+3$	q	$q+1$	$q+2$	$q+3$
Γ_1^-			$P(1, 7+n-1)$	$P(2, 7+n)$	$P(1, 7+n-2)$	$P(1, 7+n-1)$

…	$q+2n-4$	$q+2n-3$	$q+2n-2$	$q+2n-1$	$q+2n$	$q+2n+1$	$q+2n+2$	…	$q+2n+l_2+1$
…	$P(1, 8)$	$P(2, 9)$	$P(1, 6)$	$P(2, 8)$	$P(1, 4)$	$P(2, 7+n+1)$	Γ_2^-		

$q+2n+l_2+2$	$q+2n+l_2+3$	$q+2n+l_2+4$	$q+2n+l_2+5$	$q+2n+l_2+6$	$q+2n+l_2+7$
$P(1, 7)$	$P(2, 2)$	$P(1, 8)$	$P(3, 8)$	$P(2, 2)$	$P(1, t)$

$q+2n+l_2+8$	$q+2n+l_2+9$	$q+2n+l_2+10$	$q+2n+l_2+11$	$q+2n+l_2+12$	$q+2n+l_2+13$
$P(8, q+2n)$	$P(1, 7+n+1)$	$P(2, 4)$	$P(1, v)$	$P(2, 8)$	$P(5, 0)$

辅理 3.3 设$g(y, x_1, \cdots, x_n)$ 为已知的$\mathscr{P}^3$可计算函数，则由如下的模式（Ⅵ）

$$(\text{Ⅵ})\, f(x_1, \cdots, x_n) = \mu y(g(y, x_1, \cdots, x_n) = 0)$$

所定义的函数$f(x_1, \cdots, x_n)$ 仍是$\mathscr{P}^3$可计算的。

证．设Γ 为计算$g(y, x_1, \cdots, x_n)$ 的规格化了的$\mathscr{P}^3$程序，其开始指令与结尾指令的地址分别为r，s，令Γ^- 为从Γ 中去掉结尾停机指令的结果，即得所求计算$f(x_1, \cdots, x_n)$的$\mathscr{P}^3$程序为

表 13

地址	8	…	$7+n$	$7+n+1$	…	t_1	…	t_2	…	$r-2n-7$	$r-2n-6$	$r-2n-5$
内容	$\bar{x}_1$	…	$\bar{x}_n$		…	$\bar{0}$	…	$\bar{0}$	…	$P(1, 7+n)$	$P(2, 7+n+1)$	$P(1, 7+n-1)$

$r-2n-4$	…	$r-9$	$r-8$	$r-7$	$r-6$	$r-5$	$r-4$	$r-3$	$r-2$
$P(2, 7+n)$	…	$P(1, 8)$	$P(2, 9)$	$P(1, t_1)$	$P(2, 2)$	$P(1, t_2)$	$P(3, 8)$	$P(2, t_1)$	$P(1, 7)$

续表

地址	$r-1$	r	$\cdots$	$s-1$	s	$s+1$	$\cdots$	$s+2a-2$	$s+2a-1$	$s+2a$	$s+2a+1$
内容	$P(2, t_2)$	Γ^{-}			$P(1, 9)$	$P(2, 8)$	$\cdots$	$P(1, 7+a+1)$	$P(2, 7+a)$	$P(1, 4)$	$P(2, 2)$

$s+2a+2$	$s+2a+3$	$s+2a+4$	$s+2a+5$	$s+2a+6$	$s+2a+7$	$s+2a+8$	$s+2a+9$
$P(1, 6)$	$P(8, r-2a-7)$	$P(1, t_1)$	$P(2, 4)$	$P(1, 6)$	$P(2, t_1)$	$P(2, t_2)$	$P(5, 0)$

由以上辅理即知$\mathscr{P}^2 \subseteq \mathscr{P}^3$，至于$\mathscr{P}^3 \subseteq \mathscr{P}^2$，那是显然的。故得

定理 7　$\mathscr{P}^3 \equiv \mathscr{P}^2$.

我们现在来考虑另一种重复指令。

强重复指令（其符号是$P(9, a)$）其含义即将§1 所介绍的重复指令中，条件(3)“由地址$i+3$到地址a这一段地址中所存指令均不以$i+1$，$i+2$作为地址码”这一限制取消，其余内容均与§1 中重复指令相同。

令$\mathscr{P}^6 = \{P(1, a), P(2, a), P(3, a), P(9, a), P(5, 0)\}$。

辅理 3.4　设$g(y, x_1, \cdots, x_n)$为已知的$\mathscr{P}^6$可计算函数，则由关于极小运算的递归模式（Ⅵ）施于$g(y, x_1, \cdots, x_n)$所得的函数$f(x_1, \cdots, x_n)$也是$\mathscr{P}^6$可计算的。

证．设Γ为计算$g(y, x_1, \cdots, x_n)$的规格化了的$\mathscr{P}^6$程序。设其长度为$l+1$，Γ^{-}为从Γ中去掉结尾停机指令的结果。在下面所介绍的$\mathscr{P}^6$程序中，这Γ^{-}应出现二次。因两次所存放的地址不同，故这一段程序中出现的一些相应的地址码应亦随之改变，为了避免混淆，我们将Γ^{-}的这两次出现分别记为Γ_1^{-}与Γ_2^{-}。

所求之计算$f(x_1, \cdots, x_n)$的$\mathscr{P}^6$程序如下：

表 14

地址	8	$\cdots$	$7+n$	$7+n+1$	$\cdots$	$K+1$	$K+2$	$\cdots$	$K+2n-1$	$K+2n$	$K+2n+1$
内容	$\bar{x}_1$	$\cdots$	$\bar{x}_n$		$\cdots$	$P(1, 7+n)$	$P(2, 7+n+1)$	$\cdots$	$P(1, 8)$	$P(2, 9)$	$P(1, 6)$

$K+2n+2$	$K+2n+3$	$\cdots$	$K+2n+l+2$	$K+2n+l+3$	$K+2n+l+4$	$K+2n+l+5$
$P(2, 8)$	Γ_1^{-}			$P(1, 4)$	$P(2, K+2n+l+6)$	$P(9, K+2n+2l+13)$

$K+2n+l+6$	$K+2n+l+7$	$K+2n+l+8$	$K+2n+l+9$	$K+2n+l+10$	$K+2n+l+11$
$\bar{0}$	$\bar{0}$	$P(1, 8)$	$P(2, 2)$	$P(1, 7)$	$P(3, 8)$

$K+2n+l+12$	$\cdots$	$K+2n+2l+11$	$K+2n+2l+12$	$K+2n+2l+13$	$K+2n+2l+14$	$K+2n+2l+15$
Γ_2^{-}			$P(1, 4)$	$P(2, K+2n+l+6)$	$P(1, 8)$	$P(2, 4)$

$K+2n+2l+16$	$K+2n+2l+17$	$\cdots$	$K+4n+2l+14$	$K+4n+2l+15$	$K+4n+2l+16$
$P(1, 9)$	$P(2, 8)$	$\cdots$	$P(1, 7+n+1)$	$P(2, 7+n)$	$P(5, 0)$

由本辅理及辅理 2.2—2.4 即得$\mathscr{P}^2 \subseteq \mathscr{P}^6$，至于$\mathscr{P}^6 \subseteq \mathscr{P}^2$，则是易证的。因只要利用§2 所定义的函数$g(\omega, K, t)$，再令

$$\beta^{-}(\omega, K) = \mu t((g(\omega, K, t))_0 = lh(\omega) \dot{-} K)。$$

即得

$$f((\underline{\omega})_3, \cdots, (\underline{\omega})_{2+n}) = (\underline{g(\omega, K, \beta^-(\omega, K))})_4,$$

这就证明了

定理 8 $\mathscr{F}^6 \equiv \mathscr{F}^2$。

Van der Poel[10]曾得出一只含一条指令的指令系统，不过，其程序较复杂，且因该系统中表示的数码包含正负号，这一情况与我们在本文中所考虑的情况有所不同。在我们所考虑的情况下，可以证明，一只含下面这两条指令的指令系统与$\mathscr{F}^2$是等价的（证明略）。

$P^D(1, a)$ 表示以地址 a 中数减去（算术减）地址 1 中数，其结果存于 1，a 两地址中。然后问相减结果是否为 0，如是 0 作下面隔一地址的地址中所存之指令，否则，作下一地址中所存的数所表示的地址中所存之指令。

$P^D(2, a)$ 表示将地址 a 中所存之数与地址 1 中所存之数相加，其和存于 1，a 两地址中。然后执行下一地址中所存之指令。但如 $a=0$，则停机。

参 考 文 献

[1] 王浩，数理逻辑概论，科学出版社，北京，1962。

[2] Burks, A., Programming and the theory of automata, Computer programming and formal systems. North Holland publishing Company, Amsterdam, 1963, 100—117.

[3] Wilks, W. V. and Stringel, J. B., Microprogramming and the design of the control circuits in the electronic Computer, *Proc. Cambridge Philos. Soc.*, 49 (1953), 230—238.

[4] Ritchie, R. W., Classes of Predictable Complexity functions, *Tran. Amer. Math. Soc.*, 106 (1963), 139—174.

[5] Cleave, J. P., A hierarchy of primitive recursive functions, *Zeitschr. f. math. Logik and Grundlagen d. Math.*, 9 (1963), 331—345.

[6] Shepherdson J. C. and Sturgis H. E., Computability of recursive functions, *J. Assoc. Comput. Mach.*, 10 (1963), 217—255.

[7] Kleene, S. C., Introduction to Metamathematics, Van Nostrand, Princeton, 1952.

[8] Peter, R., Rekursiven Funktionen Akademischer Verlag Budapest, 1951.

[9] Peter, R., Programmierung und partiell-rekursive funktionen, *Acta Math. Acad. Sci. Hungar.*, 14 (1963), 373—401.

[10] Van der Poel, The essential type of operations in an automatic computer, *Nachrichtentechnische Fachberichte NTF*, 4 (1957), 144—145.

结构程序设计与结构程序语言

(1977)

§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[3]，“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停顿。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、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①。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帮助表示深切的谢意。

① 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。

§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"[12,62]。此外，Denning[13]，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，姑且就"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：

（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。

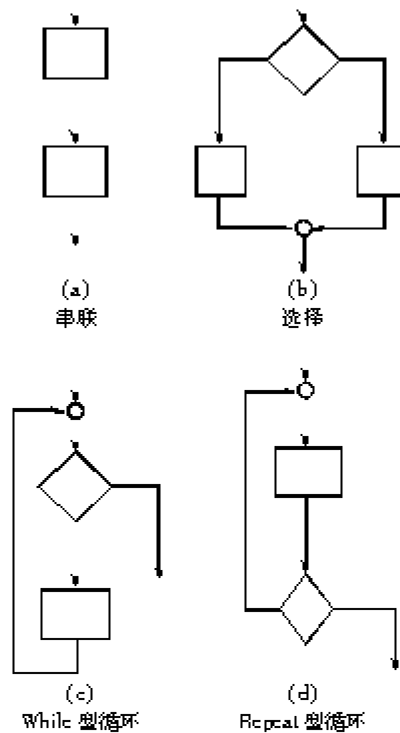


图 1

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 COTO 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 COTO 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) COTO 语句事实上有两种：一种是限于只能往前跳不能往回跳的 COTO，一种是不加这些限制的 COTO。前一种 COTO 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 COTO 为受限制的 COTO，因此，应将删去 COTO 与限制 COTO 区别开来。

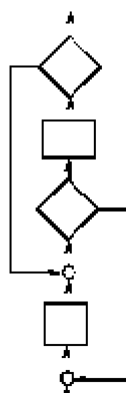


图2

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副作用局部化。文［20］指出："当一不含 GOTO 的系统被修改时，其副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副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："程序结构应该能预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[23]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 COTO 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 COTO 语句了事。

删除 COTO 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[27,28]。

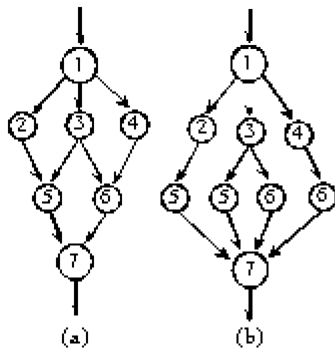


图3

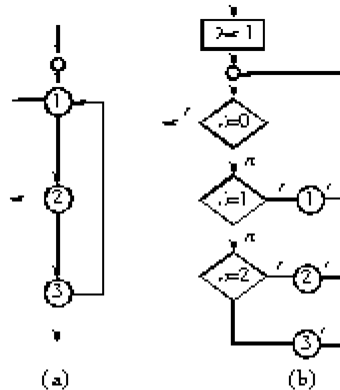


图4

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 COTO 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 COTO 语句以代替任意的 COTO 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]。

Knuth［2］中指出："关于 COTO 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 COTO 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 COTO 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 COTO 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

case E：A_1，…，A_n end

后来因为由 E 算出之整数值 i 去找 A_i 时，在后面 A_1，…，A_n 较复杂的情况下不易

看清楚其中那一个为 A_i，乃在文［29］中将此种语句改成如下的形式：

cases E of

L_1 : A_1

…

L_n : A_n

end

此处 L_1，…，L_n 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 of set 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 do C

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; resultis 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。

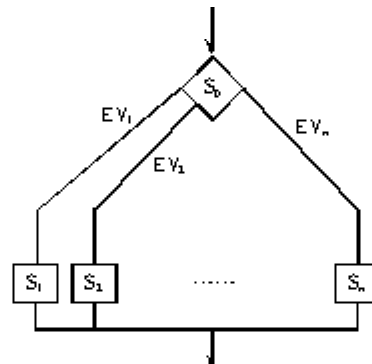


图 5

其含义是：重复执行 S_0，直到 Ev_1，…，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”（Loop with 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
esacod
```

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 τ_1，$\cdots$，τ_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 Zahn 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，则删去其前面之冒号，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 $1 \leq i \leq n$；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，docase 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[8,9,31]。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 ALGOL 60 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
    for j = 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 ALGOL 60 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]
end
```

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 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个素数，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:=j+2;
              if sqr(p[lim])≤j then lim:=lim+1;
                i:=2; prim:=true;
                while prim∧(i<lim)do
                begin
                  prim:=(j mod p[i]≠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 ALGOL 60 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"在框图中没有一合适的记号来表示"，(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[43,46]，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)）。

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)）。

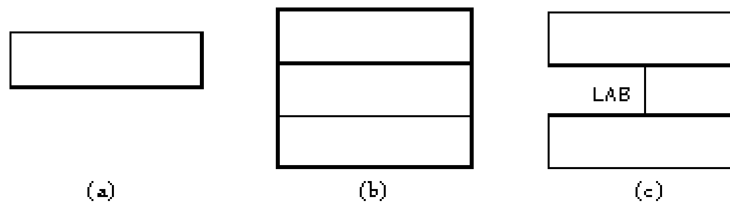


图6

椭圆型框用于表示函数（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)）；如此椭圆型框出现在一长方框内即表示此椭圆型所示之函数（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连的椭圆框（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)）。

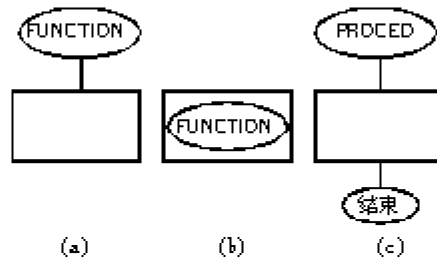


图7

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））。

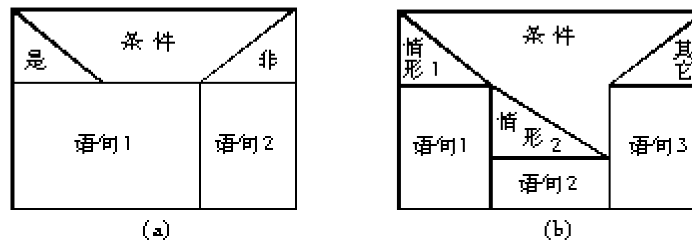


图8

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））。

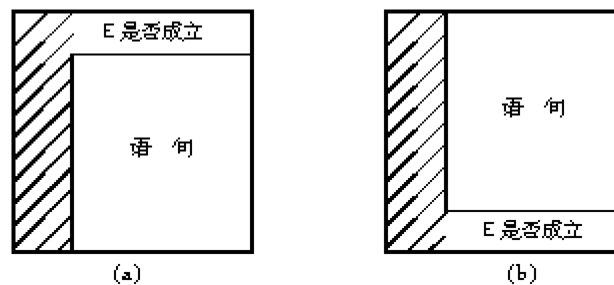


图9

结构程序设计中是否完全删去 COTO 尚有争议，故框中亦可写出 COTO 语句，表示为“→LAB”，它可在长方框的算法中出现。

如算法中引用在所给框图中未曾标出的函数或过程，则在此被引用之函数或过程前后记以“ * ”而不另用椭圆框框出。此外，在椭圆型框之旁亦可用 * 引起一个

说明表示该框所指之函数在何处被调用以及从何处调用等信息。

长方框内出现由圆圈标出的信息i，表示第i个注记。

Chapin 这种框图看来还是符合结构程序设计的要求，亦比较易于掌握，值得参考。

例

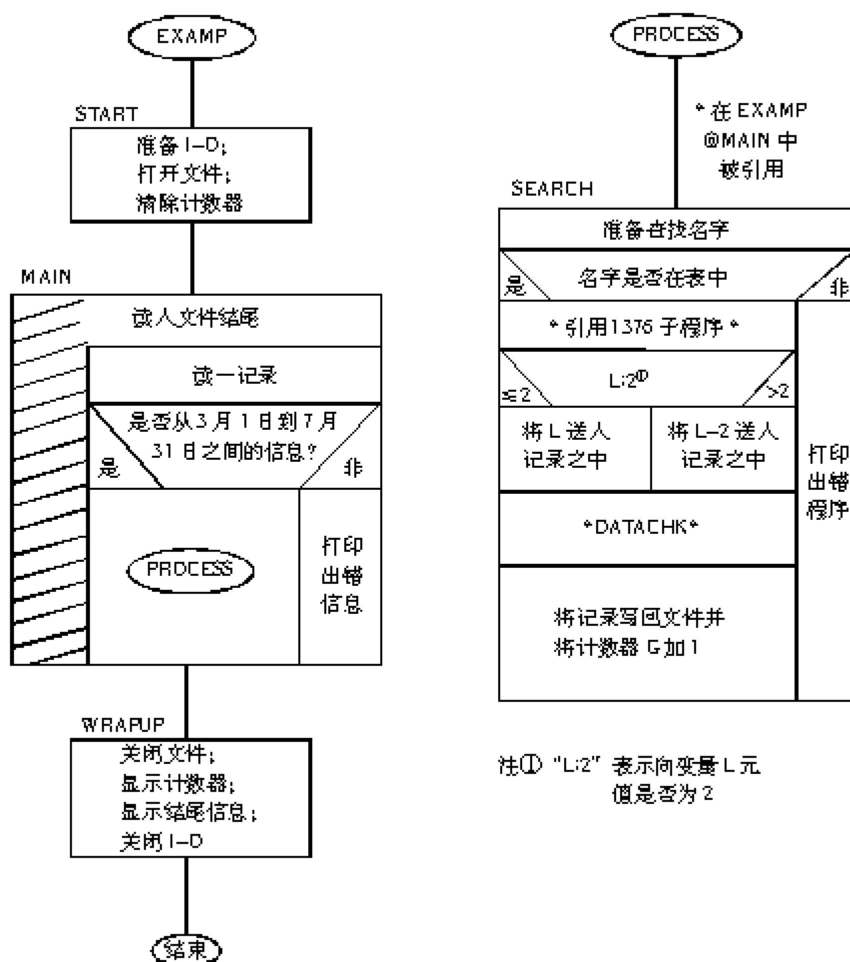


图10

由上面的介绍可以看出逐步求精方法的确提出了一种能使程序设计摆脱传统设计方法的途径。由于设计过程中先全局后局部，先整体后细节，先抽象后具体，比较容易使程序有一完整清晰的总体结构，这样写的程序较易为人读懂亦较难出错。这是此方法的优点。然而，这方法亦有令人感到不足之处：（1）由于强调先全局后局部，故要求设计者应对问题全局结构有一较为深透的理解，这一点往往给初学者带来困难。（2）逐步求精过程不能循环进行，故对那种总体结构中出现循环调用的问题这方法即难以应用。比如在设计翻译程序时，特别是采用由底向上的语法分解方法的情况下，即很难严格按此法进行。事实上，即令是采用由顶向下的方法也将遇到循环调用的问题。Ammann［47］中已遇到了这个问题。（3）逐步求精方法的

分步虽对某些问题的设计可能带来方便，但对另外一些问题，对总体结构进行综合考虑比一开始即分步进行还更方便，甚至有时是先综合考虑了以后再形式上分成步骤进行的。这已经不能算是严格意义下的逐步求精方法。通常的系统程序设计往往如此。

Baker［7］中所述IBM FSD采用的“由顶向下”设计方法即较为特殊。他们所谓“由顶向下”方法“其真实含义是：在日常生产性程序设计中……写出的代码仅仅只依赖于当时已经运转的代码。”即不允许写出一程序时，验证这程序所需要的程序和数据还没有准备好。显然，满足这样条件的设计程序的过程不一定是严格意义下的逐步求精。

Baker［62］中将由顶向下的方法应用于发展系统程序的过程，描写如图11。

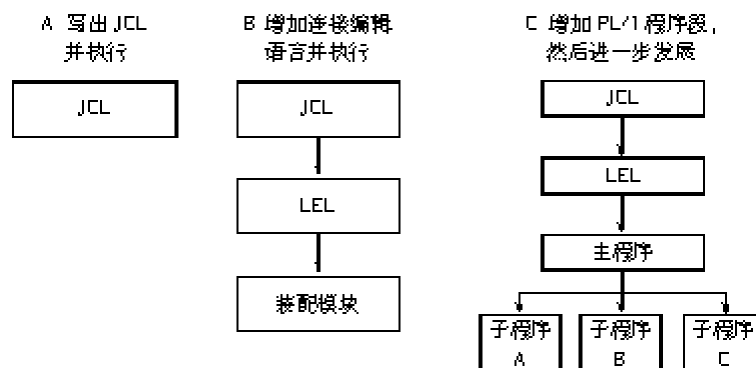


图11

图中JCL表示作业控制语言，LEL表示连接编辑语言，故其发展过程是：先定出作业控制语言，在某种意义下（即从系统内部结构看），这事实上是我们上面所谈到的一种关于操作系统的“体系设计语言”；这一层定了以后，再发展连接编辑程序，然后通过一装配程序相连即可上机，再进一步发展，即可与机器上的高级语言程序段相连，由此又可进一步发展。

Bauer［81］中讨论由顶向下方法时则更强调每一层求精之后，即对所得程序进行正确性的验证，这样，可使整个程序的正确性更能得到保证。由此可见，即在结构程序设计这一范围内，对由顶向下的方法解释也不完全是一样的[104,103]。不过，逐步求精方法的确从科学方法论的角度比较系统地提出了一种程序设计方法，比传统的方法确实是一很大的进步。但还不能认为这已经是程序设计的唯一完美无缺的科学方法。

在此值得一提的是近年来由于人工智能研究取得了较大的进展，应用这方面的成果于程序自动优化（见Knuth［2］）及自动综合（见Manna and Waldinger［14］）的研究颇被注意。后面这个问题显然更接近于开关线路综合的Huffman问题。不过，离解决尚甚遥远。甚至问题提清楚亦不容易。主要的难点有所欲综合的问题的功能用什么语言刻画？目前有人用自然语言，也有人用一阶谓词演算，也有人用某种集合论语言。本文作者感到，这种描述功能语言陈述清楚以后，恐怕所谓程序自动综

合问题不过就是前面所述自动程序设计[42]的一种不同处理的方案而已。这也可以说是另一种程序方法的研究。

(C) 分层结构与模块结构

这是关于程序的宏观结构问题，一方面是一程序的各个部分应分成怎样的模块；另一方面是各模块之间的引用关系应构成一个怎样的总体结构。而这样一种结构又恰好与如何进行设计的方法紧密相关。

在上节中介绍的逐步求精方法强调先全局后局部。故执行此方法的结果，必将产生一种条理清晰的总体结构。这样一种结构应具有怎样的特征？

(i) 这一结构之中各分块之间不能构成循环。因为逐步求精的过程是单向的，不能循环进行。循环只能在一小块之内出现。

(ii) 逐步精化必然是一层一层地实现的，故由此实现的算法总体结构必然是一种分层的结构。不过严格意义下的逐步求精，应是将上一层的某些结点，每一个分解成若干个结点（子块）。因此，严格的逐步求精所得到的程序应是一树形结构的（如图 12 (a)）。但实际上其中许多子块可能是相同的，应可合并，故应是一上半格结构的（如图 12 (b)）。

在系统程序中，除了用户程序与子程序这样两级模块外，还有操作系统这样一级更外层的模块。通常设计系统程序时，往往是先对总体结构作综合的考虑，根据与机器关系密切的程度和引用频繁程度的不同，将各子模块分成若干层，其中较外层的每一子模块原则上对较内层的任一子模块均有权引用。这样一种层次结构，如图 12 (c) 所示，有人称之为有向网结构[30]。这样一种结构自然与树形结构或上半格结构都不相同，但仍是一种层次分明的分层结构。应该说也是符合结构化的要求的。如果将逐步求精的方法理解得不那么绝对，即它不但容许由底向上的补充和修正，而且也容许在设计前对总体结构作综合的分析和安排，则这样一种结构也应该可以由逐步求精的方法所实现。

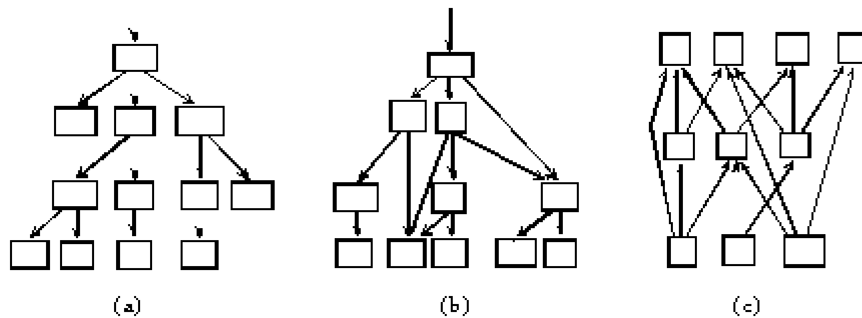


图 12

至于程序模块化的问题在结构程序设计的概念提出之前早已为人注意（有名的如 D. L. Parnas 的研究工作）。不过，结构程序设计为这一问题提供了一种较为系统

化的切实的回答。有人[62]甚至要求在一结构化的程序中，每一模块的长度不超过一页纸的范围。

所谓“模块”，即是指标准化的单元，用它可以组成具有各种功能的大系统。对于软件工程来说，Dennis 文［57］的定义是：“模块化即指将所谓程序模块的部分放在一起建成软件系统”。他认为程序模块化应具有以下两方面的目标：

（i）人们应能确信每一程序模块具有独立于用它建立的较大软件单元的上下文的正确性。

（ii）人们应能在不了解别人所写的程序模块的内部情况时，可以很方便地用这些模块组成程序。

这也就是 Boebert 的“独立于上下文”以及 Dijkstra 的“互不干扰”的概念。

事实上，GOTO 语句的删除，将程序的控制结构限于图 1 那样的图式，即有利于排除小程序块之间的互相干扰，有利于标明小块程序的交接口所表示的信息传输关系。而图 12（b）所示的上半格结构，要求循环的依赖关系只限于在一块之内出现，这也使每一小块较为独立，与上下文的关系较为简单（比如图 12（b）中各模块之间的关系都是单向的），这些要求，都有利于组成程序模块。

与程序模块化问题关系最为密切的一个问题是程序语言中应提供什么样的描写模块的手段。这种手段的内部结构及外部联系的规定不同，所描述的模块的结构也自然不同。几乎每种高级语言都提供了一些描写模块的手段，从结构程序设计的要求来看，对这些手段亦有不同的评价。

如 Fortran 中的块结构即是为了便于书写模块而设计的。它的确比较好地反映了通常设计规模较小的程序工作的要求，也比较灵活好用。但是，从结构程序设计的角度来看，这样一种模块形式有许多不妥之处：（1）模块之间的联系不是一种上半格或树形的分层结构的形式。（2）主程序块不能作为模块。（3）不同模块之间通过 COMMON 中的名字进行联系，不同人写的程序如未很好了解即很易相冲。近年来有人[33]从结构程序设计的角度对全程量提出异议，认为这是与 GOTO 语句相似的一种对程序正确性危害很大的语言成分。主张予以取消。近来亦有人在讨论 Fortran 是否能结构化的问题[39,170-179]。

ALGOL 60 中“过程”与“分程序”也是一种书写模块的重要手段，由于 ALGOL 60 是分程序嵌套结构。故其总体结构是符合（c）节中所述的上半格分层结构的要求的。但也同样存在外部公用量名词相冲的问题。特别是一过程在说明时的静态环境与执行时动态环境不同。在说明时，在一过程体内自由出现的上一层所定义的量，到执行时，其环境中很可能出现在该环境中重新说明过的同名量，这时，即很易造成混乱。在 Jossle［39］中，凡是在一过程体中出现的形参以外的非局部量均应在过程前面予以列出；也不妨设想一种限制，即在体内不允许除形参以外的任何非局部量，外部量只能在实参中出现。这样也可解决过程体内外信息的交换问题（人们往往以此法写可重入的程序）。

但是，ALGOL 60 式的过程概念以及数据结构的处理方式被认为有两方面的不足之处：（1）数据结构与施之于具有这种结构的量之上的运算（即过程或函数），是分开说明的，这样，即对于理应以这些运算来表示的对该类型含义的限制难以表

示。比如，一个后进先出栈从 ALGOL 60 的说明中孤立来看不过是一个一维场。此外，在其上分别定义了 push 与 pop 两个过程作为栈上的运算。由于栈的说明及这些过程的说明是分开作的，故对具有这样含义的栈的任意一个中间分量进行存取完全是允许的。而这恰好违反了通常我们对栈的含义所作的限制。(2) 根据 ALGOL 60 的规定，一被调用的过程应在调用它的过程结束之前结束，这样的要求是由于在编译时可以利用后进先出栈处理其局部量及中间结果而提高效率。但在系统程序中，即往往有不便之处。有时一过程 A 调用一打印程序 B。完全不必要让 A 等待 B。这一点在 ALGOL 60 中即不能做到。此外，对并发进程之间的通信问题，同样也不宜于用 ALGOL 60 的过程概念来表示。为了解决以上这两方面的不足，SIMULA 67 第一次提出了 CLASS (类组) 概念作为一个新的描述模块的手段[60,61,33]。

Class 是一种能使其每次具体调用在调用它的过程结束后仍能继续存在的过程概念；其每次具体调用称为它的一个 object，它也象过程一样由说明引入并包含参数 (在 SIMULA 67 中规定，基本类型的参数都是赋值的，构造类型的参数都是换名的)。其定义方式为

〈class 说明〉∷ = Class〈class 标识符〉〈形参部分〉;〈种类部分〉;〈class 体〉

〈class 体〉∷ =〈语句〉

在一 Class 内说明的局部量或过程称为它的 attributes。

Class 的每次调用产生它的一个 object。各参数的初值，在其调用的〈实参部分〉予以列出。Class 的调用称为 generator，其形式象一函数。执行的结果产生一个指向这新生成的 object 的指示器 (即地址)。generator 的形式为

〈object 的 generator〉∷ = new〈class 标识符〉〈实参部分〉

为了能引用所产生的 object，故应将所产生的地址存入一变量之中，这样的变量的类型为 reference。其说明的形式为

〈reference 变量说明〉∷ = ref (〈class 标识符〉)〈标识符表〉

自然这种变量亦可用于指向其它类型的量，此时“ref ( )”内即不是〈class 标识符〉而是具有其它类型的量的名字。如所指对象类型不定，则可以 none 表示其值。

对于这种 reference 变量自可如通常的一样对之赋值，并对其值进行比较 (SIMULA 中以“:—”表示赋值号。以“= =”、“=/=”，表示相等、不相等)。

例 Class C 的说明为

Class C(…); …

指向 C 的变量 X 的说明为

ref(C)X;

在进行了这些说明后，即可将 C 的一次 generator 的开始地址存入 X 中。即：

if X = = none then X: — new C(…);

如 X 是一指向 Class C 的变量，而在 Class C 中有 attribute A，则 X · A 即表示当前 X 中所指的那一 Class C 的 object 中的 attribute A。这样的 X · A 即可像通常的变量一样参加运算。

例如用 Class 表示“二进查找树”即如下：

Class tree(v); int v;

```
begin ref(tree) lft, rgt;
  proc ins(x); int x;
  begin if x < v then
       begin if lft = = none
          then lft: — new tree(x)
          else lft · ins(x)
       end
       else if rgt = = none
          then rgt:— new tree(x)
          else rgt · ins(x)
  end
  ref(tree) proc find(x); int x;
    begin if x = v then this tree
        else if x < v then
           if lft = = none then none
           else lft · find(x)
          else if rgt = = none then none
            else rgt · find(x)
    end
end
```

— Class 的后面可再定义 Subclass（子类组），也可以由 Class 拼成 Class。

设 Class A 的定义为

```
Class A (PA); SA;
  begin DA
        IA
  end
```

此处 PA，SA，DA，IA 分别表示 Class A 的形参部分、种类部分，体中数据及过程定义部分、初始算法部分（如清除，送初值等），则与 A 拼成的 Subclass B 即定义为

```
A Class B (PB); SB;
    begin DB;
          IB
    end
```

它等价于：

```
Class B (PA, PB); SA, SB;
  begin DA; DB;
        IA
        IB
  end
```

其含义即 Class B 本身是一个 Class A。其数据，过程等部分的定义是先置 A 的，然后置 B 的。

例如，设已有前述“二进查找树”的定义，然后定义一个 Class 表示名字特性表。其中名字部分依二进查找树排序。此外，再增加 type，address 等属性，在此表上规定：“送表”（enter）与“查表”（search）两个外部过程，则此名字特性表，即可作为 Class tree 的 Subclass 定义如下：

```
tree Class nametable(type,address);
         int type, address;
         proc enter ……
            ……
         proc search……
            ……
```

这样的子类组概念，显然亦表示出一种模块的层次关系。

在 SIMULA 67 中除了上述 Class 中外部过程的引用外，还设置了 resume，detach 等专用语句，利用这些语句，即可实现如下形式的相互作用关系：执行过程 M 至语句 call（X）即转入过程 X，执行至语句 resume（Y），即转入过程 Y……当执行到 detach 语句时，即回到原来 M 的 call（X）处，按当时转出时的状态往下执行。利用这样成分即可构造出像 coroutine 那样结构的程序。

Class 概念是近年来影响很大的新语言成分。关于它的讨论很多，本文后面将要介绍的一些语言概念都与此有关。显然，这样一种概念值得我国语言工作者重视和研究。

事实上，程序的结构不但与设计程序的方法密切相关（这一点上面已谈过了），而且与人员组织形式也有关系。Conway 有句名言：“程序人员组织结构怎样，则它所产生的程序结构也就怎样”。故为了保证程序有一合理结构，还应考虑其人员的合理组织形式问题。

（D）组织形式

关于结构程序设计应采取什么样的人员组织形式有许多种设想。而最受注意的是 Mill（Baker [62]）提出的主程序员组（Chief Programmer Team）。其主要内容是，在这样的程序员组中包含以下四类人员：

主程序员一人，他负责全面的设计，写其顶部的程序，规划各部分程序，并检查组中各程序员的程序。

辅助程序员一人，他协助主程序员工作，在主程序员不在时，接替他的工作，同时负责管理主要子系统的工作。

初等程序员五人，他们完成主程序员所定义的各部分程序。在一大的程序课题中，他们在其所负责的那部分程序中又起主程序员的作用。

程序资料保管员一人，按照程序员的要求修改源程序，以及将程序交付运行，负责记录有关整个课题进行中各种情况以及程序运行时的各种情况。

容易看出，这样一种人员组织形式与（C）中所讨论的树形分层的总体结构也

正好是对应的。二者可以说是一种结构的两种表现形式。

有人认为结构程序设计在保证程序正确性方面取得的成绩主要与这样一种组织形式有关。是否果真如此，尚待了解。有一点却是肯定的，即它可以充分发挥有经验的程序员的作用，减少不熟练人员对程序正确性的消极影响。现在美国某些重要的计算机厂家正在开始推行这一组织形式。在推行过程中提出了许多更为具体的问题[63]，其中不少问题则是与美国的垄断资本主义制度有关的；另一方面，由于要将这一方法进一步在工业生产中落实，组织工作的规定日益细密[7,64]，关于这些讨论即不必在本文内介绍了。

但我们应对这样一种组织形式作一些分析。本文作者认为，作为实现一个复杂软件系统的庞大的组织，没有一种较为严格的组织形式是不行的；为了保证整个程序有一合理结构，有一小部分人掌握着全局，首先集中提出一总体结构并随时负责检查这一结构的贯彻也是必要的；同时，对这一部分掌握全局的人要求他们对整个体系有一较深透的理解也不是一种过分的要求。主程序员组体现了这些方面的精神。

但是，从资本主义社会中产生的这样一种组织形式，不可避免地要深深地打上它所在环境的社会烙印。这些烙印我们必须认识到，因而不能照搬。比如：要真正发挥集体主义精神没有无产阶级政治挂帅是难以实现的，这点我们社会主义制度的优越就可以发挥巨大的威力；再就具体的组织形式看，主程序员组形式中片面强调主程序员由顶向下的主导作用，而忽视了下面工作人员的主动性。正如前面讲到由顶向下的方法时已曾指出：由顶向下的方法必须与由底向上的补充修正相结合一样，在人员组织形式中也必须是集中与民主相结合，领导与群众相结合。本文作者深信，在我们国家，以毛主席思想为指导，参考外国的先进经验，但亦批判其不合理的成分，必可创造出更好的程序工作组织形式。

此外，亦还有人[63]提出“无我程序设计组”（egoless programming group）的形式。即程序设计工作应该从程序员个人行为这样一种特点中解脱出来，而注意其集体性。为了保证程序正确，一个程序设计组中的人员应该互相传阅，互相检查彼此的程序，使得整个程序成为集体共同理解，共同实现的工作。这些意见，也值得参考。

§3　结构程序语言

结构程序语言是一种能反映结构程序设计要求的程序语言。L. Presser 在“结构语言”[66]一文中列举了该语言所应达到的目标。他指出：“一结构语言，除了适应其所要应用的领域的要求之外，还应设计成满足某些特殊的目标。常见的主要目标即要求使代码按照避免出错的方式来编写，进而要求清晰的缮写形式使其静态文本（排列）与其代码的动态行为相近似。语言设计的另一个可能目标是能将代码描写成局部性明显且在虚拟的环境中便于执行的方式。再一个目标即要使验证更为清楚。”“更进一步，希望通过结构程序语言来加强设计规程的重要课题影响着程序的最终格式。所以，由结构程序语言产生的程序的总体结构是由语言设计所确定的那一组目标所定形的”。

为了将上述这些结构程序语言的特征阐述得较为详细，我们还应回头再看看 §2 中所述结构程序设计的四个方面的特征。

（A）删除或限制 COTO 语句的问题，这个问题实质上是讨论如何使程序的静态结构和其执行时的动态结构相一致的问题。这个问题反映在程序语言上即结构程序语言应具有什么样的控制结构的问题。

（B）逐步求精的设计方法。这个问题实质上是讨论：为了得到一可靠的程序，其设计过程应以什么样的方式进行。这个问题反映在程序语言上即要求结构程序语言应具有什么样的数据结构从而能构造出与设计步骤相适应的各层语言。

（C）分层结构与模块结构问题，所谓分层结构问题也就是关于系统程序的总体结构的问题。这个问题反映在语言方面即当一结构程序语言要作为系统程序语言用时，它表示什么样的系统结构。

所谓模块结构问题，这是讨论一上机程序及其中所包含的子程序应具有什么样的结构。这个问题反映在语言方面，即一结构程序语言所表示的程序及其中所包含的子程序应具有什么样的模块结构。

（D）人员组织问题，即讨论在一程序设计课题中，其参加人员应按什么样的形式进行组织。而结构程序语言所表示的系统结构及程序模块结构，也同样应该是其相应的人员组织结构的一种反映。正如 J. R. White 与 L. Presser［67］中所明确指出的：“JOSSLE 所赋予的系统结构，本质上即反应了主程序员组的管理策略”。

上述四个方面的结构问题，前两方面（即控制结构与数据结构）是关于语言的微观结构的。而其余二方面（即系统的总体结构，程序及子程序的模块结构及人员组织结构）是关于语言的宏观结构的。以下分别进行讨论。

在讨论之前，必须注意以下基本指导思想：

（i）结构程序设计是为了要得到实际可靠的程序。故在讨论一语言成分或结构的取舍时，往往不只是去讨论该成分或结构在逻辑上可能具有什么特殊的功能或表达能力，而是着重从实际上去衡量该成分或结构在人们心理上产生什么影响。所以，讨论的结果，往往是从逻辑功能方面并无显著区别的成分或结构中强调选取那些使人阅读时感到较清晰，用起来较不易出错的方案。

（ii）一种表示形式的复杂性对人们心理上的作用，往往由于量的增加而产生质的变化。这一点 Dijkstra 在［21］等文中事实上已经反复说明过了。所以，在评价一种表示形式的优缺点时，不能单纯以简单的情形为例。因简单情形使人感到清晰的表示形式，在复杂到一定程度后很可能变得极其使人费解，ALCOL 60 中有些语言成分，往往存在这方面的问题。

下面的讨论只有在上述这两前提下才有意义，所列举的各种理由大都是根据这样的思路提出来的。

（一）程序语言的宏观结构问题

进行程序设计，常需将一较大的程序先分划成互相干扰较少、相对独立的模块。分划时，必然会遇到以下问题：（i）各模块以什么方式互相发生作用？（ii）各模块之间通过什么手段交换信息？最通常的情况即如何设置模块间交换信息的公用量？

(iii)一模块又如何再划分出它的子模块?(iv)这种模块的划分如何使参加同一课题的人员便于分工和协调?

在操作系统出现之前，程序的宏观结构主要表现为用户程序的总体结构及各子程序的模块结构这两种结构，而在操作系统出现之后，在用户程序之上还存在一操作系统，它的总体结构也就是所谓系统结构，从而也就存在系统、程序、子程序三层结构。而且，在一操作系统内出现各系统程序的模块，有时与用户程序的结构也不尽相同，我们称这种模块为系统模块。

正如在§2(C)中已经初步说明过的那样，常见的程序语言中几种较有代表性的宏观结构有如下几种：

FORTRAN 的块结构只是为了表示程序与子程序结构提出的，其程序结构如图 13 所示。其特征是：(1)程序结构与子程序结构不同，前者包含一可调用各子块而不能被子块所调用的主块，(2)公用量集中在主块中说明，(3)子块之间可相互调用，(4)子块之中只能定义本块的局部量，(5)子块之内不能再进一步分划出它所包含的子块。因此，这一结构只能由两级构成。

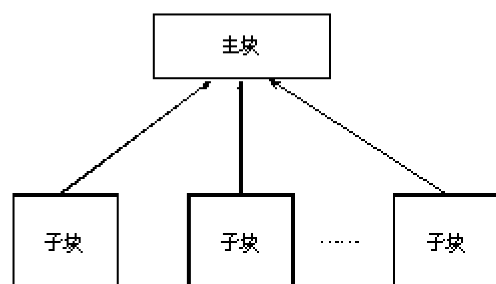


图 13

这样一种结构反映了通常中小型程序的结构。对于规模不大，参加编制的人员不多的程序，这一结构是很自然合理的。但当程序规模较大时，则变得很不方便，因为：(1)公用量集中在一处，对于子块很多，人员很多的情形，容易造成混乱，(2)各子块之间的调用关系复杂，对于大程序来说使其结构难以辨认。显然，这样一种结构不能满足系统程序分层结构的要求。

ALCOL 60 的结构如图 12(b)所示。子程序结构分为分程序与过程两大类。前者可在任何语句位置上出现，它表示了通常的开子程序概念，后者只在说明处予以定义，它表示了通常的闭子程序概念。程序本身也是一分程序，故程序的结构与子程序的结构统一起来，在概念上一致。各子程序内又可出现内一层的子程序，如此分层嵌套，在层数上没有限制，从而形成一上半格结构。公用量与局部量只有相对意义上的区别，每一模块内定义的量对其上层说是局部量，而对其内层的块而言则是公用量。各块之间的调用关系亦随上半格结构而决定。各块除能调用所属的子程序外，向上只能调用亲兄弟，亲叔伯，亲叔伯祖父…的模块，而不能调用堂兄弟，堂叔伯，堂叔伯祖父…的模块。

这一结构的优点是：(1)概念上的清晰统一，(2)公用量分层定义，既便于处

理同名问题又适于用先进后出栈来处理量的重叠分配，(3) 各级模块之间的层次分明，较便于将一较大的问题分成不同层次的算法进行处理，亦便于适当多的人员之间进行分工。

这一结构形式对于规模更大、参加人员更多而又包含较为复杂的模块之间的关系的大程序，如系统程序，即显得颇不适合。

首先是 Wulf 与 Shaw[28] 等对于公用量处理方式提出的批评。主要的问题如：(1) 副作用问题，如一过程体内出现对外层量赋值的语句，即可能产生副作用，例如：

```
begin
    procedure enclose
        begin
            real p.w;
                ......
            real procedure S(z);
                        real z;
                begin
                    w: =z+1;
                    z: =z+2;
                end S
        end enclose
end
```

对于这一程序，显然，$p: = S(k)\times w$; 与 $p: = w\times S(k)$ 即将得到不同的结果。(2) 在嵌套层次很多的情况下，在内层过程中出现的外层量，特别是在内层又定义了同名量的情况下，常常易将其说明的位置看错。(3) 在实际的程序中，并不是所有在外层定义的量在内层的每一过程中被引用。指明其应用的范围不但有助于查错，而且有时在概念上是必须的。比如先进后出栈，从结构上不过是一个一维场，其主要特点即它只限于被 POP，PUSH 等运算所加工，如不作这限制，则失去了栈的意义。而 ALGOL 则无此规定，因而不能完全表示出栈的特征。(4) 象下例这样的递归过程，对同名量 i，即易看混。

```
begin
    procedure enclose;
        begin
            integer i;
            procedure A
                begin
                    i: =i+1;
                    begin
                     real i;
                      A
```

```
                        end
                    end A
                end enclose
            end
```

显然，以上这些问题都不是由于语言的表达能力在逻辑上有何不足，而是由于书写形式当复杂到一定程度时令人感到不够清晰，从而出错。

ALGOL 60 的结构之所以出现这些毛病，通常的看法是：由于对量或过程的引用权限规定得过于一般，来龙去脉的标示信息太少，从而阅读时容易看混，亦不便于由系统进行检查。

第二是调用的层次关系。虽然 ALGOL 60 的嵌套结构的确也表示了一种层次结构，但这是一种上半格的层次结构。它与通常操作系统所要求的结构不同。比如，在同层的过程中，不能对堂兄弟位置上的过程进行调用；亦不能对上层中堂叔伯位置上的过程进行调用等等。B. Hansen 在文［80］中亦指出：“在一操作系统中的引用权限通常不是一树形结构，而是形成一有向网，这部分地说明了为什么传统的分程序结构语言的辖域规则对于并发程序（而且同样对顺序程序）感到不方便”。

第三，操作系统中往往要求处理并发的一组进程，其中一个对另一个的关系不是一种闭子程序的调用关系，而是一种相互作用的关系。即一进程在执行中可建立，启动乃至撤销另一进程。而被建立、启动的进程有其独立存在的地位，它甚至在建立者或启动者结束后仍能独立地存在。

此外，系统程序往往要求其模块中所用的量能相对独立于其算法部分的执行而存在。比如，有时要求一模块执行完毕后，其中某些量中的值仍可保留；又如，为了模块处理成可重入的程序段，除了其算法部分不许在执行时修改外，更重要的一点是引用这种模块时应各自带着有关的量。这些都是系统模块所应具有的特性。诚然，ALGOL 60 也可以实现这些要求：比如，own 变量即可在过程执行完毕后仍保留其值；可重入的程序段亦可通过将其局部量用参数来表示的方式实现。不过，系统模块应更突出这些特征。

对于上述各种问题，不同的系统程序语言往往因所强调的侧重面不同而提出特征很不相同的解决办法。归纳起来，大致可分为两类：一类是着重强调宏观结构的特征和公用量标示的明晰性，其代表为 JOSSLE[67,68,69]；另一类则着重讨论系统模块结构的特征以及引用权限的检查等，其代表即 Concurrent PASCAL[30,31]。

下面，我们分别以 JOSSLE 及 Concurrent PASCAL 为例说明之。

JOSSLE 首先在语言中将总体结构分为三级：系统结构，程序结构，子程序（即过程）结构。

系统结构由该语言中所谓“联系区”所表示，联系区的结构如图 14 所示。

此处，联系区部分指明它所管辖的成员程序的名字以及它们之间的公用量的说明，其书写格式如下：

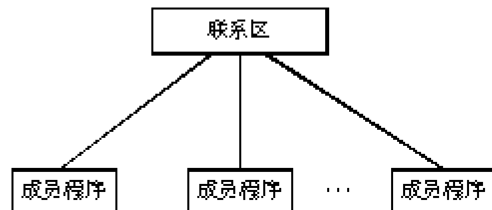


图 14

```
Communication Region〈标识符〉
  Define
    〈类型定义〉
  End Define;
  Declare
    〈变量说明〉
  End Declare;
  Members
    〈标识符〉，…，〈标识符〉
  End Member;
End Communication Region〈标识符〉;
```

其中 Define…End Define 部分定义公用量所用的构造类型的名字；Declare…End Declare 部分定义本联系区的公用量；Members…End Member 部分即列出本联系区所管辖的成员程序的名字。

联系区所管辖的成员程序可以是一程序，又可以是一联系区。如是联系区即又具有图 14 所示的结构。由此构成一树形的层次结构。

控制不能从一层的成员程序直接转入另一层的成员程序，而必须先转入下一层的联系区，再由这联系区将控制转入所管成员程序中最左的一个，这个成员称为其“主程序”，它起着较特殊的作用，只有通过它才能调用本联系区中其它的成员程序。

成员程序中不是联系区的则为“程序”，它的结构与联系区不同，而是具有“程序结构”，它的书写形式如下：

```
Program〈标识符〉
    〈公用量定义〉
    〈类型定义〉
    〈变量说明〉
    〈过程定义〉
    〈可执行语句〉
End Program〈标识符〉
```

此处公用量定义部分具有如下的形式。

```
Global
  Define
      〈类型定义〉
  End Define
  Declare
      〈变量说明〉
  End Declare
  Programs
      〈标识符〉，…，〈标识符〉
  End Programs
End Global
```

这一部分事实上是将本程序中用到的联系区中所定义的公用量重新在此予以说明，此外，其中 Programs…End Programs 内的标识符是本程序所能调用的同层的其他程序的名字。

至于其 Program 内的〈类型定义〉及〈变量说明〉部分，是用来规定本程序内所辖的各过程之间的公用量及其构造类型。其〈过程定义〉部分即本程序内所辖过程的说明。其〈可执行语句〉部分即包含本程序体内可执行的语句。

程序之内不能管辖程序，但可以有它所管辖的过程。“过程结构”又与程序结构稍异。其书写格式为：

```
Procedure〈标识符〉(〈参数表〉)
    〈已知部分〉
    〈类型定义〉
    〈变量说明〉
    〈过程定义〉
    〈可执行语句〉
End Procedure〈标识符〉
```

其中与程序结构不同之处在于其〈已知部分〉不是将所引用的外层量或程序再定义一次而只是将所引用的量或子程序的名字列在括号 Known…End Known 之中而已。此外，过程内还可再嵌套它所管辖的过程，从而过程嵌套形成一树形结构。

上述 JOSSLE 的主要特点是：(1) 将系统结构，程序结构及过程结构明确地区分开来，这不但有利于区别三类模块的特征，而且更适于系统程序工作的人员组织，正如文［67］中指出的，这种组织正好反映了主程序员组的管理形式。(2) 将各层引用的外部量及子程序在模块的开头处标明，这有利于使各级程序中引用的量或子程序来龙去脉表示清晰，既便于阅读亦便检查，因此，基本上能避免 Wulf 与 Shaw［58］所指出的问题。

但这样的结构也有一些问题：(1) 由于标明的信息太多，对大的程序固有其必要性，但对中小程序则可能令人感到累赘、多余。(2) 对于系统模块未能提出具有所需特征的方案。(3) 联系区及过程仍为树形结构，这与系统程序所要求的结构并不完全一致。而这几点（特别是 (2)，(3)），则正是 Concurrent PASCAL 所特别强

调的。

Concurrent PASCAL 中提出的“进程”（Process），“管程”（Monitor）概念是从 SIMULA 67 的“类组”（Class）变化而来[60,61]。经 Dijkstra[73]，Hoare[74]，最后由 Hansen[79,80]总结而成这两概念。

Concurrent PASCAL 是在 PASCAL 基础上扩充而成。PASCAL 的宏观结构基本上与 ALGOL 60 相同，只是取消了分程序概念，程序与子程序的结构都统一为过程的嵌套结构。Concurrent PASCAL 则在此基础上引进进程，管程等系统模块，由此而构成一种分层的系统结构。

进程与管程都看作是一种与所限定的加工运算一同定义的说明数据类型的手段，其地位与 PASCAL 中的类型相当，故称之为“系统类型”。具有这种类型的对象（量），则称之为“系统成分”。

一进程包含以下几部分：（1）局部量定义，（2）一列语句，（3）它引用其它系统成分及常量的权限，则以参数的形式表示之。

例如，名为 inputprocess 的系统进程即写成如下的形式

```
type inputprocess =
  process (var buffer: diskbuffer)
  var block: array (1…152) of char;
  cycle
    readcard (block);
    buffer. send (block);
  end
```

此处，block 即此进程的局部量的名字，这局部量的类型即由 152 个类型为 char 的分量组成的一维场（以下称为 page）。允许在此数据结构上进行加工的动作，即由 cycle…end 构成的循环语句。这进程所能引用的限于类型为管程 diskbuffer（下面将介绍）的系统成分。这形参的名字为 buffer。

具有这样的系统类型的系统成分以如下的形式引入：

var reader:inputprocess.

对于具有进程这种系统类型的系统成分，本来应可以有建立，启动，撤消等操作，不过为了简化概念，Concurrent PASCAL 中取消了“撤消”，也就是说，一进程一经发动即永远存在。同时，将建立与启动两个操作合并为发动（init）一个操作。比如，假设前面已对“reader”定义，并且设“buffer1”为一类型为“diskbuffer”的管程的名字，且已在前面发动过了，则

init reader (buffer1);

即对这一系统成分进行发动。其含义即包括分配局部量“block”，以及执行后面的语句。

也可以对几个系统成分进行并发地发动。比如，设 master（buffer1，buffer2），writer（buffer2）分别为已定义好了的用户进程及输出进程，则下面的发动语句即对三个进程作并发的发动：

```
init reader (buffer1),
     master (buffer1, buffer2),
     writer (buffer2);
```

一管程中包含以下几部分：(1) 局部于这管程的数据结构。(2) 由过程表示，允许被引用，并能在上述数据结构加工的操作。为了区别于该管程内部用的过程，称这种过程为外部过程，其说明的标记为 Procedure entry。(3) 一组表示在发动时作准备工作的语句，(4) 它引用其它系统成分及常量的权限，以参数的形式表示之。此外，有的还有 (5) 在此管程内部用的过程和量的说明。

由于一管程内的外部过程有可能被并发的进程同时调用，这里就出现一个排队的问题。事实上，排队有两种：一种即管程内的外部过程同时被调用的情形，这种排队由系统内部予以安排，使用者不需也不能过问，这种排队称为“短期排队”；另一种排队则是由写程序的人自己安排的，其排队的策略、时间的长短由写程序的人所写算法所决定，系统只给他提供一种书写手段。比如，一片磁盘缓冲区上定义 send 与 receive 两送取数的运算，当区满时即不应再往里送，在区空时则不应再从中取。这里就有一个排队等待的问题，这种排队称为“长期排队”。为了让用户能表示这种排队，Concurrent PASCAL 提供了三个手段：(1) 一种数据类型 queue (排队岗)，作为存放处于等待状态的进程名字之用。对于多道进程的排队岗，则以 queue 的序列来表示 (即 array (1…m) of queue)，(2) 一种延迟运算 delay (〈标识符〉)，即当出现需要等待时，即将调用该过程的进程暂时挂起，将其名字存入上面〈标识符〉所示排队岗之中，从此，该进程便无权再对此管程内的数据结构起作用，一直要等到其它的进程调用同一管程并对该排队岗执行一 continue 运算时，该岗内的进程才能重行恢复活动，(3) 一种继续运算 continue (〈标识符〉)，即使此〈标识符〉所表示的排队岗内在等待的进程重行恢复活动的运算。

现在即以 diskbuffer 为例说明“管程”的概念，它表示在盘上分配一片场，以 base，limit 分别表示场的起点与长度，在这场上定义两个外部可引用的运算，“送”(send) 与“取”(receive)。并约定，当场满时即不能再送，当场空时即不能再取，都应等待。送与取每次都是一块 (block) 数据 (block 在前面 inputprocess 中定义过了)。“送”是往“尾”(tail) 上送，“取”是自“头”(head) 上取，数据是在场内依次循环存放如图 15 所示。

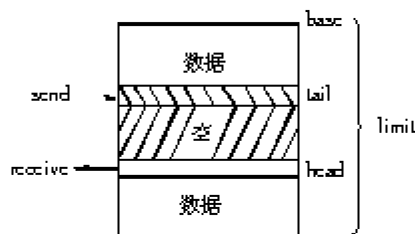


图 15

并假定，在这管程的前面已定义了分配虚拟磁盘的系统类型 virtualdisk. 在这类型内有两个外部过程 write 与 read，均带两个参数：即位置与信息类型。

管程 diskbuffer 即可写成如下的形式：

```
type diskbuffer =
    monitor (〈参数部分〉)
    var disk:virtualdisk;
      sender, receiver: queue;
      head, tail, length: integer;
    procedure entry send (var block: page);
      begin
        if length = limit then delay (sender);
        disk. write (base + tail, block);
        tail: = (tail + 1) mod limit;
        length: = length + 1;
        continue (receiver);
      end;
    procedure entry receive (var block: page);
      begin
        if length = 0 then delay (receiver);
        disk. read (base + head, block);
        head: = (head + 1) mod limit;
        length: = length - 1;
        continue (sender);
      end;
    begin "开始语句"
      init disk(〈参数部分〉);
      head: = 0; tail: = 0; length: = 0
    end
```

与进程相类似，具有 diskbuffer 这种系统类型的系统成分以如下形式引入：

```
    var buffer1, buffer2: diskbuffer;
```

同样由 init 语句来发动：

```
    init buffer1 (〈参数部分〉);
```

值得注意的是别的进程不能直接访问此管程中所定义的数据结构，而必须通过对其外部过程的调用，这种调用应以如下的形式书写：即先写具有此系统类型的系统成分的名字，然后随着一个“小数点”，再跟着此外部过程的名字及参数部分。例如：

```
    buffer. send (block);
```

以及

```
    disk. write (base + tail, block);
```

至于类组（Class），事实上与管程只有一个差别，即它不能同时被调用，因此，不需要执行时的调度，效率比较高。比如，上面 diskbuffer 中用到的 virtualdisk 即可

以用 Class 来表示。

由于系统中各系统类型及系统成分都是一个对一个引用构成一不循环的有向网，故写出时即构成一分层结构。例如，上面的例子写在一块即成如下形式：

```
type
    …
    virtualdisk = Class…end;
    …
    diskbuffer = monitor…end;
    inputprocess = process…end;
    jobprocess = process…end;
    outputprocess = process…end;
var
    …
    buffer1, buffer2: diskbuffer;
    reader: inputprocess,
    master: jobprocess;
    writer: outputprocess;
begin
  init…
    buffer1 (…);
    buffer2 (…);
    reader (buffer1);
    master (buffer1, buffer2);
    writer (buffer2);
end
```

上述处理系统结构的方案有以下的优点：(1) 它将模块的发动与执行区分开来，这正是 ALGOL 60 或 PASCAL 的过程概念所不能表示的一种系统模块所应具有的特征，(2) “管程”概念使某些需要予以保护的公用量能得到合适的保护。它不但限制能施加于这些公用量的运算，而且，通过两种排队，能够避免对这些量同时引用的问题。这些功能，正是系统程序的基本要求。

然而，就我国系统程序工作者的要求来看，Concurrent PASCAL 中这些系统模块概念却不免令人感到有不够合适之处：

(1) 此处“进程”，“管程”等概念都是按照“非常高级”的语言标准来设计的。致使许多原属操作系统范围内的工作置于语言系统中来实现。姑且不谈语言系统的担负增大的问题。至少下面两方面的困难实难回避：(i) 各系统模块的“启动”，必牵涉到建立有关模块对应的信息表（如所谓 PCB 表）等一些预备性的措施，而这些措施还应为整个操作系统的调度程序所引用，因此，与其调度算法紧密相关。故语言中的“启动”，不能脱离具体的操作系统的设计要求而实现。(ii) 管程中实现的两类排队，亦应是操作系统的调度算法的一部分，同样亦存在如何处理两者的

关系问题。由于以上两点，故“进程”，“管程”这些概念不能象其它语言成分那样，独立于所服务的对象的具体算法而实现。这样也就大大降低了这种概念作为语言成分的意义。

（2）事实上，对于语言功能的确定，可以作不同程度的选择。总之，要求的功能越强，代价也就越大。比如，对数据结构访问的权限，规定的限制越详尽，则在编译时能查出的错也就越多，越有利于提高系统的可靠性，但同时编译时的担负也越大，而且用户书写的清规戒律也越多；又比如“管程”或“类组”等概念，一方面对数据结构引用权限规定很严，另一方面并行排队等方面的功能亦很强。比如在 WSL［54］中，则以对这些方面功能的强弱程度的不同而将语言分为三层：其最高层：L_2 中包含类似于 Concurrent PASCAL 中“管程”这样的结构，即不但能实现用户自己安排的长期排队，亦能由系统自动实现短期排队；其次一层 L_1 中则只包含近似“类组”的功能，即系统不作短期排队，只能实现用户安排的长期排队；而在最低一层 L_0 中，则不作任何一种排队，而只能实现过程之间类似于 Coroutine 那样的交互作用的功能。这就表明，Concurrent PASCAL 中各系统类型所实现的各种功能，并不是不能分割的。实际上，“管程”，“进程”等概念是从 SIMULA 67 的“类组”中继承了“引用过程的限制”这方面的特征，并将“类组”的“动态生成”扩充为“发动”以与“执行”相区别，然后增加了“两种排队”的功能。我们认为“引用过程的限制”是一个必要的特征，它和其它权限的规定一样完全可以附加在通常的类型说明或变量说明之上；PASCAL 从“类组”概念中分析出专用过程“New”，它与指针结合即可很自然地实现“动态生成”的功能；至于在此之外“管程”等系统类型所增加的功能，如信息表的建立以及两种排队等，都是与具体操作系统的算法有密切关系的，这些功能加在语言成分之上是否合适值得考虑，它容易使语言与系统程序之间界限不清，而且对系统程序工作者调度算法的选择反而增加了限制，所以，我们认为还是在语言的成分上不规定这些过强的功能为好。

（3）诚然，各系统模块之间的引用关系不是 ALGOL 60 那样的上半格结构而是一种有向网的结构。Concurrent PASCAL 中各系统类型的排列的确也体现了这样一种结构。但是，当系统较大，模块较多，引用层次较复杂时，象 Concurrent PASCAL 那样按其自然的引用关系来排列就很不醒目，从而有可能看错。事实上，在有向网中仍应能分出一大范围的层次，以此为据，在语言中明白规定出系统结构，更有利于使系统模块之间关系清晰，便于阅读和检查。因此，本文作者认为，JOSSLE 在这方面的优点还是值得参考的，应该将 JOSSLE 与 Concurrent PASCAL 所代表的两方面的特点结合起来。各取其精华，融汇在一统一的系统之中。比如，与 JOSSLE 的联系区相应，可在“程序”、“子程序”之上规定“分层结构”这一级以表示“系统结构”。它具有“有向网”的形式。其表示如下：

〈0 层结构〉:: = ‘层次’0［〈联系量信息〉〈系统模块列〉］

〈$\overline{i+1}$层结构〉:: = ‘层次’$\overline{i+1}$［〈联系量信息〉〈系统模块列〉〈i 层结构〉］

此处，〈系统模块列〉是指由〈系统模块〉组成的序列。

为了表示出一数据结构上所限制施行的运算，可在类型定义时，增加一“限制部分”，其中列出所限制施行的过程名字。即：

〈类型元〉∷=〈类型名字〉=〈类型〉〈限制部分〉

〈限制部分〉∷=→[〈过程名字列〉]

这样，也就基本上可以代替“类组”或“管程”的一部分作用。

（二）程序语言的控制结构问题

微观结构包括控制结构与数据结构。我们先讨论控制结构方面的问题。

程序语言的控制结构问题在本文§2（A）中结合关于GOTO语句的讨论已作了较为详细的介绍。最后着重介绍了Zahn的事件语句。这种语句形式由于Knuth [2] 的推崇而受到广泛的重视[118,119]。不过，也还有许多不同的看法[24,83]，这个问题还值得进一步的讨论。

Zahn的事件语句事实上是PASCAL的Case语句的一种推广形式。一般对这语句都强调其后面的分情形部分的安排，认为这可使非正常出口得到一统一的整齐格式。而Halaas则着重由Until引出的事件描述部分（即Until EV_1 or EV_2 or…or EVn），他说：“Zahn构造的优点之一是它迫使程序书写者对可能出现在其基本块（即语句S_0）中的事件附以事件描述（这是促使程序书写者注意他们程序的书写形式的多么巧妙的一着！）”。但Halaas认为Zahn的事件语句有以下几方面的缺点（1）Case部分要求其基本块中执行到所示事件时要用无形的转语句转到所示的情形，（2）Case部分的安排在这种语句多层嵌套时即显得很不醒目。（3）对于一变量取不同符号分情形选取的情况，用Zahn的事件语句来描述即很累赘，反不如Case语句简明。

为了克服上述缺点，Halaas提出如下形式的“事件推动的Detect语句”（以下简称Detect语句）：

DETECT $\langle event\rangle_1$ OR…OR $\langle event\rangle_n$ IN S；NI，

此处$\langle event\rangle_i$可以是一事件常量（如标号），也可以是由事件说明所定义的一事件变量（表达式），对此变量可赋以一符号串，或一整数，或一布尔量为值。

在语句S中则可出现相应的事件常量，其出现的语句具如下形式：

$\langle event\rangle_{i1}$ OR…OR$\langle event\rangle_{im}$ $\{S_i\}$；此处$\langle event\rangle_{ij}$为前面事件描述中某一事件常量$\langle event\rangle_i$或者为前面事件描述中某一事件变量的值。当执行到$S$中这样表示事件出现的语句时，即转去执行$S_i$，执行完跳出此Detect语句。

当此表示事件出现的语句中所有$\langle event\rangle_{ij}$既不在前面事件描述中出现，亦非某一$\langle event\rangle_i$的值时，则$\{S_i\}$中$S_i$等价于空语句。如在一事件出现的语句中无$\{S_i\}$部分，则表示执行停止。

如执行到NI时仍未找出应执行的$\langle event\rangle$，则转回到此Detect语句的开始处重新执行该语句，因此可在Detect语句中表示出循环。

应用此Detect语句可以简单地表示出许多复杂的算法。

例如本文§2（A）中引用过的查找场A的分量$A[1]$，…，$A[m]$中是否有值x的算法，用Detect语句表示即为：

```
i:=0
DETECT tablematch OR nomatch
IN
```

```
i: =i+1
IF i>m THEN nomatch|m: =i; A[m]: =x; B[m]: =1;|;
IF A[i] =x THEN tablematch|B[i]: =B[i]+1;|;
NI
```

有些算法用 Detect 语句表示比用 Zahn 的事件语句简明。

例如设 v 的取值范围为（'a'，'b' …，'g'）。当 v 之值为 'a'，'b'，'c' 时，执行语句 S_1；当 v 之值为 'd'，'e'，'f' 时，执行语句 S_2；当 v 之值为 'g' 时，执行语句 S_3。这命题用 Zahn 的事件语句表示即为

```
Until L1 or L2 or L3
Do if v='a' or v='b' or v='c' then L1;
if v='d' or v='e' or v='f' then L2;
if v='g' then L3
then Case
L1:S1;
L2:S2;
L3:S3;
```

如用 Detect 语句表示则为：

```
DETECT v OR notfind
IN
a OR b OR c  |S1|
d OR e OR f  |S2|
          g  |S3|
notfind  |report error|
NI
```

显然，后面一种表示形式较为简单。

在 Detect 语句中除包含表示事件出现的语句或嵌套的 Detect 语句外，一般还可能出现 IF-THEN-ELSE 语句。Halaas 认为 IF-THEN-ELSE 语句是一种很好的语句形式，因为其语法与语义很一致，易于理解。

可是，正就是这种 IF-THEN-ELSE 语句，近年来有人认为是在 GOTO 语句之外另一有害的语句形式[123]。其论点是：虽然这种语句在简单情况下非常自然，易于理解，但在嵌套多层以后即变得极为费解，很不醒目。本文作者也感觉到，用此形式表示查找算法（如词法分析的算法）时，因需用犬牙形格式打印，只要嵌套层次较多，常使打印纸排不下或每行余下印其它信息的位置太少，以致影响阅读。文[123] 还指出，虽然对于机器说二值的结构比较自然，但对于人来说，多值的分情形语句比二值的条件语句更为自然。文中提出了一种与 LISP 的条件式较接近的分情形语句。这种语句用犬牙形格式打印即比嵌套的条件语句节省空间，因各种情形只需退一格打印。其形式是：

Multiple-Alternative

Case（条件 1）

〈情形 1〉
Case〈条件 2〉
〈情形 2〉
……
Otherwise
〈剩下情形〉
End-Alternative

特别是用这种形式语句表示多重查错的算法更为方便。这种算法往往具有如图 16 所示的图式。

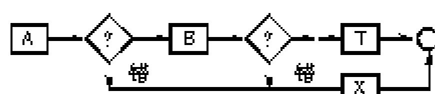


图 16

文［123］采用 M. Jackson 1972 年提出的一种表示这种算法的语句形式并称之为 Posit-Quit-Admit 语句，其中 Quit 即作出错的查询，X 即表示出错后跳出前去执行的 Admit 后的加工程序。此语句的形式即

Posit〈假设部分〉
〈语句体 1〉
Quit〈条　件　1〉
〈语句体 2〉
Quit〈条　件　2〉
〈语句体 3〉
……
〈语句体 n〉
Admit
〈加工程序〉

这种形式语句用于多次查询的算法（比如词法分析）是较为方便的。事实上，也还可以考虑进一步的推广。比如（1）在每种 Quit 情形之后先分别执行一段各自的加工程序然后再转到 Admit 后的语句，（2）还可以循环执行。

本文作者认为更有兴趣的是 Embley-Hansen［85］中介绍的 KAIL 选择句形式，它是分情形语句的一种推广形式。其定义为：

〈选择句〉::=〈前标号〉$|*|_0^1$[〈语句表〉〈控制部分〉]$|*|_0^1$〈后标号〉

〈前标号〉::=〈标号〉—|〈空〉

〈后标号〉::=—〈标号〉|〈空〉

此处“〈前标号〉”后的“*”是用来表示循环的，如没有它即表示不循环；循环时，在〈控制部分〉必须有跳出的控制符，〈控制部分〉为空时即通常的复合语句。

〈控制部分〉::=〈控制符〉〈控制表达式〉〈分情形部分〉|〈空〉

〈控制符〉∷ =if | while | until

〈控制表达式〉∷ =〈表达式〉|〈空〉

〈分情形部分〉∷ = | |〈关系符〉〈表达式〉:〈语句表〉| else:〈语句表〉| |

此处“else:”在一分情形部分中最多只能在其尾部出现一次。

下面举例说明当控制符为 if，while 或 until 时，该选择句的含义。

例如，将 x，y 中较大值送入 max，即可表示成：

(if x | ≥y:max: = x | else:max: = y)

当控制符为 while 时，即表示如下含义：如未找到所要选择的情形，则跳出，否则执行所选择的情形然后继续循环查找；控制符为 until 时，其含义则相反，即如找到所要选择的情形，则执行后跳出，否则继续循环查找。

例如，前面所述查找 A 场的分量 A[1]，…，A[m] 中是否有值 x 的算法，用选择句表示即为：

```
i: =0;
  *(i: =i+1;
    until |i>m:m: =i; A[m]: =x; B[m]: =1;
          |A[i]: =x; B[i]: =B[i]+1;) *;
```

还有如下的一些省写规定：当“关系符”为“=”，或“控制表达式”为“true”，分情形部分在第 i 种情形其“关系符”后的“表达式”的值恰为 i-2 时，相应的“=”，“true”及此值“i-2”均可省。此外，又规定了一些加强功能，如在“分情形部分”的“语句表”末尾还可以引入表示“跳出”或“继续”循环的动作等。

这种“选择句”形式还可进一步推广到表达式之上而成所谓“选择式”，此时，只需将“分情形部分”内的“语句表”改为“表达式”即成。其“控制部分”前的“语句表”则仍可保留，这就是说，在表达式的构造中可出现语句。

由以上介绍知“选择句”的主要特点在其分情形部分。它将进行比较的关系符分散在各不同情形的前面，并将这种结构推广到循环语句及表达式之上，从而使其表达能力比通常的分情形语句为强，且更接近数学的习惯。这一点是可取的。

事实上，此处所介绍的 KAIL 选择句，前面所介绍的 Zahn 事件语句，还有 Dijkstra [97] 提出的如下形式的不确定语句（或称监督命令）：

$$| * |_0^1 (E_1: S_1;$$
$$\vdots$$
$$E_n: S_n;$$
$$)$$

（其含义是，先执行所有的 E_1，…，E_n，设其中值为真的有 E_{i1}，…，E_{ik}，则在 S_{i1}，…，S_{ik} 中任选一个执行；然后循环，如果所有 E_1，…，E_n 之值皆取假值时，则跳出。如括号前没有“*”，则不循环；即只执行一次）都是分情形语句的一种推广的形式。它们与复合语句一起可以用一种共同的形式来表示。这可以说是一种新的复合分情形语句的形式。估计今后将逐步为新设计的语言所采用。可以说是语言控制结构的新发展。

(三) 程序语言的数据结构问题

程序微观结构的另一个重要的方面是数据结构。这个问题早在六十年代已为人们所注意了。正如 A. J. Perlis 在文 [86] 中所指出：语言设计的根本问题是选取什么数据结构的问题。他在 [87] Turing 报告中着重讨论了这个问题，并规定了如下的处理数据结构的方法：

(a) 少数“初始的”数据结构，如整数，实数，类型一致的数组，列表，符号串与文件等在语言中有意义，

(b) 在这些结构之上提供“足够”的一组运算，如算术性的，逻辑性的，抽选性的，赋值性的，组合性的等，

(c) 任何其它数据结构则看成是非初始的，但必须能由初始的所表示…

(d) 对于非初始的数据结构的那组运算则看作为过程来组织。

接着他还列举了关于数据结构的一般功能，比如：(a) 结构定义，(b) 对一结构赋予一标识符，(c) 给定一结构，对其部分结构命名的规则，(d) 对一标识符所指的单元赋值，(e) 引用一标识符所指单元的规则，(f) 组合、抄写及清除一结构及单元内容的规则。

Perlis 这方面的思想，结合着宏扩充的技术[88]后来即发展成为可扩充语言的概念[89]：“可扩充程序语言的提出所依据的一个基本前提是：用户能够通过修改语言的定义而得出适应他的需要的特殊语言”，故“可扩充语言的发展的原动力即企图解决语言设计目标的矛盾这一经典性的问题，这个问题可以陈述为表示的能力与概念的经济的对立”。简言之，即希望语言中只要求设置为数很少的基本概念却能方便地表示出各种不同类型的算法。经过近十年来的研究，虽然可以列举出名目繁多的扩充方法[90]，但最基本的还只是类型扩充与宏扩充。这些扩充手段现在已经是高级语言中被广泛采用的基本语言成分，也是结构程序语言实现其数据结构的基本手段。

在 Perlis 1966 年的 Turing 报告[87]之后另一对程序语言的设计思想不无影响的概念是 Wilkes 在 1968 年前后提出来的[32]。他认为，一语言中存在着两类语法，即“外部语法”与“内部语法”。前者即指语言的控制结构，后者则指其数据结构。他看到了一程序语言中控制结构可以独立于其数据结构和运算而存在，不同专用语言的控制结构（即他所谓外部语法）可以是共同的，其不同之处往往只在其数据结构及其有关的运算（即他所谓的内部语法）的选取不同而已。在文 [91] 中，他明确地指出：“人们现在开始认识到……我们应该更加注意数据存在计算机中的方式，即数据结构……我想指出，在高级语言中，大部分语法以及编译程序是关于作说明，从简单语句形成复合语句及执行条件语句的。所有这些完全与对数据进行操作的语句应如何运行以及数据结构应具有什么形式这类问题无关。事实上，我们有两类语言，其一包在另一之中；一个外部语言，它是关于控制流的，一个内部语言，它是对数据进行操作的，很可能会有一标准的外部语言——或者在少数几种中进行选取——以及一些内部语言，它可以嵌入到其中去。为了适应特殊的情况，如果必要，在嵌入时还可以构造一新的内部语言，它可以在组织控制流方面从外部语言提供的能力得到好处”。

关于结构程序语言的数据结构问题的讨论可以说是上述讨论的进一步发展。大致说来，可以总结为三方面的问题：

（A） Hoare 的数据结构理论及 PASCAL 语言。

（B） 数据结构的一致指引方法。

（C） 面向机器与面向算法的矛盾。

以下分别介绍：

（A） Hoare 的数据结构理论及 PASCAL 语言

高级语言的发展过程中，常常要求对其中出现的概念进行数学的整理，从而取得较为简洁、方便而又完整的形式。自六十年代后，关于数据结构的这类研究很多[92-98]其中尤以 Hoare[93,97]的理论影响最大。它已实现在 PASCAL 语言之中。

Mealy[93]首先将数据与它在机器上的表示区别开来。“数据被看成是表示某些实体（entity）集合上的事实（fact）的集合”，设 E 为实体集，V 为值（value）集，D 为如下的映射

$$\nu: E \rightarrow V$$

所成的集合，他称 D 为数据映射的集合，

$$\therefore D \subseteq 2^{(E \times V)}$$

设 P 为过程的集合，而过程则被定义为如下形式的一种映射

$$\pi: 2^{(E \times V)} \rightarrow 2^{(E \times V)}$$

Mealy 称四元组（E，V，D，P）为一“系统”，并称系统之间的对应为一“表示”，他认为：“独立于表示”这个词比“独立于机器”这个词更合适。过去在这方面发生的问题，“是渊源于一种承袭下来的信念：字符是存贮位置的名字，而不是由数据组织赋予它们的实体的名字”。

Hoare[97]以另外的途径发展了将数据结构与它的表示区别开来对之进行抽象研究的方向。他认为“了解复杂现象……最有力的工具是抽象”。并将“进行抽象的过程总结为四步：（1）抽象：即集中注意于实在世界许多对象或情况所共有的性质，而忽视它们之间的区别，（2）表示：即选择一组字符去表示抽象，以便用来进行交流，（3）操作：即一组对字符表示进行变换的规则，用以预测实在世界进行类似操作的后果，（4）公理化：即一组严格的语句，它表示了从实在世界中抽象出来且为实在世界的操作和表示它们的符号的操作所共同具有的性质”。Hoare 进一步认为：“这一将数学应用于实在世界时所依据的抽象过程同样也就是将计算机应用于实在世界的过程”，而这一抽象过程更为高级程序语言所加强，“应用高级语言代替机器代码的好处可能主要是由于结合了成功的抽象，特别是对于数据”，数据结构理论的关键是数据类型的概念。他比较了数学及逻辑中类型概念与高级语言中相应概念的关系，“在数学理论中，通常习惯于对于个体，个体集，个体集族等作明确的区分……在引进一新变量时，数学家通常立刻说明该变量表示什么样的类型”，而“逻辑学家们一般宁肯以不带类型的变量进行工作，但是，没有类型即可能在集合论内组成悖论”，而“罗素解决悖论的方法即对每一逻辑或数学变量联系一种类型，它说明这变量是一个个体，或一集合，或一集合的集合等等……对我们来说它

的一个有趣的特征是：类型是用来防止逻辑或数学公式中出现错误表达式的”。“数学家、逻辑学家与程序工作者应用类型概念时有高度的同一性。类型概念的显著特征可以概括为：(1) 类型决定一个变量或表达式所能取的值的集合，(2) 每一值属于一个且仅一个类型，(3) 由常量、变量或表达式所指的值的类型可以从其形式上下文中推导出来，而不必知道在计算执行时它的值，(4) 对于每一运算、其运算对象及运算结果必具有确定的类型，(5) 一类型的值及其上定义的初等运算的性质可由一组公理所表示，(6) 高级语言用类型信息去防止或查明程序中无意义的构造，以及决定如何表示及加工计算机中的数据，(7) 我们所感兴趣的是那些已为数学家所习惯的类型：即笛卡儿积，可识别的联合，集合，函数，序列与递归结构”。

基于上述 (1) 这一准则：即将类型看成是决定变量的值集的东西，他选出三种值集作为初始类型（他又称之为 Scalar type 或非构造类型）：(ⅰ) 标准基本类型，即高级语言或计算机上所固有的基本类型：如整数型，实数型，布尔型，符号型等，(ⅱ) 用户自定义的名字集，例如

type color = (white, yellow, red, black, brown),

(ⅲ) 子范围型，即已定义的类型的有序子集，例如

type year = 1900…1969,

type brightcolor = white…red。

基于上面 (7)，Hoare 发展了可扩充语言中类型扩充图式的概念[96]，并指出这种扩充图式与数学中由某些集合组成集合的运算的一致性，尤其有意义的是，它与高级语言中由简单语句组成复杂语句的控制图式也是一致的。比如：

(i) 由不同类型的对象组成的排列：
- 数学：笛卡儿积
- 类型：记录 (record)
- 控制：复合语句

(ii) 由同型的对象构成的重复：
- 数学：正则事件中的 * 运算，即 A^*
- 类型：向量 (vectors) 或序列 (sequence)
- 控制：循环语句或重复语句

(iii) 由不同型的对象构成的选取：
- 数学：集合和
- 类型：类型析取
- 控制：分情形语句或条件语句

只要考虑到文 [97] 中 Dijkstra 提出的执行顺序不事先确定可在当时任选的不确定语句，则还可增加下一种情形：

(iv) 由顺序不确定的对象构成的集体：
- 数学：集合
- 类型：集合型
- 控制：不确定语句

此外，在类型和控制中均有一种任意性过大从而破坏整齐结构的成分，而这种

成分在数学中却是找不到相应的运算的，即

(v) 指向不规则顺序对象的成分：

数学：无

类型：指针

控制：GOTO 语句

还应指出，正如本文§2 中谈过，对 GOTO 语句只要限制它只向前跳即可使之避免产生有害影响一样，对于指针也只要限制它所指对象必须在本层或包含本层在其辖域内的外层中有定义，则也同样即可将它所产生的有害作用避免掉。

由以上的类比可以看出：不论类型扩充还是语句控制的图式都不是人们主观随意安排的，它和数学中集合运算相应都反映了某些基本的逻辑关系。尽管计算机应用对象很不相同，其所需数据结构千变万化，但归根结底总可以由少数基本结构出发，经过上述几种逻辑关系组合而成。这样就为构造程序设计语言的数据结构和控制结构提供了可靠的依据。

上述 Hoare 关于数据结构的理论在 Wirth 设计的 PASCAL 语言中得到实现。Hoare 及 Wirth 在他们的有关文章中多次声言过设计语言的指导思想[97,98,99,100]，而在这些指导原则中本文作者认为有几条是最值得注意的：

(1) 为了得到可靠的程序，他们在选取各语言成分时一再强调的一点是：希望语言中的成分有助于在编译时查出尽可能多的错误。他们说，这语言中成分的“共同而显著的特征是它们为编译程序提供用于检验程序是否一致的重复信息而又能在程序执行时不致产生过分的开销”[98]。(下称“检查原则”)

(2) 他们认为在设计语言成分时应充分注意其实现效率。即表达能力与效率应能兼顾。“某些运算……虽然在设计抽象程序及描写它们的性质时是非常有助益的，但当应用于计算机内大量数据对象时却是效率非常低的……如果这种昂贵的运算作为语言的一部分……许多程序设计者很可能想不到在上机之前应将它们删去”[97]，因此“发展 PASCAL 的一个主要动力是要求得到一功能强并且灵活的语言，而又能在多数计算机上予以充分有效地实现”[98]。故他们强调语言设计与编译程序的实现应同时考虑[100]。(下称“效率原则”)

(3) 重申 ALGOL 60 以来高级语言的传统，强调面向算法而独立于机器。“在算法语言……与程序语言……之间存在重要的界限……我们着重考虑前一种语言”[97]。(下称“独立原则”)

这几条原则再加上上述作为 PASCAL 理论基础的 Hoare 关于数据结构的数学理论，是我们理解和评价 PASCAL 的关键。本文作者认为，这些原则和理论都是非常重要的，Hoare 与 Wirth 的确做了有价值的工作，PASCAL 在这些方面实有不容否认的优点；但也正是在这些方面，由于有时坚持这些原则过分执着，处理得不尽适当，不够自然，也暴露了不少值得商榷的问题[101,102,99]。以下，我们试图分别对此提出一些个人的看法：

(Ⅰ) 由于参考数学概念和形式不够适当所产生的问题

程序与数学的确存在密切的关系。高级语言能取得现今这种简洁的形式，不能不承认数学对它的影响。但程序语言毕竟不同于数学，归根结底它是要在计算机这

样一种装置上实现的，虽然程序语言应独立于某一具体机器，但不能完全脱离计算机的一般特征。忽视了这一点必然要产生削足适履的毛病。Naur [103] 中亦曾指出："我实在不能把数学看作是程序所应奋斗追求的目标，但另一方面，我的确看到了数学与程序之间极大的相似之处"。

在数据类型中提出"独立于表示"的思想是有好处的，但对"独立"应作合适的理解。类型的概念应该独立于具体机器或具体程序中的表示方式，可是它不能独立于一般的计算机概念，最终必须与计算机上一般性的"表示"相联系。这是程序中类型概念与数学中类型概念的基本区别。离开这一点即无法说明为什么要选择"整型"，"实型"，"符号串型"及"布尔型"等在一般计算机上可直接予以表示的对象作为基本类型，而通常数学中并非如此。同样，在通常数学中整数集只是实数集的真子集，施之于后者的运算恒可施之于前者，而程序语言中则不能如此。事实上，每一程序语言的背后，必有一抽象计算机作为其设计的依据，程序语言中的类型不能脱离这抽象机上的"表示"来理解。故语言的初始类型，应该就是那种在一般计算机上均能表示的那些标准类型。（我国的程序语言中应增加"汉字型"）。PASCAL将语言中的类型概念定义为"变量所能取的值的集合"，抽去了它在一般计算机的"表示"，也就抽去了程序语言的本质特征。正如 Morris[103] 指出："到数学中寻找关于类型的一致而精确的概念是一种自然的趋势。这种观点即外延论：类型是值域的一个子集。虽然这一观点在数学中完全合适地为其目标服务。但以这样的方式来定义程序语言中的类型则忽视了某些至关紧要的概念"，他还指出这些概念都是与"表示"有关的。Haberman[101] 亦正确地指出："PASCAL 语言最令人不满的方面是将子范围、类型及结构人为地统一起来……它狭隘地将类型看作仅仅是一组值的集合"。

与 PASCAL 关于类型的定义密切相关的是它对初始类型的安排，除了很牵强地不得不将标准类型（如整型，实型等）也算作初始类型外，还规定了两种新的初始类型，即用户自定义的名字集与子范围型。根据他们关于类型的定义，这些自应作为类型。同时，应该承认，这些类型在某些情况下的确有很好的表示能力。比如由下例即可看出：

```
date = record
        month: (jan, feb, mar,
                apr, may, jun,
                jul, aug, sep,
                oct, nov, dec);
        day: 1 · · 31
       end
homewk = record
          subj: (Lan, Math, Hist);
          assg: date;
          grad: 1 · · 5
         end
```

但是，如不适当地将所有“变量取值的集合”都处理成为类型，则可能产生许多很不自然的现象，正如 Haberman［101］中所指出的：“子范围不能一致地处理成为类型，反之亦然”。例如

var A：array［real］ of integer，

事实上，不但此处 array［〈下标类型〉］中〈下标类型〉位置上不能替以 real，甚至就是用较“合理”的子范围型，如前面定义过的“brightcolor”来替换也是不自然的，费解的，远不如人们所习惯的界对“1··3”更为合适。

问题是如何使语言中能表示出上述用户自定义名字集或子范围型所表示的合理的内容，而又避免其毛病。

事实上，高级语言中的类型是一种提供存贮分配信息的手段，不但初始类型如此，构造型图式亦如此。通常在作变量说明时，不但可定义其类型，亦可指明其初值。比如

var I：integer
⇐3；

既已指出其初值，则类型自明，故可以省掉，即写成

var I⇐3；

也就够了。如果我们再规定量 I 的值不能改变，也就是以 I 作为常量 3 的名字，则可写成

var I〈 = 〉3；

但是，在某些应用情况下，有时使人感到，初值（或常量）只具体规定一个值，未免太窄，而类型 integer 表示任一整数，又嫌太宽，常常需要在这两极端之间具体指明一组可能取的值。这样的情况即要求在太窄的“初值”与太宽的“类型”之外引进“值域”的概念。它是由常量（整数或用户自定的名字）组成的一个有穷序列或有穷集。设 I 的值域为｛1，3，5，7，9，11｝，则可写成：

var I ∈｛1，3···11｝；

至于“用户自定义的名字集”，不过是“值域”的一种形式而已。我们认为这样即可一方面保留下 PASCAL 初值类型的有意义的表示形式而又避免其毛病，且维持了通常习惯的类型概念。

另外，附带指出，前面谈到 Hoare［96］中曾有意义地指出了类型扩充图式与语句控制形式和某些集合运算在逻辑结构上的一致性。但可惜在 PASCAL 中对逻辑结构相同的图式并未采用相类似的记号。比如表示记录的记号是 record···end，而表示复合语句的记号则是 begin···end；表示场的记号为 array···of···，而表示循环语句的记号则是 while···do，或 repeat··· until···等等。我们认为，如以相似的记号表示相似的逻辑结构，较有助于增强使用者的理解和记忆。故本文作者认为参考数学形式采用了如下表 1 所示的记号：

表 I

类　　型	语　　句
记录 $\{C_1; \cdots; C_n\}$	复合语句 $[S_1; \cdots; S_n]$
向量 $\wedge(n)\ \{T\}$	循环语句 $\wedge(n)\ [S]$
类型析取 $\vee\{L_1:T_1, \cdots, L_n:T_n\}$	分情形语句 $[\vee \mid L_1: S_1, \cdots, L_n: S_n \mid]$
指针 $\uparrow\ \{T\}$	跳出语句 $\uparrow L$
集合 $\tau\ \{e \mid B_1: E_1, \cdots, B_n: E_n\}$	不确定语句 $[\tau \mid E_1: S_1, \cdots, E_n: S_n]$

表中 C 表示〈名字〉〈类型信息〉，S 表示语句，T 表示类型，L 表示标号，e 表示集合元，B 表示“属于”关系，E 表示表达式或条件。

本文作者认为，在表示这些逻辑关系采用近于数学习惯的符号比用自然语言文字为好，PASCAL 虽很强调引用数学形式，恰好在这方面似乎忽略了数学形式的优越性。

（Ⅱ）由于处理编译时的检查有时不够适当所带来的问题

强调编译时检查是近年来对系统程序语言的一种被重视的要求[70, 104]，这的确是保证程序正确性的有效措施之一。PASCAL 在这方面很有特色，是值得肯定的。但也有由于处理不够适当所带来一些消极影响。这方面例子之一即对“类型析取”及记录中“可变部分”的处理。类型析取的需要是容易理解的。文［99］对此作过分析。同时，这样定义的量在某一时刻具有什么类型从表示形式上很难确定，因此，也就难以在编译时进行检验。正如文［99］所述：“类型析取的危险在于可能对某一量当时的类型判断错以及难以查错，如果它出现在一赋值语句中，其后果可能是灾难性的。……在 PASCAL 中，类型析取的概念实现在记录结构的可变部分这一形式之中”。文中举了如下的例子：

```
type stackelement =
  record
    case tag: (A, B, C) of
      A: (i: integer);
      B: (r: real);
      C: (b: boolean)
  end;
var S: array [1…100] of
  stackelement;
```

PASCAL 中并规定这样的可变部分在一个记录中只能存一处，且位置必在尾部。在这例中，只有当 S[i] 的类型为 real 时，S[i].r 才是有意义的，而 S［i］的类型为 real 的充要条件是 S[i].tag = B。因此，只要检查后面这条件即可。但是，对于这样一种处理，也有不能令人满意之处。(i) 将取不同情形的条件作为名字置于各情形之前在形式上不够醒目，初学者不易理解其含义。(ii) 在有些应用中不易规定出直觉含义很显明的分情形的条件，(iii) 在有些情况下，可变部分不一定只有一个且不一定在尾部，(iv) 编译时检查 tag 部分所取之值同样是一个问题，文［99］不得不要求再增加象 SIMULA 中“inspect when”那样的语句以达到此目的。

由于有以上的毛病，本文作者感到不若稍稍降低一些编译时检查的要求，采用如下形式的类型析取式：

V{i：integer；r：real；b：boolean}

即限定在类型析取式中，各类型前应加标号。设 t 为以此类型析取式为类型的量，则不允许不带分量名的 t 单独出现，而带了分量名以后（如 $t.r$），则其类型便是确定的了，比如在赋值语句中出现这样的量，其类型的一致性即可和其它变量一样进行检查，故通常类型析取的毛病仍可避免。

由于编译时检查引起的另一个语言成分是“变界场”，这方面的讨论本文不拟介绍[99,106,107]。

（Ⅲ）**PASCAL 中另外两种引起争议的类型**

PASCAL 中有两种新的类型概念引起较多的争议，这就是集合型与文件型。

集合型本来是 70 年代以来受到广泛重视的一种非常高级的类型[108]，它在数据处理，信息检索等方面的应用是较为自然的，是值得肯定的[109]。这种有用的集合概念一般有两种形式[108,110]，一种是，从已给的集合中选出那些使某条件成立的元素，由此组成序列，集合或元素，再以它们构成集合；另一种是：具有某种给定类型元素所成的集合。这两种集合一般分别记成如下形式：

$\{e \mid B_1: E_1, \cdots, B_n: E_n\}$

set(〈类型〉)

这两种集合概念与通常数学中的集合概念是比较接近的，因此较易为人所理解，而且也较为有用。但是，这两种类型实现起来效率均较差。PASCAL 出于对实现效率（即效率原则）的考虑，抛弃了这两种较为自然，较为有用的集合型概念，而将通常集合论语言中一种关于集合的运算：即求给定集合的所有子集的集合，作为类型提出来，表示为

set of T

而此处 T 是一由基本类型表示的有穷集。为了实现起来有效，它要求 T 中元素个数不超过机器的字长，这样即可由二进位数字串来表示相应的元素是否在子集之中。设上述类型名字为 Y，而 b 为一以 Y 为类型的量，即表示 b 为 T 的一个子集。故实际上，这不过是以集合为外貌表示的一种“bitstring”。

对于这样一种集合型概念，显然可提出以下的非难：（1）对 T 中元素个数的限制很不自然，但如没有这个限制，这一类型的实现效率即大受影响，从而也就失去了这样一种集合型概念的意义，（2）这样一种集合型概念和人们所理解的集合概念差别很大，接受起来更不自然且容易用错，它违反了 Wirth 文［100］中提出的下述标准：语言成分要能“避免误解或误用”以及要“方便而且与普遍用的标准一致”。（3）这样一种类型概念即令有时有些用，也并不是通常很有用的，其作用不能与前面所述两种集合型概念相比。

关于这种类型，Mckeeman[111]表示过如下的意见：“…从让目标程序语言具有接近数学的标记并得不到什么好处，将二进位串称为集合并无助于使思维从集合跳到二进位…还有，例如 PASCAL 中的‘集合’型，可能使直觉引歪，由于明显的推广到含 61 个元素的集合，集合的集合等等并不能实现。看来，还是不要用可尊敬的但

并不是很真实的数学形象去掩饰计算机的实际局限为好，除非这种差别是如此的隐蔽以致程序员不致无意中用错”。

总之，本文作者认为，PASCAL 中的集合型概念不能认为是一种成功的创造。事实上，作为通用语言，不包含集合型是可以接受的。但如果想使语言具有较高的直接表示“做什么”的能力，则应不死守“效率原则”而采用前述两种较为自然的集合型概念。同时，也可以提醒用户，这样的类型实现效率不高，希望他能在“逐步求精”过程中换掉。用户是不难记住这些要求的。这样做，比引进 PASCAL 的集合型要自然得多，危险性少得多。

PASCAL 中另一个引起争议的类型概念是“文件型”，它事实上是企图用来表示用户自定义的顺序文件的，更具体些说，就是磁带文件及输入输出文件等。过去高级语言中直接引用机器概念作为标记处理有关这些方面的信息，用户接受起来还是较为自然的。PASCAL 文件型概念之所以引起麻烦，是因为它一方面试图以一种独立于机器的抽象形式给文件型下定义，另一方面又不得不引进一些似是而非的与机器有关的概念如“窗口”，“缓冲变量”等来补充，结果使文件型的概念很不清晰。正如 Wirth [99] 所指出的：“我们不仅旨在给文件及其操作下一简单而又协调的数学定义，而且还注意它有效的实现……”他也承认，他的处理方式是“不够老实的”，并说：文件是“一个重要的概念，也是持续引起问题的根源”。他还提出两种解决的办法：作为系统程序语言时，他主张干脆删去这一种类型概念；而作为通用语言时，他主张保留这一类型，但将 get、put 两操作删去，代之以标明信息从何处来往何处去的 read（f，x），write（f，x）这样的操作。从而即可删掉象“窗口”或“缓冲变量”这类易引起争议的概念。本文作者认为 Wirth 的这些意见值得参考。

上述“文件型”所引起的问题可以说是 PASCAL 死守“独立原则”所带来的消极影响的又一个例子。高级语言与机器的关系问题我们在下面（C）中还要讨论。

（B）数据结构的一致指引问题

这是关于数据结构的表示问题的另一种研究途径。但问题的提出与解决的方法与上面（A）中的问题均有所不同。文 [14] 对这个问题的含义作了较为清楚的介绍：“在一大的系统中，许多抽象数据结构的表示在其尚起作用的时候常常需要改动，因此，最好能将系统中的程序尽可能地与它所访问的数据结构由实现者所赋予它的具体精密的形式脱离开来。这样一来，表示的改动即不需要引起程序的变动，后者即只与抽象有关。……为精确计，我们称在抽象数据结构上的逻辑运算为它的特征。……如果一个问题，它用到一具有类型 T 的对象 X 时，恒以一种简单而一致的记号引用 X 的特征，则它便与类型 T 如何表示无关了，这就是所谓一致指引的问题”。换言之，即每一数据结构由施之于它的一组运算所刻画，一程序引用该数据结构时，不是孤立地引用此数据结构而是引用施之于它的运算，这样即可使数据结构表示的改动不致引起程序的变动[117]。

这个问题的研究最早自可联系到前面介绍过的 Wilkes 外部语法与内部语法分离的思想。但更明确地提出这个问题的首先是 Balzer 的“无数据程序”的概念[112]。

“无数据程序不只是一种新的语言，它将一道程序设想为对一数据值的集合施加一组运算的一次规划，而这一规划独立于所规划的数据值的形式。为了取得这种独立性，必须有一组说明告诉程序系统，对正在采用的某一特殊表示的数据应如何检索和存取”，这样即可以在编制该程序时“不考虑数据加工，存贮分配以及对所加工的数据表示的配比等方面的细节”。

这一思想可以说是将高级语言中“函数说明”概念推广到数据存取检索等方面的结果。正像每一函数所实现的运算算法由用户自书于函数说明的体中一样，Balzer主张将对数据结构的某些标准的存取检索操作也以类似的说明在用户程序之外予以规定。从而，当数据结构的表示改变时只要改写这些说明，而不必改写用到这些操作的用户程序。他将指引数据结构分量的运算与函数运算均表示为如下的标准形式：

〈名字〉(〈参数表〉)

他所考虑的数据结构有：Array，List，Double List，Ring，Double Ring，Structure（与 PL/1 中的相同）以及由这些结构组成的更复杂的结构。他所考虑的一组标准运算是：

DELETE〈区〉，即删去一个区

INSERT〈区〉|BEFORE|AFTER| X(〈下标表达式〉)，即在 X(i) 之前或后插入一个区

REPLACE〈区$_1$〉 BY〈区$_2$〉，即将区 1 换成区 2

ADD〈区〉TO X，即在 X 之上增加一个区

此处所谓区定义为：

〈名字〉(〈下标表达式〉)|TO〈名字〉(〈下标表达式〉)|

即一片连续的空间。显然只要有 DELETE 与 INSERT 两运算其余 REPLACE 与 ADD 也可以定义得出来。

为了增加定义的灵活性，他还规定了一些有关控制的运算等等。

这样一种设计数据的思想的优点即如他所宣称的目标所述，能使程序独立于数据的表示，修改后者不致改动前者。问题是：（1）选出少数一组有关数据的运算是否即可表达出用户可能用到的与数据表示有关的全部运算？（2）这样做等于将一些通常的有关数据的操作都变成函数调用，在效率方面显然要付出极大的代价，得失权衡是否值得？

Earley [113] 中也讨论了类似问题，他将数据结构的“语义”与“实现”区分开来，着重考虑了一套描写数据结构的语义的语言——V 图，即一种边上带名字的有向图，其中包含“结点”，“联线”与“原子”三种对象，每一结点表示结构的一个子部分，每一原子表示一个不可再分割的对象，有向联线则表示访问结构的通道。比如，图 17（a）所表示“列表”用 V 图表示即为图 17（b）。

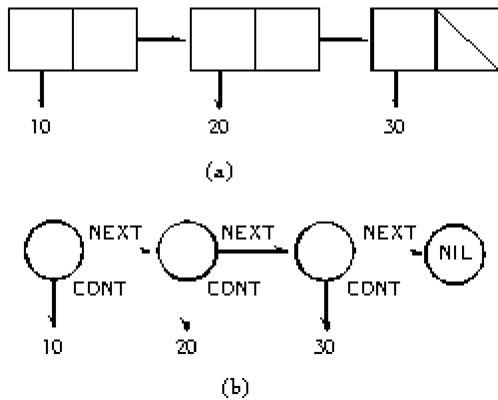


图 17

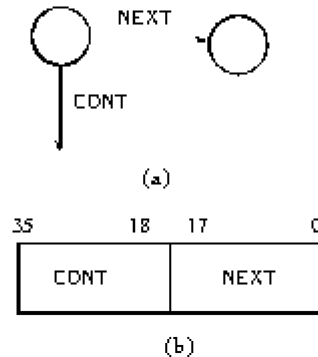


图 18

文［113］明言："事实上，V 图背后的基本思想是：其联线可以表示机器水平上不同的东西，视如何实现此图的方式而不同。一联线可以表示一指针，一下标运算，连续存放，甚至一整套的查找算法"。在 V 图上也定义了一组基本的操作，比如：

ATTACH（SELECTOR A，NODE N）= LINK

表示从结点 N 产生一联线，其名字为 A，它指向 NIL，而

DETACH（SELECTOR A，NODE N）

则表示从结点 N 删去一名字为 A 的联线。

CONTENT（LINK L）= OBJECT

即求出 LINK L 所指的对象。

这种运算还很多，此处不一一列举。所有这些运算即可看成是在数据结构上的一种变换，所有各种数据结构即可从一组表示初始数据结构的 V 图出发经过若干次这样的变换而得。

在语义之外，它还引进"实现技巧"的概念，"实现技巧的基本思想是：人们用更接近反映机器内的数据结构的低层 V 图去表示高层 V 图的实现"。比如，对于某一给定机器说，我们可以令一结点对应存贮中一串相连的字，它的联线对应于其中若干位组成的字段，每一原子即对应二进位数字的一个图式。比如图 17（b）中一个元素（图 18（a）），即可对应于图 18（b）。其中一个结点对应一个 36 位字长的机器字，其中两个联线对应两个 16 位的字段。

Earley 的 V 图似比 Balzer 的概念灵活，但仍是试图找出一组施之于数据结构上的基本运算，由此刻画出任何数据结构。因此，它仍不免存在 Balzer 无数据程序所遇到的困难。

Liskov 与 Zilles 的"抽象数据类型"[10, 72, 113, 116]则似乎在这方面向前较大地前进了一步。其中心思想仍然与前两工作相同，"一抽象数据类型定义的一类抽象对象，它们完全由可施之于它们之上的一组运算所刻画"。但他们抛弃了寻找一组基本的施之于数据结构上的运算的意图，"设有理由祈望一组先定的运算足够处理每一抽象数据对象"。他们采用了与 SIMULA 67 中 Class 概念非常近似的做法，引进一种称

为 Cluster 的模块形式，在其中一数据结构与可施于它之上的运算同时定义，用户只有通过引用所给的运算才能接近该数据结构。

一 Cluster 由以下几部分构成：（1）该 Cluster 的名字，参数以及定义此 Cluster 的一组运算的名字，（2）对象表示：即表示出此 Cluster 所定义的对象是怎样由更基本的数据类型构成的，其定义形式是

rep〈参数部分〉=〈类型定义〉

（3）对象产生程序：即列出当由此抽象数据类型产生出一新的对象时所应执行的一段程序，其作用与过程体类似，（4）运算定义。

下面即一个 Cluster 的例子。

```
Stack: cluster (element-type: type)
        is push, pop, top, erasetop, empty;
        rep (type-para: type) =
        (tp: integer;
         e-type: type;
         stk: array [1…] of typepara);
        create
              s: rep (element-type);
              s. tp: =0;
              s. e-type: =element-type;
              return s;
        end
        push: operation (s: rep, v: s. e-type);
            s. tp: = s. tp + 1;
            s. stk [s. tp]: = v;
            return;
        end
        pop: operation (s: rep) return s. e-type;
              if s. tp =0 then error;
              s. tp: = s. tp - 1;
              return s. stk [s. tp + 1];
        end
        top: operation (s: rep) return s. e-type;
             if s. tp =0 then error;
             return s. stk [s. tp];
        end
        erasetop: operation (s: rep);
             if s. tp =0 then error;
             s. tp: = s. tp - 1;
             return;
```

```
end
empty : operation return boolean;
      return s.tp = 0;
end
end Stack
```

此处 Cluster 概念与前面介绍过的 SIMULA 67 中的 Class 及 Concurrent PASCAL 中的 monitor 很相近，故不必多加解释，只需指出它们的不同之点即可。Cluster 与 Class 的不同，据文［116］说存在如下“重要指导思想的差异”，即 Class 中“每一特征和函数在该 Class 所定义的分程序内可以普遍访问，用户对其表示形式完全清楚。与此不同，一 Cluster 的 rep 在 Cluster 之外是不能访问的，只有 Cluster 中的运算才提供了访问 rep 内容的唯一途径，而且仅仅 Cluster 中所定义的运算的一部分能由外部引用”。至于 Cluster 与 monitor 的不同，由于 Cluster 中不存在并发过程同时调用的问题，故无需考虑排队。

Liskov 与 Zilles 的 Cluster 中的运算事实上也就是一种函数或过程，它可根据需要由用户来定义，故比 Balzer 等提供的有限种基本运算要灵活得多，从而克服了上述 Balzer 概念第一方面的缺点，但对于第二方面的缺点——即效率问题，则仍然存在。这恐怕是一致指引问题所难以避免的困难。

Cluster 概念正如前面谈过的“管程”一样为类型检查提供较多的重复信息，故可增强程序可靠性。

文［116］已指出，通常程序中除 Cluster 外，还常需出现常见的那种过程和变量，故一致指引问题并未因 Cluster 概念而全部解决。不过对经常需要改动的数据结构以此方式引入，可减少改动程序的机会。

Cluster 概念的作用自不止于一致指引一个方面，它还具有 SIMULA 67 的“类组”的其它功能，如数据结构的动态生成以及对引用权限的限制等。正如我们在讨论“管程”时已指出的那样，所有这些功能均可通过在类型说明上附加限制信息以及引入类似于 PASCAL 中的“New”过程那样的手段而解决。因此，对于那些经常需要改动的数据结构，亦可以限制引用过程的办法使之具有一致指引的功能，从而亦可使加工这些数据结构的程序减少其改动的机会。

（C）面向算法与面向机器的矛盾

这是程序语言领域的一个老问题。面向算法的语言（如 ALCOL 60）的优点是易懂，较难出错亦较易查错，及便于修改维护等，面向机器的语言如汇编语言亦有它的优点：由于能利用机器的各具体特征，故可用它编出时空效率较高的结果程序，亦较灵活[131]。

系统程序具有许多特点[123, 128, 131]，而其中二点尤为重要：（一）由于其程序量大，容易出错，出错后影响极其严重，故普遍要求系统程序语言特别重视保证可靠性，而这一点正是面向算法的高级语言的长处；（二）另一方面由于这程序是在机器中时刻起作用的程序，故对时空效率要求特别高，而这又正是面向机器的语言的特点。因此，可以说，系统程序语言既应面向算法又应面向机器，这一对矛盾显得

特别突出。如所周知，结构程序设计思想的产生与系统程序的要求有密切的联系，因此，面向算法与面向机器的矛盾也不可避免地在结构程序语言的设计思想中暴露出来。

概括起来，在系统程序语言方面处理面向算法与面向机器的矛盾有以下几种途径。

（ⅰ）认为单纯用面向算法的高级语言亦可解决效率问题

这是一种较为极端的观点，即认为面向机器语言所强调的时空效率问题，用面向算法的高级语言亦可解决。因此，即不必在高级语言中考虑面向机器的语言成分。这方面最有代表性的观点是 Wells [51]，而在具体语言中较有代表性的是 PASCAL。

Wells [51] 的基本思想是用高级算法语言描写算法，而依赖其编译程序来解决其效率问题。“语言与编译系统的部分区别在于算法一级的数据结构（如复数，网络图，排队岗等）与硬件表示一级的存贮结构（如指针，存贮块，按位图式，浮点及定点数等）的区别，程序编写者只要根据其算法的要求关心和选择数据结构，而编译系统则选择与其目标机器功能一致的存贮结构”。因此，“足够高级的算法语言可用来很好地完成面向机器的任务”。

此外，也有不少人认为在系统程序或问题程序中与机器有关的部分事实上只占很小的比重，因此不值得为此在语言一级引进面向机器的成分。比如 Horning 认为“罕有一个问题中有 5% 以上的实际代码是以任何有意义的方式依赖于机器体系的”。Ichbiah 也认为：“诚然，我们在系统程序中需要某些接近机器特征的成分，但这不过是一些例外情形而已，很可能这一部分的比重……是如此之小……以致完全可以用一纯粹高级语言来对付”[113]。

只要看看当前实际情况，即使人感到这些意见还难为人们普遍接受[123]。从机器体系说，面向高级语言的体系并未流行，面对种类繁多的体系结构，高级语言的效率问题仍尖锐地存在；从编译技术说，全局优化的方法还不令人满意，不但优化后的程序使人难读，而且由于其工作量过大可能影响系统的可靠性，这是系统语言所难以接受的；从语言方面说，象 PASCAL 那样试图去找一些既能保证效率又是高级的数据类型（比如记录中的可变部分或集合型等），结果是不能令人满意的，这一点我们在前面已经论及。由于 PASCAL 没有面向机器的成分使得一些系统程序工作者感到不足，比如 O. Lecarme 指出：“PASCAL 不是为写系统而是为教学设计的，它缺少那些与硬件发生密切联系的每一种功能。”[129]而恰好这一部分与机器关系密切的系统程序往往是经常起作用的程序。其效率的影响是很大的，难以忽视。

（ⅱ）允许嵌入汇编（或机器）语言程序段的高级语言①

这是人所熟知的办法。许多语言均已采用。不过，在这方面值得一谈的是 PL/S. 它较为集中地代表了这一途径。PL/S 是 IBM 公司用来书写系统程序的语言，由于长期保密而为人们所注意。

PL/S 是从 PL/1 经适当删补而成。它包含了 PL/1 中如下的一些语句形式：ASSIGNMENT，CALL，DECLARE，DO，ELSE，END，ENTRY，GOTO，IF，PROCE-

① 作为系统实现语言的宏指令系统亦包括在这一类之中。

DURE，及 RETURN；其数据类型有：BIT，CHARACTER，ENTRY，FIXED，LABEL，POINTER；复合数据结构有：ARRAY 与 STRUCTURE。

汇编语言书写的程序经二种途径嵌入到 PL/S 之中：（1）由 GENERATE 语句引入，对这语句的操作对象（事实上是汇编指令），编译程序不能指称，只能将它直接传递给后面的汇编程序去处理；（2）“内设指令语句”，即将某几条汇编语言的指令固定作为语言的语句，这种指令的操作对象是编译程序可以指称的。

除此之外，PL/S 中还有以下一些面向机器的措施：（1）Respecify 语句，它使程序编制者能限制编译程序在所产生的代码中用哪些寄存器；（2）PL/S 中不另设输入输出，存贮分配，并行操作等方面的语言成分，这些功能或者由汇编语言的宏指令表示再由 GENERATE 语句引入，或者由 PL/S 的宏请求语句[①]表示；（3）对于编译程序所产生的结果指令，用户可通过“Compiler option”或“Procedure option”提出某些要求：比如要求结果指令是在 360 系统还是在 370 系统上执行；又如所指的过程是否要求可重入等等。

显然，这样的语句可以写出质量很高的程序，但是，这样的程序在阅读清晰，查错功能方面却比高级语言要差，因为，它具有汇编语言类似的缺点，结构化的程度较低。

（iii）面向机器的高级语言

虽然对这种语言的定义可以有许多不同的说法，但一般说来，最典型的面向机器的高级语言通常都具有高级语言的控制结构而同时又能表示出汇编语言的数据结构。这方面的代表是 PL360 及 BLISS。而 BLISS 则包含更多的高级语言成分而且具有结构程序语言的一些特征，比如它是第一个删去 GOTO 语句的高级语言。

关于 BLISS 的控制结构，我们在本文 §2（A）中已谈过了，此处不赘。而其数据结构则与汇编语言颇为接近。它称一组相连的二进位为一“段”（field），每一段可以取一名字，这名字的值称为一“指针”（pointer），全字长也构成一段，这样的段的名字的值事实上即一地址。一指针一般表示为

$$E_0\langle E_1,\ E_2,\ E_3,\ E_4\rangle$$

此处 E_0 即一真地址，E_1 表示该段是这字右起第几位开始，E_2 表示该段共占几位，E_3 表示变址部分，E_4 表示是否为间接地址，如果 E_1，E_3，E_4 不写即表示其值为 0，E_2 不写即表示其值为该机器的全字长。设 X 为一名字，该语言以“·X”表示“X 的内容”。比如“X+1”表示 X 所对应的地址加 1，而“·X+1”表示 X 所指单元的内容加 1。此外，还以“global x”，“local y”，“external z”，“register w”，“own u”的形式分别直接标明所示 x，y，z，w，u 为全程量，局部量，外部量，变址寄存器及 own 变量。BLISS 中还可表示出 PDP-10 机器指令，其形式如下：

$$OP(E_1,\ E_2,\ E_3,\ E_4)$$

此处 OP 表示 PDP-10 指令的操作符；E_1 是一表达式，其值的最低 4 位即表示指令中累加器（A）段的信息；E_2 是一表达式，其值的最低 18 位即表示指令中地址

① PL/S 的宏请求语句事实上与常见的宏定义相似，定义中带参数，在编译时由预先加工程序代入定义中所示的符号串。

（Y）段的信息；E_3 是一表达式，其值的最低4位即表示指令中变址（X）段的信息；E_4 是一表达式，其最后一位即表示指令的间接地址特征位（I）。E_1 到 E_4 如果缺即表示其值为0。

BLISS 更特别规定了一种能让用户自己定义选取分量算法的 Structure。

Structure〈名字〉[〈形参表〉] =〈表达式〉 map〈名字〉,〈名字表〉

例如 Structure arr [i, j] = (· arr + (· i - 1) * 10 + (· j - 1); map arr, x: y: z; 即表示 arr 是一个二维场，其分量选取算法为“=”右边所示，x，y，z 是 arr 这种二维场。这样的“Structure”显然部分地实现了 Balzer 所述“无数据程序”的思想。

由上面的介绍可以看出 BLISS 试图将面向机器的语言成分与结构化思想结合起来。但 BLISS 所写程序中这两方面纠缠在一起，无清楚分界。分层语言则注意了这一点。本文§2 曾指出：由于（i）由顶向下的逐步求精方法要求程序编写的过程分层，从而导致描述语言的分层，（ii）系统程序的总体结构应该是一种分层结构，且愈靠近其核心部分愈与机器联系密切，从而往往要求用不同层次的语言进行描述，这又导致语言的分层[123,126]。从这样一些要求来看，BLISS 还未能提供解决面向机器与面向算法的矛盾的理想途径，更为合适的途径应该是分层语言。这一点我们在本文§2 所引 GOOS [56] 及 Knuth [2] 中的话已说得很明白。并且还应指出：在这分层语言序列中，各层语言的控制结构基本上是共同的，它们的差异主要在数据结构方面。下面是几种不同类型的分层语言。

（iv）按一致指引思想分层的二层语言

我们前面讨论数据结构的一致指引问题时，其主要出发点是为了系统便于修改从而要求程序独立于数据的表示，故所介绍的方案（如 Liskov-Zilles 的 Cluster）没有着重强调语言分层的概念；而 D. T. ROSS[117]则着重从逐步求精的程序设计方法出发提出一致指引问题。他将这种方法称为“由外向内”的设计方法，并要求有一种统一的“软件工程语言”能用来描述逐步求精的各个阶段，他将一信息加工系统分为三种模块：存贮模块，动作（ACTION）模块与选择模块。事实上这三种模块即分别对应着“存贮”，“算术运算”与“控制结构”三方面的功能。在逐步求精过程中，只是将各模块的交接信息越改越精细，越改越面向机器。为了做到这一点，只要求对各种运算的对象采用一致指引的方式，在求精过程中只需改变指引的算法而不必改变程序中对运算对象的指引。他这些思想更具体的实现即 Ichbiah 等所设计的二层语言 LIS[134]。这语言的主要特征即“在数据定义时分为二步，即第一步考虑语义性质，第二步才处理它们的有效实现…而程序算法则只用语义性质来陈述。”— LIS 程序是由一组分层联系并可分开编译的单元所组成，这种单元有二类，一是程序段（Program segment），一是数据段（Data segment），而在数据段的后面有可能跟随一实现部分（Implementation part），这种段中包含数据和类型的说明，也可能包含其它可分开编译的子程序说明（即通常所谓过程或函数说明），如图19所示。

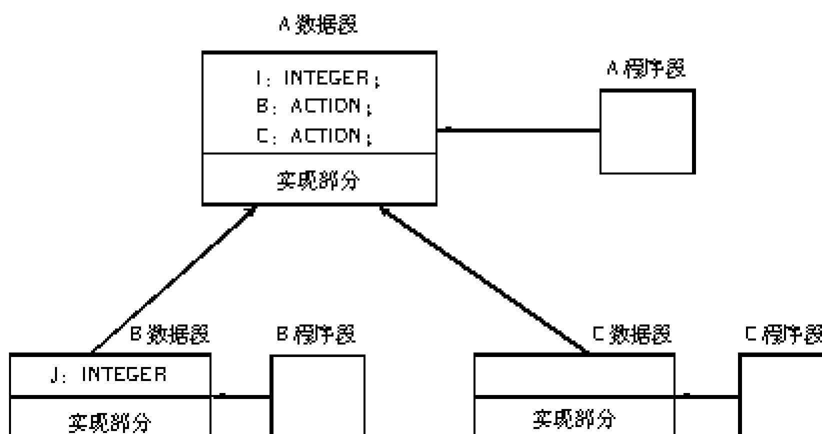


图19

在数据段或程序段中，通常只是包含高级的语言成分，而在实现部分则是用由高级成分与低级成分相掺合的语言来描写。从表2中可以看出高级语言成分与低级语言成分的对照：

表2

高级成分	低级成分
说明	实现部分
名字	地址
Domain 表示无顺序的集合	ZONE 表示有顺序的场
例 FAMILY：Domain of PERSON	例 SPACE：ZONE of 1000 WORD
带类型指针	不带类型指针
例 x，y：REF FAMILY	例 Z：REF
带类型运算	无类型运算
例 X：=Y	例 X：=Y
分量选取	当时赋型的分量选取
例 X. AGE	例（Z AS PERSON）· AGE

下例说明一致指引的含义。

例（1）

```
TYPE PERSON =
PLEX
    AGE：(0…100)；
    BIRTH：(0…2000)；
END；
…。
PERSON · AGE
…
```

（2）

```
TYPE PERSON =
PLEX
    AGE：(0…100) ACTION；
```

```
        BIRTH: (0…2000);
    END;
    …
    PERSON · AGE
    …
```

例中（1）在选取 PERSON · AGE 时即按通常记录选取分量的方式进行，而在（2）中选取 PERSON · AGE 的算法则另由 ACTION 来规定，这部分的算法则另写在实现部分之中。此处 ACTION 有些类似前面介绍的 BLISS 的 Structure 中〈表达式〉部分所起的作用。一般地说，在高级语言成分中不考虑各特征在集合（Domain）中的顺序，位置，容量等，只作为语义性质来对待；有关数据存贮顺序，有效结构，动态存贮或解除的具体算法均在实现部分来具体规定。由于通常的系统程序往往具有多层的总体结构，LIS 的结构似与之不对应。事实上较为人注意的分层系统程序语言常为多层的[123, 126]。

（v）多层的分层语言

较早的一个多层语言系统是 Burrongh 公司用于书写系统程序的三层语言 EXTENDED ALGOL，DC ALGOL，与 ESPOL，其中第一级是独立机器的而最后一级则是依赖于机器的，第二级则处于二者之间。

EXTENDED ALGOL 是 ALGOL 60 的一个扩充，主要增加的内容有：（1）按位运算，比如"X · [5:2] = 3"表示"X 的第 5 位起二位之值为 3"。（2）高级宏指令，比如在给了宏定义"DEFINE STATUS = [5:2]#"后，程序中出现的名字"STATUS"在编译时均替成"[5:2]"，宏定义亦可带参数。（3）在通常的过程定义前加"EXTERNAL"即可使该过程能独立进行编译；在通常的过程定义前加"Task"，该过程即可作为进程或 Coroutine 对待，对它可赋予各种"Task attributes"，并可根据这些"Task attributes"对它进行控制。比如"STATUS"即是一种这样的"Task attribute"，它表示此 Task 当前状态是"排队"，"活动"，"挂起"或"停止"；又如"Maxproctime"也是一种"Task attribute"，它表示允许此进程执行的最高时限。这样的"Task attribute"还很多，不必列举。

DC ALGOL 主要是为书写远程终端数据通信系统而进行的扩充。比如，它可作"Message"说明：

Message MESSAGETOCONTROLLER;

并可对所说明的标号赋予一符号串：

Replace MESSAGETOCONTROLLER BY "COMPILE MY/JOB USING COBOL";

并可为所定义的 MESSAGE 分配一定量的暂存空间：

Allocate (MESSAGETOCONTROLLER, 8);

还可设置排队存放 Message 的"queue"，比如如下的运算

Insert (MESSAGETOCONTROLLER, CONQ, BL);

即表示将 MESSAGETOCONTROLLER 中的 Message 存入名为 CONQ 的 queue 之中，至于是从头上还是尾上存入，则视布尔式 BL 的值为 0 或 1 而定。与此相对应，则有 Remove 运算：

Remove（ARRAYID，QID）；
即表示将“queue”QID 顶上所存的信息移出，存入名为 ARRAYID 的场中；此外，对于 queue，还可以有其它运算，如询问其深度，将两个 queue 联成一大的 queue，将一个 queue 分成二小的 queue 等等。

第三级 ESPOL 则是最面向机器的语言，它可直接表示出访问机器内存及其它部件的信息。它主要用于书写 B6700 操作系统最核心的部分。

由以上介绍可以看出，这个分层语言还缺乏清晰的结构和统一的思想，给人一种混杂的感觉。文［126］作者也承认：“本文所谈的许多思想在陈述时无疑将感到陈旧，这是由于用这工具实现的系统发展太快所致。…但任何与这系统有过接触的人都不怀疑，这样一种途径（即分层语言）对实现软件系统的价值”。

新进发展的分层语言系统 WSL[34,33]在结构化方面考虑得较为周到，正如其设计者所述：将语言分层“可以看成是结构程序设计的原则用于程序语言设计的一次试验”[34]。

WSL 主要根据数据结构与总体结构的复杂程度不同而进行分层，共分为 L_0，L_1，L_2 三层。

L_0 是以静态结构为其主要特征的系统程序语言。在数据结构方面其主要特点是：

（1）基本数据类型有：int，real，bit（bool），char，bits，chars，其中又区分 long 与 short，此外还有 ref。如在所有这样的类型前加 const，表示为常类型，即不可改变，如“const real 3.14”即表示常实数 3.14；“const ref int *x*”即表示一不改变的常指针，它指向一整型变量。

（2）复合数据类型有：row，table，stack，queue，structure 与 collection，collection 即集合型，包括枚举其中元素的显性集合及由条件决定其元素的隐性集合，对于 collection 可进行∩，∪，¬ 等运算；structure 即通常的记录；前面四种类型均在说明时指明其元素个数与长度，均为常界，均占线性连接的一片空间，且在 L_0 中复合型不许再包含复合型的分量。对这四种集合型，还备有相应的运算：如对于 stack，有 pop 与 push；对于 table，有送项的运算 enter；对于 queue，有存取分量的运算 put 与 get，还有指出其第一分量位置的运算 first，此外，对于 stack，table 与 queue 均有指出其最后元素位置的运算 last 及长度运算 length。

（3）还有一不规定具体类型的“类型”free。

（4）在 L_0 中无类型自动转换。

（5）此外，有 strip 与 dress 两运算分别用于剥夺或赋予类型。比如，“free strip *y*”“表示剥夺 *y* 的原有类型使之成为 free 型”；“bit dress *x*”即表示“赋予 *x* 以 bit 型”。

在总体结构方面 L_0 具有以下特征：

（1）不包含嵌套分程序结构，只有静态存贮管理，亦不包含废码收集手段。不过，系统中备有动态分配存贮的算子。

（2）以类似于“EQUIVALENCE 语句”的形式进行单元重叠分配。

（3）所有算式均具有三地址指令形式，每式中最多只许包含一运算符，无优先

性的规定，亦无括号。

(4) 有四种子程序 (Routine)：procs，functs，coops 与 coprocs，前二种即通常的过程及函数，而 coop 是由一组相互作用的 coproc 构成的子程序，它即可表示出通常的 coroutine。在 L_0 中，不允许递归亦不允许并发，过程的参量只允许是基本型且是 const。

(5) 还包含丰富的调试措施。

L_1 在数据结构方面取消了复合型不许包含复合型分量的限制。在体系结构方面引进了 ALGOL 型的嵌套分程序结构及废码收集手段。对过程参量的限制也取消了。在 coop 中取消了对 coproc 非递归的限制，可以通过一些有关调度的运算构成半并行的算法，用进程对并发进程进行模拟。

L_2 具有描述并发进程的能力，其中包含与前面所述 Concurrent PASCAL 中 monitor 概念相近的成分。

从上述关于 WSL 的简介中已可看出，该系统较注意分层标准的逻辑性。大体是按照语言成分的逻辑结构由简到繁分层。比如它将逻辑结构较复杂的类型复合，分程序嵌套，构造型参量，递归 coproc 等从基本语言中划出列入 L_1，再将更为复杂的并发进程列入 L_2，这样就使系统的结构化程度较好。这是此语言系统的优点。但另一方面这系统对面向机器的语言成分注意得比较不够，同时 L_0 似觉还不够简单。这是令人感到不足之处。

事实上，存在着两种语言分层的标准：一是按照由面向机器到面向算法的不同程度分层，这样一种分层较符合逐步求精以及系统程序分层结构的要求；另一是按照逻辑结构由简到繁的不同程度分层，这样一种分层，如能做到使其最低层语言既非常简单又功能强到足以进行自编译，则对于系统的移植以及构造“反复扩展”[14]的结构化编译系统均较有利。关于可移植性及结构化编译系统的问题我们在下节中再介绍。现在应该指出的是这两种标准的要求并不完全一致，即逻辑结构上最简单而又能满足可移植性及作为“反复扩展”的核心要求的那部分语言成分并不就是最面向机器的那部分语言成分。本文作者认为一较为理想的结构化分层语言系统应该是能使这两个标准都适当得到满足的协调的系统。

在此，我们想重提在本文§2 (*B*) 中曾指出过，并在本节讨论宏观结构时又谈到的看法：即一结构化分层语言系统最好是一“语言族”，它首先应分为两大层：作为整个语言族的基础应为一“公共基础语言”，在此之上再扩充出并列的各种面向问题的专用“体系设计语言”。由此而形成一语言族。

语言族的结构应如下图20所示，其中 C 表示“公共基础语言”，它本身又可根据面向机器到面向算法的程度不同，以及其逻辑结构由简到繁的程度不同而分成若干层。在此基础上生成各级面向问题的“体系结构语言” S_{11}，…，S_{1k} 等：

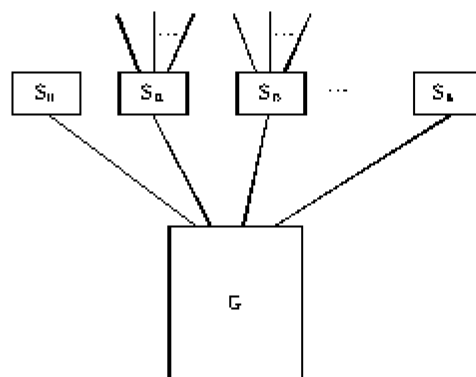


图20

目前，有名的结构程序语言中 SIMPL 语言族接近于上述结构。就文［141］的介绍看来，SIMPL-X 与 G_0 相对应，SIMPL-T 则与 G_1 相对应。在 SIMPL 族中缺少面向机器的一层语言。

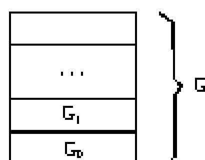


图21

据文［141］［142］的介绍，SIMPL-T 是“作为一族程序语言的基础语言来设计的”其特征是：

- 一程序由一组分开编译的模块所组成。
- 每一模块包含一全程量的集合及一过程或函数的集合。
- 语句类型为：赋值，条件，分情形，过程调用，跳出，返回。
- 数据类型为：整数，符号及符号行。
- 包含广泛的一组有关数据上的运算和内部转换函数。
- 有由具有上述数据类型的分量组成的一维场。
- 过程或函数可由用户选定是否递归。
- 基本类型可以是指引性的（by reference）或赋值的（by value），场则只能是指引性的。
- 过程或函数中不能再包含函数或过程，亦不允许以过程或函数作为参量。
- 无分程序，但有复合语句。
- 各分开编译的模块可有公共过程，函数和数据。

而作为 SIMPL-T 核心的 SIMPL-X 则在 SIMPL-T 之上增加如下的一些限制：

- 只允许一种数据类型即整型。自编译需用到符号行时，则用 FORTRAN 中将符号行压缩存于整型变量中的办法处理。

- 对基本类型的参量只允许是赋值性的。
- 所有过程与函数都是递归的。
- 标识符只前12个符号有效。
- 分情形数目限于0到99。
- 逻辑运算（与，或）的两个运算对象的值都算出来，不进行"条件跳"。

我们认为，由图20，21所示的这样一种分层语言族，较有助于解决结构程序语言所面临的各种矛盾：如统一的结构与各自的特殊要求的矛盾；面向算法（可靠，易懂）与面向机器（效率）的矛盾；功能强与语言小的矛盾；编译过程应由小到大逐步扩展的要求；体系复杂与移植简单的矛盾等等。在此，不能不使我们回想起本文引述过的Knuth［2］中提出的一些关于语言设计的理想："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）"。…"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，我们将能表示存贮控制，寄存器分配，下标范围检查的压缩等等，用一完整的系统，它将既能查错以及对程序转换进行分析，又能用高层的语言进行交流"。图20，21所示的语言族，是否可以说是走向这一理想的一种尝试？

最后，有一点应该在此说明。我们在本文§2谈到结构程序设计的定义时已经提到Dijkstra等在讨论此概念时常涉及程序正确性的证明；我们在本节开始谈到结构程序语言时又引用了Presser［66］中的话：即这种语言的"再一个目标即要使验证更为清楚"。故人们自然盼望，我们在介绍结构程序设计或结构程序语言时亦当随之介绍程序正确性证明或检验的方法。但是，我们在本文范围内将不会这样做。理由是：（ⅰ）已有的程序正确性的证明方法与检验方法很多，并无定论，很难以一种方法为依据来介绍语言的特征；（ⅱ）由于这类方法很多，只有用专文介绍才能谈清楚，将这么庞大的内容包括在本文内显然是不合适的；（ⅲ）不少专家已经指出："对于正确性证明有助的结构化手段也同时就是那些最有助于使人们读懂的手段"（［132］中Dahl语。Knuth［2］中亦有类似的看法）。因此，避开谈正确性证明和检验的具体方法亦同样可以把结构程序语言应具有的特征讲清楚；（ⅳ）就我国当前读者的需要看，首先是将结构程序设计的概念及结构程序语言的特征了解清楚。如果用很大的篇幅来谈证明和检验方法反而容易使本文所要达到的目标引偏。

当然，结构程序设计与程序正确性的证明和验证还是有十分密切的关系。读者如欲了解这方面的内容，可以参看文［140］的综合介绍，以及IFIP Congress 74与Proc. Intl Congress of Reliable Software中有关这方面的文章。此外［148］第一讲中引述的文［145］，［146］等亦值得一读。

§4 结构化的编译系统

结构程序设计方法应用最有成效的领域是操作系统，这方面著名的工作很多[149,79,149]，国内也有人开始在考虑这类问题[130]。本文作者不拟把有关这方面的内容包括在本文之内。希望有人对此作专文介绍。我们在本节内试图讨论一个范围较

小的问题，即如何构造结构化的编译系统。对于这个问题，由于侧重方面的不同，目前主要工作可分为两类：

（1）侧重强调程序模块化问题，即如何构造模块化的编译系统，如 McKeeman[23,48]。

（2）侧重强调由顶向下的设计方法，即如何用由顶向下的设计方法构造编译系统，这方面有代表性的工作如 Ammann[47]，Basili-Turner[141]。

下面分别介绍 McKeeman，Ammann，Basili-Turner 这三方面的工作：

（一）模块化的方法

McKeeman[23]认为“编译程序愈是精巧，它就愈有必要应用结构程序设计的技术，比如，模块化，……结构化技术应用还与编译算法的内在结构有关”。因此，他的“目标就是要显示出编译程序中的自然分界，由此提供将编译程序分划成模块的指导”。事实上，多年来构造编译程序的实践，已逐渐使编译程序的结构有一个对多数情况均可适用的分块的轮廓。他将这分块的轮廓，概括为纵向分块与横向分块两个方面，从纵向分块看，一编译程序应可分为界限分明的七步，如下图 22：

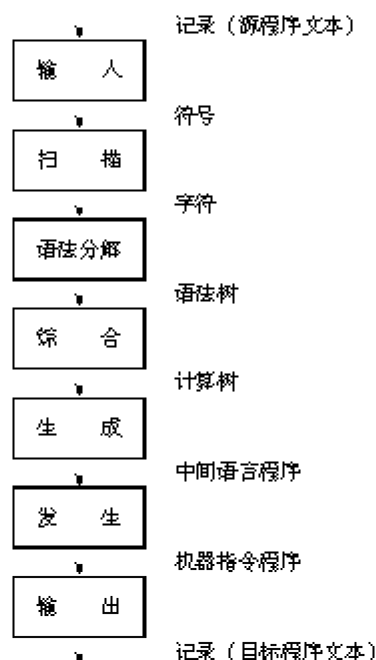
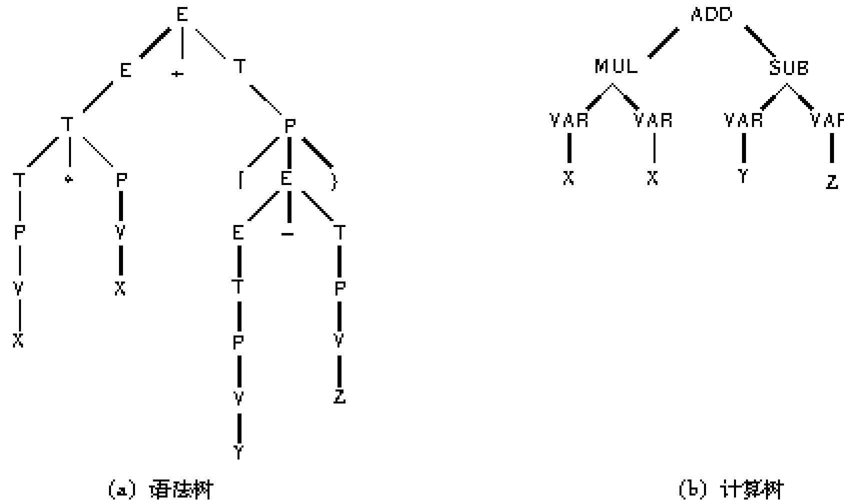


图 22　纵向分块

对于这样一种纵向分块，一般从事编译程序工作的人都是熟知的，无需多解释。不过，其中由综合块产生计算树一步，在有些系统中不太明显，它事实上是从语法树中删去中间过渡成分的一种精练化的形式，即 Lewis-Stearns[131]中所述“转换文

法”所表示的树形结构。例如表达式 X * X + (Y - Z) 的语法树与计算树即可分别表示为图 23 (a), (b):



(a) 语法树 (b) 计算树

图23

至于横向分块则是将图22 中“综合”一块再作进一步分划而得。由于一般高级语言中所包含的成分总可分解成以下四类:(1) 定义 (即说明), (2) 赋值, (3) 运算 (即表达式), (4) 顺序控制 (即语句)。而运算式又可分解成 (ⅰ) 运算对象与 (ⅱ) 运算符两个部分。因此,一源程序经过语法分解之后,其中所包含的信息,总可使之归属于这些类的某一类之中。由此,即可得横向分块的图示 (如图 24)。其中“分类”表示将语法分解树上的信息进行分类的算法,“组合”表示将各类信息再重新组合成计算树的算法。整个图 24 即可用来代替图 22 中“综合”的一框:

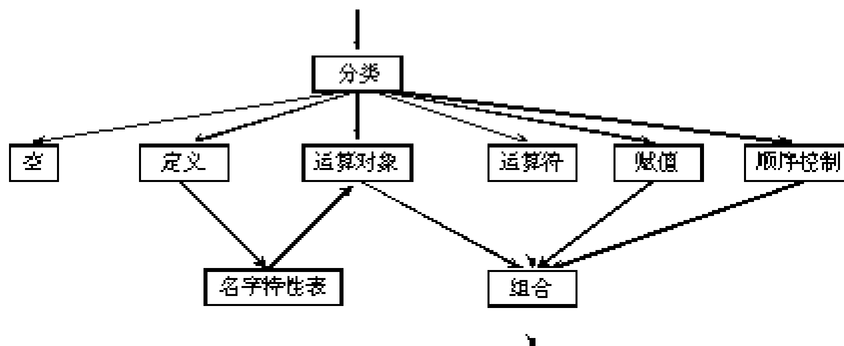


图24 综合部分的横向分块

McKeeman 认为在合理分块的基础上,应对各块之间的信息交换 (数据结构) 作精确地描述,而且不但应能像 PL/1 中“外部量”那样对此作静态地描述,还应

能描述出一块中某些数据结构经过加工后变成了什么形式的数据结构，也就是对动态行为进行描述。比如，在“扫描”之前的符号串，经过扫描之后，应变成由一对信息组成的编码，其中前一部分信息是指向该符号串存放于符号串表中的地址，后一部分信息则是为了出错时输出的信息（比如页号、行号），而符号串表中每一符号串信息又是由两部分构成的：一部分是符号串长度；另一部分是该符号串的编码。McKeeman 用一种类似于巴克斯范式的符号即可将符号串表（如图25）及扫描输出信息（如图26）描述出来。

```
string-table = string¯ ;
string = length character¯ ;
length = bit8;
bit = ‘0’ |‘1’;
character = a | b |··· | comma |···;
a = ‘11000001’;
b = ‘11000010’;
···
comma = ‘10101100’;
```

图25 符号串表

```
scan-output = string¯ ;
string = pointer-into-string-table error-into;
pointer-into-string-table = bit16;
error-into = page-number line-number;
page-number = bit16;
line-number = bit8;
bit = ‘0’ |‘1’;
```

图26 扫描输出信息（字符）

由计算树到中间语言程序之间事实上存在若干步转换。这些转换都是可以形式地予以精确描述的，而且这些转换都是单向的，也就是 McKeeman 所谓不反馈的。“不反馈是模块之间信息流的一类特别重要的特征，正是这样的信息流才使得多遍结构成为有用的结构。编译系统特别具有不反馈的内在结构，因为我们将编译过程看成是一些具有不反馈性质的形式转换序列。”这一转换序列的步骤可简述如下：

（1）由计算树（亦即抽象语法树，简写成 AST）再变成一标准形式（简写为 ST）然后转换成特征收集树（简写为 ACT），后者事实上即指向名字特性表的信息。特征收集树即可形式地描述成下面图27：

```
Program = ACT;
ACT = scope;
scope = symbol-table scope* command*;
symbol-table = (name attributes)*;
attributes = explicit-attributes | implicit-attributes;
```

图27　特征收集树

其含义即每一“scope”内除列出一组名字及相应的特性（包括用户自己标明的显特性以及由程序上下文中表明的隐特性）外，还可嵌套更内层的“scope”以及一串生成的“command”。

（2）由特征收集树再转换成特征分配树（简写为ADT），即将树中各叶端所对应的名字改成它所对应的特征（由名字特性表即可定出），例如将一名字为‘I’的VAR中名字‘I’换成其相应的类型信息（‘FIXED’）及分配的相对地址。即为：

图28　特征分配树

（3）将程序中表达式所对应的子树转换成目标程序指令串的形式（简写为SET）。

（4）将程序中语句部分转换成目标程序指令串的形式（简写为SCT）。

（5）最后得到目标程序。

要对这种不反馈的转换序列作完全精确的形式描述，看来还有待更进一步的研究[132]。McKeeman[23,48]还不能认为已完满地解决了这个问题。

上述McKeeman从模块化角度提出的结构化编译系统的概念，本文作者认为有以下几点可以注意：

（1）虽然图22，24中所示的纵向分块与横向分块的基本内容早已为许多编译系统所采用，他从结构化编译系统的角度对这种多遍扫描的编译系统的结构予以总结、分析和肯定仍然是很有意义的。他使人注意到这种结构在保证程序正确性方面的价值。至于模块的具体分法，显然图22与24的轮廓不能认为是唯一的，比如，一种一边扫描一边进行语法分解的编译系统仍不失为一结构化的编译系统。

（2）McKeeman比较强调对各模块间的信息流及转换进行形式描述，这对于精确刻画出各模块的特征及模块的变换流程是十分必要的。至于用什么形式语言进行描述较为合适则是一个尚待讨论的问题。事实上，功能较强的高级语言也都应该可以做到这一点。

（3）McKeeman［23］，［48］等文中没有讨论结构程序语言对结构化编译系统的作用这样一个重要的问题。他是以一般的高级语言为对象进行讨论的，事实上，如果源语言是一结构化的高级语言，则更有利于其编译系统的结构化。

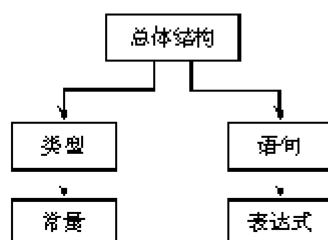


图 29

（i）为了使语法分解程序能分划成信息流不反馈的子模块，源程序的语言最好能具有如图 29 所示的整体结构：
此处，“总体结构”中包括“程序”、“过程”、“函数”这样一些基本模块的定义（说明）。图中“→”表示引用关系。请注意图中所示的引用关系是单向的，这就保证了其信息流不反馈。容易看出，像 ALGOL 60 那样的语言也不满足这样的结构，因在它的“语句”中包括“分程序”，而“分程序”中又可引用“过程说明”，而“过程说明”之中又包含“语句”。又如通常的语言中，“常量”允许在“语句”和“表达式”中出现，这样也就破坏了图 29 所示的树型结构，而在近年来出现的结构程序语言，则规定一般的常量（通常将“无符号整数”除外），均应在说明中定义一个名字。只有这些常量名字才允许在“语句”或“表达式”中出现，这样即可得到一如图 29 所示的整体结构。关于这方面的问题，详见［152］，［157］。

（ii）McKeeman［23］，［48］中没有详谈结果程序优化的问题，过去常用的线性下变优化的算法，往往使一个编译系统各部分算法互相关联，使编译系统结构复杂化。有些结构化程序语言的编译系统（如 PASCAL）即干脆不作任何与上下文相关的优化。但全局优化[133]方法则是可以在编译前对源程序独立地进行一遍扫描来完成的。它不影响编译系统内部的结构。看来，这样一种优化是符合结构化编译系统的要求的。更为重要的是，SIMPL 与 BLISS 的编译系统的经验[24,31]均已表明：由于结构程序语言中不出现 GOTO 或限制了 GOTO 的作用，它使全局优化的方法更便于实现。这一事实可以说是语言的结构化影响编译系统结构化的一有力的例证。

与此类似，可以一般地说：某些语言成分或功能，如果能像上述全局优化或宏加工那样在编译之前对源程序进行一遍加工来完成，这样的处理总是有利于编译系统结构化的。比如我们在语言族 XYZ 系统中即采用这样的方式处理“汉字”，看来，这有利于该语言族编译系统的结构化。详情见文［154］。

（4）下面两个问题，McKeeman［23］，［48］中已经涉及，我们认为在考虑编译系统模块化时是应该提到的：

（i）编译系统结构化有利于参加编译工作人员组织的结构化。像编译程序这样规模较大的程序系统，其参加工作人员组织的结构对程序正确性的影响是不容忽视的。当然，强调这一点不一定必须机械地采用“主程序员组”的方法，但人员组织总应与编译系统的结构之间有一合理的对应，而且适当加强系统主要负责人的责任和权限也是完全必要的。

（ii）采用图 22 及 24 这类多遍扫描的模块化方案势必使编译速度要放慢一些，

这对于较大的系统是值得付出的代价。但是，对于某些规模较小，又特别强调编译速度，而且，由于编制人员熟练，其正确性较能得到保证的系统，则不必机械地搬运这样一种做法。

以上介绍了侧重强调模块化的 McKeeman 方案。下面介绍侧重强调由顶向下方法的 Ammann[47]与 Basili-Turner[141]方案。

（二）逐步求精的方法

Ammann[47]的目标是“用逐步求精的方法为 Wirth 所定义的程序语言 PASCAL 发展一编译程序”。故所介绍的过程不免具有 PASCAl 语言的特征，但仍不失其一般性。而且，由于所讨论的编译系统是一遍扫描的，这对求精过程的安排也不无影响。Ammann 还指出：由于编译系统较大，将整个系统作为一个整体来逐步求精是不合适的。因此，必须将系统分割成独立的子块进行。但一遍扫描的要求又使它不能像图 22 那样按编译程序的逻辑顺序分成模块，所以，他只得将逐步求精过程与系统分块这两个方面结合起来，由此即得到图 30 所示的步骤。

由于 PASCAL 编译程序一般都是采用向前看一个符号的递归子程序方法进行语法分解，每一非终结符号对应一递归子程序。各递归子程序之间的调用关系即构成一图型（graph）结构。以当型语句为例，其语法公式是：

〈当型语句〉∷ = while〈表达式〉do〈语句〉

其相应的递归子程序为：

```
Procedure whilestat;
begin expression;
      insymbol;
      statement
end
```

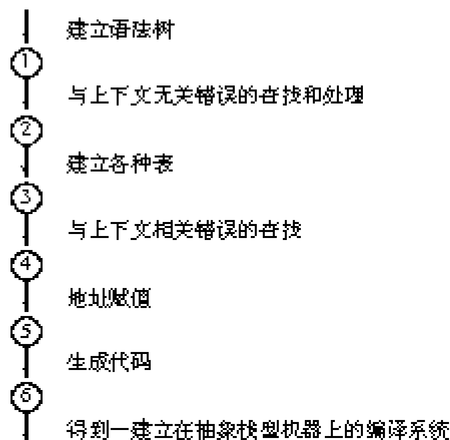


图 30

此处，expression，statement分别表示与〈表达式〉，〈语句〉相应的递归子程序，而insymbol即“读符号”子程序。建立语法树这一步除了写出对应于各非终结符号的递归子程序的基本轮廓外，还应写出insymbol这个读符号子程序。

所谓查找和处理与上下文无关的错误，就是查看insymbol所读到的字符是不是符合语法要求的合适字符。兹以setofsym表示所有合适字符的集合，以sy表示存放由insymbol读到的字符的变量，则经过这一步以后，与当型语句相应的递归子程序即精化为：

```
Procedure whilestat;
  begin expression;
        if sy = dosy then insymbol;
        else error;
        statement;
  end
```

此处，“dosy”表示“do”的内部编码，在error子程序中包括跳过源程序中不合适的字符，跳到可往下继续扫描的位置，并印出相应的出错信息等；在insymbol中包括对某些虽不完全合适但仍可断定其意义的字符如何处理和安排的等等，这一步即通常所谓“语法检查”。

再接下去便是建立各种表格，如名字特性表等，同时，写出“送表”，“查表”等专用过程。

然后，即是查找与上下文相关的错误的问题，比如查问变量的应用性出现与定义性出现是否一致；运算对象与运算符之间、运算对象之间类型是否一致等等。这些多属查找语义错误或语法限制错误的范围。在下面程序中，以comptype表示查类型是否一致的谓词，具体在whilestat中即用这过程查问前面执行expression的结果所得到的值是否为boolean型，经过这一步精化，所得过程whilestat即成为：

```
Procedure whilestat;
      begin expression;
            if ¬ comptype then error;
            if sy = dosy then insymbol;
            else error;
            statement;
      end
```

接下一步，对各变量赋予抽象机的栈内地址，这并非真机器地址，只是相对位置，一般由

(层次，层内位置)

这样一对整数偶来表示。此外，对于参量，过程调用时的保留信息，还应作适当安排，详见文［155］。

最后写出各子程序中代码生成部分，设CODE表示抽象机指令存储区，cix为其下标，geninstr为生成指令的过程，insert为反填转语句地址的过程，ic表示指令计数器。最后得到与当型语句相应的递归子程序即为：

```
Procedure whilestat;
var laddr: addrange; lcix: coderange;
begin laddr: =ic;
          expression;
          if comptype then geninstr ( 'falsejump', o)
          else error;
          lcid: =cix;
          if sy = dosy then insymbol
          else error;
          statement;
          geninstr ( 'uncondjump', laddr);
          insert (lcix, ic)
end
```

上面通过具体的例子说明了一个构造 PASCAL 编译系统的逐步求精的过程（有些参数未写出），Ammann 所分划的步骤在多数情况下对其它语言的编译系统也是适用的。当然，实际的情况可能会比较复杂一些，但以上的讨论仍不失其一般性。他的经验表明：逐步求精方法在很大程度上对构造编译系统也是适用的。

我们在本文§2（B）中已经指出："这样一种由顶向下的过程不能理解得太绝对化……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"。这一点，在 Ammann［47］中未予注意。如果机械地按这样一种由顶向下的方式去设计像编译程序这类规模较大的系统，很可能在进行到一定程度后会遇到几乎难以克服的困难。而 Basili-Turner［141］中提出的"反复扩展"的方法，正是在这一点上对逐步求精方法的改进。

（三）反复扩展的方法

文［141］指出："虽然一致同意用逐步求精方法是由顶向下模块化设计的基本准则，但是，当所要解决的课题具有相当大的规模时，这技术在实际上就不太容易应用，因为用合理模块化的由顶向下的途径建立一系统要求对问题以及它的解决办法有较好的理解……更有甚者，设计中的缺陷通常要等到实现已达到相当程度时才显露出来，到那时去纠正常常需要费极大的气力"。

"一种较实际的途径是先从所给问题的一个子集开始完成其简单初始的实现，然后经过反复修改和扩充一直到完全实现整个系统。在这过程中的每一步，不仅可扩充其原来的设计，而且对之进行修改也较易做到。事实上，随着反复过程的进行，对系统也逐渐加深理解，致使每一步能充分应用逐步求精方法使之更为有效，……因此，'反复扩展'代表了应用逐步求精的实用的途径"。

易见，"反复扩展"方法的关键在两个方面：（1）如何规定逐步扩展的内容？（2）如何选取所给问题的子集作为起点？

（1）如何规定逐步扩展的内容

扩充步骤由一个"课题控制表"所规定，其中应规定出整个课题所应完成的各

项任务以及各步所应实现的内容。“反复扩展”的过程每进行一步，则应从表中删去所已完成的任务，并选定下一步所应完成的任务，这里包括对所选定的任务的设计、编制和调试，而且应对已经实现的部分进行分析……这样逐步“反复扩展”直到“课题控制表”中所列各任务均已被删尽为止。表中每步所列的任务中，应包括“对已发现有缺陷的部分重新设计和编制，过去疏漏了的部分的设计和实现，以及某些未能解决的问题的解答”。由此可以看出：“课题控制表”的内容也是随着“反复扩展”过程的进行而逐步精细化的，它对较远的任务则只作轮廓性的规定，而对较近的任务则应规定得较明确具体。而且“作为分析的结果，课题控制表还得不断修改”。

（2）如何选取所给问题的子集作为起点

至于所给问题的子集，则应按如下原则进行选取：“一个概要性的子集应该包括了所给问题各个关键方面的适当的样板。它要简单、易于理解和实现，它实现后要是一个可为人所用的成品。其中不应包含特殊情形的分析，对它应加一些限制使它在不太妨碍使用的前提下，便于实现。它的实现在整体设计上应是简单而直接的，较低层的设计和编制应是直接的和模块化的，致使它在达到最终实现的反复过程中便于修改”。

将这些原则应用于构造语言 L 的编译系统时，“开始实现的概要性的编译系统应当由选取语言 L 的一个概要性子语言 L_0 所规定，而语言 L_0 则是 L 的一个稍加限制的子语言。它所对应的文法 G_0 则基本上是（L 所对应的）文法 G 的一个子文法”。故归根结底，即应选出一个语言 L 所对应的文法 G 的一个子文法 G_0，由它不但表示出所应开始编译的概要性子语言 L_0，而且由它亦决定了作为整个反复过程起点的概要性编译系统。

如所周知，Basili-Turner 应用此法构造了一语言族 SIMPL 及其编译系统[142, 136]。SIMPL 族的公共基础语言为 SIMPL-T，它本身具有书写编译系统的功能。为了应用“反复扩展”的方法构造一 SIMPL-T 的编译系统，先规划出 SIMPL-T 的子语言 SIMPL-X 作为概要性语言。它亦具有最低限度的自编译的功能。关于 SIMPL-T 与 SIMPL-X 的具体特征，我们已在本文§3（三）(C) 中列出，请参阅。

据设计人回顾，如果一开始不要求 SIMPL-X 具有自编译功能，则它仍可选取得更简单一些。他们还指出：虽然该编译系统的高层设计部分在整个实现过程中保持有效，但大部分低层设计与代码在“反复扩展”过程中则是需要改写的。

以上介绍了构造结构化编译系统的三类有代表性的工作。容易看出：这三项工作只是其强调的侧重面有所不同，它们之间并不是互不相容的，完全可以将其主要特征揉合在一个系统之中。即在构造一语言的编译系统时，一方面先选取一个概括性的子语言作为起点，按照一定的原则规划出其课题控制表进行“反复扩展”；另一方面，其整个编译系统按模块化的原则进行组织，也可分划出与图 22，24 相类似的模块。这样构造出的编译系统必将具有较好的结构。

此外，本文作者认为：用书写编译程序的专用语言（即 TWS）写编译程序有利于使编译系统结构化。这也就是我们在§2（B）中提到过的“编译程序的体系设计语言”。比如：用 Floyd-Evans 归约式写语法分解算法[137]，用类似于 JOSSLE 那样的

语言写语义子程序，即不但可使语法与语义两部分界限清晰，也可使编译时加工部分与目标程序部分界限清晰，便于阅读和理解。而且由于这类语言中常设有编译时专用的成分，如生成目标指令的“CODE”语句（或“geninstr”语句），为目标程序中设置标号、变量等的“ALLOCATE”运算等。这些成分的标出，突出了一编译程序的特征，可使读者概念明确。从编译系统结构化的角度看，这样一种途径是可取的，可与上述几种途径结合起来进行。

§5 结 束 语

在本文§2中我们已经指出：我们不打算在§2中对“结构程序设计”的定义多给予讨论，而只就这个概念的内容归纳出几个方面予以说明。因此前面各节多就正面进行介绍，未涉及反对的观点。为了不致使人产生片面肯定的理解，我们试图在结束本文之前介绍一些对结构程序设计持反对态度的意见，以及若干不同的解释和误解。

（一）对结构程序设计的解释

反对意见中最为极端的应是P. Abrahams的观点。他在文［161］中直言不讳地宣称“结构程序设计是有害的”，他认为：如何构造复杂的大型程序是一个老大难问题，不可能轻而易举地解决。过去曾多次有人宣传找到了一种解决办法，事实上都未成功。“现在，我们又听说可以从‘结构程序设计’的宗教中得到解救……不幸的是，这新的宗教不仅是无能的，而且是有害的，因为像其它宗教一样，它引诱其信奉者歪曲地看待现实，其结果不能有效地对待现实。”他将“结构程序设计”理解为以下的内容：

1. 用逐步求精的方法进行程序设计；
2. 程序的分层结构，或者用分层抽象作为设计的指导原则；
3. 程序设计中避免COTO；
4. 将一程序分成长度不超过一页纸的模块；
5. 避免全程量；
6. 程序正确性的证明；
7. 人员组织采用主程序员组的途径。

他认为，对这些概念的严格解释均蕴涵着一些假设，而这些假设中有些是错误的……主要的错误假设有以下一些：

1. 所有程序可以自然地表示成一树型结构；
2. 用其它的成分机械地替换COTO，总可以改进程序的可阅读性，而且所有COTO都是一样有害的；
3. 易于进行程序正确性的证明是对如何写程序起主要作用的条件；
4. 课题管理的主程序员组途径的成功足以证明对程序工作组织进行某些限制可以导致可靠的程序；

5. 程序运转的环境和它的有效时间对于程序应如何进行编制以及修改是无关的。

容易看出：如果将结构程序设计的概念作极端的理解，的确会要产生片面性，从而会要导致上述五个错误的假设。然而，如果理解得适当，这样的后果则是可以避免的。

（1）严格按逐步求精的方法由顶向下进行，每一程序应该由树型结构组成。而事实上，一程序的各个分支中往往可能有一些子模块是相同的，自应将之合并。这样，其结构即不是树型的而是上半格结构的。这说明，程序设计不能按照绝对意义下的由顶向下的方式进行，而应是由顶向下为主但又与由底向上相结合进行才较为合适，这一点我们在§2（B）中已说明白了。不过，这一事实并不意味着对逐步求精方法的全盘否定，这方法的提出，至少有两方面的积极意义：

（i）它强调了程序设计工作应是有条理地按一定步骤去进行，不能长期停留在一种手工艺工作的方式。

（ii）对于规模较大，其算法尚有待探索的课题，至少其设计过程应以由顶向下为主，较便于着手进行工作的探索。而且，正是因为设计中以由顶向下的方式为主，其由底向上的查错、修改和合并等工作较有脉络可寻，从而较易于进行。事实上，探索性算法往往以具有这样一种结构较为方便。前面所介绍的 Basili-Turner 提出的“反复扩展”方法就是以由顶向下为主以由底向上为辅的逐步求精方法。它即可避免绝对意义下由顶向下方法的片面性。

（2）理论的讨论给出了一些代替 GOTO 的机器方法[23, 27]，正如我们在§2 中所已指出的，“一般地讲，用这样的方法替换成的程序不但往往效率很低，而且更为严重的是很缺乏直观性，比原来包含 GOTO 语句时更难阅读，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。因此，不能将这些理论论证所产生的误解，归罪于结构程序设计本身。事实上，代替 GOTO 的并不是所述的机械替换方法，而是另一些较好的语句形式（如 Zahn 语句）。这一点 Abrahams[16]也是承认的。

（3）便于进行正确性证明的程序结构，往往是较为清晰、便于人们直觉验证的程序结构，这一点，Knuth［2］中已引用 Dijkstra 的意见加以说明。此处不必赘述。当然，程序正确性证明的方法也可以有许多种，不能将它与结构程序设计混为一谈。

（4）主程序员组的形式是否为最好的一种保证程序正确性的人员组织形式，的确是可以怀疑的，但无论如何，一种合理的人员组织形式，特别是适当强调程序工作的集体性，同时，在不妨碍所有参加者的积极性的条件下适当强调其主要负责人的职责和权限，看来对于保证程序（特别是大程序）正确性是十分必要的。结构程序设计促使我们注意人员组织形式对程序正确性的影响，这本身即是一件有积极意义的事。

（5）结构程序设计的方法应该说是针对规模较大、执行次数较多的程序提出来的。对于其它规模较小，并非多次执行的程序，自然不必一律按此办理。正像已经

有了大工厂生产用具的时候，并不排除某些用具仍可以手工艺方式生产一样。正确理解的结构程序设计方法应该不排除由于运转环境和有效时间的特殊条件而采用其他种方式进行程序设计的可能性。

所以，Abrahams 的批评应该说只是表明：对结构程序设计的概念和方法，不能作片面的、绝对化的理解。应该根据具体情况有分析地适当地理解这些概念和方法，否则，有可能产生有害的影响。

Denning［162］，［163］中也指出对结构程序设计的两种误解：

1. 所有程序都可以通过仅仅应用“顺序语句”，“条件语句”及“当型语句”等控制结构比较简单的程序组合而成。

2. 程序设计的方式是：“不断地对未实现的较低级的模块逐步求精，这样由顶向下进行设计，直到所有模块均完成为止”。

他接着指出：“认为由顶向下的程序设计即意味着严格地将程序设计过程解释成一序列逐步求精的过程。这是一种误解。因为，这种信念将成果的结构与得到这成果的过程的结构混淆起来了；认为许多结构定理证明了较简单的形成规则即足以产生较好的程序。这也是一种误解。因为这种信念将程序转换的机械过程与产生这些基本结构的思维模式混为一谈了。这两种误解引起更为基本的误解，即认为程序设计中存在某种固有的简单容易的因素，而且认为其中较困难的部分（模块化及形成模式）可以归约成为一机械的过程。这也是荒谬的。程序设计是一困难的思维过程”。

最后，在［162］中他表示：“我并不低估使程序具有较好风格的重要性……我只是坚持，你必须明了，不能固执地坚持程序设计必须遵守一组固定的规则，这有点像坚持要求学生按一种固定的风格作文，如果照此办理，这样的意见将把人们引入一可怕的笨拙的世界”。

Denning 的上述意见有其正确的部分，的确，程序的风格不能由一组固定的规则来机械地决定，编程序的过程中不能排除人们创造性的思维活动。但是，如果因此认为由顶向下的设计不应作为一种程序设计的过程看待，则是对这方法的一种曲解。由顶向下的设计是作为一种设计方法提出来的。它的确包含如下的含义：即为了保证所得到的程序具有一合理的结构，其设计的过程亦应具有某种合理的结构。这样一种设计思想所产生的问题并不在于将这两方面混淆起来，而是有人对这方法理解得过于片面，这一点，上面已经论述过了。

Gries[180]曾在 Denning 意见的基础上，归纳出对结构程序设计（他以 SP 表示）的十三种不同的解释：

1. 这是返回到常识。

2. 这是我们的主要程序员编制程序的一般方法。

3. 这是避免用 GOTO 的一种程序设计。

4. 这是一种控制局部任务与其环境交互作用的次数的过程，它使得交互作用的次数是任务的某些或所有参数的某种线性函数。

5. 这是由顶向下的程序设计。

6. SP 理论讨论了如何将一任意巨大的和复杂的流图转换成一种标准形式，以

致它们能够用某几种标准的基本的控制逻辑结构（通常是顺序、分支和重复），经过重复与嵌套来表示。

7. SP是一种组织和编制程序的态度，它使得程序易于理解和修改[164]。

8. SP的目的是通过理论与规范去控制复杂性[143]。

9. SP的特征不是删去GOTO而是引进结构[143]。

10. 程序结构化的一个主要功能是使得正确性的证明容易实现[21]。

11. SP的基本概念是正确性的证明[163]。

12. SP……允许在设计过程中各步内去验证其正确性，这样即自动地导致自我说明与自我捍卫的程序风格[31]。

13. SP不是万应灵丹——它确实包含一组按条理思维的形式符号，这不是一种为程序员通常固有的特性，而是一种规范，必须获得它，而且还应通过不断的有意识的努力加强它。这方面费的力气是值得的[166]。

文［148］中Basili-Turner还补充了如下一条：

14. 人们应追求程序的简单和精美，最简单的解答通常总是最易理解的。

Gries总结以上对SP的各种不同的解释，最后指出："概括起来，它们给出了这个问题的一个较好的全面的观点，我认为Hoare的定义抓住了SP的本质，即：将人们的思维按这样一种方式组织起来，使之在合理的时间内，将一计算任务表示成容易理解的形式，这样一种工作就是'结构程序设计'。所以，结构程序设计的最基本的任务，是试图讨论如何组织人们的思维，使得所编出的程序具有一合理结构。我们这样去理解结构程序设计，较不易产生片面性。

（二）结构程序设计的发展动向

结构程序设计提出已近10年了，近来主要动向怎样？

（1）这概念首先是在欧洲提出的，欧洲学术界较强调分析程序设计的基本原理[31,136]及逐步求精的设计方法[9,10]。这种讨论虽已多年，但在实际的程序实践中应用面似不很广。到现在止似未见显著的成效。结构程序设计在美国则是直到1973年才突然被IBM公司所重视，但他们所重视的主要在人员组织形式方面。有人认为"结构程序设计的问题主要……是一个管理问题"[168]。IBM公司等单位已按主程序员组的形式将软件工作组织起来。据说，还是有成效的，能较大地发挥熟练程序员的作用。在美国，结合人员组织形式，对由顶向下方法如何具体实用也有一些讨论[194,193]，程序风格问题颇受注意[197,198]。

（2）在程序语言结构化方面目前有两个很明显的对立倾向。一种是主张按照结构程序设计的思想提出新的语言成分，构造新的语言，甚至认为现在"我们正处在即将发现程序语言究竟应该是什么样子的时刻"[21]。这种观点的代表如Knuth，Dijkstra，Hoare，Wirth，Wulf等，这方面的情况，我们在本文§3中已介绍过了。我们现在应介绍一下与此对立的另一种观点，一般地讲，具有这种观点的人也承认旧的语言如Fortran等是有缺点的。新提出的许多关于程序语言的概念是有意义的。但他们更为强调的一点是：旧语言已应用十多年了，人们已习惯于使用它们。所以，与其另起炉灶设计新的语言，不如改造或扩充旧的语言使之具有结构化的特征更为妥

当。这方面最有代表性的意见为 Ralston[169]，Naur[100]。Ralston 认为：“在高级语言，至少……在通用高级语言中不会有什么革命，换言之，通用高级语言的应用领域的状态是非常稳定的，具有极大的惰性，不管是好是坏，在硬件与软件的当前水平线上还看不到或很难看到有任何东西将很大地或很快地改变这种状态”。他也承认：“并不相信 Fortran 是完美的”，但他却“相信 Fortran 是合适的”。因此，他主张“不要去打倒 Fortran 而应支持它、改进它”。“Fortran 应被看成是一以标准 FortranⅣ为核心的可扩充语言……其中包含一些槽粕，可以割弃……它们将被忘掉”。Naur[100]的观点与此相似，他以当年世界语的发展情况为例，主张不要轻易抛弃人们已经习惯了的旧语言，而应该支持它，改造它，吸收新的成分，克服掉一些原有的缺点。在这种思想指导下，近年来在美国关于“结构化的 Fortran”[170-179]，“结构化的 COBOL”[180-182]的讨论很多。这当然也是一新的值得注意的动向。不过，另外还请注意的是，这种情况在欧洲、日本等处并不显著。这也就说明只是由于在美国这些语言流行太久太广，人们太习惯了，因而才会出现这样的现象。反过来注意一下我国的现状，我们应该怎样对待这个问题，实在值得考虑。

附带还应提及的是：除了美国与欧洲以外，日本学术界对结构程序设计也日益注意。除了有人对结构程序设计作过极为详尽的综述和介绍以外[188-190]，还有人在构造他们自己的分层结构化语言[191, 192]，新近还召开了内容丰富的专门会议[193]。

总括以上的讨论，本文作者认为：结构程序设计所提出的问题是一个十分重要的问题。为了保证程序正确性，使之便于阅读和修改，我们必须注意程序设计方法、程序的宏观结构及微观结构、人员组织形式、并使与此相应的合理结构在程序语言中得到反映。至于现在已有的为了解决这些问题所提出的各种技术方案，则关键的问题不在某一方案的具体内容是否完全正确，而在于我们以什么态度对待这些方案。如果我们由于重视结构程序设计所提出的问题和希望达到的目标从而对某个具体人提出的某一具体技术方案也一概不加分析地全盘接受并教条式地执行，那是十分危险的，有可能陷入片面性的困境；反过来，如果由于有可能产生这样的后果而对结构程序设计本身一概予以否定，那同样也是片面的。应该看到：结构程序设计所要达到的目标是不可能由一二人提出的一二个方案而一举完成的。这是一个运动，是一个过程。已经提出的各种技术方案只是这一过程中先走出的几步。其中包含正确的意见，也必不可免地包含一些不够全面甚至错误的成分。因此，应该认真分析，分清主次，取其精华，去其槽粕。现在，应该肯定的是这一运动的目标和方向，而不必忙于对某种具体技术方案作最后的结论。至于结构程序语言的形式和成分，我们在§3 中已有论述，总的讲，本文作者较倾向于构造图 20，21 所示的分层语言族，其高层具有非常高级的体系设计语言的特征，其底层又是一分层的公共基础语言。其中既包含 PASCAL 中可取的高级语言成分，又包含一些面向一般计算机概念的数据结构和运算（如按位运算或间接量运算等）。在宏观结构方面，我们主张分“系统”、“程序”与“子程序”三级结构，而系统结构则是线性有向网型的，可用与 JOSSLE 中联系区相似的方式在语言一级予以规定；SIMULA 67 中“类组”的模块结构是很值得参考的，但为了使语言中概念简化，在我国目前条件下，本文作者认为可在流行的过程结构之上，吸取类组中某些重要的功能。比如：在类型定义之

上增加限制部分，规定允许在此种类型上施加的运算（过程）的名字；又如：可仿照 PASCAL 中“New”过程实现其动态存储分配的功能。在控制结构方面，我们认为：Zahn 事件语句，KAIL 选择句是可取的，这两种语句再加上 Dijkstra 的不确定语句，都是分情形语句的一种推广的形式，可以用一种共同的复合语句的形式加以概括。有了前两种语句形式即可避免 COTO 与 IF-THEN-ELSE 语句的某些毛病。不过，为慎重计，COTO 语句仍可保留，一般情形下最好留而不用，或者限制它只许往后转。本文作者还主张：最好将逻辑特征相应的控制结构与数据结构以相近似的记号来表示（如表 1）。在结构化的编译系统方面，本文作者认为可将“模块化”的途径与“反复扩展”的途径结合起来，而且认为：采用描述语义和语法的专用体系设计语言来书写翻译程序有助于编译系统的结构化。这些就是本文作者分析了近年来关于结构程序语言及结构化编译系统的讨论之后得出的初步结论[200]。

最后，我想引用 Dijkstra 最近一次谈话[199]中的一段话结束本文：“这个词（即结构程序设计）已经谈得太多了，我再也不用它了”。

参考资料

[1] Neely, P. M. “After Fortran, What?”, Cuest Editorial, Software Practice and Experience vol 5 No 1 1975.

[2] Knuth, D. E. “Structured Programming with COTO Statement”. PB 233507 或 ACM Comp. Surveys vol6. No. 4 1974.

[3] Naur, P. and Randell, B. ed. Software Engineering Report on a Conference, Carmisch Oct. 1968.

[4] Dijkstra, E. W. “COTO Statement Considered Harmful”, CACM vol 10 1967.

[5] Bauer, F. L. “Software Engineering”, Proc IFIP Congress, 1971.

[6] Datamation vol 19 No. 12 Dec 1973.

[7] Baker, F. T. “Organizing for Structured Programming”, Lect Notes in Comp. Sci. vol 23 (ed. Coos and Hartmanis) 1974.

[8] Wirth. N. “From Programming Techniques to Programming Method”, Intl. Computing symp. 1973. (ed. Cunther et al) 1974.

[9] — “Program Development by Stepwise Refinement”, CACM vol 14, 1971.

[10] — “Systematic Programming” Prentice Hall Inc. 1973.

[11] 在授与 Dijkstra. E. W. Turing 奖金时的介绍，CACM vol. 15. 1972.

[12] Kosaraju, S. R. “Analysis of Structured Programs”, J. Comp. and Syst. Sci. vol 9, No 3, 1973.

[13] Bersen, J. P. “Structured Programming Techniques”, IEEE 1973 Computer Software Reliability, 1973.

[14] Manna, Z. and Waldinger, R. “Toward Automatic Program Synthesis”, CACM vol. 14, No 3, 1971.

[15] Denning, P. J. “Is It Not Time to Define Structured Programming”, SICOPS Review vol. 8, No 1, 1974.

[16] Foulk, C. R. “Yet Another Attempt to Define Structured Programming”, SICOPS Review vol. 8, No 3, 1974.

[17] Lucena, C. J. and Berry, D. M. "Toward Definition of Structured Programming", Proc. ACM San Diego, 1974.

[18] Böhm, C. and Jacopini, C. "Flow Diagram, Turing Machine and Languages with Two Formation Rules", CACM vol. 9, 1966.

[19] 唐稚松“论指令系统的递归性”，数学学报，15卷，6期，1965.

[20] Haney, F. M. "The Architecture of Software", Software World vol. 5, No 6, 1974.

[21] Dijkstra, E. W. "Notes on Structured Programming", Structured Programming (by Dahl, et. al.) Academic Press 1972

[22] Wulf, et. al. "Programming without the Goto", Proc. IFIP Congress 1971.

[23] Mckeeman, W. M. "Compiler Structure", First Japan-USA Computer Science Congress 1972.

[24] Zelkowitz, M. V. and Bail, W. C. "Optimization of Structured Program", Software Practice and Experience vol. 4, No 1, 1974.

[25] Ashcroft, E and Manna, Z. "The Translation of GOTO Statement to While Statement", Proc. IFIP Congress 1971.

[26] Knuth, D. E. and Floyd, R. W. "Notes on Avoiding GOTO Statements", Information processing. Letters 1 (Feb. 1971) 23-31, 177.

[27] Knuth, D. E. "A Review of Structured Programming", Comp. sci. Dep. Stanford univ. Stan-cs-73-371 1973.

[28] Wirth, N. and Hoare, C. A. R. "Contribution to The Development of ALGOL", CACM vol. 9, 1966.

[29] Wirth, N. "The Programming Language PASCAL", Acta Informatica 1, 1971.

[30] Wulf, et. al. "BLISS Reference Manual", Tech rep. comp. Sci. Dept. Carnegie-Mellon Univ. Pittsburgh Penn. 1971.

[31] Wulf, et. al. "A Case against GOTO", proc. ACM 1972.

[32] 唐稚松“编译自动化研究现状”，计算机学术报告会资料汇编第二册 1972.

[33] Richards, M. "A Tool for Compiler Writing and System Programming", AFIPS Proc. vol. 34, 1969.

[34] Evans, R. V. "Multiple Exits from a Loop Using Neither Goto Nor Labels", CACM vol. 17, No 11, 1974.

[35] Zahn, C. T. Jr. "A Control Statement for Natural Top Down Structured Program", Lect. Notes on Comp. Sci. vol. 19, (ed Coos and Hartmania)

[36] Bochmann, C. V. "Multiple Exits from a Loop without GOTO", CACM vol. 16, 1973.

[37] Barth, C. W. "Notes on The Case Statement", Software Practice and Experience vol. 4. No 3, 1974.

[38] Koster, C. H. A. "Portable Compiler and Uncol Problem", Machine Oriented High Level Languages (ed. V. D. Poel-Maarssen) 1974.

[39] White, J. R. and Presser, L. "A Structured Language for Translator Construction", Comp. J. vol. 18, No 1, 1975.

[40] Clint, M. and Hoare, C. A. R. "Program Proving, Jumps and Functions", Acta. Informatica 1 1972.

[41] Cheatham, T. E. Jr. and Townley, J. A. "A Proposed System for Structured Programming", Lect. Notes on Comp. Sci. vol. 19, (ed. Coos and Hartmania).

[42] Cheatham, T. E. Jr. and Wegbreit, B. "A Laboratory for the Study of Automating Programming",

11 -71 Center for Res. Comp. Tech, Harvard Univ. 1971.

[43] Chu, Yaohan: "CDL-A very High Level Language", Tech. Rep. TR-317 NSF CJ-33013 Dept Comp. Sci. Univ. of Maryland. 1974.

[44] Discussion on "An Alternative to the Higher Level Language", Machine Oriented High Level Language (ed. V. D. Poel-Maarssen) 1974 North Holland pub Co. 1974.

[45] Nassi, I. and Shneiderman, S. "Flowchart Technique for Structured Programming", SIGPLAN Notices vol. 8, No 8, 1973.

[46] Chapin, N. "New Format for Flowchart", Software Practice and Experience vol. 4, 1974.

[47] Ammann, U. "The Method of Structured Programming Applied to the Development of a Compiler", Intl. Computing Symp. (ed. Gunther et. al.) 1974.

[48] McKeeman, W. M. "Programming Language Design", An Advanced Course on Compiler Construction, Lect. Notes Comp. Sci. vol. 21, 1974.

[49] Dijkstra, E. W. "The Structure of the "THE" Multi-Programming System", CACM, vol. 11, 1968.

[50] Wegner, E. "Tree Structure Programs", CACM, vol. 16, 1973.

[51] Wells, M. B. "Algorithmic Languages and Machine Oriented Task", Machine Oriented High Level Languages (ed. V. D. Poel-Maarssen) North Holland pub. co. 1974.

[52] Wilkes, M. V. "The Outer and Inner Syntax of a Programming Language", Comp. J., vol. 11, No 3, 1968.

[53] Liskov, B. and Zilles, S. "Programming with Abstract Data Types", Proc. very High Level Languages SIGPLAN Notice vol. 9, No4, 1974.

[54] Geiselbrechtinger, F. et al. "Language Layers, portability and program structuring", Machine Oriented High Level Languages (ed. V. D. poel-Maarssen) 1974. North Holland pub. Co. 1974.

[55] Scheidig, H. "A Brief Survey of Lo", 同上书.

[56] Coos, G. "Hierarchies", Adranced Course on Software Engineering (ed. Bauer), Lect. Notes in Econ. and Math. syst, vol. 81, 1973.

[57] Dennis, J. B. "Modularity" 同上书.

[58] Wulf, W. and Shaw, M. "Global Variable Considered Harmful", SIGPLAN Notices vol. 8, No 2, 1973.

[59] Hull, T. E. "Would you believe Structured Fortran", SIGNUM Newsletter vol. 8, No 4, 1973.

[60] Dahl, O—J. and Hoare, C. A. R. "Hierarchical Program Structure", Structured programming (by Dijkstra, et al.) 1972.

[61] Ichbiah, J. D. and Morse, S. P. "General Concepts of the SIMULA 67 Programming Language" An. Rev. in Auto. prog. vol. 7, No 1, 1972.

[62] Baker. F. T. "Chief programmer Team Management of Production Programming", IBM syst g. vol. 11, No 1, 1972.

[63] Peter, L. "Managing the Transition to Structured Programming," Datamation May, 1975.

[64] Hiemann, P. "A New Look at the Program Development Process", Lect. Notes in Comp. sci. (ed. Coos and Hartmanis) vol. 23, 1974.

[65] Weinberg, G. "The Psychology of Computer Programming", Van Nostrand Reinhold New york 1971.

[66] Presser, L. "Structured Languages", AFIPS Proc. vol. 44, 1975.

[67] Presser, L. and White, J. "Making Global Variable Beneficial", AFIPS Proc. vol. 43. 1974.

[68] White, J. and Presser, L. "A Tool for Enforcing System Structure", Proc ACM Natl Conf. 1973.

[69] White, J and Presser, L. "A Structured Language and Translator Construction", Comp. J. vol. 18, No 1, 1978.

[70] Cannon, J. D. and Horning, J. J. "The Impact of Language Design on the Production of Reliable Software", Intl. Conf. on Reliable software Proc. SIGPLAN NOTICES vol. 10, No 6, 1975.

[71] Conway, M. E. "How do Committees Invent?", Datamation vol 14, No 4, 1968.

[72] Liskov, B. H "A Design Methodology for Reliable Software Systems", FJCC. Proc. pt I. 1972.

[73] Dijkstra, E. W. "Hierarchical Ordering of Sequential Process", Acta Inf. vol. 1, No 2, 1971.

[74] Hoare. C. A. R. "The Monitor: An Operating System Structuring Concepts", CACM, vol. 17, 1974.

[75] Hansen, P. B. "Operating System Principle", Prentice-Hall July, 1974.

[76] — "The Nucleus of a Multiprogramming System", CACM, vol. 13, 1970.

[77] — "Structured Multiprogramming", CACM vol. 15, 1972.

[78] — "Concurrent Programming Concepts", ACM Comp. Rev. vol. 5, 1974.

[79] — "A Programming Methodology for Operating System Design", IFIP Congress Proc. 1974.

[80] — "The programming Language Concurrent Pascal", IEEE Trans. on Software Engineering vol SE -1 No 2, 1975.

[81] Bauer, F. L. "A Course of Three Lectures on a Philosophy of Programming", Languages Hierarachies and Interfaces (Lect. Notes Comp. Sci. 46) 1977.

[82] Lomet, D. B. "Control Structures and the Return Statement" IFIP Congress Proc. 1974.

[83] Weinberg, G. M. et. al. "IF-THEN-ELSE Considered Harmful." SIGPLAN NOTICES vol 10 No 8. 1975 Aug.

[84] Halass, A. "Event-Driven Control Statements", BIT Bd5. 1975.

[85] Embley. D. W. and Hansen W. I. "The KAIL Selector-An Unified Control Construct", SIGPLAN. NOTICES vol. 11, No 1, 1976.

[86] Perlis, A. J. "会话语言与可扩充语言", 学术报告及座谈记录 1972 年 8 月, 北京。

[87] — "Synthesis of Algorithmic Systems", J. ACM. vol. 14, No 1, 1967.

[88] Cheatham, gr. T. E. "The Introduction of Definitional Facilities into Higher Level Programming Languages", AFIPS Proc. vol. 29. 1966.

[89] Schuman, S. A. and Jorrand, p "Definition Mechanisms in Extensible Programming Languases", AFIPS Proc. vol. 37, 1970.

[90] Standish, T. A. "Extensibility in Programming Languages Design", AFIPS, Proc. vol. 44, 1975.

[91] Wilkes, M. V. "Computers Then and Now", J. ACM, vol. 15, No 1, 1968.

[92] Wegner, P. "Data Structure Models for Programming Languages", SIGPLAN NOTICES vol. 6, No 2, 1972.

[93] Mealy, C. H. "Another Look at Data", AFIPS, Proc. vol. 31, 1967.

[94] — "Data structure: Theory and Representation", IFIP Congress Proc. 1974.

[95] Hoare, C. A. R. "Notes on Data Structuring", Structured Programming (ed. by Dahl et. al.) Academic Press, 1972.

[96] — "Data Reliability", Intl. Conf. on Reliable Software Proc. 1975.

[97] Dijkstra, E. W. "Guarded Commands, Nondeterminacy and Formal Derivation of Pragrams", 同上书.

[98] Jensen, k. and Wirth, N. "PASCAL User Manual and Report", Lect Notes in Comp. sci. vol. 18,

(ed. by Coos and Hartmanis) .

[99] Wirth, N. "An Assessment of The Programming Language PASCAL", Intl. Conf. on Reliable Software Proc. 1975.

[100] — "On The Design of Programming Languages", IFIP Congress Proc. 1974.

[101] Habermann, A. N. "Critical Comments on The Programming Language PASCAL", Acta, Inf. vol. 3, 1973.

[102] Lecarme, O. and Desjardins, P. "Reply to a Paper by A. N. Habermann On The Programming Language PASCAL", SIGPLAN NOTICES vol. 9, 1974.

[103] Naur, P. "Programming Languages, Natural Languages, Mathematics", CACM vol. 18, No 2, 1975.

[104] Clark, B. L. and Horning, J. J. "Reflections on a Language Design to Write Operating Systems", SIGPLAN NOTICES, vol. 8, No 9, 1973.

[105] Morris, Jr, J. H. "Types Are Not Sets", Conf. record of ACM Symp. on Principles of Programming Languages 1973.

[106] Maclennan, B. J. "A Note on Dynamic Arrays in PASCAL", SIGPLAN NOTICES vol. 10 No 9, 1975.

[107] Wirth, N. "Comments on a Note on Dynamic Arrays in PASCAL", SIGPLAN NOTICES vol 11, No 1, 1976.

[108] Schwartz, J. T. "On Programming: An Interim Report on The SETL Project", New York, U. 1973.

[109] Leavenworth, B. M. and Sammet, J. E. "An Overview of Nonprocedural Languages", SIGPLAN NOTICES vol. 9, No 4, 1974.

[110] Earley, J. "High Level Operations in Automatic Programming", 同上书.

[111] McKeeman, W. M. "On Preventing Programming Langnages from Interferring with Programs", IEEE Trans. on Software Engineering vol SE-1 No1, 1974.

[112] Balzer, R. M. "Dataless Programming", AFIPS proc. vol. 31. 1967.

[113] Earley, J. "Toward an Understanding of Data Structure", CACM vol. 14, No10, 1971.

[114] Geschke, C. M. and Mitchell, J. C. "On The Problem of Uniform References to Data Stru cture" IFIP Congress Proc. 1974.

[115] Liskov, B. H. "Data Types and Program Correctness", AFIPS proc. vol. 44, 1975.

[116] Liskov, B. H. and Zilles, S. "Specification Techniques for Data Abstraction", Intl Conf. on Reliable Software proc. 1975.

[117] Ross, D. T. "Uniform Referents: An Essential Property for a Software Engineering Language", Software Engineering (ed. by Tou) vol. 1, 1970.

[118] Zahn, C. T. "Structured Control in Programming Languages", AFIPS Proc. vol. 44, 1975.

[119] Wasserman, A. I. "Issue in Programming Language Design", 同上书.

[120] Wegner, E. "Control Structures for Programming Languages", SIGPLAN NOTICES vol. 10, No2, 1975.

[121] Ledgard, H. F. and Marcotty, M. "A Genealogy for Control Structures", CACM. vol. 18, No 11, 1975.

[122] DeRemer, F. and Kron, H. "Programming-in-the-large and Programming-in-the-small", Intl. Conf. on Reliable Software Proc. 1975.

[123] Neely, P. M. "On Program Control Structure", Proc. ACM. 1973.

|124| —"A New Programming Discipline", Software Practice and Experience vol. 6, No1, 1976.

|125| Sammet, J. E. "Brief Survey of Languages used in System Implementaion", SIGPLAN NOTI CES vol. 6, No. 9, 1971.

|126| Lyle, O. M. "A Hierarchy of High order Languages for Systems Programming", 同上书.

|127| Hoare, C. A. R. "High Level Programming Languages-The Way Behind", Proc of British Comp. sci. Conf. on High Level Programming Language-the Way Ahead, 1973.

|128| Pyle, J C. "What the System Programmer Wants" 同上书.

|129| Sibley, E. H. "Objective and Requirements for a Future General Purpose Language", 同上书.

|130| Voda, P. J. "General Purpose Languages Revisited", Machine Oriented High Level, Language (by Poel-Manssen) 1974.

|131| Brittenham, W. R. and Melkum, B. F. "The System Program Language Problem", 同上书.

|132| Discussion on "Structured Programming", 同上书.

|133| Discussion on "What is special about Mols", 同上书.

|134| Ichbiah, J. C. et. al. "The Two Level Approach to Data Independent Programming in The LIS System Implementation Language", 同上书.

|135| Graham, R. M. "Report of Session on System Programming Languages", SIGPLAN NOTICE vol. 8, No 9, 1973.

|136| Wulf, W. et. al. "Reflections on a System Programming Language", SIGPLAN NOTICE vol. 6, No 9, 1971.

|137| Clark, B. L. and Horning, J. J. "The System Language for Project SUE", 同上书.

|138| Bergeron, R. D. et. al. "Languages for System Development", 同上书.

|139| —"System Programming Languages", Advances in Computers vol. 11, 1971.

|140| Elspas, B. "An Assessment of Technique for Proving Program Correctness", ACM Comp. survey, vol. 4, No 2, 1972.

|141| Basili, V. R. and Turner, A. J "Iterative Enhancement: A Practical Techniqne for Software Engineering", Proc. lst Natl conf. on software Engineering" 1975.

|142| —"A Transportable Extendable Compiler" Software-Practice and Experience vol. 5, No 2, 1975.

|143| Rheinboldt, W. C. "On a Programming Language for Graph Algorithms", BIT BD 12, 1972.

|144| Chu, Y. "A Methodology. for Software Engineering" IEEE Trans. on Software Engineering vols E -1, No 3, 1975.

|145| Mills, H. D. "Mathematical Foundations for Structured Programming", IBM Tech. Rept. 1972.

|146| —"The New Math of Computer Programming", CACM vol. 18, No 1, 1975.

|147| —"How to Write Correct Programs And Know It", Intl. Conf. on Reliable Software proc. 1975.

|148| Basili, V. R. and Baker. T. "Structured Programming: A Tutorial", Compcon 1975.

|149| Hoare, C. A. R. "Structure of Operating Susten" Language Hierarchies and Interfaces (Lect. Notes Comp. Sci. 46) 1977.

|150| 仲萃豪，张尤腊，面谈 1977.

|151| Lewis, P. M, and Stearns, R. E. "Syntax Directed Transductions", J. ACM. vol. 15, 1968.

|152| 唐稚松"文法的分划问题"，中国科学院计算所资料，北京 1976.

|153| Allen, F. E. "Program Optimization", ARAP vol. 5, 1969.

|154| 唐稚松"高级语言中的汉字问题"，中国科学院计算所资料，北京 1977.

|155| PASKO. H. J. "A Pseudo-Machine For code Generation", Tech. Rep. CSRG-30 1975.

[156] Basili, V. R. "The Design and Implementation of a Family of Application Oriented Languages" Proc. of 5th Texas Conf. on Comp. Syst. 1976.
[157] 唐稚松 "LR (K) 语法分解及 FPL 程序的优化", 将在数学学报 1978 年 1 期刊出.
[158] 周龙骧 "软件移植问题", 中国科学院数学所资料 1976.
[159] Beckman, A. "Comments Considered Harmful", SIGPLAN NOTICES vol 12, No 4, 1977.
[160] Gries, D. "On Structured Programming-A Reply to Smoliar", CACM vol. 17, No 11, 1974.
[161] Abrahams, P. "Structured Programming Considered Harmful", SIGPLAN NOTICES vol. 10, No 4, 1975.
[162] Denning, P. J. "On Being One's Own Programming Self", AFIPS Proc. vol. 44, 1975.
[163] — "Two Misconceptions About Structured Programming", ACM Proc. 1975.
[164] Donaldson, J. "Structured Programming", Datamation, Dec. 1973,
[165] Karp, R. Letter Datamation March 1974.
[166] Butterworth, D. Letter Datamation March 1976.
[167] Tanenbaum, A. S. "In Defense of Program Testing or correctness proofs Considered Harmful", SIGPLAN NOTICES vol. 11, No 5, 1976.
[168] Branscomb, L. 一次谈话 1977.
[169] Ralston, A. "The Future of Higher Level Languages", Intl. Comp. Symp. (ed. A. Günther, et al.) 1974.
[170] Tenny, T. "Structured Programming in Fortran", Datamation vol. 20, No 7, 1974.
[171] Meissner, L. P. "On Extending Fortran Control Structures to Facilitate Structured Programming", SIGPLAN NOTICES vol 10 No9 1975.
[172] — "Proposed Control Structures for Extended Fortran", SIGPLAN NOTICES vol. 11, No 1, 1976.
[173] Gales, L. E. "Structured Fortran With no Preprocessor", SIGPLAN NOTICES vol. 10, No 10, 1975.
[174] Reifer, D. J. "The Structured Fortran Dilemma", SIGPLAN NOTICES vol. 11, No 2, 1976.
[175] Cook, A. J. "Experience With Extendable Portable Fortran", SIGPLAN NOTICES vol. 11, No 9, 1976.
[176] Rowland, S. W. "Some Comments on Structured Fortran", SIGPLAN NOTICES vol. 11, No 10, 1976.
[177] Salomon, D. "A Design for Fortran to Facilitate Structured Programming", SIGPLAN NOTICES vol. 12, No 1, 1977.
[178] Boddy, D. E. "Structured Fortran-With or Without a Preprocessor", SIGPLAN NOTICES vol. 12, No 4, 1977.
[179] Ralston, A. and Wagener, J. L. "Structured Fortran-an Evolution of Standard Fortran", IEEE Trans. on Soft. Eng. vol SE-2 No 3, 1976.
[180] McClure, C. L. "Structured Programming in COBOL", SIGPLAN NOTICES vol. 10, No 4, 1975.
[181] — "Ordering Control Data in Structured COBOL", SIGPLAN NOTICES vol. 11, No 8, 1976.
[182] Feurman, M. et al. "Structured Programming in COBOL", SIGPLAN NOTICES vol. 11, No 8, 1976.
[183] Elgot C. C. "Structured Programming with or without Goto Statements", IEEE Trans. on Soft. Eng. vol SE-2, No 1, 1976. Erratum and Corrigendum, vol SE-2 No 3, 1976.

[184] Wtlkes, M. V. "Software Engineering and Structured Programming", IEEE, Trans. on Soft. Eng. vol SE-2 No 4, 1976.
[185] Mill, H. D. "Software Development", IEEE Trans. on Soft. Eng. vol. SE-2 No 4, 1976.
[186] Bauer, F. L. "Programming as on Evolutionary Process", 2nd Intl. Conf. on Soft. Eng. 1976.
[187] Feldman, M. B. "New Languages from old: The Extension of Programming Languages by Embedding, with a Case Study", 2n Intl. Conf on Soft. Eng. 1976.
[188] 木村泉编 "COTO 争论", BIT 1975.5.
[189] 木村泉等, "パネル讨论会", 构造的 "プログラミング" 情报处理 vol. 17, No 11 1976.
[190] 鸟居宏次等 "Structured Programming 1—8", 数理科学 2, 3, 4, 6, 7, 8, 11, 12 1976.
[191] 有泽诚 "Programming Languages with Hierarchical Structure", 情报处理 vol. 17, No 5, 1976.
[192] Nobyoshi Terashima, "The Hierarchical Language System", SIGPLAN NOTICES vol. 12, No 9, 177.
[193] Kimura, I. "A Report on a Symposium on Structured Programming and Experience With It", SIGPLAN NOTICES vol, 12, No 1, 1977.
[194] Stay, J. F. "HIPO and Integrated Program Design", IBM Syst, J. vol. 15 No 2, 1976.
[195] Van Leer, P. "Top-Down Development Using a Program Design Language", IBM Syst. J. vol. 15, No 2, 1976.
[196] Brown, P. J. "Programming and Documenting Software Projects", ACM Comp. Surveys vol. 6, No 4, 1974.
[197] Wirth, N. "On the Composition of Well Structured Programs", 同上刊.
[198] Kernighan, B. W. and Planger, P. J. "Programming Style: Examples and Counterexamples", 同上刊.
[199] Cashman, M. W. "An Interview with Professor Dr. Edsger, W. Dijkstra", Datamation vol. 23, No 5, 1977.
[200] 唐稚松 "XYZ 语言族概要" (上), 中国科学院计算技术研究所资料. 1977.

什么是计算机科学

(1978)

§1 前　言

计算机科学是在计算技术高度发展，计算机应用的领域日益拓广以及有关计算机的基础理论研究逐渐形成的情况下发展起来的一门新兴技术科学。它一出现，即引起人们的重视，在很短的时间内，世界上重要的学府都纷纷设立有关的科系或研究机构。其发展速度之快，吸引人力之多，在新兴科学之中也是比较少见的。

方毅同志在科学大会上的报告中指出："我国计算机科学技术必须有一个新的大发展。三年内，要迅速发展计算机科学和有关学科的基础研究，……到一九八五年要形成比较先进的计算机科学研究力量"[1]。

计算机科学在我国正处于即将开始迅速发展的时刻。许多大学已设立这方面的系或专业。有些研究机构正在着手组织这方面的力量。然而，据本文作者所知，我国学术界对这学科的理解却是相当混乱的。究竟什么是计算机科学，它包含哪些方面的内容？计算机科学与"计算技术"、"计算机工程"、"关于计算机的研究"等概念有什么区别，又有什么联系？计算机科学应如何进行研究？新兴技术的门类很多，为什么计算机的研究却发展成为一门独立的科学？本文作者认为：为了我国计算机事业今后健康地发展，当前对这些问题进行一些调查和讨论是十分必要的。本文试图在这方面作一些初步的探讨，以期抛砖引玉。

§2 计算机科学的形成与发展

"计算机科学"这概念并不是突然出现的。它是在有关计算机技术的发展过程中逐步形成的。只有对这发展过程有一基本的认识，才能了解它的含义，它所表示的领域的内容，以及它之所以出现的原因所在。

现代科学技术的重要特点之一是出现了许许多多门类的技术科学。计算机科学即是这种技术科学之一。因此，要了解什么是计算机科学，首先应了解技术科学的共性。

关于技术科学的特征，钱学森同志的文［2］曾有精辟的论述。技术科学是在

自然科学与工程技术都有了高度发展的基础上，使两方面能在较深刻的水平上相互结合才产生的。文中指出："如果我们要把自然科学的理论应用到工程技术上去，这不是一个简单的推演工作，而是一个非常困难，需要高度创造性的工作……。更确当地说是科学理论与工程技术的综合。因此，有科学基础的工程理论，就不是自然科学本身，也不是工程技术本身，它是介乎二者之间的，它也是两个不同部门的人们的经验的总和，是化合物，不是混合物。……而这个工作内容本身也成为人们知识的一个新部门：技术科学。它是从自然科学和工程技术的互相结合所产生出来的，是为工程技术服务的一门学问。……它自然有不同于自然科学，也有不同于工程技术的地方。因此，研究技术科学的方法，也有些地方不同于研究其它学科的方法"。

计算机科学正是一门这样的技术科学。如 P. Wegner 文［3］中所说："计算机科学既不是一技术分支，也不是一数学分支，它包含一种关于计算图式的新的思想方式。这种图式一方面是技术性的、一方面又是数学性的。而它又包含一种独特的成份，从性质上与各传统学科的内容迥异"。文［13］也说："它（计算机科学）是一门边沿科学，其中精涤的数学技巧与构造经验模型，数据归约和模拟的试探法肩并肩地共存。"

正因为计算机科学具有这样的特性，所以，它很容易被习惯于传统学科概念的人所误解和歪曲。在国外，它的性质和意义也是经过长期的实践和讨论才被人们所理解的。

计算机科学这个名称是在 1961 年左右提出来的，但在这之前美国的高等学校早已开始讲授关于计算机的课程[10]。据文［4］所述："在 1950 年，由于 IBM 公司提供 100 台可自由使用的计算机，同时要求必须讲授程序课程，这样的策略才使得计算在美国学府的门槛内有一立足之地。……但这种早期阶段很难代表现在所理解的计算机科学，也没有多少人把它看作是可以与其它学科处于同等地位，值得予以研究的一门真正学问的萌芽。"应该指出，在 1957 年，我国的钱学森同志在文［2］中已经看出了这样一门与电子计算机相联系的技术科学的前途，给予它以一门独立的技术科学的地位。但是，由于这门学科当时发展水平的局限，他在写该文时还只能把这领域称为"计算技术"，这领域的研究目标还只限于"设计更多更好的，多种多样的电子计算机，和更有效地使用电子计算机"。这领域的研究内容也只有"模拟计算机，数字计算机和复合计算机"，其成果应用也只限于"科学问题的计算和自动控制系统"。这一时期可以说是计算机科学诞生以前的时期，它的主要特征是：计算机使用尚未普及，使用范围一般只限于科学计算，因此，计算机各种应用所表示出的具有科学意义的共性尚未显露。此时，程序多数是由受过专门训练的人编制的，使用的方便性和程序的可靠性这些方面的问题尚未引起人们足够的重视，人们对于程序所关心的主要指标还只是效率问题。因此，高级程序语言的重要意义还未被人们所理解，高级程序语言所强调的形式化方法与信息结构的研究也未被人们所认识。因此，这一时期有关计算机的研究还只是一些分散的技术性的探索，尚未成为一门统一的科学。

P. Wegner 在文［3］中描述了这一时期向下一时期过渡的情况："在四十年代，

计算机的设计者和建造者如哈佛的 Aiken，宾州大学的 Eckert 与 Mauchly，普林斯顿的 Von Neumann 主要关心的是为快速算术运算生产机器，他们对于旨在使计算机便于使用的程序语言及程序系统是毫无兴趣的。……程序语言 Fortran 是在 1955—1960 这段时期发展的，开始的时候，它的发展是被怀疑的，原因是认为它的编译将占去过多的机器时间，而且所编出的程序运行不够有效。在五十年代的初期，被高度尊重的计算机先驱人物如 Von Neumann 即认为计算机的使用者将是一些有足够才智的人，因此像表示符号这样微不足道的事情不致成为障碍。但是，计算机使用人员的范围扩大得如此之快，以致对面向问题的语言的需要到 1960 年已经几乎变成普遍接受的事了。更有进者，甚至那些原来认为面向问题的语言是对不够条件的群众让步的人也已开始认识到这种语言不仅能使范围更广的程序人员使用程序语言，而且还能让出类拔萃的程序员去解决更为复杂以及更为重大的问题，要是没有这种语言，想解决这样的问题是不可能的。”

就是在这一时刻，由于高级语言的出现和发展，由于计算机应用领域的开拓，还由于与此有关的理论研究的进展，人们开始看到在这一新技术的背后存在一种新的有规律的统一性，正是这种统一性使这一技术领域成为一门科学。在这方面提倡最力的人之一是世界上最早的计算机科学系的创建者之一，斯坦福大学计算机科学系主任 G. Forsythe，他在 1961 年第一次用“计算机科学”这个名称，在文［5］中写道：“计算机正发展得如此迅速以致甚至计算机科学家也跟不上，它必定使许多数学家和工程师都感到迷惑不解……虽则其应用的方面多种多样，但用计算机处理困难问题的方法却表现出很大的统一性，计算机科学这个名称正被加到这新产生的学科之上。”在另一篇文章［6］中，他又写道：“机器表示的二进位数字串能模拟许多东西，数目只是其中之一，例如，它们能模拟大路上的摩托车，棋子、电子、乐谱、俄文字、纸上的模式、人体细胞、颜色、电子线路等等，已经知道的关于计算的各式各样的应用足以使我们认识到一种协调的技术统一体的诞生，我称它为计算机科学……不管计算机是用于工程设计，医疗数据处理，还是计算机作曲，或其他方面的应用，其计算的结构是相同的，我想计算机科学系最终要包括以下各方面的专家：程序设计，数值分析，自动机理论，数据处理、商业博弈，自适应系统，信息论，情报检索，递归函数论，计算语言学等。因为这些领域在结构中出现……”。

由于越来越多的人从高级语言的普及和计算机应用的拓广各种事实中认识到上述这种统一性，“计算机科学”这个名词很快地流行起来，同时，世界各大学也纷纷设立以计算机科学命名的系或专业。从以上的过程可以看出，这一技术领域之所以发展成为一门科学，并不单纯是由于它发展得很快或者很普及，（“汽车”、“电话”等等技术也发展得很快，很普及，但并未因此而发展成为一门“汽车科学”或“电话科学”），而是由于计算机应用的拓广使人们认识到这一技术领域的背后存在一种更为本质的东西，它使得所研究的对象具有某种内在的统一性。

人们自然要问：研究这一技术领域的内在统一的规律对于发展这一技术领域有什么意义呢？或者说，将这一技术领域的某些方面的研究，上升成为一门科学、并为之建立专门的科系，其意义何在？本文作者认为，有两方面的意义：（1）如果不对这种更为本质的统一属性进行了解和研究，认识即只能限于表面的技术细节，在

这种情况下，即只能进行技术性的改进而不可能得出本质上新的设计思想，例如，如果不对“计算”的概念进行本质的探索，即不可能设计出像 LISP 那样的语言，正如文［10］所述，“对理论的强调就起了奠定坚实的知识基础的‘补药’或者‘维生素’的作用，在这个基础上才可以对技术的进展加以吸收并进行创新”。当然，这样做时也应注意与实际的联系，脱离了计算机去研究计算的本质也是没有意义的[7]。(2) 对于培养从事这方面研究的学生来说，他对计算机的本质特征有所了解可使他站得更高，看得更远，较易于融会贯通，而不致被各门各类的技术细节所模糊眼界。对于技术科学的研究和培养方法，文［2］曾有详细的论述，那些原则对计算机科学也同样是适用的。

在 §5 中我们将举若干实例来说明计算机科学研究的意义。

§3 计算机科学的定义之争

不同观点很多，下面是较著名的几家：

Newell-Perlis-Simon 在 1967 年文［8］中表示的观点是：

“哪里有现象（Phenomena），哪里就有一门科学去描述和说明这些现象。因此，对于‘什么是植物学？’的最简单（而又正确）的回答是‘植物学是研究植物的学问’。同理，动物学是研究动物的学问，天文学是研究星球的学问等等。现象孕育科学。有计算机在，故计算机科学是研究计算机的学问。围绕计算机的现象是多种多样的，复杂而丰富的。”

这一回答甚为巧辩，但仔细考虑却存在问题：(1) 现象只是认识事物入门的向导，而科学却决定于对事物的本质（noumenon）的认识。决定计算机科学之为科学的不是那些“多种多样”“围绕计算机”的现象，而是存在于这些现象深处的共同的统一性。脱离这种统一性而只谈现象，就要丢失掉计算机科学之所以成为科学的基础。上述答案有鲜明的“现象主义”的哲学背景。本文作者认为，它是不够科学的。(2)“有计算机在故计算机科学是研究计算机的学问”。我们可以问：有电话在，为什么没有“电话科学”？有汽车在，为什么没有“汽车科学”？……之所以有研究植物的植物学，有研究动物的动物学，原因是一切植物有其所以为植物的统一性，正因为存在这种统一性，才有以研究这种统一性为对象的植物学。动物学，天文学也皆如此。新兴的技术门类很多，虽然每一门类的技术也都有一定程度的共性，但如这种共性不是深刻、普遍到足以构成一门独立的科学时，是不会被称为××科学的。而计算机科学所具有的独立性是较为突出的，前面已经提到过了。由于对于这种统一性的研究与这一技术领域的发展关系甚大，人们才将它作为一门独立的科学提出来加以研究。Newell-Perlis-Simon 的答案背离了这一基本事实，所以，是不能令人信服的。而且，如果我们按照这一定义去进行有关计算机科学的研究和教育，我们将会失去重心，无所适从。因为，像“取暖通风”、“打印输出”等等都是“围绕计算机”的现象，我们是否也应该将这些方面研究和教育工作都放在计算机科学的范围之内，如果这样做，计算机科学将成为一大杂烩。

Newell-Perlis-Simon 的观点也有其可取的一面。即强调了计算机科学的研究与计算机的关系。这一点也是应该强调的，否则，即可能脱离实际[7]。

与上述观点相对立，有些人较强调计算机科学所具有的内在统一性。问题是这种统一性究竟是什么？这就因为认识的角度不同而有不同的回答。事实上，前面谈到的 Forsythe 的话中即已有过一种回答：即“统一的计算结构”。他所提出的关于计算机科学系所应包括的各种类型专业的专家的意见，即是根据这统一的计算结构提出来的。不过，到七十年代，对这种统一性又有了进一步的认识。其主要有代表性的观点为 Wegner ［3］ 与 Knuth ［9］ 两家。

Wegner ［3］ 将“计算机文化”分为三种，即“计算机技术”，“计算机数学”与“计算机科学”。他认为计算机科学既不是一技术分支，也不是一数学分支，它包含一种关于计算图式的新的思想方式。“工业革命中起核心作用的是‘能量’，在计算机革命中被‘信息’所取代”。因为，在计算机科学中具有核心重要性的概念是“信息结构的转换”。他认为“计算机科学将会日益成为既非技术亦非数学的一门学科，一种关于信息结构转换的科学将要发展起来，以致对于科学、哲学与认识论方面的影响将如数学或物理学一样的基本和重要”[3]。

他的这种观点主要是从高级程序语言的特征出发来认识计算机科学对象的内在统一性的。他认为“数学语言与程序语言都可以看成是对信息结构进行系统转换的系统”。所谓信息结构的转换，也就是对计算的对象进行“替换”，“代入”，“取值”，“赋值”等等，这些都是基本的信息结构的转换，在此基础上即可定义“计算”的概念。“数学语言的目的是讨论与被表示的对象有关的命题，而程序语言的目的则是构造出可施于对象域上的函数，从而——数学语言的命题蕴涵着验证命题正确性的计算规则，而程序语言的函数则蕴涵计算其函数值的计算规则，在每种情形，其计算规则都可以用对该语言中表达式施之以信息结构的转换来定义”。因此，他认为构造有关信息结构转换的模型，对之进行研究是计算机科学带根本性的问题。实际上，各种高级语言也都是根据各种具体对象的要求对其信息结构的转换作出的某种具体的规定。也可以说，高级语言及其程序系统是计算机科学各种关于信息结构转换的具体的体现。

与 Wegner 观点不同，Knuth ［9］ 是从解题算法角度来看待计算机科学所体现的统一性的。文［9］中说：“我中意的描述计算机科学的办法，是说它是算法的学问。算法是精确定义的一系列规则，指示怎样从给定的输入信息经过有限步骤产生所求的输出信息，算法的特殊表示称为程序，如同我们用‘数据’这个词来代表‘信息’的特殊表示一样。也许由计算机的出现引起的最重要的发现将证明正是这样一个事实，即作为研究的对象，算法有着特别丰富的有趣的性质；再者，算法的观点也是组织一般知识的有用方法”。

用算法来说明计算机科学所讨论的对象的统一性确有其令人信服的一面。因为，计算机所能执行的计算都必须是一算法。不过，将计算机科学定义为算法的学问则有其不够全面之处。因在这样的定义下，计算机科学研究即只剩下对一种一种具体算法的分析，以及算法的效率等一般算法特征的研究，而前面所述作为高级语言或系统程序所主要关心的信息结构等方面的问题，反而变成次要的问题了。而且，算

法的讨论是历史上早有的事，如果计算机科学的内容等同于算法的研究，则在几千年以前即早已有计算机科学家了。这样的提法有使计算机科学的研究脱离计算机的危险[7]。

看来，由于计算机科学还很年轻，想要用很简单的一种概念概括全部计算机科学的内容，目前还难免不够全面。这种定义之争在许多学科也都存在。甚至象数学这样古老的学科，对于“什么是数学”，至今仍不免有争议。不过这种争议并不妨碍数学研究的进行和发展。计算机科学也是如此。但是，上面 §2 中所述的那种使计算机科学成为科学的内在统一性则是客观存在，对于这种统一性的意义亦勿容怀疑。至于对什么是这种统一性这个问题存在各种不同的答案，这只能表明这门科学的特征还表露得不够充分，尚有待进一步的研究。本文作者认为，目前不必要去纠缠于这种纯学术性的讨论，也不必去等待一个大家一致的答案。目前需要的是分清计算机科学与计算机工程的界限，并根据现已存在的客观事实，实事求是地分划出计算机科学所指的对象的范围。

下面著名的 Ramamoorthy 文［10］中对此作了明确的回答：“我们没有给计算机科学与计算机工程这两个术语下定义就已经使用它们了。更仔细地观察一下也许能清楚地看出它们的区别。计算机科学家对于计算和程序设计的理论和科学感兴趣。因此，诸如自动机理论、语言的形式理论、计算的复杂性理论、数值分析和算法数学基础、数据结构（这就是根据沃斯所说的程序设计科学），这些领域组成了计算机科学这门科学。计算机工程师则对包括软件、硬件的数据处理系统的规格、设计、实现和实用（操作）感兴趣。因此，计算机工程学可以定义为工程学的一个领域，它涉及到作为通用计算机，或更大型系统的组成部分的数字处理系统的结构、设计和实用。计算机工程师（包括软件工程师）为了计算机的具体运用，在计算机系统的规格、设计、实现和实用方面运用了计算机科学和（/或）电机工程学的原理”。

“最新的一个调查就是由斯漏思教授于 1974 年下半年为计算机学会作的。从她所调查的 222 个电机工程系和 95 个计算机科学系中，她得到了 101 个电机工程系和 59 个计算机科学系的响应。……这个调查指出，计算机科学系往往专门研究软件，数值分析和理论方面的课题，而电机工程系重点放在硬件的设计和结构以及小型计算机、微型计算机的设计方面”[10]。

本文作者认为上述回答是有充分根据的，令人信服的。为了进一步说明问题，我们想在下一节进一步就四种不同类型的材料进行一些分析。

§4　计算机科学的范围与特征

本节先就四种不同类型的材料进行一些分析，然后再希望从现存的事实中归纳出计算机科学的研究范围与特征。

（一）1968 年，美国计算机协会计算机科学课程设置委员会的报告：“68 教程”[11]。这是 60 年代美国许多大学的计算机科学家经过长期认真的讨论后提出的，对美国计算机科学的教育有十分重要的影响。其中谈到有关计算机科学的课程有以

下三个部分：

（Ⅰ）信息结构处理：包括（1）数据结构，（2）程序设计语言，（3）计算的理论模型。

（Ⅱ）信息处理系统：（1）计算机设计和组成，（2）翻译程序与解释程序，（3）计算机和操作系统，（4）专用系统。

（Ⅲ）方法学：（1）数值数学，（2）数据处理与文件管理，（3）符号处理，（4）文本管理，（5）计算机图形学，（6）模拟，（7）信息检索，（8）人工智能，（9）过程控制，（10）计算机辅助教学。

（二）1976年，日本BIT杂志连载了十期关于十个世界上著名的计算机科学系如美国的MIT，Standford，英国的Cambridge，法国的Grenoble等的调查报告。本文限于篇幅，难以作全面的分析，兹以MIT的情况为例[12]。1976年该系共开设41门课程，可分以下几类：

（Ⅰ）关于数理逻辑与可计算性理论的共5门；

（Ⅱ）关于算法分析（理论性的）的共3门；

（Ⅲ）关于程序语言的共5门；

（Ⅳ）关于自然语言处理的共2门；

（Ⅴ）关于人工智能的共7门；

（Ⅵ）关于机器的体系与组织的共3门；

（Ⅶ）关于数字系统的共2门；

（Ⅷ）关于数据库及信息系统的共4门；

（Ⅸ）关于计算机方法的共1门；

（Ⅹ）其他应用如疾病分析与模拟，代数处理，机器视觉与感知各1门；

（Ⅺ）未分类的计算机科学课程3门。

（三）美国IBM公司从事探索性和理论性研究的机构共有四处，在纽约有二处（华森研究中心和系统研究所），加州一处为西岸研究所，西欧一处位于瑞士，此外，在西德、英、法、日、意、以色列和西班牙还有其分公司的研究机构。其中以华森研究中心为最著名。

华森研究中心有1600人，其中技术人员约700人左右，下设①数学分部，②物理分部，③计算机科学分部，④应用研究分部。

关于计算机科学的理论性研究是在数学分部进行的，故在计算机科学分部中不包括这部分工作。

其计算机科学分部中包含以下的组（据1976年的材料［13］）：

（Ⅰ）识别与自动化，其中包括以下专题：（ⅰ）自动化研究，研制了一种非常高级的语言Autopass，利用它可用计算机组装机器部件，（ⅱ）口语识别，（ⅲ）手书验证，（ⅳ）超声医学（利用计算机加工医疗诊断信息的超声数据），（ⅴ）文字识别。

（Ⅱ）信号加工与应用，其中包括以下专题：（ⅰ）信号加工子系统，如用软件实现足够有效的乘法，（ⅱ）口语终端，（ⅲ）终端组的分散控制，（ⅳ）口语文

件、研究适用于未来办公室管理的系统和技术。

（Ⅲ）系统分析与算法，其中包括以下专题：（ⅰ）对复杂系统和子系统的理解，（ⅱ）对复杂系统和子系统的改进。

（Ⅳ）自动程序设计法，其中包括：（ⅰ）研究普通人写计算机程序的困难所在，（ⅱ）商业应用自动化系统，（ⅲ）研制能为许多人公用的程序，（ⅳ）对程序进行说明的系统，（ⅴ）程序优化的研究，（ⅵ）理论计算机科学。

（Ⅴ）801 小型计算机系统的代价与功效比例的分析和研究。

（Ⅵ）程序技术的研究，（ⅰ）优化微程序的自动生成，（ⅱ）微程序的验证，（ⅲ）程序验证与分析。

（Ⅶ）大型软件计划发展途径的研究。

（四）作为一种旁证，我们还可以分析一下十多年来“Turing 奖”的授奖情况。“Turing 奖”是目前国际上计算机科学领域内最高的荣誉奖，专授予这领域内在某方面有开创性成就的科学家，自 1966 年开始，共授奖十次，得奖人姓名及其主要贡献如下[14]：

获奖时间	获奖人	主要贡献
1966	A. J. Perlis（美）	计算机语言方面的贡献
1967	M. V. Wilkes（英）	微程序设计及程序设计方法
1968	R. W. Hamming（美）	编码理论及计算机应用
1970	J. H. Wilkinson（英）	计算方法
1971	J. McCarthy（美）	LISP 语言，关于计算的理论，机器人的研究
1972	E. W. Dijkstra（荷）	结构程序设计，操作系统中死锁概念，P. V. 操作，分层思想
1973	C. W. Bachman（美）	第一个可用商业数据库的创建者，网状数据库的发明人
1974	D. Knuth（美）	“计算机程序设计艺术”巨著的作者，算法分析领域的开创者
1975	A. Newell（美） H. A. Simon（美）	应用试探法研究人工智能这一领域的开创人
1976	M. O. Rabin（以） D. Scott（英）	首先提出非确定性有穷自动机概念，并在计算机理论方面作了一系列重大的贡献

以上四种类型的材料应该说在计算机科学的教育与研究方面是较有代表性的。从这些材料可以看出，至少近十年来，计算机科学的研究范围主要集中在：（1）语言与程序系统，（2）特殊应用（人工智能）等方面，自然也应该包括计算数学。至于体系设计方面，似乎足以称为计算机科学的研究工作不多（当然，也还是有的，如 Illiac Ⅳ）。对以上分析，还应补充以下几点：

（i）整机体系方面要进行实质性的创新看来是一难度很大的工作，而且，一直缺乏足够的理论基础。近年来虽经常有新型计算机出现，其中有些从工程的角度看的确可能是很好的设计，但从计算机科学的角度却未必有什么实质性的创新。无怪去年 J. Schwartz 教授在体系设计座谈会上说：“计算机体系多年来一直很稳定，看来今后也不大可能会有大的改变。计算机科学的核心问题是软件问题而不是体系设计方面的问题……”[13]。看来，这些话代表一部分计算机科学家的意见。至少应该承认，事实上从计算机科学的角度来看，体系设计方面实质性创新的工作是很少的，因此，从过去若干年的事实来看，这方面的研究的确不是计算机科学的主流。但也

应该承认，这种创新也还不是完全没有，比如Illiac IV。虽然从工程的角度看它不能算是很成功的机器，但从计算机科学的角度看来，它在设计思想上是有实质性的创新的。近年来，MIT的Dennis等人所从事的数据流机器的研究也是一种开创性的工作。

（ii）有关计算机的研究并不都包括在计算机科学之中，但并不因此就说明这些研究不重要。比如，有关存储技术以及输入输出设备的研究虽不能算是计算机科学的研究却是非常重要的研究工作。华森研究中心关于这方面的研究工作很多，但多属于计算机科学部以外的分部。同时，这类工作有些是互相交错的，比如模式识别，其理论研究应该是计算机科学研究的一个很重要的方面，而其工程实现的研究又应该属于技术研究的范围。的确有时两个方面紧密结合，很难区分。

（iii）技术性研究与科学性研究的分划在软件和应用的领域内同样存在。比如，许多著名的语言如ALGOL 60，LISP，APL，PASCAL等在其开始建立时应该说是很好的计算机科学的研究成果，但它们已经建成后，其推广应用、移植或技术上的改进和精化，或稍作技术性的改动而成另一名称的语言等等这方面的工作即不能再算作是计算机科学的工作了。

因此，可以看出，表示计算机科学特征的不是其研究的领域，而更重要的是其研究的性质，即创造性和理论性。说得更具体些，本文作者认为计算机科学的研究工作可分为以下四类：（a）新的系统，（b）新的概念和新的思想，（c）新的方法，（d）基础理论。

（a）新的系统：比如新的语言（如ALGOL 60，LISP…当其初创立的时候即是新的语言），新的程序系统（如THE系），新的体系（如Illiac IV），新的人工智能系统（如Schank关于自然语言识别的系统）等。这些系统之所以为“新”，是因为它是根据本质上（不是枝节上）新的思想和概念建立起来的。因此，第二类工作是：

（b）新的思想和概念。比如Hoare关于数据结构的理论（这是PASCAL的依据），Dijkstra关于结构程序设计的思想（这是THE以及许多新操作系统的依据），Floyd关于不确定算法的概念（这是许多人工智能语言的依据），Von Neumann关于细胞型自动机的思想（这是阵列型计算机的理论模型）。

（c）新的方法，比如Floyd优先算子文法，Allen等的优化方法，Wilkes的微程序设计方法等。这些方法中许多是要求数学论证的，所以，多数这方面工作本身又是一种应用数学理论。

（d）基础理论工作：比如算法复杂性理论，程序证明论理论等。这些方面的工作大多是数学上较为深刻的具有基础性质的工作，但它又不同于一般的数学基本理论，它的问题是从计算机领域提出的，而且它往往具有为计算机科学奠定某方面基础的意义。但它的发展前途却有许多种可能性：有些工作可能发展成为具体应用的方法或新的系统，有些工作有可能发展成数学基本理论而逐渐离开计算机科学的范围。所以，这方面的工作往往不是固定不变的，而且，常常与基础数学之间存在一界限不分明的交集。这种情况，在上述新的系统那类工作与计算机工程技术性工作之间也同样存在。这是不足为奇的，所有边缘性研究都有类似的情况。

一言以蔽之，计算机科学作为一门科学的一个最本质的特征是创新。正如文[13]所说："正是这种求新的素质使计算机科学如此独特地使具有创造才能的科学家感到激动和满足。"

归纳起来即：计算机科学的范围是关于软件、特殊应用、计算数学以及较小程度上也包括体系设计等领域的以创新为特征的探索性和理论性的研究。

§5 计算机科学主要内容介绍

（略）

参 考 文 献

[1] 方毅，在全国科学大会上的报告（摘要），（一九七八年三月十八日），人民日报，1978. 3. 29.

[2] 钱学森，论技术科学，科学通报，1957. 4.

[3] Wegner, P., Three Compute Cultures, Computer Technology, Computer Mathematics, and Computer Science, Advances in Computers 10, 1970.

[4] Knuth, D. E., George Forsythe and Development of Computer Science, CACM, 1972, 15. 8.

[5] Forsythe, C., Engineering student must learn both computing and Mathematics, J. Eng, Educ. 52, 1961.

[6] Forsythe, C., Educational Implication of Computer revolution Applications of Digitial Computer, 据文[4]转引.

[7] Hamming, R. W., One man's View on Computer Science, J ACM, 1967.

[8] Newell, A. et. al. Computer Science, Science 157, 3, 1967.

[9] Knuth, D. E., Computer Science and It's Relation to Mathematics, Amer. Math. Monthly, Vol. 81, 1972.

[10] Ramamoorthy, C. V., Computer Science and Engineering Education IEEE Trans. on Comp. EC-25 No. 12, 1976.

[11] Atchison, W. F., 68 Curriculum CACM, 1968, 3.

[12] 米泽明宪（毛丽英译）MIT（麻省理工学院）计算机科学系介绍，BIT 1976. 1.

[13] Goldberg, P. C. ed. Computer Science Department 1975—1976 Report, Thomas J. watson Research Center Yorktown Heights NY 10598 RA83 (#26828) 10126176.

[14] 1966—1970 四次 Turing 报告载于次年 JACM，1971 以后载于同年 CACM.

[15] Schwartz, J. 座谈会报告 1977. 9. 于北京.

XYZ: A Program Development Environment Based on Temporal Logic

(Programming Languages and System Design, 1983)

OVERVIEW

XYZ is a Program Development Environment conforming to three ways of programming: programming with flowchart, programming with higher level languages and programming with specification. It is built around a family of transformable programming languages whose unified logic basis is a temporal logic language XYZ/E, which is both a temporal logic system and an UNCOL-like intermediate programming language.

Among the most influential Program Development Environment systems, Cornell program Synthesizer [15] and Gandalf [12] are well known for their syntax directed editors. Indeed, they have the advantage of unifying the different phases of the program development procedure into a coherent integrated system. In spite of all these, I still find, there are some aspects which seem not very satisfactory:

(i) These systems start from the editing phase of programming that presupposes the user's program having been composed to some extent before using the system. In my opinion, it would be more useful if a program development environment system starts from the design phase of programming which would involve more try-and-error activities.

(ii) All such kind systems are oriented toward higher level languages whose syntax is based on phrase structure grammar. The embedding structure of this grammar seems not consistent with the stepwise nature of dynamic execution of a program in debugging. In my opinion, a program in flowchart structure is more suitable for this purpose.

(iii) In fact, there exist three different ways (levels) of programming: programming with flowchart, programming with higher level languages and programming with specification. Each has its advantages as well as shortages in some application fields. An ideal and universal programming development environment had better be designed to conform to all these three ways of programming.

There are many different desirable qualities required of a good programming language, such as readability, writability, suitability for logical reasoning etc. Until now, no one single language can be taken as an ideal model good in all these aspects. In my opinion, it might be impossible and even unnecessary to make effort to invent such a single ideal lan-

guage. From the standpoint of automatic programming system, if, instead of one single ideal language, a good system can be constructed, which contains a family of coherent languages; each of them has one or two of above desirable qualities and there exist in the system simple transformations which can translate a program of one language into that of the other automatically, then the system as a whole can be looked as having all these good qualities. I believe, this might be a more practical approach in the design philosophy of programming languages.

As Habermann [12] rightly pointed out: "It seems that the development of imperative languages has reached its limit. At the same time we conclude that these languages have some serious drawbacks regarding their complicated semantics and program verification". In recent years, there are a lot of efforts in constructing logical languages, such as those functional languages based on λ-calculus and prolog based on predicate calculus. In spite of all their achievements, I find that they either suffer from the limitation of application areas or depart too far from customary programming practice. What I am doing is to show a temporal logic language can serve as the unified logic basis of customary programming. Its own semantics is very simple and correspondent to the flowchart and there are simple transformations between this UNCOL-like language and the conventional higher level languages such as pascal.

This logic language has the advantage in logical reasoning such as verification [10] and formal semantics [9]. But it also has its own shortcoming, i. e. not very readable. So this is not a language for common use.

But there is an external form of this logic language that can be obtained from it by application of some simple abbreviation rules. That external form is a readable flowchart command language, connected with this language there is a flowchart-directed editor in the system. By means of it the user can design, edit, debug and implement a program with flowchart structure. The whole program development procedure is done interactively with one window of the display showing the graph of the flowchart, another showing its corresponding program and a third window showing an equivalent program written by higher level language (e. g. pascal), which is obtained automatically by the transformation built in the system. There is also in the system an analyzer of well-structureness of a flowchart. It would make a caution if any bad-structured flowchart occurs.

The system also contains a meta-language which is suitable for describing the semantic transformation from higher level languages to the temporal logic language (or its external form). The transformation so described can serve both as the formal semantics of that higher level language and as its semantic-directed compiler [9]. There is a simple way to implement this transformation.

This system not only can accept the programs of PASCAL (even hopefully, ADA), but also has its own higher level language XYZ/C. It is one oriented toward Chinese characters processing. Perlis humorously points out in his philosophical epigrams [14]: "Chinese

ought to like APL, but they spend their money in Fortran." "Natural language is not natural in computers." Our XYZ/C is a hybrid of APL and PASCAL in the sense that its representation is of APL style but its data structures are PASCAL-like. Its basic structure is also consistent with XYZ/E. Besides, it also contains Modula-like modules.

In connection with this language system, there are a group of tools, among them, there is a chinese characters processing system XYZ/H based on [3] and also a parser generator XYZ/PC based on a two-level grammar [2].

Finally, XYZ system can also serve as a tool to synthesize a program under the direction of its specification. This tool is based on a method [6], [8] that is to derive the structure of a program by analyzing the logic constants in its specification. This is an approach somehow similar to Dijkstra's [11], but we have our own particularities:

(1) Our emphasis is on the analysis of quantifiers and other logical connectives instead of the latter alone.

(2) We derive the structure of the program by analysis of both the pre- and the post-conditions, this approach can avoid lots of difficulties which would occur if only post-conditions are brought into consideration.

(3) Our formal means is an extension of flowchart called assertion flowchart instead of wp.

(4) The net result of our method is that to synthesize a program from its specification, what depends on designer's knowledge and creativity are only:

(a) to change exit assertion according to the choice of algorithm.

(b) to insert assignment statements into those nodes which determine the values of variables.

Only these parts cannot be generated automatically from the informations given by the specification.

The program so designed is a flowchart program which finally is mapped into XYZ/E.

THE TEMPORAL LOGIC LANGUAGE XYZ/E

In XYZ/E, a variable is distinguished from its name which is a character string consisting of three parts: the leftmost part is a type symbol which is either a capital letter "I", "C", "B", "PX" or empty to denote the type of the variable being an integer, a character string, a Boolean, a pointer referencing an object of type X or the rest (i. e. a label) respectively; the second part is the root of the name, it is an identifier. The last part is the index part which is either empty or a series of nodes. The node can be an identifier with a hyphen preceeding it or an integer grouped by a pair of parenthesis. The first kind of nodes represents the components of a record; the last one represents those of an array, e. g. Iabc [3]-age is a name in which I is the type, abc is the root and the rest is the index. Related to the concept of a name, there is a kind of expression whose value is a name. It is called a name schema. Its form looks like a name, except the parenthesized node of the index, in

which there is an integral variable instead of an integer, e. g. labc [# Kabc] -age. The capital letter K represents that the variable is a counter which only takes positive integers as its values.

For a name v, "# v" is a general variable. It represents its value at present time t, "o # v" represents the value of this variable at time t + 1, i. e. next time, so "o^k# v" represents the value of this variable at time t + k. To make use of this convention, we can represent "assignment", say "v ← u + 1", by an equation "o # v = # u + 1" and "jump", say "go to p", by an equation "o # lb = p" where "lb" is the name of a system variable which is used to store the current control label. Consequently, "# lb = p" can be used to represent the definitional occurence of current label, i. e. "p:". We call these two kinds of equations with # lb occurring in it "lb equations". It represents the control flow of a program.

It is easily seen that by means of these lb equations every control statement such as loop, case statement and compound statement all can be represented. Similarly, we assume I, C, B as elementary data type. Every data structure such as array, type union and record can be constructed from these elementary data type by means of control flow of iteration, disjunction and sequencing, which again can be represented by lb equation. We call "I", "C", "B" allocational formulas. They are introduced in order to omit the type symbols in the name of variables.

XYZ/E is a many-sorted temporal logic system. In this system, "If…then…else" can be used both as propositional connectives and as expressional connectives. It contains modal operators "□" (necessity), "◇" (eventuality), "o" (next time), "U" (until), A name schema is taken as an expression. A program is a well formed formula of special form:

$$\Box\ [P_1 \Rightarrow Q_1;\ \ldots;\ P_n \Rightarrow Q_n]$$

where "$P_i \Rightarrow Q_i$" represents "if P_i then Q_i else **T**". Each P_i always has the form "# lb = l ∧ R" where R is a predicate. If R is **T** (true), then "∧ **T**" can be omited from P_i. If R is not **T**, then there must be in the program a $P_j \Rightarrow Q_j$, j≠i, where P_j is of the form "# lb = l ∧ ¬ R". Each Q_i is of the form "S ∧ o # lb = m" where "S" is a predicate. Similarly "**T** ∧ o # lb = m" can be abbreviated into "o # lb = m". Each $P_i \Rightarrow Q_i$ is called a "conditional element" in the program. A program is always decomposed into subparts with brackets "% i[…]", "% o[…]", "% io[…]", "% v[…]" and "% a[…]" to represent input variable declarations, output variable declarations, input-output variable declarations, local variable declarations and algorithm part respectively. These brackets are inserted into a program only for the sake of enhancement of readability.

The following is an example:

Example 1. Square root

```
□ [# lb = Sqrt⇒o # lb = lm;
        % i[# lb = lm⇒I ∧ o # lb = ln];
        % o[# lb = ln⇒I ∧ o # lb = lk];
```

% v[# lb = lk⇒l ∧ o # lb = lp;
# lb = lp⇒l ∧ o # lb = l1];
% a[# lb = l1 ⇒o # lk = 0 ∧ o # lp = 1 ∧ o # lb = l2;
# lb = l2 ∧ # lp > # lm⇒o # lb = l4;
# lb = l2 ∧ # lp≤# lm⇒o # lb = l3;
# lb = l3 ⇒o # lp = # lp + 2 * # lk + 3 ∧ o # lb = l5;
# lb = l5 ⇒o # lk = # lk + 1 ∧ o # lb = l2;
# lb = l4 ⇒o # ln = lk ∧ o # lb = stop]].

Obviously, this language is very simple, but it is also very strong. Almost every essential construct of conventional higher level languages can be expressed in this language. As a logic system, there are many characteristics that can facilitate doing reasoning on a program. E.g. for any program C, a subgram S in it with P as its precondition and Q as its post condition can be replaced by the predicate P⊃◇Q in the process of reasoning. Based on this idea, a verification method is introduced in [10], that can be used in both topdown and bottom-up programming. Besides, intermittent proof can also be expressed in this system.

A FLOWCHART COMMAND LANGUAGE XYZ/F

The flowchart command language XYZ/F is the external form of XYZ/E. There are a group of simple abbreviation rules, to apply them to a XYZ/E program, they can transform it into its more readable external form. The statements in this language correspond to various kinds of nodes of a flowchart. So it can be used as commands in constructing a flowchart if some kind of flowchart directed editor is constructed in connection with this language.

The abbreviation rules are as follows:

(1) To change each lb equation of the form "# lb = m", "o # lb = n" into "m:", "↑n" respectively. To change each assignment equation "o # v = exp" into a conventional assignment statement, say, of the form "# v ← exp"; to omit the type symbol from the name of the variables. To transform each paired conditional elements "l: R⇒π1; l: ¬ R ⇒π2;" into an enclosed form as: "l: [R⇒π1; ¬ R⇒π2];" which is called an enclosed pair.

(2) In XYZ/E, all conditional elements in a program are commutative. Now we fix their relative position in program according to following method:

(a) to move any conditional element with definitional occurrence of a label, say l, in front of it, e.g. "l: ⇒π" or "l: [R⇒π1; ¬ R⇒π2]" to a position just subsequent to the conditional element with "↑l" at its right end, if the latter is not in the scope of an enclosed pair. If there are more than one such conditional elements, then choose any one of them as the predecessor.

(b) then, to omit those "goto" s, i.e. "↑l" or "∧↑l" from the conditional elements whose successor has definitional occurrence of this same label l in

front of it. In the algorithm part (i. e. within the brackets % a[...]), for a definitional occurrence if a label, say l (i. e. "l:"), if there is no conditional element in the program with "↑l" occuring in it, this definitional occurrence of the label can be omitted.

(3) To cancel the symbol "⇒" from the conditional elements with a definitional occurrence of a label immediately preceeding this symbol, i. e. those of the form "l: ⇒π"; and to cancel the symbol ";" from those conditional elements of the form "l:;".

(4) For any sequence of formulas separated by semicolons, say A1; ...; An, there are grouping conditions:

(a) only the definitional label of A1 can be referred to from a goto statement outside of the sequence of A1, ..., An (i. e. no abnormal entry).

(b) only the last conditional element of the sequence, say An, can contain a goto statement referring to a label outside this sequence. (No abnormal exit.)

If these conditions are satisfied by A1; ...; An, then they can be grouped by a pair of square brackets and called a bracketed sequence. A bracketed sequence corresponds to a subflowchart with one entry and one exit.

(5) If a statement S preceeded by a definitional occurrence of a label l, i. e. "l: S", this label is referenced by only one goto statement "↑l". Then this goto statement can be replaced by S and the statement "l: S" can be cancelled from the program.

After applying these rules to Example 1, the program is transformed into the following form:

Example 1 (cont.)

```
[sqrt:
  % i[m: 1];
  % o[n: 1];
  % v[k: 1; p: 1];
  % a[# k ← 0 ∧ # p ← 1;
       l2: [# p > # m ⇒ # n ← # k ∧ ↑stop;
             # p < # m ⇒[ # p ← # p + 2 * # k + 3; # k ← # k + 1; ↑l2]]]].
```

The syntax of the commands can be summarized as follows:

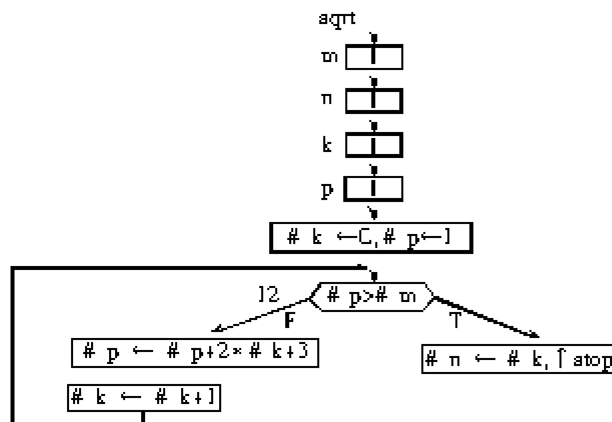
```
<command> ::= <assig> | <goto> | <para> | <pair> | <comp> | <label>: <command>
<assig> ::= <var> ← <exp>
<goto> ::= ↑ <label> | ↑stop
<para> ::= <elem seq> ∧ <end elem> | <goto>
<elem> ::= <assig> | <predicate> | <comp>
<end elem> ::= <elem> | <goto>
<elem seq> ::= <elem> | <elem seq> ∧ <elem>
<pair> ::= [ <predicate> ⇒ <para>; ¬<predicate> ⇒ <para>]
```

<command seq> : : = <command> | <command seq> ; <command>

<comp> : : = [<command seq>]

It is easily seen, that the commands other than "goto" and "pair" correspond those flowchart nodes with one entry edge and one exit edge; "pair" corresponds to a decision node; "goto" corresponds to the exit edge; the definitional occurrence of a label expresses the name of a node. If a command with the right end "goto" omitted, it means, its exit edge leads to the next node. "comp" corresponds to a subflowchart with one exit edge and one entry edge.

The flowchart corresponding to the program in Example 1 is:



Connected with this command language, there is a flowchart directed editor which contains many other commands. By means of them the user can edit, debug and implement a flowchart program.

Although the flowchart command language is much more readable than XYZ/E, it is still not a higher level language in conventional sense. In order to transform a flowchart program into one expressed by conventional higher level language, the system needs to check whether the flowchart is well structured. In the process of designing the program, if a "goto" is added which gives rise to a bad-structrued loop, the system would make a caution. For a well-structured program, the system would automatically transform it into a PASCAL-like program.

For example, the program in Example 1 would be transformed into following higher level form:

```
sqrt    (in/ m: integer; out/ n: integer)
             begin var k: integer; p: integer;
                 # k ← 0; # p ← 1;
                 while # p  > # m
                       do begin # p ← # p + 2 * # k + 3; # k ←# k + 1
                          end
```

```
        end
        # n ← # k
    end
```

SEMANTICS-DIRECTED COMPILATION

As a temporal logic system, XYZ/E's own semantics is very simple and can be easily described with denotational method without such difficulties as "continuation". On the other hand, there exist simple transformations which can map the semantics of any higher level languages such as PASCAL into this UNCOL-like intermediate language without such complicated problem as constructing symbol table. We call this approach two-level formal semantics and take such kind transformation as semantics-directed compilation.

In order to be adaptable to programming with higher level languages, XYZ contains a metalanguage β-equation (or XYZ/B) which can be used to describe the transformations from higher level languages into XYZ/E. In this description, there is a β-equation corresponding to each syntactial production. In the process of transforming, first, parse the syntax according to LR(k) grammar and then, do semantics transformations inversely; corresponding to each reduction of production i, do the semantical replacement with the ith β-equation. When this process is finished, the transformation is also done.

In following Example 2, it is easily seen how to use these β-equations to describe the formal semantics of a simple language. There are some meta symbols of this meta language used in the description: "β" is the name of the semantic mapping; "↔" is the symbol of replacement. In addition, there are some symbols used to indicate the context sensitive semantics: (i) "⌊π⌋" ("⌈π⌉") expresses that π is an upper (lower) context, here "⌊ ⌋" ("⌈ ⌉") is a pair of upper (lower) context grouping symbols. (ii) "⌊·⌋" ("⌈·⌉") is a location symbol for upper (lower) context.

In the process of replacement, when an upper (lower) context location SYMBOL "⌊·⌋" ("⌈·⌉") occurs, the action is to replace this symbol with the rightmost (leftmost) upper (lower) context which occurs on the left (right) side of this location symbol. In case a pair of context symbols occurs with an empty context in it, then the system generates a new name into it.

Example 2. The syntax and semantics of SPL.

no.

1 <decs> ::= <decl>

β <decs> ↔ β <decl>

2 <decs> ::= <decs>; <decl>

β <decs> ↔ β <decs>; β <decl>

3 <decl> ::= <id>: <type>

β <decl> ↔ ⌊β <id>⌋: ⌈β <type>⌉

4 <decl> ::= <id>; <decl>

$\beta < decl > \leftrightarrow \lfloor \beta < id > \rfloor : \lceil \cdot \rceil ; \; \beta < decl >$

5 $< type > :: = \underline{integer}$

$\beta < type > \leftrightarrow]$

6 $< type > :: = \underline{array} \; (< exp >)$

$\beta < type > \leftrightarrow \# \, k\lfloor \cdot \rfloor \leq \beta < exp > \Rightarrow \lfloor \cdot \rfloor \; (\# \, k\lfloor \cdot \rfloor) : \;];$

$o \, \# \, k\lfloor \cdot \rfloor = k\lfloor \cdot \rfloor + 1 \wedge \uparrow \lfloor \cdot \rfloor ;$

7 $< sts > :: = < st >$

$\beta < sts > \leftrightarrow \beta < st >$

8 $< sts > :: = < sts > ; \; < st >$

$\beta < sts > \leftrightarrow \beta < sts > ; \; \beta < st >$

9 $< st > :: = < var > : \; = < exp >$

$\beta < st > \leftrightarrow o\beta < var > = \beta < exp > ;$

10 $< st > :: = \underline{for} < var > : \; = < exp >_1 \underline{to} < exp >_2 \underline{do} \; < st >$

$\beta < st > \leftrightarrow o\beta < var > = \beta < exp >_1 ;$

$\lfloor \; \rfloor : \; \beta < var > \leq \beta < exp >_2 \Rightarrow \beta < st > ;$

$o\beta < var > = \beta < var > + 1 \wedge \uparrow \lfloor \cdot \rfloor \; \beta < var > \; > \beta < exp >_2 \Rightarrow$

FINAL REMARKS

These are only parts of the system. In order to make this paper not too long we have omitted many contents. E. g. there is an APL-PASCAL-like language XYZ/G which is oriented toward Chinese characters processing [7]. There is a tool that can be used to derive the program structures from analyzing the quantifiers and other logical constants in the specification [6,8]. We also have considered the issue of composing viable system versions out of component specifications as what has been done in Gandalf [12]. A verification method related to temporal logic language has been investigated [10].

ACKNOWLEDGEMENT: The author is grateful to Professors J. McCarthy, Z. Manna, R. Yeh, and N. Habermann for their help. Some essential parts of this system are based on the research done in the last three years during my visit to Stanford University, University of Maryland, and Carnegie-Mellon University.

REFERENCES

[1] Tang, Chih-sung, "Structured Programming and Structured Programming Languages", Techn. Rept. of Inst. of Comp. Techn. Acad. Sin., 1977.

[2] Tang, Chih-sung, "LBLR(k) Grammar and Grammatic Decomposition", Chinese Comp. J. no. 1, 1980.

[3] Tang, Chih-sung, "Toward an Unified Logical Basis for Programming Languages", Rept No. Stan-CS-81-865., Dept. of Comp. Sci. Stanford Univ., 1981.

[4] Tang, Chih-sung, "On the problem of Inputting Chinese Characters", Rept No. Stan-CS-81-848., Dept. of Comp. Sci. Stanford Univ., 1981.

[5] Tang, Chih-sung, "Expressing the Formal Semantics of CSP and CP with the Temporal Logic Language XYZ/E", Tech. Rept. of Comp. Sci. Dept of Univ. of Maryland, College Park, 1981.

[6] Tang, Chih-sung, "A Hierarchical Specification-directed Program Design Methodology", Draft, 1981.

[7] Tang, Chih-sung, He, T. M., and Xu, F. C., "The Common Base Language XYZ/C", Computer Science (Chinese), 1979.

[8] Tang, Chih-sung, Lin, F. M., "To Derive Program from Specification". To appear in "Research and Development of Computers" (in Chinese), 1983.

[9] Tang, Chih-sung, Zhen, M. S., and Li, X., "Two Level Formal semantics and Semantics-directed Compilation", To appear in Scientia Sinica.

[10] Feng, Y. L., Lin, F. M., and Tang, C. S., "A Correctness Deduction System for Temporal Logic Programs". To appear in Chinese Journal of Computers, 1983.

[11] Dijkstra, E., A Discipline of Programming. Prentice Hall, 1976.

[12] Habermann, A. N., "An Overview of the Gandalf Project", Comp. Sci. Res. Rev 1978-79, CMU, 1979.

[13] Habermann, A. N., Faculty Research Guide 1981-82, Dept. of Comp. Sci. CMU.

[14] Perlis, A., Epigrams, Draft.

[15] Teitelbaum, T.; and Reps, T., "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", Comm. ACM, Vol. 24, No. 9, 1981.

程序技术研究三十年

（计算机科学，1988）

§1 前 言

自五十年代末期，Backus 完成 Fortran 的编译系统以来，程序技术的研究已有三十年的历史了。三十年来，计算机硬软件技术都有了很大的发展和变化，但仍有些带根本性的目标与问题并没有完全解决，或者说变化不大。如何认识这些不变的和变的方面，如何看清这一技术领域发展中带规律性的东西以及今后的发展动向。不言而喻，对安排研究工作乃至制定有关的技术政策都是有重要意义的。

八十年代以来，各先进工业国家都在制定信息技术的战略规划，如日本第五代计算机计划、英国 Alvey 计划、西欧共同体 Esprit 计划、美国国防部 Stars 计划以及美国工业界 MCC 组织等。它们虽因各国具体条件不同而有不同的侧重面，但对一些根本性问题以及对这一技术领域的发展规律和今后动向的认识却大体上是相同的。这些共同点又是什么呢？

本文虽着重讨论程序技术，但不可避免地也将涉及与它有关的相邻领域。“程序设计”这个词，可以作广义的理解，也可以作狭义的理解。从广义说，它与“软件”这个词所指的对象大体相同；从狭义上说，我们将软件分为程序，数据与知识三个方面。本文中讨论的问题大体以程序设计为主，但必将涉及数据（数据库）及知识（人工智能）的某些方面。本文中谈到的“技术”，也应作广义的理解，即不只是狭义的技术，而且也包括与发展新技术紧密相关的理论。事实上，在程序领域，技术与理论并无截然分开的界限，往往一个时期的理论研究正是后一时期新技术的先导。

在程序技术领域方面一个根本问题是：计算机虽然是强有力的工具，但将它用得好并不容易。要使计算机能正确地运行，必须按严格的规则，应用其基本词汇（指令或语句等）编写程序。而这种基本词汇中包含了许多细节的规定，所以程序很难编、难读、易错、难改。因此，编写程序的效率很低，程序规模越大，这个问题越严重。[BF]中指出：“那些二十多年前推动着 Fortran 的力量，至今仍然是推动自动程序设计（AP）系统的实用的理由。程序设计员仍然背负沉重负担：必须描述许多细节，必须跟踪这些细节之间的许多关系，而且必须使用那种与他们解题时思维过程很不协调的程序环境。人们相信，新的程序环境可能是对这些问题的解决途

径，而且，这种程序环境所要求的软件技术已经成熟。”

但是三十年来，硬件及软件技术毕竟有了很大的变化。与这种变化相应，程序技术的发展大体可以分为三个时期：六十年代为高级语言时期，七十年代为结构程序设计时期，而八十年代则可以称之为自动程序设计时期。本文旨在说明这三个时期的一些主要情况和基本特征以及其间的联系。从而，进一步说明技术发展的内在规律及今后的动向。这一发展趋势，借用《三国演义》中的一句话来概括，即“分久必合”。具体说，继七十年代各种软件技术与理论以分散的状态蓬勃发展之后，目前，在程序设计乃至整个软件领域内，呈现出如下动向，即各种技术与理论的集成化，各种方法论的相互联系，各个技术领域的相互渗透，以及程序范型（program paradigm）及机器体系由过去的单一性，经过近年出现的分裂局面，将在更高层次上重新趋于统一。

XYZ 系统是本文作者过去长期研制的一种软件工程环境[T2-T3]。它综合了我个人对程序技术与理论各个方面的研究工作，并全面体现了我对上述技术发展趋势的理解和认识。用它可具体说明如何剪裁各种技术与理论以组成一个可用的软件开发支撑系统而符合上述发展趋势的要求。因此，作为一个模型，我们在本文最后一节对这系统作简要的介绍。我们希望，这样做，既有助于具体说明程序技术发展趋势，也有助于使人们了解 XYZ 系统的设计思想。盼得同行指正。

§2　前二十年的历史回顾

六十年代是程序技术发展的初始阶段。宽行打印装置为高级语言提供了必要的物质条件。Backus 将 Fortran 的编译程序称之为“自动程序设计”[BC]。此后十年中，设计和实现在控制结构及数据结构方面表示力强的高级语言乃成为程序自动化水平的标志，这也就是当时程序技术研究所追求的主要目标。继 Fortran 之后，针对不同的应用领域出现了 Algol 60，Cobol，Lisp，Jvioal 等等高级语言，最后到六十年代后期由 PL/1 与 Algol 68 对这一时期的语言特征作了一次总结。这一时期技术思潮的哲学概括即 Perlis 所作的第一次 Turing 报告。

这一时期中，可以说编译技术代表了整个软件技术。在开始很长一段时间内，编译系统主要靠手工编制，自动化程度很低。1960 年 K. Samelson 与 F. L. Bauer 提出用先进后出栈的技术实现表达式翻译，才在自动化翻译方面走出了重要的第一步。而更为重要的突破却是 1963 年 R. W. Floyd 关于优先算子文法的工作。他将高级语言中控制结构也看成算子，引入优先顺序概念，从而利用先进后出栈也可全面实现高级语言的语法分解。而这一工作，却是建立在形式化理论的基础之上的。

形式文法理论建立早在 1956 年。当时 Chomsky 研究语言的形式模型主要是针对自然语言的。程序领域对形式理论发生兴趣则在六十年代 Algol 60 问世之后。一方面经过严格推敲制定的 Algol 60 文本中发现了条件语句的二义性，另一方面，更为重要的是，许多严格按照 Algol 60 文本要求而编制的编译系统其运行结果都不一样。其原因是 Algol 60 文本是用自然语言书写的。虽然由 Naur 教授执笔的该文本以其精

确性著称，但仍然可作不同解释。从而提出了语义形式化的要求。所以，当时形式语义的目标主要在语义精确化。Backus在1959年提出了一种描述高级语言语法和语义的方法（即BNF）。原来认为是一种描述形式语义的元语言，后来得知它仅等价于Chomsky模型中的上下文无关文法。因程序语言中必须包括某些与上下文相关的部分，故BNF只能精确地用来描述语法（不包括语法限制）。因此，语义精确化的目标并未达到。但是，由于有了这种表示形式文法的工具，Floyd才得以建立优先算子文法。从而实现语法分解的自动化。虽然这还不是全部高级语言编译自动化，但它已可代替掉语法分解和查错等大量的手工劳动，得到了令人惊异的效果。从此，确立了形式化理论在程序技术领域的地位。并且，使编译自动化成为人们研究形式语义的另一个重要目标。

七十年代是计算技术五光十色蓬勃发展的时代。可以说是它的壮年时期。由于磁盘的问世，操作系统迅速发展起来；再加上商业数据处理等非数字应用的发展，数据库成为一个独立发展的领域；通讯设备的完善，又促成了计算机网的发展。总之，硬件的进步及应用领域的拓广，使计算机软件领域分裂成许多新的独立发展的技术领域。这一时期中，大规模集成电路也飞速发展，硬件造价下降很快，机器规模日益庞大而结构也越来越复杂；应用范围的拓广使软件系统规模增大。再加上并行性成为普遍采用的提高效率的形式，又更大程度上增加了软件的复杂性。所有这些情况汇聚在一起，产生了一种现象：即软件可靠性差。使许多耗资巨大的软件项目中途停止，甚至由于软件的错误导致巨大的经济损失。这就是所谓软件危机。在这样的背景下，致使一些计算机科学家回头从程序设计的基础到软件生产过程对各个方面提出了质疑。他们认为：（1）自高级语言出现以来片面地追求语言的表示力的思潮是不合适的，程序正确性应该比程序语言的表示力更为重要。由此引出以正确性为目标的关于各种语言成分的大量研究。（2）程序正确性的根本问题是程序设计方法论。应找到一种能保证程序员编制出可靠而有效的程序的方法。（3）软件生产管理是保证软件可靠性的另一重要条件。从人员组织到程序编制过程中的信息管理对于保证程序可靠及可重用都紧密相关，应按工程的准则来予以管理。关于上述这些方面的研究形成一个新的领域，即软件工程。

西欧和美国的计算机科学家提出这些思想以后，很快即形成一股巨大的思潮，这就是结构程序设计。这一时期思想最有代表性的总结即Dijkstra的Turing报告。[TH]

（1）软件生产管理 在软件生产管理方面，虽在美日等国进行过许多认真的试验，如主程序组等，但到现在被普遍承认的成功的组织管理经验似乎还未见到。不过，作为一种工程，软件生产需要某种合理的管理体制大概是无人反对的。到八十年代以后，人们更关心的是如何在更大程度上提高软件生产的自动化水平，由自动生成系统生产出的软件，其可靠性往往是可信赖的。

程序编制过程中信息管理对提高程序可重用性、软件开发的规范化及检查错误等方面的确是非常重要的。七十年代人们以严格规定的程序报表进行信息管理往往过于烦琐，非常耗费程序员的时间与精力，其得失很难说清。这种方式的信息管理在日本进行过较认真的试验，而在美国与西欧并不被普遍接受。如何使这部分工作用计算机来承担则是八十年代软件工程界普遍感兴趣的问题，它与软件开发方法的

研究紧密相关，成为软件开发环境的一种新的方向。它是 CASE (Computer Aided Software Engineering) 研究的一个重要方面。

(2) 程序设计方法论　程序设计方法论的研究在这时期中取得了很大的进展。其主要成果有以下方面：

(a) 由顶向下方法　这方法最先由 Dijkstra 等提出，由 Wirth 具体化为逐步求精的方法。即在设计一程序时，首先定出一最概括性的目标，然后再逐步将此目标往下分解成子目标并予以具体化，最后得到可执行的程序。这一方法更进一步演变便出现了以下两类方法：

• 非形式方法：即以表格，图形，或其它便于直觉理解的手段，以不很精确的方式表示出每一步的设计。它可以是纸上的，也可以借助于计算机。如 HIPO，SADT，以及后来更为流行的 PDL，DFD（数据流分析），Yourdan 方法，Jackson 方法等。这些方法的优点是直觉、容易掌握，且便于反复修正，多广泛流行于工业界。其缺点则是形式化程度低，因而不精确且自动化程度很低。

• 形式化方法：即以逻辑或代数等数学语言，表示出程序的抽象描述，然后以形式转换或形式分划和验证的方式最后得到一可执行的算法程序。这方法的优点是精确，其中许多部分由于形式化故可自动生成。但缺点是要求形式化训练较高，非一般用户所能掌握，故多流行于学术界。而由这方法带动的关于抽象描述语言，形式转换以及程序验证的研究长期以来一直是计算机科学的重要研究课题。

(b) 由底向上的方法　这方法强调程序设计的模块化方法。它可使程序错误及修改所引起的副作用局部化，且可加强程序的可重用性。事实上，自 Fortran 及 Algol 60 起，子程式，分程序及过程等都是程序模块的很成功的形式。七十年代以来 Parnas 对程序模块进行过长期研究，得出了分划模块一些基本准则。而最成功的程序模块概念则是由 Dahl 等在 Simula 中开始提出后来又被 Liskov 加以发展的抽象数据类型。它将数据结构看作是由可施于其上的运算所规定的对象域，由此而组成模块。模块中的变量及运算可移出模块之外为别的模块所引用，同时它也可移入别的模块中所定义的变量和运算。这种数据模块还可以根据数据的包含关系而具有层次结构。外层模块上定义的运算可自动地被内层模块所引用。程序即由这种具有层次结构的模块所组成。这种由模块由底向上组成程序的方法越来越受人重视。

事实上，不论由底向上的或由顶向下的方法，非形式的方法或形式的方法，都各有优点和不足。近年来已有越来越多的人认识到，最合理的途径应该是所有这些方法的结合。下面将要讨论这个问题。

结构程序设计思想的提出，人们多归之于 1968 年 Dijkstra 写给 CACM 的信。信中指出转 (GOTO) 语句的危害性。从此以后，掀起了一个关于高级语言各种控制结构与数据结构进行分析和批判的高潮。同年，Dijkstra 等拒绝 Wijngaarden 提出的 Algol 68 的方案也出于同一种思想。他们认为高级语言的设计首先应保证程序正确。为此，语言应设计得简明，适于阅读。因此，它应使程序具有简明的控制结构与数据结构。关于程序验证的研究表明，简明的程序结构同时也就是便于进行验证的程序结构。[T1] 中曾对结构化语言的讨论作了详细的介绍。在结构程序设计思想流行的十年中，曾出现了许多设计思想与六十年代语言不同的结构化语言，如具有结

构化数据结构与控制结构的 Pascal，具有优良模块结构的 Simula，Modula，适于进行程序验证的 Euler，Alphard，面向机器以提高执行效率的结构化语言 Bliss 等。这一潮流一直发展到 Ada，它对结构化语言作了一次全面的总结。全世界几千名计算机科学家在不同程度上卷入到关于 Ada 的讨论。从 Fortran 开始，二十多年来命令式高级语言的潮流发展到 Ada，其主流可以说已告一段落，其后即使出现一些变化也只能是枝节性的了。

(3) 语义形式化 与程序语言的发展有关，还有语义形式化的研究。前面已指出，它是六十年代提出的老问题。语义精确化及编译自动化这两个目标对七十年代结构程序设计的追求仍然吻合，因此得到了推动而有很大的发展。主要方法有下面几种。其中每一种语义形式化方法都有其适应的方面，也有其不适应的方面，没有一种能说是全面解决问题的方案[c]。

(a) 静态语义　在语义精确化方面最有影响的方法是指称语义方法。它由 D. Scott 等所创立。它的最大优点是具有逻辑学中语义形式化的那种结构性。也就是一对象的语义可由其组成成分的语义构造而成。比如“条件语句”，其语法是：

〈条件语句〉:: =if〈条件〉then〈语句〉$_1$ else〈语句〉$_2$

右边〈条件〉，〈语句〉$_1$，〈语句〉$_2$ 即构成了〈条件语句〉的组成成分，其语义，即可写成：

$$\text{【〈条件语句〉】} = \begin{cases} \text{【〈语句〉}_1\text{】 如果【条件】} = t \\ \text{【〈语句〉}_2\text{】 否则} \end{cases}$$

这种方法的优点是 (1) 结构精美，便于阅读和分解 (2) 可避免许多细节。但有如下两方面的缺点：(1) 对某些控制结构，如并发进程，很难用此法描写其语义 (2) 这种描述方法是解释性的，而实际编译系统是将高级语句翻译成中间语言，所以是翻译性的。因此，以此法描述的形式语义为基础，实在难以自动生成与传统编译系统相应的语义导引的编译系统。

操作语义采用另外的方法来描述语义，设想一抽象机，它表示出当前状态下一步操作及已得到的中间结果；然后将程序执行过程描述为这抽象机上的状态转换。早期操作语义的工作 VDL 包含细节太多不能应用。后来 Edinburgh 的 Plotkin 提出一改进方案，它形式上与指称语义很相近似，同样具有结构性，但所描述的语义是程序在抽象机上执行过程的语义。所以它既具有指称语义的优点，但同时又能方便地描述并发进程语义，故避免了指称语义的一个重要弱点。可是，这个方法同样是解释性的。作为语义精确化的工具虽确具有优越性，但从编译自动化方面来要求则嫌不足。

第三种语义形式化方法是属性文法。这文法是 Knuth 创立经长期研究使之实用化的。它是目前已知的语义形式化方法中唯一面向编译自动化的。可以说它是对上下文无关文法的扩充使之处理与上下文相关的成分的语义。其作法是将实际编译方法中名字表操作形式化。在名字表中每一名字上附有许多属性，随着语法分解（或综合）的进行而对这些属性进行变换。比如，变量的“类型”及“值”都是属性。这方法在每一语法公式之后附上一些属性计算公式以表示相应的属性变换。Kasteu 找到了一种进行属性计算的有效方法从而使这方法实用化。本方法的优点是它与实

际编译系统的名字表操作对应，故可将这部分语义加工过程形式表示出来。但也有其不足之处，首先，它所表示的这部分语义通常是在编译时可处理的那部分语义，也就是静态语义，而对程序执行过程中所具有的那部分语义它并不适于表示。所以它不是动态语义形式化方法。因此，它也没有起语义精确化的作用。此外，这方法表示形式也不具有结构性，在可读性方面对于不熟悉编译过程的用户是较差的。下面将要讨论，如何将这方法与另外具有翻译性的动态语义方法结合起来的问题。

(b) 动态语义　此外还有几种重要的语义形式化方法，它们有些是面向某种语言结构，具有特长的：如面向数据模块语义的代数语义及面向并发进程语义的时序逻辑，此外还有面向程序验证的公理化语义。

代数语义是面向抽象数据类型这种数据模块的形式语义方法。事实上 Simula 提出这种模块概念的基本思想是以施于数据域上的运算集来表示数据域的涵义，这思想本身即具有代数的意义。只要再进一步追问这些运算本身的涵义，即必然导致它们所应满足的某些关系，也就是一组公理。比如，整数域即可定义为满足 Peano 公理的那些运算所施加的对象集。接着即产生满足形式公理的模型问题。用这样的途径，的确能将各种基本数据结构及其上所可施加的运算的各种性质很精确地刻划出来。对于这些基本数据模块说将其语义用这样的方法来表示无疑是很有意义的，甚至是不可避免的。将程序的静态语义检查建立在这种语义之上既精确、有效，也很自然。但这个方法有一较大的问题：如果将这方法推广到一般用户模块，要求用户对他们所定义的模块也都写出其代数公理，则将遇到很大的困难。这恐怕是这种语义方法长期难以推广的主要原因之一。正如前面已经谈过，我们认为由底向上的方法与由顶向下的方法最好能结合起来进行。用户则以由顶向下的方法进行程序设计，直到分解到一定程度，其抽象描述可用基本数据结构上的运算来表示时，再转向系统提供的这些运算，并论证其间的一致性。这样将两类语义结合起来，也许是解决困难的较为妥当的途径（见§4）。

公理化语义也就是 Hoare 逻辑。它的直接目标是为了进行程序正确性验证。事实上，它是过程语义的扩充。所谓过程语义是指用谓词演算断言所表示的前置条件及后续条件来描述过程的功能。Hoare 将这种语义推广到语句之上，从而变成了验证正确性的规则。这种语义比指称语义要弱一些，只能在较弱的意义下，表示出语言的精确涵义。但是，它是以逻辑表示语义的很有意义的方法。下一节将谈到的抽象描述语言多以这种语义方法为基础。这种语义显然与编译自动化没有直接联系。

时序逻辑也是一种逻辑系统，用它表示程序语言的语义当然也是一种上述公理化语义的方法。但用这种方法去刻划并发进程的安全性及活性等却具有独到的优越性。Manna 与 Pnueli 等在这方面进行了令人信服的工作。XYZ 系统采用这种理论设计了一种时序逻辑语言 XYZ/E，它不但能用来对并发或串行程序的抽象功能进行描述，而且可用它表示具体算法和知识。用它可将各种程序范型表示在一统一的框架之中。就表示高级语言的动态语义而言，XYZ/E 可以起中间语言的作用。在 XYZ/CCSS 系统中，用一种具有结构性的元语言规则表示出如何将高级语言的每一种成份归约成为 XYZ/E 的程序。这种归约式与每一语法公式相应，它既起语义定义作用又起语义子程序的作用。这样表示高级语言的动态语义是翻译性的而不是解释性的。

故它与实际的编译是紧密对应的。由于 XYZ/E 作为逻辑语言，其语义是形式化的，因而这种方法表示的动态语义也是形式化的。不过，这只是表示了高级语言的动态语义，再将这部分语义与属性文法表示的静态语义结合起来，即构成一完整的建立在形式语义基础上的编译生成系统。下面§4 将要介绍。

综上所述，各种语义理论均只具有某些方面的适应性。作为理论研究，这些理论都具有独立存在的价值。但欲以这些理论为基础构造工程系统，则不能拘泥于一种理论，否则难免片面性。近年来在形式语义理论的基础上构造编译生成系统或在软件工程环境中，确定抽象描述语言时，为了具有全面的适应性，均采用将几种语义方法集成为统一框架以使各自发挥其所长的途径。下面将要说明这些情况。

§3　八十年代的发展趋势

八十年代由于集成电路的突飞猛进以及高分辨率终端的出现，为个人式微型计算机的大发展提供了条件。由七十年代开始的软件工程技术日趋成熟，再加上人工智能专家系统研究的进展，使程序技术进入成年时期。这一时期软件领域呈现的总趋势即是§1 中所指出的“由分走向合”。以下分几个方面来说明。

(1) **软件工程环境**　个人式微型计算机与软件工程相结合出现了软件开发环境。它是由过去分散编制的软件开发支撑工具，集成为整体性的系统，通过终端以交互的方式给用户以友好的使用界面。这种系统最早是由 Xerox 公司集中了许多专家共同努力创造出来的。很快即推广发展起来。早期的环境称为程序设计环境，它是围绕一种语言，将关于这语言的程序开发工具，如编辑程序、编译程序、调试程序及维护程序等集成组织而成。如 Lisp 环境、Cornell 程序综合器及 Ada 环境等。更新的环境则超出了这一范畴，称为软件工程环境，也有人称之为 CASE。它是围绕程序开发的各个环节（整个生命周期），基于一种协调的方法论，具备各种层次的表示工具且包含开发信息的计算机自动管理的巨型系统[W]。比如 Green、Balzer、Cheatham、Luckham、Rich 等合作的 KBSA，其主要特点是程序开发各个环节，在形式框架内再插入各种专家知识。又如日、美近十所大学，由美国的 Riddle 及日本的 Kishida、Seito 等领导的 SDA 系统，其特点是以模型为基础将各种分散开发的环境统一在一协调的系统之中。而西德近十所大学开发的，由 Krieg-Brückner、Broy、Gansinger 等领导的 Prospectra 系统则是以代数语义及程序形式转换为基础的环境①。这种系统往往将各种理论与技术集成在一统一的框架之中。这是二十多年来自动程序设计研究所达到的最高形式。它们的出现标志着程序理论与技术已进入成熟阶段。当然，这类系统可以组成各种形式，也将继续改进和发展。但总的讲，这类系统可说是二十多年来，程序理论与技术发展的一种总结性的工作。XYZ 系统也是属于这一潮流中的一次尝试。而在这类系统中抽象描述与程序转换往往居于核心地位。下

① KBSA 是其主要设计人在 Kestral、南加州大学、Harvard、Stanford 及 MIT 长期研制的几个系统的综合性总结，Prospectra 则是由 Bauer 的 CIP 系统发展而成。都是二十年来软件工程方面最有代表性的工作。

面着重谈谈这方面的情况。

（2）**抽象描述与程序转换** 程序技术发展过程中一个贯串始终的目标和趋势即如何在更大程度上以“做什么”来代替“怎么做”。这是自动化程序提高的标志。事实上，一计算机系统可以看成是一个由一序列语言组成的系统。最高层是最抽象的描述语言，最低层为线路语言。其中上方的各层是由软件用“翻译”的办法实现从上层到下层的转换直到机器指令，下方的各层从指令语言开始到线路语言则由硬件实现一层到下一层的“解释性”的转换。机器设计的进步可使指令系统越来越抽象以掩盖掉更多的算法细节（比如用浮点指令代替定点指令）。软件的进步则使用户使用的语言越来越抽象，也是为了掩盖掉程序中的算法细节（比如用高级语言代替汇编语言）。从七十年代以来，大家认识到高级语言程序中仍包含了大量算法细节。随着程序规模的增大及其结构的日趋复杂，高级语言中所包含的算法细节已到了使软件难读，易错，难查，难改的程度。因此，为了保证程序正确，为了提高程序的可重用性，也为了用户有更大的自由在以后选择更有效的算法，人们越来越认识到应该用更抽象的抽象描述语言以代替算法语言书写程序。而使由抽象描述到算法的转换部分地或全部地交给系统软件去实现。这样自然就大大提高软件自动化水平，提高软件可读、可写、可重用、可维护、易保证正确性等方面的质量。这也是结构程序设计时期提出各种方法论的基本前提之一。美国国防部 Stars 计划中，由 Balzer、Cheatham 及 Green 提出的“革命性”措施，即是以抽象描述为基础的。关于抽象描述语言的研究到八十年代乃形成高潮[BF]。

怎么构造抽象描述语言？除了像 Ada 中“Package”那样直接从高级语言出发，将其说明部分与实现部分分离的办法抽取出较为“低级”的抽象描述以外，绝大多数抽象描述语言都是建立在形式语义理论之上的。除了前面所述的几种语义理论外，当然也还有直接借鉴逻辑与代数的。事实上，也无非是基于上述各种形式语义理论的某种变型而已。著名的抽象描述语言有如下一些。

基于指称语义的抽象描述语言如 Meta Ⅳ。

基于代数语义的抽象描述语言如 OBJ，Larch。

基于属性文法的抽象描述语言如 ALANDIN。

基于时序逻辑的抽象描述语言如 Tempura，XYZ/E。

基于一阶谓词演算的抽象描述语言如 Prolog。

基于集合论的抽象描述语言如 Z. SETL。

基于范畴论的抽象描述语言如 Clear。

基于 lambda 演算及可计算函数理论的函数式语言也可以看成是一种抽象描述语言如 ML，FP，SASL。

早期的抽象描述语言多半基于一种理论，但随着实际软件工程环境的实现，它们往往脱离其单一的理论基础而与其它相关理论结合起来以补其不足。如前面所述 XYZ/E 虽以时序逻辑为基础，但在构造 XYZ/CCSS 的编译生成系统时，它便引入属性文法以表示静态语义；在构造 XYZ/DSPV 系统时，以时序逻辑语言表示由顶向下方法中逐步分解的过程语义，而以类似于 Larch 的方法表示由底向上组成的操作的静态语义，而将两方面结合起来。事实上，国际上许多著名系统也都去采用集成不

同语义理论的方法。如MetaⅣ正在考虑采用其它语义方法表示并发性，Prospectra系统中则采用代数语义、指称语义与谓词演算相结合的途径。

在抽象描述确定之后，即应考虑这种程序如何实现的问题。也就是如何找到一种方法，在不改变其语义的情况下将它转换成一可在计算机上有效执行的问题，这就是程序转换问题。到目前为止，不外以下几种途径。[PS]

(a) *自动转换* 即希望找出一种形式方法，可将所给抽象描述转换成可有效执行的算法程序。如慕尼黑工业大学Bauer小组所研制的CIP系统。经过十多年努力以后，虽对于许多有意义的特殊情形（如尾递归），他们找到了可以从抽象描述生成有效算法程序的方法，但在一般情况下，已感到找出自动转换方法的希望甚微。看来这一条较理想的途径事实上只能起部分的作用。当然，目前仍有人（如CMU的Scherlis）对这条道路抱很大的信心。Scherlis称其方法为“演绎程序设计”。

(b) *自动综合* 即采用定理证明的技术得出从抽象描述转换成算法程序的方法。如Manna与Wallinger所研制的Dedalus系统等。他们从程序正确性证明研究中找到了一些程序设计一般性规则。这些规则对于设计算法程序的确很有意义。可是，距离实现全面的程序综合的目标还非常遥远。

(c) *直觉主义逻辑* 比程序自动综合前进一步，Martiu-Lof的学说要求抽象描述按直觉主义逻辑的要求来表示。比如“ExA(x)”，按他们的要求即为“对于某个体c，可构造出A(c)”。关于“ExA(x)”的一个证明，也就是一个对偶〈c，d〉，此处d即关于A(c)的一个证明。而这样的c，事实上，并非毫无约束的，它必属于某一个域C（即类型）。故“关于ExA(x)的证明集”即可表示成（Σc∈C）P(A(c))，此处P(A(c))表示关于A(c)的证明的集合。以这样的意义去解释抽象描述，当对于一给定的抽象描述能给出一个证明，则按照Constable的方法即可根据这证明机械地生成出与此描述相应的算法程序。从自动转换的要求来说的确目的达到了。不过，问题的困难转换到描述或其证明本身了。显然，这样的描述或证明不是容易给出的。不过，这一理论的确可以给我们很多启示。最重要的一点是，如果我们对进行描述的语言作出适当的限制，则生成相应的算法程序的目标即可变得较易达到。这里就有一个两方面得失权衡的问题。因此，可以提出这样的问题：是否能够找到某种办法，使得在描述方面的要求不致高到用户难以使用或验证，而去求出相应算法程序方面的难度仍是可以保持在并非高不可及的程度上？这是一个值得探索的方向。在XYZ中我们容许在抽象描述中除使用时序逻辑外还可引用递归函数，它们在描述的附加部分（Where Part）予以定义。因为“ExA(x, y)”引用Skolem函数，即可转换成“A(f(y), y)”如此处“f(y)”为递归函数，则这样的存在量词即是构造性的了[TU]。

(d) *人工智能方法* 在完全自动化机械生成的途径难以找到的情况下，自不妨在某些部分采用某种专家系统进行转换。Fickas在Balzer系统中采用某些专家系统的方法得到相当的成功[F]。在已经建立较完整的形式框架的基础上采用专家知识进行某些转换，是一值得探索的途径。但这种途径毕竟只能起辅助作用，还不能认为已可以依靠来解决全部问题。

(e) *手工分解和验证* 在其它方法均失效的情况下，这是总可采取的一种方

法。即由程序设计者手工对所给的形式描述进行分解。使上一层次的抽象描述分解成下一层次的抽象描述，如人工智能解题过程中的 hierarchical planning[BF]。但为了保证这种分解的正确性，应在每次分解后进行验证。如此逐步进行直到最后得到一可有效执行的程序。这一方法实质上是逐步求精方法在形式化由顶向下方法中的应用。事实上，在非形式化的情形下，这种方法是较易运用的。而对于形式抽象描述，作这种分解就不是那么易行了。这个问题与如何从非形式方法入手形成形式抽象描述具有类似的性质。这个问题事实上就是 Balzer 的 SAFE 系统所着重要解决的描述获取（Specification Acquisition）问题[BF]。这里主要需要找到一种如何从非形式的不精确的描述逐步向形式化的和/或精确的描述过渡的方法和表示工具。我们在 XYZ 系统中采用了一种与 PDL 类似的 XYZ/SDL（Specification Design Language）语言来表示这种过渡，并以一种面向这种表示手段的语法制导的编辑系统构成交互式用户界面。事实上，以这种手工分解方法为基础，还可以结合上面几种方法的某些技术，使得分解过程中可更多地得到计算机的辅助而提高其自动化水平。而且在分解过程中，应随时对照可重用程序库中抽象数据类型模块所提供的运算，当分解过程中形成的抽象描述可由这些由底向上提供的运算来合成时，分解即告终止。看来，程序自动转换技术目前所达到的水平也就只能做到这一步了。更进一步的突破似乎十分困难。人的创造在转换过程中始终起重要的作用。

与上述从抽象描述出发进行程序设计的方法相近似，还有快速原型示范（Rapid Prototyping）。因为研制大型软件系统时，最好在算法实现以前的设计阶段就对方案进行评判。这时对系统进行修改比在实现后进行修改要经济得多。为了做到这一点，即希望在设计阶段先对方案实现一原型，然后执行这原型，看它是否符合用户的需求，符合时再进行算法细节的设计。因此，这样的原型与抽象描述相似，可忽略掉设计中许多细节而只要求突出地表示出系统的主要功能。但它与抽象描述不同，在于在示范时原型必须能直接执行，但不求执行效率。Prolog 在作为通用原型示范语言是有其优越性的。它可简洁地表示出系统的主要功能，而且可以执行。不过，原型示范系统要能起工程蓝图的作用，则应该可根据它实现出详细的算法程序。但遗憾的是，包括 Prolog 在内现存的原型示范系统都不具备这种优点。我们在 XYZ 系统中则在一定条件下可以做到这一点。即在一定条件下 XYZ 系统的抽象描述可作为 Prolog 程序来执行。当执行结果满足用户要求时，又可对抽象描述进行逐步分解和验证，最后得到有效的算法。在这种情况下，抽象描述、原型示范及相应的有效算法程序三方面相互联系起来了。

(3) 程序范型及机器体系的分裂与统一　近十年来，由于抽象描述研究的进展产生出一些十分有意义的描述语言，如逻辑程序语言，Prolog，函数式语言 ML 等。它们在某些应用领域具有传统的命令式高级语言所不具备的优越性，因而受到相当广泛的重视。可是，传统的命令式高级语言有一些特长是这些语言所不具备的。两者有完全不同的逻辑基础。它们是本质上不同的程序范型（Programming Paradigms）[FI]。从 Fortran 以来，语言变化虽然也很大，但从 Fortran 到 Ada 并未脱离命令式高级语言这样一种程序范型。因此，在程序领域内，范型是统一的。但自逻辑程序设计语言及函数式语言出现之后，范型统一的局面被打破了，出现了分裂的形

势。事实上，各种命令式高级语言的一个共同的特征，即它们都是 von Neumann 型计算机体系的反映。由于 von Neumann 机器体系长期以来一直占统治地位，故命令式高级语言也相应地一统了程序范型领域近二十余年。自逻辑程序设计语言及函数式语言出现之后，人们也随之对反映这类语言特征的机器体系产生了兴趣，特别是在函数式语言机器（包括归约型机器，数据流机器）方面，近年来取得了颇大的进展。因此 von Neumann 型机器体系一统天下的局面也随之被打破，在计算机体系领域也出现了分裂的形势[BC]。

虽然分裂局面已经形成，但明显的事实是没有哪一种范型具有足够全面的优越性而能取代其它的范型；同样，也不大可能有哪一种机器体系可取代其余的机器体系。何况，加上习惯势力形成的惰性以及长期投资的大量软件的存在，传统的命令式语言及 von Neumann 型机器仍将长期存在。因此，在可预见的将来，不同范型与机器体系并存的局面不可避免。

但范型及机器体系的分裂将给使用及教育带来极大的不便。比如以 Prolog 表示的知识及以 Ada 表示的算法对于某些用户说来很可能需要穿插在一个软件系统中来运行。因此，很自然地产生如何将这些范型和体系统一协调起来的问题。

从范型统一问题来说，过去在命令式高级语言中出现过两次统一化的努力。六十年代末期，由于将 Fortran，Algol 60，Cobol 及符号处理语言统一起来，曾出现过 PL/1 及 Algol 68；七十年代末期，由于将 Pascal，Simula 及结构化语言统一起来，曾出现过 Ada 与 Chill。这两次统一均有一共同特点，即构造一大型语言（Omnibus Language）将被统一的语言的特征表示在大语言之中。当时之所以能这么做是因为这些语言都是命令式高级语言，其逻辑基础是相同的。所以可以用这种方式由上而下的予以统一化。不过，也都付出一个重要代价，即得到的语言规模很大，用户极感不便。这一途径对于当前面临的统一化问题显然不能适应了。

目前已知的统一化办法有以下几种：

（a）从技术上，不从概念上，设法使不同范型的程序可以相互协调运行和通讯。典型的例子为 Loops。它用不同的 Smalltalk 式的数据模块表示 Lisp 程序和产生式系统，然后通过 message passing 的办法进行模块之间的信息传输。从技术上说，这不失为一可用的办法，但并没有真正解决范型统一问题。

（b）扩充逻辑程序设计使之表示算法程序。目前已存在许多这类 Prolog 的变型系统。但有一个共同的缺点，即在 Prolog 基础上实现的算法程序往往运行效率很低，无法与命令式高级语言的效率相匹配。

（c）由下而上为各种范型给出一新的语义基础，并要求各类语言在这共同基础上仍保留原有的效率。XYZ 与 U. T. Austin 的 Chandy 与 Misra 共同设计的 UNITY 均采纳这一途径，XYZ 的共同基础是 XYZ/E 时序逻辑语言，而 UNITY 的基础为由监督命令、赋值语句及循环不变式组成的一个系统。在这种统一框架中原有的各种语言仍保留原有的面貌，只是其语义基础是由统一的方式予以解释和实现。

与此相应，我们对机器体系的统一也有如下的设想：将这种统一的语义语言用作各种机器的核心语言，用这种核心语言既可表示出 von Neumann 型机器，也可以表示出函数式机器或逻辑型机器等。每一种这样的机器，事实上都可当作是一种功

能部件。既可单独使用，也可根据用户的需要，选择若干种这类功能部件连接成为一台整体机器。这种连接可以根据统一范型的要求以分布式方式组成网状。由 XYZ 系统提供的各种语言间转换规则，即构成了机器内各部分联系的结构，由于 XYZ/E 中包含了不同层次的语言，也可以包含用户所习惯的任何一种或几种语言作为使用语言，由 XYZ 系统所说明的各种语言的关系，即构成了用户程序中使用各种语言之间的语义关系。由此组成用户界面。这样的机器完全可以在当前技术达到的水平内实现。在 XYZ/E 基础上构造的这种机器体系，我们称之为 **XYZ** 机器，它是一种逻辑机，但它可有效地执行算法程序、函数式程序，当然，更适合进行逻辑程序设计[TD]。

(4) 各种技术领域的相互渗透和融合　前面已指出，七十年代形成了许多新的独立发展的技术领域。八十年代以来，这些领域之间又出现了互相渗透与融合的现象。

(a) 在数据库领域，前一阶段在程序领域中进行研究的问题后来在数据库领域内也在进行讨论。只是所用的词汇往往不同而已。这里主要讨论的仍然是两方面的问题，一个问题是数据库的“逻辑性质”问题，比如完整性限制及演绎数据库。这些问题显然与程序正确性验证及其它程序性质（如安全性、活性等）的证明相对应。两方面所根据的逻辑工具也是十分接近的。另一个问题是数据模式的“语义”问题，这个问题与程序语义形式化及抽象描述的讨论极为近似，所引用的形式化理论几乎是相同的[FM]。

(b) 数据库、人工智能及程序语言三个领域互相渗透的现象十分明显。近年关于“概念模式”的研究即说明了三者关系[8M5]。

(c) 集成电路的发展使软硬件之间的界限日益模糊。这种界限不过在不同层上以不同方式表示算法而已[M]。

关于这些相邻领域的讨论已超出本文的范围。估计这些相邻领域互相渗透的现象将进一步发展，它们将形成十分活跃的研究领域。

§4　一个实例：XYZ 系统

XYZ 系统是一软件工程环境，其基础为一时序逻辑语言 XYZ/E。针对不同的程序设计方式，有不同的环境来实现与那种程序方式相应的方法论中所要求的程序转换[TD]。

XYZ/E 是一多种类线性时间的时序逻辑系统又是一种程序语言。在这种程序语言中包含一种统一的程序框架，用它可以表示低级的或高级的算法程序，（串行的，并发的，确定性的或不确定的），还可以表示抽象描述或产生式系统。这种程序的统一框架具有如下形式：

$$\Box\ [A_1;\ \cdots;\ A_n]\ \text{where}\ \ B_1 \wedge \cdots \wedge B_k \qquad [\text{I}]$$

此处“□”表示所有时刻，“;”表示合取，$A_i (i = 1, \cdots, n)$ 为一种称为“条件元”的基本逻辑语句单元，$B_j (j = 1, \cdots, k)$ 为由等式或不等式组成的合式公式，

用它定义程序中出现的递归函数，或约束条件。这种由 where 引出的部分称为“where part”，它是程序的附加部分，可出现在程序，或条件元之后，也可出现在一组由方括号括起的条件元之后以限定它的作用域。

条件元恒具有如下的形式：

$$lb = y \wedge P \Rightarrow @ (Q \wedge lb = z) \qquad [\text{II}]$$

此处“lb”是一特殊的控制变量，它恒指向一个标号（如［Ⅱ］中 y，z），出现在条件元“⇒”左边的“lb 等式”中的标号称为定义性标号（如［Ⅱ］中 y），“⇒”右边的“lb 等式”中的标号称为转出标号（如［Ⅱ］中 z），这种 lb 等式又称为控制等式。“@”可以为“◇”（终将）或“o”（下一时刻），为时序逻辑算子。P，Q 为两合式公式，P 称为“条件”，Q 称为“动作”。由于 P，Q 的限制不同，它可以表示低级的算法运算，也可表示抽象描述。当 P 为通常高级语言中的条件时，Q 为一组如下形式的公式的合取式“ov = exp”，（此处 o 为下一时刻算子，v 为一时序变量，exp 为一表达式，其中不出现下一时刻算子，这种等式称为赋值等式），@限为下一时刻算子，则为低级形式，这样的条件元可组成算法程序。非低级形式的条件元（即 P，Q 不受上述限制），则用来表示抽象描述。为了使一程序合理，还有一些限制，本文不一一列举，其中之一是一程序中，定义性标号相同的条件元中的条件组成的析取式必恒等于 **T**（恒真）。

如一程序中所有条件元的定义性标号与转出标号均相同，则这样的程序［Ⅰ］即为产生式。如果一产生式中只包含 Horn 子句，［Ⅰ］即构成为一 Prolog 型程序。

为了表示不确定性及并发性，我们将［Ⅱ］扩充为如下的形式：

$$lb = y \wedge P \Rightarrow @ ((Q_1 \wedge lb_1 = z_1) \wedge (Q_2 \wedge lb_2 = z_2)) \qquad [\text{III}_1]$$

$$lb = y \wedge P \Rightarrow @ (Q_1 \wedge lb_1 = z_1) \wedge @ (Q_2 \wedge lb_2 = z_2) \qquad [\text{III}_2]$$

$$lb = y \wedge P \Rightarrow @ (Q_1 \wedge lb_1 = z_1) \vee @ (Q_2 \wedge lb_2 = z_2) \qquad [\text{III}_3]$$

$$lb = y \wedge P \Rightarrow @ (Q_1 \wedge lb_1 = z_1) \vee' @ (Q_2 \wedge lb_2 = z_2) \qquad [\text{III}_4]$$

此处“$\vee'$”表示不可兼析取，对于并发进程来说，不同进程应对应不同的控制变量，故加些足标 lb_i。

这就是 XYZ 系统的基础语言 XYZ/E。从形式上看它似乎结构不够高级，但由它可以定义出一种高级的形式，即 XYZ/SE 或称结构化 XYZ/E。

if-then-else 语句形式：

$$lb = y \wedge P \Rightarrow @_1 (Q_1 \wedge lb = z_1) \mid @_2 (Q_2 \wedge lb = z_2) \qquad [\text{IV}]$$

case 语句形式：

$$\begin{aligned} lb = y \wedge P_1 &\Rightarrow @_1 (Q_1 \wedge lb = z_1) \qquad [\text{V}] \\ \mid P_2 &\Rightarrow @_2 (Q_2 \wedge lb = z_2) \\ &\cdots \\ \mid P_k &\Rightarrow @_k (Q_k \wedge lb = z_k) \end{aligned}$$

此处 P_k 可写成“~”，表示其他情形。

while 语句形式为：

$$*[lb = y \wedge P; \langle 语句列 \rangle]$$

此处 y 为此循环语句的起始标号，P 真，则转到循环体语句列的起始标号，P

假，则转出循环。在循环体语句列后面，事实上省略掉一条件元，形如“lb = loop end⇒o lb⇒y”。在 XYZ/SE 中所有条件元中转出标号都只许往后转。在循环体中所有转出标号均不许直接转出循环体。任何循环外面之转出标号只允许通过循环的起始标号转入循环。此外，每一程序只允许有一入口（如程序名标号或 start）、一出口（如 stop 或 return 等）。下面看一个例子。

例 1 求 m 的阶乘

这问题的抽象描述可表示成：

$$\Box[\,lb = factorial \wedge m \geq 0 \Rightarrow \Diamond(f = m! \wedge lb = fstop)\,]$$
$$\text{where}\quad 0! = 1 \wedge \forall x((x+1)! = (x+1) * x!)$$

其算法程序可表示为

$$\Box[\,lb = factorial \Rightarrow \circ f = 1 \wedge \circ j = 1 \wedge \circ lb = l_1;$$
$$*[\,lb = l_1 \wedge j \neq m+1;\ \circ f = f * j \wedge \circ j = j+1\,]\,]$$

若写成 Prolog 型产生式，则可表示为：

$$\Box[\,lb = factorial \Rightarrow \circ FACT(0,\ 1) \wedge \circ lb = factorial;$$
$$lb = factorial \wedge FACT(x,\ z) \wedge\ w = (x+1) * z$$
$$\Rightarrow \circ FACT((x+1),\ w) \wedge \circ lb = factorial\,]$$

由于 FACT 中变量与时间无关，故条件元右边的下一时刻算子可省去。

这问题也可以将 where 部分的递归定义写成 Prolog 形式，而以查询语句形式表示抽象描述中的函数出现。后面将要谈到，我们用这种形式表示 XYZ/DSPV 系统中的快速原型示范。其形式为

$$\Box[\,lb = factorial \wedge m \geq 0 \Rightarrow \Diamond(f = ?\ w;\ FACT(m,\ w) \wedge lb = fstop)$$
$$\text{where}\quad FACT(0,\ 1)$$
$$FACT(x,\ z) \wedge w = (x+1) * z \Rightarrow FACT((x+1),\ w)$$

以上即时序逻辑语言 XYZ/E 及 XYZ/SE 的简单介绍。在 XYZ 系统中以此为基础，建立适应各种程序设计方式（范型）的环境，下面介绍主要的二种：

（Ⅰ）高级语言程序设计环境　其中包括几种工具，如通用 PDL 型设计工具，及语义导引的编译生成系统，源到源转换等。下面介绍编译生成系统 XYZ/CCSS。

在 XYZ/CCSS 中以属性文法表示静态语义，它对源程序进行语法分解及静态语义检查，同时将源程序转换成一种较标准的形式（如将嵌套作用域中同名量换名），然后转入动态语义部分：用语义方程表示将源程序转换成相应的 XYZ/E 程序。这些语义方程用一种源语言书写，其中 LAB 表示标号，如所给源程序中在此位置上有一标号，它即指向此标号，否则由系统定义一新标号，LAB_1 表示系统定义的一新标号。如下元语言符号

$$lb = y < |[\,X\,]| > \circ lb = z$$

表示求出［X］后，将 lb = y 及 olb = z 分别替换其入口及出口 lb 等式。这种语义方程与语法公式对应，它既表示其右边的语法单位的语义定义，也表示语义子程式。下面即与“条件语句”，“循环语句”相应的语法及语义方程：

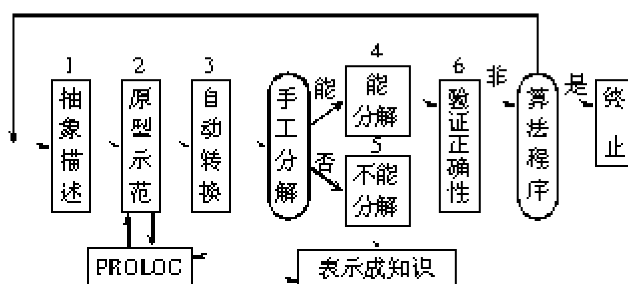
$$\langle condition\ S\rangle ::= if\langle cexp\rangle then\langle Statement\rangle_1 else\langle Statement\rangle_2$$
$$[\,LAB:\langle condition\ S\rangle\,] \leftrightarrow$$

lb = LAB ∧ [〈cexp〉] ⇒ olb = LAB1;
lb = LAB1 < |[〈Statement〉$_1$]| > olb = LAB3;
lb = LAB ∧ [¬〈cexp〉] ⇒ olb = LAB2;
lb = LAB2 < |[〈Statement〉$_2$]| > olb = LAB3;
lb = LAB3 ⇒ olb = NEXT

〈wloop〉:: = while〈cexp〉do〈Statement〉

[LAB:〈wloop〉] ↔
lb = LAB ∧ [〈cexp〉] ⇒ olb = LAB1;
lb = LAB1 < |[〈Statement〉]| > olb = LAB;
lb = LAB ∧ [¬〈cexp〉] ⇒ olb = NEXT

(Ⅱ) 抽象描述程序设计环境　其步骤可用如下框图表示



下面例2说明其中1，2，4三步：

例2　求 $\sum_{i=0}^{k} i!$

其抽象描述为

$$\Box(lb = Sumstart \wedge k \geqslant 0 \Rightarrow \Diamond(S = \sum_{i=0}^{k} i! \wedge lb = Sstop))$$

$$\text{where } 0! = 1 \wedge \forall x((x+1)! = (x+1) * x!)$$

$$\wedge \sum_{i=0}^{0} f(i) = f(0)$$

$$\wedge \forall n(\sum_{i=0}^{n+1} f(i) = \sum_{i=0}^{n} f(i) + f(n+1))$$

其原型示范为如下形式：

□(lb = Sumstart ∧ k ≥ 0 ⇒ ◇(s = ? p:
SM(k, q, p) ∧ q = ? r:FACT(k, r) ∧ lb = sstop))
where　FACT(0, 1)
　　FACT(x, z) ∧ w = (x + 1) * z ⇒ FACT(x + 1, w)
　　SM(0, h, u) ∧ G(0, h)
　　SM(i, n, u) ∧ G(0, n) ∧ G(i + 1, v) ∧ w = n + v ⇒ SM(i + 1, v, w)

其第一步分解后的程序为

$$
\begin{aligned}
&\Box[\, lb = \text{Sumstart} \Rightarrow oi = 0 \wedge os = 0 \wedge olb = l_1;\\
&*[\, lb = l_1 \wedge i \neq k+1;\\
&\quad lb = l_2 \wedge i \geq 0 \Rightarrow \Diamond\ (f = i! \ \wedge lb = l_3);\\
&\quad lb = l_3 \Rightarrow os = s + f \wedge oi = i + 1\,]\,]\\
&\text{where}\quad 0! = 1 \wedge \forall x((x+1)! \ = (x+1) * x!)
\end{aligned}
$$

这“程序”中由 $lb = l_2$ 所定义的那一条件元仍是一抽象描述，故仍应重复上述过程作进一步的分解。分解前可先查可重用程序库中是否已有程序的抽象描述与此处的一致，如果有，即可将其相应的算法程序直接引用，替入此处抽象描述即可。幸好，在例1中已有此程序，故只要更换标号名字即可将其算法程序替入本程序中的抽象描述，即得到所求的算法程序。

在例1、例2中，在从抽象描述经过分解得到程序后，应验证由此程序可推出该抽象描述。此处略。对于那种抽象描述，如找不到方法将它分解成下一层次的程序，则直接用原型示范的办法，该部分抽象描述表示成 Prolog 程序及查询语句来执行。这一部分“程序”，事实上表示了整个程序中的知识段。因此，以此法生成的程序可能是算法与知识交织而成的。

参 考 文 献

[BC] Backus, J. “程序设计能从冯·诺依曼式的设计风格中解放出来吗？函数式程序设计及其程序代数”（中译文）计算机科学，1984，No3—4.

[BF] Barr, A. and E. A. Feigenbaum ed. 《The Handbook of Artificial Intelligence.》 Chapt. X. 1982.

[BKP] Barringer, H. et. al. “Now You May Compose Temporal Logic Specification” draft. 1980.

[BMS] Brodie, M. L. et. al.（ed.）《On Conceptual Modelling》, Spring-Verlag, 1984.

[C] 周巢臣“形式语义学引论”，《计算机研究与发展》，1987，No7.

[FI] Fickas, S. F. “Automating the Transformational Development of Software”, IEEE Trans. on Softw. Eng. Vol SE-11, No11, 1985.

[FL] Floyd, R. “The paradigms of programming” CACM, No8, 1977.

[FN] Furtado, A. L. and E. J. Neubold《Formal Techniques for Data Base Design》, Spring-Verlag, 1986.

[M] Mead, C. A and L. A. Conway《Introduction to VLSI Systems》, Addison-Wesley, Reading Mass. 1980.

[PS] Partsch, H. and R. Steinbrüggen “Program Transformation Systems”, Computing Survey, 15: 3, 1983.

[T1] 唐稚松“结构程序设计与结构程序语言”《计算机应用与应用数学》，1975－1977，No12 或中国科学院计算所技术报告 1977.

[T2] Tang, C. S. “Toward a Unified Logic Basis of Programming Languages”, Stanford, Dept. Comp. -Sci. Tech. Rep. No. Stan-cs 81-865, 1981, A Revised Version, Proc. IFIP 1983, Paris.

[T3] —— “XYZ, A Programming Development Environment Based on Temporal Logic”, 《Program-

ming Languages and Systems, Proc. of TC2 Working Conf.》(ed. Bormann) Dresden 1982.

[T4] —— "A Temporal Logic Language for Behavior Modelling of Information and Expert System" 《Knowledge and Data, Proc. of TC2 Working Conf.》(ed. Sernadas) Algarve 1986.

[T5] —— "To Unify Programming with A Temporal Logic Language System, A Step Toward A Logic Machine" Tech. Rep. of Dept of Comp. -Sci, Carnegie-Mellon Univ. Pittsburgh № CMU-CS-87—160, 1987.

[Tu] Turner, R. 《Logics for Artifical Intelligence》 Ellis Horwood LTm, 1984.

[W] Wasserman, A. I. "From Programming Environment to Software Engineering Environment", 《Software Engineering Environment, Proc. Inter. Workshop in Beijing》 (ed. X. Zhong, et. al.), 1986, Beijing.

XYZ 系统的设计思想

（软件学报，1(1)，1990）

§1 引　言

本文系根据作者应邀在 1989 年 6 月 14 日日本情报产业协会（JISA）与软件工程学会（SEA）联合举办的 1989 年软件年会（Software Symposium'89）上所作的题为“XYZ 环境”基调讲演中部分内容整理扩充而成。因该会议论文集中所收入的介绍 XYZ 系统文章[1]未写入这部分内容，故特写成本文作为［1］的补充。

XYZ 系统是一基于时序逻辑且适应多种程序设计方式的 CASE 环境。它所适应的程序包括过程性的，抽象描述型的，或产生式规则型的；串型的或并发的；常见高级语言型的，面向多种对象的，函数式的或逻辑的；文本的或图形的，后者又可以是数据流型的或控制流型的等等。整个系统由经纬两维组成：一维是一具有统一程序框架的时序逻辑语言 XYZ/E。它是整个系统的核心和经络，由许多子语言组成，以适应多种程序设计方式，而且它还可用来表示常见高级语言，某些专用的抽象描述语言（如面向分布式进程通信的描述语言 Lotos），以及多种图形语言（如数据流图 DFD 以及由 Petri 网扩充而成的控制流图 CFC 等）的操作语义。XYZ 系统的另一维是由多种协调的软件工具或 CASE 环境组成。由它们既可实现由常见高级语言或专用描述语言到 XYZ/E 的可执行子集的形式转换（再加上由属性文法表示其静态语义部分而成为一建立在形式语义上的编译的编译系统）；亦可支撑由需求、抽象描述、转换、验证或快速示范等步骤组成的程序设计方法论。其中转换可以是逐步分解型的（decompositional），也可以是逐步作出决定（decision），然后以由非形式到形式，由抽象描述到有效实现进行过渡的方式。这些方法既可用于文本程序的设计，亦可用于 CFC 或 DFD 图形程序的设计。在后一情形下还可自动生成由时序逻辑语言 XYZ/E 表示的操作语义文本，并证明前后步骤所产生图形的语义的一致性。与这些方法相应，系统中还包含一统一的可重用对象库以存储程序开发过程中每一步所产生的信息，这些信息既可用来追索程序设计过程，以帮助用户对它的描述的理解和验证，也可用作修改或重用这些描述的参考。这系统是在 Sun 工作站上实现的，它是在 Unix 与 X-Window 上新加的一层通用的基于形式化方法的软件开发手段，用户可用它们开发自己的具有精确语义的专用软件。

XYZ 系统的设计思想，概括起来可分为二个方面：

（Ⅰ）试图在理性主义与实用主义之间找到一种合理的平衡；

（Ⅱ）试图将近年出现的各种有意义的程序范型，在一协调的形式框架中统一起来。

作为（Ⅰ）的直接后果，统一地以时序逻辑语言表示系统中提供的工具的精确语义界面，以及由系统提供的环境所生成的程序的操作语义，再加上可重用对象库的图式管理系统对程序设计过程所产生的信息进行管理，这样就可大大加强程序的可重用性与可适应性。

作为（Ⅱ）的直接后果，由于由 XYZ/E 表示的过程性程序，抽象描述及产生式规则具有统一形式框架与计算模型，致使不同范型的程序块可在 XYZ 系统中任意组合并协调地运行。因此，XYZ 系统可方便地支撑基于知识的程序设计以及其它多种范型混合的程序设计。

下面就（Ⅰ）与（Ⅱ）作进一步的说明。在作此说明前，先阐述两方面共同的方法论哲学基础。

§2　分与合的哲学

我在大学读书时曾学过哲学。当时最流行的西方哲学流派为著名的剑桥学派[4]，其代表人物有罗素（B. Russell），摩尔（G. E. Moore）以及稍后的维特根斯坦（L. Wittgenstein）。这一学派又名分析哲学，其方法论主要特点是对概念作逻辑分析。我现在已不大记得清这些哲学家著作的某些具体内容，但他们所提倡的分析方法却一直在我所从事的学术工作中起作用。我发现许多表面上矛盾的概念，如对之进行深层分析，有时可能变成相容。比如，常见程序语言中的变量具有动态取值的特征，也就是在程序运行中可以新值覆盖旧值；而逻辑系统中的变量概念则是静态的，其值是一次决定的。故两种变量概念是矛盾的。但如采用 Lucid 模型[3]，以可随时间延伸的向量解释变量，则上述两种不同的变量概念即可以统一起来。以此作为基础，即可用一致的逻辑框架将直言式的功能描述与命令式的状态转换以统一的方式予以表示，从而填平逻辑与程序语言之间的鸿沟。

从五十年代起，我又从毛泽东的哲学著作中学到另一种分析方法。这就是对具体矛盾进行具体分析的方法。其基本点是：从事物的发展变化过程中，历史地对事物内包含的具体矛盾进行具体分析。我深感到这方法使用得当，对发现事物的发展变化趋势，分清问题中各方面的主次，提出解决问题的办法很有帮助。我在文［2］中即试图应用这种方法对程序技术发展三十年的历史与现状进行分析。文［2］指出，不论从技术方面说，还是从理论方面说，程序技术研究在七十年代的特征是由单一走向分散，出现了许多新的技术领域，许多不同的方法与互不相容的理论体系；而到八十年代则与这种“由合走向分”的动向相反，是“由分走向合”，即将各种分散的技术与理论综合起来组成水平更先进的工程性系统。具体内容已在［2］中详述，此处不赘，下面只想从哲学的角度来谈谈。

这种具体分析矛盾的方法有以下哲学前提：（1）事物发展变化是客观存在的，

不依赖于人们的主观愿望。但人们如对之有清醒的认识，则可因对其产生变化的内在因素或其中的某些方面，施加影响从而引导变化的方向与程度，(2) 产生变化的因素有事物内在的，也有外在的，而前者是起主要作用的。(3) 这种促使事物变化的因素即矛盾。矛盾各方有主有次，这种主次地位是可以在过程中转化的，同时，矛盾的各方在一定条件下可以融合，也可以引起冲突。正是由于这些转化、融合与冲突，构成事物的发展变化。

下面我们进一步看看，事物是以什么样的方式进行发展变化的。对此，可以有三种不同的哲学观点：(a) 认为事物变化过程中只有量的变化，无质的飞跃。这是所谓进化论观点。这种观点如引向极端，则是一种不承认革命的改良主义。在 Stars 计划中，“进化论”观点属于此类。(b) 另一极端即认为事物变化过程是由接连不断的质变所组成，不承认质变之前量变阶段的意义。这是一种急躁的革命论。容易产生冒进盲动，终必引起巨大的阻力而失败。在 Stars 计划中有一种“革命性”观点，如将这观点推向极端即可产生这种后果。程序语言或方法论均与人的思维方式相关，要改变人们已习惯的思维方式是很困难的事，必然遭到习惯的阻力。形式抽象描述与人们久已习惯的高级语言编程方式完全不同，企图在短时间内要求人们放弃已习惯的后者而采用前者显然是不易被接受的。欧洲一批力量很强的专家希望在工业界推行指称语义方法已近十年，至今仍然收效甚微即说明了这一点①。(c) 显然，较妥善的态度应该是既承认量变也承认质变。质变只是在原有质的基础上进行长期量变，逐步将新的质引入并融合于旧的基础之中，在积累到充分程度以后才进行的飞跃。只有这样的逐步变化方式才可促使事物顺利的发展。虽然这一原则的正确性易为人们所理解，但具体实施远非易事，关键是既要克服主观方面的片面性，又要能实际在事物发展变化中根据具体条件不断掌握适当的分寸，并准确地分清界限。而且，在旧质基础上引入新质时，如何能使矛盾的方面作合理的安排使之协调，这就很需要进行精微的分析，分清层次与范围，绝非简单的混合。这样形成的过渡性对象，既是新旧质的融合体，必然“非驴非马”，从哪一个方面看可能都不是完美无缺的，不是很彻底纯净的。而根据当时条件从整体发展看却可能是最佳选择。如何看待或评价这种中间发展阶段形成的对象，涉及哲学方法与价值标准问题。这个问题不解决，就很难在实际事物发展过程中处理量变阶段的矛盾。我认为，一种值得推荐的哲学方法是中国古代儒家的“中庸之道”或辩证法的“合二而一”理论。一对矛盾存在于统一体中，必存在其对立的方面，亦必存在其统一的方面。无前者而不成其为矛盾，无后者则彼此无联系亦无所谓矛盾。在质的飞跃阶段，应强调矛盾的对立方面，不然就不能促成突变②。在量的变化阶段，却应强调其统一的方面，否则即不能进行融合。朱熹对“中庸”的解释是：“中者不偏不倚、无过不及之名，庸，平常也”。程颐认为，中庸的理论“始言一理，中散为万物，末复合为一理”。由此可见，中庸并不是简单的折中主义，而是告诉人们如何处理由分到合再由合到分的方法。而且在融合矛盾的两方面时，应从实际出发注意防止片面，

① 最近丹麦的 Bjorner 教授与本文作者面谈时，亦承认了这一事实。

② 我个人体会，毛泽东曾将他的方法论概括为“一分为二”，即旨在强调矛盾的对立性这一方面。在变化的质的飞跃的阶段作这样的解释是合适的。

避免偏激，掌握分寸，做到按事物的常理将对立的方面安排得恰如其分。而且这种安排应根据当时发展的具体条件来选择，所以说“君子而时中”。这是一种处理实际问题的有效方法，而不是评价理论，建立学科体系的方法。我认为，在程序技术发展的现阶段，需要强调由分走向合。将各种分散对立的技术和理论进行选取的综合，组成水平更高的软件工程环境，在这样的时期，强调一下“合二而一”，注意一下调和矛盾的“中庸之道”，无疑是有现实意义的。

XYZ 系统设计思想的（Ⅰ）与（Ⅱ）两方面，正是根据程序技术发展过程中产生的以下两方面的现象提出的：（ⅰ）软件工程技术与形式化理论的分离与脱节以及语义理论本身的分散发展，（ⅱ）程序范型的分裂。由于我们认识到现阶段的程序研究的总趋势是由分走向合，所以，在具体分析了以上各方面矛盾的基础上，根据我国以及世界学术发展现阶段的条件，试图应用“合二而一”及“中庸之道”的方法，将矛盾的各方面进行了精心的剪裁与安排，组成了这一系统。应该承认，(a) 这种集成的道路不是唯一的，XYZ 系统也不能说从各种角度看都是最优的选择，但我们认为 XYZ 系统不是草率完成的工作，至少在设计上是经过了反复推敲研究的，经过慎重权衡与修改的。(b) 既然这只是一种程序技术发展量变阶段的工作，所以从任一局部的理论或技术看，都不能说有惊人的创造。甚至，有时为了全局的需要，我们还宁可牺牲某些局部的新奇与完美。为了全面权衡得失，不可避免地只能采取“中庸”的态度。比如，现阶段作为 XYZ 系统核心的时序逻辑语言 XYZ/E，选择的是一阶时序逻辑系统。这样的逻辑有许多优越之处，用户较易接受，也较易实现，而且它的表示能力已相当的丰富，常见的程序概念多能用它描述。但是，有些深层的程序概念，它却无能为力，比如形式过程参量，受囿局部变量等均要求高阶逻辑系统。但为了保持系统的简明性，易于被接受和工程实现。我们只好分成二步走，目前面向广大用户的系统采用一阶系统，至于高阶逻辑，则留待以后扩充以面向少数特殊用户的需要。这些选择标准都带有很重的工程特征，也就是我们所谓的“中庸之道”。下面，我再应用这样的哲学方法，对前述（Ⅰ）与（Ⅱ）两方面作一些说明。

§3　理性主义还是实用主义

前几年，我多次访问了北美与西欧。我得到的印象是，虽然这些地区的国家在深刻的形式语义理论及先进的软件工程技术各方面都有很好的工作，但从总的风气讲，两地区所强调的侧重面是不同的。西欧国家更强调形式语义理论的研究以及计算机科学的学科基础的建立，而北美的技术界则更看重新软件技术的创新及其应用。应该承认，这两个方面对形成更高水平软件都是十分重要的。在七十年代，为了解决软件开发中产生的困难问题，在一定时间内分开进行某一方面单刀直入的深入探索，不同地区根据自己的条件各有侧重地进行研究，可能是必要的，大有益于最后促进软件技术发展的。可是，到了现在的发展阶段，总的潮流已趋向于综合各方面的成果组成更先进的系统的时候，如果仍然以这种南辕北辙的方式片面强调一种价

值标准，促使理论与技术向各执一端的方向发展，则势必使理论与技术日益分离，作茧自缚，最终起阻碍计算机科学与技术向健康方向发展的作用。这就是我所感到的理性主义与实用主义的矛盾。这种矛盾在一定情况下之所以有害，主要是由于脱离了当前技术发展的潮流，每一方片面强调所习惯的一种哲学价值标准而忽视了另一方的意义所致。事实上，世界上早已有许多重要的计算机科学家指出了这方面的问题。斯坦福大学的 Knuth 教授早在七十年代中期即曾在数月内在欧洲举行的两次国际会议（世界数学会及国际控制论会议）上作过两个报告，前一报告强调数学在促进计算机科学发展中的重要作用，后一报告则着重说明不适当地强调将研究数学的方法和标准用于计算技术的研究与开发，有可能起的消极影响。最近将在旧金山召开的国际计算机会上，他又将作题为“理论与实践”的基调讲演说明二者的相互影响，从而强调两方面结合的重要意义。应该说，这是一次很及时的箴言！事实上，近数年来，在软件工程领域正出现一种新的趋势，即将软件工程方法、工具与环境方面的新技术与形式化语义理论有机地结合起来，形成更高水平的 CASE 系统，这就是所谓“建立在形式化基础上软件工程环境”（Formally Based SE Environment）[6] 或“建立在语义基础上的程序设计环境”（Semantically Based Program-Design Environment）[3]。这一趋势的出现标志着程序技术发展进入一新的时期，即理论与技术有机结合的时期。它一方面表明形式语义理论研究日益成熟，已进入工程实用的阶段；另一方面也表明，软件工程技术已发展到这样一个地步，它要求使其复杂的工具具有精确的语义界面才便于安全地使用，同时由它所支撑开发的程序块，具有精确的语义，才可以合理地重用。下面，Brown 大学 Aia Giacalone 等的观点颇具代表性[6]。

“一程序设计环境中如提供建立在形式化基础上的工具与运算作为手段以在统一的形式框架中操纵程序，将比不具这种特色的环境有重要的优越性，特别是在这些工具及由它产生的程序的一致性，概念的简明与协调等方面。

概念的协调性在为用户提供含义精确的工具与运算，使他们能根据自己的特殊需要将基本环境予以扩充方面具有本质的重要性。如果这种工具能在形式化层次上予以开发，从而能被精确地理解，它就更能向用户提供关于其行为的非形式化的或半形式化的，然而却是精确的定义。”

形式化理论与软件工程技术结合虽是应受欢迎的好事，但并不是简单易行的。因为两方面各有自己的价值标准。遇到具体结合的情形，即可能出现矛盾。形式理论工作者往往看重概念的精美，表示能力的强大，及系统的完整性等。而软件工程专家则更注意用户方便性，时空效率，系统可靠性，程序可重用性及可适应性等。如果任何一方执着于自己所习惯的一种标准，则遇到矛盾时很难协调。事实上，要使双方结合组成系统，势必针对具体问题各有取舍，最后得到的系统从哪一方的标准看都有不能尽如人意之处，但只要从总体看是一最合理的选择即可。这里，就不免要让“中庸之道”发挥作用。这事实上是处理实际问题不可避免的现实主义态度。

目前已存在的形式语义理论很多，究竟应如何选择？事实上，已存在的各种理论均有其适应的方面与不适应的方面，没有一种理论具有全面的优越性。因此，当

前流行的做法是选择几种方法予以适当的剪裁、组合，使之在一协调的系统中各自起合适的那一方面的作用。当然，这并不是说，在一系统中各种理论起同等的作用。特别是一系统要求用户熟悉所有各种形式理论才能使用，必然遭到用户的拒绝。所以，在一系统中必有些理论是起主要作用的，是用户需要了解的，另一些理论只能起次要的局部的作用，用户可以忽略。

现在比较流行的形式化方法及描述语言，大致可分以下三类：

(A) 基于代数映射的形式方法。这里可包括指称语义、代数语义、带类型λ演算等。直觉主义逻辑及构造主义证明方法亦可归入此类。这些方法的优点是数学形式较为精美，理论基础深厚。在欧洲学院式研究中占压倒优势，可以说是计算领域中理性主义的代表。但从工程实际的角度看，则有许多不足之处。(i) 与长期软件工程技术界编制程序的习惯相距太大，(ii) 要求使用者有较好的抽象数学训练，(iii) 许多重要的程序概念和性质如并发进程，特别是活性，很难用此方法自然地表示，(iv) 实现的基础多要归结为函数式语言，这种语言的执行效率及表示能力均有一定的局限性。因此，这一方向虽经大量高水平的专家长期努力，仍难以为工程界广泛接受。不过在某些特殊的专用领域，如 CCS 在通信协议领域，以及代数公理方法用于描述基本数据类型等方面，这种方法仍是很有意义的。

(B) 基于编译技术的形式方法，比如属性文法及扩充的有穷自动机理论等。由于后者目前尚未构成一完整的体系，此处很难评价。至于属性文法则是目前最能为软件工程界接受的形式化方法之一。它是将语法分析方法在上下文相关部分直接扩充而成，紧密与编译技术（特别是名字表技术）相联系。所以，这方法具有理论与实践密切结合的特征。但它也有些不足之处，(i) 由于直接反映编译技术的要求，对于一些不太熟悉编译的内部结构的人，往往感到难以理解和书写，(ii) 抽象性不够，反映技术细节太多，(iii) 较适于表示静态语义，对于实际执行过程的操作语义，显得无能为力。所以，这一方法，被许多系统（Ergos，XYZ）采用来表示静态检查。

(C) 基于逻辑的形式方法。包括古典逻辑及时序逻辑等。由于古典逻辑可看作是时序逻辑的特殊情形，甚至 Prolog 这种建立在 Horn 子句上的系统亦被许多时序逻辑系统（如 XYZ/E，及 Temporal Prolog）所包括，所以，这里只需考虑时序逻辑就够了。由于 XYZ 系统采用时序逻辑为基础，所以，下面更着重地说明一下时序逻辑的优缺点。1983 年 IFIP'83 会议上 L. Lamport 对时序逻辑的优越性作了以下的概述[9]：

(i)"时序逻辑是古典逻辑的一种颇为简单的扩充。它的确提供了描述一程序的时序行为的一种自然的途径。"

(ii)"时序逻辑是适于对并发程序进行抽象描述及论证的好方法。……这种时序逻辑允许你表示出一程序的两方面性质：

· 安全性，即肯定一程序不会做坏事。

· 活性，即肯定一程序最终将做好事。"

(iii)"虽然有许多理由说明我为什么喜欢时序逻辑，但有一个理由是主要的：时序逻辑以一种简明而自然的方式支持分层描述与论证。经验表明，描述一复杂系

统的最佳途径是通过分层的抽象，从高层抽象描述起直到用一程序语言实现止，每一层是下一层的抽象描述，同时又是上一层的实现，时序逻辑提供一统一的逻辑系统描述各层次的抽象——从最高层次的抽象一直到用程序语言实现。”

（iv）“时序逻辑是形式逻辑的一个分支，它对许多人还是一新的理论。当一新的形式符号系统提出时，人们常常宣称它是简明的，自然的，而且容易学的。……但学习一新的形式符号系统，绝不是件易事，因为它要求学会用新的方式思维。你可能发现，在开始学谓词演算中的与时也是很困难的，只有当你学会改变你的思维方式时，这形式符号才变成简单而自然的，学习用时序逻辑对程序进行论证，其难度与学着用与十分相似。”

XYZ/E 是建立在 Manna-Pnueli 的线性时间时序逻辑理论[10]基础上的语言[3,1]。它在这逻辑系统之上引进一种统一的程序单元图式。这语言除了具有上述 L. Lamport 指出的特征外，还由于引进了表示赋值及控制转移的等式，使表示算法的状态转换命令与表示功能的直言描述可以用统一的形式表示出来构成统一的计算模型。这是最早提出的可执行时序逻辑系统之一。它具有统一范型的特色。

当然用时序逻辑作描述工具也不是十全十美，其主要的常被指出的不足之处，也就是逻辑理论所共有的以下两方面的缺点：“使用逻辑形式体系存在两个基本障碍。其一，一般逻辑系统中最令人感兴趣的任务（例如检测矛盾）是计算上难以处理的。……其二，对于大多数人来说，复杂的逻辑公式特别难于书写和理解”[11]。

为了避免这两方面的问题，我们认为最有效的途径不是去寻找新的或修改旧的形式符号体系，而是求助于进行描述或使用描述的程序设计过程。显然，只有复杂的多层嵌套的时序逻辑描述才会发生难于书写，理解与验证或查错的问题。对于这样的复杂问题的设计，我们在 XYZ 系统提供了一组工具与环境，它支撑一种可称为“根据用户逐步作的决定进行描述、转换与验证（或快速示范）”的方法论。从[8]中看出 Ergos 计划也采取了极为相似的途径。根据这一方法，一复杂程序的设计是由一序列的步骤组成，每一步中用户根据问题的需要分清轻重缓急作出决定，并将这决定由非形式描述逐步转换成形式描述，由一图式管理系统将所有这些信息存入可重用对象库中以备以后查询，然后验证或查证这一步作出的形式描述是否与上一步的形式描述保持一致。每一步均按这样的方式进行直到得出可有效执行的程序为止。这一方法论的优点在于将一复杂程序设计过程由一次性描述与验证变成一逐步进行描述修改和验证的过程，而且将每次为新描述作的决定的有关信息均保存在一可重用对象库中以备查找，由此用户即可了解每一步所作决定的意图，从而使由此作出的描述较容易理解和验证。这样即可使这困难的问题变得容易。一次性描述之所以令人难以理解与验证其根本原因是所有描述过程中与作决定有关的信息均已丢失或掩藏，致使阅读最后的描述文本令人如坠五里雾中。上面这一方法无非是恢复了设计过程的本来面目而已。这样一方法论及其支撑环境应该说是有明显的优越性的，但要广泛为工程界理解与接受恐怕还得需要时间，且还应增加自动化或半自动化的辅助工具及文档手段。此外，恐怕还需要写出符合工程界口味的教程并进行培训。

Ergos 与 XYZ 系统都采用了上述程序设计方法并提供了相应的支撑环境与工

具。但二者的形式化基础是不同的，XYZ 是以时序逻辑语言为基础，Ergos 则以带类型的 λ 演算等为基础，两个系统都以属性文法作为描述静态语义的方法。

以上是就通用的描述语言及支撑工具而言。至于专用领域，其描述语言可根据该领域的特殊性而定，比如 Lotos 这种专用于描述通信协议的语言，则是以代数方法为依据的。这样的专用系统与上述通用系统是可以结合起来发挥作用的。前面已指出，在 XYZ 系统中提供了一基于形式语义编译的编译系统，以属性文法表示静态语义，以一种形式化元语言表示由所给源语言到 XYZ/E 的形式转换，通过转换得到的 XYZ/E 程序表示源程序的操作语义。这系统既可用于专用描述语言亦可用于常见高级语言到 XYZ/E 的转换。

我们试图通过上述途径在理性主义与实用主义之间保持一种合理的平衡。实践将表明这目标是否已达到。

§4　范型统一问题

在七十年代末之前，计算机体系一直统一于 Von Neumann 型，而程序范型则统一于以这种体系为基础的算法语言。经过七十年代计算技术与理论的分散发展，这种统一性被打破了，出现了许多常见高级语言以外的程序范型，如函数式程序设计，逻辑程序设计，基于抽象描述的程序设计，产生式系统，面向对象程序设计，图形程序设计等。它们所反映的计算模式都超出了 Von Neumann 体系的范围。而这些程序设计方式各有其适应的范围，任何一种也不能取代其他而独占统治地位。

这种范型分裂的局面也同时给程序设计带来不便。比如抽象描述与可有效执行的算法程序之间如果存在不可逾越的鸿沟，则由前者向后者过渡时将难以保证语义的一致性，同时在过渡的中间阶段，不可避免将出现两类程序混合出现的情况，这样的程序应如何表示？此外，近年来基于知识的程序设计很为人注意，如果算法与知识由两类不同的语言表示，二者如何掺合沟通？因此，很自然地产生了范型统一化的要求。UT Austin 的 Chandy 与 Misra 在［12］中的一段话与 XYZ 设计思想很相近似：

“今天，程序设计好像日益被分裂成许多神秘的支派，每一支有它们的传教士、僧侣与符咒。我们相信，在所有各类程序设计的基础之下，存在一种统一性。我们愿为识别它而作出贡献。”

XYZ 系统正是以此为主要目标之一而设计的。我们认为，这种统一性就是各种类型程序的语义模型。而且相信，时序逻辑语言 XYZ/E，正好可以用一种一致的形式框架将这统一的计算模型表示出来。这一致的逻辑框架，既可用来表示抽象描述，亦可用来表示有效算法，还可用来表示产生式系统；不论是串型的或并发的，确定性的或非确定性的程序都可在这逻辑框架中得到自然的表示。

有了这种统一的程序框架，则在基于抽象描述的程序设计过程中，从最抽象的描述到可有效执行的算法之间各种层次的程序以及中间过渡的混合型程序均可以统一的方式予以表示和相互连接，从而可进行分层程序设计并验证上下层程序之间语义的一致性。

有了这种统一的程序框架，则在基于知识的程序设计中，不论算法程序或表示知识的规则及其查询命令，也可以统一的方式得到表示，而且可以协调运行，互相通信。

有了这种统一的程序框架，则在面向对象的程序设计中，可以面向多种类型的对象，而且可以用统一的方式表示这些不同类型对象的语义。事实上，只有在建立了这种统一性的条件下，面向多种对象的系统才是真正有意义的，因为此时不同类型的对象之间才能有“共同的语言”进行语义沟通。

有了这种统一的程序框架，则许多在软件工程中有用的图形语言，如 DFD，Petri 网等即可以找到一种方法表示出它们的操作语义，从而使对这类语言结点进行分解并验证其语义一致性变得颇为自然和方便。

目前，XYZ 系统还只实现一可运行的试验性系统，还有待进一步的改进使之工程化成为一完善的系统。我们相信，上述这些设计目标最终将可以达到。

感谢：本文作者对这次日本软件年会的东道主藤野晃延先生及 SRA 的岸田孝一先生及会议的组织者和SRA 的同行们对本人及我的助手的盛情邀请与接待，以及为进行讲演与演示所给予的多方面的支持与帮助，表示由衷的谢意。

参考文献

[1] 唐稚松，“XYZ 环境”，ソフトウエア·シンポシウム'89 论文集，1989 年6 月 14－15 日，东京。

[2] 唐稚松，“程序技术研究三十年”，计算机科学，1988，第3 期。

[3] C. S. Tang，“Toward a Unified Logic Basis of Programming Languages”，Proc. IFIP'83，1983，Paris.

[4] 约翰·查尔斯沃斯著，田晓春译，《哲学的还原》，四川人民出版社，1987。

[5] Ascheraf E. A. and W. W. Wadge. “Lucid：A Nonprocedural Language With Iteration”，Comm. ACM 20，1977.

[6] Giacalone A. et al.，“Toward a Formally-Based Programming Environment”，《Integrated Interactive Computing Systems》，(ed. P. Dgona & E. Sandewall)，1983.

[7] Cleaveland R. et al.，“The Concurrency Workbench：Operating Instructions”，Tech-Note 2/88，Computer Science University of Sussex GB，1988.

[8] Lee P. et al.，“Research on Semantically-Based Program-Design Environments：The Ergos Project in 1988”，Tech. Rep. No CMU-CS-88－118，Computer Science Dept.，Carnegie-Mellon Univ.，1988.

[9] L. Lamport，“What Good Is Temporal Logic”，Proc IFIP'83，Paris，1983.

[10] Manna Z. and A. Pnueli，“Verification of Concurrent Programs pt Ⅰ-Ⅱ”，Dept. Comp. Sci.，Stanford Univ.，Stan-CS-81－836，Stan-CS-81－843，1981.

[11] Rich C. and R. C. Water，“Automatic Programming：Myths and Prospects”，中译文，计算机科学，1989，No. 3.

[12] Chandy K. M. and J. Misra，“Parallel Program Design：A Foundation PTI”，Draft，Dept. of Comp. Sci.，Univ. of Texas，Austin，1986.

A Unified Formal Basis for the CASE Tools System

(Journal of Systems Integration,1993)

Abstract. XYZ system is a CASE tools system based on a temporal logic language XYZ/E which can represent every essential feature of conventional HLL's (sequential or concurrent), specifications of different levels, production rules, operational semantics of graphic languages in a uniform framework. With this formal language as the common basis, all the CASE tools including various kinds of graphic tools for distributed process, concurrent programs with phased memory and sequential programs, tools for verification, rapid-prototyping, language transformation, and module management can be connected freely to form more sophisticated and integrated systems.

Key Words: Temporal logic language, CASE tools, specification, verification, rapid prototyping, graphic languages, software engineering, formal language.

1. Introduction

For the past two decades, some progress has been made in the research of software engineering. Various kinds of methods, paradigms, and techniques have been invented. But the major goal of this research, to promote the productivity of software by an order of magnitude, seems still far beyond reach in the foreseeable future. I believe it is the right time for us to consider the problem of what we ought to do in the future.

1. A practical approach is to find an appropriate way to unify those significant methods, paradigms, and techniques into one coherent system in which each can play its appropriate role and avoid its shortcomings.

   Experiences have shown that most of these technical inventions only have merits in some aspects. But software development and maintenance is a very complicated process. It consists of various aspects and stages, and the ways to connect them into one process are also divergent and very problem-dependent. All those invented methods, paradigms, and teehniques can only be advantageous in some aspects and stages or good to solve some problems. However, most of them could be subject to combination, and their roles played in the whole process could be mutually complementary. According to Chinese traditional view of historical development, long-time unity gives rise to splitting and long-time separation leads to unification again. It is evident that after almost two decades of concentrated attention paid to higher-level languages and compilation techniques in the 1960s and 1970s, there is a period of the past 10 years for separate researches of various kinds of paradigms, methods, and techniques in computer science and technology. Of course, most innovative inventions in these aspects are significant. But it seems to be evident that this period has come to an end. In order to make further

progress in computing, I think what we need to do at the next step is to find an appropriate way to select, tailor, and organize; in short, to unify these inventions into one uniform, coherent, and practical system. In fact, quite a few distinguished experts have reached more or less the same conclusion. The following quotations are only a few examples.

In [3], Chandy and Misra said: "Today, programming appears to be fragmented into increasingly esoteric subdisciplines, each with its priests, acolytes and incantations. We believe there is a unity underlying all programming." More recently, the Software Development and Maintenance Session of the IFIP Congress 1992 has made the following announcement: "The technology supporting the development and maintenance of software is steadily improving along a broad front. Since it is evident that no single method or technique will increase our ability to create and extend software systems by an order of magnitude, there is considerable merit in trying to integrate various ideas and apply them jointly to both the software development process and the software product."

2. Such a unified system must be a harmonious combination of a simple and expressive formal language as its unified basis with a group of computer-aided software design (CASE) tools to support the related methodologies.

   It is natural to reduce the unification problem to the design of a language. But as Winograd pointed out in 1979 [40], a programming language alone is not enough. In order to support the whole programming development process, there is still something "beyond programming languages," i.e., CASE tools. What we want to emphasize here is the other side of the problem, to serve as the unified basis of these CASE tools, a language is also needed, which must satisfy some special requirements:

   a. In general, each of these tools can only support one method or technique. For example, there are tools to support stepwise refinement, rapid prototyping, verification, structured design of data flow diagram, knowledge-based programming, object-oriented programming, etc. Each single tool can only support one of these methods. A complete and practical process of program development is, in fact, the result of combining many of these methods and techniques. In general, the ways of combination are various and dependent very much on its application domain. Consequently, these CASE tools must be able to be freely connected together to form more sophisticated tools. In order to facilitate this kind of free connection, the precise semantics of their interfaces must be representable by a language which ought to be formal, simple, and expressive. In other words, not only programming languages need CASE tools, but also CASE tools need a unified formal language to describe their interfaces.
   b. To design such a unified language, the traditional way of designing PL/1 (Algol 68) and Ada is no longer desirable. Not only would such an omnibus language be too big, but also the unified language we need is substantially different from PL/1 and Ada, it must be a formal language which could represent the operational semantics underlying all those different paradigms with which it is to unify. As a result, this unified language ought not to be too big, but must be very expressive in representing formal semantics.

3. A desirable formal language as the unified paradigm must be the result of combination of those existing methods of formal semantics, but in which, a kind of temporal logic language called XYZ/E might be a better choice to play the central role.

As is well known, there are various kinds of methods of formal semantics, each of them has its strong side as well as its weakness. For example, the algebraic approach is a good means to represent the nonlogical axioms of the data structures, but it is incompetent to represent the operational semantics, in particular, to represent the liveliness properties of concurrent process. The attribute grammar is good to represent the static semantics of higher-level languages in compiler construction, but it is weak in representing the dynamic aspect of computation. The first-order logic and recursive functions are suitable to represent the abstract specifications, but if used as an executable language, they are very inefficient. Even in representing specification, the suitable application areas of these two kinds of formal techniques are different. For example, the declarative logic assertions are appropriate to specify the pre-post conditions of those problems as gcd, but the recursive function is a better means of representing the specification of such kinds of problems as factorial. Consequently, in order to be able to represent both kinds of problems, these two techniques had better be combined into one specification language. However, the simple-minded combination of the first-order logic with the recursive function is still not strong enough to deal with the operational semantics of many problems in conventional programming, such as loops, pointers, dynamic binding, etc. In addition, representing concurrent properties is also beyond their capability. In order to enhance the expressiveness of this unified language, I believe temporal logic is a better choice. Lamport [12] has discussed this problem very persuasively. What I hope to add is that XYZ/E is the result of embedding state-transition mechanism into temporal logic. It has some merits which other temporal logic language systems are in lack of:

a. It has a unified program framework in which efficiently executable procedural programs of conventional programming style, specification of different abstract levels, production rules, and even the operational semantics of conventional language and graphic languages (such as DFD, Petri nets, PAD, etc.) can be represented uniformly. Consequently, the transition in stepwise refinement from the specification of different abstract levels to the efficiently executable code can be described and validated in a smooth way; the gaps between the knowledge and the algorithmic programs, the graphic programs, and its textual codes can all be filled up naturally.
b. Furthermore, many difficult problems, such as those concerning dynamic binding, which are very hard to deal with by means of Hoare logic or other formal techniques, are no longer an insurmountable barrier in this framework.
c. This language is sufficiently formal and precisely defined, but does not require so much mathematical training as to be beyond the reach of common programmers.

4. A desirable approach for CASE system is to form it into a kind of two-layered structure: The lower level is a group of CASE tools to support those generally applicable methods, and the upper level is those highly integrated CASE environments for various specific application domains.

The reason to structure the CASE system in two layers is based on the consideration of the contradictory situation of integration versus flexibility. Once an environment is integrated, the steps of program development and maintenance are rigorously fixed. It loses flexibility to suit the specific interests of different users. In fact, in many applications, even though their programs are developed according to very similar methods,

there are still some minor divergences for the needs of special domains. An integrated environment is often unable to satisfy these domain-specific needs of different users. In order to solve this dilemma, we divide the CASE system into two layers. The basic layer consists of all those CASE tools or those sophisticated tools constructed from the more basic CASE tools. These tools are loosely coupled with each other. They support all those generally applicable methods. But in the upper layer, there are domain-specific CASE environments. They are designed to meet the requirements of their special application domains. These environments are highly integrated according to the habit of the users of that special domain. They consist of CASE tools chosen from the lower layer, but maybe, having some modifications and additions in order to suit their special needs. In them, the domain knowledge and domain-dependent reusable programs could be accumulated in the base structured in the most friendly way to the specific users. All these kinds of flexibility are unavailable in an integrated general-purpose CASE environment. I believe, to form the CASE system into this kind of two-layered structure is a way to solve the dilemma of integration versus flexibility. It would make CASE systems more acceptable to the industry.

XYZ (Xiliehua Yuyan Zu) is a CASE tools system designed according to the above considerations. A decade of efforts in this project has shown that in order to achieve our goals, some kind of new philosophy, departing from those influential in past decade, is needed because there are so many contradictions involved in the design decisions (e.g., formal rigidity vs. technical practicability, simplicity vs. variety, innovation vs. inheritance, etc.) and also other concrete contradictions in the development process [44]. Running to the extreme of either direction only leads to one-sidedness and obsolescence; what we need is to find some kind of balance. Neither pure rationalism nor pure pragmatism, neither radical revolutionary approach nor radical evolutionary approach could help us. We find that the doctrine of golden means proposed by Confucius could serve our purpose, because this philosophical methodology lays emphasis on appropriateness and balance in unification. This is just the way of thinking desired in software engineering in the future.

In short, the XYZ system consists of two dimensions: a temporal logic language, XYZ/E as its basis and a group of CASE tools to support various kinds of methodologies. In this article we would introduce this language first. As for the tools, because there are too many tools to be introduced in one article, we would only select one group of them explained with an example, i.e., the graphic tools for structured analysis and design.

2. The Temporal Logic Language

2.1. The Logic System

The TLL XYZ/E is based on the linear time temporal logic system proposed by Manna and Pnueli [20, 21]. One of the special characteristics of this logic language is its ability to represent every basic feature of conventional higher-level languages with their proper forms in this logic framework.

Variables are divided into global and temporal, they are typed as Pascal. The basic types include: integer (I), character (C), octagonal number (O), floating point number (F), boolean (B), stack (S(X)) or S), their semantics are defined with nonlogical axioms. This set of basic types is extensible. A variable v declared with type X is represented as **v:X**, which is synonymous with **X(v)**. The structured vector and record types are considered as abbreviations: *v;* $\hat{}$ *A(n,X)* and *v: R(m*1:*X*1,. . .*mk:Xk)* represent *v*(1):*X* ^ . . . ^ *v*(*n*):*X* and *v__m*1:*X*1 ^ . . . ^ *v__mk:Xk*, respectively. It is interesting to point out that pointers can be represented nicely in this logic language. We would discuss this problem later. Constants are represented as in Pascal. Labels and names can be treated as constants assigned to the variables of type NM.

Well-formed formulas (WFF) are defined in the regular way. In addition to the connectives and quanitifiers in classical first-order logic (FOL) (~(negation), ^(conjunction), **$V**(disjunction), − >(implication), ==(equivalence), **$A**(univ. quantifier), **$E**(exist.quantifier), **$T**(true), and **$F**(false)), there are future-tense temporal operators: [ ](always), < >(eventually, **$O**(nexttime), **$U**(until), and **$W**(unless). Only in the special sublanguage XYZ/PPE are the past-tense temporal operators allowed to occur in the condition part of a production rule.

On the basis of this temporal logic system, TLL XYZ/E is characterized with the following special features:

In temporal formulas, there is one kind called state transition equations and it has the form:

$Ov=exp (1)

here *v* is a temporal variable and **exp** is an expression of similar type as **v** containing no temporal operator. It plays the role of the assignment statement in conventional languages. In particular, if the variable on the left side of equation (1) is lb (which is a special system variable of the type NM called the control variable and always used to indicate the current label of the program) and if the right-side expression is replaced by a label, say y, then (1) becomes:

*$O*1*b=y* (2)

This equation in fact means "goto *y*," here *y* is called the forward label. Coupling with (2), an equation of the form:

1*b=y* (3)

means "the definitional occurrence of *y*, here *y* is called a definitional label. Both (2) and (3) are called control equations. An equation of following form

*$*O(*v*1,. . .,*vk*)=(*e*1,. . .,*ek*) (4a)

is an abbreviation of the conjunction

*$*O*v*1=*e*1 ^ . . . ^ $O*vk=ek* (4b)

With these equations, almost all those state-transition concepts in conventional programming can be defined.

2.2. *Conditional Elements and Units*

An implication formula of one of following forms is called a conditional element (CE):

$$lb=yi \hat{} P=>\$O(v1,\ldots,vk)=(e1,\ldots,ek) \hat{} \$Olb=yj \qquad (5)$$

$$lb=yi \hat{} P=>@(Q \hat{} lb=yj) \qquad (6)$$

here $=>$ is a new notation to represent implication; P and Q are FOL WFF's called condition part and action part of a CE respectively; @ in (6) represents $O, < > , and sometimes in special cases also [ ]. These are two basic CEs in XYZ/E. (5) is used in representing executable algorithms and (6) serves to represent abstract specification. In fact, these two kinds of CEs are sufficient to represent every kind of sequential programs for our purposes. But in order to be more expressive in representing "interrupts," a kind of structure based on temporal operator *$U* (until) or *$W* (unless) are introduced as follows:

$$(M)\$U(N \hat{} \$Olb=z) \qquad (7)$$

here U can be replaced with W. In classical TL, "until" and "unless" are represented with the form *(M)$U(N)* and *(M)$W(N)*, respectively. But in XYZ/E we always need a more complicated form: $(M)\$U(N) \hat{} N->\$Olb=z$, which is abbreviated as $(M)\$U(N \hat{} \$Olb=z)$. In fact, (M)$U(N) is only a special case of this more general form, i.e., $(M)\$U(N \hat{} \$Olb=STOP)$. In XYZ/E, this special form of "until" is always used to represent "interrupt," in which the M in (7) represents a unit as the scope where the interrupt may occur and the N and the forward label equation are used to represent the condition and the action of the interrupt, respectively.

A WFF is called a unit if it is of the following form:

$$[\ ][cel;\ldots;cem] \qquad (8a)$$

$$Where\ cons \qquad (8b)$$

here ";" is another notation for conjuction; *cel*, 1=1,. . .,m are CEs of the forms (5) and (6); and "cons" is a WFF which is a conjunction of constraints and definitions with (8a) as its scope, this part is called "Where-part" of the unit and can be empty.

In a unit there are some special labels: The execution of a unit always begins with the label "START" possibly preceded or followed with the name of the unit and halts with the label "STOP." In fact, it is not necessary to halt in the ordinary sense, since there is always an assumed CE of the form

$$lb=Stop=>\$Olb=Stop;$$

at the end of each unit but omitted. There are other special labels such as "RETURN," "NEXT," "EXIT," etc. Their meanings are similar to those used in conventional languages. Each unit can only have one unique occurrence of "START" as its starting point of execution, which we call the entry of the unit.

Example 1. Factorial *f=m!* can be represented in different ways in XYZ/E as follows.

1. As an executable program:

```
[ ][lb=fact_START^m>=0=>&Oz=1^$Oj=1^$Olb=l1;
   lb=l1^j<m+1=>$Oz=z*j^$Oj=j+1^$Olb=l1;
   lb=l1^j>=m+1=>$Of=z^$Olb=STOP]
```

2. As an abstract specification:

```
[ ][lb=fact_START^m>=0=><>(lb=STOP^f=m!)]
   Where m! <- if m=0 then 1 else m*(m-1)!
```

3. As a production rule system:

```
[ ][lb=fact_START=>$On=0^$Olb=fact;
   lb=fact^n=0^n<m+1=>$O(FT(0,1)^z=1^lb=fact);
   lb=fact^n/=0^n<m+1^FT(n-1,y)^z=y*n
            =>$O(FT(n,z)^lb=fact);
   lb=fact^n=m+1=>$Of=z^$Olb=STOP]
```

There are different sublanguages in XYZ/E corresponding to the above different representations:

XYZ/BE or basic XYZ/E corresponding to (1) is a sublanguage of XYZ/E that represents sequential executable programs. All CEs contained in a unit of this sublanguage are of the forms (5) in which the condition P cannot contain quantifiers without bound. In this kind of unit, no recursive functions are allowed to occur in the CEs or to be defined in their "Where-parts." Of course, in the program of this sublanguage, (7) is also used to represent "interrupt."

XYZ/AE or abstract XYZ/E corresponding to (2) is a sublanguage of XYZ/E that represents abstract specifications. The CEs contained in this kind of unit are of the form (6) and sometimes followed with (7). Recursive functions are allowed to be used as a part of specification.

XYZ/E or XYZ/E in production rule form, as shown in (3), is a sublanguage of XYZ/E that represents rule systems. It is one in which all definitional and forward labels are identical except for the special labels as "START," "STOP," "RETURN," etc., or labels which are the entries of other units. In this kind of unit, the control equations with identical labels can be omitted. As a consequence, the general temporal logic rule systems (i.e., those without occurrences of lb control equations) can be considered this kind of unit. According to the methodologies supported by the CASE tools of XYZ system, the process of program design can start with temporal logic specifications without control equations.

The CEs of XYZ/BE and XYZ/AE can be mixed in one unit when it is needed.

Example 2. Summation of factorials, s= SUM(i=0..k) (i!)

1. In executable form:

```
[ ] [lb=s__START=>$Oi=0^$Or=0^$Olb=l1;
     lb=l1^i=k+1=>$Os=r^$Olb=STOP;
     lb=l1^i/=k+1=>$Olb=l2;
     lb=l2=>$Of=1^$Oj+1^$Olb=l3;
     lb=l3^j=i+1=>$Olb=l4;
     lb=l3^j/=i+1=>$Of=f*j^$Oj=j+1^$Olb=l3;
     lb=l4=>$Or=r+f^$Oi=i+1^$Olb=l1]
```

2. In specification form:

```
[ ] [lb=s__START^k>=0=><>(lb=STOP^s=SUM(i=0..k)(i!))
   Where $Ai(0!=1^(i+1)!=i!*(i+1)^
         $An(SUM(i=0..0)(i!)=0!^
             (SUM(i=0..n+1)(i!)=SUM(i=0..n)(i!)+(i+1)!)
```

3. In mixed form:

```
[ ] [lb=s__START=>$Oi=0^$Oi=0^$Or=0^Olb=l1;
     lb=l1^i=k+1=>$Os=r^$Olb=STOP;
     lb=l1^i/=k+1=>$Olb=l2;
     lb=l2=><>(lb=l4^f=i!);
     lb=l4=>$Or=r+f^$Oi=i+1^$Olb=l1]
   Where $Ai(0!=1^(i+1)!=i!*(i+1))
```

To be able to represent these three forms in one program framework makes the transition from specification to an efficiently executable program a smooth procedure.

The above two examples may have given rise to a misunderstanding that specifications in XYZ/E must be represented by means of recursive functions. In fact, sometimes logic assertions are more suitable, see the following simple example:

Example 3. The specification of "z=gcd(a,b)," where a,b are positive integers.

```
[ ][lb=gcd__START^a>0^b>0
    =><>(lb=STOP^z|a^z|b^$Ax(x|b->x|z))
   Where m|n==$Er(n=m*r)
```

The production rules cannot be mixed with others in one unit. This kind of unit is called a rule unit which can be combined with other rule units to form a rule system and communicates with other kinds of units as processes by message passing.

It is easy to see that not every sequence of CEs connected with ";" must constitute a meaningful program unit. In order to guarantee its validity, some basic assumptions must hold:

1. **Frame assumption of the temporal variables**—A state transition equation with one identical temporal variable on both sides is called redundant, for example, $Ov=v. A redundant equation is always assumed to be equivalent to $T in XYZ/E. As a result, we can always assume that every temporal variable in a program is assigned once (but its value is not necessarily changed) in each CE and those with redundant equation are omitted by default.
2. **Regularity assumption of the labels**—The definitional and forward labels in a unit must be mutually matched except for those special labels "START," "STOP," etc., and those forward labels corresponding to the entries of other units. This assumption eliminates the gaps and redundant CEs from a unit.
3. **Completeness and consistency assumption of the conditions**—In a unit, all those CEs with identical definitional labels are called related. The disjunction of the conditional part of all those related CEs must be equivalent to $T (completeness) and the conjunction of any two related CEs is equivalent to $F (consistency). The former guarantees no gap and the latter no ambiguity (as for nondeterminancy, it could be represented in other more logical ways shown later).

There is another assumption of fairness shown later.

2.5. *The Structured Higher-Level Form*

CEs of the form (5) and (6) are not well structured in the sense that there are different exits corresponding to truth and falsehood of its condition. This form has its advantages for conveniently representing the semantics of graphic programs and also its shortcoming for inconvenience in correctness proof. In order to keep the merits of the former aspect and avoid the weakness of the latter, we start with this more elementary unstructured form as its basic form (i.e., Basic XYZ/E or XYZ/BE) and to define a sublanguage of the higher-level structured form statements on the same basis; it is called XYZ/SE [41]. In this sublanguage, there are four forms of statement.

The most important one is the loop statement which is of the following form:

*[*lb*=*yi*^*P*=>(*lb*=*w*¦ *$Olb*=*EXIT*); (9a)

lb=*w*{¦*R*¦}*$Olb*=*yi*] (9b)

here *lb*=*w*{¦*R*¦}*$Olb*=*yi* means that {¦*R*¦} is a group of CEs (with one entry and one exit) representing *R* and the entry and the exit of {¦*R*¦} are replaced with *lb*=*w* and *$Olb*=*yi*, respectively.

The major difference between XYZ/SE and XYZ/BE is in the case statement, it is of following form:

```
?[lb=y^P1=>$Olb=z1
      ¦P2=>$Olb=z2
      ...
      ¦~=>$Olb=zk;
```

$lb=z1\{\mid Q1 \mid\}\$Olb=EXIT;$
$lb=z2\{\mid Q2 \mid\}\$Olb=EXIT;$
...
$$lb=zk\{\mid QK \mid\}\$Olb=EXIT] \tag{10}$$

Succession is represented by the following statement:

$$lb=y=>\$O(Q \hat{} \, lb=NEXT) \tag{11}$$

Conditional statement has two kinds of form:

$$lb=y \hat{} P=>\$O(Q1 \hat{} \, lb=NEXT \mid Q2 \hat{} \, lb=NEXT) \tag{12a}$$

$$lb=y \hat{} P=>\$O(Q \hat{} \, lb=NEXT) \tag{12b}$$

They correspond to "if-then-else" and "if-then," respectively.

Example 4. Represent z=gcd(a,b) for positive integers a and b with both XYZ/BE and XYZ/SE.

1. In XYZ/BE form:

```
[   ][lb=gcd_START=>$Ox=a^$Oy=b^$Olb=l1;
     lb=l1^x=y=>$Olb=l3;
     lb=l1^x/=y=>$Olb=l2;
     lb=l2^x>y=>$Ox=x-y^$Olb=l1;
     lb=l2^x<y=>$Oy=y-x^$Olb=l1;
     lb=l3=>$Oz=x^$Olb=STOP]
```

2. In XYZ/SE form:

```
[ ][lb=gcd_START=>$O(x,y)=(a,b)^$Olb=NEXT;
  *[lb=l1^x/=y=>(lb=l2|$Olb=EXIT);
    lb=l2^x>y=>$Ox=x-y^$Olb=NEXT
              |$Oy=y-x^$Olb=NEXT;
  lb=l3=>$Olb=l1];
  lb=l4=>$Oz=x^$Olb=STOP]
```

In (4), a proof system for XYZ/BE was introduced. Since it is not well structured, the proof system for XYZ/BE looks rather complicated. In this aspect, XYZ/SE seems superior.
The proof rules for the loop and the case statements are as follows:

$$\frac{NA \mid -[\ ][lb=w\{\mid R \mid\}\$Olb=y] \hat{} \, lb=w \hat{} \, INV=>@(lb=y=> \hat{} \, INV)}{NA \mid -[\ ][lb=y\{\mid X \mid\}\$Olb=EXIT] \hat{} \, lb=y \hat{} \, INV \hat{} P=>@(lb=EXIT \hat{}\hat{} \, P)} \tag{13}$$

$$\frac{\begin{array}{l} NA \mid -[\,][lb{=}z1\{\mid Q1 \mid\}\$Olb{=}EXIT \wedge lb{=}z1 \wedge PRE \wedge P1 => @(lb{=}EXIT \wedge POS) \\ \ldots \\ NA \mid -[\,][lb{=}zk\{\mid QK \mid\}\$Olb{=}EXIT \wedge lb{=}zk \wedge PRE \wedge PK => @(lb{=}EXIT \wedge POS) \end{array}}{NA \mid -[\,][lb{=}y\{\mid X \mid\}\$Olb{=}EXIT \wedge lb{=}y \wedge PRE => @(lb{=}EXIT \wedge POS)} \qquad (14)$$

here *NA* represents the nonlogical axioms of the system; *PER*, *POS* and *INV* represent the pre- and postconditions and the invariant, respectively; *X* represents the whole program; @ represents either *$O* or $< >$. See [41] for more details.

The proof rules for assignments and succession are quite straightforward. The former needs only substitution and the latter only modus ponens.

These proof rules look like the proof rules in Hoare logic, but these are rules within one logic language instead of being related to both logic and nonlogic languages. As a consequence, the soundness and (relative) completeness of these rules can be proved straightforwardly within the framework of first-order TL. In comparing this with the complicated procedure of justification for Hoare logic, the advantage is obvious. In [41], the detailed proofs of the following theorems are given:

Theorem 1. The proof rules for case, loop, assignments, and conditional and succession statements are all derivable in the first-order temporal logic (TL).

Theorem 1 has only given justification for soundness of these proof rules. As for completeness, since first-order TL is not complete, only relative completeness can be proved for XYZ/E. See [41] Theorem 4. The following theorem is one which concerns the relationship between programs of XYZ/E and XYZ/SE [7].

Theorem 2. There is a CASE tool called XYZ/E__SE based on an interactive method that transforms a given XYZ/E program into its structured higher-level form represented by XYZ/SE.

2.4. Declarations and Programs

It is easy to find that a unit is still not sufficient to constitute a program, because all the variables used in it are not declared. For this purpose, declaration for input, output, and input-output parameters, as well as local variables, are declared with brackets around them:

```
%inp[a1:X1;...;ak:XK]
%outp{b1:Y1;...;bj:Yj]
%iop[c1:Z1;...;cm:Zm]
%var[d1:W1;...;dn:Wn]
```

The declarations of parameters and variables are grouped together by conjunction and appear either as conjuncts in the "Where-part" or in front of a program separated with ";." For example, Examples 1 and 4 can have

```
%inp[m:I]^%outp[f:I]^%var[z:I;j:I]
%inp[a:I;b:I]^%outp[x:I]^%var[z:I;y:I]
```

as the conjunct of their "Where-parts," respectively.

There are declarations other than those indicated above. In an XYZ/E program, there are declarations of channels, procedures, processes, production rule units, macros, and packages. They are enclosed with similar brackets headed with different signals such as "%chn," "%proc," "pros," "%prod," "%mac," "%pack," etc., respectively. These declarations are put in front of a program separated with ";" and followed by the main algorithmic part of the program, which is enclosed by a pair of brackets with "%alg" as its signal. A program has the following form:

```
P(par)==[ ][declarations;
            %alg[...]]
            Where B                                    (15)
```

Similarly, a recursive procedure can be defined as follows:

```
P(n;par)= =[ ][declarations;
              %alg[lb=P_START=>...;
                   lb=l1^n=0=>...;
                   lb=l1^n/=0=>...;
                   ...
                   lb=li=>P(n-1;par);
                   lb=li+1=>...;
                   ...
                   lb=lk=>RETURN]
              Where B                                  (16)
```

here P is the name of this procedure; "*par*" represents the parameters of this procedure represented as those in conventional languages; "*RETURN*" is a special label which is an abbreviation of "*POP(rst)*"; and *rst* is a recursive stack which stores return labels. A call of this procedure is represented as:

```
lb=lj=>P(r; par);                                      (17a)
```

This is in fact an abbreviation (i.e., macro) of the following two CEs.

```
lb=lj=>$O(finps)=(ainps)^$O(rst)="li+1"^$Olb==P-START;
lb=lj+1=>$O(aoutps)=(foutps)^$Olb=NEXT;                (17b)
```

here "*finps*," and "*foutp*" represent formal inputs and outputs, respectively, and "*ainps*," "*aoutps*" are actual inputs and outputs. The following example shows how to represent recursive procedures of factorial in XYZ/E.

Example 5. Recursive procedure of factorial:

```
fact(n; %iop/a:I) = =
  [][%var[w:I];
    %alg[lb=fact_START= > $Olb=l1;
          lb=l1^n=0= > $Oa=1^$Olb=l3;
          lb=l1^n/=0= > fact(n-1;w/a);
          lb=l2= > $Oa=n*w^$Olb=l3;
          lb=l3= > $Olb=RETURN]]
```

2.5. Concurrency and Nondeterminacy

Convenience in representing concurrent programs, distributed or with shared memory, and their properties (in particular, safety and liveness properties) is one of the basic concerns of the design of XYZ/E. In order to achieve this goal, some conventions are adopted.

1. Each process, say Pri, has its own control variable lbi.
2. In order to represent nondeterminacy and concurrency, there are other extended CE forms:

$$lb=yi\,{}^\wedge P=>@((Qj1\,{}^\wedge lbj1=zj1)\,{}^\wedge \ldots {}^\wedge(Qjk\,{}^\wedge lbjk=zjk) \tag{18}$$

$$lb=yi\,{}^\wedge P=>@(Qj1\,{}^\wedge lbj1=zj1)\,{}^\wedge \ldots {}^\wedge @(Qjk\,{}^\wedge lbjk=zjk) \tag{19}$$

$$lb=yi\,{}^\wedge P=>@(Qj1\,{}^\wedge lbj1=zj1)\$V\ldots\$V@(Qjk\,{}^\wedge lbjk=zjk) \tag{20}$$

$$lb=yi\,{}^\wedge P=>@(Qj1\,{}^\wedge lbj1=zj1)\$V'\ldots\$V'@(Qjk\,{}^\wedge lbjk=zjk) \tag{21}$$

Let Prm represent the instantiation of the process prm declared in the front of the program, (18)–(21) can also be represented as

$$lb=yi\,{}^\wedge P=>@(Prj1\,{}^\wedge \ldots {}^\wedge Prjk) \tag{22}$$

$$lb=yi\,{}^\wedge P=>@(Prj1)\,{}^\wedge \ldots {}^\wedge @(Prjk) \tag{23}$$

$$lb=yi\,{}^\wedge P=>@(Prj1)\$V\ldots\$V@(Prjk) \tag{24}$$

$$lb=yi\,{}^\wedge P=>@(Prjk)\$V'\ldots\$V'@(Prjk) \tag{25}$$

here @ is $\$O$ or $<>$; V' is exclusive disjunction. The difference between (18–21) and (22)–(25) is that the former represent concurrency or nondeterminacy for steps, but the latter do that for processes. It is easily seen that to represent parallelism for one step by (18) is quite suitable, but to represent parallelism for a group of processes by (22) would violate the semantics of concurrent processes which involve nondeterminacy in the execution

of steps. (19) and (23) both represent fairness (note that in these two cases, @ can only be < >). (20) and (24) both represent nondeterminism, in which the components can occur sequentially or concurrently. (20) is the exact model of the nondeterministic concurrency involved in the steps of the concurrent processes, but to represent the nondeterministic concurrency of processes by (24) is too weak because according to this model some processes may never be executed and it does not represent the nondeterminism involved in the steps. In contrast to them, (21) and (25) represent only sequential nondeterminacy as guarded commands which are represented in XYZ/E with a special form called "select statement":

$$lb=yi\hat{}P=>!![Con1\,|\,>ExeAct1,\ldots,Conk\,|\,>ExeActk] \tag{26}$$

which is synonymous to

$$lb=yi\hat{}P=>\$O[Con1\hat{}ExeAct1\$V'\ldots\$V'Conk\hat{}ExeActk] \tag{27}$$

here we assume all the forward labels in ExeActi are "EXIT."

In order to represent concurrency in XYZ/E corresponding to the realistic concept of concurrency used in conventional programming, a statement called parallel statement is introduced. It has two different forms:

$$lb=yi\hat{}P=>\,||\,[Prj1,\ldots,Prjk] \tag{28a}$$

For the sake of representing the nondeterminacy involved in the steps of the concurrent processes in (28a), to specify its semantics with the following CE is suitable:

$$lb=yi\hat{}P=>[\;](Prj1\$V\ldots\$VPrjk) \tag{28b}$$

But it has one flaw that in (28b) some Prjq may never be executed. In order to remedy this flaw, we always make an assumption of the form (23) on (28b). It is as follows:

2.5.1. Fairness Assumption for Parallel Statements. For parallel statement of the form (28a), the following property of fairness is assumed:

$$lb=yi\hat{}P=><>Prj1\hat{}\ldots\hat{}<>Prjk \tag{29}$$

Although (28b) is a good specification for (28a). It is not executable. If we desire to represent the semantics of (28a) in an executable form, its semantics had better be represented in the following way:

$$\begin{aligned}&lb=yi\hat{}P=>\$Olb1=h1\hat{}\ldots\hat{}\$Olbk=hk;\\&lb1=h1\{\,|\,Prj1'\,|\,\}\$Olb1=EXIT;\\&\ldots\\&lbk=hk\{\,|\,Prjk'\,|\,\}\$Olbk=EXIT\end{aligned} \tag{30a}$$

here *Prjq'* for $q=1,\ldots,k$ are the result of replacing every CE in Prjq by its corresponding nondeterministic form, i.e., to replace

```
lbq=lqd^M=>@(Rq^lbq=lqe)
```

with

```
lbq=lqd^M=>!![$T¦T¦>@(Rg^lbq=lqe), $T¦>$Olbq=lqd]
```

Note that in order to simplify the notations, in subsequent discussion we often represent CEs in ordinary form and assume the user understands how to change them into nondeterministic form.

Although the semantics of this group of CEs is very close to the meaning of parallel statement in conventional programming, there is still an ambiguity. For in (30), the execution control would leave this unit as soon as the earliest EXIT of some *Prjr* occurs, i.e., the execution would not stay to wait for the EXITs of other *Prjqs* for q/=r. In order to represent the fact that all EXITS of *Prijq's* must await each other until they leave the statement simultaneously, the parallel statement (28a) should be represented as the following:

```
lb=yi^P=>$Olb1=h1^...^$Olbk=hk;
lb1=h1{¦Prj1'¦}$Olb1=z1;
 ...
lbk=hk{¦Prjk'¦}$Olbk=zk;
lb1=z1=>$Obb1=tt^$Olb=yi';
 ...
lbk=zk=>$Obbk=tt^$Olb=yi';
lb=yi'^~(bb1=tt^...^bbk=tt)
        =>($Olb=yi'
          ¦$Obb1=ff^...^$Obbk=ff^$Olb=EXIT)                    (30b)
```

In order to represent specification of functionality, the nondeterminacy of steps represented by (28b) is negligible and the details of the executable code of each *Prjq* can be replaced with its pre-post conditions. Thus each *Prjq* can be represented as

```
prejq-><>(postjq^lbq=zq)                                        (31)
```

In accordance with (23), the specification of the parallel statement (30b) can be represented as

```
lb=yi^P=>(<>(prej1-><>(postj1^lb1=z1))^...
           ^<>(prejk-><>(postjk^lbk=zk)))
lb1=z1=>...^$Olb=yi';
 ...
lbk=zk=>...^$Olb=yi';
lb=yi'...=>...$Olb=EXIT]
```

or to move the operator < > outside the parentheses (...) into it, the formula becomes

$$
\begin{aligned}
&\mathrm{lb{=}yi{=}{>}[[\ ]prej1{-}{>}{<}{>}{<}{>}(postj1\,\hat{}\,lb1{=}z1))\,\hat{}\,\ldots} \\
&\qquad \mathrm{\hat{}([\ ]prejk{-}{>}{<}{>}{<}{>}(postjk\,\hat{}\,lbk{=}zk))} \\
&\ldots \\
&\mathrm{lb{=}yi'\ldots{=}{>}\ldots\$Olb{=}EXIT]}
\end{aligned}
$$

which is equivalent to

$$
\begin{aligned}
&lb{=}yi{=}{>}[(prej1{-}{>}{<}{>}(postj1\,\hat{}\,lb1{=}z1))\,\hat{}\,\ldots \\
&\qquad \hat{}(prejk{-}{>}{<}{>}(postjk\,\hat{}\,lbk{=}zk)) \\
&\ldots \\
&lb{=}yi'\ldots{=}{>}\ldots\$Olb{=}EXIT]
\end{aligned} \tag{32}
$$

because *prejq's* are FOL formulas. In order to simplify the abbreviation form, we assume (32) is abbreviated as follows:

$$
\begin{aligned}
&lb{=}yi{=}{>}[(prej1{-}{>}{<}{>}(postj1)\,\hat{}\,\ldots \\
&\qquad \hat{}prejk{-}{>}{<}{>}(postjk))\,\hat{}\,\$O(Wait\,\hat{}\,lb{=}EXIT]
\end{aligned} \tag{33}
$$

Here (33) can be considered an axiomatic system representing the specification of (28a), which is the conjunction of the specification of its components (33) also implies the following:

$$
\begin{aligned}
&lb{=}yi\,\hat{}\,(prej1\,\hat{}\,\ldots\,\hat{}\,prejk) \\
&\qquad {=}{>}{<}{>}(postj1\,\hat{}\,\ldots\,\hat{}\,postjk\,\hat{}\,lb{=}EXIT)
\end{aligned} \tag{34}
$$

provided there are no shared variables which can give rise to contradictions in "$postj1\,\hat{}\,\ldots\,\hat{}\,postjk$."

The proof rules for parallel statement are:

$$
\frac{\begin{array}{l}\mathrm{NA,\ Prj1} \mid - \ \mathrm{prej1{-}{>}{<}{>}(postj1\,\hat{}\,\$Olb{=}EXIT)} \\ \qquad\qquad \ldots \\ \mathrm{NA,\ Prjk} \mid - \ \mathrm{prejk{-}{>}{<}{>}(postjk\,\hat{}\,\$Olb{=}EXIT)}\end{array}}{\mathrm{NA},\ [w]1 \mid - \ [x'']} \quad \text{and} \quad \frac{\text{Similarly}}{\mathrm{NA},\ [w]1 \mid - \ [x''']}
$$

The proof rule for select statement can be formulated similarly. As for the real-time computing, XYZ/E contains a sublanguage for that purpose. To discuss that problem in this paper would make it too long. With this in mind, the proof rules for the parallel statement and the select statement can be formulated straightforwardly [42].

2.6. Communication

Messages are transmitted from one process to another through channels which can be declared dynamically in XYZ/E. As CSP, there are two commands related to channels for communication of messages between processes, i.e., the output command and input command of the following forms:

$$lbr=r-ST=>Ch?x\,\hat{}\,\$Olbr=r-EX \tag{35a}$$

$$lbs=s-ST=>Ch!y\,\hat{}\,\$Olbs=s-EX \tag{35b}$$

here *Ch* is the name of a channel from the output process to the input process; *y* is the output variable of the former; and *x* is the input variable of the latter. Note that in (35a) and (35b) the part *Ch?x* and *Ch!y* can be extended to $Chl?xl\,\hat{}\,\ldots\,\hat{}\,Chm?xm$ and $Chl!xl\,\hat{}\,\ldots\,\hat{}\,Chm!m$, respectively. Corresponding to commands (35a) and (35b) there are also input and output requests of the following form:

$$lbr'=r'-ST\,\hat{}\,Ch?x=>\$Olbr'=r'-EX \tag{35c}$$

$$lbs'=s'-ST\,\hat{}\,Ch!y=>\$Olbs'=s'-EX \tag{35d}$$

These formulas (35a)–(35d) are, in fact, abbreviations (i.e., preestablished macro-patterns) which represent the corresponding logic semantics in detail for synchronization communication. In order to do that, a record to store the information and to represent the conditions is needed. It can be defined as follows:

$$\begin{aligned} C(\,IPros,\ OPros):\ R(\,&buf:\ MessageType;\\ &rreq,\ sreq:\ B;\\ &otheritem:\ Type\,) \end{aligned} \tag{35e}$$

(35a) and (35b) are, in fact, the abbreviation of the following two units, respectively.

$$\begin{aligned} &[lbr=r-ST=>\$OCh-rreq=tt\,\hat{}\,\$Olbr=l1;\\ &\ lbr=l1\,\hat{}\,Ch-sreq=tt=>(\$Olbr=l2\,\vert\,\$Olbr=l1);\\ &\ lbr=l2=>\$Ox=Ch-buf\,\hat{}\,\$OCh-rreq=ff\\ &\qquad\qquad\hat{}\,\$Olbr=r-EX] \end{aligned} \tag{35f}$$

$$\begin{aligned} &[lbs=s-ST=>\$OCH-sreq=tt\,\hat{}\,\$OCH-buf=y\,\hat{}\,\$Olb=m1;\\ &\ lbs=m1\,\hat{}\,Ch-rreq=tt=>(\$Olbs=m2\,\vert\,\$Olbs=m1);\\ &\ lbs=m2=>\$OCh-sreq=ff\,\hat{}\,\$Olbs=s-EX] \end{aligned} \tag{35g}$$

(35c) and (35d) can be represented in more detailed concrete semantics similarly. These details are masked on specification level to the users. The users only need to write these CEs in accordance with the macro-patterns modified by the replacement of the corresponding actual parameters. Of course, these macros can also be realized as procedure call or process call. It depends on the taste of the users. (Thanks to Professor A. Pnueli for his elegant alternative to represent the executable semantics represented by those equations on pp. 116–119.)

As is well known, there are two types of communication. Beside the communication of the synchronization type represented by (35a)–(35d) shown above, there is communication of the asynchronization type. It can be represented similarly to (35a)–(35d) on the specification level. But their semantics are different. In its implementation, a queue is used to store the messages. Waiting for continuation of execution in input and output commands happens according to "being full" or "being empty" of the queue. Consequently, the record defined for the channel and the details in (35a)–(35d) must be changed accordingly. In

XYZ/E, channels as well as processes can be instantiated dynamically by passing actual parameters to it, this is a feature to facilitate dynamic binding. See [42] for the proof rules of these communication commands.

Example 6. A parallel statement to evaluate the maximum value of two integers m and n.

```
lb=max=>¦¦[P1, P2]
```

here

```
P1==[lb1=P1-START=>$Olb1=l1;
        lb1=l1=>C?x^$Olb1=l2;
        lb1=l2=>!![m>=x¦>$Olb1=l3, m=<x¦>$Olb1=l4,
                    $T¦>$Olb1=l2];
        lb1=l3=>D!m^$Olb1=STOP;
        lb1=l4=>D!x^$Olb1=STOP]

P2==[lb2=P2-START=>$Olb2=k1;
        lb2=k1=>C!n^$Olb2=k2;
        lb2=k2=>D?y^$Olb2=STOP]
```

Not only the distributed system, but also the concurrent processes with shared memory can be represented in XYZ/E very similarly to that in conventional programming languages.

2.7. Pointers and the Dynamic Binding Problem

Dynamic binding is a feature required by the users but very difficult to deal within a formal system. The most convenient mechanism of conventional languages to represent dynamic binding is the pointers. As is well known, to represent and verify a formal formula with pointers occurring in it is notoriously hard. In XYZ/E, since no concept of "address" can be allowed in a logic language, we consider a pointer, say f, of type P(X) being a "variable" which can "represent" any variable of type X dynamically. The fact that f represents v is represented by a formula of the following form:

$Of=:v (36a)

It is assumed that after execution of (36a), manipulation on the pointer f in the program is synonymous to manipulation on that variable v, unless a new pointer-representation to change a new represented variable to that pointer f. Obviously, this is just the role a pointer plays in conventional programming. The problem is to give a sound justification for such kind of "*pointer-representation*" from a logical point of view. In fact, the pointer in XYZ/E can be considered a notation to represent the so-called "*fixed-point function*" with the parameter part omitted, e.g., the above f is the function

f: lambda x:X .x (36b)

or

$$f(x)=x \quad (36c)$$

Thus, so-called pointer-representation (36a) is, in fact, to pass an actual parameter to the unique formal parameter of the function, i.e.

$$f\colon \mathit{lambdax.x} \backslash v = v \tag{36c}$$

or

$$f(v/x) = v \tag{36d}$$

As for the proof rule for pointers in this sense, it is very similar to that of assignment. To define pointers in this way not only can preserve all nice properties they ought to have, but also can avoid all those difficulties in verification which are inevitable in conventional programming. To make use of pointers in this sense, the dynamic binding problems can be treated in XYZ/E as naturally as in conventional programming languages. But notice now the pointer becomes a mechanism which has sound logical basis and can be subject to logical reasoning.

2.8. Packages

In XYZ/E, encapsulated modules are represented as packages. In order to enhance flexibility and extensibility, packages are separated from the main block of the program. All packages can only communicate with each other and with the main block by means of import-export mechanism. All concurrent processes can only be declared and used in the main block of the program. Hence, the main structure looks like this:

```
Program ::= %PRO ProgName==MainBlock; Package;...;Packages.

MainBlock ::= [ ][ ImportDeclarationPart;
                   ExportDeclarationPart;
                   TypeDeclarationPart;
                   GVariableDeclarationPart;
                   SharedVariableDeclarationPa;
                   ChDeclarationPact;
                   ProDeclarationPart;
                   ProgramBody ]
              WherePart

Package ::= %pack[ PackageName:
                   ImportDeclarationPart;
                   ExportDeclarationPart;
                   TypeDeclarationPart;
                   GvariableDeclarationPart;
                   SharedVariableDeclarationPart;
                   OperationDeclarationPart ]
```

In XYZ system, there is a base of reusable programs. One kind is reusable packages among which are all those used to define the basic types.

2.9. Models and Implementation

2.9.1. Kripke Model. As usual, a model (I, alpha, delta) contains a global interpretation I, a global assignment alpha, and a state sequence delta. I gives the global interpretation for all the constants, functions and predicates. Assignment alpha assigns values for every global variable. A sequence delta = S0, Sl, . . . (for future-tense TL) is an infinite sequence where each state Si assigns values to local variables and local propositional variables, etc. In short, at each state, FOL holds. This is the model on which XYZ/E is based.

2.9.2. Lucid Model. The Lucid model can be considered a geometrical image of the Kripki model. In the Lucid model [1], each temporal variable is considered a vector infinitely extensible with time. At each time i, the value space of all temporal variables constitutes a plane which correspond to the state Si in the Kripki model. Different layers of these planes correspond to the transition of states. One of the compilers of XYZ/BE (the executable sublanguage of sequential XYZ/E) is exactly implemented according to this geometrical pattern. As we have shown, all CEs of XYZ/E programs are always executed according to the pattern of transition from present time to next time and at each time only the values of a few temporal variables are changed. Consequently, at each step of transition, in order to remember the history, we need only to store those values of the changed variables into the external file which is divided into time sections to facilitate retrieval. This constitutes the basic model of the implementation of XYZ/BE. Indeed, the process is time-consuming, but if remembering history is needed in a program, spending so much time seems inevitable.

2.9.3. Von Neumann Model. If retrieving history is not necessary in a computation, then the time-consuming process of storing the values of the changed variables into the external file at each step can be avoided. This is just what the Von Neumann architecture has modeled. The future-tense TLL exactly reflects the need for this model. In XYZ/E, all sublanguages except XYZ/PPE fit this requirement. This is the reason why all those executable sublanguages of XYZ/E, such as XYZ/BE and XYZ/DE can be implemented as efficiently as conventional languages [8].

The following theorem shows the extent of expressiveness of XYZ/E.

Theorem 3. There is an effective method to transform a TL WFF into an XYZ/E program.

3. The CASE Tools

The CASE tools in XYZ system can be divided into five groups:

1. **The graphic tools for structured design:**

    - XYZ/DFD, a tool for structured design of (enhanced) DFD (data-flow diagram)
    - XYZ/CFC, a tool for structured design of XYZ chart
    - XYZ/PAD, a tool for structured design of (enhanced) PAD

Here the so-called enhanced DFD is a kind of data-flow diagram combined with synchronization or asynchronization mechanism; XYZ chart is the graphic representation of XYZ/E CE, combined with the transition of Petri nets to represent the synchronization or asynchronization mechanism; enhanced PAD is PAD (a kind of structured flowchart directly corresponding to XYZ/SE statements), combined with Petri nets transition in the above sense. All these graphic tools can be used to generate the XYZ/E program automatically from the graphs designed at each step.

2. **The tools to support the method for stepwise refinement according to decisions**
   It includes a tool to transform the informal description of the decision into formal specification (XYZ/SDL), a tool to record and to retrieve the history of design decisions (XYZ/SPEC), a tool to check the consistency of semantics of specifications of consecutive steps (see below), and a base of reusable programs (XYZ/REUSE), etc. The methodology supported by this system is more powerful than the structured design method supported by XYZ/DFD and XYZ/PAD. It can be used to design so-called "wicked problems."
3. **The tools for validation and consistency-checking:**

   - XYZ/VERI, a tool to verify the validity of XYZ/E programs and to check the semantic consistency of XYZ/E formulas
   - XYZ/PROT, a tool to execute the XYZ/E specifications with a Prolog system. It is used as a rapid-prototyping system in XYZ system
   - XYZ/RULE, an enhanced XYZ/PROT, which can be used as a rapid-prototyping system for designing XYZ/E concurrent programs

4. **The tools for languages transformation:**

   - XYZ/CCSS, a tool to transform the conventional languages into XYZ/E. The operational semantics of the source program are represented by means of a formal metalanguage XYZ/ML, the result of execution of the replacement rule represented by XYZ/ML is the target program represented by XYZ/E. The static semantics is represented by attribute grammar.
   - XYZ/E-SE, a tool to transform any unstructured XYZ/E program into a structured XYZ/SE program
   - XYZ/TL-E, a tool to transform any linear time temporal logic WFF into an XYZ/E program
   - XYZ/E-X, the compilers to transform the XYZ/E programs into other conventional languages, e.g., XYZ/E-C

5. **The tool for configuration management and version control of XYZ systems**
   It also serves as the integration base and the main entrance of the XYZ system.

Obviously, such a big system cannot be introduced in one paper. We choose XYZ/DFD as an example of the CASE to be illustrated here. In order to explain the background of this tool, it might be necessary to give some explanation of the methodology it is supposed to support. Of course, the DFD graph designed by this tool can play a role similar to a

data-flow diagram used in the so-called structured design. But there is something more than an ordinary data-flow diagram. In XYZ/DFD, the connecting line (i.e., the edge) between two nodes can serve to represent the channel between two communicating processes which are represented by the nodes on both sides of the line. In order to let the DFD graph play the role of a graphic representation of communicating processes, a kind of synchronization or asynchronization mechanism is implicitly inserted on both sides of the connecting line. We suppose the topmost layer of the software system to be designed by XYZ system is such a kind of distributed system. In the process of designing such a software system, generally speaking, XYZ/DFD is used to decompose the nodes step by step into sub-DFD graphs until a stage is reached on which all the nodes in it are no longer in need of being represented as distributed processes again. After this step, the graphic tools XYZ/CFC or XYZ/PAD can be used to decompose the nodes further into a graph to represent the routines consisting of sequential procedures and/or concurrent processes with shared memory. I hope this brief illustration can leave the users an impression of the pattern of the software systems these three graphic tools are supposed to design.

Example 7. To solve a*x*x+b*x+c=0 with a/=0 and b*b−4*a*c> =0

Step 1. Draw diagram with XYZ/DFD as Figure 1.

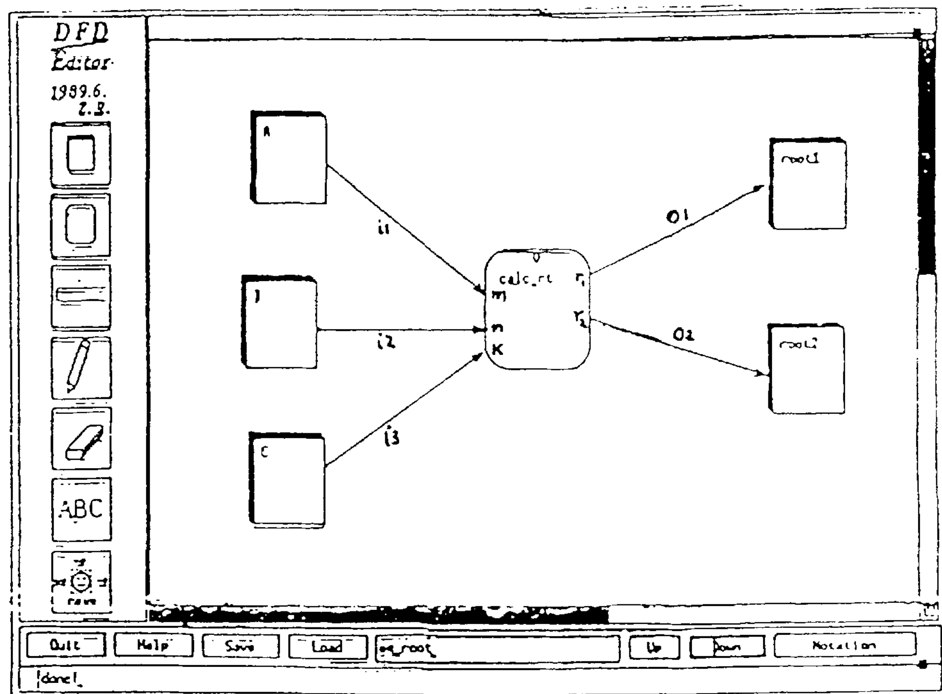


Figure 1.

Step 2. Give the semantics (pre-post conditions) of P0.

```
PRE0: a/=0^b*b-4*a*c>=0,
POST0.1: x1=(-b+sqr(b*b-4*a*c))/2*a,
POST0.2: x2=(-b-sqr(b*b-4*a*c))/2*a
```

Step 3. Generate of the XYZ/E specification from Figure 1.
Here sp0 is the formal process declared at the beginning of the program and aspO0 is its instantiation.

```
sp0(%inp/m,n,k: I; %outp/r1,r2: F;
    %chn/i1(Env,P0), i2(Env,P0), i3(Env,P0), o1(P0,Env),
    o2(P0,Env):SYN)==
  [][lb0=P0-ST=>i1?m^i2?n^i3?k^$Olb0=l1;
    lb0=l1^~PRE0=>($Olb0=P0-ST|<>(POST0^lb0=l2));
    lb0=l2=>o1!r1^o2!r2^$Olb0=P0-ST]

asp0==sp0(%inp/ a/m,b/n,c/k; %outp/ x1/r1,x2/r2;
                %chn/ j1/i1,j2/i2,j3/i3,q1/o1,q2/o2)
```

Step 4. Decompose the process node of Figure 1 into subgraph in which the node P0 is decomposed into P00, P01, P02 and P03 as shown in Figure 2.

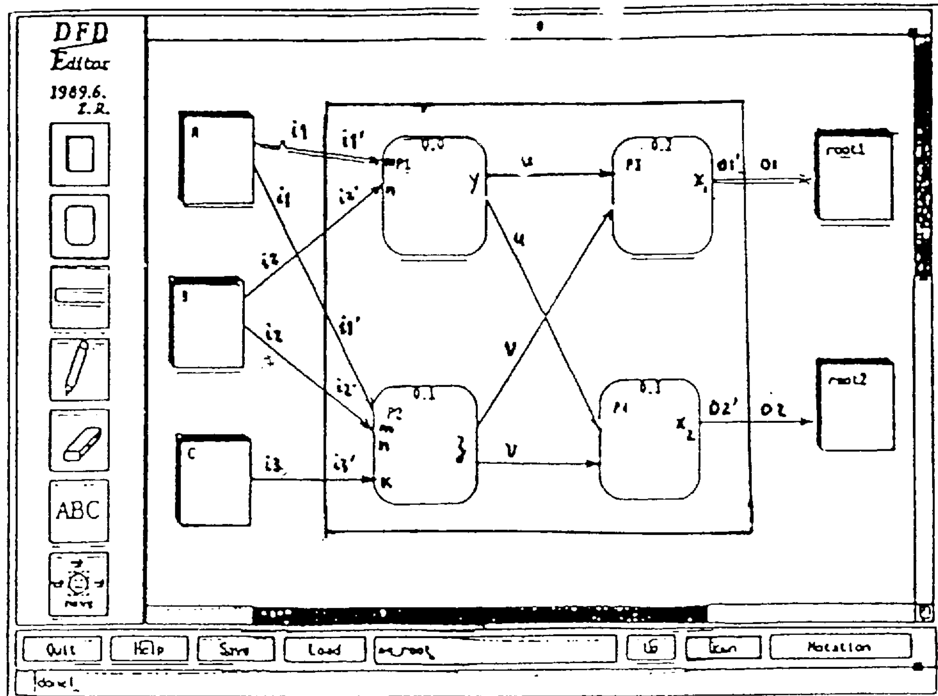


Figure 2.

Step 5. Give the semantics of P00, P01, P02, P03.

```
PRE00: a/=0,
POST00: y=-b/2*a;
PRE01: a/=0,
POST01: z=sqr(b*b-4*a*c)/2*a;
PRE02: $T,
POST02: x1=y+z;
PRE03: $T,
POST03: x2=y-z.
```

Step 6. Generate the XYZ/E specifications for P00, P01, P02, and P03 as sp00, sp01, sp02, and sp03 and their corresponding instantiations asp00, asp01, asp02, and asp03. Here only sp00 is given as an example:

```
sp00(%inp/ m,n: I; %outp/ y: F;
     %chn/ i1'(P0,P00),i2'(P0,P00),u(P00,Env):SYN)==
    [lb00=ST-P00=>i1  i1'?m^i2-i2'?n^lbr=100;
     lb00=101~~PRE00=>($Olb00=ST-
     P00|<>(POST00^lb00=102));
     lb00=102=>u!y^$Olb00=ST-P00]

asp001==sp00(%inp/ a/m,b/n; %outp/ y/y;
             %ch/ j1-j11'(Env,P001)/i1-i1',
             j2-j21'(Env,P001)/i2-i2', u1(P00,P3)/u)

asp002==similar to asp001 except to replace
             u1(P00,P3) with u2(P00,P4) and to replace
             J1-J11'(Env,P001), j2-j21'(Env,P001) with
             j1-j12'(Env,P002), j2-j22'(Env,P002)
             respectively.
```

sp01, sp02, and sp03 are similar. The whole decomposed program is structured as:

```
asp0'==[][lb0'=ST-P0'=> || [asp00,asp01,asp02,asp03]]

      Where <>asp00^<>asp01^<>asp02^<>asp03
```

In fact, the "Where-part" could be omitted because of the fairness assumption.

As we have pointed out, this process of decomposition could be carried on until every node can be represented with a sequential procedure or a group of concurrent processes with shared memory. To design these two kinds of program in a structured way, XYZ/CFC or XYZ/PAD are better graphic tools. As for this example, because asp00, . . . ,asp03 are too simple, it is not necessary to use tools as powerful as XYZ/CFC and XYZ/PAD [8]. However, these tools are useful in such a situation.

There is another problem left to consider. In what sense could we guarantee the consistency of semantics between two consecutive steps of decomposition? If the semantics of a node can be represented with pre-post condition with the form (6) shown above, the consistency can be easily validated by showing that from the XYZ/E program generated from the DFD graph after decomposition, the pre-post condition before decomposition can be deduced. In XYZ system, there are two kinds of tools provided for this purpose, one is to validate the semantic consistency by verification with the tool XYZ/VERI, another kind of tool is used to execute these two formulas with some Prolog-like systems called XYZ/PROT and XYZ/RULE and to compare the results of the execution to check their semantic consistency. The latter way of validation is called "rapid-prototyping" in XYZ system. Unfortunately, according to our experience, generally speaking, it is rather difficult to find one pre-post condition to represent the specification of a distributed system. But on the other hand, to represent the specification of such a system with a group of pre-post conditions connected with conjunction seems quite natural. It is very close to specify a system with a Prolog-like program. As a consequence, to decompose the system means to partition the specification (i.e., a conjunction of pre-post conditions) into conjunction of subconjunctions (of pre-post conditions). For example, to decompose $(C1 \wedge \ldots \wedge Ck)$ into $(Ci1 \wedge \ldots \wedge Cim) \wedge (Cim+1 \wedge \ldots \wedge Cim+l) \wedge \ldots \wedge (Cin \wedge \ldots \wedge Cim+r)$ here $Ci1, \ldots, Cim+r$ and $C1, \ldots, Ck$ are identical up to associativity and commutivity. Obviously the semantic consistency of decomposition in this sense is trivially preserved.

Now we give a brief introduction to XYZ/CFC or XYZ/PAD [8].

The basic graphic items in XYZ/CFC are following:

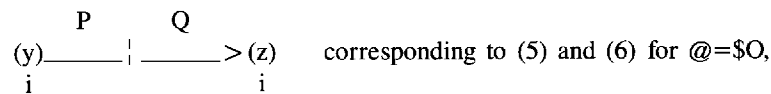


Figure 3(a).

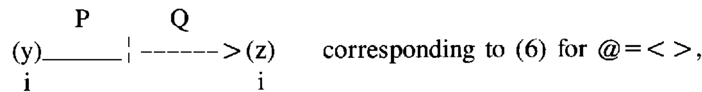


Figure 3(b).

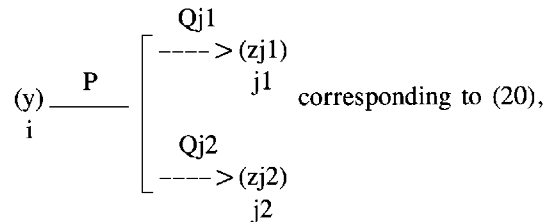


Figure 3(c).

In the concurrent cases, if in some process there are variables in the shared memory occurring in its conditions, it is possible that the truth of that condition depends upon the result of execution of other processes. When this situation happens, the process must keep waiting until the execution of its partner process changes the values of those variables. In order to represent such a situation, some notation similar to the transition of Petri nets is used in XYZ/CFC as follows:

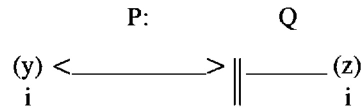


Figure 3(d).

In case any side does not wait for its partner, then change the waiting arrow <____> into the transition arrow ____> passing through the transition bar ‖, which can serve to represent asynchronized concurrency.

XYZ/PAD is the graphic editor corresponding to XYZ/SE as XYZ/CFC to XYZ/BE. Similar to the Japanese PAD, its basic forms which correspond to loop statement, case statement, and if-then statements are shown in Figure 4(a), (b), and (c) respectively; Figure 4(d) corresponds to Figure 3(d).

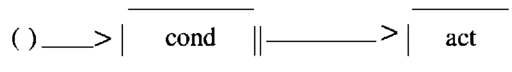


Figure 4(a).

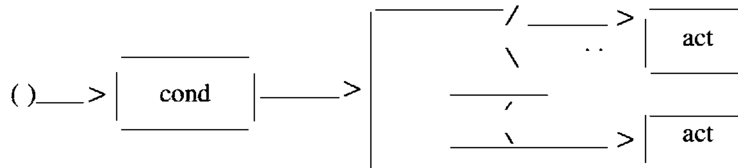


Figure 4(b).

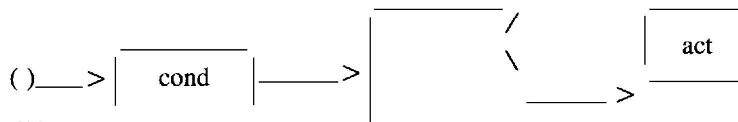


Figure 4(c).

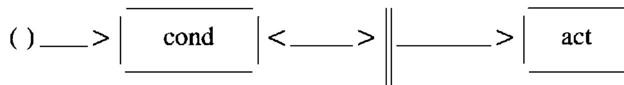


Figure 4(d).

Here the action box can be replaced not only with a PAD subgraph, but also with a specification.

These graphic tools are indeed very elegant and akin to most widely used programmer's tools. But unfortunately, they have a limitation: A basic condition of decomposibility is the so-called "well-structured property" in the so-called structured design, i.e., replacing any node of the graph with a subgraph cannot be allowed to have any side effects on any other part of the original graph. This requirement implies that in any step of decomposition, the architecture of the original graph must be preserved. Unfortunately, software development is a complicated process. A lot of software is too sophisticated to satisfy this condition. As [43, 44] pointed out, there are quite a few software problems that are intrinsically "wicked," i.e., even their specifications cannot be well-defined precisely at the very beginning and need refinement in the process of development. This fact means that in general, the process of software design is a sequence of decision making. XYZ system provides another group of tools to support a methodology for this purpose. They are XYZ/SPEC, XYZ/SDL, and XYZ/REUSE. It is beyond the bounds of this paper to introduce all these tools.

4. Related Work

The design of TLL XYZ/E started in the winter of 1979 when I was invited to visit Stanford. My host, Prof. J. McCarthy's tense logic language Elephant [22] was the first one of this kind I had ever encountered, and it influenced my thoughts. Both Elephant (its first version) and XYZ borrow from Lucid the idea of considering a variable as a vector that is extensible with time. The difference between XYZ/E and the first version of Elephant (its original version) is not only due to the fact that the former is based on TL and the latter on FOL, but also due to our divergence in orientation. In my understanding, Elephant is designed for programming with the remembrance of history while XYZ/E is designed for facilitating the representation of the methodology of software engineering. As for Lucid [1] its influence on XYZ/E is apparent in forming its semantic model and its use of next-time operator is also an important invention. In spite of all these, Lucid is still not a language based on logic and tense concept.

Moszkowski's Tempura [24] is indeed one of the earliest executable TLLs that has a close relationship with XYZ/E. Both are influenced by Manna and Pnueli's theory of TL. The difference between these two TLLs is also obvious:

1. They are based on different TL theories; Tempura is based on interval TL while XYZ/E on linear time.
2. Tempura is more oriented toward the semantic aspect of HLL, while XYZ/E is more oriented toward its relation with software engineering [29, 31]. Temporal Prolog [39] is another earlier TLL with some similarity to XYZ/E. Although XYZ/E also contains the rule form sublanguages, its efficient execution still depends upon its algorithmic sublanguages XYZ/BE and XYZ/DE. I think, this is a weakness in Temporal Prolog. Gabbay's PCFA [5], similar to Temporal Prolog, has quite a good theoretical basis and expressiveness in specification, but how to represent the sequential and communicating

processes which can be executed efficiently on a conventional machine is still a hard problem. By the way, allowing a past-tense operator to occur in the condition, although able to enhance readability, can make the execution very inefficient.

Chandy and Misra's Unity is another language sharing many common characteristics with XYZ/E. Both have the similar design goal of unifying programming: "Today, programming appears to be fragmented into increasingly esoteric subdisciplines, each with its priests, acolytes, and incantations. We believe that there is a unity underlying all programming; we hope to contribute to its appreciation" [3]. Furthermore, both Unity and XYZ/E choose unnested temporal operators to represent program properties, which can make the language more accessible to common users. But there are many major differences between these two languages:

1. I believe, as does Lamport [13], representing program and properties with one language is better than with different languages.
2. The stepwise refinement using union and superposition as a method seems too restrictive, especially if the small kernel language of Unity is brought into consideration.

There are a lot of graphic tools for program development [10], but in my understanding, very few can generate from the graph designed at each step both the specification and the executable programs of the same formal language, using a means where the semantic consistency can be checked either by verification or by rapid prototyping as naturally as XYZ/DFD, XYZ/CFC, and XYZ/PAD.

With regard to the methodology of stepwise refinement, according to design decisions made in XYZ/SPEC, Ergos [14] seems to contain some similar thoughts, but the formal basis is quite different. To the best of my knowledge, that system may still be far from finishing its design and implementation, so it is too early to compare these two systems.

As for the language transformation based on formal semantics, XYZ/CCSS is characterized by making a semantic-directed translator close to conventional compilers. In this aspect, most probably, there is still no other system of a similar nature.

As for the specialty of XYZ as a CASE tools system based on TLL and unifying various ways of programming [29, 31], I believe, XYZ might not only be the earliest one designed, but also until now, the only one implemented. As I pointed out at the beginning of this paper, this system is the result of an effort to keep some realistically good balance between rationalism and pragmatism, although it might not be very satisfactory to those who are used to either of these philosophical thoughts.

Although an experimental system of XYZ is available, this project is still not finished. Not only are a number of domain-specific CASE environments based on XYZ under investigation, but also the system itself is still subject to refinement and extension. The present system is implemented on SUNs and the total system in its present state includes 120,000 lines of C code.

Acknowledgment

XYZ system is a long-term project lasting more than 10 years. It has been sponsored by the Chinese National Science and Technology Research Fund, the Ministry of Electrical Engineering, and the Chinese Academy of Sciences. Also, during these years, I have been invited to visit many universities and institutes including Stanford, CMU, Maryland, ISSI, Bath, and Trondheim, etc. My hosts' thoughts have strongly influenced me in forming XYZ system. I must extend my deepest appreciation, in particular, to Profs. J. McCarthy, Z. Manna, R. Yeh, N. Habermann, and A. Solvberg. Many friends of mine have also been a great help and made contributions to this work, I can only enumerate a few names here: Profs. X. Ma, Y. Gu, C. Kung, Y. Feng, and H. Lin. During these years, there was a group of brilliant students working around me, it is this group who finished this research project. Some of their names can be found in the references and some are not. To all of them, I am very grateful.

References

1. E. Ashcraft, and W.W. Wadge, ''Lucid: A nonprocedural language with iteration,'' CACM, vol. 20, 1977.
2. H. Barringer, R. Kuiper, and A. Pnueli, ''Now you may compose temporal logic specifications,'' Proc. 16th ACM Symp. on Theory of Comp., 1984.
3. K.M. Chandy, and J. Misra, Parallel Program Design, Addison-Wesley, 1988.
4. Y. Feng, H. Lin, and Z. Tang, ''A proof system for temporal logic programs,'' R&D of Comp. no. 10, Beijing, 1985, (in Chinese).
5. D. Gabbay, ''A past tense condition and future tense action temporal logic language,'' draft, London, 1990.
6. A. Giacalone, et. al., ''Toward a formally-based programming environment,'' Integrated Interactive Computing System, 1983.
7. J. Gong, ''An interactive graphic CASE tool to transform a program into its well-structured form,'' to appear in J. of Softw., Beijing, 1992, (in Chinese).
8. J. Gong, R. Zhang, and C.S. Tang, ''XYZ/CFC & XYZ/PAD: Graphic editors for program design,'' to appear in J. of Softw., Beijing (in Chinese).
9. M. Hagiya, and T. Sakuragawa, ''Temporal prolog,'' Res. Inst. Math. Sci., Kyoto Univ., draft, Kyoto, 1984.
10. D. Harel, ''Statecharts: A visual formalism for complex system,'' Sci. Comp. Prog., no. 8, 1986.
11. C.H. Kung, and A. Solvberg, ''Activity modelling and behavior modelling of information systems,'' Proc. IFIP WG.8.1 Working Conf., North Holland, Amsterdam, 1986.
12. L. Lamport, ''What good is temporal logic,'' Proc. IFIP Cong., North Holland, Amsterdam, 1983.
13. L. Lamport, ''The temporal logic of action,'' draft, Palo Alto, 1991.
14. P. Lee, et. al., ''Research on semantically-based program design environment,'' Dept. Comp. Sci. CMU. Tech. Rep. CMU-CS-88-118, Pittsburgh, 1988.
15. R. Li, W. Zhang, and P. He, ''An introduction to Incaps system,'' Inst. Softw. Acad. Sin. Tech. Rep. No. IS-CAS-XYZ-91-2. To appear in J. of Comp. Sci. & Tech, no. 4, Beijing, 1992.
16. H. Lin, C. Gong, and H. Xie, ''Abstract implementation of algebraic specifications in a temporal logic language,'' J. of Comput. Sci. & Technol., vol. 6, no. 1, Beijing, 1991.
17. J. Liu, and X. Cai, ''An attributes evaluator based on partitioned grammar,'' Inst. Softw. Acad. Sin. Tech. Rep. IS-AS-XYZ-88-9, Beijing.
18. T. Liu, and C.S. Tang, ''Semantic specification and verification of data flow diagram,'' J. of Comp. Sci. & Tech., vol. 6, no. 1, Beijing, 1991.
19. T. Liu, C.S. Tang, and R. Zhang, ''A formal approach to the semantic specification of data flow diagrams,'' Proc. COMPSAC'91, Tokyo, 1991.

20. Z. Manna, ''Verification of sequential programs, temporal axiomatization,'' Theoretical Foundations of Programming Methodology, North Holland, Amsterdam, 1982.
21. Z. Manna, and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*. Specification. Springer-Verlag: New York, 1991.
22. J. McCarthy, ''Elephant'' draft, Stanford, USA, 1979.
23. X. Miao, and C.S. Tang, ''A methodology and environment for stepwise refinement according to design decisions,'' Inst. Softw. Acad. Sin. Tech. Rep. IS-CAS-XYZ-89-5, 1989.
24. B. Moszkowski, Executing temporal logic programs, Cambridge Univ. Press, Cambridge, MA, 1986.
25. B. Mu, Y. Xiao, and C.S. Tang, ''A compiler-compiler source-to-source transformation system,'' to be submitted to J. of Softw., Beijing, (in Chinese).
26. A. Pnueli, ''Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends,'' LNCS 224 Springer-Verlag: Berlin, 1986.
27. M. Pong, Y. Zhang, H. Xu, and J. Ding, ''OOMMS: A module management system based on object-oriented model,'' Inst. Softw. Acad. Sin. Tech. Rep. IS-CAS-XYZ-92-4, Beijing, 1992.
28. W. Shen, and C. Zhao, ''A compiler of distributed XYZ/E,'' to be submitted to J. of Softw., Beijing (in Chinese).
29. C.S. Tang, ''A programming development environment conforming to various ways of programming,'' R&D of Comp., vol. 11, 1982, Beijing (in Chinese).
30. C.S. Tang, ''Toward a unified logic basis for programming languages,'' Dept. Comp. Sci. Stanford, Tech. Rep. No. STAN-CS-81-865; revised version in Proc. IFIP Congress., North Holland, Amsterdam, 1983.
31. C.S. Tang, ''XYZ: A programming development environment based on temporal logic,'' Programming Languages and Systems, North Holland, Amsterdam, 1983.
32. C.S. Tang, ''A temporal logic language for behavior modelling of information and expert system,'' Knowledge & Data, North Holland, Amsterdam, 1986.
33. C.S. Tang, ''To unify programming with logic,'' I, Proc. of PREVLDB Intl. Symp. Beijing, 1986; II, Proc. Intl. Symp. Softw. Eng. Beijing, 1986.
34. C.S. Tang, ''D&R of programming technology for thirty years,'' Comp. Sci., no. 3, 1988, (in Chinese).
35. C.S. Tang, ''To unify programming with a temporal logic system,'' Dept. Comp. Sci. CMU, Tech. Rep. No. CMU-CS-87-160; Revised version in Proc. Jap. Ann. Softw. Symp. Tokyo, 1989.
36. C.S. Tang, ''Design Philosophy of XYZ system,'' J. of Softw. no. 1, Beijing, 1990, (in Chinese).
37. C.S. Tang, et. al., ''The syntax and explanation of the temporal logic language XYZ/E,'' Inst. Softw. Acad., Sin. Tech. Rep. IS-CAS-XYZ-90-1, Beijing.
38. C.S. Tang, M. Zheng, and X. Li, ''A two level formal semantics and semantic-directed compilation,'' Scientia Sinica, series A, vol. XXVIII, no. 9, Beijing, 1985.
39. S. Wang, ''Rapid prototyping with Prolog in XYZ system,'' Master thesis in Inst. Softw. Acad. Sin. 1989; to be submitted to J. of Softw., Beijing (in Chinese).
40. T. Winograd, ''Beyond programming languages,'' CACM, 1979.
41. H. Xie, J. Gong, and C.S. Tang, ''A structured temporal logic language XYZ/SE, J. of Comp. Sci. & Tech., vol. 6, no. 1, Beijing, 1991.
42. H. Xie and C.S. Tang, ''An Approach to Concurrent Programming in Temporal Logic,'' to be published in *J. of Comp. Sci. & Tech.*, Beijing, 1993.
43. R.T. Yeh, ''System development as a wicked problem,'' Intl. J. of Softw. Eng. & Knowled. Eng., vol. 1, no. 2, 1991.
44. R.T. Yeh, R.A. Schlemmer, and R.T. Mittermeir, ''A systematic approach to process modeling,'' J. of Systems Integration, vol. 1, nos. 3/4, 1991.
45. Y. Zheng, and C.S. Tang, ''Formalization of data flow diagram,'' The Role of AI in DB & IS, North Holland, Amsterdam, 1987.

《时序逻辑程序设计与软件工程》（序）

（2001）

XYZ 系统是一种以时序逻辑语言 XYZ/E 为基础的软件工程工具系统，该系统的研制计划自 80 年代初开始实施，在国家多种科研经费的支持下，共历 3 个五年计划的不断工作与改进，终于在 1995 年夏在中国科学院软件研究所实现。从 1996 年起，我们一方面开展在实时过程控制、动画片等领域的一些应用研究，一方面又从理论与技术结合方面对这系统进行改进并提出一种新的方法与工具，且将其应用于实际。为便于今后开展有关 XYZ 系统的推广与应用，也为了适应我国在计算机科学与软件工程等领域培养高层次的研究及开发应用人才的需要，我们编写了这本书。它可以说是我本人以及我所领导的研究小组近 20 年来的研究工作的总结。在本书中，我们力求明确地说明在软件工程及形式化方法等方面有哪些是应该解决的方向性的、根本性的理论与技术问题，以及我们在 XYZ 系统中是如何解决这些问题的，并将力求说明我们所走的道路与解决问题的方法与当前国外主流方向的分歧所在及各自的得失如何，以期得到读者的理解，从而做出合适的判断。我之所以认为这样做是必要的，不仅是为了使我们的工作能得到读者的认可与支持，更重要的是希望能提醒我国青年同行：虽然在软件领域中，不论理论或技术方面，我国总的讲还比较落后，还应该向先进工业国的同行学习，但请注意，千万不要盲目追赶新潮，要以分析批判的态度对待他们的工作，这些工作不但可能有待改进，而且在某种条件下，甚至有时在重大的方向性问题上还长期走在值得怀疑的道路上。在这种情况出现时，不但可能旁观者清，而且因为我们所处的文化背景不同，思维方式有异，可能我们比他们更易发现问题并认清道路。古人所谓智者千虑必有一失，其信然乎！日本是以学习西方科学技术而闻名的，软件技术更是力追美国。可是近年来，日本也有人对此提出疑问。例如，以向日本产业界介绍外界高新科技最新情况著称的经济新闻社与日经产业消费研究所合办的《日经高科技情报》（日经ハイテリ情报）在 1993 年第 200 期发了一期“东方的智慧”专号，由该所研究员岛井弘之等写了一篇总论，其中有一段话说明办此专号的目的：“现代科学技术是否已陷入某种滞塞状态？期待甚高的高温超导和常温核裂变研究虽仍在继续下去，而其现有技术已迈进成熟阶段，但是它缺乏发展革新的苗头。这种停滞的原因也许包括西方理性主义（rationalism）对待事物的看法和行动原理，至少可以说是根本原因之一。当今是重新考虑现代科学基础的思想方法的好时机。目前世界上存在着许多与西方不同的各种文化思想区域，自然其中会隐藏着打开局面的思想。比如，自古代文明发达以来，中国人发明了印刷术、指南针、火药等丰富而实用的技术，其智慧最能够作为参考

对象。”由此可见，日本人已开始注意到这个问题，我们岂可妄自菲薄？我认为，这不失为值得探索的达到知识创新目标的道路之一。当然这件事不能流于空谈，捕风捉影，只有在具体科学技术研究中作出由于具有我国文化特征的思想指导而确见实效的工作才有真正的意义。因此，我希望，我的上述看法不要被误解为鼓励人们丢掉踏实的研究作风而去做没有科学根据的比附与胡思乱想。

回顾近20年来XYZ系统的研制过程，深感在科研工作中走与世界主流方向不一致的道路是多么艰难。XYZ系统提出之初，虽得到美欧软件工程方面几位著名专家［如ISSI（国际软件系统公司）的叶祖尧[①]、CMU的N. Habermann及Trondheim的A. Solvberg等教授］的赞赏与支持，但由于这些工作在思想倾向上与西方理论研究的主流方向相背离，自然不易很快得到西方理论界的理解与赞许，因此他们虽然对于XYZ系统因其独特思路而乐意邀请我去介绍，但对其意义与价值却长期不作评述，实际上即表示怀疑。直至1988年才有两位英国著名理论家H. R. Barringer和D. Gabbay在一总结性文章中指出：“将时序逻辑应用于软件工程的主要步骤即找到可执行时序逻辑”，并承认1983年我发表的介绍XYZ/E的论文［T1］是这方面最早的“先驱”。但这一评价也不过是“一叶知秋”而已。真正表示国际理论界对XYZ系统态度的变化是在90年代才发生的。1994年，时序逻辑的创立者、1996年图灵奖得主、以色列著名理论家A. Pnueli教授第一次访问我国。他在仔细参观了XYZ系统的演示后对我说：“说实话，我过去一直认为你的野心太大，不可能成功，而这次看了XYZ系统的演示后，我发现你已经成功了。”在他与他的长期合作者斯坦福大学的Z. Manna教授的提议下，于1995年在北京召开了“逻辑与软件工程”国际研讨会。在他们向会议提交的论文前面有一段对XYZ/E的评语：“唐稚松教授……将时序逻辑的概念处理得超乎任何人的想象，并将之应用在许多方面，在他之前，没有人认为这是可能的”。接着，在他作为这次国际研讨会论文集的主编所写的序言中，更清楚地说明了他对以时序逻辑为基础的XYZ系统的认识的变化过程。他说：“我仍然记得，当我首次听唐教授谈到以时序逻辑作为软件开发全过程……的统一基础时所感到的惊讶。随着时间的流逝，这一幻梦般的系统，由软件研究所一个致力于此的小组所实现并加以扩充，这系统……随之逐步成形。我的这种惊讶也渐渐转变成为钦慕”。接着，他说明以时序逻辑这种协调的形式语义理论作为软件开发全过程基础的重要意义。最后，他说：“我盼望由唐教授所构想并发展的XYZ系统作为先驱所倡导的这一途径今后能引起软件工程系统的研究人员以及构造人员的巨大兴趣”。事实上，Pnueli教授所说明的他对XYZ系统看法的变化，正是近年来国际形式化理论界在形势迫使下已开始出现改变其原来思路的动向的一种反映。比如国际代数语义权威SRI（斯坦福研究所）的Mesequer教授将状态转换机制引入代数语义就是这方面的又一有意义的迹象。（他曾来函索取XYZ的资料，并说瑞士苏黎士ETH的Engeler教授向他推荐我们的工作。在1998年2月Dagstuhl召开

---

① 叶祖尧对XYZ的评语：“我心中绝对相信唐的XYZ语言将是一次重大突破的基础。……他的工作非常有创造性，而且有使软件生产率取得重大提高的实际应用前景。”Solvberg的评语：“XYZ语言为我们提供了一种关于信息系统……软件功能描述的令人着迷的思维方式。……我看到，如果我们将唐教授的XYZ系统组织到我们的方法论基础中去，将有巨大的潜力推进软件设计与构造的……学科技术水平。”

的会议上，我们初次相遇，又提起 Engeler 教授向他推荐我们的工作；此外在这次会上，我还第一次遇到另一位代数语义的权威学者 Wirsing 教授，他也提到 Ehrich 教授向他推荐我们的工作。）

我认为，XYZ 系统之所以过去长时间难以为西方理论家所理解，从根本上说是由于我们采用了一些我国传统哲学思想方法与西方流行的逻辑分析方法相结合的思路。这样的思路强调理论与技术紧密结合而不偏向一个极端，与西方片面重视形式化数学理论的理性主义思想差距甚大，但与国外（包括美、日、欧）一些对理论与技术不怀偏见，但较重视能实际提高软件生产率的软件工程专家（如叶祖尧、Solvberg 及日本软件工程学会主席岸田孝一等）的思路较为接近。这就说明，为什么在开始阶段 XYZ 系统从理论界与软件工程界这两方面所得到的反应是如此的不同！特别应该一提的是日本友人岸田孝一先生，他不但对西方软件技术十分了解，而且对中国传统文化有很深的素养，因此他对 XYZ 系统的哲学背景的兴趣也显得特别浓厚。他不但多次邀请我到日本作为软件学术会议的特邀主题报告人讲 XYZ 系统的哲学背景，而且亲自在《朝日新闻》（夕刊）上写专文介绍 XYZ 系统（1995 年 12 月 4 日）。文中说：“唐教授的成就之一就是花了近 15 年的时间成功地开发了称为 XYZ 的软件系统。尽管系统所采用的最基础的数学理论来源于西方，但构造此系统的基本思想却来自孔子的中庸哲学和佛教的禅宗认识论哲学。这也许可以说是东方文明对于新的 21 世纪计算机技术发展的一大贡献吧”。这一情况当然不易为西方多数理论家所理解。像瑞士苏黎士 ETH 的 Engeler 教授及德国 Braunschweig 的 Ehrich 教授等少数理论家算是例外。

在 XYZ 系统设计中，我之所以强调哲学思想指导，并不是出于为哲学而哲学的学院式兴趣，而是由于 80 年代初对国际软件理论与技术的发展潮流及其思想背景感到怀疑与忧虑所致。事实上，国际计算机科学家中有类似怀疑和忧虑的人并非少见，如斯坦福大学的 D. Knuth 教授就是较著名的一位①。

下面让我们回顾一下计算机科学技术的发展历史。如所周知，在 70 年代中以前，计算机科学领域的理论与技术是紧密相联的，如形式文法（包括属性文法）及语法分析方法的研究与编译技术是相互依存的。可是自从语义形式化问题提出以来，由于这问题难度很大，希望从传统数学理论中找新的工具与方法的要求很普遍，许多有才能的青年数学家逐渐进入这一研究领域。他们在软件技术方面的根基不很深厚，但受理性主义传统影响很大，数学的职业倾向性远远大于软件的职业倾向性。从 80 年代以来，在这类人员集中的机构、地区与会议上，其价值标准（如脱离实用意义，单纯从理论的逻辑严密性及深度与难度对工作进行评价等），以及学术气氛也就逐渐发生了变化，从而不能不影响学科发展方向，使计算机科学理论研究日益脱离软件的现实实用性考虑而成为一种较深的数学探索。这种情况自然难被关心市场效益的工业界所接受。特别是以实用主义传统著称的美国工业界，他们虽然也

① 1995 年他在访问 Duke 大学时，与我过去一位名字叫金伟的学生（现在该校作博士生）谈到过我，他说：“唐教授是我遇到的惟一一位关心计算机科学在 10 年、20 年后如何发展这个问题的中国计算机科学家。因此，我建议斯坦福大学计算机科学系邀请他来访问。” 事实上，XYZ 系统的设计思想正是这次访问时开始形成的。

很关心提高软件生产率,关心软件的可靠性与可维护性这类关键问题,但他们对日益远离工程技术实用性的形式化语义理论及规范语言的研究逐渐失去信心与兴趣,终于导致几乎完全抛弃了理论研究,而单纯从技术上找出路。他们企盼软件像硬件一样,用从技术上提高自动化水平或可重用性的途径找到提高生产率的方法。这种思路大大推动了软件工具及与之相联系的操作系统的发展。其后果是整个80年代,对于软件研究与发展来说,就是一个理论与技术相互分离脱节、各走极端的年代。这种情况在一段时间内使双方都得到一定程度的满足,理论家获得了很高的荣誉,而工业界也获得了不小的利润。可是,提高软件生产率,及提高其可靠性与可维护性等根本性问题不但没有解决,而且似乎希望日益渺茫。这种情况是否合理?这些根本性问题是否已失去意义?事实并非如此。不但像美欧各国关键技术委员会每年一次向国家最高当局提出的报告中都将这个问题放在首位;而且甚至像微软这样一贯强调软件技术且已取得大量利润的公司也已感到,由于缺少精确语义基础,大型软件的可靠性与可维护性问题已制约他们进一步的发展,因此不久前该公司已在剑桥大学等处投入巨资开展这方面的研究。事实上,语义的精确性与技术的自动化二者是相互依存不可分割的,而且只有紧密结合才能达到提高软件生产率的目标。因此,这个问题如果不从根本上找出理论与技术脱节的病根来辨明道路,单靠投入资金岂能解决。那么,产生这个问题的根本原因何在呢?

40多年来,有关计算机科学的研究实际上是围绕以某种模型为基础的三个彼此相关的层次进行的,即计算机体系、适应该体系的可执行程序语言及表示程序抽象语义的规范语言。如所周知,40多年来流行的计算机体系都是建立在冯·诺伊曼(von Neumann)模型上的,流行的程序语言即为适应这种机器体系的常见命令式语言(如Pascal,Ada,C,C++等)。这种模型主要的特征即为自动机状态转换机制。通俗地说,表现为变量在运行过程中可通过赋值命令(语句)改变其值,且其控制流可分解成“循环”、“分支”及“继续”等基本结构,其中尤以“循环”最具特殊的意义。由于这些特征使这种语言能高效地在冯·诺伊曼型计算机上运行。至于适应这样模型的规范语言,人们较为广泛接受的是Hoare逻辑中的由前置断言(pre assertion)与后续断言(post assertion)所表示的逻辑语义。以上三个层次对冯·诺伊曼模型的表示方式相互间是紧密关连,协调一致的。它有其优点与弱点,优点是其常见命令式语言执行效率高,为广大计算机用户所习用(特别是用循环表示重复计算),在Hoare逻辑的作用下,命令式语言的程序与由前置和后续断言表示的规范协调地联系在一起,而且对于数学水平不高的初学者来说,按照自动机状态转换方式设计一规模不太大的程序或模块较为直观而易于掌握;其弱点是这种基于自动机状态转换的机制不够抽象,包含细节太多(赋值本身即是一种细节),不能像数学中常用的函数那样便于嵌套与连接,平常数学中许多理论与技巧(如证明技巧)不易在其中直接应用等等;而更重要的是,在这种命令式语言与规范语言的关系中,后来发现一些语义方面的难题长期找不到解决的办法。比如,常见命令式语言中递归过程(及进程)这类模块的后续条件如何表示其递归性并如何验证其正确性?更困难而又更重要的一个问题是,对于由通信进程组成的并行语句,如何表示其语义?如何验证其正确性?事实上,在冯·诺伊曼模型的框架内,惟一可行的办

法是将并行语句的语义归结为该语句所包含的串型进程语义的合取，但这一命题成立的条件是语义可组合性。而事实表明，在通信的情况下这一条件是可能不成立的。因此，困难在于如何保证并行语句语义可组合性条件成立。容易看出，上述优点是工程界所重视的实用性方面的优点，上述弱点是理论界所强调的数学精美性方面的弱点，而上述困难问题则的确是冯·诺伊曼模型所面临的实质性问题。不解决这些问题必将严重影响程序的可靠性与安全性，将使冯·诺伊曼模型不论从理论上还是从工程上说都将难以采用。（C. A. R. Hoare 在文献［Ho2］中讨论 CSP 的数学模型时明确指出，他在文献［Ho2］中放弃了文献［Ho1］中的语义模型，而采用了与 CCS 相同的进程代数所要求的函数式模型，其中三条原因中的前两条就与这里所述的两项困难问题直接相关）。由于上述情况，从 70 年代末起，计算机科学领域出现寻找其他非冯·诺伊曼模型的高潮。其中最有影响的即函数式模型的研究，包括函数式机器体系、函数式程序语言，以及以递归性与函数映射为特征的基于可计算函数（包括 λ 演算）及代数的语义理论与规范语言等三个层次的研究。这些研究与上述基于冯·诺伊曼模型的三个层次研究可以说是相互对立的，一方的优点正是另一方的弱点，两方实际上是难以协调的。由于函数式模型具有与传统数学理论一致的许多特征，从 80 年代以来有较强数学兴趣的计算机科学家参与这方面语义理论及规范语言研究的人越来越多，在理论研究方面形成热潮，特别是西欧受理性主义传统影响较深的学术机构更是如此。因此，可以说，从 80 年代到现在计算机科学理论界（特别是以英、法、德、荷、意等国为代表的西欧）是以函数式模型为基础的理论研究占绝对统治地位的年代。但另一方面，这种理论研究所依附的基于函数式模型的计算机体系则很不成功，这类计算机造价昂贵、效率低，而且功能很有局限性。这是与现在流行而且在可预见时间内难以取代的硬件基础紧密相关的，由于 RISC 技术及工作站流行，这类函数式机器已退出市场，这方面所长期探索的研究似乎已销声匿迹。处于这类理论及机器体系之间的函数式语言研究虽多数已消失，但仍有少数（如从爱丁堡发源的 SML）因在某些领域（如作为表示形式理论的验证系统的工具）有其应用价值而在学术界保持其生命力。这一情况与下述情况颇为相似：长期以来尽管冯·诺伊曼计算机及与之相应的命令式语言占绝对统治地位，但函数式语言 Lisp 在人工智能领域仍保持其生命力。类似地，估计今后像 SML 等少数函数式语言在某些专用领域可能会长存不衰。可是，只要冯·诺伊曼计算机维持其在机器结构方面的统治地位，则函数式语言因与机器体系不一致而影响其效率与功能，故不可能在可执行语言领域占统治地位，广大用户和工业界仍将会以使用基于冯·诺伊曼模型的命令式语言为主，这一点是无可怀疑的①。现在的问题是关于形式语义理论及规范语言研究怎么样？我认为，只要机器体系及可执行语言是以冯·诺伊曼模型为主流，若理论研究却反其道而行，坚持以函数式模型为基础进行，则必然出现的结果是要么理论与技术脱节成为一种近期与技术无关的纯基础数学研究，要么

① 一些著名计算机科学家致力于提高爱丁堡 SML 语言在冯诺曼计算机上的执行效率，希望将它改造成一通用程序语言以取代常见命令式语言。对于这一目标是否能完全达到，我的观点是消极的。因为这目标等于要求建立在一种模型上的语言有可能在另一种与之对立的模型的计算机上有效执行，且其效率与建立在后一模型上的可执行语言的执行效率不相上下。这实在是件不可思议的事。

这样的理论终被抛弃。从长远来看，两者必居其一，也就是说，“皮之不存，毛将焉附?”从80年代初以来，XYZ系统的研究即是基于这样的信念开始的。我们走的是一条以冯·诺伊曼计算机为基础，面向常见程序设计的理论与技术紧密相联系的研究道路。作为少数派，孤单地走这条道路是艰辛的，为此所必须解决的问题也是十分困难的。但我认为只要思想方法正确，坚持不懈，终能找到解决困难问题的途径，因此对这条坎坷道路的前途一直满怀信心。事实上，几乎与我们同时，得克萨斯州（Texas）大学奥斯汀分校的Chandy与Misra教授很可能是在Dijkstra教授的影响下，也走上了与我们类似的面向冯·诺伊曼模型建立形式语义理论与可执行语言相结合的道路。这就是他们关于UNITY语言的研究。这是一可执行命令式语言，为了使此语言在并行情况下具有语义可组合性，他们建立了一整套复杂的理论，并对这种语言进行很多的限制，最后终于解决了以冯·诺伊曼模型为基础保证在并发情况下该语言的语义可组合性的难题。从这方面说，该系统是成功的。Dijkstra与Hoare对此语言系统作了极高的评价并予以推荐。该系统问世之初曾经引起了理论界与技术界的注意，但很可能是由于这一语言受限制太多，与常见命令式语言风格差距很大，且表示力所受影响很深，以及它所依据的理论是专为此语言所建立的，相当复杂，一般用户很难深入掌握；此外，它的目标似乎是取代流行的常见命令式语言，而不是为常见命令式语言服务，为它们提供一坚实的语义基础，因此，难以引起广大习惯于使用常见命令式语言用户的兴趣。总之，由于各种原因，这一语言系统问世10年，至今已少有人谈及，也未见到重要的应用效果。无论如何，它即使不被完全抛弃，也既不大可能取代流行的命令式语言，也未能为这些命令式语言提供一合适的理论基础，补充其这些方面的不足，以达到使冯·诺伊曼模型基础上的机器体系、可执行语言及语义理论与规范语言三个层次紧密地联系起来的目标。看来，这一使命不可避免地只有落在XYZ系统身上。为此，XYZ系统中提供一时序逻辑语言XYZ/E，它既包含表示规范的子语言XYZ/AE，也包含可在冯·诺伊曼计算机上有效执行的子语言XYZ/EE，两方均具有与常见命令式语言相同的控制结构与数据结构，其风格与表示力与常见命令式语言无异，后者可作为常见命令式语言之上的中间语言，而前者可在语义描述及规范语言方面做常见语言的补充；它的理论基础，即Manna-Pnueli线性时间时序逻辑，并不要求用户专门进行复杂的理论训练即可掌握，所以就XYZ系统而言，不存在妨碍UNITY推广应用所发生的那些理论与技术上的困难。更重要的是，前面所述的那些冯·诺伊曼模型面临的难题，如并行语句的语义可组合性问题及递归过程规范的表示与验证等问题，都已简单而自然地在XYZ系统中得到解决。因此，现在我们可较大胆地说，XYZ系统已较圆满地为冯·诺伊曼模型提供了一适应冯·诺伊曼计算机体系且理论与技术紧密联系的逻辑语言及软件工程系统。据我所知，当前世界上似乎尚未见到其他同类系统做到了这一点。事实上，正如Pnueli指出的，这一方向正是XYZ系统作为“先驱”所倡导的。

由于基于技术实现软件生产自动化的工具研究（特别是依靠人工智能实现这一目标的努力），效果并不显著。自80年代中以来，以美国工业界为基地大力发展起以提高可重用性为目标的提高软件生产率的模块程序设计的研究，这也就是面向对

象程序设计的潮流。先是以 C + + 为代表的，近年则是以 Java 为代表的程序语言系统已在美国工业界引起广泛的注意。可以说，这一以软件技术为基础的提高软件生产率的道路是颇有成效的。但在我们看来，它仍存在以下两方面的问题：第一，这一方向对语义精确性问题未予足够的重视，因此可靠性方面仍无足够的保障。比如由于面向对象程序设计的特征之一是可重用性，忽略了形式规范的情况下要保证重用时的语义一致性则只有依靠测试（testing）。可是对于具有继承性的模块而言，测试变得更为复杂与困难。第二，分布式面向对象模块中并发通信将起核心的作用。因此，由并发通信所引起的语义正确性问题是无法回避的。这个问题不可能单纯从技术上依靠模块重用来解决。

下面我们对 XYZ 系统的具体特征概括地介绍如下：

1）我们的目标是语义精确性与技术实用性的统一，精确性与实用性缺一不可。在这方面我们的要求是严格的，不容让步的，因为只有这样才能确保提高软件可靠性，可维护性的目标。可是，语义精确并不等于理论精美，虽然在理论精美性与技术实用性这两方面，我们都很重视，但如果双方发生不易调和的矛盾，则在不损伤语义精确性的条件下，我们宁可在理论精美性方面做适当的让步，以保证技术的实用性标准。因为在我们看来，计算机科学毕竟是一门技术科学，它与纯粹数学不同，忽视技术实用性必将失去技术科学的应用价值，最终导致理论与技术脱节。既然冯·诺伊曼型计算机是当前及今后可预见的时间内惟一被采用的计算机体系，计算机科学理论与可执行语言理所当然即应适应这种机器体系及其依据的模型。这才是保证理论与技术协调的合理道路。但是如果为了做到这一点而出现关于语义理论及相关技术方面的困难，则是必须解决的问题，不能回避，否则其可靠性无法保证。XYZ 系统的研制过程即是以此为目标，面对这些问题逐步加以解决的历程。事实证明，只要采取正确的思路，往往可以找到出人意料的解决问题的方法。在这一方面，我们是有较多的体会的。我坚信，将我国传统哲学思想与西方逻辑分析方法相结合，常能为我们提供一些巧妙的思路。我希望在本书中我们能够结合理论或技术问题对此有所说明。

2）时序逻辑语言 XYZ/E 的一个重要特征即强调动态语义与静态语义并重，将两者结合而不只偏向一方。由于适应冯·诺伊曼机器体系的可执行语言的特征即表示自动机的状态转换机制，其方式是命令式的，其语义是动态的；而适于这种模型的规范语言，即 Hoare 逻辑中的 Pre 与 Post 断言（用时序逻辑表示即 Pre→◇Post），其语义则是静态的，我认为，将规范所表示的静态语义与可执行语言所表示的动态语义紧密联系起来是使这方面的研究更能在实际软件工程系统中发挥实用价值的关键（这也就是前面所引 Barringer 与 Gabbay 对可执行时序逻辑在软件工程中的作用的评价）。为此，在时序逻辑语言 XYZ/E 中，应找到一种控制框架将规范的静态语义与可执行的动态语义统一地表示出来。这就构成 XYZ/E 的一个重要特色，即表示动态语义与静态语义（即 XYZ/EE 与 XYZ/AE 中）的两种命令（语句）可混合在一程序中出现，用这种混合出现的程序，即能表示出由完全抽象的规范到完全可有效执行的程序之间平滑过渡的过程。在 XYZ 系统中，就是以这样的方式表示逐步求精方法。这种逐步平滑过渡恰好起着理论（表示静态语义的断言）与技术（表示动态

语义的可有效执行的程序）相联系的桥梁作用。

动态语义与静态语义的联系，不但反映在抽象规范与可执行程序的关系上，而且也反映在程序的模块结构上。如具有流程图结构的过程与进程显然是表示动态语义的，而作为面向对象程序设计的具有封装性（encapsulation）与继承性（inheritance）的对象模块，显然是面向问题领域的，故其语义是静态的。为了将这两方面联系起来，如上所述，在 XYZ 系统中提供一种面向基于通信的计算过程的模块，称为并行语句进程（PSP），用它即可将大型程序中静态语义与动态语义联结成为一个整体。

最后，还有一个与语义有关的问题，即如前面所述，XYZ/E 可分成不同层次的子语言。其最基础的一层是表示可有效执行程序的 XYZ/EE，而且这些程序所表示的风格与常见程序语言是相同的。人们不禁要问，既然已经存在许多流行的程序语言，如 C，C + +，Concurrent C，Java 等，提出 XYZ/EE 这样的时序逻辑语言还有何必要？一方面，由于它作为具有统一结构的时序逻辑语言的最底层，可与表示不同抽象层次规范相联系，来表示逐步求精过程，从而提高所书写程序的可维护性与可靠性；另一方面，XYZ/EE 语言可以看成是一种中间语言，它的程序可在保持其执行效率的前提下自动转换成 Unix 上的 C 程序，Java 程序，或适应微软操作系统的程序。这样，即可使用 XYZ/EE 书写的程序具有适应不同软件平台的可移植性。这种特征是很具有实用价值的。

3）关于 XYZ 系统中工具所实现的方法论，有如下几方面的内容：

①归纳“2）”中所述，XYZ 系统所支撑的程序设计可以说是由纵向与横向二维正交而成。纵向方法论即前面谈到的由抽象（静态语义）到具体（动态语义）的逐步求精方法，其中包括不同抽象层次的规范描述、语义一致性检验、验证及速成原型等各种方法及相应的支撑工具，这种设计与语义由静态向动态过渡相关；至于横向方法与工具，则是为了支撑模块程序设计。对此，XYZ 系统中将针对各种不同类型模块提供相应的可视化图形设计工具。但是这两种程序设计方法及相关的工具，只是从形式化语义理论及模块程序技术两方面各强调其中的一方面，并未能将二者有机地结合起来。为了达到本书所强调的将这两个方面在时序逻辑语言 XYZ/E 的基础上协调地结合起来的目标，XYZ 系统又提出了第三种软件工程方法与工具。近年来，CMU 的学者们（D. Garlan，M. Shaw 等）特别强调软件体系结构的选择在软件开发过程中的重要意义，我们针对 XYZ 系统的特征提出一种可视化软件体系结构描述语言 XYZ/ADL（Architecture Description Language），并以它作为应用 XYZ 系统设计软件时的用户语言。它以组件（component）作为一软件系统的基本构件（事实上它是模块概念的推广）。每一组件包含二个方面：一是其外部界面（interface），即以其规范来表示；另一是其内部结构，即其体系（architecture），它以 XYZ/ADL 的图形来表示。由前者向后者转换，即构成软件开发过程的一步过渡（transition）。因一组件的体系结构中又包含下层的组件，它们亦由上述两方面构成并进行过渡。这样的逐层过渡即构成开发该软件系统的逐步求精过程。对于 XYZ/ADL 而言，每当描述一组件的体系结构后，即将自动生成一描述该体系结构语义的 XYZ/E 程序。将此 XYZ/E 程序与该组件的规范相互结合即可应用 XYZ 系统中提供的验证或模型检

验（亦称速成原型）工具检验这一步过渡的语义一致性，这样即将前二种方法的主要特征结合起来了。在XYZ系统中将提供关于软件开发的上述三类软件工程方法与工具。在开发一软件时，每一种方法与工具均可独立使用，也可将该软件划分成语义独立的模块，分别各用上述一种方法与工具进行开发，然后再（最好是以通信的方式）将它们联成一个整体。在本书下册第八、九、十这三章将分别对这三种软件工程方法与工具及相应的逐步求精过程作较详细的介绍，并将以具体例子作为示范说明。

②在大型程序设计中往往出现这种情况，即有时需要在某些部分嵌入用其他程序语言写的久经使用的程序，在软件再造工程及某些专用领域应用系统中有时也有与此类似的需要。为了提供用户处理这类问题的手段，XYZ系统中提供一种将用其他语言书写的程序转换成XYZ程序的简便的形式化方法。我们已用此工具将SDL，ESTELLE，VHDL等这些专用领域国际标准语言转换到XYZ/E，看来效果相当不错。关于这部分的讨论即构成本书下册第十一章的内容。

XYZ系统的特征也就决定了本书的特征。本书既不应是一本主要讨论时序逻辑理论的专著，也不应是一本主要介绍各种软件工程工具技术的教程。它旨在较全面而准确地介绍XYZ系统。在介绍时序逻辑语言XYZ/E时，一方面首先是面向程序工作者，将XYZ/E作为一程序语言，而不是作为一逻辑系统来介绍；但另一方面又应随处指明它仍保持其为时序逻辑理论系统的特征；在介绍各工具时，一方面首先将其作为软件工程工具来介绍；但另一方面又随时说明它与时序逻辑语言XYZ/E的内在联系。我深知，这样一本介绍兼具理论与技术双重特征的系统的书，本身存在着内在的矛盾，它所具有的“中道”性质的陈述方式很可能同时引起来自理论界与工程界双方的不满。我记得在钱学森先生所著《工程控制论》英文版的序言中也曾谈到：该书的陈述方式很可能“按数学著作的标准看不够严谨慎重”，而“从介绍工程系统的著作的标准看又包含数学理论太多”。我在书写本书时，实在也感到类似的困境，深觉无法避免。我们认为对于这种深层的矛盾，采用“中庸之道”的方法来对待是较为合适的，这也许正是“技术科学”的内在特征所在。望能得到读者的谅解。

由于XYZ系统规模十分庞大，研制时间长达近15年，且参加人员流动性很大，这就决定了它的研制人员的结构具有如下的特征：一方面这一系统只可能是集体完成的工作，另一方面为了使这一系统不致成为一松散无章的堆砌物，它又应是一个在一人的思想指导之下按照非常紧凑统一的设计思想与理论完成的系统。事实上，XYZ系统正是如此。近20年来，参加人员50余人，平均每人参加时间约为2～3年；他们绝大多数参加了各子系统或工具有关设计的讨论与实现，惟我一人一直从头到尾主持并实际参与此项工作。其语言设计、工具所支撑的方法及解决关键技术难题的方案研究等，除了少数部分外，主要皆由我负责。当然，这过程中不少人也参加了讨论并提出过很好的建议，为了节省篇幅，很难在此列举各参加人员的参与情况。不过在本书后面参考文献中，可查出对XYZ系统作出过贡献的人员的姓名（也有个别人员例外）。本书撰写情况也与上述情况相似，自然也是由我负主要责任，从总体思想到XYZ/E语言及各种软件工程方法的设计与介绍都由我完成，而对

各工具系统具体实现的技术内容的介绍，则一般由负责实现或修改该工具的人员中目前仍在我组工作或学习的人执笔，并最后在文字上由我加以统一。在书写有关部分、调试例题、对全书进行录入、编辑排版以及其他各项工作中，赵琛、沈武威起了较大的作用，谢育涛、高永祥、闫安、沈伟、唐小平、张小格、骆华俊、郭亮等都负责完成了与他们有关的部分。中国科技大学的孙淑玲教授与她所领导的科研集体，李广元副教授分别编写了本书下册第十一章（即语言转换）、第七章与第十二章有关的内容和附录Ⅱ，北京航空航天大学的张玉平副教授曾参加过附录Ⅱ初稿的撰写工作。此外，石民勇博士、王春江博士、张广泉博士先后阅读了各章，并检查了各方面可能出现的疏漏，在这方面，王、张二位其贡献尤大。我所研究员柳军飞曾在完成本系统“八五”计划的项目研究中起了很重要的领导组织作用。本系统及本书得以完成，的确是这个集体所有成员共同努力的结果。值得令人引以为慰的是，近 20 年来，XYZ 小组一直是一个团结的、有向心力的集体，其成员不但在组内工作与学习时是如此，甚至他们离开多年（现在大多数都在国外），仍对这个集体的工作予以关心，经常为 XYZ 系统的完成提供各种帮助。在本书完成之际，我应指出，如果 XYZ 系统将来被证明是一项有价值的工作，这成绩首先应归功于这个集体。还有一点应当指出，本书下册中许多关于 XYZ 系统应用的介绍都是基于我组人员与其他单位人员合作应用 XYZ 系统所得到的成果。在此我们对这些兄弟单位及有关人员表示谢意。

最后，我应借此机会感谢为 XYZ 系统的研制以及本书的出版提供各种支持、关心以及帮助的单位与友人。近 20 年来，我国国家自然科学基金会、国家科委 863 计划、电子工业部一直提供经费支持。中国科学院及我先后所在单位，如计算技术研究所与软件研究所的负责人，一直对我们的工作给予信任、支持与关心；如果没有中国科学院，特别是软件研究所基础部这样的科研环境，我相信，XYZ 系统是不可能完成的。此外，多年来国内外许多朋友也一贯为我提供了各种精神与物质的帮助。在此，我还应特别提到几位国外的朋友对我的工作的支持和帮助，他们是 J. McCarthy，D. Knuth，Z. Manna，A. Pnueli，R. T. Yeh（叶祖尧），N. Habermann，D. Bjorner，E. Engeler，H. D. Ehrich，A. Solvberg，K. Kishida（岸田孝一）等。过去我多次应邀去他们所在的大学或研究机构的访问对 XYZ 系统的形成很有影响。在本书的出版过程中，中国科学院科学出版社的有关同志也一直予以支持。最后，还应该特别提到的是 30 余年来与我冷暖相依的伴侣童恩健及我的两个儿子其琛与其放对我的工作与生活的支持与关心。对所有这些朋友与亲人，我谨在此表示由衷的谢意。

其它文章选

致中学生的信

郁毅、晴芳同学：

近半年我去国外，刚回北京，读到你们的来信。目前因为事情很多，只能简单地写数语奉复，乞谅。

我觉得“学部委员”（现改称院士，编者注），只是一种称号或头衔，不必太看重，重要的是工作。我的工作是属于应用基础研究，即设法将基础理论成果转变成为实际在生产或国防中起作用的工程技术系统。这过程很长，很复杂，很困难。我的研究虽被认为颇有成功的希望，但仍未真正成功，所以目前还不是值得庆贺的时刻。关键要锲而不舍地坚持下去，还要作艰苦的努力。我觉得，我国有许多老一辈的科学家更值得青年人去学习。他们的学术成就很大，为国家的建设实际贡献也很大。他们不计个人名利，一心为中华民族的振兴而贡献其毕生精力的精神尤为感人。你们应该向他们学习。比如李四光、钱学森……。

我觉得我国传统文化博大精深。虽然由于时代的局限，其中有不少糟粕，但更值得我们重视的是它的精华。这一点不仅我们自己应该重视，目前全世界学术界都在重视。我认为我的全部工作与我国传统文化血肉相连。这一点，一封信讲不清楚。至于说祖国的语言文字则更是表达思维的工具。语文没有学好，将对科学研究起不良影响。人是用语言思维的，语言没有学好，将阻碍你思维的条理性和深刻性。盼能注意。以上这些意见不一定正确，谨供参考。不一。

专问

学安！

唐稚松

1992 年 6 月 8 日

我国软件产业发展中几个问题

Problems in the Development of China's SoftWare Industry

（科技导报，1995）

长期以来，国内外专家均认为我国有发展软件产业的潜力，但现实情况却不尽如人意。经过十几年的努力，我国软件开发与应用虽有较大发展，但与国外的差距仍然很大，甚至落后于印度。这种情况不能不使关心我国软件产业前景的人深思。过去我国进行规划时往往盲目追随国际潮流而忽视国情，结果是钱花不少，项目很多，但收效甚微。

我国软件工作中经常混淆研究、开发、产品生产、商品销售四个不同环节的界限与联系，忽视了它们各自的目标与作用、工作的具体内容、审评标准、工作环境及组织形式、工作人员的素质要求各不相同，常常不顾各个环节的特征而胡乱安排计划，结果造成严重浪费。我国软件工作这方面的失误在欧美日甚至在港台都是少见的，因此值得认真总结教训。

一、研　　究

研究可分为理论研究与技术研究，其特征是在具有一定学术深度的基础上的创新，即创造新思想、新方法、新技术与新的设计（而非枝节性的改进）。

从产业（不是从文化）的角度说，研究工作之所以必需，是要为产业的下一步发展提供后劲，为公司准备新的市场竞争力。欧洲软件界往往靠其坚强的理论研究基础提出新的软件成果，比如 Algol 60、Polog、ML、COS，甚至包括 Ada 等都是著名的例子。而美国则以其雄厚的软件工业及应用市场为背景，在技术方面标新立异，推动软件产业的大发展，走的是另一条路，因而一直保持世界领先地位，世界上大量软件方面的创新均产生于美国就是证明。

从生产的观点看，并非任何有理论意义的研究都可能转换成生产力。因此，美国国防部与工业界在制订长远规划时，对一项理论研究的评估，总要考虑其可能在生产中起作用的时间上限，花费时间过长又难以在生产中起作用的理论研究都会排除在这类计划之外。同样，也不能说每项技术创新均具有市场价值，都能转换成实用的商品。因此，对每项研究均应作时间与条件等方面的预见性分析，否则即可能造成浪费。我国研究工作的盲目性较大，故可及时转换成生产力的项目较少。当然，决定研究成果能否转换成生产力的因素不只是这一个方面，但却是很重要的一个

方面。

与此有关的另一个问题是科研课题由上而下或由下而上提出的不同途径与方式。一般说，由下而上提出的课题较易转换成生产力，但这类课题往往层次较低、周期过短、影响面较窄小；而由上而下提出的课题则往往方向较明，如能转换成生产力则影响较大，水平较高，但是这种自上而下的课题如果组织管理不当，则较难转换成生产力。故两种方式各具优劣，最妥善的办法是根据实际情况将两方面很好地结合起来，但要将这种结合做好则又要求有很好的组织协调工作与审评水平。一般讲，美欧日软件产业多以市场为基础，故由下而上的研究工作较多。但自 80 年代以来，为了加速高技术的发展以提高竞争力，这些国家又提出了许多发展高科技的战略规划，都是由上而下制订的。这些规划多经过各方面反复的论证，进行了各种时间与条件的评估，也就是为了由上而下的计划与由下而上的市场需求能妥善地紧密结合起来，以期巨额投资能取得实效。十年来实践证明，做到这一点并非易事。美国 MCC 曾轰轰烈烈起始，但最终冷冷清清地收场。在日本则结合得较好，他们的经验值得进一步深入探索。在建立社会主义市场经济的我国，这个问题不能回避。美国国防部建立的软件工程研究所及美国工业界及珀杜（Purdue）等大学建立的软件工程研究中心似乎效果尚好，其经验也值得借鉴。

二、开　　发

开发环节追求的目标不是重大的创新，而是将已经创造出的研究成果或模型置于适宜于开发工作的环境中实现，成为高质量的成品。这一环节的主要任务是提高质量，故对工作人员的素质及工作条件与研究工作的要求完全不同，工作优劣的审评标准也不一样。

不难理解，如果“开发”这一环节没有安排好，则任何卓越的研究成果都不可能转化成有实效的成品。由此可见这一环节的重要性与必要性。在我国的软件工作中，这一环节的问题最多，这是影响我国软件质量与水平的关键所在。从组织形式上看，过去我国一方面有大量的研究机构（如各级研究所及大学），另一方面又有各部领导下的许多以生产为主要任务的工厂或公司。前一类机构强调其研究成果为达到各种等级先进水平的系统或论文，而后一类机构则强调多少产量的产品。但是，我国缺少处于以上两类之间的一类中间机构或环节，专门着重提高产品质量并使上述两方面衔接起来。正是由于这一环节的工作在我国很不明确，很不落实，严重影响了产品质量。这就是我国软件产业组织结构方面的问题所在。事实上，开发工作是最费力而又最不易看见成绩的。因无创新，从外表看，好象耗费了几倍几十倍人力物力开发出来的成品与原来研究结果并无两样，而从这一环节出来的成品又不能像生产环节那样可以计算出以数量表示的产量。这样的工作单位与工作人员如何计算成绩就不是一个简单的问题。

在国外，由于这个环节的工作要求极高而又异常辛苦，其重要意义早已为生产管理人员所熟知，所以其工作人员的工资往往很高，无名而有利。而在我国则名利

两亏，谁也不愿干这类工作。事实上，不论理论上或技术上，真正作出具有实用意义的研究创新并不容易。从软件看，我国的学术界目前还不具备提供大量进行真正创造性研究的条件。我国许多在研究的名义下进行的工作并不具有真正创新的意义，其中多数是根据国外已有的方法上或设计上，甚至是国外已经著名的商品，在我国予以重新实现或仅有一些枝节性的改动。所以，这类工作并非研究而是开发。当然，开发工作也很重要，很有价值，但必须具有开发所要求的高质量。

我国这类工作大多是在大学或研究所这类只适于作研究工作而不适于作开发工作的环境中完成的。这种环境，从工作方式、评奖标准到人员素质，事实上都不具备以提高质量为中心进行开发工作的条件。从这种研究环境作出的成品一般讲工程质量不可能很高。因此，我国由于研究与开发两环节的混乱，过去研制的软件中很大一部分从研究看并无实质性的创新，而从开发看质量又太低，是一种两不像的成品。由于没有真正合适的开发工作环境，结果使开发这一环节在整个研制过程中始终处于研究与生产两不管的状态，因此我国研制的软件系统多数质量达不到国际标准。

在国际市场上，我国除与汉字信息处理有关的软件外，一般很难有竞争力。特别是外国公司打入我国市场后，这种软件在国内市场也将失去竞争力。有人想利用由上而下的方式强制推广，这种做法既违反公平竞争原则也不可能持久，只能保护落后。实际上，软件达到先进水平的国家或地区，并不都是在研究创新方面领先的，例如日本，即主要依靠其严谨的开发工作的高质量而领先。

因此，笔者建议，为了适应开发工作的需要，我国软件产业界应着重抓好以提高质量为中心的开发基地，其价值标准、资金性质、组织管理方式、人员素质要求与工作条件等必须与大学或研究所环境截然不同。当然，也应该注意这种开发基地与大学及研究单位如何分工、联系及衔接的问题，但二者绝对不应混淆而形成混乱。当然，大学或研究所也可以附置这类基地，但它们必须按开发而不是按研究的要求来组织。

三、产品生产

产品生产与成品开发又不相同。后者是在一种较理想的环境与条件（也就是通常所谓“实验室条件”）中对个别成品进行的，而产品则应在工厂环境中进行批量生产。两者在人员素质、资源及设备条件等方面都很不相同。如果不注意这种差异，开发出高质量的成品并不一定能转换成高质量的工业产品。这又是一个十分复杂的问题，必须在开发过程中即予以注意。而且负责生产的厂家必须有一支技术队伍，专门研究如何使开发出的成品能转换成工厂环境下的产品，否则，可能前功尽弃。

遗憾的是，这一在工业界人所共知的问题，在我国软件界好象并不太为人所注意。而对于软件这种产品来说还有一个特殊问题，即它是一种智力产品，它的质量对工作人员的个人智力依赖性很大。如何将软件生产变成工业生产是国外已讨论了近 20 年的软件工程问题，目前并无定论。所以，还得根据我国的条件自己摸索出一

条合适的路子。有一种新的情况值得注意，即近年来软件工具越来越多，在不同程度上日益发挥作用。这种工具看来不仅可提高软件生产的自动化水平，而且更重要的是可逐渐使软件生产从对工作人员个人智力的过分依赖中解放出来，使之更接近工业环境批量生产的特征。从这种意义说，它对提高软件生产率起重要的促进作用。

1991 年美国国家关键技术委员会向总统提供的《国家关键技术报告》中，列举了材料、制造、信息与通信、生物技术与生命科学、航空与地面运输、能源与环境共六项，其中信息与通信中又分软件、微电子等七项。在软件这一项中即专门谈软件 CASE 工具，由此可见其重要性。

据笔者所知，近二三年来，美国各大学在计算机科学系中纷纷增设软件工程专业。这方面的意义看来国内软件界重视还远远不够。但有一点请注意，即如果我国研制的这种工具系统是围绕国际上已很流行的技术、方法与方向进行的，则很可能在我国的系统尚未制成时，国外质量较高、功能较强，且适应某专门领域的标准软件特征而又造价较低的系统已进入我国市场。此时，我国的重要用户会宁愿去买国外系统而不买我国自己长期研制的高价系统。最近，在电话软件工具方面我们即遇到这种情况。在这方面要与国际竞争，最主要的途径只有在外国人尚未注意或尚未找到解决问题的方法，而又有意义的领域或专门适应我国环境特征方面开创我国自己的新路。这就要求很高的创造性，习惯于跟在别人后面走的办法是达不到的。

这就又回到研究工作的问题。如果做不到这一点，则只有以高质量的开发工作用价廉物美的产品来取胜，走日本人的道路，此外很难有其它办法。我国软件的研究与开发工作者必须认识到，在开放改革的今天，过去关闭情况下长期通行的办法已经行不通。这里没有捷径可走，这是时代赋予我国软件工作者的重任。

四、商品销售

商品与产品不同，它有一个市场销售问题。质量不高的产品即使可哄骗顾客于一时，但最终将被市场淘汰；质量高的产品也并不一定是很畅销的商品，这是人们熟知的事实。市场需求有由下而上的，也有由上而下的。我国目前由下而上的软件市场层次似乎还比较低，大量需求的软件商品还是较低档的；但由上而下，由政府或国家大机构提出的大型应用软件系统的任务则可能是很高级的和耗资巨大的。这类由上而下的软件的需求很可能成为推动我国软件发展的巨大力量。不过这类系统目前还未完全进入市场，公开招标的似乎还不多，大多由该提出部门自己的软件人员所包揽；即令进入市场，目前我国真正有经验能指导组织这种系统的高层技术人员（相当于总工程师）也不多，很难保证高质量地完成这种任务。最近出现的新动向是国外著名软件公司纷纷来我国设立分支机构，以较高的工资挖走我国软件技术人员为他们占领我国这一市场服务。这一挑战值得严重注意。另一值得注意的问题是关于我国软件技术人员的技术水平问题，其中有许多误解。

第一种误解是，国内外均公认，我们中国人具有开发软件的潜在素质，这是否即意味着我国软件技术水平已经很高呢？显然不是。潜在素质只表示一种可能性，

绝不等于现实的技术水平。

第二种误解是，我国近年来有少数理论研究人员或中青年留学生在国外做出了很出色的成绩，这是否意味着我国软件水平已很高呢？显然也不是。一方面，少数人在国外做出成绩绝不代表我国整体技术水平；另一方面，理论研究成果与技术水平也不是一回事。好的理论成果在推动技术水平提高方面最多也只能起辅助和局部性的作用。

第三种误解是，我国各大学软件专业每年毕业人数不少，且大学软件方面课程设置和参考书籍与国外大致相同，这是否意味着我国培养的软件技术人员水平高而且数量大呢？这个问题正是我希望在此讨论的问题。根据笔者个人的一隅之见，问题正好出在我国软件教育问题上。

每年笔者所在的研究集体所招收的研究生都是各著名学府软件方面的高材生，但他们出国后往往反映动手能力不但不如欧美日的学生，而且与港台学生相比也有差距。我国学生关于软件各门课程内容记诵都很熟，考试成绩往往也比外国学生好，但动手编程、进入新的软件系统以及分析和解决实际软件问题的能力一般说来则比外国及港台学生要差。在笔者所在的集体中，先后有三位青年学者被认为对系统最熟，动手能力特强，他们一个共同的特点是大学软件本科是在香港或国外完成学习的。他们都有一个习惯就是在业余时间喜欢“玩软件系统”，即用种种办法通过动手摸清计算机内系统的“内情”，这些内情往往是手册上并未写清的。从这些青年人身上我发现一个较普遍的问题，即我国大学本科软件学生的培养存在着理论脱离实践的缺陷。

我国各大学本科着重要求学生记诵课程中的原理。这些原理当然是重要的，它们是实践经验的总结，但软件毕竟主要是一门技术，有些方面甚至具有工艺的特征（Knuth 称其讲程序技术的书为“art”）。书上的原理无论如何不能代替实践。讲操作系统的课程和书籍与实际的操作系统之间毕竟有颇大的距离。我国软件专业的老师与学生大都实践不够。而软件课程本来应是一种培养技术的课程，却被作为一种理论课程在教，这样培养出的学生动手能力往往较差。改革开放之初，因我国各方面条件不够而有此现象不足为奇，奇怪的是已经经过十多年与国外接触，且机器进口已经不少的情况下仍存在这个问题，则值得注意。据悉，无论欧、美、日还是港、台、印度的软件人员都是以培养动手能力为主。而我国各校软件负责人及老师对此问题的重要性可能认识不足。当然，由于近年来各种软件公司的出现，这方面的情况逐步有所改善，但这个问题应从大学软件本科抓起，从一开始即养成学生善于动脑、勤于动手的习惯。这才是从根本上改变我国软件人员素质的途径。我国软件产业界，应向有关大学施加影响，直接提出这方面的要求，加速解决这个问题。单纯指望教师的自觉，恐怕很难在短期内改变现状，因为这种现象的产生有很深的根源。

前面已经谈到，我国目前很缺少有经验有水平的能指导与组织大型软件应用系统工程技术方面的领导人才，这是我国软件技术水平落后的又一标志。这种人才不是靠读书本所能培养的，在有一定书本知识的基础上主要应靠积累实践经验。关于这个问题又有一种误解需要澄清。有些人希望近十年来派出的大量留学生得了高层次学位后回国来起这种作用。事实上国外大学的博士学位主要是培养研究能力，特

别是理论研究能力。他们一般说来对大型工程的实践经验也很少。据了解，我国软件方面的出国研究生，在国外得了学位后从事实际大型工程项目技术指导的人很少。除个别外，很难期望这批人来解决国内迫切需要的、指导和组织大型软件应用系统工程技术的领导人才问题。这个问题的解决恐怕只有依靠国内软件技术界的力量，但也不妨聘请一些国外（甚至一些已退休的）有经验的老软件工程师作为教练或顾问，以加速我国这方面人才的成长。

国家领导机关在抓软件政策方面也有失误。“七五”、“八五”关于基础研究的经费分配明显地对软件技术的扶持有所忽视。比如，“863”攀登计划所支持的是人工智能与新型计算机，软件基础研究只能靠很少一点基金资助。可是近年国际上新型软件平台和工具日新月异，作软件基础研究的人均无力购置。

XYZ System and It's Philosophical Background

(1996)

1 Philosophical Background

1.1 Three stages of knowledge

"Thirty years ago, before I began to do Zen Meditation training, I saw the mountains as mountains and the rivers as rivers. After a period of Zen Meditation directed by some expert, I found the mountains were no longer mountains and the rivers no longer rivers. Finally, when it reached the last stage of the training, I became mature, the mountains appear to be mountains, the rivers to be rivers again!"

Qing Yuan Wei Xin, a Zen Buddist Monk

1. First stage:
   Empirical or technical knowledge, i. e. knowledge of phenomena.
   [Programming as a technology]
2. Second stage:
   Conceptual or formal-theoretic knowledge, i. e. knowledge of noumena.
   [Programming with specification]
3. Third stage:
   Realistic knowledge, i. e. knowledge as the result of synthesis of above two sides.
   [Programming from the viewpoint of the whole process of development]

1.2 Empirical Knowledge vs. Conceptual Knowledge

Let us make some epistemological analysis of both the empirical side and the conceptual side of knowledge. Empirical item of knowledge can be analyzed into three kinds:

(i) sense data from sensation

(ii) percepts from perception

(iii) experiences from practice (including scientific experiments)

Sense data is the pure material received from outside in the process of sensation. It might be the concrete impressions of color, sound, etc. In fact, they are the result of analysis instead of real impression. No pure sense data without mixture of conception can be

found in human mental activity except, maybe, in the mind of a new born baby. What empirical knowledge can be found in adult's mind are percepts in perception, which are in fact a kind of data getting from sensation through the conceptual categorization. As for human experiences, they are formed through repeated practices which are complicated processes consisting of both the conceptual activities (such as conceptual analysis or synthesis, abstraction, deductive inference and induction) and the actions in dealing with or even making change of the external objects.

As for the conceptual items of knowledge, there are also two different views. One is the purely static view. They consider the concept to be formed by static abstraction of the common essential attributes of the set of objects. Another is the evolutionary view. Although they could agree that static abstraction plays an important role in formation of concepts, they think, factually, real formation of concepts is a process in which the static abstraction plays only a partial role. The reason is that: owing to the limitation of human cognition ability at any concrete time and the infinitude of changing status of the external world, to know the essence of a set of objects either by any individual or by the whole society of human being is a process. Concept formation cannot be done all of a sudden or statically. To be the result of a process, any concept can only be imperfect and evolutionary at any step. From this point view, concepts are not only abstract but also concrete. They have abundant contents and can evolve step by step toward the truth. I believe, the concepts in the mind of the scientists are of this kind. Philosophically speaking, this viewpoint of concept is anti-platonic but more realistic and scientific.

To summerize: on the one hand, phenomenological (empirical or technical) knowledge is the basic and important foundation of scientific truth. But it is meaningful only when it is the correct and appropriate reflection of the essence (noumena) of the objects. There are also illusive, incorrect or irrelevant phenomena occurring in the process of knowing; on the other hand, noumenalogical or conceptual knowledge is the most important item in formatism of the knowledge of the essence of the object. But abstraction, as the basic means to form concepts, could also be wrong, incomplete or misleading. Consequently, some concepts may be incorrect, imperfect or twisted images of the objects. As the result, in order to enhance the reliability and productivity of programming, following two points must be taken into consideration: (i) Programming development must be viewed as a process of successive correctness-checking and refinement not only of its technical side but also of its conceptional side. (ii) In order to carry on this process more smoothly, the technical side (or dynamic semantics) and the conceptual side (or static semantics) of a program had better be unified in an uniform formal framework. Our experience has shown this kind of unification could facilitate solution of many difficult problems.

1.3 Pragmatism, Rationalism or Confucianism

We find, this problem is deeply related to the traditional culture and philosophy, the tradi-

tional cultural and philosophical background (i. e. Pragmatism (Empiricism) and Rationalism) could push the researcher or constructor of either side in programming to its radical extremity. This kind of one-sidedness could lead to the dead end of the road and make the problem concerning enhancement of productivity and reliability more difficult to solve, because obviously as we have analyzed in 1.2 no matter how great is the difference and contradition between the characteristics of empirical knowledge and those of conceptual knowledge, these two sides can not be separated absolutely from each other and to neglect either side can only make our knowledge far from truth. For this reason, I believe, software R&D has reached a stage in need of finding a new philosophical basis to direct its way of thinking. This new approach could be characterized by following aspects:

(i) We must adopt a realistic (instead of an idealistic) attitude toward this problem. In order to synthesize the technical side and the conceptual side into an uniform formal framework, the radical attitude of either side must be avoided. I believe, Confucius's "Doctrine of Golden Means" and Aristotle's thoughts on "appropriateness" or "the intermediate" are suitable philosophical methodology, for this purpose.

(ii) Because the technical ways of programming (based on von Neumann Computer) has been widely adopted and used for almost 40 years. This conventional style of programming could not be replaced or neglected in the foreseeable future. In order to make the new approach acceptable to the industry, this conventional programming style must be kept in the future software. As well known, the most basic characteristic of conventional style of programming based on Von Neumann computer is state-transition. Consequently, to represent this characteristic in the same formal framework with specification must be the most essential step. Only in such a program framework, the wide users could find "mountains and rivers appear to be mountains and rivers again".

(iii) Conventional programming is a mixtual of logical ways of thinking and pragmatical conventions. We find, the theoreticians neglect too much the logical mechanisms existing in conventional programming and also the pragmatical conventions include some unnecessary illogical elements which ought to be eliminated. If these two sides have been treated more appropriately, I believe the gap between the theoreticians and the engineers might be much more narrowed. In order to explain this approach, in more technical details, some major concepts of the XYZ System is outlined in following section.

2 XYZ System

2.1 Basic Command Form of XYZ/BE

$$LB = y \wedge R \Rightarrow \$O(v1, \ldots, vk) = (e1, \ldots, ek) \wedge \$OLB = z \quad (1)$$

$$LB = y \wedge R \Rightarrow \Diamond (Q \wedge LB = z) \quad (2)$$

here, Eq. 1, Eq. 2 represent the dynamic (executable) semantics and the static (specifi-

cation) semantics of a program respectively; " $Ovi = ei", R, Q, represent assignment, pre and post conditions respectively.

$$LB = y \Rightarrow Q \ \$W(R \wedge \$OLB = z) \quad (3)$$

here, " $W" (or $U) represents "unless" (or "until") is a suitable command form to represent interrupt, or exception. Parallel statement is of following form:

$$LB = y \wedge R \Rightarrow \| [Pr1(\overline{par1}), \ldots, Prk(\overline{park})] \quad (4)$$

here, Pri (pari) is the call of a process or an agent instance, we also have select statement to represent non determinancy and the commands for communication which is of the form Eq. 1 by replacing " $O(v1, ... , vk) = (e1, ... , ek)" with "ch?x" or "ch! y".

```
ObjectOrientedProgram ::= %OOPROC ProgramName ==
□ [[SharedvarDeclParts;]
    [TypeDeclPart;]
    [RigidvarDeclPart;]
    [LibraryDeclPart;]
    [ProcDeclPart;]
    [ProsDeclPart;]
    [AgentDeclPart;]
    ProBody]
[WherePart]

ProcDecl ::= %PROC ProcName (n, par) ==
□ [[SharedvarDeclParts;]
    [ProcDeclPart;]
    ProBody]
[WherePart]

ProBody ::= %ALC [ce_1; ... ; ce_m]
```

here ce_i represents a Basic Command, ";" is a notation to represent "conjunction" in command level.

```
%PROC P(n, par) == □ [Declarations;
%ALC [LB = START_P⇒ ... ;
        LB = l1 ∧ n = 0 ⇒ ... ;
        LB = l1 ∧ n > 0 ⇒ ... ;
        ...
        LB = li ⇒ P(n - 1, par);
        LB = li + 1 ⇒ ... ;
        ...
```

LB = lk ⇒ $OLB = RETURN]

[WherePart]

here "RETURN" represents "POP(RST)".

Procedure call

LB = lj⇒P(r, par);

This is an abbreviated form of following two commands:

LB = lj⇒ $O(finps) = (ainps) ∧ $OPUSH(RST, l_j + 1) ∧ $OLB = START_P;

LB = lj + 1 ⇒ $O(aoutps) = (foutps) ∧ $OLB = NEXT;

Process instantiation

$OPrji = = Prj|i|

In order to represent autonomeous object in OOP, we define a new concept of agent as a pair [process, package], here the package part is a data module as class, which represents the static semantics of encapsulation, but the process part called the manager of an agent, is used to represent the dynamic semantics of communication. The relationship between these two parts is realized by exporting all the operations and variables defined in the package part into the process part as its local procedures and local variables, and this connection is implemented in compilation time. Besides, we have made analysis of the parallel statement and found a way to gurantee the decomposability of the semantics of a parallel statement. This temporal logic language can also be used to represent real time programming and distributed programming.

2.2 SE Tools & Applications

1. Basic Tools: five groups.
   (a) Verification, consistency checking and automatic generation.
      i. XYZ/VERI: Hoare logic verifier.
      ii. XYZ/RTG: Realtime program generator.
      iii. XYZ/DLF: Deadlock free condition.
   (b) Specification and rapid-prototyping.
      i. XYZ/SPEC: a tool to support stepwise refinement programming.
      ii. XYZ/PROT: a tool for rapid prototyping.
   (c) Graphic tool.
      i. Graphic tools to support distributed programming.
         A. XYZ/DPD: a graphic tool for decompositional design of distributed program diagram.
         B. XXYZ/CFC: a graphic tool for decompositional design of XYZ/BE programming.

ii. Graphic tools to support information system design.

A. XYZ/DFD: a graphic editor of Data Flow Diagram.

B. XYZ/DDA: a graphic editor and analyser of Data Dictionary.

C. XYZ/PAD: a graphic tool to support the stepwise design of XYZ/SE program diagram.

(d) Language transformation tool.

i. XYZ/NCAE: a tool to evaluate the static semantics (represented by attribute grammar) of the source language.

ii. XYZ/CCSS: a tool to transform the source language into XYZ/E in a formal way.

iii. XYZ/BESE: a tool to transform XYZ/BE into XYZ/SE.

iV. XYZ/E-C: a tool to transform XYZ/E into CC++ (concurrent Cplus modified C++).

(e) XYZ/OOMM: modular management and integration tool.

2. Some Application Experiments

(a) Domain-oriented application.

- Transformation of Domain specific languages into XYZ/E
  By means of a tool XYZ/CCSS, we have transformed International Standand languages SDL, VHDL and a major part of Estelle into XYZ/E in a formal way.
- Some special applications: Animation system design.

(b) For OOP (or DP).
We define an intermediate language CC++ (or Java) between C and XYZ/E.

(c) Reactive and control system (CIMS).
(in research)

(d) Software Development Process and reengineering.
(in research)

2.3 Examples

Example 1. Represent z = gcd(a,b) for positive integer a and b with XYZ/AE, XYZ/BE and XYZ/SE.

(i) in XYZ/AE specification form

$\Box$ [LB = START_gcd $\wedge$ a > 0 $\wedge$ b > 0
$\Rightarrow \Diamond$ (LB = STOP $\wedge$ z|a $\wedge$ z|b $\wedge$ \$A x(x|a $\wedge$ x|b $\rightarrow$ x|z))]
WHERE \$A m, n(m|n == \$E r(n = m * r))

(ii) in XYZ/BE command form

$\Box$ [LB = START_gcd $\wedge$ a > 0 $\wedge$ b > 0 $\Rightarrow$
\$Ox = a $\wedge$ \$Oy = b $\wedge$ \$OLB = l1;

```
    LB = l1 ∧ x = y ⇒ $OLB = l3;
    LB = l1 ∧ x ≠ y ⇒ $OLB = l2;
    LB = l2 ∧ x > y ⇒ $Ox = x - y ∧ $OLB = l1;
    LB = l2 ∧ x < y ⇒ $Oy = y - x ∧ $OLB = l1;
    LB = l3 ⇒ $Oz = x ∧ $OLB = STOP]
```

(iii) in XYZ/SE structured statement form

```
□ [LB = START_gcd ⇒ $O(x, y) = (a, b) ∧ $OLB = NEXT;
   * [LB = l1 ∧ x ≠ y ⇒ $OLB = l2 | $OLB = EXIT;
      LB = l2 ∧ x > y ⇒ $Ox = x - y ∧ $OLB = l1
                     | $Oy = y - x ∧ $OLB = l1];
   LB = l4 ⇒ $Oz = x ∧ $OLB = STOP].
```

Example 2. Recursive procedure of factorial.

```
fact(%INP n: INT;%IOP a: INT) ==
□ [%VAR [w: INT];
%ALC [LB = START_fact ⇒ $OLB = l1;
        LB = l1 ∧ n = 0 ⇒ $Oa = 1 ∧ $OLB = l3;
        LB = l1 ∧ n ≠ 0 ⇒ fact(n - 1; w | a);
        LB = l2 ⇒ $Oa = n * w ∧ $OLB = l3;
        LB = l3 ⇒ $OLB = RETURN]]
```

Example 3. Producer-Consumer Problem.

```
%PROC producer == □ [
%VAR [i: INT];
%PROS [
producer (%INP j: INT;%CHN ch (*, con: NM);%INP sum: INT) == [
%LOC [i, prodmo: INT];
%STM [LB = START ⇒ $OLB = NEXT;
LB = l0 ⇒ $Oi = 0 ∧ $Oprodmo = l2 ∧ $OLB = NEXT;
*[LB = l01 ∧ (i < m) ⇒ ($OLB = NEXT| $OLB = EXIT)
   ? [LB = l1 ∧ (i < sum) ⇒ $OLB = l11 | ¯ ⇒ $OLB = l1;
      LB = l11 ⇒ $Oi = i + 1 ∧ $Oprodmo = prodmo + i * 2 ∧
            $OSWRITE(j) ∧ $OLB = NEXT;
      LB = l12 ⇒ $OSWRITELN( "Producer produce" i, prodmo) ∧
            $Och!prodmo ∧ $OLB = EXIT;
      LB = l1 ∧ (i ≥ sum) ⇒ $Oprodmo = -1 ∧ $Och!prodmo ∧ $OLB = EXIT]]
      LB = l1001 ⇒ $OLB = STOP]
];
consumer(%CHN ch1(pro: NM, *), ch2(pro1: NM, *)) == [
```

```
%LOC [j: INT];
%STM [LB = START⇒ $OLB = NEXT;
        LB = l0⇒ $Och1?j;
        ?[LB = l1 ∧(j>0)  ⇒ $OLB = l111
                    |(j<0)  ⇒ $OLB = l121
                    |¯ ⇒ $OLB = EXIT;
        LB = l111⇒ $OWRITELN("Consumer receive from 1", j) ∧
                    $OLB = EXIT;
        LB = l121⇒ $Och2?j∧ $OLB = NEXT;
        ?[LB = l21 (j>0)⇒ $OLB = l1211
                    |(j<0)⇒ $OLB = l21
                    |¯ ⇒ $OLB = EXIT;
        LB = l1211⇒ $OWRITELN("Consumer receive from 2", j) ∧
                    $OLB = EXIT]]]
        LB = l101⇒ $OLB = STOP]
];
%STM [LB = START⇒ $Oi =5∧ $OLB = l0;
        LB = l0⇒ $Oa1 == producer|1|∧
                 $Oa2 == producer|2|∧
                 $Oa3 == producer|3|∧
                 $Oa4 == producer|4|∧
                 $Ob2 == consumer|1|∧
                 $Ob1 == consumer|2|∧
                 $OLB = NEXT;
        LB = l1⇒ ||  [a1(%INP 1|j;
                        %CHN c1(*, b1) | ch(*, con);
                        %INP i| sum);
                     a2(%INP 2|j;
                        %CHN c2(*, b1) | ch(*, con);
                        %INP i| sum);
                     a3(%INP 3|j;
                        %CHN c3(*, b2) | ch(*, con);
                        %INP i| sum);
                     a4(%INP 4|j;
                        %CHN c4(*, b2) | ch(*, con);
                        %INP i| sum);
                     b1(%CHN c1(a1, *) | ch1(pro, *);
                        %CHN c2(a2, *) | ch2(pro1, *));
                     b2(%CHN c3(a3, *) | ch1(pro, *);
```

```
                    %CHN c4(a4, *) | ch2(prol, *))];
          LB = l2⇒ $OLB = STOP]].
```

Acknowledgements

This work was supported by National Natural Science Foundation, National High Technology Program (863 Program), and the Ministry of Electronic Industry of China.

References

[1] Manna, Z and A. Pnueli, "Clocked Transition Systems," *Logic and Software Engineering* (eds. A. Pnueli and H. Lin), World Scientific, Singapore, 1996.

[2] Tang C. S. and XYZ Group, "An Outline of the XYZ System," *Logic and Software Engineering* (eds. A. Pnueli and H. Lin), World Scientific, Singapore, 1996.

XYZ 系统的哲学背景

（国际易学研究，1998）

一、问题的提出

（一）合久必分，分久必合

从软件过去发展的历史看，可看出如下一种规律：“合久必分，分久必合。”在70年代前，软件统一于高级语言与编译技术。后来出现形式语言与语法分解理论，但理论与技术亦是紧密结合的；硬件则统一于冯诺曼体系，高级语言也是适应这一体系而构造的。从70年代起这个统一局面被打破，先是出现各种新软件领域如操作系统、数据库、知识工程等。接着各领域内也出现了各种新技术，门类繁多，理论亦分成许多基础各异（代数、逻辑、可计算函数等）的形式语义体系；语言也分裂成本质上不同的许多类别，各有其应用范围与适应的机器体系；理论与技术这两个领域的研究也相互分离各走自己的道路。正如前面已引述过的美国奥斯汀德州大学两位著名教授 Chandy 与 Misra 在一本书中指出的：“今天，程序设计日益分裂成由许多秘密宗派组成的小团体，每一宗派有自己的牧师、符咒与仪式。”显然，对于该学术领域说，这是一种范型（Paradigm）分裂的现象。一般讲，这种现象的出现对该领域，既是一种危机，也是一种产生飞跃的机会。事实也正是如此，从80年代末期开始，逐步出现由分裂走向统一与相互结合的端倪，不论是从理论上说还是从技术上说，甚至是从理论与技术的关系上说，其总的趋势是走向“合”，要求最终将各种理论、语言、方法与工具进行合理的结合组成一种水平更高，应用效率更高，有助于提高软件生产率的统一的系统。我们研制的 XYZ 系统就是十多年前为迎接这一总趋势而提出的，其中包括一时序逻辑语言 XYZ/E 与一组实现各种方法论的 CASE 工具；其语言将各种分裂的程序设计方式（包括表示状态转换的可有效执行的常见程序设计，基于抽象规范的程序设计，面向对象的模块程序设计及可视图形程序设计等，而其中最基本的一对矛盾即静态语义与动态语义的对立）以一种一致的逻辑框架统一起来，可作为所有各种工具的共同形式语义基础；而这些 CASE 工具又实现了各种方法论，它们相互可以联结而成为统一的支撑系统。其中时序逻辑语言与 CASE 工具相互反映对方的特征而彼此渗透从而可方便地组成协调的统一体。此系统的最根本的特征即体现了一个“合”字。由于这一系统的提出是在“合”的

大趋势到来之前，故到目前为止它一直走在这一潮流的前沿而不是跟在这潮流之后。

从哲学上看，“合久必分，分久必合”这种长期在我国流传的历史观，事实上是黑格尔强调的“正反合”辩证法关于事物发展的方式的一种变形；“分”就是强调矛盾的对立性，“合”就是强调矛盾的统一性。

（二）中庸之道

在事物发展的以“分”为主，与以“合”为主的这两种不同阶段，事实上各有其强调侧面不同的哲学方法（有形或无形地）作为引导。在“分”的发展阶段通常起作用的方法是基于逻辑的分析方法，事实上，科学早期的发展多是从分类学开始的。此外，还有如“分而治之、各个击破”（divide and conquer）的方法以及各种讲兵法的书中总结的许多在斗争中克敌制胜的方法等。“合”的科学方法最典型的是综合法。从工业品的设计到语言翻译都在分析阶段后有综合阶段。“综合”就是将已分开的各部分找到其可互相适应的条件又以合适的分寸将它们组织成统一的整体。对于这样一种“合”的科学技术方法，其基本的哲学指导思想应该是什么？据我所知，在历史上东西方各有一位大哲学家讨论过这个问题，在东方即孔子的中庸之道，在西方即希腊哲学家亚里士多德，他讨论过关于适中性的中道思想。这两位哲人在这个问题上的思想是很相近的。我们此处主要采用孔子的中庸之道，概括起来有以下几点：（1）从实际出发而不是从主观的概念与形式出发，《中庸》一书中说：“极高明而道中庸。”与“高明”相对立的“中庸”的特征即在于其切近实际与平凡事物，如朱熹的解释：“庸，平常也。”（2）从变化中对具体时间地点等各种条件进行具体分析。“君子之中庸也，君子而时中”。孔子是“圣之时者也”。孟子发挥这一思想提出“权”的概念。“子莫执中，执中而无权，犹执一也”。“可以取，可以不取，取伤廉。可以与，可以不与，与伤惠。可以死，可以不死，死伤勇”。也就是不从形式上僵化地去理解“廉”、“惠”、“勇”这些道德概念，而强调其精神实质。“义”正是“权”的结果所产生的行动，“大人者言不必信，行不必果，唯义所在”，“大德不逾限，小德出入可也”。都是强调根据实际的变化了的具体情况，作具体判断以采取行动。（3）所谓“中”就是掌握合适的分寸，过犹不及，要做到恰如其分。即所谓“发而皆中节谓之和，致中和，天地位焉，万物育焉”。也就是孔子所说的，“中庸其至矣，民鲜能久矣”。“道之不行，我知之矣，知者过之，愚者不及也”。“执其两端，用其中于民。”这种根据具体情况，找出既非“过”，又非“不及”的合适的分寸，也就是亚里士多德所强调的“中道”思想。这种处理问题时要求根据具体情况找到恰到好处的分寸的中道，绝对不能与不顾具体情况形式主义地调和折中混为一谈。亚氏在这点上有很透辟的说明。他说过：“相对于我们而言的中道（the intermediate）就是既非过分亦非不足。以这样的方式，每一科学专家避免过分与不足而求中道。这就是每一科学以集中注意于中道的方式生产出好的产品。……因为他们假定过分与不足都将摧毁好的（结果）而只有中庸（mean）能够保护它。”“美德是一种适中（appropriate）……过度和不足是恶行的特性。……所以，美德是牵涉到选择时的一种性格状况，一种适中。也就是说，它为一种合理的原则所规定，这就是那具有实践智慧的人用来规定美德的原则。它是两种恶行之

间的一种中道。因此，就其实质和就表达其本质的定义而言，美德是一种中道，而从德行与道义来说，则它是最高水平的。”（亚里士多德：《尼各马可伦理学》Ⅱ）因此，实行“中庸之道”或“中道”这种方法论，必须对具体情况作具体分析，巧妙地找出最合适的“节”或“度”，而应该避免形式主义、主观主义或盲动。我认为，这是处理实际问题时能否成功的关键。但是，作为一种哲学方法，它只能告诉你这样一种原则，强调其重要意义引起你对此的重视，而不能告诉你在你遇到的一种具体情况下，什么是合适的分寸或如何找到这具体的“节”与“度”让你不动脑子去机械地执行。因此，这一方法是指导有分析能力的人根据实际情况进行判断以采取行动的方法，而不是提供给愚人或懒汉去盲目执行的方法。正是由于常常有人误解了这一方法的精神实质而陷入折中调和的困境，反过来又强加于这一方法以折衷主义的罪名。我认为，出现这种后果是出于对这一方法的误解，不应由这一方法来承担罪责。我深深感到有些理论科学家在研究规范语言时以及引用 CASE 工具时所采取的态度是令人怀疑的，因为有时他们采取的途径在我看来是不顾实际情况及问题的实质而以形式主义或以“削足适履”的方式处理问题。当然作为时序逻辑语言的研究工作，绝对不应否认逻辑分析的意义，也绝对不能在正确性问题上降低逻辑精确性要求。但我们分析应以实际情况为基础而不能以主观界定的概念为基础。我们采用“中庸之道”作为指导思想对问题进行具体分析，往往能在“山重水复疑无路”的情况下，找到一种简单而又能解决问题的实际方法，达到“柳暗花明又一村”的效果。例如在“框架假设”，“速成原型”，“实时程序设计”，“并发语句语义可组合性”以及“面向计算过程的大型程序模块”等方面都出现过这种情况。我想，应该说，正是采用了这样一种正确的哲学方法作为指导才使我们的具体技术工作避免了许多歧路。我们这种处理问题的方法常常不易为西方一些理论家所理解。最近，Manna 与 Pnueli 两位教授对我们的工作及其方法论作了如下评述：“唐稚松教授使时序逻辑概念超越了任何人的想象，他将之应用于各方面，在以前无人认为是可能的。”所有这些情况使我感到，我们所采取的途径有时与西方理论家的途径存在某些差异是与我们所出发的哲学基础的不同有关的。

（三）动与静

“分”与“合”即辩证法所说的矛盾的“对立”与“统一”。这里所说的“矛盾”是事物发展过程中出现的实际不协调因素，而不是形式逻辑所讨论的概念不一致性。在前一节中讨论了儒家“中庸之道”作为一种方法论对于 XYZ 系统设计思想的影响。但这一哲学方法只是强调了辩证法中对立的统一性这一个方面，据我所知，在整个孔子哲学体系中并未全面地着重讨论辩证法。我国传统哲学中，不少流派都强调辩证法，而《易经》更是我国哲学史上最全面最深刻的一部弘扬辩证法思想的经典著作。首先它以抽象的概念“阳”与“阴”表示宇宙间一切事物的最基本的一对矛盾元素，并予以符号化（乃至数字化）的表示，即“—”与“- -”。实际上，这对基本元素是宇宙间“男”与“女”，“天”与“地”，“刚”与“柔”等等人类或自然界各种对立元素的抽象概括，它们表示着一切矛盾的最基本的共性。以这对根本矛盾为基础，《易经》讨论了天、地、人，万事万物的发展变化过程。而这些

变化规律中一个重要的方面即“动”“静”结合，这是它与“佛学”及魏晋“玄学”最基本的差别所在。例如：“生生之谓易”，“天行健，君子以自强不息”，“刚柔相推而生变化”，“穷则变，变则通，通则久。”在《系辞》中，更明确提出：“动静有常，刚柔断矣”，“居则观其象而玩其辞，动则观其变而玩其占”（此处“居”即指稳定静止的状态），“夫乾，其静也专，其动也直，是以大生焉。夫坤，其静也翕，其动也辟，是以广生焉。”我们恰好在计算机软件的“动态语义”与“静态语义”的对立与统一中，找到了一种解决软件发展面临的重大问题的方案。事实上，自有冯诺曼计算机以来，每一程序均由两部分组成，即其变量与操作（过程或进程）的定义部分（常称说明部分）与其算法过程的执行部分（常称程序体）；前者的语义是静态的，后者的语义则是动态的；前者是关于其领域情况的规定，后者是关于其计算过程的描述。这两方面是相互依存，都是不可缺少的。可是从过去四十多年的发展史来看，国际计算机界前二十年专注于动态语义，几乎忘记了静态语义；后二十年专注于静态语义，又几乎忘记了动态语义。我们认为，各有其片面性。让我们回忆一下过去的情况。1950 年出版的《Automata Studies》（ed. C. Shannon and J. McCarthy）是计算机出现初期讨论计算机模型的会议论文集。与会科学家提出二类自动机（即图灵机与有穷自动机）作为计算机的模型，它们的共同之处即由下表 1.1 所表示的有穷状态转换。

表 1.1

条件 / 状态	C_1	…	C_j	…	C_n
…	…	…	…	…	A_{1n}，S_{1n}
S_i	A_{i1}，S_{i1}	…	A_{ij}，S_{ij}	…	A_{in}，S_{in}
…	…	…	…	…	…
S_m	A_{m1}，S_{m1}	…	A_{mj}，S_{mj}	…	A_{mn}，S_{mn}

此处，$i=1$，…，m；$j=1$，…，n；A_{ij}表示动作，S_{ij}表示下一状态。表中有穷状态转换即表示了自动机执行过程的动态语义，至于自动机的构造（包括带子，读头等）、条件、状态、动作等的定义（即其静态语义）都包含在自动机的概念定义之中，在具体用自动机解题时，已被隐藏；作为计算机的模型完全由表 1.1 所表示的动态语义所代表。这一片面强调动态语义的做法一直从 50 年代初维持到 70 年代中叶。当程序语义形式化问题首先被提出时，J. McCarthy 所提出的方案即试图用一阶逻辑表示动态语义。这一方案后来又被采纳为第一个维也纳定义语言，但终于以失败而告终。此时正好 D. Scott 以递归函数及 λ 演算为基础提出表示程序静态语义的指称语义方案，它很快即被广泛采纳，后来又扩充了与模块程序、并发程序等方面有关的形式化方法。但就总体情况来看，从 80 年代起，动态语义方面从未引起足够的重视。而从我们的观点来看，一些关系软件生产率的关键性问题，如可靠性、可维护性及可重用性等的解决与如何合适地将静态语义与动态语义两方面结合是紧密相关的。XYZ 系统的一个主要特征，即以中庸之道为指导统一有关软件的理论与

技术的许多方面的矛盾，而其中最基本的一种即以统一的程序框架表示动态语义（赋值）与静态语义（pre 与 post 断言）；在此基础上即可将逐步求精、程序规范、验证与速成原型表示在一平滑的过渡过程之中。这是问题的一个方面，即西欧学术界所强调的以大型程序模块的可重用性为基础提高软件生产率的道路。他们所强调的面向对象程序设计，的确有其重要意义，但它们主要是关于领域情况的描述，其结构主要由数据结构所决定，通讯在其中只起局部的（为了实现操作调用这一种特殊情况）及附属的作用，故其语义主要是静态的。而近年来由于大型通讯系统的发展，分布式与反应型程序的重要性日益明显。因此，在面向对象这种面向领域的模块之外，还需要另一类面向计算过程的大型程序模块，其语义则是动态的，其结构特征主要由动态通讯所决定。但这种模块的语义只有在满足语义可组合性条件的前提之下才成立。如何保证这条件成立是长期以来并发性研究中的一个难题，有些语言（如 Chandy 与 Misra 的 Unity）为了使这条件成立而被加以过多的限制，致使它们无法在工业中推广应用。XYZ 系统则是应用动态语义与静态语义相互有机结合的方法（即用静态语义的形式验证的方法作为手段找出防止起破坏作用的动态语义性质）解决了保证此一条件成立的难题，它对语言的表示力不加任何限制，因此具有很好的实用性。到目前为止，还未见其它系统做到了这一点。由以上情况可见，动态语义与静态语义相结合不论是基于形式语义保证软件可靠性与可维护性方面或是基于模块化保证软件可重用性方面都为提高软件生产率提供了一条新的道路。这是 XYZ 系统应用我国传统哲学指导思维方法将动态语义与静态语义紧密结合起来的主要收获。

（四）理性主义与实用主义

如上面已经指出的，在增强可靠性提高软件生产率这一目标如何达到这个问题上，在过去十多年中，西欧学术机构中的理论计算机科学家与美国工业界的工程技术人员实际上走了很不相同的两条道路。西欧这些科学家强调研制以形式化理论为基础的抽象规范语言及相应的形式语义理论，而美国工业界则更强调支撑各种方法论的 CASE 工具。就本身说，我们认为这两个方面的研究都是有意义的，在实现加强可靠性与提高软件生产率的目标方面都是不可缺少的。但我们又认为这两方面要取得实效应该相互紧密结合互相渗透，彼此考虑另一方为自己一方服务以适应己方特征。可是过去十多年的情况，不幸的正是沿着各执一端的道路在发展，其后果是两方的距离日益扩大而不是日益接近，各从一个方面走向片面性。西欧一方发展了一整套远离工业界要求的数学理论。花费巨资构造了巨大的理论性空中楼阁，其结果只能使工业界对它敬而远之；美国工业界一方构造了许多 CASE 工具，这些工具在某些方面的应用中还是可起一定作用的，但因它们缺少一共同的形式化语义基础，这些工具很难灵活地连接起来组成能适应各种不同需求的，其功能更为强大、结构更为复杂的工具环境，因此其作用有很大的局限性。几乎所有单纯依靠软件技术构造的规模较大的集成化环境，除设计者本身外，很难推广应用。我认为产生以上两种片面性的根本原因在于这两方面的研究均无形中受到所在地区传统哲学思想的引导从而走向极端。即西欧的理性主义与美国的实用主义（实用主义的认识论为经验

主义，不过这一哲学流派是美国工业社会流行的个人中心与实用化意识影响下的经验主义）。这两种哲学在一定限度内各有其正确的一面，但走向极端也均有其片面性。这里我想引用我国古代禅师青原惟信的话：“老僧三十年前未参禅时，见山是山，见水是水。及至后来亲见知识，有个入处，见山不是山，见水不是水。而今得个休歇处，依旧见山只是山，见水只是水。”（《五灯会元》卷十七）。从知识论的角度来解释这段话，人们认识真理的过程可分为以下三个阶段：第一个阶段是认识事物对象的现象阶段，也就是经验性、技术性阶段。在此阶段，即按事物所表现的现象认识事物，故见山是山，见水是水，计算机程序即为人们从技术特征所了解的计算机程序。随着认识的深化，经过人们对其经验（或技术）进行理性加工（也就是分析、抽象、推理、归纳），人们对于事物对象的认识就上升到第二阶段，亦即认识对象的本质阶段，或者理性认识阶段。此时，由于“山”与“水”的本质是决定“山”与“水”所表现出的现象的基础，它更具有“山”之所以为“山”，“水”之所以为“水”的内在特征，通过理性认识阶段所形成的对山或水的认识就更具有真理性。它与第一阶段所形成的关于山或水的现象的认识有本质的不同与飞跃，这就是为什么在这阶段见“山”不是山，见“水”不是水。类似地，对于计算机程序的理性认识即构成其抽象描述，也就是规范，这是对程序的更为本质的特征揭示，自然是更为深刻的认识。此阶段所得到的关于程序的抽象描述显然与第一阶段单纯从技术方面以高级语言形式所表示的常见算法程序大大不同。西方大多数以理性主义为背景的理论家所得到的成果（形式语义理论及规范语言）大多属于这一类。我们认为这些研究成果是认识深化的表现，是重要的，但不能止于此。如果认识仅止于此，则仍仅达到“见山不是山，见水不是水”的阶段。但人毕竟是生存于现实世界中，技术科学研究不能仅止于理性主义的理念世界，最后仍应回到现实世界之中。事实上，现象与本质，技术与理论既是对立的又是统一的。本质不能脱离现象而存在，理论不能脱离技术而独立存在。那种脱离现象而独立存在的理念只是一种柏拉图式的想像，作为理论陈述的方便自然是允许的，但作为指导实践的技术科学则不可。事实上，本质从来就是存在于现象之中的，现象有些是歪曲本质的，有些则是反映本质特征的。这些反映事物本质特征的现象（或具体事物）即是通常所谓具有典型性或代表性的事物现象。事实上，通过对感性认识的分析、抽象、推理、归纳等过程，使认识越来越反映事物本质的特征，删去不反映本质甚至歪曲本质的那些感性材料，然后将剩下的能从不同方面反映事物本质、具有代表性的现象按该事物的本来面目加以重新综合使认识的对象一方面具有事物的原貌，另一方面又能更直接地反映该事物的本质。这就是说，这种反映本质的各方面现象的综合物才是对象的本质最全面的反映。此时见山仍是山，见水仍是水，可是这样认识的山和水已和第一阶段认识的山和水在深刻性方面有本质的不同了。我认为我们对待计算机程序的认识也应如此，既不能否认西方理论家关于规范语言的研究，又不能止于他们的研究，而是应该在实用主义与理性主义之外（或之上）走既同于二者又异于二者的第三种道路。由于这条道路是以唯物辩证法与逻辑分析方法相结合而不是相排斥为特征的，所以我们认为，当前最重要的是需要提出一种统一化的哲学理论（特别是知识论理论），它应包含了中西各重要哲学流派的精华（也就是其正确的方面），但

也规定了这些思想的局限性，也就是其不应逾越的合理的界限（这也是孔子与亚里士多德均强调的适中）。我们在研制 XYZ 系统时，深深地感到这样一种知识论体系的重要意义。它可以在工作的关键时刻帮助从事研究的科技人员辨明方向，鉴别是非得失以作出决定，为我们树立一个标准与原则。事实上，近年来在国外也有人有类似的要求。比如日本经济新闻社会与日经产业消费研究所出版的《NIKKEI 高技术情报》在 1993 年 1 月份（第 200 期）发行的一期纪念专号，名为“东方的智慧”。其中岛井弘之等写的题为“渴望与现代科学融合——中国哲学思想的启发”的总论中，有如下一段开场白：“现代科学技术是否已陷入某种滞塞状态呢？……这种停滞的原因也许包括西方理性主义（rationalism）对事物的看法和行动原理，至少可以说是根本原因之一。当前是重新考虑现代科学基础的思想方法的好时期。目前世界上存在着许多与西方不同的文化思想的区域，自然会隐藏着打开该局面的建议。……其中，自古代文明发达以来，中国人发明了印刷术、火药、指南针等丰富而实用的技术，其智慧最能够作为参考对象。”据我所知，日本软件专家熊谷章及日本著名软件工程权威 SRA 技术总裁岸田孝一等均有这方面的论述。岸田孝一曾对 XYZ 系统的哲学背景作了如下的评论：“统一已有的事物，创造新的统一环境，是具有中国面貌的特征的。中国哲学思想中很多是取自‘独创的解释形式’，也许这正是中国哲学思想的真谛吧。对于面向应用软件开发的软件工程来讲，也理应好好参考一下唐氏的方法论。”（见上引《NIKKEI 高技术情报》1993 年第 200 期中，原孝二等人的文章：“吸收欧美长处用中庸之道统———对重视应用的日本有参考价值”）在 1995 年，他在《朝日新闻》（夕刊）（1995 年 12 月 4 日）曾发表专文介绍 XYZ 系统，在该文末尾，他写了如下的一段话：“虽然这系统（XYZ 系统）所采用的基础数学理论来源于西方，但构造此系统的基本思想却来自于孔子的中庸哲学以及佛教禅宗的论理哲学。这也许可以说是东方文明对于新的 21 世纪计算机技术发展的一大贡献吧。”

（五）立足于我国实际的中西文化融合

这样一种统一化知识论体系不仅为软件工作或某些高技术研究所需要，它还是当前所处时代的要求。我认为，我们目前正处于一个新时代到来的前夕。不难理解，以西欧北美为代表的资本主义工业化社会的基督教文明经过了过去四百余年的发展，已经由兴盛繁荣而逐渐走过了其鼎盛时期，不论从政治、经济、社会到科技文艺都已显现出式微的趋势。而东方亚太地区则由百多年前其封建社会在西方殖民者兵舰的威胁下崩溃，经过中间殖民地受欺压的阶段以及最近二十年的逐渐兴起而成迅猛发展之势已成为举世瞩目的事实（这种发展的趋势可能因一时的错失而走一些弯路，但看来不致改变其基本方向）。在这一地区，不论从人口的众多和地域的广阔，还是从文化的悠久与影响而言，我国自然是处于一种十分特殊的地位。从文化与哲学而论，为迎接这一新时代的到来，中西文化与哲学的关系问题自然是一个意义重大的问题，其影响必将十分深远。只要看看上面引述的日本专家从高科技的发展出发追索到中国哲学这一点即不难看出其意义。我在此想顺便就中国与外来文化思想的关系谈一点个人的体会。回顾一下历史，数千年间外来文化思想进入中国最大规

模的有三次：第一次是从汉代开始的印度佛学的传入。第二次是随着西方传教士与商人的到来以及与之相配合的炮舰的侵入，西方基督教文化与思想的传入。第三次则是随着十月革命的一声炮响送来的马列主义。在这三次外来文化的传入中，第一与第三次都在经过一段时间的碰撞与摸索之后，终于找到了中外文化相互渗透与融合的途径。唯独第二次西方文化与中国传统文化的融合问题，虽已经近百年的各种形式的碰撞，应该说，至今尚未很好地解决。这的确是一值得深入探索的问题。它不但与我国乃至亚洲现代化有关，而且与如何维护世界和平、人类进步、新时代的顺利发展以及如何协调东亚文化传统与西方文化的关系都直接相联系。我国著名文史大师陈寅恪在 30 年代所作“冯友兰‘中国哲学史’的审查报告”中指出：外来文化思想“即令一时输入，非与我国固有思想相融合，决不能保其势力”。又说：外来思想“输入后，若久不变易，则决难保持。……能于我国思想史上发生重大久远影响者，皆经国人吸收改造之过程。其忠实输入不改本来面目者，……虽震荡于一时之人心，而卒归于消沉歇绝。……窃疑中国自今日以后，即使能忠实输入北美或东欧之思想，其结局当等于玄奘唯识之学，在吾国思想史上既不能居于最高之地位，且亦归于消沉歇绝。……其真能于思想自成系统，有所创获者，必须一方面吸收输入外来之思想，一方面不忘本民族之地位，此二种相反相成之态度，乃……二千年吾民族与他民族思想接触史所昭示者也”。从这段深刻的史识论断中，我们不但要认识中西融合的重要意义，而且还应考虑这种融合远非易事。我认为过去长期以来中西文化融合问题之所以难以解决，既有社会的历史的原因，也有这两种文化本身存在的内在深层矛盾的原因。下面我们想就后一方面的原因进行一些讨论。

从 20 年代以来关于中西文化异同的讨论很多，在我们研究如何建立一种适应新时代要求的哲学体系时，应该考虑这些文化哲学的特性。在此先引述一些熟悉中西文化的哲学家的观点。正为金岳霖先生在“中国哲学”一文中所概括的：(1) 中国哲学非常简洁、富暗示性，易包容独创思想，而在中国古代逻辑与知识论不发达；希腊文化是理智文化，哲学思想观念明确，为科学的发展提供了工具。(2) 中国哲学强调“天人合一”，人与自然的统一，主体与客体的融合；而西方则将人与自然分割。强调人征服自然，这也刺激人去寻求关于自然的知识以期得到征服自然的力量。(3) 中国哲学强调哲学与政治思想的统一、个人与社会的统一；而西方哲学家将哲学与他本人分开，他进行推理、论证等技术性的哲学讨论而不传道，不讲究身体力行。张岱年先生在《文化与哲学》(教育科学出版社：1988 年) 一书中也讨论了这个问题，他认为“中国传统文化比较重视人与自然，人与人之间的和谐与统一的关系，西方文化，从希腊一直到近代，比较重视人与自然、人与人之间的对立和斗争的关系”。此外，“中国古代有一个思维方式，就是富于辩证思维”。“中国哲学家特别重视直觉方法，即‘体认’或‘体验’。……而中国的分析方法不发达。我们的思维方式，要保持、提高辩证思维的长处，但需补上分析方法。……中国过去对分析很不重视，……有些范畴表现为模糊笼统，没有严密性。同时，也没有伽利略的科学实验方法。……应当做到分析与综合的统一，才是比较健全的科学方法”。王浩先生在“西方哲学与中国”(《九州学刊》1986 年 9 月) 一文中也谈到中西文化异同问题：“有几种常见的说法，彼此相关又不完全相同。中国文化注重人生，

注重完美的感性与直觉，注重合，注重和谐，注重身份。西方注重知识，注重理论概念的构造，注重分，注重竞争，注重契约。”又说：“以分与合的对比为例，很容易想到两者间的辩证关系及发展。……合与直觉与具体感在一起，分与形式化及抽象为偶。”他接着说他曾经写过一篇文章讨论直觉与形式化相结合在生物科学上的地位，举例说明成功的科学研究，有赖于直觉和形式化的恰当配合。“……生物界对这篇文章兴趣很大，看作是他们所要的哲学。”王先生在该文结尾列了一个表，“包括了若干形式化与直觉到这一对照多少有关的成对的概念”。如形式与内容，公与私，客观与主观，普遍与特殊，抽象与具体，分析与综合，形式逻辑与辩证法等等。从以上几位哲学家的这些讨论，可以看出，中西文化哲学所强调的重点的差异，事实上是代表着人类思维方式的根本差异。两者各有所长，也各有所短。哪一方强调过头均可能产生片面性。这样的对立的双方不是一方简单地否定另一方能解决问题的。事实上，这种差异或不同，归根到底是辩证思维与形式逻辑思维的差异。王先生在生物学的研究中已看到这两方面合适的结合是成功的关键，本文作者在 XYZ 系统的研究中，也有类似的体会。我认为，为了建立符合新时代要求的哲学体系，应从更高的角度看待中西文化的各方面特征，将各方面的优点综合成统一的系统。正如张岱年先生在上述书中所说的，“要发展新的文化，改造思维方式很重要。一方面要保持过去辩证思维的优点，一方面要注意分析方法，有所改进”。虽然，看到这种综合双方优点的新的思维方式的必要性是很重要的，比过去总是站在一方否定另一方的意义与价值的观点是一很大的进步，但如何能实际做到这一点并非轻而易举的事。我认为，为了迎接新时代的到来，从学术研究来说，首先，需要在知识论的层次上构造一融汇辩证思维与逻辑思维于一体的协调的体系，从理论上先说明两者的位置并如何联系；然后，还要能将这样的思维方式付诸实践（包括科学实践）并在认识世界与改造世界中灵活运用，得到预期的效果，即知见于行。我们在 XYZ 系统的研制过程中已实践过这两方面的结合，下面各节即主要介绍这一知识论体系的提纲，试图从哲学理论上说明这两种结合的基础。这样一种哲学体系的意义在于现实的需要。自从清代以来，在中西文化关系上经过不断的反复的争论，从最早最保守的“西方文化大逆不道”论，到后来的“西方科技文明中国古已有之”论，“中学为体西学为用”论，“全盘西化”论以及具有怀古幽情的“新儒学”等，多数是以站在一方的立场否定另一方为特征的；只有解放前夕清华大学哲学系的金岳霖先生与冯友兰先生的哲学体系，才具有融合中西的特点。但是在他们的体系中均避开了辩证法问题，而是以逻辑分析方法为主的，所以，也未做到思维方法的统一。在思维方法上如何融合辩证思维与逻辑思维，这样一个根本问题一直在我国及国外均未解决。在《实践论》与《矛盾论》中实际上已触及这个问题，但未明确地加以讨论。前面引述张岱年先生的话中已指出了这两种思维方法结合的重要意义，他从迎接新的时代这一大目标的高度提出了这种需要。故改造思维方式使之融合这两类思维方法已经不只是一个学术讨论的问题而是与科学实践密切相关的事了。但据我所知，张先生迄今也尚未能提出一融合这两类思维方式的知识论体系。下面提出的一种统一化知识论提纲，即在为弥补这一空白而作的一种尝试。而我们进行了近二十年的 XYZ 系统的研制，即为这一知识论理论的提出提供了一次在高科技研究

实践中的具体例证。

二、哲学基础

(一) 什么是哲学

几乎每一哲学流派都对这个问题有自己的回答，此处不想一一列举，因为我们所试图建立的哲学不能以现在的任一东西方哲学体系的已有的前提为基础，所以我们需要从“什么是哲学”这一更根本的问题谈起。事实上，只需引用20世纪初英国牛津大学哲学教授布拉德雷（F. R. Bradley）在《现象与实在》一书中的一段发人深思的话：

> 形而上学是一种为人类本能找‘坏理由’（bad reasons）作为根据的学问，这种找坏理由作为根据的行动本身也是一种人类的本能。

对于一位希望从哲学中求得“伟大真知”的人来说这段话无疑是一瓢冷水，大杀风景。但对哲学文献稍有接触的人只要仔细一想，即会感到这是一句老实的真话，任何人只需翻阅一下那些大部头的职业哲学家的著作，即不难发现，其中绝大部分内容都是在讲一些“坏理由”，即一些不成其为实质性道理的道理。就西方哲学著作来说，哲学家为了说明其哲学思想为“伟大的真理”，一般采取两种途径：一种是讲“兜圈子”的道理。其做法是为了表示其哲学思想为“无可怀疑”的真理，常企图使其哲学具有欧几里德几何学那样的逻辑体系。因这体系中的定理是经过严谨的逻辑推理来论证的，因此，颇具“无可怀疑”的外观。可是，内行人都知道，这种逻辑体系的公理或公设以及其推理规则本身事实上是被假设其成立，不能在这体系内再加论证的。因此，这种逻辑系统所赖以成立的起点本身并非“无可怀疑”的。事实上，就以欧几里德几何为例，在它之外尚有非欧几何体系并且已在物理学得到应用。因此，欧几里德几何也并非“无可怀疑”的了。哲学家明知这一情况，常常采用并不光彩的办法论证其理论成立。即用“兜圈子”的办法，用其理论的一部分为依据“论证”其另一部分“成立”，然后，反过来，又用后者“论证”前者“成立”。比如，常见的办法是用其方法论（逻辑或辩证法）论证其本体论成立，然后再又用后者作为方法论的基础以说明其方法论的正确性；另一种情形即用其知识论与本体论互相论证。众所周知，这样的论证事实上没有任何意义，最多只能说明其哲学体系中这两部分是互相协调的而已。除了这一类“坏理由”外，还有另一类“坏理由”。因为诉诸经验与具体事例总不能得出“无可怀疑”的结论，有些哲学家（即所谓怀疑论哲学家）因对一切论断均予以怀疑，最后只有诉诸个人自我意识的反思，即用个人反思来保证其理论的基础绝对“无可怀疑”。典型的例子即笛卡儿（理性主义的开山祖）的“我思故我在”，与主观唯心论者巴克莱主教（经验主义的主将）的“存在即被知觉”。对笛卡儿来说，什么命题他都可怀疑其真实性。唯有他反思到他“在怀疑”（也就是他的思想）这一事实他主观上再也无可怀疑。由此

为基点，再得出作为思想的主体“我”的存在，由此而得出其整个哲学体系。类似地，巴克莱对任何事物均表示怀疑，唯有他直接感觉到的印象他不能再怀疑，由此建立其主观唯心论的知识论。显而易见，这样的本体论或知识论的基点是个人的主观意识的反思，必然导致以自我为中心建立整个的哲学体系。在这样的知识论中，甚至“外在世界”和“他人”是否存在以及如何认识都成了问题，连常识都难以说明，又如何谈得上作为科学知识的基础？我认为这样一种知识论唯一说明的问题是由于从它只能得出荒谬的结论，根据归谬法，说明其出发点是错的。也就是说，将知识的基础限于主观感觉经验与逻辑两个方面是不够的。还应该在更广的基础上构造知识论体系以作为科学的依据。但应指出：职业哲学家的著作中虽充满着以上这几类可笑的“坏理由”，这并不等于说哲学工作没有意义。人类思维活动的各个方面如科学、道德、文学艺术等领域除有其本领域的专业概念与原理外，最后都有一些更为根本的概念与原理，它们对其相应的领域的深入发展起着引导与奠基作用。这些带基础性的理论与原理即构成知识论（真）伦理学（善）或美学（美）的哲学内容。当然，一种体系所包含的这些基本理论与原则不应互相矛盾。应可组成协调的理论体系，形成不同的流派。这种理论体系内部的逻辑关系可疏可紧，也可以包含一些实例或用更易为人理解的语言加以说明，但归根到底无法再用什么更根本的道理来逻辑推理予以论证。事实上，追求“无可怀疑”的道理予以论证是难以达到目的的。哲学说到底是一种信仰。当然信仰选择也可有各种标准，讨论这些选择的标准已超出哲学的范围，而属于元哲学（metaphilosophy）。这种元哲学标准往往带有社会学的含义，因为哲学家毕竟是社会中人，他个人所在的社会环境及其位置，无疑将影响他选择哲学的标准。当然，关于这种选择的讨论也是多种多样的。这已超出我们这里讨论的范围。但有一点应指出，许多哲学体系往往是哲学体系与元哲学体系的混合物，比如实用主义就是经验主义知识论与实用主义元哲学的混合物。王浩先生在《超越分析哲学》一书中对现代分析学派哲学家的分析中得出许多与元哲学有关的精辟见解，可供读者参考。

（二）一种统一化知识论体系的提纲

1. 目标

本节目标是建立一种将经验知识与理性知识、逻辑思维与辩证思维相互协调结合的知识论体系。

2. 经验知识与理性知识的对立与统一

经验知识与理性知识是构成知识的两要素。前者通过认识主体的感官以获取关于外在世界的感觉材料，后者通过理性思维活动以加工整理材料使知识具有条理且更能反映事物的本质。

经验主义者（以英国的洛克、巴克莱、休谟等为代表）强调经验是一切关于外在世界的知识的基础这一面，而理性主义者（以欧洲大陆的笛卡儿、斯宾诺莎、莱布尼兹等为代表）则强调理性是认识事物本质的基本途径这一面。事实上这两面都是构成知识的必不可少的要素，各起其不可缺少的重要作用。一切关于外在世界的知识都是这两方面结合起来构成的，二者不可分割。不可仅强调其一面而忽视其另

一面。片面强调其中的一面，只能对人类认识真理起误导作用。

经验知识归根到底是由外在事物作用于认识主体的感官构成的印象为基础形成的。这种印象如看见一片颜色，听到一阵声音等，在哲学上称为感觉材料（sensedata）。人们将这种材料或这一类认知活动简称为感觉（sensation）。但单有感觉还不能构成概念（concept）。概念形成必须经过人类的理性思维活动，也就是经过去粗取精（即抽象〔abstraction〕），去伪存真（即分析〔analysis〕），由此及彼（即演绎推理〔deduction〕），由表及里（即归纳〔induction〕）等思维活动才能形成概念。感性思维与理性思维的关系还并非如此简单。事实上，感觉与概念的关系是相互的。即不但感觉是形成概念的基础，反过来，一般说来，人们在获取感觉印象时，还依赖于认识主体原来已形成的概念作为接收的框架。实际上，除了初生婴儿外，人们很少有完全纯粹的感觉。比如“红色”本身即是一概念（请注意：事实上任何语言只能表示概念），人们在一次视觉活动中所感到的那片具体的颜色才是真正纯粹的感觉材料，它是瞬息即逝不具有可抓住的规定性的。只是因为人们头脑中过去已形成了“红色”、“绿色”、“朱红色”、“浅朱红色”……的概念，才能在这框架中将那片感觉到的具体“红色”印象留存住，成为一感觉材料。概念在认识活动中的这种作用，哲学上称为概念的范畴性（categorization）。康德是首先着重指出概念这一作用的哲学家，这是他在认识论方面的重要贡献。但他将概念的这一作用发挥得过分了，他把人类思维活动中这一作用看成是一先验的本能，忘记了这种作用本身也是人类在长期实践中逐步形成的这一事实。这样的片面性导致了他的哲学体系的唯心论观点。

由此可见，经验知识可分为三个层次：最基本的一层即感觉。第二层即在纯粹感觉之上经过了概念加工的感性认识，哲学上称之为知觉（perception），我们通常谈到的感性认识的内容实际上主要是指知觉。但感性认识还不止于此。因为人类并非总是静观世界。人是为了生存、发展……，在各种实践活动中去认识世界的。事实上，实践（包括科学实践，社会实践等）是一切认知活动的基础。以科学实践为例，它必然是一过程，首先它具有一定的目的性（比如，为了验证某一科学结论或试验某一制成品等），为此，他作出判断并进行推理（理性思维）；然后，他还要进行一些活动（比如为了验证，作出判断以及对试验的对象进行切割、掺合、加温、加压等的改造）；再对结果进行观察（知觉）；然后对观察的结果形成概念与判断，以之与原来判断进行比较，推演，再作出结论（理性思维），如此等等。所以实践是包括目的、感性认识与理性认识、改变外在世界的行动等一连串活动的过程。感性认识总是在实践活动中进行的。如果重复进行类似活动中一种知觉反复出现，人们自觉或不自觉地通过理性思维（归纳或总结）由这些重复出现的知觉形成经验（experience）。这是人类感性认识的最高形式。

上面所述的人类认识活动中，感性认识与理性认识的关系，可以说构成一种螺旋型循环（或称递归）的关系图。它之所以是螺旋型（递归）的，是因为归根到底它有一起点，即感性认识。所以感性认识（知觉）与理性认识（概念）的相互作用关系是一种辩证的关系而不是完全相对的关系。此图如下：

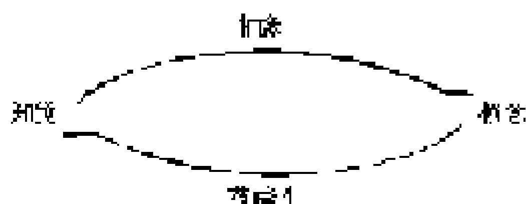


图 1.1　知觉与概念的关系图

[STHZ] 3. 概念（或理论）的发展进化过程

任何概念、理论或思想，无疑都是人类理性思维活动的结果。因此，都具有抽象性。但是，人类理性思维不应单纯归结为神秘的顿悟。不但由于感性认识或理性认识都有其本身认知能力的局限性，而且也由于外在世界是不断发展变化的，表示本质的特征常常是要经过一相当长的过程才能最终为人们所认识。故人类对外在世界的认识在任一具体时刻都可能出错，也可能不全面。所以，人们对外在世界的认识（不论感性认识或理性认识），经常需要修正与改进。因此，认识事物是一过程，很少一次完成。故而感性认识虽是认识外在事物的基础，一次感觉印象并不能代表事物的真实外貌；同样，虽然理性认识是人们认识事物本质的必要的途径，但也不能认为一次分析、抽象、演绎或归纳即能全面深刻地认识外在事物的本质。因此，真理也不是理性思维的一次活动完成的，更不能说认识是由偶然的顿悟或是由于人类具有某种先验的能力所取得。概念或理论虽是抽象的，但也是具体的。它们都有其具体内容。概念的这些具体内容是在人类长期实践中，通过不断修正改进而使之进化发展日益接近真理的。黑格尔首先提出了具体概念的思想，马克思、恩格斯等人使这一认识论具有了唯物论的含义，毛泽东的实践论与矛盾论对唯物认识论进行了很系统的总结。

概念（或理论）的进化是按照以下辩证过程发展的：

首先，在概念形成中作为一相对稳定的正题（thesis）阶段，在此时，可对之进行理性思维加工（即对之进行静态的分析、抽象、演绎、归纳等）。这是理性思维起主要作用的阶段。但随着概念内容的发展，逐渐从中出现了对立的内容而逐步产生分裂，进入下一阶段。

第二，反题（antithesis）阶段。随着对立概念的发展，即可能产生以下几种可能发生的情形：

（Ⅰ）对立的概念主宾地位发生变化我们称之为易位（transposition），即原来概念内容由主变为宾，后来新生的概念内容由宾变为主。二者虽仍维持在同一整体中，但因其地位变化，这概念或理论的含义也随之变化。中国古代《周易》与《老子》，即比较着重讨论这种主宾易位的变化。在一定条件下的“物极必反”往往是这类变化的一种基本模式。

（Ⅱ）第二种情形是：随着变化发展，事实说明所出现的新生的对立面不能成立。因此，概念又恢复到原来的正题状态，我们称这种情形为复原（restoration），过分强调这种状态是保守思想的特征。

（Ⅲ）第三种情形是：随着变化发展，证明新生的对立面可完全否定原来概念的内容。这时概念的内容彻底改变。我们称之为革命（revolution）。概念又回到新的正题阶段。在科学发展的过程中，事实上这种革命性变化是很少出现的。不问具体条件强调革命，往往导致形成偏激的概念或理论。

（Ⅳ）第四种情形则是：新旧两方面呈相持状态。此时，一种选择是再等待一段时间看概念的发展，等到一定的时间，如果不大可能出现上面（Ⅰ）—（Ⅲ）的情形，则应找到一合适的时刻，以合适的分寸将对立的方面综合起来成为一协调的整体，此时概念进入到下一阶段。

第三，合题（synthesis）阶段。我国儒家哲学最强调这一阶段，可称为中和（unification）。这就是“致中和，天地位焉，万物育焉”思想的本质所在（希腊哲学家亚里士多德的中道思想也具有很类似的含义）。从我国传统哲学的观点看来，宇宙万物发展正是基于这种对立统一的方式进行的。如阴阳、天地等本来就是对立的统一。事实上，程序概念也是以数据结构与控制结构、静态语义与动态语义等等的对立统一为基础的。直到现在，面向对象程序设计的领域结构的模块化，与面向开发程序设计的进程结构的模块化的对立与统一仍未逃脱这一框架。经过这一阶段的统一之后，概念又回到正题阶段，开始一个新的发展过程。以上这样一个关于概念的辩证发展过程，可以用图 1.2 来表示。

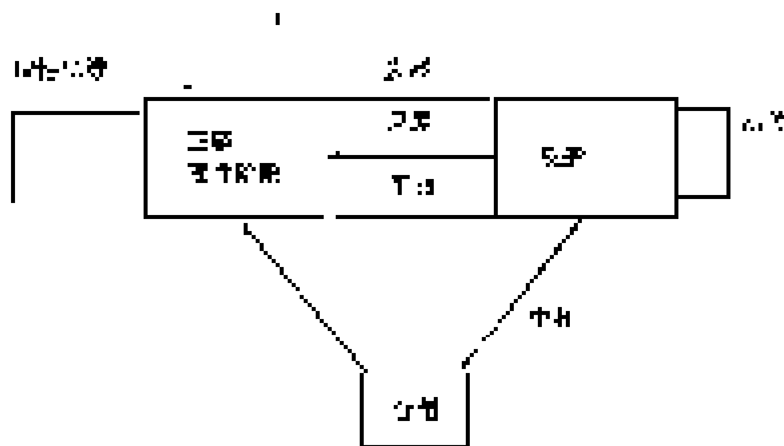


图 1.2　概念的辩证发展过程示意图

由此可见，逻辑思维（即理性思维）与辩证思维并不是互相对立的两种思维方式，所谓辩证思维只是概念的辩证发展过程在思维方法中的反映。而逻辑思维则是适用于概念发展一个阶段（即正题阶段）的思维方法。二者完全可以相辅相成。从图 1.2 易见在辩证思维这一统一的名称下事实上包含了几种不同的思维方式，它们在矛盾发展中各有其适应的情况。不能不问具体发展阶段的具体条件一概予以否定或肯定。

以上是我认为比较适合本书内容的一种知识论的提纲。这样一种知识论对程序

设计具有如下的含义：

（Ⅰ）不论是技术经验方面还是形式化理论方面，不应强调一面或忽视另一面，也不应截然划分，而应使二者在软件开发的各个方面均有机地结合。

（Ⅱ）包括规范与可执行程序在内的程序开发是一个逐步进化的过程，不可能一劳永逸一次性地完成。所以应强调支撑这样一个过程的语言、方法与工具。从形式化理论说，应能作为程序开发全过程的基础，而不仅仅强调其在某一方面的意义。

（Ⅲ）计算机科学与技术的发展也是一个发展进化的过程。虽然这一学科中比较经常出现新技术、新方法、新理论，包括各类的飞跃，但不能认为这一学科要求不断推翻旧有的一切方面，总是处在革命的状态之中。事实上，革命在科学发展中并非经常出现，更非不断革命。以计算机而论，数十年来冯诺曼体系思想虽有很大改进，但它一直仍维持作为现在流行计算机的基础。因此，程序之为程序的基本概念不应轻率地予以否定。在学科从初级阶段发展到高级阶段后，其最根本的本质特征仍不能抹杀。山仍应是山，水仍应是水。以状态转换为基本特征的冯诺曼机上的程序设计还应是这种程序设计。不过，更高阶段的程序设计既应维持原有的本质特征，又应包容了后来发展的各种正确的程序技术与理论的新内容。这中间并非不存在矛盾，问题不在于出现矛盾，而在于从技术与理论的深层上对此进行分析。使之找到新的平衡而予以统一。这并非易事，但我们应该努力去克服困难做到这一点。这就是 XYZ 系统面临的挑战，也是我们引以自豪之处。

在结束哲学讨论之前，还有一个新的问题，即计算机科学与技术有一与其它传统的科学与技术不同之处，它们讨论的对象不是直接关于世界的某一方面事物的知识（如物理、化学、生物、水利工程、矿冶工程等），而是实现表示知识或信息的一种工具或语言，也可以说它是一种关于语言的科学或技术。当然它所表示的知识也存在是否正确的问题；但就这一学科本身而言，它对它所表示的内容是否正确不能负责，它本身也不具备讨论它所表示的内容是否反映客观事实的条件。正如语言本身无法讨论用它表示的命题是否为真理一样，关于计算机及其语言的学科只能讨论它是否正确地、有效地、方便地表示了用户所希望它表示的内容。这是这一特殊学科所面临的往往不易为人们理解的处境。人类虽已使用语言达数千年，但真正形成具有重要经济与社会价值的以语言为对象的科学与技术还是从计算机科学与技术开始的。这个问题既有哲学意义，也有科学意义。下一节我们将讨论这个问题。

三、语言问题

语言是一种表示知识（思想）的符号系统。显然，语言只有通过表示知识（思想）来与客观实在（世界）发生联系，而不是直接与后者发生联系。

语言的理论问题有三个方面：一是讨论由符号如何组成语言中的合式公式，这是符号间的关系问题，称为语法学（syntax）问题。二是讨论语言中的合式成分如何表示知识（思想），这是讨论语言中合式公式与知识（思想）的关系问题，称为语义学（semantics）问题。三是讨论语言的使用者对语言的一些实用性要求，比如

使用效率问题，这是关系语言的使用者与语言的关系问题，称为语用学（pragmatics）问题。这三方面合在一块统称为符号学（semiotics）。因为语言已存在数千年，绝大部分是自然形成的，如汉语、英语等，这叫自然语言。自然语言都不是形式化的。这类语言当然也各有其符号学的研究，也在一定程度上已找到以上三方面的一些规律。对这方面的研究，自不在我们本书讨论的范围之内。我们感兴趣的是形式化语言方面的问题。事实上，从历史上看，从希腊哲学家研究形式逻辑开始，即已有这方面的萌芽。比如，当时即已有用符号代表某些逻辑成份，希望以某种可机械地执行符号变换来代表某种意义的演绎推理。此外，在欧几里德几何研究中也体现了比较完整的形式化方法。不过，真正系统地研究形式化方法应该说还是在20世纪初，从Frege、罗素等研究符号逻辑或数理逻辑开始的。而形式化语言超出数学或逻辑学的范围进入表示日常生活中的问题，则是计算机出现以后才开始发生的情况。不过，常见程序语言中形式化程度并不很高，即令是逻辑语言Prolog中也包含了许多非形式化的成分。事实上，绝对彻底形式化的实际可用的程序语言目前尚不存在，也很可能不会出现。我们认为实际可用的可在常见的计算机上有效运行的形式化语言中，很可能时序逻辑语言XYZ/E是功能最强的。显然，关于XYZ/E的研究必然涉及到上述符号学三个方向的讨论。这里产生一个问题，即作为一个实际应用的形式语言，XYZ/E如何与外在世界联系？显然，任何一程序语言如不能与外在世界联系将是没有意义的。但是，这里所谓“外在世界”，实际上是指程序使用者关于程序所处理问题所在客观世界的“信息”，而并非这世界本身。程序与外在世界联系的窗口，事实上是一些由软件或硬件表示的信息，如软件中由某些变量或公式（为特殊函数）表示“数据”或“事实”，或硬件的某些部件表示的特殊“键盘”、“指示灯”或“时钟”等。至于通过这些信息所表示的“知识”是否符合外在世界的实际，多大程度上可信，在计算机软硬件内是无法验证或测量的。只能留交给使用者自己在计算机软硬件以外根据这些信息去进行判断。事实上，任何语言系统都是这样处理问题的，不过在计算机这一工程系统中这一事实表现得特别明显而已。这一事实说明，在计算机语言或任何其它语言领域，实际上不直接涉及知识论问题，因为它处理的是符号系统与知识信息的关系（这种关系是一种关于“表示”（representation）的关系），而不是信息知识与外在世界的关系（这关系是知识论所讨论的关系）。像计算机或语言学这些领域的科学技术的研究应称为信息科学技术的研究，它们与传统的以物理、化学、生物学等为代表的自然科学以及建立在自然科学基础上的工程技术是完全性质不同的两类科学技术。即信息科学所研究的对象不是外在客观世界本身，而是表示这个外在世界的信息（知识）。由于计算机科学技术的出现与发展，这一新领域即信息科学技术的特征日益为人们所注意。一个十分有趣的问题是：逻辑与数学应属于那一种科学技术？有相当可信的理由将它们划入信息科学而不是自然科学的范围。如果事实果然如此，则过去许多涉及逻辑与数学的知识论问题的争论则应重新考虑。因为，在信息科学技术方面，根本不应直接涉及知识论问题。也许正是由于过去将数学划入自然科学，所以一些知识论问题总是极难解决。也许逻辑与数学既非自然科学又非信息科学而处于二者之间兼具两方的某些特征的一类科学，因此用这两类科学的标准都只部分适合而不完全适合。这很可能是

数理哲学的困难所在。

对于信息科学技术来说，由于知识（思想）与信息处于语言符号系统与客观实在世界之间，这就发生一些新的问题。这处于中间的知识及其信息这一部分理应受客观世界与符号系统双方的制约，即它不但应反映客观世界的情况，而且应与语言的符号系统的结构及功能相互协调。比如说，如果一语言的符号系统不能表示状态转换，则这语言所表示的知识部分也不能有与状态转换相应的信息。如 Prolog 这语言不包含表示不确定性的机制，显然它所表示的知识也都是确定性的，并不能因此就说明客观世界的规则都只能是确定性的。一语言系统只选择了与它自身结构相适应的那部分“知识世界”予以“表示”，这并不意味着它所表示的就是整个客观外在世界。这样的事实只是说明该语言只与它们能表示的世界有关而已。也可以说，一语言所表示的“知识世界”，只能多少是这语言的一个影子。这一情况与传统的以自然科学为基础的知识论的讨论的“知识”很不相同，知识既非完全由于客观实在在知识主体方面的反映，也非完全由于康德式的主体方面思维框架作用于客观世界所提供的材料的结果。对于信息科学技术而言，居于在语言符号系统与客观实在世界之间的知识或信息世界，有它居于中间地位的固有特征。我认为维特根斯坦的《Tractatus》（在一定解释下）首次较明确地提出了这个问题。他在这本书内，一方面构造了一个他认为理想的逻辑语言的符号系统，一方面设想了一个与之对应的知识世界。至于在这知识世界后面的客观实在，用他的话来说是“不可说的”。根据王浩教授在《超越分析哲学》一书中的解释，维特根斯坦的“理想逻辑语言”基于以下的假设：（Ⅰ）将“概念”与“事物”混为一谈，统称为“对象”；（Ⅱ）有穷化原则；（Ⅲ）原子化原则；（Ⅳ）外延化原则。由此得出的“命题”的全体构成语言。命题可分解成初等命题的真值函数，这些初等命题的真值函数表示的是初等事实（即原子事实或事件的状态）是否成立。与之对应的是一具有类似结构的对象世界。用维氏自己的话说：

1.1　世界（应为对象世界）是事实（facts），而不是事物（things）的全体。

2.01　一初等事实是一些对象的组合。

2.011　事物是由初等事实所组成。

2.203　每一名字（name）指称（denote）一对象，对象是其指称。

2.81　如果表示的形式是逻辑形式，则所构成的图象（picture）即称为逻辑图象。

3.　事实的逻辑图象即思想（thought）。

4.　一思想即一具有含义的命题。

4.001　一语言即全体命题组成。

4.21　最简单的一类命题，也就是初等命题，肯定初等事实的存在。

5.　一命题即由初等命题所成的真值函数。

由此可见，维氏所谈“对象”及“对象世界”或“事实”是与“事物”不同的。“事物”是指我们所说的“客观实在世界”，而“对象”及“对象世界”或“事实”是他的逻辑语言所表示的内容，也就是说，这些概念是指我们所谓“知识世界”中可被维氏逻辑语言所表示的那一部分，即该逻辑语言的“语义世界”。这

世界是属于“知识”或“思想”范围之内的。不过，根据我们前面介绍，一语言的“语义世界”与该语言仍有区别，因后者只是一符号系统。从“4”及“4.001”看维氏未将二者加以区别。当然，由此即可能引出“知识”与“客观实在世界”之间的关系问题，也就是知识论问题。如果将语言理解为符号系统，则对于语言学而言只存在符号系统与所表示的知识之间的关系问题，也就是语义学所研究的问题，而不存在语言所表示的知识与客观实在世界的关系（即知识论）方面的问题。因此知识论问题即从这一学科领域取消了。

按照维氏的看法，这样一“理想的”逻辑语言即构成了我们可说清楚（或表示清楚）的东西极限，在此之外，即什么也不应该说。

“只有能说清楚的话才能说，凡不能说的则应报之以沉默。”

这句话往往从哲学上被解释为神秘主义或不可知论。但如果从形式语言的角度说，事实上这句话可以理解为“语言所能表示的知识与客观实在世界的关系（即知识论所研究的问题）不是有关语言的符号学所能讨论的问题”。在这样的解释之下，也就不存在什么神秘主义或不可知论了。事实上这正是近三十年来在计算机语言研究中发生的情况。维氏上面这一段话，在第一个较严格陈述的程序语言 ALGOL 60 正式文本的开端，被引作该文本的指导思想。从 ALGOL 60 的角度引用这段话，从来未引申出神秘主义或不可知论。也就是说，维氏在《Tractatus》中所陈述的思想，事实上是该书出版三十余年后才问世的计算机语言所要求的原则。《Tractatus》提出的逻辑语言所应满足的（Ⅰ）—（Ⅳ）四条假设，从计算机语言看均十分合理。维氏心目中所设想的可说清楚的思想实际上恰恰就是计算机语言所能表示清楚的内容。因此，从这一角度说，维氏可以说是计算机这一技术的先知。这也说明为什么在欧洲不少计算机科学家如维也纳的 H. Semanek、特拉德海姆的 A. Solvberg 等均认为维氏哲学是计算机科学的哲学。从计算机技术看，维氏思想不存在什么神秘，他只是将许多知识论的讨论排除在语言的符号学之外而已。这也是很合理的。事实上，计算机语言历来即如此处理其语言问题。维氏《Tractatus》中的思想在过去数十年中之所以引起那么多争论，主要是因为他本人及注意他的思想的人都是从哲学的角度来看待他的思想，这里自然就引起大量的问题。比如他的“有穷化原则”，显然使许多哲学的讨论难以适应；它甚至使作为数学基础的集合论也成了问题。所以，他的语言与哲学思想难以为哲学界及数学界所接受则是很自然的事了。以致他本人在后期也放弃了《Tractatus》的思想。这些事实只能说明《Tractatus》的创造性思想摆错了位置。这并不能埋没这一天才思想在技术方面的预见性。这里只有一点值得注意的是维氏在《Tractatus》提出其思想时，总是将逻辑、数学与语言学摆在同一范围之内。这使我联想到前面提到的信息科学与自然科学的区别。的确，历史上关于知识论的讨论，一个居于核心的困难问题即如何解释逻辑及数学真理。前面已指出，从来关于知识论的讨论，都是以物理、化学、生物等自然科学为模型的。在哲学史上关于知识论问题的讨论在解释自然科学及建立在其上的工程技术问题时很少遇到困难。这些事实不能不使我想到如果将逻辑与数学也像计算机科学与技术及语言学一样划在信息科学的范围内，即将它们看成不是直接表示客观实在，而是通过表示知识（信息）来与客观世界联系，也许过去一贯认为十分困难的数理哲学问题

即将不成为问题。我认为这是一个值得哲学家注意的重要问题，它涉及到改变知识论的面貌。与这个问题有一点联系的是下面三个问题：

（一）逻辑主义问题。罗素是数理哲学方面有名的逻辑主义的先驱。他在《数学原理》中，通过在逻辑的基础上构造集合论从而将数学归结为逻辑。后来，他本人的工作虽因遇到悖论而引起麻烦，但这个问题一直是数学基础方面引人注意的问题。在某种意义下，通过构造时序逻辑语言，我们在时序逻辑基础上可以将各种程序语言的特征均以统一的框架表示出来，因此可以说将程序语言从而将计算机归结为逻辑。从这个意义上说，XYZ系统也可以说体现了一种逻辑主义。不过，我们的工作比罗素的研究范围要小得多，因为程序语言不需要表示全部集合论，所以，野心要小一些。XYZ系统只是将计算机程序的语义归结为逻辑，可以说是计算机世界的逻辑主义。但在XYZ出现之前，用逻辑统一程序语言也遇到了一些新问题。因为在直言式逻辑中表示状态转换从来就是一个困难问题，如克林（Kleene）即未能在一阶逻辑内表示与有穷自动机等价的正则事件。我们是因为借时序算子之便才做到这一点。

（二）统一化问题。我们所进行的统一程序语言的工作很容易使人联想到卡尔纳普（Carnap）的统一化科学的工作。这两项研究从外貌上看确有其相似处。但事实上很不一样。他那研究涉及所有各种学科，范围极广，内容极复杂，而我们仅涉及计算机科学技术这一领域，自然范围狭窄得多；更重要的区别是，他从哲学的角度以相信逻辑与经验知识是构成知识的两种因素为出发点的，从一开始即很少注意到每一学科内部即有不统一的情况（如相对论与量子力学的统一问题长期未得解决），更何况各不同学科之间的不统一性就更明显了。所以，他从一开始即以由顶向下的精神贯彻其信仰。从某种意义上说，那条道路是很难走通的。而我们的出发点一开始即由底向上的。不只是因为我们相信程序语言需要或应该统一，而是从具体各种语言出发发现在时序逻辑上可能实现统一才进行此项研究的。虽然研究过程中也遇到了许多困难，但因找到了本质上的统一性，所以，这些具体困难都找到了合适的途径予以解决。因此，XYZ系统统一各种程序设计方式的工作与卡尔纳普所倡导的统一科学计划是完全不同的两类工作。

（三）验证问题。XYZ系统与维也纳学派均讨论验证问题。事实上，这同一名词代表了具有完全不同含义的两种工作。这里涉及到前面讨论过的符号学的问题。XYZ系统是信息科学的一项研究。程序语言中虽亦有表示与外在世界联系的成分，但在语言系统范围内部我们无法讨论语言所表示的内容与外在世界的关系，故XYZ系统中所讨论的验证，事实上不是讨论计算机语言程序表示的语义与外在世界的关系，而是讨论系统内两种子语言语义之间的关系。即一种是可执行子语言XYZ/EE表示的程序，设为P；另一种是用规范子语言XYZ/AE表示的程序性质或规范，设为Q；所谓“验证”，是根据严格的逻辑推理，检查从所给程序P是否能推导出所要求的性质Q，也就是证明：

$$P \models Q$$

成立。这里不涉及到一程序与外在世界的关系。而逻辑实证主义者谈论验证时，事实上是根据通常知识论中经验主义者的思想，即以通常物理或化学实验为模型，对

于语言所表示的判断要求，通过实验观察等方式检查该判断是否正确。这里涉及到语言表示的内容与外在世界的关系。由此可见他们所谓验证与 XYZ 系统中的验证完全是两回事。但有一点值得注意，维也纳学派哲学家往往同时是主张语言分析的思想家，语言符号学正是他们提出的。他们自应知道，从形式语言学角度看，语言学内部是不能讨论知识论问题的。而他们在哲学讨论中所谈的验证，显然不是形式语言学范围内所讨论的“验证”。如果说，“验证”表示“检验一个论断是否正确”，则通常以物理化学为模型的知识论所谈的“验证”是指“检验一个命题是否正确地表示了所指的客观实际情况”，而计算机科学中所谓的“验证”是指“检验一道可执行的程序是否正确地表示了它的相应的规范所指的语义”。至于它的相应的规范所指的语义与客观世界的关系，这是计算机科学范围内无法讨论的问题。

诗　词　选

登山

久历崎岖意未消，
风光无限聚峰娇。
山行愿化云梯石，
早送来人上九霄。

京都岚山

暮云凝聚锁中流，
雨外斜阳一角楼。
醉眼飘飖忘客意，
瀛洲终不是神州。

登山

久历崎岖意未消，风光无限聚峰娇。
山行愿化云梯石，早送来人上九霄。

京都岚山

暮云凝聚锁中流，雨外斜阳一角楼。
醉眼飘飖忘客意，瀛洲终不是神州。

杭州

晚林如墨数灯孤
一抹遥山淡欲无
十载前缘忘不得
桂花秋雨别西湖

阿尔加夫海边

夕阳西坠水悠悠
人到西欧地尽头
莫把西天当彼岸
美洲西更有神州

杭州

晚林如墨数灯孤，一抹遥山淡欲无。
十载前缘忘不得，桂花秋雨别西湖。

阿尔加夫海边

夕阳西坠水悠悠，人到西欧地尽头。
莫把西天当彼岸，美洲西更有神州。

初春登北海揽翠轩

古木重城曙色开
蓟门天净见楼台
槛前春水破冰出
原上白云吹雨来
旧野断园新瓦换
高枝先夺早春回
东风欲醒神州梦
近似无声远若雷

初春登北海揽翠轩

古木重城曙色开，蓟门天净见楼台。槛前春水破冰出，原上白云吹雨来。
旧野断因新瓦换，高枝先夺早春回。东风欲醒神州梦，近似无声远若雷。

闻國家自然科學奖讯

卅载崎岖步履辛，
烂柯缘力敝车薪。
浮生易老心难老，
世态趋新我未新。
自古痴缘多似梦，
已迟花讯岂成春。
疲驴直近荒山暮，
明月清風自可人。

闻国家自然科学奖讯

卅载崎岖步履辛，烂柯缘力敝车薪。浮生易老心难老，世态趋新我未新。
自古痴缘多似梦，已迟花讯岂成春。疲驴直近荒山暮，明月清风自可人。

金陵怀古

四十流年去不回，
秦淮歌舞梦成灰。
寻根我独临荒渡[1]，
扫叶谁堪托古台[2]。
元佑是非终两败[3]，
乌衣纨绔竟重来[4]。
江山易使行人老，
岂止沧桑动客哀。

金陵怀古

四十流年去不回，秦淮歌舞梦成灰。寻根我独临荒渡，扫叶谁堪托古台。
元佑是非终两败，乌衣纨绔竟重来。江山易使行人老，岂止沧桑动客哀。

夏夜宿香山 調寄虞美人

小樓高樹風淒咽
雨過松林月
此時情似那年秋
一榻青燈花下說紅樓
而今人似孤鴻渺
我亦他鄉老
喚回十載少年心
夜半蛙聲猶作故園音

夏夜宿香山　调寄虞美人

小楼高树风凄咽，雨过松林月。此时情似那年秋，一榻青灯花下说红楼。
而今人似孤鸿渺，我亦他乡老。唤回十载少年心，夜半蛙声犹作故园音。

桃李芬芳

唐先生不仅在科研工作中做出了杰出贡献，还培养了大量人才。数以百计的学者和工程师在不同时期先后得到唐先生的指导。他们当中有不少已经成为学术界和工业界的佼佼者。下面仅列出部分人员名单，主要包括唐先生直接指导过的研究生、博士后以及与唐先生合作过的有关科研人员。

何天敏　徐书润　顾毓清　倪惜珍　杨　均　范植华　龚振和　冯玉琳
叶大兴　郑茂松　周咸青　顾元祥　李　新　龚　淳　李　东　钱振宇
印卫平　周巍宗　林惠民　汪健白　谢汉东　谢洪亮　蔡晓莉　董　春
王　新　郑　毅　杨　红　丁　征　彭　敏　龚　洁　何家伟　何　锫
刘激扬　刘　彤　缪　旭　庞民治　王　杉　肖　宇　张　钱　黎仁蔚
郭瑞阳　徐祗祥　邵　中　张文辉　贾燕斌　穆　斌　王铁军　严海林
赵　琛　金　伟　沈武威　唐若鹰　闵关山　王　宵　张　健　王飓安
杨　莉　杜文亮　史剑平　谢育涛　高永祥　陈秋胜　张小格　柳军飞
沈　伟　唐小平　闫　安　郑小立　卢　铁　骆华俊　石民勇　李广元
郭　亮　王春江　周子毅　舒　明　郑建丹　张广泉　赵海云　朱雪阳
刘春明　徐雨波　吴荣杰