

第0章 绪 论

语言是交换信息的媒介,是人们相互通信的工具,也是信息的载体,人们借助这些载体来记忆、加工信息,“语言是思维的工具”。

自然语言是人类生产、生活与自然斗争中发展起来的语言。是人类群体为群体内交换信息的一种约定。汉语、英语、法语、俄语、西班牙语……都是自然语言。我们知道,一种自然语言主要是一组发声规则、语音语调的声音语言,以及对应的一套基于符号文字的书面语言。然而,哑语、盲文又扩大了自然语言内涵的范畴,至今难于准确定义什么是自然语言。为与本书讨论的高级程序设计语言比较,我们先看看自然语言的性质。

0.1 语言的一般性质

• **媒体性** 人们传递信息常常要用到自身器官所能表达的一切手段,声音是最重要的传递手段,但姿态、动作、情绪(形、象)却不可少。为了记录转瞬即逝的信息,辅助信息记忆(存储),人们发明了与声音对应的文字符号语言。符号文字实质上是图形的抽象,图形又是自然景象的抽象。声、图、文、象成了表达自然语言的基本手段,它们相互补充。随着人类历史的进展,加上媒体本身固有的性质,它们既相似又各不同。在每一个发展方向上又派生出若干分支,例如,符号语言派生出数学符号语言以表达逻辑和推理思维,派生出音乐符号的五线谱便于对乐器的操纵。声音语言总是当前用于交换的语言,它有时间性,并且是驱动自然语言发展的最活跃分支。文字语言相反,它有很强的稳定性。古文、时文留下不同历史时期的自然语言版本。文字语言也往往是语言规范的依据。

声、图、象随着计算机技术的发展,它将成为表达人类形象思维的有力工具。符号文字、语言则成为逻辑思维的表达工具。不同媒体语言都是自然语言的组成,相互补充而又无法完全取代。

• **规范性** 语言是通信工具,通信者之间若无事先约定的规则,信息传递必然失真。任何语言都有一组约定的基本元素(音素、图素、符号集、象元)以及它们合法组合的规则集,即语法;以及这些基本元素在合法的组合中的意义,即语义;还有这些有意义的组合在不同应用环境下所表达的真实含义,即语用。一种语言,有一致的基本元素、语法、语义、语用的规则本来是不言而喻的。然而,由于使用语言的是人。人们往往寻求自己意志更好的表达方式:向趋简、扩大概括范围、随意抒发,加上不同行业、阶层、地域的差别,使得方言、行话、俚语、黑话层出不穷。简化字,缩略语、外来语比比皆是。在交通通信不发达的古代,只能做到统一文字规范,方言甚至发展到全然不懂的程度,例如中国的南北方言。规范化和方言化这一对矛盾始终推动着语言的发展,即使在通信高度发展规范化要求很高的今天,行话、俚语也不能完全消除。

• **演进性** 语言必然随社会信息量的总增长而发展。随着社会文化的发展语言也向纵深发展。它往往从专业的专门需要而扩展到全社会。例如,符号语言的数学分枝,当社会上数学普及及时就成为通用语言的一部分,如算术符号,等号,代数公式,积分号等;商业词条随着商业文化普及,也成为大众语言的一部分,如“投入”、“产出”、“透支”。作为语言表达基本概念部分的字典日益增厚,大英牛津词典词条已达130余万。作为语言规则的另一部分,也从古英语变为现代英语,古汉语变为现代汉语。

自然语言的演进随社会经济、文化发展有其自身的规律，沿规范化——方言化缓慢但不停滞地交替进展，并深深受到政治影响。秦始皇的“书同文”，清代的“康熙字典”，以及我国解放后的“汉字简化方案”、民国的“国语”和以后的“普通话”，都是规范化的重大步骤。而每次规范化都纳入方言化中合理的进展。如果经济文化未发展到某种程度，政治影响不足，即使是非常好的演进文字，其效果也不理想。例如，本世纪初的世界语由于没有经济文化托，至今敌不过英语的普及。以致于人们产生错觉：自然语言是难于进化的。其实是人们运用语言的惯性。

演进性与惯性这一对矛盾在自然语言中也确实存在。人们往往习惯于从小养成的某种表达方式而拒绝“更好”的表达方式。方言地区的普通话一听就可以听出来，“日本英语”很难听就更不用说了。但惯性的影响只在一代、半代人最多十几代人的时间，从语言发展的历史看演进性是明显的，而且愈来愈快。

• **抽象性** 客观事物总是普遍联系相当复杂的，且处于永恒的运动变化之中。人们表达的信息总是抽取某个需要的侧面。以其主要特征概括该事物。从表达的简易性来看人们总是力图抽象，愈是抽象它表达的事物外延愈大，如“人”，它包括“白人”，“黑人”，“亚人”，“非洲人”……甚至包括“泥人”，“铁人”。所以，在语言中主干词往往是比较抽象的，形容词、副词使之具体化。代词一般更抽象。汉语中的虚词（之、乎、也、者、矣、焉、哉）更是无所不能。

信息表达中利用简单的抽象构造复杂的抽象是扩大表达能力的基本手段。随着人类文明的发展，从组合抽象上升到概括抽象是非常明显的。愈是文明古国的语言，抽象概念愈多。例如：爱斯基摩人说“雪”有十六种具体的雪，唯独抽象不出“雪”字。但抽象语言有时也会失于贴切和准确。我国翻译家常常为找不到贴切的形容词和副词翻译英文而头疼。

• **冗余与多义性** 随着信息质和量的扩大，借鉴表达和重复表达同时存在。一词多义、一意多词在自然语言中比比皆是，语言情况更严重，汉语早就有“九Li十八Chen”的说法。这带来释义的困难。必须根据上下文的语义才能完成释义。尽管如此，误解、误用比比皆是。

绝对的冗余随着规范化的进展会逐渐消除。但有些冗余已深入到语言本身已无法消除。例如，汉语中“感觉”、“平平稳稳”，无法用“感到”、“觉得”、“平稳”替代。有些冗余甚至成为法定语法，例如，英语中的时态与时间副词的重用：

I finished the work yesterday.

finished若不用过去时即为错句。双重保险冗余是必要的。再如，同一语义可用主动句语法也可以用被动句语法。这种冗余带来表达的方便。

多义源于借喻和类比，最为典型的是成语，成语一般只取其抽象涵义，如“一针见血”在绝大多数情况下不是讨论“针如何扎出血”之类的问题，而任何不接触实质问题都可用它评判。另一个来源是一词多词性，如英语SET一词有33个解。汉语“画画”“扇扇”……等等，有时造成难以分辨的混乱，如：

中国队大胜。 中国队大败。

中国队大胜美国队。 中国队大败美国队。

正是由于冗余和多义性，自然语言在短期内不可能作为程序设计语言。因为机器还不能像人那样聪明。

0.2 程序设计语言的一般性质

程序设计语言是表达软件的工具，任何软件都是计算机系统行为的预先规划。这个规划就是以某种语言编写的程序系统。计算机系统理解了程序并按程序的规定完成一步步的动作。所以，程序设计语言也是人——机通信工具。从用户的观点，程序设计语言是一台虚拟的机器。语言的表达能力和效率，就是机器的能力与效率。学会了某种语言（当然包括系统命令）就是学

会了使用机器，甚至可以不懂硬件工作原理。从用户观点，程序设计语言也是人——人交换信息(如何用程序模型“问题世界”的运动)的工具。这点对当今软件开发的工程方式尤其重要。

我们从表达、通信、使用三个方面描绘了程序设计语言功能实质。作为语言它也有和自然语言相似之处(如下述)，但最大的不同是：

- **程序设计语言是人工语言** 这里所说“人工”指一个语言从定义——实现——到流行不象自然语言那样要经过数百年到数千年，而是几年到几十年。因此，规范化——方言化，演进性——惯性矛盾加骤。新语言、新版本层出不穷。1975年美国国防部曾统计当时美国软件所用语言及方言多达1500种，当今开发一个新语言由于有完善的工具，不用一人年，因此，其数量不可胜数。

- **主要通信对象是机器** 这首先要程序语言定义、实现要精确，决不能有含混和遗漏。所以，到目前为止程序设计语言都是有严格定义的符号语言，即形式语言。计算机科学家积极研究形式语法，形式语义，以求得到准确的解释。其次，机器由于使用目的不同，制造厂家不同，容量、速度、指令及格式差异是客观存在的。而用某种程序语言表达的程序要能在各型机上运行，它就不能有依赖机器的特征。然而，哪怕有一个重要特征被抽象略去，要充分发挥硬件设备的能力也无异于空谈。所以早期语言没有一个是完全独立于机器的，甚至包括(Pascal、C)。同样，语言也不应有依赖运行支持(即OS、运行支持例程和支持库)的特征。

- **用于表达软件** 如同数理符号系统对于数学、五线谱对于音乐，程序设计语言应方便于表达软件，能反映当前软件开发技术，能为用户提供所需要的特征。但要弄清楚的是，软件开发技术和程序语言的表达没有直接关系。即技术高超的程序员能用过时的老语言编制反映最新技术的程序。但往往以牺牲可读性、易维护性、安全性为代价。好的程序设计语言使用方便，符合软件工程诸原则，从而有助于提高软件的开发效率。

除以上三点与自然语言不同之外程序设计语言也有媒体性，特别是当今多媒体技术蓬勃地发展，声控程序，图形编程，动画设计陆续走上实用。但当今整个的程序设计语言学仍然建立在符号学的基础上，即以正文字符作为程序设计语言媒体。其他形式媒体只是被处理的对象，故本书讨论的程序设计语言仍限于正文符号语言。

程序设计语言的人工性决定了演进很快，从1954年FORTRAN研制至今已有七个正式版本。Turbo-Pascal, Borland-C, MSC 更快，几乎1年半就有一新版本。程序设计语言演进这么快说明了两个问题：一是软件技术发展很快，二是人们还没有找到相对满意的程序设计语言。估计在今后一段相当长的时间也不会稳定。这也是本课程开设的目的。制约语言演进的因素是与老版本开发出的软件兼容。把老版本改造成新版本的投资量并不比重新开发少很多(这是软件工业的特点)。不利用又造成极大浪费，所以老语言规范改版比新语言困难、慢得多。

语言惯性在程序设计语言中依然存在。所以第一门入门语言对软件工作者影响极大。语言惯性也制约着新型性能良好的语言推广。

程序设计语言更能说明抽象性。因为说到底机器只会取出一条指令执行一条指令，改变内存某些单元的数据。人们对机器一次运行的结果作出的解释就是程序的语义。机器语言只有操作码(存、取、比较、四则算术)和地址码的定义。人们制造了变量、数组、赋值、控制函数、过程、块等语义概念，经翻译成为操作码、地址码实现这些概念，从而人们可以编制过程式程序。如果再把数据结构和操作进一步封装起来，就可以实现对象式程序……总之，愈向前发展愈是提供更接近问题域的概念编程，愈远离原始的地址码和操作码语义。基本的技术是高层抽象。其示意图如图0-1。

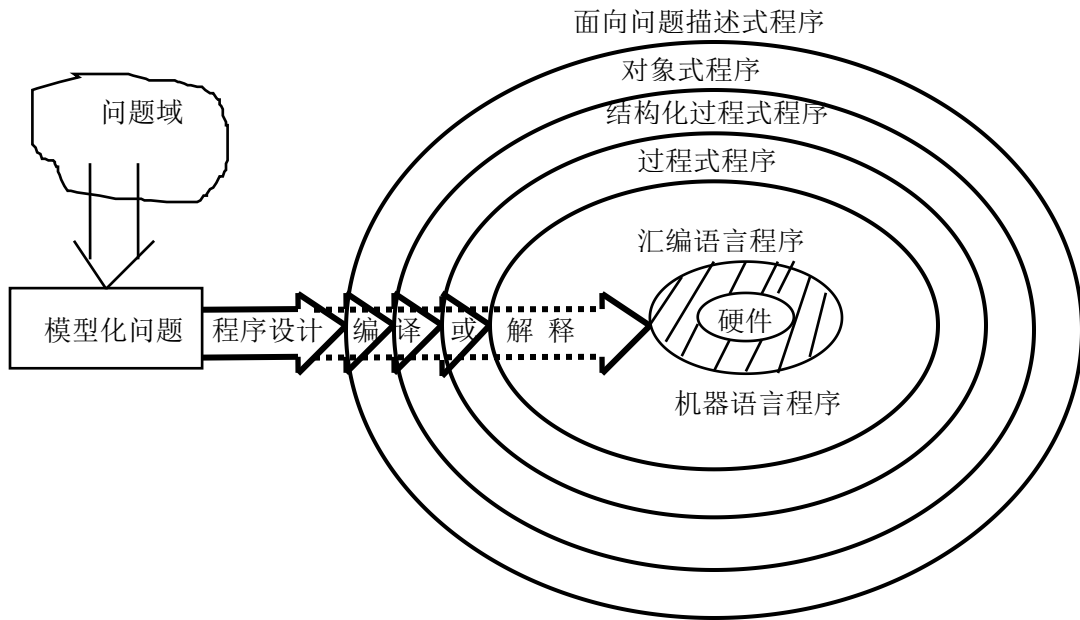


图0-1 程序设计语言的抽象层次

一个问题域中的问题，经需求分析建立解题模型，如果是函数、逻辑、对象式模型则可直接以程序中函数，谓词推理规则、对象的表达机制编制程序而不涉及指令及地址细节。如果是过程式编程，除了程序对象及其相互关系之外，还必须设计对象实现的过程，但也不涉及指令和地址细节。汇编程序则需由程序员把计算模型转换成面向机器的指令和地址。最后这些不同层次语言编制的程序都经编译(或解释)器变为可执行目标码。面向机器→面向过程→面向对象→面向问题这是当今程序设计语言发展的趋势。

程序设计语言的多义与冗余也同样存在：特殊符号从最早的高级程序设计语言即有多义，如FORTRAN中的括号()，可声明数组大小、维度；数组元素引用；条件语句中的条件；函数和过程的参数表；隐循环；格式语句的格式说明；表达式中优先算符。现代语言把多义扩充到标识符，如Ada, C++的函数重载，即同一函数名代表不同的函数。这个趋势随着编译分辨能力增强会更明显，其目的是方便。因为一个系统由若干程序员分别开发常用名词术语都差不多。只许唯一的名字管理实在太麻烦。

冗余也一样，我们都知道结构化程序设计理论指出只要有三种特征：顺序、条件和循环即可编出任何程序，迭代只要while-do语句就够了。然而，标准的结构化语言Pascal循环有三种(while_do, for_do, do_until)。此外，从控制执行流程说来只要条件表达式加上goto句，任何控制特征均能表达。然而，所有保留goto语句的语言都定义了许多控制特征，即使是转语句FORTRAN中就有四种(转、计算转、标号转、算术条件转)。这些冗余无非是为了方便和编程效率。

0.3 为什么要研究程序设计语言

自1945年有现代计算机以来，程序设计语言研究始终是最活跃的领域。到目前为止，还没有趋于稳定。传统的程序设计语言已暴露出不适应新技术的需求，然而又离不了它们。新语言虽层出不穷，又一时难于占据统治地位。其中的原因、趋势很值得研究。具体说来：

- 它是计算机软硬件技术的窗口 软、硬件技术发展最终必然要反映到人——机界面的语言上。例如，硬件高速、存储容量增大，图形界面、可视计算就有可能，相应图符语言得到发展。软件重用技术的发展，可重用库标准化，才有第4代语言的诞生。这些新发展都会改变传

统的使用计算机方式和程序设计语言的概念和定义。所以，只要软硬件技术在发展，新版本还会不断出现。

• **它是人们研究计算表达的形式** 计算机科学的本质是研究如何把问题世界的对象及其运动映射为程序对象及其交互通信。即从计算的角度观察、模型客观世界，然后在模型理论的基础上建立表达(即语言)、实施计算。程序设计语言学不涉及模型理论本身。但是是表达成果的手段。例如模糊逻辑要求模糊程序设计语言，函数式模型要求表达高阶函数的函数式语言。所以，随着人们对计算本质的认识加深，新范型，新语种总会出现。

• **它和计算机理论研究联系最紧密** 计算机科学就是从集合论(1895)、符号逻辑(1910)、不完全理论(1931)发展起来的。至30年代Post系统、递归函数论、可计算性理论奠定了计算机科学基石。从研究停机问题的Post系统和形式语言理论基础上，发展出完善的形式语法，编译理论，编译器的编译器至可扩充语法的EL/1系统。从递归函数论的一个分支，建立了操作和指称语义学，直至类型理论。此外，当今并发、分布、协调理论，CSP、CCST均与并发分布式程序设计语言相关。有关计算科学理论基础和形式语义学发展见图0-2，图0-3。因此，学习程序设计语言的理论和实践为进一步学习计算机理论打下基础。

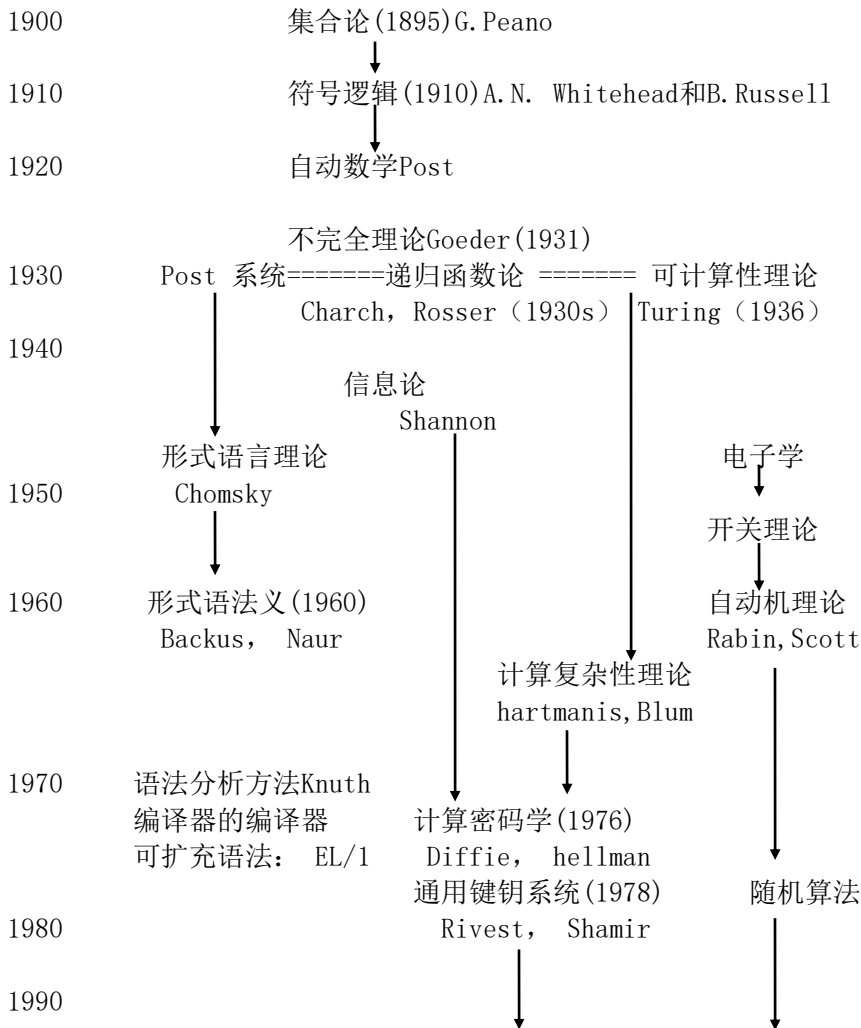


图0-2 计算机科学理论基础

• **研究程序设计语言还有利于提高软件开发人员素质** 对程序设计语言清晰的理解能扬长避短地使用某种语言；善于捕捉潜在的软件缺陷；会选择适宜的语种或版本。

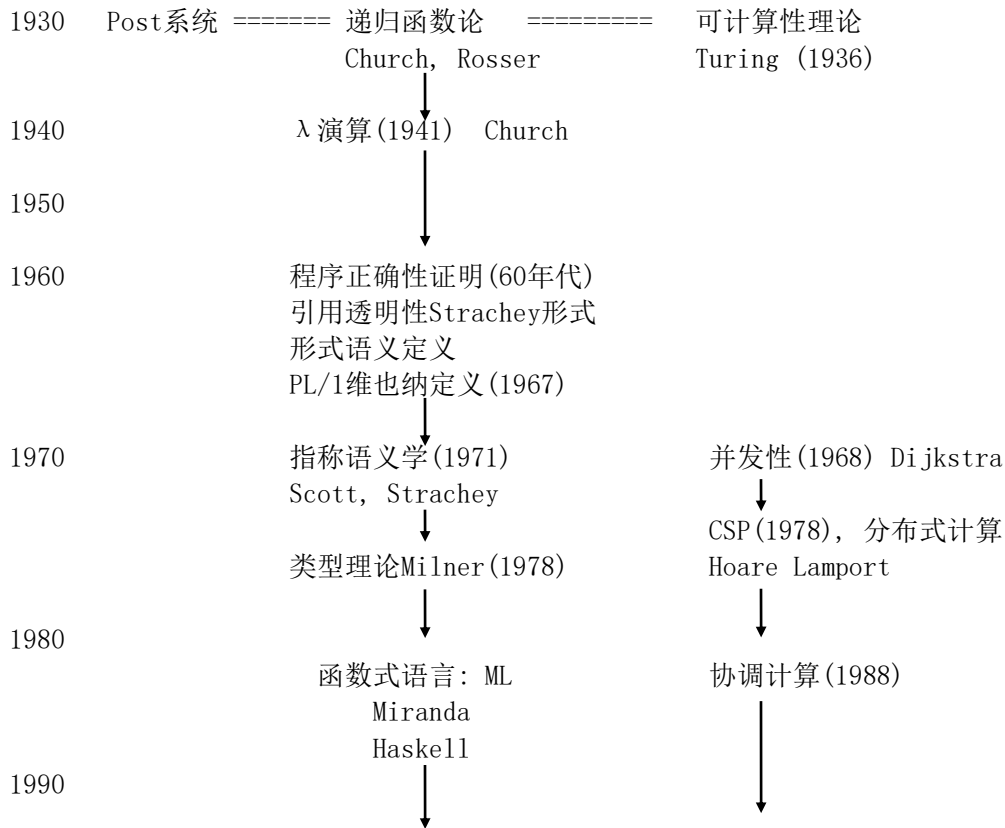


图0-3 形式语义学理论的发展

• **有利于开发专用语言或界面语言** 专用语言可大幅度提高软件生产率，使用户使用更方便。随着语言技术向更抽象方向发展，计算机技术向各行业渗透，开发近于各行业术语和工作规程的非常高级的语言是必然趋势。这样各行业人员经过极少的计算机培训即可用计算机解决自己的业务问题。

• **有利于新领域语言的发展** 当前处于热门的规格说明语言、并发程序设计语言、4GL、多媒体语言(5GL)远不及顺序过程式语言成熟。对于不太成熟语言的使用者，必须有较好的程序设计语言的理论功底，一方面能减少不必要的错误，另一方面也能促进新语言的发展。

• **有利于通用语言的标准化规范化** 对于广大的软件工作人员说来，如果不是专业的程序设计语言工作者，语言标准化就不是份内的工作。但熟知语言的理论和实践对理解、推行的标准，不无端制造方言都有好处。而“必要的方言”又是下次标准化的候选扩充版本。

0.4 程序设计语言定义与处理器

一个程序设计语言的物理存在就是该“语言的引用(参考)手册”(LRM)，俗称“语言定义文本”，术语是该“语言的规格说明”。它是一个文档，引用手册规定了该语言的语法和它的语义，以及用该语言元素写的程序如何组织(即如何提供一个完整的程序)。

一个程序设计语言的实际存在是该语言程序的处理器。它是一个软件，机器配备了这个软件，机器就具备处理该种语言程序的能力。处理器将独立于机器的源程序翻译为可在该种机型上运行的目标代码程序。因此，按执行的模式处理器有编译型的和解释型的：

解释器读入一段相对完整源代码(多数情况下是一句)翻译为目标代码后,立即解释执行。交互式环境的语言即采用这种方式。一般说来,解释器能够迅速给出正确结果,但程序效率较差。这不等于是只适合于交互式的小语言。庞大的Ada程序设计语言的第一个实现就是解释型的。因为,当时急于验证语言定义是否完整正确。

编译器读入一个独立的编译单元(程序的逻辑单元,如主程序、函数、子程序。或若干个逻辑单元组成的编译单元,如文件、模块、包)翻译为可执行的目标代码单元,经过连接必要的库支持例程成为可执行内存映象,加载到内存中运行,由于是按单元翻译,可以经过上下文分析作若干次优化,目标代码质量高。对于运行量大要求高效的应用则采用编译型语言,此外,还有:

翻译器 从一种语言翻译为另一种语言的软件统称翻译器。编译器、解释器将源代码翻译为目标码是它的两种特例。将用户定义的界面语言翻译为某种高级语言(如Pascal, C, Ada), C语言的预处理器就是翻译器的实例。预处理器将语法定义以外的特征翻译为符合该语法定义的表示,是扩充程序语言的手段。

一般说来,程序设计语言的定义是第一性的,不依赖于处理器的实现。但验证语言的功能和性能却要在某个具体的机器上运行该语言的处理器。这是目前验证语言设计仅有办法。Ada的ACVC测试程序用例近一万个,通过不了则不承认你是Ada的编译器。正因为如此,在定义程序设计语言的机制时,它一方面要根据应用域的需要定义各种语法特征(提供相对独立语义概念的语法规定),同时也要顾及到该种特征实现的可能性。例如,重载标识符,必须有分辨信息,故一般变量不宜重载,函数名可以。

早期的语言引用手册以元语言(metalanguage)表达每一个语法特征,并以自然语言对各特征作语义说明或限定。早期元语言的概念指“描述语言的语言”,而不是当今“更加本原”的高抽象“元”的概念。元语言一般包括:

- 字符集、词条表由它们提供标识符、运算符、关键字、空格、限制符、各种数字量、字符串常量等语法概念。
- 注释 注释符号之内(或以后)的符号串是为人们阅读,处理器要略去的。
- 语法结构定义定义以上符号组合的合法结构。它给出语句、表达式、程序单元、块(如FORTRAN的循环域)、作用域、嵌套与结合等语法概念。早期的语言语句定义是最主要的语法结构。

元语言 是严格形式定义的,例如,COBOL的ADD语句定义为:

$$\text{ADD} \left\{ \begin{array}{l} \text{标识符} \\ \text{数值} \end{array} \right\} \left\{ \begin{array}{l} , \text{ 标识符} \\ , \text{ 数值} \end{array} \right\} \cdots \text{TO } \text{标识符} [\text{ROUNDED}] [, \text{标识符} [\text{ROUNDED}]] \cdots \\ \text{[; [ON]SIZE ERROR 语句]}$$

其中: { } 括着的列,其各项是可任取一的。

[] 括着的项是可选的。

... 表示重复项。

字词下面横线表示基(相当于关键字),有的文本以黑体给出,若选此项必写此词。

元语言描述是面向程序员使用的。其实它的词条表,就是语法中的词法标记(Lexical token)。重复项,替换项在巴库斯(BNF)范式中以更加数学化的符号“|”代替。早期元语言的形式定义最大的问题是,它只定义了语法结构的形式,没有规定这些结构是如何产生的,致使在说明中作更多的附加说明。尽管如此,语言编译的实现者还经常因理解的不同产生微妙差异。而验证编译器是否正确目前尚无简易办法,若无大量测试,只能投入市场实际使用以求反馈信息。这种代价是巨大的。

计算机理论家、语言学家认为如果语法和语义能以更加数学化地描述,则不必经过大量测试在纸上即可验明其正确性,发现它的问题,即时修改。经过60-70年代的努力,形式语言学得到发展。形式语法尤其完善,形式语义尚未完全成熟;尽管如此,程序设计语言的实现者仍力图借助这些形式理论,解决实现语言时的许多含混问题。

目前，程序设计语言引用手册（为使用者所用）已不回避BNF、EBNF等面向实现者的描述方法。语义方面仍然是自然语言和示例解释，是非形式的。形式语义仅当需要时由语言实现者、研究者另外作出。但是，一些基于经验的语言（如C），至今拒绝EBNF描述语法，仍用元语言形式，因为严格的数学化会取消它们的许多灵活性。

0.5 本书的目的与组织

本书是为有一定编程经验的系统程序员、应用开发者编写的，作为深入了解高级程序设计语言，选用设计语言的教材。书中也有一些实现编译器或解释器的内容，但不是设计实现语言处理器的指导教材。

通过程序设计语言发展和更新，读者可以了解软件发展中与计算和表达有关的主要理论和实践问题。也许这比学会设计出小型程序设计语言处理器更重要。本书力图复盖美“ACM/IEEE-CS计算1991教程”有关程序设计语言部分。

本书的具体目标是：

- 对各种类型高级程序设计语言有清晰的理解。它们的基本原理、概念和程序设计范型。
- 了解常用高级程序设计语言的优缺点。以便有选择，有限制地使用或更改扩充。
- 了解程序设计语言发展中的问题与趋向，以及程序设计语言在软件技术中的地位。
- 掌握程序设计语言各主要成分设计中的关键问题。学会分析、选择、调合、折衷，以便设计新特征。
- 掌握设计程序设计语言的主要步骤和表示法和理论分析、表达的基本技能。

本书共分四大部分：第零章至第二章是有关程序设计语言的一般论述并简述形式语法。第三章至第九章是程序设计语言的基本元素和特征机制。第十章至第十五章是各种程序设计范型。第十六章至第十七章为形式语义，第十八章是附录。

第 1 章 历史的回顾与程序设计语言分类

程序设计语言发展迅速。到目前还丝毫没有规范到统一语言的迹象，我们要研究它，最好办法是从它的历史发展开始，展开它的全貌。从发展中了解为什么老的不行要有新的，这其中的困难和技术正是我们研究的内容。分类使我们简化问题只研究一类中的一个、两个代表就可以。

1.1 程序设计语言简史

1945 年第一台现代计算机 ENIAC 问世，它用真空管做计算，一下子比当时最快的电动机机械计算机快 300 倍(每秒 300 次乘法)。存储器非常小，计算指令(即程序)由外部插座和开关馈入。还不能称之为完全自动计算。1946 冯·诺依曼在一篇论文中建议: (1) 计算机应采用二进制。(2) 计算机的指令和数据都可以放在存储器内。这就是奠定现代计算机的著名的冯·诺依曼原理:

CPU 逐条从存储器中取出指令执行，按指令取出存储的数据经运算后送回。

数据和指令(存储地址码、操作码)都统一按二进制编码输入。数据值的改变是重新赋值，即强行改变数据存储槽的内容，所以说它是命令式的(imperative)。

第一台按冯·诺依曼原理制成的通用电动计算机是 1951 年美国兰德公司的 UNIVAC-1。人们就开始了机器语言的程序设计: 指定数据区编制一条条指令。由于任何人也无法记住并自如地编排二进制码(只有 1 和 0 的数字串)，则用 8、16 进制数写程序，输入后是二进制的。程序的外部表示和内部的存在一开始就是分离的。

单调的数字极易出错，人们不堪其苦，想出了将操作码改作助记的字符，这就是汇编语言，汇编语言使编程方便得多。但汇编码编的程序必须要通过汇编程序翻译为机器码才能运行。尽管汇编码程序和机器码程序基本一一对应，但汇编语言出现说明两件事，一是开始了源代码——自动翻译器——目标代码的使用方式，一是计算机语言开始向宜人方向的进程。

1.1.1 50 年代高级语言出现

1954 年 Backus 根据 1951 年 Rutishauser 提出的用编译程序实现高级语言的思想，研究出第一个脱离机器的高级语言 FORTRAN I。其编译程序用 18 个人一年完成(用汇编语言写)。到 1957 年的 FORTRAN II，它就比较完善了。它有变量、表达式、赋值、调用、输入/输出等概念; 有条件比较、顺序、选择、循环控制概念; 有满足科技计算的整、实数、复数和数组，以及为保证运算精度的双精度等数据类型。表达式采用代数模型。

FORTRAN 的出现使当时科技计算为主的软件生产提高了一个数量级，奠定了高级语言的地位。FORTRAN 也成为计算机语言界的英语式世界语。

1958 年欧洲计算机科学家的一个组织 GAMM 和美国计算机协会 ACM 的专家在苏黎士会晤起草了一个“国际代数语言 IAL”的报告，随后这个委员会研制了 ALGOL 58 得到广泛支持和响应。1960 年欧美科学家再度在巴黎会晤对 ALGOL 58 进行了补充，这就是众所周知的 ALGOL 60。1962 年罗马会议上对 ALGOL 60 再次修订并发表了对“算法语言 ALGOL 60 修订的报告”。由于该报告对 ALGOL 60 定义采用相对严格的形式语法。ALGOL 语言为广大计算机工作者接受，特别在欧洲。在美国，IBM 公司当时经营世界计算机总额 75% 的销售量，一心要推行 FORTRAN，不支持 ALOGL，以致 ALGOL 60 始终没有大发展起来。尽管如此，ALGOL 60 还是在程序设计语言发展史上是一个重要的里程碑。

1959 年为了开发在商用事务处理方面的程序设计语言，美国各厂商和机构组成一个委

员会。在美国国防部支持下于 1960 年 4 月发表了数据处理的 COBOL 60 语言。开发者的目标要尽可能英语化,使没有计算机知识的老板们也能看得懂。所以像算术运算符+、*都用英文 ADD、MULTIPLY。COBOL 60 的控制结构比 FORTRAN 还要简单,但数据描述大大扩展了,除了表(相当于数组)还有纪录、文件等概念。COBOL 60 虽然繁琐(即使一个空程序也要写 150 个符号),由于其优异的输入/出功能,报表、分类归并的方便快捷,使它存活并牢固占领商用事务软件市场,直到今天在英语国家的商业领域还有一定的地位。

50 年代计算机应用在科学计算和事务处理方面有了 FORTRAN、COBOL,因而应用得到迅速发展。工程控制方面刚刚起步,仍是汇编语言的市场。1957 年,美国 MIT 科学家 McCarthy 提出 LISP,并把它用于数学定理验证等较为智能性的程序上。但 LISP 在当时只是科学家的语言,没有进入软件市场。

1.1.2 60 年代奠基性研究

60 年代计算机硬件转入集成电路成本大幅度下降。应用普及的障碍是语言及软件。这就促使对编译技术的研究。编译技术的完善表现在大型语言、多种流派语言的出现。

1962 年哈佛大学的 K. Iverson 提出 APL 语言。它是面向数学(矩阵)的语言。它定义了一套古怪的符号,联机使用非常简洁,深得数学家喜爱。它提出动态数据(向量)的概念。

1962 年 AT&T 公司贝尔实验室 R.Griswold 提出正文处理的 SNOBOL,可以处理代数公式、语法、正文、自然语言。以后发展为 SNOBOL 3, SNOBOL 4。80 年代后裔叫 ICON,用于测试。

1963-64 年美国 IBM 公司组织了一个委员会试图研制一个功能齐全的大型语言。希望它兼有 FORTRAN 和 COBOL 的功能,有类似 ALGOL 60 完善的定义及控制结构,名字就叫程序设计语言 PL/I。程序员可控制程序发生异常情况的异常处理、并行处理、中断处理、存储控制等。所以它的外号叫“大型公共汽车”。它是大型通用语言的第一次尝试,提出了许多有益的新概念、新特征。但终因过于复杂、数据类型自动转换太灵活、可靠性差、低效,它没有普及起来。但 IBM 公司直到 80 年代在它的机器上还配备 PL/I。

1967 年为普及程序语言教育,美国达特茅斯学院的 J.G. Kemeny 和 T.E.Kurtz 研制出交互式、解释型语言 BASIC(“初学者通用符号指令码”字头)。由于解释程序小(仅 8K)赶上 70 年代微机大普及, BASIC 取得众所周知的成就。但是它的弱类型、全程量数据、无模块,决定了它只能编制小程序。它是程序员入门的启蒙语言。

LOGO 语言是美国 MIT 的 S. Papert 教授于 1967 年开发的小型语言。其目的是无数学基础的青少年也能学习使用计算机,理解程序设计思想,自己编制过程或程序。但直到 80 年代初微机普及到家庭它才受到重视。LOGO 是交互式语言,用户编程就在终端前定义过程命令,并利用系统提供的命令构成程序。LOGO 的数据是数、字、表。由于它能方便地处理符号表,可以利用人工智能成果开发情报检索、演绎推理、自然语言会话小程序。青少年可设计各种智能游戏。屏幕上的海龟使青少年直观地构造各种图形。递归程序的表达能力使青少年可受到自动程序良好训练。1979 年 MIT LOGO 小组推出 Apple LOGO,及 TI LOGO(德州仪器公司 TI 99/4A 机)。这两个版本最为普及。LOGO 是青少年入门的启蒙语言。

LOGO 的近于自然语言的命令及海龟、键盘、程序、图形并用的使用风格,对以后的命令式语言、用户界面语言有一定的影响。

1967 年挪威计算机科学家 O.J. Dahl 等人研制出通用模拟语言 SIMULA67。它以 ALGOL 60 为基础,为分层模拟离散事件提出了类(Class)的概念。类将数据和其上的操作集为一体,定义出类似类型的样板。以实例进入运算。这是抽象数据类型及对象的先声。

60 年代软件发展史上出了所谓的“软件危机”。事情是由 1962 年美国金星探测卫星“水

手二号”发射失败引起的。经多方测试在“水手一号”发射不出错的程序在“水手二号”出了问题。软件无法通过测试证明它是正确的。于是,许多计算机科学家转入对程序正确性证明的研究。这时,著名的荷兰科学家 E. Dijkstra 提出的“goto 语句是有害的”著名论断,引起了一场大争论。从程序结构角度而言,滥用 goto 语句会使程序无法分析、难于测试、不易修改。这时也提出了全程变量带来的数据耦合效应、函数调用的副作用、类型隐含声明和自动转换所带来的难于控制的潜伏不安全因素等等过程语言中的一些致命性弱点。60 年代对大型系统软件的需求大为增长(如编制较完善操作系统、大型军用系统),要求使用高级语言以解决生产率之需,加上高级语言使用以来积累的经验,加深了人们对软件本质、程序语言的理解。人们积极研制反映新理论的语言。

1964 年,ALGOL 工作组成员 N. Wirth 改进了 ALGO 60,提出结构化的语言 ALGOL W。由于它结构简洁、完美。ALGOL W 成为软件教程中示例语言。1968 年,他带着 ALGOL W 参加新一代 ALGOL 的研究委员会,即开发 ALGOL 68 的工作组。

ALGOL 68 追求的目标也是能在多个领域使用的大型通用语言。1965 年以 Wijngaarden 为首的一批科学家开始研究新 ALGOL。强调了语言设计中冗余性(少作隐含约定)、抽象性(数据抽象与控制抽象)、正交性(一个语言机制只在一处定义并与其它机制无关)。强化了类型定义和显式转换;有并发、异常处理功能;保留 goto 允许有限制的函数边界效应:过程可以作为参数传递;用户可定义较复杂数据结构、定义运算符。语法定义是半英语半形式化的。语言作成可扩充式,也就是说,有一个相对完备的小语言核心,可以不断增加新特征以增强语言表达能力。表达式采用利于快速编译和提高目标码效率的逆波兰表示法。ALGOL 68 集中了当时语言和软件技术之大成。但因学究气太重,一般程序员难于掌握。强调语言简单的人持有不同看法。但文本草案在 Wijngaarden 坚持下通过了。为此,Dijkstra 等人发表了“少数人声明”。N.Wirth 带着竞争失败的 ALGOL W 回去研究以后著称于世的 Pascal。

Pascal 的研制者一开始就本着“简单、有效、可靠”的原则设计语言。1971 年 Pascal 正式问世后取得了巨大的成功。它只限于顺序程序设计。是结构化程序设计教育示范语言。

Pascal 有完全结构化的控制结构。为了方便,除三种最基本的控制结构(顺序、if-then=else、while-do)外,又扩充了二种(do-until、for-do)。程序模块有子程序(过程和函数),分程序,可任意嵌套,因而有全程量、局部量、作用域与可见性概念。保留 goto 语句但不推荐使用。

Pascal 的数据类型大大丰富了,有整、实、字符、布尔等纯量类型:有数组、记录、变体记录、串等结构类型;增加了集合、枚举、指针类型。为用户描述复杂的数据结构乃至动态数据提供了方便。所有进入程序的数据都要显式声明、显式类型转换。并加强了编译时刻类型检查、函数的显式的值参和变量参数定义便于限制边界效应。在人们为摆脱软件危机而对结构化程序设计寄予极大希望的时代,Pascal 得到很快的普及。它也是对以后程序语言有较大影响的里程碑式的语言。

1.1.3 70 年代完善的软件工程工具

硬件继续降价,功能、可靠性反而进一步提高。人们对软件的要求,无论是规模、功能、还是开发效率都大为提高了。仅管 Pascal 得到普遍好评,但它只能描述顺序的小程序,功能太弱。在大型、并发、实时程序设计中无能为力。程序越大越要求高的抽象力、安全性、积少成多的模块拼合功能。为了对付日益加剧的新意义上的软件危机。70 年代语言继续发展,在总结 PL/1 和 ALGOL 68 失败的基础上,研制大型功能齐全的语言又一次掀起高潮。

70 年代是微机大发展的时代。设计精巧的小型过程语言藉微机普及得到发展。软件市场 FORTRAN、COBOL、汇编的三分天下开始缓慢地退却。结构化 FORTRAN、COBOL 力

图在新的竞争中保全自己的地位，专用语言丛生。一旦证实它的普遍性，它就变为通用语言。C 就是在这种情况下成长起来的优秀语言。

硬件的完善使得过去难以实施的组合爆炸算法得以缓解。人工智能的专家系统进入实用。LISP 发展为 INTEL LISP 和 MAC LISP 两大分支，其他智能语言陆续推出。特别是 Backus 在 1978 年发表“程序设计能从冯·诺依曼风格中解放出来吗？”一文。非过程式语言、高抽象模式语言大量涌现。

70 年代继 60 年代的形式语言语法研究，形式语义取得重大成果。最先是 IBM 维也纳实验室集合欧洲著名的计算机科学家于 1972 年写出 PL/1 的操作语义。该语义用维也纳定义语言 VDL 表达，长达 1500 页，终因抽象层次太低，而此时(1971 年)牛津大学 D.Scott 和 C.Strachey 提出了更加数学化的指称语义学，维也纳实验室转而研究 PL/1 的指称语义描述。1973-1978 年 D. Bjorner 和 C. Jones 开发了维也纳开发方法 VDM。所用语言是 Meta IV。虽然巨大投资(3.5 亿)未见可见效益，IBM 终止了维也纳实验室的语义学研究方向，但 VDM 方法及指称语言学对计算机语言发展影响是深远的。1973 年 C.Hoare 和 Wirth 用指称语义写 Pascal 语义发现了 Pascal 的设计上的许多问题竟做不下去。VDM 以后还用于多种语言如 CHILL, Ada, 指导编译器的开发。

1971-72 年，DEC 公司和卡内基·梅隆大学的 Wulf 合作，开发了 PDP(以后是 VAX)上的系统程序设计语言 Bliss。它是无类型的结构化语言，没有 goto 语句。有异常处理和汇编接口，面向表达式。直到 80 年代末 DEC 公司还用它作系统设计。

1972 年，AT&T 公司贝尔实验室 Ritchie 开发了 C 语言。C 语言的原型是 1969 年 Richard 开发的系统程序设计语言 BCPL。K.Thompson 将 BCPL 改造成 B 语言，用于重写 UNIX 多用户操作系统。在 PDP-11 机的 UNIX 第五版时用的是将 B 改造后的 C。C 扩充了类型(B 是无类型的)。1973 年 UNIX 第五版 90%左右的源程序是用 C 写的。它使 UNIX 成为世界上第一个易于移植的操作系统。UNIX 以后发展成为良好的程序设计环境，反过来又促进了 C 的普及。

C 语言是个小语言，追求程序简洁，编译运行效率高。是一个表达能力很强的、顺序的、结构化程序设计语言。它给程序员较大的自由度，下层数据转换灵活。程序正确性完全由程序员负责。上层是结构化的控制结构，有类似 Pascal 的数据类型。它的分别编译机制使它可构成大程序。输入/出依赖 UNIX，使语言简短。语言学家极力反对的 goto 语句、无控制指针、函数边界效应、类型灵活转换、全程量这些不安全的根源 C 全部具备。在某种意义上 C 得益于灵活的指针、函数副作用和数据类型灵活的解释。易读性又不好，偏偏程序员都喜爱它。因为它简洁，近于硬件，代码高效，并有大量环境工具支持。C 程序写起来又短，调试起来又快。微机上的各种移植版本的 C 语言，对 C 成为通用的程序设计语言起到了推波助澜的作用。C 语言正以席卷系统程序设计领域的势头发展，并向应用领域扩展。以后的发展把与它同期出现的 Pascal 远远抛在后面，成为系统软件的主导语言。

1972 年法国 Marseille 大学的 P.Roussel 研制出非过程的 Prolog 语言。Prolog 有着完全崭新的程序设计风格，它只需要程序员声明“事实”“规则”。事实和规则都以一阶谓词的形式表示。Prolog 规则的执行是靠该系统内部的推理机，而推理机按一定的次序执行。在这个意义上它又有点过程性。以回溯查找匹配，Prolog 的数据结构类似 Pascal 的记录或 LISP 的表。它是以子句为基础的小语言，最初被解释执行，编译 Prolog 是很久以后的事，由于 Prolog 是以逻辑推理作为模型的语言，它可以直接映射客观世界事物间逻辑关系。在人工智能研究中得到了广泛的应用，80 年代日本声称研制的五代机以 Prolog 作为主导语言并研制 Prolog 机。

70 年代，在传统语言中出现了以下有代表性的语言：

为了开发大型可维护程序，施乐公司 1972-74 年由 Geschke 领导研制了 Mesa 语言。Mesa

是强类型结构化语言，有模块(若干子程序集合)概念和抽象数据类型。支持并发程序设计，由监控器协调各模块执行。有分别编译，异常处理机制。保留 `goto` 语句，也可以抑制类型检查。Mesa 可配置语言编译后的各模块。

1974 年 MIT 的 Liskov 和 Zilles 提出 CLU 语言，它突出的是数据抽象。数据抽象是 70 年代类型强化和抽象技术的重要成果，它允许用户定义抽象的数据类型。这些类型的定义和它的实现可显式地分开。定义描述了语义，实现对于使用该数据的用户是无关紧要的，因而，利于修改。增强模块性和独立性，从而易于保证程序正确。数据抽象可定义更远离机器而近于人类的数据概念。如堆栈就可定义为抽象数据类型。人们可通过压入数据、弹出数据的操作对栈体进行操作。其外在行为就是后进先出的数据栈，而栈体可由数组、或链表、或记录任一种数据结构实现。

CLU 的抽象数据类型称之为簇(Cluster)。由构造算子(Constructor 按簇的样板建立运算对象(实例))。CLU 无 `goto`，无全量程概念。用户可定义新的迭代(通过迭代算子 Iterator)。

数据抽象在 1975 年卡内基·梅隆大学 Wulf 和 Shaw 开发的 ALPHARD 语言中是数据模型 FORM。ALPHARD 的特点是支持程序的验证。程序设计和验证同时进行。

另一个支持程序验证的语言是加州大学 Poperk 和加拿大的 Horning 于 1976 年到 77 年开发的 EUCLID。为了易于验证，无 `goto` 语句，指针仅限于集合类型，类型兼容有严格的定义，函数调用绝无边界效应。编译自动生成验证用的断言。EUCLID 以后发展成数据流语言。

在并发程序方面，1975 年丹麦学者 B.Hanson 开发了并发 Pascal。它没有追求大而全，只是将 Pascal 向并发方面作了扩充，希望用 Pascal 写操作系统。有抽象数据类型的类(Class)机制。控制方面提出进程类和管程类的概念。通过 `init` 语句激活类实例，`cycle` 语句使进程无限循环地运行。通过管程(管理资源的模块)实现进程通讯。有较强的静态类型检查，可查出静态“死锁”。

令人不解的是正当人们对进程、管理概念充分评价时，B.Hanson 本人放弃了这些概念。1981 年发表了小型系统程序设计语言 Edison，仅用并发语句控制并发进程的执行。Hanson 极力推崇语言的简单性，所以 Edison 比并发 Pascal 小得多，普通(不大的)微机都可以运行。但其表达能力比 C 差多了。没有达到并发 Pascal 那样的影响。

Pascal 在结构化程序设计方面是一个示范性语言，在推行结构化程序设计教学上发挥了卓越的作用，但在工程实践上暴露出设计上的许多缺点。Pascal 除了无独立模块和分别编译机制不能编大程序而外，原来它的强类型是有漏洞的。类型等价似乎是按名等价，实现是按结构等价。最后的结论它是“伪强类型”。数组定长对处理字符串很不方便，布尔表达式求值定义不严，I/O 规定太死，难于写出灵活的输入/输出。声明顺序过严，无静态变量概念(局部量一旦所在局部块执行完毕就消失)都给程序设计带来不便。从小而灵活方面，它又不及 C，没有位(bit)级操作，指针操作限制过死。于是 Pascal 的设计者 1975 年又开始开发 Modula 语言，1977 年正式发布为 Modula-2。

Modula-2 除了改进 Pascal 的上述弱点而外，最重要的是有模块结构。可分别编译的模块是用户程序的资源。系统资源也以模块形式出现。模块封装了数据和操作(过程)，模块定义和模块实现显式分开。程序员在定义模块中通过移入，移出子句控制程序资源(类型、变量、过程、子模块)的使用。

Modula-2 增加了同步进程机制以支持并发程序设计，有有限的低级设施直接和系统打交道。取消 `goto` 语句、增加 `case` 语句中 `otherwise` 机制，封装的模块可作抽象数据类型设计。它是用于系统设计的强类型语言。西欧的计算机科学家对 Modula-2 是欢迎的，但它不巧与美国开发的 Ada 非常近似，与 Ada 竞争处于非常不利的地位。尽管它的 9000 句编译器具有 Ada20 万句编译器 80% 的功能，也没有取得 Pascal 那样的成就。

70 年代中期美国软件的最大用户美国国防部(美国软件市场约 2/3 经费直接或间接与它相关)深感软件费用激增并开始研究原因。研究表明,在硬件成本降低和可靠性提高的同时,软件费用不仅相对数,绝对数也在增加。美国军用的大量大型、实时、嵌入式系统软件开发方法落后、可靠性差。语言众多(常用 400-500 种,加上派生方言多达 1500 种)造成不可移植、难于维护。为摆脱这种新的软件危机下定决心搞统一的军用通用语言。从 1975 年成立高级语言工作组开始投资五亿美元,前后八年研制出 Ada 程序设计语言。Ada 是在国际范围内投标设计的,法国的一家软件公司中标,J.Ichbian 成为 Ada 发明人。多达 1500 名第一流软件专家参与了开发或评审。它反映了 70 年代末软件技术、软件工程水平。为了提高软件生产率和改善软件可移植性,提出开发语言的同时开发支持该语言的可移植环境(APSE)。

Ada 是强类型结构化语言。封装的程序包是程序资源构件。用户只能看到程序包规格说明中显式定义的数据(包括抽象数据类型)和操作。数据结构和操作(过程或函数)的实现在程序包体中完成。封装支持模块性和可维护性。规格说明和体的分离支持早期开发(可延迟决策)。分别编译机制可组成复杂的大型软件。

Ada 有并发、异常机制。可定义精确的数据(如浮点数小数点后任意多的位数)。有将数据对象、类型、过程参数化的类属机制。有为目标机写目标程序(机器语言的或汇编语言地)的低级设施,可对字节、字位操作。Ada 的私有类型支持数据隐藏,程序包可实现数据抽象。标识符和运算符重载概念,既方便程序员又使程序好读且安全。强调大型系统可读性胜于可写性,Ada 程序自成清晰的文档。

Ada 语言的开发过程完全按软件工程方式进行。严格禁止方言。美国国防部有一个严格管理 Ada 及其环境的机构 AJPO(Ada 联合规划办公室)负责 Ada 的确认、修改、维护、培训。从业界转向软件工程方法开发软件的意义上,Ada 也称之为里程碑式语言。

由于 Ada 过多强调安全性和易读性,Ada 编译程序要做较多的静态检查,因而体积庞大(约 20 万句、512KB 的微机装下了微机 Ada 编译就剩不下工作空间了)。程序代码较长,虽不像 COBOL 繁琐但要比 C 语言程序长 60%。运行效率,特别是嵌入式实时控制应用中,通过交叉编译得到的目标机代码一时还难满足要求。环境工具发展缓慢,因为除军方外民间公司更乐于开发对所有语言通用的计算机辅助软件工程环境(CASE)。自 80 年代第一个语言版本,83 年修改定型至今,Ada 没有达到投资者预想的成就。目前已看到 Ada83 只反映 80 年代初期的软件工程技术。随着软件工程本身向集成化、可重用、面向对象方向发展,Ada 已有一些不适应了。但美国军方还在全力支持,1995 年 Ada 完成面向对象改造推出了 Ada-95。增加了标签类型、类宽类型、抽象类型;放宽了访问类型;使一个静态强类型语言可以支持 OO 的动态束定。这种改造非常痛苦,使 Ada-95 语法规则增至 277 条,成为最庞大臃肿的语言。耐人寻味的是 Ada-95 很快被 ANSI 和 ISO 接受,它成为世界上第一个有法定标准的面向对象的语言。

70 年代末到 80 年代初值得一提的还有 FORTH 语言。FORTH 是典型的中级语言。它是汇编语言指令码向用户自定义方向的发展。也就是说,用户可以面向一个堆栈机器模型定义操作命令——字(word)。最低层的字是指令码,逐层向上,上层字由下层组成。因此,FORTH 系统有良好的继承性。系统提供核心字、解释器字、编译字和设备字。对于简单的计算解释器字直接执行命令(字),复杂计算可将字定义编译成目标码存入堆栈供以后执行。有汇编字集合以使用户直接使用机器,要求字集合操作数值计算,引用字集合使用户可以引用系统和用以前已定义的字。FORTH 程序员首先查看系统中字的字典,以它们组合成新字,进而构成程序。FORTH 程序的逆波兰表示法便于解释和编译,这对长期从事汇编编程的程序员并不生疏。FORTH 把具体的机器抽象为堆栈机,既可以使程序员直接操纵机器又不涉及具体机器指令码、操作码、存储安排。而且良好的继承性使程序越编越短,在最终用户层一两

个命令就完成了程序设计。它大受控制领域、要求单片、单板计算机(例如仪表工业)领域的程序员喜爱。

FORTH 是 C.Moore 一个人开发的语言, 他于 1968 年在 IBM 1130 机器上实现第一个 FORTH。他说他的语言是第四代的(Fourth-Generation)。由于 1130 只允许五个字符的名字才叫 FORTH。1973 年成立 FORTH 公司并把它投入航天工程应用, 发展了通用商务 FORTH 系统。此后世界各地开始重视 FORTH。1976 年成立欧洲 FORTH 用户小组(EFUG), 1978 年成立 FORTH 标准化国际组织, 80 年发布 FORTH-79 标准文本。与此同时美国 FORTH 爱好者小组 FIG)也制定了标准 fig FORTH。各国天文行业, 仪表行业纷纷以其为行业用计算机语言。FORTH 并不好读, 也不宜编大程序。但在它自己的领域简单好用、扩充方便、编译迅速。与传统语言追求的目标大相径庭。给人耳目一新。

1.1.4 80 年代的面向对象发展

Ada 的大、功能齐全、开发耗资可以说是程序设计语言之最。但它还没有普及就有些落伍了。可能今后不再有人再投入巨资去开发大型过程语言。

80 年代继续向软件危机开战, 但软件工程以陈旧技术难于作出庞杂的软件工具。为了改善这种情况, 人们乞灵于面向对象技术。程序设计语言纷纷面向对象靠拢。正如上一个 10 年程序设计语言结构化一样。这是主要特点。

80 年代的第二个特点是“用户友好”的所谓第四代语言的出现。

80 年代的第三个特点是各种技术相互渗透各种更高级非过程性语言出现。

1972 年美国施乐公司保罗·奥特研究中心的 A.Kay 领导的软件概念小组为方便不同用户处理各种信息在小机器上搞了一个 Dynabook 计划。它以全部的“对象”概念建立自己的系统。1980 年 Smalltalk-80 作为正式的发布版本。

Smalltalk 语言是该系统的软件。专用的硬件机、Smalltalk 环境、用户界面、面向对象程序设计风格构成了整个系统。

“对象”是有自己的数据和操作的实体, 各对象相对封闭。程序设计就是建立应用系统中的对象群, 并使之相互发消息。接到消息的对象就在自己封闭的存储中响应执行操作。操作的结果是改变这组对象的状态、完成计算(发出输出消息, 对象响应后完成输出)。

为了使各自封闭的对象数据和操作不致多次重复定义。Smalltalk 有类和继承的概念。类如同传统语言中的类型, 只有类的实例进入实际运算(叫实例对象)。类对象中的数据和操作可为它的子类继承, 它自己的数据和操作也继承自超类。于是一个 Smalltalk 系统就必须有支持它的一个类库, 它象一棵大树, 所有的行之有效类都在这棵大树的某个位置上。用户只要选取其中某些类稍加修改变成自己问题所需要的类(子类), 定出通讯协议, 让它们的实例相互通讯完成计算。无论是系统对象和用户定义的对象都按不同的抽象层次放在统一的类库中。例如, 向编译对象发消息并传送用户对象, 该用户对象(程序)就被编译了。Smalltalk 中只有不同抽象层次的对象, 小到一个整数、大到一个系统都叫对象, 且别无其它计算单元。面向对象程序设计的概念因此而出。

Smalltalk 因为它天然的封装性体现了模块性和数据隐藏, 利于维护修改。它的继承性实质上是软件的重用。这对困惑于大程序难于管理的软件工程学无疑是一条绝好出路。

Smalltalk 类对象概念来自 SIMULA 67, 响应消息的方法表达式求值类似 LISP 的归约。它本身是在小机器上开发的小系统。庞大的类库占去了很大的空间, 难于编制大型程序。加上它独特的编程风格, Smalltalk 本身并未发展起来。但面向对象思想, 语言和环境一致性; 交互式和极端用户友好(用菜单和鼠标即可上机), 对 80 年代语言和计算系统产生了巨大影响。它也可称之里程碑式的语言。

各种过程语言，甚至汇编语言都借鉴对象思想，以求能支持面向对象程序设计。82-86 年相继出现 Object Pascal、Objective-C、Object Assembler(68000 汇编程序改造)、Object、LOGO、Object FROTH 它们以原有语言采纳对象——消息编程模式。

另一些语言向类、对象延伸。以对象——引用编程模式编程。如 85 年 AT&T 公司推出的 C++。87 年 Borland 公司的 Turbo Pascal 5.5。

传统的人工智能语言也向面向对象上发展。施乐公司 1983 年在 Intel LISP 基础上研制了 LOOPS。是面向对象 LISP 美国西海岸版本。85 年美国符号处理公司在 MIT 的 LISP 机上开发了 Flavos，称之为东海岸版本。Flavos 有更为灵活和复杂的多继承性。85 年施乐公司又作出 Common LOOPS。1988 年，ANSI X3J13 组将它们统一为 CLOS。85 年，IBM 日本分公司开发了 SPOOL 是 Prolog 面向对象的扩充。86 年 Valcan 和施乐联合开发的并发 Prolog 预处理程序，也是支持面向对象的。

70 年代在软件发展史上是数据库成熟的年代。数据库有数据描述语言 DDL 数据操作语言 DML。它们都是为实现某种模式数据库的专用语言。目标简单，在所应用的领域高效。因此，不能以通用程序设计语言代替。数据库给用户以界面(即查询)语言。对于简单程序没有必要转到通用语言再编程序。查询语言进一步扩充，就形成一系列查询命令加上约束条件控制的非过程语言。如 SQL。1988 年 SQL 成为正式 ANSI/ISO 标准。

80 年代软件环境大发展。操作系统原有的作业控制语言(JCL)和系统调用命令也逐步发展为该环境的统一的界面语言。如 UNIX 的 Shell。

80 年代系统软件中开发环境的思想向各专业渗透。各专业都为本专业的最终用户提供简便的开发环境。即事先将程序模块以目标码存放计算机，用户只需简单的命令，甚至本专业常用的图形就可组成应用程序。这些图形、菜单、命令即用户界面语言。

这些语言共同的特点是声明性(只需指明要做的事)、非过程性、简单、用户友好。而应用程序的实现可由系统自己完成(低层有固定不变的计算模型，如关系运算，也可以连接备用模块智能推理)。这就所谓的第四代语言(4GL)。

4GL 并没有为程序设计语言学带来什么新概念和新特征。一般用传统技术做出界面语言解释器。要求环境有较好的工具支持(应用程序、程序库、各种测试、调试、文档工具)。最简单的 4GL 是图形——菜单，用户不用击键即可完成计算(用鼠标器)。

4GL 是硬件高速发展和快速降价必然的结果。因为不可能在短期内培养出与硬件发展需要相匹配的那么多的程序员。但 4GL 不易编制开创性程序。

1.1.5 九十年代网络计算语言

九十年代计算机硬件发展速度依然不减。每片芯片上晶体管数目仍然是一年半增加一倍。计算机主频从 12-25 兆赫增加到 500-600 兆赫(每秒钟可执行 750MIPS 指令)，价格进一步低廉。使用方式也从多人一机的分时系统到一人一机局域网计算，到每人都成为拥有全球资源的客户。建立在异质网上的多媒体环境已成为客户端使用环境的主流。支持“所见即所得”的用户界面的“语言”大量涌现。

- 由于有良好环境支持，程序设计重点从算法加数据结构实现技术向规模说明描述方面转移。规模说明语言在 80 年代已有研究。(META-IV, EPROL, HOPE, CLEAR, SPECINT, Z)在个别具体领域也能实用。但作为通用，哪怕是某个行业建立在域分析基础上的规格说明语言尚未出现。

- 环境智能化、规模说明语言自动生成目标码客观上要求加入人工智能技术。异质网环境推行后各结点上数据库资源共享都要求各语言间不要有人为的断层。因此，多范型语言研

究会有较大发展。80 年代向面向对象扩充已经出现多范型语言，如 C++，Ada 95(命令式加对象式)、CLOS(函数式加对象式)。此外小型研究有 TABLOG(关系式加逻辑式)、Funlog(函数式加逻辑式)值得注意的是加拿大的 Nial(1983-88)和美国的 G(1986)语言，它们试图将五种范型：命令式+对象式+逻辑式+函数式+关系式统一在一个语言之中。虽然问题重重，但可由此发现许多新概念和新特征，对于程序设计语言研究是非常有利的。

- 随着面向对象数据库和面向对象操作系统的成熟，完全消灭“语义断层”的数据库程序设计语言(DBPL)和持久性程序设计语言(PPL)终将汇合并标准化。这样，程序运行时大量文件到内存转换则可以取消，从而增大了计算机的实时性，甚至取消文件概念。

- 4GL 有了较大发展，种类花色增多，行业标准出现。支持它们的通用程序设计语言是 C、C++、Ada。

各通用程序设计语言的发展及其相互影响见图 1-1。

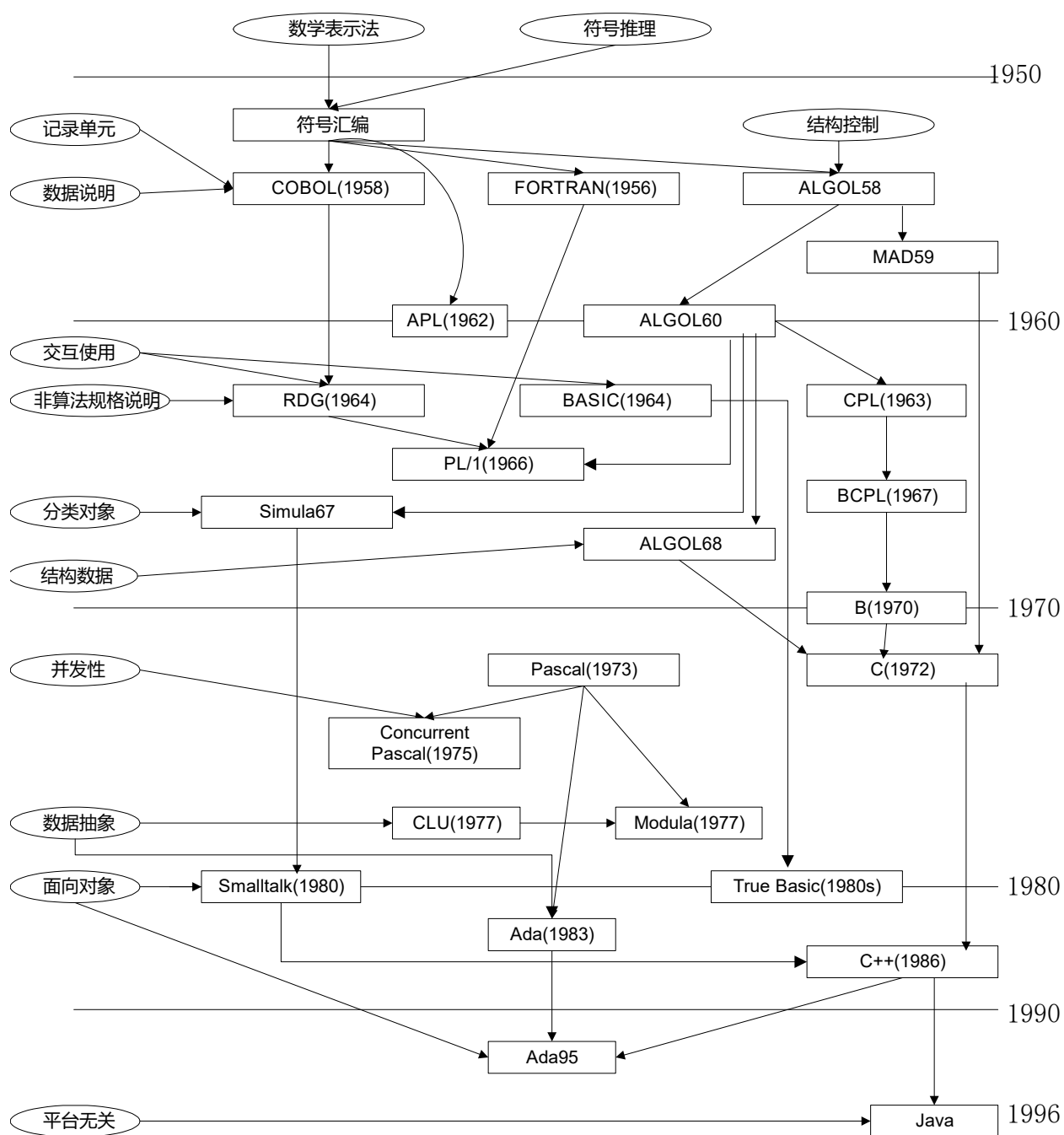


图 1.1 主要通用程序设计语言的发展和相互关系

1.2 程序设计语言的分类

可以从不同的角度对程序设计语言分类。而且一个语言可以分在几个类别中。尽管目前尚无分类标准但大致清楚其所属类别,有利于我们选择使用语言。也可以澄清一些术语概念。

1.2.1 按对机器依赖程度

(1) 低级语言 面向机器,用机器直接提供的地址码、操作码语义概念编程。机器语言和汇编语言,宏汇编虽然抽象层次逐渐提高仍属低级语言的汇编语言(如 8086 汇编,68000 汇编)。

(2) 高级语言 独立于机器,用语言提供的语义概念和支持的范型编程。如命令式(Pascal, C, Ada)、函数式(LISP, ML)、逻辑式(Prolog)、关系式(SQL)、对象式(Smalltalk, C++)。

(3) 中级语言 可以编程操纵机器的硬件特征但不涉及地址码和操作码。如字位运算,取地址,设中断,开辟空间、无用单元收回,用寄存器加速等。高级汇编, C, FORTH 属此列。

1.2.2 按应用领域

(1) 商用语言 处理日常商业事务。有良好文字、报表功能,数据量大和数据库密切相关。代表语言是 COBOL, RPG。

(2) 科学计算 数值计算量大,支持高精度、向量、矩阵运算。代表语言有 FORTRAN、APL。

(3) 系统程序设计 支持与硬件相关的低级操作,编写系统程序(操作系统,编译、解释器,数据库管理系统,网络接口程序)的语言,如 C, Ada, Bliss, FORTH。

(4) 模拟语言 模拟应用主要是以时间为进程模拟客观世界的状态变化。分离散事件模拟和连续模拟。代表语言有 GPSS、SLAM、SIMULA 67。

(5) 正文处理 主要操作对象是自然语言中字符(英文)。很方便产生报告、表格、代表语言是 SMOBOL。

(6) 实时处理 其特点是能根据外部信号控制不同的程序段并发执行。这类语言有并发 Pascal、并发 C、Ada、Mesa、OCCAM, FORTRAN-90, LINDA。用于通讯领域的程序设计语言都要有实时功能,如 Gypsy、CHILL。

(7) 嵌入式应用 在一个大型机器(宿主机)上为小机器(或单片机)开发程序,经调试后将它译为小机器(目标机)的目标码在小机器上运行叫嵌入应用。如机载、弹载计算机。这类程序一般都有实时要求,并近于系统设计。代表语言 Ada。

(8) 人工智能应用 这类程序是对人们的智力行为仿真。包括自然语言理解、定理证明、模式识别、机器人、各种专家系统。这类语言要能描述知识、并根据推理规则推断合理的结论。在符号运算上作谓词演算或 λ 演算是其推理运算的基本方式。代表语言有 LISP 和 Prolog

(9) 查询和命令语言 这是一类新兴的语言,是各种早期系统程序简单的用户命令的发展。数据库语言 dBASE、SQL,操作系统 UNIX 的 Shell 语言为其代表。现代软件环境的用户界面语言更是丰富多采使用方便。其特点是与程序员的交互性和非过程性。

(10) 教学语言 为了培养程序员或使学生很快入门,人们设计了教学用语言。例如,过程程序设计有 BASIC,结构化程序设计有 Pascal,青少年启蒙有 LOGO。但往往由于对程序语言的作用没有全面理解,初学者学到一门启蒙语言后就企图用它作软件设计,这是十分有害的。

(11) 打印专用 图文并用在各种打印机(包括激光)打印字体优美的报告、图形、图象。代表语言有: Postscript、Tex、Latex。

(12) 专用于某类数据结构

(a) 串处理 专用于处理正文字符串,抽取字符串,引用串函数,串形式匹配,回溯与穷举查找。代表语言有: SNOBOL、ICON。

(b) 数组处理 构造和操纵矩阵,可整体操纵数组。代表语言有 APL, Visicale, Loutus。

(c) 表处理 支持表的各种切割、连接操作。直接表 I/O、动态堆栈分配。代表语言有:

LISP、T、Scheme、Miranda。

(13) 数据库应用

(a) 数据库专用, 可完成简单应用, 复杂应用要嵌入通用程序设计语言, 如 SQL、Foxpro、Delphi、PowerBuilder。

(b) 数据库程序设计语言 既支持一般应用操纵临时对象(程序终止即消失), 也可操纵数据库中持久对象, 且可混合在一个程序中, 如 C++, CO2, OPAL。

1.2.3 按实现计算方式

(1) 编译型语言 用户将源程序一次写好, 提交编译, 运行编译得目标码模块。再通过连接编辑、加载成为内存中可执行目标码程序。再次运行目标码, 读入数据得出计算结果。大多数高级程序设计语言属于这一类。

(2) 解释型语言 系统的解释程序对源程序直接加工。一边翻译, 一边执行。不形成再次调用它执行的目标码文件。大多数交互式(interactive)语言、查询命令语言采用解释型实现。典型的例子有 BASIC、LISP、Prolog、APL、Shell、SQL、Java。它们的特点是所用翻译空间小, 反应快, 但运行效率慢。

1.2.4 按使用方式

(1) 交互式语言 程序在执行过程中程序可陆续添加和修改, 以对话方式实现计算。一般是解释型的。由于程序设计支持环境的发展, 交互式语言可方便为用户调用各环境工具, 有日益发展的趋势。

(2) 非交互式语言 多数编译型语言的目标码文件执行期间, 程序员不能干预, 只能在执行完毕再修改。

1.2.5 按程序设计范型

(1) 单范型语言 范型即程序组织和实现计算的模式

(a) 命令式语言 计算实现的模型如果按冯·诺依曼原理强制改变内存中的值叫命令(或译指令、强制 Imperative 式)的。所有过程语言都基于这个原理。由于强制改变值, 程序状态的变化没有一定规则, 程序大了就很难查错, 很难调试, 不易证明其正确。组织程序的范型即: 算法过程+数据结构。到目前软件开发主导语言仍是命令式语言。近代命令式语言增加了模块强制类型检查、抽象数据类型、类属等机制可开发较安全、可靠的大程序。

(b) 面向对象语言 将数据和其上的操作封装于对象中。对象归属于类对象, 类对象有继承, 实例对象上的操作可动态决定。程序是相互发消息通信的对象集合。代表语言有 Simula-67, Smalltalk。

(c) 数据流语言 传统过程语言中以程序控制保证程序功能实现, 数据是分散的, 为控制流服务。数据流语言以数据对象为核心加工过程为其服务。藉以提高运算速度。这种语言的程序设计方法学模型是基于数据流。数据流语言有 Val 和 EUCLID。

(d) 函数式语言 程序对象是函数及高阶函数, 组织程序的范型是函数定义及引用。代表语言有 LISP、FP、ML、Miranda。

(e) 逻辑式语言 程序对象是常量, 变量和谓词、组织程序的范型是定义谓词并写引用谓词的公式, 并构造满足谓词的事实库和约束关系库。代表语言有 Prolog。

函数式和逻辑式语言不需要强制求值故也称施用式语言(Application), 又因无需涉及求

值过程，只定义“求什么？有什么(函数)关系”故也叫声明式(declarative)语言。

(2) 多范型语言

一个程序设计语言不止支持一种程序设计范型，最初为将一个常用的语言扩充具有另一范型的能力，§ 1.1.5 老语言对象化就有很多实例，即它们是两范型的，有目的研究五种范型组合是 Nail (1983)和 G(1986)。

并发程序设计应该说也是一种范型，因为它在组织程序时程序执行的非顺序性和各成分相互通信的时间要求和单主机上顺序程序是不同的。但在各程序成分内的组织与设计与该种语言顺序部分没什么不同，即各种范型语言均向并发语言发展如 Ada, Concurrent Smalltalk, Concurrent Prolog。并发语言必然是顺序部分的超集。

1.2.6 按断代划分

由于硬件中从真空管到超大规模集成电路已经五代了。程序设计语言的断代当然也要按自己的特征：

(1) 第一代语言 1GL 50 年代

主要特征是面向机器，离不开地址码，操作码，存储空间分配。机器语言、汇编语言即是。

(2) 第二代语言 2GL 60 年代

主要特征是脱离机器面向算法过程的高级语言。有变量、赋值、子程序、函数调用概念。少量基本数据类型，有限的循环层次(三层至多七层)。一般无递归调用。FORTRAN、BASIC、ALGOL 60、COBOL 即是。

(3) 第三代语言 3GL 70 年代。

主要特征是结构化控制结构，块级控制。有作用域和可见性概念。有丰富的数据类型，除基本类型增加了布尔、集合、字符类型而外，用户可自行定义结构数据，枚举数据，枚举数据，还可通过指针定义动态数据。第三代语言的典型代表是 Pascal。第三代晚期，出现了程序(或模块)定义和实现显式分离特征：规格说明(Specification)只定义程序的功能，体(body)是规格说明的过程实现。Ada 和 Modula-2 就是典型例子。

(4) 第四代语言 4GL 80 年代

主要特征是极端用户友好。最终用户(end user)只经过几天甚至几小时训练即可上机。它是声明式、交互式、非过程化语言。依赖于环境支持，一般都要有大的数据库。编制一个程序要比第三代语言所花时间少一个数量级。但由于把许多编程工作放在支持系统中自动完成，往往只有某一方面的功能，所以 到目前为止还没有通用的第四代语言。如 LOTUS1-2-3 只适合表格处理。dBASE 适合数据库查询和应用。MANTIS、IOEAL、NATURAL、APPLICATION FACTORY 声称凡 COBOL 程序都能编，但也仅限于数据处理。

(5) 第五代语言 5GL

习惯上把用于人工智能程序表达的语言称之为五代机语言或五代语言。实际上 LISP 早在 1957 年就设计出来了。它是对抽象的符号进行表处理而不是象 ALGOL 和 FORTRAN 那样算出数值，所以 一开始就用于数学定理证明之类的智能程序。70 年代出现的 Prolog 更是为智能推理而设的，它们都是小语言，而早期都是解释执行的。它们只用一种程序设计模型在人工智能方向试探。它们的特点都是在上层按某种模式去开发程序，下层实现则千篇一律的笨办法(例如，Prolog 用的是递归树查找的匹配方式)。于是，有人研究 LISP 机 Prolog 机，下层直接是推理单元而不必用冯·诺依曼机去模拟推理、找出匹配。但在非它所长的智能应用中 LISP 机、Prolog 机笨拙无比。所以五代机及其语言目前只能说有了萌芽。真正五代机通用语言是什么，目前还很难预言。

还有一种说法是把除正文语言之外的其他媒体语言称为 5GL。从断代是革命而不是改良的角度多媒体语言研究应该称做第五代。但目前仅仅是把正文与其他媒体可等价的部分用其他媒体表达,从而简化程序设计(它还是基于正文的),如前所述,各媒体有其自身不可等价性(如声调、声音速度、强度带来的语义反意),怎样程序设计?目前尚不可知。

1.3 本章小结

- 本章按年代给出了程序设计语言发展的各阶段,程序设计语言研究最主要是高级程序设计语言。它的出现,奠基研究、完善、向对象式发展,九十年代只给出六个可能的发展方向,并发式、多媒体(5GL)、4GL 完善、规格说明式、数据库程序设计语言、多范型式。

- 可以按多种准则为程序设计语言分类。一个语言可以在多个类别,本书按对机器依赖程序、应用领域、实现计算方式、使用方式、编程范型和断代划化,六个角度分类。所指语言一般至少在某个领域是比较通用的。特别专门的*语言不是本书研究范围。

- 历史上有较大影响的语言是:

FORTRAN, COBOL, Algol-60, PL/1, LISP, Algol-68, BASIC, Pascal, APL, Simula, C, Ada, Smalltalk, Prolog, ML, C++, SQL, Java

本章习题

1.1 学习本教程之前你会哪些语言,它们是哪个时代的产物,它的祖先,它的类别,它的特点,全部列出之。

1.2 考证一下 dBASE 是从什么通用语言演变而来的。它的特点是什么,与祖先语言同异。

1.3 BASIC 能不能编人工智能程序?为什么?

答:一般不能。

首先,人工智能应用,这类程序是对人们的智力行为仿真,包括自然语言理解、定理证明、模式识别、机器人、各种专家系统。这类语言要能描述知识,并根据推理规则推断合理的结论。在符号运算上作谓词演算或 λ 演算是其推理运算的基本方式。因此此语言最好是非过程的,以直接反映客观事物之间的关系。BASIC 是过程性语言、是交互性、解释型语言,这是由于它的全程量数据、无模块,决定了它只能编制小程序,它不能定义数据类型,没有递归,这对匹配回溯是致命的。因此一般情况下,不能满足人工智能的要求。但它有判断,转移和条件,可以在相对窄小范围内模拟实现专家系统程序。所以不能绝对。

1.4 什么是嵌入式语言,它有哪些特点,有什么好处,有什么麻烦?

1.5 就你知道的软件工程原则,评价 C 语言作大型软件开发的优缺点。

1.6 同 1.5,评价 Ada 语言作大型软件的优缺点。

1.7 小结第三段列举的 18 种语言,你用一两句话说明它在历史上重要性。至少要说 10 种。

1.8 程序设计语言的成功有以下因素:

成功=设计好坏+实现难易+权势支持+社会需要

试比较 FORTRAN-Algol-60; Modula-2-Ada; BASIC-Pascal; Pascal-C; PL/1-Algol-68(任选一组)。

答: Pascal-C

Pascal 的研究者一开始就本着“简单、有效、可靠”的原则设计,因此一问世,就取

得了巨大的成功。在人们为摆脱软件危机而对结构化程序设计寄予极大希望的时代，Pascal 得到很快普及，以上说明设计好坏和社会需要使 Pascal 取得成功，但随着社会发展，硬件继续降价，功能和可靠性进一步提高，人们对软件的要求，无论是规模、功能、还是开发效率都大为提高。因此只能编顺序小程序的 Pascal 已不能满足需求了，也就慢慢只在教学示范中使用。

C 是一个表达能力强、顺序的、结构化程序设计语言，它给程序员较大的自由度。下层数据转换灵活。上层是结构化的控制结构，它的分别编译机制使它可以构成大程序。输入/输出依赖 UNIX，使语言简短。C 得益于灵活的指针，函数副作用和数据类型的灵活。它设计上虽不完美，但它简洁、近于硬件、代码高效、并有大量环境工具支持，以及实现上的优势，使它成为系统软件的主导。

FORTRAN 与 ALGOL

FORTRAN 语言的出现使当时科技计算为主的软件生产提高了一个数量级，奠定了高级语言的地位，1960 年的“算法语言 ALGOL 的修订的报告”对 ALGOL 的定义采用相对严格的形式语法。ALGOL 语言为广大计算机工作者接受，当时 IBM 公司当时经营世界计算机总额 75% 的销售量，一心要推行 FORTRAN，不支持 ALGOL，以致 ALGOL60 始终没有大发展起来。以上说明了“权势支持”是影响程序设计语言成功的因素之一。

Modula-2 和 Ada 语言。

Modula-2 和 Ada 在设计上是相似的，二者皆是强类型语言，封装的程序包是程序资源构件。

从实现上来看，Modula-2 语言 9000 句编译器就具有了 Ada20 万句编译器 80% 的功能，明显优于 Ada。但是，Ada 有美国国防部的支持，故 modula 在与 Ada 的竞争中处于不利地位，最终也没有取得很大成就，不及 PASCAL 语言的。而 Ada 在美国国防部的支持下，发展迅速。该语言的开发完全按软件工程方式进行，但是 Ada 过多强调安全性和易读性，因而编译程序要做许多静态检查，故体积庞大。且由于 Ada 的环境工具发展缓慢，故自其第一个版本以来，并未取得投资者预想成就。并且，随着软件工程本身向集成化、可重用、面向对象方向发展，Ada 已有些不适应，然而美国军方仍然支持，故 95 年，Ada 完成面向对象的改造，但这种改造使其成为最庞大臃肿的语言，可是它却被 ANSI 和 ISO 接受，成为世界上第一个有法定标准的面向对象语言。由此可见，程序设计语言的成功不仅在于其设计的好坏和实现的难易，权势支持和社会需要也是重要因素。

1.9 试述程序设计语言 Programming Language(PL)与程序设计语言 Program Design language (PDL)有什么不同。

1.10 探讨一下程序设计语言是否向人类自然语言靠近？有没有可能不用程序设计语言，用规范化自然语言上机？

第2章 程序设计语言的设计概述

本章介绍程序设计语言的设计目标以及为实现这些目标所遵循的设计准则, 设计语言最后要给出语言的参考手册, 为此, 从表示的角度介绍与语法, 语义和上下文约束有关的概念和表示法. 目前在程序语言语法文本中, 语法形式表示是完整的, 语义表示多半是非形式的. 本章仅给出各种形式语义的初步概念, 有关理论后文还要讲述。

2.1 表示与抽象 (Representation & Abstraction)

程序就是程序设计语言表示的计算。所谓表示是人为制造的符号组合以表达我们需要表达的意思。每个合乎语法的表示都表达了某种语义，例如：

```
float n; //n是C语言中的浮点数变量
sqrt(n); //对n取平方根
```

由于我们约定float是浮点类型的关键字，故float n这个合法表示表明“n是浮点数”，是一个静态的说明。同样，sqrt(n)说明要做“对n取平方根”的动作。

表示是对事物抽象的表达。抽象是对论题本质的提取。当我们为某软件投入市场作市场调查并写出分析报告时，我们已先完成了抽象（从社会效益、技术效益、经济效益列出定性或定量的分析），也完成了表示（报告正文）。表示的规则是语法的，抽象的，内容是语义的。同一事物可在不同抽象层次表示它。同一程序可用高级语言表示、汇编码表示以及机器码表示。当然，高级语言上面还可以有甚高级、超高级语言不断抽象上去。所以，每种语言都是在某个抽象层次上的表示。

从日常概念中，我们可以得知概念愈是抽象其内涵愈小(只包含主要特征，略去其余)而外延愈大(能满足该主要特征的事物愈多)。例如“张三在种地”也可以说“张三在改变生态环境”，还可以说“张三修理地球”都没有错。之所以不错是表达了主要概念：“地表情况有了变化”。但张三用锄种玉米的细节全被抽象(忽略)掉，且“李四砍树”也可以说成“改变生态环境”、“修理地球”。当然，这种近乎失真的外延是不希望的，同样，能满足高级语言描述的计算的机器码实现也可以不止一个。例如：给变量x增值1， $x=x+1$ ；这类赋值语句编译后的机器码可以有如下两种做法，按语义描述为：

- | | |
|------------------------------|-------------------------|
| 1. 从内存代表x的地址中取出值放在运算器中。 | 1. 从内存代表x的地址中取出值放在运算器中。 |
| 2. 加1，将结果放于某临时单元。 | 2. 加1将结果放入x地址中。 |
| 3. 将临时单元内容作类型检查(必要时转换)并放入x中。 | $x+=1$; |

两者都对，但效率不同。左边按赋值常规做法，保证赋值安全并没有错。右边则认为编译时表达式 $x+1$ 已作了类型检查，结果必然是x的类型，根据这个特殊情况节省了一步，C语言的 $x+=1$ ；的写法就是后者。其它语言两者均可，完全由编译器决定了。

2.1.1 上层抽象可用多种下层抽象实现。程序设计的四个世界。

客观世界的问题描述了客观世界对象相互之间的作用，经过分析建立解题模型。

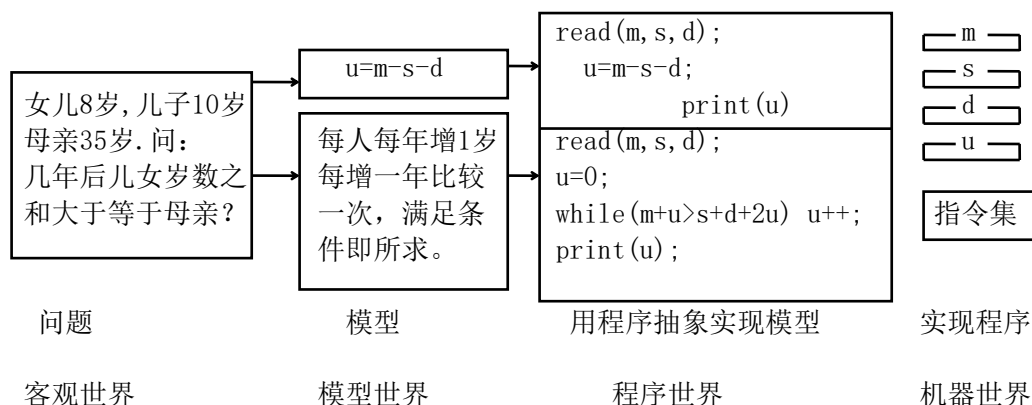


图2-1 计算机解题的四个世界

一种模型只抽取与该层抽象有关的数据属性。例如按图2-1的问题，儿子、女儿、母亲的姓名、身高体重、文化程度……与本题无关。问题中的对象在本例中是三人年龄和要求解的年数。数学模型是最高级的抽象，解题公式 $u=m-s-d$ 即本题的数学模型。程序对象可以是 u, m, s, d ，它直接和数学模型对象对应。由于程序对象在程序世界有自己的行为，如输入/输出操作。到计算机实现时又映射为存储对象和指令集。它又要加上分配存储、查找地址和往运算器、存储器和输入/出设备送数的操作(输入/出还要有数、码变换)。所以，同一问题在问题世界，模型世界，程序世界，机器世界中对象及其相互作用是不同的。我们研究程序设计语言的设计是从问题—模型世界的需求出发，在机器世界可能实现情况下，定义自己的对象表示及其相互作用的表示。对于从问题到程序，我们只介绍已有程序设计范型，对于程序到实现，我们在以后章节将陆续分析程序各元素和各机制(即每种特征)以及它们的实现。

2.1.2 显式表示和隐式表示

程序对象及其相互作用的表示可以是显式的。如上例中的 `float n`，显式说明 n 是浮点变量；也可以是隐式的，例如数组元素的次序(按下标序不可改)；一组连续的语句(语句次序不可改)；程序中的缺省声明，如FORTRAN中符号名首字母为 $1..N$ 的程序对象自动为整型，其余为实型。再如Ada中：

```
for J in 1 .. 5 loop
  ...
  ...
end loop;
```

其中 J 的类型自动为循环域 $(1..5)$ 表示给出的整型。一般说来，编译时，信息越多越准确，所以，强类型语言强调显式表示。但显式表示带来不必要的噜嗦，使程序不够精炼。所以，在语义明确，不易出错之处，也采用隐式表示，例如Ada的常量声明：

```
PI: constant FLOAT:=3.1415926;
```

可写为:

```
PI: constant := 3.1415926;
```

因为以3.1415926初始化时，等于声明了PI必须是浮点类型。

2.1.3 聚合表示和分散表示

程序对象、操作过程(或函数)在早期语言中为了快速翻译，只能用较原始表示组合起来表示复杂的实体，或类似实体用不同的表示指明微小的差异，即分散表示。例如，FORTRAN中没有记录数据结构，则将每个域定义为数组，各数组按同一序号取一元素构成一个记录。再如，FORTH语言中不同类型“加”运算的分散表示：

```
+      单字长加号
D+     双字长加号
M+     混合加号
```

其它语言中一个+号就概括了。这叫聚合 (Coherent)表示。分散表示并不一定都坏。例如，早期语言的函数(或过程)定义的类型构(Signature)和体是不能分开的：

```
FUNCTION SUM (V, N)           //函数型构
REAL V (N)                   // 局部变量
SUM = 0.0
DO 200 I = 1, N
SUM = SUM + V(I)
200 RETURN

END
```

} // 函数体

而Ada等现代语言为上层表达方便, 型构和体可以分开，只写：

```
function SUM (V: VECTOR, N: INTEGER) return FLOAT;
```

函数体可以写在别处，只要在联编时有一个完整定义的SUM函数存在，内部内联成一“内聚式”即可运行。

2.2 程序设计语言的设计目标

设计程序设计语言即定义一组能表示某种范型的特征集。每个特征有严格的定义且可以在机器上实现。程序员灵活运用这些特征便可表达他所希望任何计算。在某种意义上说，只要能想得出巧妙而高效的实现，就能提供更多更方便的特征，但就一个语言来说，这一组特征要能体现它的设计目标。

语言设计的目标在不同时代的软、硬件背景下不能同一而语。例如，早期语言追求时空效率，近代软件追求可靠性(易验证)，可维护性(易读易改，改了不产生旁效)，可移植性(尽可能远离具体机器)。不同的应用范型也有不同的要求。这里仅就当今最一般的目标，给出定性的描述：

(1) **模型能力(Model Power)** 语言规定的语法特征应使程序员能充分、精确、容易地表达他的计算。

(2) **语义清晰(Semantic Clarity)** 每个合乎语法的表达式和程序都定义一个清晰、无二义性的涵义，且不因程序执行引起语义改变。

(3) **实用性(Utility)** 只设计使用频繁或没有它不行的特征，即每个特征都常用。

(4) **方便性(Conveniency)** 越是频繁使用的特征越要简便。

(5) **简单性 (Simplicity)** 力求简单, 一致, 少生僻, 除非十分方便不轻易增加冗余。

(6) **可读性 (Readability)** 程序员往往从其特征定义的表达形式上理解其内涵, 力求直接表达, “相当于”表达。

(7) **可移植性 (Portability)** 一种语言的程序在一般的机器上均可实现其计算, 且不产生歧义。

(8) **效率 (Efficiency)** 程序设计语言的效率指三个方面: 编程效率; 预处理和编译效率; 执行效率。它们相互相关, 对语言追求目标有不同的选择, 总的希望都很高, 但非常不易, 只能折衷。

(9) **灵活性 (Flexibility)** 对语法以外的限定在不带来歧意的前提下允许放松。这样能更好地发挥程序员的能动性并可简化实现。C语言做得较好, 如数组名也可以是指针, 字符也可解释为整数等。

2.3 设计准则

以上目标有时是相互冲突的, 设计者只能按具体应用目标折衷。为达到上述目标应注意以下准则:

(1) **频度 (Frequency) 准则** 频繁的特征应语义清晰、方便、简单、可读。例如, `if_then_else`很频繁, 结构化后更清晰。

(2) **结构一致 (Structure Consistency) 准则** 一个程序的静态结构, 直观上应与动态的计算结构一致, 即表示结构与逻辑结构统一。这是E. W. Dijkstra提出的著名的准则, 并导致结构化程序普及。违反这个原则, 程序难以阅读、测试和维护, 例如:

| | | |
|-----|---------------------|--------------------|
| 100 | IF (P) GO TO 150 | while NOT P do |
| | A段 | A段 |
| | IF (EXP1) GO TO 110 | if (EXP1) then |
| | B段 | C段 |
| | GO TO 140 | elsif (EXP2) then |
| 110 | C段 | D段 |
| | IF (EXP2) GO TO 120 | elsif (EXP3) then |
| | GO TO 140 | |
| 120 | D段 | E段 |
| | IF (EXP3) GO TO 130 | else |
| | GO TO 140 | B段 |
| | | endif |
| 130 | E段 | F段 |
| 140 | F段 | end do |
| | GO TO 100 | |
| 150 | | |

单看左边程序不花上10分钟很难理解程序是怎样执行的。

(3) **局部性 (Locality) 准则** 一个模块 (或函数、过程) 只用自己声明的局部数据做一件事。这是保持语义清晰最有效的措施。这个模块的修改不影响其它的部分。近代程序小函数、小过程特别多, 八件事调用八次, 势必增加运行时调用界面上的类型检查, 使运行速度降低, 为此, 语言设计上增加了内联 (`in_line`) 机制。将函数代码插入到调用之处 (插入时作类型检查)。既保持了局部性, 又保持了速度。

因为公用数据极易产生副作用 (也称边界效应), 造成程序错误, 而且难以查错。强行将公用数据局部化又会造成冗余和重复, 逻辑上增加混乱。故近代语言有不同层次控制数据公用措

施，如堆变量、自动变量、静态变量、全局量等。函数式语言全部采用局部量，Pascal、Ada、C有限制地使用全局量，BASIC全部是全局量。

(4) **词法内聚 (Lexical Coherence) 准则** 逻辑相关的代码表示上应相邻。为了编译方便，早期语言将声明部分和语句部分明显分开，如 Pascal。声明集中在顶部，过程在中间，主程序在最下面。这就天然适合于小程序(不超过一页)。程序大了阅读主程序时来回翻以查对数据类型，是十分不方便的。所以C++的声明也是语句，可以插在语句出现的任何地方。

这样，要用什么变量，声明了就使用，这种分散表示又带来了缺乏总体概念(到底有几个变量要到处找)的缺陷，所以，C++规定声明可以多次，定义只能有一次，就缓解了这个矛盾。

许多语言的新版本都注意到词法内聚准则，如：

Common Lisp 也对早期LISP作了改进，例如：

| | |
|-------------------|----------------------------|
| (lambda (<哑参数名表>) | (let (<哑实参数对>) |
| (<函数体>)) | (<函数体>)) |
| <实参值表>) | |
| ((lambda (x y) | (let ((x 3.5) (y (+ a 2))) |
| (+ (* x y) | ((+ (* x y) |
| (- x y))) | (- x y))) |
| 3.5 (+ a 2)) | |

早期LISP的入表达式

Common LISP的改进

如果函数体很长，就近哑实结合是很方便的。

(5) **语法一致性 (Syntactic Consistency) 准则** 看上去相似的特征，其语法定义应一致。这是为了减少程序员的记忆负担。例如，FORTRAN 的计算转移语句：

GO TO (L1, L2, ..., Ln), I

I是整变量，算出第几(1..n中的某个值)就转向第几个语句标号处。但FORTRAN同时有赋值转语句：

GO TO N, (L1, L2, ..., Ln)

N是要显式赋值的语句标号：ASSIGN Li To N。N和I相似，但意思、执行方法相差很多。

(6) **安全性 (Security) 准则** 没有违反语法规则的程序错误是不能检查出来的。例如，上述FORTRAN计算转语句，程序中规定了八个语句标号，编译很容易通过。运行时I=9也符合I的类型规定，但无法执行，是停机还是转到所在标号之下的第一语句呢？只好由编译器实现者定。这样，同一程序在不同编译器的环境下就有不安全因素。Pascal的 Case语句也有类似情况。近代语言的Case都增加了Otherwise分支，以防指示变量的值未按程序员预想的出现。

安全性导致**静态强类型**，即在编译时刻所有程序对象的类型均应检查。这样运行时检查减至最小，可保持较高的运行效率，同时有利于程序员修改程序。然而，静态强类型不利于动态继承，不利于程序扩充修改。于是，有人研究**动态强类型**，编译时插入运行检查代码，等到有了确定类型值时检查。这样势必减低运行效率。所谓“强”是每一个运算的对象都要检查。早期的C(K&R版本)是不安全的，算术运算和函数调用的参数一般不做类型匹配检查，安全性由程序员保证。

(7) **正交性和正则性 (Orthogonality & Regularity) 准则** 每种语言特征都由独立的机制实现，即与其他特征的实现机制无关。也就是说，使用或修改某个特征对其他特征没有影响。所以一般语言都以互不相关(正交)的基本机制来构造更有表达力更好用的机制。与此有关的，用基本特征组合为更加复杂的特征时要符合正则性原则，所谓‘正则性’(Regularity)是指设定的某种规则或约束应无一例外。例如，数据类型是一个特征，函数求值是一个特征，设计时它们是正交的。FORTRAN, Pascal均不能满足正则性原则，即函数不能返回数组。C语言可以(返回指针即数组)。然而，若基本特征本身就不正交，正则性一定不能保证。正交设计是语言设计的基本目标。正则性减少程序员对过多例外的记忆。

(8) **数据隐藏 (Data Hidding) 准则** 程序模块应有显式的接口(也称界面)，有了这个界面，用户就能正确使用本模块，模块实现者就有了实现本模块的所有信息。这是D.Parnas提出的著名的模块设计准则。即实现中增加的辅助数据全部对用户隐藏。只要不改界面，对模块内部的任何修改都不会影响到其他部分。只重视显式接口也是抽象性、简单性的体现。所

以, Ada等近代语言把模块(包、函数、过程、任务)的规格说明和体显式分开。规格说明即接口集, 并把规格说明放在一起, 阅读、查找大程序特别方便。如果模块体是事先测试好的库程序、预定义的程序, 那么编制程序可以在接口的层次上, 工作量可以减少一个数量级。这是实现软件重用、提高软件生产率的语言措施。

(9) **抽象表达(Abstract Representation)准则** 若某表达陈述不止一次, 则应抽取因子使其以递归形式表达。例如:

$ax + bx$ 应可表达为 $(a+b) x$

因为递归是使有限表达无限计算, 极其精炼, 也是符号计算中常用的手段。故新语言或老语言新版中都增加了递归表达机制: 递归函数调用和递归数据类型。

抽象表达也借鉴自然语言中主干词加修饰词的方式。即主干词定义较概括的抽象, 修饰词使其具体化。例如, C++中:

```
static const int max_index=MAX_LENGTH-1;
```

这种多层修饰近代语言愈来愈明显。

抽象表达的第三种措施是提供高层抽象。早期语言只有语句级语法机制, 近代语言有命名的包和模块(Ada, Modula-2)。名字代表整个块, 对其内部成分的引用, 则用多层点表示法(由抽象到具体):

```
MATHLIB. TRIANG. COS(X);
```

意思是数学包中的三角形包的余弦函数。

(10) **可移植(portability)准则** 程序语言不应有依赖硬件或环境的特征。完全不依赖硬件或环境则会影响表达性和效率, 例如, Pascal 就没有硬件操作的抽象, 位模式运算, 表达力比C差。两者兼顾是较难的事, 因为该语言的程序总要在具体的环境和机器上运行。为了可移植把语言核心定得完全脱离机器, 而每个机器提供一预定义环境是近代语言策略。Ada的基本数据类型和输入/输出机制完全由预定义程序包(用Ada加汇编写、不能够移植)提供。因此, Ada可以不受机器物理字长限制控制数学精度。输入/输出在FORTRAN, COBOL中是语言的一部分, 然而在C, Pascal, Ada中是预定义函数。移植, 扩充, 修改都比较方便。特别是, 程序员在编程时勿需过早把注意力集中在输入/输出的格式上。

除了以上十项准则而外, 还有可扩充—子语言准则, 类型一致性准则, 可见性控制原则等已见诸文献, 但未获普遍推荐。本书后文中遇到时再介绍。

2.4 程序设计语言的规格说明

如前所述, 定义一个语言文本是定义语法和语义, 暂不涉及语用(Pragmatics)。用于表达程序设计语言的语言统称之为元语言(Meta Language)。语言规格说明必须采用一种元语言。不是以有限制的自然语言(如C)就是建立在符号学上的形式语言(Ada)。前者称为非形式表示, 早期语言采用。由于自然语言不准确、有歧意, 近代语言多采用后者。本节介绍近代语言采用的描述方法。

2.4.1 语法的规格说明

从符号学的观点, 语法定义了一组符号, 以及这组符号合法的组合规则。例如, ASCII码或EBCDIC码的基本字符集即一般程序语言的基本符号。组合规则分两部分:

微语法 也叫词法(Lexicon), 定义什么样的符号组合可以构成有意义的词法短语(也叫标记token), 并规定注释、空白的意义。词法短语一般分五类: 字面量、标识符、保留(关键字)、操作符和特殊符号。组合后的标记就有了自己的词义。相当于英语的单词(word)或汉语

的‘词’。

宏语法 定义什么样的词法短语的组合是合法的语法特征。这包括声明、命令、表达式和保留字规定的语法单元。它们往往是相互嵌入的，其组合规则由文法规定。

一般程序设计语言的语法由统一的文法一道给出，包括微、宏语法，只在作编译释意时区分它们。

2.4.1.1 上下文无关文法

(1) 文法 (Grammar)

文法可导出该语言所有可能的句子，形式地，一个文法G是一个四元组：

$$G = (S, N, T, P)$$

其中，T是终结符号串的有限集。

N是非终结符号串的有限集，是文法中提供的成分概念，相当于英语动词短语、名词短语或定语从句等句子成分的符号表示。

$T \cap N = \Phi$ ，即它们是不相交的。

S是起始符号串， $S \in N$ 。

P是产生式，一般形式是：

$$\alpha \rightarrow \beta \quad \alpha, \beta \in (T \cup N)^*$$

“ \rightarrow ”表示左端可推导出右端，如 $\alpha \rightarrow \beta$ ， $\alpha \rightarrow \tau$ ， $\alpha \rightarrow \delta$ 则可写为：

$$\alpha \rightarrow \beta \mid \tau \mid \delta$$

如果产生式将语言的非终结符中的每一个标记都推得为终结符号，则这一组产生式集即为该语言的全部文法。

(2) 文法的递归表示在形式文法中是必须的

例2-1 整数的产生式表示法：

$$\langle \text{digit} \rangle \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$$\langle \text{Integer} \rangle \rightarrow \langle \text{digit} \rangle \quad \text{一位数字是整数}$$

$$\langle \text{Integer} \rangle \rightarrow \langle \text{digit} \rangle \langle \text{digit} \rangle \quad \text{两位数字也是整数}$$

$$\langle \text{Integer} \rangle \rightarrow \underbrace{\langle \text{digit} \rangle \cdots \langle \text{digit} \rangle}_{n \text{ 个}} \quad n \text{ 位数字也是整数}$$

n个

这势必造成产生式臃肿，如果写成：

$$\langle \text{Integer} \rangle \rightarrow \langle \text{digit} \rangle \mid \langle \text{Integer} \rangle \langle \text{digit} \rangle$$

$$\mid \langle \text{digit} \rangle \langle \text{Integer} \rangle$$

这种形式，我们要几位就几位，一条产生式就够了。我们只要不断以 $\langle \text{Integer} \rangle \langle \text{digit} \rangle$ 置换右端的 $\langle \text{Integer} \rangle$ 最后代之以 $\langle \text{digit} \rangle$ ，即可得到任何整数。读者也许注意到上述最后的产生式第二、三个替换取一个就可以了。但它们在文法上是不同的， $\alpha \rightarrow \alpha \beta$ 是左递归产生式，而 $\alpha \rightarrow \beta \alpha$ 是右递归产生式，也叫尾递归的。

不同型式的产生式决定了不同型式的文法。

(3) Chomsky 文法

• **0型文法** 如果对产生式 $\alpha \rightarrow \beta$ 左端和右端不加任何限制，即 $\alpha \in (N \cup T)^+$ ， $\beta \in (N \cup T)^*$ 。这种文法对应的语言是递归可枚举语言。在编译理论中，图灵机(或双向下推机)可以识别这种语言。

• **1型文法** 如果产生式形如：

$$\alpha A \beta \rightarrow \alpha B \beta \quad \alpha, \beta \in (N \cup T)^*, \quad A \in N, \quad B \in (N \cup T)^+$$

则叫做上下文相关文法，对应的语言是上下文敏感语言。线性有界自动机可识别这种语言。

• **2型文法** 如果产生式形如：

$$A \rightarrow \alpha \quad \alpha \in (N \cup T)^*, \quad A \in N$$

左端不含终结符且只有一个非终结符。这种文法叫上下文无关文法。对应的语言即上下文

无关语言。非确定下推机能识别这种语言。

• **3型文法** 如果产生式形如：

$$A \rightarrow aB \mid Ba \quad a \in T^*, A, B \in N$$

左端不含终结符且只有一个非终结符。右端最多也只有一个非终结符且不在最左就在最右端。这种文法叫做正则文法，对应为正则语言。有限自动机可识别这种语言。

显然，这种文法经置换可消除右端非终结符，使每一产生式均可用一终结符的正则表达式表达。

例2-2 所有产生式的非终结符均可置换为终结符表达式。

设产生式是

$$N = \{S, R, Q\}, T = \{a, b, c\}$$

$$P = \{S \rightarrow Ra, S \rightarrow Q, R \rightarrow Qb, Q \rightarrow c\}$$

则有 $S \rightarrow Ra \rightarrow Qba \rightarrow cba \mid S \rightarrow Q \rightarrow c$

$$R \rightarrow Qb \rightarrow cb$$

$$Q \rightarrow c$$

每一型文法的语言都是上一型文法语言的子语言，如图 2-1。一般说来，简单语言用3型文法即可描述。如汇编语言、程序设计语言的词法子语言。对于程序设计语言中的嵌套结构3型文法无能为力，所以采用2型文法(自然包含3型)即上下文无关文法。显然，对于更加复杂的结构采用1型文法更好。但1型文法有复杂的二义性问题，使语言处理器复杂且不可靠。目前0、1型文法尚停留在语言理论研究领域。对于程序设计语言中出现个别上下文相关特征，用上下文约束予以说明。

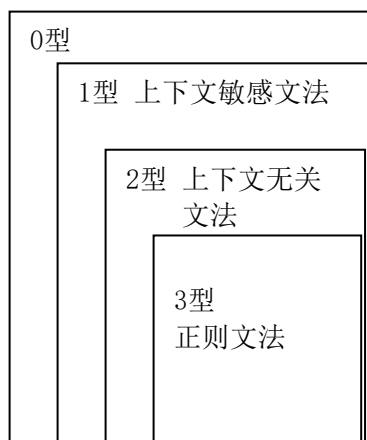


图 2-1 Chomsky 文法的关系

2.4.1.2 BNF和EBNF

1960年J. Backus在Algol-60语言中采用巴库斯范式BNF描述语法。BNF就是上下文无关文法的表示法。其元语符号是：

$::=$ '定义为'，即产生式中的“ \rightarrow ”符号。

$\langle \rangle$ 表示所括符号串是非终结符号串。

$|$ '或者' 表示左右两边符号串序列是可替换的。终结符、关键字、标点符号直接写在产生式中。最初这三个符号就够了。

例2-3 BNF示例

```

<unsigned integer> ::= <digit>
                    | <unsigned integer> <digit>
<integer> ::= +<unsigned integer>
            | -<unsigned integer>
            | <unsigned integer>
<identifier> ::= <letter>
                | <idenfitier> < digit>

```

| <identifier> <letter>

以后,又增加了以下符号使之更精炼:

[] 表示括号内的内容是可选的。

{ } 表示括号内的内容可重复0至多次。

C+ 念'Kleene加'表示语法类C可重复一到多次。

C* 念'Kleene星'表示语法类C可重复0至多次。

例2-4 例2-3进一步化简的表达

<integer> ::= [+|-]<unsigned integer>

<identifier> ::= <letter> {<digit> | <letter>}

<unsigned integer> ::= <digit>⁺

<identifier> ::= <letter> {<digit> | <letter>}^{*}

1974年Niklaus Wirth在作Pascal语法定义时将BNF与正则表达式结合起来,使表达更精炼。这就是扩充的巴库斯范式EBNF,也是当今使用最普遍的表达法。

• 其元语符号变动是: 增加[]、{ }、() (表示成组)、. (表示产生式终结)。^[]、{ } 意义同前,旨在消除或减少递归表达。

• 取消非终结符的尖括号,至少是产生式左端,为此符号串中空白用'_'连接。

• 为区别元符号和程序符号(程序中也有[]、.、()、.),程序中的终结符加引号,如'(', ')', '.', ' '。

例2-5 按ISO标准部分Pascal语言的EBNF产生式:

program ::= <program_heading> ';' <program_block> '.'.

program_heading ::= 'program' <identifier> ['(' <program_parameters> ')'].

program_parameters ::= <identifier_list>.

identifier_list ::= <identifier> { ',' <identifier> } .

program_block ::= <block>.

block ::= <label_declaration_part> <constant_declaration_part>

<type_declaration_part> <variable_declaration_part>

<procedure_and_function_declaration_part> <statement_part>.

variable_declaration_part ::= ['var' <variable_declaration> ';']

{ <variable_declaration> ';' }].

variable_declaration ::= <identifier_list> ';' <type_denoter>.

statement_part ::= compound_statement.

compound_statement ::= 'begin' <statement_sequence> 'end'.

statement_sequence ::= <statement> { ';' <statement> }.

statement ::= [<label> ':'] (<simple_statement> | <structured_statement>).

simple_statement ::= <empty_statement> | <assignment_statement> |

<procedure_statement> | <goto_statement>.

structured_statement ::= <compound_statement> | <conditional_statement>

| <repetitive_statement> | <with_statement>.

当今在各种书刊中,EBNF中全用、全不用< >都有,终结符号用黑体以取消引号,用*代替{ }也有。关键在于用正则表达式表示,且表达能力EBNF和BNF等价,只是书写简化一些。

2.4.1.3 语法图

80年代许多程序设计语言手册不仅给出EBNF,还给出语法图,语法图和EBNF完全等价,只是更直观一些。

例2-6 与例2-5 Pascal EBNF '程序' 和 '语句' 产生式对应的语法图,如图2-2,图2-3

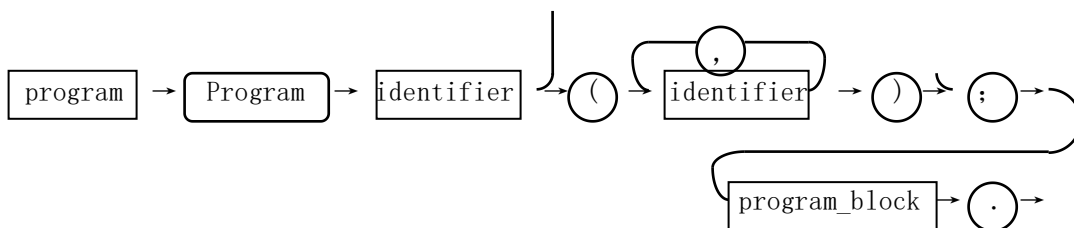


图2-2 Pascal “Program” 语法图

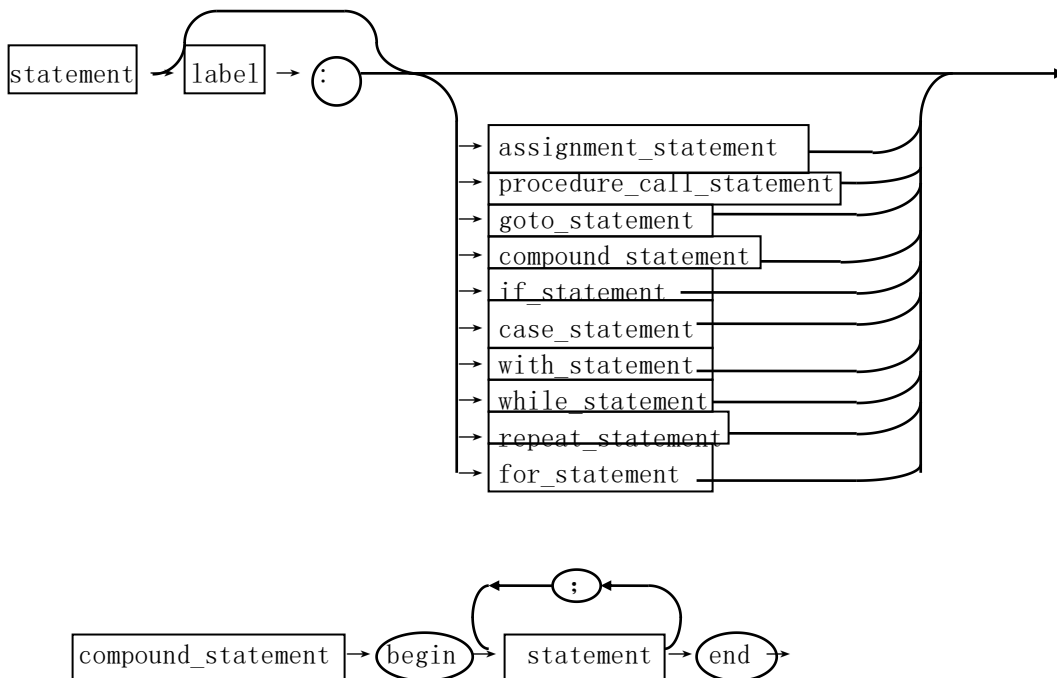


图2-3 Pascal “复合语句” 的语法图

其中方框为非终结符，圆和椭圆形为终结符。箭头指向构造流向。每个非终结符又可开始一个语法图（一条产生式规则）。与EBNF完全对应，[]以‘短路’绕道表示，{ }以迴环表示，小圆弧是有意义的，表示流向。有的语法图在环线上注上数字表示最多转几次。

2.4.1.4 语法分析

语法规格说明定义了该语言程序合法的特征和语句。语言处理器则通过语法分析接受合法的程序，这就叫做程序释义（Parsing a Program），释义过程是产生式生成句子的逆过程。编译器通过释义过程，将读入的源代码逐一分析，看它是否和文法相配。释义过程的输出是文法结构的树型表示，也叫释义树或语法分析树。

语法分析有许多方法，这在编译课程中有详细的论述，不是本书主要内容。总的说来，语法分析的算法可归为两类：“自顶向下”和“由底向上”。“自顶向下”释义则从文法的起始符开始，按可能产生的表达式寻找语句相同的结构匹配。每一步都产生下一个可能的源符号串，找到再往下走。“由底向上”释义则相反，它先查找源代码的各个符号串，看它能否匹配归结为产生式左边的非终结符，如果有含混则向前多读入k个符号串，为此归约下去，一个短语一个短语，最后到达起始符号串，归约的过程就形成了释义树。图2-4是由底向上语法分析的例子。

只要语法是正确的程序，语法分析总是能成功的，如果程序有错误，编译程序就会按它自己的理解给出错误提示。经验表明错误提示可以发现很多近处错误，如果错误源头较远，则给出信息是不太确切的

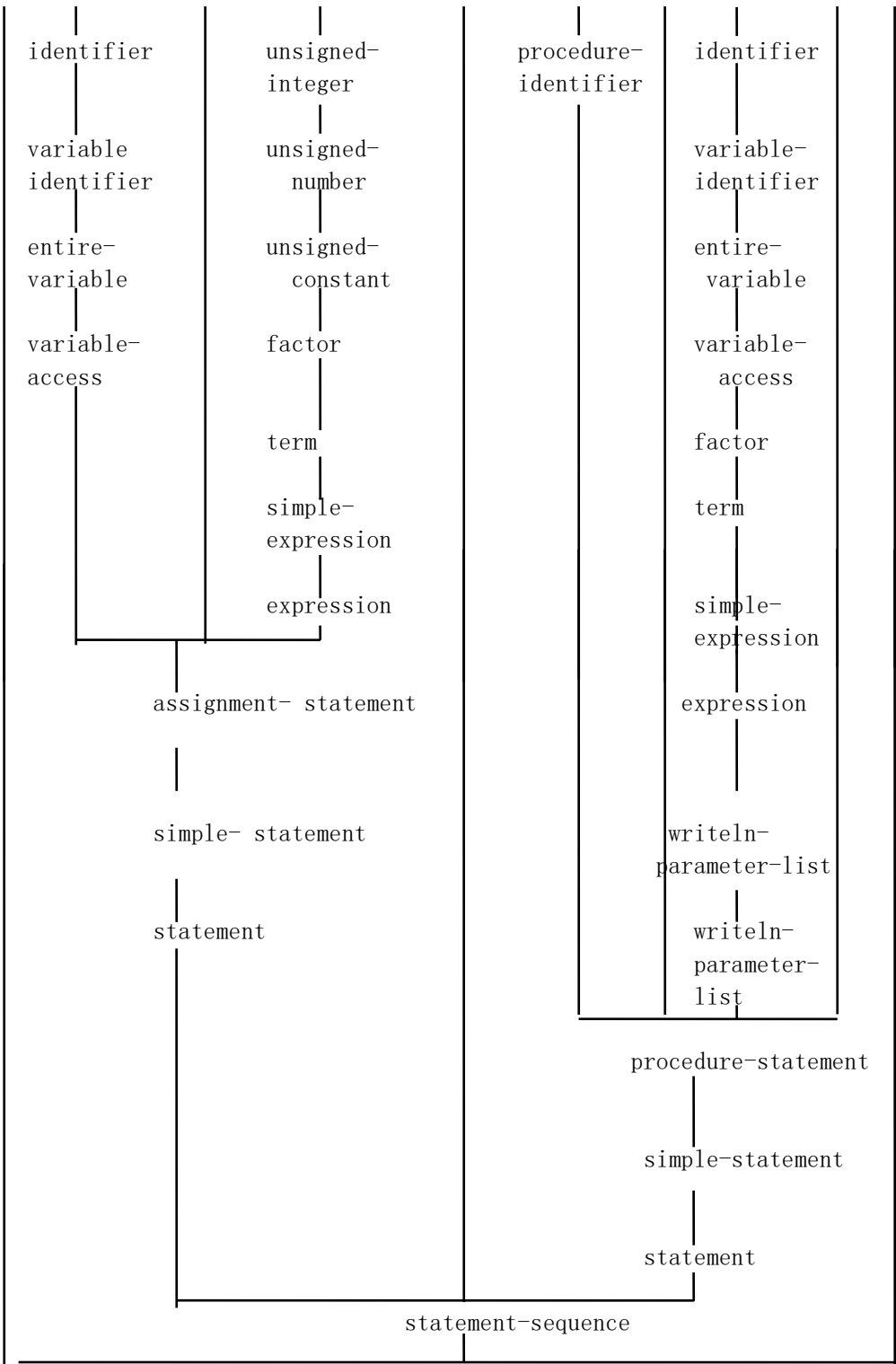


图2-4 由底向上释义过程图解

2.4.2 语义规格说明

程序的语义是它在运行时显现出来的行为。我们怎样说明这种行为呢？这里有好几种方法，每种方法都构成了一类语义规格说明。程序通常要在计算机上执行，因此按照程序的运行步骤或操作来说明程序设计语言是很自然的，这就是所谓的**操作语义学**。大多数程序设计语言的非形式化语义都是用这种方式说明的。例如 Pascal 中 while 命令的操作语义可说明如下：

执行命令 while E do C，其步骤为：

- (1) 对表达式 E 求值，产生一个真假值。
- (2) 如果值为 true，则执行命令 C，然后从 (1) 开始重复。
- (3) 如果值为 false，终止。

操作语义的特点是，强调程序的每一执行步骤。这样能清楚地洞察到程序实现的路径，但在实际中，过分注重细节则难于识别程序执行的净效应。

另一种观点是把程序看成是一个数学函数的实现。命令式语言的程序都可以看作是函数实现，它将程序的输入数据映射为输出数据。函数式语言程序更明显地实现函数。我们就把这个函数理解为程序的意义。这种思想是指称语义学的基础。例如，在 Pascal 语言里的 while 命令，其指称语义可以说明如下：

```
execute [while E do C] =
  let execute-while env sto =
    let truth-value tr=evaluate [E] env sto in
      if tr
      then execute-while env (execute [C] env sto)
      else sto
    in
  execute-while
```

这个语义表示可读作：

执行 while E do C 的语义函数是 execute-while 函数，该函数定义为映射到环境 (env) 和存储 (sto) 上以得到存储的新值。为求出 execute-while 的净效应，先设真值函数 tr，它等于在环境 env 存储 sto 上对 E 求值。如果 tr 为真值，则递归执行 execute-while 的函数，即在 env, sto 下执行 C 从而改变 sto 的值。每次执行时 env 没变 sto 改变了。当 tr 为假时，结果就是 sto 的值。

指称语义学是由牛津大学的 Dana Scott 和 Christopher Strachey (1971) 开发的。这种描述方法可在 Stoy (1977) 和 Schmidt (1986) 的论述中找到。其基本思想是：为每个程序短语指派一数学实体 (指称) 为其意义。典型情况是将其输入映射为输出的函数。例如，一个表达式的指称是将环境和存储映射为值的函数。一个命令的指称是将环境、初始存储映射为最终存储的函数。一个复合命令的指称是它的子命令的复合指称等等。更一般地，任何复合短语的指称都是各子短语指称的组合。

指称语义学也称数学语义学，它不像操作语义学那样涉及操作的细节。实际上是每个语法结构的数学模型，尽管它比较抽象，目前仍然是准‘标准’的语义学。本书我们还要重点讨论它。

除此而外，公理语义和代数语义是近代软件工程和面向对象技术研究经常用到的。

Robert Floyd 于 1967 年提出了可以在程序流程图的弧上加上断言来推测程序的正确性。每一个断言都是与程序变量的当前值有关的逻辑公式，例如：断言 $x < 0$, $y = x + 1$ 是说 x 的当前值非负， y 的当前值比 x 的值大 1。这种方法的目的是在程序的末端产生一个将程序的输入与输出联系起来的断言，Tony Hoare (1969) 将这种思想用于高级语言的控制结构，并指出语言本身的语义可以用这种方式说明。这就是所谓的**公理语义学**。Hoare 和 Wirth (1973) 对 Pascal 的一个大子集进行了公理语义化。Hoare 和 Lauer (1974) 研究了概念操作语义和公理语义的互补 (仍保持一致性)。

公理语义学把语言的公理描述看作是语言的一个理论，该理论由三部分组成：

• 公理集 元语言描述的不加证明的公理集如： $\vdash 0 < \text{succ } 0$ 即自然数0的后继大于0是一个定理(由 \vdash 表示)也是一条公理。

- 语法规则集 以它来确定什么是合式公式。
- 推理规则集 从已确立的定理演绎新定理。

推理规则表示为:

$$\frac{f_1, f_2, \dots, f_n}{f_0} \quad \text{前提} \quad \text{推论}$$

意即, 若 f_1, f_2, \dots, f_n 是定理, 那么 f_0 也是定理。其中 f_1 是形如 PSQ 的公式, 证明后即定理。P, Q为S操作的前后置断言。示例如:

| | |
|----------------------|--------------|
| $x=a$ and $y=b$ | 前置断言P, 证明的前题 |
| $t:=x; x:=x+y; y:=t$ | 操作集S |
| $x=a+b$ and $y=a$ | 后置断言Q, 结论 |

公理语义不像指称语义那样能为每个语法结构给出一个数学指称, 并能得出指称域上的值。它只定义证明规则(公理及推理规则), 以此证明程序的某些性质。这些证明规则在某种意义上就是抽象的语义。

公理语义主要用于程序验证, 语言理解, 语言规范/标准化等方面, 对编译、解释器的没有直接的作用。

同一客观事物, 如果从功能角度抽取模型进而可编制基于功能的软件, 即注重程序的行为。这样的软件最害怕修改。因为修改功能往往是最直接的要求。如果从模型客观世界对象的性状抽取模型, 则可编制基于对象的软件, 即注重数据类型对客观世界的映射。愈是抽象程序则愈经得起修改。因此, 基于抽象数据类型编制的软件有良好的稳定性、重用性、可修改性。抽象数据类型也是对象式程序设计的理论基础。

从数学的观点, 数据类型刻划了某些数据集以及在这些数据集上的操作集, 就是经典的多类代数, 抽象数据类型即泛代数, 它是抽象数据类型的数学模型。从泛代数和范畴论的角度, 一个抽象数据类型对应为项代数的初始代数, 代数规格说明示例如下:

```
specification LISTS
  sorts
    List
    NATURALS
  formal sort Component
Operations
  empty_list : List
  cons(, ) : Component, List → List
  headof_ : List → Component
  tailof_ : List → List
  lengthof_ : List → NATURALSs
variables  c: Component
          l: List
equations
  headof cons (c,l)=c
  tailof cons (c,l)=l
  tailof empty_list = empty_list
  lengthof empty_list = 0
  lengthof cons (c,l)=succ (length of l)
end specification
```

这是抽象表数据类型的代数规格说明, sorts是说明表用到的数据类别, 包括一形式类别 Component。operations部分列出五个函数, 即LISTS的行为(操作)。下横线是占位符表示要用到的参数。后面是参数类别到结果类别的映射。equations部分是约束行为的等式, 即公理

部分。为了表达的普遍性，设变量 c, l 。行为加上约束就是 LISTS的代数语义。

2.4.3 上下文规格说明

至此，我们对程序设计语言的语法和语义的形式方法勾画出了大体轮廓，可从中看出明显的差别。但在实际中，它们的界限是模糊的，例如，静态类型程序设计语言(如Pascal)的表达式 $x/2$ 。若孤立地看，该表达式是合乎格式的，但是，如果 x 的类型是character又会怎样呢？这显然是个错误。但它是语法错误还是语义错误呢？有两种不同的观点：

- 这是一个语义错误。因为有一个‘/’的操作数不是数字，导致对表达式的求值失败。
- 这是一个语法错误。因为不能孤立地看这个表达式而应将它作为程序的一部分。因为 x 的使用与其声明时不一致，整个程序就不能认为是合格的。因此 $x/2$ 的语义就无关紧要了，我们仅关心合格程序的意义。

在术语解释方面也有争议，持第一种观点的人把类型规则看成是静态语义，程序中的语义行为在编译时就能预断；持第二种观点的人将类型规则看成是上下文敏感的语法，一个合格的短语的语法要受它出现的上下文环境的影响。

在本书中，我们采取的是折衷观点，使用的术语是上下文约束(contextual constraints)，它与语法和语义都有区别。

上下文约束可定义为限定程序短语为良构的规则，例如：

- 标识符只能在声明它的块中使用。
- 赋值语句左右两侧必须有相同类型。
- 在while $\langle \text{exp} \rangle$ do $\langle \text{statements} \rangle$ 中 exp 必须是布尔表达式。

这样，就简化了语法和语义的形式化。如前所述，大多数程序设计语言并非上下文无关文法能全部描述的。对个别上下文敏感部分(如类型规则)则在上下文规格说明中予以说明。

2.5 小结

• 本章首先介绍程序语言中非常重要的概念：表示与抽象，即同一事物可在不同的抽象层表示它。程序语言的实现实质是不同抽象层次的等价变换。继而讨论抽象的内涵与外延问题。

• 从抽象层次的角度可以把计算机解题分为四个世界。用户看到的程序设计语言是程序世界使用程序对象，实现者要涉及机器世界，要处理存储对象。

• 即使只限定程序世界，对象表示有显式也有隐式的。

• 本章讨论了语言设计目标。它们是模型能力、语义清晰、实用性、方便性、简单性、可读性、可移植、效率高、适当的灵活性等九个方面，只能定性举例比较难于量化。

• 为了实现以上九个目标应遵循的设计准则：频度、结构一致、局部性、词法内聚、语法一致、安全性、正交性/正则性、数据隐藏、抽象/修饰表达、移植性等。

• 本章从表示的角度介绍了程序设计语言的规格说明。它们的术语、基本概念。

• 语法规格说明一节介绍上下文无关文法的基本概念与表示。介绍了形式语言理论中按文法的产生式划分的Chomsky四种文法和四种语言。指出程序语言规格说明中一般采用上下文无关文法(包含正则文法)。

• 上下文无关文法的元语表示是BNF或EBNF范式，EBNF由于纳入正则表达式表示法比BNF简炼但表达能力一样。EBNF和语法图完全一致对应，只是语法图更直观，语法定义中一般两者都给出。

• 本章介绍了当前广泛研究、使用的形式语义。虽然定义程序设计语言文本时并不是必

须的。形式语义描述程序执行中的行为。本节给出了每种语义学如何描述程序行为的初步概念。

- 操作语义将每个操作抽象并形式表达它的操作后果。注重细节、表达复杂，非行家一时难看出程序净效应。但在重要的涉及硬件、控制的应用中还有用到它。

- 指称语义把程序看作是一个数学函数的实现。用语义函数在语义域上如何取值表征语法特征的行为，是较为抽象的表示，这样只关心程序净效果。是当前语义学研究中的‘标准’语义。

- 公理语义并不给语法特征以确切的数学映射值，它只定义公理及推理规则以此证明程序的某些性质。这些证明规则就是抽象语义。

- 代数规格说明用以表达抽象数据类型，它以代数方法处理满足给定逻辑系统的各种模型，即可模型程序的各种代数结构。刻划了语义范畴。

- 上下文说明是使形式语法和语义不致过于复杂的折衷。以自然语言给出相应约束。以便语言处理器的实现者遵照。

习题

2.1 举出日常生活中的三个例子，说明高层抽象内涵减小，外延增大。

2.2 计算机解题涉及四个世界的对象及其相互运动(作用)。程序设计语言要涉及几个世界对象？为什么？

答：程序设计语言的设计要受到问题世界和模型世界的影响，一旦它定型，它只涉及程序、机器两个世界对象，这是因为程序设计语言的只是实现软件的描述工具并自动转到机器中运行。

2.3 举例说明什么时候分散表示好，什么时候抽象(集中)表示好。

2.4 指出以下说法的正误

(a) 程序设计语言越简短越好。

错，有时十分方便的使用未必简便。

(b) 复杂的语言可读性都差。

错，例如Ada虽复杂，但可读性也很好。

(c) 已有程序设计语言(Pascal, C, Ada, Lisp, Prolog等)早已实用不存在二义性。

错, Pascal还存在。

(d) 高级语言由于硬件速度提高快不用追求效率。

错, 效率是高级程序语言必须考虑的一个问题。

(e) 当今语言没有一个在语言的层次上就能保证可移植。

错，很多语言都有力图脱离机器。Ada、C等。

(f) 有副作用的函数百害无一利。

错，C正是得益于这种副作用。

2.5 将以下BNF表示的Algol60部分产生式画成语法图

```
<unsigned integer> :: = <digit>
                        | <unsigned integer> <digit>
```

```
<integer> :: = +<unsigned integer>
              | -<unsigned integer>
              | <unsigned integer>
```

```
<decimal fraction> :: = . <unsigned integer>
```

```
<exponent part> :: = 10<integer> //10为下标。
```

```

<decimal number> : : = <unsigned integer>
                  | <decimal fraction>
                  | <unsigned integer> <decimal fraction>
<unsigned number> : : = <decimal number>
                  | <exponent part>
                  | <decimal number> <exponent part>

<number> : : = +<unsigned number>
          | -<unsigned number>
          | <unsigned number>

```

2.6 下面的Pascal程序有错误，识别每一个错误并将它们按语法错、语义错、上下文约束错归类：

```

program p;
  a: array 10 of char;
  b: Integer;
begin
  a [0]:=b;
  c:='*'
end.

```

2.7 当前的程序设计语言说明得都很严格，其编译程序也同样严格。程序不能有任何语法错误，而我们人类在讲话时对语法错误不在意，我们可以猜测讲话人的真正意思。设想一个容错编译程序，一旦发现语法错误它能猜测程序员原来想写的是什么内容，这种编译程序能实现吗？(提示：研究PL/1的历史)

2.8 设想一个自身就能适应新程序结构的智能化编译程序。该编译程序并不简单地拒绝使用新的程序结构，而是请求程序员进行解释，以便进行学习。这种编译程序能实现吗？

2.9s 在下面几种情况下，你能举出例子说明哪些二义性是人们希望的，哪些是可容许的？哪些是不希望的？

- (a) 人们谈话中的二义性。
- (b) 某本书或文章中的二义性。
- (c) 交互式查询语言中的二义性。
- (d) 程序设计语言中的二义性。

2.10 假设下面的文法是为一个简单的函数语言设计的，试证明它有二义性：

```

Expr ::= pExpr | Expr pExpr | fn Id. Expr
pExpr ::= Id | Lit | (Expr)

```

第3章 值与类型

数据是计算机加工的对象，无论是什么计算机程序不外乎希望经过计算得到我们需要的数据值或者计算机的一组动作（一般都包括数据值的改变）。值是对事物性态的表征或度量。没有值就没有计算。对人类最有价值的东西是数据值，如电话号码簿、人口普查数据、卫星采集地球数据、金融数据、军事情报数据、气象天文数据……计算机科学把数据研究看作是重要课题就不足为奇了。

本章讨论程序世界的值与类型。因为和客观世界值一样它总是分类的。程序中的值寓于常量、变量之中。变量由于有时、空特性因而引起一系列特点：左值、右值、当前值、赋值，以及名、值分离的引用等。过程式语言特征无不与此相关。接着介绍值应是头等对象概念。这与类型完整性原则是一致的。

本章第二部分是讨论一般语言内定义给出的基本类型、复合类型和递归类型，以及它们的数学表示。并初步引入类型系统概念，以及有关类型的若干术语、准则。

本章第三部分是讨论与求值密切相关的表达式。从表示法，类别到求值顺序的优先级和结合性，以及混合计算时类型的兼容性。本节的论述力求不陷于具体语言，但仍以具体语言为说明的例子。

第四部分给出小结。

3.1 值

值是对事物性态的表征和度量，而表征和度量总是在某个论域的某个抽象层次上，讨论问题的抽象层次不同取值也不同。例如，与一个人的“年岁”相关的值：

医疗档案中取值：{ 成人，儿童，婴儿 }

接班人会议记录中：{ 老年，中年，青年 }

户口登记簿：{ 1..100 }

从操纵值的角度看值是对数据指称的语义内含的表征。‘年岁’是有关一个人多么大的数据指称。它可以在语义域 {1..100} 或 {老年，中年，青年} 或 {成人，儿童，婴儿} 中取一个值。数在度量上有其准确性，但如上例示，值不一定是数值。我们把数据的指称叫做数据的‘名字’，其语义内容的表征为‘值’。每个数据都有其所在论域上的名、值。

然而，每个表征又可以用更详细的语义表征表示它，例如：成人：{老年、中年、青年} 或 {18..100}。再如，青年：{15..25}。这样上层抽象的值是下层抽象的名。那么‘最’下层的值有没有名呢？从表示法约定有最下层。例如在数学系统、计算机程序世界整数123、实数123.45就是最下层了。它的名就是数字‘123’序列，是它的指称！值就是约定的语义解释“一百贰拾叁”。如果约定不同，名为‘123’的值是‘83(八进制)’。我们把一个符号的直接语义解释叫原子值（即在该论域层不可分为更细的解释了）。‘原子’事实上是相对的，进入到机器世界123更详细的表征是0000000001111011，‘位’（bit）的语义才是‘最’原子的。

值总是存在于约定的表示之中。

3.1.1 值与类型

客观事物是多种多样的，度量它们的值，就应分门别类。

程序世界的值一方面根据映射客观事物的需要，一方面根据计算机实现的可能也有不同类别。分类的准则是表示结构的一致（同样语法），同样的语义内涵，以及容许有同样的运算，这就是类型。每一个值都属于一个类型。一般说来，同类型的值相互运算结果仍为该类型的值。计算机中最基本的类型是：

| | |
|--------|--------------------------------------|
| 整型 | {..., -2, -1, 0, 1, 2, ...} |
| └ 定点类型 | {... -1.0, ..., 0, 1.0, ...} |
| 实型 | |
| └ 浮点类型 | {... -. dddE-dd, ..., . dddEdd, ...} |
| 字符型 | {'A', ..., 'Z', 'a', ..., 'z', ' '} |
| 真值型 | {TRUE, FALSE} |
| 枚举型 | {枚举符号集} |

以简单的基本类型可以构造结构类型：

元组 同/不同类型值的集合
 数组 可索引的同类型值的集合
 记录/结构 有标记不同类型值的集合
 表 可递归操作的同类型值的集合

还可进一步组合成更复杂的复合类型：

串 可作串操作的字符序列。

其他专用或更复杂的类型。如整型用于指针操作的指针类型。用户定义的复合类型等。

同一数据类型的外延构成域(domain)，计算机中一般约定有限域。

3.1.2 字面量、变量与常量

计算机中只能用名字操纵值。最直接的名字是字面量、变量、常量，即值的称谓。

字面量也称直接量，约定它的表示(名)就是它的值，且从它的表示可以得知其类型，如123, 1.23e10, 's', "stock", TRUE, dec, 它们是整、实、字符、串、真值(布尔)、枚举型的字面量。前四个比较清楚，后两个是符号串容易和其他类型名字混淆，所以对无保留字概念的FORTRAN用.TRUE.表示第五个，其他语言则约定写法，如黑体、大写、斜体、大小写等。因为TRUE、FALSE一共只有两个值，不难判定。第六个dec也是约定字体(如斜体)，但不查看声明和上下文是难以判断的。

变量是代表任何值的标识符。一旦声明该标识符是什么类型，它在程序中只可取得该类型的某个值。变量只是一个数据的名字，它的值随时可变。所以有初始值(程序开始运行时的值)，当前值(引用该变量瞬间的值)和赋值(强行改变该变量的值)等概念。它所遵循的运算规则与所在类型的值相同。

常量也可以是标识符。一旦某标识符声明为某类型的常量，则必须给它赋初值，且在程序中不得改变，它就成了某个值的代名词。如C中：

```
const int con= 3176;
const int errcon;           //不给初值是错误
.....
errcon = 4231;              //运行时赋值是错误
con=4403;                   //运行时赋值是错误
.....
```

字面量也是常量只是不能再给它取名字。

计算机中名、值分离正好对应为存储单元的地址和存储的内容。名字在编译后消失只

留下地址码。而地址码若作为一个值放在另一个地址码的存储单元中，该存储单元就是指针。它的地址码就是指针名。

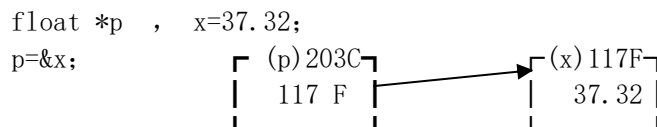


图3-1 指针存储示意图

名、值可显式分离导致程序变量和数学变量有不同的涵义，如 $x=x+1$ 在数学中是不可理解的，因为数学中没有时、空概念。 x 虽然可以取该类型的任何(变量的含义)值，一旦有了值，所有上下文中的 x 都一样。在数学表达式中它是作为某个暂时未知的确定值在作符号运算。

程序变量在程序中出现可以没有值(声明后没赋初值)；可以有**右值**，即变量名出现在命令式语言赋值号的右边，此时是对它存储内容的引用，常规的值；也可以有**左值**，即变量名出现在赋值号的左边，此时是对它存储单元地址或大对象的首地址的引用。所以变量不是具体的值，变量引用才是值，还要看在什么地方引用，它可以是风牛马不相及的左值和右值。

左值也可出现在赋值号的右边，如 C 语言中：

```
y = x ++;
```

先用 x 的右值赋值，再为 x 的左值增值。

近代语言如C++专为操纵左值增加了引用类型，如：

```
int &Rint = Svar //声明一个引用变量并赋初值
```

Rint的类型是引用整型的类型。声明时一定要赋初值(把Svar的地址给Rint)。它相当于指向Svar的常量指针。也是Svar的别名。

3.1.3 程序中的求值方式

运行程序的目的是为了改变或得到我们需要的值，即将输入值“变成”输出值。我们这里说的值是广义的，它包含输出一组计算机动作。这一组动作过程的指称就是过程调用或函数调用(既输出函数值也输出动作值)。近代语言为了概念清晰，主张函数不输出动作只返回值，所以和变量一样叫函数引用，这样，程序中求值就可以有两种方式。

(1) 在表达式中求值

‘表达式’是运算对象和运算符组成的序列，执行表达式的结果得到值，如：

```
(a+b)/(sin(x)-1.0)-hope[i]
```

| | | | |——成分元素引用
 | | | | |
 | | | | | 字面量
 | | | | |
 | | | | | 函数引用
 变量引用

表面上看，都是符号表示的算术表达式。和数学中代数表达式完全一样，但意义不同。程序中表达式在计算‘当时’可以没值，当应当有值。所以，没有初始化或事先赋值的运算对象进入表达式是程序错误(代数中是这样的吗?)。

(2) 通过函数引用、过程调用改变值

和数学一样，一个函数只要将自变量代入值，则可求出函数值：

```
sin (pi/6.0)=0.500000 //函数引用
float sin (x:float) //函数型构
{ //函数体
    .....
    return; //函数定义
}
```

程序中自变量称为变元(argument)，从参数意义上叫它实参，实参与形参结合，返回计算结果值，如果把从实参进入到返回之间的一步运算(在函数体内)看作黑箱，则此种求值方式就是从输入值到输出值的函数映射。可记为 $I \xrightarrow{f} O$ 。过程调用是从输入值到输出一组动作的映射。

有了函数映射的概念，广义来说，表达式中求值也是映射：两个(或一个)操作数按运算符约定的规则“变”成了结果值。这个“变”术语上是映射。运算符即指定的函数映射。表达式是多个连续的映射。为此又定出上层规则：自左至右，自右至左的结合规则，算符优先，括号优先等等。所以语义学中“函数映射”成为最重要的基础。运算符既是函数名，则可把运算符统一为函数表示：

把运算符统一为函数表示，形式地：

单目运算： $\oplus E$ 等价表示为 $\oplus(E)$

双目运算： $E1 \oplus E2$ 等价表示为 $\oplus(E1, E2)$

\oplus 表示‘某’运算符，沿用函数的前缀表示法，E，E1，E2是变元，也是操作数或子表达式。所以，传统的表达式：

$a*b+c/d$

均可等价复合函数调用：

$+(*(a,b), /(c,d))$

从上层抽象的值是下层抽象的名的观点，在高阶函数系统中函数名(抽象)也是值，如sin、cos、fun、sum。

3.1.4 值应是头等程序对象

至此，程序中的值包括：

- 字面量(整、实、布尔、字符、枚举、串)。
- 复合量(记录、数组、元组、结构、表、联合、集合、文件)。
- 指针值。
- 变量引用(左值、右值)。
- 函数或过程抽象。

每一语言不全部具有以上种类值。例如，C的结构相当于记录，ML则都有，且有差别，Pascal有集合其他语言没有。ML没有指针等等。

值既然相当于数学中的计算对象，它就应该具有和数学对象一样的权利，如可出现在表达式中求值；可作参数和函数结果值；它可以构成复杂的结构值(如矩阵、向量、张量等)。在程序语言学中我们把相当于数学对象的程序对象叫做头等对象(First Class Object，也可叫作第一类对象、头等公民)，因为它们作为运算对象的权利未受到任何限制。程序对象的权限具体说来是：

- 可作为操作数出现在表达式中求值。
- 可作为单独的存储实体。
- 可作为参数传递到过程或函数。
- 可作为函数返回值。
- 可以构成复杂的数据结构。

但并非所有程序设计语言的对象均具有以上权利。函数抽象(即函数名)在多数过程式语言中是不能作函数返回值的。在Ada中甚至不能作为参数传递，更不能构成复杂的数据结构。只有函数式语言可以。显然，一个函数的输入参数是函数抽象，那么该函数必然是高阶的。若返回值又不能是函数抽象就有点不伦不类了。语义上难以描述。递归函数是特殊的高阶函数，要回避它又不行，所以，现代语言希望把函数抽象、变量引用都作为头等对象。

数组元素(单个)不是可存储体，也不能作为函数返回值。

3.2 类型

类型是值的集合，以及在这个集合上容许的操作集合。一个程序设计语言总要为用户提供：

(1) 一组直接可用的类型（内定义或预定义的基本类型、结构类型、递归复合类型等），以及提供一组用户定义类型的机制。

(2) 一组类型规则，指明类型的性态及相互关系。如不同类型值混合计算结果类型等等。

(3) 一种类型检查机制，保证类型定义和规则成立。因为程序设计语言类型规则与采用检查时刻密切相关。例如编译时刻作类型检查容易实行单态、静态类型，运行时作检查，可实现多态、动态类型等等。

我们把上述三方面统称语言的**类型系统**。类型和类型系统是程序设计语言研究中最复杂且最本质的内容，本书大量篇幅均与此有关。这里先介绍一般程序设计语言内定义的基本类型、复合类型和递归定义的类型，它们的表示和形式描述。

3.2.1 基本类型

类型是值的一种属性，在3.1.1中我们介绍了五种基本类型，各语言中叫法不一（如真值叫布尔值，浮点叫实型），但数学意义是一致的：定义一个值的集合，以及操作集合，基本类型的值集表示法是在名字之下列举该集合：

```
Truth_value= { FALSE, TRUE }
INTEGER = { -maxint,..., -2, -1, 0, 1, 2, ...,maxint }
REAL     = { ..., -1.0, 0, 1.0, ... }
CHARACTER= { 'A',..., 'Z', 'a',..., 'z', ' ' }
```

基本类型的操作集合比较简单，抽象表示是：

```
op1: T → T      //单目
```

```
op2: T × T → T   //双目
```

各基本类型的具体操作留作习题3.6。

按类型的数学意义真值、字符、枚举型的值域是有限集，整型、实型是无限集，由于机器字长限制，程序中的值域是有限集。所以大写的INTEGER，REAL表示它是实现预定义（或内定义）的，所以虽然同为INTEGER，16位机上值域是-32768.. 32767，32位机是-2147483648..2147483647。有的语言还设有signed/unsigned整数，无符号整数域在16和32位机上是0..65535，0..429467295。同一类型名各机器按“自己”的理解，这对移植造成巨大的潜在危险。不仅如此，在物理字长长的机器上收敛的数值计算程序到字长短的机器也可能发散，所以，FORTRAN-77利用一字节8位来刻画程序中字长，用户可选定字节数：

```
INTEGER*4 ICJ, PCK,      //4字节32位， 值域如上
```

```
INTEGER*1 MIN1, SMAL     //1字节8位-128..127
```

Ada更加显式地让用户指定值域和精度（对浮点）：

```
type INT is range -130 .. 130;  //两字节实现
```

```
type REAL is digits 6 range 0.0 .. 1.0E35 //由4字节实现
```

这样，就引出了泛整（实）型、模型数、安全数的概念。对于强类型语言Ada，没有指明值域和给出类型名的类型（如直接写出字面量）称泛类型。对于浮点数，每当指明值域和精度，则实现必须解出相应的字位表达，离散表达的浮点数集即为模型数集定点也类似。然而，为了处理方便，实现解出的字位数都是按一个或半个字节圆整化的，例如程序员指定了一个浮点类型，阶码要5位，尾数要22位则实现解出7位作阶码（加符号位是8位），尾数

24位（4或8的倍数）。这样，表示的精度和值域都增大了。大出的部分虽超过指定的值域仍然是安全的，可通过类型检查，故称安全数。

C语言虽有长整型，短整型，双精浮点等规定，是指定字节的隐含说法，但实现仍有差别，不如Ada。

Pascal还内定义了集合类型和子界类型，是程序语言中仅有的。事实上，集合使用并非最频繁，用枚举型再预定义一个集合操作的程序包即可实现其功能。每当需要用时连接上，不用时不作为语言的一部分，可以减少编译器的大小。至于子界其概念比后文讲的子类型要窄。

除REAL外其余四种都是离散基本类型，离散基本类型是其值和整数(域)有一对一关系的基本类型。在Pascal和Ada中这是很重要的概念。离散基本类型的值可以用于如记数、情况选择、数组索引之类的操作，在早期语言中，只有整数可用于这些操作。

集合(或某类型)中元素的个数叫基数。我们用#S表示S中值的个数，例如：

```
#Truth-Value = 2           (如上述)
#INTEGER = 2 * maxint+1     (在Pascal中)
#INT = 2*130+1              (在上文Ada示例中)
```

3.2.2 复合类型

复合类型(或称结构数据类型)，其值由更简单的值复合或构造而成，程序设计语言支持各种各样的结构数据：元组、记录、变体记录、联合、数组、集合、串、表、树、顺序文件、直接查找文件、关系等等。这么多结构令人眼花缭乱，实际上只要很少几个结构概念就可以实现这些类型，这些概念是：

- 笛卡尔积(元组和记录)
- 不相交的联合(变体记录和联合)
- 映射(数组和函数)
- 幂集(集合)
- 递归类型(动态数据结构)

本小节讨论头四个。递归类型在第3.2.3节中讨论。

本教程介绍的数学表示法是对以上定义的复合集合给出的最简单、标准、适用的表示法。该表示法的表达能力足以描述数据结构的各种变体。

(1) 笛卡儿积

最简单的一类值的复合是笛卡尔积，它是两种(可以是不同的)类型值组成的对偶，我们用表示法 $S \times T$ 代表所有值的对偶集合，其中每一对偶的第一个值取自集合S，第二个值取自集合T，形式地：

$$S \times T = \{ (x, y) \mid x \in S; y \in T \}$$

这可用图3.2 说明：

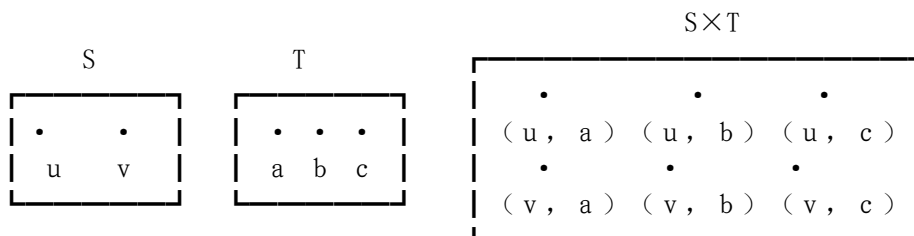


图3.2 两集合的笛卡尔积

对偶上的基本操作只有选择，即选取它的第一成分或第二成分。

笛卡尔积的基数极易推断：

$\#(S \times T) = \#S \times \#T$ //右式里的‘ \times ’是算术乘含义

这就是我们把‘ \times ’号用于笛卡尔积的原因。

可以把笛卡尔积的概念从对偶扩充到三元组，四元组等等。一般地，表示法 $S_1 \times S_2 \dots \times S_n$ 代表所有 n 元组的集合，其中每个 n 元组的第一成分选自 S_1 ，第二个选自 S_2 ，……，如此等等。

ML中的元组，COBOL、Pascal、Ada、ML中的记录，Algol-68和C中的结构，都可以理解为笛卡尔积。

例3.1 以下Pascal记录类型：

```
type Month = (jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec)
type Date = record
    m:Month;
    d: 1..31
end
```

其值的集合为 $Date = (\backslash month / \times \{1..31\})$ 。这些值有下述的372对：

```
(jan, 1)    (jan, 2)    (jan, 3)... (jan, 31)
(feb, 1)    (feb, 2)    (feb, 3)... (feb, 31)
...
(dec, 1)    (dec, 2)    (dec, 3)... (dec, 31)
```

显然Date中的某些值和真实的日期不对应。要真实表示用简单记录颇不容易，读者可以试试。

用标识符 m 和 d 访问记录类型Date的单个成分要加记录名，如：

```
var someday: Date;
...
someday.d:=29; someday.m:=feb;
```

这两个命令分别访问了someday的第二和第一成分。用指明的成分访问，程序员就不必记住各成分的次序了。

ML有直接和笛卡尔积对应的表示法。形如 $T_1 * T_2 \dots * T_n$ 的 n 元组类型，其值的集合即笛卡尔积 $T_1 \times T_2 \dots \times T_n$ 类型， T_i 是类型名。

例3.2 ML的元组类型举例。

```
type person=string * string *int *real
```

其值的集合为 $person=String \times String \times Integer \times Real$ 类型。

类型为person的someone其值可分解如下：

```
val (surname, forename, age, height)=someone
...
if age>18 then ... else ...
```

这样，若要访问元组内某个成分，程序员就必须记住它在元组类型中的位置。元组类型定义并未为使用单个成分提供什么线索。如，我们往往一时记不清person的第一成分是forename还是surname。为此，ML又提供了记录类型：

```
type person = surname : string,
    forename: string,
    age      : int,
    height   : real
```

实际上，这就和Pascal的表示法一样了。

笛卡儿积的一种特殊情况是所有元组成分值均取自同一集合。这种情况的元组我们称它为同构的。写为：

$$S^n = S \times S \times \dots \times S$$

就是所有成分均取自集合 S 的同构 n -元组集合。同构 n -元组的基数是：

$$\#(S^n) = (\#S)^n$$

最后再分析一个非常特殊的情况， $n = 0$ 。由上可知 S^0 的值为1。该值是0-元组 $()$ ，

即什么成分都没有的元组。因此定义成该类型的一个单元值，这种表示很有用：

$\text{Unit} = ()$

Unit在ML中对应为unit，在Algol-68和C中为void。请注意Unit并非空集，它是一个元组，只是它没有成分。

(2) 不相交的联合

另一类复合值是不相交的联合，其值取自两个(通常是不同的)类型。用表示法 $S+T$ 表示这种值的集合，其中每个值都取两个，来自集合S和集合T。为了标明值的来源，我们给各个值加上标签，形式地：

$$S+T = \{ \text{left } x \mid x \in S \cup \text{right } y \mid y \in T \}$$

其中取自S的值标以left，取自T的值标以right。加上标签只是为了区分各值的来源，因为它们要有区分，要不然就可任选了。

不相交的联合可由图3.3说明：

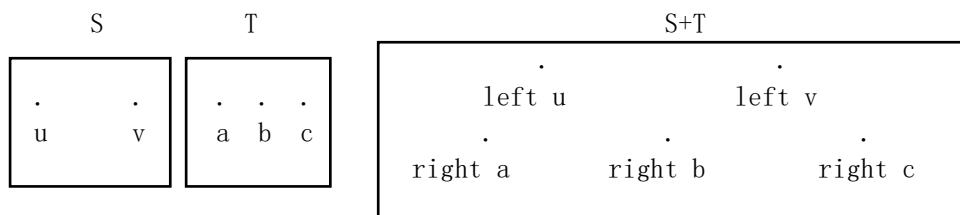


图3.3 两集合的不相交的联合

对 $S+T$ 中值的基本操作是：(a)测试它的标签以判明它是取自S还是T。(b)还原为原有集合值，根据具体情况还原为S或T。例如，若测得值为right b我们就可知其来自T，因而还原为T的b的值。

不相交的联合的基数也容易推断：

$$\#(S+T) = \#S + \#T \quad // \text{这里的‘+’是‘算术加’含义。}$$

这就是采用记号‘+’来表示不相交的联合的原因。

可以把不相交的联合扩充到任意多个集合上。一般地，表示法 $S_1+S_2+\dots+S_n$ 代表一个集合，其中每个值是从集合 S_1, S_2, \dots, S_n 之一选取的。

Pascal和Ada的变体记录，Algol-68的联合，ML中的构造，Mirenda中的代数类型都可以看作是不相交的联合。

Pascal的变体记录可有以下最简单的形式：

```
record
  case I: T of
    L1: (I1: T1);
    ...
    Ln: (In: Tn)
  end
```

其中字面量 L_1, \dots, L_n 是离散基本类型T的值。这个变体记录具有类型为 $T_1+T_2+\dots+T_n$ 的集合值，每个来自 T_i 的值以标记 L_i 值标记。

例3.3考察下述Pascal的变体记录类型：

```
type Accuracy = (exact, approx);
Number = record
  case acc: Accuracy of
    exact: (ival: Integer);
    approx: (rval: Real)
  end
```

该类型值的集合是 $\text{Number} = \text{Integer} + \text{Real}$ ，其值为：

$\{ \dots, \text{exact}(-2), \text{exact}(-1), \text{exact}0, \text{exact}1, \text{exact}2, \dots$
 $\cup \dots, \text{approx}(-1.0), \dots, \text{approx}0.0, \dots, \text{approx}1.0, \dots \}$

这里以枚举值 exact 和 approx 作标签。本例说明，任何离散基本类型的值均可作Pascal的标签。

Pascal里的变体记录的标记和变体记录成分都可以象其他非变体成分一样地访问。这就要冒不安全的风险。设有类型Number的变量num，若num的当前值为7，则 num.acc的值显然是exact，num.ival值为7。若因某种未曾想到的运行错误，程序使num.acc成了approx而当前又没有实数值，则它创建一个未定义值的num.rval，并产生了冲掉num.ival的副作用。num的值一下子就从exact 7变为approx未定义。

ML中的datatype声明，允许引入不相交的联合类型，可由下例说明：

例3.4 分析ML的类型定义：

```
datatype number = exact of int
                | approx of real
```

该类型值的集合是Number=Integer + Real，其值域同例3.3。其中标签的标识符是exact和approx。

ML中带标签的值叫构造。构造由表达式组成，如exact(i+1)和approx(r+3.0)。它们只能由型式匹配分解。若num类型为Number，下述case表达式计算与num表示的数的最接近的整数：

```
case num of
  exact i => i
| approx r => round (r)
```

假定num的值是approx 3.1416，其值要与型式approx r匹配，再按子表达式对r求值，即取整3.1416得3。因为按标签匹配，ML不存在Pascal变体记录的不安全性。

变体记录一般情况下是笛尔儿积和不相交的联合的混合。请分析以下类型定义：

例3.5 Pascal的变体记录类型：

```
type Shape=(point, circle, box);
Figure=record
  x, y: real;
  case figureshape: Shape of
    point : ( );
    circle: (radius:Real);
    box    : (helght, width: Real)
  end
```

该类型值的集合是Figure=Real × Real × (Unit+Real+(Real×Real))

这里有几个值：

```
(1.0, 2.0, point( ))           //点(1.0, 2.0)
(0.0, 0.0, circle(3.0))        //半径3.0, 原点在圆心的圆
(1.5, 2.0, box(3.0,4.0))       //中心在(1.5, 2.0)处的3.0×4.0的方框
```

Figure的每个值都是三元组，每个三元组头两个成分是实数，第三个成分是不相交的联合。后者以值point, circle, box作标签。以point标记的值有点特殊是0-元组(); 以circle标记的值是一个实数；以box标记的值是一对实数。

请注意，不相交的联合和普通的并集不是一回事。标签使我们识别S+T中任何值的来源，一般不是S∪T，除非S和T刚巧不相交。实际上，若有T=a, b, c，则：

$T \cup T = a, b, c = T$

$T + T = \text{left } a, \text{ left } b, \text{ left } c, \text{ right } a, \text{ right } b, \text{ right } c \neq T$

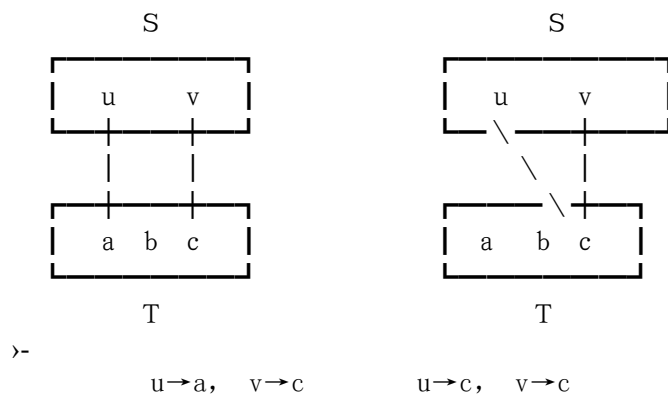
(3)映射

从一个集合到另一个集合的映射(或函数)概念在程序设计语言中是极为重要的，并以多种形态出现。设有一个映射m，它把集合S的每个值x映射为集合T中的值，集合T中的映射值叫做x在m之下的映象，我们约定写为m(x)。我们把：

$m: S \rightarrow T$

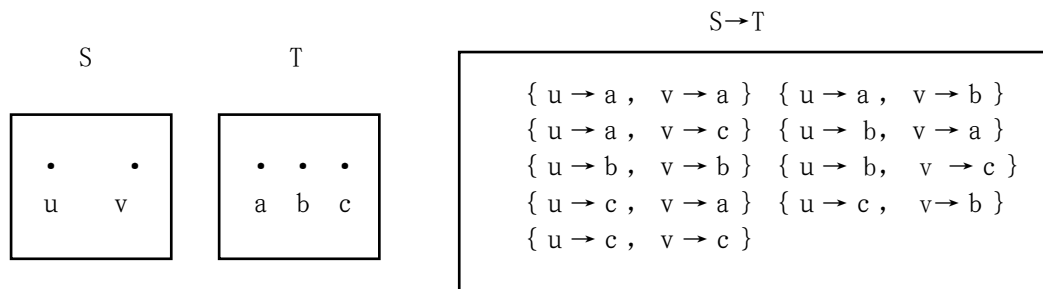
解释为m是从S到T的映射。

图3.4 说明了从S={u, v}到T={a, b, c}的两种不同的映射。符号‘→’读作‘映射为’，于是u→a, v→c表示了u到a和v到c的映射。u→c, v→c表示u, v均映射为c。

图3.4 $S \rightarrow T$ 的两种映射

表示法 $S \rightarrow T$ 代表了这种映射的全集，其图示如图3.5。形式地：

$$S \rightarrow T = \{ m \mid x \in S \Rightarrow m(x) \in T \}$$

图3.5 $S \rightarrow T$ 映射的全集

由图中看出：S中的每个值，在 $S \rightarrow T$ 的映射时可能的映象数是 $\#T$ ，S中有 $\#S$ 个值。因此 $S \rightarrow T$ 的基数是：

$$\#(S \rightarrow T) = (\#T)^{\#S}$$

几乎每种程序设计语言都有数组，数组可以看作是映射，并且是有限映射。即从一个有限集（称数组的索引集合）到某个别的集合（称数组的成分集合）。

多数程序设计语言把数组的索引集合限定为整数域。某些语言甚至固定索引域的下界（典型的为0或1）。Pascal和Ada的数组索引集合可以是任何离散基本类型。形如

array [S] of T

的数组类型，其值为集合 $S \rightarrow T$ 的有限映射。一般S，T写类型名和取值域（范围）均可。

例3.6 分析以下Pascal数组类型的值集

```
type Color=(red, green, blue);
Pixel = array [Color] of 0..1
```

这个数组类型值的集合是 $Pixel = Color \rightarrow \{0, 1\}$ ，其中 $Color = \{red, green, blue\}$ ，Pixel的值为以下八个有限映射：

$\{ red \rightarrow 0, green \rightarrow 0, blue \rightarrow 0 \}$ $\{ red \rightarrow 1, green \rightarrow 0, blue \rightarrow 0 \}$
 $\{ red \rightarrow 0, green \rightarrow 0, blue \rightarrow 1 \}$ $\{ red \rightarrow 1, green \rightarrow 0, blue \rightarrow 1 \}$
 $\{ red \rightarrow 0, green \rightarrow 1, blue \rightarrow 0 \}$ $\{ red \rightarrow 1, green \rightarrow 1, blue \rightarrow 0 \}$
 $\{ red \rightarrow 0, green \rightarrow 1, blue \rightarrow 1 \}$ $\{ red \rightarrow 1, green \rightarrow 1, blue \rightarrow 1 \}$

若P为Pixel类型且c为Color类型变量，则P(c)即可访问P的成分，其索引值为对c的引用。

多数程序设计语言都有多维数组，n维数组成分的访问要用n个索引值。可以把n维数组看作是n元组的简单索引。也可以把多维数组看作是数组元素为多（一）维数组的一维数组，

这样就有着较大的灵活性，如C语言中：

```
char v[7][9][3]
```

对应的数学表示是：

```
Array v = int  $\rightarrow$  (int  $\rightarrow$  (int  $\rightarrow$  char))
```

除数组以外，程序设计语言还有以抽象形式出现的映射。函数抽象以算法的手段实现 $S \rightarrow T$ 的映射，该算法取 S 中某个值(参数)计算它在 T 中的映象(结果值)。集合 S 不一定是有限集。

例3.7 同一Pascal的函数抽象不同的算法实现

```
function even (n: Integer): Boolean;
begin
    even := (n mod 2 = 0)
end
```

此函数抽象实现的映射是 $\text{Integer} \rightarrow \text{Truth_value}$ ，即：

```
{ 0 $\rightarrow$ true,  $\pm 1 \rightarrow$ false,  $\pm 2 \rightarrow$ true,  $\pm 3 \rightarrow$ false, ... }
```

我们可以用另一种算法：

```
function even (n: Integer): Boolean;
begin
    n := abs (n);
    while n > 1 do
        n := n-2;
    even := (n=0)
end
```

此函数抽象实现的和上一个函数抽象同样的映射。

函数抽象和数学中的函数不是一回事，它通过某个算法手段实现了一个函数，算法具有数学函数不能享有的性质(如效率)。更加本质的差别是程序设计语言中的函数抽象可访问(甚至更新)非局部变量中的值。例如，利用函数抽象返回随机数或一天中当前的时刻，这都是常规数学函数办不到的。为此，在程序设计语言中用到术语“函数”时，必须搞清楚我们的意图是数学函数，还是函数抽象。

多数程序设计语言的函数抽象可以有几个参数，调用时为其传递 n 个变元，也可以把它看作是接受了一个变元，该变元是一个 n -元组。

例如以下pascal函数抽象：

```
function power (b: Real; n: Integer): Real;
...
```

实现了 $\text{Real} \times \text{Integer} \rightarrow \text{Real}$ 的映射。它把数对(2.5, 2)映射为6.25，数对(2.0, -2)映射为0.25等等。

虽然数组和函数抽象都可以按映射来理解，但函数抽象参数传递比数组复杂得多，第6章我们还要详细讲述。

(4) 幂集

程序设计语言中集合的概念可以以不同形式出现。考察一个值的集合 S ，我们感兴趣的是 S 的子集的那些值，所有子集的集合则称为 S 的幂集，写为 ΦS ，形式地：

$$\Phi S = \{s \mid s \subseteq S\}$$

集合上的基本操作即集合论中的常规操作：成员测试、蕴含测试、并、交、差、基数测试。

S 中的每个值可以是也可以不是 S 中的某个集合的成员。因此 S 的幂集的基数可由下式给出：

$$\#(\Phi S) = 2^{\#S}$$

Pascal是程序设计语言中绝无仅有地直接支持集合的语言。Pascal集合类型的形式是：

```
set of T
```

它有 ΦT 中各集合的值。所有的基本集合操作Pascal都提供。

例3.8 Pascal的集合类型定义示例

```
type Color = {red , green, blue};
Hue = set of Color
```

该集合类型的值集是 $Hue=\Phi Color$ ，即所有 $Color=\{red, green, blue\}$ 的子集的集合，有以下八个集合：

```
{ } {red} {green} {red, green}
{blue} {red, blue} {green, blue} {red, green, blue}
```

Pascal只容许初等量的集合(即真值集合、字符集合、枚举集合、整数集合)。由于有了这样的限制，集合实现的效率非常高。如果允许复合值的集合(如，字符串集合、记录集合)，许多程序设计问题就可迎刃而解，只有少数复杂的程序设计语言支持这种概括性的集合。

要小心不要弄混了 ΦS ，它是值的集合，值本身又是集合。 $S \times T$ ， $S+T$ ， $S \rightarrow T$ 和 ΦS 每个都是值的集合，头三种情况其值分别是对偶、联合和映射，第四种情况 ΦS ，其值本身又是集合。

3.2.3 递归类型

(1) 表

表是值的序列，一个表可以有任意多个元素，包括空。元素个数称为表的长度。没有元素的特殊表称为空表。

若表的所有元素都是同样类型则称同构的，我们只考察同构表。

假定我们希望定义一个类型，其值为整数表。我们可以这样定义表：其值要么为空，要么是一个整数(它的头)和另一个整数表(它的尾)的对偶。这个定义是递归的，则表的集合可描述为：

$$Integer_List = Unit + (Integer \times Integer_List)$$

或换种写法：

$$Integer_List = \{nil() \} \cup \{cons(i, L) \mid i \in Integer, L \in Integer_List\}$$

式中我们选用 nil 和 $cons$ 作为标记，为简化起见，把 $nil()$ 缩略为 nil ，意思是空表， $cons$ 意思是将 i 和 L 组成一个表，是一个谓词。这些等式确切的意义是什么呢？分析以下值的集合：

```
{nil}
∪ {cons(i, nil) | i ∈ Integer}
∪ {cons(i, cons(j, nil)) | i, j ∈ Integer}
∪ {cons(i, cons(j, cons(k, nil))) | i, j, k ∈ Integer}
∪ ...
```

即以下集合：

$$\{cons(i_1, cons(\dots, cons(i_n, nil) \dots)) \mid n \geq 0; i_1, \dots, i_n \in Integer\}$$

它是所有有限整数表的集合。极易看出，另一个解是无限整数表的集合，即 $n \rightarrow \infty$ 是有限 $Integer$ 集的超集。我们暂不研究它，因为我们感兴趣的是可计算的值。而无限表在有限的时间内是无法算(完)的。

将上述等式更概括一些，递归集合方程是：

$$L = Unit + (S \times L)$$

对 L 有一个最小解，它对应于所有从 S 中取值的有限表集合。所有其他解都是这个最小解的超集。

表(或序列)当我们引入表示法 S^* 后则更加有用， S^* 是由 S 中取值的所有有限表的集合，于是：

$$S^* = Unit + (S \times S^*)$$

在Pascal和Ada中递归类型必须以指针来定义。现代函数式语言，如ML，可直接定义

递归类型。

例3.9 ML声明定义的整数表类型：

```
datatype intlist = nil
                  | cons of int * intlist
```

请注意短语 ‘nil | cons of ...’ 是 ‘nil of unit | cons of ...’ 的简写。

按此定义，intlist类型的值可如下示：

```
nil
cons (1, nil)
cons (2, cons (3, cons (5, cons (7, nil))))
```

实际上，ML具有完整的表类型族，如 ‘int list’ (其值为整数表)，‘bool list’ (其值为真值表)。
‘int list list’ (其值为整数表的表)。list是它预定义的类型构造函数。

对表的基本操作是测试是否为空，选取(非空)表的头或尾。在这些预定义操作之上可定义更多的操作，如表的连接和表长的测试(它们可由基本操作作出显式定义)。

(2) 用户定义的递归类型

递归类型是其值由该类型值复合的类型，递归类型由其自身定义。上述表类型即递归类型的一种，用户还可以定义许多种递归类型。

一般地，递归类型值的集合T，由形如：

$$T = \dots T \dots$$

的递归集合方程式表达，一个递归集合方程式可以有多个解。幸而每个递归集合方程式总是有个最小解。这个解是其他各个解的子集。就计算而言，我们最感兴趣的就是这个最小解。

上述方程式的最小解可按下述步骤迭代求得，以空集置换右边的T，这就得到T的第一次近似值。然后以第一次近似值再置换右边的T，这就得到T的第二次也是较好的近似值。连续做下去。每一步都是拿前次近似值置换右边的T。连续的近似值愈来愈接近最小解。

递归类型的基数是无穷大。前述整数表的集合为无限大，尽管每一单个的表都是有限的。因此不能枚举递归类型的所有值，我们只能通过足够多的连续迭代接近枚举。

例3.10 设用ML声明定义的二叉树类型

递归类型inttree，其叶子值为整数的二叉树：

```
datatype inttree = leaf of int
                  | branch of inttree * inttree
```

以ML表示法表达的某些inttree类型的值如下：

```
leaf 11
branch (leaf 11, leaf 5)
branch (branch (branch (leaf 5, leaf 7), leaf 9), branch (leaf 12, leaf
18))
```

这个递归集合方程的类型定义：

$$\text{Integer_Tree} = \text{Integer} + (\text{Integer_Tree} \times \text{Integer_Tree})$$

我们先以空集置换其中的Integer_Tree，得到第一次近似：

$$\{\text{leaf } i \mid i \in \text{Integer}\}$$

它实际上是深度为1的所有二叉树的集合。

再把第一次近似值置换右边的Integer_Tree，得到第二次近似：

$$\begin{aligned} &\{\text{leaf } i \mid i \in \text{Integer}\} \\ &\cup \{\text{branch}(\text{leaf } i, \text{leaf } j) \mid i, j \in \text{Integer}\} \end{aligned}$$

它实际上是深度最大为2的所有二叉树的集合。再把第二次近似值置换该式右边的Integer_Tree，可得：

$$\begin{aligned} &\{\text{leaf } i \mid i \in \text{Integer}\} \\ &\cup \{\text{branch}(\text{leaf } i, \text{leaf } j) \mid i, j \in \text{Integer}\} \\ &\cup \{\text{branch}(\text{leaf } i, \text{branch}(\text{leaf } j, \text{leaf } k)) \mid i, j, k \in \text{Integer}\} \\ &\cup \{\text{branch}(\text{branch}(\text{leaf } i, \text{leaf } j), \text{leaf } k) \mid i, j, k \in \text{Integer}\} \end{aligned}$$

$$\cup \{ \text{branch}(\text{branch}(\text{leaf } i, \text{leaf } j), \text{branch}(\text{leaf } k, \text{leaf } l)) \mid i, j, k, l \in \text{Integer} \}$$

它实际上是深度最大为3的所有二叉树的集合，连续做下去，就得到任意有限深度的二叉树的集合。

(3) 串

串是字符的序列，所有现代程序设计语言都支持串，但串应归到哪个类型，类型语言的设计者看法不一。问题在于(a)串是初等量还是复合值？(b)应该提供什么样的串操作？典型的串操作是相等测试、串连接、选定字符或子串、按字典排序。

定义串的一种办法是把它看作基本类型，其值为任意长度的串。ML就采用这个办法。因此，ML的串操作(如相等测试、串连接、分解为单个字符等)全部都是内定义的，用户无法再定义它们。

定义串的另一个办法是把它看作是字符数组，Pascal 和Ada就采用这个办法，所有数组上可用的操作均自动地用于串上。特别是用数组索引就可选定串上的任何字符。按Pascal/Ada的办法，其结果是给出的串变量(类似于数组变量)只能包含定长的串。Algo168就避免了这个缺点，它把串定义为可变长的字符数组，这样给出的串变量就包含了任意长度的串(关于数组变量的这个性质，第9.1.4(2)节中还要讨论)。如果是字符数组，一般不提供数组上的串的操作，如连接和按字典排序，只能由使用者写出专用的操作过程。

把串定义为字符的表可能是最自然的方式。Miranda和Prolog就采用这个办法，所有表上可用的操作均自动地用于串，特别是选定串上的第一字符，但选第n个字符却不能。串上特有的操作，如按字典排序和选定子串都必须在一般表操作的基础上予以增补。

3.2.4 类型系统初步

如前所述同我们把类型定义、类型规则、类型检测统称类型系统，但如何定义，给出什么规则，还得讨论与类型本身有关系的几个问题。

(1) 静态类型和动态类型

为确保能防止无意义的操作，语言的实现必须提供对操作的事先类型检查。例如，在一个乘法执行之前要检查两个操作数，保证它们是同类型或可兼容类型的数。同样，在一个逻辑运算执行之前要检查两个操作数，保证它们能得出真值。

在复合数据的情况下，我们要同时考虑到复合的形式和各单个成分的类型。例如，记录成分的选定和数组中的索引是不同的。类型检查必须保证某种选择操作适用于复合值的类型。

尽管在操作之前类型检查必须进行，但在类型检查的时间上还有相当大的自由度，可以在编译时刻进行也可以在运行时刻进行。这个貌似语用的问题实际上是把程序设计语言分为静态类型语言和动态类型语言的重要依据。

在静态类型语言中，每个变量和参数都有程序员指定的类型。这样，每个表达式的类型都可演绎得到，且在编译时刻对每个操作数作类型检查。多数高级语言都是静态类型。

在动态类型语言中，仅值有固定类型。变量和参数一般不指明类型，但它在不同的时刻可以取不同的类型的值。这就隐含着在运行时刻执行一个操作之前要对操作数作即席的类型检查，Lisp和Smalltalk就是动态类型语言的例子。

例3.11 考察下述两个Pascal的函数定义：

```
function even (n: Integer): Boolean;
begin
    even := (n mod 2 = 0)
end
```

尽管我们并不知道n将取什么具体的值，但可以断定该值只能是整数，因为n就是这样声明的。由此断言可推演出‘mod’的两个操作数都是整数，且其结果值也是整数。接下来，可推演‘=’的两端操作数也是整数。最后，可推演出赋给even上的值是真值，该值与声明的结果类型Boolean一致。

接着，再看一个类似的函数定义，这次假定为动态类型语言：

```
function even ( n);
begin
    even := n mod 2 = 0
end
```

这里n值的类型事先很容易通过编译，所以运行时只能根据mod两端操作数一致作类型检查。如果左操作数是一个整数，则表达式正常执行。再检查‘=’两端类型，本例为整数0，也可执行，并得到真值类型，本例正常结束。如用even (true)调用则运行时可查出错误。它使程序异常简洁。动态类型有时非常有用；请看下例：

例3. 12 动态类型的简洁性

```
procedure readliteral (var item);
begin
    读一无空白的串
    if 串构成一整数字面量then
        item: = 该串的数值值
    else
        item: =该串自身
    end
```

动态类型因隐含有运行时刻类型检查，它将使程序执行变慢，还隐含着要为每个值作标记以识别它的类型。静态类型语言就免去了这些时间上的开销，因为所有的类型检查在编译时刻就做过了。

(2) 弱类型和强类型

早期语言的类型系统主要是为分配存储、存储访问、控制预定义函数的应用而设，只要数据格式兼容就可以兼容，不象近代语言类型系统涉及语义学方面。例如，在程序世界，字符和整数混合运算就没有什么实际的意义，然而在机器世界，实现单板机的控制应用时，这种灵活性非常有用。随着程序编大和语义复杂对类型系统要求更高，希望不发生类型错误，有了错误也能查出。纵观历史，程序设计语言的类型系统逐步增强，大致分为以下五类：

- 无类型语言 只有值有类型变量无类型，在程序执行期间用到什么值就是什么类型，它只能是动态类型系统，多用于交互式，早期LISP, Smalltalk之类语言也采用此方案。

- 弱类型语言 变量有类型，但只要数据格式相同类型即可兼容，例如早期FORTRAN没有字符类型，字符值可放在任何其他类型变量中编译运行均不作检查，程序员负责。

- 强制类型语言 变量有类型，并作类型检查，但类型兼容灵活。这是从早期语言赋值语句规则学来的。如表达式结果是实数向整变量赋值，则自动截尾圆整，所用到的操作指令编译时自动加上，程序员不用问。这叫隐式的类型强制，PL/1, C, Ada把插入代码的要求放给程序员，程序员可在程序中用显式类型强制表示，如：

```
int a, b
float c, d
a= float(b)+d;
```

式中b的转换是显式的，赋值a是隐式强制。

强制在多数语言中均有，然而PL/1 为了既灵活又争取效率(时、空)设计了多达16种隐式转换规则。程序员不堪其苦，所以显式强制才普及。

- 伪强类型语言 类型规则订得愈死，检查越认真，程序出错后自动排错才容易。因此，软件工程要求强类型，即每个操作数在计算之前均被检查。由于动态类型检查时空开销较大，强类型检查放在编译时，由于程序编译一次，可运行十数百次，静态检查开销

再大也是值得的，所以叫静态强类型。Pascal就是按照这种模式设计的。不幸的是类型系统设计不够严密，如类型等价准则(见下文)不严、类型域过宽等使得许多潜藏错误仍然发生，所以叫伪强类型。

- 强类型语言 它的特点是类型多，只要客观世界的值有区分，就可以定义出某种类型，如年龄取正整数值一般0..150就够了，就把它定为年龄类型。它虽然是整数类型的子类型，按强类型的观点它是新类型。不能自动转成其它整类型，要显式强制，好处就是若年龄变量运算时得出2531的值，运行支持程序就报错。因为谁也不会有2531岁，超出年龄类型域！所以，一个程序光整型就可以定义几十上百个。第二个特点是检查严，每个运算对象都要通过类型检查，不能在编译时查出就在编译时插入检查代码。早期的强类型一般指静态强类型。Ada语言就是按此设计的。80年代中后期发现强调安全增强类型检查，不利于软件继承、重用。类型多态是个出路，但不是无类型的多态类型，而是动态强类型，即类型强化在执行时动态束定之后进行。当然，这要建立在90年代机器速度、容量大为提高的前提下。C++在往这个方向发展，但目前尚无动态强类型语言。

(3) 类型等价

数学中整数是实数的子集，所以它们相同的值，如2和2.0是等价的。然而，我们知道在程序中它们是不等价的，因为实现它们有的结构不一样。所以，程序中最早是按结构等价判定：

结构等价可以定义为：类型 $T = T'$ 当且仅当 T 和 T' 可存储相同的值的集合。

所谓结构等价是因为在它作检查时比较的是类型 T 和 T' 的结构(没有必要枚举比较它们所有的值！一般说来也不可能做到。)Alog1-68就采用了结构等价。

以下规则说明如何判定类型 T 和 T' 是否等价，其中类型定义按笛卡儿积，不相交的联合、映射等。(可根据程序设计语言的不同类型用类似的规则措辞。)

- 若 T 和 T' 均为基本类型，则 $T \equiv T'$ ，当且仅当 T 和 T' 存储值的结构完全一致。如， $\text{Integer} \equiv \text{Integer}$ 。
- 若 $T = A \times B$ 且 $T' = A' \times B'$ ，则 $T \equiv T'$ ，当且仅当 $A \equiv A'$ 且 $B \equiv B'$ 。如， $\text{Integer} \times \text{Character} \equiv \text{Integer} \times \text{Character}$ 。
- 若 $T = A + B$ 且 $T' = A' + B'$ ，则 $T \equiv T'$ ，当且仅当 $A \equiv A'$ 和 $B \equiv B'$ ，或 $A \equiv B'$ 和 $B \equiv A'$ ，如 $\text{Integer} + \text{Character} \equiv \text{Character} + \text{Integer}$ 。
- 若 $T = A \rightarrow B$ 且 $T' = A' \rightarrow B'$ ，则 $T \equiv T'$ ，当且仅当 $A \equiv A'$ 和 $B \equiv B'$ 。如， $\text{Integer} \rightarrow \text{Character} \equiv \text{Integer} \rightarrow \text{Character}$ 。
- 否则， T 不等价于 T' 。

这些规则虽然简单，但却不易看出两个递归类型是否结构等价，请看下式：

$T = \text{Unit} + (A \times T)$

$T' = \text{Unit} + (A \times T')$

凭直觉。 T 和 T' 是结构等价的。然而，推断任意两递归类型是否结构等价的决策本身就使得类型检查极为困难，因为递归次数不等结构不等。

按结构等价极易造成语义混乱。

例3-13 试比较以下两类型及声明

| | |
|--|---|
| <pre>type man is record name: String; age : Integer; end record; m: man;</pre> | <pre>type dog is record name: String; age : Integer; end record d: dog;</pre> |
|--|---|

它们结构相同，若按结构等价，则可以有：

`m.age := d.age;` //允许，都是Integer结构元素。

按强类型是不可以的，编译应检查出错误，所以，结构等价往往是伪强类型。于是又有按名等价规则：

按名等价可定义为：程序对象 O_1, O_2 ，若 $O_1' \text{ type} \equiv O_2' \text{ type}$ 当且仅当两类型具有相同的名字。

按照这个定义例3-13的man和dog是两个不同的类型，尽管它们的结构相同，再看一例：

例3-14 Ada的类型等价

```
type A is array (range 1.. 100) of INTEGER;
type B is array (range 1..100) of INTEGER;
    OA1, OA2:  A;
    OB1, OB2:  B;
    OC1: array (range 1.. 100) of INTEGER;
    OD1, OD2: array (range 1..100) of INTEGER;
    OE1: A;
```

这里声明了两个有名类型A, B，两次声明匿名数组对象(匿名声明时强类型语言内部给名)，它们的类型结构是完全一致的，根据按名等规则：

| | |
|---------------|--------------------|
| OA1, OA2 | 是同一类型(都用A声明) |
| OA1, OB1, OC1 | 是不同类型(类型名为A, B, 无) |
| OD1, OD2 | 是同一类型(同时声明，虽无名) |
| OA1, OC1 | 是不同类型(两次声明) |
| OA1, OE1 | 是同一类型(虽两次声明，但同名) |

显然，按名等价的前题得结构等价。但如果程序员把不同结构的类型写上了相同的名字，它们等价吗？例如：

```
type A is range 1..100;
type A is array (range 1..100) of FLOAT;
```

显然会引起名字冲突。

当然，按名等价不会都只带来好处，它也带来一系列类型兼容问题。如字面量是什么类型？它和一系列相同结构的类型变量能兼容吗？Ada的泛类型，基-子类型继承且兼容、父-派生类型(继承不兼容)的规定就是解决这些问题的。

(4) 类型完整性规则

这是语言设计中正则性原则在类型设计中的体现。既然类型是值的集合以及在这个值的集合上的操作集合，那么我们指定的操作，特别是共性的操作，不能对不同类型的值有所区别。前述第一类值(头等值)就是因为程序对象中有不能保证类型完整性的值而分出来的。第二类值有的是不能单独存储(结构成分)，有的是不能单独求值(结构类型只能成分求值)、还有不能作函数返回值(变量引用，函数抽象)，然而，它们确实又具有值的特性(可求值、可作参数传递、可构成更加复杂的数据结构等)。

类型完整性准则是：涉及值的类型中不能随意限定操作。

简而言之，程序设计语言的设计者应力求没有第二类值。因为过多的限制影响语言表达能力并违反正则性。但实际情况是Von Neumann机计算模型的制约和程序语言技术发展水平的制约，完全没有第二类值是不可能的，例如，Pascal的函数返回值不能是向量，Ada却可以。Ada的函数抽象不能作函数返回值，ML却可以(因为ML完全取消了赋值，并有高阶函数)。

类型完整性原则的另一好处是减少语言形式化的麻烦。从而为完全自动编程计算打下良好基础。这也是追求类型完整性的实质。

3.3 表达式

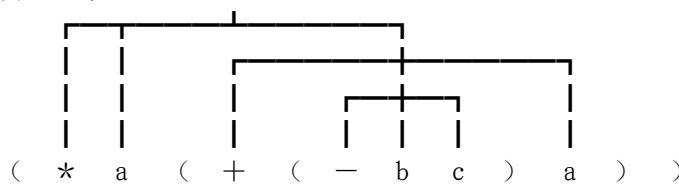
前述表达式是操作数，运算符的序列，目的在于求值。也谈到运算符本质上可看作函

本节讨论五个问题：表达式一般有哪些表示法？哪些程序对象可以作为操作数？表达式的种类。求值顺序和基本类型的兼容。

3.3.1 表达式表示法

各种语言表达式表示法不同，多半是为了与语法分析机制一致。表达式有双重作用，一方面程序员藉以表达它所要求的计算，它的表达形式又(隐含)指明了一个语法释义树。释义树本质上是建立操作数和运算符的关系，它根据约定的结合规则、优先规则、加上括号优先可以决定该表达式的整个运算次序。因此，按树模型编译出的目标代码的动作步骤和程序员预计的计算是一致的。

- 前缀表示法 运算符在前后跟一个或两个操作数或子表达式。括号优先是为了清晰，如LISP的表达式：



```
( + A 1) //一个双目运算
```

(-B) //一个单目运算

- 中缀表示法 运算符在两个操作数之间，单目运算符的操作数算第二操作数，如Pascal的表达式：

$$(a * ((b - c) + a))$$
 $A+1$

-B

中缀表示法和数学表示法一样，可读性好，通用程序语言这种办法最多。

- 后缀表示法 运算符在操作数之后, 如 FORTH:

a b c -a + *

A 1 +

B -

- 直陈修饰法 用关键字代替括号，并指定操作，例如，COBOL的一组ADD...TO...关键字相当于括号，同时指明“加”操作，如COBOL：

ADD A B 3 TO C 相当于 Pascal:

$$(a + b + 3) + c$$

注意，修饰法和Pascal中缀表示意义略有不同，后者表达式有一返回值(放在中间变量中)，前者直接把A，B，3之和“加到”C中。

此外，本节例子用算术运算符比较直观，关系运算符一样。不用括号就要加上规则，上述FORTH例子中如果没有规则约定，就不知道‘-’号是单目还是双目。多数语言有了规则同时可以加括号，这种冗余性是为了方便、清晰，有时是必要的。

3.3.2 表达式种类

可以以不同方式组合表达式，此刻先综述表达式基本类别：

(1) 一般表达式

传统意义下的表达式，见诸于传统语言文本。由项、因子、初等量、括在括号中的

表达式逐级定义，不包括显式的赋值运算符和其他关键字。

传统语言中Ada最丰富，我们列举如下，它们均能出现在赋值号右边，还可赋初值：

| | |
|----------------------------|---------------------------------|
| 4.0 | --字面量 |
| PI | --常量引用 |
| (1..10=>0.0) | --数组聚集， 10元素均赋0.0值 |
| (DAY=>1, | |
| MONTH=>OCTOBER, | |
| YEAR=1949) | --记录聚集 |
| SUM | -- 变量引用 |
| INTEGER 'LASE | --属性表达式、求预定义INTEGER的最大值 |
| SINE(X) | --函数引用 |
| COLOR' (BLUE) | --限定表达式，取COLOR类型中的枚举值'BLUE ' |
| REAL (M*N) | --类型转换表达式，将整子表达式结果值强制转成 REAL |
| (LINE_COUNT+3) | --括在括号中的表达式 |
| new CELL ' (0, null, null) | --分配算子表达式， 分配算子new为记录指针赋 初值 |
| • 其他简单表达式： | |
| "Valid code above " | --初等量 |
| not DESTROYED | --因子 |
| 2 * LINE_COUNT | --项 |
| B ** 2-4.0 *A *C | --简单表达式 |
| PASSWORD (1..3)="BTV" | --关系表达式 |
| COUNT in SMALL_INT | --关系表达式(集合元素表示) |
| INDEX = 0 or ITEM_MIT | --关系表达式 |

聚集表达式给结构类型变量赋(初)值。Ada给出三种表示法以方便应用。

例3-15 Ada的数组聚集赋初值

```

type TABLE is array (1..10) of INTEGER;
A:  TABLE:= (1, 2, 3, 1, 4, 5, 1, 0, 0, 0)      --按位置
A:  TABLE:= (1=>1,  2=>2,  3=>3,              --按下标名指明
              4=>1,  5=>4,  6=>5,
              7=>1,  others =>0)
A:  TABLE:= TABLE( 1 | 4 | 7 =>1,           --选择指明' | '的意义是' 或'
                    2=>2, 3=>3, 5=>4,
                    6=>5, others =>0)

```

按位强调次序不能错；按下标名次序可以颠倒；选择指名之后不能按位；others只能出现在最后。读者自然会想到什么时候用哪种。

例3-16 ML的三种聚集表达式

元组聚集表达式：

(a* 2.0, n/2.0)

构造了元组类型real * real的一个值，注意用圆括号。

记录聚集表达式：

{y= thisyear +1, m= "Jan", d=1}

构造了记录类型y: int, m: string, d: int的一组值。注意用花括号。成分名必有。

表聚集表达式：

[31, if leap (this year) then 29 else 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

构造了类型int list的一组值。注意用方括号，和第二项的条件表达式。

(2) 扩充的表达式

为了扩充表达式的表达能力，许多语言把表达式概念扩大，凡是求值，经比较得出真假值都认为是表达式。这些扩充是条件表达式、命令表达式、块表达式。本段先说条件表达式，其余见后文。

条件表达式引入if_then_else或case结构其中嵌入的是子表达式(只求值不赋值)。

例3-17 ML语言的条件表达式

- if_then_else结构:

```
if x > y then x else y
```

本表达式求x, y中较大者的值。

- case结构: 设thismonth本身是串String类型变量，它可以取得英文月份名简写的串。

```
case thismonth of
  "Feb"=> if leap (thisyear) then 29 else 28
  | "Apr" => 30
  | "Jun"=> 30
  | "sep"=> 30
  | "Nov"=>30
  | - =>31
```

整个是分情形的条件表达式，其中'-'即otherwise。此外，在"Feb"子表达式中又嵌入了一个if条件表达式。leap (thisyear)是函数引用作判断条件，它返回真假值以得出正确的二月份日期。每当得到(或读入)一个月份串，本表达式得出该月份的正确天数。当然是放在临时变量中。

3.3.3 优先级和结合性

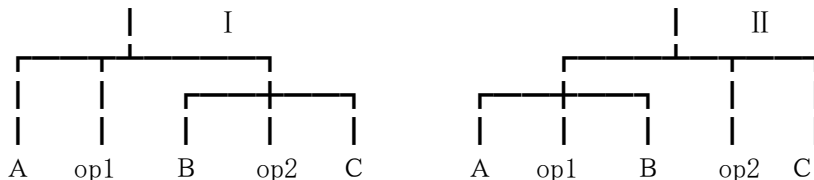
本书把运算符和函数视为一体，因此，表达式求值次序对单个运算符其规则与函数是一致的，在第 章我们一并介绍。现在是整个表达式(多个运算符函数)谁先谁后的问题。这就是运算符的优先级(precedence)和表达式结合性(associative)规则。

在3.3.1中介绍的前、后缀表达式似乎有些别扭，不太符合人的习惯，如果仔细分析，它们的释义树清晰，无二义性，比中缀表达要好。

例3-18 中缀表达式的二义性

设表达式A op1 B op2 C

很自然可以得出以下两株释义树:



因此，优先级pre规则是:

- [1.] 若pre op1 > pre op2 是释义树 II
- [2.] 若pre op1 < pre op2 是释义树 I
- [3.] Pre op1= Pre op2 由结合性规则决定。

如果按结合性规则:

- [1.] 自左至右释义，本表达式释义树是 II
- [2.] 自右至左释义(先扫描到表达式终端)本表达式释义树是 I。

每个运算符的结合性由语言设计者定。例如‘-’号一般是自左至右结合，‘:=’号、‘**’号是自右至左结合。

每种语法定义已经给出运算符的优先级。例如，Algol 60开创的BNF设的初等量-因子-项-表达式概念，即数学表达式中先乘除，后加减的反映。优先级越高的运算符(作为释义树的一个结点)它在释义树中位置最低。问题出在中缀表达，相同优先级谁先谁后，就要靠结合性规则了。一般在语法说明的正文中给出。

括号极易体现优先和结合性，早期语言几乎就是括号语言。FORTRAN I 和早期的LISP差不多，表达式的每一步都靠括号，过多括号反倒使可读性降低，虽然对翻译效率影响不大，它确是以辩证观点看待事物的最好例证：加一点清晰，过头是累赘。

3.3.4 类型兼容性

即使在强类型语言中也避免不了不同类型值的混合运算。特别是算术运算符作数值计算。所以语言手册在给出算术运算符定义时要规定不同类型运算结果值的类型。即语言给定的隐式转换。这种转换往往是根据实现的方便和可能。例如，C语言声明了：

```
char a, b
```

若有a+b则a, b转成int再计算，结果值为int。因为从运算器中取出字节要复杂得多。这种规定似乎毫无道理。short int 也是同样命运。按数据在存储中占位(bit)的多少分层，是一般语言转换时分层依据。例如，C语言规定类型层次是：

```
char<<short<<int< unsigned< long<unsigned long< float<double
```

每逢混合运算低层自动升格为高层，结果值是高层。请注意，所谓低层转高层并非真换了一个存储单元，而是在运算器内“看待”它或拷贝到中间寄存器时是真换了，原有数据是不能动的。低层向高层转增加的位数补0。

对于静态强类型语言，只有基类型和子类型可以混合运算且隐式转换为基类型。对于父类型和派生类型程序员必须显式规定转换。在Ada有：

例3-19

```
type BASE is INTEGER;
subtype SON_TYPE is BASE range 1..1000;           --子类型
type DIVERSE is new BASE range 1..1000;          --派生类型
A, B: BASE;
C, D: SON_TYPE;
E: DIVERSE;
...
A:= B+C,                                         --合法， 结果为B类型赋给A
A:= C+E;                                         --不合法，
A:= C + SON_TYPE(E);                           --合法， 有显式强制
A:= E                                           --不合法， 两个类型
D:= E'+BASE(E);                                --合法
```

对于用户定义的运算符(重载)，其隐式部分应与基本类型隐式规则同，显式部分则按用户定义。

例3-20 Ada的重载运算符

```
type UNIVERSE is (< >);
type SET is array (UNIVERSE) of BOOLEAN;
...
function "+" (ELEM: in UNIVERSE; A_SET: in SET) return SET;
```

定义了某个离散类型UNIVERSE的元素ELEM可以加到该类型的集合中，返回值是原集合类型SET。有了这个定义，程序中可写E+S这种表达式了。其中E，S是与ELEM和A_SET对应的实参元素和集合。

3.4 小结

- 值是对事物性态的表征和度量。同一事物在不同论域不同抽象层次上有不同的值。按抽象层次上层的值可以是下层的名(概括抽象的值)。
- 程序世界的值是有类型的，并寓于某种表示之中。程序世界的基本类型一般是整、实、字符、真值、枚举型;结构类型有元组、数组、记录、表、串; 用户定义的递归复合类型。
- 除类型分类而外，另一维分类是字面量、常量、变量。变量的时、空特性是过程式语言最大特点。
- 程序世界的值是字面量、复合量、指针值、变量引用(包括函数、过程调用)、和函数和过程抽象。它具有这两维的特点。
- 程序世界中求值方式是利用表达式和函数引用。表达式本质上是嵌套复合的函数引用。一个运算符就是一个函数。
- 第一类(头类)对象在该类型所有运算面前不受任何限制。具体到程序世界它们的运算是可作操作数求值;可存储;可作参数传递;可作返回值;可进一步构成复杂数据结构。类型完整性原则要求所有涉及值的对象都是头等对象。
- 类型是值的集合以及在这个集合上的操作集合。类型定义、类型规则、类型检查统称类型系统。
- 简单集合、笛卡儿积、不相交的联合、映射、幂集、递归函数等数学概念是以描述一般程序设计语言的各种类型的数学模型。
- 给出类型定义和规则之前要设计类型，按它的性态分有动态/静态，弱/强，因此有不同的等价规则。按名，按结构等价是决定类型强弱的重要因素。
- 表达式是操作数、运算符的序列。运算符有前缀、中缀、后缀表示法。一个表达式对应一棵语法释义树，不同结点上的运算符有不同优先级。同等优先级靠结合性决定其求值次序。括号是表现结合次序最有利的工具。
- 本书把表达式概念扩大。凡为得到值的运算集都可以叫表达式，这样，有条件表达式、命令表达式、块表达式。
- 类型兼容性规则是类型系统重要组成。基本类型兼容根据客观世界类型兼容的需要和机器世界实现的可能。每种语言文本必需给出。
- 在表达式形式之中，值得注意的是Ada的聚集、属性、限定、类型强制、分配算子表达式代表了近代语言的发展。

习题

- 3.1 选定程序设计语言的值的类型，根据需要和可能，能否举出三个例子说明虽然需要但不可能，至少是不方便？
- 3.2 函数抽象和函数引用这对概念能否推广到变量、常量、指针、字面量上？
- 3.3 标识符、符号名、变量、常量、字面量参数(变元)名这些术语相互什么关系，为什么不能去掉一些？

答： 计算机之中只能用名字操纵值。变量、常量、字面量都是标识符，符号名是标识符的名字。变量是代表多值的标识符，仅是一个名字，它的值在程序中可以

改变；常量的值在程序中不能改变，成为某个具体值的代名词；字面量也是常量，只是不能再给他取名字。参数是只用于调用的变量。这些术语相互区别，相互联系，不能去掉。

3.4 程序运行之前要联编和加载到内存。每次数据的绝对地址都是不一样的，指针的内容是地址，一个模块被多个程序连编怎样保证地址不指错？

3.5 试述把程序对象都定为头等对象的优缺点，类型能不能作为头等对象？

3.6 试列举真值、整型、实型、字符、枚举、指针类型的操作(一般机器有的非用户进一步定义的)。选两种写全。

3.7 写出以下Pascal程序声明的各类型的数学模型：

```
type Suit=(club, diamond, heart, spade);
      Rank = 2..14;
      Card = record s: Suit; r: Rank end;
      Hand = array [1..7] of Card;
      Turn = record
        case pass: Boolean of
          false: (play: Card);
          true: ( )
        end
```

3.8 试比较数组映射和函数映射同异，对以下问题写出(常量)数组和函数抽象的实现(语言不限，Ada更方便)。

(a)逻辑非。 (b) $n \leq 7$ 的阶乘 $n!$ 。

3.9 试写出

```
array [S] of array [T] of U
和array [S, T] of U
```

两数组定义的数学模型S, T为可枚举集合, U为实数集。比较它们的同异。

3.10 描述屏幕窗口可用以下数组类型：

```
type Window=array [0..511, 0..255] of 0..1;
w: Window;
如果有一记录:
type Point = record
  x: 0..511;
  y: 0..255
end
```

能否写出：

```
type Screen = array [Point] of 0..1;
s: Screen;
```

如果行，对记录有什么限制？它们值集有没有差别？写出与w(35, 22)相当的S元素引用。

答：对记录有限制。不能通过，如果行只能限定为两整形记录成分。

Point相当于一个 $\{(x, y) \mid x \in (0, 511), y \in (0, 255)\}$ 的集合，与window定义的范围相同；它们的值有差别。Screen类型的值集是Point $(0, 1)$ 全集。即对应每一个像素点坐标 (x, y) Screen的值都有两个，即0或1。与W(35, 52)相当的S元素引用为S $(\{35, 22\})$ iff $s(\{35, 22\}) = W(35, 22)$ 。

3.11 C语言的++x, x++是左值还是右值，为什么？

答：++x先作左值后作右值，是因为 $x=x+1$ ，在把X赋值给它的左值；

x++先作右值后作左值，因为先赋值给它的左值，再 $x=x+1$ 。

3.12 指出C语言的运算符(), [], →, ., 它们的操作数目度，与操作数结合方式，操作数类型，结果类型？

3.13 用运算符，操作数术语说明以下每个表达式，它们表达的是什么？

```
b [ 3] [5]
* (b[i]+j)
(* (b+i)) [j]
```

```
* ((*(b+i)) + j)
* (&b[0][0] + 5*i + j)
```

答: `b[3][5]`: 二维数组的一个操作数, 表达的是二维数组**b**的一个存储操作数。

`*(b[i]+j)`: 一维数组**b**的第*i*个操作数**b[i]**的地址与*j*相加的和作为*运算符的操作数, 即指向**b[i][j]**操作数的指针值。

`((*(b+i))[j])`: 操作数**b**和操作数*i*相加的和作为*运算符的操作数, *运算求得的结果作为起始地址指针操作数, 与操作数*j*相结合决定了以*运算结果为指向**b[i][j]**操作数的指针值。

`((*(b+i))+j)`: 操作数**b**和操作数*i*相加的结果作为*运算的操作数, *运算求得的结果作为操作数和操作数*j*通过+运算符求得结果有作为*运算符的操作数得到最终结果指向**b[i][j]**操作数的指针值。

`(&b[0][0]+5*i+j):&b[0][0]`是二维数组**b**的起始操作数的地址指针, `5*i+j`是操作数5和操作数*j*, 通过运算后得到的结果, 通过‘+’运算符运算后, 其结果作为‘*’的操作数, 意即从起始地址增加的位移, 最终结果为指向**b[i][j]**操作数的指针值。

3.14 两模块分别编译带来编程、扩充的许多方便, 但类型安全性必须用以下检查之一:

编译时类型检查

连接编辑时作类型检查

运行时类型检查

试述每种检查的作法和优缺点, 你若设计一小型强类型语言将作何选择, 为什么?

3.15 查一查手册, 按类型完整性原则比较Pascal、Ada、ML谁最符合这个原则, 即查以下值是否头等对象。基本类型值/复合类型值 (Pascal的串、集合文件单列出)/指针值 (ML无)/引用值/函数抽象。按它们定作常量/操作数/运算返回值/函数返回值/变元/成分时列表给出。最后说出你的结论。

3.16 指针有没有类型, 它的类型是什么, 值与操作集? 求值规则?

3.17 变量是抽象的值, 类型是抽象的变量, 也是抽象的值, 按类型层次可以不可以定义类型的类型? 它们的值集、操作集又是什么?

解: 变量是抽象的值, 类型是抽象的变量, 也是抽象的值, 按类型层次可以定义类型的类型, 它们的值集应是所有类型组成的集合。也可以由某种具体的语言限制其值集, 只允许几种特定的类型值。它的操作集可以包括组装 (几种不同的类型组成一种复杂类型), 比较相等 (两种类型是否是同一类型)、赋值由已定义的类型产生新的类型拷贝) 等等。

3.18 C语言的指针和数组是同一类型吗? 试述它们的同异。

—

第4章 存 储

如前所述，程序世界、机器世界是同一事物的不同层次描述。在每个层次上各有自己的术语概念。我们的目标虽然是为程序世界提供表达程序的术语和语言，因为它们密切相关。我们也应该了解机器世界里存储对象有哪些，怎样工作，可以实现哪些机制。这些工作机制反映到程序世界是什么？在某种意义上说，越能在上层构造出更新、更多的符合软件工程原理的程序对象，越要在下层想出更多的实现办法，如果都能实现，则程序设计语言就进了一步。

有关存储对象的基本术语和机制，我们假定读者已在汇编语言课程学过，此处不重复，只是在涉及到机器世界术语和机制时我们才提到它。请注意，我们讨论问题的立场是程序世界。还要注意，程序对象和存储对象并非一一对应的，赋初值的简单常数一般也是不作为独立的存储对象，直接放在被赋值的变量中。所以研究存储对象的核心是抓住“变量”。

本章主要讨论变量的时空特性，第二节介绍各种实现计算的存贮模型，第三节讨论因时空关系而出现的悬挂指升问题，第四节讨论各种语言变量更新机制，第五节介绍有副作用的表达式。

4.1 程序变量的时、空特性

如前所述，程序世界中的变量不同于数学变量在于它的时、空特性。我们先从空间特性开始。我们知道，计算机内的变量的存储空间是随时可变的，它可以在外存磁盘上，也可以在内存的某个地方，而且同一程序两次加载运行变量也不会在一个地方，即非固定地址。为此，一般目标程序中都采用相对地址的办法，绝对地址就不重要了。对于程序，我们要找到某个计算对象，永远从该段程序的首地址开始。为了简化，我们在虚存空间讨论存储对象。

4.1.1 引用和指针

由于变量的名值分离，当程序操纵变量时，首先寻址再取内容。这是两种操作‘取地址’（引用）和‘取值’（递引用）。命令式语言两者都要，且有时是隐式的。自从Pascal,C引入指针变量可显式操纵变量的地址，给程序员带来极大方便。例如，一个大单位，随机录用上千个员工。每个员工履历是一复杂记录，现登记现录入无序存放。我们只要按它的某个属性特征对指向它们的首地址（指针）排序，即可按排好序的指针表次序打印各种报告，而无需将庞大的记录重排多次。

指针本身也是程序对象，它也有地址，于是C语言允许多级指针的机制。其它语言编译只允许一级指针，除非指向复杂数据结构内嵌的指针。用 \uparrow (Pascal)和 $*$ (C)标记指针变量标识符以识别是操纵指针变量的内容(地址)，还是操纵指针变量所指对象的内容(值)。

例4-1，C程序中的指针

```
int i;
```

```

int * p,  *q;
    ⋮
*p=i;      //p指向整变量i
p=i;       //错误,  p中只放地址值
p=&i;      //正确,  效果同*p=i, &取地址符
q=p;       //正确,  同类型赋值, q, p都指向i
*q=*p+1;   //q指向i,  此时i已成为i+1
q=p+1;     //'1'仅是名,  值取决于所指对象每个单元占多少字节。

```

所谓寻址即要给出编译(或解释)时程序变量名与地址的对照表。每当给程序对象分配存储时就在表中填入, 这样就创建了引用(reference)。由于程序对象名字编译后不存在, 每次操纵用的是地址, 这个在对照表中有的地址即为该对象的别名。早期语言认为这是属于‘内部’的事, 外叫‘变量’, 内叫‘地址码’。而C++语言把这个占据存储的存储对象也上升为程序对象, 叫做引用, 使用户直接操纵地址。引用, 指针、变量关系的示意图如图4-1。

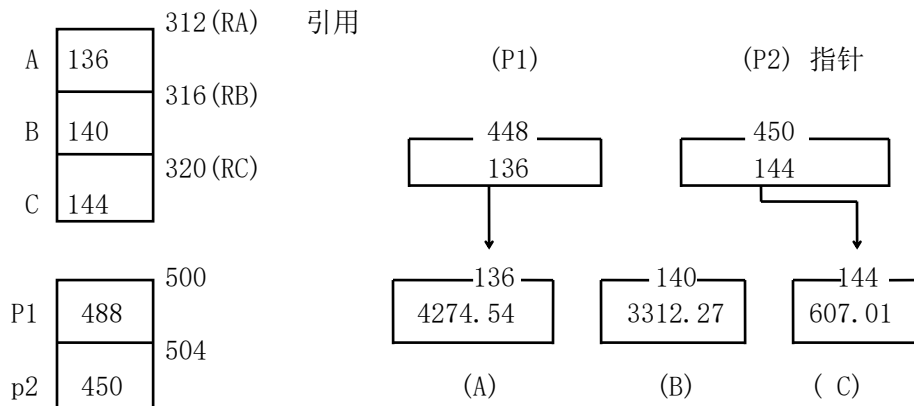


图4-1 变量、指针、引用示意图

变量A的地址码分配为136放在312中, 每当程序中用A的左值, 查出136即是, 若用A的右值查到136再找136的内容。既然312也是存储单元地址, 则把它对应为一引用名RA。只是一旦给出136再也不许改变, 它成了A的别名, 相当于常量指针。指针P1, P2既是程序对象, 它们被分配在448, 450存储单元内, 地址码 448, 450同样成为P1, P2的代替物。如果在448中给予136地址则P1指向A。给140则指向B。其内容是可变的, 所以和引用RA不一样。RA虽然也放一个地址, 但更强调的是引用内容。即当RA, A同时出现在赋值号右边, 其效果完全一样, 这一点和常量指针又不一样。它和变量A不同之处仅在实现上, 效果是相同的, 而且引用‘变量’必须赋初值, 一旦赋初值则在其作用域在其所在程序单元的范围不得改动。

C++和C引入引用主要是用于函数调用的传地址。形、实结合正好相当于引用变量赋初值。(这在第6章中还要讲)。

4.1.2 递引用

一般说来, 通过指针变量引用某变量的内容叫做递引用(dereference)。多层指针则多

次递引用。指针用‘↑’或‘*’显式地表达了它的递引用。但多数程序设计语言都有隐式的递引用。例如，下标表达式，除了引用下标变量值再引用该元素的值，即使出现在赋值号的左边的下标表达式也是递引用。我们称这是自动递引用。

FORTH语言是唯一不容许自动递引用的，它只有显式递引用运算符@：

例4-2 FORTH的递引用运算符@

```
[1] 13 VARIABLE xx           (声明变量xx并赋初值13)
[2] 0 VARIABLE Y             (声明变量Y并赋初值0)
[3] xx @ 2 * Y !             (相当于Y=xx*2)
```

如果只写xx 2 * 则为将xx的地址乘以2放在Y之中。

4.1.3 变量的时态

除了空间名值分离导出指针、引用、递引用程序世界的概念之外，变量还有时间特性，以它的状态刻划：

(1). 分配/未分配/除分配

为程序对象分配存储就创建了存储对象，程序对象就可以访问了。如果在编译时做这个工作我们叫静态分配，程序中变量多数如此。如果在程序运行时刻分配存储，我们叫动态分配，程序中的指针(或访问)类型变量则按动态分配，一般由程序员显式指定(用new操作)。若程序对象声明了但未分配存储对象即投入运行，此程序对象处于**未分配**状态。撤消程序对象即除去已分配的存储对象，我们叫除分配或除配(deallocate)。除分配可由程序员显式指定(用delete操作)，也可以约定由语言的执行系统或操作系统自动实现。自动将当前无用的程序对象除配，并收回待用称无用单元回收(Garbage collection)机制。动态(或无)类型语言一般有此机制，静态类型语言虽然也有动态分配部分，往往不设此机制(如C语言)

(2). 定义/未定义/失去定义

分配的存储单元总是有残值的(上个程序运行后留下的)，这些值无任何意义(与本程序无关)。我们说，变量**未定义**。如果通过赋值或赋初值，就是**定义变量**，除了它的值对本程序计算是有意义的。如果该变量执行超出了我们指定它有意义的区域(例如，一个局部块的末端)，只要它的存储没有撤消，它总是存在，但其内容(值)已**失去定义**。以下举例说明。

例4-3 变量的状态(Ada)

```
procedure MAIN is
declare BLOCK;
    type CELL;
    type LINK is access CELL;
    type CELL is record
        VALUE: INTEGER;
        NEXT: LINK;
    end record;
    type VECTOR is array (INTEGER range< >) of FLOAT;
    LA: LINK;
    V: VECTOR;
    VB: VECTOR(1..5);
begin
    LA:=new CELL;
    LA.all:=(3120, null);
```

--声明访问类型变量LA，但未分配
 --声明数组类型变量V，未分配
 --声明数组类型变量VB, 运行前已分配但未定义
 --动态分配无名CELL对象由LA代名
 --定义LA所访问对象

```

V:=(1=>5.0, 2=>4.0; 3=>3.0, --分配V为六元素数组, 且得到定义, 动态完成
    4=>2.0, 5=>1.0, 6=>0.0) --只有VB(3)得到定义, VB的其余元素未定义

VB(3):=V(1);
end BLOCK;
L:=LA; --LA值已失去定义, 出错
        --V, VB(3)的值均失去定义
        --Ada有无用单元收集, 此处可将失去定义的变量存储回收
end MAIN;

```

4.1.4 可存储值

就变量访问和更新而言, 我们把可以直接访问和更新的存储对象的值称为可存储值(storable)。它们是可访问的最小存储单元。因此, 复合值如果它们的成分不能选择更新, 该复合值即为可存储值。例如, Pascal的集合值, 不能单独更新其中某个元素, 它是可存储值。反之, 数组元素是可以选择更新的。整个的记录、文件也都不是可存储值。所以, Pascal中可存储值是:

- 基本类型值
- 集合值
- 指针值

变量引用、函数过程抽象也不是可存储值, 它们本身不是可存储的。C++语言设置了引用类型变量, 变量引用有了名字, 它是可存储的。

ML的可存储值有:

- 基本类型值
- 记录、构造、表
- 函数抽象
- 变量引用

ML的记录、构造、表是可存储值, 因为它们不能选择更新, 更新其中一个元素存储要重整一遍。由于函数抽象和变量引用全部采用类似C++引用类型的办法, 借助于引用可以有选择地更新复合值。事实上除数组而外, ML所有的值均为可存储值。

例4-4 ML的函数抽象是可存储值

函数名即函数抽象, 在Pascal中, $f(x)$ 的 f 是不能单独存取和更新的, ML中却可写为:

(if exp then sin else cos) (x)

这个条件表达式的返回值(sin或cos)是函数抽象, $\sin(x)$ 是函数定义, 也叫型构(signature), $\sin(a)$ 是函数引用。后二者都不是可存储值。

4.2 组织存储对象的存储模型

存储管理模型是翻译器必须选定的关键部分, 各语言不尽相同, 但从总体说有三种存储模型, 静态存储, 堆存储, 栈存储。后二者也称动态存储模型。

4.2.1 存储对象的生命期

每个存储对象都有其创建(生)、可用(活着)、撤消(死)的生命期。每当分配存储即创建存储对象。当所在程序块或整个程序执行结束,程序对象死亡,存储对象也应自动撤销。但多数对象生命期不会像程序执行期一样长,也有少量对象的生命期比程序还要长。我们把寿命和程序一样长的变量称为**全局变量**;寿命和程序中一个或几个模块一样长的变量称**局部变量**;寿命大于程序执行期的变量称持久变量。文件变量即**持久变量**。与此对应,程序中变量都叫临时变量(包括全局、局部、静态、堆变量)。

活着的对象必须在内存中才可用。然而,由于有虚拟存储技术。外存上虚存中的对象也是活对象,除非所占存储被撤销并分配给其它程序对象。

在程序执行期间,一个存储单元可多次分配给局部变量,甚至同一存储对象可由多个程序对象共享,如C的联合,Pascal、Ada的变体记录。FORTRAN等价语句指明的各变量。

反过来,一个程序对象可由多个存储对象实现。如前述的引用、指针。

程序对象死了,对应的存储对象没撤销,此时引用该对象就会引起难以预计的错误。分配了没有赋(初)值就引用也会引起错误。这两种错误在程序调试中屡见不鲜。

4.2.2 静态存储对象

在程序运行之前分配的程序对象都叫静态存储对象。编译时分配自不用说。近代语言有在装入内存后,运行前分配,Ada称之为确立(elaboration)期间(确立除分配某些对象外还计算初值表达式定义初值)。之所以称静态,因它们在分配之后整个程序执行期间不再改动。静态分配常伴之以初始化。

所有语言的全局变量都是静态对象。某些语言(COBOL)只有静态对象;某些语言(Pascal)除全局变量而外没有静态对象;而C和Algol允许程序员把局部变量声明为静态的,即该局部块执行结束静态变量并不失去定义,该块下次还可用,但同一程序其它局部块却不能用。所以Algol把它叫做OWN(自己的),声明中显式指明。C则用Static,C的extern变量全局于所在文件也是静态的。

静态存储对象直观易于查错,但它不能支持递归,因为多次递归调用相关的动态参数无法分配(不知道递归几次)。

静态存储对象仅限于全局变量的第二个缺点是被迫使用易引起副作用的全局量。对于某些问题(如求随机数,输入/出程序指示当前输入/出位置的指针)需要在程序多次执行时值有连续性,因而没有执行后仍保持值的变量(若完全是局部变量执行后自动销毁)无法模型客观世界。然而,采用全局变量其它无关模块也可使用就会引起副作用,C的静态变量即为局限于一个文件(模块)的局部变量,即保持值的连续性又有保护作用。

静态存储对象,无论是全局量还是局部量的第三个缺点(也是它的优点)是占据的存储不到程序执行完不释放。对于大程序且有众多只需短寿命变量的应用就浪费了大量存储。即使有无用单元回收机制不到执行完也收不到它,FORTRAN的等价语句(新一代语言的联合、变体记录)就为(起到)缓解作用而设。

请注意,静态分配的不一定是静态变量,程序中的局部变量(C语言叫自动变量)是静态(编译时)分配的,运行时存贮的地址可变。

4.2.3 动态存储对象

动态存储对象在程序执行期间诞生(分配和定义)，故名动态。多数动态对象它们的大小和结构形态往往依赖输入，编译时无法确定。例如，递归定义的二叉树，其结点、叶子、层数完全由输入值定。因此，动态存储对象生命期长短不同，大小不同，类型各异，如何使存储管理高效是一个中心问题。管理不好对程序执行的时、空效率影响极大。在某种意义上说，它是发展先进语言或软件技术的关键。例如，Prolog、Smalltalk出现初期都是执行效率难以和传统语言匹敌的，慢5-30倍。不断改进才能进入实用。影响最大的因素是动态存储对象的生命期，而不同生命期存储对象搭配使用，恰巧是增加表达能力，减少程序错误的进步。所以一般语言提供管理不同的生命期对象的模型。最简单的模型是在内存中开辟一个堆(heap)空间，放入其中的动态对象的寿命完全由程序员控制，语言系统全然不知。每当创建一新存储对象，存储管理则找出一个足以放下该对象的堆空间。每当存储管理得知(出了所在程序块)它已死亡，它就记下该对象不再使用。随来随堆，谁死谁释放，新来的没地方就等到老的死了再存入，故名堆。一般的指针动态生成的对象即按此型式，故指针变量也叫堆变量。

管理堆型式最大的问题是多次存储后留下存储碎片，因为新对象的大小往往不能填满老对象的空间，时间长了形成星星点点的许多“小洞”，管理很麻烦。拿出记录该碎片信息所费存储比这些“小洞”本身小不了多少，而且运行时一个到一个链接十分低效。再者还没有能找出合适的识别算法将“小洞”合并。这些问题各语言系统都采取了各自的做法。本节将介绍主要做法。

由于堆型式管理存在着麻烦，人们寻求并得到第二种管理型式：嵌套生命期型式，按照这种型式任何两同时出现的对象其生命期长短可以不同，但必须完全嵌套，即不能交错。也就是在时间上一个包容一个，我们就可以把相近寿命变量归成一组，最长寿命组放在堆栈的底部，最短的在顶部，逐级释放和重新占据存储，可以得到成片‘干净’的存储。如图4-2所示。

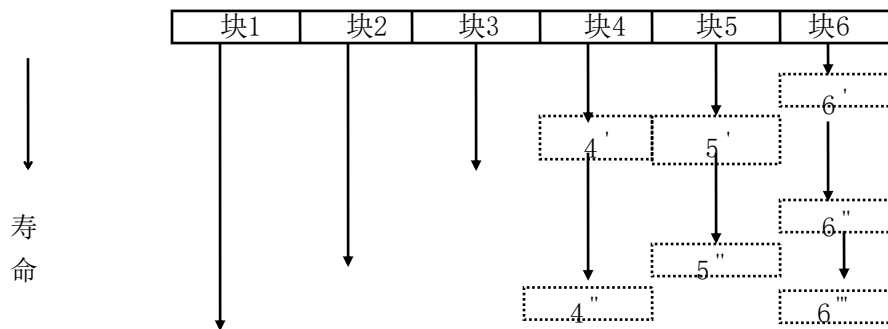


图4-2 嵌套生命期存储原理图

本例依次分配了6组不同寿命的存储块。由于块6中对象寿命最短，全部重新分配3次，块4，块5可重新分配两次…而块1的一次生命期还没完。如同倒下来的堆栈，栈顶的对象生得快死得也快，死了清除再分配第二批。而栈底的对象一直活着。事实上，动态对象寿命长短本来就和所在嵌套模块相关，只要把寿命特长的(如文件，全局量)排出来归到栈底的某一组，把寿命特短的(如循环控制变量)另立嵌套组，这个分组的问题也就解决了。

4.2.4 动态堆栈存储

按照嵌套生命期存贮原理，对于嵌套块结构的程序设计语言如(Pascal, Ada, C)动态堆栈型管理特别有效，因为多数变量和所在块寿命一样长。变量生死随所在块的激活和消亡高效自动实现。所以C语言把局部于块结构的变量叫做auto(自动)变量。但多数语言(包括C)堆、堆栈管理同时并用(原因见后文)。

4.2.4.1 动态堆栈式管理机制

为了说明堆栈式管理，我们再参照图4-2运行时堆栈，结合执行过程，介绍其实现机制，如图4-3所示。

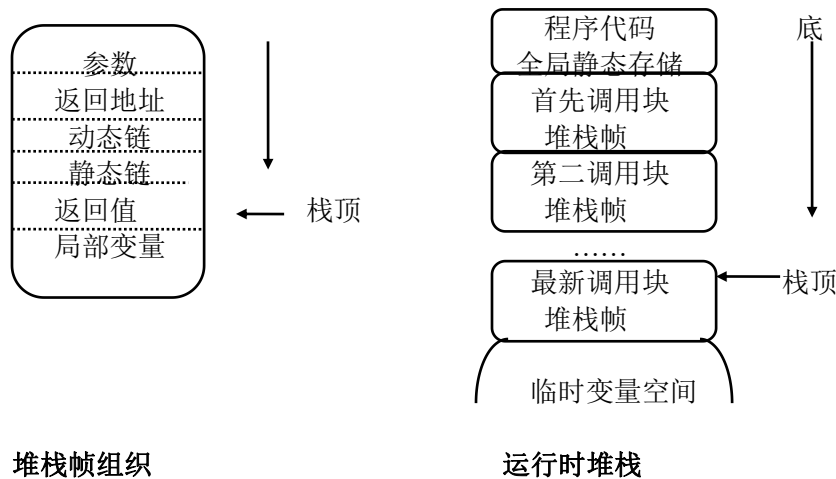


图4-3 运行时堆栈和堆栈框架实现

运行时堆栈(Run-time stack)是在内存开辟一个空间，如同堆。首先将程序代码和全局、静态变量装入，示意为图4-3的右图最上一块。这一块的词法上辈(Lexical Parent)是操作系统。当执行控制开始时，首先执行这一块，当执行到调用时，按编译时生成的嵌套关系，找出被调用的块，将该块的数据(包括实参和局部量)作为新的一帧，记下静态链(连接词法祖先(Lexical ancestor))，压入运行时堆栈后记下动态链(连接执行顺序)。由于编译时词法祖先块的地址无法确定，静态链要根据祖先块分配存储对象去找，有时要用到局部变量和参数。动态链指出当前块的动态执行的上辈，以便在块出口时弹出栈内的执行指令。动态和静态链在压入栈时利用局部分配指针从栈中第一个‘自由’位置逐个向上(下)分配，记下该指针的值，以便以后弹栈。

每个堆栈帧(stack frame)的组织如图4-3的左图，它按以下事件序列压栈：

- [1] 调用程序将实参值置于新块底部，用局部分配的指针，一般说，最后一个实参先放入，倒数第二个第二…第一个放在最上面。
- [2] 接着放入新块的返回地址。
- [3] 把当前栈顶指针值拷贝到栈内，这是新块动态链的一个域(存放当前栈顶地址)。
- [4] 填写静态链，将编译时生成的标记码拷贝进来，对于调用块静、动态链标记是一样的。
- [5] 再留足够的位置以放下局部变量的返回值，如已有初始化则拿来就可以了。
- [6] 控制转回父块代码执行，直至到生成另一个内嵌块的堆栈帧，如此，运行堆栈又生

成一层…

如果某块对应的程序代码出口则根据动态链返回父块，再生成复盖的新的一层堆栈帧…
老块被复盖也就是除分配了。

此处为了简化，返回值假定在块内，实际上常用寄存器。

4.2.4.2 用堆栈式存储实现递归

显然，递归调用是堆栈存储管理最简单的形式。因为它的运行栈每次压入的是同一模块新的数据对象版本，静态链自然按逐层递归调用展开直至到达边界，现举例说明。

例4-4 Pascal 的递归函数

求整数之连乘积

```
function product (jj: Integer): Integer;
var kk: Integer;
begin
  if jj <= 0 then product:=1
  else begin
    readln (kk);
    product:=kk * product(jj-1)
  end
end;
```

其执行图示于图4-4。为简化令jj=2，读二次kk=25，7。

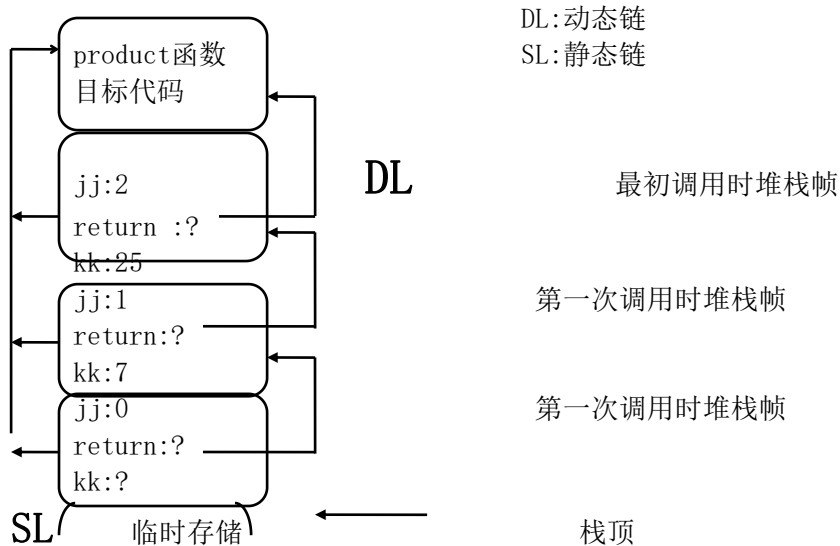


图4-4 递归函数的堆栈帧

4.2.5 动态堆存储

动态堆栈存储虽然解决了碎片问题，但是存储对象寿命只能和堆栈帧同时生死，限制太严。而且在每一个块的进口处并不知道存储对象的大小和个数，以及要求存储对象比创建它的

块的寿命长。所以堆存储有时是必不可少的。我们再详细讨论堆存储。

4.2.5.1 堆分配

许多语言允许程序员显式分配动态存储，分配语句可以出现在程序中的任何地方，办法是调用操作系统ALLOC函数。ALLOC将存储对象分配在堆中并返回对该对象的引用。如前所述堆分配有时在连接的较小的空间上，算法比较复杂。所以要建立一处自由表登录已死存储对象的空间，并查清相邻的碎片予以合并。这些算法还必须高效，否则影响性能。故往往是折衷，追求快速牺牲一些过小碎片（这又是潜在危险，当指针错误地指向这些无用小碎片时，后果极难预料）。

各个语言组织动态堆及分配后的返回值也不完全一样。现例举说明：

(1) FORTH的堆分配/除配

- 分配： HERE <表达式>ALLOT

FORTH在存放符号表和全局对象的字典中作动态分配。HERE是系统变量，程序员通过HERE访问字典顶端指针。先把HERE的当前值放入堆，然后对表达式求值得一整数N，ALLOT将N个字节加到字典指针。另设一小栈记下新增区域存入，用户只存入指针变量。

- 除配： 无无用单元回收。重新分配由用户自己写例程。

(2) LISP 的分配/除配

- 分配： (cons<表达式1><表达式2>)

见到这个表达式就分配一新表，并返回对新表的引用。新表左域由表达式1的值填入，右域用表达式2的值填入。

- 除配： 由无用单元收集机制自动完成。

(3) C的动态分配/除配

- 分配： malloc (sizeof(T))
 malloc (N)
 calloc (N, sizeof(basetype))

式中T、basetype是类型名，N是整数。第1式是分配T类型的一个对象；第2式是N个字节；第3式是分配一个数组，类型为basetype，元素是N个，并初始化为0。malloc和calloc均返回对新对象的引用。程序员必须将返回值赋给某个指针类型或强行转成所希望的指针类型。

- 除配： free (ptr)

ptr必须指向已分配过的堆对象。该对象经此函数即可重用(再分配)。

(4) Pascal动态分配/除配

- 分配： new (<指针名>)

<指针名>必须是声明为指向某类型的指针。该指针存放引用返回值。

- 除配： Dispose (<指针名>)

<指针名>所指对象置入自由表。

(5) Ada动态分配/除配

- 分配： new <TYPE>' (<表达式>)

分配一TYPE类型的对象，如果提供了可选表达式，对它求值并用以初始化该新对象，new是函数返回对新对象的引用。该返回值程序员应赋给ACCESS类型变量(即指针)。

- 除配： 有无用单元回收，显式除配一般不用，而是所在块执行完自动完成。

4.2.5.2 死堆对象处理

死堆对象叫无用单元或垃圾(garbage)。堆对象死亡是当它引用的对象已失去定义(即出了创建该对象的块)，我们称这种死亡是自然死亡。强行用(或类似)操作系统KILL命令也能显式使之死亡。自然死亡一般程序员和运算支持系统都是察觉不到的，不知不觉在一个未知的地方留下垃圾。

KILL命令在除配之后将该对象的引用(地址)置入自由表。在语言中实现KILL并拿出大量存储以便追踪，一直存在着很大的争议。

回收死堆对象的存储比堆存储复杂得多。死堆对象往往不在堆的末端而在中间，简单地

减少堆高指针是不行的。要把它做成登录可重分配存储对象的自由表。效率高低全看自由表的设计，曾经有过以下三种办法：

- 忽略死对象，这种表当然最容易实现。这看上去近乎胡来的策略还真有实现的，如AOD-VS操作系统上(数据通用公司的MV8000机的Pascal，在其参考手册中明文写出Dispose命令无其它操作。该操作系统是有虚存、页式管理和分时的)。

编译实现者的哲学是：页式管理若有浪费充其量不到一页。如果一页上的对象全都死了操作系统就不会把这页装入内存。事实上，当存储对象的创建时间、寿命大致相差不多时，这种策略能工作得很好。否则浪费巨大。

- 保持一个自由表它将所有自由空间连接在一起。这样，每个自由空间都要有几个字节记录它的大小并指示链接(多数硬件是8个字节)，小于此数的空间也只能放弃了。多数C，Pascal编译版本采用这个方案。8字节的头部说明是需要的：两个指针以构成双向循环链表，一个字节指示该小块是自由还是工作的。各小块编排次序和内存地址一致，这样相邻的小块合并就很方便了。死对象除配也很方便：只要把指示位设成“自由”。相邻小块都自由则合并。

扫描到第一个自由区并接着找到足够大的空间是很费事的，因为多数区域都在工作。故扫描从正在工作的顶部开始，倒着来，如果额外增加一些记录信息空间，扫描效率会高些。这里有时、空互易问题。

- 保持多个表按类型的长度，每种长度一个表，表的元素可以交换，次序就不重要了。这样就不需要标识区域的开销。归并、重分配实现起来就容易得多。Ada即采用这种办法。

堆式管理除产生碎片之外仍有悬挂指针问题，其原因是当有多个指针指向同一存储对象时，除配该存储对象及其一个指针而忘掉消除所有指针，其它指针就是悬空(其实是指向无定义存储)了。两对象共享一数据结构时更容易出现。因为一个死了一个活着，既要除配又不应除配，一除配就产生悬挂指针。而识别这种情况比较困难。因此，有人主张不要显式的除配(KILL)操作。Pascal的Dispose备受攻击，许多版本取消，并有反复。

LISP语言就是自动处理重用死堆对象。没有显式KILL命令，堆对象依然要死的，但存储管理不知道什么时候死。它采用无用单元回收机制，每当堆接近满的时候，从头检查起，如果是静态的、堆栈分配的它就认为是活的，凡有一活对象指向的存储也是活的。一律作上标记。最后将无标记的存储重新分配。

尽管如此，程序员依然有义务删除对不再需要的对象的引用(无用单元收集只解决最后全部不用了的对象除配)。此外，无用单元收集费时低效。随着存储廉价可配备大内存，就不致于在一个程序运行期间多次运行它。随着机器速度进一步提高，无用单元收集似乎是存储管理有希望的出路。

4.3 悬挂引用

指针指向一个已死或无定义的对象，就称为悬挂引用(Dangling References)，或悬挂指针。程序设计语言采用堆式管理，同时提供显式除配命令(KILL)时最容易产生。对于堆栈式管理，如果外块的指针指向内块的存储对象时也会产生悬挂指针，因为当内块执行完除配，外块指针依然活着可用，它就指向无意义的单元的。

悬挂指针是极为有害的。它甚至可以无缘无故地修改另一个程序。这对另一个程序是无论如何也查不出的错。

正因为如此，指向堆栈内(即内块)的指针Pascal是不允许的。Pascal只限指针为堆对象，构造链表和树数据结构的指针都在堆中分配。简单变量和数组在堆栈中分配，不提供地址运算。因此，只能用下标处理数组，不能用指针-地址操作快速索引。

C对指针的使用完全没有限制。取地址“&”运算符随处使用。当然也可以用到已除配的

堆栈对象上。即使C不许函数嵌套，主函数对函数的一层调用用堆栈实现时也会产生悬挂引用，试看以下例题：

例4-4 悬挂引用(C)

```
int * dangle (int ** ppp)           //参数是指针的指针
{
    int p=5;
    int m=21;
    *ppp=&p;                        //传回的指向p的指针
    return & m;                     //返回m的地址
}

main( ) {
    int k =17;
    int * pm, *pk=&k;               //pk指向k
    pm = dangle (&pk);              //返回时pm, pk均指向已失去定义的指针函数
}                                   //局部量, 即p和m
```

main()调用dangle后,堆栈中dangle所据空间可重新分配,而pm, pk指向其中的m, p就成了悬挂引用。

当然读者会问,既然对指针如此自由使用,为什么不全由堆式管理实现呢?原因是堆栈可实现高效的递归。C语言追求高效也支持递归,所以采用栈-堆结合的管理方式。只好把悬挂引用的问题留给程序员解决。C设计是为系统程序员用的,它的哲学是程序员不需要更多的限制。C本来就简单,提供特征不多,限制多了会成为无用的小语言。

函数抽象作为第一类值是另一个诱发悬挂引用的原因,请看以下示例:

例4-5 假定Pascal将函数扩充成第一类对象

```
var fv: Integer → Boolean;
procedure P;
    var V1, V2: Real;
    function f (m: Integer): Boolean;
    begin
        V1: =...;
        V2: =...
    end;
begin
    fv: =f           //第一类对象
end;
begin
    P;
    Writeln(fv(0));
:
end
```

P把局部函数抽象f赋给全局函数变量fv。执行P后引用fv(0)时等于间接引用f(0)。然而,此时V1, V2均已失去定义, fv(0)的返回值成了悬挂引用。

把局部函数值作为返回值产生的这类引用,在函数式语言把函数作为第一类值时悬挂引用尤其严重。所以ML, Miranda采用的方法是把所有变量作为堆变量处理,且不提供强行KILL指令。所以在程序块启动结束时,只要(直接或间接的)引用变量还存在,那么每个变量就继续有效。但存储空间利用率就没有堆栈式分配有效了。

Algol68部分解决了悬挂引用问题:对引用赋值作些限制,引用局部变量不赋给比它生命长的变量。在对抽象的赋值方面也作了相应限制。但实施这些限制是增加运行时的检查。增加了开销。

4.4 变量更新

有了变量的存储我们接着讨论这些存储里的值如何更新。变量更新实则是把存储看作一自动机其中状态(以共值表征)有了改变。存储对象更新只有两个途径：赋值和初始化。也可以用程序运行作界线划分为静态赋值(初始化)和动态赋值。

4.4.1 变量初始化

变量分配了没有定义就使用，是程序错误主要来源之一。为此早期曾有自动赋初值的做法，即分配了一律赋初值0，或某个可区分的位模式。事实证明，它不是个好办法，不显式赋初值带来查错困难，再者除数值变量而外，其它类型初值约定也易产生误解。

变量显式初始化由程序员在变量声明中给出。许多语言都有初始化子句。以下举出FORTRAN和C两个例子：

例4-6 FORTRAN 初始化声明

```
CHARACTER * 3 EOFLAG
```

```
DIMENSION      A(8)
```

```
DATA EOFLAG, ISUM/' NO ', 0/(A(I), I=1, 8)/8.2, 2.6, 3.1, 17.0, 4* 0.0/
```

FORTRAN赋初值在DATA语句中完成，用一对'//'区分，先写变量再写初值，交替表示按位置对应，(A(I), I=1.8)是循环表达的数组元素，甚至可以写出更复杂的嵌套循环。后面的值是与元素出现次序一一对应的，多个相同连续值用n*表示。句中ISUM为隐含声明的整变量。

例4-7 C的初始化声明

```
static char end_of_file_flag[ ] = { "no " };
```

```
int isum=0;
```

```
static float a[8] = {8.2, 2.6, 3.1, 17.0};
```

这是和上一个FORTRAN例子一样的初始化。C语言字符数组(第一行)的长度可由初始化值表达式长度定。第三行只赋了半数初值，也按位对应，其余自动为0。C称后者为初始化符(initializer)，由字面量、常量、常量表达式构成。表达式在编译时求值。只要复合变量中一个域(元素)初始化，所有域都要初始化。唯一的偷巧是自动为零。

C语言设计者认为FORTRAN初始化，有不必要的灵活性。它们比FORTRAN简单直接。FORTRAN有复杂的嵌套循环只能在装载后运行前完成。另一个现代风格初始化例子见前述例3-4。

现代语言支持堆栈式嵌套结构的初始化(ANSI C, C++, Ada)，即给自动变量初始化，在编译和初装时是无法完成的。因为这个堆栈框架还没生成。于是由编译算出表达式存在某个地方，并生成几个装载指令，待帧生成后装入。老C版本是不支持这种自动数组初始化的。概念上还属于静态初始化。

4.4.2 动态更新

赋值是以表达式的值强行置换存储对象中的内容。正因为如此，才有一个存储对象在不同的时间代表不同的程序对象，这是命令式语言造语义复杂和混乱的根源。我们称强行赋值为破坏性赋值(Destructive Assignment)。

(1) 简单赋值与聚集赋值

单个变量、数组元素、记录成分值的改变即通过简单赋值。它不涉及新值。只涉及两个对象：一个引用(在赋值号左边)，一个值或有值对象。多数语言只允许简单赋值，初始化中静态聚集赋值我们已介绍过了。近代语言几乎都扩充了动态聚集赋值。但仅限于同类型复合变量之间。例如，ANSI C中有：

```
typedef struct {int age, weight; char sex;} person;
person a, b = {10, 70, 'M'}; //b有初值, a没有。
a = b; //动态聚集赋值, a也有了。
...
```

(2) 通过函数赋值

除了强行赋值之外，如果通过READ例程把待赋值变量作为返回参数也能实现赋值。此时引用作为参数，值由输入介质提供。READ实现一系列动作完成赋值，没有返回值。在这个意义上，READ仅仅是过程，不是求值的函数。然而，真有一些语言赋值按函数实现。LISP及所有函数式语言，APL，C就采用函数赋值。

例4-8 C语言赋值的函数实现

```
#define MAXLENGTH 100
float ar[MAXLENGTH];
int high_sub, num_elements;
high_sub=(num_elements=MAXLENGTH)-1;
```

最后一行'='号如一般运算符是中缀函数名，前后两个操作数是参数，它返回一个值(就是num_elements的值)。正因为如此，它可以出现在表达式之中。LISP的'赋值'函数是replaca, replacd, 返回值是对参数的引用。

以下将常见语言赋值实现列表如下：

| 语言 | 赋值号 | 聚集赋值 | 多重赋值 | 实现机制 |
|------------------------|--|-----------------|----------------------|------|
| COBOL | MOVE = (在COMPUTE语句中) ADD, SUBTRACT, MULTIPLY DIVIDE | 可 否 否 | 可 可 可 | 语句 |
| FORTRAN | = | 否 | 否 | |
| ALGOL | := | 否 | 否 | |
| PL/1 | = | 可 | 可 | |
| FORTH | ! | 否 | 否 | |
| Pascal | := | 可 | 否 | |
| Ada | := | 可 | 否 | |
| LISP APL C(1973) | replace, replaced ← = | 某些版本可 可 否 | 返回结果 引用 值 值 | 函数 |
| | | | | |

ANSI C = 可 值

表4-1常见语言赋值的实现机制

其中多重赋值指形如 $a=b=c=3.0$ 的连接赋值。C语言是允许的。

既然函数可以动态地改变值，则函数式语言每当有破坏性赋值时就用参数束定的函数调用替代。每当要把一个计算值存入变量时，函数式语言则把该值作为参数传递给函数。赋值例程(或函数)体内的动作继续以嵌套函数实现，这样就避免了变量。参数束定在一个例程入口和出口的语义是不变的。因而是可以得到语义清晰的语言。

4.5 有副作用的表达式

我们把赋值 $V := E$ 称之为赋值命令(也就是语句)，它对表达式E求值，并以该值强行更新V中的原有值。一般表达式在求值过程中是不会更新其中操作数的值的，它只引用各操作数的值。它更新的是赋值号左边的值。如果E是一个复合的表达式会怎样呢？例如，其中一个子表达式是函数引用，在对E求值时必然要将此函数执行一遍。函数体中显然又有许多声明和赋值语句，因此，除了求出函数返回值之外，其中的赋值语句更新了函数体中的变量。也就是除了求函数值的主要作用而外，也起了更新变量的副作用(side effect)。如果只更新了函数局部变量，下次求该函数值时并没有影响，如果更新了全局变量，就会影响到其它表达式求值，这就难于控制了。

4.5.1 块表达式

块(Block)是嵌套在程序中的局部程序，由封闭的声明和语句集组成。如果表达式包含的子表达式是一个声明或定义，该表达式即为块表达式。前述函数引用是隐式的块表达式，函数体中既有声明又有语句。如果该函数没有副作用则与变量引用无异，如果有则以块表达式看待。

有些语言具有显式块表达式，如ML：

例4-9 已知三角形三边a, b, c的长度求三角形的面积：

```
let val s = (a+b+c)*0.5
    in sqrt(s*(s-a)*(s-b)*(s-c))
end
```

抽象形式是let D in E end，其中D是说明E的表达式，一起叫块表达式。D的作用仅限于说明E。

4.5.2 命令表达式

如果没有声明，块中只有命令，嵌入到表达式中则为命令表达式，C语言有显式的命令表达式：

例4-10 C语言读字符常见的表达式

```
(c=getchar())!=EOF
```

该表达式求真值，若C中读入是EOF，本表达式求值为0(假)。其它字符时值为1(真)。子表达式c=getchar()是赋值命令。故本表达式是(嵌入了)命令(的)表达式。

这个getchar()肯定是有副作用的。因为两次同样无参调用可得出不同的字符。从当前读的位置移到下一个字符位置就是副作用，但这个副作用正是C程序员希望的。

块表达式和命令表达式都是具有副作用的表达式，我们扩充它们无非是希望它的副作用能增强表达能力，事实上有了它们程序清晰可读，如上面的例子。ML没有全局量不会有什么坏的副作用。C语言就靠程序员保证了。

4.6 小结

- 程序变量具有时、空特性，故引入存储对象概念。它既是程序对象的实现又是独立的存储体。

- 变量名字编译后成为地址码，将此码存放在存储对象中，该存储对象对应的程序对象叫引用(C++)，引用变量是该变量的别名。指针可以存放任何程序对象的地址，由程序员操纵。

- 通过指针变量引用程序对象的值叫递归引用。

- 变量有分配、未分配、除分配，定义、未定义、失去定义的因时而变的状态。

- 存储模型有两大类：静态存储和动态存储。静态存储在运行前分配变量的存储，动态存储在运行中分配和除配。动态存储模型又分两类：堆存储和堆栈存储。

- 堆栈存贮，堆存贮模型是单今多数模块语言模型。

- 存储对象是有其生命周期的。一般局限于生成该存储对象所在模块的生命期。程序员可显式控制存储对象的生命期，在一模块内使用多生命期对象，以程序运行周期为准，大于此周期为持久变量，等于为全局，小于为局部，局部尚可内嵌局部。除持久对象外其余随程序运行终止而销毁，均为临时的。

- 静态存储对象一般由编译分配后在整个程序执行期间不会改变。全局的、外部的变量均静态对象。也有语言将局部变量定为静态的。

动态存储对象往往取决于输入值和程序执行情况无法在编译时确定所需存储，只能动态分配/除配。堆栈式管理和程序模块执行自然相适应，且天然支持递归，高效。堆式管理自由方便，存储利用相对耗费大。多数语言是两者相结合。指针对象一般是堆对象。

- 显式除配操作易引起悬挂指针，外块指针指向内块对象也易引起悬挂。悬挂指针是程序出错根源之一。一般认为命令式语言有三大害：任意的goto语句、悬挂指针、函数副作用，本章涉及一个半。函数副作用第6章还要讲。

- 程序中变量更新只有两个途径：赋值和初始化。赋值通过赋值命令或调用读例程。调用例程或函数是避免破坏性赋值重要途径。是函数式语言得以发展的原因。

- 当今语言赋值有按赋值命令模型有按函数调用模型。C语言按函数模型故有赋值命令出现在表达式中的命令表达式。

- 如果一个程序块(分程序)嵌入在表达式中则称块表达式。函数体是块表达式，嵌有函数引用的表达式是隐式块表达式，ML有显式块表达式。

- 如果块中只有命令没有声明即为命令表达式。扩充它们是为了扩大表达能力，但也会带来副作用，故称有副作用的表达式。

- 好的函数副作用使程序简单清晰，一旦放开了函数副作用又易产生难调易出错问题。这取决于语言设计的宗旨。但目前所有命令式语言都有一定程度的的函数副作用。

习题

4.1 一般变量和指针变量有何不同？引用类型变量与它们又有何不同？

答：在寻址对照表中，一般变量对应的地址码即为该变量值的存储单元，而指针变量对应的地址码则为该指针所指向的存储单元的地址。

引用型变量与指针变量的区别在于它是常指针，一旦对照表建立，也就创建了引用，不可改变。

引用是变量的别名，但它必须赋初值。

4.2 C语言中数组名的含义是什么，当把数组的地址赋给某指针时是否要用算符，为什么？

答：C语言中的数组名可以看作是一个指针变量名，它代表数组元素的首地址。但其值不可改变，这一点与引用型变量有些类似。当把数组的地址赋给某指针时不用运算符，因为数组名本身就是指向数组首地址的指针。

4.3 现今语言有静态数组、动态定义数组、可变长灵活数组，试说出它们的定义，并举出哪种语言采用它的例子（一个语言也行）。

4.4 找一个你所熟悉的语言所写的程序，指出哪些变量应用静态实现？哪些动态实现？该语言真是这样的吗？

4.5 试总结存储对象生命期有哪几种，C语言与此对应各叫什么变量？

4.6 何谓运行时堆栈？堆栈帧？堆？

答：运行时堆栈是在内存开辟一个空间，如同堆栈。首先将程序代码和全局静态变量装入。在执行控制开始时首先执行这一块；当执行调用时，按编译时生成的嵌套关系，结合动态链和静态链来增、减堆栈帧。

堆栈帧是指在运行时把每次调用所需的数据作为一帧，并包括返回地址、动态链、静态链、返回值和局部变量，压入运行堆栈栈顶。当对应的子程序执行完毕，该帧失去定义（可为新调用覆盖）。

堆是内存中预留的一片自由空间，当程序在运行过程中需要动态分配空间时，由操作系统从堆中按照一定的算法分配所需的变量的存储空间。

4.7 堆栈帧中设静、动态链两项的意图是什么？

4.8 设计一个程序输入随机的一千个职工记录。用悬挂指针的方法，按姓名的字典序，年龄（由小到大），按小单位名且姓名字典序，按工资多少（由大到小）各打印一个报告，即重排的原记录。

```
#include "stdafx.h"
#include "stdio.h"
#include "string.h"
struct crew
{
    char    name[20];
    char    unit[50];
    int     age;
    float   income;
    crew    *next;
} *head;
void sortbyname( );
void sortbyunitname( );
void sortbyincome( );
```

```

void sortbyage( );
void print( );
void main( )
{
    int I;
    struct crew *p,*sav;
    head = new crew[101];
    for(I = 0; I < 100; I ++) head[I].next = &head[I+1];
    p = head;
    for(I = 0;I < 100;I ++)
    {
        scanf("%s %s %d %f",p->name,p->unit,&p->age,&p->income);
        if(I) p=sav->next;
        sav = p;
    }
    sortbyname();
    sortbyunitname();
    sortbyincome();
    sortbyage();
}
void sortbyname( )
{
    int I=0,j=0;
    crew *tmp1,*tmp2;
    while (I<100)
    {
        for (j=I+1;j<100;j++)
            if ( strcmp(head[I].name,head[j].name) > 0)
            {
                tmp1 = &head[I];
                tmp2 = &head[j];
                head[j-1].next = &head[j+1];
                head[I-1].next = tmp2;
                tmp2->next = tmp1;
            }
        I++;
    }
    print();
}
void sortbyage( )
{
    int I = 0, j = 0;
    struct crew *tmp1,*tmp2 ;
    while(I<100)
    {
        for (j=j+1;j<100;j++)
            if (head[I].age>head[j].age)
            {
                tmp1=&head[I];
                tmp2=&head[j];
                head[j-1].next=tmp2;
                tmp2->next=tmp1;
            }
        I++;
    }
}

```

```

        print();
    }
void sortbyunitname()
{
    int I=0, j=0;
    struct crew *tmp1, *tmp2;
    for(I=0; I<100; I++)
        for(j=I+1; j<100; j++)
            if (strcmp(head[I].name, head[j].name)>0)
            {
                tmp1=&head[I];
                tmp2=&head[j];
                head[j-1].next=&head[j+1];
                head[I-1].next=tmp2;
                tmp2->next=tmp1;
            }
    for(I=99; I>=0; I--)
        for(j=I-1; j>=0; j--)
            if (! (strcmp(head[j].name, head[I].name)>0) )
            {
                tmp1=&head[I];
                tmp2=&head[j];
                head[j-1].next=&head[j+1];
                head[I-1].next=tmp2;
                tmp2->next=tmp1;
            }
    print();
}
void sortbyincome()
{
    int I=0, j=0;
    crew *tmp1, *tmp2;
    while(I<100)
    {
        for (j=j+1; j<100; j++)
            if (head[I].income< head[j].income)
            {
                tmp1=&head[I];
                tmp2=&head[j];
                head[j-1].next=&head[j+1];
                head[I-1].next=tmp2;
                tmp2 -> next = tmp1;
            }
        I++;
    }
    print();
}
void print()
{
    int I=0;
    for (I=0; I<=100; I++)
    {
        printf("%10s%10s%10d%10d", head[I].name, head[I].unit, head[I].age,
            head[I].income );
        printf("\n");
    }
}

```

```

    }
}

```

4.9 全局变量可不可以动态生成，为什么？

4.10 设计一个具有悬挂引用的程序。（见例4—4）

4.11 如果一个语言只支持静态存储行不行？如果只支持动态存储呢？如果都行则指出它们的优缺点。按第2章语言设计准则列表说明。

4.12 想一个办法上机测试C语言寄存器变量、宏变量的生命期，按本章讲述它是静态还是动态的？是局部还是全局的？是临时的还是持久的？

4.13 C语言数组和Pascal数组有什么不同。可以用函数映射来描述C的数组吗？如果可以又怎样描述。

4.14 试用汉语写出扩充某个语言具有无用单元收集功能的需求，并给出初步实现原理。

4.15 指针变量如同语句中的goto，愿意指谁就指谁，没有章法，goto的问题已用结构化解决了，你能想出有效管理指针的方案吗，至少要谈出途径。

解： 悬挂引用主要是指指针指向一个已死的或无定义的对象。

主要出错情况有两种：

1. 未给一个指针变量分配空间就给它赋值；
2. 函数返回局部变量指针。

解决方案

①在对指针实施运算的地方，检查指针变量是否已赋初值，即指针变量是否已指向合法位置。

②在对指针变量赋值的地方，检查指针变量是否先前已指向其他合法位置，避免悬挂引用。

③在对指针变量赋值（特别是数组）时，记录其“元素”的个数，以便能在其后的操作中检查是否越界。

④在对指针变量赋值的地方，记录引用对象已被哪些指针指向，为此引用对象被除时，提出编译警告，告诉程序员哪些指针已悬挂。

4.16 用图解方式说明4.2.5.2节中以一个自由表、多个自由表处理死堆对象的全过程。

4.17 函数式语言变量更新是如何实现的？为什么它没有副作用？

4.18 试述语言赋值采用函数赋值机制的优缺点。

第5章 束定

在程序中我们靠各种名字操纵程序对象，以此编制有声有色的程序。上一章因名、值分离我们引出存储对象概念，但只有当名字和存储对象结合在一起才构成程序对象，所以，名字不等于程序对象，它只有通过束定(Binding, 我国有译绑定, 定连, 联编的)才能成为程序对象。例如，一个变量声明了未分配存储，此时该变量名没有束定，只有在运行时分配存储才成为定义的程序对象。

发明名字是高级程序语言重大的进步，名字能为程序员提供可见可识的语义。但机器只把名字看成符号串，它没有识别名字表达的语义的能力，它只知道束定，把 $\sin(x)$ 与它的函数体束定起来 $\sin(x)$ 才能实施它的语义“对 x 求正弦”。这样人们希望的语义才成为现实。

本章学习束定和各种束定机制，声明、声明的种类和作用域等有关基本概念。本章还讨论由于嵌套块结构语言带来较为深入的几个问题；不同束定机制对语言释义的差异；名字的作用域与程序对象生命期匹配问题；束定机制与语言翻译器的关系。

5.1 名字与束定

如果一个名字只指称一个程序对象，这本来是没有值得讨论的问题，多数程序设计语言动态创建程序对象时一个名字可以在不同时候代表不同的程序对象。如嵌套程序外块、内块可以有相同的名字。一个程序对象可以有好几个名字，如引用变量名，指针名。极端情况下程序对象可以没有名字(无名类型)，名字可以没有对象(悬挂指针)。所以块结构、参数传递、递归、指针、引用、别名等机制使名字空间大为复杂化了。它打破了“一个程序对象对应一个名字”的简单概念。

把名字和存储对象联系起来叫束定，更概括一些，束定是将名字(标识符)和可束定体联系起来。所谓可束定体(bindable)是能反映(操作)语义的存储块，如常量、变量的存储体、函数体、过程体、类型、异常。

指明束定一般由程序员在程序正文的声明中作出，如：

```
int a, *p;
```

标识符 a , p 束定为 int 类型的变量、指针。然而真正束定是编译器或解释器完成的：它给 a , p 分配了存储对象并填上名字—地址对照表，某个地址上的存储对象就“是”该名字的对应物。反过来说也行。

可以把束定看作是完成名字指向存储对象指针的过程，但它和指针不同，首先它是编译(或解释)器做的。为每一个名字分配其语法要求的存储，也可以跨越时间，编译时占个位置，运行时再分配(实现束定)。再者，翻译器可以自动递引用束定而不能自动递引用指针。它是跨越程序世界和机器世界的概念。

在一个程序的生命期期间，一旦束定不再改变叫静态束定，反之，一个名字束定多个存储对象(不同时间)叫动态束定。还有一种介乎动静之间叫块结构束定。静态束定一般在运行之前完成(编译时做一部分，连接时做一部分，装载后确立(elaboration)时再做完)，也叫早束定。程序运行时动态完成叫晚束定。一个程序设计语言采用什么类型机制和程序运行机制决定了它以早束定还是以晚束定实现。我们现在分析已有语言的束定机制。

5.2 各种束定机制

除无类型语言而外，一般常见语言 (Algol, FORTRAN, COBOL, C, Pascal, Ada) 都是类型语言，每个程序对象均与类型相关，类型在声明 (或缺省声明) 中给出。

5.2.1 静态束定

较老的非结构化语言一般采用静态束定。在编译时建立一个符号表，最简单的情况，该表用三个域：类型，名字，地址束定即可实现，见图5-1用虚箭头表示束定。

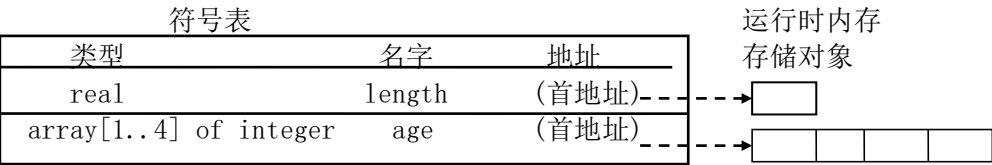


图5-1 符号表和束定

编译根据类型为名字分配适合大小的存储对象，按相对地址算出被束定对象的首地址，填入表，然而，运行时只有存储对象，符号表在执行代码中是没有的，多数语言此时均销毁。所以说，它跨越时间。编译时开始，运行开始时 (动态) 到程序结束运行时 (静态) 结束。

静态束定运行前完成，一旦束定不再改变 (符号表已销毁)。静态束定也可以实现多重束定，如下例：

```
例5-1 FORTRAN的等价语句
DIMENSION P1(3)
EQUIVALENCE (P1, P2), (P1(2), P3)
```

DIMENSION语句声明了一三元素实型数组 (P隐含声明实型) P1(3)。分配存储后实现P1束定。紧接着EQUIVALENCE又指明束定。名字的意思是P2就是数组P1，P3是P1数组的第二元素。如图5-2所示：

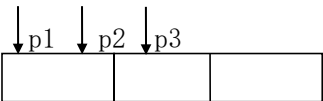


图5-2 多重束定

其他语言的多重束定是：

```
COBOL REDEFINES 子句
Pascal 无标签域的变体记录
C 联合类型
```

5.2.2 动态束定

动态束定是程序对象标识符在运行中分配存储并束定，而且可在运行之中改变束定。程序中的声明到运行时才束定显然要根据运行值判定。无类型语言解释执行，动态束定很自然 (见5.2.4)。类型语言在运行中解释类型并束定变量效率较低。方法是一样的。

FORTH是类型语言，它完全动态束定。不过为了提高效率它同时有编译器和解释器，相互切换使用。编译器不完全生成目标码，只生成解释起来更快一些的中间码。FORTH的程

序对象是有类型的字词（word）。在统一的字典中处理。字典是一个大的堆栈。系统的、预定义的、以前定义过的字都压在栈底，新的应用压在栈顶。新、老字都可以使用。每当处理一项声明，字典创建一个项存储用户定义的字。每个项分四个域：名字域：放名字，第一字节是名字长度；链接域：使该项成为可查找的数据结构（实为指向以前项的指针）；代码域：相当于类型域，标识名字是哪类程序对象（函数、变量、常量、用户定义的类型）；参数域：或称体存放字的特定含义，如常量就放纯值，变量放存储对象，函数则放可解释的代码。

代码域实则是指向运行例程的指针，这个例程就束定为该名字的语义，代码域定义了该类型对象的解释方法。最初的类型仅有“函数”、“变量”，用户可不断增加，每当声明一个新的类型符，则给出两段代码，一个是为该类型对象分配足够的空间并初始化，另一段是该类型的运行例程。指向这个新类型的指针成了该类型唯一的标识符，也成了今后声明该类型对象代码域中的内容。现举例说明如下：

例5-2 FORTH 的动态束定

| | |
|------------------|----------------------|
| 0 : 2by3array | (‘:’表示编译开始，后为类型声明符) |
| 1 create | (编译动作：将类型声明符装入字典项) |
| 2 2, 3 | (在字典项中存入 2×3维数) |
| 3 12 allot | (为六个短整数分配12个字节) |
| 4 does> | (运行时动作指令，取下标) |
| 5 rangecheck | (函数调用，检查下标) |
| 6 if | (如果不越界) |
| 7 linearsub | (函数调用，计算线性下标值) |
| 8 then | (给出数组基地址和位移) |
| 9 ; | (‘;’切换成解释执行，数据类型定义毕) |
| 10 | |
| 11 2by3array box | (声明并分配名为box的数组变量) |
| 12 10 1 2 box | (给box(1, 2)赋值10) |

CREATE起到第3行和DOES起到第8行即为上述两段代码。按类型声明符2by3array的束定，将编译后的1-8行压入字典堆栈的项中。解释执行时，按2by3array指针找到编译块，运行CREATE分配box的存储对象（在该项的顶端），接着解释执行第12行，运行does检查(1, 2)是否越界。若未越界计算下标准确值，在该地址下赋值10。

如果第12行以后用户把box定义为另一类型变量也可以，因为类型名束定于指向另一编译块的指针，再出现box则按最新释义。

当见到用户的FORGET命令则将至此定义的新字全部撤消，栈顶指针退下一项。如无FORGET命令，栈越积越大，可重用的字越多。

5.2.3 块结构束定

块结构束定使名字和反映操作语义的代码联系，更为复杂。它和FORTH动态束定不同的是，新程序块中的名字可以和外块相同即重新定义，但出了该块还要恢复该名字和原来束定。详细的实现机制见5.4节。在此处我们先介绍一个有趣的问题，即常量束定会因块结构动态运行机制，变成常量非恒值。

程序中为常量给出一个名字方便程序维护。因为若常量有了改动只需改定义常量名的那一次，程序中出现数十次都用名字代替了不用改。有的语言还扩充了常量表达式：

例5-3 Ada的常量束定

```
P1: constant FLOAT := 3.1416;
N: INTEGER :=5;
M: INTEGER;
MAX_LENGTH: constant INTEGER :=N*100;
```

```
MAX_INDEX: constant INTEGER := MAX_LENGTH-1;
```

最后两行都是要计算的常量表达式。它们通常在装入内存后运行前计算。结果值束定于常量名。P1, MAX_LENGTH, MAX_INDEX都成了只读变量(由编译加指令保护), 而符号对照表运行时已撤消, 这是正常情况。如果把这组常量声明放在过程PROC内。而调用PROC的主子程序MAIN内又把N定义为一般变量。且N和PROC都在一循环体内, 而循环次数不知, 每循环一次读一次N, 由于采用堆栈式管理, 每调用一次装入一次PROC并计算一次常量表达式(确立时, 简单常量编译时已赋值不必计算)。此外, 由于PROC中的N和

```
procedure MAIN is
...
  N: INTEGER;
...
  loop
    GET(N);
    PROC;
    when (A>B) exit;
  end loop;
...
end MAIN
```

MAIN中的N同名, 按作用域规则它们束定于同一存储对象, 出了PROC还原。显然, 因‘全局’量N取决于输入, 常量非恒值。

有时这种非恒值常量还非常有用。

5.2.4 无类型语言的束定

无类型语言更加依赖束定, 因为一个变量名可完全动态地束定到任何类型的值或操作集上。完全动态束定只可用于解释类型语言, 或像LISP那样只有简单、统一的类型结构, 这种语言类型直接和存储对象结合, 而不通过名字, 并和对象一起存在内存。符号表中就没有类型这个域了, 如图5-3所示:

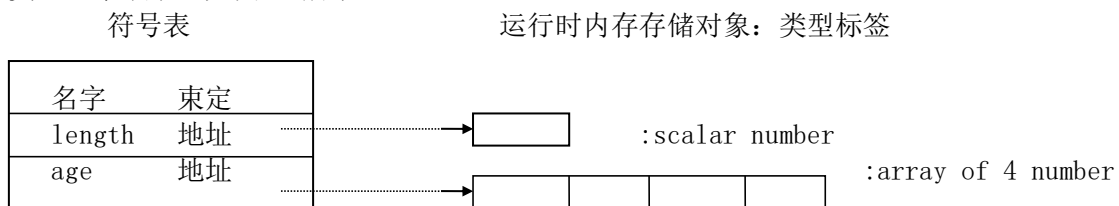


图5-3 APL的束定

由于无类型, 一个名字即使在一个块内, 只要程序员发出命令即可束定到另一个存储对象上, 与存储对象大小无关。APL、SNOBOL就是这种无类型语言, 只有按当时束定解释它的意义。

无类型语言“变量”是不用声明的。它只有定义(简单声明), 即将标识符束定到表达式的结果值上, 第一次引用则此标识符即为该值类型的“变量”。APL甚至连赋值概念也没有, 程序员显式操纵束定, 达到计算并改变值的效果。如下例:

例5-4 APL计算税金的程序

```
▽ TAXCALC
[1] 'ENTER GROSS PAY'
[2] GROSS ← □ //输入什么值GROSS就是什么类型, □表示终端输入
[3] →LESS × GROSS < 18000 //条件表达式为真转LESS标号句
[4] TAX ← .25 × GROSS //表达式结果值束定到TAX
[5] →DISPLAY //转到标号DISPLAY句
[6] LESS : TAX ← .22 × GROSS
```

[7] DISPLAY: 'THE TAX IS \$', Φ TAX

[8] ∇

' \leftarrow '即为束定，把表达式结果值所据的存储单元束定到TAX。' \rightarrow '相当于goto，' \times '指示后面表达式为条件，' \square '为终端输入，' Φ '显示TAX的十进制值。引号中的字符串是照录显示出来的。

程序员可把标识符束定在任何表达式上，不同类型值都可以。

5.3 声明

声明指明了本程序用到的所有程序对象。实质上，它给出预想的束定集合(实现世界)，即每个标识符和什么样的存储对象束定。声明的作用，一方面供翻译器处理时所需信息，一方面为人们阅读便于调试。对于类型语言，一般有显式声明部分或声明语句。强类型语言每个标识符都要显式声明。对于其它类型语言允许隐含声明。例如，FORTRAN的标识符第一个字符在(I..N)范围为整型，其余为实型，无类型语言无显式声明部分，直接给出束定(通过表达式)或定义。或隐含由上下文给出束定。即使是强类型语言为使程序清晰也有隐含声明。

例5-5 Ada的显式和隐式声明

Ada是静态强类型语言，它的声明有典型性，且最丰富，它的显式声明有：

| | | | | | |
|-------|---|---|--------|--|----------|
| 基本声明： | : | = | 对象_声明 | | 数_声明 |
| | | | 类型_声明 | | 子类型_声明 |
| | | | 子程序_声明 | | 包_声明 |
| | | | 任务_声明 | | 类属_声明 |
| | | | 异常_声明 | | 类属_设例_声明 |
| | | | 换名_声明 | | 延迟_常量_声明 |

Ada的隐式声明包括：块的名字、循环名字、语句标号(且为可选)以及循环控制变量。有些操作也可以直接出现不用声明，如预定义运算符，派生子程序等。声明的确立产生事实上的束定，下文讨论中为了方便不强调确立过程。声明就有了静态束定。

5.3.1 声明的种类

这里讨论的不是声明的程序对象有哪些类别，而是声明本身的类别，有：

- 定义
- 顺序声明
- 并行声明
- 递归声明

我们详细说明如下：

(1) 定义

定义为标识符束定提供完整信息，使标识符可束定于确定的存储对象上。定义就是声明，而声明不完全等于定义。例如，C语言的外部变量声明，Ada的带有(with)子句声明都是给编译提供信息的。所以，一般说来，一个标识符可以声明多次而定义只能一次。否则产生名字冲突。在这个意义上，定义是简单的声明。

具有完整定义的声明为完全声明，否则为不完全声明，不完全声明在相互递归的类型定义中是常见的，例如，二叉树的每个结点由结点值和左、右两指针组成，是先定义指向结点的指针类型，(此时结点是什么不知道)，还是先定义结点(以记录类型表达)的类型(此时指针成份未定义)？所以，只能先说一半再说整个，如例5-6。实现上不会有困难。读者想

想为什么?

有些语言把定义仅限于用已知信息定义程序中需用的信息,而声明可创建新信息。

例5-6 ML的类型定义与声明

ML定义两个类型

```
type book = string * int    //书名和版本号
type author = string * int //作者名和出版年号
```

book, author两个名字都束定于string和int组成的结构存储对象上。这是不安全的,编译后两标识符去掉后容易出错。ML还有新类型声明:

```
datatype book = bk of string*int
datatype author = au of string * int
```

它创建了两个新类型,除束定于string*int的存储对象上外另有标记au, bk则与原有的类型都不同了。前者按结构等价,后者按名等价。

Pascal, Ada, C++则认为每个类型构造子,如record...end, array[] of...,都引入新类型,定义只把标识符束定于类型。

如果把变量定义的概念仅限于用已有变量定义新变量,则许多语言为防止别名的混乱没有变量定义,只有引入新变量的变量声明,只有Ada有变量定义,即它的换名声明:

```
POPULA: INTEGER rename POPULATION (STATE);
```

将以国名为下标的人口数组元素换名为POPULA。

与此相反,函数式语言没有变量概念,当然也没有变量声明,只有和值束定的参数(变元)。即只有定义,无类型动态语言更是如此,全靠定义。

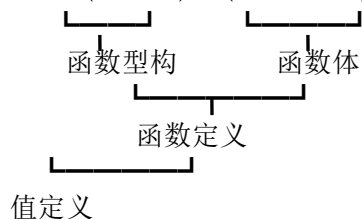
例5-7 ML的值定义

```
val count = ref 0
```

ref是引用相当于传统语言分配算符new,即将标识符count束定于数0,因而引入了该变元。

例5-8 ML的函数定义和值定义

```
val even = fn(n: int)=>(n mod 2=0)
```



以上是标识符even的值定义(因有val)。将标识符even束定于函数,其变元n为整数,体的表达式返回真值,则even的值为Integer→Truth_Value。以上还说明ML的函数抽象(even)是第一类值可用作值束定。如果定义为函数也是可以的:

```
fun even (n: int)= (n mod 2=0)
```

函数抽象 函数体

└───────────> <束定> ─────────┘

但值定义更好用,因为=>右边可以是任何值表达式。

(2) 顺序声明与并行声明

程序讲究的就是次序,所以声明一开始都有一个隐含约定:声明是顺序的,即后声明的声明符(declerator)可立即使用刚声明的声明符。如果把声明符集写作D则有顺序声明:

```
D1; D2
```

声明确立次序先D1后D2(分号表示)。因此,D1可以影响到D2的声明

例5-9 Ada程序包声明

```
package MANAGER is
  type PASSWORD is private;
  NULL_PASSWORD: constant PASSWORD;
  function GET return PASSWORD;
```

--声明程序包规格说明

--声明私有类型未定义

--立即用私有类型声明变量

--返回私有类型函数

```

function IS_VALID(P: in PASSWORD) return BOOLEAN;
Private
  type PASSWORD is range 0..700;          --定义私有类型
  NULL_PASSWORD: constant PASSWORD:=0;    --此时才定义
end MANAGER;

```

尽管PASSWORD在第2行声明时未定义，但第3，4，5行就立即用它声明其它对象。end MANAGER，表示程序包结束，以上声明都限于本包，如果第3行像第8行那样赋了初值，本声明就出错了。

并行声明不怕次序调换。ML中就有并行声明，其一般形式是：

D1 | D2 或 D1 and D2

两个子声明D1，D2是独立的，即它们确立相互无影响，确立先后不会改变声明的意图，因此，就不能立即声明立即用了。

例5-10 ML的并行声明

```

val pi= 3.14159
  and sin = fn(x: real) =>...
  and cos = fn (x: real)=>...

```

是合法的，加上

```

  and tan = fn (x: real) => sin(x)/cos(x)

```

就不合法了，因为在确立tan时它也许最先。

对于讲究副作用、次序的命令式语言是不会采用并行声明的。除非同一程序描述并行的子程序部分。而函数式、逻辑式因排除副作用，故可采用。

(3) 递归声明

递归声明是标识符以自身束定的声明，一般形式是：

```

D = ...D...           //D是包含标识符D的声明符
或 D1 = ...D2...      //D1是间接递归或称相互递归
D2 = ...D1...

```

递归声明通常限于类型、过程、函数、值定义。至少到目前还没有扩大到更大的方面，如类、模块、程序包、类属、异常等。

有的语言采用自动递归，有的语言则由程序员显式指明递归，后者当有重名时程序员有主动性。

例5-11 Pascal和ML的递归

```

FUNCTION eof(VAR f: Text): Boolean;
BEGIN
  eof:=eof(f) OR (f↑='*')
END;

```

这个Pascal程序原意为用标准函数eof检索正文文件f的内容，当该文件以'*'结束时检索结束。如不递归调用怎么也到不了'*'。但本程序自动递归调用自己定义的eof，而不会调用标准的eof(其中有f↑)，所以什么也查不到。

ML可显式指明递归：

```

val rec power=
  fn(x:real, n:int)=>
    if n =0 then 1.0
    else if n < 0 then 1.0/power (x, -n)
    else x * power (x, n-1)

```

其中rec是显式指明符。

5.3.3 块声明

程序均由模块组成，我们暂且把嵌套块和并存块的声明作用域，以及由此而引起的程序对象生存期问题放在下节讨论。这里先说程序单元中定义的块。Algol 60最早引入的分程序实则是块命令BEGIN D; C END; 即把一个语句(命令)扩大到可有自己声明集D，命令集C的局部块，前文我们已经介绍过了。并且把这个局部性的思想用于表达式(表达式中有局部声明)即块表达式，这里进一步问，声明集也是一个块，能不能再带有声明呢？回答是肯定的。

块声明是含有局部声明的声明，局部声明确立产生的束定仅用于块声明。

事实上有此需要，现代语言ML、Ada、C++都扩充了块声明，请看下例：

例5-13 ML的块声明

```
local
  fun multiple (n: int, d: int) = (n mod d = 0)
in
  fun leap (y: int) = (multiple (y, 4)
    andalso not multiple (y, 1000))
    orelse multiple (y, 400)
end
```

本例声明一求闰年函数leap，它给出函数定义，只要输入整数年即可判断是否闰年。函数体中用到multiple函数，它在local ... in之间定义(局部声明)。块外的用户只能见到leap，而multiple是见不到的，后者只为本块服务。所以它是声明的声明。前述例5-9是Ada块声明的例子，该例定义的Password类型、常量和两函数包外用户都可见，唯独私有类型PASSWORD本身的定义不可见(放在private中)。它是其它声明的辅助声明、读者就会看到块声明的好处：控制外部的可见性，使程序更清晰、简明。修改私有类型定义对外无影响。

5.4 束定的作用域与释义

前面我们介绍了束定的概念、机制及类别，声明即为预计的束定集合等一般概念。束定作用域复盖是造成语言释义混乱的根源，本节我们研究这个问题。

5.4.1 束定与环境

声明是给出预期的束定，而束定总是在以前束定过的标识符的基础上进行，即承认已有束定。我们把以前的束定集的作用域范围称为环境(environment)，当然环境还包括系统预定义的字面量，关键字，特殊符号(从符号学意义上的早期束定)。一旦对某标识符作出束定，它也成为新环境的一部分。环境用以解释程序中的一切活动，例如有声明：

```
const c=7;
```

将标识符C与常整型存储对象束定，其中放值7。完成束定(确立)后，c就是常量程序对象'7'的指称。

再如：

```
var c: Char
```

c就是字符变量的程序对象的指称，两个c束定不同是它所在的环境不同，当然同一环境下，同一标识符不能有两处束定(请注意，同一程序同一标识符可声明多次为什么？)。环境是束定集合(实现世界)，也可以说，环境是有效声明的集合(程序世界)。以下是Pascal环境示

例。

例5-14 Pascal程序的环境

| | |
|--|---|
| <pre> PROGRAM P; CONST z=0; VAR c Char; PROCEDURE Q; CONST c=3.0e6; VAR b: Boolean; BEGIN (2) END; BEGIN (1) END. </pre> | <p>(1) 处环境为:</p> <p>z→整常量; c→字符变量 Q→过程抽象</p> <p>(2) 处环境为</p> <p>c→实数3000000.0 b→布尔变量 z→整常量 Q→过程抽象(如有递归)</p> |
|--|---|

5.4.2 词法作用域与动态作用域

我们把程序正文给出的嵌套声明作用域叫词法作用域(Lexical scope)嵌套块是词法子辈,被嵌套块是父辈,它们的作用域相互复盖,以最近原则束定。Pascal是词法作用域的语言。

例5-15 以词法作用域解释本程序结果

```

PROGRAM A;
  VAR x, y: Integer;
  FUNCTION B(d: Integer): Integer;
    BEGIN B:=x+d END;
  FUNCTION C(d: Integer): Real;
    VAR x: Real;
    BEGIN x:= 5.1; C:=x+B(d+1) END;
  BEGIN x:= 3; y := 0
    Writeln (C(y))
  END.

```

读者如跟踪其打印结果C(0)=9.1(当x=3, y=0)。

词法作用域一般以上章介绍的运行时堆栈和堆栈帧实现。编译后的执行代码和全局量先入运行时堆栈成第一个堆栈帧,如图5-5所示。当程序执行进入子程序或块时,它的参数,局部变量另辟堆栈帧存放。本例为函数C的d, x。当然还要包括子程序的返回地址(动态父辈,并且成动态链)和指示词法父辈的指针(静态链)。静态链以全名束定实现,它提供了词法作用域描述。

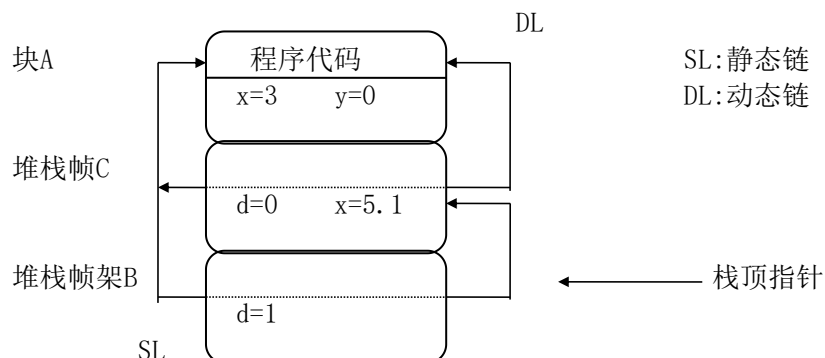
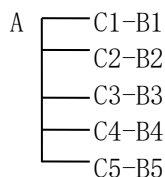


图5-5 运行时堆栈

本例中词法作用域是A<B, C。C, B之间的引用是兄弟间引用。动态调用链是A-C-B。这个例子过于简单，但已看出差别。如果A的语句中有循环（例如五次）则动态链是：



每次新的 $B_i - C_i$ 覆盖老的 $B_{i-1} - C_{i-1}$ 。如果C中又有引用B，则形成递归，其动态链是：A-B-C-B-C-B-C.....但词法作用域依然不便。

每个程序的程序对象一旦交付编译，它的名字个数是确定的，唯独递归定义可在运行时创建无数个堆栈分配的对象。每次递归地调用本函数都要作新的存储分配形成新一层堆栈帧。新的存储块（堆栈帧）一直保留对参数和局部量的束定，直至该程序块出口。我们说这个堆栈帧是动态作用域。

递归函数中声明的一个标识符对应为不同动态作用域中的好几个对象。递归五次就会有六组存储对象束定到同一组局部量的标识符上。块结构可以利用全名束定解决一个标识符多次声明的问题。利用词法作用域可在编译时查清。这个办法在递归情况下就无能为力了。因为每次调用的都是“同一个”词法块。只能采用动态作用域的分辩规则：只用最当前的活束定。从当前的堆栈框架开始找某个标识符的束定，如果找不到，则沿动态链找调用它的堆栈帧，直至沿动态链找到最初的一块有该标识符定义的束定为止。

5.4.3 词法作用域和动态作用域的求值差异

词法作用域规则和动态作用域规则所产生的计算结果只有在非常简单的情况下是一致的，而多数情况下不同。以下例子是和例5-15完全一样的程序，只是所用作用域规则不同。

例5-16 LISP动态作用域

```

(defun A                                     //定义一个函数名为A
  (x, y)                                     //具有两个参数x, y
  (printc( C y)))                           //函数体是打印函数printc;
                                              它打印函数C应用到y上的结果值

(defun B                                     //定义一个函数B
  (d)                                       //只有一个参数d
  (+d x))                                   //函数体是d和x相加的返回值，注意，
                                              x未定义，它是全局参数引用。

(defun C                                     //定义一个函数C
  (d)                                       //只有一个参数d
  (let (x' 5.1))                           //设置局部参数x的值为5.1
  (+x(B (+d 1))))                           //函数体是把局部的d加1，后结果值应用
                                              其返回值再加局部的x，得C的返回值。

(A 3 0)                                     //把以上定义的函数的A应用于实参是3，
                                              0的调用。
  
```

其引用过程是：

```

(A 3 0) =(printc (C 0))                    //x=3, y=0
        =(printc(+5.1(B(+ 0 1))))          //x=5.1, d=0
        =(printc(+5.1 (B(1))))
        =(printc(+5.1(+1 5.1)))            //d=1, x就近取值=5.1
        =(printc(+5.1 6.1))
        =(printc 11.2)                     //例5-15是9.1
  
```

这两种作用域的束定机制当有标识符多次定义时结果不同。Pascal, C, Ada按词法作用域将标识符换成全名, 只有递归情况下, 无法换成全名, 才按动态作用域释义。所以它们两者都用。

5.4.4 作用域与生命期匹配的问题

名字的作用域和它所代表的程序对象的生命期并不是总是匹配的。最突出的例子是函数或过程调用中的引用参数传递 (VAR参数)。调用期间形参的名字与实参表示的程序对象束定。这个程序对象的生命期要比形参名字在其整个作用域的执行期要长。因为被调用的块要先出口, 自然比调用块寿命短。就过程或函数而言对象的生命期比名字作用域要长。这种情况还不止形、实参调用。

Pascal 和C中用指针动态分配的链表和树结构也有这种情况。因为这些指针是结构的成分可不在堆中分配。每当产生一新堆栈帧的结点时, 指向它的指针在其父结点的成分内填上地址就可以了。而指向该结构的头指针是在堆中分配的, 只要不除配它的寿命持久。还可以把它拷贝到外块寿命更长的指针上。

C语言的局部静态变量也正是利用这种不匹配, 达到数据保护的目的。静态量顾名思义和全局量一样, 静态分配到全局数据区。而它的名字的作用域仅限于该局部块。每当执行进入到该局部块, 这个静态变量的名字束定到静态分配的存储上, 此时是可访问的。每当出了该局部块, 名字失去定义。而又没有其它名字和它束定。任何局部块都不能访问它。数据安全保存。再次进入重新束定…。

类似的FORTRAN的公共语句静态分配的公共数据区, 其中的程序对象的生命期, 也比过程或函数模块内公用区中变量的名字作用域长。意思和C的静态变量相近, 但它必须共享, 不安全, 不如C的独享。

5.5 束定机制与语言翻译器

我们已经看到束定机制不同对程序语义有影响, 语言翻译器是采用编译型还是解释型对实施束定时采用什么束定机制也有影响, 本节讨论这个问题。

编译器和解释器最大的不同是解释器一面翻译一面执行, 因此它的可用信息比编译器在运行前编译要多多了。根据运行上下文解释器可以知道当前待分配的数据的数量和大小, 特别是存放大量数据的数组。编译器于运行之先就要分配存储, 它不得不每次都开辟较大的数组以放下最大可能尺寸的数据。若程序数据大小差异很大且变化难以估计, 往往耗费大量空间。

类型语言的类型声明给数据大小提供了准确信息, 所以编译器倾向于类型语言。数据束定解决了, 命令(语句)一般是比较容易的。一个程序块从进口到出口逐条增减堆栈分配的指针, 而无类型语言正因如此不宜编译(不等于说不行)。语法分析、符号表都可以照做, 只是存储管理要困难一些, 要动态管理。

解释器采用动态作用域的模式实现非常自然, 符号表在执行时是存在的。分配存储对象并将其地址和标识符束定的工作也在执行中做。因此, 对标识符引用的解释总是按执行上下文中的最当前的束定。一般说来, 程序的执行路径要取决于输入和条件(语句)求值情况, 是难以预测的。因此, 动态束定会因每次运行情况不同, 同一名字和不同存储对象束定。这样, 就会引起对全局引用作出不同的解释, 而且出了错极难追踪。

编译器中则正好相反, 符号表在编译结束后销毁(除非保留作排错而用), 所有存储对象都必须在编译时完成束定。对于那些一定要在执行中间分配的局部量, 编译器要决定它的地址, 即相对于当时的堆栈框架的首地址, 以便分配到正确位置上, 这样, 对于有多意的名字编译器必须保存一个与分配域堆栈平行的束定堆栈, 直到程序执行时为止, 每当翻译一个子

程序块，编译器就把局部束定压入堆栈，每当该子程序块编译完即可全部弹出栈。构成一个新堆栈帧。用于解释符号引用的束定总是在栈顶。这种翻译模式正好实现词法作用域：解释局部量的束定总是在最小封装块中定义。词法作用域是实现编译器的一种方法，编译器可以决定如何嵌套，谁是父块谁是子块，但不能知道执行顺序。

词法作用域的优点也是它的缺点：要事先知道如何束定，因而排错比较容易，再一个是执行效率高。

正因为词法作用域和动态作用域对程序语义有影响，作用域应该成为程序设计语言形式化的一部分。

本来一个程序设计语言的设计与编译实现还是解释实现没有什么关系。许多语言开发早期是解释型的以后由于运行效率难以容忍改成编译型，BASIC，LISP，Prolog都有这种经历。但正如上所述，语义有微妙差异，两种翻译器不完全等价。

5.6 小结

- 束定是将程序中代表程序对象的标识符和实现程序对象的存储对象联系起来的工作。标识符只有通过束定才能成为程序对象。
- 束定联系标识符和存储对象类似指针，但不是指针，它跨越编译—装载—运行阶段才完成。它必须由系统自动递引用。
- 静态束定(早束定)在运行前完成一旦束定不再改变。动态束定(晚束定)在运行期间实施，一个名字可动态束定到另一存储对象上。
- 符号表即实施束定最主要手段，它列出全名、类型、它所对应的被束定体的首地址。无类型语言将值类型附记在存储对象后。编译完成符号表消失。无类型动态束定的变量运行中类型也是可变的。
- 可以独立地与名字束定的存储对象叫可束定体。
- 声明就是束定，声明有定义、顺序声明、并行声明、递归声明四类。
- 静态束定语言一个标识符可声明多次，定义只能一次。动态束定语言一个标识符可以束定多次，因而它们表征的程序对象不同。
- 声明的有效范围叫作用域。一个标识符在其作用域内用简单名即可引用叫可见，否则叫掩蔽。块结构语言都采用最近声明有效原则。被掩蔽的标识符要用作用域分辨符。
- 块声明是带有声明的声明，它使部分声明隐藏。
- 环境是束定集合的有效范围。声明指明新的束定集合，它在原有环境内指定，声明确立构成新环境。
- 当一个标识符在一个程序中有多次束定时，相互复盖的作用域造成释义混乱，可以按词法作用域理解并进行束定，也可以按动态作用域理解和进行束定。但两者的释义结是不同的。
- 递归过程(无论是递归函数还是递归数据结构)只能按动态作用域束定和释义。嵌套块结构语言按词法作用域束定和释义。把标识符和全名分开是块结构语言解决名字冲突的办法。
- 标识符和它所代表的程序对象的生命期不一定匹配。在块结构和嵌套块结构的语言中这是常见的。C语言的静态存储类就利用了它的优点。
- 语言的翻译器不因解释型和编译型对语言释义有所不同。但束定的词法作用域和动态作用域对语义确有影响。编译型倾向有类型、嵌套结构语言并按词法作用域。解释型倾向于动态作用域。

习题

5.1 程序对象、标识符、名字、存储对象它们各自的含义并有什么关系？

答：程序对象是名字和存储对象的结合体；

标识符和名字都是用以代表程序对象的，标识符是一字符串用于命名程序中某个(语法)元

素。名字也是字符串用以表示程序中的实体。以使人们从语义角度使用它。一般情况下，名字对应一简单标识符。有时代表多个构件的组合。

存储对象是机器内部程序对象的具体实现。

关系：一个名字可对应几个程序对象，一个程序对象也可对应多个名字，这些是由不同的束定机制来决定的。

5.2 符号表有什么作用？怎样加入新名字？

5.3 程序对象的语义如何体现？

5.4 无类型语言的‘变量’是如何声明的？

5.5 何谓束定？束定和指针、引用有何不同？

答：束定是将名字（标识符）和可束定体联系起来，所谓可束定体是能反映出语义的存储块。如常量、变量的存储体、函数体、过程体、类型和异常。

或者束定是将程序中代表对象的标识符和实现程序对象的存储对象联系起来的动作。

指针是程序可以令其指向任何程序实体，引用是常指针。

不同：①束定是编译（或解释器）做的。，为每一个名字分配其语法要求的存储，也以跨越时间，编译时占个位置，运行时再分配（实现束定）。

②翻译器可以自动递引用束定而不能自动递引用指针。

5.6 何谓动态、静态、块结构束定，比较它们的同异。

5.7 用束定能代替赋值吗？为什么？

5.8 在一个块中各声明的标识符其生命期一样长吗？程序语言的编译(解释)器怎么知道？

5.9 有以下C程序片段

```
int x, y, z
fun1( )
    int j, k, l;
    int m, n, x;
    ...
    fun3 ( )
    int y, z, k;
    ...
main( )
    ...
```

它是嵌套块程序吗？如何嵌套，用框形图绘出示意，标以A, B, C...为块名。

a. 定义了多少名字作用域，指出每个名字作用域的起止。

b. main, fun1, fun3中各能访问什么变量？

c. 哪些变量作用域相互复盖？

d. 变量j是全局量还是局部量，为什么？

e. 在哪个块中能同时使用k, x两个变量，若有两外以上它们的使用意义一样吗？ 5.10 何谓名字冲突？不冲突不行吗？

5.11 指出以下Pascal程序各标号处的束定环境：

```
PROGRAM A;
  CONST x = 999;
  TYPE Nat=0..x;
  VAR m, n:Nat;           (1)
  FUNCTION f(n:Nat): Nat
  BEGIN
    (2)
  END;
  PROCEDURE w(j:Nat);
```

```

(3) CONST n=6;
BEGIN
    (4)
END;
BEGIN
    (5)
END.

```

(1) 处环境 $x \rightarrow$ 整常量999
 $\text{Nat} \rightarrow$ 类型 $0 \dots \text{maxint}$
 $m, n \rightarrow$ nat类型变量

(2) 处环境为 $x \rightarrow$ 整常量 (3) 处环境 $x \rightarrow$ 整常量999
 $m, n \rightarrow$ nat类型变量 $m, n \rightarrow$ nat类型变量
 $j \rightarrow$ nat类型变量

(4) 处环境: $j \rightarrow$ Nat类型变量的参数 (5) 处环境: $x \rightarrow$ 整常量
 $n \rightarrow$ 常量6 $\text{Nat} \rightarrow$ 类型变量
 $x \rightarrow$ 整常量999 $m, n \rightarrow$ nat类型变量
 $m \rightarrow$ nat类型变量度 $f \rightarrow$ 函数抽象
 $w \rightarrow$ 过程抽象 (如有递归) $w \rightarrow$ 过程抽象
 $f \rightarrow$ 函数抽象 (如有调用)

5.12 何谓束定出现? 应用出现? 函数抽象的 $F(x: T)$ 的参数 x 是什么出现?

5.13 试述块声明的作用? C++的类声明是块声明吗?

5.14 为什么递归定义只能用动态作用域?

答: 动态作用域是指新的存储块(堆栈帧)一直保留对参数和局部量的束定, 直至该程序块出口才释放的堆栈帧, 递归定义的函数在每次递归调用本函数时都要作新的存储分配形成新一层堆栈帧, 所以只能用动态作用域。

5.15 C语言的联合、Pascal和Ada的变体记录、FORTRAN的等价语句, 一个存储对象与多个标识符束定, 束定的是一个程序对象吗? 如果不是它们的作用域复盖怎么没有区分问题呢?

5.16 试举一例说明常量非恒值的用途。怎样避免常量非恒值。

5.17 C++中声明也是语句, 可以出现在任何语句出现的地方。它如何实现束定? 动态/静态?

5.18 对于非并发语言并行声明有什么作用?

5.19 用FORTRAN能仿真作出C的静态变量吗? 如果行, 试述仿真要点。如果不行说明原因。

5.20 试述标识符的作用域和它所代表的程序对象寿命不一的根本原因。

第6章 函数和过程

程序表达的计算本身近似于代数演算，演算在符号上进行。只要给出实际的输入值即可得到输出值或所需要的计算机动作。一般说来，程序是通用的，适合于一组输入值。如果把程序中某一段实施某种功能的计算单独划出来，给它取个名字并给出本段所需的参数，即将名字束定于执行某种功能的代码块。在程序世界里这段程序叫子程序。50年代人们就理解了这种分割的好处：它实施的功能单一，便于调试；它相对独立，便于多人分工完成，且时间不受约束；它相对封闭，所有数据从接口出入，人们易于控制，是分解复杂性的有力措施。

命令式语言中子程序有两种形式：函数(必须返回值)和过程(实施一组动作)，有时叫函数过程和子例程subroutine。它们是程序的第一次分割。

子程序开创程序世界里局部性、模块性的先河。由于子程序是程序的一个相对独立部分，它和主程序联系的接口特别重要。在这个界面上要指出该例程的数据特征，即输入什么输出什么。而整个子程序体是完成从输入到输出的实现手段。因此，界面指出“做什么？”而子程序体回答“怎么做”。于是，在80年代程序完成第二次分割：将子程序定义(即界面)和子程序体显式的分开，成为相对独立的规格说明(Specification)和体(body)。这样在程序设计的早期只写规格，详细设计时再选算法、数据结构实现它的体。只要规格说明不变，即使过程体作重大的修改或换掉也不会影响整个程序系统。Ada 和C++即是完成第二次分割的实例。

本章我们首先研究函数和过程的定义和调用。关注的焦点在第二章中讨论，它们的接口和数据传送机制。

原则上所有的程序值都可以作参数传递，第三节讨论实参求值策略，最后讨论参数本身为函数或返回值及函数的高阶函数。

6.1 函数和过程抽象

函数抽象是用一个简单的名字抽象代表一个函数。该函数由函数型构(Signature)和函数体(body)组成。函数计算的目的是求值。函数体等同于一个复合的表达式。所以，函数抽象是对表达式的抽象。同样，过程抽象是用一个简单的名字抽象代表一个计算过程。该过程由过程型构和过程体组成。过程调用的目的是执行一组命令以更新数据，过程抽象是对命令(即语句)集的抽象。

6.1.1 函数定义与引用

函数定义定义了一个程序世界的函数，它有名字，输入、输出参数(组成函数型构)，和实现函数从输入到输出的映射的函数体组成：

```
function FUNC (fp1, fp2, ...): returntype; //函数型构
```

```
B; //函数体，可包括任何声明和语句
```

函数型构中fp1, fp2, ...为形式参数，也叫形式变元(argument). returntype为返回值类型，函数引用是应用函数的唯一手段，它在同名的函数名之下给出实在参数(实在变元)：

FUNC (ap1, ap2, ...);

其应用过程是：先匹配函数名，找到函数定义模块，然后匹配形、实参数。匹配后程序控制转而执行函数体。实参以形参别名参与运行直至函数体结束，或至显式的Return语句。主调程序的引用处此时得到一函数返回值。函数引用可出现在任何求值的表达式中。返回的参数值在程序的下文显式引用。

(1) 函数定义的形式

早期语言函数定义和子程序一样，都是过程的一种。但函数要返回值，于是函数名同时起到携带返回值的作用，函数返回值的类型加在函数关键字之前，如例6-1(a)。函数名至少要出现在赋值号之前一次；局部量和参数的类型在型构之后语句开始之前声明；至少要有有一个RETURN语句。

Pascal和Ada函数返回值的类型放在型构后，且参数类型在参数表中声明，和局部量声明明显分开，且Return只作中途返回之用，不用Return从块末出口也可以。由于Pascal支持递归，它承袭了函数名至少在赋值号左边出现一次，于是造成名字含义不同，见例6-1(b)，左边的函数名是变量引用，右边是函数引用。Ada（见例6-1(d)）就避免了这个缺点：函数名仅仅是函数名，返回值由return关键字后跟的表达式表示，其求值结果为某个临时的中间变量，程序员不可见。return机制同pascal。这样，和充当块结束括号的关键字end分工也明确了（一指程序行文结束，一指执行结束）。

C语言，见例6-1(c)，在函数型构和局部量声明的形式与Pascal同。用return加表达式指明返回值与Ada同。但由于C只有函数形式，若为主函数或过程则无return语句，且缺省的函数返回值类型等同于int型。ANSI C和C++进一步吸取在参数表中指明参数类型的优点。

例6-1 各种语言函数定义

(a) FORTRAN的函数定义

```
INTEGER FUNCTION FACT(N)
  INTEGER N, I, F    //参数类型在此声明
  F = 1
  DO 10 I = 2, N
    F = F*I
10 CONTINUE
  FACT = F          //必须至少定义函数名一次
  RETURN            //至少有一返回语句
END
```

(b) Pascal的函数定义

```
FUNCTION fact (n:Integer) :Integer;
BEGIN
  fact := 1;
  IF n = 1 OR n = 0 THEN
    Return
  ELSE
    fact := n*fact(n-1) //两fact意义不同
  END.
```

(c) C的函数定义

```
int fact (n) {    //和fact(n)等效
  int n;
  int i, f;
  f = 1;
  if(n>1)
    for (i = 2; i<= n; i++)
      f *= i;
  return (f); } //返回表达式f的结果值
```

(d) Ada 的函数定义

```

function FACT (N: POSITIVE) return POSITIVE
begin
  if N = 1 then
    return(1);
  else
    return (N*FACT(N-1));
  endif;
end FACT;

```

(e) ML的函数定义

```

fun fact (n :int) =
  if n = 1 then 1 else n*fact (n-1)
val fact = fn(n:int) =>
  if n = 1 then 1 else n*fact (n-1)

```

ML是函数式语言，见例6-1(e)，它的函数定义即为对表达式的求值。示例中给出函数抽象的型构后，函数体为一条件表达式。在函数引用处得该表达式结果值(临时中间变量传递)。参数类型在参数表中声明。即使是ML的值定义也用函数定义实现，因函数必然求值，此时则用抽象符号fn代表函数，指明参数后用‘=>’带出函数体，以便和‘=’带出的函数定义的函数体区别。语义是一样的。

(2) 引用或调用的形式

函数引用源于代数中参数置换的思想。以同名的函数名带实参表，运行时置换形参表，谓之参数结合(association)。显然，形—实参数表中元素个数，次序，类型应一致。早期语言都严格遵此准则。近代语言提供了较多的灵活表示法。

例6-2 Ada的参数结合表示法

本例为一自动洗衣机控制程序的函数型构部分。设事先已将CLOTH_TYPE(衣服类型)、SOIL_TYPE(弄脏类型)、ERROR_TYPE(错误类型)作了枚举类型定义。

```

function WASH_CYCLE
  (WEIGHT      : in INTEGER;
   FABRIC      : in CLOTH_TYPE := COTTONKNIT;
   SOIL_LEVER  : in SOIL_TYPE := AVERAGE;
   DARK        : in BOOLEAN := FALSE;
   DELICATE    : in BOOLEAN := FALSE;
  ) return ERROR_TYPE is...

```

其中五个形参四个给出初值。Ada允许以下引用：

```

RESULT := WASH_CYCLE (5, WOOL, SLIGHT, TRUE);
  --5公斤毛织品，不太脏，深色，不娇贵的衣服
  --实参与形参按位置一一对应。最后一个有缺省值则不另重复。
RESULT := WASH_CYCLE(10);
  --10公斤，棉织品，一般程度的脏，浅色，不娇贵
  --除第一参数外其余全用缺省值
RESULT := WASH_CYCLE(8, SOIL_LEVEL=>FILTHY);
  --8公斤，棉织品，脏得很，浅色，不娇贵
  --第三个是指名参数结合，第一个按位，其余按缺省。
RESULT := WASH_CYCLE(WEIGHT=>2,
  DELICATE => TURE,
  SOIL_LEVEL => SLIGHT,
  FABRIC    => WOOL,
  DARK      => TRUE);
  --2公斤毛织品，深色，不太脏但娇贵
  --全部用指名结合，故参数次序可以随意颠倒。本例未用缺省值。

```

除此而外，C语言允许任意多个参数的调用。例如，内定义函数printf()调用时可以写：

```
printf("The sum is %d and the average is %d\n ", sum, sum/count);
//第一参数中两个%d指明它有两个实参sum和sum/count。
printf("%3c %3c %3c %5d %17e %17e\n ", c1, c2, c3, k, x, y);
//第一参数指明有六个实参要打印。c1, c2, c3是占3位的字符,
//k为占5位的整数, x, y为占7位的长浮点数
也就是说, 它的参数个数随意, 但必须和第一参数中格式符个数、位置对应。
```

6.1.2 过程定义与调用

过程子程序定义形式与函数定义极为相似:

```
procedure PROC (fp1, fp2, ...) //过程型构
      B; //子程序体包含局部声明
```

对应的过程调用是:

```
PROC (ap1, ap2, ...);
```

早期语言在过程名前加关键字'CALL', 近代语言取消。过程子程序的调用与函数过程同。因不返回过程值, 故不能出现在表达式中, 只能作为单独的命令(语句)。过程返回的修改值只能在参数表之中。因而参数表是它的输入/输出必由之路。既然有此区别, 近代语言把函数和子程序的区别有意拉大, 不主张对函数参数表的值作修改, 只返回函数的结果值。如Ada函数的参数表只有输入模式(in)没有输出模式(out)。

(1) 多重入口和指定返回

早期语言强调算法, 一个复杂的科学计算过程可以有上千语句。为了充分利用部分程序代码, FORTRAN设有ENTRY入口语句, 如例6-3。

例6-3 FORTRAN 的多重入口示例

```
SUBROUTINE DEG (R, THETA, X, Y)
      C = 3.14159/180.0
      THETA = C * THETA
      ENTRY RAD (R, THETA, X, Y)
      X = R * COS(THETA)
      Y = R * SIN(THETA)
      RETURN
END
```

若THETA是度数值时, 则调用语句为:

```
CALL DEG (R, THETA, X, Y) //入口在子程序顶部
```

若THETA是弧度值时, 则调用语句为:

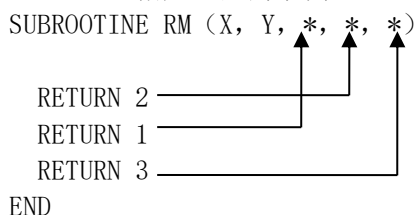
```
CALL RAD(R, THETA, X, Y) //入口在子程序中部
```

这个子程序有两个入口, 多一个ENTRY, 当然可有多个入口。多重入口违背结构化一入口一出口的要求。

FORTRAN除了多重出口而外, 还有指定返回, 如下例:

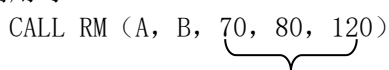
例6-4 FORTRAN指定返回的示例

```
SUBROUTINE RM (X, Y, *, *, *)
      RETURN 2
      RETURN 1
      RETURN 3
END
```



主程序调用时

```
CALL RM (A, B, 70, 80, 120)
```



本程序块中的语句标号

END

多个出口的多重返回直到近代语言都是允许的(为什么?)FORTRAN语言的规定把语句标号作形-实参数,依次RETURN i(i = 1, 2, 3, ……)返回到主调程序指定的实参标号上(按形参顺序)。即它调用后返回不一定是调用处,而可到任何指定的位置,但这种方便带来程序控制混乱,相当于goto语句。近代语言无论从哪个地方返回都返回到调用处。

(2) C语言的过程子程序

C语言虽然没有子程序形式,一切过程,包括主程序都是函数过程。它以void(无值)关键字代替函数类型指明符,实施子程序过程语义。因而这些过程中没有return语句。有时程序员写一子程序过程忘了加上void,它仍会返回值1。因为缺省类型指明符的函数是int型,执行成功又未指定返回什么,它就返回整数1。

6.1.3 无参过程

函数和过程的参数表均可为空。有的语言保留(),有的干脆只有一个名字。对于充作主程序的无参过程很好理解,它不被调用,只调用其它过程。然而,一般被调用的过程,若不给出调用参数那么两次调用不是同一结果吗?除了少数过程子程序每调用一次执行同样的几个动作而外,一般无参过程也要更新过程内部的值。函数过程还会返回不同的值。其原因在于函数在包容它的主调程序的作用域之内,全局量在函数中有效。改变了全局量两次调用结果值当然不一样。这就是函数的副作用(side effect)。

例6-5 有副作用的函数

C在很大程度上利用函数副作用,例如,当需要跳过空白时写:

```
while ((c = getch()) == '');
```

当c为字符时出循环,若无副作用则为死循环。因第一次调用getch()为空白,永远为空白,才是无副作用的。再如,Ada中常用的随机数:

```
function RANDOM return FLOAT range 0.0...1.0;
```

引用时,若FIELD已声明为常量:

```
RESULT := RANDOM*FIELD;
```

RANDOM若无副作用两次引用RESULT值不可能改变。

如果在程序员没有想到的地方出现了函数副作用,则程序错误是极难查出的。无参函数极易产生副作用。当然有参函数也会产生副作用,即同样参数的两次调用结果值不一。读者能举出一例子吗?

6.2 参数机制

参数(parameter)因环境条件可变的值的系统,在函数或过程中,它既可以是常量也可以是变量。给定一组参数值,函数或过程就有不同的计算(结果值)。我们把传递到函数或过程上的值叫做变元(argument)。相当于数学函数自变量取值。变元一般指实参。

程序中哪些值可以作变元呢?应该说所有类型值均可作变元最好,但多数语言不行。例如Pascal的文件类型值,Ada的函数或过程抽象及文件都不能作变元,其它初等量、复合量、指针、变量引用均可作变元。ML的任何值都可以作变元。

由于前叙变量的时空特性,传递变元的形-实参数可以有许多不同的实现结合的办法,即所谓参数机制。不同的机制使得同一函数得出不同计算结果,我们要仔细分析它们。为了简化我们‘函数引用’都叫‘调用’。把‘过程子程序’叫过程。

6.2.1 传值调用

形参表和实参表不管表示法如何，它们在结合时类型、个数总要一一结合(用缺省值补足)。它们相结合最简单的办法是把实参值拷贝到形参上，所以也称为值调用(call-by-value)。值调用语义明晰，容易实现。结合步骤是：

[1]实参表达式先求值。

[2]将值复制给对应的形参。显然，除指针指向的存储不复制而外(只复制指针值)，形参和实参有同样大小的存储。

[3]过程运行后一般不再将改变了的形参值复制回实参。因实参如是表达式是复制不回去的。所以，实参成了只读参数。

只读参数的好处是在子程序内对形参的任何修改都不会传到主调程序中，是限制函数副作用的最有力的机制，程序模块干净，数据出入清晰，但耗费存储。为了减少双倍的存储开销，对于大的链表和记录则以指针作为形实参。

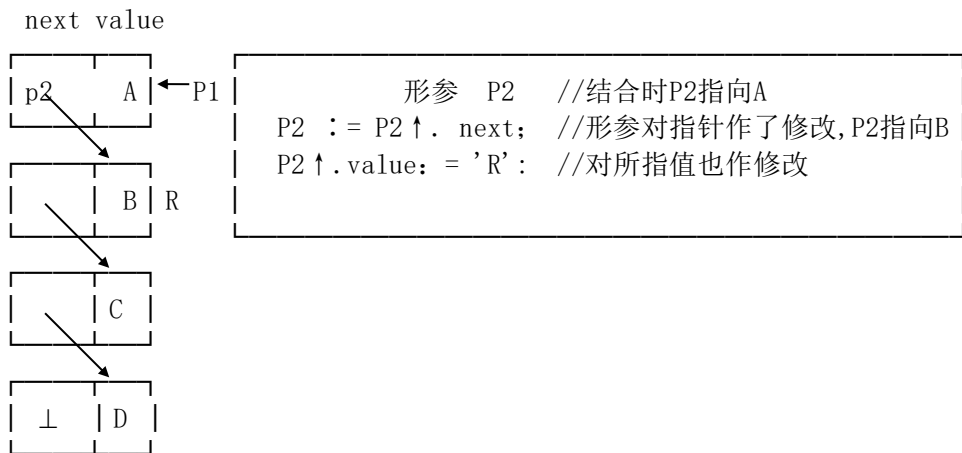


图6-1 指针复制机制

但在复制机制下，虽然子过程对指针值的任何修改都不会影响实参指针原有的指向，如图6-1，形参P2与P1结合后它指向主调程序链表。此时子程序第一语句将P2改为指向下一元素(值为'B')，返回时P1仍指'A'处。然而，对形参指针指向的内容作了修改，是会影响主调程序的数据值的。如例中把P2当前指的内容换成'R'，返回后链表中的'B'也被改为'R'了。

我们用下例演示传值调用的图示6：

例6-6 Pascal中的传值调用

```
PROCEDURE test1(J2,A2:Integer;P2:list)
BEGIN
  Writeln(J2,A2,P2↑.value);
  J2 := J2 + 1;
  P2 := P2↑.next;
  Writeln(J2,A2,P2↑.value)
END.
```

调用程序有：

```
test1(J1,A1[J1],P1↑.next);
```

其调用图示如图6-2

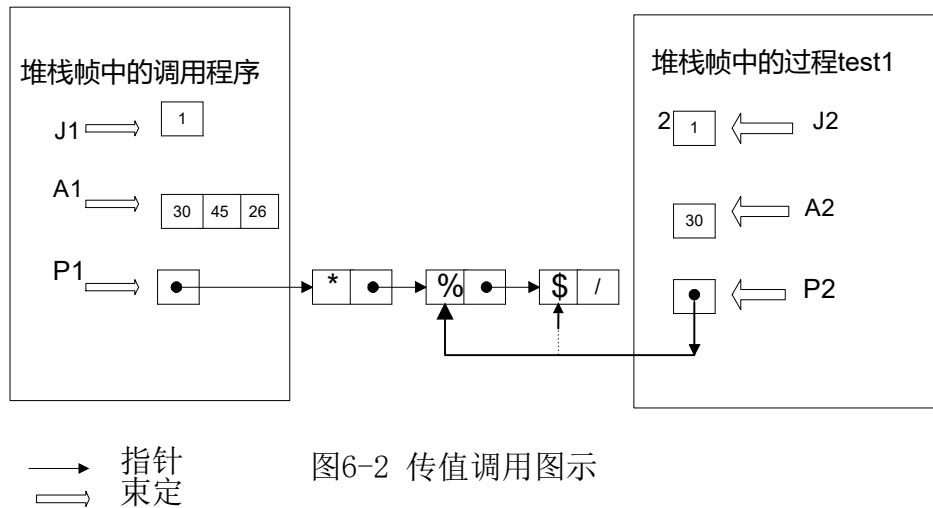


图6-2 传值调用图示

调用程序中J1, A1, P1束定于存储对象并赋初值如图示。J2, A2, P2编译后束定自己的存储对象，复制结合后按过程中语句

第一次打印为：1 30 %

第二次打印为：2 30 \$

返回后调用程序中值未变。

6.2.2 传名调用

传值调用只能返回单个的函数结果值（早期语言必须给函数名赋一次以上的值）。如果是个交换两个数的过程，交换的结果无法返回。于是Algol想到传名。变元的名字就是地址。如果在过程内修改形参的值，它就按结合的变元的名字（地址）找出变元值修改之，返回后实参值不就改过了吗。Algol既实现传名也实现传值。传名在过程/函数中加工的就是实参已分配的值，因此不需付出双倍存储代价。但传名过程是虚实结合时将程序体中所有形参出现的地方均以实参变元名置换。在函数/过程执行中先换变元名再算值，这样出现几次算几次效率是低的。此外，如果计算变元表达式时（如数组元素）如对其他值有影响（即副作用），则副作用要扩大影响多次。图6-3和例6-7是图6-2和例6-6的同一题目的对比：

例6-7 传名调用程序示例

由于Pascal无传名机制，此处作一点扩充：

```
PROCEDURE test2 (NAME J2, A2: Integer; NAME P2: List);
  函数体同test1
```

执行同样调用：

```
test2(J1, A1[J1], P1 ↑ .next);
```

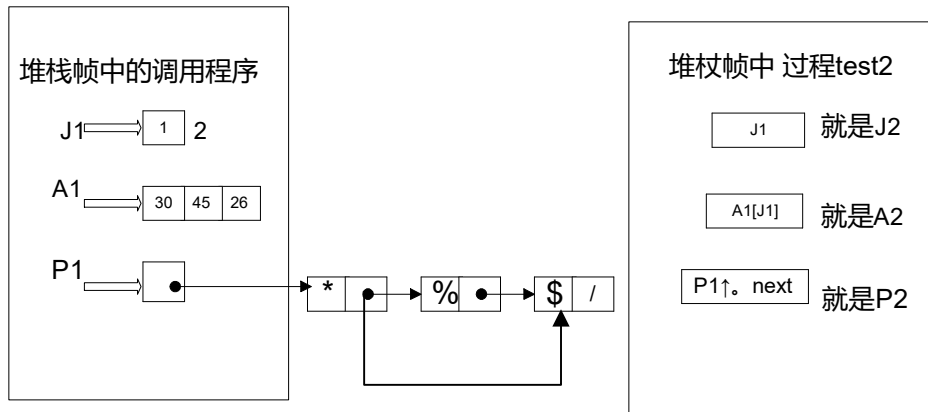


图6-3 传名调用图示

名结合后如右图所示，此时打印：

1 30 %

执行后J1的值为2，A1[2]的值是45，将P1↑.next的指针换指下一个‘\$’值，结果是：

2 45 %

名调用在发明引用调用后已不再采用，但C语言的宏，在预处理宏扩展中仍按名调用。

6.2.3 引用调用

Algol 60的传名机制，即将程序对象名字(地址)直接传给子程序。后来发现除了简单变量名外总会有副作用使程序难以控制，Pascal则采用引用调用(call_by_reference)或称地址调用(call_by_address)，即Pascal的VAR参数。引用参数传递的是存储对象。实参必须是变量名或能得出地址的表达式，它把实参名束定的存储对象改为形参名束定。

引用参数实现时，编译器在子程序堆栈中设许多中间指针，将形参名束定于此指针，而该指针的值是实参变量的地址(在主程序堆栈框架内)，于是在子程序中每次对形参变量的存取都是自动地递引用到实参存储对象上。引用参数的束定实则是设常指针，对程序员是透明的。引用参数在子程序中占空间少，递引用比复制效率高，所有对形参变量的更改直接反映到实参存储对象上。这点和传名是一样的，只是减少了不必要的副作用。图6-4和例6-6是前述问题的对比。

例6-8 引用调用的Pascal过程

```
PROCEDURE test3(VAR J2, A2: Integer; VAR P2: List);
```

函数体同test1

相应的调用程序是：

```
test3(J1, A1[J1], P1↑.next);
```

其中J1、A1、P1取值和上例一样如下图。编译时J2、A2、P2在子程序中束定于三个指针对象，虚实结合时它们指向参数(填入地址)，也就是完成了束定，执行test3的第一次打印是：

1 30 %

第二次打印是： 2 30 \$

运行后J1值变为2，P1的链表变为了虚线的。出了test3, 该框架消失。

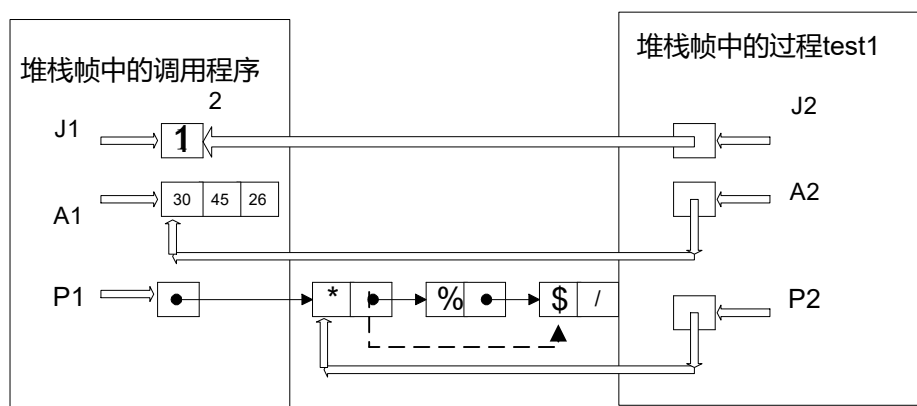


图6-4 引用调用图示

FORTRAN所有的参数传递全是引用调用，C语言的数组参数也是，其它参数和Pascal一样由用户指定 (VAR 或&)。Ada的输出模式(out)参数也是用引用调用实现的。

6.2.4 参数模式与返回调用

传值和引用各有特点, 程序设计语言往往同时用到。传值实现只读参数很方便, 引用机制可实现读-写参数, 只写参数可用引用实现但不安全, 特别是当程序员误用时编译器是无法知道的。例如, 在子程序中给值调用的参数赋值; 在多机系统多道程序中, 程序返回参数被无意修改, 还有多重束定在并发程序中出现的时序控制问题都迫使近代语言进一步完善参数传递机制。

显式指明参数传递模式, 可以为编译实现提供信息。例如, Pascal的参数表显式指明的表示法较FORTRAN的改进多了:

```
fun_name(x, y:Real; VAR s, q:Integer)...
```

一看就知, x, y传值实现, 它只读。s, q引用实现, 可读/写。Ada倒反不限定如何实现它只规定参数模式in, out, inout, 即只读、只写、读写三种参数传递方向的模式(mode), 这更有利于程序员控制程序的语义。表示法为:

```
PROC_NAME (X,Y:in Real;  
           S:inout Integer;  
           Q:out Integer)...
```

in模式可不写出(缺省)。函数只能有in的模式, 过程都有, 且出现次序不受限制。这样, x, y因在子程序中只读, 传值实现可保证不受破坏。s读/写用引用实现, 而q是只写参数, 传值和引用都不能保证“只”写。于是有返回调用(call_by_Return)机制。

实现返回调用机制有两种办法: 其一是复制。主调程序开设和返回数据一样大的实参空间, 子程序运行完毕将局部变量写入只写参数, 拷贝回去。另一种办法是引用实现将参数束定于实参变元, 但增加“只写”保护。

6.2.5 值--返回调用

看起来返回调用似乎比引用调用笨拙, 但是安全。特别是在多进程、并发程序。并发程

序中引用机制的不安全可说明如下, 若有两进程P1, P2都接受了实参变元ARG, 作为写参数, 则:

- 若P1先执行完P2执行, ARG最终值按一般顺序执行理解, 先是P1的, 最后是P2的。
- 若P1开始未完P2又开始了, P1结束后P2再结束, 则P1对ARG的加工会被P2加工复盖, 则结果按P2。采用引用只写参数变成读/写参数, P2输出不可靠。
- 若P1开始未完又开始P2, 但可能P2先结束再P1结束, 则P2对ARG的加工为P1的加工复盖。结果按P1。同样结果, 若采用返回机制时, 得出的P1是可靠的。

采用值与返回调用机制(call_by_value_and_return)是把值引用和返回调用组合起来, 以实现调用程序和被调用程序双向通道, 这对于有多个存储器的多处理器系统和网络分布式系统值调用极度安全, 返回调用可靠。有利于分布式并行计算的复杂性控制。在子程序执行期间因不是束定, 形参变量的值不会中途改变, 复制回去和拷贝进来处可设断点检查。因此, 值与返回调用被认为比束定机制更适合于并发程序。

6.2.6 指针参数

指针作为参数其实现方式一般是复制机制, 它复制的是地址(指针内容)。这样, 主调程序的指针和子程序指针同指主调程序中的同一存储对象。指针调用(call_by_pointer)和引用调用都可以实现读-写参数。虽然它们都是用指针实现(一为显式、一为隐式)的, 但指针调用容易引起悬挂指针, 而引用调用的隐式指针随引用块的堆栈帧消失而消失。相对安全。以下举例说明。

例6-9 Pascal的引用调用和指针调用

引用版: 交换两变量的内容

```
PROCEDURE swap1( VAR a,b:Integer);
VAR t:Integer;
BEGIN
  T := a; a := b; b := t
END;
```

调用程序片断:

```
j = 3; k = 5;
swap1(j,k);      //结果j = 5, k = 3
```

主程序堆栈帧中有j = 3, k = 5, 子堆栈帧中a, b束定于j, k, 并有局部量t, 执行后t中值为3。其图示读者可参照图6-4给出。

指针版: 同上题

```
TYPE int_ptr = ↑ Integer;
VAR jp, kp:int_ptr;
PROCEDURE swap2 (a,b:int_ptr);
VAR t:Integer;
BEGIN
  T := a↑; a↑ := b↑; b↑ := t
END;
```

相应调用程序片断:

```
NEW (jp); jp↑ = 3;
NEW (kp); kp↑ := 5;
Swap2(jp, kp);
```

本程序调用图示如图6-6:

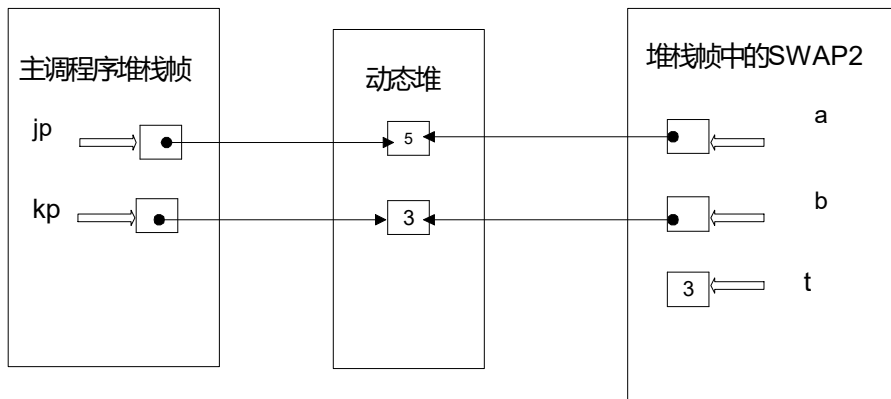


图6-5 指针调用图示

主调程序堆栈帧中有指针 `jp`，`kp`，由于是 `new` 生成的，在动态堆存储中开辟所指存储对象 5, 3。子程序堆栈帧中有指针 `a`，`b` 和局部量 `t`。调用时，`jp`，`kp` 值复制到 `a`，`b`，`a`，`b` 也指向堆存储中的 3，5。动态堆存储增加了悬挂指针的可能性。

C 语言由于有寻址操作 `&`，一般可不在堆存储中分配所指对象。以下是同一例子的 C 版本。

例6-10 C语言的指针参数传递

```
void swap3(int *a, int*b)
{ int t;
  t = *a; *a = *b; *b = t;
}
```

形参是两指针，实参不用指针的版本：

```
main() {
  int j = 3; k = 5; //声明并初始化两整数
  swap3(&j, &k); //类型匹配吗?
}
```

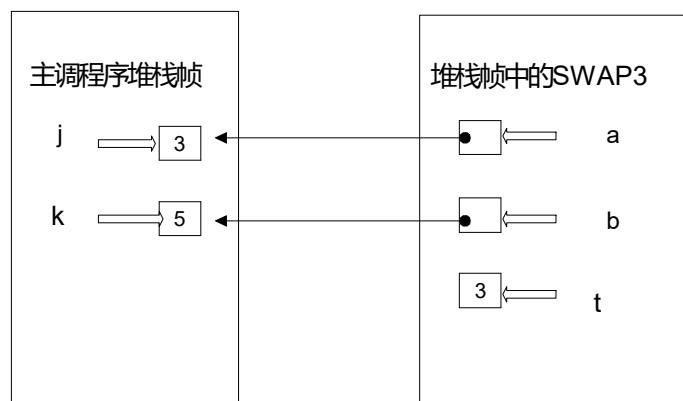


图6-6 C语言的指针——引用调用

主调程序堆栈帧中 `j`，`k` 束定的存储对象中放 3，5 值。子程序堆栈帧 `a`，`b` 束定的存储对象内容放 `j`，`k` 的地址(指向主堆栈帧的存储对象)。另有一局部量 `t`，运行后其中有值 3。

实参是指针的版本：

```
main()
  int j = 3, k = 5;
```

```
int *jp = &j, *kp = &k;
swap3(jp, kp);
```

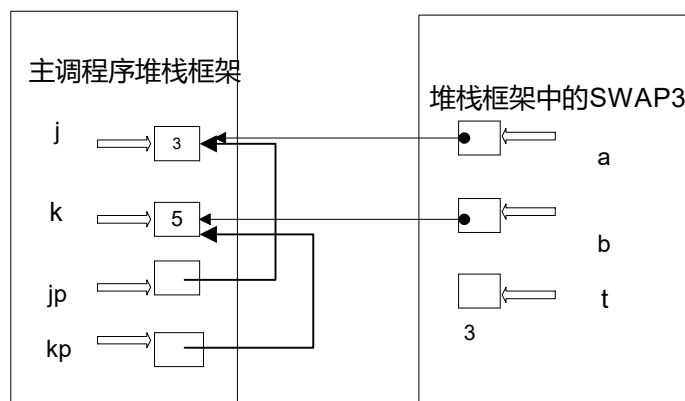


图6-6 C语言的指针调用

此时，主堆栈帧中增加了jp, kp它束定了存储对象，其内容是j, k的地址。执行过程是：swap3模块调入堆栈帧，指针结合后，a, b也指向j, k, 执行后t中有值3, 然后swap3堆栈框架消失，返回后继续执行完main()。读者自然可看出这两个版本的差异。

6.3 变元求值策略

形实参数置换，如果实参是表达式它在什么时候求值？对整个函数调用求值有什么影响？是本节讨论的题目。我们不妨先看一个例题：

例6-11 ML纯函数(无变量和副作用)求值

设有函数定义：

```
fun sqr(n:int) = n*n
```

若 $p = 2$, $q = 5$, 以 $p+q$ 作实参，调用此函数，可以有三种求值方案。

第一种急求值：

$$\text{sqr}(p + q) = \text{sqr}(2 + 5) = 7 * 7 = 49$$

即在形实结合时，实参变元(或表达式)先求值，置换后进入程序体执行。

第二种是正规求值：

$$\text{sqr}(p + q) = (p + q) * (p + q) = (2 + 5) * (2 + 5) = 7 * 7 = 49$$

即形实结合后把实参变元表达式直接代入函数体中每个形参出现的地方。求值一次算一次表达式。

第三种求值方案是懒求值：

$$\text{sqr}(p + q) = (p + q) * (p + q) = (2 + 5) * 7 = 49$$

表达式第一次求值后，将结果值暂存，再出现形参则以结果值置换。

对于本例这三种求值方案的最终结果是一样的。因为它们都是ML的纯函数。满足Church_Rosser性质：

若一表达式可完全求值，仅当它能前后一致地按正规求值次序求值。若一表达式能以几种不同的求值次序求值(包括混合使用各种求值方案)，则所有这些求值次序得到的结果值应该是一样的。

这是衡量一个函数和表达式是否有副作用的一个重要性质。我们把满足Church_Rosser性质的函数称为纯函数。以下我们分析不满足Church_Rosser性质时造成的问题。

(1) 急求值

急(eager)求值对应为前述值调用。实参变元表达式先求值,再复制。这对于需要部分求值情况是很不利的。请看以下函数定义:

```
fun cand (b1: bool, b2: bool) =
  if b1 then b2 else false
```

有函数调用:

```
cand (n>0, t/n>0.5)
```

当 $n = 0$, $t = 0.8$, 本调用失败, 因为第二实参变元表达式 $0.8/0 > 0.5$ 无解。若用正规求值则有解。因为它相当于求:

```
if n>0 then t/n > 0.5 else false == false
```

正因为如此, 急求值也是严格(Strict)求值, 即在函数调用之前所有实参变元表达式都求值完成, 也叫完全求值, 它满足Church-Rosser性质第一句话。

(2) 正规求值与懒求值

之所以叫正规(normal_order)求值是源于 λ 演算的置换规则。即若函数定义为 $F(P)$, 函数应用为 $F(Exp)$, 则在函数(体)中的每次 P 出现均用 Exp 置换。这相当于前述名调用。显而易见, 正规求值要作多次 Exp 计算。如果 Exp 有副作用, 多次计算的 Exp 值不会是同一个值, 整个函数求值就难以预料了。

正规求值支持递归程序, 因为递归程序递归次数往往在编译时是不知道的, 它展开多少次置换多少次似乎更好实现一些。Algol 60是最先实现递归的, 所以, 它采用名调用。但如前述, 由于多数语言存在副作用, 正规求值本身采用较少。

懒(lazy)求值是正规求值的特例, 它只用到到被结合的形参变量的值时才对实参变元表达式求值(第一次出现), 求值后将结果值束定于形参变量, 以后形参变量再次出现就不用求值了。这是懒的第一个意思, 例子见例6-11。对于复杂的表达式如果子表达式求值对整个表达式求值没有影响就不再求它。这是懒的第二个意思。事实上, 很多逻辑表达式都可以部分求值:

```
E = Exp1 and Exp2 and ... Expn
```

若 $Exp1$ 求值为false, 则 E 值已定为false。再如:

```
E = Exp1 or Exp2 or ... Expn
```

若 $Exp1$ 为true, 则 E 值为true不论其它表达式取何值。

这也叫短路(short circuit)求值, 一般用作条件表达式, 也叫短路条件。不仅效率高, 而且能避免不必要的出错。例如, Pascal中,

```
WHILE (scan<101) and (a[scan] <>key) DO
```

当 $scan = 101$ 时应停止本循环。但Pascal做不到, 它采用急求值。当 $a[101]$ 数组 a 已越界, 程序才停止执行。然而, 同样语句在C中执行得很好。C采用懒求值, 当第一子表达式求值为false时, 整个表达式不再求了, 跳出WHILE。

多数命令式语言采用急求值。C, APL, 纯函数式语言(因无副作用)采用懒求值。

6.4 高阶函数

以函数或过程作为实参变元或返回值的函数或过程, 我们统称高阶函数。以 λ 演算为模型的纯函数式语言都有高阶函数机制。命令式语言对函数作变元早就有此需求, 例如, 一个求定积分的程序, 只要输入某个具体函数和积分上下界, 它都可以求出面积。再如, 在统计事务应用中, 一张报表, 可以按字母字典序, 数字的升、降序排序。这样, 在排序程序中比较运算符($>$, $<$, $=$)既要比较字符又要比较数字, 就只好编成参数化的。如前所述, 运算符就是函数, 则排序程序就以抽象的运算符(函数变元)作参数。本节我们讨论函数作为变元和函数返回函数结果值。

6.4.1 函数作为变元

函数作为变元的需求主要是为通用化程序单元而提出的。FORTRAN最早就直接允许在参数表中写虚、实函数名。只要在主调程序写上该实函数是外部的(EXTERNAL FA, FB), 连编程序就可以找出FA, FB, 并在每个虚函数名F处以FA或FB置换。这样, 科学计算程序的通用性大为提高。是程序库, 程序包程序主要的形式。

第二种用途是作映射函数(mapping function)。它把单目、双目运算扩充到多个数据对象的数组或表上。映射函数本身以简单运算函数和表(或数组)作实参变元。请看下例:

例6-12 LISP 的mapcar函数

设程序上文已有四个表x:(4 9 16 25), y:(1 2 3 4), z: NIL, w:((3 4 5) (2 1) (7 9 3 6))。mapcar要求的实参函数用' 标记: 则有:

| 表达式 | 解释 | 返回表 |
|---|--------------------------|---------------|
| (mapcar' +1 x) | 把加1函数用于x诸元素 | (5 10 17 26) |
| (mapcar' +x y) | 加对应诸元素 | (5 11 19 29) |
| (mapcar' +1 z) | 把加1函数用于空表 | NIL(不知应加几次) |
| (mapcar' (lambda (a b) (-(* 3 a) b)) x y) | 把函数((3*a)-b)应用到x y对应的元素上 | (11 25 45 71) |
| (mapcar' caar w) | 消去每子表的头项两次并销毁 | (5(3 6)) |

将一个函数作为参数传递给另一函数是十分容易实现的, 只要传一个指向函数的指针。仅有的问题是对函数的类型作检查, 也就是实参函数的变元类型, 返回值类型. 这在编译处理调用本函数的函数时都应该知道。这就是Pascal和ANSI C以函数作变元与函数式语言主要的不同之处。因为函数式语言往往可以从函数定义来推断类型而又是解释执行的。以下是典型的以函数作输入参数的例子(C)。

例6-13 凯人勒函数求根:

```
#define M 2.2
#define E 0.5
double root(f, a, b, eps) {
    double (*f) ( ); //f是参数化函数
    double a, b, eps;
    double m;
    m=(a+b)/2.0
    if (f(m) == 0.0 || b-a<eps)
        return(m);
    else if ((*f)(a) * (*f)(m)<0.0)
        return(root(f, a, m, eps));
    else
        return(root(f, m, b, eps));
}

Double kepler(x) { //实参函数
    double sin( );
    return (M-x+E*sin(x));
}

main( ) {
    double kelper( ), root( ), x;
```

```

x=root(kelper, 0.0, 7.0, 1e-12);
print("\n%17s%25.16f\n%17s%25.16f\n\n", "approximate root:", x,
"function value:", kepler(x));
}

```

6.4.2 函数作为返回值

C语言以指向函数的指针形式支持函数作为变元，C++进一步对预定义的运算符重载并将它们用于函数变元。其函数返回值可以是指向函数的指针。但C和C++均不能像函数式语言那样，在函数中创建一个函数并把它作为返回值返回。这是两种函数作为返回值的区别。

函数式语言的这种功能在复合函数、闭包、和组成无限表上是十分有用的。在本书函数式程序设计范型中我们还要介绍。这里先以闭包说明高阶函数的应用。

闭包(closure)是可用到表达式上的操作。我们可以把每个表达式上的自由变量束定到另一表达式的值上以封闭本表达式。闭包可创建一个函数作为闭包处理的返回值。闭包最有用和最容易理解和应用是部分参数化。例如，有 n 个变元的函数，我们将其中一个变元束定于局部定义的值上就得到一个 $n-1$ 个变元的新函数。

例6-14 ML的函数部分参数化

设 $n > 0$ ，求 b^n ，则有ML程序：

```

fun power (n, b) = if n = 0 then 1.0
                  else b* power (n-1, b)

```

这是两个变元的普通函数。变元类型 $\text{Integer} \times \text{Real}$ 。ML可以改写为。

```

fun powerc(n) (b) = if n = 0 then 1.0
                   else b* powerc(n-1) (b)

```

函数抽象`powerc`作用于第一个变元，返回的仍然是函数，再作用于第二变元，得 b^n 结果值。我们可以把隐式的返回函数显式写出：

```

val sqr = powerc 2
  and cube = powerc 3

```

当`powerc`中的 n 束定于2时得平方函数`sqr`，束定于3时是立方函数`cube`。那么函数引用只有一个变元 $\text{sqr}(b) \equiv b^2$ ， $\text{cube}(b) \equiv b^3$ 。

例6-15 闭包应用

若国际银行已有各国通货转换程序`Convert`，当我们得知1马克 = 0.6美元时，如果`Convert`是用ML之类的有高阶函数语言编的程序，则立即可派生出两个新函数，即闭包：

```

val MarksToDollars = Convert 1.67
  and DollarsToMarks = Convert 0.6

```

这两个新函数是`Convert`的第一参数束定于变元值的结果，以后若有`DollarsToMarks` 72,000，72,000就是第2变元值了。

尽管闭包产生新函数，但它增加不了多少代码。闭包函数代码和原有代码极为相似。实现的办法是利用包含束定符号的记录和指向函数的指针。对于上述把`Convert`的参数束定于0.6或1.67这种常量，填入记录，加上指向`Convert`的指针问题就解决了。

然而，当我们束定`Convert`的第一参数如果不是常量问题就出来了。例如， $x * 0.6$ 。这当然会因 x 值不同影响到返回函数的意义，就不是美元到马克了。如果设计者说 x 是某个修正数不会有大的出入，那么是先求值再束定还是把 $x * 0.6$ 作为一个串和第一参数束定呢？如果懒求值当然是后者。到使用时求值一次！为此在求闭包函数的环境中应有对 x 的束定，且它的生存期不短于闭包块。

懒求值在动态中实现，对于静态强类型语言是不能实现真正的闭包函数的。但不等于说不能实施闭包的功能。请看下例。

例6-16 用Pascal模拟闭包函数

设一数组ar: 7 4 2 6 3 1 8 5 9以某个运算符对其归约, 例如, OP = '+' (累加), 则sum = 45。若OP = '-' (累减), 则sum = -31, 若OP = '*' (累乘), 则sum = 362880。OP是变量随输入而定。则归约程序是:

```
TYPE row_type = ARRAY[0..8] OF Integer;
FUNCTION Reduce (FUNCTION f(x:Integer;y:Integer):Integer;
                  ar:row_type):Integer;
VAR sum,k:Integer;
BEGIN
    sum := ar[0];
    FOR k = 1 TO 8 DO
        sum := f(sum, ar[k]);
    Reduce := sum
END;
```

再给出算符定义:

```
FUNCTION AddOp(x:Integer,y:Integer):Integer;
BEGIN AddOp := abs(x)+abs(y) END;
```

同样可以给出SubOp, MulOp..., 于是可写出归约的闭包函数:

```
FUNCTION MySum (ar: row_type):Integer;
BEGIN MySum: Reduce (AddOp, ar) END;
```

请注意MySum只有一个参数。另一函数参数已和AddOp值束定为累加。但这全部是静态完成的, 故只能模拟闭包。

6.5 小结

- 函数和过程抽象是将部分程序封装, 并设输入/出接口, 过程抽象的输入出接口仅有参数表。函数还有另一返回值的出口。
- 函数(过程)型构和函数(过程)体构成函数(过程)定义。函数(过程)调用是同样名字并带同样类型、个数的实参变元表。函数调用返回值, 过程调用只改变参数和环境的值。
- 无参函数(过程)是极易产生副作用的函数(过程)。但有时非常有用。
- 理想情况是程序中的所有值均可以作实参变元。
- 值调用以复制作形实结合是最可靠的参数传递机制。它时空效率低。
- 引用调用是形参名束定于实参存储对象上的形实结合, 时空效率高可返回值。
- 参数模式in, out, inout是在已有调用机制上更加安全的限制。
- 返回调用是实现out模式的机制, 实质也是复制。
- 值返回调用是双重复制, 实现inout模式。在多机网络并发环境下更安全。
- 名调用是形实参名字(在程序对象上)的置换。因多次求值不宜出现在有副作用场合。
- 指针调用机制是复制指针值。容易引起悬挂指针。C语言一般变量、引用变量、指针变量均可作参数比较灵活。
- 形实结合时急求值是在调用界面上求值。它安全可靠, 但不能实现部分求值, 增加了不必要的出错机会。
- 正规求值是用到形参则对实参求值。多次用多次求值。效率低当有副作用时不安全。
- 懒求值是第一次用到形参值才求值, 以后则以该值为准。可以实现短路条件。效率高, 安全。
- 有副作用的情况下是不能满足Church_Rosser性质的, 该性质指出表达式求值: 可前后一致地按正规求值才是完全求值; 按所有求值方式求出的值应是一致的。
- 函数的参数或返回值是函数则该函数是高阶函数。高阶函数扩大程序概括能力提高通用性。
- 函数闭包同一组代码可派生出一系列有用的函数, 方便于应用。

习题：

6.1 参数、变元、变量有何不同？

6.2 试述以下函数调用的优缺点：

(a) 利用缺省参数形实参数个数可不匹配(Ada, C++)。

优点：调用灵活、简洁，使用方便；效率高；

缺点：易带来副作用，引起参数丢失、误用。

(b) Ada指明参数调用次序可随意。

优点：灵活、安全、不需要记住参数名。

缺点：要求程序员记住所有的参数名。

(c) C语言某些函数参数个数随意。

优点：灵活、可通过参数个数，类型的多样化实现过程函数，高阶函数等。

缺点：不安全、易引起函数功能误用。

(d) ANSI C的参数原型。

优点：安全、有利于进行强类型检查，易于区分参数与局部变量。

缺点：不够灵活方便，在函数调用时参数个数、类型、位置对应关系限制较严。

(e) Ada有参数模式C++为什么没有。

开发并发进程、多进程时易只读，只写。Ada的参数模式中的Out模式保证了在多进程、并发程序中的安全性，所以多用于开发多进程，而C++一般不用于开发多进程。

6.3 “过程调用只要注意检查参数表，程序就不容易出错了”。这话对吗？

6.4 Pascal的常量参数为什么不能是VAR参数？

6.5 C参数传递依赖值调用实现的，它返回改变了的参数值靠什么？

6.6 详细解释多任务应用中引用调用为什么不及值-返回调用？

6.7 有以下Pascal程序

```
PROGRAM ParameterDemo;
VAR i:Integer;
    a:ARRAY [1..3] OF Integer;
PROCEDURE Strange (VAR x:Integer;y:Integer);
    VAR temp:Integer;
    BEGIN temp := x; x := x+y; y = temp END;
BEGIN
    FOR I := 1 TO 3 DO a[I] := 10-i;
    I := 2;
    Writeln 'Initial values', a[1], a[2], a[3]);
    Strange (a[i],i);
    Writeln ('Final values', i, a[1], a[2], a[3])
END.
```

(a) 画出堆栈帧图，标明各数据间关系。

(b) 跟踪执行说明存储对象中的值如何改变的。写出输出结果。

(c) 如Strange中x, y都是VAR参数会如何？

6.8 指针调用与引用调用有什么不同。C语言指针调用为什么相对安全？

6.9 何谓懒求值？应用程序员能控制吗？

6.10 试说明引用调用和指针调用在以下情况的同异：

(a) 写过程调用时。

(b) 写过程定义的类型构造时。

(c) 写过程定义的过程体时。

6.11 按以下表格回答12个问题

| 变量类型 | 创建时间 | 死亡时间 | 何处访 |
|------|------|------|-----|
|------|------|------|-----|

| | 问 | | |
|--------|----|----|----|
| 全局变量 | 1 | 2 | 3 |
| 静态局部变量 | 4 | 5 | 6 |
| 一般局部变量 | 7 | 8 | 9 |
| 动态堆变量 | 10 | 11 | 12 |

3, 6, 9, 12问可回答：“声明所在块”；“除恢复之外的任何地方”；“其它(详细说明)”。

其余可回答：“装入时”；“块入口时”；“块出口时”；“程序终止时”；“任何时候”；“其它(详细说明)”。

答：1：装入时 2：程序终止时 3：整个程序 4：装入时 5：程序结束 6：声明所在块
7：声明处 8：块出口 9：声明所在块 10：声明处 11：块出口 12：声明所在块

6.12 以下Pascal程序打印什么？画出堆栈框架图说明变量内值的变化情况。

```
PROGRAM trick (INPUT, OUTPUT);
VAR j, k, l: Integer;
FUNCTION f(k: Integer; VAR l: Integer): Char;
VAR k: Integer;
BEGIN
  K := l+j; l := k; j := k;
  IF k>10 THEN f = 'Y' ELSE f := 'N'
END;
BEGIN
  J := 3; k := 15; l := 4;
  Writeln (f(l, j));
  Writeln (j, k, l)
END.
```

6.13 何谓高阶函数，它有什么用？编制一个仿真高阶函数的应用。

答：以函数或过程作为实参变元或返回值的函数或过程，我们统称为高阶函数。

作用：①函数作为变元的需求主要是为通用化程序单元而提出的。

②作映射函数。把单目、双目运算扩充到多个数据对象的数组或表上。

见书“凯人勒函数求根”

6.14 一个条件表达式if E1 then E2 else E3可改写为函数if_then_else(E1, E2, E3)形式。如语义和原表达式一样，三种参数求值方案均可吗？

6.15 有函数定义fun F(x: T) = E，函数调用为F(Z)。按正规求值，刻划置换：

$$F(Z) \equiv \text{subst}(E, x, Z)$$

其中subst为置换函数，E中x的自由出现均以Z置换。按此形式写出急求值的置换函数表达式。

6.16 写出一个C程序，函数调用后结果值为函数。

6.17 试评价例6-9中指针版和引用版程序的优缺点。

第7章 程序控制

冯·诺依曼机器模型变量的时空特性对程序中求值的次序是十分敏感的。程序员用这类语言要得到预想的计算，就要善于驾驭求值次序。表达式的求值次序是最低层的程序控制，在前面章节中我们已经介绍过了。在它的上层是四类控制：顺序控制、选择控制、重复(迭代)、函数或过程调用。本章要讨论它们。

再上二层是对程序模块的控制。包括一个程序的各模块组织以及它们与环境(软、硬件平台)的相互关系。本章部分讨论它们。

并发控制也是一类控制，它可以在语句级，特征块和模块级实施并发控制。涉及并发结构模型和通信机制后文第13章还要讨论。本章只讨论顺序程序控制。

结构化程序最重要的成果是程序控制。它使程序成为可分析的、分层结构的，程序正文和程序的执行逻辑比较一致。同时也使编译器易于实现，尽管它已定论，在前述章节中也零散讨论某些部分。本章对此作一系统的小结。

7.1 一般概述

程序控制是控制机器执行计算的次序。我们把表达式求值规则看作是微观的。则最基本的控制对象是“语句”(Statement)所表征的计算。在英语中statement一词有‘陈述’的意思，易于和声明混淆，所以，有的书上主张叫“命令”(command)。对于命令式(Imperative)语言叫命令也许更好，本书这两个词通用。

早期的高级语言脱胎于汇编语言，语句和汇编语言的一条指令对应。它相当于程序世界的一条“指令”。是可以独立处理的最小单元(当编译或解释时)。程序员通过语言代码陈述的先后表达是程序最基本的顺序控制。但是一个复杂的计算单有顺序指令是编不出来的。汇编语言的JUMP指令在程序世界就成了GOTO命令。它打破单一的顺序，使程序有声有色。可以毫不夸张地说：命令式语言只要有赋值语句 $V := EXP$ ，简单的逻辑条件IF(e)和GOTO语句就可以编出一切计算程序(输入/出除外)。按IF条件跳过一段代码是很容易的，GOTO返回到某起始语句即可实现重复。早期的语言都脱离不了GOTO。

GOTO和语句标号把顺序的程序代码切分得七零八落。编译时只能把它们分成很多可连续执行的程序小块(显然，GOTO语句的前后语句不能在一块)的语句组。这就是模块(Module)一词的来历。对于重复执行的块单给出关键字和截止处语句标号，并引出循环域和嵌套循环的概念。这是模块封闭性的开始：GOTO只能从循环域内转向域外，反之不行。但当时重点还在想出方便的GOTO表示上。单GOTO语句FORTRAN就有五种之多。流程图的发明使得编程思路清晰，而程序代码仅仅是实现流程图的表示工具，它本身的可读性未引起足够重视。随着程序尺寸的增长，脱离了流程图，程序正文越来越难读(见第2章图2-3示例)。即使不太大的程序，几十个GOTO来回穿梭就成了戏称的“乱面条”程序。难读、难调、难修改。但当时人们醉心于精巧的设计。终于酿成60年代初的软件危机。

自从1965年E. Dijkstra提出“GOTO语句是有害的”以来，60年代中叶在西方软件界引起一场争论，因为它动摇了赖以构成巧妙计算转移的根基。既然它有害，那么，首先要回答，不用GOTO行不行？1966年Boehm和Jacopini回答了这个问题：任何流程图的计算逻辑都可以用顺

序组、条件选择组、迭代组三种程序结构实现。这三种结构严格一个进口一个出口。也正因为如此可以随意嵌套(将某一对进出口替换成另一结构)，构成极为复杂的程序。用这三种构件块构造程序可完全不用GOTO语句。这就是结构化程序设计的基本思想。程序控制在块一级，块相对封闭，即不许有控制从块的一部分处转移到另一块内。这样程序控制就成了组织顺序、选择、迭代的结构了。

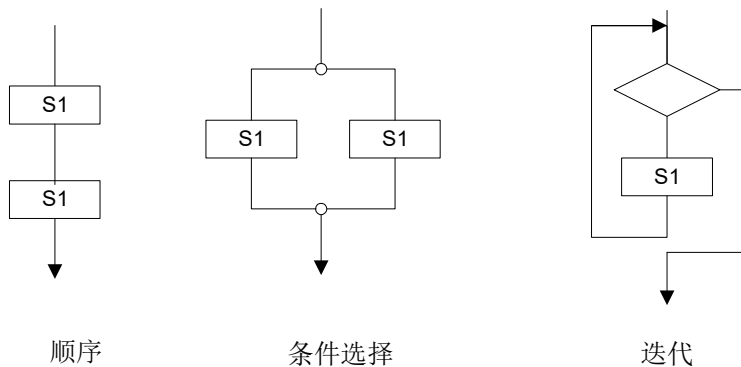


图7-1 三种最基本的程序结构

理论探索完成后要回答的第二个问题是用结构化程序代替非结构化程序会有哪些问题？是否一切都好？事实上，结构化程序结构确有不方便之处，至少是低效。例如，在一计数循环中查找一个数，头几次就找到了，余下空循环不做到底是不能出来的。保留GOTO一跳出来不是更好吗？此外，若想利用某块中的部分代码不能从需用处进入非得从头上进入，这势必要保证不用部分空执行(没有GOTO，就要增加很多不必要的代码)。这就是是否取消GOTO值得争论的原因。最后大家接受了Knuth(1974)对GOTO使用的折衷意见：GOTO是有害的，但有时对改善效率带来好处。可以保留GOTO，但要加以限制使用。只能向前转移，不能向后。GOTO的迹线不能交叉，只能从块内转到块外，不得相反。问题基本解决，争论的双方依然坚持各自信念，所以70年代以后发展起来的语言有保留GOTO的，如Pascal, Ada, C，但不提倡使用。有完全取消GOTO的，如Modula, Adison等等。有的保留GOTO的积极作用限制GOTO的副效应，把它们改头换面变为比较安全的顺序控制器(sequencer)，详见后文。

结构化程序除了顺序、条件选择、迭代三种最基本控制而外，子程序调用也是一种重要的控制。虽然在子程序体中依然是三种基本控制结构的复合，子程序调用本身是显式地控制抽象。多次子程序调用和返回，构成程序的深层嵌套。对于非命令式语言，如纯函数式语言，顺序控制并不十分重要。它控制计算更多的依靠嵌套。

7.2 顺序控制

顺序控制在程序结构上是一组可顺序执行的简单语句，如果每一个顺序成分可用一顺序结构置换，顺序结构可形式地表示为：

S1; S2

进一步扩展可为：

S1; S2; ... Sn

其中Si为简单语句(命令)。一般过程语言有哪些简单语句呢? 我们用最丰富的Ada简单语句作解释。

```
简单_语句 ::= 空_语句
              | 赋值_语句   | 过程调用_语句
              | goto_语句   | 入口调用_语句
              | 出口_语句   | 返回_语句
              | 引发_语句   | 夭折_语句
              | 延迟_语句   | 代码_语句
```

其中空_语句, 赋值_语句, 代码_语句不影响控制和转移, 代码语句是某些信息要用机器(或汇编)码。延迟语句为控制同步, 指定在原地停留的时间, 但无控制转移。

连续的赋值是最基本的运算。goto语句已如前述, 可以往前、往后跳过任意行代码。当然它和语句标号要配合。标号的表示法各语言不一, 有用数字, 有用字符串的。Ada用<<字符串>>。

exit(出口)语句, raise(引发)语句, abort(夭折)语句, 都是顺序控制符。

exit将程序控制跳到所在块的末端, 或由它指明的外嵌套块的末端。C语言中与其相当的是break。

raise是引发异常。当程序执行到此句(一般是程序员预感要出异常设在此处)时, 跳到有exception关键字处执行异常处理段(一般在本程序块末尾), 且不返回。异常引发规则比较复杂, 详见本章7.5节。

abort是强行夭折即就地停止, 有的语言是HALT命令。这类就地停止和暂停命令在FORTRAN早已有过。FORTRAN的STOP和END分工是一为执行停止, 一为代码结束。由于它把程序正文表示和执行逻辑分开。任设STOP为多出口。与结构化宗旨有悖, 在以后的版本中取消了。FORTRAN的PAUSE为暂停语句, 它中止程序执行, 以便程序员中途作些调整, 调试程序, 而且必须重按回车才能恢复运行。这些早期单机单用户, 惜机时如金时代的便于调试的语句, 以后都取消了。完全可以设“断点”, 追踪排错解决。恢复abort和HALT是新一轮并发程序和增加了异常处理后的要求。强行终止程序执行一般是信息不足, 事先考虑的异常处理段不足以处理已发生的异常。而进一步运行又无意义, 只好终止程序的执行。一般终止前应将当前信息尽可能收集、显示出来, 以便调试员分析。

入口调用和子程序调用相似, 前者是专用于并发程序的任务调用。也要作参数匹配。每当执行到此句, 一个任务被激活, 任务体开始执行。它对程序控制转移到另一程序单元作用是一样的。

同样, return是显式指明执行到此返回至调用点。因为返回是同一点, 多个return一般都是允许的(违反一出口的规定吗?)。早期子程序过程必有return, 而现代语言则不一定(为什么)?

程序总是按程序正文自上至下, 自左至右地(隐含)顺序执行的。表达式求值规则和顺序控制器是对这个隐含执行的“修正”, 从而达到控制计算的目的。

7.3 条件选择控制

计算机“聪明”最基本的基础是会根据条件选择它应执行的代码。所有的程序设计语言都离不开条件控制。

7.3.1 结构式条件控制

简单的条件语句IF(e)，根据逻辑表达式的真假值在程序中产生分枝。它本来就是想做if... then... else计算逻辑的。不过早期语言用的IF(e)是语句级的，这种程序语言编的程序不仅不宜于阅读，且T-部分和F-部分得不到保护，甚至它有多大都没有明确界线，其它地方的goto可随意进入，程序极易出错，Algol-60正式把它看作复合语句，那么，这两部分代码叫T-子句和F-子句。子句以语句结束符终止没有其它标记。

if语句的退化型是if_then，正规型是if_then_else，T-子句和F-子句可以是任何语句组。嵌套if一开始就发现了问题，在Algol60草案讨论中，就发现了‘悬挂else’问题。例如，当有if嵌套时，有：

```
if E1 then if E2 then S1 else S2
```

它可以解释为：

(E1=true) \wedge (E2=false) 执行S2，else属于内部if，也可以解释为：

(E1=false) 执行S2，else属于外部if

即不管E1为真假S2均可能执行。原因是可以画成多个语法树，它有二义性。

解决这个问题是很容易的，加上语句括号DO-END(PL/1)，BEGIN-END(Pascal)就可以：

```
if E1 then begin if E2 then S1 else S2 end
```

```
if E1 then begin if E2 then S1 end else S2
```

它的简化形式(FORTRAN-77，Ada)是：

```
if E1 then if E2 then S1 endif else S2 endif
```

```
if E1 then if E2 then S1 else S2 endif endif
```

└────────就近匹配────────┘

语句括号既使语义清晰又可以封闭块，所以，Pascal，C的程序员要指明块时(两个语句以上)，都要用语句括号begin_end或{ }。

既然不要goto，早期的流程图也可以结构化。与此对应的选择结构有nassi_Schneiderman式流程图。每个语句和复合语句都是块连接，如图7-2所示。块中分成T-，F-子句分别画成T-域和F-域。可随意嵌套(将-S块又扩充为选择结构)。

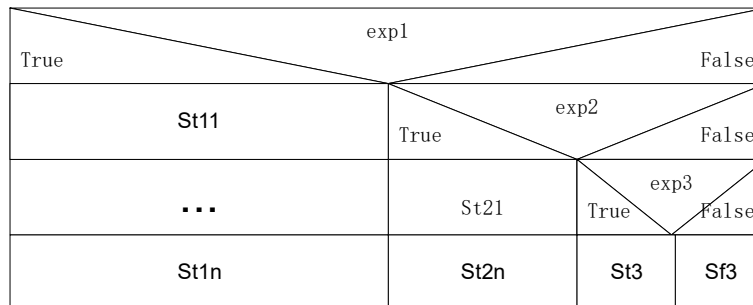


图7-2结构选择的Nassi_Schneiderman

7.3.2 case和switch

在图7-2所示结构选择中是连续嵌套的选择结构对应的程序结构是：

| | |
|------------------|--------------|
| IF exp1 THEN | IF exp1 THEN |
| ST1 | ST1 |
| ELSEIF exp2 THEN | ELSE |
| ST2 | IF exp2 THEN |
| ELSEIF exp... | ST2 |
| ... | ELSE |
| | IF exp ... |
| ELSE | ELSE |
| SF3 | SF3 |
| ENDIF | ENDIF |

左右两表示法语义是一样的，但左边出了新关键字ELSEIF。这是在FORTRAN-IV到FORTRAN-77结构化改版时增加的，增加后显得自然方便，只是在“都不”满足条件时执行最后一个ELSE。不用费心记对应是那一层。ELSEIF在Ada中更成为可区分的elsif，和else if是两种表示。

嵌套 IF 结构的执行，永远是执行一组代码就跳到endif。对于不同条件，if_then_elseif...else结构方便，清晰。然而，对于同一条件表达式而多次判断其值的嵌套IF结构，这种表示显得累赘，且多次错缩进编排，程序反倒歪斜。于是多数结构化语言提供case选择。

两种语言的分情况语句的表示示例如下：

| | |
|--------------------|--------------|
| case Exp is | switch (exp) |
| when v1=> S1; | case v1: S1; |
| when v2=> S2; | break; |
| ... | case v2: S2; |
| when vm vn=>Sn; | break; |
| when others => St; | default: St; |
| end case; | |

Ada 的case语句

C的switch语句

这两种表示法略有差异，大体一致。Ada的Exp不限于真值表达式(只两种情况)，可以是任何离散类型表达式(C限于整数)。根据取值Vi的情况跳到对应的case标号那个子句Si执行。Ada是执行完子句跳到end case，而C若无break则不跳，接连执行下句.C语言设计者认为会对组合分枝带来方便，但这种过分灵活性，实际上是设计缺陷。others和default选择都是在取值超出了case标号的值域时执行。Pascal类似Ada，但无others选择。

7.3.3 以条件表达式实现选择控制

函数式语言没有语句，只有表达式，选择控制只能由条件表达式完成。C语言是面向表达式语言，它有条件表达式，且编译处理最小单位也不限于语句。例如，C语言语句行中可以出现：

```
++x; x++;
```

这种无赋值号的表达式。它还有 ? : 双目运算符表示的表达式，例7-1示出条件表达式和条件语句差别。

例7-1 将b, c中小值存入a (C语言)

```
a=b<c ? b:c      if (b<c) a=b; else a=c;
条件表达式      条件语句
```

一个语言并没有必要既提供条件表达式又提供条件语句。事实上, 任何条件表达式描述的计算条件语句都可以做到, 只是更臃杂一些。但是条件表达式返回的是一个值, 它可以做函数的实参变元, 在封闭的程序中继续使用。而条件语句不能作为实参变元, 它与程序的其它部分通信只能通过环境。事实上, 赋值就在改变环境(的状态)。为此, 函数式语言, 例如LISP, 就用了许多括号封闭各层嵌套调用(ML用文字括号如let... in等)。程序设计风格也是嵌套, 没有或不强调顺序子句。可以这么说, 命令式语言大量依赖顺序命令, 纯函数式语言完全依赖写嵌套函数调用。可惜LISP受到早期过程语言高效的影响也包含了rplaca(破坏性赋值)函数和prog控制结构。程序员可以使用变量并列出顺序执行的表达式。这样一来就没有纯嵌套和纯顺序的语言了。命令式语言主要是顺序赋值, 但也有嵌套和递归函数调用, 参数结合, 甚至像C语言那样有条件表达式。而LISP主要是嵌套, 但也有赋值, 和顺序执行。ML, Miranda, FP力图回复纯函数式, 以摆脱变量时空特性的影响。

7.4 迭代控制

迭代一般用于处理同构的、聚集数据结构的重复操作, 诸如数组、表、文件。对于命令式语言迭代一般是显式的循环, 例如上节讨论的结构化语言的while_do。对于函数式语言迭代一般是隐式的, 如LISP的map函数, 它通常返回一个不断延长的表。

结构化语言的设计者虽然接受任何迭代都可以转为while_do表示的结构化程序理论, 但为了便于表达, 迭代的表示法多种多样, 赛过50年代FORTRAN的GOTO表示法, 即使是同一种表示法其表达的灵活性也是空前的。C语言有以下例子:

例7-2 用for循环计算表中元素之和

设表list是一简单结构, 一个成分的值value, 另一个成分是指向下一表元素的指针next。可以有以下六种写法:

[1] 先赋sum初值进入正规for 循环:

```
sum=0;
for ( current=list; current!=NULL; current=current->next)
sum += current ->value;
//循环控制变量current就用表list
```

[2] sum在循环内赋初值:

```
for (current=list, sum=0; current != NULL; current =current -> next)
sum += current->value
```

[3] 循环控制变量可以在for以外赋初值, 增量在体内赋值:

```
current = list, sum=0;
for ( ; current != NULL; )
sum += current->value, current = current->next;
//用', '号连接的两语句成为一个命令表达式
```

[4] 循环条件为空, 用条件表达式完成:

```
current = list, sum = 0;
for(; ; )
if (current == NULL) break;
else sum += current->value, current->next;
//不提倡此种风格
```

[5] 完全不用循环体，全部在循环条件中完成

```
for (current = List, sum = 0; (current = current->next) != NULL;
    sum += current->value);
//效率最高. 但list不能为NULL表
```

[6] 逻辑混乱，也能正确计算：

```
current = list, sum = 0;
for (; current = current->next; sum += current->value)
    if (current == NULL) break;
//最坏的风格，list也不能为NULL表
```

这个例子说明语言的语法规则的灵活性，C语言的哲学是只要不给实现带来过多开销，尽量灵活，不管程序员如何写，说得过去就行，例如，[4]中的if的T-子句是语句，F-子句是表达式，整个既是命令表达式也是条件命令。这种过多的灵活性也带来排错调试的困难。

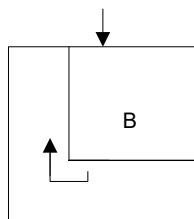
7.4.1 显式迭代控制

控制迭代可以在一段循环代码开始处设条件，也可以在执行一次后设条件，决定是否重复。也可以规定次数，虽然都可以用最基本的while_do仿真。但各有特点，语言选用也不完全一致。

(1) 无限循环

在顺序式命令语言中无限循环被认为是死循环，早期语言均没有，对于并发程序，为了随时接受另一进程发来的信息，它要处于活动的运行状态，就必须用无限循环。以下是Ada的无限循环及其对应的结构图：

```
loop [name]:
    B
end loop;
```



名字name是可选的，有的语言用关键字BEGIN_REPEAT(FORTH83)表示，C语言用空记数f(;;)B；当然，进去出不来无限运行下去是不可接受的。这就要在循环体内设exit命令跳出（或调用另一子程序或任务，在另一程序块中终止）。

(2) 有限循环

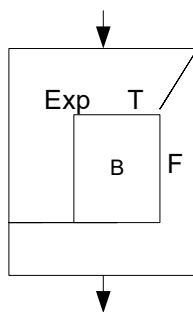
满足某个条件不再迭代，其迭代次数是有限的。

- 如果先测试条件再进入循环体叫‘当循环’(while_do)，形式如下(Ada)，在无限循环体之上加while子句：

```

while Exp
  loop
    B:
  end loop

```



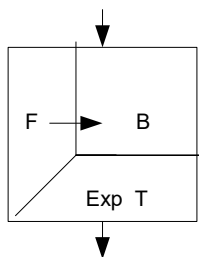
条件表达式为真执行循环体。

• 如果先执行循环体再测试条件叫‘直到循环’(do_until)，形式如下(Pascal)；在无限循环体之下加until子句：

```

REPEAT
  B
UNTIL Exp;

```



条件表达式为假再次执行循环体。这种循环的缺点是条件不完全或条件已满足至少要执行一次。所以是条件比较清楚时用它。

两种循环中条件表达式的变量，或进入循环被加工数据的初始化可以在循环体或子句内完成，也可以在循环外完成。一般说来，这两种循环的次数难以预知，以满足条件为准。

这两种循环实质是一种，故有些语言只有while_do，没有do_until，每当需用do_until时，可用以下等价形式：

```

REPEAT          B;
  B;             =>  while TEXP loop
UNTIL Exp        B;
                  end loop;

```

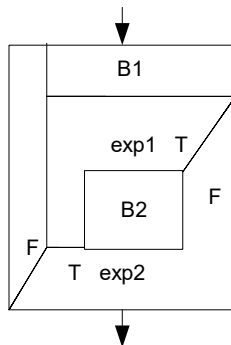
• 通用有限循环 (general loop)

如果循环控制条件可以设在循环的任意位置(即通用)，它相当于while_do和do_until的功能复合。结构化理论叫它do_while_do，其形式是(FORTH)：

```

BEGIN
  B1;
  WHILE Exp1
  B2;
  REPEAT Exp2

```



这种结构最方便，若B1, Exp2为空，即while_do。若Exp1为空，即do_until。多数结构化语言没有专门的通用循环，但提供的其它机制可以实现，如下例：

例7-3 将屏幕上输入的数求和以便打印(Ada)，约定当输入负数时停止。

```
function READ_ADD return Integer is
  N:Integer;
  SUM: Integer:=0;
  loop
    WRITE_PROMPT ("Please enter a number and type a CR");
    N:=GET_NUMBER;
    exit when N <=0;
    SUM:=SUM+N;
  end loop;
  return SUM;
end READ_ADD;
```

(3) 计数循环

如果已知数据结构的长度，将while子句的条件表达式量化，即给循环控制变量以初值、终值、增量，当控制变量中的值达到终值即出循环。这种循环叫计数循环(counted loop)。它是效率最高的循环，也是程序设计语言最早采用的循环(FORTRAN)：

```
DO L I=EXP1, EXP2, EXP3
```

其中I是事先声明的控制变量，整数类型，三个表达式是在此刻能求出值的整表达式。用EXP1赋初值后进入循环，执行完循环体检查是否超出EXP2。若未超出，则增量EXP3后执行，否则出循环。以后陆续研制的语言与此大同小异，主要是在循环控制变量和表达式约定上不同。几十年来一直在演变。

循环控制变量，最初限于非负整数，相应表达式也是整数类型。FORTRAN曾经有过许可实数表达式，但未推广。Ada容许所有表达式均为离散类型(整、真值、字符、枚举)。

控制变量声明，早期是全局静态变量，如从循环中跳出，控制变量值仍可引用，但如果正常出口该值总要大于或等于终值。而终值本身在不同语言中也是可变的(PL/1, COBOL可在循环体内改变表达式EXP2的值，FORTRAN不可)。这样就难于控制了。所以，现代语言即使是静态变量出了循环也不能引用，Pascal的循环控制变量规定了局部于子程序，并明文规定，循环内不得改变终值和增量表达式的值。虽然出了循环该值仍存在(局部变量)，但使用无效。Ada更进一步，循环控制变量仅限于该循环，事先不用声明，给变量定义的取值范围自动作了声明，也不能在循环体内改变取值范围大小，且出了循环无效(编译可查出)。

关于增量表达式，在仅限于正整数的语言中，缺省为+1，Algol60和Pascal容许递减，可以是负整数。Ada的增量随类型，它只有正序和逆(reverse)序，如：

例7-4 Ada的for循环

```
for I in reverse 1.. N loop
  --I自动为整型, I=N , N-1 , ... , 1
for TODAY in DAY range MON.. FRI loop
  --TODAY 自动为DAY(枚举)类型并取子域MON..FRI,
  -- 其增量按DAY中定义顺序——枚举
for BACKDAY in reverse DAY 'RANGE loop
  --按枚举类型DAY的逆序,即SUN, SAT, FRI, ... , MON
  --DAY' RANGE是属性表达式, 意即DAY类型的值域
```

还有一个要说明的问题，是表达式EXP2在顶部求值还是在底部求值？EXP1在顶部或外部，EXP3在底部求值这都没有问题。早期FORTRAN的EXP2在底部求值，以后FORTRAN-77改为顶

部，这样一开始条件不满足就不要进入循环。

(4) 迭代元素

1963年美国Galler和Fischer对FOR循环提出迭代元素(Iteration element)概念，以后C语言的发明人D. Ritchie把它用于C。迭代元素使FOR语句的语义通用化，即循环表达式和循环语句能等价地实现，它们的一般形式是：

```
iterate (<控制元素>)<迭代域>
= iterate (<a>; <b>; <c>)<迭代域>
```

执行过程是：

[1]. 进入迭代域之前先执行一任意表达式表<a>，目的是为了初始化，执行后控制转到表达式。

[2]. 是任意表达式但其结果值只以逻辑值解释：0为‘假’，非0为‘真’。若为‘真’执行迭代域后执行表达式<c>。若为‘假’则跳过迭代域出口。

[3]. <c>是任意表达式表，执行完后控制转移到。

控制元素后接迭代域。如果是语句，则此域中是任意简单或复合语句，执行这些语句都可能产生副作用，如输入/出语句、赋值语句。如果是迭代表达式，每次执行迭代都是在计算程序对象。这些程序对象往往是聚集数据结构，它们就是整个迭代表达式的返回值。

迭代元素和一般FOR循环的最大差别是对迭代元素不加任何限制。语句，表达式，连续赋值，空都可以。所以有例7-2同一问题的六种表达。迭代域中做的事也可以在迭代元素中做，效率更高但牺牲了可读性。从另一个意义上讲，为程序员提供了互易的灵活性。

正是由于迭代元素可以是任何表达式和语句。一般说来，它不能像FORTRAN那样把控制变量放在索引寄存器中，得到高效（利用简单寄存器变量精心设计的个别问题也能达到）。

7.4.2 隐式迭代控制

所谓隐式迭代是语言系统自动实施迭代。这对第四代语言，声明式语言(函数式逻辑式)都非常重要，而且是语言发展方向上的重要机制。

隐式迭代一为对聚集数据每个元素自动逐个加工，一为实现某种特定算法，现分述如下：

(1) 对聚集对象的迭代

APL语言最突出的特征是隐式迭代，因为它是专门处理向量语言的。有数据元素和向量的混合计算，有向量和矩阵的混合计算，如果象常规语言每次计算只能在基本数据类型元素上进行，程序非常臃肿，所以一个运算符自动迭代计算变量所代表的对象上的每个元素，变量根据值定义情况既可以代表简单对象，又可以代表向量、矩阵。如前所述，这种语言是无类型动态存储分配的。

例7-5 APL的隐式循环

```
▽ ANSWER ← A FUNC B           //声明定义两参数函数名为FUNC
[1] ANSWER ← (3×A) - B         //函数定义
▽
X← 4 9 16 25                   //X, Y束定为整数向量
Y← 1 2 3 4
Z← (3 4) ρ 1 2 3 4 1 4 9 16 1 8 27 64 //Z束定为3×4矩阵
```

若有以下表达式，见右下栏的解释。运算符‘ρ’为比较。

| 表达式 | 解释 |
|-------------|--|
| X+1 | X的每一元素加1=(5 10 17 26) |
| X+Y | X, Y对应元素相加=(5 11 19 29) |
| X FUNC Y | (3 *X)-Y =(11 25 45 71) |
| (X+1)=(X+Y) | (5 10 17 26), (5 11 19 29)=(1 0 0 0) 只有一个元素一样。 |
| Z=1 | 将1与Z的每个元素比较, 相同为1, 否则0, 得((1 0 0 0) (1 0 0 0) (1 0 0 0)) |
| +/Z=W | 如果W是纯量, 结果是一向量其元素是W 在Z中每行出现的次数, 如W←4时为(1 1 0)。 如果W是和Z一样的矩阵, 其结果是一长 度等于Z的行数的向量, 元素为每行中W, Z 元素匹配的位序。 |

类似的LISP的map函数的例子见例6-12。它处理的数据类似APL的向量, 主要是表。数据库查询语言如dBASE加工的文件和记录也是聚集数据对象。在处理时也要用隐式迭代, 见下例:

例7-6 一种类似dBASE的dBMAN查询的隐循环

设每个雇员一个文件, 当前记录类型域是:

lastname sex hours grosspay with held

有以下dBMAN命令:

[1]. display all for last name = 'Jones' and sex='M'

[2]. copy to lowpay.dat all for grosspay < 10000

[3]. sum grosspay, withheld to grosstot, whelddtot all for hours >0

第一行找出所有男性名为Jones的雇员。第二行是创建一新文件lowpay.dat把当前文件中税前工资小于10000的记录复制上去。第三行求税前工资总合并扣除那些小时工资已登记了的雇员的那些部分。

至于它内部是如何迭代的算法程序员不用过问, 只知道它在多个顺序组织的文件上进行的。

(2) 回溯

有一些计算是在已有数据集上寻找匹配实现的, 程序员只需写出匹配表达式, 执行这些表达式时是在数据集上一一搜索, 如果沿某个搜索路径无一匹配成功, 则自动回溯到上一数据结点, 直到穷举所有数据库(集)。Prolog, SNOBOL, ICON就是这类语言。我们把回溯(Backtracting)也归到隐式迭代一类。

先看Prolog的一个例子:

例7-7 Prolog查询的回溯实现

如果数据库中已列出以下事实:

| | |
|-----------------------|----------------------------|
| f1.likes (Sara, John) | f9. does (Mike, skating) |
| f2.likes (Jana, Mike) | f10. does (Jane, swimming) |
| f3.likes (Mary, Dave) | f11. does (Sara, skating) |
| f4.likes (Beth, Sean) | f12. does (Dave, skating) |
| f5.likes (Mike, Jana) | f13. does (John, skating) |
| f6.likwa (Dave, Mary) | f14. does (Sean, swimming) |
| f7.likes (John, Jana) | f15. does (Mary, skating) |
| f8.likes (Sean, Mary) | f16. does (Beth, swimming) |

说明8名青年相互爱慕和喜好的运动(溜冰或游泳)。今有一查询: 找出相互爱慕且有共同

运动嗜好的一对，并指出是什么运动。Prolog查询目标为：

?-likes (X,Y), likes (Y,X), does (X,Z), does (Y,Z)

执行过程是逐一匹配这四个谓词，我们暂叫g1, g2, g3, g4四个子目标，都能匹配即为解。具体步骤是：

| 匹配 | 结果 |
|--|---------------------------|
| [1]. 沿数据库匹配第一子目标 (f1<->g1) | X←Sara, Y←John |
| [2]. 匹配g2, 沿f1->f7. Y对X不对 | 失败 |
| [3]. 回溯重配g1, (f2<->g1) | X←Jana, Y←Mike |
| [4]. 匹配g2, 沿f1->f5 (f5<->g2) | g2成功 |
| [5]. 匹配g3, 沿f9->f10 (f10<->g3) | X←Jana, Z←swimming |
| [6]. 匹配g4, f9, Y对, X不对, | 失败 |
| [7]. 回溯第[5]步f11->f16无新匹配 | 失败 |
| 回溯第[4]步f6->f8无新匹配 | 失败 |
| 回溯第[3]步重配g1 (f3<->g1) | X←Mary, Y←Dave |
| [8]. 重匹配g2, (f6<->g2) | g2成功 |
| [9]. 匹配g3 (f15<->g3) | X←Mary, Z←skating |
| [10]. 匹配g4, 按 Z找Y (f12<->g4) | Y←Mary, Z←skating |
| [11]. 查询全部谓词满足 (f3<->g1) (f6<->g2) (f15<->g3) (f12<->g4) | X=Mary, Y=Dave, Z=skating |
| [12]. 从 (f4<->g1)再查找有无另一组解重复第[7]-[11]步, | 结果无新解。 |

本例是在一顺序的数据结构(子目标结点, g1, g2, g3, g4)上做回溯，基本思想是从某一结点出发一试探新子目标一无解回溯到原目标另取新值(直至穷举所有值)一再试探完成所有子目标。回溯的数据结构更一般的是多叉树，回溯的每一步都有许多逐个匹配。这就要靠隐式迭代完成。显然，回溯算法效率是不高的。

7.5 异常处理

程序运行中常常有意想不到的程序错误，例如，数组越界，栈溢出，被零除等，都会引起程序无法执行下去，也就是出现了异常(exception)情况。在早期语言的程序中，出现了这种情况就中断程序的执行，交由操作系统的运行程序处理。操作系统先调用诊断程序，如果诊断程序能处理，处理后继续执行。否则打印尽可能多的信息后，停止这个程序。

军用语言的嵌入式应用不能容许程序有了错误交由监控器处理，或停止工作。这样，正在执行飞行或战斗任务的军器会因失控而毁灭。因此，即使有了程序错误，程序员也应事先要想好应急措施。使之降级运行或实施某种动作以求安全。这些非常规程序叫异常处理段，待异常引发后执行。因此，Ada引入异常处理机制。90年代非军用语言C++也有了异常。异常中断程序执行转而执行异常处理段，在这个消极意义上，它也是程序控制。

7.5.1 异常定义与异常处理段

异常是引起程序不能正常执行的事件。为了处理异常必须先给异常(事件)命名。Ada语言就预定义了五种异常名：

- CONSTRAINT_ERROR 凡取值超出指定范围叫约束异常。

- NUMBER_ERROR 数值运算结果值实现已不能表达叫数值异常。
- STORAGE_ERROR 动态存储分配不够时，叫存储异常。
- TASKING_ERROR 任务通信期间发生的异常叫任务异常。
- PROGRAM_ERROR 除上四种而外程序中发生的一切异常，都叫程序异常。
如子程序未确立就调用等。

用户可在任何程序的声明部分定义异常名：

 <异常名>: exception;

即为用户定义的异常，预定义的异常可以显式也可以隐式引发，而用户定义的异常必须显式引发。用以下引发语句：

 raise [<异常名>];

引发语句可出现在任何可执行语句所在的地方。一旦执行到本句，则本程序块挂起转而执行本块异常处理段中同名的异常处理程序。

每个程序块都可以在该块末尾处设立异常处理段：

declare:

 <正常程序声明>; [<异常声明>];

begin

 <正常程序代码>; [<引发语句>];

exception

 when<异常名>=><处理代码> --异常处理段

 when ...

 [<引发语句>] --一般再次引发原异常（缺省名字）

end;

Java/C++的异常处理和传统的异常机制基本一样，只是它是面向对象语言。一切都是对象，异常也是对象。系统提供一个throwable类，由它派生多个子类，定义各种异常。不仅如此用户定义的异常也由它的子类派生。这样，随着长时间的积累各种程序中见到的异常都可以处理。

Java/C++的异常引发也是两种：显式引发通过throw/throws(抛出)语句；自动引发又运行时系统判断是哪种预定义的异常。Java/C++认为程序员定义异常总有自己的估计。给捕捉异常划定一范围。可以减少捕捉开销。当然，范围之外的异常就靠自动引发了。

7.5.2 异常引发与异常传播

用户定义的异常必须显式引发，而系统定义的异常一般自动引发。因为数组越界，被零除程序员是无法指定出错地点的。但在处理段中可再次显式引发，其目的是让外嵌的程序块的处理段来处理。

如果没有exception段，或有了exception段找不到匹配的处理程序。它自然要往上找，看各外嵌程序块中有没有与此同名的异常处理程序。如果没有，则在调用语句所在的块中引发这个异常（隐式）。如果还找不到，就到更高层调用的程序块中找。如果还找不到则主程序停止执行交由操作系统处理。这就和没有异常设施的程序一样了。我们把这种逐级引发找处理段称为异常传播(propagating exception)。

例7-8 Ada的嵌套异常及异常传播

```
with TEXT_IO, MATH_FUNCTIONS;
use TEXT_IO, MATH_FUNCTIONS;
procedure CIRCLES is
  DIAMETER:Float;
  MAXIMUM: constant Float:=1.0e6;
```

```

[0]TOO_BIG_ERROR: exception;
    PUT ("please enter an float diameter.");
    begin
[1]loop
    begin
        GET (DIAMETER);
        if DIAMETER > MAXIMUM then
[2]    raise TOO_BIG_ERROR;
        else
[3]    exit when DIAMETER <=0;
        end if;
[8]    ONE_CIRCLES(DIAMETER);
        PUT("Please enter another diameter(0 to quit):");
[4]    exception
        when TOO_BIG_ERROR=>
            PUT ("Diameter too large ! please reenter.");
[5]    end ;
[6]end loop;
        PUT ("processing finished.");
        PUT New_line;
    end CIRCLES;
    procedure ONE_CIRCLE (DIAM:Float) is
        RADIUS, CIRCUMFERENCE, AREA:Float;
    begin
        CIRCUMFERENCE:=DIAM*22. 0/7. 0;
        RADIUS:=DIAM/2. 0;
        AREA:=RADIUS*RADIUS*22. 0/7. 0;
        PRINT_CIRCLE (DIAM, RADIUS, CIRCUMFERENCE, AREA);
[7]exception
    when NUMBER_ERROR =>
        raise;
    end ONE_CIRCLE;

```

这是一个非常简单的程序，每读入一个直径计算并打印半径、圆周长和面积。是一个交互式程序，直到输入直径 ≤ 0 停止。CIRCLES过程中在[0]处声明异常TOO_BIG_ERROR。每当输入直径值大于最大值MAXIMUM时显式引发异常，在[2]处。引发后程序控制跳到[4]进入异常处理段。名为TOO_BIG_ERROR的处理程序。本程序打印一行提示 后经[5]、[6]又回到[1]，循环再做。如果直径 ≤ 0 ，它就跳到[6]出循环，打印“处理完了”信息后结束本程序。

如果不大不小，则调用ONE_CIRCLES过程，这个过程中没有定义用户的异常。但显式指明系统定义的异常NUMBER_ERROR的处理。即由于不可知的原因计算 π 值(220/7.0)或其它数值时发生了问题，它就自动引发NUMBER_ERROR，引发后首先在自己的程序块中找处理段，进入[7]终于找到，但其处理是再次引发NUMBER_ERROR，本段没有其它处理了，它就把这个引发通过CIRCLES中调用语句传出，则在[8]处引发该异常。于是它又在CIRCLES中找处理段，进入[4]后没有这个处理程序，这才迫令本程序停止运行，把NUMBER_ERROR异常交给操作系统的运行管理程序处理。这就和其它传统语言处理一样了。

异常引发和处理后不像子程序调用返回原处，而是从处理它的处理段出口，执行以下程序。只有交由操作系统处理，程序才结束悬挂，终止执行。否则主进程一直等着(悬挂)异常进程的处理。

Java的异常我们在第15章中还要介绍。此外，只演示其抛出过程及传播路径。

例7-9 Java的异常定义及异常传播

```
[1]  class UserException extends IOException{//定义用户异常类
        private int i;
[3]  UserException(int j){                //构造一个用户异常对象
        i=j
    }
[6]  public String OnlyTest(){                //测试异常对象构成否
        return "UserException"+i;
    }
[9]  }
[10] public class Example {
[11] static void dme(int j) throws UserException{
[12]     system.out.point/n("dme"+j);
        if(j>0)
[14]         throw new UserException(j);
[15]         system.out.point/n("good");
    }
[17]     public static void main(string args[]){
[18]         try{
            dme(4);
            dme(15);}
[21]         catch(UserException e){
            system.out.point/n("Exception is"+e)
        }
    }
[25] }
```

从[1]到[9]行是用户定义的异常类。一个方法（[3]）是构造本类实例，一个方法只是测试该实例用上没有，返回调用号。

第[10]到[25]行是一个示例Example类，它有两个方法，一为dme，一为main.main()是Java的规定，不能像C++那样作为单独的主调函数。只是一主调方法，由它向dme发消息dme(4), dem(15)。

第[14]行的throw是因为j>10时要扔出一个新的UserException, 有了new则按第[3]行的构造方法构造新扔出的异常。

第[11]的throws xxxx必须和方法联用, 指明该方法只(检测)扔出xxxx类的异常, 当然常规检测除外。

第[17]行的主调方法内容很简单, 在try块中故意放dme(15) (它大于10). 当调用发生时它经过[11]到[13]行扔出"UserException15"那么它就在[21]行中作参数匹配. 结果相配就执行打印语句得出:

"Exception is UserException15"

如果在[21]行中找不到匹配, 本例交操作系统处理.

这两个例子说明用户定义异常一般引发和传播的方式. 表明异常引发和处理的控制此函数/过程调用要复杂得多. 但它归于顺序控制器.

7.6 小结

- 表达式求值次序是最低层的程序控制。它的上层是语句组的四类控制：顺序、选择、迭代、子程序调用。再上一层是块间通信和约束。
- 只要有赋值、简单的if和goto三种语句就可以编写一切命令式计算过程。
- goto语句是有害的，但它有利于提高效率。当今程序中依然保留它或它的变体叫顺序控制器（符）。
- 结构化程序是只有一个进口，一个出口的三种基本结构(顺序、选择、迭代)的复合。
- 语句括号消除了选择结构的二义性。if_endif 比begin_end更好。
- 选择控制可以用命令表达式和条件表达式实现，函数式语言更仰赖条件表达式。
- FORTRAN在GOTO上想出许多表示法，近代语言在迭代上想出的表示法更多。为了方便(灵活性)保持更多的冗余性。
- 迭代控制有显式的和隐式的，声明式语言、第4代语言更多仰赖隐式的。隐式迭代程序清晰简炼，但程序员无法干预。
- 通用循环是while_do和do_until的复合，无限循环加上不同的约束子句可以得到不同的循环控制。
- 计数循环是最老的控制结构之一，循环控制变量、三个表达式各语言都有差异。控制范围在程序体中可否改变也有差异。
- C语言的计数循环用迭代元素概念。它对迭代元素不加任何限制，所以C语言计数循环变通写法最多，最有用，但可读性差。
- 隐式迭代是系统自动在同构的数据结构上完成迭代。回溯是隐式迭代。
- 异常处理是一种复杂的程序控制。是将系统程序控制程序的办法让程序员使用。用户定义异常、指定引发、写出异常处理段。引发的规则是先查本段再查调用段，直到交给操作系统为止。

习题

- 7.1 为什么函数式程序对于顺序控制显得不重要。它的选择、迭代控制靠什么实现？
- 7.2 为什么结构化程序结构比早期非结构程序易于排错？详细列出它们的优缺点。
- 7.3 为什么直到近代语言，子程序的返回语句可以多个这不违反了一进一出原则吗？
- 7.4 试述case语句和嵌套if同异。那个效率高？
- 7.5 无限循环是一进一出的结构吗？它一般用在什么地方？只有它才能出现死循环吗？
- 7.6 有以下类pascal程序片断：

```

k:=3
FOR i:=1 to K do
BEGIN
    Writeln (i,k);
    k:=k+1
END;
Writeln(i);

```

这个程序终止吗？为什么？最后一句写出什么？

这个程序终止，因为循环中递归变量值改变，并最终达到终值，而pascal程序中又不能改变循环终值最后一句写出4。

7.7 何谓迭代元素？它和常规计数循环有什么不同？

7.8 设计一个for 循环要考虑哪些问题？试写一设计需求规格说明。

考虑的问题。

①循环控制变量（初值）、终值、增量表达式、条件表达式、循环体；

②循环控制变量类型：整型、离散、实型；循环控制变量：全局静态、局部静态、局部变量；

③终值和增量表达式在循环中允许改变否？

④条件表达式EXP2在顶部求值还是在底部求值。

7.9 写出一个Prolog程序，它在一颗多叉树上作回溯匹配。

7.10 为什么异常处理后不能返回到异常引发的地方。如果一定要那样有什么问题。

7.11 试编写一C程序，利用系统调用实施绝大部分异常处理功能，该程序要求。

[1] 打开输入文件infile，若打开失败打印必要信息。跳出程序。

[2] 打开输出文件outfile，若打开失败打印必要信息，跳出程序。

[3] 若打开无问题，扫描数据不符，打印必要信息，关闭文件后跳出。

[4] 若输出文件已为EOF，写不进去，打印必要信息，关闭文件跳出。

7.12 C语言中有一顺序控制器continue它的作用是什么？写出continue出现在for循环时等价语义的while_do形式：

```
for (Exp1, Exp2, Exp3)
    .
    .
    .
    continue
    .
    .
```

作用：结束本次循环，即跳过循环体中下面尚未执行的语句，接着进行下一次是否执行循环的判定。 等价语义的while_do形式：

```
Exp1;
while(Exp2) do
{
    Exp3;
    (1);
    if(!Exp4)
    {
        (2);
    }
}
```

7.13 FORTRAN也有一顺序控制器CONTINUE，它的作用是什么？

7.14 用户定义异常时，名字复盖NUMBER_ERROR并写出了自己的异常处理段when NUMBER_ERROR=>...可以不可以执行为什么？

7.15 有以下Ada程序片断：

```
type MONTH is (JAN, FEB, MAR, ... , NOV, DEC);
RAINFALL: array (MONTH) of Float;
procedure GET_WEATHER_DATA is
begin
    for AMONTH in MONTH loop
        begin
            GET (RAINFALL(AMONTH));
```

```

exception
    when DATA_ERROR=>
        PUT ("Invalid data for "); PUT (AMONTH);
        SKIP_DATA_ITEM;
        RAINFALL (AMONTH):=0.0;

    end ;
end loop;
—
end GET _WEATHER_DATA;

```

该程序读取每月雨量数据以判断天气。若雨量RAINFALL数据格式不对，引发异常，处理时RAINFALL数组中该项为零，并读下一项。

异常DATA_ERROR是系统定义的异常还是用户定义的？它在哪里引发？不用异常能否处理？若能改写之。

答：异常DATA_ERROR是系统定义的

它在语句GET (RAINFALL (AMONTH)) 处引发。

不用异常可以处理，改写如下：

```

begin
    for AMONTH in MONTH loop
        begin
            GET(RAINFALL (AMONTH));
            if ( RAINFALL( AMONTH ) /= float ) then
                PUT ("Invalid data for "); PUT (AMONTH)';
                SKIP_DATA_ITEM;
                RAINFALL (AMONTH):=0.0;
            endif ;
        end loop ;
    end GET_WEATHER_DATA ;

```

第8章 程序的抽象与封装

我们早已指出抽象是程序设计语言的基本手段和技术，有了抽象我们才能脱离指令码和地址码，才有语句和表达式。第6章函数和过程抽象使我们能在更高一个抽象层次上设计程序，这样，我们在组织更复杂的程序时，直接用诸如求平方、求正弦、交换两个数、B树查找等函数或过程，而不必过问它们的实现细节，特别是利用抽象将规格说明和体分开，可以使程序清晰，便于修改。近代语言不希望一个函数或过程做很多事，一个函数或过程只实施一个功能。所以，一个应用程序系统就可能有成千上万个程序单元(函数或过程)。至于程序的数据，前面已经说过，最好分到每个函数和过程之中，但这是不可能的。试想，运行1000个各不相同的函数或过程能算出什么呢？仍然是1000个小程序。程序本质需要数据既相关又独立，Pascal解决这个问题的办法如图8-1的嵌套共享数据。

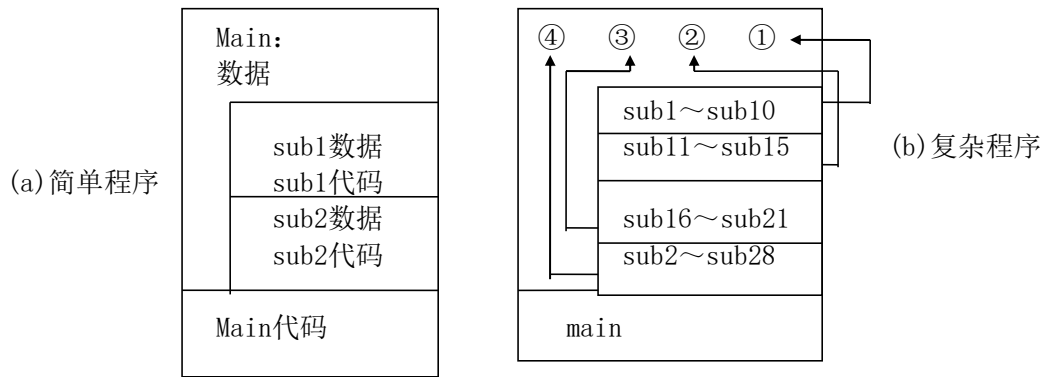


图8-1 Pascal的嵌套共享数据

在图8-1(a)中，sub1代码除加工自己的数据而外不能加工sub2的数据，但能加工Main的数据。如果需要，sub1还可作类似的更深一层嵌套，这样，很大的既独立又共享的程序就可构造出来。70年代以前就是这样做的。但程序进一步扩大问题就出来了，如图8-1(b)所示。如有28个子程序分四组加工四组数据，我们怎能保证程序代码中有了错误不去加工其它组的数据？除了人为地再把它分成四个嵌套层之外，别无办法，这样结构化程序带来的易识性完全没有了。更有甚者是一个小子程序有了修改，整个大程序要重新编译。所以，Pascal是不能编写大应用系统的。

于是，能不能在函数或过程以上再提供一层抽象的语言机制，以方便程序表达？70年代末就是在这种思想上发展了Modula-2和Ada语言，它们提供新的封装机制，把数据和子程序分别包装起来，构成可分别编译(separated compilation)模块(module)或程序包(package)。编译后的包模块可以自由地组成一个程序系统。其逻辑结构如图8-2所示：

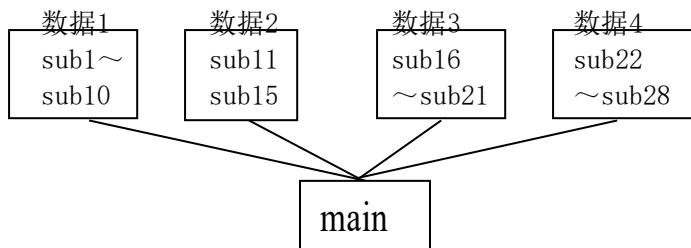


图8-2 模块组成程序系统图

本章我们研究程序包(模块)的作用、表示、实现, 以及更为抽象的封装一类属。

8.1 模块和包

模块在程序设计中是一个很通用的概念, 小到几条指令或几个数据, 大到一个子系统都可以叫模块。我们这里把模块定义为封装的一组共享数据和一组子程序的程序单元, 模块(或包)名是对这个封装的抽象。当子程序为空时, 即为数据包, 相当于FORTRAN的BLOCK DATA。当数据为空时, 即为程序包, 相当于一般语言系统的库程序包(Library package)、数学包。

封装的模块在现代程序设计语言中有不同的术语。有的只是分别编译的手段, 无独特语义, 有的在程序世界代表不同的实体。随着抽象数据类型ADT技术的发展, 它们日益成为有用的程序对象。当今不同语言模块术语如下:

| | |
|------------------------------|--------------|
| C | 分别编译的文件集 |
| Ada | 程序包(Package) |
| Modula | 模块(Module) |
| CLU | 簇(Cluster) |
| C++, Smalltalk, Simula, Java | 类(class) |

从封装的角度, 模块只是把数据和子程序体现的操作封装起来。它可以把杂乱无章的数据和操作放在一个模块里, 只不过为了转储、编译、使用方便, 不提供抽象层上的语义。也可以把数据和操作有机地组织起来成为客观世界对象的程序对象映射。例如, C++的类。

模块封装和子程序封装一样, 必然有一个与外界通信的界面。和子程序封装不一样的地方是不仅可以传递数据还可以传出类型声明, 并通过界面作子程序调用。程序员只要善于控制界面的大小, 即可使模块成为有声有色的程序对象。为程序提供上层抽象的表达能力。

从程序世界的角度, 即使是C语言文件模块, 我们也希望按它的上层语义来使用。以下的讲述都是这个前题。

8.1.1 模块的一般形式

```
package PKG_NAME is
    <私有数据, 操作的声明>
    <公有数据, 操作的声明>
    [<保护数据、操作的声明>]
end PKG_NAME;
```

名字PKG_NAME封装了整个模块, Ada, Modula-2把程序包分成规格说明(specification)和体(body)两部分, 它们代码物理相连, 包名一致, 包规格说明中声明的数据全是公有的, 且对体中有效, 体中声明的数据全是私有的。子程序的规格说明放在包规格说明中是公有的, 其定义部分放在包体内。包体内定义的其它子程序是私有的。Ada的程序包规格说明就是外界可见的界面。

这里公有和私有是以程序包为程序单元, 能为包外程序实体使用的数据(包括类型)与操作作为公有的(public)否则私有(private)。所谓保护性(protected)部分是对有继承性的子模块(类)而言, 以后(第10章)还要讨论。无论公有私有数据, 包内的子程序都可以用。程序包的示意图如图8-3(a), 相应代码表示法见例8-1。

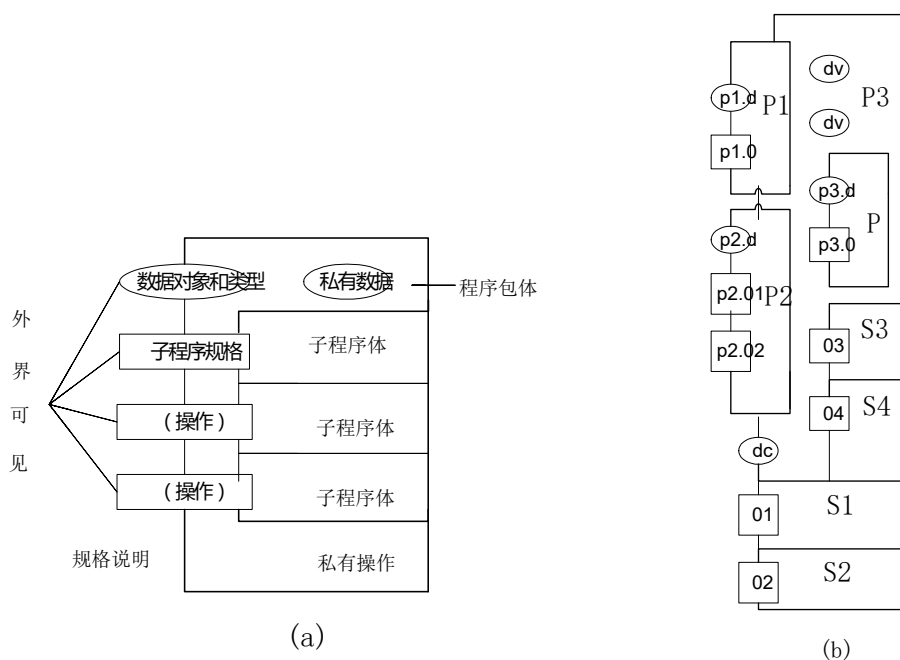


图8-3 程序包示意图

例8-1 Ada 的复数程序包

```
package COMPLEX is
  type NUMBER is record
    REAL_PART:FLOAT;
    IMAG_PART:FLOAT;
  end record;
  function "+"(A, B: in NUMBER) return NUMBER;
  function "-"(A, B: in NUMBER) return NUMBER;
  function "*" (A, B: in NUMBER) return NUMBER;
end COMPALEX;
```

有了这个程序包我们可以编出复数应用程序:

```
with COMPLEX;
use COMPLEX;
procedure MAIN is
  COMP_1: NUMBER := (1.0, 2.0); --1+2i
  COMP_2: NUMBER := (3.0, 4.0); -- 3+4i
  W, X, Z: NUMBER;
begin
  W := COMP_1 + COMP_2;          --W = 4+6i
  X := COMP_2 - COMP_1;          --X = 2+6i
  Z := COMP_1 * COMP_2;          --Z = -5+10i
end MAIN;
```

程序包中公有的类型NUMBER, 操作"+", "-", "*"在写了with COMPLEX的MAIN主子程序中直接可用。写use只是为了不写前缀, 如COMPLEX.NUMBER可省为NUMBER. 至于如何实现复数的加、减、乘, 则见程序包体中的函数体的定义(此处略)。用户定义的"+", "-", "*"为运算符重载。

模块本身仍可嵌套如图8-3(b)，图中用P代表程序包，S代表子程序，d代表数据，o代表操作界面。模块P界面上嵌套两模块P1，P2，因而模块P的用户可见P1.d，P1.o，P2.d，P2.o，P.dc。当然也可以见到S1,S2的O1，O4，O2。P内嵌程序包P3，和子程序S3，S4 私有数据dv包外不可见。在P的包体内P3.d，P3.o，dv，o3以及P的界面上所有实体均可见。

8.1.2 模块程序的结构

模块本身可以作为提供数据和操作的程序对象，是程序系统的资源。它是被动的如同扩大的子程序定义，没有其它程序对象触发它(通过对界面上子程序的调用或引用)，它是不会动作的，例8-1就说明了这个问题。with子句是主子程序MAIN和程序包COMPLEX联系的纽带。有了它相当于在编写主子程序MAIN时刚编了COMPLEX包。当MAIN中第一个赋值语句为W赋值时，它引用运算符“+”，把COMP_1和COMP_2与A，B匹配，得到一个返回值REAL_PART = 4，IMAG_PART = 6(即4+6i)。

类似这样的程序结构，如图8-2，我们叫它主程序驱动结构。主程序(它本身也是一模块)驱动各模块，不排除P1模块执行时向P2模块发出调用(通信)，P2向P4，P4向P3…即各模块相互通信。

还有一种程序结构是无主程序的模块体系结构。即我们把图8-2的MAIN取消得图8-4。

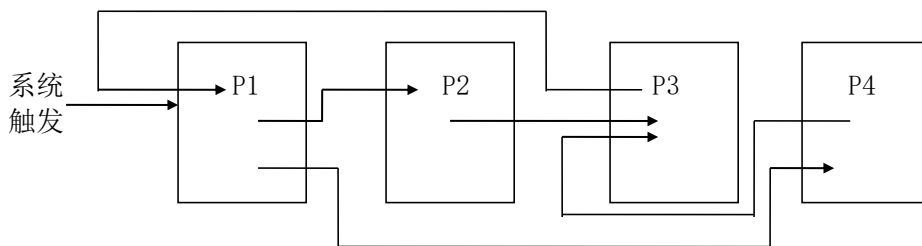


图8-4 无主驱动模块体系结构

操作系统或运行环境，一上来触发P1，(调用一次P1.o1，不一定要做有意义的计算)，P1激活后执行时向其它模块发调用(如图中向P2)，接着P1→P4，P2→P3，P3→P1…相互发调用。这种无主驱动程序执行的结果是什么？那就是将各模块的私有数据从它的初态变成执行后的终态。你认为哪些终态的数据是你需要的就在哪个模块里设一输出子程序把它打印出来，或者集中输出到某个模块的打印子程序。也是靠通过通信(相当于调用)传递参数。

无主驱动模块体系是更加概括的情况，也是当今的发展趋势。但按顺序程序时代传统有一个主控模块似乎更方便一些，对于分布式、并发程序设计范型，无主控更好。因为每个模块都在自己网络结点上被激活，相互发消息。反正本模块的执行控制也转不到接受者的模块上去。所以，模块间的调用意义就变了，形式和调用一样控制传不过去，就等对方响应了。它只能说是相互之间通信(communication)。通信的过程调用叫做消息(message)发送，主动模块(actor)向服务模块(server 也叫接受者)发消息(如同过程调用还要传递数据)，接受者按自己认为合适的时候响应消息(执行消息所要求的动作)。又接受又发送消息的模块叫当事者(agent)。

8.2 抽象数据类型

如前所述，模块仅仅是一个语言结构。用得好可以为我们提供共享数据包或程序包，例如，一个导弹的计算系统，少不了万有引力常数、重力加速度、重要城市经纬度、海面声速、地球半径…等等共享的不变常数。放在一个包里作数据资源。再如，把矩阵转置、求逆、求特征向量、二次型…等等相关函数收集作为程序包以支持所有矩阵运算。也可以既要数据也要加工这些数据的子程序，构成一个相对封闭的状态机，以解决能以状态机模型的问题，例如，词法分析程序。但从程序语言的角度我们最感兴趣的是模块能提供抽象数据类型 (Abstract Data Type简称ADT)。

8.2.1 数据抽象与抽象数据类型

数据抽象是1977年后发展起来的程序设计方法学，图8-1(b)说明图8-1(a)的结构化程序设计不足以处理大的软件系统。必须是如同图8-2的结构。但分解的原则是什么？是数据，有了数据再加上加工它的操作(过程或函数)构成一模块。

为了设计大系统就有必要把封装的界面和它的实现分开。这就是我们介绍过的规格说明和体。“做什么”和“如何做”显式分开后对软件开发的好处我们不再赘述。这个“做什么”里面有什么呢？类型和对象声明(数据集)和加工这些数据集的过程和函数(操作集)的说明，这恰好符合类型定义，是数据集V和该集上的操作集Op组成的二元组：

$$T = (V, Op)$$

只不过我们不能把它按单一特征叫上什么类型罢了，(如同按值叫整、实型，按结构叫记录、数组型)。只好叫它抽象数据类型。抽象的意义还在于它的界面是整个ADT的抽象。正如函数型构是函数的抽象一样。

数据抽象是用到抽象数据类型的程序开发方法学。总结起来它的好处是：

[1] 数据抽象将数据类型的使用和实现分开，先决定数据是如何使用的（外部性态）而不是它的实现内部构造。提高了程序开发的抽象层次，较易控制大型系统的全局。

[2] 数据抽象简化了程序正确性问题。即如果程序不正确，改起来方便。只要规格说明不变，只改变错误程序的体，不影响使用该资源的应用程序。增加了软件的可维护性。

[3] 利用抽象数据类型可以准确地构造新类型。

[4] 数据抽象能实现数据隐藏，即非规格说明界面上的数据，外部用户是不可见的。增加软件安全性, 可靠性。

[5] 作为软件设计技术，模块性利于软件开发，应用程序和模块实现的开发可独立进行。

正是因为数据抽象是一种程序设计方法，非模块语言也可以实现数据抽象。例如，用Pascal定义一个堆栈类型：

```
TYPE stack = ARRAY [1..100] OF Integer;
FUNCTION is_empty: Boolean; BEGIN...END;
FUNCTION is_full: Boolean; BEGIN...END;
PROCEDURE push (el: Integer; S: stack);
    BEGIN...END;
PROCEDURE pop (VAR el: Integer; S: stack);
    BEGIN...END;
```

stack类型面向应用有四个操作is_empty, is_full, push, pop也能体现数据抽象的思

想，但由于没有封装，其它程序有可能操作stack类型的数据，不安全，不能保证可靠性。再者，在Pascal中类型表示，函数和过程体的细节都要写完（此处以‘…’略去）。如果程序员精心用它，也是抽象数据类型。例8-1的NUMBER是语义上的抽象数据类型。如果把NUMBER类型改写为：

```

type NUMBER is limited private;
    .
    .
    .
    类型行为（即操作）
    .
    .
    .
private
    type NUMBER is record
        REAL_PART: FLOAT;
        IMAG_PART: FLOAT;
    end record;
    .
    .
    .

```

更是抽象数据类型了。因为，即使在程序的其他地方声明了NUMBER的对象，它只能用我们在包中定义的几个操作，其它操作均不可以，正如预定义的定点、浮点、整形一样。

8.2.2 利用抽象数据类型构造新类型

由于用户可以自己定义值的集合和操作的集合。我们在以预定义类型组成复杂的新类型值时可以随意增加约束使之描述准确。更有胜者，可以从基始的常量出发，完全由操作隐含描述该类型的值集及其所遵循的行为。例如，程序语言中一般没有有理数这个类型，其值集是：

..., -1/2, -1/1, 0/1, 1/2, 1/3, 2/3, ...

在ML中可用一对整数描述：

```

datatype rational = rat of (int * int)
val zero = rat (0, 1)
    and one = rat (1, 1)
fun OP++ (rat (m1, n1):rational, rat (m2, n2):rational) =
    rat (m1*n2+m2*n1, n1*n2)

```

第一句定义的集合值是：

$\text{Rational} = \{\text{rat}(m, n) \mid m, n \in \text{Integer}\}$

它是带标签“rat”的整数对。m是分子，n是分母。而这种类型是新类型，ML中用datatype而不用简单的类型定义：

```
type rational = int * int
```

因为type只限于用已有类型定义rational名。其它两句一为0元素zero，一为么元素one，OP++定义的是加操作。显然，还可以定义其它操作，此处略。

但这个定义太泛不能避免rat(0, 0), rat(1, 0), rat(m, 0)的情况，也不能分辨rat(3, 2), rat(6, 4), rat(-3, -2)，实质上是一个有理数，例如，当有

```

val h = rat (2, 1)
if one ++h = rat (6, 4) then ... else ...

```

时，不能分辨`rat(3, 2)`就是`rat(6, 4)`作出错误选择执行`else`部分。我们需要的是：

`Rational = {rat(m, n) | m, n ∈ Intege: n>0; }` 且`m`和`n`没有公因子

后面的约束加到`datatype` 定义中是可以的，但没有函数调用它不会自动检查这些约束，于是，ML按抽象数据类型的思想提供了抽象类型。

例8-2 用ML的抽象类型定义有理数

```

abstype rational = rat of( int * int)
with
  val zero = rat(0, 1)
  and one = rat (1, 1)
  fun OP // (m:int, n:int) =
    if n < > 0 then
      rat (m, n)
    else (打印非法，或不是有理数)
  and OP++ (rat (m1, n1):rational ; rat ( m2, n2):ratinal) =
    rat (m1*n2+m2*n1, n1*n2)
  and OP == (rat (m1, n1): rational, rat (m2, n2):rational ) =
    (m1*n2 = n2*n1)
end

```

请注意，`with`到`end`是对值构造函数`rat(int*int)`的约束模块。这个定义产生以下束定：

```

rational → 抽象数据类型
zero → 有理数的0元素
one → 有理数的1元素
// → 构造有理数的函数抽象, 检查被零除
++ → 两有理数相加的函数抽象
== → 测试两有理数相等的函数抽象

```

其中‘//’、‘++’、‘==’是为了区别于预定义的‘/’、‘+’、‘=’，它们是以下函数映射：

```

//: Integer × Integer → Rational
++: Rational × Rational → Rational
==: Rational × Rational → Boolean

```

每当程序员写出一有理数`m//n`时就自动调用`//`函数作了非零检查，所以：

```

val h = 1 // 2
.
.
.
if ont ++h == 6//4 then ... else ...

```

就不会出现错误判断问题。

尽管模块的实现者还是以有标签的`rat(m, n)`实现有理数，但用户只在界面以外以

```
zero , one, //, ++, ==
```

这些函数表达和使用有理数而不会出现歧意 这再一次说明使用 and 实现(或表示)的脱离。

8.2.3 构造函数和析构函数

抽象数据类型和内定义类型相似, 我们在使用Integer, Truth_Value类型时, 从不注意它们在机内的实现, 只知道它们的值集和可用的操作。也没有过问操作如何实现, 可以想象整除时(/)当第二操作数为零必然引发“被零除异常”。抽象数据类型要由程序员一一定义出来, 而操作的用户不必“记清楚”, 知道能做什么就够了。

对于复杂的抽象数据类型, 它的值集并不像Integer和Truth_Valae这么简单。上述有理数就说明这了点。我们通过zero, one, //才把它们描述清楚, 每当用户写:

```
rat (0, 1)
rat (1, 1)
OP // (m, n)
```

函数调用时我们就知道它在要求有理数zero元, one元和m/n的值, 所以, 它们是构造出该类型值的构造函数(constructor)也叫构造子。

构造子并不是什么新思想。ML面向值, 它的构造子是值构造子, 我们在Pascal, Ada的数组声明中:

```
AR: array [ 1..10] of Integer ;
```

array ... of ... 就是一个数组类型构造子。它构造一个十元素整数的数组, 并将标识符AR与其束定起来。

值构造子构造本程序需用的值, 对象构造子构造本程序需用的变量(对象), 类型构造子构造出本程序需用的类型(从抽象类型构造具体类型, 见下章)。它在概念和实施上早期语言都有, 只是在面向对象中才要求用户显式定义它。

与构造函数对应的, 如本程序不再需用这个类型的值或变量时, 则有析构函数(Destructor)或称析构子, ML中面向值一般不必说该值不要了。如果真要写一个析构函数, 上述有理数的例子的析构函数是:

```
fun float (rat (m, n): rational) = m/n
```

即把有理数划归浮点数(包括无理数近似值), 该类型的这个值就不存在了。

8.2.4 以分别编译文件实现抽象数据类型

C语言和Pascal一样也没有提供程序模块机制, 但C语言的设计者由于和UNIX操作系统的渊源, 一直利用操作系统的命令(语言以外)来完成近代的新技术。这也是C不易打倒的原因。

C语言实现模块性不是简单地使用操作系统命令, 因为这样做程序员免不了有疏忽遗漏导致错误。UNIX的新近版本都发展了make文件设施。ANSIC全部采纳这些设施, 以下C语言的例题均为ANSI C的。

一个文件可单独编译, 通过连接程序连成一个可执行程序。C语言的模块程序至少由四个文件组成, 其意图是:

- 由make文件联成整个程序。它将各个成分列表并指出相互依赖关系。该文件是由make设施解释的操作系统命令集。经解释后得到可执行的机器码就绪于装载待运行。

- 头文件将其它文件(模块)共享的符号的定义和声明放于其中, 这些符号多数是程序对象但有些更低级只能统称符号(标识)。

- 主模块文件, 包含净主程序本身代码, 其中声明的对象不与其它文件(模块)共享。

- 一个或多个由主程序调用的函数文件, 它们通过主程序共享头文件, 但可单独编

译。这些模块通常用于处理缓冲和存储管理的协例程。

例8-3 用C语言定义抽象类型

这是一个交互式对数组元素作归约的程序。其中数组元素类型NUMBER 即为抽象类型。此外，运算符和数组元素个数也是参数化的，以大写标出。这四部分程序如下：

[1] 头文件：“ modules.h”

```
# define LEN 4           //数组长度参数化
# define D “%d”         //所希望的格式指明符
typedef long int NUMBER; // NUMBER实例化为标准类型
extern NUMBER reduce ( NUMBER (*) ( ), NUMBER)
```

最后一行告诉编译器reduce另有模块。前三句是LEN, D, NUMBER的实例化，当然可以是其它数据和类型。

[2] 主模块文件：“ sumup.c”

```
# include <stdio.h>
# include “modules.h”
# define OPS 6           //运算符有6种例化
NUMBER fi_plus(a, b)     NUMBER a, b; { return a+b; }
NUMBER fi_times(a, b)    NUMBER a, b; { return a*b; }
NUMBER fl_or(a, b)       NUMBER a, b; { return a||b; }
NUMBER fl_and(a, b)      NUMBER a, b; { return a&&b; }
NUMBER fb_or(a, b)       NUMBER a, b ; { return a|b; }
NUMBER fb_and(a, b)      NUMBER a, b ; { return a&b; }
static NUMBER (*op_ar [OPS] ) ( ) =
    fi_plus, fl_or, fb_or, fi_times, fl_and, fb_and;
static char * label [OPS] =
    “ plus”, “ logical or”, “ bitwise or”,
    “ times”, “ logical and”, “ bitwise and”;

main ( )
{
    NUMBER ar [LEN];
    NUMBER * ar_last = &ar[LEN-1]; //标记数组最后元素
    NUMBER * p;                     //扫描数组的指针
    int k;                           //函数数组索引
    puts (“This programs demonstrates some whole_array operations.\n\n”);
    do
        printf (“please enter %d integers seperated by spaces.\n”, LEN);
        p = ar;
        while (p<= ar_last && scanf(D, p) == 1 )++p;
        if (p<=ar_last)
            printf (“Premature EOF or conversion error job terminated.\n”);
            exit(1);
        while (getchar( ) != ‘\n’);
        putchar (’\n’);
        //将每个运算符数组的操作用于数字数组上
        for (k = 0; k<OPS; k++)
            printf (“ %s ”D”\n”, label [k], reduce (op_ar[k], ar));
        printf (“Do you want to enter more data? (y/n) \n”);
```

```

while (getchar ( ) == 'y');
}

```

本程序中static指针数组私有于本文件，最主要的是归约调用reduce(op_ar[k], ar)，其它都是为此准备的。

[3] 函数文件：

```

"reduce.c"
# include " modules.h"
NUMBER reduce (NUMBER (* op) ( ),  NUMBER ar)
{
    int k;
    NUMBER sum;
    sum = ar [0];
    for (k = 1;  k<LEN;  k++)
        sum = (* op) (sum,  ar[k]);
    return sum;
}

```

由于连接其它模块定义的大写词本模块可见，调用本函数OPS次，将“加”、“乘”、“逻辑或”、“逻辑与”、“位或”、“位与”全做一遍。

[4] 用户通过makefile指明各文件关系

UNIX所有的应用程序都以makefile为根，makefile还有自动版本管理功能，将最当前的版本组成可执行程序。如果模块的源程序修改而又没有重编译，则不会将其目标模块送去联编，且编过的模块会自动跳过，不会重复编译。用户在自己目录下键入：

```

make sumup
sumup:  main.o reduce.o
    //告诉连接程序两目标码文件联编后名为sumup
cc-o sumup main.o reduce.o
    //调用编译、连接组成名为sumup的目标文件
main.o: main.c modules.h
    //两源文件一起编译后成main.o
cc-o main.o main.c
    //调用编译产生名为main.o的目标文件
reduce.o:  reduce.c modules.h
    cc-o reduce.o reduce.c

```

以后按sumup名字即可运行了。

用模块文件实现抽象数据类型最大的缺点是不完全属于语言系统，排错要涉及操作系统命令和语言两方面。

8.3 类属

我们已经知道函数是表达式集的抽象、过程是命令集的抽象，类属(generic)是声明集的抽象。也就是声明的变量、类型、子程序都是参数化的。这对抽象数据类型的表达更为有用。因为在高层开发程序时，我们不太关心数据的具体类型，例如，交换两个数的程序，是整数、字符、自然数、全偶数、浮点数、定点数对我们都无所谓。我们可以把它写成(Ada)类属子程序。

例8-4 Ada类属子程序

```

generic
  type ELEMENT is private;           --类属类型参数
  procedure EXCHANGE (FIRST, SECOND: in out ELEMENT);
  TEMP: ELEMENT;                     --程序中用类属形参
begin
  TEMP := FIRST;
  FIRST := SECOND;
  SECOND := FIRST;
end EXCHANGE;

```

按照Ada的类属形实参数匹配规则，类属参数是私有类型则可和任何类型参数匹配。使用时可实例化：

```

procedure INT_EXCHANGE is new EXCHANGE (Integer);

```

预定义的Integer与ELEMENT匹配，即在procedure中任何出现ELEMENT的地方以Integer代。这个实例子程序就是交换两整数的过程。同样，

```

procedure CHAR_EXCHANGE is new EXCHANGE (Character);
procedure SMALLINT_EXCHANGE is new EXCHANGE (MY_INT);

```

这两个声明实例化了类属过程，一个实例是交换字符过程，另一个实例是交换两个小整数的过程。不过后者作实例声明时，要求前此已经定义了MY_INT类型。

类属在不同语言叫法不同，如C++中称为模板(template)，实质是一样的。

8.3.1 类属定义一般形式

```

generic
  <类属值> | <类属变量> | <类属类型> | <类属子程序>
package GENERIC_PACKAG is
  --用类属参数的包体
end GENERIC_PACKAG;

```

把procedure 关键字换掉package就是类属过程。

类属实例化是在有了实在参数以后作实例声明：

```

package INS_GENERIC_PKG is new GENERIC_PACKAG (<实参变元>);

```

类属形实参数的个数、次序要匹配。以下是类属值参数的例子。

例8-5 不同终端要求屏幕上行列不同，我们写了一个终端程序包，将行、列参数化。

```

generic
  ROWS: in Integer := 24;
  COLUMNS: in Integer := 80;
package TERMINAL is
  .
  .
  .
end TERMINAL;

```

对于小屏幕、字处理可设两例：

```
package MICRO_SCREEN is new TERMINAL (24, 40);
package WORD_PROCESSOR is new TERMINAL (ROWS => 66, COLUMNS => 132)
```

8.3.2 类属参数

原则上声明中的所有程序对象都可以参数化，其实现机制如同子程序中参数结合，类属值一般是复制机制，类属对象(变量)、类型、子程序用引用机制。取决于实现。

Ada语言类属参数及其实例化已举出两个例子。例8-5是值参数，以in模式指示，为变量赋的初值(24, 80)是缺省参数。如果指出实参值是(24, 40)，(66, 132)则按实参。对象(变量)参数形式和本例大体相同，只是模式是inout。实例化时要有声明过的变量(ROW_V, COL_V)。

例8-4是类型参数化的例子，由于Ada是强类型，只有私有类型形参可以和任何类型实参相匹配。如果换成(< >), range< >, delta< >, digits< >则只可以和任何枚举、整型、定点、浮点类型实参匹配。

如果类型T是应用到类型T值上操作的规格说明，编译器将检查每一个类属实例保证：

作用到实参类型上的操作 \supseteq 指定应用到T上的操作

并检查类属抽象本身，保证：

指定应用到T上的操作 \supseteq 类属抽象中用到T的操作

类属类型参数和子程序参数不同之处在于类属类型参数可以相互依赖(这是顺序声明带来的)请看以下综合例题。

例8-6 Ada的类属类型和子程序参数

程序包中封装一分类和归并程序。类属包按数组类型设计，它的元素类型，按升序还是降序的优先操作，都将它们参数化：

```
generic
  type ITEM is private           --数组元素类型
  type SEQUENCE is array (Integer range < >) of ITEM;
                                   --数组类型参数化。ITEM马上就用了
  with function PRECEDES (x, y:ITEM) return Boolean;
                                   --参数化操作未定升降序

package SORTING is
  procedure SORT (S: in SEQUENCE);
  procedure MERGE (S1, S2: in SEQUENCE; S: out SEQUENCE);
end SORTING;
```

有了with说明函数PRECEDES是参数化的，它的参数也是参数化的ITEM。它实际上是一个比较运算符。在SORTING. SORT过程中以如下形式用：

```
if PRECEDES( S(j), S(i)) then ... else ... endif;
```

如果我们已经写了两个重载函数：

```
function "<=" (X, Y: Float) return Boolean;
function ">=" (X, Y: Float) return Boolean;
```

那么就可以作以下设例声明：

```
type FLPAT_SEQUENCE is array (Integer range < >) of Float;
```

```
package ASCENDING is new SORTING (Float, FLOAT_SEQUENCE, "<=");
package DESCENDING is new SORTING (Float, FLOAT_SEQUENCE, ">=");
```

这样，就有了一个升序，一个降序的两个程序包。

这个例子说明的类属子程序参数匹配。其关键字、返回值类型、参数都要作检查。顺便说一句，Ada一般子程序的参数不允许是子程序，而类属则可以。所以，Ada在涉及子程序参数化之处均用类属处理。

8.3.3 类属实现的静态性

类属是一种概括抽象，类属程序本身是可编译的。编译时类属参数是不能执行的抽象参数。在编译类属设例程序时，相当于宏替换，将所有抽象参数换成实际参数。替换了的实例目标码是一个可执行的新拷贝，设几次例复制多少次，对于这种静态实现，熟知C语言预处理宏的人是很容易理解的。

就类属程序的参数而言，例如类属类型参数，它可以实例化为若干类型，我们叫它多态类型(polymorphic type)。多态类型提高了程序的表达能力，因为强类型是为了程序不出错而为之，每个程序对象编译时都要确知其类型，参数匹配，表达式求值赋值均要作类型检查。但强类型对于通用操作的程序是十分不利的。例如前述交换两个数的程序，其实它对任何基本类型的两个数据都可以交换，然而强类型语言都要对每种数据类型另编一个完全类似的程序，Ada的类属解决了这个问题。只不过不由程序员重复，翻译器替你重复。它是静态替换。

很自然，人们要问能不能把这种灵活性扩充到运行时刻呢？对于解释型语言就比编译型有利多了，它可以根据上下文执行的当时情况决定它是与哪一个实例类型、实例过程相结合。也就是说，不必静态作出很多版本。而是像子程序那样运行之中将参数束定，类属程序只需一个版本(当然，参数版本一个也不少)。我们把这种参数例化的束定称为动态束定(dynamic binding)或后束定。动态束定的好处在于程序的可扩充性，它可以随时增加实例参数版本。动态束定技术在面向对象中得到充分地体现。

8.4 对象和类

封装为程序员提供的抽象数据类型使程序语言上了一个新台阶，可以更接近地描述客观世界对象，而不必拘泥于语言系统提供的有限的基本类型，正如高级语言提供的抽象让程序员不必拘泥于汇编指令地址码和操作码一样。

封装又是一个隐藏数据的机制，与使用无关的内部数据（虽然也重要）使用者不可见。这样我们要表达的概念更加贴切。请看以下例子：

例8-7 Ada程序包提供的对象

```
with SYSTEM;
package STACK is
  function PUSH (k:Integer) return Boolean;
  function POP return Integer;
  function TOP return Integer;      --返回栈顶元素
end STACK;
package body STACK is
```

```

STKLEN : constant := 10;           --堆栈有十元素
STK:array (1..STKLEN) of Integer;  --整数数组实现的堆栈体
TOS: Integer range 1..STKLEN;
function PUSH (k:Integer) is
begin
    if TOS < STKLEN then
        TOS := TOS + 1;
        STK(TOS) := k;
        return True;
    else return False;
end PUSH;
function POP return Integer is
begin
    if TOS > 0 then
        return STK(TOS);
    else
        return SYSTEM.MIN_INT;      --交SYSTEM包处理
    endif;
end POP;
function TOP return Integer is
    ANS:Integer;                    --工作变量，局部量
begin
    ANS := TOS;
    TOS := TOS - 1;
    return ANS;
end TOP;
begin                               --程序包是资源，以下执行代码在装入后，
    TOS := 0                         --程序运行前首先执行，为了初始化。
end STACK;

```

STACK程序包为我们提供三个操作PUSH，TOP，POP其它都隐藏了。这三个操作操纵一个隐藏的整数堆栈(包外用户可从PUSH的参数看出是要压入整数得知)，但这个堆栈没有名字。例如：

```

with STACK;
procedure MYJOB is
    ...
    STACK.PUSH(My_Value);
    ...
end MYJOB;

```

STACK包就是这个程序对象的代名词。程序对象本身确实存在，但隐藏在包的内部。事实上，根据Ada语法，若用了use STACK，子句PUSH操作的前缀STACK都可以省去。那么，MYJOB中就完全在透明情况下PUSH，POP，TOP这个隐藏的对象(Object)了。我们可不可以从正面研究一下对象呢？什么是对象它给程序设计带来什么好处？

8.4.1 对象

不管是隐式表示还是显式表示：

对象是一个程序实体，它有私有的数据集表征该对象上的状态。它有私有的操作集来改变它的状态。这正好刻划客观世界的对象，例如，一辆汽车，一台电视，一个人，一颗树…都可以用一组数据描述它。每个对象都有各自的行为，行为的结果是数据的改变。汽车跑了路油量减少了；天线正常打开电视，显示画面；吃了饭，做了事，身体重了，技能有了长进…这些行为就是程序对象容许的操作（函数或过程）。所以对象是相对独立的实体。它的操作不是传统意义上的过程调用，尽管语法形式极为相似。它并没有把操作调到主程序中用，而是通过通信（让）对象自己操作自己的数据，一个对象程序就是对象间相互通信。

由于对象的自主性，封装性，可以使程序错误局部化，也有利于程序系统扩充。加一个对象减一个对象对别处没影响（没有耦合的操作）。

在程序设计中对象由类刻划，正如变量由类型刻划一样。类是对象的样板，程序员在程序设计中定义类对象。程序在执行中创建实例对象。由于面向对象中类也是对象，所以用术语“实例(Instance)”来区分它们。

类和抽象数据类型一样，定义一组数据类型，以及可施于这些类型上的操作集（过程和函数）。这样说来，类、类型、抽象数据类型不是一样了吗？都是值集加操作集。诚然，类型和抽象数据类型都是类型。正如“人”和“黑人”都是人一样。但沿用历史的概念，类型指基元类型和简单的结构类型，且操作是自然隐含的。用户对一组类型定义一组操作则是抽象数据类型。即操作‘施加’于类型体系结构之上。强调值集可以接受‘操作’。

类是实例对象的概括是某一类对象的抽象。虽然它也是在类型（传统含义）体系结构上‘施加’操作集，但它强调的是对象行为。‘施加’仅仅是实现行为的手段。

如果类和抽象数据类型都定义了值集和操作集(A, Op), 也都声明了实例(或变量)v.

实例对象的行为是：

v. Op1 向对象v发消息Op1, 该v操作自己的属性A.

ADT的变量的操作是：

Op1(v) Op1施加于变量v的值集A上.

一个是自主的，一个是被动的，正因为它们在概念上抽象是一致，C++把类和类型同等对待。类和类型的区别在子类/子类上更明显。子类是更加特化的类，语义内涵是增加的，子类形是类型的子集，语义内涵是缩少的。

8.4.2 类和实例

C++的类虽不像Smalltalk的类那样清晰。为了比较还是用它。

例8-8 C++的类定义

```
class char_stack{           //以下是该类的私有部分
    int size;
    char * tos, *end;
    char * s;
public:                     //以下是该类公共部分
    char_stack(int sz)
        s = new char [size = sz];
        tos = s;
```

```

        end = tos + (size - 1);

    ~ char_stack( ) delete s;
    int push (char c)
        return (tos <= end ? (* tos++ = c):0);
    char pop ( )
        return (tos > s ? * -- tos: '');
    char top( ); //top的声明
}; //类定义结束

char char_stack::top( ){
    return (tos > s ? * tos : ''); } //top的定义

相应主程序
main( ){
    char c;
    char_stack stk1(100);
    char_stack stk2(10);
    ...
    stk3 = new char_stack (20);
    stk1.push ( '#' );
    stk2.push ( '%' );
    c = stk2.pop( );
}

```

这是和例8-7相似的堆栈只不过是字符元素。它有push, pop, top操作, 因为在public关键字之下, 它们可与外界通信, public以上为私有数据, 也叫属性(Attribute)。操作的体可以直接给出, 如push、pop, 也可以只给声明后给定义, 如top(), 这不过是为了程序清晰, 也就是写和看方便些而已! 在类外定义top()如例中所示。如果再加上关键字inline就和push效果一样是内联(体)的。要注意的是和类名一样的两个函数char_stack(), ~char_stack(), 它们是构造子和析构子。

我们暂且把main看作是一个对象, 声明确立char_stackstk1(100)时, 实际上是向char_stack类对象发消息: 构造一个名为stk1的实例。构造时括号中参数应匹配。char_stack类响应这个消息执行构造子char_stack(int sz), 生成100元素字符数组并令指针和指针限初始化。同理, 生成stk2(10)。stk3是创建一无名对象将它的值赋给对象stk3, 当main执行完每个对象自动调用析构子撤消这些对象。主对象中stk1.push('#'); stk2.push('%'); 分别向两对象发消息, 引起stk1执行自己的push, stk2执行自己的push(其实都是执行类中代码)。本例每个对象中只有私有数据值。

类把其中的成分叫成员(member), 数据成员是属性, 操作成员叫方法(method)。公有成员类外可见, 私有成员只有本类成员可见。

类声明实例对象和类型声明变量一样都要构造一个程序对象, 只不过实例对象一般比变量复杂, 它的构造函数中需要指定更多的信息。有时为了提供多种构造的方式(带不带、带几个参数, 或参数类型差异等等), 去生成实例对象, 此时, 只看消息指定什么参数就执行某个能与其匹配的构造函数。而类型声明变量不是提供一个有初始值的对象就是提供一个对该类型对象的引用, 比较简单, 编译或解释器自动做了, 故早期语言无需用户提供构造子。读者可以体察到构造子名与类名相同的好处。用抽象数据类型创建复杂对象就没有类方便。

8.5 小结

- 利用抽象和封装提供了新的语言机制：模块、包和类是大型软件的需要。
- 模块或包是封装一组数据(变量和类型)和一组操作(过程或函数的抽象)，也是可单独编译的程序单元。
- 程序包的封装就是要分出公有(外界可见)和私有部分，这样给外界以清晰简单的界面，有利于实施高层语义。
- 数据抽象是一种编写大型程序的方法学，它基于以数据和其相关操作作模块分解。构成抽象数据类型，数据抽象即按抽象数据类型开发程序的方法学。传统语言也可以实现数据抽象，但无封装不安全。
- 抽象数据类型是用户定义的，但其效果和内定义类型一样，可以用来定义新类型。
- 构造函数(子)构造一个例化的程序对象。析构造函数撤消这个程序对象。
- 类属是声明的抽象，类属程序单元是参数化程序单元，值、变量、类型、子程序均可以参数化。类属单元可独立编译，但一定要设例后才可使用(执行)。
- 类型参数化该类型则为多态类型，Ada的类属是静态设例的类型多态，动态的类型多态更有利于程序设计。
- 封装提供数据隐藏。对象是一个程序实体，它封装了私有数据和操作，对象是自主的，只能通过界面的触发自己的操作加工自己的数据。类是对象的样板，也是抽象数据类型。类对象和实例对象是抽象到例化的对应。
- 类中的成分叫成员，数据成员也叫属性。操作成员叫方法，方法的调用通过消息，在类可控的界面上有私有、公有、保护成员。

习题

- 8.1 本书模块结构指的是什么？为什么一个函数只表示一个操作？
- 8.2 为什么需要构造函数(子)、能定义值构造子、对象构造子、类型构造子吗？试定义之。
- 答：构造 是用于构造抽象数据类型的值的构造函数。之所以需要，是因为抽象数据类型很复杂，它的值集也很复杂，只有在函数调用的时候，我们才可以知道它在要求什么样的值，所以需要构造出该类型值的构造函数，也叫构造 。
- 可以定义
- 值构造 ：构造本程序需用的值的构造 。
- 对象构造 ：构造本程序需用的变量（对象）的构造 。
- 类型构造 ：构造本程序需用的类型（从抽象类型构造具体类型）的构造 。
- 8.3 大型程序设计需要哪些特征支持？
- 8.4 设计以下抽象类型(任何语言表示均可)
- COMPLEX
- MONEY
- FUZZY (具有值yes, no, unknown和逻辑操作)
- SET
- 如果用无包(模块)机制的语言能不能做？做出来有什么问题？
- 8.5 何谓抽象数据类型？尽可能全面解释。
- 答：类型是 $T = (V, OP)$ ，由若干基元类型组合成复杂类型，也是 $T' = (V', OP')$ 。其中

$$T^* = \text{comp}(T_1, T_2, \dots, T_n)$$

$$V^* = V_1 \cup V_2 \cup \dots \cup V_n$$

$$OP^* = \{OP_1 \cup OP_2 \cup \dots \cup OP_n\} \cup \{\cup \text{复合域操作}\}$$

由于复合是任意的，它们的统称叫抽象数据类型。

用抽象数据类型开发程序的方法为数据抽象，好处是将数据类型的使用和实现分开，提高了程序开发的抽象层次，较易控制大型系统的全局；简化了程序正确性问题；利用抽象数据类型可以准确构造新类型；数据抽象可以实现数据隐藏，私有数据外部不可见，增加软件安全性，可靠性；模块性利于软件开发。

8.6 C语言用文件实现模块程序，能实现类属吗？能实现面向对象的类吗？

8.7 何谓类属？类属子程序和子程序重载有什么不同？类属是动态实现还是静态实现？

答：①类属是声明集的抽象。即是变量、类型、子程序都是参数化的声明。

②类属子程序中程序名、变量、类型均可以参数化，而子程序重载是同名函数的复用，即只有变量（程序参数和返回值）是参数化的；类属子程序的所有实例是一组同构的程序对象，而子程序重载只是同名的不同程序，且程序参数个数可不相等。

③类属的实现类似于宏替换，将所有抽象参数换成实际参数，因此是编译时静态实现的。

8.8 Ada的私有类型和类属类型参数有何不同？

8.9 试比较C、Ada、Modula、Clu、C++实现模块程序之同异。

8.10 何为私有成员、公有成员、保护成员？

8.11 C++是否实现规格说明和体的显示分离？

答：C++中实现了规格说明和体的显示分离，且增加了inline（内联），保持程序正文分离而运行时是不分离的。

8.12 设想一下类属程序是如何实现的，为什么它可以编译（不设例）不能运行？

答：编译本身只编译符号实体并不运行。符号何时有值与编译无关，一般程序运行时符号可立即得到值，如不替换得不到有意义的值。故“不能”运行。替换了的实例目标码是一个可执行的新拷贝，设几次例复制几次，是一种静态实现。

8.13 试述类、类型的同异，给它们各下一个定义。

8.14 试述消息和过程调用之异同。

答：类之间方法的调用通过消息。

函数/过程之间的调用通过过程调用。

相同之处都是触发了一个新的过程或类中操作对象的执行。

不同之处在于消息的触发，调用的双方是平等关系，进程控制比较自由，可同步，可异步；可等待可不等。而过程调用之间则存在父子、嵌套关系，调用期间子程序享有副程序的环境，副程序一直等待子程序完毕。控制是线性的。

8.15 从计算机只能按结构处理计算说明抽象，封装在程序语言中的意义。

第9章 类型系统

第2章中我们曾把定义类型的机制、类型规则和类型检查统称之为类型系统，虽然说到底类型是值集和可施于值集上的操作集，更抽象地说类型就是一特定域(Domain)上的一类值。然而，在第8章中我们看到抽象数据类型和类属类型，它们确实提出了新的类型定义机制，以及达到实施计算的各种规则，因此本章再研究类型系统。

早期程序设计语言的类型系统都很简单，即每个程序对象都必须有一个可供类型检查的、单一的、特定的类型。用这个特定类型可以作出有意义的计算，但随着程序设计语言的发展，不知不觉地走出了单一类型。例如：

- 我们把运算符看作函数，如“+”： $\text{Integer} \times \text{Integer} \rightarrow \text{Integer}$ 。但我们把“+”到处用，浮点、定点甚至在抽象数据类型时把它们作为复数、集合的“加”或“并”运算用。“+”的是什么类型上的操作？

- 第6章我们举过C语言printf()函数的例子，它的变元不仅类型随意，个数也随意。printf()到底输出什么类型？

- 第8章讲的类属过程它的变元类型参数化之后可以和任何(极端情况下)类型结合，类属函数(过程)是什么类型？

为了程序表达方便，设置了很多更加概括的抽象设施，反过来打乱了原来单态(monomorphic)类型的初衷，使类型检查变得复杂。不从理论上分析类型系统，我们无法描述程序设计语言的语义。

本章我们首先介绍实现类型的一般方法，对类型的一般操作。接着讨论类型的另一个(语义)侧面：类属域。从类属函数种类导出四种类型多态性，这是当前语言中已实践过的。由多态性又引入多态类型，它已在函数式语言中出现是一个有发展前途的方向。最后，介绍类型变量的一般推理技术。

9.1 类型表示和实现

类型显式和隐含声明都指明了程序对象的性质，机器怎么知道？翻译器显然要有一种表示法使计算时可作类型检查，能按约定的大小分配存储，能准确按类型规定找出所需元素。

我们暂且设想把有关这些信息存放在某个地方，其信息结构叫**类型对象**，是逻辑结构。物理上各语言实现时，可以集中成一个对象，也可以分散到各个表。大体有三部分信息要存放：一为名字(也有少数无名的)；一为名字的类型，由预定义或基类型名指示的属性；一为信息体，即该类型的特定属性，如图9-1所示。我们用T形图给出类型信息的三部分。

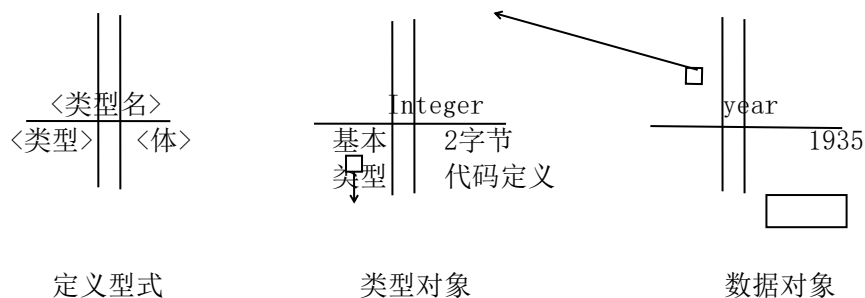


图9-1 翻译器类型实现示意图

名字的类型一般以指针实现，对于最基本的整数、实数等类型它指向由编译约定的一组识别符号，调用的一组函数。这组函数包括计算、取值域、约束(如短整型、长整型)等。这些符号和函数调用用以解释类型对象体中的信息。而体中的信息又是对数据对象的体的约定。

例如，整变量year取值1935，它也有三部分：名字一般出现在符号表；该名字的类型是个指针，它指向类型对象Integer的名(头)，对象体即前述的存储对象。其分配和操纵应符合的规定，由Integer的体中信息给出。也如前所述，类型语言数据对象的类型一般放在符号表变量名(如year)之后，束定建立名字和存储对象之间的联系。且一旦束定不再改动，对于无类型语言类型指针，一般放在存储对象之后，束定的作用更大，动态地束定到各个名字上，但组成这三部分信息是实现类型不可少的。以下介绍各种类型表示，也是类型实现的设计。我们不一一介绍基本类型，仅从实现角度把它们分为有限(长)的、约束的、指针的。再由它们组成复合类型。

9.1.1 有限类型

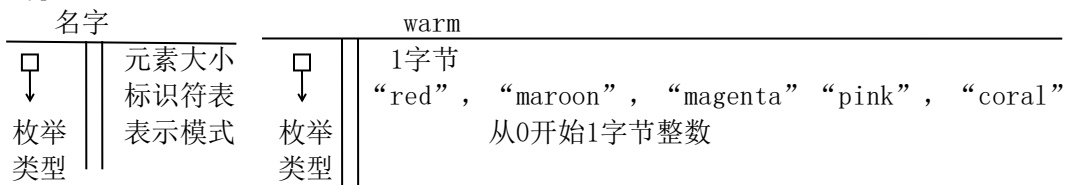
当一个类型和已有类型无关，且有不多的成员时，实践上把它们叫做为枚举类型，即列出该类型成员的标识符表。它所支持的操作是，该类型两对象的相等性测试、求某成员的前驱、后继，有时还提供输入/出操作。

枚举元素的标识符叫类型常量，编译要把它换成内部表示，为了实现查找前驱和后继的方便往往换成从零开始的整数。C语言更是把它们等同于整常量，可遵循整数运算。Pascal和Ada则不然，整常数只作为枚举类型语义载体和整型不兼容。

枚举类型的运算是很容易实现的，主要困难是输入/出，Pascal没有输出，程序员不得不为此编出一段程序，其它语言倾向于增加，系统代码因此就增大多了。这是一种互易。类型对象如下示：

例9-1 枚举类型的类型对象

```
type warm = (red, marroom, magenta, pink, coral);
```



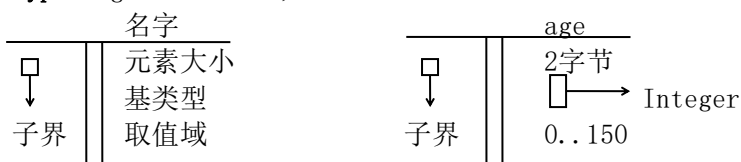
标识符作为字符串保留以便输入/出。

9.1.2 约束类型

子类型是在已有类型上声明约束，派生类型实现方式和子类型同，只是与被派生的父类型不兼容。Pascal和Ada均有子类型。其类型对象如下例：

例9-2 Pascal的约束类型

```
type age = 0 .. 150;
```



类型对象

对于Ada的子类型和派生类型，在类型的类型一格中指明subtype 或derivetype就不会混乱了。取值范围是为了运行时作越界检查。C语言没有子类型概念，主要是C语言设计时类型只用于定义存储配置，而不着重语义清晰，约束对表示的大小和译码不起作用，只对值语义起作用，即所有表示的值对基类型均合法，有了约束按约束拒绝某些值。

9.1.3 指针类型

所有指针都表示机器的地址，指针本身取值域的大小唯一只和寻址长度(机器)有关。

地址值用自然数表示是很自然的。有些语言，如FORTH，指针用整数实现后则遵从整数算术，指针与所指类型绝对无关。程序员要小心地组织这些运算。

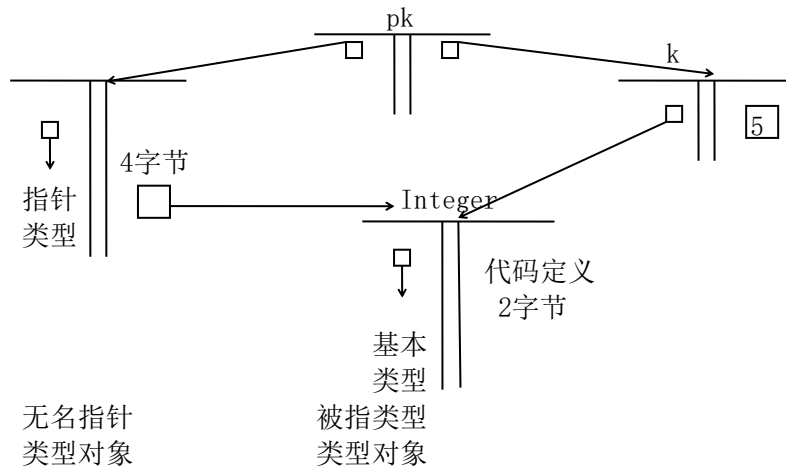
然而多数语言在指针类型对象的体中增加了与所指类型有关的信息。以便编译区分有二义性的运算，例如当指针和所指类型相“+”时能很快地作类型检查；便于快速存取一个所指类型对象或成分；也有利于动态分配所指类型对象。因此，一般语言要求程序员显式指明所指类型。不设显式的指针声明如:TYPE POINTER P;

例9-3 C语言指针的类型对象

C语言有以下声明:

```
int k = 5, * pk = &k;
```

则类型对象示意图如下:



同样声明各语言指针语义很不相同，即类型对象体中信息不同。我们用C和Pascal对比说明。设p是指针，BT是所指基类型。对于C语言:

- [1] 指针算术按BT定义， $p+1$ 按BT对象长度增加1个。
- [2] 递引用和取值的含义按BT，如 $*p+1$ 将所指对象值取出加1。因此，BT只能是简单基本类型。BT是复杂类型时 $*p+1$ 无意义。
- [3] 不作类型检查。
- [4] 指针类型不用于分配BT类型对象。动态分配BT类型对象不靠指针，直接说出BT对象名即可。

对于Pascal:

- [1] 指针本身不定义操作，仅有的操作是递引用。以便所有指针语义一致。
- [2] 任何BT类型的对象b都可以通过 $b := p \uparrow$ 赋值，它按复制n个字节(n是BT大小)实现。
- [3] 严格的类型检查。
- [4] 动态分配函数NEW的变元必须是指针p，返回值是对BT类型对象的引用，存入p。

9.1.4 复合类型

复合类型由简单基本类型复合而成，也叫结构类型。按复合原则不同它们可分为：

- 取决于位序(数组、串)，不取决(集合)
- 元素同构(数组、串、集合)，异构(记录)
- 定长(数组、记录)，变长(串、变体记录、集合、灵活数组)

本节介绍典型的复合类型的类型对象。

(1) 数组类型

数组类型的元素是以其所据位序访问，从第一位置到最后一位置称为数组界。一般以离散类型描述元素所在位置，故也叫索引类型(index type)，实现时索引也都映射为整型的值。

数组以下标(subscript)表达式索引，数组名所在的地址加下标表达式(以方、圆括号表示)的值。下标表达式的值应在数组界之内。实现术语是基地址加位移，称之为下标索引值。为了快速计算基地址，一般是把它放入地址寄存器。索引值放入索引寄存器，由索引存取指令快速动态计算。

数组对科技计算是极重要的。早期语言多维数组都换算成一维数组，为此提供了快速专用的语法和语义以支持高阶数组。但后来加强了用户定义的类型，数组元素的基类型和索引类型都扩大了类型范围。高阶数组则以数组的数组来解释，这样，无论几维只要再来一次“元素是数组”就可以了，近代语言两种表达都可以。

例9-4 Pascal多维数组两种表示法

[1]: 适于数学的表示法:

```
VAR Sugared: ARRAY [1..3, 1..10, 0..3] OF Real;
d := sugared [ 1, 5, 2];
```

[2]: 基于实现表示法:

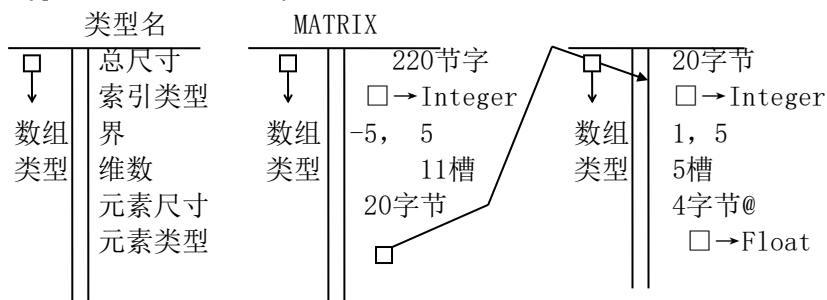
```
VAR Plain: ARRAY [ 1..3] OF ARRAY [1..10]OF ARRAY [0..3] OF Real;
d := Plain [1] [5] [2];
```

内部实现按[2]。计算一维地址简单。C的数组按流的思想，一维均为零基地址。没有非零基地址概念。也没有类似早期FORTRAN的、如上例[1]中的表示法(它有较多换算)。因为C追求快速。

数组类型的类型对象如下例。

例9-5 Ada数组的类型对象

```
type MATRIX is array (-5..5, 1..5) of Float;
```



显然其中界和维数有冗余信息，若用零基地址就取消了上下界这一项，图中@号为最后元素类型(不再有子数组)。

由于C的数组作为参数时总是引用传递，而且也不作类型检查，子程序内只要分配一个指向基类的指针，只要知道基类型即可，维数、界、总长度均可不要。而Pascal, Ada要作类型检查因而信息多，运行时开销大，换来可靠性。

(2) 串类型

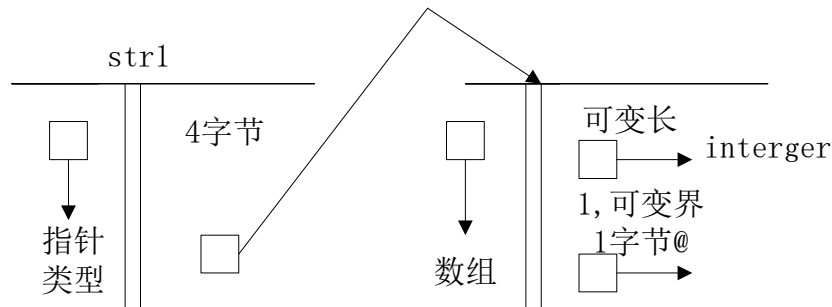
许多语言把串看作是下界为整数1的一维字符数组，其长度可变，是一种特殊的数组(Ada、Pascal)，但未能反映串的本质。

一般说来：串对象长度编译时不知长度不同的串混用(计算并、截取子串)串对象可以包含其它串对象函数以串作变元或返回值，则任何长度串都要引用有特定语义，如不同长度串也能比较字典序。

Ada和Pascal的串长不是可变的，只是串值可变长，用定长数组表示变长串工作不可靠。解决的办法有两个一为记数串(counter string)，即在构造或输入一个串时，将其长度以无符号数放于0下标处，这样对串元素施行迭代算法处理极为方便。另一个是终止符(terminated string)在串的终止处置置空(ASCII的0)或0，这样在串做迭代处理时见零自动终止，且易于从错误中恢复。C语言就采用这种方案，以下是它的类型对象图。

例9-6 C语言的串

```
char * str1 = "This is a string Interla.";
```



现在再来讨论可变长，可变界数组始终是数组中较为复杂的问题，对于编译型语言，如Pascal，Ada见到串变量编译就给它一定长空间(比较大)分配到运行时堆栈框架中，这一般是比较浪费的。BASIC和SNOBOL这类解释型语言，在专设的动态存储区中分配串变量，把串变量作为指向该区的某个(头)单元的指针。串处理函数也都用指针作变元并返回指针结果值。由系统动态管理串存储。

C实现的串存储，一半如同Pascal取定长串(按声明初值串)，一半则如BASIC把串换成字符指针，但不包括串处理函数的存储语义，即处理串的函数不分配到存储。例如，C的串函数strcat就要求有一个变元是指向预分配的存储对象的指针，这样，就有足够长度放下返回值。它之所以不考虑支持自动存储动态管理是为了简单、高效，动态管理要用到紧凑算法作无用单元收集，效率太低。

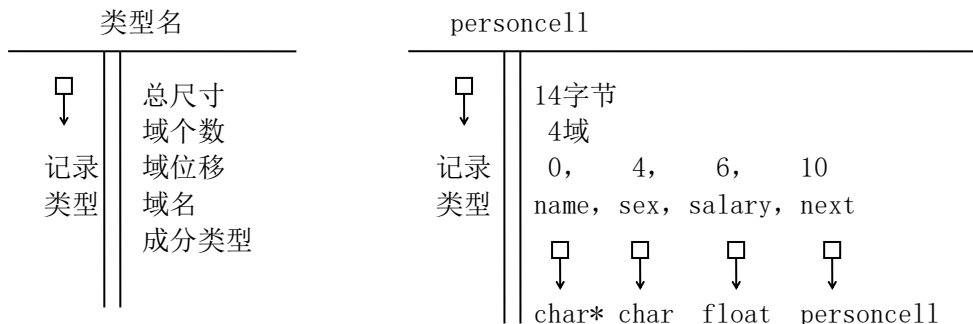
(3) 记录类型

记录(或结构)是异构成分集合，其类型对象体中的信息支持记录对象整体操纵和每个成分的单独立操纵(分配、存取、更新)，理论上，对整个对象分配各成分所需域长度即可，但实际上往往要大一些，总要带一点垃圾，原因是要按字或长字归整成线(word alignment)，而每个成分都从双字节地址开始，长字甚至要二倍双字节，故总长总要大于部分和。

每个域分域的起始处叫位移(offset)由编译器处理声明时算出。记录类型的类型对象所含信息如下例：

例9-7 结构类型的类型对象

```
struct personcell { char * name; char sex;
                    float salary; personcell * next ; }
```



尽管记录型与数组型有许多相似之处，对其成员访问都要通过选择子(selector)，即指明成

分的点表达式或下标表达式。但最大的不同一个是处理单一类型域，一个是多类型域。所以选择子的表示法一为括号，一为带点的名字。有语言尝试过全用下标表达式，因太复杂未推广。

(4) 联合类型

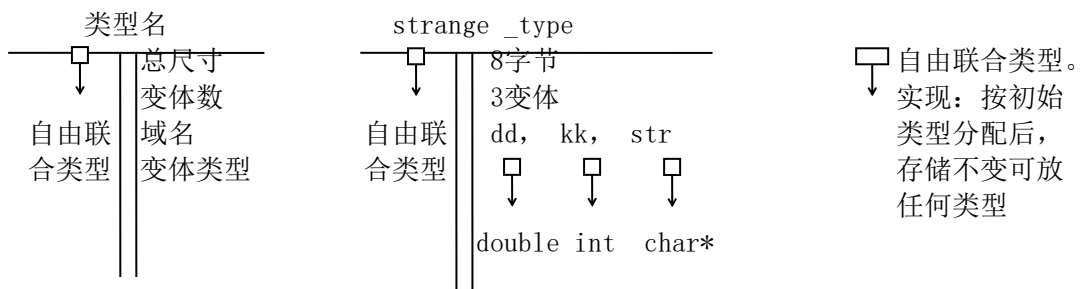
同一存储对象其语义可以不同，则称此种类型为联合类型。有自由联合和约束联合，它们的语义也不相同。

- 自由联合类型是语义不健全的类型，同一存储对象的内容可作多种解释。取决于该类型变量出现在使用场合的上下文。一般它不是复合类型。

例9-8 自由联合类型的类型对象

C语言用户可定义自由联合：

```
typedef union {double dd; int kk; char* str} strange type;
```



如有声明和语句：

```
union strange_type s;
*s.str = 'a';           //先赋一字符
printf ("k:% 15.10d", s.kk); //却能打出正确整数值
z = s.dd;               //也是正确赋值
```

与前述FORTRAN的EQUALENCE的多重束定类似，比它更自由。C语言的各成分域寻址总从零开始，故无域位移一项。

- 约束联合类型，理论上语义健全的，因为它的语义变体由情况指明符(case specifier)控制。Pascal和Ada的变体判别式记录就是这一类。

例9-9 Ada的判别式记录

```
type GENDER is (Male, Female, Unknown);
type PERSON (SEX:GENDER := Unknown) is
  record
```

```
    BORN:DATE;
```

```
    case SEX is
```

```
        when Male => BEARDED:Boolean;
```

```
        when Female => CHILDREN:Integer;
```

```
        when Unknown => NULL;
```

```
    end case;
```

```
  end record;
```

```
SAM:PERSON;           --声明该类型变量
```

```
  .
```

```
  .
```

```
  .
```

```
SAM := (Male, (1970, Jan, 3), FALSE);
```

```
--正确
```

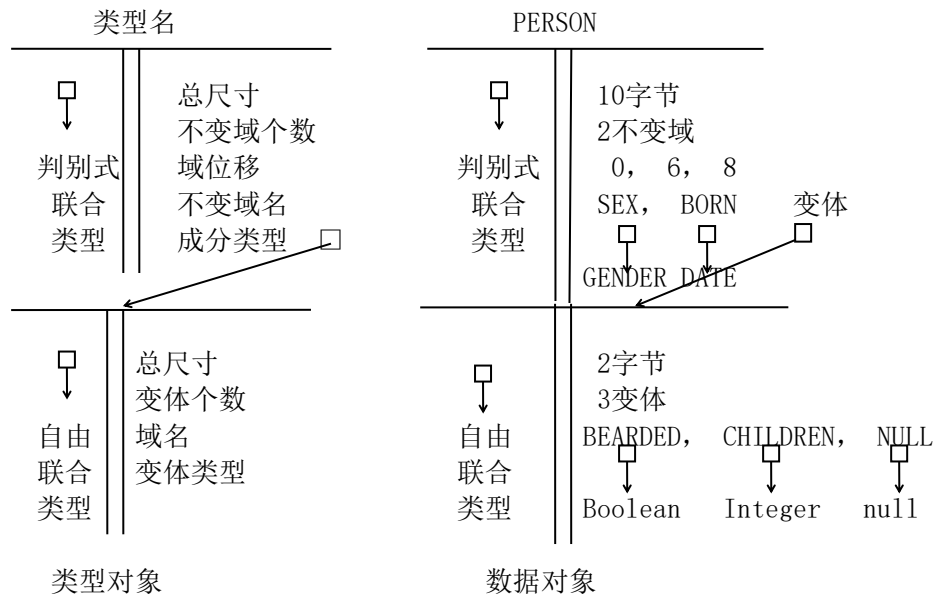
```
SAM.sex := Female;
```

```
--禁止
```

```
SAM.CHILDREN := 2;
```

```
--无意义
```

与本例对应的类型对象是：

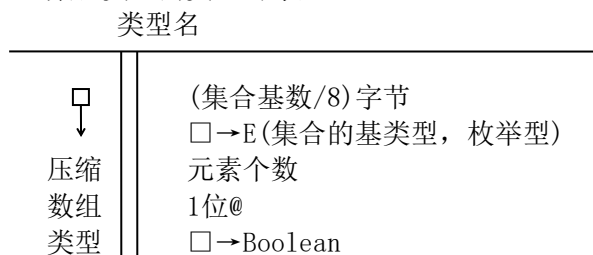


(5) 集合类型

唯有Pascal语言提供了集合类型，Pascal的集合用Boolean数组实现。数组元素是压缩的Boolean类型，也可以说是枚举类型和子界类型的常量，集合是枚举的特例。

集合成员是唯一无序的，集合成员个数在计算中不确定，它在用布尔数组实现时，每个元素对应一个二进制位(BIT)，借用枚举、整数计算存储对象的大小，利用位模式计算实现集合。所以要回避数组的有序，并要准确确定长度，C语言由于提供了位模式计算所以不提供集合类型，Ada有较强的枚举类型，且一维布尔数组均用位模式表示和计算(只不过对用户是隐式的)又提供了聚集赋值表达，用户很方便地设计集合类型，所以也不提供集合类型。

集合类型的类型对象是：



再看一个Pascal集合类型的实例。

例9-10 有12种颜色组成色彩集合。已知画水仙、狗尾花、树三种画的颜色，问它们共用什么颜色？调色板有多少颜色就够了。

```

TYPE color = (red, pink, orange, yellow, green,           //枚举
               blue, violet, magenta, brown, black,
               gray, white);
combo = SET OF color;                                     //集合
VAR tree, daffodil, iris:combo;
    palette, common:combo;
daffodil := [white, yellow, orange, green];               //实测的颜色
iris := [white, yellow, blue, violet, green];
tree := [white, gray, brown, black, green];
palette := iris + tree + daffodil;                        //'+'是集合并
common := iris * tree * daffodil;                        //'*'是集合交
    .
    .
    .
  
```

内部表示：

```

daffodil: 001110000001           //该模式指示的集合
iris:      000111100001
tree:      000010001111
palette:   001111101111
common:    000010000001

```

9.2 复合对象上的操作

类型是变量的样板。我们有了类型对象提供的信息，就可以‘照方抓药’构造变量。这个构造工作，在早期静态编译型语言中，是由编译做的：分配存储，填以初始值。程序中声明了多少变量就构造了多少个。当然，初始化也可以留待程序负责，但该存储对象具有该类型一切属性，这是定死了的。堆变量(指针)一般可动态生成。程序员可在程序中用new算子构造——这就是最初为用户开放的构造(函数)子(constructor)。以后用户可以定义复杂的数据类型，根据不同的使用要求作不同的初始化，例如，构造出对象并填以初值；构造出对象部分填入初值；为使用方便要求构造对象后返回该对象的引用……因此，近代语言向用户提供了可自行设计构造子(函数)这一语言特征。以便用户能更好地控制程序对象的构造。

类型对象一般也是复合对象。构造子将单个的程序对象按类型对象样板构造成复合对象。它的逆操作是选择子(selector)，即从复合对象中选取某个成分，查询某种属性。选择子在早期语言的构造类型早已有之，如数组的下标选择，记录的点‘·’表示法的成分选择。但它们的操作是内定义的。

本节我们讨论复杂对象上的操作：构造子、选择子、引用、递引用，以及它们的关系。

9.2.1 构造子

构造子是构造程序对象的操作。它必然返回一程序对象，可以是对象本身，也可以是对该对象的引用。不同语言采用不同构造子。

(1) 值构造子

一个程序对象，分配了存储并赋以初值才真正得到定义。多数程序设计语言以都以直接可识的字面量定义基本类型的程序对象。这往往是由编译器做的，直至简单的值表达式计算后填入。然而，对于取决于束定环境的值表达式(如嵌套块)，则要调用构造函数进行。值构造函数的输入是该类型的表达式，输出是程序对象(在运行堆栈中)，这些构造函数是隐式的，程序员只在值表达式上表达后隐式调用它们。

C语言的值表达式在语法中叫初始化子(符initializer)，只允许出现在声明部分。这似乎没什么道理。C++作了扩充可放在语句出现的任何地方。

对于复合类型的复合对象，理应能提供一组字面量来初始化。但Pascal却不许可，它只能在程序开始执行时，一个成分一个成分赋值。Ada, C, C++扩充了聚集赋初值，也就是隐式调用构造子完成。

APL是显式使用值构造子最早的语言，它以‘ι’(希腊文‘约塔’，构造从1递增数组)，‘,’(并)，‘ρ’(组形)三个符号作值构造子。

例9-11 APL的值构造子

以下APL语句创建数组A,B,矩阵M1,M2。

```

N←10
A←ιN           //A = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
B←11, 2, N, 17 //B = [11, 2, 10, 17]
M1←(3 5) ρ 11 12 13 14 15 21 22 23 24 25 31 32 33 34 35

```

M2 ← (3 4) p 0

构造的M1, M2如下:

| | | | |
|-----|----------------|-----|---------|
| M1: | 11 12 13 14 15 | M2: | 0 0 0 0 |
| | 21 22 23 24 25 | | 0 0 0 0 |
| | 31 32 33 34 35 | | 0 0 0 0 |

这三个构造子是内定义的。它可以在运行中不加限制地构造任何值。

Ada是强类型语言值构造子形式如下:

例9-12 Ada的值构造函数

若声明以下记录类型:

```
type IMAGE is new Float;
type COMPLEX_PAIR is
  record
    RP: Float;
    IP: IMAGE;
  end record;
```

声明变量并以值构造函数为其分配初值:

```
C:COMPLEX_PAIR := COMPLEX_PAIR (0.0, IMAGE(5,6));
```

‘:=’号右边是值构造函数, 其实它是类型强制表达式。即括号内的聚集值是记录类型COMPLEX_PAIR的对象的值。构造子名和类型同名。

(2) 引用构造子

引用构造子是构造子的返回值是对创建的对象引用。引用构造子隐含着动态分配和使用指针。多数通用语言并不支持它。然而, 在LISP中却是非常重要的基本操作。LISP的cons函数就是引用构造子。每次使用时分配一个新槽并初始化它构成一新表。将第一变元和其余变元作为两个连续的部分拷贝进去, 并返回对新表的引用:

例9-13 LISP的引用构造子

cons是LISP的构造函数, 它将两个表合成一个表。

```
(cons '(a b)' (c d)) ≡ ((a b) c d)
```

而不是 ≡ (a b c d)

9.2.2 选择子

选择子为复合对象选择指明成分, 或选择符合对象的某些部分。通用语言除数组下标是所有数组类型的选择子而外, 每一个成分(对记录或结构)名配上‘.’和复合对象名即为选择子名。如例9-12的COMPLEX_PAIR.RP, 即为一选择子。当选择子出现在赋值号右边时它是递引用。

例9-14 Pascal的选择子

若Pascal中有声明:

```
TYPE Stack = RECORD
  top: Integer;
  store: ARRAY(1..100) OF stackItems
END;
```

它有下列[...]及“.”两种选择子。可写函数:

```
FUNCTION Pop(VAR s: Stack): stackItems;
BEGIN
  Pop := s.store[s.top]; [1]
```

```
s.top := s.top -1      [2]
END;
```

[1]行右侧是复合选择子，它以递引用的值赋给Pop。[2]左边s.top是选择子引用，右边s.top是递引用。

C++中选择子和Pascal类似，有下标[……]，‘.’，对于指针指向的无名对象用‘→’，如：

```
p→print();    //引用p所指向对象中的print()成员函数。
b = p→age;    // 递引用p所指对象中数据成员age的值。
```

APL选择子更多，‘↑’（取部分值）‘↓’（消去部分值），‘/’（筛选），‘[….]’（下标）

例9-15 APL选择子矩阵

设两数组M1, M2, (如例9-11)构成了3×5矩阵和3×4的零矩阵。变量E, M2重束定为：

```
E←001010/ "RANDOM"
M2 [1 4; 1 2 3] ← M1[ 1 3; 2 4 5]
```

选择子‘/’按第一向量非0元素筛选字符向量“RANDOM”，得出“NO”束定到E上。选择子[1 3; 2 4 5]选出M1的第1、3行的第2、4、5列元素，将它束定到M2的[1 4; 1 2 3]上，即选择子选定的第1、4行和第1、2、3列的元素上。得：

```
12 14 15      12 14 15 0
32 34 35 →    0  0  0  0
               0  0  0  0
               32 34 35 0
```

同样，选择子的体也是隐含的，用户不能定义选择子的操作。

9.2.3 用户定义的构造子和析构子

无论是数组、记录构造子、选择子，还是APL的多种构造子和选择子，其函数体均为内定义的。面向对象中对复杂对象的操作，可由用户定义，即可定义构造子、析构子的函数体。我们以C++为例。

C++中的类在语法上等同于类型，即抽象数据类型。类定义之后，在主控对象main中，可以按类型定义变量一样定义实例对象，如：

```
class student {...}
wang: student(初始化符表);
```

这个工作可由编译做，它仍然要有一个隐含的构造(子)函数将初始化符表中的参数一一存入对应的属性表之中。但复合类型定义不象基本类型那样简单，有时十分复杂。这个隐含的构造子难以胜任任何复杂情况，这是其一。更重要的面向对象它要动态(在运行中)生成实例对象，这时就要求在类定义时，也定义本类如何生成实例对象的构造子。向这个类发构造子消息，并给出初始参数它就生成一个实例对象。这也保持了语言的正规性：对象是封闭的，对对象的操作只能是发消息。构造子和其它操作一样，它也是界面上等待响应消息的方法。构造实例对象由类对象自己完成。

由于用户自定义构造函数，因此不再强调值构造子或引用构造子了。愿意定义为什么样的构造子都行。而且，用户在生成实例时往往需要不同初态的实例对象。构造函数就写出多个，即它是重载(overloading)多态的。按指明参数匹配时分辨多态。此外，仅当没有自定义构造函数时，编译才调用预定义的构造函数。请看下例：

例9-16 C++的构造函数和析构函数

字符堆栈类定义如下：

```
class stack {
private:
    char * u;           //字符数组未定长度
    char * p;           // 栈指针
    int size;           // 栈尺寸
public:
    stack ( ) {          //缺省构造子
        size = 100      // 总给100字符长
        u = new char [100];
        p = u;
    };
    stack (int sz) {     // 要求指定sz的构造子
        u = new char [size = sz];
        p = u;
    };
    void push (char ch) {...}
    char pop ( ) { ...};
    ~ stack ( ) {        //构造子
        delete [ ] p;
    }
};
```

构造函数和类同名(参见例9-12)，且可重载(本例一个无参一个有参)。在主对象中：

```
void main ( ) {
    stack s1;           //和无参构造子匹配，堆栈s1长度 得v[100]
    stack s2(25);       //和带参构造子匹配，堆栈s2长度得v[25]
    :
    ~stack ( );
}
```

当执行main中声明时，自动隐式调用并匹配构造子。若编译时能作重载分辨则在编译时做完。若参数类型个数只能在动态时确定，则动态束定(匹配)。

当执行到~stack()时，撤销实例对象。析构函数体内指明撤销哪一个(或所有)对象。对于运行堆栈框架中生成的实例对象，它们随所在块执行完而自动消失。在堆空间分配的实例对象则必须显式调用析构子才能撤销。析构子是构造子在分配存储上的逆操作。

9.2.4 复合对象操作之间的关系

图9-2示出复合对象、简单对象、选择、引用、递引用之间的关系。并非每种语言全都具备这些操作。

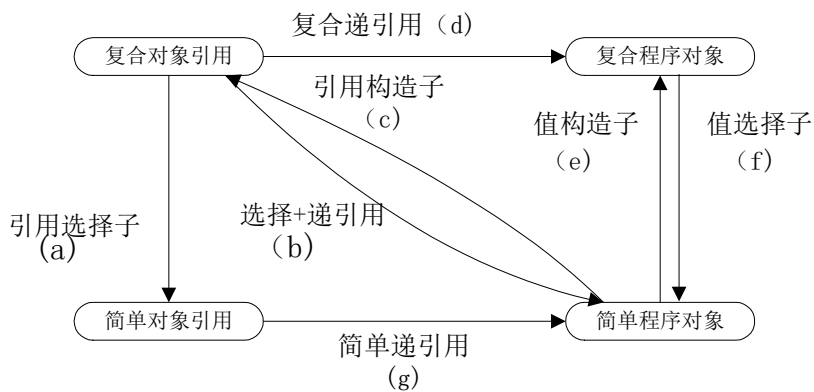


图9-2 复合对象上各操作间的关系

解释如下：

- (a) 引用选择子，如Ada，Pascal对数组、记录成分的引用，并出现在赋值号左边的选择子或VAR参数。
- (b) 选择+递引用，上述选择子出现在赋值号右边，则自动递引用简单程序对象。
- (c) 引用构造子，通过引用构造子分配存储，以简单对象构造复合对象，赋初值后返回对复合对象的引用。如前述LISP的cons函数。
- (d) 复合递引用，在赋值号右边，以数组、记录名递引用整个复合对象；复合对象的相等比较，拷贝，以及作值参传递时。
- (e) 值构造子，以简单程序对象(的值)构造复合程序对象，见例9-11。Ada数组的聚集表达式的初始化。Pascal无此机制。
- (f) 值选择子，通过值选择子(见例9-15)从复合对象中选取成分。
- (g) 简单递引用，赋值号右边的表达式或变量名。

9.2.5 对类型的操作

系统程序员在设计库函数，C++应用开发者实现类库，都不能只用简单的语言提供的声明。因为往往需要该类型对象更多的信息，如整个结构的大小，束定时间等。出了问题可以有目的地测试。总之对内部情况不能完全透明。至少在重要的地方可以在程序语言的层面上操纵。

(1) 类型选择子

类型对象是有关类型信息的源泉，我们把它作为一个对象，并作为变元传递到比较容易实现的类型谓词上。谓词描述类型有关的属性。于是，同样有类型对象上的选择子。选择子通过指示类型的类型指针可以找到所需的属性，每个属性在语言层面上都对应为一个选择子函数。

C语言的sizeof算子可用于任何类型上，它访问类型对象并返回类型实例的大小。在C语言移植中sizeof是很重要的。在动态分配中也极频繁地使用。特别是要作动态分配时。例如，TA是正在设计的复杂类型，我们暂时还不知道它的具体结构，但可以写出：

```

TA* pt1, *pt2;
pt1 = (TA*) (malloc (sizeof TA));
pt2 = (TA*) (calloc (n, sizeof TA));

```

这样，就可以开始设计TA对象的应用，pt1指向TA的对象，pt2指向n元TA型数组。

Ada支持类属程序包，它在语言中就提供类型选择子，选择类型的属性。每当属性表达式出现在程序中，可以得到该属性的值。以下部分列出Ada属性选择子：

| 属性 | 意思 |
|-----------------|-------------|
| 任何类型都有的: | |
| T' STORAGE_SIZE | 所需存储单元(字节)数 |
| T' SIZE | 一个对象所需的字节数 |
| 整型、子(派生)类型、枚举型: | |
| T' FIRST | 该类型最小值 |
| T' LAST | 该类型最大值 |
| 浮点型: | |
| T' DIGITS | 小数点后十进制位数 |
| T' EMAX | 最大幂值 |
| T' EPSILON | 离散浮点到下一个值之差 |
| T' LARGE | 最大正值 |
| 数组类型: | |
| T' FIRST(N) | 第N个索引位置的下界 |
| T' LAST(N) | 第N个索引位置的上界 |

属性选择子在Pascal中也有，如maxint，只不过没有Ada完备罢了。

(2) 类型构造子

程序员在声明类型时用以描述类型结构的关键字，如Pascal的“record”，“array”，和特殊符号，如Pascal的“↑”，都是类型构造子。每当类型构造子出现即创建了一个新类型。类型名字则束定于新类型的域。9.1节类型对象的T型图已作了清晰的说明。

类型构造子除了构造复杂类型之外，也用于变量和参数声明，如：

```
ar: array [1..10] of Real;
```

类型构造子创建了一个无名的10个实元素的数组类型。ar是该类型变量。

然而，在C中“*”，“[]”不是类型构造子。和记录对应的struct和union是类型构造子。C++把它们都算作类型构造子，和基本类型一起，导出新类型。

```
type *           //指针，指向type 类型。
type &          //引用type类型。
type []         //数组，具有type型元素。
type ( )        // 函数，返回type类型。
```

Ada的类型构造子最丰富：

array(数组)、record(记录)、access(访问)、range(整型)、digits...range(浮点)、(-,-, -, -)(枚举)、new(派生)、<>(类属抽象)、dleta...range(定点)、private, limited private(私有、受限私有)。

9.3 类型的语义

类型规定了程序对象的物理性质和语义性质。上两节我们讨论了类型的物理性质：表示和实现。即它规定了对对象的存储及操作约束，以及对类型这个复杂对象上的操作。本节我们研究类型的语义侧面。

类型指明一组对象，这组对象享有共同的物理性质(尺寸、结构)，容许同样的操作(定义在其上的函数)，这组对象构成一个数据域，域(domain)的性质即类型的语义。

声明一个类型，即指明了一个域。这个概念在程序语言初期并不明确。例如，汇编语言就没有类型声明。但程序的实现者却记住它有三个隐含的域：地址、整数和索引。因为它们的行为(操作)不同。goto，存数，取数操作施加于地址对象上；算术运算和比较施加于整数对象上；

“装入索引寄存器”，“加到基址上”这两个操作只可施加于索引对象上。

早期的类型语言，FORTRAN，Algol，引入了基本类型域，它们是固定(预定义)的域。程序中只作变量声明。原则上各域彼此不相关，表达式中遇到混合运算，则按自动升级强制(见后文)处理。这当然要包含类型检查。但是，除了内定义函数，用户定义的过程和函数，其参数类型一般的不作类型检查。

早期的无类型语言，如LISP，Snobol，APL，也有自己的基本域。变量直接动态地束定到域上。它不作变量声明，但在每个基本操作之前均作域检查。如前所述，域标记在值后。当函数要求的变元域不匹配则发出错误信息。LISP提供的域检查谓词，程序员也可以用。

例9-17 LISP的基本域及检查谓词

基本域：(整)数、符号(对象名)原子(非表实体)、表

基本域谓词：numberp, symbolp, atom, listp, null(用于空表)

如：(numberp s)若s是数返回T，否则F。

9.3.1 域的一致性

70年代开放了用户定义的类型。用户可以用类型名，类型构造子来声明一个类型。用户声明的类型一般由基本类型复合。情况就比早期的一个预定义类型就是一个独立的域要复杂些，要作类型兼容性检查自然是检查两类型是否同一域。那么首先要问类型声明和域是什么关系？怎样才算相同的域？

(1) 内部域和外部域

我们把程序语言在程序世界向程序员提供的域称为**外部域**。即类型声明和类型构造子每次出现都指明了一个域。而在实现世界由语言翻译器或实现者使用的域称**内部域**。如，程序语言中的枚举型，在实现世界中是整数。任何基本类型，实现世界中都是字位串。

同是字位串定义了不同操作约束即可映射为不同的外部域。对表示的结构解释，连同其上的操作约束就构成一个域。所以，域相等不仅表示一致，容许的操作也要一致。反过来说：如果某域上定义了一个函数，不能用于另一域的表示上，则两域彼此**独立**。也就是不兼容。如果某域A上定义了一个函数，可用于B域表示的对象上，但反过来不行(B域上的函数不能用于A域)，则称两域彼此**半独立**。显然，两者定义的函数彼此均可用，则称**可合并域**。一般说来。内部域和外部域是独立的或半独立的。整数域和字符域是两个独立的外部域，因为它们的运算函数不同。多数语言整数、字符和位串是内部独立的不同域。

(2) 内部合并域

C语言和FORTH采用的是内部合并域，这给哈斯表索引带来高效。C语言中有真值域，但无真值类型，外部的真值域用约定内部整数表示。1或非零整数为TRUE，0为FALSE。因此，真值的运算用整数表示。这带来极大的方便。凡外域解释不通之处按内部域，反之亦可。

例9-18 内部合并域带来的方便

在C语言中真值和整数的互易。

把真值当整数：

```
difference_count = difference_count+(item1 != item2);
```

若两项item1, item2不相等时，括号中表达式为1，差别记数加1。相等为0，则无差别就不计数了。

把整数当真值：

```
while (item) { ...};
```

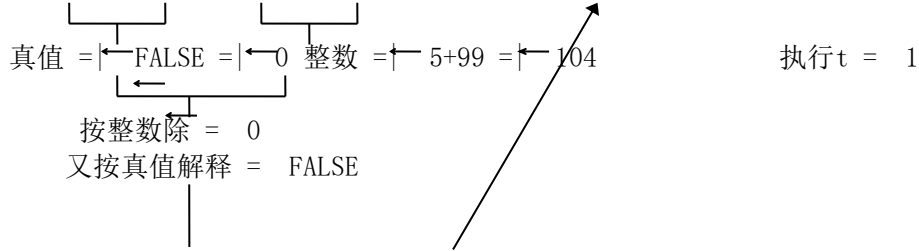
若输入整数item 不为0, 循环继续做。

这样的程序极为简洁, 系统程序员欢迎, 自然也会带来问题。

例9-19 无意义的求值

以下是C的合法语句, 但除了难懂也没什么意义:

```
if (( 'a' > 'b' ) / ( 5 + 'c' )) x = 1; else t = 1;
```



‘c’ 的ASCII编码是99, ‘a’, ‘b’ 是97, 98。

一个表示只要能按整数或真值之一解释得通, 它都是合法的计算。这在Pascal和Ada中是绝对不可以的。

一般说来, 有显式可区分内部和外部域对程序的检错和编译带来方便。但程序员要到处写显式类型转换。所以程序员宁预冒风险也喜欢简洁的C。内部合并域不能取消。

(3) 映射域

用内部的, 原有的域定义外部的, 新的域叫映射域, 这是用户定义类型增强表达能力的基本手段。一般说来, 它们是两个域, 从一个域的每一元素到另一域的元素存在着映射关系。其值的物理形式一般是不变的, 只是解释(容许的操作)不同, 因此, 一个域的对象转换为另一域的对象只在对象上换了域标识。反过来也一样。

换了标识就是新域, 新域和老域一致(兼容)吗? 特别是子集域。说它兼容是有同样的表示, 容许同样操作, 似是有理, 早期语言就是这样处理的。说它不兼容是, 定义新域就是为程序员处理数据方便, 如人的年龄不超过150岁, 年龄域是整数域的子集, 它们是两个域。混合运算时必须显式作转换。编译极易查出2050岁这种荒谬错误。

映射出的子集域可以看作是可合并域, 混合运算不作任何转换。也可以看作新域, 要作类型变换。这就要看语言采用什么机制了。

Ada的子类型定义subtype和派生类型定义type...new...为程序员提供了方便。前者是可合并域, 后者是新域。

9.3.2 构造域的一致性

类型声明实则是利用类型构造子指明一新域。按强类型观点每构造一新类型域它都是独立的。将构造出的域束定于类型名, 这个名字就代表了这个构造域。于是, 又出现了类似变量“名实分离”的情况。

相同的构造不同的名字, 是别名还是新域? 这就是前面说的类型“按名等价”和“结构等价”之分。按名等价安全, 但频频类型转换极为麻烦, 特别要小心处理别名(没有创建新域)的情况。按结构等价灵活方便, 但不安全。

相同的名字不同的构造, 这在静态编译型语言是不会出现的。谁也不会重载类型名。但它恰巧是类型技术发展的方向。即多态类型。一个类属类型名可以静态设例为多个类型。一个类型变量名可在运行中动态束定为多个类型。下一节我们专门讨论类属域。

(1) 无名类型域

名、实分离已经使类型系统复杂化, 然而, 程序语言中允许用类型构造子直接构造出无名类型。这就是我们引入域解释类型语义的又一原因。先看下例:

例9-20 类型构造子创建新域

```

TYPE Vitalstates = ARRAY[1..3] OF Real;
VAR p, q: ARRAY[1..3] OF Real;
    r, s: ARRAY[1..3] OF Real;
    v: Vitalstates;
FUNCTION Vol (t: ARRAY [1..3] OF Real) : Real;
BEGIN Vol := t[1]+t[2]+t[3] END;

```

这个程序按结构等价p, q, r, s, v, t均属于一个域，是一个正确程序。按名等价的语言，它定义了四个域：Vitalstates, p和q, r和s, t。因为类型构造子用了四次，特别是函数参数中用类型构造子声明，则不可能有任何实参类型与之匹配。同一程序改作如下就对了：

```

TYPE Vitalstates = ARRAY [1..3] OF Real;
BoxDimensions = Vitalstates;
VAR p, q: Vitalstates;
    r, s: BoxDimensions;
    v: vitalstates;
FUNCTION vol(t: BoxDimensions): Real;
BEGIN vol := t[1]+t[2]+t[3] END;

```

后改的程序为同一类型提供了很多别名。这在Pascal中正确，在Ada中：

```

type BOXDIMENSIONS is VITALSTATES;

```

是不容许的。可以有：

```

TYPE OTHER_INT IS new INTEGER;

```

但它定义了一个和INTEGER 一样的新类型。

(2) 类型映射出非独立域

用老类型定义新类型不一定创建新域。C语言的typedef声明就是例子。

例9-21 C语言的类型别名

```

typedef struct pt { float c1, c2; } polar_point;
typedef pt xy_point, twisted_point;
polar_point pp;
xy_point xp;
twisted_point tp;
struct pt sp;

```

pt是标签，实则域名。Polar_point, xy_point, twisted_point是pt的别名，并未创建新域。故pp, xp, tp, sp是可兼容的变量。C语言的typedef 不是类型构造子。

Pascal的类型定义，如果不用类型构造子也会产生别名。

例9-22 Pascal的合并域

```

TYPE LengthInFeet = Real;
LengthInMeters = Real;
VAR r: Real;
    f: LengthInFeet;
    m: LengthInMeters;
FUNCTION FeetToMeters (f:LengthInFeet): LengthInMeters;
BEGIN FeetToMeters := f *12.* 0.00254 END;

```

r, f, m 完全兼容，虽然它们有不同的外部类型。其中的 ‘*’ 可作混合运算。以下表达式均合法：

```
m := 3.0; r := m/0.5;
f := FeetToMeters(m);
r := FeetToMeters(f);
```

定义类型名，并未为程序员带来区别。这是Pascal成为伪强类型的事例之一。

(3) 类型映射形成可区分的域

Ada 给类型映射以较多的选择：

```
Subtype Length is Float;           [1]
type Length is new Float;          [2]
```

[1]行是子类型映射，Length就是Float的别名。[2]行是映射的新域，即Length和Float是不相兼容的域。既然不相容那么就要为新域重新定义基本操作。否则在它上面什么也不能定义。这似乎代价太大了，我们原本希望它和Float有一样的基本操作，只是程序员用起来方便一些。要解决这个问题则在编译时增加改型操作：REP——将新域对象临时换成老域标记，对象的物理表示不变；MAKE—REP的逆操作，将新域标记再换回来。这样新域用的全是老域的基本操作。当然，还可以额外定义自己的新操作。Ada，C++在实现这种换型 (cast) 操作时，只简单冠以类型名。

例9-23 Ada的换型

```
TYPE TONNAGE is new Float;
T1, T2 : TONNAGE;
FF : Float;
FF := 3.2;
T2 := TONNAGE(2.0);
T1 := TONNAGE(Float(T2) * FF);      [1]
T1 := T2/T1;                         [2]
```

上例说明，Ada派生类型有以下规则。设老类型R派生出新类型T：

- 新类型T的字面量可以沿用R的字面量，上例[1]行的换型不必要。
- R类型上的预定义函数可用于T类型上。上例TONNAGE中未定义 ‘/’ 运算，[2]行的 ‘/’ 是Float中预定义的，
- T类型上定义的函数不能用于R类型上。
- R，T类型值，通过显式换型操作互换类型。
- 换型操作只在混合表达式和赋值时是必须的。

显然，Ada怕过于庞杂，牺牲了一半语义保护。但也为强类型少付一半麻烦。

9.3.3 类型的换型、变换和强制

强类型增强了安全性，但定义在一个域上的操作不能施于另一域，类型转换是必须的了。上节已经提到换型，本节我们进一步讨论类型转换。

类型转换分两类，一为换型 (cast)，即映射域间类型标记的转换，物理表示不变，语义解释变了。一为变换 (conversion)，不仅语义变了，对象的物理表示也变了。这两种转换可以显式也可以隐式调用相应转换函数实现。如果隐式自动调用转换函数则称强制 (coercion)。所以，换型、变换、强制是三个不同概念。当今文献和译名多有模糊之处。本书暂时这样定名。

(1) 换型

类型换型指明映射域之间的转换，数据存放物理结构不变，类型标记换了它的语义就变了。许多语言的基本类型都由整数映射(整数由字位串映射就不再细说了)。例如，Pascal允许以下基本换型函数：

| 被表示域D' | 表示域D | 换型函数 | |
|--------|------|-------|------|
| | | D' 到D | D到D' |
| 枚举型 | 整型 | ord | 非基本 |
| 字符型 | 整型 | ord | chr |
| 地址 | 整型 | 不允许 | 不允许 |
| 真值 | 整型 | ord | 非基本 |

从整数到真值的换型没有预定义的基本换型操作，程序员可以自己写出换型函数：

例9-24 Pascal实现整型到真值换型

```
FUNCTION IntToTrue (k: Integer): Boolean;
BEGIN  IF k = 0 THEN IntToTrue := FALSE
      ELSE intToTrue := TRUE END;
```

但Pascal的地址禁止换型，使得它难于做系统程序设计。C语言就灵活多了。换型更典型的例子是复数中的虚数。它的表示、译码和实数完全一样，只是操作结果不同。

例9-25 Ada换型实例

```
type IMAG is new FLOAT;
function “+” (Q,R: IMAG) return IMAG is
begin return IMAG(FLOAT(Q)+FLOAT(R)) end “+”;
function “*” (Q,R: IMAG) return FLOAT is
begin return (-1 * FLOAT(Q) * FLOAT(R)) end “*”;
```

函数体中“+”定义在IMAG域上，类型是 $\text{IMAG} \times \text{IMAG} \rightarrow \text{IMAG}$ 。“*”定义在IMAG域上，结果是 $\text{IMAG} \times \text{IMAG} \rightarrow \text{FLOAT}$ 。

我们说，从表示域D到被表示域D'的换型为升级(promotion)换型。它要新增语义，即再定义一些限定的操作。从D'域到D的换型为降级(demotion)换型，它要失去一些语义。即降级域的对象只可用表示域上的操作。无论升降级，换型函数均为待换成类型的名字。先降后升，利用老域上的操作定义新域上的操作，这样，就可与新声明的域定义出所需的任何操作。重载运算符就尤其需要了。这在用户定义抽象数据类型时是很基本的。

(2) 类型变换

类型变换改变了类型对象的物理性质：存储大小，译码，引用层次。因此，存在着三种变换。

大小变换 只改变用于表示值的字节数，值的译码和其它语义约定均不变。现代机器物理字长并不统一。许多语言支持统一译码但不同字长的硬件特征使语言有些灵活性，这对可移植性、保持高的时空效率都是致命的。当有混合运算时，不同操作数的尺寸必须调整到能匹配。

C语言就是这种语言，它的整型就有三种长度，1字节也是char，2字节为short，4字节是一般整数。这是为了提高存取效率。短整数和个人机物理字长对应(386以下)。

所谓大小(size)变换，加长为升级变换，缩短为降级变换。加长总是保险的，缩短要丢掉一些信息，甚至是致命的。运算中混合运算时均采用升级变换。尺寸调整是较安全、最容易实现的变换。往往自动进行，称大小强制(size coercion)。Ada只支持大小强制。

译码变换 同一外部域有多个内部译码时，译码(encoding)变换即将一种译码模式转换为另一种最接近的译码模式。理想情况当然是变换前后的两个译码都表示同一外部域。整数到浮点和浮点到整数的变换近似地属于这一类。许多语言都提供了预定义的变换函数。整数到浮点一般比较安全和近似。而浮点到整数则要圆整或截尾，如FORTRAN的：

```
INT (32.7) = 32 //截尾
```

```
NINT (32.7) = 33 //圆整
```

Pascal也提供两个函数，Ada只支持圆整。

引用变换 同一存储对象我们可以在数据值层次引用它，叫做0层引用。这是最直接有效的。也可以在变量层次引用它，叫1层引用。它找到地址(变量名)后再递归引用。也可以在指针层次引用它，叫2层引用，是两次递归引用。也可以通过指针的指针引用它，叫3层引用，也就是三次递归引用，请看下列：

例9-26 C程序对象引用层次

```
#define SAMPLE 50000L           //第0层：长整数
long int k = SAMPLE,           //第1层：变量
    *pk = &k,                  //第2层：指针
    **ppk = &pk;               //第3层：指针的指针
```

所谓引用(reference)变换，即改变它的引用层次，而大小、译码均不变。通过取地址将程序对象变为存储对象是降级引用变换。反之是升级引用变换。这在C中常见。

(3) 类型强制

类型换型和类型变换可以显式地写在源代码中，称显式转换处理。如果在源程序代码中不出现，由编译隐式自动调用这两类转换函数称之为强制(coercion)或自动变换(automatic conversion)。每当程序中的运算符或函数无法匹配时，翻译器就得事先考虑在此处自动强制，改变对象的译码、大小或引用层次。

表达式中的混合运算的操作数，经常是需要强制的。转换的方向可以是升级也可以是降级、系统保证自动强制总偏向较安全、保守。

例9-27 Pascal的类型变换与强制

```
VAR c:Char;
    Number: Real;
    DigitValue: Integer;
DigitValue := ord(c) - ord('0');           [1]
Number := Number * 10.0 + DigitValue        [2]
```

这个例子有两个显式换型，ord将字符转成整数。[2]行的‘+’运算按实数加则触发译码强制，将DigitValue的整数表示改为浮点实数。[1]、[2]的右边表达式触发引用强制，从变量c，Number、DigitValue中抽取值。

类型强制和类型变换一样，也有三种情况：

- 大小强制

大小强制(size coercion)多数程序语言都有此机制。表达式中混合运算是常见的，处处写显式变换太麻烦。此外，字符串处理也是极频繁的，串长度无法预计。APL，Basic 动态分配串的存储，然后将串变量名束定于串值。多长都可以，然而编译型语言(FORTRAN，Pascal，Ada，C)则要预分配某个长度的存储区。待串值读入后，要作结束标志。运算中有时要截出空白。这类操作包括加长、截尾都是自动进行的(Pascal例外)。

- 译码强制

在处理语义相关译码不同的程序对象时译码强制(encoding coercion)带来方便。整、实数互换是极常见的。多数语言都有译码自动强制。APL和Basic走得最远，甚至不需要在源代码上有什么区分。它们根据上下文自动动态译码和强制转换。FORTRAN直到-77才允许自动强制。因为译码变换如前所述容易丢失信息。所以许多语言对于安全的变换给以强制机制，不安全留给程序员作显式变换。这在类型转换规则中规定(如Pascal，C)。Ada干脆禁止译码强制。

- 引用强制

当程序上下文中以提供程序对象要求得到存储对象时，自动的引用变换就是引用强制(reference coercion)。自动递归引用提供了变量地址要求取值也都是触发引用强制。它方便自

然，消除了很多显式递引用。在程序中更多的时间是显式和自动递引用同时出现，要十分小心地写出正确组合。引用强制在C语言中最普遍。

例9-28 C语言中复杂的强制

解释以下声明：

```
int k, *kk, *aa;           //声明一个整数和两个指向整数指针。
int a[5];                  //5个整数的数组。
int f( );                  //返回整数的函数声明。
int *ff( );                //返回指向整函数的指针。
kk = &k;                   //取地址，引用强制
aa = &a[0];                 //取地址，aa指向数组头，引用强制。
aa = a;                    // aa指向数组头，非引用强制，无“&”
ff = f;                    // ff指向函数头，非引用强制，无“&”
```

请注意，C语言的函数名，数组名就是它的首地址，因此无需引用强制操作。这似乎又违反了语言的正规性。

9.4 类型检查

上节我们研究了类型的语义。程序中类型构造子指明了程序中的数据域，域以其上容许的函数(操作)刻划类型的语义。

本质上计算机只有二进制位串一个域，它容许存、取、增长、减短操作。但它太低级了。于是，设定它们的编排，以及在不同编排(译码)上的操作，就映射为高层的域。各个程序设计语言都提供了独立的基本类型域，再由这些基本域组成复合域。程序就是完成定义在这些域上的计算。这些域如果是独立的，明确的，每次计算之前作检查并不是难事。所谓类型检查就是对每个计算操作数(输入)所属域的检查(实现上按9.1节讲的类型对象提供的信息)。操作函数使用限定即该类型规则。

但情况并非如此简单。首先，从低级域映射为高级域的基本类型域，并非完全独立的，且基本类型域各语言并非一致。在整数、浮点、定点、字符、枚举、真值域中，C语言整数、字符、真值不分，Ada的真值以枚举实现导致了内部合并域。这样，粗口径的检查对程序员查错、划分类型初衷没有什么好处。其次，类型定义的类型名、类型构造子带来的名实分离，导致是名字代表一个域？还是实际构造指明一个域？同构造不同名是否一个域？无名构造算一个域，还是出现一次算一个域？类型别名的处理……一系列问题。各个语言都根据自己的特征给出相对完整的说法。这就是“按名等价”和“结构等价”原则。说不通的则不作检查。由程序员负责语义一致性，如C语言函数调用不作类型检查。而Ada力求说得通，导致静态强类型。它按名等价。在声明中每定义一个类型都是独立的。其三，一个程序总是由多种类型域混合计算，强类型更增加了域的数目，我们不可能对每个可能的域组合定义出计算函数。定义复杂计算时将新域对象转换成老域对象，利用老域原有函数定义新域函数。这就涉及到各域之间转换问题。换型是对相同表示的对象的语义约束的更换，适用于映射域。变换是外部相容而内部域差异较大的域间强行作内部替换。包括大小、译码、引用层次。强制则是对换型、变换的自动执行。为此要定义出一系列规则。类型检查就是查这些规则和操作函数是否有符合预定类型的输入/出。

(1) 强类型类型兼容规则

大型软件总是希望程序尽可能自动查出错误，少仰赖程序员的技巧，因此从无类型、弱类型、强制类型、伪强类型、到强类型发展，代表语言是LISP, Basic, PL/1, Pascal, Ada。伪强类型和强类型是一回事，只是在发展初期想加强类型但漏洞太多而得名。值得一提的是强制类型PL/1认为最下层的域是二进制位串，上层映射的各基本域都是可以相互转换的，如果研究

清楚各域转换规则并让程序系统自己做，即强制，会给程序员施展技巧提供很多方便，语言设计者研究了各域之间关系：整数—实数；字符串—数；字位—真值；位串—其它；同种译码的长—短对象。即所有基本类型对象均可自动变换为其它类型。为此定出转换条件，即规则(PL/1有16条之多)。反映到语法上，许多在别的语言中是禁止的本语言也合法。正是由于完全自动，程序员只能巧妙地运用而无助于他作域检查。用得不高明往往会产生莫明其妙的错误。自动过头适得其反。是一个十分有益的教训。

Ada在类型转换上，走的是另一个极端，除了大小强制而外，所有强制都不行。而要程序员显式使用换型和变换。这样无论在程序正文面上检错还是控制安全都有好处，但带来程序臃肿和低效。

从程序语言表达一个安全可靠、易维护、易修改的软件的角度，强类型是一出路。强类型基本观点是：

- 所有程序对象都是有类型的。
- 所有类型域不相覆盖(子集域例外)。
- 所有函数调用(即程序操作)均作类型匹配与检查。

这些工作全部在运算前都必须做完。所以，可以动态实施强类型(把检查代码插入目标程序)，也可以静态实施强类型，编译时尽可能多查错。这就导致语法复杂，规定太死。

静态强类型除以上三原则外，还增加：

- 一个对象严格属于一个类型。
- 一个变量只能存入同一类型的值。
- 类型允许多有多个变体，但在运行前必须查明运算对象当前属于哪个变体(类型)。
- 两类型对象兼容仅当它们属于同一类型或可合并的子类型。

(2) 双类型

静态强类型的严格条件减少了许多运行时检查，使其成为可行的强类型方案。但牺牲了灵活性。这对系统程序员无疑带来许多不便，例如，将一实数作为Hash表计算输入，希望是位串类型。一个下标希望它既是一维又是二维的。实数按整数取模等等。这类灵活性是高级语言一开始就有的，叫双类型(dual type)。请看下例：

例9-29 FORTRAN 以等价语句实现双类型

FORTRAN的EQUIVALENCE可将两个不同类型的变量束定到一个存储对象上。有声明：

```
REAL Z , X
INTEGER INTZ , M1(100) , M2(4 , 25) , K , N
EQUIVALENCE (INTZ , Z)
EQUIVALENCE (M1(1) , M2(1 , 1))
```

则可写以下语句：

```
Z = X           //将实数X拷贝到Z
N = INTZ-1       //Z的值当整数解释，减1
K = M1(29)       //将第29元素拷贝到K
K = M2(1,8)      //K值依旧。M1(29) = M2(1,8)
```

Pascal虽然要强类型也为这类双类型开了逃避检查的窗口，即非判别式的变体记录类型：

例9-30 Pascal的变体记录

```
TYPE TwoVariants = 1..2;
   IntReal = RECORD CASE TwoVariants OF
       1: (IntName: Integer);
       2: (RealName: Real);
   END;
   OneDim = ARRAY[1..100] OF Integer;
   TwoDim = ARRAY [1..4, 1..25] OF Integer;
   DualDim = RECORD CASE TwoVariants OF
```

```

1: (vector: OneDim);
2: (matrix: TwoDim);
END;
```

作类似例9-29的计算;

```

R. RealName := x;
N := R. IntName - 1;    // 按整数解释
K := A.vector [29];
K := A.matrix[2,4];    // 按一维下标解释
```

请注意, Ada也有类似的变体记录, 但它仅仅是给程序员写结构变动的记录以方便。实现时是多个存储对象, 而不是一个存储对象多种解释。

C语言走得更远, 它有自由联合类型union, 也是类型构造子。但它只按联系成分中最长的分配存储:

例9-31 上例以C的自由联合实现双类型

```

typedef union { long int_name;           //和float一样长
               float real_name; } int_real;

int_real r;
typedef int one_dim[100];
typedef int two_dim [25] [4];
typedef union {one_dim vector;
              two_dim matrix; } dual_dim;

dual_dim a;
```

有了这个声明, r.int_name, r.real_name, a.vector[k], a.matrix[m][n]可以随意用。更有甚者, C语言使用指针换型马上就改变了对所括内容解释。请看下例:

例9-32 计算0到某实数间的Hash值(C语言)

```

#define TABLESIZE 1000
long int j, k;
long int * plong = &k;           //指向long的指针
float * pfloat = (float*) (&k);  //指针换型 (cast)
*pfloat = 3.1;                   //浮点值放于k的槽
j = *plong % TABLESIZE          // 该值取模得Hash数
```

指针换型其所指对象长度必须一致。只要有了换型, 绝不可移植(因两类型长度, 各机对“一致”的解释不同)。

(3) 强类型回避检查方法

上述指针技巧, 自由联合类型, 不加限制地换型, Ada都是不允许的。为的是保持静态强类型原则。Ada才能保住大型军用软件的高可靠性和可移植性。所以, Ada 83的程序易读不易写, 一旦写出能经久好用(程序员“可移植”)。

Ada的变换是在非常有限的映射类型间, 变换必须显式, 其语法和换型一样。自动强制除长度(大小)强制外其它都不许。

但Ada的使命是军事应用, 机载、弹载计算机都是嵌入式的。嵌入式程序以Ada开发和调试, 然后交叉编译到一个芯片、一个单板机上, 往往要嵌入一个小的监控程序(起操作系统作用)。那么也要作十分低级的系统程序设计。为此, Ada有低级设施提供降级的映射类型。可以定义操作码地址的长度、格式, 同时它又是Ada程序世界中的用户定义类型。请看下例:

例9-33 Ada的机器代码程序包

```

package MACHINE-CODE is
```

```

BITS : constant := 1;
WORD : constant := 8;
type OPCODE is (MOU, SUB, ADD);
  for OPCODE' SIZE use 2*BITS;           --规定枚举值长度
  for OPCODE use (MOV=>2 # 00 #;
                  SUB=>2 # 01#;
                  ADD=>2 # 10 #);        --定义操作码格式
type REGISTER is range 0..7;
  for REGISTER'SIOZE use 3* BITS;        --定义8个寄存器号
type INSTRUCTION is                      --指令，从什么寄存器取数，执行命令后
  record                                  --送回什么寄存器
    COMMAND      : OPCODE;
    SOURCE        : REGISTER;
    DESTINATION   : REGISTER;
  end record;
  for INSTRUCTION use
  record at mod 1 ;
    COMMAND at 0 WORD range 0..1;
    SOURCE   at 0 WORD range 2..4;
    DESTINATION at 0 WORD range 5..7;
  end record;
end MACHINE_CODE;

```

这个程序包描述的机器码是一个小型八位字长机器指令。头两位是操作码，两个后三位是寄存器代号。可以写出汇编程序如下：

```

with MACHINE_CODE;
use MACHINE_CODE;
procedure ASM_PRG is
begin
  INSTRUCTION' (COMMAND=> MOV, SOURCE=> 0, DESTINATION=> 1);
  INSTRUCTION' (COMMAND=> ADD, SOURCE=>1, DESTINATION=>2);
  INSTRUCTION' (COMMAND => MOV, SOURCE=>0, DESTINATION=>1);
  INSTRUCTION' (COMMAND)=> SUB, SOURCE=> 1, DESTINATION => 3);
end ASM_PRG;

```

执行这个程序如同汇编，则常规枚举，整数类型约束则不能施加于OPCODE，REGISTER类型上。故允许两个类属子程序：

```

with UNCHECKED_DEALLOCATION;  --不检查无用单元是否回收。
with UNCHECKED_CONVERSION;  --不检查类型变换

```

用上UNCHECKED_CONVERSION之后，类型检查由程序员负责且代码不可移植。所以尽可能限制在最小的包模块内。

前述指针换型技巧在面向对象C++中实现继承和动态束定时，极为普遍。Ada-95的标签类型(tagged)的标签(tag)，实则为指针。父、子指针的换型实现动态束定就是利用这个技术。但其它高层映射类型依然是强类型的。

9.5 类属域

前几节我们讨论类型域、类型检查、和类型转换以及什么是强类型。这些都是基于类型是

固定域的，它给出类型的结构并指出域。研究这些域的关系时，一般是把这些域看作是独立的，每个域中只有一种类型对象。我们也看到可兼容子域、变体记录，以及第8章中的类属机制。事实上，程序中最频繁最有用的是类属域。所谓类属域是该域中可包括多种特定类型的对象，即所有特定类型域的总和。

9.5.1 类型的多态性

一个程序设计语言，每个常量、变量、形参、函数返回值都是有且只有唯一类型叫单态 (Monomorphic) 类型系统。与之对应的函数叫单态类型函数 (输入/出倒不一定是同样类型)。

Pascal 的程序世界里，用户定义的函数和过程都是单态的。但Pascal内定义的函数和过程却不可回避采用多态。例如，运算符‘+’、‘-’、‘*’的两端操作数可以是(整，整)，(实，实)，(整，实)，(实，整)四种类型对的混合运算。再如write(E)函数，若表达式E是字符、串、整数均可以正确输出，write是什么类型域上的函数呢？此外，Pascal的内定义测试函数eof。它的实参数变元类型是任何文件类型，它测试该文件变量是否终止。该函数是File(τ) (‘ τ ’是任何文件类型的符号)即从 τ 到真值的映射：

$\text{File}(\tau) \rightarrow \text{Truth_value}$

我们称这个函数是多态函数，因输入类型是多态 (polymorphic) 的。

Pascal是编译型语言，它根据多态内定义函数的输入/出参数分辨“重载”，编译后仍然是单态类型且不作运行时检查。但Pascal还是留下了用户可用的多态函数：任何类型的子界类型均可继承其基类型的操作。这些操作(函数)都是多态的。

使用单态无非是便于管理和简化，但限制了表达能力，例如自顶向下设计开发时，早期类型怎能定得那么死。有些操作天生来就不讲求类型；交换两数据存储的位置；堆栈和队的操作；二叉树的算法…特别是数据抽象其类型总具有概括性。还有继承中如果不想继承某个数据或操作则用重名覆盖…这都是近代语言要求多态系统的需求。因此，原来在内部实现的技术向用户开放。典型的多类型系统的应用是：

- 运算符重载。
- 灵活数组 (长度可变)。
- 参数化类属包、类属函数/过程。
- 实现具有继承的类体系。
- 多类型 (类型是变量、运行中类型可变)。
- 在语言中指定类型域之间关系。

9.5.2 类属多态的类别

类属域是一个抽象的域，它表示的是各子域共享的结构和语义性质。类属域是抽象数据域，即抽象数据类型的域。与之对应，定义在该域上的函数称抽象函数或虚函数 (virtual function)。虚函数也表达过程抽象。程序设计语言中的抽象数据类型由类属域和一组虚函数声明表示。

(1) 虚函数

和一般域上定义的函数、类型一样，虚函数也有它的语义内涵和外延。所谓概念的内涵是指该概念的本征特征，外延是本征特征概括的范围。例如，“国家”概念的内涵是“土地、人民、政权、主权”。“中国、美国、法国、俄罗斯…”都是“国家”概念的外延。当然，每个本征特征又有自己的内涵和外延，例如，“主权”的内涵是“独立外交，独立立法，司法，…”。香港有土地、人民、政权但无主权，符合“地区”概念内涵，不符合“国家”概念内涵，因此，不是国家的外延。即“国家”不包括它。愈是抽象概念内涵愈小外延愈大。如果把“表”定义为“有表头、表尾的数据串，表尾也是表”。那么“整数表、真值表、字符表、表的表…”都是表的外延。如果只说“整数表”，那么除上述定义外尚加“表元素必须是整

数”这个内涵。而0元，1元，n元整数表均为其外延。

我们可以把表定义为抽象数据类型，它所对应的域是类属域，图示如下：

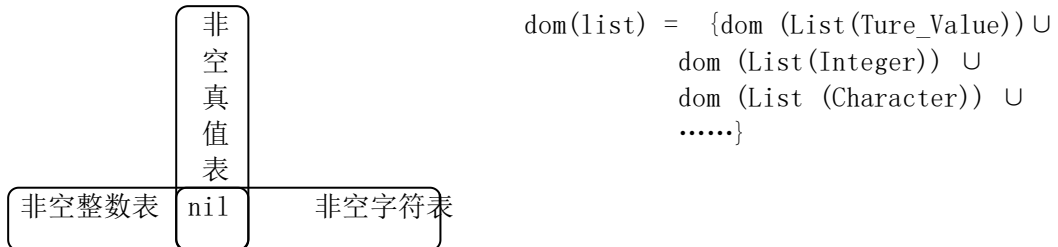


图9-3 表的类属域

表的类属域即它的外延域，也就是符合“表”内涵定义的所有表域的并集域。而真正属于表类属域“自己的”元素只有一个：“空元素nil”，是各子域的交集域。虚函数是定义在类型域上，各子域均可用此虚函数。当子域未定义时，虚函数只可以施行于自己的(空)元素上(当然有的类属域子域交集可以是非空的)。所以叫它虚函数，它是什么也不干不了，什么也能干的函数，只要不断增加新子域。类属域的大小是当前各子域大小的总和，没有上界。下界为一个空元素。虚函数也叫类属函数。虚函数可以实例化为许多类型的函数，它是多态的。

(2) 类属实例化的方式

定义新子域的过程就是类属实例化的过程，对于一个类属域上的抽象数据类型，它的类属实例化包括两个方面：数据的实例化和操作(即函数)的实例化。由于各语言原旨不同，在采用类属机制上不完全一样，呈现不同的类属多态特征。现总结如下，其图示如图9-4：

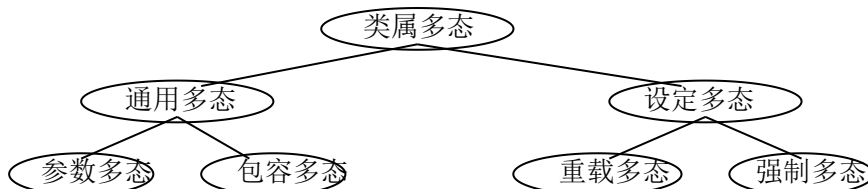


图9-4 类属多态的图示

类属多态实现，即实例化不外乎三种情况：

- 数据例化，操作不用动。操作原本就是通用的叫通用类属函数(universal generic function)。如父类型(域)上的函数可用于子类型(域)上，这种多态叫包容多态(including polymorphic)。不同类型值直接代入参数化函数叫参数多态(parameterization polymorphic)，函数例化后使用。
- 数据例化，操作改动。只保持类属函数的语义。这叫设定多态(ad hoc polymorphic)即为不同类型数据设定不同的函数体，甚至变元的个数也可以不一样。如果利用类型强制则改动的函数体要少得多。这就称之为强制多态(coersion polymorphic)。一般情况下同一函数名要配备多个函数体，则此名称为重载的，故名重载多态(overloading polymorphic)。

多态例化的时间上也有区别，上述识别不同例化变量，择匹配者连接，置换或束定，都是在编译时完成的，称静态束定。如果到运行时完成，则可设为显式的类型变量，运行时检查约束条件后匹配。例化数据的选择，匹配操作可任意设计，则谓多类型(polytype)。

下面我们分别介绍各种多态实现技术。

9.5.3 设定多态

最老的语言，表达式中的四则运算符都是多态的。因为“+”，“-”号既可以整型操作数也可以是实型的。既可以单目，又可以双目。一般情况下不能预定义一个机器的模块，完成所有希望的语义，编译时拿出来连接上，目标码就完成预想的计算了。而一个模块实现一种类型语义，如整数减或实数减或取负…。传统上把实现同一函数的不同代码块叫方法(method)。编译首先识别操作数，然后派送(dispatching)一个正确的方法，将它束定到操作(函数)名上，运行时不再检查就是正确的。这些方法除了对不同类型完成相同的外部语义而外，结构上，算法过程上没有什么共同之处，所以叫设定(ad hoc)多态。设定多态又分两种：

(1) 重载多态

尽管老语言内定义函数是重载的类属函数，但在程序世界依然是单态类型。甚至任何程序实体都不能有相同的名字，这时编译要方便得多。但语义相同的操作，多个程序员同时开发一个大程序，都要设计这种操作时，谁能顾得上你取了这个名字我就不能再用了。重名是大型软件开发必然的需求。于是，近代语言将传统语言中识别运算符操作数，派送合适的方法，这种技术开放到程序世界。即允许程序员为一个函数名定义不同参数的多种实现(函数体)。每个实现都束定到同一个名字上，该名字如同承受重荷，故名重载(overloading)。

有关重载函数的定义和使用方法，本书已多处举例此处不再赘述。值得注意的是，重载分辨中每种语言都要有自己的约定。否则造成混乱。

例9-34 Ada重载分辨准则

Ada是强类型语言混合运算要显式重载运算符。如整数相除按实数进行并返回实数：

```
function "/" (M, N: Integer) return Float is
begin
    return Float (M) / Float(N);
end;
```

如程序中有以下声明和语句，我们分析结果值：

```
J, N: Integer; X, Y: Float;
X := 1.0; J = 1;
Y := X+7.0/2.0;      --Y = 1.0+3.5 = 4.5 [1]
Y := X+7/2;          --Y = 1.0+3.5 = 4.5或1.0+3.0 = 4.0 [2]
Y := J+7/2;          --Y = 1+3 = 4.0 [3]
N := (7/2)/(5/2)      --N = 3/2 = 1 [4]
Y := (7/2)/(5/2)      --Y = 3/2 = 1.0或3.5/2.5 = 1.4 [5]
```

[2]行Ada调上述重载“/”函数得Y = 4.5。Pascal, ML则调用常规整除，Y = 4.0。[3]行[4]行无歧意都调常规整除。[5]行Ada调新重载的“/”，Y = 1.4。Pascal, ML是Y = 1.0。

Ada采用上下文相关重载的分辨约定。即下一步计算“希望”是什么类型就按什么类型。例如，上例中[3]行，子表达式两操作数均整型。再查看整个表达式“希望”得出什么。因为J是整型当然希望整型，故常规整除。[5]行因Y是实型，则用新重载的“/”。中间“/”则调常规实型“/”。

Pascal和ML均采用上下文无关的分辨规则，即只看两操作数是什么类型。和重载函数的参数匹配。

顺便说说上下文相关重载的概念不仅用于函数，也可用于变量和字面量。例如，字面量32根据上下文它可以是32, 32.0, (26)₈, (50)₁₆，只写“32”这样的表达式方便多了。但一般说来，这种方便性带来不安全，近代语言又恢复了显式的标记，如C语言的字面量规定。

同样道理，重载函数技术一旦向程序员开放，怎能禁止程序员将语义不相关的函数取上同样的名字？特别是和人们习惯的概念相反的语义。例如，重载运算符“*”实则是两向量相加。尽管程序没错，这样定义也许出于不得已而为之，但A*B在一般程序员看来是两向量相乘，就给

程序维护造成麻烦。所以，重载函数仅仅是实现类属函数的一种手段，决不等于怎么写也是类属函数。

(2) 强制多态

重载多态分辨后为某单态类型在系统程序中并不多见。因为不同类型域混合计算是常见的。例如，FORTRAN可作算术运算的基本类型有四种(INT, FLOAT, DOUBLE, COMPLEX)，即使只考虑“+”的双目运算，它们不同的组合有16种，且每种方法都不相同。如果有F个运算符作双目运算，有R个不同表示的域(FORTRAN是以上四种)，则方法总数是：

$$M = F * R^2$$

FORTRAN实现四则运算的方法有64种。也就是要准备64个方法体以便派送。为了减少方法数，很早就使用了类型强制技术。例如： $+(FLOAT, INT) \rightarrow +(FLOAT, FLOAT(INT))$ 。FORTRAN实际上只有16种方法(读者想想为什么不强制为四种方法，每种算符一种?)。由此，FORTRAN内定义了以下强制：

| 用户可用的强制 | 返回值 |
|--|---------|
| INT (REAL), NINT (REAL), INT (BOUBLE) | Integer |
| REAL (INT), REAL (DOUBLE), ABS (CMPLX) | Real |
| DBLE (INT), DBLE (REAL) | Double |

这里之所以仍用“强制”一词是因为它们有换型，有变换都用一种表达形式。

强制类型变换给实现者提供了选择。例如，Pascal认为整数和实数，外部域是可兼容子集关系，内部只设实数方法，整数转实数，算完再转回。(int, real), (real, int)更包括在内了。牺牲了效率、简化了系统。PL/I走得最远，任何基本类型均可相互自动转换。但任何类型的组合映射出的外部域比程序需要的外部域大得多。于是给程序员许多使用规定，但事情并非为设计者想的那样，稍有不慎产生许多无意义的转换，又因内部自动，查错极其困难。Ada则总结这个教训，宁肯增加方法数，也让程序员控制类型转换，以便查错。

例9-35 Ada的扩充运算符

```

type MASS is new Float;
type VECTOR is array(1..5) of Integer;
function "+"(X:Integer; Y:MASS) return MASS is
begin return MASS (Float(X)+Float(Y))end "+";
function "+" (X:MASS ; Y:Integer) return MASS is
begin return MASS (Float(X)+Float(Y)) end "+";
function "+" (X,Y:VECTOR) return VECTOR is
Z:VECTOR;
begin
  for K in 1..5 loop
    Z(K) := X(K)+Y(L);
  end loop;
  return Z;
end "+";

```

其中MASS到Float是降级换型，Integer到Float是变换，算完后Float到MASS是升级换型。

重载和强制经常是一起使用的。只是在概念分类上它们不同。一个是同一外部域有不同的内部域。一个是多个内部域又映射为一个或少数几个内部域。

9.5.4 通用多态

存数、取数、相等比较这些操作原本对任何类型都是通用的。调用这些函数形实参数不用作类型匹配检查。它们的形参就是最原始类属类型。这些函数通用于类属域上的各子域类型。

类型强化并开放用户定义类型之后，有许多与数据类型关系不大的操作或算法，也要按不同类型独自写一段程序，如交换两个数，快连查找，堆栈的弹出和压入等等。于是人们想到早期的类属类型。能否写一个函数适合多个类型。

子类型和父类型虽是两个类型，它们的语义行为一样。定义在其中一个上的函数自然能用在另一个上。所以类属函数通用于类属域上各子域。

在强类型时代，函数调用是要作参数匹配检查的。类型参数化，和包容子类型是易于实现的，不过到了面向对象时代这个本来比较简单的包容多态就复杂得多了。

(1) 参数多态

类型参数化在早期语言的内定义函数中是隐式的。编译按通用的意图生成目标码，并没有显式的参数化类属函数/过程。Ada提供类属机制(见第8.3节)，其实现技术类似于C语言中的宏替换。因为每个类属程序包或过程使用前都要设例，则将例化的实例类型更换类属类型。其中要做如同函数调用时的参数匹配检查，不过类属程序包或过程是先编译成目标码放于库中，在编译类属设例时调出目标码置换。Ada类属包中不仅类型可以参数化，值、变量、函数/过程都可以参数化。值参数化技术在早期可变长数组中，其长度值是参数化的，已相当成熟。对象(变量)参数化用换成常指针实现。实例化后是对实例对象的引用。过程/函数参数也是用函数头指针(C语言中的技术)实现。

Ada中只有完全符号化的私有和受限私有类型可以和任何类型匹配，也就是它的类属域最大。其它若指明了

类型关键字<>或(<>)

则只能和指定类型或枚举型匹配，类属域的范围是可以控制的。

编译型语言类属参数必须设例，实例编译后才能运行，这多少还有些死板。如果在运行中根据上下文选例化实参，表达能力更好。此时形参符号就是类型变量符了。这就是多类型，我们在9.5.5节中介绍。

(2) 包容多态

Ada的子类型和派生类型形成的类属域是不一样的，尽管有时它们的结构完全一致：

```
subtype SMALL_INT is Integer range 1..100;
type DEV_INT is new Integer range 1..100;
```



图9-5 派生类型和子类型不同的类属域

定义在子类型上的函数自动能用于父类型，反过来也一样，只是要做值域范围检查，这是一般强类型操作数在运行之中都要做的。派生类型类属域上的函数如果是通用类属函数，则变元显式换型。如果非类属则只适合子域，不属包容多态。

在面向对象的对象继承中，父类的类属域外延包容所有子类域。自然，父类定义的操作适用于子类域中的对象，而子类域上定义的操作不能用于父对象。其原因如下图所示：

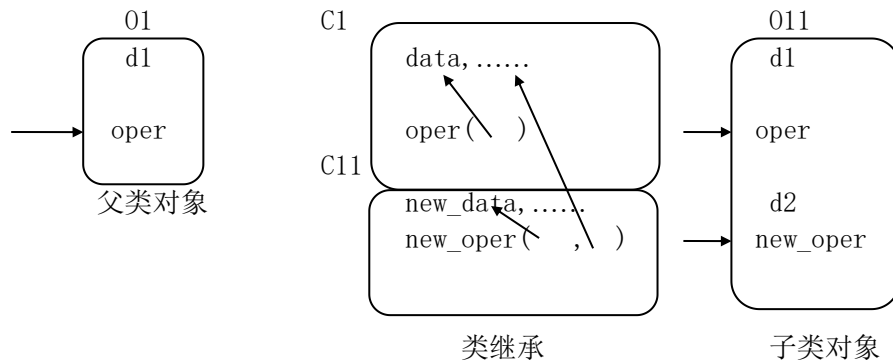


图9-6 父-子类继承域示意图

因子类后定义，向子类实例对象011发消息oper()时，它自己类中虽没有，但可沿继承树到父类中找匹配，也就是系统的调度器派送到父类的块中。反过来，向01发出new_oper()它只能往上不会往下找匹配，故只好报错。如果继承更高层类的数据就往更高派送。这个oper()显然是类属函数，通用于C1的实例对象集合和C11的实例对象集合。new_oper()若不加可见性控制也是类属函数，它被它的子类继承。如果总是图示这种匹配和派送，静态编译也可以胜任。即根据01.oper()和011.oper()分别派送。但面向对象中派送分辨的函数体是动态运行中完成的(如C++)，这是因为直接使用类属对象程序表达能力强得多，例如，obj.oper()可向任何具体类对象发消息。这样对OO中对象横向协作表达计算是十分方便。例如，在雇员类之下有拿月薪的，有计时的。经理是特殊的雇员，它虽然是雇员的子类，当它做工资表时就要向拿月薪、计时的对象发消息obj.computing_pay()。这个obj就是适合于类属域上任何子域的对象。当运行中根据上下文动态决定发给谁。其示意图图9-7

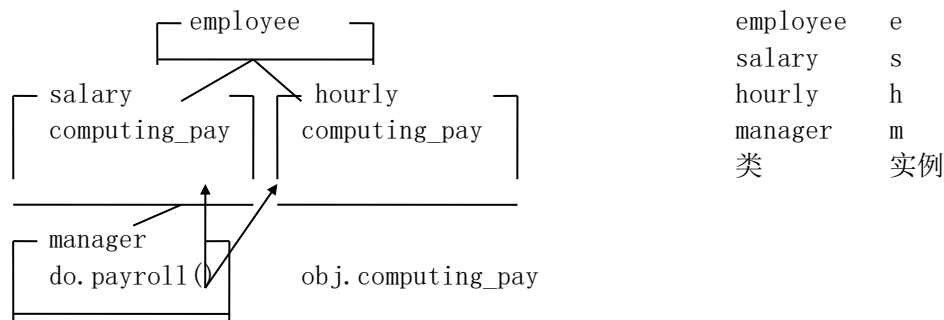


图9-7 雇员类体系消息发送示意图

C++用指针完成这个工作。因为基类指针C*p指向的对象同时也可以指其子类对象。C*p是真正的类属对象，若在运行中有p = &s, p = &h, p = &m, 则p就指向不同类型对象，因此employee *p->computing_pay();是可以发向任何对象(在employee类体系之下)的类属函数。动态中给出对象首地址完成束定，则派送就不是静态中完成的。由运行时(run_time)调度器完成。

不幸的事是C++是编译型语言，选择子'→'在静态束定。即编译完之后p指向employee，选择p的成分已定死了，即使对象的首地址改变，其原有束定关系无法相应改动，也就只好将p = &s中的&s作强制(相当于p(employee*)&s)。动态改变首地址根本不起作用，于是，C++引入显式virtual关键字。virtual函数只编译不束定，在最上层第一次出现virtual的函数中建立一束定表。本类的virtual函数体(方法)只作为缺省方法束定，其它virtual方法在各子类中，按动态填入的子类名，完成束定并调用。沿类体系从上到下匹配派送。

C++的虚函数只是将静态语言动态化的机制，它的束定表相当于变体记录中的判别式。由于是动态自动填写判别式的值，不需要列出每一种case一起编译。它优于判别式记录类型。这样，OO程序可以任意派生而不需重新编译。只编译新派生子类就可以了。

当然，包容多态还涉及构造函数的初始化继承问题，虚构造函数、析构造函数等较为复杂的规定。知道基本思想后都是不难掌握的。

9.5.5 多类型

多类型(polytype)是程序语言提供显式的类型变量, 该变量无需静态例化在运行中可表示任何类型。类型检查在运行匹配时进行, 早期无类型语言只有类型对象(如APL)它以值后标以标签(tag)来束定实现。过程式程序语言中多类型的雏型是联合和变体记录, 程序根据运行中判别式的值自动呈现结构不同的类型, 但在引用和处理中还要显式指明处理的是哪个类型变体。以便静态编译。多类型一般动态检查。不仅表达灵活, 而且可避免程序员错写了判别式, 其代价是运行检查耗费时间。

(1) 多类型的表示和用法

ML语言可以直接定义类型变量, 我们先看下例:

例9-35 ML的类型变量。

ML可显式写出以下形式函数:

```
fun second (s:  $\sigma$ , y:  $\tau$ ) = y
```

其中希腊字是为讲述方便设的, 程序中是某个标识符前加“ σ ”, τ 是类型变量可为任意类型。设函数second的类型是 $f: \sigma \times \tau \rightarrow \tau$ 。当我们引用该函数时(无需例化和强制), 其返回值为:

```
second (12, 15)          15      //整型
second (13, true)        true    //真值型
second (2.1, 4.2)        4.2     //实型
second (13)              //错误, 变元不匹配
second (name)            "watt"   //如果name是"Jeffrey", "watt"
second (1993, 2, 23)     //错误, 变元不匹配
```

该函数是多类型函数, 只要类型匹配正确都可以得到正确的解:

```
Integer  $\times$  Integer  $\rightarrow$  Integer      //匹配
Integer  $\times$  String  $\rightarrow$  String        //正确
Integer  $\times$  Truth-Value  $\rightarrow$  Integer  //错误, 次序不匹配
String  $\times$  Integer  $\times$  String  $\rightarrow$  String //错误, 个数不匹配
```

类型变量使函数的类型成为整个类型族, 该函数的类属域就相当大。例如:

```
fun id (x:  $\tau$ ) = x
```

就定义了一个多态一致性函数, 其类属域是:

```
dom(id) = {真值集, 整数值, 实数集, 字符集, 串集,
          ... 一切能与  $\tau$  类型匹配的类型域}
```

多类型大为提高了语言表达能力, 在描述同构的数据结构上极为方便。例如, 在ML中:

```
type 'a pair = 'a * 'a
datatype 'a list = nil | cons of ('a * 'a list)
```

其中'a即为类型变量的语言具体规定表示法, 第2行的"|"是“或者”可写出若干个, 它是典型的设定类属域。datatype为定义新类型而设, type是简单类型声明。它描述了:

```
pair( $\tau$ ) =  $\tau \times \tau$ 
list( $\tau$ ) = Unit + ( $\tau \times$  list( $\tau$ ))
```

多类型除了能表达抽象类型而外, 在高阶函数中是必不可少的, 例如, 定义复合函数的复合操作“o”, 我们有

```
fun op o (f:  $\rho \rightarrow r$ , g:  $\alpha \rightarrow \rho$ ) =
  fn (x:  $\alpha$ ) => f(g(x))
```

这个函数是将f,g复合成 $h(x) = f(g(x))$ 。形参是两个函数f, g, 其类型是 $\rho \rightarrow r$ 和 $\alpha \rightarrow \rho$ 。函数的返回类型是 $\alpha \rightarrow r$ 。读者可以设想不用类型变量, 就是五种基本类型的组合就有25种。

有了这个定义, 可以如下使用:

```
val even = not o odd           //not (odd)
fun twice = (f: t→t) = f o f   //f(f(t))
```

(2) 多类型表示的抽象数据类型

上小节9.5.4. (2) 指针变量C*p是多类型对象, 但它只能局限于表达类属各子域中的任何对象。类型变量更自由:

例9-37 ML的抽象数据类型

ML定义集合类型, 若用任意类型的表表达则有:

```
abstype 'a set = set of 'a list           // Set of是构造子
  with val emptyset = set([])              //值定义
      fun singleset e = set([e])            //以下操作定义, 本行构造单元素集合
          and union(set(l1), set(l2)) = set (l1 @ l2)*
              //集合并操作'@','*'表示去冗余, 集合元素不重复
          and member (e, set[]) = flase
            |      member(e, set(h::t)) = (e = h) orelse member(e set t)
      end;                                  //对集合中元素成员测试
end;
```

类型名set也是类型构造子, 故接 '=', 它由可变类型的元素的表组成, with以下是该类型的约束, 直至end。member为重载函数, 当空集匹配返回false值。若与形如 $h::t$ 的表组成的集合匹配, 则测 $e = h$ 表头为真返回, 或另测表尾中的元素。

有了这个抽象类型环境, 可以写:

```
val l = singleset 2;           //l == {2}, 构造1值
val l = singleset 'b';        //l == {'b'}
```

函数singleset作用于变元2上, 类型检查时2的int型与'a类型匹配, 同理与'b'也是匹配的。

(3) 从无类型到多类型

多类型内涵趋于零也就是无类型。哲学上本是一回事。无类型没有显式的类型符号, 用户不必声明类型, 依靠值构造子。在无态类型语言中我们已经介绍过了, 它把变量名字和类型值任意束定。值的类型标注在值对象之后, 变量是多态的。也如前指出, 动态类型在执行期间作类型检查, 效率较低, 然而, 无类型也是动态时作类型检查哪为什么还要多类型呢? 原因很简单, 对于描述复杂数据特别是抽象数据类型的一个模块没有名字极不方便, 也不利于检查。抽象数据类型要有类型, 但不必很具体, 以便运行时扩大外延包容, 多类型正好应了这方面的要求。smalltalk的类起着多类型的作用, 所以, 它的变量没有类型声明, 只要消息模式能匹配就计算。声明只解决变量“所属”是哪个类的, 是否共享。不指明变量的类型结构, 变量可以引用任何类的对象。

有类型, 抽象类型必然的结果是引入多类型。是否它可以统一前述四种类型的多态性? 特别是设定多态是当前研究的一个重要方面, 读者可以试探之。

9.6 类型推理

类型推理是程序已隐含有明确的类型，但程序正文上没有显式给出，需要从上下文进行推理，它貌似类属，但不是类属。例如：

Pascal 常量声明 `const I = Exp`中的I。

Ada的循环控制变量 `for I in 1..1000 loop`...

I的类型要按Exp，或1..1000给出的类型，由于没有显式的Exp和1..1000的类型抽象，I的类型虽是确定的，但编译时要推知。

(1) 单态类型推理

类型本身是单态的。如果按强类型思想每个类型编译前已显式得知，那就没有什么好推理的。但多数语言的翻译器都有一定的推理功能。例如，ML可谓是强类型了，但它允许：

```
fun even(n) = (n mod 2 = 0)
```

代替 `fun even (n:int) = (n mod 2 = 0)`

因为，当根据字面量2得知mod的一个操作数是整型，则另一个操作数必然是整型，从而推断n是int，显式声明n: int似乎冗余了。

翻译器保留一定的推理能力是为了程序员不致为琐碎细节困扰。但程序员还应养成显式说明类型的好风格。这样就不致于在调试程序时，调试员也要作推理。冗余没有坏处，这是一个问题的两面说法，翻译器尽可能多一些推理能力，程序员尽可能少用，不用它。

(2) 多态类型推理

多态类型和类型变量是离不开类型推理的。特别是高阶函数。

例9-38 ML多态表的推理

```
datatype  $\tau$  list = nil | cons of ( $\tau * \tau$  list)
fun length(l:  $\tau$  list) =
  case l of
    nil => 0
  | cons(h,t)=>1 + length(t)
```

由于l是 τ list类型，它必然是nil及cons两种情况：由定义cons中h是 τ 类型，t是 τ list类型。由最后表达式中的'l+'得知length(t)的返回值为int，而由nil子表达式的数字0可知它为 τ list \rightarrow int。故length是list(τ) \rightarrow Integer类型。

如果每一步都标注它的类型则程序十分繁琐。类型推理是编译或解释器要实现的功能。

9.5 小结

- 类型对象是实现类型的信息集中描述，它由T型图表示。T型图分别注明名、型、体，体中有存储信息和说明值域的代码定义。这些信息由内部指针实现。

- 对类型或复杂对象的操作可归结为两大类：告诉用户怎样创建程序对象，现代语言提供类型构造子、变量(对象)构造子和值构造子。另一为如何从复杂对象选取引用成分或其值(递归引用)。程序语言提供了多种选择。

- 为了作系统程序设计近代语言为系统程序员提供更多的信息(取自类型对象)，于是属性表达式出现在程序语言中。

- 构造程序对象可以对该类型的值构造，也可以构造对该程序对象的引用。

- 域是有共同物理属性和语义属性的对象集合，该集合上定义计算所需的函数。

- 类型指明的域上的操作、约束和约定的译码即类型的语义。
- 外部域为程序员理解使用的域，内部域为编译理解、实现的域。从内部到外部映射的域统称映射域。
 - 从简单类型映射出复杂类型可构成非独立的域，也可构成新域。
 - 类型变换有三种，两类，一为换型(cast)，一为变换(conversion)，前者物理结构不变束定语义变，后者都变。这两类由机制由系统自动实现作称强制(coercion)。
 - 强类型准则是：所有类型对象均有声明类型。所有类型域不复盖；所有函数调用均作匹配检查。
 - 类属域是语义相关的子域的集合。单态类型指明的域是单态域。类属类型、多类型的域是多态的。
 - 两单态域上没有共用的函数则称彼此独立。一方通用为半独立，完全共用为类属域。
 - 设定类属域其子域人为设定(对象结构不相关)，并可用于同一函数名。它除了类属函数同名而外，变元、方法均不同的称重载类属域。各方法内数据域经强制再次映射为更基本的域称强制类属。
 - 通用类属域各子域不仅语义相关，对象结构也相关。它分为将类型参数化的参数类属和子域间有包容关系的包容类属。
 - 类型强制是外部域相关内部域之间的转换，包括换型和变换。变换的三种形式是译码、引用、尺寸变换。
 - 判别式记录域是内部域结构相关外部域无关的变换。
 - 重载有上下文无关重载和上下文相关重载。后者容易引起释义时的二义性。
 - 参数多态，参数必须设例。一旦设例在程序执行中不变，类型变量则动态可变。
 - 包容多态，两子域有包容关系，其语义相关结构也相同，仅域的大小有差别。定义在基域上的函数和对象可被子域继承。反过来不行。
 - 类和类型相同之处：
 - 都是数据集和操作集的抽象；都是实例和变量的样板，单看都是类型，子类(型)对象也是父类(型)对象。
 - 不同之处：子类继承父类内涵并可扩充内涵，子类型只继承缩小的内涵，类型上的操作(函数)可通用于子类型。类上的类属函数，虽然名字通用，因函数体不同有动态束定问题。类型没有，即使变体记录类型静态亦可分辨和束定。
 - 多类型是有类型变量的类型系统，多类型函数引用时无需例化和转换，可以和任何类型结合。多类型在高阶函数中是必不可少，多类型可以描述抽象类型，增大了语言的表达能力。
 - 多类型的类属域是能与之结合的各子域的并集。多类型本身的域是各子域的交集。
 - 多类型理解和识别要通过类型推理。翻译器应增大识别能力，程序员应尽可能用显式表达类型的风格。

习题

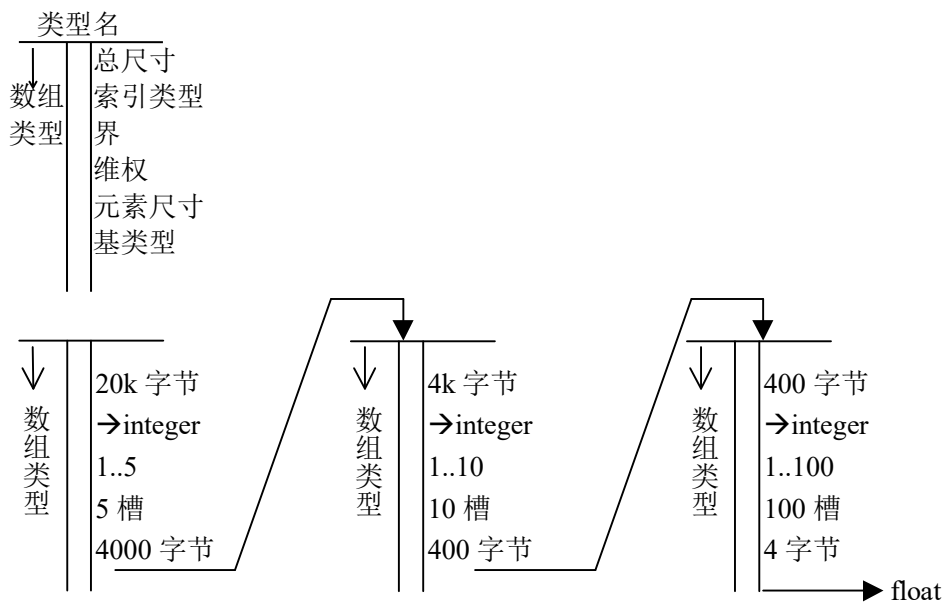
9.1 基本类型和程序员定义的类型有何不同？

9.2 何谓类型对象，它的组成是什么？

答：类型对象是一种信息结构，其中存放的信息用于描述特定域，即值集和可施于值集上的操作集。其组成包括三部分：一为名字的类型，由预定义或基类型名指示的属性。如整、实、字符、枚举等。一为信息体，即该类型的特定属性。

9.3 图解以下MATRIX类型的类型对象：

```
type VECTOR is array (Integer range 1..100) of Float;
type MATRIX is array (1..5, 1..10) of VECTOR;
```



9.4 什么是类型的内部域，外部域？

类型的内部域指实现世界由语言翻译或实现者使用的域

类型的外部域指在程序世界向程序员提供的域

9.5 总结一下Pascal为什么是伪强类型的，它的哪些机制靠程序员负责安全、正确？

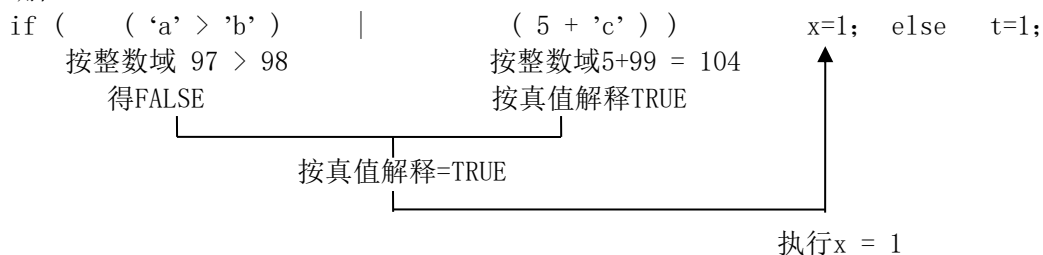
以下条件语句在C中是可执行的：

```
if ('a' > 'b') | (5+'c')) x = 1, else t = 1;
```

'c' 的ASCII码是 Integer 99, 'a','b'是97, 98。

试用释义树画出其结果应是什么？在pascal, Ada中这个语句能执行吗？

答：Pascal的强类型机制并不完善，如类型等价是按结构等价，易引起语义混乱。它采用了非判别式的变体记录类型，为双类型逃避检查提供了可能。类似这样的地方都要由程序员控制正确性。



Ada, Pascal中这个语句不能执行。

9.6 分析以下C程序，若a为零进入本语句循环体重复几次，为什么？

```
do <B> while (a = 0)
```

答：若a为0进入本语句循环体运行一次，因为本语句是先进入循环体，执行一次后才进入while语句部分判断循环出口。

9.7 为什么C++设虚函数，Smalltalk语言没有？Ada的类属函数和C++虚函数的同异。

9.8 为什么“无类型”是“动态类型”语言的代名词？

9.9 何谓类型的构造子？举出例子，它定义了新域吗？

9.10 类型的cast, coercion, conversion各为何意？各有什么不同？给出某个语言的例子说明之。

答：类型转换分两类，一为新型（cast），即映射域间类型标记的转换，物理表示不变，语义解释变了，一为变换（conversion），不仅语义变了，对象的物理表示也变了。这两种转换可

以显示也可以隐式调用相应转换函数实现。如果隐式自动调用转换函数则称强制 (coercion)。所以, 换型, 变换, 强制是三个不同的概念。

举例:

```
cast:   type   IMAG is new   FLOAT;
        T:IMAG ;
        R:FLOAT ;
        R: = (FLOAT)T ;

conversion   a:char ;           coercion: in b;
            b:integer ;           float c;
            b=(integer)a ;           c=b;
```

9.11 Ada 为了安全封闭了不加限制的从一个类型强制到另一个类型; 禁止自由联合类型; 不通过低级设施声明在语言层次上不能象C那样访问位(bit)级实现, 等等不安全漏洞, 你对此评价如何?

9.12 何谓派送(dispatching)? 什么情况用到派送, 我们把实际执行派送的系统程序称为派送器, 它在什么地方?

9.13 重载多态和参数多态有什么不同?

答: 同一函数名要配备各个函数体, 则此名称之为重载的, 故名重载多态, 属于设定多态。数据例化, 操作改动, 只保持类属函数的语义。为不同类型数据设定不同的函数体, 甚至变元的个数也可以不一样。重载函数仅是实现类属函数的一种手段。

不同类型值直接代入参数化函数叫参数多态, 函数例化后使用。数据例化, 操作不用动, 属于通用多态, 即操作原本就是通用的。参数必须设例, 一旦设例在程序执行中不变。

9.14 实现重载的机制是什么? 实现参数多态的机制是什么?

9.15 从检查准确性, 检查对运行时间的增长、协助程序员排错等各方面比较强类型的类型检查和动态类型的运行时检查的优缺点。哪个更安全可靠?

答: 强类型的类型检查优点: 编译时尽可能多查错, 它的严格条件减少了许多运行时检查, 提高了运行效率, 且有利于程序员排错。

强类型的类型检查缺点: 牺牲了灵活性, 这对系统程序员无疑带来许多不便, 例如双类型的类型检查, 要根据上下文进行检查, 所以提前考虑所有可能的错误, 语法复杂, 规定太死。

动态类型的运行时检查: 优点: 运行时根据上下文对所有情况进行检查, 灵活的语义检查, 检查准确性高。

缺点: 运行效率低, 不利于协助程序员排错, 因为它的查错都是根据上下文的特定环境进行的。

强类型的类型检查更安全可靠。

9.16 变长数组(灵活数组)在过程式语言中一直很伤脑筋, 试述Ada和C在解决变长参数数组时各采用什么技术, 它们的优缺点?

9.17 一个字面量是T类型用T'类型重载它, 这种重载是上下文相关的还是无关的? T和T'相关吗?

9.18 以下ML程序中函数not是Truth_Value→Truth_Value, o的类型是 $(\rho \rightarrow \gamma) \times (\alpha \rightarrow \rho) \rightarrow (\alpha \rightarrow \gamma)$:

```
fun negation(p) = not o p;
fun cond (b, f, g) = fnx = >
    if b(x) then f(x) else g(x)
问negation, cond各是什么类型?
```

9.19 以下ML程序中, '+'的类型只限于Integer×Integer→Integer, 问函数sum1, sum2, 各是什么类型?

```
fun sum1(l) =
    case l of
        nil =>0
        | cons(h, t)=>h+sum1(t);
fun insert (z, f, l) =
```

```

case 1 of
  nil=>z
  | cons(h,t)=>f(h, insert (z,f,t))
fun sum2(l) = insert(0, (opt), 1)

```

9.20 Ada中父类型和子类型兼容，同样结构的两子类型：

```

subtype SUB1 is Integer range 1..100;
subtype SUB2 is Integer range 1..100;

```

SUB1, SUB2兼容吗？

```

type VECTOR is array(range 1..100) of Float;
subtype V1 is VECTOR(5..10);
type V1, i2 new VECTOR(5..10);

```

V1, V2兼容吗，上机试一试。

9.21 Ada的类属程序包是：

```

generic
  CAPACITY: in Positive;
  type ITEM is private;
package QUEUE_CLASS is
  procedure APPEND (NEWITEM: in ITEM);
  procedure REMOVE (OLDITEM: out ITEM);
end QUEUE_CLASS;

```

它能作为面向对象中的队类对象用吗？为什么？假设package body是正确无误的。

9.22 程序员为什么不能设计新的运算符，只能重载运算符？

答：因为如果定义新的运算符，则要扩充程序设计语言的语法，如用增加关键字。运算符之间存在优先级等复杂的关系，如果把运算符定义下放到程序员级，将引起极大的混乱。

程序系统中基本类型已定，即使是新的类型分开后亦可划分为基本类型操作而基本类型的运算符也就是运算功能已定，所以只能重载运算符。

9.23 类的继承体系只是一种机制。可以按概括方式组织：

物质—生物—动物—人—青年—学生

也可以按聚集方式组织：

原子—分子—细胞—器官—人—生物

你认为哪种方式好？为什么？

答：我认为按概括方式组织更好。

因为继承体系要求子类能够有父类的特征，能够执行父类的一些操作。继承体系要求内涵不断扩大，而外延不断减小，由此可见，概括方式较好。

9.24 按静态强类型两个结构不相同的域能表示同一外部域吗？

9.25 设TYPE是单态类型 τ 是类型变量，举出程序语言中，什么机制可实现以下函数：

```

f1:  $\tau \rightarrow \text{TYPE}$ 
f2:  $\tau \rightarrow \tau$ 
f3:  $\tau \times \text{TYPE} \rightarrow \text{TYPE}$ 
f4:  $\tau \times \text{TYPE} \rightarrow \tau$ 

```

第 10 章 面向对象程序设计语言

当今席卷软件界的面向对象技术,近因是 xerox 公司 1980 年推出的 Smalltalk-80 语言。当时,美国国防部正在推行投资五亿历时 8 年的 Ada,试图先在军中统一再推广到全行业,使之成为软件开发的主导语言。Ada 是过程语言的新阶段,提供数据封装、数据抽象机制、并发、异常处理;良好的易读性使 Ada 源代码能自成文档;分别编译支持软件叠加和重用;强类型增加软件的可靠性;Ada 率先提出 Ada 程序设计支持环境 APSE,把与机器相关的机制作为预定义环境(包括基本类型和输入/出机制),从而支持可移植性,能较好的满足软件工程可靠、易维护、可移植、可重用的目标。但是,由于 Ada 太大(编译及环境当时的微机放不下),统得过死(廉价编译和环境推出太晚,一定要通过 ACVC 测试才能得到 AJPO 承认),且其本身反映了 70 年代末 80 年代初软件工程思想:严格管理和评审,导致软件开发周期难于缩短,费用依然居高不下。80 年代软件工程推行并不顺畅,一些民间开发小系统的制售商急于寻求新路。Smalltalk-80 带来的面向对象思想为软件技术发展带来了生机。于是,80 年代中期出现了一批 OO 语言,几乎所有老语言都作了面向对象改造。不仅如此,数据库、分布式协作计算、多媒体技术、软件工程工具的集成,甚至计算机硬件都采用 OO 技术。OO 软件开发方法学改变了传统软件开发模式。90 年代 OO 技术为各大公司、各种高技术软件接受,并走向工业化、规范化、主流化的途径,其影响至今尚无消减迹象。

学习面向对象语言和技术,最正宗、最有效当从 Smalltalk 语言开始。我们首先接受它原有的新范型,与传统过程一模块式大不相同的软件构建思路,再去观察各种技术如何引入、结合 OO 技术,就会清晰得多。本章第一、二节介绍原理和实现技术;第三节总结面向对象的基本特征及对应的代表语言;第四、五节分别介绍两个重要的面向对象语言 Ada95 和 Eiffel。

10.1 Smalltalk 语言

对象的思想最早源于人工智能研究,60 年代末描述智能对象的帧(frame)即封装了许多槽(slot),槽既可以是属性(数据)也可以是行为(操作)和约束。但最早见诸文献是 sketchpad 提到的 OO 图形学(1963)。

60 年代挪威的 Dahl 和 Nyard 为模拟系统研制了 SIMULA-67 语言,首先提出封装的类和动态生成实例对象的概念。

60 年代末,美国犹他大学 Alan Kay 在 FLEX 项目中为改善个人计算环境,研制窗口菜单的动态生成和交互时,想到了 SIMULA 的类和对象。由于受到当时软件限制,其成果未推广。70 年代初他到 Xerox 公司 PaloAlto 研究中心参加了 Dynabook 项目。该项目的硬件是 Star(个人机的前驱)软件是 Smalltalk,使用键盘、鼠标操纵图符、窗口、菜单,这在当时是开创性的。1972 年 Dan Ingalls 完成 Smalltalk-72 第一个实用版,以后又经过-76-80 两次改进,Smalltalk-80 成为向外发行的正式版本。由于 Smalltalk 必须在专用机器和环境上运行,它的全新编程范型,以及它的小型系统目标,没有得到很快的推广。但它第一个正式提出“面向对象”完整的概念,在软件界产生深远的影响。

10.1.1 Smalltalk 系统

Smalltalk-80 是专用机上的语言，只是在它成名之后才有一般操作系统(Unix, OS2)上的版本。当时 Smalltalk 系统有四个组成：

- **语言核心(Kernel)** 规定了 Smalltalk-80 的语法和语义，它是交互式、无类型的、编译—解释执行的程序设计语言。
- **程序设计系统** 即 Smalltalk 程序设计支持环境，包括它的界面工具软件和类库支持。主要是类库，它的类库既支持系统(如解释器就是一个类)也支持应用(提供一组预定义类)都挂在基类 Object 为根类继承树之下。
- **程序设计范型(Paradigm)** 由于 OO 程序设计和传统过程语言的程序设计在当时差别较大，它在类这一级设计程序，且以派生新类重用老类为主。程序中的对象向类对象发消息，可动态生成实例对象。它所提供的整套设计方法比较新颖。
- **用户界面模型(User Interface Model)** Smalltalk 系统的哲学是用户在终端上以鼠标、键盘、菜单、图符完成应用程序的编辑、修改、运行。是“用户友好”，以图文并茂的图形用户界面简化使用的先行者。

以下先介绍用户界面模型，再介绍语言核心和程序设计范型，最后给出一完整 Smalltalk 程序。

10.1.2 用户界面模型

Smalltalk 的窗口叫视图(View)，是多任务可复盖的窗口，如图 10-1 所示，每个窗口都有可弹出的菜单及子菜单。用鼠标选择菜单选项即可操作该窗口。有以下标准窗口：

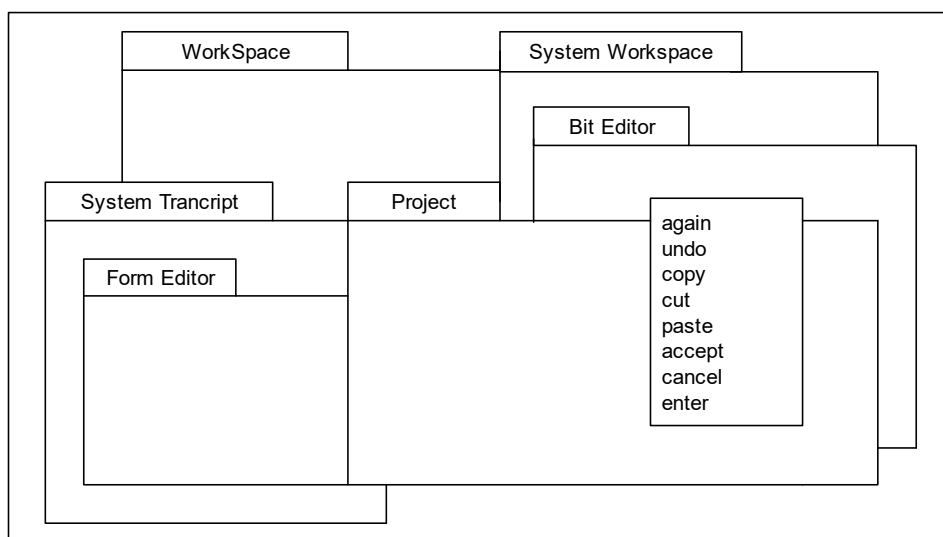


图 10-1 Smalltalk 标准视图(窗口)

- **系统工作空间(System WorkSpace)**

是一特殊工作空间，提供编辑、求值、文件访问、系统查询。使用各种消息表达式以及从故障恢复的功能表达式。

- **工作空间 (WorkSpace)**

从系统菜单中可以弹出任意个工作空间，用户在此编辑、修改(切割与粘贴)程序正文，运行、调试、显示结果。

- **系统副本 (System Transcript)**

是一个特殊的工作空间，其职能是正文收集器，它有一组全局变量，记录用户当前调试的类，当其它机制出故障时，它可以得到反馈的另一拷贝。

- **项目 (Project)**

项目窗口用于组织项目。一个项目窗口创建的变量在另一个项目中可以改变，项目按树形继承组织，子项目完成父项目特定的功能。工作空间开发的正文可纳入某个项目窗口内。是多任务管理机制。

- **两种图形编辑窗 (Form 和 Bit)**

Smalltalk 有面向图形的用户界面，用户可以在位编辑器 (Bit Editor) 之下给出位模式图形，也可以用画笔 (Pen) 和表达式画出位模式图形，得到一图形块。图形块(屏上一个块区域)在 Form Editor (格式编辑器) 窗内操纵，编出更为复杂的图。

- **系统浏览器 (System Browser) 窗**

Smalltalk 程序开发是基于类派生新类，这种环境必须提供浏览器窗口，以使用户浏览系统类库。系统浏览器窗口有五个子窗和两个菜单项(标以 Class 和 Instance)，如图 10-2 所示：

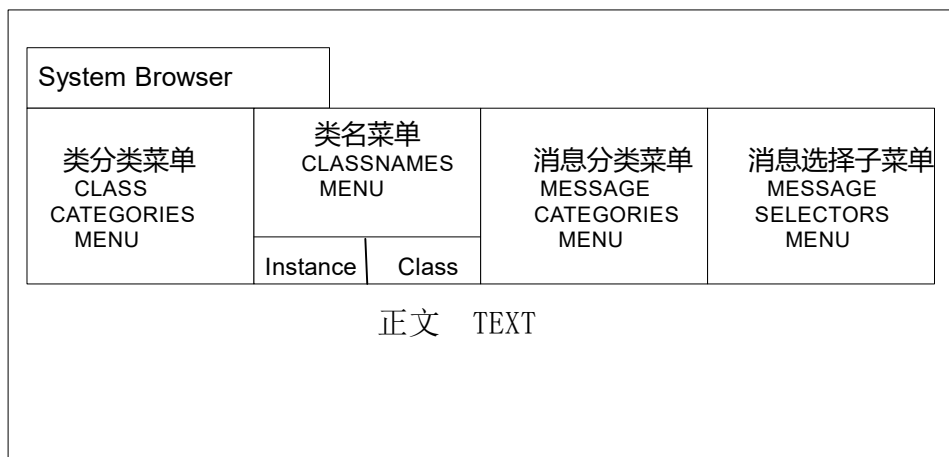


图 10-2 Smalltalk 的浏览窗

Smalltalk 的类按分类组织，浏览时先查第一个分类子窗，子窗中都是下拉式菜单，每选定一项分类，正文框中就显示该项的正文模板。该项(分类)中的成员立即显示在相邻的子窗(类名菜单)中。第二个子窗下的 Instance 和 Class 选项因正文框中显示所选类名项的程序正文模板时，发向实例和类的消息不同而设。选定类名项后，该类名中的消息分类立即在其右邻子窗显示。相应的该类的模板示于正文框。同样，选定消息分类子窗中的选项后，在该选项下的所有消息选择子，在最右子框中列出。与消息分类选项对应的正文模板于正文框中显示。

最后, 选定消息选择子则与其对应的方法显示于正文框, 程序员即可按给定模板写出程序(在工作空间中)。每个子窗都有操作菜单(如运行(doit), 打印(printit), 复制(copy), 切割(cut), 粘贴(paste)……)和本子窗专用菜单(如类名子窗中, 有 file out, printout, spawn, spawn hierarchy, hierarchy, dtfinition, comment protocols, inst var refs, class var refs, rename, remove)。用户就是按浏览窗中显示的模板填写程序。

10.1.3 语言核心

Smalltalk 语言核心非常小, 只提供表达对象、类、实例、消息/方法的表达式的语法:

(1) 保留字

Smalltalk 的保留字只有五个 nil, true, false, self, super, 前三个是常值, 后两个是伪变量, 其值根据上下文, 不能像变量那样任意赋值。true, false, nil 也是同名类的唯一实例, 语义显然。self, super 是在写方法定义时实例对象不在现场的代名词, 都是消息的接受者。self 是本类的某个实例, super 是本类的直接超类的某个实例。

(2) 字面量

Smalltalk 提供五个类(Character, Number, String, Symbol, Array)的字面量实例, 它们各有自己的子类, 如 Number 子类有 Integer, Float, Fraction。

- 字符字面量 所有 ASCII 字符集前冠以 \$。
- 数字字面量 类似其它语言的整、浮点、分数。也有 2, 8, 16 进制表示, 如 2r111, 16rFF.C, 以及复数 22.6+j8r22 和虚数 j22.9(即 22.9i)。
- 符号字面量 标识符、双目选择子、关键字选择子前冠以 ‘#’, 如:
 - #aSymbol 标识符
 - # * 双目选择子
 - #at: put: 关键字选择子
- 数组字面量 在(……)界定数组前冠以 ‘#’

(3) 限定符和特殊符号

不包括算术、比较运算符(它们在 Object, Number 类中定义), 以下符号不得重定义。

- " 注释限定符
- ' 串限定符
- \$ 字符字面量的前缀
- # 符号和数组的前缀
- #() 数组限定符
- , 并运算符
- ; 消息分隔符
- : 关键字消息终止符, 块参数前缀
- | 临时变量限定符, 块参数分隔
- :=或← 赋值号(分别于不同的 Smalltalk 版本)
- ↑ 对象前缀表示该对象的返回值
- [] 块限定符
- () {} 优先控制符

(4) 变量

标识符作变量名，约定第一字符小写用于实例，大写用于类，标识符的假德文写法影响至今，如 isFalseGerman。

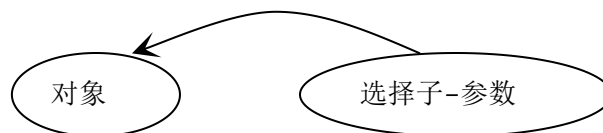
- 实例变量 私有于一个类的一个实例，且只能由该类及其子类的实例方法访问。
- 类变量 一个类及其子类的所有实例可共享的变量。该类的类方法和该类及其子类的实例方法均可访问。
- 临时变量 私有于定义它的方法，仅当所在方法成活才存在。
- 全局变量 Smalltalk 系统中所有方法均可访问的变量。所有类名均为全局变量，系统字典 Smalltalk 查询所有全局变量。
- 汇聚变量(Pool Variables) 是仅能由定义为汇聚字典的一些类访问的全局变量。
- 参数 是定义在方法中消息选择子带的对象，私有于方法。

(5) 消息表达式与语句

Smalltalk 是基于表达式的语言，‘语句’并不重要，只是多个顺序表达式用 ‘.’ 隔开来称语句。

消息表达式的一般格式是：

对象 选择子 参数



选择子(selector) 带着参数向对象发消息，此对象叫做接受子(receiver)。Smalltalk 的消息表达式有三种：

- 单目的 不带参数
 - tree class 消息 class 发向 tree，得到 tree 的类
 - 0.3 sin 消息 sin 发向 0.3，得 sin(0.3)
 - Array new 消息 new 发向 Array，创建一 Array 的实例
- 双目的
 - 3+4 消息 ‘+’ 带参数 4 发向对象 3，得对象 7
 - 100@ 50 消息 ‘@’ 带参数 50 发向对象 100，得 (100, 50)
 - (sum/count) * reserve amount

双目，括号优先

单目优先

双目

消息选择子 ‘/’ 带参数 count 发向 sum 得一对象(总数的几分之一)，消息选择子 amount 发向对象 reserve 得一对象(储金数额)作为参数，由选择子 ‘*’ 带着发向对象 ‘总数的几分之一’，得一对象(储金数)。

- 关键字消息表达式

用关键字(带有 ‘:’ 的选择子)描述的双目表达式，也是自左至右释义。关键字虽非保留字，一旦定义不得改动，且惟一：

anArray at: 3 put:100

finances totalSpentOn: ‘food’

前者释义：选择子 ‘at:’ 带参数 3 发向对象 anArray，得对象(数组第 3 元素)。接着选择子

‘put:’ 带参数 100 发向第 3 元素对象，使之有值 100。

后者释义：用于食物的总金额。

• 赋值 变量在不同时间可赋以不同对象，任何表达式加上赋值前缀 ‘←’ (有的系统是 ‘:=’) 即可将表达式的值束定到左边对象上：

```
quantity←19.
name← ‘chapter 1’.
foo ← array at:4.      “数组第 4 元素与 ‘foo’ 同名”
BicPen←Pen new home; turn: 89.
```

最后一个赋值是：新创建一个 Pen 类的实例在起始处(home)，‘;’ 是级联关键字表达式，省写接受子，相当于 Pen turn: 89(画笔转 89 度)。做完之后束定于 BicPen。即画笔实例 BicPen 被创建于起始位置并转 89°。

• 块表达式

括在[]中的上述四种表达式。块还可以带一到多个参数(前缀以 ‘:’), 并用 ‘|’ 与右边表达式隔开，如：

```
[:x:y | BicPen goto: x@y]
```

意即将 BicPen 移到 $x \times y$ 处，此时 x, y 坐标是参数化的，如写：

```
[:x:y | BicPen goto: x@y] value: 100 value: 250
```

相当于：

```
BicPen goto 100@ 250
```

后面两参数置换 x, y 并按选择子 value: 操作。整个块表达式也是一个接受子对象。

块表达式一般表达一组可延迟执行的动作，例如：

```
| aBlock |
aBlock←[‘This is a String’ displayAt: 500@ 500].
Display white.
aBlock value
```

aBlock 是一临时变量。第一语句的块不完成赋值束定。先执行第二语句。使 Display(即在屏上)对象显示白底，待第三语句执行(求 aBlock 的值)时，块表达式才执行，其值是将串显示在屏中(500, 500)起始的位置上。

(6) 控制结构

有条件选择(conditional selection)和条件重复(condition repetition)，由关键字识别：

条件选择一般形式是：

布尔子表达式

```
ifTrue: [ ‘真’ 块执行]
ifFalse: [ ‘假’ 块执行]      “可以不出现”
```

如：number<0

```
ifTrue: [absValue←number negated]
ifFalse: [absValue←number]
```

执行是：‘<’ 选择子将数 0 发向对象 number 返回一布尔对象，若其为 ‘真’ 执行 ‘真’ 块(上例为一赋值表达式)，否则 ‘假’ 块。

同样，条件重复一般形式是：

```
[布尔块表达式]
whileTrue: | whileFalse: [重复块]
```

如: [index>listSize]

```
whileFalse:[list at:index put: 0.
            index ←index + 1 ]
```

每次对布尔块表达式求值, 若为‘假’对象执行‘假’块直至布尔块表达式求值为‘真’。

(7) 消息/方法

方法(method)是消息的实现, 是静态书写的消息(message), 方法描述对象如何实施其操作, 方法的一般形式是:

消息模式 | 临时变量 | 语句组

单目选择子, 双目选择子带参数, 关键字选择子带参数, 连续的关键字选择子带参数都可以是消息模式。

以一队竖杠(| |)限定的变量集合均为临时变量, 临时变量可出现在语句组之中:

```
newAt: initialLocation | newBox |
    newBox←self new.
    newBox setLoc: initialLocation tilt: 0 size: 100 scribe: pen new.
    newBox show.
    ↑ newBox
```

消息模式是带参数 initialLocation 的关键字选择子 newAt:, 临时变量 newBox。方法体中第一句 self 指本消息要发向本消息所在的类, 本类接受 new 消息生成一实例对象赋给 newBox。第二句是 newBox 接受后面一串连续的关键字表达式发来的消息, 这里 setLoc: tilt: size: scribe: 带的都是实参, 它要找出与之匹配的方法, 计算后的对象 initialLocation 作为参数随 setLoc:发向 newBox。第四句是显示结果(在屏幕上, 如何 show 也要找匹配), 第五句有左‘↑’的表达式返回表达式(本例为对象)求值后的当前值。

如果在这个类 Box 中还定义了实例方法:

```
setLoc: newLoc tilt: newTilt size:newSize scribe: newScribe | |
    Loc←newLoc. titl←newTilt.
    size←newSize. scribe← new Scribe
```

则上一方法体中第二句的消息就和这个消息模式匹配。其结果是 Box 的实例对象 newBox, 其属性 Loc 有值 initialLocation, tilt 值为 0, size 值为 100, 自备的画笔 scribe 已由类 Pen 生成(即 Pen 的新实例)。

10.1.4 Smalltalk 文件系统与虚机

Smalltalk 语言核心只提供表达式、语句、方法、控制结构语法。如何写出 OO 程序和系统如何支持程序设计见后文, 本节先说 Smalltalk 的程序执行。

Smalltalk 是编译-解释执行的, Smalltalk 源程序经编译器得到虚映象(Virtual image), 虚映象由字节代码中间语言编写, 由 Smalltalk 虚机解释执行。相应的文件系统管理三种文件: 源文件、变更文件、映象文件。源文件是共享的以便多个用户浏览。如果用户有了更改, 使用时, 不宜立即施于源文件, 而是作为变更文件为用户私藏。变更文件上有关于用户使用更改的登录。映象文件是被解释执行的文件。

由于 Smalltalk 是交互式的, 被编译的方法在执行期间出了问题要反映到源程序, 则要对映象文件施行反编译(decompilation)得到源程序(但注释和临时变量、形参有丢失)。

Smalltalk 的虚机是一个软件, 它有三个功能部分:

- 负责存储管理器

- 虚映象解释器
- 基本例程 用汇编码写出的底层方法实现，包括输入/出、整数算术，数组下标索引，屏幕图形的基本操作等，其目的是为了改善性能。

存储管理器负责 Smalltalk 对象的存储管理。实现数据隐藏和抽象数据类型的管理。其实现方法本章 10.2 节再作介绍，其他模块(或对象)对存入的对象仅有的操作是：

- 取出一个对象的类
- 存取对象的各属性(字段)
- 创建新对象

存储管理器还要负责无用单元收集并管理自由空间。无用单元回收一般用引用计数法。每当建立引用，该对象标志位置“1”，即该对象可访问。每当撤销引用的指针或束定用的常指针，标志位置“0”，即该对象不可访问。当自由分配空间耗尽，则作无用单元收集，将置“0”的空间列入自由表。

虚映象的中间码文件以堆栈作为虚拟机模型设计代码。解释器逐条按代码(其形式如图 10-4 所示)压、弹栈执行。和一般解释执行的程序设计语言没什么两样，只是效率更快一些。

Smalltalk 系统除虚拟机软件用汇编编写(约 6—12KB)外，其它系统软件(包括编译、反编译、排错、编辑、文件系统)均用 Smalltalk 语言编写(约占整个系统 97%)，保持了良好的可移植性。

10.1.5 Smalltalk 程序设计范型

在 Smalltalk 系统中什么都是对象，大到一个编译器、解释器，小到一个数(如‘3’)都是对象。每个对象都有自己的数据存储和施加于这些私有数据上的操作。数据(即属性)表征了对象的状态，操作在外部激发下操作自己的数据，从而改变了对象的状态。操作在外部看来如同对象的行为(behavior)。而同类型对象有同样的行为。数据属性如用变量描述，就可以用类对象描述同类实例对象。成为它们的样板和产生它们的工厂。程序设计在类的层次上进行，由类静态(于工作空间指明向类发出消息)或动态(方法运行时)生成实例对象。每个对象当接受某消息并执行其方法的消息表达式时都是在向其它对象发消息。接着在被发消息的接受子对象的类中找消息模式匹配，并执行其方法……

(1) 一个简单的 Smalltalk 程序

例 10-1 统计字母出现频率

我们在工作空间中写：

```

| s f |           “定义了两个临时变量”
s ← Prompter prompt: 'enter line' default: '' .
                  “s 是 Prompter 的实例，将关键字表达式的结果束定于 s”
                  “意即输入一行字符串，若不输入，则为空串”
f ← Bag new.      “f 是 Bag 的实例”
s do: [:c | c isLetter ifTrue: [f add: c asLowerCase]]
                  “s 在 Prompter 中找方法 do: 的模式，块表达式是 do: 的参数”
↑ f               “返回 f 中的值”。

```

c 是块变量，意即从 s 中拿出某字符，isLetter 是消息模式，判 c 是否字符，若为真执行内块。内块中 f 找 add: 消息模式，从 Bag 直至上层父类，找到先执行右边子表达式。

c asLowerCase 是单目表达式，同样要在 Prompter 中找 asLowerCase 匹配，也是不成向上找。它返回是“第 k 个”小写字母，add: 把它发送到对象 f 的第 k 个位置上并与原数相加。

这个程序一共四句。如果掀鼠标使菜单项 ‘doit’ 工作并输入：

“Smalltalk is a Programming Language for developing soluions to both simple and complex problem.”

则输出的 f 值是：

```

7 1 1 2 4 1 5 1 5 1 7 4 4 7 3 3 6 3 2 1
a b c d e f g h i k l m n o p r s t u v

```

表示 ‘a’ 出现 7 次，‘b’ 1 次，‘d’ 2 次…系统只处理 80 字符，子串 “omplex problem”，未计入。英文字母是本书为说明而加的。

这个程序用到系统的方法很多，我们几乎看不出串是什么数据结构。实施的算法是什么。以下是尽可能多地写出算法，并用 Paseal 程序作对比：

例 10-2 字频统计对比程序

| | |
|---|--|
| <pre> Pascal PROGRAM Frequency CONST Size = 80; VAR s: string[size]; k,i: Integer; c: Char; f:ARRAY[1..26] OF Integer; BEGIN Writeln('enter line'); Readln(s); FOR i:= 1 TO 26 DO f[i]:= 0; FOR i:= 1 To size DO BEGIN c:= aslowerCase(s[i]); if isLetter (c) THEN BEGIN K := ord(c) - ord('a')+1; f[k] := f[k]+1 END END; FOR i:= 1 To 26 DO Write(f[i], ' ') END. </pre> | <pre> Smalltalk “无消息模式方法，宜写算法” s c f k “定义了四个临时变量” f←Array new: 26. “f 是 Arrey 实例长度 26” s←Prompter prompt:'enter line' default: '' “s 是 Prompter 的实例，装输入字符串” 1 to:26 do:[:i f at:i put:0]. 1 to: size do:[:i c←(s at:i) asLowerCase. C isLetter ifTrue: [k←c asciiValue-\$a asciiValue + 1. F at:k put: (f at:k) + 1]]. ↑ f </pre> |
|---|--|

(2) 类协议

从上例看来，Smalltalk 编程广泛利用以前系统定义的方法。而方法是在各个类协议 (protocol) 中定义。类协议的一般格式是：

| | | |
|---|-----------------------------------|--|
| 类名 超类名 实例变量名 类变量名 汇集变量名 | 标识符 标识符 标识符表 标识符表 标识符 | 单继承，只一个 用于实例对象 用于类对象 充作若干类共享的汇聚字 |
| 类方法： 方法 1 方法 2 . . 方法 n | | 用于创建实例，并初始化 如上例 Array 中的方法 new: |
| 实例方法： 方法 1 方法 2 . . 方法 n | | 刻画实例对象行为 如上例中 asLowerCase, at:put:, isLetter 是对象 s, f, c 的方法。 |

显然在设计类协议之先首先要利用浏览窗查看类库中已有的类，它有什么变量和方法。这里有三种情况：

- 如果所有类的变量和方法都不合要求，则作类定义时写超类名为 Object，继承它的最一般操作，如 new, at:等，其它全由程序员设计。
- 如果有一个类很接近你需要的类，则以它为超类，增、删、派生类的变量和方法。对于‘删去’的变量和方法是以同名变量和方法‘覆盖’，重新定义或置空(动作)。
- 如果一个类完全满足你的需要，则在工作窗如同例 10-1，直接向它发消息，生成实例并初始化，并将它束定到实例变量名上。

Smalltalk 的程序设计除了作类协议而外，就是刻画实例对象如何接受消息，其示意图如图 10-2。运行时 di1, di2, ai1, ai2, ci1, ci2, ci3, 动态生成，并相互发消息。接受消息后，在自己所属类的消息模式中找匹配，若找不到找父类，直至 Object。找到匹配的方法后作行形实参数结合执行方法体，执行中又有子表达式发消息，再找匹配继续执行...

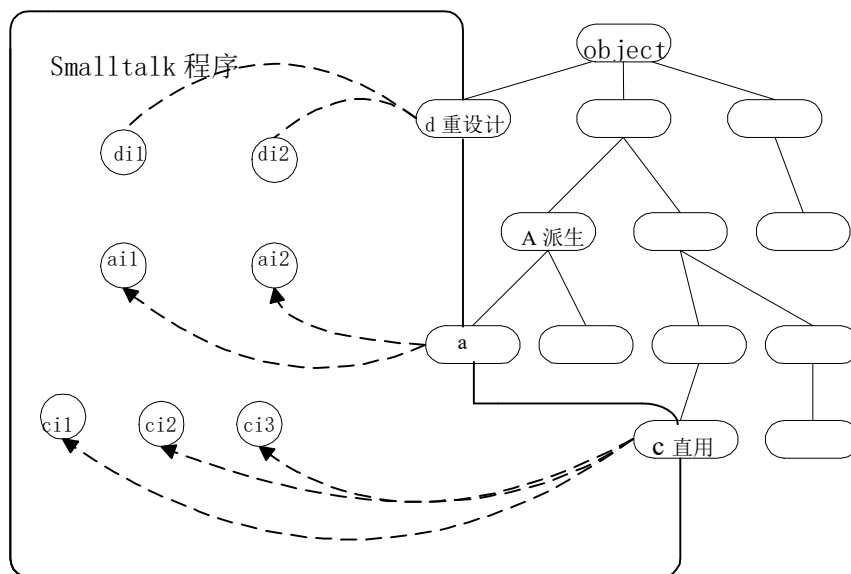


图 10-3 Smalltalk 程序示意图

(3) 一个完整的 Smalltalk 程序

例 10-3 家庭财务帐目

建立全部流水帐类，直接挂在 Object 上

class name FinancialHistory

superclass Object

instance variable names 'caseOnHand incomes expenditures'

category 'Financial Tools'

class method

initialBalance:amount | | “建立流水帐本初始为 amount(元)”

↑ super new setinitialBalance: amount.

new | | “建立流水帐本初始为 0(元)”

↑ super new setinitialBalance:0 “这是两个构造子”

instance method

receive: amount from: source | |**incomes at: source put:**(self total ReceivedFrom:source) + amount.

“从来源 source 接收到的钱数”

cashOnHand←cashOnHand + amount. “因而手头现金增加”

incomes changed

spend: amount for: reason | | “为事由 reason 支付的钱数”，

expenditures at: reason put: (self totalSpentFor: reason) + amount.

cashOnHand← cashOnHand - amount. “因而手头现金减少”

expenditures changed

CashOnHand | | “回答当前手头现金”

↑ cashOnHand

expenditures | | “回答支出细目”

↑ expenditures

```

incomes | | "回答收入细目"
↑ incomes
totalReceiveFrom: source | | "回答自 source 收钱总数"
(incomes includesKey: source)
ifTrue: [↑ incomes at: source]
ifFalse: [↑ 0]
totalSpentFor: reason | | "回答在 reason 项上总支出"
(expenditures includesKey: reason)
ifTrue: [↑ expenditures at: reason]
ifFalse: [↑ 0]
private
setInitialBalance: amount | | "实例变量初始化"
cashOnHand ← amount.
incomes ← Dictionary new.
expenditures ← Dictionary new

```

有了以上 FinancialHistory 类协议，则可创建一实例 HouseholdFinances(家庭财务)并记录收支：

```

Smalltalk at: # HouseholdFinances put: nil.
HouseholdFinances ← FinancialHistory initialBalance: 1560
HouseholdFinances spend: 700 for: 'rent'.
HouseholdFinances spend: 78.53 for: 'food'.
HouseholdFinances receive: 820 from: 'pay'.
HouseholdFinances receive: 22.15 from: 'interest'.
HouseholdFinances spend: 135.65 for: 'utilities'.
HouseholdFinances spend: 146.14 for: 'food'.

```

表达式中 HouseholdFinances 是一全程变量，先向 FinancialHistory 发创建初帐的消息，将返回对象赋于全程量 HouseholdFinances。完成以上表达式，随时可查询“手头现金”，“收入帐目”，“支出帐目”。

程序中 includesKey:, at:put:, changed 是继承自 Object 类的方法。Dictionary 是系统提供的字典类。category 是分类，为浏览归类存放，查询时，按类名，分类名均可。

10.1.6 Smalltalk 程序设计系统

在 Smalltalk 中，系统支持程序也是作为类挂在 Object 之下，包括算术运算、数据和控制结构的实现、输入/出、随机数生成器等。

还有一些类是辅助程序设计过程的，语法分析器、编译器、解释器、反编译器这些对象的方法都有源代码，目标码两种形式。还有一些对象表示类和 methods 的结构，以便程序员追踪系统。还有将方法和向其发消息的对象联结起来的对象，这些对象统称环境(contexts)类似其他语言实现中的堆栈帧和活动记录。

Smalltalk 的窗口、菜单、以及可视信息均以位模式对象表示，并提供编辑器对象，和外部介质的通信，主要是磁盘文件系统。对象和字典(数组)都作为单独的文件，以便通信网络使用。

最基本的类库如下示：

Object

| | |
|------------------------|--------------------|
| Magnitude | Stream |
| Character | PositionableStream |
| Data | Read Stream |
| Time | WriteStream |
| Number | ReadwriteStream |
| Float | ExternalStream* |
| Fraction | FileStream |
| Integer | Random |
| LargeNegativeInteger | File |
| Large PositiveInteger | FileDirectory |
| Smallinteger | FilePage |
| Lookupkey | UndefinedObject |
| Association | Boolean |
| Link | False |
| Process | True |
| Collection | ProcessorScheduler |
| SequenceableCollection | Delay |
| LinkedList | SharedQueue |
| Semaphore | Behavior |
| ArrayedCollection | ClassDescription |
| Array | Class |
| Bitmap | MetaClass |
| DisplayBitmap | Point |
| RunArray | Rectangle |
| String | BitBlt |
| Symbol | CharacterScanner |
| Text | Pen |
| ByteArray | Display Object |
| Interval | Display Medium |
| Ordered Collection | Form |
| SortedCollection | Cursor |
| Bag | DisplayScreen |
| MappedCollection | InfiniteForm |
| Set | OpaqueForm |
| Dietionary | Path |
| Identity Dietionary | Arc |
| | Circle |
| | Curve |
| | Line |
| | LinearFit |
| | Spline |

此外，支持浏览、巡视、表示语法错，以及编译、语法分析、扫描等等还有 20 余个类及子类。最后还有一个追踪器，追踪程序的新版本：

SystemTracer

这些类库支持的类今天看来和 C++, Java, ada-95 非常类似, 在当时的确是非常不习惯的。许多程序员不愿去记忆它并产生抵触情绪, 一个类相当于一个文件, 类库以树状文件目录实现。

10.2 Smalltalk 的对象、类、方法的实现

Smalltalk 的源程序, 即工作空间中写的类协议和消息指令, 经编译生成中间代码, 中间代码只对类中各方法体生成。类、实例对象、活动记录分别存放。全靠常指针联系, 方法匹配靠内部指针变量动态束定。匹配要快速搜索消息模式, 找到后把中间码压入解释执行栈, 逐一弹出执行。

(1) 类的存储

定义了类协议之后。编译时将类对象 (设为 Box) 分配存储并填上中间代码, 其格式如下图:

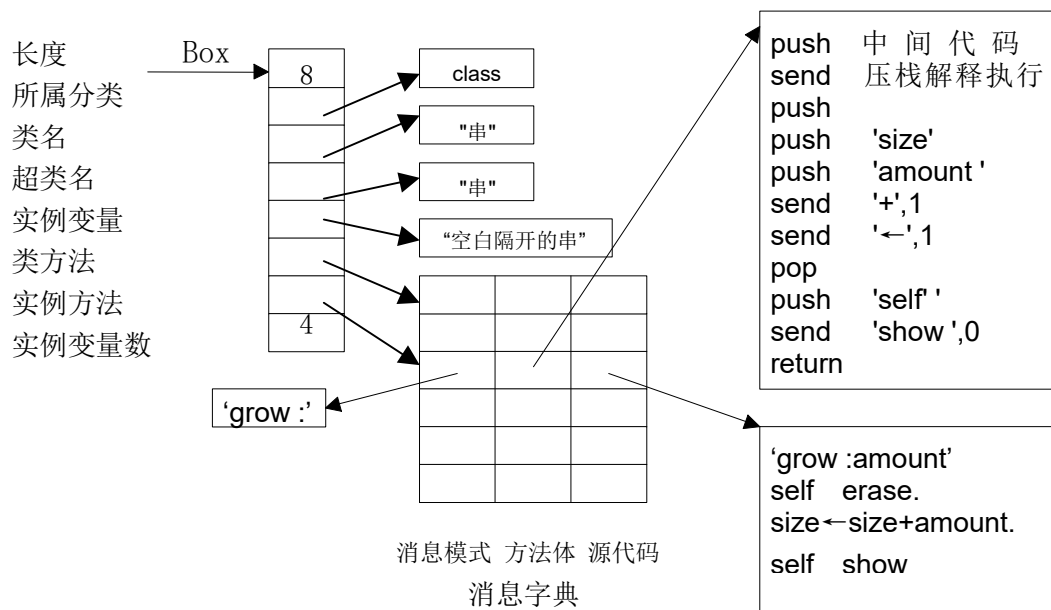


图 10-4 Smalltalk 类协议的存储实现

Box 类表示为 8 个单元, 除第一个填本类协议的长度外其它 6 个全是指针, 指向定义内容, 内容除关键字之外 (如 class) 均以串形式存放。类协议最后一项 “实例变量数” 填 4 (Box 例子中的个数), 是为了在解释执行中分配实例存储快一些, 不必每次数 “空白隔开的串”。“类方法” 和 “实例方法” 指向 “消息字典”, 每条消息都有三个指针域: 指向消息模式、方法体、消息源代码。方法体是中间代码形式, 语义是压栈、弹栈、送数操作。实现的是它所指向的方法体的源代码 (也以串形式), 方法源代码是为了窗口显示而保留的。

(2) 实例对象的存储

实例对象只存放数据, 其存储格式如下图:

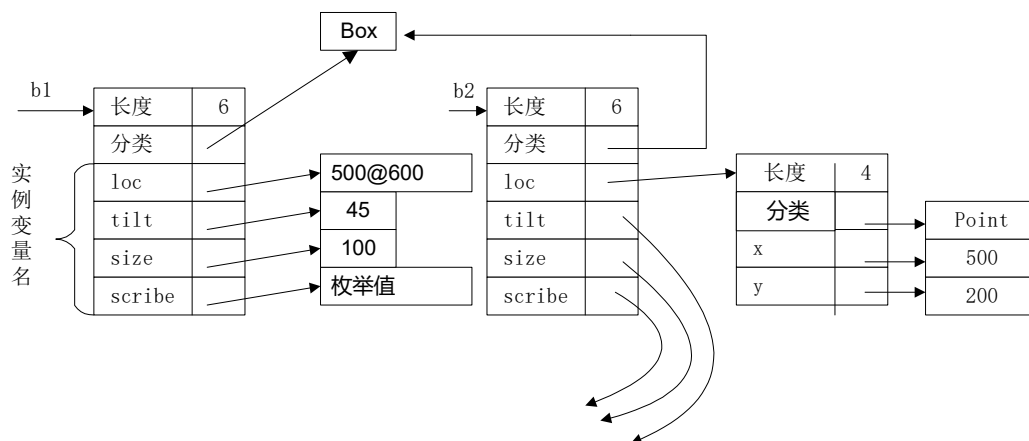


图 10-5 Smalltalk 的实例对象存储实现

图中 b1, b2 是 Box 类的两个实例，他们指向的表中第一列是为了说明才加上的，他们只放实例变量的值。所属分类指向 Box，而 Box 又是指针指向自己的存储片。实例 b1 存放直接值 b2 的 loc 是一点的坐标，则其指针指向 Point 类的实例，x, y 束定于具体值。Scribble 描述画笔抬、落、转向的“枚举”变量。类似的复合数据结构，全按 b2 形式扩充。

(3) 活动记录

方法每次执行如同过程语言的过程调用，执行一个展开的堆栈帧。堆栈帧按活动记录描述的数据展开。活动记录一般分为三部分：

- 环境部分：方法执行用到的上下文：分局部 local 和非局部 non-local 两部分。
- 指令部分：方法执行的指令。
- 发送者部分：消息发送者。

活动记录如下图所示：

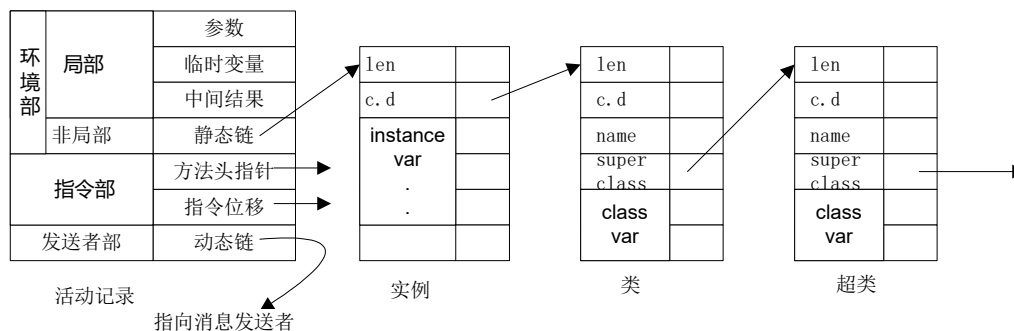


图 10-6 活动记录示意图

其中“静态链”指向本方法执行中要用到的实例变量的值(只能到该实例中去找)。实例和类、超类的连系如同图 10-4。指令部中并不放执行指令，指令仍在类和超类中，此处只放方法头指针和特殊指令相对位移。发送者部的动态链，连接“调用”本方法的(实例)对象。活动记录是方法执行中所有相关信息的总索引图。

10.3 面向对象的基本特征

面向对象是在传统语言和软件技术基础上发展起来的：结构化程序表达导致了封装、

局部性的重视。比过程/函数上一层的封装使之成为可表达高层语义的程序对象。抽象数据类型是程序设计语言的重要发展。有了 ADT 程序员可以显式定自己的构件(component)类型,即新类型,以它为半成品构筑自己的应用。封装性和自主性必然导致继承机制的出现,因为各自为政的封装必然导致大量相同的机制重复定义,不利于维护。然而,继承又为构件重用大开方便之门。极度的继承应用者几乎不用编程序。只要在窗口上指定实例对象,给出参数即完成计算。基于继承的重用则要求函数类型是多态的,否则继承效用有限。由于我们不知道这个方法为今后的什么类型对象用到,只能把它定为多态的,多态有利于程序扩充。Smalltalk 是无类型语言,它的类和子类是包容关系,即子类实例也是父类实例,只要消息模式匹配,即可把方法的体束定于该对象的方法上。寻找匹配(是在动态执行中完成的,称动态束定(Dynamic Binding))。

Smalltalk 表示出的面向对象的诸多优点,正好符合软件工程的局部性、概括表达,可重用,易扩充的要求。然而,它本身是编译—解释型的,效率较低(1/30 C 的速度)。对象概念过于泛化难于编制大型系统(小到一个数,大到一个工具,一个系统都叫对象)。加上它的独特的归约求值方式(沿袭自 LISP)程序员不习惯且不说,它无法利用多年积累的软件资源,于是自它出现之后传统老语言纷纷转向 OO,这里面分三种情况:

- 以老语言模拟 Smalltalk,下层则可利用老语言的资源,代表是 Objective-C。
- 改造老语言,在原有风格基础上增加对象、类机制,尽可能全面反映上述五大特征。代表是 C++, Ada 95 以及 Object COBOL, Object Pascal, Object FORTH……。这是多范型语言之路。人工智能领域有 COMMON (LISP 的对象系统)、Object Prolog。
- 按上述五大特征设计全部的强类型面向对象语言,典型的是 Eiffel。它虽然纯正,但比较学究化,程序员一时难于为每个对象写出约束其行为的公理,推广需要时日。

正是由于面向对象的诸多优点,各种语言结合自己应用域对象化。“面向对象”名噪一时,谁都说自己是面向对象语言。其实有的只是简单封装就宣称支持面向对象编程了。再者,各领域应用的要求也不相同,例如,图形领域特别需要继承, I-CASE 工具只需封装。1994 年, P. Wegner 总结了 OO 语言的发展,给出以下图示澄清了概念:



由以上表中可以看出,这五个特征对“面向对象”是缺一不可的。从语言学的角度,有的文献把面向对象语言又分成“最小面向对象”和“纯面向对象”,前者多半是多范型由原非对象式改造过来的,如 C++和 Ada95。语言只提供实现上述五个特征的可能,在语法结构上保留了很多“不纯”和“不一致”。例如 C++的主函数 main(),只能在概念上把它“看成”是主控对象。Java 对这个“不纯”作了改进。Ada95 更是连显式的类机制都没有,利用无语义的包和输出用户定义的类型实现类及面向对象其它机制。然而,具有讽刺意义的是 Ada95 居然被 ISO 批准成为世界上第一个面向对象程序设计语(ISO/IEC8652:1995),Smalltalk80,

C++, Eiffel 均无此殊荣。

以下我们先讨论几个带共性的问题。

(1) 封装与继承带来的问题

• 访问权限

在封装的可控界面上有 public, private, protected 三种可见性。公有成员既可以接受类外发来消息也可接受类内发来消息。私有成员只可接受类内发来消息。被保护成员只可接受类内或子类发来消息。如果有了继承, 继承后子类中这种关系能否保留? C++ 作以下规定:

| C++类 | 继承方式 | 子类 |
|------|---------|----------|
| 公有 | Public | 公有 |
| 公有 | Private | 私有 |
| 保护 | Public | 私有(仍受保护) |
| 保护 | Private | 不继承 |
| 私有 | Public | 不继承 |
| 私有 | Private | 不继承 |

这种访问权限是为了保证类的继承体系概括抽象的实施。读者可以把其实现方法作为习题。

• 对象初始化次序

C++ 设置了显式构造子, 在继承体系中构造子显然也要继承。定义派生类构造子时也必须指明要继承成员的初始化值:

```

derived :: derived (int i, int j, int k): base1(i), base2(j) {
    d=k;
}

```

第一基类构造函数
第二基类构造函数

如果只是使用缺省的构造子声明派生类实例:

```
derived d;
```

在构造实例 d 时, 编译先调用基类的缺省构造函数再调派生类的缺省构造函数, 是先基后派。同样, 对缺省的析构子其调用次序相反, 先派后基。

(2) 强类型语言的动态多态问题

强类型语言的多态, 我们在函数重载已有介绍。重载分辨是在静态(编译)时分辨: 只按函数型构中的参数的个数、类型、次序匹配, 而不是看谁来调用。特别是以指针指向的对象调用其复盖的函数时, 它无从分辨。

例 10-4 C++ 的无法执行的多态函数

设父类 ellipse 子类 eircle 均有求周长

求面积 area() 函数, 有以下主程序:

```

main( )
{
    ellipse *pe, e(8.0, 4.0);
    eircle c (5.0);
    pe = &c;
    cout << pe -> area( );
}

```

这个程序完全合于 C++ 语法, 父类指针可以指向子类实例。原本希望打印半径为 5.0 的圆面积。然而输出的是胡乱结果或无结果。其原因是 pe 在编译时按 ellipse 类确定各函数调用的

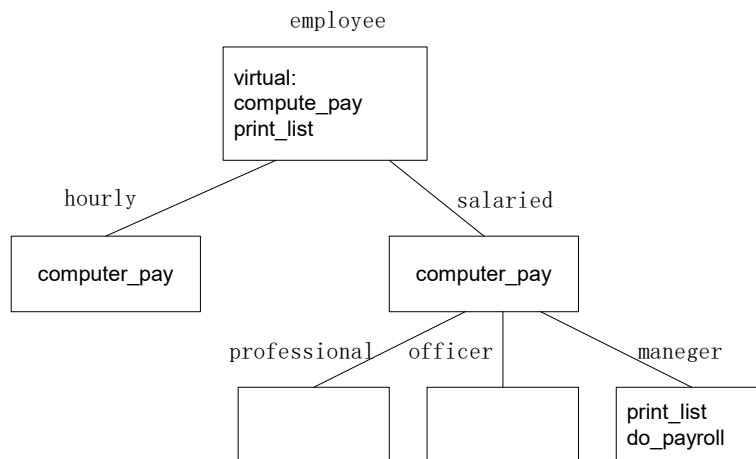
位移。运行时虽然换了首地址 (pe= & c) 它仍然按 ellipse 中 area 的位移找计算函数。而 c 中 area 应该位移多少此时是不知道的。编译后符号交叉引用表已消失。所以, 编译型强类型语言无法实现动态束定。于是, C++引出虚函数机制: 在函数名前冠以 virtual 表示该函数在子类另有定义(可冠可不再 virtual)可以实现动态束定:

```
virtual float area( );
```

实现办法是建立一个虚函数表记下本函数在各子类中的位移。当 pe 代表不同子类对象时, 可以准确找到要计算的函数。再看一例子:

例 10-5 计算并打印职工工资表

设经理、专业人员、行政人员领年薪, 其它雇员领计时工资。其类继承体系是:



若前此已设计 card, pay_data 类, 部分程序是:

```

class employee {
    char name [ ], soc_sec [13], *dept_code, *job_code
public:
    employee * Link;
    employee (int); //构造函数; 变元是名字个数
    void print_paycheck;
    virtual pay_data compute_pay;
    virtual void print_list;
    void employee :: print_empl ( )
    { cout << "Name: " << name << "\n\t" << soc_sec
        << "\t Dept: " << dept_code << "\t Job: "
        << job_code << "\n";
    }
};

class salaried : public employee {
    int annual_salary, vocation_used;
public:
    pay_data comput_pay( );
    void take_vacation( );
};
  
```

```

class manager : salaried {
    employee * staff;
public:
    void add_employee( );
    void print_list( );
    void do_payroll( );
    manager( );      //构造函数
};

class hourly : public employee {
    float pay_rate, hours_worked, overtime;
    int vacation_used;
public:
    pay_data compute_pay( );
    void record_time_card ( card *);
    hourly ( );      //构造函数
}

class officer : salaried { ...};
class professional : salaried {...};

```

其中 `employee :: print_list()` 定义为虚函数, `manager :: print_list()` 也是虚函数。它们的各自定义是:

```

void employee :: print_list( ) {
    employee * scan;
    for (scan = link; scan != NULL; scan = scan ->link)
}

void manager :: print_list( ) {
    cout << "\n\n Manager: ";
    print_empl ( );
    cout << "\n Employees Supervised: \n";
    employee::print_list( );
}

```

用 `scan` 指针扫描每个雇员实例, 如为一般雇员则打印表头和相应数据。如为经理则加打“Manger”和“辖下雇员”再加表头和相应数据。在主控程序中随机读入雇员数据, 它会自动按不同实例对象正确地打印工资表。

10.4 Ada95 的面向对象机制

Ada83 是静态强类型语言, 它的设计思想是运行前尽可能地发现程序错误, 即所有束定均在编译时(最多是在装入后确定时)完成, 运行中动态束定是十分致命的, 以致于 Ada95 的语法扩充至 277 条产生式, 这就潜在着它竞争不过对手。但是, 从学习语言定义, 实现每种机制的角度来看, Ada95 为我们提供了绝妙的教材。

(1) 定义类和实例对象

在第 8 章中我们已经介绍过 Ada83 可以在程序包中定义抽象数据类型。Ada95 以抽象数据类型实现类。类的封装性由包实现，类的继承性则扩充了标签（tag）类型和抽象类型。标签类型只限记录类型。类的继承性利用 Ada 83 的类型派生机制实现子类。示例如下：

例 10-6 Ada 的类定义

```

package Object is
  type Object is tagged           --此类型的数据，即对象的属性
    record                         --无 tagged 即一般的 ADT，有它为了类继承
      X_Coord: Float := 0;
      Y_Coord: Float := 0;         --初值为缺省时用
    end record;
  function Distance (O: Object) return Float; --Object 对象的行为
  function Area (O: Object) return Float;
end Object;

package body Object is
  type ...                         --此处若定义属性是包外不可见的私有属性
  function Distance (O: Object) return Float is
    type ...                       --此处若定义属性局部于 Distance
  begin ...end;
  function Area (O: Object) return Float is
    begin return 0.0 end;
end Object;

with Object, use Object;
package Shapes is                 --这个包封装了三个子类（型）
  type Point is new Object with null record; --只继承不扩充的子类
  type Circle is new Object with      --继承并扩充
    record
      Radius: Float;                --此属性
    end record;
  function Area (C: Circle) return Float; --覆盖 Object 中的 Area
  type Triangle is new Object with    --继承并扩充
    record
      A,B,C: Float;                 --这三个属性
    end record;
  function Area (T: Triangle) return Float; --覆盖
end Shape;
package body shapes is
  --定义三个子类各操作实现
end shapes;
  这两个包定义的类型用 tagged 和 new ...with 建立它们的继承关系

```

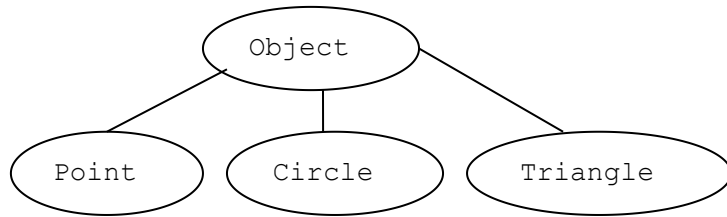


图 10-7 一个继承关系

这些类（型）包外可见（可输出）在主子程序中声明实例，如同类型声明变量，以初值表达式作值构造子：

子类的实例

也是父类的实例

P: Point; --声明实例对象 P

O: Object:

C: Circle: = (0.0, 0.0, 34.7);

P: (O **with null record**);

T: Triangle: = (3.0, 4.0, 5.0);

C: = (O **with** 34.7);

T: = (O **with** 3.0, 4.0, 5.0);

如果动态生成实例，可将此声明放在类的方法（过程/函数）中，调用时生成。

（2）以类宽类型实现多态

Ada95 的每个标签类型都有一个与之对应的类类型属性 T' Class，并把它叫做类宽类型（Class Wide Type）。这是对强类型的泛化，即所有由 T 类型派生出的类型都是类宽类型子域，类宽类型是该类型所有可能派生类型域的总和，显然类宽类型不是具体类型不能用作需要定义的变量声明，除非给予实例化：

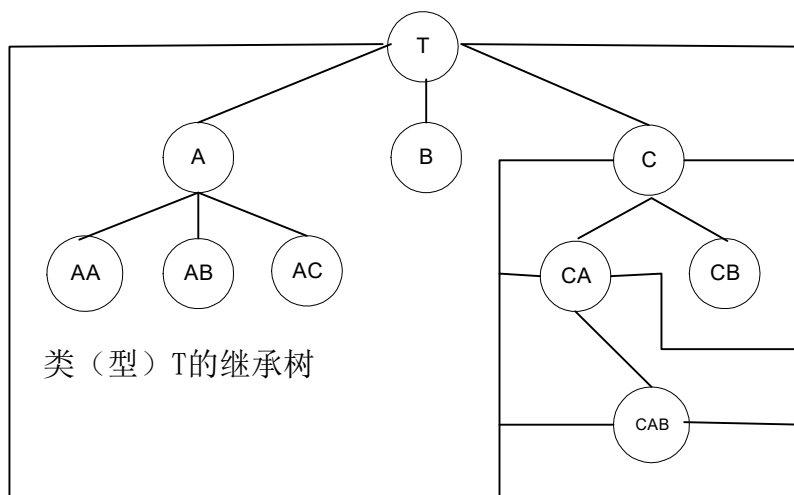
设已声明 T 类型，及 T' Class 的变量 V，则

Y: T; --一般声明，正确

Y: T' Class; --不可以

Y: T' Class:= V; --可以，T' Class 束定为 V 的类型

类宽类型的范围示意如下：



T' Class 域包括图示所有子域

C' Class 域包括 C, CA, CB, CAB 的域

CA' Class 域包括 CA, CAB 的域

CAB' Class 域中只有 CAB 的属性和方法

图 10-8 类宽类型图解

Ada 用类宽类型实现单继承，请看下例：

例 10-7 民航订票系统

乘客向售票员提出订票要求，订票系统处理这个要求，并通知售票员能否满足这个要求。有 3 种座位可供选择：经济舱、公务舱、头等舱。公务舱乘客可选靠窗 和靠过道的座位。经济舱乘客不订餐（只配发），其它舱订餐可选择食品类型有鱼、鸡、牛肉、猪肉。头等舱乘客可选择地面接送。

经分析，将订票基本数据作为基类；舱班号，日期、座位号。经济舱、公务舱作为两个子类。头等舱是公务舱的子类。程序如下：

```

package Reservation-System is
  type Position is (Aisle, Window);
  type Meal_Type is (Fish, Chicken, Beef, Pork);
  type Reservation is tagged
    record
      Flight_Number: Integer;
      Date_Of_Travel: Date;
      Seat_Number: String (1...3); = " ";
    end record;
  procedure Make (R: in out Reservation);
  procedure Select_Seat (R: in out Reservation);
  type Economic_Reservation is new Reservation with null record;
                                                    --子类 1
  type Business_Reservation is new Reservation with
    record
      seat_Sort: Position;
      Food: Meal_Type;
    end record;
                                                    --子类 2
  procedure Make (Br: in out Business_Reservation);
  procedure Order_Meal (Br: in out Business_Reservation);

  type First_Class_Reservation is new Business_Reservation with
    record
      Destination: Address;
    end record;
                                                    --子子类
  procedure Make (Fcr: in out First_Class_Reservation);
  procedure Arrange_Limo (Fcr: in out First_Class_Reservation);
end Reservation_System;

  有了以上类（型）定义，就可以编写一个处理订票程序，无论哪种客户按混合次序队列
  买票都可以处理：

  procedure Process_Reservation (Rc: in out Reservation' Class) is
    --形参可以是类宽类型，不必最初限定某特定类型

  begin
    ...

```

```
Make (Rc);    --它按相结合的 Rc 的具体类型出票
...
```

```
end Process_Reservation;
```

这个程序在编译时 Make 函数无法束定到某函数体上，只有在运行中（动态）束定。束定时按 Rc 对应的实参的标签 tag 值派送，编译时只要做出派送表，本例编译后的派送表是：

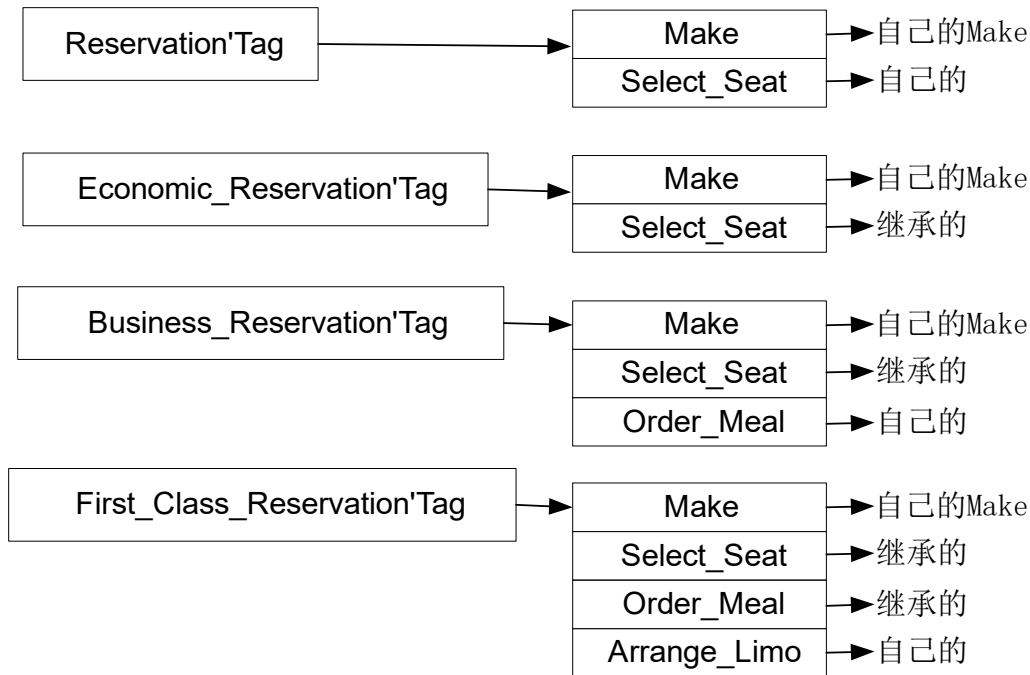


图 10-9 标签和派送表

由此看出 Ada-95 扩充标签类型的目的：编译时内定唯一的、可区分的标签值，给动态束定时以必要的信息。此外，强类型的继承在某类型和派生类型之间若有运算也要显式转换，且只能向继承体系根部转换。本例在包体中实现业务舱 Make（出票）时，可以写：

```
Procedure Make (Br: in out Business_Reservation) is
begin
    Make (Reservation (Br));
    Select_Seat (Reservation (Br));           --显式转换才能调用
    Order_Meal (Business_Reservation `Class (Br)); --再派送
end Make;
```

最后一行是指明类宽类型，它的子类（型）可转换到此新根。凡实参是类宽类型，形参是特定类型就又有派送，称再派送（redispatch）。

（3）扩充程序包机制实现继承的类体系

请注意，以上各子类（型）均在一个程序包内，即使包中用了私有类型（包中成分共享）也不会影响到派送，可以实现动态束定。但如果这个订票系统使用了一段时间，有超音速客机也参与订票，我们如果写一程序包：

```
with Reservation_System;
package Supersonic_Reservation_System is
    type Supersonic_Reservation is
        new Reservation_System.Reservation with
```

```

    record
        champagne: Vintage;
        --其它超音速成分
    end record;
procedure Make (Sr: in out Supersonic_Reservation);
    .....
end Supersonic_Reservation_System;

```

由于有 **with** 子句, 本包与 Reservation_System 建立平面联系, 它输出的 Reservation 类型本包直接可用。本包编译后原包不重新编译立即可用。但是原包中若有私有类型, 新包就不可见了。如果要继承该类型就得修改并重新编译原包, 于是 Ada 95 增设了子辈单元 (child unit) 和私有子辈单元。

- 子辈单元

子辈单元是父单元的扩展部分, 它在父单元声明作用域中, 可以见到父单元的所有部分, 包括私有部分。子辈单元名用父单元名加点表示法, 且无需 with, use 子句, 上述超音速订票系统可改写为:

```

package Reservation_System.Supersonic is           --'.' 后是子辈单元名
    type Supersonic_Reservation is new Reservation with private;
private
    type Supersonic_Reservation is new Reservation with
        record
            Champagne: Vintage;
            .....
        end record;
procedure Make (Sr: in out Supersonic_Reservation);
procedure Select_Seat (Sr: in out Supersonic_Reservation);
    .....
end Reservation_System.Supersonic;

```

这个程序包和普通程序包一样, 对外部用户可见, 叫公有子集, 然而, 它又可见到父辈的私有部分和体。这样, 外部用户可通过子类的公有成份访问父辈的私有细节, 这违反了原封装的原意, 故本子辈包只提供私有成份。一个父单元可以有多个子辈单元, 子辈单元又可以有孙辈单元, 这样形成的树状联系, 可以在程序结构上形成 (单) 继承树。但子辈单元间, 即同辈, 相互的可见性仍然要用 **with** 子句。即在无 **with** 子句时, 子辈单元只可见到父单元内的成分, 见不到同辈单元内的成分。为了安全, 且无论可见性规则如何制定, 编译时的次序是十分重要的, 可见单元必须先于新编单元。

- 私有子辈单元

一个复杂系统, 子辈单元只能扩展父单元时私有部分是非常不方便的, 允许子辈单元扩充公有部份则有可能泄露私有部分。Ada95 提供了私有子辈单元的机制, 它只是父单元私有部分的扩充, 外界不可见, 而它可以见到父辈单元的私有成份和同辈的可见部分, 请看下例: 例 10-8 操作系统的框架程序

```

package OS is                                     --父包 OS
    --OS 的可见成份
type File_Descriptor is private;
    ...

```

```

private
    type File_Descriptor is new Integer;
end OS;
package OS.Exceptions is           --OS 的子辈程序包
    File_Descriptor_Error,
    File_Name_Error,
    Permission_Error: exception;    --所定义异常 OS 各子辈包均可用
end OS.Exceptions;                 --公有，但不会泄露

with OS.Exceptions;
package OS.File_Manager is        --OS 的子辈程序包
    type File_Mode is (Read_Only, Write_Only, Read_Write);
    function Open (File_Name:String ; Mode:File_Mode) return
                                                File_Descriptor;
    procedure Close (File: in File_Descriptor);
    ...
end OS.File_Manager;               --公有，只用私有类型。也无泄露

procedure OS.Interpret (Command: String);    --命令解释过程，等同子包

private package OS.Internals is    --私有子辈程序包，不用 with
    ...
end OS.Internals;

private package OS.Internals_Debug is    --OS 的私有子辈程序包
    ...
end OS.Internals_Debug;

```

两个私有子辈程序包 OS.Internals 和 OS.Internals_Debug 相互可见，且可见 OS 的公、私有所有部分及 OS.Exception 和 OS.File_Manager 的公、私有部分（包括它们的包体），整个包相当于 OS 的私有部分、外界不可见，这样，为复杂的程序包分解为易于修改的子包又不泄露原有的封装性提供了极大的灵活性。

（4）其它面向对象扩充

由以上看出 Ada95 设有把包升格为类（如 C++），只是把记录类型标签化，从标签类型扩充类体系，并在结构上以子辈包配合才能实现面向对象的类体系。类体系是单继承的。多继承只能交由程序员间接实现：

例 10-9 Ada 的多继承

```

with Abstract_Sets;
package Linked_Sets is
    type Linked_Set is new Abstract_Sets with private;
    --再定义 Linked_Set 的各种操作
private
    type Cell;
    type Cell_Ptr is access Cell;

```

```

type Cell is
  record
    E: Element;
    next: Cell_Ptr;
  end record

function Copy (P: Cell_Ptr) return Cell_Ptr;
type linner is new Controlled with
  record
    The_Set: Cell_Ptr;
  end record;

procedure Adjust (Obj : in out linner);
type Linked_Set is new Abstract_Sets with --继承 Abstraet_sets
  record
    Component: linner:           --其扩展成分又继承了 Controlled
  end record;

end Linked_Sets;

```

此外, Ada95 也提供抽象基类, 它本身无实例, 其子类 (型) 才有实例。它只具有共性成分:

```

type Person is abstract tagged           --抽象类 (型) 无实例
  record
    Name: String;
    Birth: Date;
  end record;

type Man is new Person with               --特定子类
  record
    Beraded: Boolran;
  end record;

type Woman is new Person with            --特定子类
  record
    Children: Integer;
  end record;

```

10.5 Eiffel

面向对象技术发展中 Eiffel 语言可称之为—枝独秀, 它是以面向对象技术建造正确软件而设计的。Eiffel 是设计人 Bertrand Meyer 在 1988 年“面向对象软件构建”一书中提出的语言。他以抽象数据类型作为模型, 构建软件系统。比较理论化, 有严格的形式描述, 是全新的面向对象程序设计语言, 引起业界兴趣。从 1988 年到 1992 年陆续研制了多种编译版本, 在 Eiffel 3 的基础上, B. Meyer 又出了一本书“Eiffel : The Lanaguage “(1992), 即今日定型的 Eiffel 版本。1997 年 Meyer 又在“面向对象软件构建”的第二版中扩充了对并发程序的支持。

Eiffel 是强类型, 面向对象, 并发式程序设计语言。有许多新颖的思想, 我们学习它可以从另一个角度对面向对象有清晰的理解。

10.5.1 Eiffel 的对象

Eiffel 有传统语言的赋值、变量、控制（三种）、函数/过程、调用/参数匹配、类属、异常等概念和机制，它在这些概念和机制之上定义类和对象。

Eiffel 把类和类型同等看待，类是抽象数据类型的实现。抽象数据类型就是客观世界对象的规格说明，即一类相似对象的严格数学定义。Eiffel 对象分类准则是看有无相同的行为，数据只是表明对象的状态，类对象中的数据也被看作是存储对象行为的参数（输入和结果）。

类型分为两种：一为尽头（expanded）类型，这就是其它语言中的基元类型，如，Integer, Real, Character, Double, Boolean 共五种，其它均为引用（reference）类型，以这些基本的尽头类型构造的复杂类型，即抽象数据类型，也是类。

有了类型就可以声明实体（entity），相当于 C++ 中的引用变量：

P: PERSON --实体 P 引用 PERSON 类（型）。

Eiffel 中的实例对象没有名字，但可借助实体，并可在执行中生成：

!! P

双惊叹号即创建指令，创建一 PERSON 类的实例对象。

I: INTEGER --实体 I 就是尽头类型 INTEGER 的实例不用双惊叹号，就是实例对象声明。图 10-10 示出它们的区别：

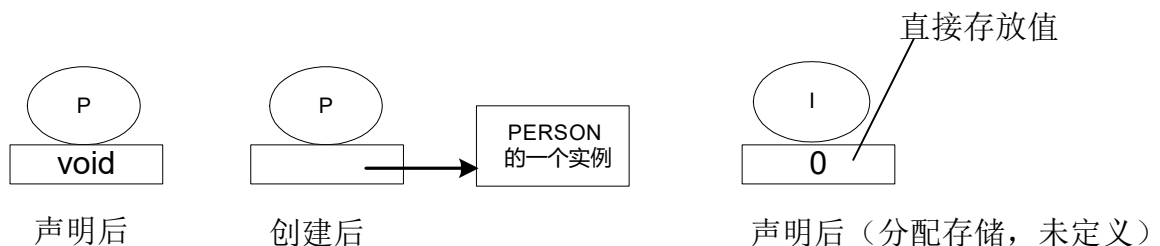


图 10-10 Eiffel 中两种实例对象

对于引用类型的实例对象一定要附着（attached）在实体上，也就是实体绑定到对象，而以尽头类型声明的实体，如同 C++，就是实例对象并为它分配存储（按类型），如果有初值还能得到定义。Eiffel 有缺省初值，数为零，字符为 '%U'（空），布尔为 false，引用类型的初值为 'void'（空引用）。

10.5.2 抽象数据类型

抽象数据类型不因语言而异，它描述的是该类型对象最本质的特征（feature），抽象数据类型的实例是以它的行为而不是表示来定义的。Eiffel 用数学语言描述 ADT，以此才易写出正确的软件。

设想一个队列 Queue，它所有可能的行为是：

| | |
|-------------|--------------|
| ADDTQ | 在队的末端加上一个数据项 |
| REMOVEFROMQ | 在队的前端去掉一个数据项 |
| FRONTQ | 决定队的前端数据项的值 |
| ISEMPTYQ | 探查队中是否已无数据项 |
| ISFULLQ | 探查队列是否满 |

LENGTHOFQ 队中当前拥有数据项数

CREATEQ 创建一个新的空队

我们再引入几个集合，并用函数映射表示它们的操作：

I 数据项集合

Q 所有队列集合

B 布尔数据集合

N 自然数集合（包括零）

则有：CREATEQ: $N \rightarrow Q$ --输入 n 得 q (n) 长的空队
 FRONTOFQ: $Q \rightarrow I$ --输入 q 得到前端 i 的值
 ADDTOQ: $Q \times I \rightarrow Q$ --将 i 加到 q 中得 q'，仍在 Q 中
 REMOVEFROMQ: $Q \rightarrow Q$ --q 去掉 i 得 q'，仍在 Q 中
 ISEMPYQ: $Q \rightarrow B$ --输入 q 查出真假值
 ISFULLQ: $Q \rightarrow B$ --输入 q 查出真假值
 LENGTHOFQ: $Q \rightarrow N$ --输入 q (n) 查出 n

这就叫操作的型构 (Signature)，再加上对每个操作的语义描述就是我们在第 17 章中还要讲述的抽象数据类型的代数规格说明。语义描述可以是公理性的，不因时间延续而改变；也可以在每个操作前后加上条件：

$\{P\} O \{Q\}$

$\{P\}$ 为前置条件，当为‘真’时执行本操作 O。 $\{Q\}$ 为后置条件，操作 O 之后应满足本条件（为‘真’）。队列的规格说明是：

name QUEUE [I]

include LISTS

sort

I, Q, B, N

signature

CREATEQ: $N \rightarrow Q$

FRONTOFQ: $Q \rightarrow I$

ADDTOQ: $Q \times I \rightarrow Q$

REMOVEFROMQ: $Q \rightarrow Q$

ISEMPYQ: $Q \rightarrow B$

ISFULLQ: $Q \rightarrow B$

LENGTHOFQ: $Q \rightarrow N$

sementics

let i \in I, q, r \in Q, n \in N, b \in B

pre_conditions

CREATEQ(n) ::= n > 0

ADDTOQ(q, i) ::= **not** ISFULL (q) --括号中逗号表示分开的输入

FRONTOFQ(q) ::= **not** ISEMPY (q)

REMOVEFROMQ ::= **not** ISEMPY (q)

ISEMPYQ ::= true

ISFULLQ ::= true

LENGTHOFQ ::= true

post_conditions

CREATEQ (n:r) ::= r = CREATELIST (n)) --括号中的分号表示前面

```

ADDTQ(q, i; r) ::= (r=APPEND (q, i))    --是输入, 后面是输出参数
FRONTQ(q, i) ::= (i=HEAD (q))
REMOVEFROMQ(q, r) ::= (r=TAIL (q))
ISEMPTYQ(q; b) ::= (b=ISEMPTYLIST (q))
ISFULLQ(q; b) ::= (b=ISFULL LIST (q))
LENGTHQ(q; n) ::= (n=LENGTH (q))
end

```

请注意, 后值条件的定义表达式中, 借用了许多已定义的表 (LIST) 的函数。这是一般的做法。由已定义的简单操作定义更为复杂的操作。

10.5.3 Eiffel 的类和程序运行

抽象数据类型是类的模型, 类是特征的集合, 类可以继承多个父类, 由 `creation` 关键字指定某个 (些) 特征是实例构造。特征即 C++ 中的方法, 其可见性由特征后 {ANY} (公有) 和 {NONE} (私有) 注明, 缺省为 {ANY}。

特征的规格如同过程语言中的过程 (指定参数及类型) 或函数 (指明返回类型) 的型构特征的体叫例程 (routine)。以下是前述队列的类:

例 10-10 Eiffel 的队列的类定义

```

class QUEUE[ITEM] inherit
    OBJECT                                --可以多个父类
[3]    creation
        make                             --只指明构造特征的名字
[5]    feature {NONE}                    --私有特征
        size: INTEGER
        head: NODE[ITEM]                --引用 QUEUE 类外的类型 NODE
[8]    tail: NODE[ITEM]
[9]    feature{ANY}                      --公有特征
[10]   make (n:INTEGER) is               --构造 n 结点实例对象, 只记 n 和当前长度
        do
            size:= n
            length:= 0                  --当前长度为 0
[14]   end--make
[15]   front: ITEM is                    --取队前端数据项
        require                          --前置条件关键字
            not is_empty                --队不空
        do
            Result:=bead.item
[20]   end--front
[21]   add (i: ITEM) is                  --在队尾处加一项
        require
[23]       not is_full
[24]   local
        new_node:NODE[ITEM]

```

```

do
[27]    !! new_node.make (i)
[28]    if head = Void then
        head := new-node
    else
        tail.Change_next (new_node)
    end
    tail := new_node
    length := length+1
[35]    ensure          --后置条件关键字
        tail=new_node
        length=Old (length) +1
[39]    end--add
[40]    remove is          --取消队头项 (结点)
    require
        not is_empty
    do
        head: = head.next
        if head = Void then
            tail := head
        end
        length := length - 1
    ensure
        --后置条件
    end--remove
[52]    is_empty:BOOLEAN is    --查队是否空
    do
        Result := (length=0)
    End--is_empty
[56]    is_full:BOOLEAN is    --查队是否满
    do
        Result: (length=size)
    End--is_full
[60]    length: INTEGER
end--class QUEUE

```

这个类有三个私有特征，[5]到[8]行，一个实例对象 size，和两个以已定义类 NODE[ITEM]声明的实体 head, tail。声明确立时如下图示：

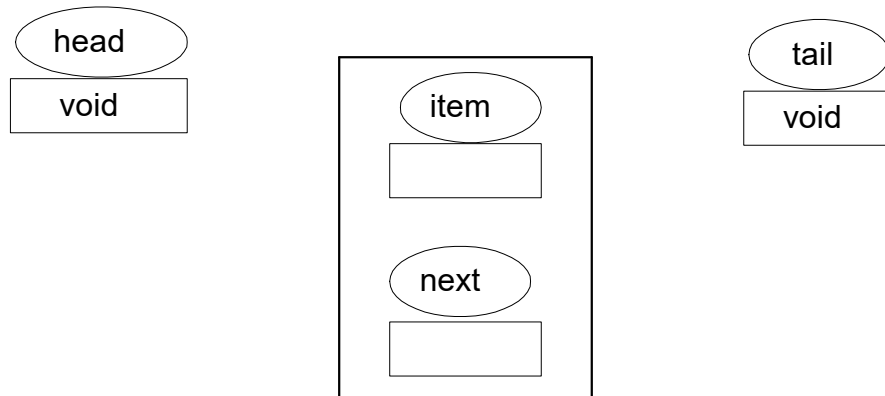


图 10-11 声明确立时对象实体示意图

head, tail 都是可能引用 NODE[ITEM] 对象的实体, 但此时未引用。从 [9] 到 [60] 行, 是公有特征共 7 个: [10] 到 [14] 行是构造子, 它只记下实例对象最多有 n 个结点, 和当前实例队的长度 (为零即没有队)。以下为本类提供的五个操作, 分别从 [15], [21], [40], [52], [56] 开始。

Eiffel 没有主程序概念, 由程序员指定一根类 (root class) 实例开始运行, 最简单的根类就是只有单个创建过程的类, 如:

```

q:QUEUE[STRING]
[63]    !! q. make (10)
[64]    from
        j := 0          --循环初值
    until
        j := 10         --循环终值
[68]    loop
        j := j+1
        io read_line
        s :=io.last_line
[72]    q.add (s)
[73]    end

```

| |
|-------|
| Zhang |
| Wang |
| Lee |
| Zhao |
| Chow |
| Huo |
| Ling |
| Wu |
| Shu |
| Bai |

这个程序声明了一个队实体 q, [63] 行附着后 q 即代表 10 元素实例对象。[64] 到 [73] 行是一个循环, [64] 到 [68] 行是循环初、终值, [68] 到 [73] 是循环体。它读入一个名单表 (如上右框所示)。每次读入一行字符串, 读后在 [72] 行向对象 q 发“把 s 串加入到队中”的消息。例如, s 此时有“Zhang”。

执行 [72] 行要找 q 的类中特征匹配, 查到第 [21] 行, i 和“Zhang”结合。至第 [23] 行前置条件 is_full 是调用特征, 找匹配至 [56] 行。因 [63] 行执行过, 此时 length = 0, 故返回 false, [23] 行为 true, 往下执行。生成新结点, [27] 行完成后结点图如下:

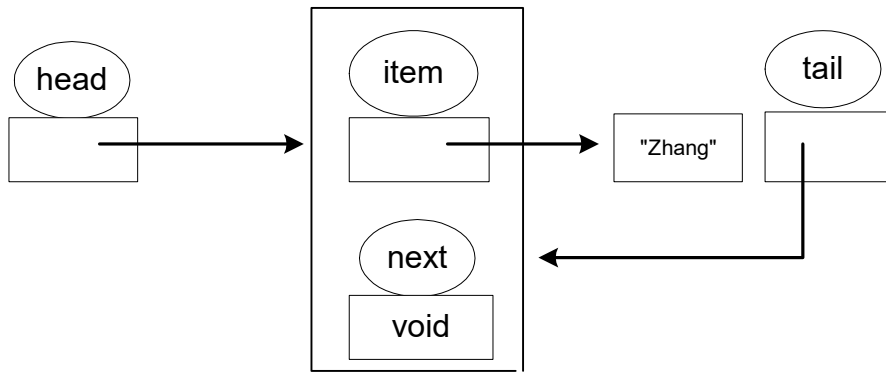


图 10-12 链表第一个结点生成

[28]和[39]行完成后 head, tail 都指向第一个生成的结点。[68]到[73]行的循环接着做下去, 就形成如下结点的链表:

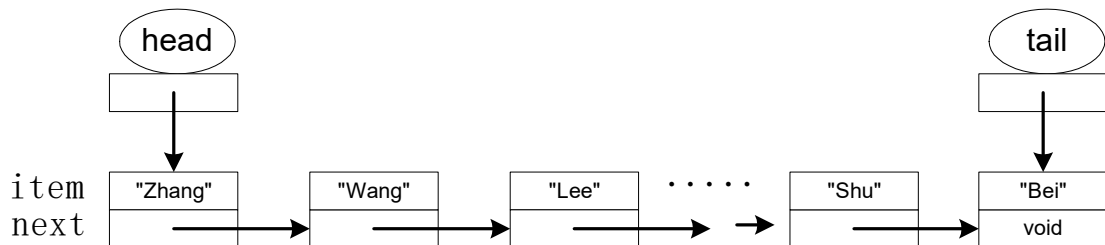


图 10-13 实例队 q 完成初装

从以上定义中可以看出 Eiffel 的特征以例程实现, 其中指令 (instruction) 如同传统语言语句, 也是顺序、条件、循环。也有局部于特征的实体, 如[24]行。第[5]到第[8]行的私有实体, 为本类所有特征共享。第[60]行是个特征, 它返回实体 length 的值。每一特征的前、后置条件以 require_ensure 插入到例程 (特征的体) 中。可以省略, 因为它要增加运行开销, 只在必要时按 ADT 规格全部加上。除此之外在循环指令和类中可以加上不变式, 如:

例 10-11 查找数组 a (索引从 1 到 size) 中, v 值第一次出现的位置。

```

from
    position := 0
    s := a.item(1)
invariant                --不变式
    0 <= position and position <= size
variant                  --变式
    size - position
until
    position = size or else v = a.item(position)
loop
    position := position + 1
end

```

类中的不变式加到末端 **end** 之前, 没有 **variant** 关键字。require, ensave, invariant, variant。这 4 个关键字引出的子句是 Eiffel 的断言。用户还可以用 **check_end** 子句在例程的任何地方插入断言, 如:

check

j >= 1 **end** j <= length

end

v := a.item(j)

Eiffel 的类还有一种写法是只写界面：

class interface CLASSNAME

--其余只写特征的型构没有体例程

--相当于 Ada 的没有体的规格说明

10.5.4 Eiffel 的继承与多态

Eiffel 是多继承的，它有非常丰富的继承机制，共六种：

- 被继承特征可换名
- 被继承特征可改变可见性
- 被继承特征可重新定义行为
- 可以使被继承特征的行为延迟发生作用
- 选择继承
- 被继承特征不作定义

其书写格式如下示例：

class HEIR **inherit**

PARENT1:

--无适配子句全部继承此父类特征

PARENT2

rename

f1 **as** parent2_f1,

--将 PARENT2 中的 f1 换成后者

f2 **as** parent2_f2

export

--改变输出（外部可见）状态

{ANY} f3, parent2_f1

--PARENT2 中 f3, f1 在此类中公有

{NONE} f4, parent2_f2

--PARENT2 中 f4, f2 在此类中私有

{AClass}f5

--只许 AClass 使用

redefine

--内容有了修改的关键字

parent2_f1, f3

undefine

--被覆盖的关键字

f4, parent2_f2

select

--在同名特征中先定继承路径

end

--继承来的 PARENT2 使用 5 种适配子句

PARENT3

rename

g1 **as** parent3_g1

end

--只选一种适配子句表示有了修改

creation

--本类构造子

feature

--本类定义的特征

end--class HEIR

下面我们分别解释：

(1) 换名 (rename)

只改变名字 (表示) 不改变特征原有行为 (内容)。好处是清晰, 且减少不必要的动态分辨, 更重要的是可以实现多继承中的重复继承。例如, 房子的租金是按 (单位租金×面积) 算出。一个家庭事务所, 既是住宅又是事务所, 其继承图如图 10-14。两条路径都是继承 ‘租金’。

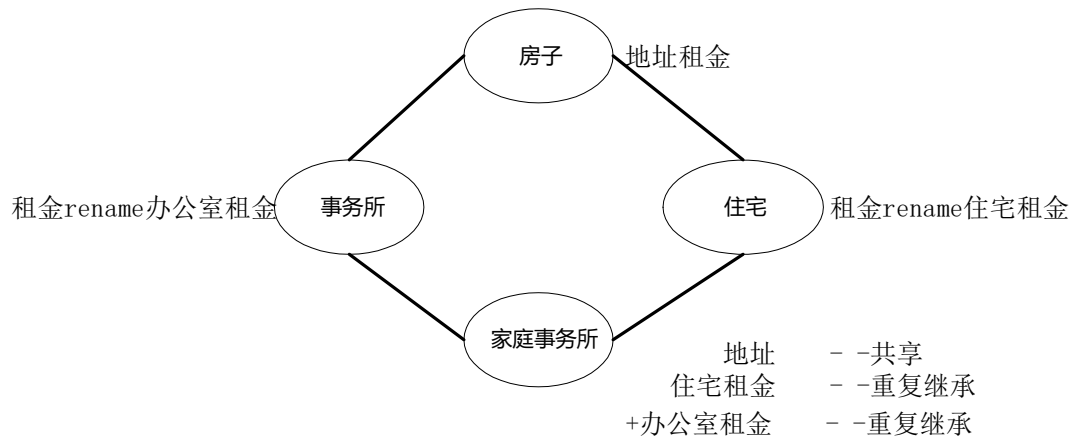


图 10-14 重复继承

由于换了名, 两者都继承, 叫重复继承。

(2) 改变可见性 (export)

不像 C++ 可见性的继承有死规定, Eiffel 由程序员灵活控制。例如, 一个平面图形类, 有 “display” 特征, 它只显示图形轮廓; 彩色平面图类, 继承平面图形类。显然, 它有自己的 “display”, 而继承过来的黑白线条就没有意义了, 无需公有 {ANY} 调用, 该特征则改为私有 {NONE}。先画轮廓, 上色后显示。

(3) 重新定义行为 (redefine)

特征名不改, 修改例程。这就是多态性中的静态覆盖 (overriding)。如, 矩形是多边形的子类, 求面积所用公式是不一样的, 继承下来的名字一样 (Area), 例程重写。

(4) 延迟 (deferred)

是一种支持抽象设计和动态束定的机制。因为 OO 设计时是概括抽象, 即子类实例就是父类实例, 抽象设计中把所有共性特征放在父类。例如, 图形类的求面积求周长都是在平面坐标内积分计算, 一旦子类有了确切形状这个特征才能实行。故程延迟, 相当于 C++ 的虚函数。Eiffel 中, 凡有延迟特征类均为延迟类。延迟类即抽象基类, 没有自己的实例, 子类的实例都是它的实例。

例 10-12 车辆注册的延迟类

deferred class VEHICLE_REG

--无创建实例对象指令

feature

register:LIST[VEHICLE]

reregister_veh

deferred

--延迟特征、子类中重定义

```
--其它访问注册表的例程
end--VEHICLE_REG
```

| | |
|--|--|
| <pre>Class TRUCK_REG inherit VEHICLE_REG Redefine register Feature Register: LIST[TRUCK] reister_veh --重定义 do --卡车注册的各动作 end --其它访问本注册表的例程 end -TRUCK_REG</pre> | <pre>class CAR_REG inherit VEHICLE_REG redeine register feature register: LIST [CAR] register_veh --重定义 do --轿车注册的各动作 end -- 其它访问本注册表的例程 end -CAR_REG</pre> |
|--|--|

这两个类可以有实例构造子,但本题作为引用类在例 10-13 中组成复合构造子,此处略。

(5) 选择 (select)

因为 Eiffel 允许重复继承,在有换名的多继承中,同一特征有可能继承多次,为此,设选择继承机制:

```
class EMPLOYEE inherit
    PERSON
        rename                --虽然在本类中已经换了名
            display as person_display
        end;
    PERSON                --指示动态束定时,继承的是:
        redefine
            display        --重定义的确良 display 版本
        select
            display        --被选用
        end
    ...
end--class EMPLOYEE
```

(6) 无效继承 (undefine)

当父类是多继承时,若继承两个非常相近的特征,子类只想继承一个的使另一个无效。不必用一个同名的空特征使之无效,直接指出就可以了。这是为了清晰:

```
class INHER inherit
    PARENT1
        undefine    feature2
    end
end--INHER
```

从以上六种机制看来,由于引入多继承和重复继承,继承机制比较复杂,换名增加了名字空间复杂性,它虽然为重复继承不得已而为之,但增加了程序正文的清晰性。静态的多态性其分辨方法和其它语言类似。类属多态和例化与 Ada 同,且只限于参数多态。我们再看 Eiffel 的动态束定。

10.5.5 指定动态类型和动态束定

Eiffel 是强类型语言，前述的延迟类（deferred）显式告诉编译，运行时再束定。父类的某个特征到子类继承后重定义，该特征是一个名字，指出它是动态类型后在运行中分辨。

（1）指定动态类型

设有继承关系如图 10-15，有声明

```
plist: FIXED_LIST [PERSON]
P1, P2: PERSON
e1, e2: EMPLOYEE
s1, s2: STUDENT
t1, t2: TUTOR
```

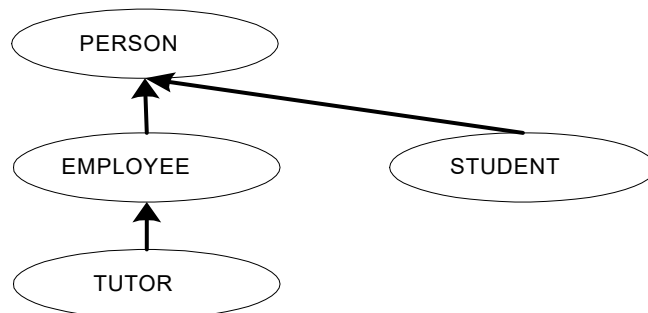


图 10-15 一个继承关系

一种办法是用类属类型作动态使用不必设例

--先将 8 个对象读入 plist 表

```
!! plist. make (p1, p2, e1, e2, s1, s2, t1, t2)
```

--建立了一个混合表的对象 plist，子类型相符可入表

```
from --循环依次打印各对象名字
```

```
    plist.start
```

```
until
```

```
    plist.exhausted
```

```
loop
```

--动态类型，相当于 Ada 的类宽类型

```
    io.putstring(plist.item.name) --多态函数，动态束定
```

```
    plist.forth
```

```
end
```

其中 item 在 FIXED_LIST[PERSON] 中的定义中有

```
feature
```

```
    item:PERSON
```

第二种办法是创建相符类型对象，创建中即借用了动态类型概念，以父类型定义子类型。

若有：

```
P:PERSON
```

```
!EMPLOYEE! p.make(...)
```

p 的静态类型仍然是 PERSON，用它创建了一个 EMPLOYEE 类型的对象，运行中执行的是：

```
e:EMPLOYEE
```

```
!!e.MAKE(...) --两个 make 是不同的特征
```

(2) 动态束定

如上段声明, Eiffel 中可以写指令:

```
plist.item.display
```

其中 display 在各子类(型)中定义同名, 但例程不一样。执行时可按 item 当前类(型)找出它的 display 特征执行, 即动态束定。

最后我们再看一个动态束定的例子, 附带介绍 Eiffel 的其它几个机制。

例 10-13 车辆登记人机界面程序

设如例 10-12 已定义了一个延迟类 VEHICLE_REG 和两个子类 CAR_REG 和 TRUCK_REG, 则有:

```
class VEHICLE_HCI creation
    make
feature {NONE}
    veg:VEHICLE_REG
    creg:CAR_REG
    treg:TRUCK_REG
    Cartype, Tucktype:INTEGER is unique --整数代表枚举值, 只许渐增
    answer:INTEGER
feature {ANY}
    make
        --创建 VEHICLE——HCI 连同 VEHICLE——REG 对象
    do
        ...
    end--make
    get_and_execute
    do
        --找出用户的车辆类型并引用之
    inspect --相当于 case 语句
        answer
        when Cartype then
            veg := creg --向相容实体赋值, veg 动态类型
        when Truck type then
            veg := treg
        else
            --输出出错
        end
        vreg.register_veh --动态束定例程
    end--get_and_execute
end--class VEHICLE_HCI
```

10.6 小结

•“面向对象”始于 Small talk 语言, 它是早期抽象、封装及抽象数据类型研究的新成果, 早期实用系统是 Simula 67。

• Small talk 系统由语言核心(定义语法语义)、程序设计系统(支持设计的类库)、程序设计范型(有别于传统程序设计、动作的新方式)、用户界面模型(提供实施新范型的用户界

面)组成。

- Small talk-80 是有类、无类型的基于表达式的小语言。它的表达式是选择子带参数向接受子发消息。它是编译解释型，工作空间写的每个类协议经编译为中间代码，由 Small talk 虚拟机解释执行，执行中动态束定每个发向实例对象的消息。

- Small talk 虚拟机管理三种文件：源文件、变更文件、映象文件，虚拟机负责对象的存储管理；虚拟机提供最低层与操作系统联系的支持基本例程。基本例程由汇编语言编写。

- Small talk 程序设计主要是写类协议和使用表达式。类协议一为完全自定义；一为在原有协议下派生（支持重用）；一为直接使用类库中已定义好的类（完全重用）。

- “Small talk 的支持编程的类、工具性的类、用户保留的有用类均统一组织在单继承的类库中。且随用户积累可逐步增大。

- Small talk 用字符串和指针实现对象间的语义联系。静态编译后的消息字典供动态匹配使用。

- 面向对象的基本特征是：封装、抽象、继承、多态、动态束定。只有这五个特征全具备才是面向对象程序设计语言。

- 封装和继承带来可见性继承问题。原则是不因为类型扩展改变原定义类的可见性。各语言为此设的机制不同，C++较死，Eiffel 较活，Ada 居中。

- 对象构造子在继承体系中也要继承。为此，对象初始值也必须给出，其初始化次序是先基后派，析构时其撤销次序是先派后基。

- 强类型语言编译后即要束定，实现动态束定的比较麻烦，解决的办法是留出束定表，编译时只作缺省束定，每当新派生一类型即往其中注册（常指针指向该例化类型的地址），使用该操作时按结合类型特征值派送。显然，派送表要留到运行时。Ada 是标签值，C++是子类（型）名，Eiffel 是实体名。为此，C++设虚函数机制。

- Ada 95 以抽象数据类型实现类，类的封装由包实现，为此扩充了子库单元和私有子库单元的概念。类完全由记录类型加标签表示，沿用 Ada 83 的派生类型概念生成子类。扩充的抽象类型和标签类型均是记录类型加关键字。

- Ada 95 的对象构造子如同类型声明变量，但一定要加初值表达式（即 Ada 的值构造子）。

- Ada 95 扩充了类宽类型实现多态和动态束定，因类宽类型出现而引出的新一轮派送为再派送（以此为根限定子辈单元可见域）。

- Ada 95 是单继承系统，多继承通过扩展成份间接实现。

- Eiffel 提供了实体概念，实体即引用的常指针，以实体作对象的静态描述，运行时通过构造子（!!）生成实例对象完成束定，动态束定极其方便。

- Eiffel 以抽象数据类型为模型定义类，每个操作都有严格的形式定义（型构），其行为以公理或前/后置条件描述。前置条件对应到语言是 require 引出的断言，后置条件是以 ensure 引出的断言，断言以临时变量代入前面已证明过的操作混以逻辑运算符表达。

- Eiffel 的操作叫特征，构造对象的特征由 creation 关键字指定，它只指定实体和对象体的关系，还需构造子（!!）生成活的对象。

- Eiffel 没有数据的具体概念，属性对象只是存储对象特征的（输入/出）值。

- Eiffel 可以为循环和类列出不变式，不变式用断言表达，当类中无例程（函数/过程）组成时，为限定属性取值范围就要用类不变式限定。

- Eiffel 是多继承的，有六种继承机制：原特征到子类中可换名；继承后可改变可见性；继承的特征行为（例程）可以重定义；支持重复继承，为避免用一特征重复继承设选择继承；同名的特征覆盖等。给程序员以较大灵活性。

习题

10.1 试述面向对象五大特征满足了软件工程对软件要求的何种特性。

答：面向对象的五大特征是封装、抽象、继承、多态和动态束定。

面向对象的五大特征满足了软件工程的局部性、概括表达、可重用、易扩充的要求。对象的封装与抽象满足了对数据隐藏和局部性的要求、比过程/函数更抽象成为对象，继承为构件重用大开方便之门。基于继承的重用则要求函数类型是多态的，否则继承效用有限，多态还有利于程序扩充。

10.2 试述虚函数和重载函数的同异。

答：虚函数和重载函数都是同名函数。重载函数通过函数型构中的参数个数、类型、次序进行匹配，静态束定函数的机制，虚函数是根据实际赋予的对象类型动态束定此特定对象的方法，它建立一个虚函数表记下本函数在各子类中的位移，是一种根据对象束定方法的机制。

10.3 对象是封装的数据和操作

- [1] 单封装数据
- [2] 单封装操作
- [3] 一个函数
- [4] 一个关系 $R(a, b)$

能不能作为一个对象？为什么？

答：单封装数据：是对象，因为满足面向对象的五大特征，只不过封装和隐藏的仅是数据而已。单封装操作：是对象，因为它也满足面向对象的五大特征。一个函数：不是对象，因为它不能被继承、重用、和抽象、动态束定，既不能满足面向对象的五大特征。一个关系 $R(a, b)$ ：也不是对象，因为它也不满足面向对象的五大特征。

10.4 试比较 Small talk、C++的继承机制。各有什么优缺点。

10.5 解释以下 Small talk 程序片断的语义

```
| array |
array ← MonitoredArray new: 20
1 to 10 do: [: i | 1 to 10 do:
    [: j | array at: i | j ]
↑ array access Counts
```

10.6 解释以下 Small talk 程序

```
| input answer f c |
input ← File pathName: 'chapter, 7'.
answer ← writeStream on: String new.
f ← Bag new
[input atEnd]
    whileFalse: [ (c ← inputnext) isLetter
        ifTrue [ f add: c aslowerCase ] ]
0 to: 25 do: [: i | c ← ($a asciiValue + i) ascharacter
    answer
        cr ; nextPut: c ; space;
        nextputAll: (f occurrencesOf: c) printString].
↑ answer contents
```

10.7 试比较第 9 章类型对象 T 形图和本章介绍的对象一类存储之间异

答：相同：都是体现存储信息的信息结构，具有名字、名字的定义及信息体都通过束定建立名字和存储对象之间的联系。

不同：T 型图的类型并不预分配存储空间，并且分配和束定的都是该类型的具体类型对

象，类型对象由它的类型来决定分配的存储空间。而类一对象存储是编译时将类对象分配存储并填上中间代码，实例对象只存放数据，利用指针指向自己的所属分类，共享所属分类中的方法和消息。

10.8 上机验证 10.3[1]C++访问权限，打出各种结果的报告。

10.9 完成例 10-5 打印职工表程序，输入 10-20 雇员其中经理 2-3 名。打出结果报告。

10.10 C++中构造函数有没有虚函数？析构呢，为什么？

答：构造函数没有虚函数，因为没有意义。析构函数必须是虚函数，因为 C++为了解决强类型的动态束定问题，即在运行中根据不同的（子）类对象束定不同的函数上，虚函数定义在类型域上，各子域都可以用虚函数，它是动态实现的，以实现多态性。

10.11 Smalltalk 变量没有类型计算为什么不出错？

10.12 试述 C++之友员 friend 设施之优缺点。友员和嵌套子类的同异。

10.13 试述 10.4 (3) 中 Supersonic_Reservation_System 和 Reservation_System, Supersonic 两个程序包的同异。

10.14 总结 Ada 95 库单元、子单元、子辈单元、私有子辈单元的可见性规则，查看 Ada 95 手册，你的总结是否全面。

10.15 总结 Ada 95 程序包和类体系结构的关系，你能提出这种相互配合不好能产生什么问题吗？

答：①Ada95 以抽象数据类型实现类。类的封装性由包实现，类的继承性则扩充了标签（tag）类型和抽象类型。标签类型只限记录类型。类的继承性还扩充了 Ada83 的类型派生机制以实现子类。

②以类宽类型实现多态。

③以扩充程序包机制实现继承的类体系。

这种相互配合不好会导致运行中动态束定的混乱，并且与子辈包配合不好就不能实现面向对象的类体系中的继承。

10.16 Ada 95 抽象基类本身能否为实际类型的子类、父类？为什么？

答：不能为实际类型的子类。因为 Ada95 类的封装性由包实现，类的继承性是由子辈单元来描述，子辈包是父辈包的扩展，在父单元声明作用域中，必须可以见到父单元的所有部分，包括私有部分，实际类型是不可见父辈的私有细节的，因此抽象基类本能不能作为实际类型的子类。

10.17 试评述 Eiffel 的 expended 类型和 referance 类型。

10.18 抽象数据类型规格说明分哪几部分，各部分的作用是什么？

10.19 试比较 C++和 Eiffel 的继承机制，评价它们的优缺点？

10.20 假定有以下 Eiffel 的类定义：

```
class A
feature {NONE}
    a1: Type_1      --属性
feature {ANY}
    a2:Type_2      --属性
    ra (t: Integer) --例程
end A
```

设若要建立一 B 类，它输出属性 b1，例程 rb。试绘一示意图表示 B 类对象的域连同与其结合的输出例程，且当：

[1] 利用 A 类中隐藏的属性定义 B 类。

[2] 以 A 类作为 B 类的父类。

第11章 函数式程序设计语言

过程式程序设计语言由于数据的名值分离，变量的时空特性导致程序难于查错、难于修改。从60年代初对程序本质的研究，直至80年代软件工程对软件本质的研究，软件表达能力，软件规模，运行效率都大为改善(前九章介绍的各种特征，读者应有初步印象)，但命令式语言天生带来的三个问题只解决了一半：

- 滥用goto，通过结构化程序设计和限制少量可控制的跳转，已经完全解决。
- 悬挂指针，尽可能多用引用和在堆栈框架中分配指针，可消除大量悬挂指针。但引用的对象依然有时、空问题。没有完全解决。
- 函数副作用，由于冯·诺依曼机的本质是改变变量的存储从而改变程序状态，但是状态时空效应不可免，副作用不可能消除。

70年代研究正确性的学者得出结论：传统的程序设计语言之所以难于正确带来调试大量问题是两个方面：一为写不出正确的规格说明：在一个程序开发之前如何准确地指出程序怎样才算“正确”，于是开展了形式方法的研究。另一个是过程式程序存在的先天问题难于用数学模型，用术语说是程序状态的易变性(Mutability)和顺序性(Sequencing)。1977年J. Backus在图灵奖的一篇演说：《程序设计能从冯·诺依曼风格下解放出来吗？》(下称《解放》)，极力鼓吹发展与数学连系更密切的函数式程序设计语言。经过80年代，非冯范型语言得到巨大的发展，正如第0章介绍过的函数式、逻辑式、数据流、关系式、综合型，虽然它们还存在着各种不同的问题，暂时还无法代替命令式过程语言，但对程序设计语言的研究提供了不少有益的启示。在其所长之处，传统过程式语言难于与它们匹敌。例如，人工智能专家系统。所以，也不失为程序设计语言百花园中的几支奇葩。

本章以对比方式介绍函数式语言的一般语言特征，以及设计的主要思想，如何解决过程式语言的不足，而又带来什么不足，函数式语言的模型是 λ 演算。本章也作初步介绍。

11.1 过程式语言存在的问题

程序设计语言总得在机器上运行实现，J. Backus的文章从根本上动摇了冯·诺依曼的硬件体系，在《解放》一文中指出，存储器和CPU之间地址传送强行赋值是“冯·诺依曼瓶颈”。程序以数据存储的状态表达计算。变量的时空特性决定状态是易变的。瓶颈传送必须有顺序，按照这个模型的高级程序设计语言是赋值和对顺序的控制。如前所述，过程式语言三种语句就够了(赋值，简单if加goto)。顺序就是时序，有时间性。

(1) 易变性难于数学模型

数学本质上是符号的指称系统，符号一旦在它的上下文中得到定义，它指称的是同一事物。数学靠对真假值的陈述来证明某个事情的正确性。一旦陈述不随时间改变。我们说过，代数中的变量是未知的确定值，而程序设计语言的变量是对存储的抽象。它的值可以由程序的任何部分发出命令修改。早期的程序正确性证明是对非封装模块语句集进行的。这样就避免不了错误的命令改变某个变量的内容，封装和模块性大大地缓解了出错的可能。但在模块内，有变

量这个“改变值的随意性”依然存在(总有错误的语句)，根本解决：能不能不要程序意义的“变量”只保留数学意义的“变量”？是函数式程序设计语言的主要问题。

程序总是一种计算，计算就一定要改变值。无论是通过运算符，还是读/写都可以把它们看作是改变值的函数(本书一再表明这个观点)。函数映射将自变量域(domain)上的对象映射为函数返回值域(range)上的对象。一旦定义，每次同样自变量就能返回同样的返回值，不因时因地有所不同，数学上这是根本。如果 $\sin(30^\circ)=0.5$ 这种函数应用出了别的结果值，数学证明就无法进行了。然而，在程序设计语言中可轻而易举写出同样自变量调用得出不同结果值的函数。

例11-1 有副作用的函数

```
int sf_fun(int x)
```

```
    static int z = 0;        //第一次装入赋初值
    return x + (z++);
```

`sf_fun(3) = {3 | 4 | 5 | 6 | 7 ...}` //随调用次数而异。不是数学意义的确定函数。

由此，程序必须消除函数的副作用，不管有无全局量和局部静态量。

(2) 顺序性更难数学模型

数学家建立数学理论时，先定义符号，然后写这些符号之间关系的事实，这些事实的先后次序并不重要，只要这些符号使用时有定义。证明只是验证这些事实是不是，而不用这些事实去做什么。证明过程极其重要，但反映不到它的表示上，只能表示结果，程序恰巧与此相反，写下程序正文是描述的一组动作的顺序，而不是写下变量关系的事实。即使从顺序的角度讲它也不能像议事日程表一样准确，几点几点干什么。例如，一个求值语句在循环域内时，求值次数往往依上下文而定。如果真想描述程序执行的全过程，单是程序代码是不够的，还要有堆栈的拷贝，全局量、静态量、寄存器变量，程序计数器的当前值，以及输入的拷贝。最重要的还是顺序性影响计算结果，例如，前述急求值、正规求值、懒求值同一表达式就会有不同的结果。没有副作用还好处理一点，有副作用更甚，因而难于为程序建立统一的符号数学理论。

由此，应寻求与求值顺序无关的表达方式，才能在程序设计代码完成后，不用上机，从正文中就可验证或证明其正确性。

(3) 理想的改变途径

说到底命令式语言是因为有依赖于时空的变量。程序刻划的是以时间为因素的程序状态。计算和程序状态改变成了两件事情，赋值是改变下一次计算的环境。如果没有变量，就没有破坏性赋值，也不会有引起副作用的全局量和局部量之分。调用通过引用就没有意义。循环也没有意义，因为只有每次执行循环改变了控制变量的值，循环才能得到不同的结果。实际上语句都可以不要。因为语句(命令)是改变机器状态而不是返回值的。但程序总得计算改变值，所以还需保留“变量”以返回值，但此时变量意义不同于命令式语言的。

没有程序意义的变量，没有语句程序结构还留下什么呢？那么只剩下表达式，条件表达式，递归表达式。这三种机制对于一个程序设计语言够吗？基本够，但还差一点。因为输入/输出无论如何也是在改变环境，无论如何也要依赖文件变量且有副作用。好在它的副作用只在端点上，不影响程序和程序设计本身，只是最初进来最后出去，我们暂且放下它们。

至于留下的三个机制按照程序设计语言学家的研究它是足够的！因为程序总要改变值，有规则地改变值程序才是可证明的。如果用函数映射的办法，写得出函数就是找得出规则。运算符是函数，它有严格的规则，组合到表达式中也是有规则的。逻辑上是通的，易于证明正确。有破坏性赋值的命令语言程序在大逻辑框架中是有规则的，如条件、循环控制，但每个赋

值语句无章可循。恰巧出错都在这些上面。以表达式和嵌套表达式代替语句组就避免了赋值的武断性。内嵌套表达式的结果值又作外套表达式变元，即数据一直在有规则的表达式之间传递，不涉及具体环境和实现，这就是所谓引用透明性原理。虽然仍在同一环境的运算器、存储器上实现，只借用它的计算和存放中间值和结果值的功能。所以可以从数学的指称意义上控制整个程序。因而能准确地写出语义(几乎完全脱离机器，当然好写!)。

数据结构当然要有，但数据结构的改变靠重新映射，复杂的数据结构当然需要同一操作的迭代。递归函数就可胜任对这些数据结构操作。至于条件表达式则可完全替代条件语句，使程序能有选择地作计算，它也是必要的逻辑控制。

λ 演算是符号的逻辑演算系统，它正好只有这三种机制，它就成为函数式程序设计语言的模型，1957年发明的LISP语言就以 λ 演算为模型。可惜LISP不够“纯”函数，由于输入/输出要向表达式传递值，LISP又增加了PROG机制，即命令式语言的变量、赋值甚至于标号和goto语句等过程式语言的特点，但纯LISP部分一直被认为是它体现函数式语言的特点。为了理解这些特点我们还是先介绍它的模型： λ 演算。

11.2 λ 演算

λ 演算是一个符号、逻辑系统，其公式就是符号串并按逻辑规则操纵，它提供无goto和赋值的语义元语最小集。Church的理论证明， λ 演算是个完备的系统，可以表示任何计算函数，所以任何可用 λ 演算仿真实现的语言也是完备的。 λ 演算本身并不是可执行程序设计语言，但它简单、数学表达清晰，将它用于研究基础概念和特征是非常有用的。是多数近代函数式语言的语义基础。

λ 演算使函数概念形式化，是涉及变量、函数、函数组合规则的演算。其目标为可计算性理论建立模型，20年代和30年代Schonfinckel, Curry, Rosser, Kleene和Church等人做了大量奠基性的工作。

λ 演算基于最简单的定义函数的思想：一为函数抽象 $\lambda x. E$ ，由 λ 说明的 x 在函数体 E 中出现均为形参变元。 E 是一个 λ 表达式。一为函数应用 $(\lambda x. E)(a)$ ，即 E 中的 x 均由 a 置换变成 $E(a)$ 。

这两个基本思想贯穿于符号系统的 λ 演算之中，居然可描述一切可计算函数，在这里还要说明，在 λ 演算中：

一切变量、标识符、表达式都是函数或(复合)高阶函数。如 $\lambda x. C$ (C 为常量) 是常函数。

11.2.1 术语和表示法

(1) λ 演算有两类符号：

- 小写单字符用以命名参数，也叫变量（数学含义的）。
- 外加四个符号 $'(, ')$, $'.'$, $'\lambda'$ 。

由它们组成符号串叫 λ 表达式，核心 λ 演算表达式是非常冗长的(见下文)。为了清晰和简炼，再加上一类符号：

- 大写单字符、特殊字符(如+、-、*、/)、大小写组成的标识符代替一个 λ 表达式。

(2) 公式

符号组成的串。其组成规则如下：

- 变量是公式，如 y 。
- 如 y 是变量 F 是公式，则 $\lambda y.F$ 也是公式。
- 如 F 和 G 都是公式，则 (FG) 也是公式。

(3) λ 表达式

上述三种公式及其复合均称 λ 表达式。其中：

• 形如 $\lambda y.F$ 为 λ 函数表达式，以关键字 λ 开始，变量 y 为参数，也叫约束变量。公式 F 为函数体，' $\lambda y.$ '指明 F 中所有的 y 均为形式参数。

- 形如 $(F G)$ 为 λ 应用表达式，也称组合表达式、运算符_操作数表达式。
- 为了清晰， λ 表达式可以任加成对括号。

以下表达式有相同的语义：

(fa) , $f(a)$, $(f)a$, $(f)(a)$, $((f)(a))$

例11-2 λ 演算公式举例

| | |
|---|-------------------|
| x | 变量、公式、表达式。 |
| $(\lambda x. ((y)x))$ | 函数，体内嵌入应用。 |
| $(\lambda z. (y(\lambda z. x)))$ | 函数，体内嵌入应用，再次嵌入函数。 |
| $((\lambda z. (z y))x)$ | 应用表达式。 |
| $\lambda x. \lambda y. \lambda z. (x \lambda x. (u v)w))$ | 复杂表达式 |

(4) 简略表示

- 缩写与变形表达 下例各表达均等效：

$$\begin{aligned}\lambda a. \lambda b. \lambda c. \lambda z. E &= \lambda abcz. E \\ &= \lambda (abcz). E \\ &= \lambda (a, b, c, z). E \\ &= \lambda a. (\lambda b. (\lambda c. (\lambda z. E)))\end{aligned}$$

- 命名 以大写单字符或标识符命名其 λ 表达式：

$$\begin{aligned}G &= (\lambda x. (y(yx))) \\ ((\lambda x. (y(yx))) (\lambda x. (y(yx)))) &= (G G) = H\end{aligned}$$

由于 λ 演算中一切语义概念均用 λ 表达式表达。为了清晰采用命名替换使之更易读。

$$\begin{aligned}T &= \lambda x. \lambda y. x && // \text{逻辑真值} \\ F &= \lambda x. \lambda y. y && // \text{逻辑假值} \\ 1 &= \lambda x. \lambda y. x y && // \text{数1} \\ 2 &= \lambda x. \lambda y. x(x y) && // \text{数2} \\ \text{zerop} &= \lambda n. n(\lambda x. F)T && // \text{判零函数}\end{aligned}$$

zerop 中的 F 、 T 可以用 λ 表达式展开，其它用法同。

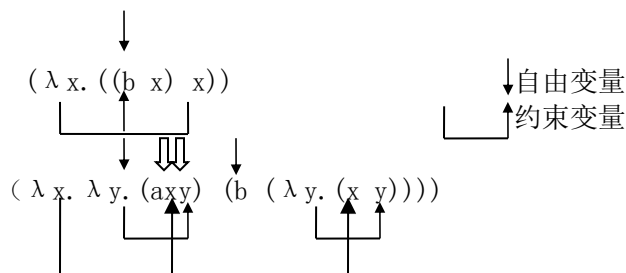
(5) 形式语法

λ 演算可以看作是描述函数的表达式语言。由于核心的 λ 演算没有类型，没有顺序控制等概念，程序和数据没有区分。语法极简单：

$$\begin{aligned}\langle \lambda\text{-表达式} \rangle &::= \langle \text{变量} \rangle \\ &\quad | \lambda \langle \text{变量} \rangle. \langle \lambda\text{-表达式} \rangle \\ &\quad | (\langle \lambda\text{-表达式} \rangle \langle \lambda\text{-表达式} \rangle) \\ &\quad | (\langle \lambda\text{-表达式} \rangle) \\ \langle \text{变量} \rangle &::= \langle \text{字母} \rangle\end{aligned}$$

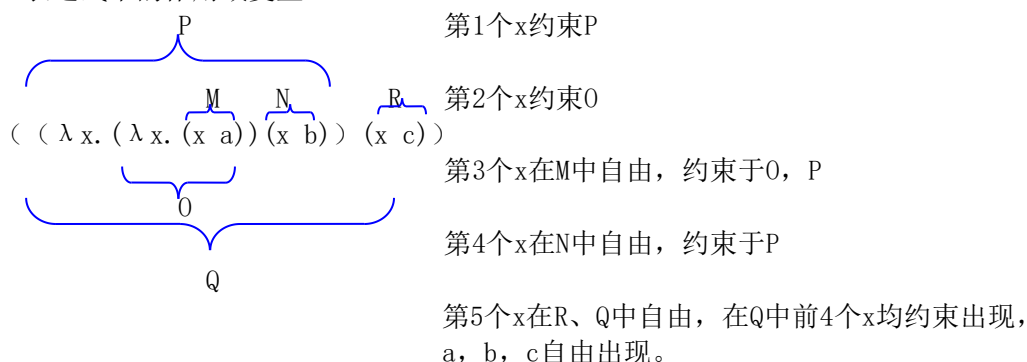
11.2.2 约束变量和自由变量

λ 表达式中参数名是局部名，它约束函数体中出现的所有名字，其约束作用(域自该参数‘.’之后至整个函数体。未约束的变量名为自由变量。



第一个 λy 的作用域是整个函数体，但第二个 λy 与之重名，有作用域复盖。作用域概念与第5.4节同。

例11-3 λ 表达式中的作用域复盖



11.2.3 基本函数

用 λ 演算表示计算首先要表示域，由于 λ 演算系统的值也是公式，则每个域中先定义最基本几个公式，其它值通过演算实现。这样，表示计算的问题全集中在 λ 演算上了。

• TRUE 和FALSE的 λ 表达式

$T = \lambda x. \lambda y. x$ T是参数为x, y的“真”函数，体中无y返回x。

$F = \lambda x. \lambda y. y$ F是参数为x, y的“假”函数，体中无x返回y。

• 整数的 λ 表达式:

$0 = \lambda x. \lambda y. y$ 与F的 λ 表达式相同

$1 = \lambda x. \lambda y. x y$

$2 = \lambda x. \lambda y. x (x y)$

$n = \lambda x. \lambda y. x (\underbrace{x (\dots (x y) \dots)}_{n \text{ times}})$

n个

• 基本操作函数

有了表示数的基本函数就可以构造表示操作的基本函数。

以下是逻辑操作函数：

```
not = λ z. ((z F) T) = λ z. ((z λ x. λ y. y) (λ x. λ y. x))
and = λ a. λ b. ((a b) F) = λ a. λ b. ((a b) λ x. λ y. y)
or  = λ a. λ b. ((a T) b) = λ a. λ b. ((a λ x. λ y. x) b)
```

以下是算术操作函数举例：

```
+ = add = λ x. λ y. λ a. λ b. ((z a) (y a) b)
* = multiply = λ x. λ y. λ a. ((x (y a)))
** = sqr = λ x. λ y. (y z)
identity = λ x. x //同一函数
succ = λ n. (λ x. λ y. n x (x y)) //后继函数
zerop = λ n. n (λ x. F) T = λ n. n (λ z. λ x. λ y. y) (λ x. λ y. y) //判零函数
```

此外，positive(取正整数)、neg(取负)、divide(除法)、subtract(-、减法)，pred(前驱)等函数也可以写成λ表达式形式，最底层的计算都是以λ表达式形式表示的。

核心的λ演算只有单目运算，例如：add 1它返回的结果是函数，再应用到后面的表达式上。这样，3+4就写add 3 4，add 3返回“加3函数”应用到4上当然就是7。写全了是：

```
(λ x. λ y. λ a. λ b. ((x a) (y a) b)) (λ p. λ q. (p p (p q))) (λ s. λ t. (s s s (s t)))
≡ λ a. λ b. (a (a (a (a (a (a b))))))
```

所以，为了清晰我们用高层数据名和函数名。

11.2.4 归约与范式

所谓演算就是归约。λ函数的语义是它应用到变元上所实现的数学函数计算。从符号演算的角度应用就是符号表达式的归约(reduction)，归约将复杂的表达式化成简单形式，即按一定的规则对符号表达式进行置换，先看一例，相互归约的符号标以下横线。

例11-4 归约数1的后继

```
(succ 1) => (λ n. (λ x. λ y. n x (x y))) 1 //写出succ的λ表达式
=> (λ x. λ y. (1 x (x y)))
=> (λ x. λ y. ((λ p. λ q. p q) x (x y))) //写出1的λ表达式
=> (λ x. λ y. ((λ q. x q) (x y)))
=> (λ x. λ y. x (x y)) = 2 //按定义
```

本例中succ和1都是函数(1是常函数)，归约实则为置换。第一步是λ n束定的n被1置换。展开后，x置换p，(x y)置换q，最后一行不能再置换了，它就是范式，语义为2。

以下是λ演算的归约规则：

(1) **β归约**，β归约的表达式是一个λ应用表达式(λ x. M N)，其左边子表达式是λ函数表达式，右边是任意λ表达式。β归约以右边的λ表达式置换函数体M中λ指明的那个形参变量。形式地，我们用[N/x, M]表示对(λ x. M N)的置换。

有以下规则：

- [1] 若x不在M中自由出现，结果为M；
- [2] 若M = x, 结果为M；

- [3]若 $M = LR$, 结果为 $[N/x, L][N/x, R]$;
 [4]若 $M = (P)$, 结果为 $([N/x, P])$;
 [5]若 $M = \lambda y. k$, $x \neq y$, y 不在 N 中自由出现, 结果为 $\lambda y. [N/x, k]$;
 [6]若 $M = \lambda y. k$, $x \neq y$, y 在 N 中自由出现, 结果为 $\lambda z. [N/x, [z/y, k]]$, $z \neq x \neq y$ 且不在 M, N 中自由出现。

关键的问题是注意函数体中要置换的变量是否自由出现, 如:

$((\lambda x. x (\lambda x. (x y))) (z z)) \Rightarrow (z z) (\lambda x. ((z z) y))$ //错误, 第二 x 个非自由出现。
 $\Rightarrow (z z) (\lambda x. (x y))$ //正确

例11-5 高层表示的 β 归约

$(\lambda n. \text{add } n \ n) \ 3$
 $\Rightarrow \text{add } 3 \ 3$ //置换 n 后取消 λn
 $\Rightarrow 6$
 $(\lambda f. \lambda x. f(f \ x)) \ \text{succ } 7$
 $\Rightarrow \lambda x. \text{succ } (\text{succ } x) \ 7$
 $\Rightarrow \text{succ } (\text{succ } 7)$
 $\Rightarrow \text{succ } (8) = 9$

注意, $\text{add}, 3, \text{succ}, 7, 9$ 是为了清晰没进一步展开为 λ 表达式。

但 β 归约有时并不能简化, 如:

$(\lambda x. x \ x) (\lambda x. x \ x)$

归约后仍是原公式, 这种 λ 表达式称为不可归约的。对应为程序设计语言中的无限递归。

(2) **η 归约** 是消除一层约束的归约, 形如 $\lambda x. F \ x$ 的表达式若 x 在 F 中不自由出现, 则:

$\lambda x. F \ x \Rightarrow F$

它的逆是任何一个 F 都可以加一套 $\lambda x(\)x$, 只要 x 在 F 中不自由出现。

显然, $\lambda x. f(x) \Rightarrow f$ 是 β 归约的特例。

(3) **α 换名** 归约中如发生改变束定性, 则允许换名(λ 后跟的变量名), 以保证原有束定关系。例如:

$(\lambda x. (\lambda y. x)) (z \ y)$ // $(z \ y)$ 中 y 是自由变量
 $\Rightarrow \lambda y. (z \ y)$ //此时 $(z \ y)$ 中 y 被束定了, 错误!
 $\Rightarrow (\lambda x. (\lambda w. x)) (z \ y)$ //因 $(\lambda y. x)$ 中函数体无 y , 可换名
 $\Rightarrow \lambda w. (z \ y)$ // 正确!

(4) **归约约定**

- 顺序 每次归约只要找到可归约的子公式(λ 应用表达式, 即一个是 λ 函数表达式, 一个是变元)即可归约, λ 演算没有规定顺序。
- 范式 符号归约当施行(除 α 规则外)所有变换规则后没有新形式出现, 则这种 λ 表达式叫范式。并非所有 λ 表达式均可归约为范式。前述 $(\lambda x. x \ x) (\lambda x. x \ x)$ 为不可归约的。
- 解释 范式即 λ 演算的语义解释, 归约后形如 $\lambda x. F$ 它就是函数, 形如 $x \ x, (y (\lambda x. z))$ 就只能解释为数据了, 只可以把它放在另一个应用中作变元进一步归约。

上述基本函数均为范式, 在它的上面取上有意义的名字可以构成上一层的函数, 如:

$\text{pred} = \lambda n. (\text{subtract } n \ 1)$

是一个前驱函数, subtract 展开归约后, 即 pred 的 λ 表达式。我们这样是为了阅读清楚。

(5) **综合规约例题**

例 11-6 以 λ 演算规约 $3**2$

$3**2 = ** (3) (2)$
 $= \lambda x. \lambda y. (y \ x) (3) (2)$

$$\begin{aligned}
& >_{\beta} (\lambda y. (y \ 3)) (2) \\
& >_{\beta} ((2) \ 3) \\
& = (\lambda f. \lambda c. f \ (f \ c)) \ (3) \\
& >_{\beta} \lambda c. (3 \ (3 \ c)) \\
& = \lambda c. (\lambda f. \lambda c. (f(f(f(c)))) (3c)) \\
& \hspace{15em} // \text{ 有c不能置换c} \\
& =_{\alpha} \lambda c. (\lambda f. \lambda z. (f \ (f \ (f \ (z)))) (3c)) \\
& >_{\beta} \lambda c. (\lambda z. ((3 \ c) ((3 \ c) ((3 \ c) (z))))) \\
& \hspace{15em} // \text{ 再展3} \\
& = \lambda c. \lambda z. (((\lambda f. \lambda c. (f(f(f(c)))c) ((3c) ((3c) (z)))) \\
& =_{\alpha} \lambda c. \lambda z. (((\lambda f. \lambda w. (f(f(f(w)))c) ((3c) ((3c) (z)))) \\
& >_{\beta} \lambda c. \lambda z. (((\lambda w. (c(c(c(w)))) ((3c) ((3c) (z)))) \\
& \hspace{15em} \text{同理展开第二个c, 第三个c} \\
& = \lambda c. \lambda z. (((\lambda w. (c(c(c(w)))) ((\lambda p. (c(c(c(p)))) ((\lambda q. (c(c(c(q)))) (z)))) \\
& >_{\beta} \lambda c. \lambda z. (((\lambda w. (c(c(c(w)))) ((\lambda p. (c(c(c(p)))) ((c(c(c(z)))))) \\
& >_{\beta} \lambda c. \lambda z. (((\lambda w. (c(c(c(w)))) ((c(c(c(c(c(c(z)))))))))) \\
& >_{\beta} \lambda c. \lambda z. (c(c(c(c(c(c(c(c(c(c(z)))))))))) = 9
\end{aligned}$$

11.2.5 Church-Rosser定理

我们在第六章已非形式地介绍过Church-Rosser定理，有了 λ 演算我们可以更严格表述它。

按照 λ 演算的运算规则，谁先归约最后归约其结果是一样的。
我们先看一复杂 λ 表达式：例11-7 不同归约次序的正规 λ 演算得同一结果

$$\begin{array}{c}
((\lambda x. (x \ y)) (\lambda u. (u \ v))) (\lambda p. (p \ q) \ r) \\
\swarrow \quad \searrow \\
((\lambda u. (u \ v)) \ y) (\lambda p. (p \ q) \ r) \quad ((\lambda x. (x \ y)) (\lambda u. (u \ v))) (r \ q) \\
\swarrow \quad \searrow \quad \swarrow \quad \searrow \\
(y \ v) (\lambda p. (p \ q) \ r) \quad ((\lambda u. (u \ v)) \ y) (r \ q) \quad ((\lambda u. (u \ v)) \ y) (r \ q) \\
\swarrow \quad \searrow \quad \swarrow \quad \searrow \\
(y \ u) (r \ q) \quad (y \ u) (r \ q) \quad (y \ u) (r \ q)
\end{array}$$

但是由于存在着非严格函数，求值次序影响着结果。Church和Rosser系统地研究了 λ 演算中归约特性，并给出定理。这里，我们只介绍与求值次序相关的定理。

(1) 严格函数

归约直到不可归约时正常终止，除开不可归约函数而外，有些函数尽管在符号上可归约，语义上无任何意义，到机器上不可执行的。这就是所谓停机问题，一般说来，停机问题是不可判定的，但我们在表示上给它以符号，例如：

$\text{divide } m \ 0 \Rightarrow \perp$

表示该函数非正常终止， \perp 是一个不代表任何值的值。不终止的计算结果值为 \perp 。

若 $f \ \perp \Rightarrow \perp$ 则称 f 为严格函数。

若 $f \ \perp \Rightarrow v$, v 为确定的值，则 f 为非严格函数。

正是由于有非严格函数的存在，就产生了求值次序的问题。 λ 演算如前所述归约次序没有严格规定。因为它是如前所述的正规求值：只要找到可规约的子表达式，哪种归约路径都能得到同样的结果。然而，由于非严格函数也能得出结果，就需要规定求值次序。

(2) 非严格函数取决于次序的例子

例11-8 λ 演算不同求值次序的归约

```

(( $\lambda n$ . multiply  $n \ n$ ) add 3 4)           //先做右端子表达式
=> ( $\lambda n$ . multiply  $n \ n$ ) 7                //急求值
=> multiply 7 7 => 49
=> multiply (add 3 4) (add 3 4)           //先做  $\beta$  归约
=> multiply 7 (add 3 4)
=> multiply 7 7 = 49                      //正规求值
若为 => multiply (add 3 4) (add 3 4)      //两个表达式一样，后一个不归约，
                                           取前一个的值。
=> multiply 7 7 = 49                      //懒求值
正常情况下，三种求值结果一样。因为它们符合  $\lambda$  演算。
若有表达式：  $\lambda n$ . 7 (divide 2 0) =>  $\perp$    //急求值
=> 7                                       //正规求值

```

正规求值是置换进行时，保留原样符号，直到最后使用时才自左至右求值，随用随求，用 n 次求 n 次。懒求值是一经有结果以后不用再求。且所有中间结果只求一次，作为值替换。

(3) 重述Church-Rosser定理

如按正规求值次序对某表达式 E 求值结果为正常值 v ，则按其他求值次序可能是 v 也可能是 \perp 。若按正规求值结果为 \perp ，即非正常终止，则按任何求值规则结果也为 \perp 。

正规求值是由外向内(Outside-in)对应为函数中的各调用。Algol, LISP早期语言按正规求值设计，如前所述，当有副作用时正规求值就不如懒求值了。为了安全，近代函数式语言均采用懒求值。与此对应，急求值(值调用)是内向求值(Inside-out)。

从上文看出， λ 演算是很正规、很基础的，可演算起来是个无限伸展的表(函数)、变元均作为表元素，不断嵌套。纯LISP完全按此风格，因而得到表处理语言的名声。不断向高抽象换名，表达起来要清晰明了得多，为增强表达能力 λ 演算一般扩充了以下表示法。

11.2.6 增强 λ 演算

核心 λ 演算的计算完备性已经证明。我们也看到只用最底层 λ 演算是极其复杂的。用高层命名函数，语义清晰。不仅如此，保留一些常见关键字，语义更清晰。例如，我们可以定义一个 `if_then_else` 为名的函数：

```

if_then_else =  $\lambda p$ .  $\lambda m$ .  $\lambda n$ . p m n

```

当 p 为‘真’时，执行 m 否则为 n 。我们先验证其真伪。

例11-9 当条件表达式为真时if_then_else函数的归约

```
(if_then_else) t m n => (λ p. λ m. λ n. p m n) t m n
=> (λ m. λ n. (t m n)) m n => (λ m. λ n. (λ x. λ y. x) m n) M N
=> (λ m. λ n. (λ y. m) n) m n => (λ m. λ n. m) m n
=> (λ n. m n) m n => m
```

同理p为F时可归约为取n的值。LISP的条件函数cond(E1 E2 E3)就是这样做的。但这样太不直观了。

在核心λ演算基础上增强λ演算增加了以下表达式：

(1) if表达式

可保留显式if-then-else形式：

```
(if_then_else) E1 E2 E3 = if E1 then E2 else E3
```

其中E1, E2, E3为λ表达式。表达式中的形参可用λ p. 形式抽取到最右边：

```
λ m. λ n. if(zerop n) then 0 else (divide m n)
```

zerop是除零函数，当n为零时令其为零，其次序仍为先归约E1，根据返回值选择E2 或E3。

(2) Let/where表达式

如果有高阶函数：

```
(λ n. multiply n (succ n)) (add i 2)
```

我们以add i 2置换变元n得：

```
multiply (add i 2) (succ (add i 2))
```

这样当函数较长时仍不清晰，于是设关键字let ... in...

```
let n = add i 2 in
  multiply n (succ n)
```

这样就清晰多了。它的一般情况是：

```
let a = b in E ≡ (λ a. E) b ≡ E where a=b
```

对于更为复杂的λ表达式：

```
(λ f. E2) (λ x. E1) = let f = λ x. E1 in E2
                     = let f x = E1 in E2
```

其中形如f=λ x. E1的λ x. 可移向左边为f x = E1。因为f是λ x. E1 的替换名，它应用到变元x上和说x是E1的形参效果是一样的。如：

```
sqr = λ n. multiply n n           //整个是λ函数表达式
sqr n = multiply n n             //两应用表达式也相等
```

let表达式在ML. LISP中直接采用，Miranda用where关键字使程序更好读，let直到E完结构成一个程序块。Miranda只不过把where块放在E之后。见本章例11-17。

(3) 元组表达式

我们先看形如(E1, E2)的有序对，再扩展到多元组。

例11-10 屏幕上的点处理

设p表示点对(x, y)则

```
(x, y) ≡ pair x y           //pair构造函数
```

一旦构造成统一元组，我们要处理时有：

```
let (x, y) = E1 in E2 ≡ let p = E1 in
  let x = first p in
    let y = second p in E2
```

其中first p => first pair x y => x

`second p => second pair x y => y`

为选择函数，当E2中有x，y出现时选出后即可计算。

一般情况下n元组是 $p=(x_1, x_2, \dots, x_n)$ ，建立在p上函数有：

```

`let f(x1, x2, ...xn)=E1 in E2 ≡ let fp=E1 in
                                let x1=first p in
                                let x2=second p in
                                .
                                .
                                .
                                let xn=n_th p in E2

```

若E1中用到元组值，从以上表达式中可得到确切定义。更进一步简化，通过名字把E1传到E2，如：

```

let max(x,y)=if x>y then x else y
in
  sqr max(x,y)

```

11.3 函数式语言怎样克服命令式语言的缺点

我们学习了λ演算自然就想到以它为模型的LISP，对于程序数据不分，高阶函数古怪的嵌套程序设计风格，函数抽象作为第一类对象，函数引用遵循引用透明原则理解就不困难了。但为了效率和I/O，LISP保留了PROG特征，所以依然有变量、赋值、副作用、执行顺序。它虽不“纯”由于时间长积累的软件多，一直还在发展。它的后裔有Common LISP，Scheme，CLOS(面向对象)。

新一代的函数式语言ML、Miranda、Hashell则力求更纯地实现Church定义的λ演算语义，并保留LISP在函数是第一类对象、条件表达式、隐式迭代和递归等方面成功的经验。但必须

- 严禁破坏性赋值
- 完全不考虑语句顺序
- 全部采用懒求值

我们进一步讨论这些问题。

11.3.1 消除赋值

消除赋值不等于说程序员不能给值取个名字(很象变量)，只是一旦这个名字和值束定后在名字作用域内至程序终结也不会改变，名字就成了常量且不取决于执行顺序。这样，理解程序或作正确性证明就大大简化了。

我们先看赋值语句在过程语言中起什么作用。在函数式语言中取消会有什么问题：

- [1] 存储计算子表达式的中间结果。
- [2] 条件语句的重要组成。
- [3] 用于循环控制变量。
- [4] 处理复杂数据结构(增删改某个成分)。

如果函数式语言全能实现上述功能，赋值当然可以消除。

Miranda保留全局量、局部量(符号名)以及参数名。全局符号即按 λ 演算的符号,声明时给出,参数名束定于变元表达式。where子句内的是局部符号名,整个的where部分以堆栈帧实现,局部名存放中间计算结果。where可嵌套产生块结构的名字束定。这样就完全达到作用[1]。

用条件表达式完全可以代替条件语句,其返回值通过参数束定或where子句束定于名字,达到作用[2]。

函数式语言都要定义表数据结构,因为归约和递归计算在表上很方便。对整个表操作实则是隐式迭代,不用循环控制变量。对于顺序值也都用表写个映射函数即可隐式迭代。部分达到作用[3],其它显式循环要用递归。见下文。

作用[4]是最麻烦的,禁止赋值意味着数据结构一旦创建不得修改,数据结构小是不成问题,每次求值重来一次,相当于值调用。每次调用复制一套,新拷贝束定于函数的参数名上,但是数据结构一大,修改又少(例如,1000元素的表只改一个元素)如此耗费是受不了的。

这个问题一直未根本解决,问题是要不要保留原拷贝,如果不要只将改动的换上新值只使当前版本有效还说得过去。如果要保留就不好办,一个办法是如图11-1:

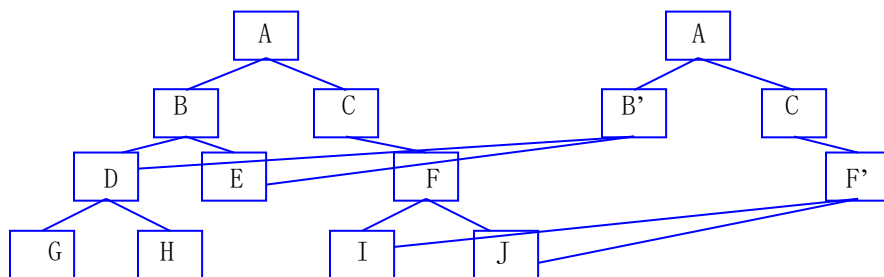


图11-1 函数式语言结构数据修改

它的基本思想是每次起个头,直到有修改节点。重复联上关系。以下就不用拷贝了。

如果此树只有 $B \rightarrow B'$; $F \rightarrow F'$, A, B, C, F拷贝并将B'联到D, E, F'联到I, J成为一新树,老树并不冲掉。

这种办法当修改在最底层时,拷贝工作少不了多少,再者,多次修改链太多,给访问带来麻烦。Miranda非常注意对数组的操作。

11.3.2 以递归代替while_do

我们在结构化程序设计一节中虽说过,顺序、条件选择、迭代(while_do)可以写出一切程序。现在看来,它们只是充分条件并非必要。顺序和while_do可以取消。

程序设计中重复必不可少,但不必由手工显式写出,由计算隐式实现,即用递归。

我们在《程序设计技术》或《程序设计方法学》之类的书中学过尾递归,尾递归是递归调用后不再有操作,把它们变成循环迭代是极方便的,变量置换之后它实质上就是无限循环,加上约束条件就是while_do或for循环。因而函数式语言的程序员,只要会组织条件表达式和递归函数,一切需要用重复计算之处均可仿真,而效果几乎是一样的。函数式语言递归之后一般不会有操作(没有语句好加入)几乎自然是尾递归的。

组织仿真的要点是把递归函数体写入条件表达式。循环终止条件是条件测试部分,函数如有返回值放在then部分,递归函数体放在else部分,如果不需返回值则取消这部分(else)。测试的循环控制变量作为函数的参数束定引入。控制参数的增量由另一函数计算或参数本身就是自增的函数,以下用Pascal写出的递归仿真重复,读者学了任何一种函数式语言即可翻译为

函数式程序版本。

例11-11 递归实现重复

以下程序打印整数数组的值，为调用Print_out(LL, N)时，它打印数组LL的头N个值。这个过程仅为一个初始化的外壳，实际是调用一个递归过程print_recursive，外壳还设置了一记打印次数的局部量count，它只有一个拷贝，而递归函数每调用一次在堆栈框架顶端生成一个数据、信息的版本，返回后退栈释放所有版本。

```
PROCEDURE Print_out (list:list_type; list_size:Integer);
  PROCEDURE Print_recursive (count:Integer);
  BEGIN
    Writeln (list [count]);
    IF (count < list_size) THEN Print_recursive (count +1);
  END ;
BEGIN
  Print_recursive(1);
END.
```

当然递归实现没有递推实现效率高，尾递归转成递推也是很容易的事，但在实现上可大不同。begin。递归每调用一次要新建一堆栈帧以记录递归信息，而递推则不必，因而效率高多了。LISP家族的语言能在识别出尾递归后自动转成递推实现。

11.3.3 消除顺序

传统语言中控制计算次序本来就有两手：嵌套和顺序。语义相关密切的写成一个嵌套块，其它就是安排顺序。由于命令式语言允许大量顺序语句，程序员往往把语义相关的也写成顺序的。函数式语言虽是一个函数、一个函数地写，但要求逻辑无关。一旦相关无法传递数据，非得写成嵌套函数不可(返回值自动绑定到外套函数的变元上)。函数式语言只有一条路，而 λ 演算证明这条路足矣(至少在理论上)。

嵌套函数和嵌套调用虽然可以完全取消顺序性，但代码实在难读，近代函数式语言为改进可读性，表示上回归，原理还是嵌套。先开门见山读出函数，然后再读，其中(where)什么是什么。

(1) 用嵌套代替语义相关的顺序

相关语句都可以看个某个主要语句的子句。 λ 演算的let/where块结构，就是局部符号定义，函数调用靠束定，把子句放在局部定义之中写一个主函数即可，请看示例：

例11-12 嵌套代替顺序

有一Pascal语义相关的顺序语句片断：

```
c:= a + b;
s:= sin(c * c);
write(a, b, c, s);
```

//上面计算不进行是无法打印的
//逻辑上要有顺序。

LISP 实现：

```
(PRINT (LET ((C(+ A B))
  LET (S (SIN (* C C)))
    (LIST A B C S ))))
```

//仍有顺序但在一个表达式内。自左至右

// 处理即隐式顺序。

Miranda实现:

```

Answer a b = (a b c s )
  where
    s = sin (c* c)
    c = a+b

```

//多么清晰，全然没有顺序

(2) 用懒求值代替顺序

懒求值是到了“用的时候才求值”，而不象命令式语言“有了值才能用”。这就在一定程度上取消了次序。就拿上例说，LISP可用急求值实现，即顺着let，let执行到list也可用懒求值实现，先不执行子句到list时再去执行子句。Pascal实际是急求值，没有上一步就没有下一步，Miranda则完全是懒求值。在给定参数后打印表(a b c s)。c，s，无值到where中去找，找到了s求值时又无c值，最后找到c = a + b，此时一古脑全求出来！因此，s，c，子定义随意颠倒也无所谓。

所以，懒求值是取消顺序的有力措施。

(3) 利用卫式进一步消除顺序性

一般说来，取消了赋值，以嵌套表达式描述计算，描述性就相当好了，但Miranda进一步打破条件表达式的求值顺序。它用可任意并行执行的卫式(guarded)表达式来选择执行应选部分。

例11-13 Miranda的卫式表达式

```

gcd a b= gcd (a-b) b, a>b
        = gcd a (b-a), a<b
        = a,           a=b

```

逗号后为卫式表达式。这三个表达式并行求值，谁为真选谁的对应子表达式。一般情况程序员总会使它只有一个为‘真’，当有两个以上为‘真’时系统就不可判定了，与此对比的LISP程序如下。

例11-14 LISP的条件选择

```

(DEFINE GCD (a b)
  (COND ((Greaterp a b) (GCD ((Difference a b) b)))
        ((Lessp a b) (GCD (a (Difference b a)))))
  (T a)))

```

它首先测试(Greaterp A B)，不成功再测试(Lessp A B)，再不成功才是(T A)，因为COND函数测试的表达式对可以很多(相当于常规语言case条件)，这样顺序求下去很费时，但它相对安全，不会出现两个同时为‘真’的情况。

11.3.4 输出问题

有了上述消除顺序的措施，剩下的就是输入和输出了。

懒求值对多数计算是很好的，但它不宜程序的输出，因为根据定义输出过程是有副作用的，并不需要返回值。且根据懒求值规则跳过变元在输入/输出中是不允许的。所以要么允许程序员指定严格求值，仅为了输出取消懒求值，这导致大量可以不求值的变元也求值了。再一个办法是利用数据对象内部原有的顺序。复合对象类型的值(如表和元组)都是有序的，它们都

可以动态构造。由于在函数式语言中，结构对象是第一类对象，语言支持的任何形式(交互、非交互)的输出都可以用在表和元组上。Miranda就是用无限表动态实现的，见下节。

11.4 函数式语言Miranda

Miranda语言是1986年出现的新风格的函数式语言。完全没有变量、赋值和顺序语句的概念。程序间的值传递靠函数调用和参数束定。由于采用懒求值故可以提供“无限表”，程序只有严格静态定义的对象、类型、和函数，能充分说明函数式程序设计的主要特征。

Miranda程序是脚本(script)定义的集合。脚本定义了命名对象，每个定义的左手边是符号名，‘=’号右边是定义它的子句(函数)，可以有多个子句。对象有简单数据类型，此外就是表、元组还有函数。表达方式和 λ 演算中 λ 表达式相对应。但比LISP更注意接近数学传统表示。

例11-15 简单的Miranda脚本

| | |
|------------------------------|---|
| Miranda | λ 演算 |
| <code>sq n = n * n</code> | <code>sq = $\lambda n. n * n$</code> |
| <code>z = sq x / sq y</code> | <code>z = / (sq x) (sq y)</code> |

函数调用是函数名后跟变元表。变元可以是简单数据项、表和元组，只是不用括号。脚本定义以行自然结束，不可有‘;’或其它符号。

Miranda的基本类型有字符(char类型，加单引号的字母)，真值(bool类型，值为True和False)和数(num类型，包括整数、实数)。数据结构只有表和元组，串是字符表，串字面量可以用双引号括着的字符串，也可以写作字符表。

11.4.1 数据结构

Miranda只有两种复合数据结构，元组和表。

(1) 元组(tuple)

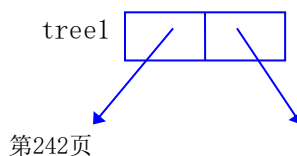
是异构(不同类型)值和序列，用圆括号括着的简单对象的表。相当于常规语言记录数据类型，可以用一指向元组成员的指针表实现。括号括着的值序列即为元组字面量。函数的实参变元表也是元组，只是它可以不要圆括号。

元组用于定义枚举数据类型，并组成复合的数据结构，定义和访问元组类型的方法，

例11-16 树类型定义

| | |
|--|-----|
| <code>tree ::= Leaf Integer Node tree tree</code> | [1] |
| <code>leaf1 = (Leaf 3)</code> | [2] |
| <code>tree1 = (Node leaf1 (Node (Leaf 17) (Leaf 49)))</code> | [3] |

其中‘::=’为类型定义符，以资与函数定义‘=’区别，‘|’为‘或’。tree是多态类型，它可以是叶子Leaf(整型)或中间结点Node(两个树类型的元组)类型，这种形式同时也定义了构造子，[2]行即按构造子Leaf构造leaf1对象，将名字leaf1束定于两元元组，意即对象值为3。同理，[3]行构造三元元组，意为如下图所示的树：



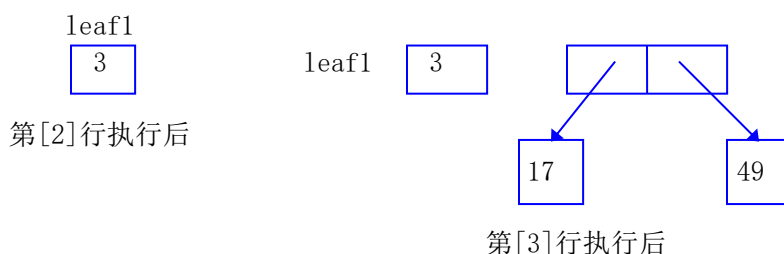


图11-2 例11-16定义的树

有了类型和对象定义可写出函数。

例11-17 多态函数定义及调用

设函数 `max_tree` 为求树中叶子值最大。定义如下：

```
max_tree (Leaf ldata) = ldata [1]
max_tree (Node n1 n2) = max1, max1 > max2 [2]
                        = max2, max2 > max1 [3]
                        = max1, otherwise
                        where
                        max1 = (max_tree n1) [4]
                        max2 = (max_tree n2) [5]
```

`max_tree`定义了两种型式的变元，当为叶子变元时值就是叶子的数据`ldata`。当变元为树时继续递归并选取值大的叶子值。[4]行以下是`where`子块，其中`max1`，`max2`是函数调用，意即对树变元的左（`n1`）右（`n2`）子树分别求最大值，然后作比较，如此递归下去到达叶子，则和[1]行匹配。再利用卫式条件检查留下最大者。

如有应用：

```
(max_tree leaf1)      //结果值为3, leaf1用上例
(max_tree tree1)      //结果值为49, tree1用上例
```

调用按变元类型找匹配`tree1`是`Node tree tree`类型，故匹配函数定义的[2]行。求值时外向内，求到`max1`或`max2`再去计算`max1`，`max2`的函数调用，即懒求值。

(2) 表(list)

表是同类型值的序列，用方括号括着的简单对象列表，有四种表示法：空表，无限表，有限表，间隔表。

例11-18 Miranda表的表示法

```
[ ]                //空表
[1..n]             //1到n, 域表示
odd_number = [1, 3, ..100] //1到100内奇数表，头两项及最后项必写
eleven_up = [11...]  //10以上，无限表表示
evens = [10, 12..100] //10以上偶数表至100，头两项及最后项必写
evens_up = [12, 14...] //10以上偶数无限表，
week_days = [ "Mon", "Tue", "Wed", "Thur", "Fri" ] //五个串值的表
```

11.4.2 内定义操作

Miranda定义了常规的算术运算符(+、-、*、/、div、mod)并按中缀表示使用,括号和优先级和传统概念没什么两样,和传统函数式语言一样,把表特别是不定长的表作为符号演算的空间,以体现 λ 演算的归约。故内定义了一些有用的表操作:

```
L1 ++ L2           // 表L2并接到表L1的末尾
item>List         // 将项item加到表List的头前
List ! n          // 从表List中选取第n项
L1 -- L2          // 从表L1中剔出L2的值
#List             //返回表List的项(基)数
```

例10-18 Miranda表操作示例

设已读入下表:

```
week_days = ["Mon", "Tue", "Wed", "Thur", "Fri"]
```

| 表达式 | 结果 |
|------------------------------------|-------------------------------|
| # week_days | 5 |
| days = week_days ++ ["Sat", "Sun"] | days中成为7元素 |
| 0:[1 2 3] | [0 1 2 3] |
| week_days ! 2 | "Wed"(Miranda采用0基下标) |
| week_days -"Fri" | ["Mon", "Tue", "Wed", "Thur"] |

11.4.3 定义函数

Miranda把函数定义叫方程(equation)因它力图接近数学表达,简单函数一行一个例11-17中已示出。先写名字接着是形式参数表(可以是空,不用括号),紧接'='号后是包含形参的表达式,一行写不下后续行必须缩排(不超出上行'=')。

Miranda提供卫式(guarded)表达式,以实施条件控制。卫式表达式和左边的表达式要在一行并由逗号隔开,它本身是Boolean型表达式。一个函数条件控制可定义多个方程(一般三个足矣,>、=、<)三个子句的卫式表达式并行求值,哪一个求值为真执行哪一个子句,并返回值。若多个卫式表达式求值为真(>=或<=情况)则返回任何一个均可。最末一行不写卫式表示“otherwise”。

例11-20 卫式表达式

斐波那契数的函数定义,用卫式表达式实现递归

```
Fibonacci n = 1,           n=0
              = 1,         n=1
              = Fibonacci (n-1)+Fibonacci (n-2)  n>1
```

卫式表达式的值集应复盖所有的可能,否则用otherwise。

函数式程序以束定代替赋值,函数调用为求值表达式提供了进行束定的机会,将结果值束定于参数名。当有多个嵌套的函数调用时,程序难读程序员也难于构造。Miranda提供了where子句,为参数名一值束定以更灵活的第二方式。在函数定义时,where子句为该表达式定义一些中间值的局部名字,这些中间“赋值”和传统语言的赋值形式无异(注意,意义不同是束定到返回值)。子束定表达式可以多个,并可以嵌套到任意层次。产生块结构的名字束定。当然,内层应以缩排方式给出,以便释义时信息识别。

例11-21 利用where解二次方程

```
quadroot a b c = error "complex roots",      delta < 0
                = [term1]                    ,      delta = 0
                = [term1 + term2, term1 - term2], delta > 0
                where
                delta = b*b - 4*a*c
                radix = sqrt delta
                term1 = -b/(2*a)
                term2 = radix /(2*a)
```

如果读者编过类似的Pascal或C程序会发现，Miranda的这个程序要简洁得多了，其原因是卫式表达式简化了if_then_else语句。where子句先后无所谓，它按需要求值，程序员不必考虑求值顺序，（因懒求值且没有副作用的二义性）。如果子束定表达式不需要就不会去求值。上例中若delta=0，where子句中除计算delta和term两项而外，其它均不求值。

Miranda的设计者将指称哲学用于为返回表提供自由束定，由于是懒求值和变长表，在括号内程序员可随意写下表项，组织表达式非常方便。Ada、Pascal构造表和初始化就很费事。

Miranda完全按 λ 演算模型，每个函数都是一元运算，当有多个变元时，函数名也是第一类对象，它逐一应用到各个参数，中间返回函数可以任意取名，这种中间函数称Curry函数（为纪念逻辑学家Haskell Curry得名）。

例11-22 直角三角形求斜边长函数

```
hypotenuse a b = sqrt (a * a + b * b)
```

调用时

```
hypotenuse 3 4          //=5
```

也可写作：

```
(hypotenuse 3 ) 4  → f 4
```

Miranda则写作为：

```
f 4 where f = hypotenuse 3 //f=sqrt(9 + b*b)即 Curry 函数。
```

Curry 函数充分体现高阶函数思想，即使是一简单多元函数，也可以看作一元函数返回函数，再作用到第二元……直到结果值的一个高阶函数。与此相应Curry函数的类型是：

〈函数名〉：：〈变元类型〉→〈结果值类型〉

高阶函数时重复‘→’号直至最后结果。成对圆括号可任加，表示是函数类型。

高阶函数定义，表、递归的灵活运用形成Miranda的简捷风格。我们看一对比程序。我们先用类Pascal程序写出一行向量求和（归约）程序，然后给出对比的Miranda。

例11-23 Miranda用高阶函数实现类型参数化。

```
TYPE row_type=ARRAY[0..9] OF Integer;
FUNCTION Reduce(FUNCTION f(x:Integer,y:Integer):Integer;
                ar:row_type):Integer;          //函数f是参数化的。
VAR sum,k:Integer;
BEGIN
    sum:=ar[0];
    FOR k=1 TO 9 DO
        sum:=f(sum,ar[k]);
    reduce:=sum
END;
FUNCTION MyOp(s:Integer,y:Integer):Integer;    //此处定义一实例函数。
BEGIN
```

```

MyOp:=abs(x) + abs(y)
END;
FUNCTION MySum(ar:row_type):Integer;
BEGIN MySum:=Reduce(MyOp,ar) END;

```

本程序仅将 Reduce函数操作参数化。数组元素类型、长度还是固定的。Miranda都可以，它设3个参数：操作、表、单位元。单位元的值可用于归约过程的初始化值，也是空表时的返回值：

```

reduce f[] n = n [1]
reduce f(a : x) n = f a (reduce x n) [2]

```

为了理解上不引起歧意，写明reduce的类型：

```

reduce::(num→num→num) //第一变元f是函数类型（有括号），它是二元算子
      →[num] //返回值是表，表元素是num类型
      → num //若空表映射为数，类型是num.
      → num //最后映射返回值是num类型。

```

[2]行中（a:x）是表头a和表尾x，右边函数体是表尾归约后与表头归约。[1]行指明空表规约 n 次是单位元的 n 次复合。

Miranda的num是数类型既可以是整数也可以是实数。

11.4.4 表闭包

表闭包(comprehension)是一个纯函数式结构，它提供了型构表上的迭代方式且不需赋值和顺序语句。相当于传统的循环。

表闭包是一个表达式，它包括两部分，‘|’的左边是表_表达式，右边是子句序列也叫限定符(qualifier list)表，由‘;’号隔开的产生器(generator)和过滤器(filter)组成。限定符表说明了值束定到表达式中自由变量的序列。它由1到多个表表达式的生成器组成，紧跟着零个或多个布尔表达式的过滤器。过滤器实则是约束条件，表闭包是一个任意复杂结构的(无限)表。其语法是：

```

<ZF表达式> ::= ‘[’ <体> ‘|’ <限定符表> ‘]’
<限定符表> ::= <产生器> { ‘;’ <产生器> | <过滤器> }
<产生器> ::= <变量> ← <表_表达式>
<过滤器> ::= <布尔表达式>
<体> ::= <表_表达式>

```

其中ZF表达式是为纪念逻辑学家Zermelo和Franled而得名。它们是开发集合论的先驱。

例11-24 表闭包示例

| ZF 表达式 | 解释 |
|-----------------------------------|-------------------------|
| [n*2 n←[2, 4, 6, 8, 10]] | [4, 8, 12, 16, 20] [1] |
| [n*n n ←[1, 2, ...]] | [1, 4, 9, ...] [2] |
| [x+y x← [1..3]; y←[3, 7]I | [4, 5, 6, 8, 9, 10] [3] |
| [x*y x ←[1..3]; y←[1..3]; x>=y] | [1, 2, 4, 3, 6, 9] [4] |

[1]行只有一个生成器，表值2, 4, 6, 8, 10束定于n得出2倍表。[2]行是无限表也只有一

产生器，[3]行[4]行有两个产生器，[4]行还有一过滤器，否则要对称出9个数。

表闭包类似于for循环，其自由变量相当于循环控制变量。自由变量依次束定后加工各元素。闭包与循环不同之处是它必须返回一个表作为结果值。单产生器相当于单循环，多产生器相当于嵌套循环。有了过滤器，比for循环就要复杂了，而且按需要可以设多个过滤器，即将不合要求的值排除在外。

体中有几个变量就得要几个产生器，就相当于嵌了套。嵌套运行规则如同常规语言的嵌套循环，也是先内后外层地满足。上例最后一行即按此次序作出的表。即使是多个产生器的嵌套闭包结果也是一维表。过滤器每当按体中取出一组值就检查一次，有了过滤器结果表长度不一(当然，表结果要有内部约定，如同C)。

有了闭包写程序可以极为精炼，因为不用显式表示递归或迭代，以下是快速分类有名的例题。

例11-25 用Miranda编写快速分类

```
sort [ ] = [ ]                                [1]
sort (pivot: rest) = sort [y | y ← rest; y<=pivot] [2]
                    ++ [pivot] ++              [3]
                    sort [z | z ← rest; y>pivot] [4]
```

[1]行定义函数sort的变元是空表它返回一空表(调用同一函数)，一般定义递归它都有[1]行。它同时指出sort是表变元。[2]行的圆括号不是表，而是说明变元一个表，它由元组pivot和rest组成，即表头、表尾。以局部的名字给出了与之结合的实参表表头、表尾。

sort函数描述的函数定义是：构造三个新表，第一个表从rest中取出小于或等于pivot的y值构成，第二个新表是pivot，第三个表是大于pivot值的表。++操作指明了正确连接为一个整表。然而，[2]行[4]行的求子表依然是对sort函数的递归调用它们又各自建立三个新表，直至都成空表与[1]行匹配返回，那么所有元素都分类完了。

这个程序比任何其它语言都要简洁。表闭包并未定义过程而是指明(denote)哪些元素可进一步作为变元递归调用sort函数。这个程序仅仅是如何分类的陈述，要做正确性证明非常容易。

但这个程序的时空开销是比较大的，因为每个表都要查两次，分成两个新表。没有有效的无用单元回收，老表所占无用空间也是个问题。

11.4.5 无限表

上述各例均已用到无限表了，我们再讨论如何实现无限表。

我们可以把无限表看作两部分：有限的头部，它放λ已经算好了的值，和一个无限的尾，它由生成新表尾的代码组成。这相当于一个函数，也采用懒求值，即不到表不够时不计算它。程序可以用类似于其它的语言的下标来访问表中各值。

很明显，无限表尾不表示任何值，它是函数对象，每当调用到它时，它按规定计算表头值，并构造一新的函数对象放在表尾，以便再展其它项，它就是新的无限表尾的头，这个过程一直延续到需要的表长已达到。

这个函数可由记录当前表位置的闭包实现。由当前位置算出下一位置，新函数对象又创建了新闭包，它又记下了表的新位置。

有了无限表，我们看输入/输出。

程序设计语言没有输入/输出显然是不行的。但它们天生来是依赖于时间，通常的输入概念是执行READ过程使存储对象获得值，这和函数式不一致，但无限表提供了模型输入/输出的方式：输入流就是一个值的无限表，每次处理输入流一个新值就附在表尾并为程序访问。

同样，也用无限表模型输出流。问题在于对整个表求值之时实际的输出已开始发生，如

果让输出函数返回一个“成功”和“失败”的响应，并测试这个响应，即使是懒求值这也是可以做到的，测试在于查出输出函数是否求值，还全靠求值中的副作用使输出实现呢！

11.5 问题与讨论

函数式程序设计表达能力强、程序简洁是用过这类语言的人的口碑。我们从LISP，ML，到Miranda的发展可以看出Miranda广泛利用数学符号和块子句，几乎和数学式子一样，程序和数学公式极为接近，更为宜人。这对程序验证，正确性证明是极为有利的。

(1) 模拟状态不易

在人工智能领域，模式匹配、变换证明， λ 演算都是胜任的。以 λ 演算为模型的函数式语言当然都胜任。函数式语言的出现，使程序设计方法学设计算法时更多从数学公式上下手。对于符号逻辑、推理、泛函和科学计算都是它的强项。但描述状态、随时输出状态变化之类的问题能力较弱。主要原因在于一个函数只能通过其参数和返回值与操作系统通信，且非嵌套函数之间数据传递困难。状态模拟要设懒表并施以较大技巧。

本书不是讨论应用范型只从语言设计实现上提到它，有兴趣的读者可参阅M. C. Henson 1987年出的《函数式语言元素》一书。

(2) 效率还是问题

除了难以模拟状态而外，效率普遍认为是函数式语言的大问题，到目前为止虽然新语言版本在效率问题上作了许多改进，从过去比顺序的命令式语言慢200-1000倍到近来的3-5倍，其原因是：

- 函数是第一类对象，局部于它的数据一般要在堆(heap)上分配，为了避免悬挂引用，要有自动重配的检查。
- 无类型(如LISP)要在运行中检查类型，即使是强类型的(如ML, Miranda)减少了类型动态检查，但函数式语言天然匹配选择模式的途径也是运行低效原因。
- 懒求值开销大：每次用到函数的参数，每次从复合值中选出一个值时都要进行检查，以免出现未求值的表达式。在急求值的语言中，(如命令式的)就不需这种检查。ML也采用急求值。
- 中间复合值一多费时费空间。如本章所述复杂对象每中间修改一次就要重新生成，通过程序变换减少中间值则可提高效率。
- 无限表动态生成，计算一次增长一个元素！效率也很低。

但这不是说函数式语言比命令式语言低一个等级(就效率而言)，在它所长之处的某些程序，命令式语言的并不及它，它的表达能力及处理复杂问题的简捷性在今天硬件速度日益增长的背景下是依然有竞争力的，因为效率只要能接受表达能力自然是追求目标。

(3) 并发性

函数式程序设计的并发性也是值得讨论的，因为并发性是该领域发展的动力之一。函数式语言被认为是非常适用于处理并发性问题的工具，共享值不需加特殊保护，因为他们不会被更新。所以在函数式语言中，显式同步结构是不必要的，并且在分布式实现过程中能随意复制共性，因为并行进程之间不会互相干扰，而这大大简化了推理和测试(在有些并行性应用中，也希望提供非确定性，但通常我们对并行性感兴趣的动机只是它能改善程序运行情况)。

函数式程序的功能并不依赖于特殊表达式的计算和程序的其它部分是否并行地执行。许多研究人员正设法使编译程序能够识别哪些表达式可被同时求值，以代替程序员来作这种判断。

这类似于FORTRAN编译器，它能判断给定程序中对大数组可进行怎样的并行处理。原则上函数式语言的编译利用了非正规并发性的优点，而数组处理器则利用了正规的并行性。此法的好处是一个确定性程序在不同的并行(且顺序)系统上不作修改就可被重新编译并运行，当然，程序应避免选择一个固有的顺序算法。要实现一个好的并发性处理，还有一些困难问题要解决。在将表达式求值分配给不同的处理器这一点上就有隐藏的额外开销：用于求表达式值的数据必须从一个处理器传到另一个处理器，而表达式的计算结果还得被传回来。如果表达式求值所需时间极少，则用于通讯的额外开销将占用分配给计算表达式值的时间，所以必须在没有程序员帮助下，使这段额外开销时间尽量短，寻找解决这个问题的先进技术是一个热门的研究课题。

11.6 小结

- 命令式语言变量时空特性使程序难于用数学模型，导致了函数式语言的发展。
- 程序中无变量，副作用和难以查错的问题，迎刃而解，但无变量是一巨大革命：没有语句、没有循环，剩下的是函数表达式，条件表达式和递归函数。
- 没有和存储相连的变量和赋值，函数式程序中值的修改得每次重新构造。
- λ 演算是一个符号逻辑系统，可提供无goto和赋值语言的语义元语最小集。可以表达任何可计算函数。 λ 演算的表达式可看作一小语言。
- λ 演算基于函数抽象 $\lambda x. E$ 和函数应用 $E1E2$ 两个最基本的思想，应用归约即作 λ 演算。
- λ 演算归约规则有BC求值)、 η (化简)、 α (换名) 归约， λ 表达式除不可归约的而外，归约最后的结果得到范式。范式是原表达式的语义解释，可以是数据可以是函数。故以 λ 演算为模型的语言，程序、数据没有明显区分。
- Church-Rosser定理指出：正规求值无结果其它求值次序也无结果，反之，其它求值次序也可能无结果。
- 增强 λ 演算扩大了核心 λ 演算表达能力是一种块结构。
- 新一代函数式语言严禁破坏性求值，完全不考虑语句顺序，采用懒求值。函数的参数束定取消赋值，递归函数代替循环(隐式代显式)嵌套函数、子表达式块消除顺序。懒求值、卫式表达式也消除顺序。
- 表和元组一般是函数式语言最重要的数据结构，定义好了处理使用方便。
- Curry函数是多目运算转为 λ 演算的单目运算的形式，也是函数抽象是第一类值的体现。Curry函数就是高阶函数，高阶函数表达能力强可提供抽象函数类型，支持类型变量。
- 函数式语言宜于形式匹配、变换与证明、逻辑推理、泛函及科技计算，不宜于模拟状态，效率有待进一步提高，是非冯语言有力的竞争者之一。

习题

11.1 何谓函数式程序设计风格？尽可能描绘它。

11.2 求 b^n ($n \geq 0$) 一般是将 b 自乘 n 次，有一个更好的算法是：

$$\begin{aligned} b^0 &= 1 \\ b^{2n} &= (b^2)^n \\ b^{2n+1} &= (b^2)^n * b \end{aligned}$$

用任何一种函数式语言写此程序。或以任何过程语言模拟函数式风格写此程序

11.3 递归与迭代有何不同，说明后，将10.2程序用递归和迭代各编一个，证明你的说法。

11.4 Pascal 和C均可用函数作为变元，说明为什么它们不能编制高阶函数 程序？

11.5 为什么程序语言界要用 λ 演算研究程序设计语言?

11.6 归约以下 λ 应用表达式

$$[1] ((\lambda x. \lambda y. x)u) \Rightarrow \lambda y. u$$

$$[2] ((\lambda x. \lambda y. z)u) \Rightarrow \lambda y. z$$

$$[3] ((\lambda x. \lambda u. ux)u) \Rightarrow ((\lambda x. \lambda w. wx)u) \Rightarrow \lambda w. wu \Rightarrow u$$

$$[4] ((\lambda x. \lambda x. uu)u) \Rightarrow \lambda x. uu \Rightarrow u$$

11.7 用核心 λ 演算证明LISP的cond函数, 当所有子表达式均为NIL时执行

(T fun)

结果就是fun。

11.8 用核心 λ 演算计算:

$$[1] \quad 1+2 = 3 \quad [2] \quad 3^2 = 9$$

$$\text{add } 1 \ 2 \Rightarrow \lambda x. \lambda y. \lambda a. \lambda b. ((xa) (ya) b) 1 \ 2$$

$$\Rightarrow \lambda a. \lambda b. ((1a) (2a) b)$$

$$\Rightarrow \lambda a. \lambda b. (((\lambda p. \lambda q. pq) a) (\lambda x. y. x(xy) a) b)$$

$$\Rightarrow \lambda a. \lambda b. (\lambda q. aq) (\lambda y. a(ay)) b)$$

$$\Rightarrow \lambda a. \lambda b. (a(a(ab)))$$

$$\Rightarrow 3$$

11.9 归约以下 λ 应用表达式:

$$[1] ((\lambda x. \lambda y. x(xy)) (pq) q)$$

$$\Rightarrow ((\lambda y. (pq) (pqy)) q)$$

$$\Rightarrow (pq) (pq(q))$$

$$[2] ((\lambda x. \lambda y. y) (pq) q) \Rightarrow (\lambda y. y) q \Rightarrow q$$

$$[3] ((\lambda z (\lambda y. yz)) \lambda x. xy) \Rightarrow (\lambda z. (\lambda y. yz)) y \Rightarrow y. yy \Rightarrow w. wy \Rightarrow y$$

$$[4] ((\lambda x. \lambda y. y(xy)) (\lambda p. pp) q) \Rightarrow ((\lambda x. \lambda y. y(xy)) pq)$$

$$\Rightarrow q(pq)$$

11.10 设 $\text{twice} = \lambda f. \lambda x. f(f\ x)$

$$\text{试证: } \text{twice twice } gz = g(g(g(g\ z)))$$

$$\text{证明: } \text{twice twice } gz$$

$$\Rightarrow (\lambda f. \lambda x. f(fx)) (\lambda f. \lambda x. f(fx)) gz$$

$$\Rightarrow \lambda f. \lambda x. f(fx) g(gz)$$

$$\Rightarrow (g(g(gz)))$$

\therefore 得证

11.11 试证

$$\text{zerop } 1 = F$$

$$\text{zerop } 0 = T$$

11.12 写出 λ 函数表达式, 用本章给出高层符号。

$$[1] \quad n+n$$

$$[2] \quad \text{函数变元为整数 } n, \text{ 即 } f(n), \text{ 有}$$

$$f(n) = \begin{cases} 1 & n > 0 \\ 0 & n = 0 \\ -1 & n < 0 \end{cases}$$

[3] 函数变元为一个点用点对 (x, y) 表示, 该函数返回该点在 x 轴上的映象, 即 $(x, -y)$

$$[4] \quad \text{函数变元为两函数 } f, g \text{ 其结果为 } f.g.$$

11.13 orelse 函数有两个变元, 满足以下真值表:

$$t \ t' \ \text{orelse}$$

| | | |
|---|---|---|
| T | T | T |
| T | F | T |
| F | T | T |
| F | F | F |

[1]试写出orelse的扩充 λ 演算表达式。

现有判零函数调用,如(zerop 1)和判正数函数调用,如(positive 5),用急求值和正规求值对以下式子求值:

[2] orelse (zerop 1) (positive 5)

[3] orelse (zerop 0) (positive 0)

解: [1]orelse (t,t')= $\lambda t. \lambda t'. ((t\ T)t')$

[2]orelse (zerop1) (positive 5)

\Rightarrow orelse (F) (T) //急求值。先求zerop1和positive 5的值

$\Rightarrow \lambda t. \lambda t'. ((t\ T)\ t')FT$

$\Rightarrow \lambda t. ((t\ T)F)T$

$\Rightarrow (TT)F$

$\Rightarrow (\lambda x. \lambda y. x)TF$

$\Rightarrow (\lambda y. T)F$

$\Rightarrow T$

\Rightarrow orelse (zerop1) (positive 5) //正规求值

$\Rightarrow \lambda t. \lambda t'. ((t\ T)t')(\text{zerop1})(\text{positive5})$

$\Rightarrow \lambda t. ((t\ T)(\text{zerop1})(\text{positive5}))$

$\Rightarrow ((\text{positive5})T)(\text{zerop1})$

$\Rightarrow (TT)F$

$\Rightarrow T$

[3] orelse (zerop 0) (positive 0)

\Rightarrow orelse (T) (F) //急求值

$\Rightarrow \lambda t. \lambda t'. ((t\ T)t')TF$

$\Rightarrow ((t\ T)T)F$

$\Rightarrow ((FT)T)$

$\Rightarrow \lambda x. \lambda y. yTT$

$\Rightarrow T$

$\Rightarrow \lambda t. \lambda t'. ((t\ T)t')(\text{zerop } 0)(\text{positive } 0)$

$\Rightarrow \lambda t. ((t\ T)(\text{zerop } 0))(\text{positive } 0)$

$\Rightarrow (\text{positive } 0)T(\text{zerop } 0)$

$\Rightarrow F\ T\ T$

$\Rightarrow \lambda x. \lambda y. y\ T\ T$

$\Rightarrow \lambda y. y. T$

$\Rightarrow T$

11.14 以下是什么函数调用? 请用数学式子表示:

$\lambda m. \lambda n. \text{if zero } P\ n \text{ then } 0 \text{ else divide } m\ n$

以 $m = 12, n = 0$, 对此 λ 表达式求值。

11.15 懒求值和严格求值有何不同?

11.16 何谓无限表, 如何用它实现灵活数组?

答: 可把无限表看作两部分: 有限的头部, 它放入已经算好了的值, 和一个无限的尾, 它由生成新表尾的代码组成. 这相当于一个函数, 也采用懒求值, 即不到表不够时不计算它.

程序可以用类似于其它的语言和下标来访问表中各值.

11.17 何谓表闭包? 限定符? 过滤器?

答:表闭包(comprehension)是一个纯函数式结构,它提供了型构表上的迭代方式且不需赋值和顺序语句.相当于传统的循环。

表闭包是一个表达式,它包括两部分:‘|’的左边是表_表达式,右边是子句序列也叫限定符表,由‘;’号隔开的产生器(generator)和过滤器(filter)组成。限定符表说明了值绑定到表达式中自由变量和序列,它由1到多个表达式的生成器组成,紧跟着零个或多个布尔表达式的过滤器,过滤器实则是约束条件,表闭包是一个任意复杂结构的(无限)表。

11.18 递归加条件怎样取代while_do,举例说明之。

11.19 用Pascal或Ada写一个快速分类的过程并和例10-22的Miranda程序比较,说出你的评价。用prolog能否写出呢?如能请写出。

11.20 将例11-10写成任何一种函数式执行。

11.21 有以下函数:

$$Y = \lambda g. (\lambda f. g(f f)) (\lambda f. g(ff))$$

试证 $YF = F(YF)$

证明: $YF = \lambda g. (\lambda f. g(ff)) (\lambda f. g(ff)F) = (\lambda f. F(ff)) (\lambda f. F(ff))$
 $= F((\lambda f. F(ff)) (\lambda f. F(ff))) = F(YF) \quad \therefore \text{得证}$

11.22 将以下递归函数程序改写为逆推程序,并打印出结果。

```
PROGRAM
  CONST N = 4;
  TYPE String = PACKED ARRAY [1..N] OF Char;
  PROCEDURE Permute(s:String; k: Integer);
  VAR I :Integer;
  BEGIN
    IF k = N THEN
      Writeln(s)
    ELSE
      FOR I:=k to N DO
        BEGIN
          SwapChar(s, k, I)
          Permute(s, k+1)
        END
      END
    END
  PROCEDURE SwapChar(VAR s:String; p1, p2: Integer);
  VAR Temp : Char;
  BEGIN
    Temp:=s[p1];
    s[p1]:=s[p2];
    s[p2]:=Temp;
  END
  BEGIN
    Permute('ABCD');
  END
```

_11.23从计算逻辑(过程逻辑)和计算结构模型本评价函数式语言。

第12章 逻辑式程序设计语言

无论是命令式还是函数式程序都把程序看作是从输入到输出的某种映射。当然命令式语言有时没有数据输出，但也要“输出”某些动作。为了实现这种映射，程序要对数据结构实施某个算法过程，算法实现该程序功能。算法又是以程序语言提供的控制机制实现计算逻辑。所以，R.Kowalski说：

算法 = 逻辑 + 控制

然而传统语言计算逻辑在程序员心里，隐式地体现在程序正文之中，为此程序正确性证明还要把它隐含的逻辑以断言形式地写出来。

自然人们会想到能不能把描述计算的理论基础命题演算和谓词演算直接变为程序设计语言。这样，也许不必用求值来判定某件事情的真、伪，直接根据事实和规则判定真伪，“找出”解。事实上，这是可行的，而且在人工智能的专家系统、语言理解、数据库查询中非常需要这种程序设计语言。1970年诞生的Prolog长久不衰就是例证。

逻辑程序设计的基本观点是程序描述的是数据对象之间的关系，它的抽象层次更高而不限于函数(映射)关系。关系也是联系，对象和对象、对象和属性的联系就是我们所说的事实。事实之间的关系以规则表述，根据规则找出合乎逻辑的事实就是推理。因此，逻辑程序设计范型是陈述事实，制定规则，程序设计就是构造证明。程序的执行就在推理，和传统程序设计范型有较大的差异。

本章我们从逻辑程序设计理论基础，谓词演算导出逻辑程序语言的理论模型，并介绍逻辑程序设计语言Prolog的主要特征和实现要点。

12.1 谓词演算

谓词演算是符号化事实的形式逻辑系统，它也是逻辑程序设计语言的模型，谓词演算在所有计算机理论的书籍中均有论述，本章仅简单复习，主要目的是引入术语。

12.1.1 谓词演算诸元素

用形式方法研究论域上的对象需要一种语言，它能表达该域对象具有什么性质(properties)，以及对象间有些什么关系(relations)。为了一般化还要有变量(variable)指明域上某个(些)对象，以及准确说明对象情况的量词(quantifiers)。当然这个语言还需要一些辅助的符号(继承自初等集合论)。描述以公式(Formulas)表达，即描述一般命题的谓词。谓词公式中各元素按一定逻辑规则变换即谓词演算(predicate calculus)。以下是谓词公式的示例：

$\exists X p(X)$ —存在具有性质 p 的对象 X 。

$\forall Y p(Y)$ —所有的 Y ，即域上任何对象，均具有性质 p 。

$\forall X \exists Y (X = 2 * Y)$ —对于所有的X都可以找到Y，其值为X的一半，其中的逻辑连接“=”是中缀表示。等效于 $=(X, 2 * Y)$ 。

谓词公式各元素符号表示法在各教程中不尽相同，谓词演算的表示由以下元素组成：

(1)公式 由一组约定的符号组成的序列，它包括常量(指明域上的某个对象)、变量(域上任意对象)、逻辑连接(指明对象状态及对象间的关系，有数量上的 $>$ 、 $>=$ 、 $=$ 、 $<$ 、 $<=$ 、 \neq 和逻辑上的 \vee 、 \wedge 、 \neg 、 \rightarrow 、 \leftrightarrow)、命题函数(指明对象间的函数性质)，谓词(对象间约定的关系)，量词(逻辑连接的一种，指明对象状态)。一阶谓词演算系统中公式里不得嵌套谓词。

(2)常量 指明论域(universe of discourse)上的对象，也是数学对象。通常一个逻辑系统要引入多个论域，一般有十进制整数域、真值域、字符域，可用不同的符号表达有区别的域。常量可以看作是退化了的函数(没有变元)。常量以字面量或小写字母表示。

(3)变量 可束定到特定域上某个范围的对象上，在演算(归约)期间它可以例化为具体对象(常量对象)，变量是数学意义的，一旦束定整个演算期间不变，变量以大写字母表示。

(4)函数 表征对象具有的映射关系，函数带参数，从变元域映射到结果值域(可以是不同的域)。变元一般是常量、变量、函数结果值、变元在语法上都叫项(term)。函数以小写标识符表示。

(5)谓词 表征对象某种性质的符号，谓词带上一到多个变元(对象)即为断言(assertion)。它断言这些对象具有谓词所指性质，谓词形如函数，当该变元确实具有所指性质时隐含真值，否则为假。

当谓词应用到的变元是常量或已被束定的变量上时，就叫做句子(sentence)或命题(proposition)，查询时能返回真假值。当谓词应用到变量或包含变量的项上时，它依然是谓词但不能形成句子，只有例化才能判定真伪，形成句子。

谓词变元的个数称作目(arity)，故有单目、N目谓词之称。

例12-1 N-目谓词的例子。

| 谓词 | 目 | 含义 |
|--------------------|---|------------|
| odd(X) | 1 | X是奇数 |
| father(F, S) | 2 | F是S的父亲 |
| divide(N, D, Q, R) | 4 | N除D得商Q和余数R |

| 谓词例化 | 结果值 |
|-------------------------------|-----------|
| odd(2) | False |
| divide (23, 7, 3, 2) | True |
| father (changshan, changping) | True |
| divide (23, 7, 3, N) | N未例化，不知真假 |

(6)量词 变量可以代表域中任何对象，量词可以指定某种集合，量词有两个，一为全称‘ \forall ’、一为存在‘ \exists ’，量词限定的变量名作用域是整个公式，它说明同名变量。有些谓词经量词修饰后即没有变量，可成句子或命题。

例12-2 量化谓词

| 量化谓词 | 结果值 |
|---|-----------------|
| $\forall X \text{odd}(X)$ | False |
| $\exists X \text{odd}(X)$ | True |
| $\forall X (X = 2 * Y + 1 \rightarrow \text{odd}(X))$ | True |
| $\forall X \exists Y \text{divide}(X, 3, Y, 0)$ | False |
| $\exists X \exists Y \text{divide}(X, 3, Y, 0)$ | True, 如X=3, Y=1 |
| $\exists X \forall Y \text{divide}(X, 3, Y, 0)$ | False, 但很难证明 |

证明一个全称谓词是比较难的，因为最可靠的证明方法是枚举例证，这对于论域大的公式就很困难了。于是采取反证的方法，如能找出一个反例则原公式(也是命题)为假，这种假言推理的表示恰好将全称改为存在，全称量化的谓词取反：

例12-3 量化公式

| 量化谓词 | 取反 | |
|--|---|-----|
| $\forall X \text{odd}(X)$ | $\exists X \text{not odd}(X)$ | [1] |
| $\exists X \text{odd}(X)$ | $\forall X \text{not odd}(X)$ | [2] |
| $\forall X(X=2*Y+1 \rightarrow \text{odd}(X))$ | $\exists X \text{not}(X=2*Y+1 \rightarrow \text{odd}(X))$ | [3] |
| | $\exists X \text{not}(X=2*Y+1) \text{or odd}(X)$ | [4] |
| | $\exists X((X=2*Y+1) \text{and not odd}(X))$ | [5] |
| $\forall X \exists Y \text{ divide}(X, 3, Y, 0)$ | $\exists X \forall Y \text{ not divide}(X, 3, Y, 0)$ | [6] |
| $\exists X \exists Y \text{ divide}(X, 3, Y, 0)$ | $\forall X \forall Y \text{ not divide}(X, 3, Y, 0)$ | [7] |
| $\exists X \forall Y \text{ divide}(X, 3, Y, 0)$ | $\forall X \exists Y \text{ not divide}(X, 3, Y, 0)$ | [8] |

第[4]、[5]行实际就是通过谓词演算得到更加直观的等价式，因为局部not是极容易找出例证的。

(7) 逻辑操作 and, or, not, \rightarrow (蕴含) \Leftrightarrow (全等)是最基本的逻辑运算符，然而，最本质的有and(\wedge), or(\vee), not(\neg)就够了，如：

$$\begin{aligned} A \rightarrow B &= \text{not } A \text{ or } B = \neg A \vee B \\ A \Leftrightarrow B &= (A \text{ and } B) \text{ or } (\text{not } (A \text{ and } (\text{not } B))) \\ &= (A \wedge B) \vee \neg(A \wedge \neg B) \end{aligned}$$

命题演算即原子命题通过逻辑连接，等价变换，达到验证命题真理性的目的。谓词演算以准确的谓词描述论域上对象的逻辑关系，通过等价变换达到证明一般命题真理性的目的。命题演算是谓词演算的特例。

12.1.2 谓词演算的等价变换

等价变换即演绎推理，不同的逻辑连接有其等价变换的规则。命题，谓词演算规则可以查有关文献。本节演示一般谓词公式变换为子句的实例。'⊢'号为“可推出”。

[1] 以 \wedge , \vee , \neg 如上例消除 \rightarrow 、 \Leftrightarrow 符号。

[2] 化为前束范式，消除最外的 \neg 符号，即否定符号内移。

$$\neg(\exists X p(X)) \vdash \forall X(\neg p(X))$$

[3] 利用斯柯林变换消去存在量词。如：

$$\forall X \exists Z(c(Z, X) \wedge b(X)) \vdash \forall X(c(z, X) \wedge b(X))$$

再如： $\forall X(a(X) \wedge b(X) \vee \exists Y c(X, Y))$

$$\vdash \forall X(a(X) \wedge b(X) \vee c(X, g(X))) \quad (11.1)$$

其中 $g(X)$ 是对任何 X 可得到(存在)的 Y 。

[4] 消除前束范式的全称量词。既然设定所有变量均为全称，要不要符号都是一个意思，即都要搜索全域才能证实该变量，则(11.1) 式为：

$$\vdash a(X) \wedge b(X) \vee c(X, g(X)) \quad (11.2)$$

[5] 利用分配率 $P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$ 化成合取范式，(11.2)式成为：

$$\vdash (a(X) \vee c(X, g(X)) \wedge b(X) \vee c(X, g(X))) \quad (11.3)$$

经过以上变换任何一复合公式均可成为如下形式:

$$F = C1 \wedge C2 \wedge \cdots Cn \quad (11.4)$$

且其中 C_i 称为子句, 如果以', '代替' \wedge ', 且其次序无关可表达为集合形式:

$$F = C1, C2, \cdots, Cn$$

其中 C_i 内部没有其它逻辑连接符号, 只有' \vee '. 若以';'代' \vee '则有:

$$C_i = L1 \vee L2 \vee \cdots L_v = L1; L2; \cdots; L_v \quad (11.5)$$

因此, 任一公式均可化为' \vee '连接的子句的集合, 证明' \vee '连接的子句为真是很容易的, 但证明整个公式为真则需所有子句均为真, 即子句逻辑。

12.2 自动定理证明

自1879年Frege正式研究符号逻辑以来, 至本世纪30年代末的五十年期间, 数理逻辑发展比较完备, 它们一直用作基础数学的证明系统。电子计算机问世后, 人们不仅借助计算机完成手工不能完成的证明工作(如1987年的四色定理证明), 更感兴趣的是用逻辑定理对程序作正确性的自动证明。本节简述证明系统的概念和模型。

12.2.1 证明系统

用谓词演算公式描述的事实即证明系统中的公理(axioms), 证明系统(proof system)是应用公理演绎出定理(theorems)的合法演绎规则的集合。所谓演绎, 也叫归约(deduction), 是对证明系统中合法推理规则的一次应用。在一个简单的演绎步骤中, 可以从公理导出结论(conclusion), 中间可利用以这些规则演绎出的定理。

证明(proof)是个语句序列, 以每个语句得到证明而结束, 即每个句子要么演绎成公理, 要么演绎成前此导出的定理。一个证明若有 N 个语句(命题)则称 N 步证明, 反驳(refutation)是一个语句的反向证明。它证明一个语句是矛盾的, 即不合乎给定的公理。

同一命题的正向证明和反驳有时会有天壤之别, 证明长度和复杂性差别很大。构造一个证明或反驳要有深入的洞察、联想还要有点灵感。写得好, 脉络清晰, 句子简明, 反之臃肿晦涩。即使是较差的证明构造起来也要有点技巧。

一个语句若能从公理出发推演出来, 则称合法语句, 任何合法语句也叫做定理(theorem)。从某一公理集合导出的所有定理集合称为理论(theory)。一般说来, 理论具有一致性, 它不包含相互矛盾的定理。

12.2.2 模型

从公理集合中导出定理集称之为理论, 有了理论我们要解释它的语义必须借助某个模型(model)。因为形式系统只是符号抽象, 借助模型我们可为每个常量、函数、谓词符号找到真理性的解释。即定义每个论域, 并表明域上成员和常量公理之间的关系。公理的谓词符号必须派定为域中对象的性质, 函数派定为对域中对象的操作。不一致的理论就没有模型, 因为无法找到同时满足相互矛盾定理的解释。

公理集合一般情况下只是定义的部分(偏)函数和谓词, 是问题域的一个侧面。所以能满

是该理论的模型往往不止一个。如，下例是一“间隔数理论”为找出该理论的一个模型，先找一个论域，它必需要能解释给出的三个公理，常量'1'和'2'，函数符'+', 谓词interval。

例12-4 一个最简单的理论

公理集:

$$\forall X \text{interval}(X) \rightarrow \text{not interval}(X+1) \quad (\text{a1})$$

$$\forall X \text{not interval}(X+1) \rightarrow \text{interval}(X) \quad (\text{a2})$$

$$2=1+1 \quad (\text{a3})$$

从间隔数公理可导出定理:

$$\forall X \text{interval}(X) \rightarrow \text{interval}(X+2) \quad (\text{t1})$$

$$\forall X \text{interval}(X+2) \rightarrow \text{interval}(X) \quad (\text{t2})$$

如果我们设定论域是整数，则'1'，'2'可解释为整数一和二，且 '+' 就是整数加(符合公理 a3)，即加函数。谓词 interval(间隔数)在整数域上显然有两个子域 odd(奇数)、even(偶数)都能够满足。如果奇数都是间隔数，在本理论中偶数定然不是。反之，奇数不是。既然我们论述在整数域上，对任一整数我们倒反不能确定它是否为间隔数了，间隔数理论不能证明 interval(3)，也不能证明 not interval(3) 为真命题。这就是 Milbert 讨论过的可判定性(decidability)问题。1936年 Church 和 Turing 证实谓词演算可判定性问题是没解的，因为不存在也不可能证明一个算法能正确地验证非真命题。对于真命题即使有算法过程存在，也要在无限长的时间才能做完证明。

然而，一旦我们断言 interval(3) 或 interval(2) 是真命题，我们立刻可通过演绎证明按这个理论写出的每一个谓词为真。这就是 Goedel 和 Herbrand 1930 年证实的谓词演算具备的完整性(completeness)。它可以证明该理论所有模型里的每一语句为真。谓词演算可用来开发每一逻辑真命题的形式证明系统。甚至存在机械的证明方法(Gentzen 1936)。

12.2.3 证明技术

从谓词演算具有完整性，理论上可作出自动生成程序并证明按公理集合建立的任何理论，它们是公理集合的逻辑结论。但实际做起来，要找到切题的证明如同大海捞针，效率难以容忍。

即使把问题限制在证明单个命题，关键仍然是效率。如果我们从公理出发做出每一个步骤，在新的步骤上仍然要查找每一个公理找出可能的推理。如此下去就形成一个庞大的树行公理集，每层的结点都是表示一个公理的语句，其深度和宽度随问题和最初给出的公理而定，一层一步，N层的树就是N步推理。

对于自动定理证明程序，只有穷举每条可能的证明步骤才能说它是完全的。然而，穷举完所有路径马上遇到组合爆炸问题，无论是深度优先还是广度优先，百步演绎可能的路径数都是天文数字。以下是证明路径示例。

例12-5 试证65534是个好整数

设谓词 good(X) 表示 X 是好整数。谓词 Powtwo(Y, Z) 表示 2 的 Y 次方为 Z，显然，powtwo(0, 1) 是事实。若 A 是好整数与 2 作整数运算之后结果仍应为好整数。若 $2^p=A$ 且 p 为好整数则 A 也是好整数， $2^{p+1}=2*A$ 也都是显然的公理。所以有:

公理

$$\text{f1} \quad \text{good}(0)$$

$$\text{f2} \quad \text{powtwo}(0, 1)$$

$$\text{r3} \quad \text{good}(A-2) \leftarrow \text{good}(A)$$

$$\text{r4} \quad \text{good}(A+2) \leftarrow \text{good}(A)$$

```

r5  good(A*2)←good(A)
r6  good(A)←powtwo(P, A), good(P)
r7  powtwo(P+1, A*2)←powtwo(P, A)

```

七条公理中有两个事实5条规则，以下证明过程：

| | | |
|-----|-------------------|----------------|
| s1 | good(2) | f1和r4 |
| s2 | good(4) | s1和r5 |
| s3 | powtwo(1, 2) | f2和r7 |
| s4 | powtwo(2, 4) | s3和r7 |
| s5 | powtwo(3, 8) | s4和r7 |
| s6 | powtwo(4, 16) | s5和r7 |
| s7 | good(16) | s2, s6和r6 |
| s8 | powtwo(5, 32) | s6和r7 |
| s9 | powtwo(6, 64) | s8和r7 |
| s10 | powtwo(7, 128) | s9和r7 |
| s11 | powtwo(8, 256) | s10和r7 |
| s12 | powtwo(9, 512) | s11和r7 |
| s13 | powtwo(10, 1024) | s12和r7 |
| s14 | powtwo(11, 2048) | s13和r7 |
| s15 | powtwo(12, 4096) | s14和r7 |
| s16 | powtwo(13, 8192) | s15和r7 |
| s17 | powtwo(14, 16384) | s16和r7 |
| s18 | powtwo(15, 32768) | s17和r7 |
| s19 | powtwo(16, 65536) | s18和r7 |
| s20 | good(65536) | s7,s19和r6 |
| s21 | good(65534) | s20和r3 得证 |

这个证明人工花了21步，机器怎么会知道？它必须把每个事实和已证明的定理逐个代入到这5个规则中验证得出新的定理，每得出一个 $\text{powtwo}(p+1, A*2)$ 后要像第七步s7和第二十步s20一样把幂和值分出来，连试算到计算至少要操作 $7^{21}=4.7173 \times 10^{17}$ 次(4.72亿亿次)。

12.2.4 归结定理证明

不加限制地写出公理，穷举证明显然不是一条自动证明的出路。因为要机器自动查匹配演算到预期的合适公式，代价是极大的。于是对谓词公式的写法加以限制，即在12.1.2节把任何公式化为子句逻辑的办法可以省去一大片的归约演算，即使如此，穷举证明计算量也相当可观。

一个技术是J.A.Robinson1965年提出的归结(resolution)法，归结法是命题演算中对合适公式的一种证明方法。为了证明合适公式F为真，归结法证明 $\neg F$ 恒假来代替F永真。归结原理是：设有前题 $L \vee P$ 和 $\neg L \vee R$ 则其逻辑结论是 $P \vee R$ ，因为两个子句中含有一个命题的正逆命题($L, \neg L$)。若L为真 $\neg L$ 一定为假，P若为真， $P \vee R$ 也为真。若L为假 $\neg L$ 为真，P若为真， $P \vee R$ 也为真。这个推理把两子句合一(unification)并消去一对正逆命题，故归结也译作消解。归结证明的过程并称之为归结演绎，其步骤如下：

[1] 把前题中所有命题换成子句形式。

[2] 取结论的反, 并转换成子句形式, 加入[1]中的子句集。

[3] 在子句集中选择含有互逆命题的命题归结。用合一算法得出新子句(归结式)再加入到子句集。

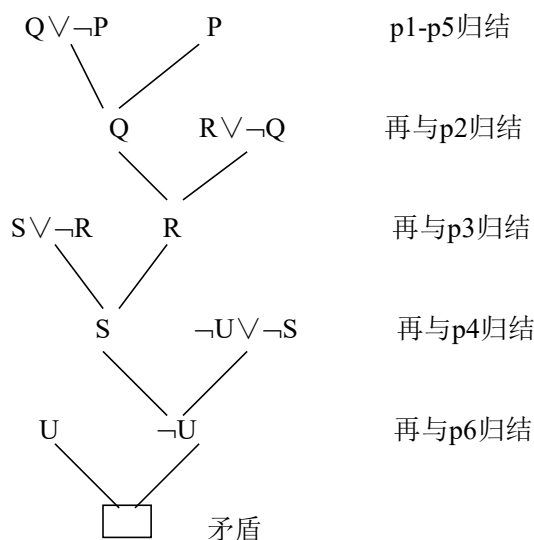
[4] 重复[3], 若归结式为 \square 则表示此次证明的逻辑结论是矛盾, 原待证结论若不取反则恒真。命题得证。否则继续重复[3]。

例12-6 归结证明

| 若有前题 | 待证命题 | 取反得新子句 |
|-------------------------|----------------------|--------|
| p1 $Q \vee \neg P$ | $\neg P \vee \neg U$ | p5 P |
| p2 $R \vee \neg Q$ | | p6 U |
| p3 $S \vee \neg R$ | | |
| p4 $\neg U \vee \neg S$ | | |

取待证命题的反, 得 $P \wedge U$, 它是 \wedge 连接的两个子句 P , U , 把它们加到前题子句集, 为p5, p6。

归结演绎如下图:



由于归结为 \square , 则 $\neg P \vee \neg Q$ 得证。

由本例可以看出两个问题, 第一, 归结法是由合一算法实现的。所谓合一是找出行式匹配的两子句, 将它们合一为归结式, 相当于代数中的化简。

更广义的合一算法是找出两子句最通用的实例(对于谓词演算)即找出变量的实例, 使两子句等价。例如:

| 公式1 | 公式2 | 广义合一 |
|--------------------------|--------------------------|----------------------|
| $pp(a, b)$ | $pp(X, Y)$ | $X=a, Y=b;$ |
| $p(a, b) \wedge q(c, d)$ | $p(X, Y) \wedge q(Z, W)$ | $X=a, Y=b, Z=c, W=d$ |
| $q(p, g(X, Y), X, Y)$ | $q(Y, Z, h(U, k), U)$ | $U=Y=p, X=h(p, k)$ |
| | | $Z=g(h(p, k), p)$ |

第二是如果得不出矛盾那么归结法要无休止地做下去, 中间归结式出得越多, 匹配查找次数越多, 每一步都做长时间计算, 显然, 要想出比这种原始归结法的更好办法, 即利用切断(cut)操作, 并对子句形式进一步限制的超级归结法(Hyperresolution)。

12.2.5 Horn子句实现超归结

Horn子句是至多只有一个非负谓词符号的子句。这就等于说，通过谓词演算一个语句只包含一个蕴含运算符连接前题和结论，前题是由'∧'连接的几个谓词，结论就是单一的谓词符号。

Horn子句形式示例如下：

$$\neg P \vee \neg Q \vee S \vee \neg R \vee \neg T \quad (12.6)$$

其中只有一个非负谓词S，可作以下演算，先将S移向右方

$$\vdash S \vee \neg P \vee \neg Q \vee \neg R \vee \neg T \quad (12.7)$$

按德·摩根定律

$$\vdash S \vee \neg(P \wedge Q \wedge R \wedge T) \quad (12.8)$$

'∨¬'即'→'，则

$$\vdash S \leftarrow (P \wedge Q \wedge R \wedge T) \quad (12.9)$$

此即条件Horn子句，因为(12.9)的意义是if($P \wedge Q \wedge R \wedge T$) then S。显然，若S为空，则为无条件Horn子句，是一个断言(事实)。

超级归结实质上是将无条件Horn子句中的谓词符号和条件子句中的对应谓词符号合一。找出所有子句中变量的实例集，使每一条件子句为真。如果不满足则寻找新的实例(回溯算法)，如果满足了也要找出所有实例。

为了消去不必要的匹配以提高超级归结的效率，cut操作是必须的，它可以由程序员指定。当找到一个解之后不再搜索其它解。它本身是个无变元谓词，当执行到它时，不再回溯。cut操作也可用于分情形动作控制与fail(失败)谓词联用，12.4节还将举例。

12.3 逻辑程序的风格

基于自动定理证明的逻辑语言，有其独特的程序设计风格，因为它不描述计算过程而是描述证明过程。例如，有一个问题：对数组A按升序排序，我们怎样编逻辑程序呢？我们只好构造一个和A那样大的数组B，而且它是排好升序的，我们证明命题：

“存在一个数组B它是数组A重排升序”

为此，可以陈述更严谨一些：

B是A的重排，当且仅当B的元素就是A的元素，且 $B[i] < B[j]$ ， $i < j$ 。

于是，自动定理证明系统依靠环境，构造一个希望的解(B)，并证明它的存在。构造的办法不外乎匹配查找，例化或置换。证明是本题的主旨排序成了副产品！

除了证明性风格而外，逻辑程序的第二个特点是描述性，请见下例：

例12-7 求平均成绩的逻辑程序

打开一分数文件scores，读入分数求和并用的数N除之得平均成绩。

```
average:-see(scores),
    getinput(Sum, N),
    seen(scores),
    Av is Sum /N,
    print('Average = ', Av)
getinput(Sum, N):-ratom(X),
    not(eof),
    getinput(Sum1, N1),
```

Sum is Sum1 + X,

N is N1 + 1.

getinput (0, 0):-eof.

要使求平均成绩程序正确，必先打开(see)文件，从中得到输入，得N个数的总和记以N和Sum，关闭(seen)文件后求平均值Av，打印之。至于如何得到Sum和N，细化谓词getinput(,)。若要getinput成立，先读入一原子X(ratom是系统提供的谓词)。若未至文件末尾，读其余分数并求和。getinput是递归定义，并相应断言Sum，N和Sum1，N1的关系。如果已至文件结束标记eof则输入为(0, 0)。这个程序为求平均分数给出三条规则，‘，’号即子句的‘^’连接，意即所有子句为真，左端谓词才成立。

这个程序是用Prolog写的，我们虽未介绍它的语法，一经简单解释我们即可读懂该程序。它的风格是：若要A成立，做B，做C，做D，…再细化，若要B做出则要做P，做Q，做R，…这与过程式语言自顶向下描述没什么差别，且比较自由，没有严格的顺序性。当然程序执行还需有事实的陈述，以及需求证明的查询。

逻辑程序第三个特点是大量用表和递归实现重复操作，递归特别利于谓词描述，我们只要能说明特征谓词的一步动作为真，其余如法炮制，程序就设计完了，上例中，getinput，读一数X，求和并令门数加1，其余照做。

关于表与递归联用的例子，见12.4节中Prolog的例子。

12.4 典型逻辑程序设计语言Prolog

Prolog是典型的逻辑程序设计语言，它基于一阶谓词逻辑，直接陈述问题世界的事实、规则和目标，非常简明清晰，所以人们称之为自文档的。Prolog的程序将逻辑和控制显式分离，它使程序正确性证明简化(只涉及知识，即逻辑部分)，程序的优化(只涉及控制部分)不影响程序正确性，是正交设计成功的范例。

12.4.1 Prolog的环境

Prolog很长一段时间都是交互式语言，它必须要有支持环境，即管理事实和规则的数据库，并为操纵它们提供元语言。小的数据库可以文件形式联接上，大的库在键盘上调用输入函数consult(filename)。一旦进入系统，即可查看库内容(用listing (predicate_name))或编辑它们(用retract(z))，还可以扩充数据库(用asserta(X)或assertz(X))。

联上数据库，程序员即可交互使用。查询、测试假设、断言更多的事实。为响应查询，Prolog系统执行它的证明过程并查找能满足查询变量的事实集。查找成功，写出找到的事实，若程序员满意并打回车键后，系统给出“yes”并显示新提示，如果数据库查完还未找到要的事实则给出“no”，并显示新提示。

12.4.2 数据对象和项

Prolog的基本成分是对象(常量、变量、结构、表)、谓词、运算符、函数、规则。

- 常量 预定义有十进制整数，用户定义的常量叫原子，有小写字母开头的名字，

Prolog是无类型的，程序是理论，可以对应多个模型(也可以没有)，因而原子在不同模型中是不同的对象。

- 变量 大写字母开头的名字。一个规则中同一变量名都将束定到同一对象上。符号‘_’是变量的占位符，对于确有一变量但暂时不能指明时则补上‘_’，一个规则中，‘_’出现几次就束定几个变量。

- 结构 复合的数据结构，如记录类型可用函子(函数名)和括在括号中的成分(域)表给出：
<函数名>(<成分>, ..., <成分>)

例如，course(C_no, C_name, Teacher, Precno)

- 表 表是最频繁使用的数据结构。以方括号表示，[] (空表)，[Head | Tail](递归定义的一般表)[X,Y,Z](定长表)[a, b | Z](表头为常量，表尾为变量Z)。表中元素叫项(term)可以是原子、表、空。

表的规格说明有两种形式，它们等效：

[X, Y, Z] • (x, •y, •(z, []))

[Head, Tail] •(Head, Tail) // '•' 指出连接关系

开端表(open)是表尾长度可变的表。证明过程中，开端表可以和任何与其表头有相同常量的表匹配(合一)。

选择子函数head(X)，tail(X)可操纵表X返回表头或表尾变元。

- 谓词 预定义有=、\=、<、=<、>、=>。用户可定义自己的谓词，以小写字母给出谓词名。机器是不理解谓词的语义的，它只知道匹配，所以，程序员应将谓词的意义记清并按它写程序。谓词带参数，即项，但参数不能是谓词、函数和关系。

- 运算符 预定义了整数运算符+、-、*、/和mod，可以写成标准的函数式+(3, x)也可以写成中缀形式 3 + x，使用中缀形式时有通常的优先级，算术等式还有一种is的形式，如X is(A+B)mod C，当Prolog处理这样的公式时，它用A, B, C的当前束定来执行这个计算其返回值束定于X，如is右边变量未束定值则返回错误。

从纯语法意义上Prolog的项什么都可以表示：

<项>::=<常量>|<变量>|<结构>|(<项>)<表><后缀算符>
|<项><中缀算符><项>|<, 项><前缀算符>

从语义角度，以下语法描述提供了处理时的语义概念：

<程序> → <子句>
<子句> → (<事实> | <规则> | <查询>)
<事实> → <结构>
<规则> → <头> :- <体>
<头> → < 结构>
<体> → <目标> , <目标>
<目标> → /*形如p或q(T, ...,)的字面量*/

12.4.3 Prolog程序结构

Prolog程序由子句组成，如前所述，子句模型是Horn子句。Horn子句本质上提供最基本的形式逻辑，即if then条件推理，把多个子句放在条件中，当条件(即前题)均满足，then部分即结论。如果条件为空，即只有以谓词表达的结论，就是断言，即事实。条件子句即为规则。

为了简明，Prolog以' :- '代替‘←’以', '代替'and'，以';'代替'or'，'!'代替cut。

和其它语言环境一样，系统配备了一批支持程序设计的预定义谓词(相当于其它语言的内定义函数)。

和其它语言一样，程序分两部分，定义和应用。程序定义该程序所需的公理集，即事实和规则。应用部分即查询，即写出待证的目标。整个程序是一证明系统。

(1) 事实与规则

Prolog程序先定义公理集，我们看下列：

例12-8 Prolog的规则和事实

| | |
|-----------|--|
| 条件子句(规则) | pretty (X) :- artwork(X) pretty (X) :- color(X, red), flower(X). watchout (X) :- sharp(X, _). |
| 无条件子句(事实) | color (rose, red). sharp (rose, stem). sharp (holly, leaf). flower(rose). flower(violet) artwork (painting (Monet, haystack_at_Giverny)). |

一般说来，一个谓词可以定义好几个规则来表达它。如同每个规则用‘or’联接起来表达这个谓词，只有一个(第一个满足该谓词的)规则来解释对该谓词的调用。所以可以写出一系列的规则来表达通用的条件语义结构。

规则可以递归，这样就可以实现算法需要的重复。当然，递归谓词必须至少要有两条规则，一为归纳基础，一为递归步骤。

(2) 查询

Prolog中查询(query)是要求Prolog证明定理。因为提出的问题就是证明过程的目标，所以查询也叫目标(goal)。语法上查询是个由逗号分开的项表。语义上是同时满足的谓词，逗号即and。如果查询中没有变量，Prolog从已给定的规则和事实证明它，查询以‘?-’开始，请看下列：

例12-9 Prolog的查询

```

?- pretty (rose).
yes
?- pretty (Y).
Y=painting (Monet, haystack_at_Giverny).
Y=rose.
no
?-pretty(W), sharp(W, Z)
W=rose Z=stem
no

```

第一个查询问“玫瑰美丽吗？”它到例12-8的事实库中找出玫瑰是花，颜色是红的，满足第二条规则，故查询结果是yes。

如果查询中包含变量，Prolog的定理证明系统将找出满足查询的实例集合，查询中的所有变量均隐含具有量词 \exists ，也就是查询问的是否存在满足子句的对象。它给出一个就等待，当程序员键入‘;’，它就给出下一个直至没有，给出no。例12-9中的?- pretty(Y)查询，它找到例12-8 中第一条规则匹配，再找到最后一个事实匹配，找出Y=莫奈的油画《吉曼尼的草堆》。第二次再查，找出玫瑰。第三次查不出则给出no。

查询所包含的谓词项，它将自左至右依次处理。仅当任何一谓词项均不满足才夭折本次处理过程。在这种处理中，前项结果“好象”对后项有效和过程语言非常相似。

例12-10 最大公约数的欧基里得算法

最大公约数欧基里得算法可用三条规则描述:

$\text{gcd}(A, 0, A).$

$\text{gcd}(A, B, D) :- (A > B), (B > 0), R \text{ is } A \bmod B, \text{gcd}(B, R, D).$

$\text{gcd}(A, B, D) :- (A < B), \text{gcd}(B, A, D).$

其中A, B是输入参数, D是输出参数。第三条规则的执行, 非常象if(A<B) then gcd(B, A, D)。当测试(A<B)为'真'时调用谓词gcd(B, A, D)。如果成功结果值束定于D。

第二条规则相当于执行了两次 if 测试, 做一次赋值(将A mod B束定于R), 再做一次谓词匹配, 结果值束定于D。

12.4.4 封闭世界内的演绎过程

查询建立了演绎过程的目标, 它以项的逻辑“与”连接给出。Prolog逐一满足查询中的各个子目标。子目标也可以是含变元的谓词, 它首先到数据库中查变元个数(目)相同的规则头, 如果找到, 则暂时列出匹配的变元实例。接着处理规则体的另一个子目标。看找出的实例能否满足新的子目标, 如果成功, 继续匹配下一个子目标, 直到所有子目标均满足。因为各子目标是“与”的关系。

如果有某个子目标查遍数据库也找不到能满足的事实, 该子目标失败, 但不等于整个目标的失败, 它就回溯到上一个子目标重选事实继续匹配。这种回溯的执行过程和例题我们在第7.4.2节(2)中已介绍过了。即使是整个目标最后失败, 也不等于这个目标追求的命题是否定的, 因为限于数据库存放的规则和事实有限, 它是“封闭世界假说”之下的失败。

回溯是在树行模型之下以递归下降法自动进行的, 也就是说, Prolog实现程序计算的算法相对固定。程序员也就没有必要对程序作出什么显式的控制, 它的程序控制是隐式的。提高程序质量就在于程序员如何安排一个规则内条件的次序了, 各规则之间的次序并不重要。这是因为它采取的是深度优先, 穷举搜索, 不到整个数据库按规则子目标所有可能的组合全部查完是不能说出fail(失败)的。

12.4.5 函数和计算

Prolog解题是模型一个系统, 从输入开始以某种给定的形式希望能导出输出。对于其它语言这恰好是定义一个从输入到输出的函数, 设计算法(安排对数据结构的加工步骤), 做过程式程序设计。Prolog是公理化的语言, 它只以公理化的方式描述输出结果, 并用证明过程代替传统的计算过程, 即它找出数据库中具有和所希望输出的对象有同样性质的对象, 证明它们是一致的。然而, 作为一个程序设计语言它也要有最低限度的计算。

(1) 函子完成逻辑设计中的计算

Prolog为整数预定义了五种运算符, 它们都是计算函数, 所以, 函子以结构形式出现, 如:

| | |
|-------------|-----------------|
| 中缀表示 | 前缀表示 |
| $X + Y * Z$ | $+(X, *(Y, Z))$ |
| $A - B / C$ | $-(A, /(B, C))$ |

中缀和前缀表达等效, 前缀表达虽可按谓词理解, 两变元具有+、-、*、/、mod关系, 但它不求真值而有求值结果, 所以, 它不是谓词仅仅是一特殊的结构:

<函数名>(<变元>, ..., <变元>).

函数求值的结果一般通过谓词is(<变元>, <表达式>)绑定到变元上, 也可以写成中缀。它的意思是表达式(复合的函数结构)计算的值返回到变元之中, 该变元以表达式的值实例化。所以, 若要做计算, 一条规则中至少要有一个is子句, 前述最大公约数程序的第二条规则:

gcd(A, B, D);-(A>B), (B>0), R is A mod B, gcd(B, R, D).

其中A mod B就是计算余数的, 计算后以返回值例化R, 再代入后边的递归谓词。

用户定义的带数值计算的谓词, 必须增加一个存放返回值的变量。如求最大公约数的欧基里德算法是两数(变元)辗转相除, 只要两个变元。上式gcd(A, B, D)变元有3个, D就是为了返回值, 是gcd的输出变元。

把函数改写为约束, 很容易写出program程序, 请看下

例12-11 求斐波那契数的Prolog程序

斐波那契函数以下述公式生成以下数列:

1, 1, 2, 3, 5, 8, 13, 21, ...

Fib(0) = 1

Fib(1) = 1

Fib(n) = Fib(n - 1) + Fib(n - 2)

编制Prolog程序时这样分析:

第一、二式是事实也是公理, 把结果值作为变元照写。第三式说明, 若n为斐波那契数, n-1和n-2的斐波那契必须成立, 且这两个数之和是n的斐波那契数, n>1。于是有Prolog程序:

Fib(0, 1).

Fib(1, 1).

Fib(n, f) :- Fib(m, g), Fib(k, h), m is n-1, k is m-1, f is g+h, n>1.

当有查询 ?- Fib(5, f)时, f返回8。

(2) 逻辑程序的算法表达

算法怎样用公理表达呢? 我们拿一个最典型的Quicksort分类程序讨论。分类在表上进行。我们可以按快速分类写出它的公理。

quicksort(未分类表, 分类完的表) :-

(从未分类表拿出第一元素, 以它为基准, 分成两个表), [1]

quicksort(小表, 分类完小表), [2]

quicksort(大表, 分类完大表), [3]

append(分类完小表, 基准元素和分类完大表, 分类完总表) [4]

这样把快速分类的总目标变成了四个子目标, 第二、三个子目标显然清楚, 继续递归分类。第四个是以连接谓词append把它们连成一个分完类的总表输出。第一个子目标是将一个表分解为两个表, 我们细化如下:

先构造一个“劈分”谓词, split(基准元素, 未分表, 小于基准元素的表, 大于基准元素的表)。前两变元输入, 后两个输出。谓词意义明显, 但算法过程还没说清如何分。于是, 我们从待分类表拿出第一元素, 若小于基准元素放入小表, 否则放入大表, 接着再重复直至都成为空表。最后的Prolog代码如下例:

例12-12 快速分类的Prolog代码

r1 split(_, [], [], []).

r2 split(Pivot, [Head | Tail], [Head | Sm], Lg) :-
Head < Pivot, split(Pivot, Tail, Sm, Lg).

r3 split(Pivot, [Head | Tail], Sm [Head | Lg]) :-
Pivot < Head, split(Pivot, Tail, Sm, Lg).

r4 quicksort([], []).

r5 quicksort([Head |], Head).

```

r6    quicksort ([Pivot | Unsorted] AllSorted):-
        split (Pivot, Unsorted, Small, Large),
        quicksort (Small, SmSorted),
        quicksort (Large, Lgsorted),
        append (SmSorted, [Pivot | LgSorted], AllSorted)。

```

规则r1, r4虽然是递归到头的一条公理,但也起到声明数据结构的作用。请注意从表中取出元素的技巧,人为写上[Head | Tail]它将表的第一元素束定到名字Head,其余部分束定到Tail名字上,由此分出Pivot, Head直至递归分完。

规则r2, r3的‘<’谓词,是实现条件选择很典型的方法,规则r5是处理只有一项的表的特殊情况。

(3) 逻辑和控制分离

Prolog无通常意义的控制结构,也就是该程序动作次序(显然也有)和计算的子句逻辑没有必然的关系。例如,把例12-12中, r4, r5, r6写在r1, r2, r3前面并不影响本程序的执行结果。如有查询它总是从第一个规则到最后一个规则查找匹配。但是,写得好与不好却影响效率。为了效率总是把易于找到匹配的规则写在最上面。控制优化可以大为改善效率。程序逻辑和控制的无关性决定了逻辑程序设计方便易行:先分析程序逻辑,解决正确性问题。再分析程序效率,优化控制,使程序完善。但要注意,一味追求高效会丧失易读性。例如,例12-12中, r1, r4是到达递归边界的规则,完全可以放在r3, r6之后。但r1, r4有表达数据结构的功用,如r1是有一个变量,三个表作变元的谓词公理。一上来就是r2倒反看不清晰。所以,例12-12是递归逻辑程序惯用的写法。

12.4.6 cut和not谓词

如前所述,超级归结依赖cut(割断)操作使归结证明人为截止,因为Prolog的归结模型只能完整地证明正命题,是否有解无法判定。如果明知继续下去没什么意思了就人为截断。显然,一旦提供了这个机制,它是一双面刃,用得不好会破坏证明系统的完整性。

(1) 安全cut

非形式解释cut,它如同一篱笆,由程序员任意置放在规则之中,以停止无意义的回溯。例如,有如下规则:

```
P :- Q, R, S, ! T, U, V
```

证明系统首先查找Q、R、S的条件合一, Q满足R不满足则回溯改Q,如此下去直至S也满足,控制经!传到T,此时满足Q、R、S的变量实例均予冻结。回溯只能在T、U、V之间,所有条件均满足本规则成功,否则失败,并不再回溯修改!号左边变量实例。

所谓安全cut即不可能导致可证明的目标失败,且大为改善效率。

例12-13 安全cut示例

求1到N的整数之和

```
r1    sum_to(N, 1) :- N=1, ! .
```

```
r2    sum_to(N, R) :- N1 is N-1, sum_to(N1, R1), R is R1 + N.
```

当有查询:

```
?- sum_to(1, X)
```

//匹配r1

```
X=1;
```

//打';'号由于有!不致无限查找第2个

```
no
```

```
?- sum-to(6, X)
```

//匹配r1失败,匹配r2连续r2

```
X=21;                                //直至成功，打';'号也不再找
no
```

其实本例r1写作事实sum-to(1, 1).也可以求出，但本例写做规则，它就要找右端子句匹配，当用户键入';'号时，如果没有‘!’，它将无限运行以查找第二个满足sum_to(1, X)的解。实际是不可能找到的。

(2) cut 实现not操作

在归结证明系统中，一般要求规则和查询都用非反条件，尽管not操作完全可以代替cut且程序行文中可读性更好，但用not时必须仔细，因为它是cut实现的，其定义如下：

```
r1  not(X) :- X, !, fail.
r2  not(_).
```

其语义是：若X为‘假’则条件not(X)成功。若X成功，not(X)失败。其推理过程是：

- 若X为假，匹配r1，在未达到！时已失败，则匹配规则r2，由于r2什么变元都可以且总为成功，所以，not(X)是成功的。
- 若X为真，匹配r1后，X为真，控制通过！传到fail，则r1失败。于是回溯到！过不去，只好失败。由于用了!就地失败，它不再匹配r2，故not(X)为假。

正是由于这个原因，谓词p和not(not(p))求值结果不能保证一样，有时not(p)和not(not(p))求值结果倒是一样的，以下是not谓词出毛病的例子：

例12-14 不可靠的not谓词

假定一规则test有以下定义：

```
test(S, T) :- S = T.
```

运行以下查询时有：

```
?- test(3, 5)
no
?- test(5, 5)
yes
?- not( test(5, 5) )
no
?- test(X, 3), R is X+2.
X = 3
R = 5
?- not (not test (X, 3))), R is X+2.
! error in arithmetic expression : not a number
```

由于第二次not(外部的)求值时用到上例规则r1，其中X是not(test(X, 3))的结果值，故X+2不是数加2。

这个问题原因在于子句逻辑的不可判定性。即我们有时可以证明理论T，有时可以证明它的反，not T，有时两者均不可证明。如果是后者，说明定理只适用于某些模型而对另一些模型不能用。我们不能因不能证出T为真而断言T为假，但Prolog确是这样，不能证明T为真则断言not(T)成功。所以，使用not要特别小心。

(3)不安全的cut

在人工智能领域的启发式策略中，证明树有时极其庞大，程序员就要割掉部分明知无解的推理树，有时由于机器太慢，树虽不大，时间太长还非这样做不可。但程序员稍有疏忽就会酿成大错，把仅有的解都割了去。这时就破坏了证明完整性。cut使我们处于两难的境地，它的高效是以风险为代价得到的，如同60年代goto技巧对非结构化程序。只要模型是超级归结，cut的两面性是不可以解决的。

12.5 Prolog评价

Prolog提供一种证明风格的声明式程序设计，推理清晰，概括能力强，程序和数据没有明显分离。在复杂的人工智能程序中，简明的表达有利于程序员写出好程序。特别是逻辑和控制分开的正交性简化了正确性证明，因为程序员只需关注证明的逻辑部分，而优化控制提高效率并不影响证明的正确性。

Prolog程序具有自文档性，由于论域直接就是问题域，谓词清晰描述了论域对象的特征与关系，而实现是单一的递归下降匹配算法。没有影响描述的过多实现细节，自文档性好是必然的。它的非过程性和没有“隐藏的语义”使程序更具有安全性。即程序员控制整个程序比较方便，且不需了解更多实现细节。

在程序员尚不清楚如何组织数据和计算过程的应用中，Prolog有极大优势，因为它证明的是后果而不是过程，且对程序的顺序性要求不严。只陈述应如何如何，要如何如何，并未详细安排计算过程。

正是由于非过程性，它也成为潜在的并行程序设计语言的候选者，所以，当今在高度并发的连接机(Connection Machine)上，采用Prolog作为软件语言。

尽管由于编译技术的改进(有了编译的非解释型Prolog)，Prolog程序的效率由于依靠参数束定的合一匹配，它的效率仍不及传统过程语言。也正是由于它的声明性质，程序员在优化算法时作用有限。因为它的基本推理算法是确定的，且程序正文看不出计算过程，以不变的递归下降算法应万变，当然就没有过程式语言设计精巧算法效率高。

此外，Prolog的基本推理规则基于最简单的形式逻辑，if_then_else，所以它的功能过程式语言很容易实现，效率大为提高，只是表达不如Prolog清晰，所以，近年专家系统的开发工具不少以C语言实现，这样做还利于得到C语言环境工具的支持。

第二个不足是，一般复杂的大型系统一开始很难作为证明系统开发，程序不大运算量惊人。而Prolog本身也只有局部量，天生来也不是大型软件开发的工具。因此，Prolog只能作为逻辑程序设计的独枝存在，解决大型应用多范型语言是个出路。

12.6 小结

- 逻辑程序设计的基本观点是程序描述论域上对象的属性及其相互关系，对象和对象、对象和属性及其相互关系。对象和对象、对象和属性的连系就是事实，事实之间的关系以规则表述。根据规则找出合理的事实叫推理。

- 谓词以公式表达对象及对象间的关系，谓词公式中的常量、变量、函数、谓词、查询按一定逻辑规则的变换(等价置换)即谓词演算。

- 子句逻辑是经典逻辑的特例，任一谓词演算公式均可化为'或'连接的子句的集合(子句内部由'与'连接)。

- 证明系统是应用公理演绎出定理的合法演绎规则的集合，公理即证明的前题是事实和规则的集合。演绎也叫归约，是证明系统中合法规则的一次应用。演绎的结果是从前题(公理)导出逻辑结论。证明是一个语句序列，证明过程是要么使每个语句演绎成公理，要么演绎为前此导出的定理，反驳是反向证明一个语句是矛盾。

- 一个语句若能从公理推演出来称定理，定理集合称理论。理论不含相互矛盾的定理。

- 理论要借助模型得到语义解释，模型即特定论域上的约定。一般说来，一个理论可以有多个模型。

- 命题演算是谓词演算的特例。谓词演算对于真命题的证明是完整的，但是否能找到证

明是不可判定的。不存在，也不可能找到一个算法验证非真命题。

- 经典逻辑从前题到结论的推理一般采取穷举证明技术，在实现上效率是根本问题，因为各公理组合实现的推理树往往是天文数字，即证明系统的组合爆炸问题。

- 基于子句逻辑的归结证明技术以归结—合一方法证明反命题恒假来说明命题为真。归结原理是将两个含有正逆命题的子句消解为更简单的归结式，如果归结得不出矛盾要无休止地归结下去直至空间耗尽。反之原命题得证。

- 对命题形式进一步限制(用Horn子句)和利用切断技术(cut 操作)是能付诸实用的超级归结法。它是Prolog的理论模型。

- 逻辑程序设计的风格不是描述计算过程而是证明过程。一般构造一个希望的解，证明它就是所希望的解。构造过程就实施了计算。第二个特点是描述性。第三个特点是大量利用表的数据结构和递归。

- Prolog程序公理部分是事实和规则(无条件条件和条件子句)，查询是求证目标。它在封闭世界(限于规则和事实库)完成证明的演绎过程。回溯是实现各子目标同时满足的唯一方法。

- 子句逻辑的不可判定性要求规则和查询用非反条件，且慎用not操作。

- Prolog的优点是自文档性、非过程性、逻辑表达能力强。描述性使程序清晰、具有潜在的并行性。Prolog的缺点是只适合较小程序。计算效率不高。

习题

12.1 试述逻辑程序设计的基本风格与过程式程序函数式程序的同异。

答：无论是命令式还是函数式程序都把程序看做是输入到输出的某种映射，为了实现这种映射，程序要对数据结构实施某个算法过程，算法实现该程序功能。而算法又是以程序语言提供的控制机制实现计算逻辑。但这种逻辑是在程序员心里，被隐式体现在程序正文中，为此，程序正确性的证明还要把它隐含的逻辑以断言形式地写出来。而逻辑程序设计的范型是陈述事实，制定规则，程序设计就是构造证明。程序的执行就是在推理。

它们的共同之处在于：程序描述的都是数据对象之间的关系，只是逻辑式程序设计抽象层次更高，而不仅限于函数（映射）关系。

12.2 试述归结证明的基本思想。为什么要超级归结？

答：归结法是命题演算中对合适公式的一种证明方法。其基本思想为：为了证明合适公式F为真，归结法证明 $\neg F$ 恒假来代替F永真。归结原理是：设有前提 $L \vee P$ 和 $\neg L \vee R$ ，则其逻辑结论是 $P \vee R$ ，这个推理把两个子句合并消去一对正逆命题。将新的子句再加入归结子句集中，继续归结。若最终归结式为 \square 则表示此次证明的逻辑结论是矛盾，原待证结论若不取反则恒真，命题得证。

使用超级归结的原因是：由以上思想可知，如果得不出矛盾，那么归结法要无休止地做下去，中间归结式也越多，匹配查找次数越多，每一步都做长时间计算。于是想出比这种原始归结更好的办法，即利用切断操作，并对子句形式进一步限制即为超级归结法。

超级归结实际上是将无条件Horn子句中的谓词符号和条件子句中的对应谓词符号合一。

12.3 何谓变量例化？在演绎过程中变量如何实例化？

12.4 何谓命题演算，谓词演算，它们的同异。

12.5 谓词与关系的同异？

答：关系是泛指事物之间的相互关系，从逻辑程序设计的观点来看，关系包括事实（即对象和对象，对象和属性的联系）及规则（即事实之间的关系）两种，指数据对象具有的性质及不同对象的联系。

谓词是关系的特例，是因公式表达的对象间的关系，即某种事实，是表征某种对象性质的符号。

12.6 以下限定量词是否为真？若为真取反。

[1] $\forall Y \text{ even}(Y)$

[2] $\exists X \text{ even}(X)$

[3] $\forall X X = X + 0$

[4] $\exists X \forall Y X = Y + 1$

[5] $\forall X \exists Y X = Y * 1$

12.7 公理和理论有何不同。

答：公理是用谓词演算公式来描述的事实。理论是从某一公理集合导出的所有定理集合，所以理论是公理的演绎结果。

12.8 定义一个理论举出它有多少个模型。

12.9 何谓证明系统？何谓反驳？

证明系统是应用公理演绎出的合法演绎规则的集合，反驳是一个语句的反向证明，它证明一个语句是矛盾的，即不合乎规定的公理。

12.10 试证Horn子句的完整性。

12.11 给出事实和规则：

flower(crocus, spring, white).

flower(voilet, spring, blue).

flower(iris, summer, blue).

flower(rosr, summer, red).

flower(marigold, summer, orange).

tree(holly).

color(F, C):-flower(F, S, C).

color(T, green):-tree(T).

pretty(F):-color(F, red).

pretty(F):-color(F, blue).

grows(X):-tree(X).

grows(X):-flower(X, Y, Z)

求能与以下子句合一的实例化集合：

[1] flower(F, spring, Y), pretty(F).

[2] flower(F, summer, Y), pretty(F).

[3] flower(E, Y, orange), pretty(E).

[4] grows(X), pretty(X).

答：[1] flower(F, spring, Y), pretty(F)

F=voilet, Y=blue

No

[2] flower(F, summer, Y), pretty(F)

F=inis, Y=blue,

F=rose, Y=red

No

[3] flower(E, Y, orange), pretty(E)

No

[4] grous(x), pretty(x)

x=violet

x=inis

x=rose,

No

12.12 用prolog写出完整的矩阵相乘程序。

domains

Vt=integer *

Ut=integer *

```

predicate
    form_Vector(Vt, integer)
    form_Matrix(Ut, integer, integer)
    Matrxinx_Multi(Ut, Ut, Ut)
    Vm_Multi((Vt, Ut, Ut)
    Vector_Multi((Vt, Vt, integer)
Main_fun(Ut)
Clavses
form_Vector([H|T], X):-0
    readist(H),
    form_Vector(T, X-1),
    form_Vector([], 0).
Matrix_Multi([H|T], L2, [H3|T3]):-
    Vm_Multi(H1, L2, H3).
    Matrix_Multi(T1, L2, T3).
    Matrix_Multi([], _, []).
    Matrix_Multi(_, [], []).
Vm_Multi(L1, [H2|T2], [H3|T3]) :-
    Vector_Multi(L1, H2, H3).
    Vm_Multi(L1, T2, T3) .
    Vm_Multi([], _, []).
    Vm_Multi(_, [], []).
Vector_Multi([H1|T1], [H2|T2], [H3|T3]) :-
    S1=H1*H2,
Vector_Multi(T1, T2, S2) :-
    S=S1+S2.
Vector_Multi([], _, 0).
Mach_fun(1):-
    Readint(X,)
    Readint(Y),          X>0, Y>0,
    form_Matrix(L1, X, Y),
    form_Matrix(L2, X, Y),
    Matrix_Multi(L1, L2, L).

```

12.13 总结prolog中cut的用法，怎样是安全的？怎样是不安全的？

12.14Prolog程序的模块性体现在哪些方面？

12.15写一个C语言的快速分类程序，比较例12-12的Prolog程序，你能得出什么结论？

```

int k[30]
Void main( )
{int I;
  for(I=0;I<30;I++)
    Scant("%d",&k[I]);
  Quick_sort(0,30);
  for(I=0;I<30;I++)
    printf("%d\n", k[I]);
}
void quick_sort(int m,int n)
{int temp: i,j,t;

```

```
i=m;
j=n;
temp=k[i];
do{do i++;
    while((temp>k[i]&& i!=n)
    do j--;
    while((temp>k[i]&& j!=m)
    if(i<j)
    {
        t=k[i];
        k[i]= k[j];
        k[j]=t;
    }
}while(i<j)
t=k[m];
k[m]= k[j];
k[j]=t;
quick_sort(m, j-1);
quick_sort(j+1, n);
}
```

结论：prolog解题是模拟一个系统从输入开始从某种给定的形式希望能导出输出。对于其他的语言，这恰好是一个从输入到输出的函数。设计算法，做过程式程序设计，prolog是公理化的语言，它是以公理化的方式描述输出结果，并用证明过程代替传统的计算过程。

第 13 章 程序的并发性和进程交互原语

前面章节我们已多次提到并行(parallel)命令和并发(concurrent)程序,但主要是讨论顺序程序。所谓顺序程序是指一个程序作业从起始到终了的每个作业步骤顺次地执行完毕。程序员写程序时就默认有一台顺序执行的机器,为它排出先后执行的动作步(语句)。然而,自然界发生的事,大多数是并行的。只是我们在讨论之中把它孤立和顺序化。这是为了早期计算机昂贵,一台机器 CPU 只能顺序地执行,自然是首先发展顺序计算。在实时控制领域,即使在早期也无法顺序化。例如,飞行姿态控制程序,每时每刻都要计算敌我双方飞机的速度、高度、方向才能决定控制措施。再如,民航机票发售系统,各站点售票是同时进行的,既让乘客随意订座还不许卖重了。这都要求多个计算同时平行地执行,并协调一致。我们称这种程序为并发程序(concurrent)。显然,并发的前题是并行(平行, parallel)。然而,并发和并行的概念在不同文献上也少有出入。本书把“并行”定义为程序可相关或不相关平行执行,有时仅指不相关的平行执行。“并发”执行必然相关。

由于近年硬件成本大幅度下降。网络计算快速发展,用多台机器,多个 CPU 完成一个计算是极普通的事。因而原先人为顺序化的顺序计算就没有必要了。而且今后为了利用信息资源,一上机就上网是应用计算机主流。并发程序设计、并发语言将成为主流。顺序程序倒成了并发的特例。

操作系统本身是一个高度并行的软件,它本身一般是某种高级顺序语言(如 C)扩充了一些支持并发进程低级原语,加上汇编语言编写而成的。在这个意义上,这些扩充了低级原语的语言就是并发高级语言。

其它更高层的通信机制是建立在这些低级原语上的模块(相当于该语言的库模块)语言的机制与特征并没有扩充,编译也没有大改动。

完全独立于操作系统,在高级语言内提供全套支持变形程序设计的并发,高级程序设计语言是 Ada,因为它要编制机载、弹载的嵌入式程序。这些程序从单板、单片开始,编制监控程序(小 O.S)连用应用程序。这种语言并不多见。

因此,研究并发高级程序设计语言,勿宁说首先研究并发程序设计。搞清了各种并发模型和通信机制再去看已有语言的扩充就很容易理解和实现了。

本章介绍并发程序设计的基本概念。并发程序带来的问题和要解决的基本问题。基于共享变量和基于消息传递的两类并发机制反映了不同物理模型,它们共同要解决的是对共享变量的互斥访问和进程间如何通信协调。

本章着重介绍概念和术语,是下章讨论各种并发机制的基础。

13.1 基本概念

并行程序是同时执行两个或多个程序,在程序执行完之前必然有时间上的重叠。我们知道,一个程序的一次执行叫做一个进程(process)。研究并发程序首先要了解进程及其相互作用。

13.1.1 程序与进程

源程序经编译、连接编辑成为可执行代码。它们是可执行的“半成品”,可以作为用户文件寄存于外存。可以作为库文件和系统文件。一旦需要则调入内存,然后开始执行。进程是个动态概念,即程序以一组数据的一次执行。我们先看看进程在内存中的执行状态及实现过程。就单个

进程而言,它有四种状态:

- | | |
|-----------------|------------------|
| • 就绪 ready | 可执行代码装入内存立即可运行。 |
| • 运行 running | 执行进程。 |
| • 阻塞 blocked | 停止本进程执行,随时可恢复执行。 |
| • 终止 terminated | 停止,且不可恢复执行。 |

除了这四种状态而外,控制进程的逻辑操作是激活(activating)触发(triggering)和中断(interrant)。所谓激活是创建一个进程并使之进入就绪或立即运行状态。所谓触发是使就绪或阻塞状态转入运行态。所谓中断即使运行的进程转入阻塞或终止态。然而,这些逻辑操作是机器指令层次上的,在语言层次上借助于各种原语(primitive)或约定实现。所谓原语是程序语言中定义的例程名。例如,早期的操作系统低级原语 fork 和 quit 就是用来控制进程状态的。一个进程调用 fork 例程,则中断本进程创建新的子进程并执行之。一个进程调用 qait,将终止本进程。再如, cobegin...coend 或 par...end 命令,并发程序的开始执行,用某种约定自动激活各个进程,如 Ada 主控程序开始所有任务均激活。一个进程终止则自动触发阻塞的父进程使之执行。

在更低的实现层次上中断分为外部中断和内部中断(由外部触发和本进程触发)。触发中断后,调用中断处理器(interrupt handler)例程,再调用原语例程,并将原语例程要求创建或终止的例程提交调度器例程,完成进程控制。其控制流示意图如下:

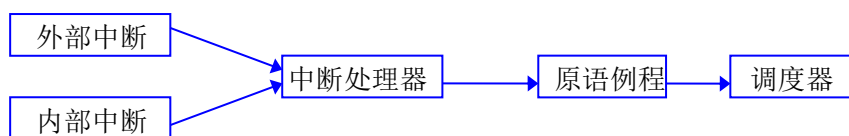


图 13-1 进程执行控制流

我们把一个进程执行中再次创建的进程,称为子进程。一个进程可多次创建子进程,一次可创建多个子进程。子进程还可以创建子进程。如果被创建的子进程不分配一套资源则称线程,例如,多 CPU 机上主进程将大型科学计算并行分配到各 CPU,且共享同一内存。如果分配资源,如一般程序进入打印语句,为此要分配打印机、驱动程序资源,则称子进程。

13.1.2 并行政式的模式

进程是程序执行的控制流,众所周知,代码执行要加工数据,与进程执行同时存在的还有一个数据流。根据 Flynn 的分类法:

- 一组可执行代码装入一个机器内存后,以一个 CPU,一组数据执行一次。它属于 SISD 执行模式(即单指令流单数据流的简写)。物理上对应为单处理器的顺序程序。
- 一组可执行代码装入后,可以依次执行多个进程,它属于 SIMD,单指令流多数据流。对应为单机多处理器的主机或单 CPU 的分时系统、阵列机组。
- 在多机或多处理器上各有自己的可执行代码。协同完成一组数据的计算,是 MISD 多指令流单数据流系统。对应为分布数据流机。
- MIMD 则为多指令流多数据流系统,对应为一般分布式系统(有多个不同的处理机,运行各不相同的进程)。局域网和广域网就属于此列。如果网上协同运行一个程序作业,则为以 MIMD 系统实现的并发程序。

这四种模式包括了单机单处理器，单机多处理器，同型多机多处理器(阵列机)和分布式多机多处理器(异质机联网)的各种物理实现。MISD 一般并发系统不多用，并发程序主要在 SIMD, MIMD 上实现。MIMD 在实现多机协作计算时又可以分为两类：共享存储和分布式存储。共享存储多用于同类多 CPU 的单机上，所有 CPU 处理的进程都共享公共的数据。这样，各进程间因数据共享而紧密耦合。进程间的关系最初是主/奴（master/slaver）式，即 OS 的核只执行某个‘主’处理器（进程），它统管共享数据并派遣任务到各‘奴’处理器的进程。优点是设计控制简单，缺点是主进程易成瓶颈。当今最流行的是对称多处理器，即 OS 的核可以执行任何处理器，每一处理器都是自调度（self-scheduling）的，即从待执行进程表中取出一个执行。它也派送子进程给其他处理器，也接收其它处理器发出来的子进程，故称对称多处理器 SMP。并行处理器谱系如图 13-2:

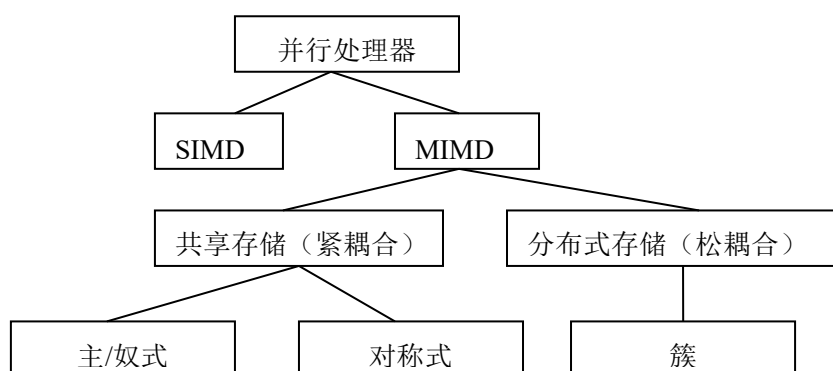


图 13-2 并行处理器谱系

由于分布式存储是松耦合的计算机群体，它对应为多计算机的簇（cluster），和一组计算机的集合不同之处在于：它们各自的存储是被大家共享的，它们互连，每个计算机只是“整个”计算机中的一个节点，是今后高性能、可伸缩、高可靠性计算机的发展方向。

13.1.3 线程与进程

早期计算机一个进程就是一个执行的线索，它可以和其它进程交替执行，即又开始另一线索。执行一个线索是断断续续的（冻结、就绪、运行、终止态等），完全由 OS 调遣。一个进程可以看作是 OS 派送的单元（Unit of dispatching）。此外，进程执行占有资源（数据、设备、CUP 的时间片），所以进程又可以看作资源拥有单元（Unit of resource ownership）。后来发现，这两种单元属性是相互独立的。在一个资源拥有单元之下可以派生出多个派送单元，即多线程执行。它们同样可以交互，这就是线程。

线程是共享资源的轻量级进程（lightweight process），它也是有线程执行状态，也有其静态存储和局部变量。

传统的 OS 支持单线程的计算模式。单用户的 MS-DOS 和多用户的 UNIX 就是例子，即使 UNIX 是多线程交互，每一进程之中只有一线程。如图 13-3 左侧图示。右侧上图，一个进程多个线程对应为 Java 虚机的计算模式。而当今所有 OS 均发展为右下角的多进程和多线程的计算模式。



图 13-3 线程与进程计算模式分类

正是由于线程具有进程的所有派送特性。当不涉及资源派送时，线程交互和进程交互是一样的。下文讨论只谈进程交互。

13.1.4 原子动作

并发和抽象一样可以在不同层次研究它。一个数据占据 32 位，我们如要拷贝它，可以从第一位拷贝到第 32 位。也可以同时拷贝 32 位(在字位级并发)。一个表达式有若干项，每项同时计算最后汇总求值(子表达式级并发)。同样并发可以在语句级，程序块级，程序单元级，模块级…。在级以下则认为是原子的，不再分成子部分并发执行。原子动作是一次“立即”执行完的“顺序”动作。至于是否真正不再分就不一定了。例如，一般顺序程序的输入/出进程和主进程都是并发执行实现的。如原子动作定义在程序级上，它们也是“立即、顺序”地执行的。并发的讨论则在此级以上。原子动作在进程内完成，一个进程可以有多个原子动作(但不得有半个)。

在高级语言层次上，原子动作一般定义在语句级的事件(event)上。所谓事件，是本程序表示的状态有了变化。例如，执行了赋值语句，作了初始化、调用、终止…当然，事件也是相对的，一个语句对数据的加工可以是一个事件，若干语句的一个循环，也可以是一个事件。

例 13-1 PL/1 的多任务

PL/1 的并发进程是任务 TASK，它可以定义语句级的事件。P 是一个进程，它并行执行 Q 进程，则 P 进程的正文可以写：

```
DECLARE X EVENT
```

```
:
```

```
CALL Q(APT) TASK (X) //激活 Q
```

事件变量 X 的一次取值表示一事件，它可取值 IN __ EXCEPTION(相当于 true)和 TERMINATED(相当于 false)。当 P 进程执行到 CALL 时激活任务 Q，实参表 APT 和 Q 的形参表匹配，Q 进程和 P 进程并行执行。事件变量 X 调整两进程的同步。X 变量是 P, Q 共享的。

13.1.5 进程交互

并发程序主要是研究进程之间的关系。所谓交互即两进程有数据或操作通信。如甲进程向乙进程发一条消息送给它数据，或触发乙进程中另一操作，这叫直接通信。如果甲进程据有某资源改变了该资源中的数据，以后乙进程也据有该资源，这些改变了的数据对乙进程有了影响，这是共享数据的间接通信。一般说来，并行进程有三种类型：

(1) 独立进程 两进程并行但不相关。

设进程为事件序列，若有 C, K 两并行进程，可表达为 $C \parallel K$ 。其中：

$$C = \{C_1, C_2 \dots C_n\}$$

$$K = \{K_1, K_2, \dots K_m\}$$

独立进程是两进程内任何事件 C_i, K_j 的执行都不依赖对方。因此与进程执行速度无关, $C \parallel K$ 执行时在时间上可任意复盖, 也可以按 $C;K$ 或 $K;C$ 或 $\{C_1, C_2, K, C_3, K_2 \dots K_m \dots C_n\}$ 秩序任意交错执行, 均能得到正确结果。

(2) 竞争进程 两进程竞争同一资源。

设 C, K 是两竞争资源的进程, 事件 C_i, K_j 要使用同一资源 r 。则必须保证 C_i, K_j 的互斥访问 (mutual exclusion), 即两(访问)事件时间上没有复盖, 同一时间只有一个事件据有资源 r , 按 $C_i; K_j$ 或 $K_j; C_i$ 次序执行均可。我们把据有并加工资源的代码单划出来, 叫做临界段(Critical section)。一般说来, 这段代码是同一个, 所以 C_i 中有 CS, K_j 也有 CS 。显然 CS 两次执行, 谁先谁后对各自进程计算结果是不一样的。因而 $C \parallel K$ 结果不确定。如果推广到多个进程竞争同一资源, 情况更难预料。因为各进程输入的数据只有运行时才知道, 因而, 进入临界段的时间不确定。此外, 为了保证互斥则各进程均设进出临界段的协议。竞争进程一般形式是:

| | |
|---|---|
| \textcircled{C} : loop 入口协议 临界段 出口协议 非临界段 end loop | \textcircled{K} : loop 入口协议 临界段 出口协议 非临界段 end loop |
|---|---|

入口协议一般是按所设共享变量(多为布尔型)判断能否进入, 出口协议则改置正确值。为确保进程的确定性, 利用共享变量“通信”协调。

(3) 通信进程 两进程有协议的信息交换。

设 C, K 定义如前, C_i 必须先于 K_j (K_j 要用到 C_i 的结果)的执行, 即其它事件先后无所谓, 一定要保证 C_i, K_j 的执行顺序。对于多个进程则有如 UNIX 和 DOS 的管道命令:

$C_1 \mid C_2 \mid \dots \mid C_n$

规定事件先后。它是通信进程的一种。实现通信的机制有许多种: 同步的、异步的; 单向的, 双向的; 定向的, 广播的。这与环境提供的运行机制和并发语言设计需求有关。也正是程序并发性要研究的问题。我们这里只给出初步概念。

- 同步(synchronous)通信 指两进程进度各不相同, 但必须同步到达通信点(注意不一定同时, 同时是实时(realtime)程序的概念)。若一方未到, 另一方等待, 直至完成信息交换。交换后各自执行各自进程则为单向同步通信。如果交换后, 发送方一直等待接受方执行的结果, 拿回结果后再各自执行自己的进程为双向同步通信。

- 异步(asynchronous)通信 一般要借助相当大的邮箱。两进程以各自速度执行, 发送方有了信息投入邮箱, 并继续执行自己进程。接受方在认为合适时从邮箱获取信息。一般不竞争邮箱且为单向通信, 当然也可做成双向的。

- 定向/广播式通信 所谓定向是发送方指明接受方。而广播式通信发送方只向公共信道发送信息, 任何共享该信道的成员均可接受, 所以是异步通信、单向的。

13.2 并发程序带来的问题

一般说来, 顺序程序和并行程序, 与程序执行的速度无关, 软件制售商在任何速度的机器开发出的商品, 可以到其它速度的机器上运行而不影响其结果。并发程序可不这样, 因而, 并发程序是比较难开发、测试和移植的。我们先讨论它带来的新问题。

(1) 速度依赖

并发程序执行结果, 取决于顺序成分进程执行的相对速度。对于并发且有实时(real time)要求的程序, 执行结果还取决于绝对速度。

竞争进程最为明显的, 例如, 一个进程对资源 r 中的变量 x 求平方; 另一进程是对 x 作加一操

作。结果值可能是 $(x+1)^2$ (后者快、前者慢)或 x^2+1 (前者快后者慢)。显然,速度依赖导致结果不确定。即使是速度相同的多 CPU 处理机上,输入/出设备速度有了小扰动也会影响结果。

并发程序调整相对速度的办法是延迟快进程。把进程挂起来(进入悬置态)待到指定条件满足才唤醒该进程。其基本原语是:

`await<表达式> do <语句 | 进程>`

进程执行到本句,测试表达式不满足则不作 `do` 以后的操作。即用原语 `await` 实现条件同步。表达式即条件。

(2) 输入值依赖

同一并发程序两组数据输入可能会有很大差别。因为若有一个判断因值不同跳过一段程序(跳过某个进程)则会打乱原有速度依赖关系。使执行顺序难以预测。使并发程序难于设计、测试、修改,就要更多地依赖软件工程技术,加强文档管理。

(3) 不确定性

顺序程序两次同样值的测试,一般情况下都是一致的。即所说的再现。并发程序因上述原因往往没有确定的结果值。对于有副作用的函数或表达式这种先后次序的差异影响则更大。所以,软件工程对副作用比较谨慎,并发程序是其原因之一。正是由于并发程序的不确定性。某一系统上测试通过的并发程序到另一系统上通不过。一组输入通过另一组数据通不过。这是常有的事。即使是同一套系统、同一语言处理器,同样一组输入,有时结果也不一样。因为这一系统不单运行这一并发程序,别的作业也会对程序干扰。只有在完全一致的条件下测试结果才可以再现。在个别临界值的情况下,出现间歇再现,所以,并程序的测试需要高超的技术。

(4) 死锁 (deadlock)

死锁是一种状态,由于进程对资源有互不相兼容的要求而使进程无法进展。表现为:

- 受到排斥 进程永远访问不到所需资源。
- 循环等待 进程资源分配链形成一封闭回路。它虽无明显把持,但对资源要求通过一组进程最后还是回到自己,不改为先释放已据资源,只能死锁。
- 无占先(no preemption) 进程无法放弃所占的、其它进程需要的资源。所谓占先,只要所据资源的进程未处于使用状态,另一优先级高的进程有了要求,则此资源被后者占去。
- 把持(wait and hold) 相互以占有对方资源为放弃已占资源的先决条件。

解决死锁的方法:

到目前为止还无法保证不出现死锁状态。即使采用无死锁的通信和同步机制。因为意想不到的条件出了错也会产生死锁。解决的方法是:

- 利用工具作静态死锁检测,可以避免或减少死锁出现的可能,事前防止。
- 或事前,让进程同时提出所有需要的资源,消除把持条件,或强行给资源排序,按此顺序满足要求,消除循环等待条件。
- 或事前,为调度程序声明最大的资源需求。少竞争资源死锁可能也小。好的调度程序资源稍大即可避免或少出死锁。
- 一旦出现,最笨的办法是重新启动,试换数据,找出原因改正之。这是事后重试解决。
- 一旦出现,找出死锁地点,夭折某些事件或进程,自动从死锁状态恢复。显然夭折的损失要求最小。为此,设置检测点,一段一段倒转(roll back)查找,直至段分得最小,夭折这个最小段。这对实时程序仍不可取。

(5) 死等 (starvation)

相互竞争的进程如果都满足进入某一资源条件,一般采用排队的先来先服务原则。相对最公平,但有的进程占用一种资源时间过长,致使其它资源长期闲置。适当地让它等待可以解放很多占时少而重要的进程,这样更公平。于是,除了先来先服务而外,在调度例程中约定或在条件中加入优先级表来达到此目的。调度程序则按此优先级和先来后到统一调度。如果优先级不当就会造成某些进程永远处于阻塞态,死等(但不是死锁)。死等是不公平调度引起的,解决的办法是在改

变某些进程的优先级，在公平性和合理性上作某种折衷。

13.3 并发程序的性质

安全性(safety)和活性(liveness)是一般程序均有的性质(程序属性)。所谓安全性是程序在执行期间不会出现异常的结果。对于顺序程序指其最终状态是正确的。所谓活性是程序能按预期完成它的工作。对于顺序程序指程序能正常终止。对于并发程序不仅如此还增加了新解释。

并发程序的安全性还要保证共享变量的互斥访问和无死锁出现。活性指每个进程能得到它所要求的服务；或进程总能进入临界段；或送出的消息总能到达目的进程，活性深深受到执行机构调度策略的影响。

正因为如此，公平性(fairness)也是并发程序重要性质之一。所谓公平是指在有限进展的假设下没有一个进程处于死等状态。调度策略如能保证每个无条件的原子功能均能执行则称无条件公平性。在具有条件原子动作时，若条件原子动作能执行并依然保持无条件公平性，则为弱公平性。条件原子动作一定能执行，则为强公平性。

13.4 低级并发机制和并发原语

无论程序设计语言上层采取何种机制实现程序的并发，最底层不外乎创建进程(装入内存、初始化使之就绪)；起动执行；阻塞(或叫冻结)；停止执行；阻塞父进程创建子进程；撤销进程等六种操作。这六种操作更低层的实现是机器指令。例如，停止和阻塞则利用中断陷阱(trap)，在机器指令中填入陷阱地址码。利用开关指令从一个进程跳到另一个进程。

原语是包含这些底层指令的例程。由于支持上层不同的并发机制，原语为了表述方便不同语言原语的差别在于所选组合指令的不同。

例如：

| | |
|-------------------------------------|------------------------|
| { fork/multifork | 分股创建多个子进程并执行。 |
| { quit/join | 合股新创进程回到原进程。 |
| { wait(e) | 等待 e 为真进入临界段 P 操作 |
| { signal(e) | 示信 e 为真临界段可执行, V 操作 |
| { sleep (value) | value 满足使所在进程阻塞 |
| { wakeup(value) | value 满足使所在进程唤醒(恢复执行)。 |
| cobegin s1 s2 ... sn coend | 开始多个进程 s1...sn 并发执行。 |
| { coroutine N | 指定协例程 N。 |
| { resumeM | 转入协例程 M。 |
| { send(Exp) to... | 将表达式值送至...进程 |
| { received(V) from... | 接受来自...消息，值由变量 V 传入。 |

13.4.1 基于共享变量的同步机制

如前所述，并发程序中进程交互分两大类，一为基于共享变量，一为基于消息传递。本小节先介绍基于共享变量的同步机制。

我们把加工进程共享变量的一组语句叫做临界段(critical section)。竞争进程竞争进入临界段。一旦进入就独享共享变量资源。从安全性而言，这是必须的。为了竞争进程互斥地访问共享变量。最简单的办法是另设一公共变量指示是否已有进程进入临界段。为此，早期的并发程序设置入口

协议和出口协议。在协议中设置、判明指示变量的值，以求保护临界段一个时间只有一个进程进入，一般形式见 13.1.4(2)。那么不能进入的进程就只好跳到循环末端，再开始一轮循环测试。直至用完此资源的进程在其出口协议中改变了指示变量的值，它才能进入。该进程据有所有足以运行的资源(占有内存，得到 CPU 的执行)周而复始地测试，实质上是在等待进入临界段。我们把这种等待称为忙等待(busy wait)。

(1) 忙等待

设我们把指示变量叫做 lock (锁)，每次测试临界段是否锁定。竞争进程以测定进入条件(锁)保持协调地进入临界段，我们说它在语义上保证了条件同步。锁就是条件，协调就是同步。请注意，此时未设同步原语。程序员也无法阻塞停止某个进程。如果有多个进程竞争进入临界段，则每个进程都要轮流测试锁。这就是著名的自旋锁 (spin lock)，其算法如下例。

请注意，以下给出的算法描述语言，类 pascal_Ada，不是可运行的某种语言的简化。

例 13-2 自旋锁

```
program SPIN__LOCK:
  var Lock := false;
process P1::
  loop
    when not lock do           // 条件同步
      lock := true;           // 入口协议
      临界段;
      lock := false;           // 出口协议
      P1 的非临界段;
    end do;
  end loop;
process P2::
  loop
    when not lock do
      lock := true;
      临界段;
      lock := false;
      P2 的非临界段;
    end do;
  end loop;
end P2;
process pn
  :
end SPIN__LOCK
```

其中 lock 为共享指示变量。本程序运行时每个进程同时启动。每一进程执行进入 loop 后首先查 not lock 只要有一个进程(设 P1)进入临界段工作，它把 lock 改成 true。其它进程不断测试，均不满足，跳到循环末端，周而复始再测。直至 P1 在临界段工作完将 lock 改成 false，则其它任何一个测试为 false 的进程即可进入临界段。同样，其余进程一直测试，均不满足只好等待。

忙等待在实现条件同步上是比较方便的。但不能保证互斥。对于分时系统竞争进程有个时间差还好一点，如果在多处理器的条件下，进程严格同时到达，对资源(临界段的加工对象)的竞争变成对指示变量查询和更改的竞争。要取决于操作系统对公用主存储器的存取访问的排序。如果某进程进入循环且正在更新 lock 为 true 期间，第二个进程又访问了 lock(为 false)，那么它也进入

临界段。互斥得不到保证。为此，寻找以忙等待实现互斥同步的算法，从 65 年到 81 年有许多名家(如 Dijkstra、Dekker、Knuth、deBruijn、Eisenberg、Peterson 等)上百篇论文，最后 peterson 的算法(1981)获得满意的解。算法如下：

例 13-3 忙等待实现互斥访问

```

program MUTUAL__EXCLUSION
  Var  enter1 : Boolean := false;
        enter2 : Boolean := false;
        turn   : String := "P1"           // 或赋初值 "P2"
process P1::
  loop
    enter1 := true;                      // 以下三行入口协议
    turn   := "P2" ;
    while enter2 and turn = "P2" do skip; // 跳至循环末端
    临界段;
    enter2 := false;                      // 出口协议
    P1 的非临界段;
  end loop;
end;
process P2::
  loop
    enter2 := true;
    turn := "P1" ;
    while enter1 and turn = "P1" do skip;
    临界段;
    enter1 := false;
    p2 的非临界段;
  end loop;
end;
end.

```

这个算法依然是忙等待，但用 enter1, enter2 turn 三个变量代替例 13-2 中的 lock 变量。其作用是保证互斥，即使 P1 进入 while 时，P2 更新了 enter2，只要有一点点时间差就可保证互斥。除非绝对一致(此时都在 while 上反复测试，谁也进不了临界段。形成死等)。因此一般情况下会保证正常互斥。

这个算法可保证无死锁且公平。任何一个想进入临界段的进程最终都可进入 *就是活性所要求保证的有限进展的假设。即若同步机制是公平的，则没有一个进程在等待无限长时间才能满足的条件。这个算法很容易扩充到多个进程。Peterson 并给出互斥、无死锁、公平性的操作证明。Dijkstra 1981 年也给出公理证明。

由于设置了多个变量，使用忙等待设计的同步并发程序，比较难读和理解，作正确性证明也麻烦。再者，忙等待比较浪费处理时间，因为在自旋处理周期中完全可以干点别的什么。最后，忙等待过于低级，程序员要从同步机制选取一直做到实现测试，设计起来比较麻烦且易出错。

(2) 信号灯

Dijkstra 首先理解到忙等待的低级和设计麻烦，提出了完整的信号灯(semaphores)理论(1968)。

信号灯是一个非负整值变量 s。在其上定义了两个操作 P, V(取自荷兰语字头，即 wait(等待)和 signal(示信))。V 操作发信号指示一个事件可以出现，P 操作延迟所在进程直至某个事件已经出现：

```

P(s) : await s>0 do s:= s-1;           // ‘await’ 表达延迟的表示
V(s) : s := s+1;

```

P 操作的等待(用原语表示), 判断、设置信号量一般看作原子操作, 一气呵成。可用硬件复合指令 TS(测试和设置)实现。信号灯变量 s, 当只有一个资源时取值 {0,1} 就够了, 此时称为二值信号灯。当有多个资源时初始化 s 等于资源数, 这时称通用或记数信号灯。

信号灯可保证进程互斥且公平, 如下例。

例 13-4 以信号灯实现的两进程互斥

```

Program MUTEX__EXAMPLE;
  var mutex : Semaphore := 1;
  process P1::
    loop
      P(mutex);           // 入口协议
      临界段;
      V(mutex);           // 出口协议
      P1 的非临界段;
    end loop;
  end p1;
  process p2::
    loop
      P (mutex);
      临界段;
      V (mutex);
      P2 的非临界段;
    end loop;
  end;
end.

```

由于 P, V 均为原子动作, 故可以保证互斥且无死锁。和前述自旋锁对比, 它清楚、简单、对称。由于用原语则不用忙等待在等待期间 CPU 可干别的事。它们至少在进程正文上是公平的。

P, V 操作很容易扩大到 n 个进程竞争一个临界段。也可以将二值信号灯劈开分别实施同步警卫功能。以下是生产者/消费者著名问题的同步解。

例 13-5 生产者/消费者的同步与互斥

```

program PRODUCER__CONSUMER
  var buf : TYPE;           // 任意类型 TYPE
  var empty : sem := 1, full:sem := 0; // 两信号变量初始化
  process PRODUCER [i:1.. J]::
    loop
      PRODUCER[i] 产生一条消息 m;
      deposit : P(empty);   // 存入消息 m 的三个操作
      buf := m;
      V(full);
    end loop;
  end;
  process CONSUMER [j:1..N]::
    loop

```

```

        fetch : P(full);           // 从 buf 取出消息 m 的
        m := buf;                 // 三条操作
        V (empty),
        CONSUME R[j]消费取出这条消息 m;
    end loop;
end;
end PRODUCER__CONSUMER.

```

本程序有 M 个生产者，每位每次生产一条消息，只要 buf 空则存入，否则等待。有 N 个消费者，每位每次消费一条消息，只要 buf 中有消息(以 full 标志)则可进入临界段，取出资源 buf 上的信息。信号变量 full , empty 的类型为 sem ，是定义的抽象数据类型。

当程序启动时 M 个生产者和 N 个消费者同时激活。激活的微观次序由实现定。

将信号变量 S 一分为二(empty , full)简化了传递方向。称劈分二值信号灯 ($\text{split binary semaphore}$)。这个算法保证了互斥,无死锁。

将 P , V 操作扩充到多进程，多资源(多个临界段)也是很容易的。例如，我们可实现 q 个缓冲区的多生产、消费者问题。其算法如下：

例 13-6 多进程的互斥与同步解

```

program PROD__CONS
var buf [1...q], mi, mj: TYPE;
var front :=0, rear :=0;           // buf 的下标变量
var empty : sem :=q , full: sem :=0, // 劈分二值信号
var mutexD: sem := 1, mutexF: sem:=1;
process PROD[i:1..M]::
    loop
        PROD[i]产生一条消息 mi
        deposti : P(empty); P(mutexD);
            buf[rear] := mi;
            rear := rear mod q+1;
            V(mutexD); V(full);
        end loop;
    end;
process CONS [j:1..N]::
    loop
        fetch: P(full) ; P(mutexF);
            mj := buf(front);
            front := front mod q+1;
            V (mutexF) V (empty);
        CONS[j]消费消息 mi;
    end loop;
end;
end PROD__CONS.

```

buf 的每个元素放一条消息，它是一个消息队，装入的消息放在缓冲区尾 $\text{buf}[\text{rear}]$ ，消费的消息从缓冲区头 $\text{buf}[\text{front}]$ 中取。两组劈分信号，协议中是两次 P, V 操作。一组劈分信号管进入临界段 (empty , full)，另一组劈分信号管开放的是头还是尾(mutexD , mutexF)。

选择互斥是更为复杂的同步机制。不仅多进程，多资源，而且某进程只能选定使用某些资源。

用 P, V 操作也可以实现选择互斥。经典的例子是哲学家就餐问题。

例 13-6 五位哲学家就餐问题

一张圆桌坐了五位哲学家, 如图 13-2 所示。桌上放有一大盆通芯粉, 但只有五把叉子。而吃通芯粉必须有两把叉子。一旦据有两把叉子的哲学家就可以吃, 否则它只好利用等待的时间思考。如果每人拿一把叉子等待其它人放下叉子, 就产生死锁。

通芯粉当然是共享资源, 但叉子是只有两个哲学家共享的资源。每个哲学家的行为过程即为一进程。第 i 个进程只和第 i 和 $i+1$ 个叉子有关。故算法如下:

```

program DINING__PHILO:
var forks [1..5]: sem:= (5*1);
process PHILOSOPHER [i: 1..4]::
  loop
    P(forks [i]); P(forks[i+1]);
    吃通芯粉;
    V(forks[i]); V (forks [i+1]);
    思考问题;
  end loop;
end;
process PHILOSOPHER [5]::
  loop
    P(forks[1]); P(forks[5]);
    吃通芯粉;
    V(forks[1]); V(forks[5]);
    思考问题;
  end loop;
end;
end DINING-PHILO.

```

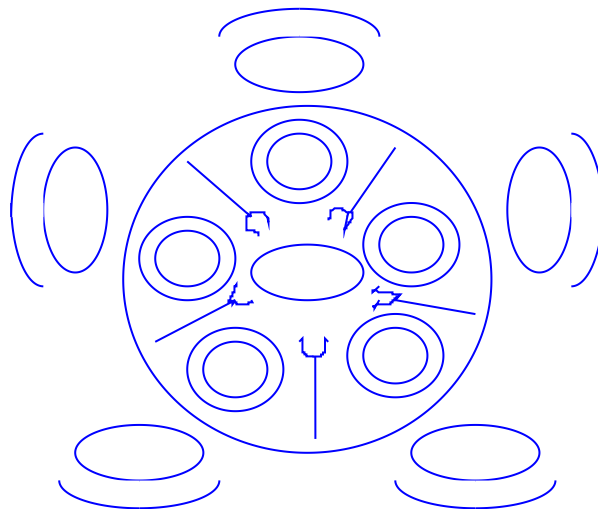


图 13-2 哲学家就餐问题示意图

以上以信号灯实现互斥同步均隐含使用了 await 等待原语。事实上多数分时处理的机器, 单主机多处理器的机器是用系统调用并行核(或叫管理程序)实现的。进程创建就绪后将该进程的描述子(descriptor)入队, 即就绪进程表, 等待 P 操作的完成。这样, 进程处于就绪或阻塞状态且未运

行。此时 P, V 操作的语义解释是:

```
P(s): if s=1 then s:= 0
      else 置进程于 queue 中等候
V(s): if queue != empty then 从 queue 中消除一进程,令调度程序执行它
      else s :=1
```

无忙等待的实现更通用,且可用以实现其它上层机制。核的工作原理图如图 13-1,它的主要职责是为进程分配处理器周期。进程在它分配前是阻塞的。当然核例程名,调用细节因各同步机制和具体机器不尽一致,但原理是一样的。如果为多处理机或分布式系统写一个核,就要复杂一些,就要拿出一个处理器专用于维护就绪表并为其它处理器分配进程。这样,就绪表的访问成了共享的,也要保证互斥。于是在就绪表上用忙等待保证互斥。在分布式处理机上,不仅有一个核管理的就绪表。每个点上都要有自己的核以维持本机的就绪表,进程从一个处理机迁移到另一个上,则置于另一核管理之下,情况就复杂多了。

13.4.2 基于通信的原语

分布式或网络上由于没有共享主存,自旋锁、信号灯实现起来都不太方便(上段最后已说明)。网络提供的通信设施是交换消息(message)。这样的系统只能由发送(send)和接受(receive)消息实现进程交互。基于消息的进程交互一般说来,效率比共享变量的分时或多处理器系统要低。虽然 1973 年 Hanson 首先把它用于操作系统。但直到九十年代这类机制的普及应用才成为可行。

消息是信息传递的单元,按 shannon 的模型,信息源借助信道(channel)向信息目标发送消息。信道成了并发进程共享的资源。信道是通信网的抽象,泛指进程间通信的路径。信道由两个原语访问: send, receive。当某进程向信道发送一消息,通信就开始了。需要该消息的进程,从信道上接受(获取)这条消息。数据流也随发送者传递到接受者。同步也就自然实现了,即不发送没法接受,发送了必须接受(那怕暂行存放不处理),否则丢失。

由于信道本身不能存储,变量只能存放在各个进程中,因而不能共享地访问,所以也用不着互斥机制。由于只有所在进程能考察变量情况,条件同步编程与基于共享变量的大不相同。程序也不一定非要在一个处理器上执行,可以分布在多个处理器上,分布式程序因而得名。反过来,分布式程序却可在单主机或多路处理器的(分时)系统上执行。此时把信道改成共享存储就可以了。

(1) 信道与进程

消息传递利用信道和两个原语可以实现不同的通信模式。信道即信息(数据)传递的路径。是网络的抽象。在分布式网络中信道不同的组合可组成四类不同逻辑功能的进程:

- 过滤器(filter)进程 它是一种数据转移器,进程接受数据后按需要加工,然后传出,如同过滤器滤出需要的数据,故得名。如果将一系列过滤器首尾相连则形成管道(pipeline)或称流水线信道。每个进程独立激活,物理上可分布在网络各结点或多处理器的单主机上。典型过滤器进程均设原语:

```
receive<变量表>from<源进程>
send <表达式_表> to <目的进程>
```

管道信道在操作系统中是最常见的,它把每一个设备抽象为进程,终端—CPU—打印机,即构成运行一个程序的信道,三个进程均为过滤器。当验明消息来源无误,接受原语将表达式表中的值“赋给”变量表。

- 客户/服务器进程 就分布式网络而言,它的拓扑是一个异质结点的图。管道通信只是图

上的一条路径。即数据流通过管道跨越各进程。然而就一般图而言,一个结点可以有多对一,一对一,一对多,多对多的数据流(进程交互)。在实际网络应用中有一类进程即这种情况:客户/服务器,一个客户(client)进程它要求一个或多个服务器(server)进程为其服务,反过来一个服务器也可以为一到多个客户服务。显而易见,一个发出服务要求,一个响应要求后,完成服务,回复应答。通信是双向的。客户/服务器每完成一次通信客户方是发送(请求)接收(应答)两次消息传递,服务器方相反是接受(请求)发送(应答)两次消息传递。这里也有隐含先来先服务原则。网络上的文件服务器就是非常典型的例子。

- 对等(peer)进程 另一类进程是既是客户又是服务器的 peer。一组对某进程共同完成(并行地)一个复杂计算。例如,阵列计算中各结点上的进程都是 peer,它们作相似的部分计算。既发消息,也接受消息,协作完成一计算。非阵列式网络上的 Peer 到 Peer 的进程交互最能代表网络上通信的一般情况。然而,由于算法复杂到目前还没有通用的解,是网络通信追求的最终目标。

(2) 两类通信模式

无论是哪种进程,通信既可按同步也可按异步实现,所谓同步即两进程都要执行到同步点直接交换数据。所谓异步是两进程谁也不等谁按各自速度执行自己进程,就完成了数据通信。事实上完全不等待是不可能的,一个进程执行到 receive 如果没有任何进程 send,它只好等待(跳过 receive 语句再次空循环就是忙等待,否则阻塞在 receive 的等待队中)。与此相反,一个进程执行到 send,它总可以把消息发出去。不是直接发给接受者就是发向信道(哪怕没人接受)。

所以,判定是否异步只看含 send 的进程是否阻塞,等待接受进程的到来。

异步通信的实现,可以看作在某个进程中附带一无限大的邮箱(抽象说也是信道,缓冲时间差)。发送者按自己进程执行,将一条条消息发到邮箱。而接受者在自己合适的时候处理一条条消息各不相扰,犹如邮递员给居民投信,可以彼此不见面。如果是服务器它就将结果消息投入邮箱回复到客户。如果邮箱容量为零,即为同步通信,则非等不可,“见面”交换信息。至于怎样激活、怎样等、见面后怎样分手,各种语言有不同的规定。到下一章高层通信机制一节再介绍。

13.5 小结

- 并行程序是两个或多个程序执行时,时间上有覆盖,并行程序间若共享数据或有数据传递,即它们是相关的,称并发程序。

- 程序代码的一次执行叫一个进程。研究程序的并行性即研究进程交互。一个进程执行中可处于四种状态:就绪、运行、阻塞、终止。控制进程状态的操作是激活(创建进程并进入就绪态)。触发(结束就绪进入运行态)、中断(结束运行进入阻塞态)。程序设计语言以这些操作的组合提供原语。

- 并行程序的模式有四种:单指令单数据流 SISD;单指令多数据流 SIMD;多指令单数据流 MISD;多指令多数据流 MIMD。并发程序主要研究 SIMD, MIMD。

- 原子动作是不可再分成并行子部分的计算机动作。它们粒度大小是相对的,一般以事件(程序的状态有了改变)来定义原子动作。

- 进程有三种类型:独立进程;竞争进程;通信进程。竞争资源和进程通信是并发程序研究的两大类问题。

- 并发程序和顺序程序的差别在于:依赖执行速度;计算结果不确定;产生死锁;得不到执行。

- 并发程序的基本问题是活性(会不会死锁)公平性(会不会死等)、安全性(会不会出错)。它们也是并发程序的基本性质。

- 基于共享变量的进程交互的基本问题是对共享资源的互斥访问和竞争进程间的同步。基于共享变量的进程交互的低级机制有自旋锁和信号灯。

- 进程据有运行资源且处于运行中实现的等待称为忙等待。简单自旋锁的忙等待只解决同步

协调不解决互斥。复杂自旋锁不利于编程。

- 信号灯提供 P, V 操作保留临界段, P 操作隐含延迟以调整同步。是基于共享变量最基本的机制。可在其上构造任何进程交互的通信机制。

- 基于通信的进程可分为四类:

过滤器, 接受消息加工后发送出去。

客户, 发送要求服务的请求, 得到已服务的应答。

服务器, 接受服务请求、服务后给出应答。

等同, 既是客户又是服务器, 也是双向的过滤器。

- 基于通信的原语是:

`receive <变量-表>from<源进程>`

`send <表达式-表> to <目的进程>`

进程原语匹配实现消息传递。

- 异步消息传递, 交互进程间均向信道发/收消息, 彼此可不“见面”。可借助公共邮箱(附在某进程上)实现。

- 同步消息传递, 信道邮箱容量为零, 进程必须“见面”。一般利用原语将快进程延迟到同步点实现。

习题

13.1 试述顺序、并行、并发程序之同异。

顺序性是指一个程序作业从开始到终止的每个作业步骤, 顺序地执行完毕, 并发程序是指多个计算同时平行的执行, 并协调一致。并发的前提是并行, 并行是指程序可相关或不相关串行执行。并行程序是两个或多个程序执行时, 时间上有覆盖, 并行程序间若共享数据或者有数据传递, 即它们是相关的, 称为并发程序。并发程序与顺序程序之间的差别在于: 依赖执行速度; 计算结果确定, 产生死锁; 得不到执行。

13.2 试述并发程序的优点、缺点、难点。

优点: 有利于多机, 多处理器、分布和网络等, 提高效率;

缺点: 依赖执行速度; 计算结果不确定, 产生死锁; 得不到执行。

难点: 安全性, 活性, 公平性。

13.3 客观世界问题是并发的且能用顺序程序模拟实现, 什么情况下用并发程序什么情况下用顺序程序, 尽你可能讲出理由。

13.4 何谓同步进程交互。是否两进程一定要到同步点?

13.5 怎样判定一个程序是异步通信。

13.6 在有分时的机器上编一程序实现自旋锁程序。

13.7 在有分时的机器上编一程序实现信号灯程序。(语言无此功能, 可利用操作系统命令)。

13.8 编一程序, 模拟实现五位哲学家就餐问题, 就餐 1000 人次或打出死锁信息(语言不限, 可利用操作系统命令)。

13.9 分布式网络上进程可分哪四类, 能举出这以外的进程型式吗? 举出三个实际问题说明它们是哪种进程型式解最好。

13.10 假定有一个原语 `start C`, 插入程序后可拉出一新进程, 新进程共享原有变量, `start` 以下创建的变量不共享。将以下矩阵求和的顺序程序改成并发程序, 使计算时间尽可能的少。

```
type Matrix is array (1..n, 1..n) of Float;
```

```
procedure add (a,b : in Macrix;
```

```
sum : out Matrix) is
```

```
begin
```

```
for i in 1..n loop
```

```
for j in 1..n loop
```

```
        sum (i,j) := a(i,j)+b(i,j);  
    end loop;  
end loop;  
end;
```

在你的设计中有多少个进程并发执行？什么环境下本并发程序比原顺序程序快？什么情况下难于判定？

13.11 写出一死锁程序(语言不限)。说明它真是死锁的。

13.12 查阅文献，写出信号灯的不变式。

第14章 进程交互机制和并发程序设计语言

如果说忙等待是人们设计、实现并发程序的最初尝试。信号灯理论则为进程交互的同步与互斥的研究打下了基础。特别是P操作的等待原语最后调度派送实现的做法，把语言和下层执行进程打通，从而任何并发程序均可用信号灯实现。

然而，信号灯是语句级，直接作为程序设计语言层次偏低。程序员设计中稍有失误不是死锁就是死等。例如，信号灯中将P，V操作写颠倒或遗漏就会造成灾难性后果。此外，同步并不一定能保证互斥。都用P，V操作实现，语义不直观，程序难查错。

70年代结构化程序设计已经比较成熟，人们利用结构程序的概念为并发机制提供较高层的机制，并设计了一系列并发程序设计语言。方便了用户、促进了并发程序技术的发展。

在信号灯理论的基础上，基于变量共享的并发机制提出了条件临界区、监控器、路径表达式等指导高层程序结构的技术。基于通信的并发机制提出了借助异步通信原语的异步通信和直接匹配的同步以及卫式同步通信。70年代末把结构化与通信相结合提出了远程过程调用和会合机制。不仅适合于多处理器的并发程序，更适合于分布式网络上的客户/服务器的应用。80年代又有多原语范型的机制出现。因而并发程序语言百花齐放，有的基于某一机制，有的综合各种机制。

本章对以上各种机制作相对完整地原理性介绍，并以四种进程反复作对比的例子。以期读者对以上技术有清晰的概念。最后，列出重要的并发程序设计语言及所采用的机制。

14.1 基于变量共享的高层并发机制

1972年Hoare和Hansen几乎同时提出条件临界区的概念。直到1975年D. Gries和S. Owichi才开发出完整的推理规则。B. Hansen随后把它们用于DP和Edison(1978, 1981)语言。但由于它的低效和其它缺陷很快地被监控器所取代。然而它对并发程序技术的研究有其历史功绩，对多种并发语言的发展有影响。直到90年代的Lynx(1991)语言还采用它的部分技术。

监控器被认为是信号灯以后最成功的成果，特别适合于单机多处理器的共享变量的并发，以及工业控制机中的监控机(器)。为了设计操作系统中的进程调度，必需用到嵌套调用监控器的问题，因而导致了路径表达式 研究。本节也作简略介绍。

14.1.1 条件临界区

竞争资源，人们自然想到把资源独立出来描述。条件临界区(Condition Critieal Region 简称CCR)将共享变量显式地置于叫做资源的区域内。每个共享变量至多只能置于一个资源，且只能在条件临界区的**region**语句中访问。每个进程在自己的进程体内指明要访问的条件临界区，而同一临界区可出现在不同进程之中(谁进谁用)。由于资源r中的共享变量，只能在CCR语

句中根据布尔表达式(条件)的取值访问。因而实施条件同步是显式且容易的,而互斥则自然地实现(谁满足谁进,并使其它进程不能满足条件)。我们还是先看它们的表示法和例子。

(1) CCR的表示法和实例

首先在资源中声明共享变量:

resource r(共享变量声明)

例: **resource** sema(s:int :=n)//信号灯资源中有共享变量s。

共享变量只能在条件临界区中访问。其形式是:

region r **when** B **do** S **end**

其中B是布尔条件, S是对**resource**中声明变量的操作语句集。意即条件不满足时,执行本**region**语句的进程处于等待(条件同步)直至满足执行S并退出**region**。用于信号灯资源,我们有:

region sema **when** s>0 **do** s:= s-1 **end** // P操作

region sema **do** s:= s+1 **end** // V操作

用于完整的并发程序中见下例:

例14-1 用CCR实现例12-4的生产者与消费者

本程序有M个生产者和N个消费者。每次生产一条消息装入临界段的buf中。buf虽然共享但与控制进程无关。用CCR实现对它的互斥访问。而用于控制的共享变量纳入CCR。程序中的deposit和fetch是作为语义说明而加的。注意**resource**和**region**的呼应。

```

program PRODUCER_CONSUMER_CCR
  var buf:TYPE;
  var empty :sem, full:sem;
  resource sema:(empty := 1, full := 0);
  process PRODUCER [i:1..M]::
    loop
      PRODUCER[i] 产生一条消息m;
      deposit:
      region sema when empty>0 do empty := empty - 1 end;
      buf := m;
      region sema do full := full + 1 end;
    end loop;
  end;
  process CONSUMER [j:1..N]::
    loop
      fetch:
      region sema when full>0 do full:= full - 1 end;
      m := buf;
      region sema do empty := empty + 1 end;
      CONSUMER[j] 消费者取出的这条消息m;
    end loop;
  end;
end PRODUCER_CONSUMER_CCR.

```

条件临界区的**region**语句中不仅有资源**resource**中声明的共享变量。也允许有所在进程中的变量,如buf。这样,我们可利用嵌套的**region**语句,把不同资源的共享变量复合起来(外层**region**的变量相当于所在进程变量)可以更清晰地控制同步。但请注意,本例清晰但不能保证互斥访问buf(正常情况是没有问题的,如例中误写full:=1就会产生)。为此,对例14-1的CCR

稍作改进。例14-2将buf放进临界区**resource**中，它不仅互斥，还可以并发地访问一个循环队(缓冲区)。

例14-2 CCR实现对共享缓冲区的并发访问

```

program PROD_CONS_CCR
  var buf[1..q]:TYPE;
  var front, rear :int;
  var slot, mi, nj:TYPE;
  resource buf[1..q] (slot;full :bool := false);
  resource f(front := 1);
  resource r(rear := 1);
  process PROD[i:1..M]::
    loop
      PROD[i] 生产一条消息mi
      deposit:region r do
        region buf [rear] when not full do
          slot := mi; full := true;
        end;
        rear := rear mod q+1;
      end;
    end loop;
  end;
  process CONS [j:1..N]::
    loop
      fetch:region f do
        region buf[front] when full do
          nj := slot; full := false;
        end;
        front := front mod q+1;
        CONS[j] 消费一条消息nj(一般情况mi≠nj)
      end;
    end loop;
  end;
end PROD_CONS_CCR.

```

本程序共声明q+2个资源，每一缓冲区元素均为一资源，故其中消息放于slot中。当M+N个进程激活后，它们各自进入自己的临界区，谁满足条件被存入(或取出)消息。

(2) 条件临界区评价

条件临界区最主要的优点是概念清晰。此外：

无需辅助标志和变量即可描述共享变量的任何进程交互。

- 程序编译时即可保证互斥。
- 一个进程创建一个条件不需顾及其它条件是否与此条件有关。
- 易于程序正确性证明。
- 体现了共享数据传递的方便。

它的致命缺点是低效(和信号灯相比)。此外：

- 进程和共享变量耦合太紧。
- 临界区利写不利读，一多了就太散，因而也难修改。

由于它的缺点严重，在Hoare 和Hansen相继研究它们不久，Dijkstra就提出了改进建议。

既然是共享数据为什么不把加工它们的操作也集中起来，每个进程调用这些操作，不就真正达到监控目的了吗。这个非常小的改进反映了基于模块控制的思想. 对70年代 程序模块化有深远的影响。

14.1.2 监控器

Dijkstra提出的建议是把分散在整个程序中的**region**语句进一步集中成为一个模块叫做监控器 (monitor)。监控器在Hansen研制的并发Pascal中叫管程，即与进程对应，管理所有共享资源控制各进程同步与通信的模块。各进程和共享数据的耦合因而松弛，变成向监控器发出调用才能使用共享数据的关系。对共享变量的加工限于监控器内定义的过程。这也就是最早的对象封装思想，事实上它也得益于Simula的类封装的启发。显然有利于全局控制并发程序(一目了然)，便于维护、扩充、修改。由于进程和监控器相对独立，易于开发和调试。监控器成为实现并发程序设计语言的主要技术。

14.1.2.1 表示法和进程同步

监控器程序结构的一般形式是：

```
program monitor
monitor Mname::
    共享数据声明并初始化;
    proc op1 (<形参表1>) is <op1 体> end;
    ...
    proc opn (<形参表n>) is <opn体> end;
end;
process Pname [i:1..N]::
    局部数据声明并初始化
begin
    :
    call Mname.opi (<实参表>);
    :
end
begin
    初始化, 激活进程
end monitor.
```

monitor模块是所有进程共享的。它本身只是一被动实体，仅当外部进程通过调用操作进入到监控器，才能加工共享数据。如果每次只允许一个进程进入作一种操作就能自动保持对共享变量的互斥使用。仅当已进入进程处于阻塞态或已返回，第二个进程才能进入，(且必须进入，因为阻塞是等待条件改变，若没有新进程来改变条件，只能死等)。每个进程都有描述它当前状态的描述子，因此在每个进程中要有显式控制进程同步的wait(等待)和signal(示信，向其它进程发出信号，本进程对共享变量的使用已经完毕)的原语。这样就控制了每个进入监控器内操作过程的进程。

以监控器实现同步有许多方案，我们先介绍Hoare的条件变量方案(1974)。

条件变量用以延迟监控器中过程的执行，也就是延缓了使用它的进程。在条件变量上定义了两种操作：

```
var c: cond    //cond是抽象类型可定义为布尔值
wait (c)       //延迟执行本句的进程直至c为真
```

signal(c) //唤醒处于c的等待队列中的头一个进程。如队列中为空则如同skip
例14-3 有界缓冲区的监控器实现算法

```
monitor BOUNDED_BUFFER::
  var buf[1..q]: TYPE;
  var front :=1, rear :=1, count := 0;
  var not_full cond;           //当count < q示信为真
  var not_empty cond;         //当count > 0 示信为真
  proc deposit (data :TYPE) is
    while count = q do wait (not_full) end;
    buf [rear] := data;
    rear := (rear mod q) +1;
    count := count+1;
    signal (not_empty);
  end;
  proc fetch (VAR result :TYPE) is
    while count=0 do wait (not_empty) end;
    result := buf [front];
    front := (front mod q) +1;
    count := count -1;
    signal (not_full);
  end;
end BOUNDED_BUFFER.
```

它控制最多q个进程可进入缓冲区buf。当缓冲区已满则进程处于**wait(not_full)**等待至not_full为真，而not_full为真必须feteh被使用。同理，如进入fetch的进程发现buf中已无数据可取，也要在**wait(not_empty)**处等待。这就保证了buf的互斥使用。每个条件变量均对应一个等待队列。例中WHILE是为了保证它能响应多个进程，即一个挂起，它循环响应第二个的调用…

此例进程声明和其他程序略。

一个并发程序可以有多个监控器，分管不同共享资源。**wait** 和 **signal**操作相当于信号灯中的P.V操作，但略有不同，差别是[1]当没有进程在条件变量上等待时，**signal**操作不起作用。[2]**wait** 使进程等待到最近一个**signal**操作执行的一段时间。[3] **signal**先变条件变量再唤醒进程。

为了完善monitor，以后又增加了一些辅助操作，如有优先级的**wait**、广播式**signal**、empty查询和最优先级的minrank。以下c为指示等待队列的条件变量，rank是优先级的整值。

- **empty(c)** 与c对应队列中无进程等待为真，否则为假。
 - **wait (c, rank)** 下次以c指示队列中具有rank值的进程先唤醒。
 - **signal_all(c)** 所有等待在c中的进程全部唤醒。
 - **minrank(c)** 返回所有等待c指示队列中最优先进程的rank值。
- 以下是操作系统中中期调度时，以耗时最短的进程作下一进程的算法。

例14-4 耗时最短的下一作业

```
monitor SHORTEST_JOB_NEXT::
  var free := true;
  var turn :cond; //该条件当资源可用示真
  proc request (time :int ) is
    if free then free := false
```

```

        else wait (turn, time) endif;
    ...
end;
proc release ( ) is
    if empty (turn) then free := true
        else signal (turn) endif;
    ...
end;
end SHORTEST_JOB NEXT.

```

每个进程注明耗用时间来调用request, 请求获取资源。如不空则等待条件变量turn变值。当turn变为真以后则time 值最小的进程最先唤醒。rank按递升值优先级递减。

14.1.2.2 以监控器实现条件同步的技术

除了设条件变量、有优先级参数的条件变量而外, 还可采用以下技术:

(1) 复盖条件变量

一般说来, 有优先级参数的条件变量效率是高的。为了记住众多的优先参数, 进程等待队中要有长长的记录表, 在某些情况下, 空间耗费不允许。作为一种时、空互易, 可设复盖条件变量(即两级条件变量)。“上一级”条件变量, 一旦满足, 则唤醒所有(利用signal_all)在队列中等待的进程, 然后各进程再按“自己的”条件变量检查。该等的继续等, 该醒的则被唤醒。这种以现算代替查表的作法在操作系统的定时时钟中采用过。

(2) 传递条件

我们先讨论一个占先(preemptive)概念。当某进程要求使用某资源, 资源控制器则为其分配资源, 在使用期间原则上是不再分配给其它进程。如果在此期间有一更高优先级的进程要使用该资源, 只要前一进程当时未处于运行态都要转而分配给后一高优先级进程, 谓之占先。

有无占先对竞争的并发进程影响是很大的, 如果用监控器实现信号灯, 我们有:

例14-5 以监控器作信号灯

```

monitor SEMAPHORE::
    var s:= 0, pos:cond; //当s>0, pos示信为真
    proc P ( ) is
        while s=0 do wait (pos) end;
        s:= s-1;
    end;
    proc V ( ) is
        s := s+1; signal (pos);
    end;
end SEMAPHORE.

```

当有多个进程调用P ()而处于pos的队列中等待时, 一般情况下每进行一次V ()操作即可唤醒一个进程执行完P ()。其唤醒次序保证是先来先服务的。由于signal不是占先的, 在被唤醒进程执行之前, 监控器不能拒绝另一进程进入它。此时, 调用P ()的外来进程发现s>0就先于被唤醒进程执行完了P ()。这种信号被“盗窃”的情况使程序员难于控制程序的正确。

为了防止条件信号被偷, 发信号的进程不利用全局数据(s)而是直接将条件传入被唤醒的进程。改进写法是, 先变V ()的过程体。当调用V ()时, 它只唤醒一个进程而不忙于为s增量。若无等待进程则按常规增量。再改P ()的体, 设检查条件s>0。仅当s>0才去对s减量后返回, 否则于pos处等待(如下例)。当pos的队列上等待的进程被唤醒, 它就从P ()返回不对s减量以补救V ()操作中的signal执行之前的S不增量。逻辑上pos条件是直接传递的。代码如下:

例14-6 以监控器实现的FIFO信号灯

```

monitor SEMAPHORE::
    var s:=0;
    var pos :cond;    //当V中pos队列非空示真
    proc P( ) is
        if s>0 then s:= s-1 endif
        if s=0 then wait(pos) endif;
    end;
    proc V( ) is
        if empty(pos) then s:= s+1 endif
        if not empty (pos) then signal(pos) endif;
    end;
end SEMAPHORE.

```

注：本例中“ ”号表示和前一个语句并行执行的语句，以下同。

(3) 会合同步

有些进程，如打印机，磁盘驱动程序，总是为其它进程服务的，它本身得不到服务。而其服务对象是客户进程，进程交互是客户/服务器(client /server)关系时，为此两交互进程必须会合(rendezvou)才能得到服务。如不能到达会合的同步点则要相互等待。以下是经典的贪睡的理发师问题：

例14-7 贪睡的理发师的模拟解

一个小理发店有一位贪睡的理发师，他睡在理发椅上等待顾客，顾客随机地从一个门进入。如果没有人他就得到理发师理发，理完后理发师打开后门让理完了的顾客出去。接着第二位。如果顾客进入，理发师正忙着，他就座在椅子上等待。如果没有顾客，理发师又在理发椅上睡觉。示意图如图14-1。

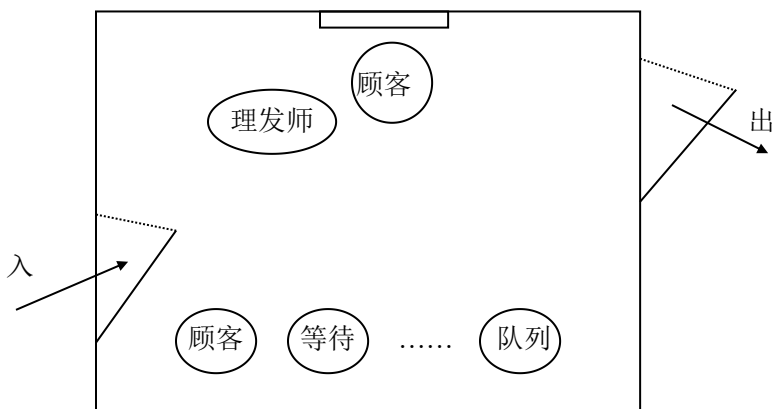


图14-1 贪睡的理发师的理发室

如果每一顾客是一个进程，理发师也是一进程。用一个监控器正好模拟理发店内进程交互。这里有两会合，一次是等待顾客坐上理发椅，一次是理发师开着后门等顾客出去。顾客进程在理发店(监控器)中要得到理发get_haircut()，理发师进程则要得到下一位顾客get_next_customer()并给他理完finished_cut()，再开门送客。

控制同步的条件是理发师空不空(barber_available)；理发椅坐上有没有人(chair_occupied)；后门开了没有(door_open)；顾客走了没有(customer_left)。并以barber, chair, open三个变量指示理发师有空、理发椅可坐、后门的开闭。算法如下：

```

monitor BARBER_SHOP::
    var barber := 0, chair := 0; open := 0;
    var barber_available :cond    //当barber>0 示真

```

```

var chair_occupied :cond           //当chair>0示真
var choor_open:cond               //当open=0示真
proc get_haircut( ) is             //顾客调用
    while barbe = 0 do wait (barber_available) end;
    barber := barber - 1;
    chair := chair + 1; signal (chair_occupied);
    while open = 0 do wait (door_open) end;
    open := open-1; signal (customer_left);
end;
proc get_next_customer( ) is       //理发师调用
    barber := barber + 1; signal (barber_available);
    while chair = 0 do wait (chair_occupied) end;
    chair := chair - 1;
end;
proc finished_cut ( ) is //理发师调用
    open := open + 1; signal (door_open);
    while open>0 do wait (customer_left) end;
end;
end BARBER_SHOP.

```

其中顾客两次等待(理发师有空、门打开)，理发师两次等待(顾客入座、顾客离去)。各有两次示信(可以入座、顾客已去)，(理发师已空、门已开)。

14.1.2.3 各种语言实现监控器时的原语语义差异

以上定义监控器有三个特征：第一，监控器封装了共享变量，共享变量仅能由监控器内的过程访问。第二，监控器内的过程都是互斥地执行的。因而共享变量不能并发访问。第三，条件同步由wait和signal操作实现。wait延迟正在执行的进程，此时监控器不再排斥其它进程进入，即临时放弃互斥。执行signal则唤醒一个被延迟的进程，但发信号的进程仍排斥其它进程进入监控器，被唤醒的进程仅当它在能获得互斥执行的某个未来时间开始恢复执行。

程序设计语言Mesa包括以上三个特征。UNIX采用上述条件同步。然而，监控器有时不一定必须互斥。也可以采用其它办法实现条件同步。本节讨论这两方面，先看后者。

(1) 实现条件同步的各种信号机制

以下各种机制的前提是监控器内过程都是互斥执行的。

● 自动信号 A S：

wait 和 signal虽然是一对互补操作，但当signal用错了也不影响wait的执行。一个也等待十个也可以等待。于是Hoare想到只要wait 加上条件就可以不用signal原语了。这样可省去检查signal是否执行的开销，程序员也不必操心是否用错。在wait(B)中条件B不满足则等待，满足了，自动(发出信号)恢复执行，因此，称这种同步机制为自动信号AS。其缺点是B中变量一般不是共享变量，难于编制更复杂的调度问题。

● 信号和继续 S C

第二种机制是我们已讲过的显式信号机制，当无占先时发信号的进程继续执行。直至它进入等待或返回之前其它进程是不许进入监控器的。因此称这类机制为信号和继续(SC)机制。modula3即采用此种机制。

显式wait、signal操作，当采用占先的signal时，并发语言曾经采用过以下三种语义稍有差别的机制。

● 信号和出口 S X

第三种机制是既然被占了先，发信号的进程也就不等了。立即从监控器出口或从过程返

回，故称信号和出口(SX)机制。并发Pascal即采用此种机制。

- 信号和等待 S W

第四种是发信号的进程被人占先之后处于监控器内等待，直到它能再次获得互斥访问，恢复执行。故称信号和等待(SW)机制。Modula和并发Euclid采用这种机制。

- 信号和急等 S U

最后一种是第四种的修正，称信号和急等(SU)机制。发信号进程被人占先之后也要等待，但保证在监控器有新的进程进入之前先使它得到恢复。Pascal_plus即采用这样的机制。

以上五种信号机制语义略有不同，但可从理论上证明它们是等价的。即以一种机制可模拟另一种机制。实现费用不同，对某些类型问题表达的方便性不同。也正是不同语言各自钟爱它们的原因。

(2) 嵌套监控器中的互斥

在磁盘调度器之类的应用中，一个进程首先要争取进入磁盘去寻址，找到地址后读/写，这样就要设计两个监控器。一个管理粗的磁盘资源，进程进入或释放。另一个管理读/写区，进程互斥地读写。这两个监控器是嵌套的。

如果像单监控器那样保持互斥访问；每一时刻只有一个进程进入监控器，调用某个过程，我们称它是闭式调用。显然，闭式调用共享变量是得到互斥保护的，但是在嵌套监控器之中，这种方式容易引起死锁。因为当它在下层监控器内等待时，由于它上层是互斥的，就没有别的进程能进入到下层去唤醒等待的进程。除非唤醒条件有某种技巧。所以人们自然想到开式调用。所谓开式调用是若有嵌套调用发生时上层互斥自动解开，待调用返回后上层监控器又重新闭合(获得)互斥。显然，开式调用的语义要复杂一些，它要求在每次嵌套调用时监控器不变式为真。且共享变量不得作为下层调用的引用参数(共享变量成了上下层并发访问的数据了)。

一个程序设计语言，可以不允许嵌套监控器(因为一般问题用多个同级监控器均可实现)。也可以选择某种调用机制。然而，既然允许嵌套，最好提供两种调用机制以备用户选择。

14.1.3 路径表达式

监控器内的过程(操作)是互斥执行的，操作的同步靠程序员显式地用wait和signal控制。不同的信号机制有不同的缺省约定。究其本意是对共享数据规定一谁先谁后的次序(用条件是否满足控制)。来早了则以延迟控制同步。于是1974年Campbell和Habermann提出以路径表达式直接控制进程顺序的建议。当然前提是封装共享变量的模块，其本质是对模块中的操作显式规定执行次序(给出约束)。操作的对象是资源(以数据表征其状态)，所谓路径表达式是就每一资源在其开始声明时就在其上定义操作的约束。而每个操作过程中不再使用同步原语wait、signal。

路径表达式的语法是：

```
path <路径表> end
```

其中路径表是操作名和路径算子组成的表。路径算子指明它的前后操作是顺序的(;)还是并行的(,)，并以[...]和'N'：指明同时激活的操作或路径的数量。以下以有界缓冲区的两个操作作说明：

```
path deposit, fetch end
```

只说明deposit 和fetch是并发执行的。对先后次序，被激活操作的个数和路径数没有任何约束，即无同步要求，它等价于：

```
path [deposit], [fetch] end 和
```

```
path [deposit, fetch] end
```

如果指明顺序要求：

```
path deposit; fetch end
```

则说明deposit必须先于fetch执行。可以同时激活多条这样的路径，只要激活和完成的fetch操作不超过deposit的个数且遵守次序(可以空缺)。

```
path1: (deposit; fetch) end
```

说明只能有一条路径(但可多次执行此路径)，两操作交替互斥执行。

```
path N: (1:(deposit); 1:(fetch)) end
```

说明deposit和fetch是一一对应地互斥激活，先执行deposit，完成的deposit个数不超过N次且可多于fetch完成的个数。由路径表达式指明的同步约束，编译时即可保证。

路径表达式优点是程序员可直接控制过程的执行，正文清晰，它是监控器中派生出来的一个重要分枝。80年代被认为是很有希望的同步表达模型。事后表明，当同步化依赖过程参数或监控器的状态时，表达能力差，因而不适于一般的调度程序，只对高层并发描述带来方便。此外，过程交错引用表达比较难，因而难于作为并发程序语言的核心模型。

14.2 基于消息传递的高层并发机制

基于消息传递的并发机制，中心问题有两个：信道命名方式和同步化。它们是正交的，因而出现了各种各样的组合。但从并发程序的角度，互斥访问、无死锁、公平性仍然是高层机制要考虑的。这里的互斥访问指两进程不能同时和一个进程通信。我们按两类通信模式讨论之。

14.2.1 异步消息传递

异步消息传递要借助于信道，信道可以看成是已发送但未接受的消息队，一个并发程序可以声明多个信道(本书以下述抽象表示法)：

```
chan<信道名> (<标识符1>:<类型1>, ..., <标识符n>:<类型n>)
```

其中<标识符i>:<类型i>为传到信道上的数据域名(可缺省)和类型。例如：

```
chan input (char);
chan disk_access(cylinder, block, count : int,
                 buffer: char*);
char output [1...N] ([1...M]: char);
```

第一行声明一个信道，名为input且这条消息中只有一字符域。第二行一个信道，一条消息中三个整数和一个指向字符的指针)。第三行声明了N个信道每个信道都是M个字符的行。

有了声明，在进程中就可以写send和receive原语向/从某个指明的信道发送/接受消息，其中发送的表达式的个数、类型、次序，接受的变量的个数、类型、次序应和声明的信道中的域一致。

信道如声明为全局的，即并发程序所有进程均能向它发消息，则称邮箱。如果信道只在接受进程中声明，则称该进程的输入端口(port)。当然，同一并发程序的发送者均可见到这个端口。我们假定发送/接受一条消息是原子的。且每个信道都是先进先出的队。队是无限长的。

如果一个信道只有一个发送进程和一个接受进程，则称它为一链(link)。

异步通信进程执行send原语是从不等待的。而receive原语则不然，消息队中无消息，或进程调度从消息队中选择耗时最短的下一条消息时，进程只好处于receive处等待。以下举例说明各种进程的异步通信。

(1) 异步通信的过滤器

以下例14-8是从一个信道input接受字符流的过滤器进程char_to_Line。每当它接受满一行或见到CR符号就向输出信道output送出该行。CR为行结束符。

例14-8 过滤器异步实现的算法

```

chan input (char), output ([1..MAXLINE]:char);
char_to_Line::
  var line [1..MAXLINE]:char, i:int :=1;
  loop
    receive input (line [i]);
    while line [i] =CR and i<MAXLINE do
      i := i+1;
      receive input (line [i]);
    end
    send output (line); i:= 1;
  end loop;
  ...

```

//以下可写其它送字符或消费行的进程。形成流水线，是过滤器典型用法

本例中进程char_to_Line的私有数据line，它的元素出现在**receive**原语中是变量。当装满值出现在**send**原语中是表达式。信道input，output的物理实现可以附在进程(本例即char_to_Line进程)内。各为单字符和MAXLINE个字符的无限长的队(一般由动态链表实现)。

若有N个生产者进程向信道送字符，依其先后输入，没有互斥和同步要求。只有在极端情况下严格同时到达，则由系统决定进入信道次序。输出信道也如是。M个消费进程，每位每次消费(接受)一行，先来先取走。也没有互斥和同步要求。

信道只管收发，虽然也是一数据空间，由于没有临界区的访问、修改等概念，故不是共享变量。

(2) 客户/服务器异步通信实现

客户/服务器结构中服务器往往是处理主体，而客户只是一触发器。例如，终端是客户，打印机、数据库均可看成服务器，其执行设备档次比较高，以支持一对多的服务。以下是用客户/服务器进程实现的资源分配器。

例14-9 资源分配器的客户/服务器算法

我们把可用资源抽象为整数代号放在集合units中，每个资源名是unitid变量中的整数值。由插入(insert)和撤除(remove)操作操作这个集合。客户/服务器，每次只能申请/处理一个单位的资源。

设一信道为request，客户经此信道发送消息指出自己的号数(index)。对资源要求的性质或是占据(ACQUIRE)或是放弃(RELEASE)。如果是放弃还要指明放弃的资源号(unitid)。另一信道是reply(应答)，服务器应答第m号客户可以给你分配第n号资源。算法如下：

```

type op_kind = enum (ACQUIRE, RELEASE);
chan request (index, op_kind, unitid; int);
chan reply [1...N] (int);
Allocator ::
  var avail:int := MAXUNITS, units :set of int;
  var pending :queue of int;
  var index:int, kind :op_kind, unitid :int;

```

```

//units可用资源以整数代号初始化, reply初始化为零
loop
  receive request(index, kind, unitid);
  case kind = ACQUIRE =>
    if avail > 0 then                                //正常分配
      avail := avail-1;
      unitid := remove (units);
      send reply [index] (unitid);
    endif;
    if avail = 0 then                                  //记住index号客户未分
      insert (pending, index)                          //插入阻塞队pending
    endif;
  case kind = RELEASE =>
    if empty (pending) then
      avail := avail+1;
      insert (units, unitid);
    endif;
    if not empty(pending) then                          //分配unitid号资源
      index := remove (pending);
      send reply [index] (unitid);
    endif;
  end case;
end loop;
client [i:1..N]:
  var unitid :int;
  send request (i, ACQUIRE, 0);
  receive reply [i] (unitid);
  //使用unitid号资源然后释放它, 发以下消息:
  send release (i, RELEASE, unitid);

```

由于是异步的, 当分配器从信道request获取请求(ACQUIRE)分配资源的消息时, 如果可用资源没有, 即avail=0, 它就把请求接受下来, 将客户代号index放入阻塞队(pending)中, 跳出case执行空循环。一旦有释放请求, 它查到pending不空, 则从pending中消去队尾等待的那个客户, 并把这个刚释放的单位(unitid号)给了这个客户, 并发送应答。

客户/服务器进程的特点是**send, receive**是对等的, 服务器要应答消息。本算法有N个客户一个服务器。用其它主控程序创建client[i], 当它发送request后, 接着执行reply[i], 如发现未分配(未应答)则在本处等待, 直至确已分配unitid/=0再使用它。这些动作是**receive**原语实现的。

下面一个例子更有意思一点, 是操作系统中常用的文件服务器。它保持客户与服务器交谈的连续性。为了访问外存上的文件, 客户首先要打开文件, 文件存在表示成功, 客户才能发出一系列读/写要求。最后由客户关闭文件。打开之后作读、写、关闭等操作, 但打开的和操作的是同一文件(即连续交谈)。为了简化, 文件都是一种类型, 差别只在空不空, 所以文件服务器也是N个进程, 一个管一个。每一个对应一个访问信道。但打开文件的信道是全局的。为了简化, 客户号、服务器号均以整数表示。

例14-10 文件服务器和客户

```

type kind = enum (READ, WRITE, CLOSE);

```

```

chan open(fname:string *, clientid:int);
chan access[1..N] ( kind, TYPE);           //可增细节信息
chan open_reply[1..N] (int);
chan access_reply [1..N] (RTYPE, FLAG);
File_Server [i:1..N]::
    var fname:string*, clientid:int;
    var k:kind, args:TYPE;
    var more:bool := false;
    var 局部缓冲区, 快速缓存, 磁盘地址等;
    loop
        receive open (fname, clientid);
        //打开文件fname, 不成则等, 若成:
        send open_reply[clientid](i);
        more := true;
        while more do
            receive access [i] (k, args);
            if k=READ then 执行读操作;
            if k=WRITE then 执行写操作;
            if k=CLOSE then 关闭文件;
            more := false;
            endif;
            send access_reply [clientid] (results);
        end;
    end loop
client [j:1..M]::
    send open ("foo", j);                  //第j个客户要打开foo文件
    receive open_reply[j] (serverid)      //取server的号
    //准备访问数据k.args
    send access [serverid] (k, args)
    receive access_reply[j] (results)     //取结果信息

```

本例是一对一地访问，只是把多个进程写在一起。一旦j和某个i对应，它总是那个文件。如果文件服务器不止N个。最好的办法是按需要动态创建服务器进程。当然，信道也要随之创建。

14.2.2 同步消息传递

异步通信由于**send**操作没有等待，则信道中必须有一个无界的消息队。这会产生几个问题。首先，发送进程的消息可能堆积很多，而发送者如果想接受者确实得到才敢去作以下步骤的话，就要等很长时间才能得到应答。第二，如果等了很长时间，难以知道是接受进程有故障还是没有处理完。第三，需要有存储空间作消息缓冲，大小还要试着来。造成测试麻烦。总之，消息一多**send**本来不等的也变成等了。违反异步初衷。同步消息传递可避免以上三种情况。

同步消息传送**send** 和**receive**两原语都是可阻塞等待的原语。仅当两进程都达到同步点时才通信，因而不需缓冲，交付就交付了不会出问题。本质上相当于分布式赋值语句，一个进程

求出表达式值，通过信道赋给接受进程的变量，中间无缓冲。

14.2.2.1 通信着的顺序进程CSP

1972年Hoare提出的通信着的顺序进程(CSP)模型首先完整引入同步通信概念，CSP本身不是完整的并发程序设计语言，但对同步通信有较大影响。我们先介绍它。

(1) 通信语句

CSP不用**send**, **receive**原语，采用紧耦合的进程通信，它直接用语句指明使用某个进程的变量。假定A进程要和B进程通信并把表达式e的值传给它。则在通信进程间对等地写：

```
A: ... B!e...    //B!e 是输出语句，目标是B
B: ... A?x...    //A?x 是输入语句，源自A进程
```

当表达式e和x的类型一致时则称它们是匹配的。A, B并行执行，谁先到有“!”或“?”的语句谁就等待，直至对方到达。完成数据传递(分布式赋值)后各自执行自己进程。输入/输出语句更为一般的形式是：

```
<目标进程>! 信道名 (<输出表达式表>)
<源进程>?   信道名 (<输入参数表>)
```

其中信道名应一致。在同步通信中信道不再是邮箱而是有向链。信道用来区分不同类的消息，但程序中可以不显式声明信道名(CSP的后裔Occam却要显式声明)。它们的语义解释如：

```
B! port(e1, e2, ... en)
A? port(x1, x2, ... xn)
```

向目标进程B以名为port的信道传送表达式 e_i ($i=1..n$)的各个值；从来源进程A的信道port中获取值，完成 $x_i := e_i$ 赋值。

当只有一个信道时，信道名可省。

例14-10 求两整数的最大公约数

我们把求最大公约数做成一服务器进程GCD。客户进程client每向它发送一对整数，它求出结果发回客户。

```
GCD :: var x, y: int;
loop
  client ? args( x, y)
  while x/=y do
    if x>y then x:= x-y endif
    if y>x then y:=y-x endif;
  end;
  client ! result (x);
end loop;
```

client进程在准备好数据v1, v2和结果存储空间r之后，应有以下语句：

```
... GCD! args(v1, v2);
GCD ? result (r)...
```

其中args, result是信道端口名。

(2) 选择通信

以上通信进程双方要指明对方写传递消息的输入/出语句，它的最大的限制是不能一到多地通信。为此，引入Dijkstra 1975年提出的(警)卫式命令。即设一布尔条件B，仅当B为真才

执行消息传递语句。所以，也叫卫式通信。卫式通信语句的一般形式是：

$B; C \rightarrow S;$

其中B是布尔条件，C是可选的通信语句，S是一组消息传递的语句。B; C组成卫式通信条件G。当B不出现隐含B为真。当C不出现卫式通信语句就变成了般的卫式语句。选择通信语句一般形式是：

```
if G1→S1
  G2→S2
  ...
  Gn→Sn
endif
```

只要条件Gi为真，各子句执行顺序不限。以上GCD服务器可用选择通信为多个客户服务：

```
GCD::var x, y: int;
while (i:1..N) client [i]? args(x, y) do
  while x /= y do
    if x>y then x:= x-y endif
    if x<y then y:= y-x end if;
  end;
  client ! result(x);
end;
```

其中i下限定为1..N即条件B，每当第i个客户得到一个数对x, y时才求它们的最大公约数。而i值的顺序是不限的。

CSP的选择通信只许可输入语句作为警卫条件，输出不可以。这样，实现容易，表达带来不方便，有些语言有扩充。CSP成功之处在于它基于非常简单的输入/输出语句思想，易于与其它机制集成，表达能力强。

14.2.2.2 过滤器的同步实现

阵列计算时，各结点形成一矩阵，其中每个结点都是一个过滤器。将输入的数据加工后送出去，当然要通过进出的信道。而每个结点的计算都是并行的。它们共同完成一阵列计算。以下我们用Eratosthens筛法求素数作为过滤器管道(矩阵的简化)通信的例子。

例14-11 利用管道通信求素数

按照Eratosthenes筛法求素数，例如N以内，先将候选数(除2以外的所有奇数)放于一维数组：

```
2 3 5 7 9 11 13 15 17 19 21 23 25 27 29 .....N
  ↑
  k
```

按顺序程序做法是设一指针k，查k所指元素直至N，凡有k指素数的倍数则筛除。筛完k:=k+1。显然，当前k指的一定是素数，再以当前k所指素数将候选数再筛一遍，直至k所指向的素数近于N。当用并行程序实现时，每筛一次都可以看作是一过滤器进程sieve[i]，于是就有L个进程(L为N以内素数个数)。每个进程的任务是向自己“私有”一维数据结构中填筛过的数。上述数表可以看作是sieve[2]进程做完的结果，经过sieve[3]。sieve[4]完成：

```
2 3 5 7 11 13 17 19 23 25 29 31 25 37 ... N //sieve [3]完成
  ↑
  无3的倍数
2 3 5 7 11 13 17 19 23 29 31 37 ... N //sieve [4]完成
  ↑
  无5的倍数
```

即每个进程i把第i-1进程筛过的数接过来，看它是不是本进程的“第一个”素数P的倍数。如果不是则放弃。如果是，则传到i+1进程，让它再筛。直至“第一个”素数值 $\geq \sqrt{N}$ 。

写并发程序时，“私有”的一维数组是不必要的。因为发送来的数是筛过了的，i-1进程

不发i进程就不必筛，筛后立即发出。所以只要两个变量p，next就足够了。算法如下：

```
sieve [1] ::var p:= 2, i:int;
  print (p); sieve [2]! p;
  for i := 3 to N by 2 do          //发送候选奇数
    sieve [2] !i
  end.
sieve [j: 2..L]::var p:int, next:int;
  sieve [j-1]? p; print (p) ;      //只打印“第一个”收到的
loop
  sieve[j-1] ? next;
  if next mod p /=0 then
    sieve [i+1] ! next endif;
end loop;
```

每个进程到了通信语句，如对方未到达就等待，其信道为sieve [i:1..N]。连接用输出/λ语句体现。

14.2.2.3 客户/服务器的同步通信实现

为了对比，我们依然用文件服务器作例子，它的异步实现见例14.10。

例14-13 文件服务器的同步实现算法

```
File_Server [i:1..N]::
  var fname:string, args:TYPE;
  var more :bool;
  var 局部缓冲区, 快速缓存, 磁盘地址等;
  while (c:1..M) client [c]? open (fname) do
    //如名为fname的文件已打开:
    client ! open_reply( );
    more := true;
    while more = true do
      if client [c]? read(args) then
        调处理读入数据的过程;
        client [c]! read_reply(results);
        if client [c]? write(args) then
          调处理写的过程;
          client [c] ! write_reply(results);
          if client [c]? close() then
            调关闭文件操作;
            more := false
          endif;
        end;
      end.
    client [j:1..M] ::
      var serverid : int;
      while (i:1..N) File_Server[i]! open(“foo”) do
        serverid := i;
        File_server[i] ? open_reply(serverid);
      end.
    //以下写使用该进程的代码。例如，读可写：
```

```

File_Server[serverid] ! read(访问参数);
File_Server[serverid] ? read_reply(results);
...
//最后要有关闭文件的消息

```

14.2.2.4 同步通信的底层实现

异步通信机制依靠**send**, **receive**原语和显式的信道, 在具体机器上实现这些原语和信道如前所述, 还要有一下层实现程序。同样, 同步通信, 如CSP, 直接指明对方发送消息, 也要有底层实现的程序或例程。这些程序, 是由并发语言系统提供的。我们称它为并发核(kernal)。现在先说异步通信的核。

分布式网络上, 各结点机上均有一并发核。并发核一般由三部分组成:

- 描述子和缓冲区 信道描述子记录了程序中信道的名字和所需存储大小, 数据类型以及信道所在的机器号(名), 缓冲区为等待队列提供空间。
- 原语实现例程 它有近似机器指令的**send**, **receive**等核层原语; 以及创建、撤消信道指令; 建立、处理等待队列的指令。使原语按其语义实现。每个原语例程最后都调用调度例程dispatcher()。
- 网络通信接口 即使信道附在某一结点, 通信时, 它首先要求远程核创建一 远程同名通道并经网络发送数据。对每一结点而言都要有通信接口。

分布式并发核示意图如下:

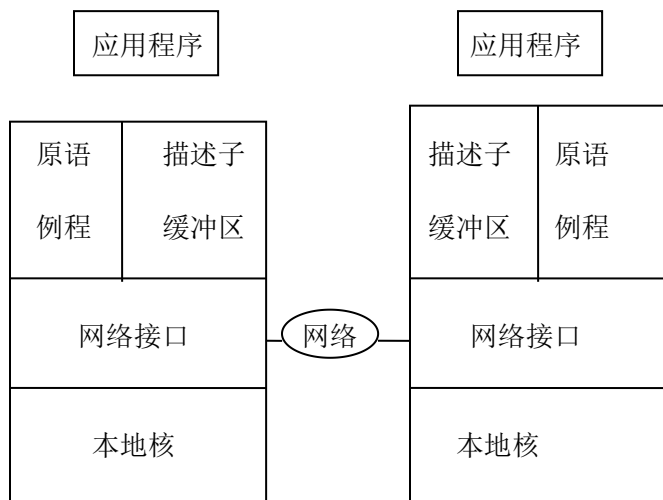


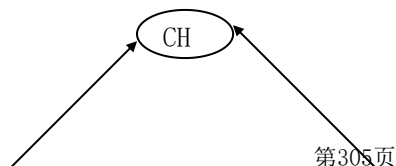
图14-2 分布式并发核示意图

显然, 对于共享内存的单主机多处理器系统, 并发核就简单得多(没有网络接口部分)。

同步通信虽然不用原语, 直接用选择通信语句实现, 它除了比异步更为复杂而外, 在最基本的操作上和异步是一样的。因此, 同步通信的核是在异步核的基础上实现的。例如, 向目标进程发消息, 在异步的不等待的**send**原语上加上等待条件即可实现输出语句的语义。这样, 问题就集中在如何实现同步上了。当前实现同步有两种策略:

(1) 集中式的同步实现

虽然同步通信语句直接写出要与之通信的对方, 实现时人为增加一个服务器, 管理本并发程序的所有通信要求, 这就是所谓的“澄清室(clearing house)”技术, 它将要求通信的进程匹配。我们图示说明:



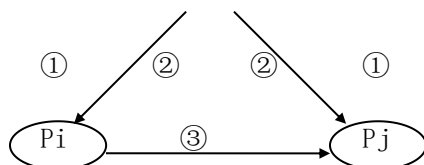


图14-3 利用澄清室的交互模式

在图14-3中，若进程Pi要执行向Pj(作其目标进程)输出的语句，且假定程序正文上端口名和参数类型是匹配的。Pi, Pj首先向服务器CH发消息，即第①步，CH暂存第一条，等第二条(Pi的 ①或Pj的 ①)的消息到达后，匹配。并向Pi, Pj发回可以通信的应答，即第②步。得到应答后Pi才将输出语句中的表达式求值后送给Pj，对Pj的输入语句的变量作远程赋值，做第③步。此后，各进程执行自己的代码。

为了记录等待，澄清室内设一记录，登录要求通信尚未匹配的进程。正因为集中管理，它易于检查进程匹配情况。然而，在分布式系统中，澄清室 要附在某台机器上执行。这样多次来回通信增加了系统通信开销，澄清室往往成为并行核的瓶颈。这样，自然要寻求分散式同步(直接)通信的解。

(2) 分散式同步实现

用异步通信原语实现分散式同步通信，为每个要求通信的进程设立一个匹配信道(输出语句为一端，输入语句为另一端)和一个应答信道。并于有输入语句的进程设阻塞等候队列。所以处理输入语句要复杂一些。如果输出/入语句不出现在警卫子句内处理程序要简单得多，输出语句出现在警卫子句中情况会变得更复杂。因此CSP和Occam的最初版本均不允许在警卫子句中有输出语句。以下给出实现算法框架：

例14-14 分散式协议：显式子句中无输出语句

```

chan match [1..N] (sender, port :int, data:TYPE);
chan reply [1..N] ( );
//各进程均设以下局部变量
    var pending: set of (信道match参数表) :=  $\Phi$ ;
    var sender, port:int, data :TYPE;
//不在卫式子句中的输出语句对应如下:
    send match[目标进程] (myid, port, 输出语句中各表达式);
    receive reply [myid] ( );
//不在卫式子句中的输入语句对应如下:
    if port 端口的pending(阻塞队)中有消息then
        从pending 中撤除一条消息;
        将data 存入输入语句的变量;
        send reply [sender] ();
    endif;
while 若未找到匹配do
    receive match [myid] (sender, port, data);
    if port 匹配 then 将data 存入输入语句的变量;
        send reply [sender] (); endif;
        if port 不匹配 then insert (sender, port, data)插入pending阻塞
    endif;
end.
//若卫式子句中有输入语句，则对应如下:
for 每个为真的布尔表达式do
    if port 端口的pending中有消息then

```

```

    从pending中撤除一条消息;
    将数据data存入输入语句中的变量;
    send reply[sender]();
    执行相应的卫式子句;
    exit;           //跳出for循环
endif;
end;
while 若未找到匹配do
    receive match[myid](sender, port, data);
    if port 以真值与某个输入语句匹配then
        将data存入输入语句的变量;
        send reply[sender]();
        执行相应的卫式子句; endif;
        if 消息不匹配 then
            将insert (sender, port, data)插入pending;
        endif;
    end.
end.

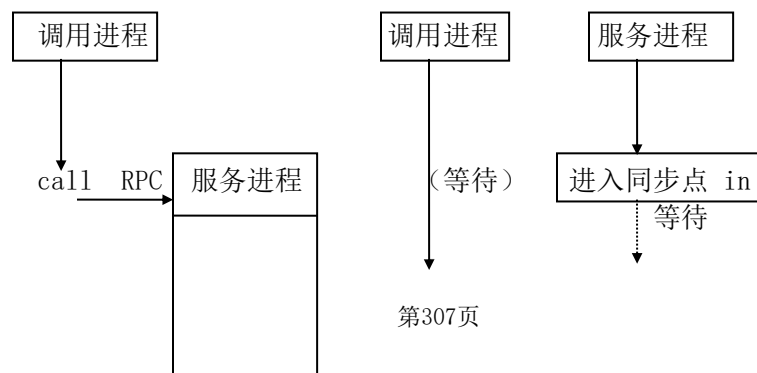
```

至此，我们介绍了消息传送的同步和异步机制。它们是互补的。有些问题用同步解方便且不要求动态缓冲分配。对于其它一些问题，如广播式通信，若采用扩张收缩算法则同步就不及异步了。但请注意正是由于同步是用异步原语实现的，它们都少不了信道，而信道通信在原语级是单向的。如像客户/服务器之类的应用，发消息和应答要设两个信道。特别是当一个同步进程向多个服务器发消息时，则有二倍如此多的信道。这就导致了大量信道。相比之下实现过滤器信道要少一半，而实现等同(peer)进程，信道虽然是双向的，但不设专为应答的利用率不高的信道。

14.3 远程过程调用和会合

以上讲的异步和同步通信是以嵌入有通信语句或原语的进程为基本的通信单元。70年代末，结构化程序发展为模块的高级形式，并发程序研究中也提出将监控器模块与同步消息传送结合的机制，即远程过程调用(Remote Procedure Call 简称RPC)。

远程过程调用特别适合于分布式网络中各结点间作客户/服务器式通信。每个结点上是一个封装的模块。模块内存共享数据和一系列操作。模块的对外界面上程序员可指定输出的操作。外部(另一模块)可用call语句调用其中的操作。如同顺序进程一样，调用进程(客户)发出调用之后，被调用进程首先创建这个服务进程，与之通信，执行服务进程，将返回值返回到调用进程。在此期间调用进程一直处于等待。因此调用、返回应答和结果均走一个信道，大大减少了信道数目。RPC完成调用进程执行自己原进程，服务进程终止。其执行示意图如图14-4的左图。



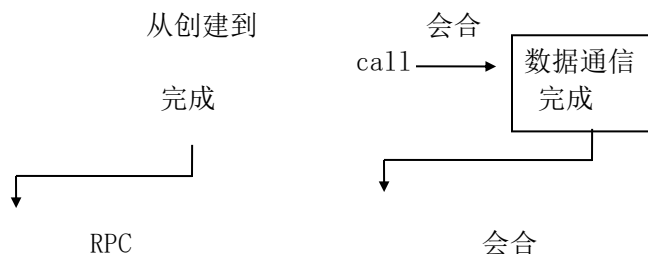


图14-4 RPC和会合的执行示意图

RPC仅仅是提供模块间的通信机制，在模块内部的各进程依然有是否并发执行问题。例如，某网络结点是一多处理器的工作站。这种情况下，模块内的并发进程的同步与互斥与块外的RPC调用引起的同步与互斥要求就要统一考虑，似乎也有些复杂。70年代末的Ada设计者在此基础上发展了会合 (rendezvous) 机制。

会合把远程过程调用的操作建立在进程内，由进程输出操作。虽然，它们依然封装在一更大的模块内。大封装模块控制本地机上的并发，而远程调用直接对进程调用。会合执行的示意图如图14-4 的右图。下面我们分别举例说明。

14.3.1 远程过程调用RPC

如上所述，RPC适于分布式网络通信。我们为某一结点写一并发模块(对于多处理器的机器，一结点即单一处理器)，其一般形式是：

```

module Mname
    输出(外部可见)过程型构;
body
    本模块内共享变量及初始化代码;
    输出过程的体;
    局部于本模块的过程和进程;
end.

```

其中过程型构与过程体写法和顺序程序没什么两样。例如，当有外部模块对本模块的一个操作调用时，可写为：

```
call Mname.opname(AP_list);
```

本模块内调用和一般模块内的过程调用一样：

```
call opname(AP_list);
```

(1) RPC实现模式

每个模块在各自的地址空间。每当有远程调用时，服务模块创建一服务进程，调用进程一直延迟到整个服务完成。此时两进程的同步是隐式的。如果同一时间只允许一个远程调用进入本模块，就实现了对共享数据的互斥访问。多个调用者们就要排队等待。这也许对于想进入本模块调用另一个操作进程的远程调用似乎不公。但由于共享数据，另一操作也会更新共享数据，一个一个来要安全些。因而，在上层调度中若能设计出更好的算法可进一步挖掘并发性。

现在的问题是块内各操作进程在有了远程调用之后如何同步与互斥。一个办法最简单：块内同一时间只能有一个进程执行(不管是局部调用还是远程的一起排队)。这样最安全，隐地实现了互斥，共享变量无需保护。但不可取，丧失了大量潜在的并发性。

现在多数RPC实现是支持模块内并发执行的。其实现模块可以如前所述的异步、同步、卫

式同步模式。远程调用只是其并发成分之一。我们举例说明。

(2) 分布式系统文件缓存器

在分布式系统的文件系统中有一个统一的文件服务器File_Server模块管理磁盘上的文件。文件以固定大小(1024字节)的块存放在服务器所管磁盘上,并为读写块提供两个操作readblk, writeblk。每个结点(机)上都设有文件高速缓存File_Cache的局部模块以管理本地的文件高速缓存区。应用程序要访问文件它就调用本地File_Cache中的read和write过程。

当应用调用读/写时File_Cache首先检查所需处理数据其字节是否超出缓存块(仅1024字节)。如不超出则立即满足客户的调用。若超出或不是所需数据,则调用File_Server中的readblk过程以得到所需数据。

写操作情况类似,应用程序写下的数据一般放在局部缓存之中。存块满了或有其它要求时则调用File_Server中的writeblk过程,将它入磁盘。

例14-14 分布式文件系统的文件缓存

每一应用进程均有一File_Cache模块。由于读和写不同时执行,本块内无同步要求。如块外有多个应用进程访问本块,设信号灯变量保护。

```

module File_Cache                                //每一无磁盘工作站均存一份
  op read (count:int; res buffer[1..1024]:char);
  op write (count:int; buffer[1..1024]:char);
body
  var cache_block:TYPE;                          //文件缓存块
  var 有关文件记录信息描述子变量;
  var 按需要的信号灯变量;
  proc read(count, buffer) is
    if 所需数据不在cache_block中then
      选一适用的cache_block;
      call File_server.readblk(...);
    endif;
    buffer := 从cache_block 中取count字节;
  end;
  proc write(count, buffer) is
    if 不宜写then
      选取适合写的cache_block;
      call File server. writeblk(...);
    endif;
    cache_block := 将buffer的count字节写入;
  end;
end File_Cache.

```

逻辑上为文件服务器的机器装以下模块:

```

module File_Server
  op readblk(field, offset:int; res blk [1..1024]:char);
  op writeblk (field, offset: in; blk(1..1024):char);
body
  var cache_block:TYPE;                          //磁盘缓存块
  var queue :QTYPE;                              //等待访问磁盘的队
  var 按需要的信号灯变量

```

```

proc readblk (field, offset, blk) is
    if 需要的块不在cache_block之中then
        将读要求存入磁盘的等待队;
        等待处理读;
    endif;
    blk := cache_block;
end;
proc writeblk (field, offset, blk)
    选取一可写cache_block;
    如必要, 将写要求入等待队;
    等待至磁盘可写;
    cache_block := blk;
end;
Disk_Driver :: loop
    等待磁盘访问要求;
    启动磁盘操作; 等待中断;
    唤醒等待此要求的进程并执行之;
end loop;
end File_Server.

```

其中Disk_Driver是处理要求访问磁盘的驱动进程。**res**指结果参数。

14.3.2 会合

会合由进程输出操作, 因而, 每当有远程调用时进程已经是激活了的。这和模块输出操作, 远程调用时再激活进程不一样, 故而得名, 即以两活动进程相会合以求得同步。服务进程一般定义为:

```

Pname::各操作Opi的声明;
    局部变量声明;
    in/accept 语句和条件;           //接受RPC的入口点
    opi (FRG_list);
    其它局部操作;
    调用时和RPC一样:
    call Mname.Pname.Opi (ARG_list)

```

(1) 会合实现模式

简单的会合除了服务进程已激活, 且服务后继续自己进程而外, 和RPC没什么两样。必须通过call语句求得同步的会合, 即调用者/被调用者谁先到达call/in谁先等待对方到达。但RPC是进程级的(一个过程一个进程), 且允许有多个并发进程进入封装这些操作的模块。会合是操作级的(一个进程有若干个操作), 调用的是进程输出的操作, 因而, 操作不能是并行的, 只能同时有一个操作在执行。而每个操作都有自己的等候队列。也就是进程执行到本操作再查看等待队列, 如无候选调用, 处于等待。如有则与最优先的会合。通信后完成服务, 再做下一操作。正是由于定义输出操作, 带来了卫式通信的方便性。会合的一般形式是:

```

in Op1 and B1 by e1→s1
    ....
    Opn and Bn by en→sn

```

end in

其中 O_{pi} 是一个进程中的第 i 个操作， $B_1 \cdots B_n$ 是 n 个条件，也称同步表达式，谁满足条件执行谁。还要看满足条件的候选队列中是否有调用者。如有，则表达式集 e_i (也称调度表达式) 与变量结合执行语句集 s_i 。否则，处于第一个满足的 B_j 且为空队的进程上等待。

如果略去 $B_1 \cdots B_n$ 则为会合的一般式。只要有了 O_{pi} 调用则执行 O_{pi} 对应的 s_i ，否则进程等待在 in 处，轮番测试(查看各等待队列)是否有调用。

如果 O_{pi} 同名，则 $B_1 \cdots B_n$ 构成条件选择(当然要用到重载概念)。同一名字 O_{pi} 下根据不同 B_i 选择不同的 s_i 执行。取消某一 B_j 可构成缺省选择。这样，可增加中断、中止等命令，增加了表达能力。

卫式通信同样可以用在调用方：

```
call  $O_{p1}$  and  $B_1$  by  $e_1$ 
```

```
....
```

```
 $O_{pn}$  and  $B_n$  by  $e_n$ 
```

```
end call
```

仅当卫式操作与调用进程中的其它参数有关时，这样使用。

(2) 会合的实例算法

会合中同步由会合机制保证，而互斥对于同一进程中各操作也是自然保证的。多进程的互斥(不一定是共享变量，资源)访问则要靠程序员编程，以下是会合实现的不同进程。

• 会合实现过滤器

有界缓冲区是一典型过滤器，一些进程向缓冲区存入(deposit)数据，另一些进程从缓冲区取出(fetch)，缓冲区(buffer)本身是一个进程。实现算法如下：

例14-16 以会合实现有界缓冲区

有界缓冲区 $buf[1..q]$ 是本地机上的共享变量。 $front$ ， $rear$ ， $count$ 为控制和警卫变量，所有操作共享。

```
Buffer ::
```

```
  op deposit( $data:TYPE$ ), fetch(var  $result:TYPE$ );
```

```
  var  $buf[1..q]:TYPE$ 
```

```
  var  $front := 1$ ,  $rear := 1$ ,  $count != 0$ ;
```

```
  loop
```

```
    in deposit ( $data$ ) and  $count < n$  do
```

```
       $buf[rear] := data$ ;
```

```
       $rear := rear \bmod n + 1$ ;
```

```
       $count := count + 1$ ; end;
```

```
    in fetch ( $result$ ) and  $count > 0$  do
```

```
       $result := buf[front]$ ;
```

```
       $front := front \bmod n + 1$ ;
```

```
       $count := count - 1$ ;
```

```
    end;
```

```
  end loop.
```

当本进程激活进入**loop**则测试deposit和fetch的等待队(不分先后，次序由系统定)，如无则等待，否则检查警卫条件 $count$ ，会合后进入过程体，执行完重新循环测试。

• 会合实现客户/服务器

用会合实现客户/服务器结构是最适宜的。只需注意写服务器，因为客户进程只需发出操作调用就行了。以下是操作系统中最常用的时间服务器的例子。

例14-17 用会合实现时间服务器

```
Time_Server :: op get_time( ) return  $time: int$ ;
```

```

        op delay(waketime: int);
        op tick();    //仅由时钟中断处理器调用
var tod :=0;
loop
    in get_time() return time do
        time := tod;    end;
    in delay(waketime) and waketime <= tod do
        skip;    end;
    in tick() do tod := tod + 1;        //再启动时钟;
    endin;
end loop.

```

本服务器可为任何客户进程提供获取时间和延迟操作。其中tod记录当前时间。若每秒中断一次，tod+1。唤醒时间是tod+延迟的‘绝对’时间。

14.3.3 Ada的任务

Ada是军用嵌入式语言，并发是其重要的一部分。Ada的并发机制是任务（task一般指一个进程），任务是三大组成程序单元之一（程序包、子程序）。但由于激活的关系它本身不是独立的编译单元。必须嵌入在子程序或程序包之中。如：

| | |
|----------------------------------|--|
| procedure PNAME is | package PKG_NAME is |
| 任务和其它声明; | 任务和其它声明; |
| 任务体; | end PKG_NAME; |
| begin | |
| 顺序语句; | package body PKG_NAME is |
| end PNAME; | 局部声明; |
| | 任务体; |
| | 过程体; |
| | begin |
| | 初始代码; |
| | end PKG_NAME. |

当子程序开始执行时（到begin处）子程序内定义的任务全部激活，顺序部分也是一任务。程序包规格说明和体分别表示的方法也决定了任务的分别表示。当引入程序包到内存并确立时（在包体的begin处）包体内定义的任务全部激活，并发执行。这是因为它采用会合机制。任务先激活再同步通信。

14.3.3.1 Ada的任务结构

任务规格说明和体是同名对应的。Ada任务规格说明中只有入口（entry）声明，它与体中accept语句对应。accept相当于前述in：

```

task TNAME is
    entry ENAME (FP_list);    //可以多个
end TNAME;
task body TNAME is
    局部声明;
begin
    accept ENAME (FP_list) is    //对应多个

```

```

    语句序列;
  end ENAME;
end TNAME;

```

任务激活后就是一个进程。其它进程通过
`[call] TNAME.ENAME (AP_list);`
 调用入口，会合后通信得以实施。语句序列中可以继续有入口调用和过程调用。

14.3.3.2 通信与同步

以上task结构实现Ada的简单通信：调用进程执行到**call**，被调用进程执行到**accept**，谁先到谁等待(同步点)至会合(形实参匹配并执行**accept**操作的体)后，调用、被调用进程分离，各自执行以下动作。调用进程一直等到送回返回值(如果有)。每个**accept**语句有一等待队列，当多个调用者竞争会合时，按先来先服务原则一一会合，解决了多对一的通信。当有多个客户发出多个入口调用时，Ada提供选择会合机制以便程序员控制同步，实现公平性。

(1) 简单选择

```

select          --进程执行至此，若E1, E2, ...
  accept E1;    --的等待队列中有调用者在等待，则与其中之一会合，会合完跳
or              --至末端。若全无等待会合者，本进程一直处于select处等待
  accept E2;
or
  ....
end select;

```

(2) 否则选择 防止在select处死等

```

select
  accept E1;      --到达此处，若E1, E2, ...的等待队列中均无候选者，
or                --立即执行else的op操作。
  accept E2;
or
  .....
else.
  必选操作op;
end select;

```

(3) 卫式选择 实施条件同步

```

select when B1=>    条件B1满足且E1的等待队列中有候选者，E1会合，或E2
  accept E1;        否则一直处于select处等待
or when B2 =>
  accept E2;
end select;

```

(4) 延时选择

```

select          各等待队列中均无候选者，延迟一分量执行TAKE_A_BREAK。
  accept E1;     否则会合后跳过延迟部分。
or
  ....
  delay 1.0 * MINUTS;
  TAKE_A_BREAK; --无需匹配的过程调用。
end select;

```

在调用(客户方)同样有**select...or...else**和**delay**语句，语义同。

14.3.3.3 任务类型及实例任务

Ada的任务是一进程，也是任务类型的一个实例。从管理角度Ada给出显式任务类型，非类型的实例直接写作task。由于任务类型化，其实例在程序中用法和一般抽象数据类型的实例完全一样。以下是Ada任务类型的例子。尽可能结合了其它同步机制。

例14-18 哲学家就餐问题Ada解

本例和前述解题意略有不同。哲学家是随机进入餐厅，吃完就走。等不到叉子才利用时间思考。本程序是一模拟程序：

```

procedure DINING is
  task type FORKS is
    entry PICKUP;
    entry PUTDOWN;
  end;
  task body FORKS is
    begin loop
      accept PICKUP;           --过程内有记录叉子拿放的操作和真值表。
      accept PUTDOWN;         --此处略
    end loop;
  end FORKS;
  task type PHILOSOPHERS is
    entry LIFE (IC :in Integer);
  end;
  task body PHILOSPHERS is
    I: Integer;
    begin
      accept LIFE (IC :in Integer) do
        I:=IC; end;
      loop                    --模拟哲人入座后行为
        ROOM. ENTER;
        FORK(I). PICKUP;
        FORK((I+1) mod 5). PICKUP;
        delay EATTIME (I);    --可以是随机数发生器实现
        FORK(I). PUTDOWN;     --的函数调用
        FORK((I+1) mod 5). PUTDOWN;
        ROOM. EXIT;
      end loop;
    end PHILOSOPHERS;
  task ROOM is              --简单任务
    entry ENTER;
    entry EXIT;
  end;
  task body ROOM is
    OCCUPANCY:Integer := 0;
    begin
      loop
        select
          when OCCUPANCY < 4 =>    --不让同时有5人进入，以免死锁
            accept ENTER do

```

```

        OCCUPANCY := OCCUPANCY+1;
    end;
or
    accept EXIT do                --已有4人先出再进
        OCCUPANCY := OCCUPANCY-1;
    end;
end select;
end loop;
end ROOM;
FORK:array (0..4) of FORKS;
PHILO:array (0..4) of PHILOSOPHERS;
begin                            --FORKS. PHILOSOPHERS, ROOM同时激活
    for I in 0..4 loop
        PHILO (I).LIFE(I:Integer);
    end loop;
end DINING.                      --所有任务终止, 全部停止。

```

主程序使五位哲人都活动起来, 显然第三位以后就有竞争。一共11个进程。

14.4 多原语的并发机制

RPC虽集模块化和同步通信之大成, 但不宜于等同(peer)到等同进程的交互。因为远程过程调用带回的信息(通过一个信道)仅仅是返回值。服务器难于远程调用客户进程为其服务。这对于频繁与相邻结点通信的等同进程是致命的。会合也好不了多少。虽然进程可直接通信, 但一个进程不能同时执行call和输入语句。要实现频繁与相邻结点通信就要利用辅助进程或执行异步算法。因此, 人们想到能否综合各种机制的优点。为此, 我们对上一章和本章讲的各种机制作一小结和回顾。

14.4.1 进程交互主要技术的回顾

在单CPU主机上, 人们最初用忙等待的询问保护锁的办法来实施进程间的同步与互斥, 但由于编程技巧性能强语义不直观, 加上低效, 导致了信号灯理论的建立。早期的操作系统设计得益于信号灯。它用原语级的P、V操作, 协调竞争进程的同步, 并保护对共享资源的互斥访问。原则上, 无论哪种背景的进程交互均可用信号灯模拟出来。

以后的发展分成了两路, 如图14-6, 以单主机多处理器的背景环境的并发程序中, 利用资源共享(一块存储或叫一组变量)来实施进程的同步与互斥。将信号灯对临界段的保护发展成为条件临界区和监控器技术。由于并发进程访问共享存储不必经过线路传递。把控制同步的原语wait, signal嵌入在过程(进程)之中, 而过程又集中于模块内。这种结构化的编程使进程的同步与互斥易于显式控制, 利于程序的开发维护和修改。这一分枝由监控器的嵌套又引伸出路径表达式技术。共享存储确定后, 并发程序的编制着眼于过程, 因而, 也叫它们是面向过程的。

基于分布式网络背景的并发程序处于另一个极端。几乎没有共享的存储。所有进程交互全靠通信完成。它发展了信号灯的示信同步的一面。借助send, receive原语和信道实施的异步通信可以描述网络上的四类进程(过滤器、客户、服务器、等同), 并可方便地实现广播式, 令

牌传递、伸缩(心跳)等算法。然而，当网络结点众多，同步要求较高时，异步通信又不及同步通信。总的说来，它们是面向消息的。

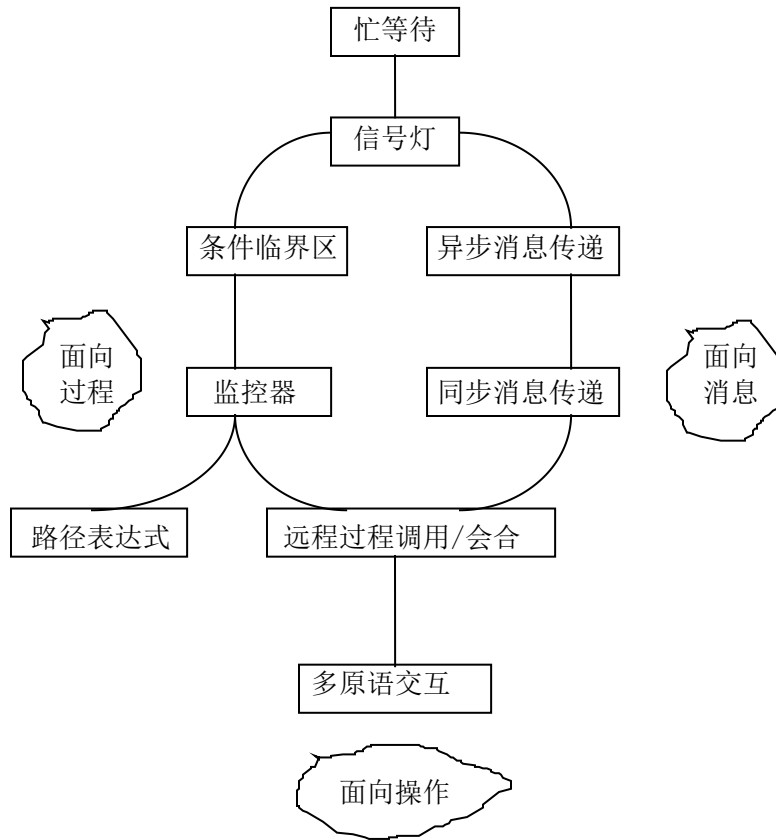


图14-5 进程交互技术的发展

和信号灯一样，面向消息的进程交互是原语和语句级，把结构和模块集中控制的技术与其结合就有了远程过程调用和会合。在分布式结点内的并发仍可用数据共享，各结点间用远程通信，信道容量为零也就成了单主机多处理器的模型。适应面好，表达方便，是当今软件平台上客户/服务器交互的主要模型。但进一步的等同进程通信则要求其更加灵活，于是多原语进程交互成为当今研究热点，因为它可以实现人们多年积累的各种并发算法，适合于各种背景。显然，和其它软件一样，“万能”意味着“低效”。所以，也是大有可为的研究领域。

14.4.2 多原语并发程序表示法

程序结构采用模块，可对应分布式某结点：

```

module  Mname
    可见Op(操作)声明;           //输出操作
body
    局部变量声明;
    初始化代码;
    过程声明(包括输出的和局部的); //proc或in
  
```

进程声明 (Mname 内的进程);
end.

同步和异步调用Op:

```
call Mname.Op(ARG_List)      //同步用call
send Mname.Op(ARG_List)      //异步用send
```

同步调用**call**，调用者等待得到返回值。如果被调用是**proc**，则要创建服务进程，匹配、计算、返回。如果是**in**则对方已激活，是会合。同样，异步调用**send**，不要求等待返回值，其余同。这样组合形成了四种不同的并发机制：

| 调用者 | 被调用者 | 效果 |
|-------------|-------------|----------|
| call | proc | (远程)过程调用 |
| call | in | 会合 |
| send | proc | 动态过程创建 |
| send | in | 简单异步消息传递 |

其中如果

```
in Op (f1, ..., fn) do
  u1:= f1;
  ...
  un:= fn;
endin
```

其效果等同于**receive Op (v1, ..., v2), in**语句退化为**receive**。

以下是多原语用于数据库(或文件)读/写的例子。

例14-19 带封装数据库的读写器

显然ReadersWriters模块要为客户进程输出read, write操作。由于多个客户可同时读，则采用RPC机制创建多个读进程。但写只设一个进程，且仅当无读者进入数据库时才写，设保护：读者数nr=0。多个写者以会合方式进入数据库写。Writer进程可设在本地也可以在另一模块中。以下按本地给出程序如下：

```
module ReadersWriters:
  op read (res results), write (new values);
body
  op startread( ), endread( );
  var buffer...;                //被数据库或文件数据的变量
  proc read(results) is
    call startread( );          //RPC call_in 至writer 进程
    读数据库数据的操作
    send endread( );            //send_in 至writer进程
  end;
  Writer ::
    var nr :=0;
    loop
      in startread( ) do dr := nr+1; end;
      in endread( ) do nr := nr-1; end;
      in write(values) and nr=0 do
        写数据库的操作;
```

```

        end in;
    end loop;
end ReadersWriters.

```

客户进程可以**send** 或**call** read 和write操作。nr=0保护了对数据库的互斥访问。

14.4.3 SR语言

SR是同步资源的字头简称，设计于70年代末，从多消息传递原语扩充了共享内存而来。它基于小量正交概念设计，动态命名，进程动态创建，位置设置等。是一个表达能力很强的语言，用于写类似Unix操作系统。

(1) 程序结构

SR的基本单元是可分别编译的模块。有**resources**（资源）和**globals**（全局共享）模块。其模块规格说明是：

| | |
|----------------------------|--------------------|
| resource name | global name |
| 输入规格说明； | 输入规格说明； |
| 输出操作和类型声明； | 输出操作和类型声明； |
| body name (FP_list) | body name |
| 变量和局部声明； | 变量和局部声明； |
| 初始化码； | 初始化码； |
| 过程； | 过程； |
| 进程； | 终止化代码； |
| 终止化代码； | end name |
| end name. | |

resource 和**global**模块均为样板，通过创建语句创建它的实例模块：

```
rcap := create name (AP_list) [on machine]
```

实参表AP_list和形参表FP_list匹配。匹配后执行body内的初始化代码，创建body内的进程，支持动态数组，并按程序员规定的先后初始化。初始化完成后**create**返回并赋给rcap，它现在即实例模块。体中的过程包括输出的和本块内的。

global模块一般是不带参数的，用来提供共享的类型、变量、操作、过程。**global**的实例存放在需要它的虚机(地址空间中)。每当创建**resource**实例时，若有输入的**global**名字，则该**global**实例自动隐式创建。

在缺省情况下，执行**create**语句所在的虚机得到一rcap实例，否则按[on machine]的指示在分布式名叫machine的结点的虚机上创建rcap实例。

SR程序总有一个主资源，一开机就隐式创建它的实例并执行之。主资源初始化代码执行时它就创建其它资源的实例。所有资源均动态创建，因此设销毁语句**destory**动态地销毁：

```
destory rcap;
```

如果有终止化代码，执行后所在实例撤销，并释放所分配的存储。只有当所有进程终止或冻结SR程序才终止。此时，运行系统执行主资源中的终止化代码以及各**global**中的终止化代码。

(2) 通信和同步

在SR中，同一资源内的进程可以共享变量，可以使用各种原语实现通信和同步：信号灯、异步消息传递、RPC、会合。SR既适合分布式也适于共享内存的多处理器机器。

操作声明可以出现在资源规格说明、资源体、甚至在进程之中。出现在进程之中则谓之局部操作。该进程可将局部操作的功能传递到另一进程，即后者可引用前者的操作，这样就可以支持连续调用。

调用操作用**call**则为同步，**send**为异步。在资源中声明的操作，实际上是提供了一种可直接引用的功能。在资源间可以自由传递，故通信路径是动态的。资源中所有的进程执行都是并发的。

输入语句**in**实施会合，语句包含同步和调度表达式，还可以有**else**部分作为必选动作。

SR保留**receive**语句是输入语句的退化。它和**send**的无参特例，正好实施P,V操作。中断由**send**语句异步完成。此外，还保留显式的**reply** 和**forward**语句。前者从**proc**返回值。**forward**传递对另一进程调用以得到进一步的服务。如果执行了**forward**，调用者阻塞，直至另一进程结束后返回。

终止化代码是SR的独创，当所有进程终止或冻结，甚至死锁时，它可以打印出结果状态和有关时间的信息。

14.5 并发程序设计语言

高级程序设计语言扩充并发机制，最早可溯源至PL/1和Algol-68。20多年以来，随着各种并发机制的研究推出了数十种具有并发机制的高级程序设计语言。直到目前，还没有一种并发程序设计语言能统治并发程序设计整个领域。其原因是硬件背景、应用领域和程序设计模型的多样性。

本章给出有一定影响的并发高级程序设计语言(如下表)，并列出它们的同步和通信机制。从表中看出：第一，相当多的语言不止一种并发机制，其原因也是由其背景和用途不同。第二，有些语言是从有一定影响的顺序语言扩充而来。第三，有一些只是试验室使用或小型应用，谈不上正式语言。但对学术研究有较大影响的。如CSP，DP，PLITS，Joyce，StarMod等。有一些力图成为工业产品，如Mesa，Ada，Concurrent C，C++，Java。本表从学术观点选取。

表14-1 主要的并发程序设计语言

| 语言 | 年代 | 并发机制 | 备注 |
|-------------------|------|-----------------|----|
| DP | 1978 | CCR+RPC | |
| Edison | 1978 | CCR | |
| Argus | 1982 | CCR+RPC+原子事务 | |
| Lynx | 1991 | RPC+会合+CCR | |
| Concurrent Euclid | | 监控器(SW) | |
| Concurrent Pascal | 1975 | 监控器(SW) | |
| Modula-3 | 1985 | 监控器(SC)+协例程和锁的包 | |
| Path Pascal | 1979 | 监控器+路径表达式 | |
| Pascal Plus | 1979 | 监控器(SU) | |
| Turing Plus | 1983 | 监控器(SC+SW) | |
| Mesa | 1979 | 监控器(SC)+RPC | |
| Emerald | | 监控器+RPC(面向对象) | |
| Actor | | 异步消息传递(基于对象) | |
| PLITS | 1979 | 异步消息传递 | |

| | | | |
|----------------|------|--------------|----------|
| NIL | 1982 | 异步消息传递+会合 | |
| Gypsy | 1979 | 异步消息传递 | |
| CONIC | | 异步消息传递 | |
| CSP | 1980 | 同步消息传递 | |
| Joyce | | 同步消息传递 | |
| Occam | 1987 | 同步消息传递 | |
| Concurrent C | 1985 | 会合+异步发送 | |
| Concurrent C++ | 1987 | 会合 | |
| Ada | 1983 | 会合 | |
| SR | 1982 | 多共享+消息原语 | 92年SR2.0 |
| Star Mod | 1980 | 多消息原语 | |
| Linda | 1986 | 带共享元组空间的消息原语 | |
| Java | 1996 | 类库中支持多原语 | |

本表中的某些语言，本书已作轮廓介绍。由于篇幅所限，不能对这27种语言一一介绍。以作者的观点，读者如对今后并发语言发展有兴趣，请注意以下语言：

(1) Ada是专为嵌入式使用的军用语言，并发是Ada的重点。它所采用的会合机制直接支持客户/服务器应用。也是当今软件工程平台(异质网上)最普遍的技术，所以并发C和并发C++也都采用会合机制。95年3月Ada95新版本通过，并发机制保留增加面向对象功能。在并发程序设计语言中它依然是有较大影响力的一支。Ada语言的并发部分我们在14.3.3中已经介绍。

(2) Linda它本身并非程序设计语言，它是访问所谓元组空间的一组原语。任何顺序程序设计语言都可以用Linda原语扩充以产生并发程序设计的变体。已有C_Linda、FORTRAN_Linda。

Linda综合了共享变量的同步和异步消息传递功能。元组空间(TS)是共享的，其中放的是有标记的记录(元组)，也是一共享的通信信道。各进程对TS有六种操作原语：

| | |
|-------------|------------------------|
| out | 相当于send 向TS 送数据。 |
| in | 相当于receive 从TS 取元组。 |
| rd | 共享变量向局部变量赋值。 |
| eval | 创建进程元组。 |
| inp | 谓词，若TS中有匹配元组返回真值。 |
| rdp | 谓词，若e延迟等到TS中有匹配元组返回真值。 |

元组空间是被动的数据元组和主动的进程元组的无序集。数据元组看上去如同一般的带标记的记录。表达计算的共享状态：

(“tag”，value1，…，valuen)

它可以分布存放在网络各结点上，即分布式数据结构。其中每个值都由一进程完成：

out (“tag”，expr1，…，exprn)

括号中是n元素的进程元组，几个进程异步地计算各表达式，终止后就成了数据元组，按out送入TS(逻辑空间)。从TS中取出元组则按in描述的样板：

in (“tag”，field1，…，fieldn)

fieldi可以是表达式，也可以是?var，即执行进程局部变量的值。执行本句的进程一直延迟至TS中至少有了一组和此样板一样的值。取回执行进程，并在TS中消除此一样板。

eval (“tag”，expr1，…，exprn)

则创建了n个子进程，且异步并发执行，每个表达式即函数或过程。执行完成即成为被动的数据元组。

利用成对的out，in可实现点到点的消息传递。进程交互是在向TS读/写或生成数据元组时

进行的。

(3) Occam它是CSP的直接后裔，即同步消息传递，Occam程序的基本单元是三个基本的进程：赋值、输入、输出(相当于其它语言语句)。它有顺序和并行构造子将基本进程组成传统的进程(以PROC过程作为大进程样板，过程参数即信道。这样一个网络结点机的内部并发和外部并发可自由控制)。例如，若信道display和keyboard已和物理设备连接。以下Occam程序实现从键盘读入字符ch，再给信道comm送出显示。程序不用关键字，只以错位排列表达嵌套段落关系。

```

CHAN OF BYTE comm          //BYTE为许可通过的数据类型
PAR                          //以下两段并行执行
WHILE TRUE
  BYTE ch:                  //BYTE型局部变量
  SEQ                        //以下消息是顺序的
    Keyboard ? ch           //从信道keyboard输入ch
    comm ! ch               //将ch输出至comm
  WHILE TRUE
  BYTE ch:
  SEQ
    comm ? ch               //如果comm中无字符则等待
    display ! ch

```

本程序由PAR指出了两个进程，SEQ指示一个进程顺序执行。

Occam是独立的并发语言。但它常用于VLSI实现的具有分布式存储的多处理器作传送机(transputer)，充传送机的机器语言。

(4) Java及SR是多原语语言，92年已发展至SR2.0版。程序员可通过SR控制分布式计算成分置放在什么结点上。是其它语言不及的。SR我们在14.4.2节中已作了介绍。

(5) 各种程序包和程序库，在并发程序语言未发育完全之前各厂商为了支持并发程序，都在不同程度上提供程序包和库例程。例如，Sun工作站的操作系统SunOs以socket包支持远程过程调用和异步消息传递。以Sequout并行程序设计库例程支持自旋锁，信号灯。intel超立方中的通信库可支持同步、异步消息传递。但它们都是特定机器系统中的库例程，没有移植性。在这些系统上写并发程序，首先写出顺序程序，然后用库例程创建进程，同步化，通信。事实上，当前不少的并行应用都是按此方式作出的。当然，程序员要非常了解本机的体系结构和OS，并作机器间的类型检查。尽管它们不是方向，但各包、库应用的经验是新的高层并发语言的原材料。

Java是当今网络上编制并发程序的最重要语言，也是基于低级原语的并发语言，在语言层次上只扩充了两个关键字Synchronized和threadsafe，其原语在thread类中提供支持线程编程。由于该语言为网络上需要而设计，并发线程编程只是其综合性能的一个方面。本书第15章还要综合介绍。

14.6 小结

本章介绍了并发程序中进程交互的各种机制，以及重要并发程序设计语言采用的机制。

- 基于变量共享的并发机制有条件临界区(CCR)、监控器和路径表达式。
- 条件临界区把共享变量置于资源并在与其对应的region中操纵它。易于显式控制同步；编译时即可查出不互斥；创建进程只看自己条件，简单；易于正确性证明。缺点是低效；进程与共享变量耦合太紧；利写不利读。学术上价值大，实用采用少。

- 监控器把共享变量和其上操作置于一个模块。显式控制同步。易于开发、测试、维护。是封装对象程序结构的先驱。

显式控制同步的两个基本操作是wait和signal每次只有一个进程进入监控器，仅当它阻塞或返回时才能有第二个进入。由于有了等待和唤醒原语加上同步条件控制就很方便了。

- 以监控器实现条件同步有多种做法、常用的有复盖条件变量(两级条件变量)；在无占先情况下传送条件，使同步控制安全；在监控器内组织进程会合。

- 各种语言在采用监控器技术时，对于控制进程同步的wait和signal两个原语有少量语义不同的解释。其原因是应用目的和实现背景的差异。它们是：

在无占先情况下：

自动信号AS 只设条件取消signal自动唤醒。

信号和继续SC 执行signal的进程发信号后继续进行。其它进程不得入监控器。

在有占先情况下有：

信号和出口SX 发信号的进程发完后返回或出口，不阻塞等待。

信号和等待SW 发信号的进程发完后等待执行。

信号和急等SU 同SW，但在新进程进入监控器之前保证被唤醒。

理论上这五种机制是等价的。

- 为保持互斥访问，每一时刻只有一个进程进入监控器，称闭式调用监控器。当有嵌套监控器应用时，这种方式易于死锁。为此有开式调用监控器，即当嵌套调用下层监控器内过程时，上层互斥自动解开，待下层返回后又自动恢复互斥。

- 如其在监控器内以wait, signal 控制监控器内各进程执行次序，不如显式指出各进程执行顺序。路径表达式即取消wait, signal原语直接指明进程执行路径的表达式。

- 基于通信的并发机制，中心问题是信道命名方式和同步化。异步通信必须有显式信道，然后借助原语send和receive收发消息。同步通信必须指明两进程的同步点，通过输出、输入语句，进程间直接发消息，信道概念退化为路牌。

- 异步通信信道可为全局的，称为邮箱。可为局部于某进程的，称为端口。两进程固定共享一信道称为链。

进程执行异步原语send从不等待。receive一般情况也不等待，但无数据时也要等待。异步通信可实现无实时要求的过滤器进程和客户/服务器进程。

- 同步通信直接以对等方式写输入输出语句。当只有一个信道时，信道名可省。

CSP是同步通信的典型模型，它采用卫式通信： $B; C \rightarrow S$ 。B是条件C是通信(输入/出)语句，S是有关此次通信的消息或操作。当C不出现即顺序程序中的卫式语句。当B不出现即一般同步通信。通信语句中的“?”、“!”指明消息传递方向。

- 同步通信下层实现用并发核。分布式系统上各结点均有一核。核由描述子和缓冲区、核原语、网络通信接口三部分组成。核原语依然借助send、receive借助核实现同步的策略有集中式(安全、低效)和分散式。为了减低复杂性卫式语句中不得有输出(通信)语句。

- 远程过程调用借用调用这一概念，达到同步。在分布式系统上，调用通过某一信道(网路)只等到对方进程服务完。PRC是模块之间通信机制。被调用过程，在同步交换信息期间，从创建到撤销。

- 会合把远程调用建立在进程内。即进程已经是激活的，远程调用调用它允许(输出)的操作。进程封装在更大的模块内(一般放在一个结点上)。这样，本地进程交互和远程进程交互可以直接显式控制。为并发程序的编制提供了更大的方便。

- 远程过程调用是进程级的(一个过程一个进程)，且允许多个调用进程进入同一封装模块。会合是操作级的(一个进程有若干个操作)，同一时间只有一个操作在执行。因此，为方便编程设选择操作的控制以实现卫式通信。

- 远程过程调用和会合适合于客户/服务器进程。等同进程通信要求更加灵活的机制，多原语 即综合模块的监控器、异步通信、PRC、会合各种机制(后者已包括同步)。如假定程序是模块封装进程。调用者call、send调用过程而接受者以proc, in使调用者进入就组合成四种并

发机制：

| | |
|-----------|-------------------|
| call_proc | (远程)过程调用 |
| call_in | 会合(本模块进程间和各模块进程间) |
| send_proc | 动态过程创建 |
| send_in | 简单异步消息传递 |

in语句的退化即receive原语。

- 值得注意的并发语言有：

| | |
|-------|--------------------|
| Ada | 会合同步机制 |
| Linda | 带元组空间的一组原语，可附于顺序语言 |
| Occam | 同步消息传递 |
| SR | 多原语并发机制 |

习题

14.1 试述条件临界区和信号灯的同异，为什么说CCR是对信号灯的进步？

答：信号灯和条件临界区都是基于变量共享的开发机制。信号灯是语句级的，直接作为程序设计语言层次偏低，程序员设计中稍有失误不是死锁就是死等。另外，同步并不一定能保证互斥，都用P、V操作实现，语义不直观，程序难查错。

条件临界区是在信号灯理论的基础上，基于变量共享的高层并发机制，它将共享变量显式地置于叫做资源的区域中。每个共享变量至多只能置于一个资源，且只能在条件临界区的region语句中访问。它易于显式控制同步，编译时即可查出不互斥，创建进程只看自己条件简单，易于正确性证明。因此CCR是对信号灯的进步。

14.2 按例14-2 表示法，试写出：

北京站一批旅客(m人)到达至出租车总站候车。有n辆出租车载人，设两种小车一为4人，一为8人必须满足才开车，设 $n*4(或8) > m$ 。编一模拟程序，描述需要几辆车把旅客载完。设一随机数表示旅客愿坐小车和大车的意向。

14.3 CCR在什么情况下保证同步不保证互斥？

答：在CCR中，由于资源r中的共享变量，只能在CCR语句中根据布尔表达式（条件）的取值访问，因而实施同步是显式且容易的，但互斥则是自然隐式地实现，要求是谁满足谁进，并使其它进程不能满足条件，而其它进程容易满足条件时，则不能保证互斥。

14.4 试述监控器的基本原理以及它和条件临界区的同异。

答：监控器把分散在整个程序中的region语句进一步集中成为一个模块叫监控器（monitor）对共享变量的加工限于监控器内定义的过程。监控器本身只是一个被动实体，仅当外部进程通过调用操作进入监控器，才能加工共享数据。每个进程都有描述它当前状态的描述子，因此在每个进程中要有显式控制进程同步的wait（等待）和signal（示信，向其它进程发出信号，本进程对共享变量的使用已经完毕）的原语，这样就控制了每个进入监控器内操作过程的进程。

条件临界区同监控器相比进程和共享变量耦合太紧，临暂区所写不利读，一多了就太散，难于修改。

14.5 试述监控器中wait和signal和P，V操作的同异。

14.6 试用图解方法说明贪睡的理发师(例14-6)例题中各进程交互关系。

14.7 何谓占先？它是人为设定的概念还是非有此约定不行、查有关操作系统，哪些有占先。

答：占先（preemptive）：当某进程要求使用某资源，资源控制器则为其分配资源，在使用期间原则上是不再分配给其它进程。如果在此期间有更高优先级的进程要使用资源，只要前一进程当时未处于运行态都要转而分配给后一高优先级进程，谓之占先。

它是人为设定的概念，并且必须有此约定。

操作系统中有占先的是Windows NT，UNIX，LINUX等。

14.8 试述各种语言实现wait 和sigal时微妙的语义差异。为什么说它们是语义等价的？设想一种算法以SC实现SX。

14.9 嵌套监控器中何谓开式调用何谓闭式调用？试评述其优缺点。

答：开式调用：是若有嵌套调用发生时上层互斥自动解开，待调用返回后上层监控器又重新闭合（获得）互斥。

闭式调用，每一时刻只有一个进程进入监控器，调用某个过程，称为闭式调用。

闭式调用共享变量是得到互斥保护的，但是在嵌套监控器中，易引起死锁。

开式调用的语义要复杂一些，它要求在每次嵌套调用时监控器不变式为真，且共享变量不得作为下层调用的引用参数，不易引起死锁。

14.10 何谓路径表达式？它的根本弱点是什么？

14.11 信道是逻辑概念还是物理实体。从实现的角度理解它应是什么。

答：信通是一种逻辑概念。信通可以看成是已发送但未接受的消息队，在异步通信中，信通如果声明为全局的，即并发程序所有进程均能向它发消息，则称邮箱。

如果信通只在接受进程中声明，则称该进程的输入端口。

14.12 同步通信和异步通信 最本质的差别是什么？

答：异步通信必须有显式信通，然后借助原语send和receive收发消息，执行send原语是从不等待的，若消息队中无消息，或进程调度从消息队中选择耗时最短的下一条消息时进程处于receive处等待。信道中必须有一个无消息队。

同步通信必须指明两进程的同步点，通过输出、输入语句，进程间直接发消息，传送send和reive两原语都可阻塞等待的原语，仅当两进程都达到同步点时才通信，因而不需要缓冲。

14.13 试述异步通信实现机制的并发核的组成各成分的作用。

14.14 为什么同步通信中早期语言禁止在卫式子句中有输出语句。如果有你设想如何实现。参照例14-13写出实现框架。

14.15 会合和远程过程调用的基本差别是什么？它们是同步还是异步通信？RPC能实现异步消息传递吗？

14.16 试在Unix操作系统上实现例14-15有界缓冲区，编出实际的程序运行之。

14.17 试用Ada实现例14-16客户服务器算法，在Ada环境上运行之。

14.18 为什么Ada的任务不是能分别编译的程序单元？

14.19 Ada的选择语句有几种形式，试比较和CSP中卫式通信的同异。

14.20 用Ada上机验证例14-17。模拟1000次检查有无死锁。什么情况下会产生死锁，验证之。

14.21 用图解方式表明例14-17的11个进程交互情况。

14.22 试述多原语并发语言的优缺点。

14.23 Linda的主要特点是什么？它的新颖思想对你有什么启发。

14.24 写一评论；并发高级语言展望，一定要包含：为什么至今难统一并发语言；是带并发包好还是扩充语言机制好；多原语前景。

开发高级语言，最早可溯源至PL/1和ALGOL-68。20多年来，随着各种开发机制的研究推出了数十种具有并发机制的高级程序设计语言，这么多的语言也有多种开发机制，直到目前，还没有一种开发程序设计语言能统治开发程序设计整个领域，其原因是硬件背景，应用领域和程序设计模型的多样性。

在并发高级语言的发展过程中，在并发程序语言未发育完全之前，各厂商为了支持并发程序，都在不同程度上提供了程序包和库例程，这些程序包和库例程都是特定机器系统中的库例程，没有移植性。程序员要非常了解本机的体系结构和OS，并做机器间的类型检查、这种方法不应成为并发高级语言的发展方向。扩充语言机制相对较好一些，可以用原语扩充顺序程序设计语言以产生并发程序设计的变体。

由于并发高级语言的发展中，进一步的等同进程通信要求更灵活，多原语进程交互成为当今研究热点，因为它可以实现人们多年积累的各种并发算法，适合于各种背景。如多原语语言Java已成为当今网络上编制并发程序的最重要语言。

第 15 章 平台无关语言

高级程序设计语言宗旨之一是平台无关,即用户会了某种语言到任何能实现该语言的环境上即可编制程序而不需其它知识,在该语言的某个环境上编制的程序拿到有该语言实现的另一个环境上可以照样运行,程序的计算,语义不变。传统上将其称之为可移植性(portability),前者程序员可移植、后者程序可移植。由于高级语言源程序最终总要在不同操作系统、不同指令系统、不同内码格式的具体机器上实现,可移植性始终不能完全彻底解决。例如两不同指令系统的目标码程序语义完全等价是不可能的,只能使之差异最小,一般情况下不影响计算逻辑,就算很好了。业界几十年来一直为此努力,到 80 年代初 Ada 已声称可以做到程序和程序员可移植。它的办法是:

[1] 语言定义是与机器无关的,凡与机器相关部份(如基元类型和输入/输出等)做成预定义程序包。保证 Ada 写的程序可移植,执行语义微小差别可由预定义包查出。

[2] 提出完整的可移植环境(APSE),分三层。第一层核环境(KAPSE)是操作系统的抽象,即应用程序接口集。接口是规范化(大体一致)的。第二层,最小 Ada 程序设计支持环境 MAPSE,保证 Ada 程序编译,连接、加载、运行,排错……程序动作基本一致。当然要包括支持程序包、函数库,也是可移植性的基本保证。第三层,支持程序开发和动作的各种工具,尽可能相同。其中最主要的一环是编译器要经过 ACVC 测试。

[3] 语言中向用户开放属性机制,使程序员能得知每种类型实现状态以及运行时的某些状态值。当有不一致时可采取补救措施。

90 年代网络计算兴起,对平台无关语言提出了更高要求。因为一个不断扩充的异质网不可能每个站点都提供庞大的可移植环境,更不能统一操作系统。经常是几个不同的机器计算一个应用(系统),即一处开发加载到另一处立即可用。联合开发的程序交互可用。可移植性上升为平台无关性。

1995 年出现的 Java 就是在这种需求之下开发出来的。本章讨论 Java 语言设计特点及其保证平台无关的机制。

除此而外,网络计算中还用到两类平台无关语言:置标语言(严格说不属于程序设计语言)和脚本语言(严格说不能做到平台无关)是网络客户必须掌握的语言,而且随着网络计算技术的成熟,它们的重要性会与日俱增,本章一并介绍。

15.1 平台无关语言的实现策略

网络协议保证网络上两站点间文件的传输。文件中的数据(即使有较复杂的数据结构)一般可以复原。如果文件中是程序,接受站点只能看作字节流。无法在本机上执行。除非机器同构操作系统相同。网络能做到的是把程序作为字符流数据传过去。

实现程序传输,并在任何站点上执行是平台无关语言的第一需求。现在的问题是传什么与怎样执行。

(1) 传输源程序

要求语言的定义完全是机器无关的，每一接受站点配备一翻译器以屏蔽本机的特殊性，并配备与机器相关的预定义程序包(类似 Ada)以支持程序连接和运行。

翻译器可以是编译器可以是解释器。编译器可以使目标代码效率高，但经本机优化其语义差异偏大。解释器则相反效率较低、语义差异狭小。对于小型语言可采用此种方案(如脚本语言)。Ada 不完全移植性证明此方案不可取。更重要的是传过去之后要编译—连接—加载才能运行，无法用于实时性较高的协作程序。

(2) 传中间代码

编译技术中中间代码是抽象机器码，是对实现世界的抽象，故平台无关。传中间代码是一个可行方案。编译时所需环境(词法分析、语法分析)在源程序所在机器上配备，接受站点无需，且不必传递。传出去的中间码文件，如果中间码标准化做的更好，并有同样的解释器(网络可做到这一点)平台无关性就基本保证。再一个优点是可以尽早发现源程序错误，增加传输有效性。解释器屏蔽本机特性，不同操作系统上的解释器的一致性平台无关性的根本保证。最初的 Java 即采用编译—中间代码—解释执行方案。

(3) 传中间代码再编译后在执行

解释执行效率是低的。如果接受到中间代码程序后对它进行编译生成本机目标代码并优化，此时可得到高质量目标码，优化的语义差异比没有统一的中间码程序作优化语义差异小。这样编译要付出时间。当程序较大或有多次执行的程序块时，它比解释执行效率高。当程序较小时得不偿失。

(4) 传中间代码编译——解释执行

为了弥补(3)的不足可对中间代码解释执行。若得知有重复执行的大块，则将该块编译，如循环块、多次调用的过程/函数块，作为目标码模块暂存，下次执行直接引用。新版 Java 即采用这种方案。

总之，站点间传某一机器的目标代码绝不可行；传源代码有实现上语义“微小差别”问题，当程序较小时可行；传中间码可使语义差别最小，分散处理使双方处理程序小巧，中间有一次代码校验(这也是网络传输必须的)质量较高。

解释执行处理程序小巧，效率较低，动态联接较易；编译执行代码质量高，但编译时耗用时间和资源；混合执行最理想，但又带来处理的安全和复杂性，并要配备两个处理程序。传中间码的混合执行要三种处理程序。

15.2 Java 语言

Java 是为家用电器而开发的。由于硬件芯片发展快速，为了市场竞争，家电商就必须频频更换这些芯片。软件要求可靠不希望多改(只是跑起来更快)。这就要求软件和芯片无关。开发者发现 C 和 C++ 是满足不了这种要求的，于是在 C++ 的基础上开发了 Java(1990–1993)。

时逢其惠，90 年代初网络计算大发展，1993 年 SUN 公司设计了 Web 浏览器 Hot Java 把它联到 HTML 文档浏览器中，Java 程序作为 applet 嵌入 HTML 文档，从而为 Web 主页提供交互式可执行的内容，使静态描述的页而活动起来了，引起了业界广泛兴趣。在广泛试用基础上 1995 年底发布 Java1.0, 1997 年发布了 Java1.1 及相应的 JDK1.1 (Java 开发工具箱), 1998

年推出 Java2.0 平台,至此,Java 相对成熟,成为网络计算中真正的平台无关语言。之所以叫平台是它提供了丰富的支持类库(Java 核心 API),方便 Java 程序开发。

Java 类似 C++,但比 C++更面向对象,更简洁,相同部分采用相同语法。不比 C++大,这使熟悉 C++的程序员看起来很面熟,易学易推广。为了支持网络应用,Java 配备了 Java.net 程序包,其中的 URL 类使打开网上文件和打开本机文件一样方便,Socket 类支持网络数据报文和流(stream_based)等低级连接;远程方法调用 RMI API 使向远程对象发消息如同本机一样。

Java 的浏览器是可以根据需要动态地下载网络上的资源,使其成为真正的分布式语言。

Java 的解释执行,省却了编译后的链接(Linking),从而支持快速建模。Java 沿袭 C++的强类型并除去了 C++、C 的不安全性,增强了运行时的检查,不支持指针,配备无用单元自动收集。故比 C++更安全、健壮。

Java 的语言(扩充)包 Java.Lang 提供线程 Thread 类支持线程编程。

Java 本身平台无关,Java 系统也具有有良好的可移植性。Java 的编译是用 Java 语言写的 Java 的解释器(即运行时系统)用 ANSI/C 编写。

Java 可以把一个类动态地链接到一个正在运行的系统。

15.2.1 Java 平台无关性实现

Java 采用传递中间代码,实现程序在各站点上移动。在每个站点设置浏览器,解释执行中间码程序。浏览器屏蔽了各机的特殊性。执行中间码比直接解释源代码快得多。前期的编译又可以查出许多错误,且能得到充分的支持(这对面向对象程序尤其重要)。所以它是编译——解释执行的,如同 Smalltalk。

解释器和运行时系统(run_time System)统称 Java 虚机,即如同在虚拟的机器上运行。典型操作系统的虚机网上都有,可以按本机操作系统下载。

Java 源程序实现计算的过程其示意图如下:

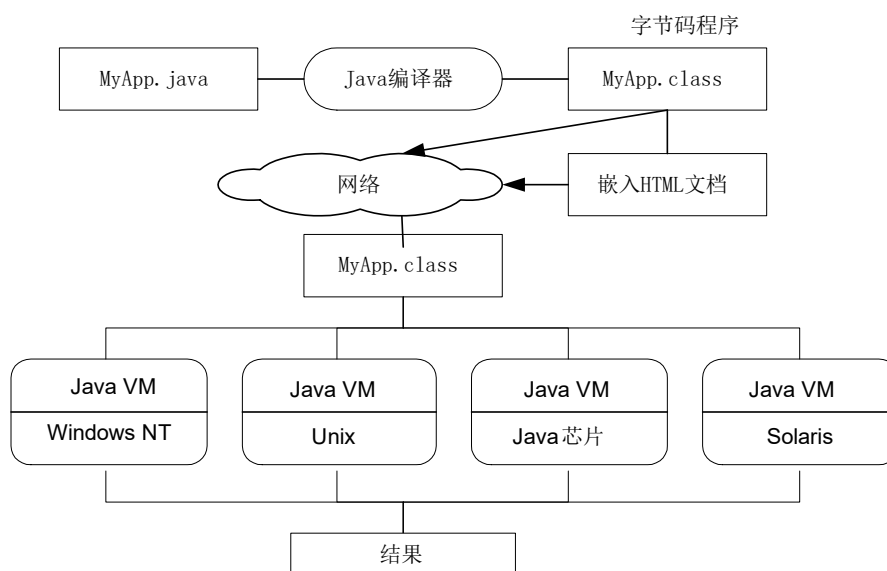


图 15-1 Java 源程序实现过程

15.2.2 Java 的字节代码

Java 选用 16 位双字节的 Unicode 作为中间代码。Unicode 原为 Apple 公司研制的 Machintosh 的多种语言文本的体系结构，以后为各大公司接受，1992 年正式成为美国行业标准“Unicode，全球字符编码”。

国际标准化协会 ISO 早在 1984 年开始了各国各地区使用的书面文字及符号统一编码。1993 年正式公布了 ISO 10646。我国等同的标准是 GB13000.1-93。ISO10646 规定了通用多八位编码字符集(UCS)。其中的 UCS-4 用四个八位位组表示一个字符。它的子集 ISO10646-1 是基本多文种平面(BMP)，为双八位的 UCS-2，其中 I 区 20992 个位置是中、日、韩汉字区。

Unicode 形式上和 UCS-2 是一样的，但体系不同。ASCII 和 Unicode 字符码相同，但只占它的低八位。为了使多八位编码字符与 ASCII 易于转换 ISO 10646 引入了通用转换格式 UTF-8，使得空位大量消除，以变长的字符流传输。

UTF-8 用 1~6 个八位位组表示 BMP 中的一个字符。从第一个八位位组左边数起有几个‘1’就表示后面接几个八位位组。以‘0’标志右边全是“有效位”。为了区分只有一个八位组，所有位组左边全多一个‘1’。编码格式如下：

| 有效位 | 最小数(H) | 最大数(H) | UTF-8 格式(B) |
|-----|----------|-----------|---|
| 7 | 00000000 | 0000007F | 0VVVVVVV |
| 11 | 00000080 | 000007FF | 110VVVVV 10VVVVVV |
| 16 | 00000800 | 0000FFFF | 1110VVVV 10VVVVVV 10VVVVVV |
| 21 | 00001000 | 001FFFFFF | 11110VVV 10VVVVVV 10VVVVVV 10VVVVVV |
| 26 | 00200000 | 03FFFFFFF | 111110VV 10VVVVVV 10VVVVVV 10VVVVVV 10VVVVVV |
| 31 | 04000000 | 7FFFFFFF | 1111110V 10VVVVVV 10VVVVVV 10VVVVVV 10VVVVVV 10VVVVVV |

Java 程序内码转换如下图：

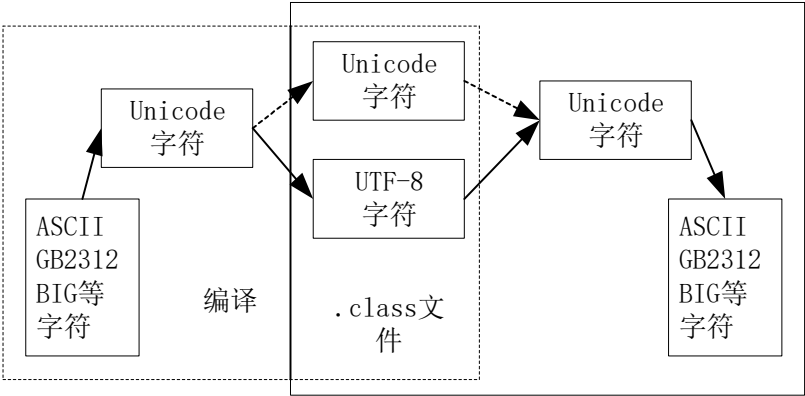


图 15-2 Java 程序由码格式转换图

图中 BIG 为台湾汉字标准，虚箭头是只传递不转换，实箭头转换。

15.2.3 以 C++为母语实现面向对象

Java 语言在变量声明、运算符、表达式、语句、参数传递、程序控制、面向对象机制的

表示法和概念与 C++ 基本一致。它取消了 C++ 某些不安全、依赖实现、对象不纯的机制：

(1) 取消指针，增加安全

指针是地址的抽象，如果一个程序到处移动其所到之处，存储格式显然不一样，指针变量运算各机解释不同。只能取消。然而程序员熟悉的指针引用计算则以引用变量代替，也就是保留‘常指针’，常指针保证不会把空指针传出去。没有指针变量，悬挂指针问题不会出现，增加了安全性。

(2) 取消头文件和预处理机制

在 C 和 C++ 之中头文件放了许多与实现相关的常量定义和宏、以及背景为单机时的全局变量，这在移动之后是难以实现的。取消头文件之后全局量以静态 static 量附于某文件。便利编程的库函数、类原型以包机制解决。以更方便的定止化 (final) 机制定义常量。

一般说来，预处理为编译提供信息，简化编译中的数据，为程序员提供扩展部份语言功能的方便，但它不像语言那么标准化。只为该机的编译器所识，取消后扩展部份功能以更正式的程序包代替。

(3) 取消 goto 语句。

C/C++ 本可用 break、continue 和异常来复盖编程时需要 goto 的全部理由，只是为了照顾历史才保留 goto。Java 由于没有历史包袱，故取消。

(4) 更加面向对象化

程序中只有类——对象，以及在其上层的包 (Package)。取消非面向对象的结构；全程序和 main() 函数。main() 作为方法放于类中，其执行句柄的功能照旧。

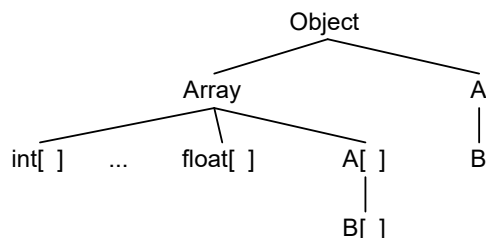
Java 的类机制和 C++ 相同，为了纯净只有单继承。把复杂的多继承放在处理复杂联结的包中以界面 (Interface) 实现。Java 类中的方法均缺省为虚函数，Java 的类均可动态加载。

Java 取消了 C++ 中的模板 (template)。模板是参数化 (即类属) 类型，是半静态的。面向对象支持完全的多态类型。运行之中动态束定，样板就没有必要了。

Java 取消了运算符重载 (operator overload) 这种优雅的多态机制。这似乎没什么道理，只是为了阅读清晰，方法重载、复盖 (overriding) 保留。

Java 取消了 struct。C++ 中把 class 作为带显式操作的类型处理。struct 类型是带隐含操作的 class。而 Java 中全是类和对象，数组、字符串都作为对象，复杂类型都是多种属性的类，struct 就冗余了，可以取消。

值得一提的是 Java 有一个虚基类 Array，它是 Object 的子类，所有基本类型和所有其它类的数组都是它的子类。如下图标，左半：



然而我们所有的类又都是 Object 的子类且有继承关系，如图的右部。那么 A[] 自动是 B[] 的父类。因此，数组 a[] 是 A[] 的实例，是 Object 实例：

```
Object 0;
```

```
int a[] = new int[10];
0 = a;      //数组可以赋给 Object 的对象反过来要遵守强制转换规则:
a = (int[ ])0;
```

综上所述, Java 是较 C++ 更纯粹的面向对象语言。除此之外, Java 增加以下特征和机制改进 C++ 使之更适宜作平台无关的网络语言:

(5) 引入界面机制

Java 中的界面是以 Interface 关键字代替 class 的类, 它只提供封装的方法协议(抽象的方法)没有方法体, 属性都是 static 常量。但它可以作为引用类型有其实例的界面对象。界面本身是多继承的。

任何一个类总是一个子类有其父类, 都可以去实现界面(中定义的方法)。设界面 INF 已声明:

```
class B exten A implements INF {
    继承 A 的属性
    可以声明自己的属性
    定义 INF 声明的方法 MI( ){
        方法体中可用 A, B 属性和 INF 的常量};
    定义 INF 声明的方法 M2( ){ 同 M1( )    };
    定义 INF 所有方法, 及父界面中所有方法;
}                                     //B 是 INF 的实现
class C exten D {                     //单继承, 通过界面 INF
    INF inf = new B( )                //引入类 B, 实现多继承
    ...
    C.method1( ){可用 B 中的属性和方法, C, D 的属性和 D 的方法}
    ...
    C.methodn( ){...};
}
```

同一界面可由多个类实现。执行时按所在的包(见下文)和引入的包找匹配, 动态绑定实现的类。

多继承的随意性导致类体系结构混乱, 父辈继承孙辈), Java 保持单继层清晰的树状态层次体系(便于静态存储管理)。利用界面动态实施多继承。

(6) 包与编译单元

C++ 的头文件提交分别编译时有关类继承、联接信息。一个文件中可以只有一个类, 也可以多个类。Java 取消头文件提供包 package 的概念。

Java 为每个类生成一个字节代码文件, 且文件名和类名同。因此同名类有可能发生冲突。增加包机制即为了管理类名空间, 消除冲突。包是一种程序结构, 由一组类和界面组成, 并为编译指示与其它包或类联系的信息(通过 import 语句)。提交编译的编译单元就是一个包, 当包语句不出现时, 系统按无名缺省包处理。

包对应为文件系统的目录。包名按目录路径组织。例如, package

Java.awt.image.mypicture 指定该包中的文件存储在目录 path/Java/awt/image/mypicture 之下。path 是由环境变量 CLASSPATH 确定的根目录。

有了包机制, 程序访问的可见性控制又扩大一级。不注明 public 的类, 只在本包内访问。类中成员访问控制与 C++ 相似。在有包的情况下总结于下:

| | 同一个类 | 同一个包 | 不同包的子类 | 不同包非子类 |
|-----------|------|------|--------|--------|
| private | ✓ | | | |
| default | ✓ | ✓ | | |
| protected | ✓ | ✓ | ✓ | |
| public | ✓ | ✓ | ✓ | ✓ |

Java 提供的开发平台, 即应用程序接口集, 就是以包的形式提供的。详见下一小节。

(7) 支持线程编程

当今从网上接收 CAI 授课, 一方面在后台接收画面, 前台播放视频, 同时播放声音, 这要求多个进(线)程并行地工作。线程是共享同一进程资源的轻进程, Java 支持多线程编程以改善图形用户界面的可交互性。

由于进(线)程的并行和调度最终由操作系统实施。在高级语言层次上只要编写若干例程利用操作系统命令和原语也可以实施并发线程, C 和 C++ 就是这样做的。这要求程序员熟悉操作系统, 按选定的并发机制锁定、释放例程。工作繁琐且易死锁。

Java 提供了语言内置的多线程机制, 从而大大地简化了多线程应用程序的开发。Java 在语言核心上并未提供线程制定、控制、调度等机制, 只有两个关键字 synchronized(同步化) threadsafe(线程安全)。前者指明某个对象或方法不能被并发执行, 后者指明某个变量当一个线程使用时其它线程不能用。所有线程控制原语 start(), sleep(), suspend(), wait(), run(), stop(), resume(), notify() 均以方法放在 Java.lang 包的 Thread 类中。也就是利用预定义的 Thread 类和 Runnable 界面扩充语言支持线程编程。这和预定义的 Java.swing 支持 GUI 编程是一样的。保持语言核心精巧。

Thread 提供的原语以共享变量的监控器(管理程)为模型。线程同步机制采用 CSP 机制, 由于 RMI API 屏蔽了对象的分布性、封闭的对象本来就是利于通信的。故不设 send, receive 原语, 以方法调用编制基于通信的并发程序。

(8) 增设的新关键字

final(定止)关键字指明类、变量、方法是最终的, 即类不能有派生子类; 变量值一经定义不再变(即常量的一种表示法); 方法当有重名时不被复盖。这主要是为了保证程序清晰和安全。因为一个应用都是最终类生成的对象相互发消息。如果某个最终类又有了子类, 子类实例对象也是父类实例对象, 极易产生错误。重要方法被复盖可能是黑客引入病毒途径。

finally 关键字出现在捕获异常的 try-catch-finally 语句块中, 只要进入 try 语句块, 不管是否捕获到异常(进入某个 catch 处理段), 都要执行 finally 块。

finalize(终止化)虽不是关键字, 它是 Java.lang, Object 类中的一个重要方法名。在对对象作无用单元收集前, Java 运行系统会自动调用 finalize() 方法释放系统资源(已打开的文件或 socket), 用户可在某个类中显式定义该方法以撤销该类对象所占资源。

用 **native**(本地)修饰的方法, 其方法体可由本地非 Java 语言实现(如 C, C++, Ada),

这是沟通 Java 与本地资源重要手段。

transient (瞬态) 当类的实例作持久对象处理时标注 **transient** 指明该变量非持久数据 (不入库)。

abstract (抽象) 修饰类时, 指明该类不能生成实例。修饰方法时无方法体, 为所有子类定义的一个统一接口。子类中再次定义该方法 (体)。

抽象类不一定要有抽象的方法, 但有抽象方法的类一定是抽象类。

abstract 不能修饰构造方法、静态方法、私有方法。不能初始化抽象类, 不能调用抽象方法, 不能复盖超类中的抽象方法。

instanceof 指明表达式的值是某个类/界面的实例, 表达式可以是简单变量名。

15.2.4 Java 预定义程序包

Java 采用小语言、多支持的构成思想, 由于它的实现小巧, 任何站点都可以下载它的编译器和解释器, 然而, 复杂的程序表达则以大量的类库支持。面向对象的继承机制正好满足这种设计。Java 以包 (package) 提供类库支持。Java 的包是一组类和接口的集合, 包利于名字空间管理, 以点表示法提供类和接口, 以便编程开发直接引用。一组包的集合就是 Java 编程支持环境。一个基本的编程环境提供以下包:

- **Java.lang** (语言包) 提供基本数据类型对象, 数字库对象, 支持线程编程和具有异常, 出错处理程序, 提供访问系统资源的类, 支持应用程序动作的类 (classloader, Compiler, Runtime), 提供动态调用和安全保护的基础。它是 Java 语言编程最必须的支持, 自动缺省链接。

- **Java.io** (输入/出包) 提供一整套支持输入/出的类, 即支持对字符或字节流的读/写操作。包括线程通过管道流的操作, 以及通过网络对远程流的操作。Java 的对象可以串联化作为字节流传输 (入流通过 **ObjectInput stream** 类, 解串通过 **Object Output Stream**)。

- **Java.Util** (工具包) 提供各种编程工具, 如 **Date**, **Dictionary**, **Hashtable**, **Stack**, **Vector**, 以及有关的异常类。

- **Java.net** (网络包) 为实现网络通信功能的各种类, 为 **URL**, **URLConnection**, **Socket** 等, 以及有关的异常类。

- **Java.awt** (抽象窗口工具包) 包括一套标准的图形用户界面 (GUI) 的元素, 如按钮、对话框、菜单、选项、容器、事件、格栅包、图形、图象、面板、正文域、窗口、标号、表、编排管理等, 以及相关异常。支持用户开发图形用户界面。Java 2.0 在它的上层提供 **Java.swing** 使 GUI 设计更方便。

- **Java.awt.image** (AWT 图形包)

是 **Java.awt** 包的子包, 提供高级图形处理的类和接口。

- **Java.awt.peer** (AWT 对等体包)

是 **Java.awt** 包的子包, 提供 AWT 构件与系统相关的窗口实现 (对等体)。

- **Java.applet** (Applet 包) 提供编写小应用程序 **Applet** 的各种类与接口。

- **sun.tools.debug** (调试工具包) 提供调试 (特别是远程虚拟机上运行各类) 的各种类与接口。

以上各色中的类约计 235 个。此外支持基于数据应用有 JDBC API；支持构件开发有 Java Beans 构件系统，该系统使用户定制 Bean 构件通过 CORBA 和 COM 连接排 Java 构件；支持复杂媒体应用的有 Java.media；支持电子商务应用的有 Javax.commerce，这些包和机制构成 Java 平台，也有约 300 个类。这些包和机制大大地方便了应用开发，它们相对独立，可按需克隆下载，这也是网络利用资源的优势。

15.2.5 一个 Java 的简单实例

例 15-1 一个小应用程序，支持在屏幕上画图。

```
import Java.applet *;
import Java.awt *;

public class Scribble extends Applet {
    private int last_x, last_y;           //鼠标当前的座标
    private color current_color = color.black; //当前颜色
    private Button clear_button;          //“清除”按钮
    private choice color_choice;          //颜色下拉列表
    //小应用程序没有 main() 方法
    public void init() {
        this setBackground (color.white); //屏幕初始化设置屏幕背景色
        clear_button = new Button ("clear"); //创建按钮
        clear_buffon.setForeground (color.black);
        clear_buffon.setBackground (color.lightGray);
        this.add(clear_buffon);
        //创建菜单颜色，并把它加到小应用程序中
        color_choices=new choice();
        color_choices.additem("black");
        color_choices.additem("red");
        color_choices.additem("yellow");
        color_choices.additem("green");
        color_choices.setForeground(color, black);
        color_choices.setBackground(color, light Gay);
        this.add(new Label ("color: ")); //加一标号
        this.add(color_choices);
    }

    // 当用点击鼠标开始画图时，调用以下方法
    public boolean mouseDown (Event e, int x, int y)
    {
        last_x = x; last_y = y;
```

```

return true;
}

//当用户拖动鼠标时，调用以下方法
public boolean mouseDrag (Event e, int x, int y)
{
    Graphics g = this.getGraphics( );
    g.setColor(current_color);
    g.drawLine(last_x, last_y, x, y);
    last_x = x;
    last_y = y;
    return true;
}

//当点击鼠标选择颜色时，调用以下方法
public boolean action (Event event, Object arg)
{
    // 若点击 Clear 按钮，作以下操作
    if (event.target == clear_button){
        Graphics g = this.getGraphics( );
        Rectangle r = this.bounds( );
        g.setcolor (this.getBackground( ));
        g.fillRect(r.x, r.y, r.width, r.height);
        return true;
    }

    //若选择颜色，作以下操作
    else if (event.target == color_chioces){
        if (arg.equals ("black")) current_color=color.black;
        else if (arg.equals ("red"))
            current_color = color.red;
        else if (arg.equals ("yellow"))
            current_color = color.yellow;
        else if (arg.equals ("green"))
            current_color = color.green;
        return true;
    }

    // 否则，交由超类处理
    else return super.action (event, arg);
}
}

```

这个程序只是展示 Java 与 C++的相似性。其中 color, Button, Choice, Label, Graphic,

Event 类继承自 Java.awt.*。color_choices 是实例对象，color_choices.addItem() 是向它发消息。this 是 Scribble 的实例的代名词，super 是其超类实例的代名词。

15.3 Java 虚拟机技术

虚拟机技术作为一门综合技术几乎体现了当代程序设计语言设计与实现技术的各个方面。Java 虚拟机要实现 Java 的面向对象程序：类装入及实例对象生成，对象交互(引用方法)；获得核心类库支持；线程运行；处理异常；作无用单元自动收集。虚拟机执行过程如下：

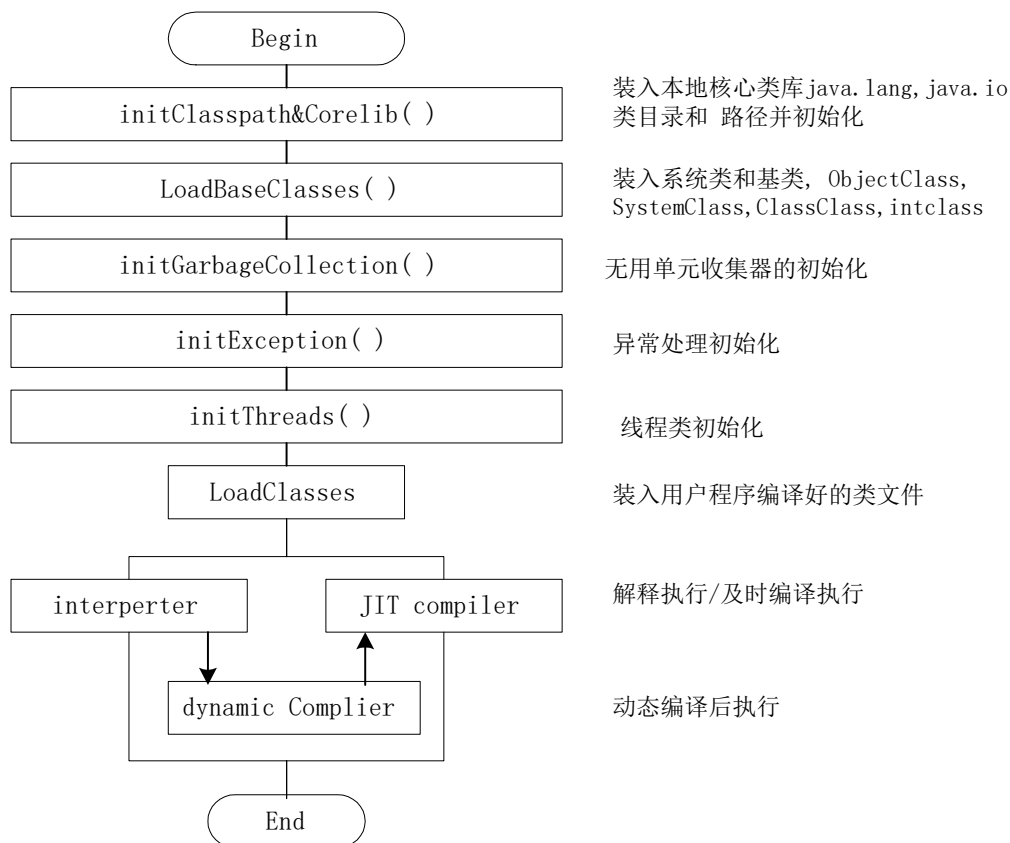


图 15-3 Java 虚拟机执行过程

下面我们逐一介绍虚拟机中与 Java 语言最密切相关部分。

15.3.1 类的加载

Java 是 00 语言，它的每一个类对应为一字节代码文件。字节码保留了源文件的信息：类名、域名、方法名，并反映了封装及继承的父类信息，以及创建实例对象操作的信息。因而反汇编成源文件十分容易。

(1) 类文件格式及存储结构

类文件包括以下四部份

[1] 常数池(constant pool)相当于传统编译的符号表，包括类名、方法名、域(即属性)名、

所属类名、字符或数值常量、标记(tag)和数据值分别存放。

[2] 方法表 存放各方法有关信息,包括访问标识、方法名、方法属性:代码属性和异常属性。代码属性存放方法字节码、方法运行堆栈、局部变量空间的大小。异常属性表中包括各异常类型、处理段的入口(自定义的、预定义的)。Java 虚机根据该属性转而执行相应处理段。行号表记录了方法的字节码和源文件对应的关系。

[3] 域表 记录类变量和对象变量的类型、空间大小、和对表头偏移量。

[4] 类表 以上三种表均由类表索引。类表中还有静态数据区存放类变量。对象域字节数记录该类创建对象的大小。方法索引表中存放本类方法表指针及父类方法表指针。当有对象引用本类方法时可在方法索引表中快速查询。并限定实例对象不可引用类方法,但可引用父类对象方法等,以及体现访问控制(公有、私有、保护)信息的约束。访问控制标识、父类标识、域的个数、常数池个数等信息存放在类表的其它属性域中。

虚拟机为方便对类的查询,将类存放在一 Hash 表中,Hash 值相同的类存放在由类的 next 域建立的类链表中。类表中 Class*为该类的 Hash 值。

类文件的格式与类在内存中的存储构基本一致如图 15-4。Java 中的界面与类有相同的存储结构。因为它只定义方法型构(或称原型)没有方法体,故存储结构中设有方法表和方法索引表。

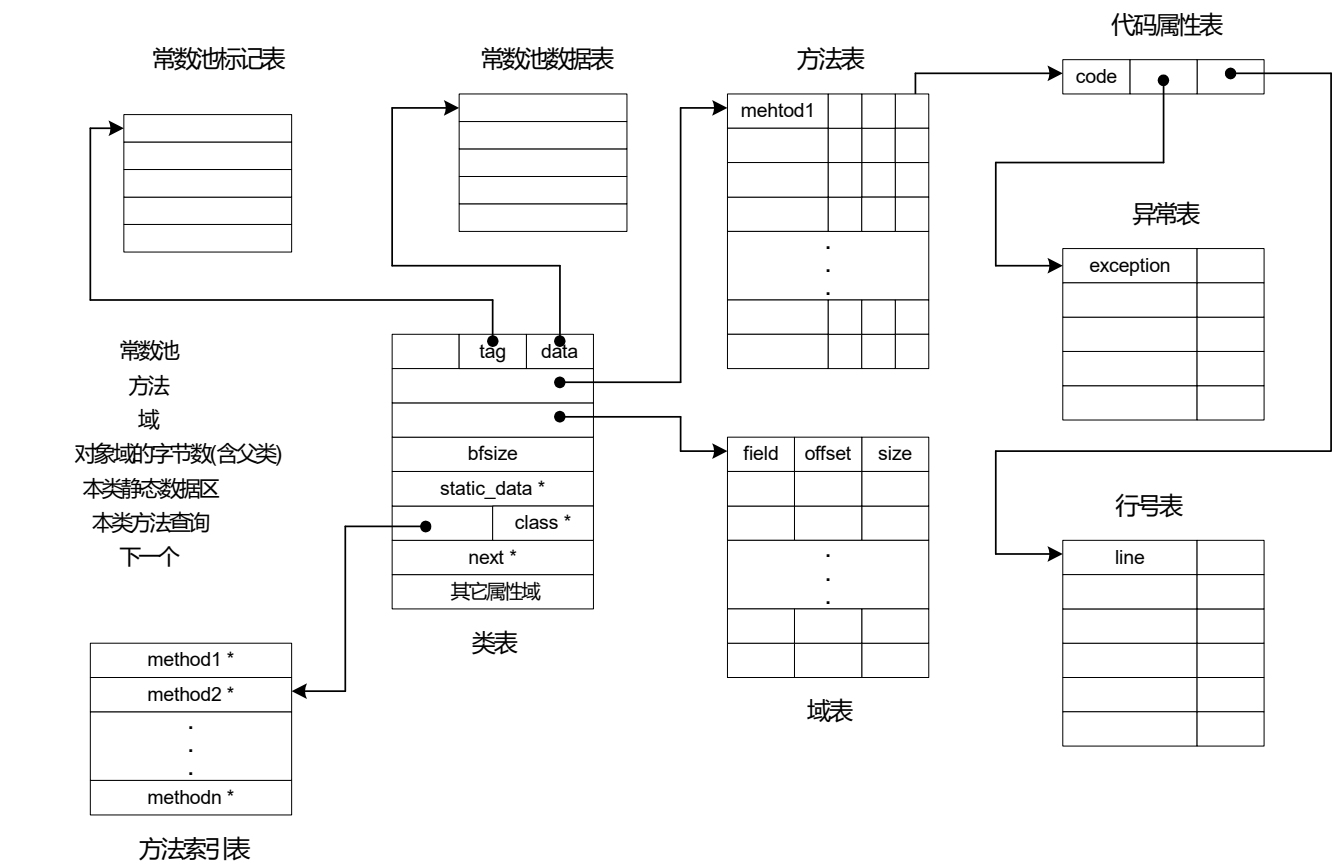


图 15-4 类在内存中的存储结构

(2) 在虚机内建立类

Java 的类文件传入后,由类装入器(classloader)完成装入、链接、初始化的工作。若有父类先链接父类。链接工作分三步:

校验(Verification)确保类格式文件结构的正确性。每条指令码的正确性。

制备(preparation)创建常数池的两个表。创建类/界面的域(Field)表。分配类的静态(类变量)的域。创建方法表和方法索引表,处理类方法重载,即修改方法索引表指针使其指向被重载的子类方法。

解析(Resolution)常数池存放各种类型的常量以及该类对其它类域和类方法的引用。在创建常数池的数据表时,这些引用是以全名(Full qualified name)形式,即符号引用。当解释器执行方法代码时用的是符号引用,若该符号是对类的引用则将被引用的类加载到内存,并将符号引用替换为该类在内存中地址引用,同时在标记 tag 表中标明“已被解析”。常数池的解析包括类/界面名、它的域和方法、字符串的解析。

解析过程可在装入后,也可以在运行时首次使用时解析。只有经过解析,装入的类才是可执行的。

15.3.2 Java 虚机的体系结构

Java 虚拟机从平台无关概念出发,建立独立的堆栈机器,它假定有以下公共设备:

- 各种寄存器
- Java 堆栈
- 对象堆
- 方法区

其体系结构如图 15-5 所示:

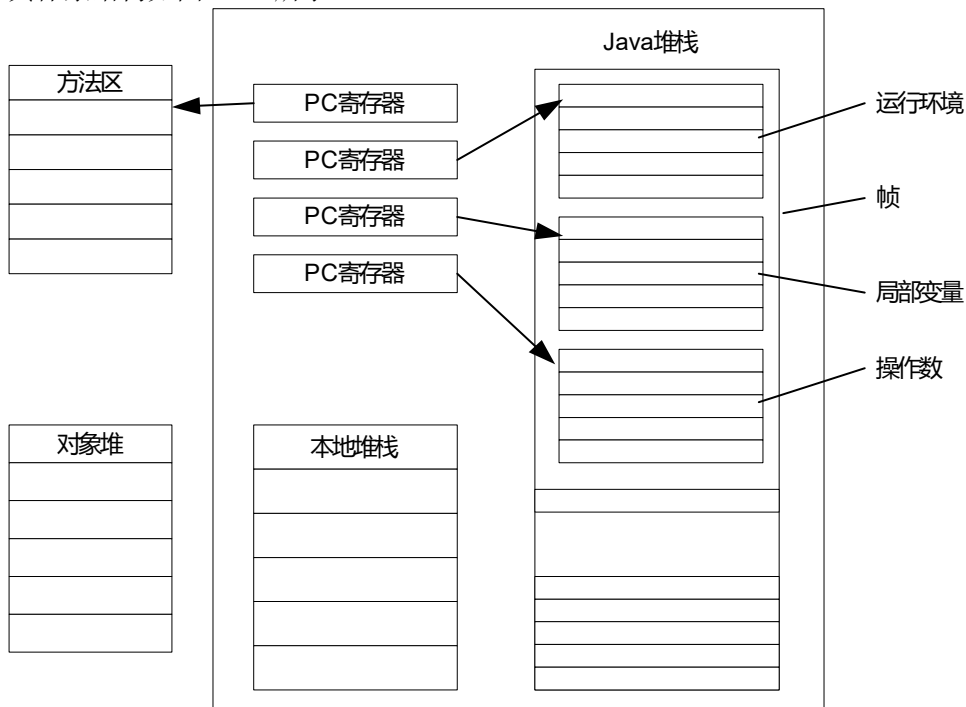


图 15-5 Java 虚拟机线程运行的体系结构

JVM 为每个 Java 线程分配一个 Java 栈和一个本地方法栈。所有的线程共享一个方法区

和动态生成的实例对象(只有数据)区, 初始化阶段 JVM 将与此线程相关的方法装入方法区, 以 PC 寄存器引用。每当某一方法被调用, 该方法作为一帧(Frame)压入栈。帧分三部分: 方法的局部变量放入局部变量区, 由 Var 寄存器引用; 方法的动态链、返回地址、异常传播信息压入运行环境区, 由 Frame 寄存器引用; 方法指令的字节码, 解析后放入操作数区, 由 Optop 寄存器引用。

压入后随后弹出逐一执行, 方法结束后, 返回调用该方法的帧, 若有新的方法调用再压入新帧。若新帧在执行中又调用方法, 则在新帧上方再压入一帧……。

15.3.3 虚拟机技术进展

由于 Java 解释执行, 其效率约为 C++ 的 1/2-1/6。越是安全措施多执行效率越低。因此, 一直在为提高效率而改进虚拟机。从 1996 Java1.0 正式发布以来虚拟机的实现机制已更迭了三次:

第一代 JVM 直接解释执行字节代码文件, 其工作管理示意图为图 15-6。

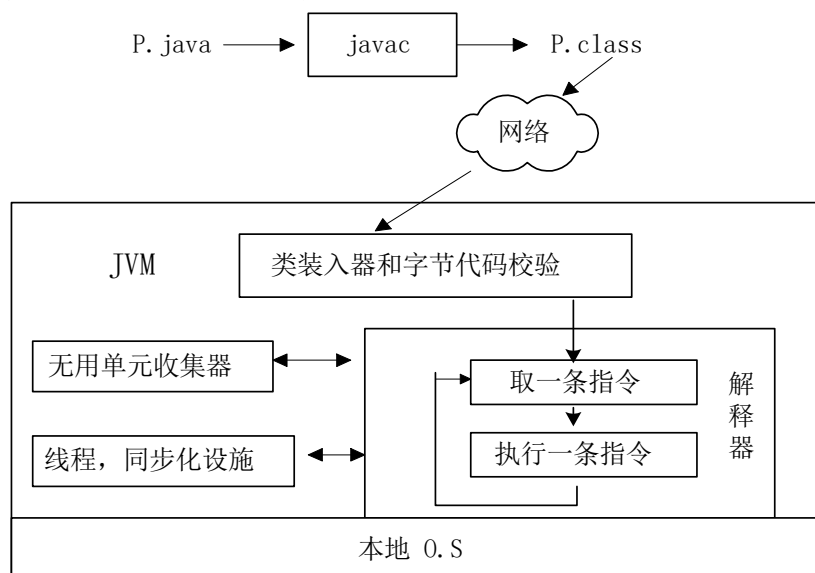


图 15-6 直接解释字节代码的 JVM

类装入器、字节代码校验器、无用单元收集器、线程同步化设施统称运行时支持。这种 JVM 主要追求正确、可靠、平台无关。但其效率仅为 C 的 1/20。于是 SUN 公司着手效率, 自然首先想到编译, 即字节代码文件传入后, 及时(Just in time)编译为本机的目标代码, 经优化后执行。生成的本机目标代码只放在内存, 不另作文件保留, 即使同一程序下次运行重新编译。其工作原理示意图为图 15-7:

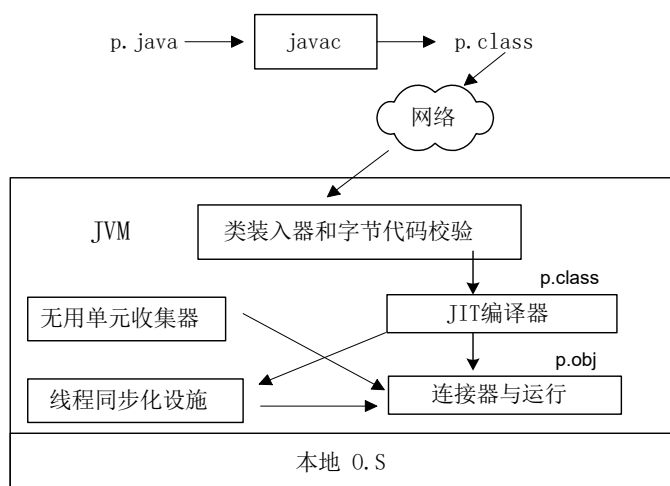


图 15-7 及时编译的 JVM

这种及时编译方案的 JVM 也叫第二代 JVM。它的执行效率大为提高,约为解释执行的 6-10 倍。Java 程序的动态特性和安全性均未受到影响。移植性稍有影响但不甚重要。无用单元收集器效果差。两种代码同时存在内存翻倍。在内存廉价增容下, 这个问题不算致命。

这种方案对反复执行的代码(如循环调用)特别有效。一次执行的代码有时不太有效,甚至花在编译、优化的时间比起直接解释执行时间还长。所以,有的系统把 JIT 编译作为选项,根据解题需要选用。

第三代 JVM 采用近年发展出的动态编译技术。软件工程界早就发现 80-20 规律,即 80% 代码只占 20% 运行时间,而 20% 的代码占用 80% 的运行时间。如果能用某种办法(甚至人工智能技术)找出这 20% 的代码,只编译优化它。对于 Java 只对这 20% 代码作及时编译,编译后存放在文件中,每次解释执行到此处,连上优化后的代码,这样,执行效率大为提高。其它对性能影响如同 JIT 编译。只是无用单元收集的损害要更少些。

SUN 公司 99 年春天发布的 JDK2.0 其中 Hot Spot (热点) 性能引擎即采用这种动态编译技术。

采用动态编译技术的 JVM, 其原理示意图为图 15-8。

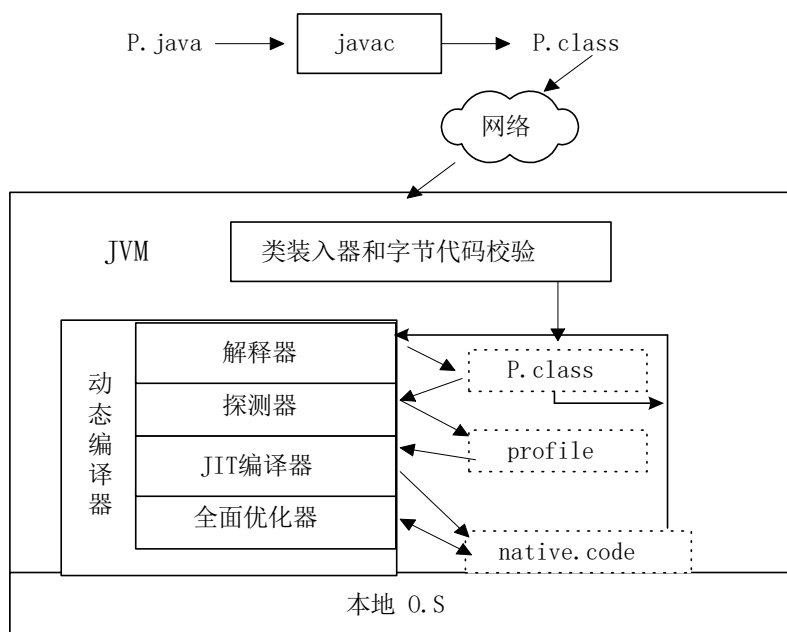


图 15-8 动态编译器的 JVM

动态编译器包括字节代码解释器(含无用单元收集器和线程同步设施), 及时编译器、探测器和全面优化器。当字节代码程序装入并校验后, 首先解释执行。执行中探测器记录执行时的信息, 如方法被调用次数和执行时间。当得知某个方法要多次执行, 进入循环, 或某段代码解释执行时间超过预定的阈值, 即找到了热点(Hot Spot), 停下解释器, 返回到热点起始处, 将整个热点代码作为一单元, 提取放入 profile 文件。调用及时编译器和全面优化器对 profile 文件作及时编译并优化。执行优化后的本地代码并保存入文件。再接着恢复执行以下字节代码程序, 发现下一个热点, 做同样的及时编译和优化, 直到这段程序执行完毕。

下次再装入这个类的字节代码程序时, 若发现是已释义(parsed), 解释执行到热点, 直接取出优化后的本地代码执行……

Hotspot 性能引擎目前尚未采用人工智能技术。由于当今程序的风格是方法小、个数多, 以方法调用、被调用作为探测器分析的重点。此外, 调用的内务开销也是应优化的热点。Hotspot 采用自动内联(in_line)技术把被调用方法嵌入到调用方法中, 省去了运行时数据传递和匹配检查的时间(这也是 Java 取消 C++的内联语句的原因)。

HotSpot 性能引擎的全面优化器集当今优化技术之大成。可作死代码删除、循环变量提升、冗余子表达式删除、常量传播等。还有针对 Java 特点的优化技术: 空检查、值域检查和删除等等。

热点代码体积越大, 优化的余地越大。优化后的代码成了面目全非的快速“黑箱”, 不宜于动态插装新的 Java 代码, 减低了 Java 的动态性能。于是又发展了动态逆优化技术。也就是记录下热点优化前的插入点和主要特征。

15.4 置标语言

网络计算的发展导致电子文档的发展, “无纸办公” 实则是由电子文档代替维系现代社

会人类通信、记录、交往活动的一切介质文档。这是一个极其巨大的领域、据悉，美国在 90 年代初一年社会上产生、运作的文档为 920 亿个。当今软件开发 2/3 的工作量是开发的是各种文档(程序也是文档)。“无纸”电子文档要能表达各行各业需要的各种文字，各种字体格式，各种文体形式(诗、散文、报告、经书法律条文、字典)，远远超出基于正文文件的计算机网络节点间信息传递的需要。多媒体技术出现之后，声、图、文并茂的电子文档成为桌面、工作站出版技术一个分枝。

电子文档突飞猛进的发展迫切需要标准化和规范化，围绕以下三个方面：

- 各国各种信息是否都能表示？为此制定正文(字符集)、图、声、象的各种标准。其中以正文，声音最为复杂。
- 每种信息的语义特征和内在联系能否表示？即信息的逻辑结构。为此制定结构文档标准。
- 电子文档如何表示(显现)为各种媒体？为此制定各种格式、字体和彩色、绘制标准。

上节提到的 ISO/IEC 10646 就是国际标准化组织历九年时间(1984-1993)制定的“通用多八位字符集”(UCS)。虽然一个字符占用四个八位耗用较多空间，但它一劳永逸地解决了多文种的各种表示，且后面一个八位与 ASCII(已通用于计算机内码)兼容。理论上和发展上 UCS-4 是站得住脚的，可工业界希望少占空间。于是 ISO 又推出 ISO 10646. 1 BMP(基本多文种平面)字符集。它是 ISO 10646 的子集，也叫 UCS-2。每个字符只有两字节，可以表示 $256 \times 256 = 65536$ 个符号。UCS-2 与 Java 的字节代码 Unicode 形式一致。

有关字符集还有 ISO 8859 分别为各国地区语言是定义了 15 个拉丁族语言子集。

第二类制定各种置标语言标准，用以描述文档语义内容和结构，本章介绍的 SGML，XML，HTML 即为此列。

第三类仅就正文文档有 DSSSL 文档样式语义和规格说明语言(ISO 10179: 1996)和 CSS 层叠样式表等。

第一类是基础，第二、三类是相互关联的，有的置标语言包含两方面内容。

15.4.1 从过程式置标到描述式置标

网络传递的信息是字节流文件，它虽然可区分正文文件中的记录，但只能按本机的理解恢复“原样”。为了使发送者和接收者看到原样就在电子文档内部作标记(tag)，指明题头、段、节、句；大小写、文种(英汉混合文档)；字体、大小号；何处插图、编排、出声……

置标语言就是定义标记及其使用规则的语言。它不属于程序设计语言，因为它不能按任意计算模型，设计算法进行计算，仅仅是描述文档，围绕文档的增、删、改、并的“计算”。

如何将电子文档变为电子出版物，80 年代早已有许多软件系统采用标记，如 TeX，PostScript 等。它们采用过程式置标，即告诉计算机为何一步一步地生成合乎预想的文档。如“空 7 格，#3 字、正体、打印一行；换行；空 2 格，#5 字，打印一行，跳过一行……”。这样生产出的文档只能是一种固定格式，固定介质且能有某种软件可以实现。更重要的是其中内容无语义性质(只是行)，程序员记不住，查不出则无法删改。

然而当今软硬件更改特别快，业务变更特别频繁，因此大量信息往往需要重排格式，重

排数据就要修改实现过程置标的软件，工作量有时占文档制备的 15%-50%。因此，必需改为不受实现过程影响的置标方式。

描述性置标不管过程，只按文档最后形式描述文档的结构。标记是“标题”、“第一段”、“第二节”、“空一行”……最后的显现各机可用不同的软件、不同的过程实现。描述只关注数据的内在相互关系。

从过程性置标到描述性置标是一个很大的进步。文档的拥有者和使用者可以把精力集中于文档数据内部组织上，最后显现可设计自动生成工具。这对于数据共享性，长久性，可移植性、重用、修改、集成都是十分有利的。

我们介绍的三种置标语言均为描述性置标。

15.4.2 SGML

标准通用置标语言 SGML 是 ISO 组织 1984 年研制 1993 年发布为 ISO 8879 标准。SGML 实质上是描述如何置标，置什么标的元 (Meta) 语言。

SGML 把文档看作是不同类型的对象集合是实例对象组成的结构。一段正文被置标之后即为一文档实例。

(1) SGML 文档结构

SGML 的文档结构由有类型的元素 (实例对象) 组成、元素是由命名的起始标记 <tag_Name> 和结束标记 </tag_name> 括着的正文单元。正文单元也叫内容。内容可以为空，可以是简单的一段正文，也可以内嵌另一种类型的元素。标记的名字并不涉及内容的语义，只用于区分其它元素名且指示相互关系。

内容嵌入其它元素，还可以再次嵌入另一类元素……这样就形成复杂的 (既可并行，又可嵌套) 结构，不同文档结构以内容模型区分。请看以下文档实例：

例 15-9 SGML 的文档实例

这是一个文档实例，就元素 <anthology> 而言，它给出了内容模型：

```
<anthology>
<poem><title> THE SICK ROSE </title>
  <stanza>
    <line> O Rose thou art sick. </line>
    <line> The invisible worm, </line>
    <line> That flies in the night </line>
    <line> In the howling storm: </line>
  </stanza>
  <stanza>
    <line> Has found out thy bed </line>
    <line> Of crimson joy: </line>
    <line> Does thy life destroy </line>
  </stanza>
```

```
</poem>
```

```
<!--more poems go here--> //SGML 的注释 </anthology>
```

<anthology>是一个元素，它嵌套多个<poem>(诗)元素。每首诗都有题目<title>，按着是若干诗段<stanza>，诗段内是四行<line>诗句。它们也都是成对标记的。

这个文档实例为什么排成这个样子而不是别的，例如，标题是否可出现在<line>中？<line>能不能出现在<stanza>之间？结束标记是否多余？也就是说，它要显式给出规则来定义它。

(2) 定义 SGML 文档结构

在创建 SGML 文档实例时，首先要给出形式规格说明以指明该文档结构，即文档类型定义，简称 DTD。DTD 是对正文的解释。为了分析可以从不同角度定义 DTD。

例 15-9 中出现的各元素其形式声明如：

```
<!ELEMENT anthology -- (poemt)>
<!ELEMENT poem -- (title? stanza+)>
<!ELEMENT title - 0 (#PCDATA)>
<!ELEMENT stanza - 0 (linet)>
<!ELEMENT Line 0 0 (#PCDATA)>
```

其中‘<!’是终结符，大写 ELEMENT 是关键字指出它是元素声明。后面三部分：第一部分是元素名，第二部分是最小规则，第三部分是内容模型。

最小规则指明有无起始/结束标记，中横线‘-’表示有，‘0’表示无。按此例说明例 15-9 中</ltitle>，</lstanza>，</line>，</line>可以不要。

内容模型由其它元素或带#号的保留字指明。此处 PCDATA 是“可释义的字符数据”的简写，意即任意字符。元素名后跟的字符是出现指示符：‘+’出现一次或多次，‘?’号是 0 次至多 1 次，‘*’号是 0 次到多次。

此外，还有成组连接符‘，’，‘|’，‘&’。意思是‘后跟’，‘或者’，‘调换乘后’。‘|’可为 SGML 提供成组模型概念，例如，多个元素有同样内容：

```
<!ELEMENT (Line | line1 | line2) 0 0 (#PCDATA)>
```

同一元素可以有不同结构：

```
<!ELEMENT poem - 0 (title?, (stanza | couplet | line)+)>
```

poem 元素的内容结构是，可能有一个题名，后跟 stanza 或 couplet 或 line 的多次重复，每次重复是三者重新任选。

这样定义的元素隐含着一个文档是一颗树形结构：诗集由若干首诗组成；诗由题名连同若干诗段组成；诗段由行组成；行由#PCDATA 组成。行中不能有诗集、诗。这是一棵层次分明的正常树。

然而文档中常需要某个元素出现在任意层次的其它元素旁，例如标注 Annotation)。这样，SGML 以异常(exception)放松对它的出现限制。包容(inclusion)异常可以把声明的元素加到文档结构的任何层次上。例如，我们先声明：

```
<!ELEMENT (note | variant)--#(PCDATA)>
```

若作以下声明：

```
<!ELEMENT poem-0(title?, (stanza+ | couplet+ | line+))+ (note | variant)>
```

最后一个‘+’即把 (note | variant) 包容到 poem 中, 即前面括号内的任一元素中, 包括 title。

如果我们因某种理由不希望 (note | variant) 出现在<title>中, 其它三元素任插。就可以用排斥(exclusion)异常:

```
<!ELEMENT title - 0 (#PCDATA)-(note | variaut)>其中的 ‘-’ 号就把<title>中的
(note | variant)排除了。
```

这样, 文档类型定义可如下式:

```
<! DOCTYPE anthology[                               //可与最上(根)元素同名
<! ELEMENT anthology - - (poem+)>
<! ELEMENT poem - - (title? Stanza+)>
<! ELEMENT stanza - 0 (line+)>
<! ELEMENT (title | line) - 0 (#PCDATA) >
]>
```

关键字 DOCTYPE 指明 anthology 是一个文档类型其结构定义在 [] 之内。

(3) 并发结构

同样一个文档结构, 可从另一观点定义它, 设按页的诗集<p. anth>:

```
<!DOCTYPE p. anth [
<!ELEMENT p. anth - - (page+) >
<!ELEMENT page - - ((title?, line+)+)>
<!ELEMENT (title | line ) - 0 (#PCDATA) >
]>
```

因为它最低层行题名和行都是一样的(#PCDATA), 可以写到一起:

```
<(anthology) anthology>    //A 类型的 A
<(p. anth) p. anth>        //B 类型的 B
<(p. anth) page>           //按 B 类型的 DTD 定义 page 元素
<!-- other titles and lines on this page here -->
    <(anthology)poem><title> THE SICK ROSE
    <(anthology)stanza>
    <line> O Rose thou art sick.
    <line> The invisible worm.
</(p. anth) page>          //page 结束
<(p. anth) page>          //另起一页
    <line> That flies in the night
    <line> In the howling storm:
    <(anthology) stanza>    //按 A 类型的 DTD 定义 stanza 元素
    <line> Has found out thy hed
    <line> Of crimson joy:
    <line> And his dark crecret loe
```

```

    <line> Does tyh life destroy.
  </(anthology) poem>
<!-- rest of material on this page here -->
</(p. anth) page>           //B 类型的 page 结束
</(p. anth) p. anth>        //B 类型的 B 结束
</(anthology) anthology>    //A 类型的 A 结束

```

这样定义的电子文档，当处理器只识别其中一个类型(有其定义)，则自动跳过另一个类的标记。若二者都识别则由人指定其一。

(4) 属性

SGML 的每个元素都可以定义属性或属性表。属性并不影响元素的内容，只是为显示，处理提供方便。在文档实例中要给出属性值，如：

```
<poem id = PI status = "draft" >...</poem>
```

<poem>元素定义了两个属性一为标识 id 一为状态 status，它们均在起始标记之中，只看有无‘属性名=属性值’。一首诗是草稿(draft)则可继续加工为可出版的版本。此属性说明重要程度)。标识属性是给该元素以唯一的名字以便在该文档内交差引用。属性定义在声明中完成：

```

<!ATTLIST poem
      id      ID              #IMPLIED
      status (draft | revised | published) draft >

```

属性表声明从关键字!ATTLIST 开始，在元素名之下列出属性名，属性值，缺省值三部分，中间以空白隔开，其符号意义如同元素声明中的约定。上面 status 属性可取值‘草稿’，‘校订稿’，‘可出版稿’，而缺省值为‘草稿’。请注意，当出现在实例元素标记中应加引号对。ID 是系统识别的保留字，意即唯一名字集合，其它系统保留字是：

CDATA 该属性可取任何合法字符数据。标记名也可以出现在其中但不作标记处理。

IDREF 该属性取指向某其它元素指针值。

NMTOKEN 该属性取名字记号，即字符串。

NUMBER 该属性取数字值。

大写的保留字也是 SGML 中的类型，在缺省部份中有以下系统保留字：

#REQUIRED 实例元素出现时此处必须指定值。

#IMPLIED 实例元素出现时此处可不提供值。

#CURRENT 实例元素出现时此处可不提供值，其值同本文档中第一个同名元素已取得的值。

重写上例：

```

<! ATTLIST poem
      id      ID              #IMPLIED
      status (draft | revised | published) #CURRENT >

```

当 poem 以简单标记<poem>出现在<anthology>之中，可以看作以前出现过的 poem 有相同的 status 的值。如果#CURRENT 换成#REQUIRED，则释义器报告出错，因为它要求括号中三值

以外的值。

为了支持元素交叉，即跨元素引用，SGML 提供特定标识符/标号机制，但必须在 DTD 中声明。如同上例，在 poem 的 ATTLIST 中，声明 id 是 ID 类型(并非必须)，声明了即可给以特定的标识符：

```
<POEM ID=Rose>
    此段 poem 正文具有标识符 ‘ROSE’
</POEM>
<POEM ID=P40>
    此段 poem 正文具有标识符 ‘P40’
</POEM>
<POEM>此段 poem 没有标识符</POEM>
```

下一步定义一个交叉引用的元素。该元素实际是指针，一般没有结束标记：

```
<!ELEMENT poemref - 0 EMPTY>
<!ATTLIST poemref target IDREF # REQUIRED>
```

它的属性名为 target，其值是 IDREF(系统提供的关键字，跨文档引用指针类型)，必须提供值。

有了这个声明可在正文中随处引用，如：

```
Blake's poem ou the sick rose <POEMREF TARGET=Rose>
```

尖括号中引用的即上述声明中的那段诗。

(5) 实体

为了跨文档间的可移植性，SGML 提供了实体(Entity)概念。实体是置标文档的某一部分，可大可小，小到一字符串，大到整个正文文件。如有声明：

```
<!ENTITY tei “Text Encoding Initiative” >
<!ENTITY ChapTwo SYSTEM “sgmlmkup.txt” >
```

第一句声明了名为 tei 的实体，也叫内部实体，其值(即内容)是引号中的那段正文，故也称正文实体。第二个声明的实体是 ChapTwo，指明它是 SYSTEM 实体，本文档以外的外部实体，其值是名为 sgmlmkup.txt 的正文文件的全部内容。若某文档一正文段是：

“The work of the &tei has only just begun”

‘&’ 为引用符号，SGML 提供串置换机制，本段等效于：

“The work of the Text Encoding Initiative has only just begun”

外部实体，可以是外部正文文件，也可以是图形文件，此时叫外部文件实体和图形实体，均以指针实现。

除了实体引用之外，SGML 还提供字符引用机制，以便处理不能直接打印的字符。此外还有不常用的参数实体，用于在 DTD 的声明中增加特殊的异常。

(6) 标记节

标记节(Marked Section)是为了将一个主文档做成多个版本，即在一段正文中做出条件正文域标记，其形式是：

```
<![关键字[被标记正文段]]>
```

其中关键字有：

- INCLUDE 被标记正文段应包括在文档中，并正常处理。
- IGNORE 被标记正文段应略去，不输出这部分。
- CDATA 被标记的字符串若有 SGML 标记名或实体名，则当做一般字符串处理。
- RCDATA 被标记的字符串若有 SGML 的标记，如同 CDATA，若有实体名则按实体处理。
- TEMP 被标记的正文段在文档中是临时的。

这些关键字可用参数实体置换，可使文档更好读。先以 ‘%’ 声明参数实体：

```
<!ENTITY % Shanghai 'IGNORE'>
```

在正文段中有：

```
In such cases, the bank will reimburse the customer for all losses.
```

```
<![ % Shanghai; [
```

```
Liability is limited to ¥ 50,000.
```

```
]]>
```

% Shanghai; 是 IGNORE 的替换，意即 ‘上海客户无五万元债务的限制’。

(7) 文档样式

SGML 只处理文档结构、内容，没有为文档样式(style)制定标准。它常沿用两个标准，一为美国国防部“格式输出规格实例 FOSI”和 ISO 1996 年发布的“文档样式语义和规格说明语言 DSSSL”标准。

SGML 十分庞大，不同业务需求取其不同子集。关键的问题是‘定制’实现。至少，当前学习使用全集，既耗费资金又浪费精力。但作为学术研究是极有价值的。

15.4.3 HTML

1989 年 Tim Berners-Lee 在欧洲核物理实验室(CERN)开发出了超文本置标语言。它从 SGML 出发作出了它的应用实例。让用户只在 HTML 类型定义下为用户文档置标。简单地说，它只是一组给定标记集合。没想到这种固定格式的标记，由于其简洁性大获成功。1994-1995 年 Web 技术发展起来 HTML 成为开发页面最广泛使用的工具。

HTML 是 SGML 应用程序，它的文档模型也是树状结构模型。标记的约定和置标方式和 SGML 一样，只是文档类型定义 DTD 已由系统定义为 HTML，用户不必学习 SGML 复杂的语法，直接用标记定义实例文档。标记分以下四类：

- 文档结构定义
- 字体字型定义
- 版面布局定义
- 链接定义

它的最大特点是超文本和多媒体。实现超大本的方法是动态束定。即在文档任意地方插入一锚元素可在其中指明此处可链接到 Web 上的任何资源(URL)。例如：

```
<A NREF= “要连接的 URL 处资源” > hotspot </A>
```

当浏览器解释本元素时，在 hotspot 处加亮变色，用鼠标单击它即可跳到引用属性 HREF

指定的 URL 文档上。

HTML 以结构正文为框架发展起来的，多媒体信息以图形元素。插入 WAVE
//===== Add a example on WAVE

HTML 写的 Web 主页成了网上主要传递信息手段，脚本语言以<SCRIPT LANGUAGE=...>. Java 语言以<APPLET>元素插入。当然，相应的浏览器也必须识别这些语言的程序才能执行。

以下分述标记：

(1) 文档结构标记

```
<HTML>一个 HTML 文档</HTML>
<TITLE> 题名</TITLE>
<!--注释-->
<H1>下一级标题</H1>      //可以嵌套 6 级，至 N6
....
<P> ....                  //另起一段，无结束标记
<BR>.....                //另起一行，无结束标记
```

“头”标<HEAD></HEAD>和“体”标<BODY></BODY>是对 HTML 文档另一观点的置标，相当于 SGML 中的并发结构，可随意插入，逻辑上<HEAD>括住一个文章的标题及摘要，<BODY>是文章全文。

(2) 文档页面格式标记

分逻辑字符模式和物理字符模式。因为是逐词标记都是成对的。逻辑的有：

```
<DFN> 定义的单词      //一般斜体
<EM>  强调的单词      //一般斜体
<CITE> 书或电影主题名 //一般斜体
<CODE> 程序代码段     //打印字体
<KBD> 键盘输入
<SAMP> 计算机状态信息
<STRONG> 特别强调词   //黑体
<VAR>   代替变量的实例 //斜体
```

物理的：

```
<B> 黑体
<I> 斜体
<TI> 打印机字体
```

(3) 版面布局标记

• 各种列表 标记列表及表项：

```
<UL> //无号列表      <OL> //有序号列表
<LI> //表项          <LI> //表项 1
<LI> //表项          <LI> //表项 2
```

```

</UL> //至此结束    </OL>    //至此结束
<DL> //定义列表
<DT> //第一条目
<DD> //与<DT>交替出现，对<DT>的测试
<DT> //第二条目
<DD> //对<DT>的测试
</DL> //结束
嵌套列表，最多可嵌三层：
<OL><LI><UL><LI><OL><LI>...</OL></UL>...</OL>

```

- 预设格式

```
<PRE>
```

完全按 HTML 文档页面上给出的编排格式，其它标记无用且不许用。浏览器照原样显示，只可以有超文本链接标记。

```
</PRE>
```

- 内联图形

```
<IMG SRC=image_URL> //不成对使用的标记
```

image_URL 是将 URL 处的 .gif 或 .xbm 图形文件内联到 HTML 文档置标处。当显示页面时图形就在其中了。

 标记除了 SRC 属性之外

```
//=====
```

15.4.4 XML

XML 是扩充置标语言 (eXtensible Markup Language) 旨在 WWW 的 Web 上使用 SGML。它是与 SGML 完全兼容的子语言。和 SGML 一样它仍是描述其它置标语言的元语言。它不像 HTML 为某类文档提供一组预定义标记，而是定制某类标记的置标语言。

XML 同样按 20/80 规则对 SGML 最常用部分抽取子集，而略去“功能虽强，很少用到”的部分。它易于实现和使用，是 SGML 和 HTML 之间交互操作的桥梁。它有 SGML 的文档类型定义 DTD 和完整的结构模型，适合于各种文种文档的置标。也有 HTML 易于连接网上搜索引擎的优点。1996 年 11 月 XML 1.0 正式发布以来，成为当今最活跃的研究领域，各种扩充附加子语言不断出现：如 XLL→Xlinks→xpointers 扩展 XML 文档的链接能力；数字领域应用 MathML；化学领域的 CML；同步化多媒体语言 SMIL；通道标记语言 COF；古籍标记语言等等。

(1) XML 文档的三种形式

SGML 核心技术是文档类型定义 DTD。HTML 系统按 DTD 给出 300 个 (HTML4.0) 标记，用户只能用这些标记不能置新标记。前者对用户要求较高，后者又规定得太死。XML 取中，所以它的文档有三种形式：可以完全和 SGML 一样写出文档实例，即第一种文档形式先形式化地定义文档的类型、元素、属性、实体的标记，再以这些标记定义文档实例。第二种形式是不

作 DTD 的形式定义，但遵照文档模型直接置标写出文档实例。这样，文档的标记必须是良定义 (well-formed) 的。即严格的标记配对，并作出文档独立声明，即解释系统不要找形式定义，就页面上的标记解释。由于用户可以自定义标记，它比 HTML 强大得多。第三种形式也不作完整的 DTD 形式定义，但可以引用已有的 DTD 定义。称之为合法 (valid) 的 XML 文档。

指出 XML 文档是否独立 是非常重要的，为此，所有 XML 文档的第一行必须是处理指令。即使是注释也在本行之后。以下用例子说明：

例 15-10 一个出版商清单

```
<?xml version = "1.0" standalone = "yes"? >           //处理指令
<!DOCTYPE document [                                     //文档类型定义
    <!ELEMENT document ANY>
    <!ELEMENT title (#PCDATA)>
    <!ELEMENT publisher (name, email, homepage, address, voice, fax)+>
    <!ELEMENT name (#PCDATA)>
    <!ELEMENT email (#PCDATA)>
    <!ELEMENT homepage (#PCDATA)>
    <!ELEMENT address (#PCDATA)>
    <!ELEMENT fax (#PCDATA)>
]>

<document>                                               //以下文档实例
<title>
    publishers of the Music of New York women composers
</title>
<publisher>
<name> ACA-American composers Alliance </name>
    <email> info @ composers.com </email>
    <homepage> http: //www.composers.com/</homepage>
    <address> 170 west 74th st.NY NY10023 </address>
    <voice> 212-362-8900 </voice>
    <fax> 212-874-8605 </fax>
</publisher>
<publisher>
    <name> Alfred Publishing </name>
    <email></email>                                     //***
    <homepage / >                                       //**
    <address>15535 Morrison South Oaks CA 91403 </address>
    <voice></voice>
    <fax/>
</publisher>
```

```
</document>
```

这个例子也是 SGML 文档实例，我们并不陌生。形式文档类型定义中<document>作为根元素。由于起始、结束标记在 XML 中必须出现，最小限定规定指示符“—”则省略。内容模型用了保留字 ANY，它只用于根元素，指明所有 PCDATA 及其它字符数据、元素均可出现在此处。<document>有两个子元素。<publisher>又五个子元素。

以下是文档实例，XML 允许整个形式类型定义不要，此时处理指令中的 standalone?必须是“yes”。由于各元素都是良定义的文档实例可单独使用，浏览器就按各元素出现的次序，位置解释。显然，元素必须简单，若有复杂关系还得形式定义。

请注意，第二个 publisher(出版商)没上网，无<email>号和主页，“***”，“**”两行是空元素的两种写法。

第三种形式的 XML 文档也举一例子：

例 15-11 一个合法的 XML 文档

```
<?xml version="1.0"? >
<!DOCTYPE advert SYSTEM "http://www.foo.org/ad.dtd">
<advert>
  <headline>...<plcl>...</headline>
  <text>...</text>
</advert>
```

这个例子的处理指令没有 standalone，意即它解释本文档标记时不限于本文档，要通过系统引用其它文档中的 DTD，它作文档类型声明，却不给出元素定义，它依然是“合法”的。显然，该类型实例文档也必须是良定义的。

无 DTD 文档的任何元素均可以有属性(用名字、‘=’和属性值给出)属性的类型隐含是 CDATA。因为文档内容中如果有‘<’‘>’符号存在，在无 DTD 可匹配情况下，浏览器必然把它当做标记解释。因此，XML 把它们预定义为<(<)、>(>)、&apos(‘)、"(“)、&(&)五个字串，除非它们在 CDATA 节中。

(2) XSL 可扩展样式语言

XML 文档通过 XSL 可扩展样式语言描述的样式表(它本身也是 XML 的文档)，执行后产生一个 HTML 文档，即 XSL 把 XML 文档翻译为 HTML 文档显现。

在 XML 看来<xsl>是一个根元素，它由一到多个规则<rule>元素组成：

```
<xsl>
  <rule>
    <target-element type = "tagname"/>
    action
  </rule>
  <rule>
    <target-element/>
    <children/>
  </rule>
```

```
</xsl>
```

`<target-element>`本身是一个标记，是指明本规则对 XML 对应的文档中的某个元素，由 `type` 指明。`action` 是一组动作，将 HTML 标记，和目标元素内容如何组合起来。例如，XML 的元素是：

```
<book>Principia Discordia </book>
```

合并为新文本时是：

```
<p>Principia Discordia </p>
```

`action` 就应用写成：

```
<p><children/></p>
```

上例中第二个规则`<target-element/>`未指明与 XML 上例中第二个规则`<target-element/>`未指明与 XML 的哪个元素匹配。意指所有未指明规划的元素均与本规则匹配，其中`<children/>`未指明与 HTML 哪个标记组合，意即按 XSL 解释器缺省规则组合。此外，在缺省的情况下，当遇到内容中有子标记怎么办？按层叠样式表单 (CSS) 原理，无论是否缺省，均按第一次匹配的规则对子元素再次迭代匹配。

XML/XSL 把文档的逻辑结构和显现格式显式分离，获得较大的灵活性。

例 15-12 为例 15-10 出版商清单写一 XSL 的样式表

```
<xsl>
  <rule>
    <root/>
    <HTML>
      <children/>
    </HTML>
  </rule>
  <rule>
    <target-element type = "title"/>
    <HEAD>
      <TITLE><children/></TITLE>
    <HEAD>
  </rule>
  <rule>
    <target-element type = "publisher"/>
    <DT/>
    <select-elements>
      <target-element type = "name"/>
    </select-elements>
    <DD/><UL>
    <select-elements>
      <target-element type = "email"/>
```

```

</select-elements>
<select-elements>
  <target-element tyle = "homepage"/>
</select-elements>
<select-elements>
  <target-element tyle = "address"/>
</select-elements>
<select-elements>
  <target-element tyle = "voice"/>
</select-elements>
<select-elements>
  <target-element tyle = "fax"/>
</select-elements>
</UL>
</rule>
<rule>
  <target-element type = "address"/>
  <target-element type = "email"/>
  <target-element type = "voice"/>
  <target-element tyle = "fax"/>
<LI><children/></LI>
</rule>
<rule>
  <target-element type = "homepage"/>
<LI><a Href = "=text"><children1><children/><1A></LI>
</rule>
</xsl>

```

例中的大写标记是 HTML 的出版商的 name 作为定义项<DT>出现，其余元素<DD>并列表(不加序号)。

这里还要说明以下几点：

- 根元素<root>

一个新文档就是一个根元素，多数情况下要特殊处理，也许是不在 XML 中置标的元素，所以 XSL 第一条规则就是根规则。本例因无其它特殊处理，只说把 XML 文档将记为<HTML>括着的一个 HTML 文档。

- 选择元素<select-elements>

XML 的元素有其给定顺序，当 xsl 规则以 <children/> 输出时，按其原顺序。如果需要重排，或 XML 相同元素的子元素不一致。以选择元素标记套在目标元素之外，则输出文档按选择元素给出的次序给出。本例两次序一致。

• 每个 XSL 规则可以有多个目标元素(为例 15-12 第二规则)，一个目标元素当有多个规划时只能和一个规则匹配，它会选最精确的。如果一个也不合适就用缺省的(例 15-12 中第二个出版商与第二个规则匹配)。

• 链接到 XML

把编好的 XSL 样式单链接到 XML 声明中按以下格式：

```
<?xml version = 1.0 standalone = "no"?>
<?xml_stylesheet href = "publisher.xsl type="text/xsl"?>
<!DOCTYPE document SYSTEM"pabliher.dtd" [
]>
<document>
...

```

意思是把 xsl 样式表单作为一文档类型定义文件 dtd 连接出来。

(3) 样式属性

从上例中可以看出 HTML 的标记 xsl 是采用的，除了、<I>、<TT>过于简单，<CODE>、和不具体之外。其余 HTML 标记在文档物理结构编排上已经很好了。只是数据格式化、字体等 XML 作了扩充，主要针对<DIV>(用于块一级元分区)、和(用于元素内部的小分区)两个标记，其形式是：

```
<xsl>
  <rule>
    <target-element>
      <DIV font_size = "zopt" font_weight = "bdd" font_family = "Courier">
        <children/>
      </DIV>
    </rule>
  </xsl>

```

为<DIV>标记增加了字体大小，粗组，字形等属性。每种属性有多个值(查手册)。也可以写作：

```
<DIV style = "font_size: zopt; font_weight: bold; font_family: courier; ">
```

除了字形字体属性外，还有 color(颜色)、text(正文)、box(框)和分类属性(display, white-space, list-style-type, list-style-image, list-style-position)

• 样式表元素也可定义属性，一般列在置标元素之后：

```
<attribute name = "attname" value="attvalue"/>
```

例 15-13 如果希望把出版商 IDG Books 出版的\$39.95 元的书名都加黑体的标题。

可写规则

```
<rule>
  <element name = "book">
    <attribute name = "publisher" value = "IDG Gooks"/>
    <attribute name = "price" value = "$39.95"/>
    <target-element type = "title"/>
  </element>
</rule>

```

```

    <SPAN style = "font_weight:bold">
        <children/>
    </SPAN>
</element>
</rule>

```

样式元素预定义属性有：

ID 标识属性 它的值必须全局唯一，为目标元素增加一ID 值，使同一规则可用
 限于多个元素。

CLASS 分类属性指出 ID 属于哪类，类属性可作为一元素出现在某元素内：

```

    <class attribute = "class_name"/>

```

ONLY 位置属性 指出是父元素的唯一子元素(为其写格式)

POSITION 位置属性 指出一组元素的首尾(作不同格式)

- 规则本身也有优先级属性，这是为了一个元素匹配多个规则

```

<rule importance = "10">
    <target-element type = "author" position = "last_of_any"/>
    <children/>
</rule>

```

(4) 嵌入 JavaScript

XML 和 JavaScript 直接相通把 JavaScript 解释器作为插件插入 XML 解释器之后，可以直接写表达式：

```

<rule>
    <target_element type="note"/>
    <SPAN font_size="10+2+'pt' ">
        <children/>
    </SPAN>
</rule>

```

10+2+'pt' 即 12pt，它用了 JavaScript 表达。

可以用求值标记 EVAL

```

<rule>
    <target-element type = "formula"/>
    <DIV>
        <eval>
            "2+2="+ (2+2)
        </eval>
    </DIV>
</rule>

```

(2+2) 是的 JavaScript 表达式，求值为 4 再联到 "2+2=" 字符串上。还可以声明函数：

```

<define-script><![CDATA[

```

```

function faetorial(n) {
  var result = 1;
  for (I = 1; I <= n; I ++){
    result = result*i;
  }
  refurn result; }
]]>
</define-script>

```

JavaScript 程序必须嵌入 CDATA 节中，引用时：

```

<rule>
  <target-element type = "formula"/>
  <DIV>
    <eval>
      factorial (10);
    </eval>
  </DIV>
</rule>

```

(5) 非 XML 数据的处理

XML 和 SGML 有一样的实体 ENTITY 机制，可以引用内部实体和外部实体，当 Web 页面引用 GIF 和 JPEG 图像、Java applet、Active X 控件、以及各种声音实体时，XML 对这些数据是不识别的，称 UNPARSED(未释义)。把未释义实体当点箱引用到处理它的程序(也是黑箱)，它们自己相互识别也能工作。

XML 提供表示法 NOTATION 元素，表示法指定的元素是非 XML 实体，它遵循不同于 XML 的一组规则。实际上是外部标识代表的一段程序。

一般形式是：

```
<NOTATION name SYSTEM "externalID">
```

单 NOTATION 用法：

```

<!ELEMENT sound EMPTY >
<!ATTLIST sound source ENTITY #REQUIRED>
<!ATTLIST sound player NOTATION #REQUIRED>
<!ENTITY spacemusic SYSTEM"/sounds/space.wav">
<!NOTATION sm SYSTEM "mplay32.exe">

```

有了这个定义，文档中可写：

```
<sound src = "&spacemusic" player = "sm">
```

也可以用于枚举列表：

```

<!ELEMENT sound EMPTY >
<!ATTLIST sound source ENTITY #REQUIRED>
<!ENTITY spacemusic SYSTEM "/sounds/space.wav">

```

```

<!NOTATION mp SYSTEM "mplay32.exe">
<!NOTATION st SYSTEM "soundtool">
<!NOTATION pdf SYSTEM "acrobat.exe">
<!NOTATION sm SYSTEM "Sound Machine">
<!ATTLIST sound player NOTATIONS(mp | sm | st)#REQUIRED>

```

声音元素的演精器属性可以是 mp(可执行文件), sm(扬声器), st(声音工具)。请注意 NOTATIONS 加了“S”。

15.5 脚本语言

无论哪种置标语言都只是处理数据(对象)。它的“计算”是由该处理系统的软件实现的,这种语言不为使用者提供编制计算程序的功能。有时很不方便。例如,用 HTML 文档显示五个各占一行的数,可以写:

```

<body>
1
2
3
4
5
</body>

```

浏览后这些数显示出来,这些操作用“计算”程序写:

```

<SCRIPT Language = "JavaScript">
<!--for (i = 4;i <= 5;i ++){
    document.write("“+i”+”<BR>”);
} -->
</SCRIPT>

```

这个程序不难看懂。这个与 Java 非常相似的语言是 JavaScript 脚本语言。它在 HTML 文档中附上一处理该文档的“脚本”,不需经过编译-连接,即可直接解释执行。脚本语言是为服务于某种业务的小的程序设计语言,一般解释执行。虽有较大的局限性,但在其特定功能方面有明显的优点,例如,处理、显示、打印正文文件的 perl、PostScript,客户/服务器应用中的 VBScript, JavaScript。

本节介绍 JavaScript 目的是希望读者了解当今小语言设计采用的机制,及其如何为其目标服务的。

15.5.1 JavaScript 概述

JavaScript 是为网络客户端用户编制处理 Web 页面上数据、增加交互性的小程序而设,减轻服务器端的负担,从而可减少网络传输,增加用户的方便性。这些工作 Java 小应用程序

全可以做，但 JavaScript 做起来更简洁、利索。

JavaScript 的世界只限于 HTML 文档浏览器（JavaScript 的解释器作为插件、集成到浏览器）在这个意义上它是平台无关的。它却与浏览器密切相关，因为运行支持由浏览器提供。

JavaScript 是对象语言但不是面向对象的。它只能建立预定义（内置）类对象的实例对象不能设计（派生）新的类对象。故无继承概念。

JavaScript 不能打开、读/写、保存本机和服务器上的文件。它只能处理页面上见到或引入的信息。因而它也是安全的，程序有毛病最多只能破坏本页，对系统无法损害。

JavaScript 是全动态束定的，因为它解释执行，对象在执行中动态建立，引用检查也是动态的。所以它的变量声不指定类型，也不作类型检查是弱类型的。这适于最终用户使用，也保持解释器小巧。脚本语言大多如此，如 AppleScript HyperTalk、dBASE。

Web 浏览器可以操作硬盘以便保存下载下来的文件。由于 JavaScript 可以让浏览器做更多的工作，当然“继承”了写硬盘操作，它可以利用浏览器创建文件以保存信息供以后使用，但必须通过 Cookie 文件。比 Java 小程序更适用。

JavaScript 程序内嵌于 HTML 文档（见序言中示例），<SCRIPT>本身只是 HTML 中的一种标记，为使旧式浏览器（无 JavaScript 解释器）能兼容地工作，JavaScript 程序放在注释中，它使 Web 程序员能访问、修改所有的 HTML 标记、表单、窗口、图象、书签、链接、锚点。

JavaScript 是事件驱动的，它预定义了 9 个事件，把要运行的对象（即函数）交由事件句柄，它就运行起来了。

JavaScript 提供以下功能：

- 访问 Java 小程序的方法
- 与其它插件、小程序、脚本通信
- 客户端表单验证（发送到服务器之前）
- 动态生成 HTML 文件
- 创建简单的交互程序

JavaScript 与 Java 的对照表

| JavaScript | Java |
|----------------|----------------|
| 代码直接解释 | 编译后存放在服务器下载运行 |
| 基于对象、不能定义类、无继承 | 面向对象 |
| 弱类型 | 强类型 |
| 动态束定 | 编译时静态束定 |
| 限制磁盘访问（可写盘） | 限制磁盘访问（不能自动写盘） |
| 脚本限制浏览器功能 | 编译后可作小程序，可独立运行 |
| 无标准库 | 有大量的库 |
| 尚在变化之中 | 相对稳定 |

15.5.2 JavaScript 语言特征

JavaScript 是为客户端用户使用，类似于 Java 但比 Java 简单

(1) 常量、变量、表达式、运算符

与 Java 基本相同。变量需作声明：

```
var foo = 23;    //初值不必须
```

保留了 C 的条件表达式：

```
is Real = (Imagination <= Reality) ? true: false
```

基本类型为数值（整型、浮点、布尔、字符串、null 以什么类型值初始化变量，则变量即为该类型。以后可赋其它类型值。

赋值 ‘=’ 是运算符，为变量赋值仍用表达式，它是基于表达式语言。

变量分局部变量（局部于函数）和全程变量（在一个 JavaScript 程序中）

(2) 控制语句

有条件语句（if ...else），循环语句（for,while）及循环操作语句（continue,break），对象操作语句（for...in、new、this、with）和注释语句（//、/*...*/）。也有花括号括住语句组的规定。

break 使循环提前结束，continue 指示本次循环结束，继续循环。

for...in 是操作对象的特殊循环语句，可依次访问对象中所有属性：

```
for (propertyName in document) {  
    document.write (propertyName + "<BR>");  
}
```

可以打印文档对象 document 中所有属性。因有 “
” 指示一行一个，其余空白。

New, this 用法同 Java

With 语句是对某对象的属性操作时缺省对象名的写法。例如，对 document 的两属性赋值：

```
Document.fgColor="#000000"
```

```
Document.bgColor="#FFFFFF"
```

可写为：

```
with (document) {  
    fgColor = "000000"  
    bgColor = "#FFFFFF"  
}
```

(3) 函数

脚本语言大多以函数为其核心部分，函数的定义形式和其它语言无差别，先装入后使用：

例 15-14 使用函数生成 HTML 文档的例子

```
<HTML>
```

```
<HEAD>
```

```
<SCRIPT Language = "javascript">
```

```
<!--remember me?    //注释内嵌 Java script 程序
```

```
var age = 0;
```

```
function myHeader(age) {    //函数定义
```

```

document.write (“<TITLE> the “tag + Year Old Page</TITLE>
}
function myBody(date, color) {
    document.write(“<BODY bgcolor = ”+color+”>”);
    document.write(“<H3> Welcome to My Homepage!</H3>”);
    document.write(“The date is” +date+”<BR>”);
function manyLinks(index) {
    if (index == 1) {
        return http://www.yahoo.com;
    }
    else if(index == 2) {
        return http://home.netscap.com;
    }
    else return http://www.idsoftware.com;
    //return the title                                //注释
    my Header(33);                                    //函数调用
    //done for the moment!-->                          //HTML 注释至此
</SCRIPT>
</HEAD>
<SCRIPT Language = ”javascript”>                    //又开始一段程序
<!--
my Body(“July22, 1996”, “#FFFFFF”);                  //函数调用
document.write ( “<A href = ”+manyLinks(2)+”>Here’s a link </A>” );
//-->
</SCRIPT>

```

JavaScript 的函数参数传递采用引用机制，返回值是任何有效类型。如果运行后不返回值（只有操作）则自动返回 `true`;

JavaScript 提供的内置（预定义）函数有：

| | |
|--------------------------------------|----------------------------|
| <code>escape (str)</code> | 将串转换为 HTML 字符表示，如空串为%20 |
| <code>unescape(str)</code> | <code>escape()</code> 的逆转换 |
| <code>eval(str)</code> | 对字符串求值 |
| <code>parseFloat (str, radix)</code> | 将字符串转换为浮点数 |
| <code>parseInt(str)</code> | 将字符串转换为整数 |

（4）事件

JavaScript 是事件驱动，装载主页、卸载、击鼠标键、按下按钮、在某个域中输入都是事件。事件由事件句柄确定，一组语句，一个函数均可作为事件处理代码，一般按以下语法：

```
<HTMLTAG event Headler = ”JavaScript Code”>
```

例如，标准函数 `alert()` 在屏幕上产生一个警告对话框。On Click 是鼠标事件句柄，当单击一接收此事件的对象时激活 Click 事件（预定义的）

在 HTML 的公共趋文本链接中嵌入 JavaScript 事件句柄。单击该链接时对话框中显示事件处理的一段正文。单击框中 OK 钮，则返回到页面。

```
<A HREF="#" onClick = "aler +(‘Wow! It Works!’>; “> Click here for a message!</A>
```

JavaScript 有两类事件：系统的（无须用户介入即可发生），鼠标的（以用户行为触发）。

JavaScript 预定义事件有九个：

| 事件 | 句柄 | 触发事件 |
|------------|-------------|------------------------|
| blur | onBlur | 用户单击离开表单元素，focus 的逆事件。 |
| change | onChange | 对表单正文、正文域、选择对象，用户改变值时。 |
| click | onClick | 用户单击表单元素或链接时。 |
| focus | onFocus | 用户单击表单元素时。 |
| mouse over | onMouseOver | 鼠标指针移动到某对象时，自动激活。 |
| select | onSelect | 表单中用户选择输入字段时。 |
| submit | onSubmit | 用户提交表单时。 |
| load* | onLoad | 主页装载后自动激活。 |
| unload* | onUnload | 用户离开页面自动激活。 |

以下是事件的例子。当鼠标指针移到某对象上时该事件触发。

例 15-15 当鼠标指针指向不同正文，示警窗出现对该文的解释；

```
<HTML>
<HEAD>
<SCRIT Language = "JavaScript">
<!-- hide from non-JavaScript browsers
function explain1() {
    window.alert ("This is a link to the white House.");
}
function explain2() {
    window.alert ("This is a link to the QUE Home page.");
}
function explain3() {
    window.alert("This is a link to the Auther's Home page");
}
//end hide--->
</SCRIPT>
</HEAD>
```

```

<BODY>
<H3 > Chater4: On Mouse Over</H3>
<HR>
<A Href = http://www.mcp.com onMouseOver =
    “explain2( )”>QUE Publishing </A><BR><BR>
<A Href = http://www.whitehouse.gov” onMouseOver =
    “explain2( )”>White House </A><BR><BR>
<A Href = http://www.lobosoft.com” onMouseOver =
    “explain2( )”>Lobo Soft Soft Ware </A><BR><BR>
</BODY>
</HTML>

```

句柄 `onMouseOver` 在 `<A>` 标记内定义，当鼠标指针指向 `<A>here` 中的正文时激活。激活后执行相应函数。

(5) 数组

JavaScript 有内置数组 `Forms`(窗体)和 `Anchor`s (锚)不用声明即可操作其元素。它是 `document` 对象的属性，所以有：

```

document.forms[0]    //第一窗体，下标从 0 开始
document.forms[1]    //第二窗体

```

在 HTML 画面中每出现一个锚元素则自动加入 `Anchor`s 数组。有了例 15-页面可以写出 JavaScript 程序段验证 `Anchor`s 数组元素的值：

```

document.write(“There are “ +anchors.length+”anchors.<BR>”);
for (var i\0; i<anchors.length++){
    document.write(“Anchor”+i+1+”:”+anchors [i]. value +<BR>”);
}

```

运行后即例 15-15 中的三个 URL 值。

用户定义的数组是利用函数创建数组对象：

```

function MakeArray(n) {
    this. Length = n;
    for (var i =1; i<= n; i++){
        this[i] = ” “; //this 是数组对象代名词
    }
    return this;
}          //此函数可创建无名 n 个空元素的数组

```

若我们要创建 10 个人名的数组时：

```
nameArray = new MakeArray (10);
```

我们就可以使用它：

```
nameArray[1]="George Washington"
```

```
nameArray[2]="John Adams"
```

```
nameArray[3]="Thomas Jefferson"
```

(6)对象

JavaScript 提供了卡在 HTML 文档上操作的一组预定义对象（类），每个类对象都有其属性和方法（即函数）。熟读了这些类创建实例对象就很方便了，创建方法很简单如同上例创建数组对象用 `new` 方法，属性和方法的操纵采用‘`·`’表示法。

JavaScript 预定义的对象体系结构如图 15-9 所示。

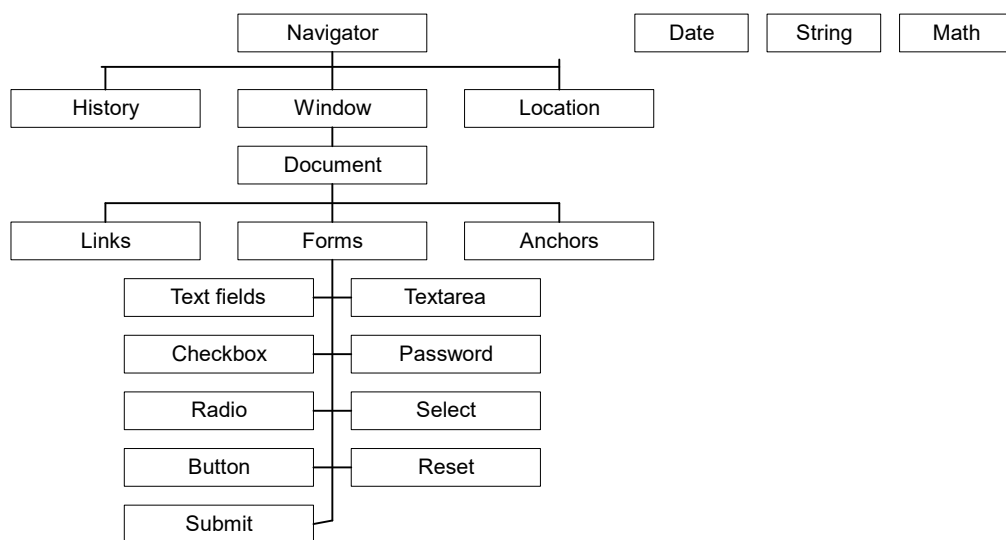


图 15-9 JavaScript 的类体系

现对其功能简略介绍如下：

- Navigator 只有六个属性无方法，属性给出有关浏览器的信息，引用查询。
- History 有关访问过的链接 URL 的历史列表，只有一个长度属性，有 `go()`, `back()`, `forword()` 方法。
- Location 识别查找完整 URL，有协议、宿主机名、端口、路径名、查找、哈希属性，没有方法。
- Window 有有关窗口信息的六个属性，操纵窗口的七个方法。
- Document 有关当前文档信息的 17 个属性，操纵文档的五个方法 `clear()`, `close()`, `open()`, `write()`, `writeIn()`。
- Forms HTML 中 `<FORM>` 标记的出现即创建—Forms 的实例对象，它包括多个子对象（也是属性）和八个自己的属性，只有 `submit`（提交）一个方法。
- Links 有关超链接的处理对象
- Anchor 记录 URL 的数组对象
- Math 相当于其它语言数学包，八个属性，17 个方法，包括三角、对数、随机数等函数。

- String 只有一个长度属性，有关字符串各种操作转换方法 18 个。最常用的对象。
- Date 客户攫取设置当前年、月、日期信息的对象。没有属性，只有 20 个方法。

(7)实例

例 15 -- 16 每秒钟显示一次当前的时间。

```
<HTML>
<HEAD>
<TITLE> JavaScript Clock </TITLE>
<SCRIPT Language = "JavaScript">
<!-- Hide me from old Browsers-hopefully
//Netscapes Clock-start
//this code was taken Netscapes JavaScript document ation
//at www.netscape.com on Jan ,25. 96
var timerID = null;
var timerRunning = false;
function stopclock() {
    if (timerRunning)
        clear TimeOut(timerID);
    timerRunning = false;
}
function startclock() {
    //Make sure the clock is stopped
```

15.6 小结

- 高级程序语言的宗旨是平台无关，由于机器指令系统不同，运行能力不同，**Java** 以前所有高级程序设计语言不能做到平台无关。
- 平台无关语言实现策略，传递统一的中间代码比较理想。它接近实现代码语义丢失少，解释器小巧，先便宜可利用复杂的查工具，不会增大传递信息量。为了平台无关各种环境均应配备解释器。
- **Java** 类似 C++ 比它更面向对象，更简洁，更安全，有较强的动态性，支持线程编程，取消了 C++ 不利于平台无关的机制。如头文件和宏。指针等。
 - **Java** 的字节代码 Unicode 和 ISO lo646-1 BMP 兼容，后字节和 ISO ASCLL 兼容。
 - **Java** 的新机制：
 - 提供虚基类 **array**, **Java** 的所有基本类型都可以有数组对象。
 - 界面机制，可以实现多继承。
 - 引入包和编译单元。包是类和界面集合。为了简化各字面向管理。实现可见性控制。预定义包提供了程序设计环境。

--提供线程包支持以 `csp` 模型开发并发程序。

--新设关键字有 `final,finally,mafiva,fransient,abstract,instanceof`

- **Java** 虚拟机由类装入器、字节代码解释器、无用单元收集器、线程同步设施组成。为了提高代码运行速度增加及时 (**JIT**) 编译器。
- 第三代虚拟机只便宜运行时间长的热点，混用解释和编译后的机器码，效率最高。是当今发展前沿。
- 置标语言不是程序设计语言，是电子文档发展以后，以计算机或网上自动处理电子文档而设计的语言。它们一般解释执行，平台无关。
- 电子文档三类标准，信息的基本表达，信息结构描述，电子信息再现处理。
- **SGML** 把文档看作是对象集合，对象即元素。每个元素是标记括着的内容，内容又可嵌入元素，这就形成树形分层体系结构。也是文档的逻辑结构。**SGML** 不涉及类容的语义。
- **SGML** 以形成的方法定义文档结构几其中的元素、属性、实体。属性、实体也是元素，实体可以引用内部元素和外部元素，外部则通过链接。
- **SGML** 只是元置标语言，它的文档实例还要通过其他描述格式、字体、编排的语言。现它采用的是 US DOD 的 FOSI 和 ISO 的 DSSSL。
- **HTML** 是一组标记构成的置标语言，它用预定义的标记直接写 **HEML** 文档，并可立即上网传输。是当今网上主页的主要工具。
- **HTML** 也有文档结构定义，知识它用预定义标记；还有字体字型定义的标记和版面布局定义的标记（如列表）。最重要的是链接定义可以利用。显现网上资源。
- **XML** 是 **SGML** 在网上使用的子集,它也是置标语言的元语言.它描述的是文档数据的逻辑结构.它有一个子语言 **XSL** 按 **CSS** 层叠样式标型转换为 **HTML**.
- **XML** 由三种使用形式：和 **SGML** 一样写文档实例；不作 **DTD** 形式定义直接置标写文档实例；不写 **DTD** 形式定义，引用 **DTD** 文写，将其实例置标写文档实例。后者称合法的，后二者称良定义的。
- **XSL** 语言是 **XML** 有关表示和格式的元素集合。它的元素、内容、属性、实体的概念和 **SGML**、**XML** 完全一样。
- 目前 **XSL** 文档可以按外部实体链接。
- **XML** 和 **JavaScript** 直接相通，嵌入 **JavaScript** 表达式或函数可对页面处理生成更加灵活的 **HTML** 文档。
- 实体引用可以把非 **XML** 数据和处理程序当黑箱引用，引用到一起，它们相互识别。可播放声音、绘图，在页面上完成多媒体实现。
- 脚本语言为服务于某种业务的小型程序设计语言，一般解释执行，有较大的局限性。但在其工作目标上尤其独特的优势。
- **JavaScript** 是 C/S 计算中客户端编制小应用程序的脚本语言，有浏览器执行可以减轻服务器端的“计算工作量”。**JavaScript** 本身的解释器作为插件插入客户端浏览器。
- **JavaScript** 是基于对象的不是面向对象的，它只能在浏览器提供类库之下声称自己的实例对象，不能定义类。**JavaScript**+程序可嵌入 **HTML**，**XML**，增加处理文档的能力。

- JavaScript 和一般过程小语言相近，特别是 C。函数是它最重要的机制。在 HTML 注释内（对于只识别 HTML 不识别 JavaScript 的浏览器，整个程序略去）声明变量，写表达式或函数即为 JavaScript 程序。
- JavaScript 是事件驱动的，系统给出 9 种预定义的事件支持 JavaScript 程序运作。
- JavaScript 提供了 Navigator 对象体系及 Date, String, Math 支持编程的类对象。对象以 new 生成。用户编程是给出实例对象的值，并引用方法（发消息）。

习题

- 15.1 试述 Java 取消指针后带来的好处与坏处。传统二叉树以带指针的结构(C++)定义,试写出 Java 的二叉树定义(方法略)。
- 15.2 Java 程序结构是:应用程序是含有一个 main()方法的类的集合;小应用程序是 APPLET 类的子类.如果小应用程序中有 main()方法,能运行吗?为什么?
- 15.3 Java 的包是类和接口集合,包是程序运行的单位吗?为什么与 Ada 不同?
- 15.4 Java 方法调用有值调用和引用调用,试写出这两种调用的示例.
- 15.5 Java 只有实例对象的构造子没有析构子,它是用什么方法销毁对象的?
- 15.6 Java 没有虚函数概念,它是如何实施动态束定的,试写一例说明之.
- 15.7 Java 的类继承不分公和私有继承,为什么?比较 Java 的 FINAL(定止)修饰符 CONST,为什么 FINAL 最后能取代 CONST?
- 15.8 将例 15-1 编译后,嵌入主页运行之.
- 15.9 为一本教科书写出 SGML 语言的文档类型定义.
- 15.10 设计一个软件规格说明实现 SGML 的 DOCTYPE .
- 15.11 设计一个解释器实现 XSL 的子集.
- 15.12 设计一个解释器的规格说明实现 XML 的子集.
- 15.13 试定义 JavaScript 的解释器与 XML 的接口.

第16章 指称语义的原理与应用

指称语义学是Christopher Strachey和Dana Scott在1970年提出的。在形式化语义学的早期，人们都把注意力放在操作语义上，而Strachey和Scott则在纯数学的基础上进行语义学研究。这种方法最初也被称为数学语义学。它的优点是：不需要在计算机上实际运行程序，就能预测程序的行为，了解程序设计语言全部的涵义。另一个潜在的优点是：能够对程序进行推理，如证明两个程序等价。

指称语义学的一个显著特征是：程序中的每一个短语（表达式、命令、声明等）都有其意义。而且，每个短语的意义是由它的子短语来定义的。这就对语义加上了结构，它是与语言的语法结构平行的。每个短语的语义函数就是短语的指称意义。其现代名称为指称语义学。

本章先给出指称语义的基本概念，接下来研究用指称语义说明程序设计语言的基本概念，如：束定、存储、抽象与参数、基本类型与复合类型。接着讨论怎样将这些技术应用到一完整语言中，最后讨论指称语义的其它应用。

16.1 指称语义原理

从数学的观点，一个程序可以看作是从输入到输出的映射 $P(I) \rightarrow O$ ，即输入域(domain)上的值，经过程序 P 变为输出域(range)的值。沿用这个办法，我们将程序短语集 P 中的某个短语 p ，映射为数学域上的值 d 。 $\phi \llbracket p \rrbracket \rightarrow d \quad (p \in P, d \in D)$ 。

ϕ 为短语 p 的语义函数， D 为语义域。当然，一个短语 p 可以有多个语义函数 ϕ, ψ, ξ, \dots 。同一语义函数也可以输入不同的短语 p, p_1, p_2, \dots 。

我们把语义域 D 中的数学实体 d ，或以辅助函数表达的复杂数学实体 d' ，称为该短语的数学指称物，即短语在语义函数下的指称语义。这样，程序的行为可以完全以数学实体表达。所以，指称语义最早称为数学语义。

指称语义描述的是语义函数映射的后果，不反映如何映射的过程，更没有过程的时间性。而程序设计语言的时间性只能反映到值所表达的状态上。因此，如何把存储、环境、状态抽象为数学表达是指称语义关心的事。

本节以简单的命令式语言演示如何用指称语义描述程序语义。

16.1.1 语义函数和辅助函数

语义函数的变元是语法中的短语（当然可以是代表整个程序的短语，如PROGRAM），其映射指称物即语义。

例16-1 描述二进制数的语义

二进制数的语法：

| | |
|---------------|----------|
| Numeral ::= 0 | (16.1-a) |
| 1 | (16.1-b) |
| Numerals 0 | (16.1-c) |
| Numerals 1 | (16.1-d) |

我们给出求值的语义函数:将Numerals集合中的对象映射为自然数:

valuation: Numerals \rightarrow Natural (16.2)

这样, 通过语义函数valuation, 任何二进制符号串都可以得到自然数的指称解释。因此, 按语法的产生式, 我们给出以下语义函数:

valuation $\llbracket 0 \rrbracket = 0$

空心括号中的0是语法符号, 后面‘0’的自然数。

| | |
|---|----------------------------|
| valuation $\llbracket 1 \rrbracket = 1$ | |
| valuation $\llbracket N0 \rrbracket = 2 \times \text{valuation } \llbracket N \rrbracket$ | // $N \in \text{Numerals}$ |
| valuation $\llbracket N1 \rrbracket = 2 \times \text{valuation } \llbracket N \rrbracket + 1$ | |

这样, 对任一二进制数, 如1101, 反复用以上四个公式:

$$\begin{aligned}
 \text{valuation } \llbracket 1101 \rrbracket &= 2 \times \text{valuation } \llbracket 110 \rrbracket + 1 \\
 &= 2 \times (2 \times \text{valuation } \llbracket 11 \rrbracket) + 1 \\
 &= 2 \times (2 \times (2 \times \text{valuation } \llbracket 1 \rrbracket + 1)) + 1 \\
 &= 2 \times (2 \times (2 \times 1 + 1)) + 1 \\
 &= 13
 \end{aligned}$$

即得到‘1101’的值的语义解释, 为自然数13。

对于最简单命令式语言我们再举一例。

例16-2 计算器命令的语义描述

计算器命令的抽象语法:

| | |
|---------------|----------|
| Com ::= Expr= | (16.3) |
| Expr ::= Num | (16.4-a) |
| Expr + Expr | (16.4-b) |
| Expr - Expr | (16.4-c) |
| Expr * Expr | (16.4-d) |

Num ::= Digit | Num Digit (16.5)

Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 (16.6)

每条命令的执行都可以得到一整数, 语义函数是:

execute : Com \rightarrow Integer (16.7)

每一表达式的求值可得到一整数，则有：

$$\text{evaluate} : \text{Expr} \rightarrow \text{Integer} \quad (16.8)$$

每一个数都对应一个自然数：

$$\text{valuation} : \text{Num} \rightarrow \text{Natural} \quad (16.9)$$

这样我们就有了三个语义函数。为了细化表达式的四种定义，则增加以下辅助函数：

$$\text{sum} : \text{Integer} \times \text{Integer} \rightarrow \text{Integer} \quad (16.10)$$

$$\text{difference} : \text{Integer} \times \text{Integer} \rightarrow \text{Integer} \quad (16.11)$$

$$\text{product} : \text{Integer} \times \text{Integer} \rightarrow \text{Integer} \quad (16.12)$$

以下定义每个短语的语义函数：

$$\text{execute } \llbracket C \rrbracket = \text{execute } \llbracket E = \rrbracket = \text{evaluate } \llbracket E \rrbracket$$

其中 $C \in \text{Com}$, $E \in \text{Expr}$ 。意即每条命令执行都要对 E 求值。

再定义 Expr 的四个表达式：

$$\begin{aligned} \text{evaluate } \llbracket N \rrbracket &= \text{valuation } \llbracket N \rrbracket & (N \in \text{Num}) \\ \text{evaluate } \llbracket E_1 + E_2 \rrbracket &= \text{sum} (\text{evaluate } \llbracket E_1 \rrbracket, \text{evaluate } \llbracket E_2 \rrbracket) \\ \text{evaluate } \llbracket E_1 - E_2 \rrbracket &= \text{difference} (\text{evaluate } \llbracket E_1 \rrbracket, \text{evaluate } \llbracket E_2 \rrbracket) \\ \text{evaluate } \llbracket E_1 * E_2 \rrbracket &= \text{product} (\text{evaluate } \llbracket E_1 \rrbracket, \text{evaluate } \llbracket E_2 \rrbracket) \end{aligned}$$

再定义 Num 的两个表达式：

$$\begin{aligned} \text{valuation } \llbracket D \rrbracket &= D' & (D \in \text{Digit}, D' \in \text{Natural}) \\ \text{valuation } \llbracket ND \rrbracket &= 10 \times \text{valuation } \llbracket N \rrbracket + D' \end{aligned}$$

其中 $E_1, E_2 \in \text{Expr}$ 。 D' 是以 D 表示的自然数域的值。

这样，对任一命令的执行其语义函数可立即得出，例如：

$$\begin{aligned} \text{execute } \llbracket 40-3*9 \rrbracket & \\ &= \text{evaluate } \llbracket 40-3*9 \rrbracket \\ &= \text{product} (\text{evaluate } \llbracket 40-3 \rrbracket, \text{evaluate } \llbracket 9 \rrbracket) \\ &= \text{product} (\text{difference} (\text{evaluate } \llbracket 40 \rrbracket, \text{evaluate } \llbracket 3 \rrbracket), \text{evaluate } \llbracket 9 \rrbracket) \\ &= \text{product} (\text{difference} (\text{valuation } \llbracket 40 \rrbracket, \text{valuation } \llbracket 3 \rrbracket), \text{valuation } \llbracket 9 \rrbracket) \\ &= \text{product} (\text{difference} (40, 3), 9) \\ &= 333 \end{aligned}$$

我们把定义短语指称的式子叫做语义方程。等号左边是某个短语的语义函数，等号的右边是子短语的语义函数和辅助函数组合的定义函数，如上述。定义函数如同普通数学函数定义，可引入新变量，如：

$$\begin{aligned} \text{let } s &= 0.5 \times (a + b + c) \text{ in} \\ &\quad \text{sqrt } (s \times (s - a) \times (s - b) \times (s - c)) \end{aligned}$$

还可以引入新函数，如：

```

let succ n = n + 1 in
  if n ≥ 0 then
    sqrt( succ n )
  else
    0.0

```

定义函数也可以引入多个参数，如上式中的a, b, c。也可以如前所述多层嵌套块的局部定义。有时引入无名函数(即 λ 表达式)更方便。

16.1.2 语义域

无论作为指称物，还是辅助函数的变元和结果值都要研究域(domain)。域是具有相同性质元素值的集合。

(1) 基本域

本书前述的基本域即最简单的域，它们是：

- Character : 域元素取自某字符集。
- Integer : 域元素为整数。
- Natural : 域元素为正整数。
- Truth-Value : 域元素只有‘true’ ‘false’ 值。
- Unit : 域元素仅有一空元组()。

用户可定义枚举域：

```
Weeks=mon, tue, wed, thu, fri, sat, sun
```

或偶数域：

```
Even=...-2, 0, 2, 4, 6, 8, ...
```

以及以基本域构造的复合域。前已讲过的：

(2) 笛卡儿积域

$D \times D'$ 元素为对偶 (x, x') 其中 $x \in D, x' \in D'$ 。

$D_1 \times D_2 \times \dots \times D_n$ 元素为n元组 (x_1, x_2, \dots, x_n) ，其中 $x_i \in D_i$ 。

(3) 不相交的联合域

$D + D'$ 元素为对偶 $(\text{left } x, \text{right } x')$ 其中 $x \in D, x' \in D'$ 。

更加复杂一些，可以定义复合域。每个子域上都可以是元组。复合域是不相交的n维域 $D_1 + D_2 + \dots + D_n$ 。例如定义形状域：

```
shape=rectangle( Real×Real ) + circle Real + point
```

其中point是point Unit的简写。

请注意，矩形和圆都借助Real描述值，由于有标记rectangle和circle它们是不同的且不相交的域。

(4) 函数域

$D \rightarrow D'$ 元素为将D域值映射为D' 域的值的函数或映射。例如 $\text{Integer} \rightarrow \text{Even}$ 。

如果D域中每个元素的映射为D'域中的元素,我们称全函数。

如果域D中有的元素不能映射为D'域中的元素,我们称部分函数或偏函数。为了表达的一致性,我们假定每个域中都包含一特殊元素fail,记为 \perp ,则有:

| | |
|------------------------------|----------------|
| $f(v) \rightarrow \perp$ | 偏函数, $v \in V$ |
| $f(\perp) \rightarrow \perp$ | 严格的偏函数 |
| $f(\perp) \rightarrow v$ | 非严格函数 |

引入偏函数,偏函数域上元素间具有偏序关系,偏序关系‘ \leq ’的性质是:

- D域若具偏序性质,它必须包含唯一的底元素,记为 \perp ,且 $\perp \leq d$,d为D中任一元素。通俗解释是d得到的定义比 \perp 多。 \perp 是不对应任何值的‘值’。
- 若 $x, y \in D$, $x \leq y$ 此二元素具有偏序关系‘ \leq ’,即y得到的定义比x多。这一般就复合元素而言,即x中包含的 \perp 比y多。
- 若 $x, y, z \in D$, 则偏序关系‘ \leq ’必须是:

- [i] 自反的,即有 $x \leq x$;
- [ii] 反对称的,即若 $x \leq y, y \leq x$, 必然有 $x = y$;
- [iii] 传递的,即若 $x \leq y, y \leq z$, 必然有 $x \leq z$ 。

偏函数对研究程序语义是极为有用的。因为任何程序运行的语义都可写成:

$run: Program \rightarrow (Input \rightarrow Output)$ (16.13)

如果递归不收敛或停机得不出输出则:

$run: P(Input) \rightarrow \perp$ (16.14)

此函数即偏函数。推而广之,我们在指称语义中讨论的域均具有偏序性质。

(5) 序列域

序列域 D^* 中的元素是零个或多个选自域D中的元素有限序列,即同构序列元素。或为nil元素,或为 $x \cdot s$ 的序列,其中元素 $x \in D$,序列 $s \in D^*$,‘ \cdot ’为连接符。例如: $String = D^*$ 是序列域。它可以有以下元素:

| | |
|---|-----------------|
| nil | // 一般写法是 “ ” |
| ‘a’ \cdot nil | // 一般写法是 “a” |
| ‘B’ \cdot ‘u’ \cdot ‘s’ \cdot ‘y’ \cdot nil | // 一般写法是 “Busy” |

16.1.3 命令式语言的特殊域

有了以上基本域和复合域,我们可写出表达式求值和简单命令执行的语义,但还不足以描述命令式小语言,因为变量的行为是和环境,存储密切相关的。我们先讨论两个特殊的域。

赋值语句总是改变变量中的值。前文已经说过这在纯数学中找不到对应。指称语义是数学语义,因此,必需为储存变量的存储和影响变量值的上下文环境作出模型。当然,有的理论为这两者的作用结果建立状态模型。本书是前者。

(1) 存储域

一个抽象定义的存储器，由存储单元(槽)的集合组成，其中放可存储值。槽中的值随赋值而变。如果设想某一时刻存储槽中的值是一个存储(状态)，那么我们可以把所有存储(状态)的集合称之为Store域，其中每一元素sto就是一帧存储槽集合的快照。有一些已存入可存储值；有一些已分配了但未存入值(未定义)；有一些未分配(未使用)。例如，以下图示了Store的一个快照sto。

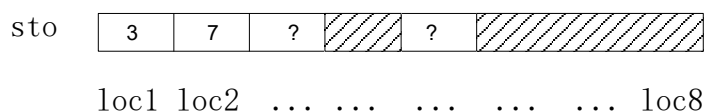


图16-1 存储快照sto的示意图

其中1, 2单元(以loc1, loc2指示)已赋了可存储值, 即3, 7。loc3, loc5分配了未定义。其余阴影槽未使用。我们把Location称(存储)单元域(相当于地址), 其元素为loci。把Storable称可存储值域。其元素为stble。于是, 存储快照集合即为每次将存储单元映射为其状态State。为了避免混乱不设State域, 则直接写:

$$\text{Store} = \text{Location} \rightarrow (\text{stored Storable} + \text{undefined} + \text{unused}) \quad (16.15)$$

括号内即为状态域, 若设状态变量state它可取上述三种值。

为了刻画存储快照, 引入以下辅助函数:

$$\text{empty_store} : \text{Store} \quad (16.16)$$

$$\text{allocate} : \text{Store} \rightarrow \text{Store} \times \text{Location} \quad (16.17)$$

$$\text{deallocate} : \text{Store} \times \text{Location} \rightarrow \text{Store} \quad (16.18)$$

$$\text{update} : \text{Store} \times \text{Location} \times \text{Storable} \rightarrow \text{Store} \quad (16.19)$$

$$\text{fetch} : \text{Store} \times \text{Location} \rightarrow \text{Storable} \quad (16.20)$$

它们的意义是明显的。empty-store是所有单元都映射为unused的Store的元素。allocate是某个存储快照sto映射为sto' 其中某些单元loc由未使用变成未定义(即分配了)。deallocate是将存储快照sto中某些单元loc变为未使用, 从而映射出新的sto'。update是在sto中以stble值将loc单元更新, 从而映射出新的sto'。fetch是从某sto的loc单元上取出stble值。显然, 我们有:

$$\text{deallocate}(\text{allocate}, \text{sto}) = \text{sto} \quad (16.21)$$

$$\text{fetch}(\text{update}(\text{sto}, \text{loc}, \text{stble}), \text{loc}) = \text{stble} \quad (16.22)$$

(16.21)是互逆操作, 和预期的语义行为一致。

现在我们给出以上辅助函数形式化的定义:

$$\text{empty_store} = \lambda \text{loc}. \text{unused} \quad (16.23)$$

$$\begin{aligned} \text{allocate } \text{sto} = \\ \text{let } \text{loc} = \text{any_unused_location}(\text{sto}) \text{ in} \\ (\text{sto} [\text{loc} \rightarrow \text{undefined}], \text{loc}) \end{aligned} \quad (16.24)$$

deallocate (sto, loc) = sto [loc \rightarrow unused] (16.25)

update (sto, loc, stble) = sto [loc \rightarrow stored stble] (16.26)

```
fetch (sto, loc) =
  let stored_value (stored stble) = stble
    stored_value (undefined) = fail
    stored_value (unused) = fail
  in
    stored_value (sto(loc)) (16.27)
```

其中any_unused_location, stored_value是语义明显的下一层辅助函数, 因为sto本身是从loc \rightarrow state的快照, sto[]中的方括号所表达的loc到state的映射是sto的等价物或解释。

(2) 环境域

由于命令式语言数据对象的名值分离, 标识符集就有已束定和未束定的不同状态(请注意这和上段的状态内涵不同, 故本书避免定义状态域State), 此外, 声明产生束定并有一定的作用域。任何表达式或语句都是在某个作用域的环境内求值。因而我们把环境模型为标识符映射成的束定状态, 并以此间接刻画作用域。当然, 束定必须有可束定体。于是用三个域 Environ, Identifier, Bindable来刻画变量束定后的环境。其中

Environ = Identifier \rightarrow (bound Bindable + unbound) (16.28)

括号内是束定状态。

为此, 定义以下辅的函数:

empty_environ : Environ (16.29)

bind : Identifier \times Bindable \rightarrow Environ (16.30)

overlay : Environ \times Environ \rightarrow Environ (16.31)

find : Environ \times Identifier \rightarrow Bindable (16.32)

它们的非形式解释是: empty_environ是所有标识符均处于未束定状态的空环境; bind是只要有了标识符束定于可束定体, 则环境状态就改变了; overlay是两束定环境组成新环境 env+env', 若某标识符在两环境中均有束定, 则新环境中以后束定复盖先束定; find是在环境 env中找出标识符I所对应的可束定体bdbl。显然, 若未束定则返回 \perp 。

我们举例说明:

例16-3 环境模型中辅助函数示例

设已定义环境 env = {i \rightarrow 1, j \rightarrow 2}, 则以下函数的解释是:

bind (k, 3) = k \rightarrow 3 // 标识符k束定于3, 花括号内是新环境

overlay ({j \rightarrow 3, k \rightarrow 4}, env) = {i \rightarrow 1, j \rightarrow 3, k \rightarrow 4} // 得已复盖的新环境

find(env, j) = 2

find(env, q) = \perp // 若env环境中, k未束定

这些辅助函数的形式定义如下:

empty_environ = $\lambda I.$ unbound (16.33)

bind(I, bdbl) = $\lambda I'. \text{if } I' = I \text{ then bound bdbl else unbound}$ (16.34)

overlay (env', env) =

$$\lambda I. \text{ if env' (I) } \neq \text{unbound then env' (I) else env (I)} \quad (16.35)$$

$$\text{find(env, I) =}$$

$$\quad \text{let bound_value(bound bdouble) = bdouble}$$

$$\quad \quad \text{bound_value(unbound) = } \perp$$

$$\quad \text{in}$$

$$\quad \quad \text{bound_value(env(I))} \quad (16.36)$$

其中 $\text{env}(I)$ 表示 env 中的 I ， bound_value 是下层辅助函数，意即求束定值。

16.2 指称语义示例

我们定义一个极小的命令式语言IMP，它只有整数和布尔值，可声明常量变量。有赋值，分枝，循环，跳出等命令。主要是说明指称语义的写法。

16.2.1 过程式小语言 IMP

抽象语法是：

$$\begin{aligned} \text{Command} &::= \text{Skip} & (16.37\text{-a}) \\ &| \text{Identifier} := \text{Expression} & (16.37\text{-b}) \\ &| \text{let Declaration in Command} & (16.37\text{-c}) \\ &| \text{Command; Command} & (16.37\text{-d}) \\ &| \text{if Expression then Command else Command} & (16.37\text{-e}) \\ &| \text{while Expression do Command} & (16.37\text{-f}) \\ \text{Expression} &::= \text{Numeral} & (16.38\text{-a}) \\ &| \text{false} & (16.38\text{-b}) \\ &| \text{true} & (16.38\text{-c}) \\ &| \text{Identifier} & (16.38\text{-d}) \\ &| \text{Expression} + \text{Expression} & (16.38\text{-e}) \\ &| \text{Expression} < \text{Expression} & (16.38\text{-f}) \\ &| \text{not Expression} & (16.38\text{-g}) \\ &| \dots \\ \text{Declaration} &::= \text{const Identifier = Expression} & (16.39\text{-a}) \\ &| \text{var Identifier : Type_denoter} & (16.39\text{-b}) \\ \text{Type_denoter} &::= \text{bool} & (16.40\text{-a}) \\ &| \text{int} & (16.40\text{-b}) \end{aligned}$$

该语言的上下文约束是：

- 标识符 I 只能出现在声明的作用域之内。
- 赋值语句两边类型一致。
- 条件和while命令中的表达式为布尔型。

- 关系运算结果为布尔型，操作数为整型。
- 逻辑运算两操作数均为布尔型。
- 算术运算两操作数和结果值均为整型。
- 常量声明以表达式类型决定标识符类型。
- 变量和两种类型值均假定只占据一个存储单元。

16.2.2 IMP的语义域、语义函数和辅助函数

IMP中操作数的值都是第一类值，我们定义为Value：

$$\text{Value} = \text{truth_value Truth_Value} + \text{integer Integer} \quad (16.41)$$

意即Truth_Value和Integer 域中的真值和整数的不相交的联合，该域元素记为value。而可存储值只有这两者：

$$\text{Storable} = \text{Value} \quad (16.42)$$

只有第一类值和变量是可束定体：

$$\text{Bindable} = \text{value Value} + \text{variable Location} \quad (16.43)$$

执行命令的语义函数是：

$$\text{execute: Command} \rightarrow (\text{Environ} \rightarrow \text{Store} \rightarrow \text{Store}) \quad (16.44)$$

每条命令的执行是将Command域中的元素(一条命令C)映射为程序状态的改变，即某一环境env下的存储快照sto映射为另一sto'。具体可以写作：

$$\text{execute } \llbracket C \rrbracket \text{ env sto} = \text{sto}'$$

表达式求值的语义函数是：

$$\text{evaluate: Expression} \rightarrow (\text{Environ} \rightarrow \text{Store} \rightarrow \text{Value}) \quad (16.45)$$

每个表达式的求值是将Expression域中的元素E映射为从环境中取出改变了状态值，即在某一环境env的存储快照sto下求值value。可写作：

$$\text{evaluate } \llbracket E \rrbracket \text{ env sto} = \dots$$

声明确立的语义函数是：

$$\text{elaborate: Declaration} \rightarrow (\text{Environ} \rightarrow \text{Store} \rightarrow \text{Environ} \times \text{store}) \quad (16.46)$$

每个声明的确立是将Declaration域中的元素D映射为将环境env下存储快照sto变为有了新束定的环境env' × sto' 的快照。具体可写作：

```
elaborate  $\llbracket D \rrbracket$  env sto = (env', sto')
```

辅助函数有如前所述的empty_environ, find, overlay, bind, empty_store, allocate, deallocate, update, fetch。以及sum, less, not等辅助函数。此外,再增加一个取值函数:

```
coerce: Store  $\times$  Bindable  $\rightarrow$  Value (16.47)
coerce(sto, find(env, I)) (16.48)
  = val // 若I束定于val (常量) (16.48-a)
  | fetch(sto, loc) // 若I束定于loc (变量) (16.48-b)
```

16.2.3 IMP的指称语义

我们现在按16.2.1定义的抽象语法逐一为各语法短语写语义,即给出语义函数的语义等式:

```
execute  $\llbracket \text{Skip} \rrbracket$  env sto = sto
```

意即在环境env和存储sto下执行Skip的结果是不变的存储sto。

```
execute  $\llbracket I := E \rrbracket$  env sto =
  let val = evaluate E env sto in
  let variable loc = find(env, I) in
  update(sto, loc, val)
```

意即先在env sto中对E求值,并放入临时变元val中;再在env找出I束定的单元放于临时变元loc中;最后以sto, loc, val作参数调用辅助函数update。loc中的值被修改,I因而得赋值。

```
execute  $\llbracket \text{let } D \text{ in } C \rrbracket$  env sto =
  let (env', sto') = elaborate D env sto in
  execute C(overlay(env', env)) sto'
```

意即先在env sto中确立声明D。由于确立D既改变了env又改变了sto,则将它们作为序偶取出。然后在复盖了的新环境env'和存储sto'下执行C。而env'=overlay(env', env)。

```
execute  $\llbracket C1; C2 \rrbracket$  env sto =
  execute C2 env(execute C1 env sto)
```

连续执行两命令的语义是:第一命令C1在env sto下执行,不会改变束定的环境只改变存储。因此C2在改变了的存储,即括号内执行结果,中执行。

```
execute  $\llbracket \text{if } E \text{ then } C1 \text{ else } C2 \rrbracket$  env sto =
  if evaluate E env sto = truth_value true
  then execute C1 env sto
  else execute C2 env sto
```

意即在env sto下执行条件语句,首先在此env sto中对E求值,若为真值域中的true,则在同

—env sto下执行C1，否则C2。E已约定为布尔型。

```
execute [while E do C] =
  let execute_while env sto =
    if evaluate E env sto = truth_value true
    then execute_while env (execute C env sto)
    else sto
  in
  execute_while
```

意即执行while命令的语义是execute_while函数的在env sto之下执行结果。该函数是递归定义的，每次执行都会改变sto，否则，sto不变。

至此各命令短语语义等式写完，再看表达式的各个短语。

数字和布尔字面量的语义等式如下：

```
evaluate [N] env sto = integer(valuation N)
evaluate [false] env sto = truth_value false
```

true的语义等式类似。

标识符求值借助前述取值函数coerce：

```
evaluate [I] env sto = coerce(sto, find(env, I))
```

带运算符的表达式语义等式如下：

```
evaluate [E1 + E2] env sto =
  let integer int1 = evaluate E1 env sto in
  let integer int2 = evaluate E2 env sto in
  integer(sum(int1, int2))
evaluate [E1 < E2] env sto =
  let integer int1 = evaluate E1 env sto in
  let integer int2 = evaluate E2 env sto in
  truth_value (less(int1, int2))
evaluate [not E] env sto =
  let truth_value tr = evaluate E env sto in
  truth_value(not(tr))
```

其中每个临时变量int1, int2, tr 都前缀了标签(相当于说明取值的类型)。其语义一看自明。

现在再看声明的确立：

```
elaborate [const I = E ] env sto =
  let val = evaluate E env sto in
  (bind(I, value val), sto)
```

意即在env sto下作常量I的声明确立。先在env sto下对表达式E求值，放入临时值变元val，得到的是(env, sto)对偶，而env' 是通过辅助函数bind得到。因为是常量束定，sto未变。

```

elaborate  $\llbracket \text{var } I:T \rrbracket$  env sto =
  let (sto', loc) = allocate sto in
    (bind(I, variable loc), sto')

```

意即在env sto下作变量I的确立。先用辅助函数allocate在sto中分配一个单元名为loc，得到改变了的存储和该单元的对偶(sto', loc)。然后将I与loc束定返回(env', sto')新环境与存储。同样，bind中如果是变量束定(以variable标签调用)则分配的loc是未定义的，如为常量束定(以value标签调用)，则I的单元是已定义的。

IMP的语义至此写完，它只是IMP的语义规格说明，还不是实现。如编出各辅助函数的实现，则可在编译时进行语义检查。请注意，指称语义虽只看后果不看执行次序，但本语义规格说明中实际也反映了程序执行的次序，如C1; C2命令以及其它控制结构。然而，指称语义在执行结果无相关时可完全排斥次序，例如，x:=1, y:=2; 和y:=2; x:=1; 按指称语义解释是一样的，操作语义则可能不同。

本节给出指称语义的写法，并演示一个简单的小语言IMP, 这只是初步印象，还不足以用于一般的命令式语言。以下三节补充较复杂的语义描述方法。

16.3 程序抽象的语义描述

函数抽象关心的是返回值，过程抽象是要得到动作和存储的改变。当然，命令式语言由于环境和存储，它们可以有副作用。这一点正是本书一再强调的，函数抽象不能等同于数学函数。而指称语义的函数全按数学函数理解。我们看如何解决这个矛盾。

16.3.1 函数抽象

按指称语义观点，函数也是一个论域：

$$\text{Function} = \text{Argument} \rightarrow \text{Value} \quad (16.49)$$

它是将变元域中的元素映射为值域中元素。在命令式语言中要考虑到函数先访问调用它的变元的存储，但最终还是要得到值，所以有：

$$\text{Function} = \text{Argument} \rightarrow \text{Store} \rightarrow \text{Value} \quad (16.50)$$

在命令式语言中，函数体内部和一般命令式程序无异。增加的域和语义函数主要在函数调用方面。Formal_Parameter, Actual_Parameter是参数束定的两个输入域。语义函数是：

$$\text{bind_parameter}: \text{Formal_Parameter} \rightarrow (\text{Argument} \rightarrow \text{Environ}) \quad (16.51)$$

$$\text{give_argument} : \text{Actual_Parameter} \rightarrow (\text{Environ} \rightarrow \text{Argument}) \quad (16.52)$$

bind_parameter 将形参标识符束定到相应的变元上。实质上是改变了函数体求值的环境。give_argument是求出实参的值，即在环境中求出Argument域的值。与表达式求值非常相

似。

(1) 扩充IMP语法

现在我们扩充IMP使之抽象语法中包含函数和过程抽象。不过这里先假定形参只有一个常量参数(多参数见下节):

```

Command ::= ...                                // 同式 (16.37 a-f)
          | Identifier(Actual_Parameter)        (16.53)
Expression ::= ...                             // 同式 (16.38a-g)
          | Identifier(Actual_Parameter)        (16.54)
Declaration ::= ...                            // 同式 (16.39 a-b)
          | func Identifier(Formal_Parameter) is Expression (16.55-a)
          | proc Identifier(Formal_Parameter) is Command   (16.54-b)
Formal_Parameter ::= const Identifier: Type_Denoter (16.56)
Actual_Parameter ::= Expression                (16.57)

```

增加的上下文约束是: 函数和过程声明中的标识符必须是函数名或过程名, 形参和实参的类型必须相同。

前文16.2.4节中IMP的所有第一类值都可以作为变元传递, 故有:

Argument = Value (16.58)

我们约定: 值, 变量, 函数均为可束定体:

Bindable = value Value + variable Location + function Function (16.59)

(2) 写IMP函数的指称语义

参数束定的语义等式是:

$$\text{bind_parameter } \llbracket I:T \rrbracket \text{ arg} = \text{bind}(I, \text{arg})$$

束定的结果值是改变了的环境(一个形参对应一个实参变元)。

相应实参的语义等式是:

$$\text{give_argument } \llbracket E \rrbracket \text{ env} = \text{evaluate } E \text{ env}$$

注意, 其中env是实参表达式所在的环境。

函数调用的语义等式如下:

```

evaluate  $\llbracket I(AP) \rrbracket$  env =
let function func = find(env, I) in
let arg = give_argument AP env in
func arg

```

意即函数调用如同表达式求值。在环境env中对I(AP)表达式的求值是, 首先在环境env中找到标识符I, 它就是函数func。再在env中对实参表达式AP求值得到变元域中变元arg。则函数调用的语义是将func作用于arg。

函数声明的语义等式是:

```

elaborate [fun I(FP) is E ] env =
  let func arg =
    let parenv = bind_parameter FP arg in
    evaluate E(overlay(parenv, env ))
  in
  (bind(I, function func))

```

在环境env下确立函数声明。最后是要得到I与func的束定。而该函数func是对形参FP和变元arg先作束定得到新参数环境parenv，复盖了原有环境env之后，对E求值。其结果是作用于arg的函数func，最后将此函数束定于I。

以上都假定是静态束定。

16.3.2 过程抽象

过程抽象和函数抽象类似只是不返回值。

上小节扩充IMP的抽象语法定义中，过程proc是过程域的一个元素。过程域是变元域到存储域，到存储域的映射：

$$\text{Procedure} = \text{Argument} \rightarrow \text{Store} \rightarrow \text{Store} \quad (16.60)$$

第一次映射由于变元改变了存储，第二次映射由于过程操作得到最后的存储。

同样按扩充了的IMP，它的第一类值都可作为变元：

$$\text{Argument} = \text{Value}$$

相应可束定体是：

$$\text{Bindable} = \text{value Value} + \text{variable Location} + \text{function Function} + \text{procedure Procedure} \quad (16.61)$$

命令的语义函数是execute，表达式的语义函数是evaluate，声明语义函数是elaborate。各语义函数的映射域如前。形参和实参的语义函数bind_parameter 和give_argument如(16.51) (16.52)。因为过程调用是命令，故其语义等式是：

```

execute [ I(AP) ] env sto =
  let procedure proc = find(env, I) in
  let arg = give_argument AP env sto in
  proc arg sto

```

相应过程调用声明的语义等式是：

```

elaborate [proc I(FP) is C ] env sto =
  let proc arg sto' =
    let parent = bind-parameter FP arg in
    execute C(overlay (parenv env)) sto'
  in

```

(bind(I, procedure proc), sto)

其解释与上小节同，只是过程是执行命令C，不是表达式E求值。

16.3.3 参数机制的语义描述

上两小节我们为讲述方便作了最简单的假定：只有一个常量参数，变元是第一类值。更为一般，变元可以是变量，即变量参数；可以是函数或过程抽象，即函数或过程参数。参数还可以是多个。我们扩充IMP的抽象语法，并作相应语义描述。

(1) 常量和变量参数

现在还是先说只是一个参数的情况。先细化参数定义语法

Formal_Parameter ::= **const** Identifier: Type_denoter (16.62-a)

| **var** Identifier : Type_denoter (16.62-b)

Actual_Parameter ::= Expression (16.63-a)

| **var** Identifier (16.63-b)

为此，细化变元域，除第一类值外增加变量域：

Argument = value Value + variable Location (16.64)

相应的参数束定的语义函数，如下：

bind_parameter : Formal_parameter → (Argument → Environ) 同 (16.51)

give_parameter : Actual_Parameter → (Environ → Store → Argument) (16.65)

形参的语义等式是：

bind_parameter $\llbracket \text{const } I:T \rrbracket$ (value val) = bind(I, value val)

bind_parameter $\llbracket \text{var } I:T \rrbracket$ (variable loc) = bind(I, variable loc)

实参的语义等式是：

give_argument $\llbracket E \rrbracket$ env sto = value (evaluate E env sto)

give_argument $\llbracket \text{var } I \rrbracket$ env sto =

let variable loc = find (env, I) in

variable loc

其余与上小节同。

(2) 复制参数机制

采用复制机制，形参标识符是过程的局部变量，参数结合时将实参表达式 值复制给这个变量。返回时将结果值复制回实参变量。相应扩充IMP语言的抽象语法：

Formal_Parmeter ::= value Identifier: Type_denoter (16.66-a)

| result Identifier : Type_denoter (16.66-b)

Actual_Parameter ::= Expression (16.67-a)

| var Identifier (16.67-b)

增加复制入、出的两个语义函数:

copy_in: Formal_Parameter \rightarrow (Argument \rightarrow Store \rightarrow Environ \times Store) (16.68)

```
copy_in  $\llbracket$  value I:T  $\rrbracket$  (value val) sto =
  let (sto', local) = allocate sto in
    (bind(I, variable local), update (sto', local, val))
copy_in  $\llbracket$  result I:T  $\rrbracket$  (variable loc) sto =
  let (sto', local) = allocate sto in
    (bind(I, variable local), sto')
```

第一个式子是, 在sto存储下将值val复制到传值的标识符I上, 其语义是先分配一临时变量local, 将值val更新Local得新环境下的sto', 再将形参I与变量local束定, 得到更新了的环境和存储。

第二个式子是在sto'存储下, 将变量loc束定到返回结果值的形参标识符I上, 因为在sto中分配了local, 才有了新存储sto'和变量local的对偶。因为是结果参数, 没有值更新。

同样, 语义函数copy_out是:

copy_out: Formal_Parameter \rightarrow (Environ \rightarrow Argument \rightarrow Store \rightarrow Store) (16.69)

具体定义如下:

```
copy_out  $\llbracket$  value I:T  $\rrbracket$  env (vlaue val) sto = sto
copy_out  $\llbracket$  result I:T  $\rrbracket$  env (variable loc) sto =
  let variable local = find(env, I) in
    update (sto, loc, fetch(sto, local))
```

第一个式子是, 在sto存储和env环境下将值参数I和变元值结合, 当从过程返回时它不起任何作用, 拷贝出的仍是sto时的。第二个式子是在env和sto下将变量loc和结果参数I束定的值拷贝出来。其语义是先在环境env中找到I, 它就是变量Local。取出sto中的local的值, 在sto中以它更新loc即为要拷贝的结果。

最后, 过程声明的语义等式作以下修改:

```
elaborate  $\llbracket$  proc (FP) is C  $\rrbracket$  env sto =
  let proc arg sto' =
    let (parenv, sto'') copy_in FP arg sto' in
      let sto''' = execute C (overlay (parenv, env )) sto'' in
        copy_out FP parenv arg sto'''
  in
    (bind (I, procedure proc), sto)
```

在环境env和sto下确立proc(FP) is C声明。其语义是, 将变元arg在调用时刻的sto'下复制到形参FP, 从而得到参数环境parenv和改变了的存储sto''。然后在改变了的环境和sto''之下执行C, 使存储又一次改变得到sto'''。最后在参数环境和sto'''之下将变元值拷贝到形参FP。这

就是调用时刻sto'下过程对arg参数的作用。将能作这种拷贝的过程proc和过程名I束定,得到sto下的新环境。即过程调用的过程声明的语义。

(3) 多参数

程序设计语言中形、实参数的个数从零个(无参函数或过程)到多个都是允许的。我们将上述语义描述扩充到多个参数。首先修改函数和过程域为:

$$\text{Function} = \text{Argument}^* \rightarrow \text{Store} \rightarrow \text{Value} \quad (16.70)$$

$$\text{Procedure} = \text{Argument}^* \rightarrow \text{Store} \rightarrow \text{Store} \quad (16.71)$$

形参序列则用形参表,实参也一样。于是参数束定的语义等式是:

$$\text{bind_parameter} : \text{Formal_Parameter_List} \rightarrow (\text{Argument}^* \rightarrow \text{Environ}) \quad (16.72)$$

$$\text{give_argument} : \text{Actual_Parameter_List} \rightarrow (\text{Environ} \rightarrow \text{Store} \rightarrow \text{Argument}^*) \quad (16.73)$$

(4) 递归抽象

命令式语言中递归抽象是重要的。语义的递归描述我们在16.2.2的『while E do C』的语义等式中直接定义execute_while函数,把它放在if部分。执行时env是不变的,每次以执行后的存储代替原有sto。直到求值E条件为假,不再递归。

函数抽象的递归是每次将I与递归的函数func重新束定,这样得到一新环境env,在新复盖后的环境下对函数体求值。故递归函数声明的语义等式如下:

```

elaborate [ fun I(FP) is E ] env =
  let func arg =
    let env' = overlay(bind (I, function func), env) in
    let parenv = bind_parameter FP arg in
    evaluate E(overlay(parenv, env'))
  in
  bind(I, function func)

```

语义解释如16.3.1 非递归函数声明的确立。只是其中env'是每次重新束定后的环境。这样的语义等式只说明func是递归的,并未给出递归终止描述。下节还要进一步讨论递归的数学表示。

16.4 复合类型

简单变量只占据一个存储单元,用fetch和update辅助函数即可取出并更新。复合变量就没有这么简单。首先它的成分可以是不同类型,且要部分更新。而一条命令往往是更新一个成分的值。

如果不允许部分更新,我们的存储模型仍能描述复合变量,因为最小存储单元是可存储体。这样,在表达式和赋值语句中语法只要稍作改变。因为标识符既要表示复合变量,也要表示某个成分。为了分辨和操作还要增加辅助函数。

(1) 最简单的复合变量的语义描述

我们暂不考虑函数和过程抽象,回到16.2.1节的IMP抽象语法,扩充复合变量。这次只增加最简单的复合量对偶(A:T1, B:T2)。先扩充抽象语法:

$\text{Command} ::= \dots$
 $\quad | \text{V_name} := \text{Expression}$ (16.74)
 $\quad | \dots$
 $\text{Expression} ::= \dots$
 $\quad | \text{V_name}$ (16.75-a)
 $\quad | \dots$
 $\quad | (\text{Expression}, \text{Expression})$ (16.75-b)
 $\text{V_name} ::= \text{Identifier}$ (16.76-a)
 $\quad | \text{fst V_name}$ // 相当于V(1) (16.76-b)
 $\quad | \text{snd V_name}$ // 相当于V(2) (16.76-c)
 $\text{Type_denoter} ::= \text{bool} \mid \text{int}$ (16.77-a-b)
 $\quad | (\text{Type_denoter}, \text{Type_denoter})$ (16.77-c)
 对偶值本身是一个域:

$\text{Pair_Value} = \text{Value} \times \text{Value}$ (16.78)

现在还要加上对偶变量的域:

$\text{Pair_Variable} = \text{Variable} \times \text{Variable}$ (16.79)

第一类值是:

$\text{Value} = \text{truth_value Truth_Value} + \text{integer Integer} + \text{pair_value Pair_Value}$
 (16.80)

由于对偶值可部分更新，它的成分域有自己的单元。所以不是可存储值:

$\text{Storable} = \text{truth_value Truth_Value} + \text{integer Integer}$ (16.81)

变量也是一个域:

$\text{Variable} = \text{simple_variable Location} + \text{pair_variable Pair_Variable}$ (16.82)

相应取变量和更新变量的辅助函数是:

$\text{fetch_variable}: \text{Store} \times \text{Variable} \rightarrow \text{Value}$ (16.83)

$\text{update_variable}: \text{Store} \times \text{Variable} \times \text{Value} \rightarrow \text{Store}$ (16.84)

它们是通用的，全靠标签区别是哪一种变量:

$\text{fetch_variable}(\text{sto}, \text{simple_variable loc}) = \text{fetch}(\text{sto}, \text{loc})$ (16.85)

$\text{fetch_variable}(\text{sto}, \text{pair_variable (var1, var2)}) =$ (16.86)

$\text{pair_value}(\text{fetch_variable}(\text{sto}, \text{var1}), \text{fetch_variable}(\text{sto}, \text{var2}))$

$\text{update_variable}(\text{sto}, \text{simple_variable loc}, \text{stble}) =$

$\text{update}(\text{sto}, \text{loc}, \text{stble})$ (16.87)

$\text{update_variable}(\text{sto}, \text{pair_variable (var1, var2)}, \text{pair_value (val1, val2)}) =$

$\text{let sto}' = \text{update_variable}(\text{sto}, \text{var1}, \text{val1}) \text{ in}$

$$\text{update_variable } (\text{sto}', \text{var2}, \text{val2}) \quad (16.88)$$

有了这两辅助函数，我们增加识别(identify)和分配变量存储(allocate_variable)的语义函数，就足以描述对偶变量的语义了。

$$\text{identify: } V\text{-name} \rightarrow (\text{Environ} \rightarrow \text{Value_or_Variable}) \quad (16.89)$$

其中Value_or_Variable也是一个域：

$$\text{Value_or_Variable} = \text{value Value} + \text{variable Variable} \quad (16.90)$$

值或变量(名)的语义等式是：

```

identify [I] env = find(env, I)
identify [fst V ] env =
  let first (value (pair_value (val1, val2))) = value val1 |
    first (variable (pair_variable (var1, var2))) = variable var1
  in
  first (identify V env)

```

其中用到辅助函数first，它将对偶值或对偶变量映射为它的第一子域。‘snd V’的语义等式完全相似。

赋值命令的语义等式是：

```

execute [V:= E] env sto =
  let val = evaluate E env sto in
  let variable var = identify V env in
  update_variable (sto, var, val)

```

我们再看对‘值或变量(名)’的元素V_name如何求值。若V_name出现在产生式右部时，它应该是值。识别函数identify若识别为值正好；如果识别为变量，则取变量中的当前值(即递归引用)，且值可以是简单的也可以是对偶值。其语义等式为：

```

evaluate [V] env sto =
  coerce (sto, identify V env)

```

辅助函数coerce 也作相应修订：

$$\text{coerce: Store} \times \text{Value_or_Variable} \rightarrow \text{Value} \quad (16.91)$$

$$\text{coerce (sto, value val)} = \text{val} \quad (16.92\text{-a})$$

$$\text{coerce (sto, variable var)} = \text{fetch_variable (sto, var)} \quad (16.92\text{-b})$$

除识别复合变量而外，还要为其分配存储单元。我们借助类型指明符Type_denoter域到分配符Allocator域的映射来实现变量分配allocate_variable。这个语义函数是：

$$\text{allocate_variable: Type_denoter} \rightarrow \text{Allocator} \quad (16.93)$$

分配符(器)域是一个抽象的描述，它是在存储快照sto中，分配一个变量得到(sto', var)对偶：

$$\text{Allocator} = \text{Store} \rightarrow \text{Store} \times \text{Variable} \quad (16.94)$$

例如，我们为类型指明符bool分配存储的语义是：

```
allocate_variable [[bool]] sto=
  let (sto', loc) = allocate sto in
    (sto', simple_variable loc)
```

为对偶指明符分配存储的语义是：

```
allocate_variable [[(T1, T2)]] sto=
  let (sto', var1) = allocate_variable T1 sto in
  let (sto', var2) = allocate_variable T2 sto' in
  (sto', pair_variable (var1, var2))
```

有了分配变量的语义函数，我们就可以写变量声明的语义了：

```
elaborate [[var I:T]] env sto=
  let (sto', var) = allocate_variable T sto in
  (bind(I, var), sto')
```

即在环境env和存储sto中确立声明‘var I:T’。先在存储sto中按类型T分配一变量名为var，再将它与标识符I束定，得到(env', sto')的对偶。

(2) 数组变量的语义描述

数组是同类型元素的序列，如果元素也是数组则为多维数组。命令式语言数组的更新只能更新元素，所以前述IMP扩充的value_or_variable(名)域，自然可用以表达数数组元素。增加以下抽象语法：

V_name ::= Identifier (16.95-a)

| V_name [Expression] (16.95-b)

Type_denoter ::= bool (16.96-a)

| int (16.96-b)

| array Numeral of Type_denoter (16.97-c)

增加的上下文约束是：[]中的表达式必须是int型；它前面的V_name必须是数组变量名；

Type_denoter可递归定义深度不限，形成多维数组。Numeral指明一维的大小。

我们用成分值序列模型数组值，用成分变量序列模型变量，并用隐式索引序列。它的域是：

Array_Value = Value* (16.97)

Array_Variable = Variable* (16.98)

对数组变量作索引操作的辅助函数：

component : Integer × Array_Variable → Variable (16.99)

```

component (int, nil) = ⊥           // 取空序列成分只能失败           (16.100)
component (int, var.arrvar) =      // var · arrvar为数组变量序列
if int = 0 then var                // 索引为0取序列头
else component (predecessor (int), arrvar)           (16.101)

```

其中用到辅助函数predecessor是为了递归调用时取出第int 个arrvar元素的变量。
数组值可以赋值，因而也把它看作第一类值。故这个扩充的IMP其第一类值是：

```

Value = truth-value Truth_Value + integer Integer
      +array_value Array_Value           (16.102)

```

如果数组变量可个别更新，其成分只能以单独的存储单元存放，因此，统一存放的array_value不是可存储值。IMP的可存储值域仍然是：

```

Storable = truth_value Truth_Value + integer Integer           (16.103)

```

变量域既有简单变量也有数组变量：

```

Variable = simple_variable Location + array_variable Array_Variable (16.104)

```

更新和取数组元素值的两个辅助函数：

```

fetch_array: Store × Array_Variable → Array_Value           (16.105)

```

```

update_array : Store × Array_Variable × Array_Value → Store (16.106)

```

以下fetch_array (sto, arrvar)是在sto下从数组变量arrvar中取出数组值。请注意，arrvar是序列尾的变量(与LISP表尾相似)。如果序列为空则取值为空：

```

fetch_array (sto, nil) = nil           (16.107-a)

```

```

fetch_array (sto, var · arrvar) =
  fetch_variable(sto, var) · fetch_array (sto, arrvar)           (16.107-b)

```

以下update_array(sto, arrvar, arrval)是在sto下以数组值arrval更新数组变量arrvar中的值。它的做法是一个变量var一个值val递归地更新：

```

update_array (sto, nil, nil) = sto           (16.108-a)

```

```

update_array (sto, var · arrvar, val · arrval) =
  let sto' = update_variable (sto, var, val) in
  update_array (sto', arrvar, arrval)           (16.108-b)

```

有了这些辅助函数，我们进一步综合fetch和update的定义，即变量可以是单个变量，也可以是数组变量：

```

fetch_variable: Store × Variable → Value           // 取变量中的值           (16.109)

```

```

update_variable: Store × Variable × Value → Store           (16.110)

```

// 更新变量中的值得到新存储sto'

这两个辅助函数因第一变元不同有不同结果：

```

fetch_variable(sto, simple_variable loc) =          // 和一般简单变量取值无异
    fetch(sto, loc)                                (16.111-a)
fe_tch_variable(sto, array_variable arrvar) =
    array_value(fetch_array(sto, arrvar))            (16.111-b)
update_variable(sto, simple_variable loc, stble) =    // 如同一般变量更新
    update(sto, loc, stble)                          (16.112-a)
update_variable(sto, array_variable arrvar, array_value arrval) =
    update_array(sto, arrvar, arrval)                (16.112-b)

```

我们再给出数组变量的语义函数。先扩充识别函数：

identify: $V_name \rightarrow (Enviro \rightarrow Store \rightarrow Value_or_Variable)$ (16.113)

和式(16.88)定义的区别在于增加了Store。因为抽象语法中增加了下标表达式，它经Store得到值才映射为‘值或变量(名)’。语义等式如下：

```

identify [I] env sto = find(env, I)
identify [V[E]] env sto =
    let variable (array_variable arrvar) = identify V env sto in
    let integer int = evaluate E env sto in component (int, arrvar)

```

赋值命令的语义等式是：

```

execute [V := E] env sto =
    let val = evaluate E env sto in
    let variable var = identify V env sto in
    update_variable (sto, var, val)

```

变量更新即赋值语义。其余命令的语义等式不变。

当V_name出现在表达式中时用的是它的值。故用求值语义函数给出：

evaluate [V] env sto = coerce(Identify V env sto)

和上小段(1)中简单复合变量求值函数形式完全一样，只是V可以是数组变量(名)。显然coerce的两个等式也是一样的。

同样，变量声明的语义等式也相似，只是分配变量allocate_variable语义函数用到数组上有点不同：

```

allocate_variable [array N of T] sto =
    let allocate_array (sto, size) =
        let (sto', var) = allocate_variable T sto in
        if size = 1 then (sto', var • nil)
        else
            let (sto'', arrvar) =
                allocate_array(sto', predecessor (size)) in

```

```

      (sto", var • arrvar)
in
  let (sto', arrvar) = allocate_array (sto, valuation N) in
    (sto", array_variable arrvar)

```

其中allocate_array辅助函数的映射域是：

$$\text{allocate_array} : \text{Store} \times \text{Natural} \rightarrow \text{Store} \times \text{Array_Variable} \quad (16.114)$$

意思是：在sto下分配类型为T的N元数组的语义是得到存储sto"和已分配的数组变量var • arrvar，其中arrvar是在sto下对N求值后调用分配数组这个辅助函数得到的。显然，分配了数组变量，存储变为sto'。第一个let...in是辅助函数allocate_array的定义。它是递归的，每分配一个类型为T的变量则加入序列。序列头叫var以后的都叫arrvar，每加一个变量，sto多一撇，sto"仅仅是一变量符号，N元数组最后应是N-1撇。

16.5 程序失败的语义描述

数值上下溢出，数组越界都会引起程序失败。指称语义怎样描述这个“失败”呢？在纯数学里我们只能如前所述假定每个域中均有一特殊元素fail记为 \perp ，只要它作为函数变元则函数结果也为 \perp 。这个假定非常简单而且粗糙，但能描述失败及其传播。

在重要之处，程序员可以显式写出。例如，辅助函数sum求和：

$$\text{sum} : \text{Integer} \times \text{Integer} \rightarrow \text{Integer} \quad (16.115)$$

$$\begin{aligned} \text{sum}(\text{int1}, \text{int2}) = & \text{if } \text{abs}(\text{int1} + \text{int2}) \leq \text{maxint} & (16.116\text{-a}) \\ & \text{then } \text{int1} + \text{int2} \\ & \text{else } \perp \end{aligned}$$

$$\text{sum}(\perp, \text{int2}) = \perp \quad (16.116\text{-b})$$

$$\text{sum}(\text{int1}, \perp) = \perp \quad (16.116\text{-c})$$

一般情况下有了任何变元为 \perp ，函数结果必为 \perp 的约定，当int1取值为 \perp 时，sum自动为 \perp 。即我们以前写的语义等式就隐式地具有传播失败的功能。例如：

```

evaluate [E1 + E2] env sto =
  let integer int1 = evaluate E1 env sto in
  let integer int2 = evaluate E2 env sto in
  integer(sum(int1, int2))

```

当E1或E2 求值失败则传到int1或int2中使本语义等式失败。同理，程序的任何一部分失败都会传播为整个程序失败。

16.6 指称语义应用

指称语义用作定义程序设计语言的工具，它把每个程序的行为都用数学函数描述。为理解程序提供了一相对严格的基础。且无需在机器上运行即可研究程序的性质。当然颇受计算机科学家欢迎，也成为当今程序设计语言的“标准”语义定义工具。

除了设计程序设计语言时定义语义而外，借助指称语义可用数学推理证明程序的某种性质。第三种应用是作语言原型。因为完整的指称语义可以看作是语言的解释器。如果用某种语言模拟实现所有语义函数，则相当于早期考察语言行为，以节省正式开发时的开支。

本节讨论这三方面问题。首先围绕一个小语言ESMP(扩充的Pascal的小子集)介绍开发出完整指称语义(全文见附录A, B, C)的全过程。接着讨论其它两个问题。

16.6.1 指称语义用于设计语言

为一个程序设计语言写指称语义的步骤是：

- 分析(所设计的)程序设计语言的规格说明写出抽象语法。
- 定义该语言的指称域，并为这些域定义恰当的辅助函数以模型值上的操作。
- 建立语义函数。为抽象语法中的每个短语(即短语类)指定一个域(语义函数的输入域)，定义输入域到其指称域的语义函数。
- 为每一短语类写出语义等式。

以下分述之：

(1) 分析ESMP规格说明写出抽象语法

我们选用的实例是ESMP语言。附录A给出非形式化规格说明。和一般语言一样，语法按EBNF给出，语义用自然语言描述，当然很容易包括了上下文约束。

本语言较IMP扩充了记录类型、块命令、块表达式。过程参数，并提供一标准环境。用户借助它的输入/出与程序通信。

附录B给出了ESMP的抽象语法。词法到标识符、整数字面量、字符字面量、运算符等词法单元后，不再细化。程序是命令(包括过程调用)集合。

(2) 定义语义域

附录C的C.1节定义了ESMP的语义域，以及每个域上的辅助函数和相应的说明。基本值域是Truth_Value, Integer, Character(C.1.2-4)。复合类型域有数组和记录(C.1.5-6)。这五类值均为第一类值，Value是它们的不相交的联合(C.1.7)。其它类型的值有Variable(C.1.8)，Procedure(C.1.10)和Function(C.1.11)。运算符均属函数域，包括传统的算术、逻辑、关系运算、变元(C.1.9)是值、变量、过程、函数域的不相交的联合，是高层复合域。过程和函数域是从变元到存储和值域的映射。本身是动态的(因为Store域的元素是随程序执行而变的快照sto)。分配器(C.1.12)域也是动态的，在存储中分配变量从而改变了存储。

标识符成为可束定体，环境域描述了束定状态。可束定体是变量、值、过程、函数、运算符、分配器(符)的不相交联合(C.1.13)，意即标识符可束定为其中之一。请注意Allocator域，它分配一存储单元使之归于变量域，可以和标识符束定。四个辅助函数empty_environ, bind, overlay, find描述了标识符域，可束定体域和环境域之间的关系。

ESMP语言中的过程、函数、运算符、分配器非第一类值，因而不是可存储体。text域也非第一类值，但ESMP把它列入可存储体(C.1.14)。存储快照，存储单元，可存储体描述了程序的存储状态。五个辅助函数empty_store, allocate, deallocate, update, fetch描述了存取、更新值的行为。

输入/输出在IMP中未定义。一个完整的程序的语义应以其输入到输出的行为来说明。我们在式(16.13)中讲过，现再录如下：

$$\text{run: Program} \rightarrow (\text{Input} \rightarrow \text{Output})$$

其中Input和Output是输入/输出域。因为它们都是第一类值，更本质地：

$$\text{run: Program} \rightarrow (\text{Value} \rightarrow \text{Value}) \quad (16.117)$$

一般在更上一层描述输入/出，它是对正文文件的读/写操作：

$$\text{run: Program} \rightarrow (\text{Text} \rightarrow \text{Text}) \quad (16.118)$$

ESMP用字符序列模型正文Text文件，除了结束符之外，作了最大的简化。C.1.15给出了正文文件域 $\text{Text} = \text{Character}^*$ 及读、写、重写整数和字符的辅助函数。还有跳空白，跳字符，换行等总共11个辅助函数。

(3) 定义语义函数

语义函数为每种短语类指明指称域。ESMP语言的短语类是：词法短语、命令、表达式、声明、名字、参数、类型指明符、程序(附录C的C.2-C.9)。对于词法短语是求值；命令是执行；表达式也是求值；声明是确立束定。函数和过程抽象则分成三部分：抽象和体在声明中确立；抽象的调用归于命令；只有参数另给出语义函数。类型指明符的指称域为 $\text{Environ} \rightarrow \text{Allocator}$ 。‘程序’的语义函数已如前述，但为了程序运行，除给出标准环境而外，还定义了解释这些环境的10个辅助函数。从整数字符互换码到置行结束的过程。

ESMP是无副作用简单命令式语言，为了进一步理解如何定义语义函数和指明指称域，我们列出其它类型语言对照：

| 语言类别 | 短语类 | 指称域 |
|-----------|-----------------|--|
| 纯函数式语言 | 表达式 声明 | $\text{Environ} \rightarrow \text{Value}$ $\text{Environ} \rightarrow \text{Environ}$ |
| 无副作用命令式语言 | 表达式 命令 声明 | $\text{Environ} \rightarrow \text{Store} \rightarrow \text{Value}$ $\text{Environ} \rightarrow \text{Store} \rightarrow \text{Store}$ $\text{Environ} \rightarrow \text{Store} \rightarrow \text{Environ} \times \text{Store}$ |
| 有副作用命令式语言 | 表达式 命令 声明 | $\text{Environ} \rightarrow \text{Store} \rightarrow \text{Value} \times \text{Store}$ $\text{Environ} \rightarrow \text{Store} \rightarrow \text{Store}$ $\text{Environ} \rightarrow \text{Store} \rightarrow \text{Environ} \times \text{Store}$ |

表16-1 不同类型语言短语类的指称

函数型语言与存储状态无关。表达式求值总是在某些变量束定的环境之下，它只受嵌套声明的环境的影响。命令式语言部必须与存储状态(用快照集Store描述)有关。声明的确立是在原有环境下新分配了变量，改变了存储得到新的环境和存储。改变存储总是会有副作用的。所谓无副作用语言是，语义指称和实现上都不计及存储改变的副作用。例如：

```

evaluate  $\llbracket E1 + E2 \rrbracket$  env sto =
  let integer int1=evaluate E1 env sto in
  let integer int2=evaluate E2 env sto in
  integer (sum (int1, int2))

```

是无副作用的，而

```

evaluate  $\llbracket E1 + E2 \rrbracket$  env sto =
  let (int1, sto') = evaluate E1 env sto in
  let (int2, sto'') = evaluate E2 env sto' in
  (integer som (int1, int2), sto'')

```

是有副作用的。它通过存储状态的改变影响后面的计算。请注意，表达式求值一般认为是原子的。若某个语言允许子表达式的副作用，则读者可以按此分解子表达式并描述它们的影响。有副作用的表达式求值，不仅得到值，还有存储改变(所以是笛卡尔积)。至于命令本来就是改变存储的，纯函数式语言就没有命令。命令式语言由于有存储状态要分配存储，才有分配器域 Allocator。声明的确立其指称是 $\text{Environ} \rightarrow \text{Allocator}$ ，代入 Allocator 域的等式即为上表的域指称。

另外还要谈一谈函数和过程抽象的指称。对于纯函数式语言，函数抽象体是表达式。表达式执行是 $\text{Environ} \rightarrow \text{Value}$ 。这个环境是定义函数体的环境加上静态束定时，形实参结合增加的束定对原有环境的复盖。整个抽象的指称域是 $\text{Argument}^* \rightarrow \text{Value}$ 。然而，对命令式语言的函数都要计及存储的影响，函数体是 $\text{Environ} \rightarrow \text{Store} \rightarrow \text{Value}$ 。这个环境也是函数体定义时的环境加上静态束定变元对该环境的复盖，以及改变了的存储 $\text{env}' \text{ sto}'$ 。在这个状态下求值。同样，整个抽象域的指称是 $\text{Argument} \rightarrow \text{Store} \rightarrow \text{Value}$ 。类似地，有副作用命令式语言的抽象，不仅求值也改变存储。以下是它们差异的汇总。

| 语言类别 | 抽象型式 | 抽象体指称 | 抽象域 |
|-----------|-----------------------|--|--|
| 纯函数式语言 | Function | $\text{Environ} \rightarrow \text{Value}$ | $\text{Argument}^* \rightarrow \text{Value}$ |
| 无副作用命令式语言 | Function Procedure | $\text{Environ} \rightarrow \text{store} \rightarrow \text{Value}$ $\text{Environ} \rightarrow \text{Store} \rightarrow \text{Store}$ | $\text{Argument}^* \rightarrow \text{Store} \rightarrow \text{Value}$ $\text{Argument}^* \rightarrow \text{Store} \rightarrow \text{Store}$ |
| 有副作用命令式语言 | Function Procedure | $\text{Environ} \rightarrow \text{Store} \rightarrow \text{Value} \times \text{Store}$ $\text{Environ} \rightarrow \text{Store} \rightarrow \text{Store}$ | $\text{Argument}^* \rightarrow \text{Store} \rightarrow \text{Value} \times \text{Store}$ $\text{Argument}^* \rightarrow \text{Store} \rightarrow \text{Store}$ |

表16-2 静态束定语言的抽象域

(4) 写出语义等式

ESMP语言的语义等式在附录C的C.2-C.9中给出。语义函数的指称，按给出的非形式语义并利用域中定义的辅助函数(当然，还需要增加一些辅助函数：一些取自于语言环境，如C.9中的 get, append, code, decode, ship-blanks, skip_line等。另一些是为语义清晰作的补充定

义，如C.2的integer_valuation IL, id 0, character_valuation CL等)写出语义等式。

写语义等式时要注意两个问题。一是迭代一般用递归函数实现。如while命令的例子。另一是注意顺序，按书写的先后理解，所以如前所述的E1+E2表达式求值，先求值的必然对以后计算有影响，除非显式写明原有环境和存储状态env, sto。

在阅读附录C的形式语义时，要注意标签和函数名，函数名有时带括号有时如同 λ 演算中的函数fab = f(a)b = g(b)。

16.6.2 指称语义用于程序性质研究

给出程序的数学含义我们就可以脱离机器的执行研究程序的某些性质。这里只举两个方面的例子。

(1) 上下文约束的静态描述

以上我们用到第一类值Value，和程序状态(Store \times Environ)，面向程序动态执行描述程序的语义。也叫动态语义。然而，有一些研究不依赖执行，例如短语是否良定义？类型、程序是否等价。这些研究是利用指称语义的方法作静态语义的研究。

在程序设计语言的文法产生的所有句子之中只有一部分是良定义的。例如，早期FORTRAN的标识符不多于六个字母数字字符。如果只在第7, 8个字符上有差异的标识符就认为是一样的，也就是非良定义。语法往往不能给出明确的表示，要依靠上下文约束。

用指称语义的方法描述程序设计语言的上下文约束要建立类型环境的概念。类型环境是类型的束定集。即语义函数将语法类映射到类型域上，而不是该类型的值或变量域上。语言中各类型之总称即为Type域。例如，在前述IMP语言中类型域是：

$$\text{Type} = \text{truth_type} + \text{integer_type} + \text{var_type} + \text{error_type} \quad (16.119)$$

类型环境：

$$\text{Type_Environ} = \text{Identifier} \rightarrow (\text{bound Type} + \text{unbound}) \quad (16.120)$$

它们和动态语义非常相似，读者不难写出所需域。有了指称域，再给出辅助函数：

$$\text{equivalent: Type} \times \text{Type} \rightarrow \text{Truth_Value} \quad (16.121)$$

可测试两种类型是否等价。

$$\text{constrain: Command} \rightarrow (\text{Type_Environ} \rightarrow \text{Truth_Value}) \quad (16.122)$$

检查命令在类型环境中是否遵从约束，即是否良定义的。

$$\text{typify: Expression} \rightarrow (\text{Type_Environ} \rightarrow \text{Value_Type}) \quad (16.123)$$

验明表达式的类型，即在类型环境中的具体类型。

$$\text{declare: Declaration} \rightarrow (\text{Type_Environ} \rightarrow \text{Truth_Value} \times \text{Type_Environ}) \quad (16.124)$$

在类型环境中给出声明是良定义的真值，以及所产生的类型束定。

```
type_denoted_by: Type_Denoter → Value_Type
```

产生类型指明符的真实类型。类型环境域有以下辅助函数：

```
empty_environ : Type_Environ (16.126)
```

```
bind : Identifier × Type → Type_Environ (16.127)
```

```
overlay: Type_Environ × Type_Environ → Type_Environ (16.128)
```

```
find: Type_Environ × Identifier → Type (16.129)
```

我们现在给出语义等式说明这些函数的用法。先说声明：

```
declare [const l = E] typenv =
  let typ = typify E typenv in           // E求值结果类型
  if equivalent (typ, error_type)
  then (false, empty_environ)
  else (true, bind(I. typ))              // I也是typ类型
```

```
declare [var I:T] typenv =
  let typ = type_denoted_by T in
  (true, bind (I, var typ))              // I已束定于typ型
type_denoted_by [bool] = truth_type
type_denoted_by [int] = integer_type
```

再说命令。用语义函数constrain：

```
constrain [Skip] typenv = true
```

在任何类型环境下skip都是良定义的。

```
constrain [I := E] typenv =
  let typ = find (typenv, I) in
  let typ' = typify E typenv in
  equivalent (typ, var typ')
```

在typenv之下，若本赋值命令是良定义的，当且仅当，在环境typenv下，I束定于typ，且E在typenv下求值为typ'，两个类型是等价的。

```
constrain [let D in C] typenv =
  let (tval, typenv') = declare D typenv in
  if ok
  then constrain C(overlay (typenv', typenv))
  else false
```

在typenv之下若本块命令是良定义的，当且仅当，在typenv中声明D得新类型环境，且C的类型环境限定为它们复盖后的环境。否则非良定义。

```

constrain  $\llbracket C1; C2 \rrbracket$  typenv =
  constrain C1 typenv  $\wedge$  constrain C2 typenv

```

在typenv下若 $C1; C2$ 是良定义的，当且仅当， $C1$ 和 $C2$ 在typenv之下都是良定义的。

```

constrain  $\llbracket \text{if } E \text{ then } C1 \text{ else } C2 \rrbracket$  typenv =
  equivalent (typify E typenv, truth_type)
 $\wedge$  constrain C1 typenv  $\wedge$  constrain C2 typenv

```

在typenv下，若本条件命令是良定义的，当且仅当 E 为真值类型，且 $C1, C2$ 都是良定义的。

```

constrain  $\llbracket \text{while } E \text{ do } C \rrbracket$  typenv =
  equivalent (typify E typenv, truth_type)  $\wedge$  constrain C typenv

```

在typenv下，若本while命令是良定义的，当且仅当 E 为真值类型，且 C 为良定义的。
再说表达式。用语义函数typify验明类型：

```

typify  $\llbracket N \rrbracket$  typenv = integer_type

```

在任何类型环境下 N 都是整类型。

```

typify  $\llbracket I \rrbracket$  typenv =
  let value_type_of (truth_type) = truth_type
    value_type_of(integer_type)=integer_type
    value_type_of(vartyp) = typ
    value_type_of (error_type) = error_type
  in
    value_type_of (find (typenv, I))

```

由单标识符构成的表达式若为良定义的，当且仅当它在typenv下束定于某个类型。

```

typity  $\llbracket E + E2 \rrbracket$  typenv =
  if equivalent (typity E1 typenv, integer_type)
     $\wedge$  equivalent (typify E2 typenv, integer_type)
  then integer_type
  else error_type

```

在typenv之下若表达式 $E1+E2$ 是良表达定义的，当且仅当 $E1, E2$ 均为整类型。

综上所述。用指称语义方法说明程序语言的上下文约束。如果写出模拟执行以上等式的解释器。则程序中任何非良定义结构均可查出。以后只为良构的短语类写动态语义。

(2) 程序推理

利用语法类的指称语义等式，可以推导出两语法类的语义是等价的。例如，我们一看即可知， $C; \text{Skip} \equiv C$ 。要证明相等，即指出两端指称一样即可：

```

execute  $\llbracket C; \text{Skip} \rrbracket$  env sto

```

```

= execute [[Skip]] env (execute C env sto)
= execute C env sto

```

从我们在16.1.4(3)中导出的 $\llbracket E1; E2 \rrbracket$ ， $\llbracket \text{Skip} \rrbracket$ 的语义等式可知以上推导是正确的。再看一例。我们证明：

```

while E do C  $\equiv$  if E then
    begin C; while E do C end
else Skip

```

通过证明两端指称相等证明它们恒等。设语句括号有以下解释：

```
execute [[begin C end]] = execute C
```

即知道如何执行就可以成对取消。

```

execute [[if E then
    begin C; while E do C end
    else Skip]] env sto
= if evaluate E env sto = truth_value true
  then execute [[C; while E do C]] env sto
  else [[Skip]] env sto
= if evaluate E env sto = truth_value true
  then execute [[while E do C]] env (execute C env sto)
  else sto
= execute [[while E do C]] env sto

```

对照16.1.4 (3)中 $\llbracket \text{while } E \text{ do } C \rrbracket$ ， $\llbracket C1; C2 \rrbracket$ 和 $\llbracket \text{Skip} \rrbracket$ 的语义等式推导，即可证实上述推导正确。

16.6.3 语义原型

正是因为指称语义是形式化的数学语义，如果有一函数式语言，我们可以直接把它翻译为该语言的程序。运行这个程序我们就可以得到该语言的语义解释。这个程序就是快速开发(严格说抽象语法树)的解释器。反复改进，反复运行即可导致完善的目标语言设计。这和软件工程中原型开发是一样的，我们叫它语义原型。

本书定义语义函数，辅助函数，数组数据时，有意按ML语言的函数风格。无非是便于直接对应为ML程序，也正因为函数式语言无副作用，不会使语义运行结果复杂化。以下给出IMP语言(16.1.4节给出的抽象语法和指称语义)改写为ML的示例。其步骤如下：

(1) 先将抽象语法改写为ML的datatype 定义：

```

type Identifier = string
  and Num eral = string;
datatype Command =
  Skip
  | IbcomesE of Identifier * Expression

```

```

| letDinC of Declaraton * Command
| CsemicolonC of Command * Command
| ifEthenCelseC of Expressiion * Command * Command
| whileEdoC of Expression * Command
and Expression =
  num of Numeral
| flase'
| true'
| ide of Identifier
| EplusE of Expression * Expression
and Declaration=
  constIIsE of Ldentifier * Expression
| varIcolonT of Ldentifier* Typerdenoter
and Typedenoter=
  bool'
| int'

```

按ML把Command定义为类型，其值域可以是竖杠分割的任何一个。而每个实则是抽象语法树上的短语类，如IbecomeE 即 $I:=E$ ；CsemicolonC 即 $C; C$ ；whileEdoC即while E do C。同样，Expression 类型的值域也用竖杠分割，false' 是ML预定义值；EplusE即 $E + E$ ；还可写 ElessthenE即 $E < E$ 。Declaration类型也一样，constIIsE 即 $\text{const } I = E$ ；VarIcolonT即 $\text{var } I:T$ 。这些连写的短语符号是值域的名字，具体的值则按以下形式：

```

Skip
IbecomesE(I, E)                // IbecomesE域上的对偶(I, E)值
CsemicolonC( C1, C2)
num " 365"                      // num域上的值 365
ide "x"
EplusE(E1, E2)

```

(2) 将域的各等式也转成ML的datatype定义：

```

type Location = int;
datatype Value=
  truthvalue of bool
  | integer of int;
type Stroeable = Value;
datatype Bindable =
  value of Value
  | variable of Location;

```

(3) 现在可以定义语义函数(16.44)–(16.46)了。

按ML定义函数类型的写法：

```

execute : Command → (Environ→Store→Store)

evaluate : Expression→ (Environ→Store→Value)

```

elaborate : Declaration \rightarrow (Environ \rightarrow Store \rightarrow Environ * Store)

再写出具体函数定义:

```

fun
  execute(Skip) env sto =
    sto
  | execute(IbceomesE(I, E)) env sto =
    let val val' = evaluate E env sto in
      let val variable loc = find (env, I) in
        update(sto, loc, val')
      end
    end
  | execute(letDinC (D, C)) env sto =
    let val (env', sto') =
      elaborate D env sto in
      execute C (overlay (env', env)) sto'
    end
  | execute (CsemicolonC (C1, C2)) env sto =
    execute C2 env (execute C1 env sto)
  | execute(if E then C else C (E, C1, C2)) env sto =
    if evaluate E env sto = truthvalue true
    then execute C1 env sto
    else execute C2 env sto
  | execute (whileEdoC(E, C))=
    let fun executewhile env sto =
      if evaluate E env sto = truthvalue true
      then executewhile env (execute C env sto )
      else sto
    in
      executewhile
    end
and
  evaluate (num N) env sto = integer (valuation N)
  | evaluate (false') env sto = truthvalue false
  | evaluate (true') env sto = truthvalue true
  | evaluate (ide I) env sto = coerce(sto, find (env, I ))
  | evaluate (EplusE( E1,E2 )) env sto =
    let val integer int1 = evaluate E1 env sto in
      let val integer int2 = evaluate E2 env sto in
        integer (sum( int1, int2 ) )
      end
    end
  | ...
and

```

```

elaborate(constIIsE ( I, E )) env sto=
  let val val'=evaluate E env sto in
    (bind(I, value val'), sto)
  end
| elaborate (varIcolonT( I, T )) env sto=
  let val (sto', loc)=allocate sto in
    (bind( I, variable loc ), sto')
  end
and
valuation(N) =
  integer(stringtoint N)

```

以上按IMP 抽象语法套写语义函数execute, evaluate, elaborate. 还要把辅助函数改写为ML:

```

fun
  coerce(sto, value val')= val'
  | coerce( sto, variable loc ) = fetch( sto, loc )

```

此外, 再定义Environ, Store类型和相应的辅助函数: emptyenviron, bind, overlay, find, empty-store, allocate, deallocate, update, fetch. 这些留作习题。

(4) 设置初始条件运行ML程序

有了以上定义, 即可运行抽象语法树的ML解释器, 例如:

```

val env0=...;           //初始环境
val sto0=...;           //初始存储
val prog=...;           //一条IMP命令的抽象语法树
execute prog env0 sto0;

```

这将得到更新了的存储, 即执行一条IMP命令的结果。

请注意, 这个程序运行效率是极低的, 空间耗费也大, 只是做小语言原型时试用。

16.7 小结

- 指称语义的基本原理是每个程序短语通过语义函数可映射为某数学域上的值。这些值即该短语的指称。语义函数变元是程序短语。借助辅助函数和已定义过的语义函数定义语义函数, 叫语义方程。

- 无论是语义的指称物还是辅助函数的变元和结果值都是论域(简称域)。计算机中和常规类型域对应的基本域有字符、整数、自然数、真值... 用户定义的枚举域, 以及以这些基本域复合的复合域: 笛卡儿积域、不相交联合域、函数域、序列域。

- 在程序语言的研究之中, 用函数映射描述和表达域。若D域中每个元素均可映射为D' 域上的元素称全函数, 否则为偏函数。程序有停机问题, 每个域都增加了fail(记为 \perp)元素, 故其上描述均为偏函数。

- 按命令式语言固有特性, 指称语义中将存储快照集合模型为Store(存储)域, 将所有束定集合模型为Environ(环境)域。以分配、未定义、未使用和已束定、未束定刻画程序变量的状

态。

- 一般程序设计语言指称语义开发步骤是：

[1]. 写出目标语言的抽象语法和上下文约束。

[2]. 定义语义域：所有抽象语法树的子树根结点均为输入域；指称域按语言给出。对命令式语言有值(包括基本值和复合值)、可存储体、存储、环境、变量(存储单元)域。

[3] 定义每一域上的辅助函数。

[4] 以包含辅助函数的语义等式定义语义函数。为输入域中每个语法短语写出语义方程，并兼顾上下文约束的规定。

- 程序抽象本身是函数域：从变元映射为存储或值。用束定并给出参数的语义函数刻画抽象程序的调用。本章给出复制参数机制的语义函数，并给出由单参数扩大到多个参数的递归抽象描述。

- 本章从对偶变量扩充到数组变量，演示变量或值域如何通过识别和成分语义函数作复合变量的部分更新。

- 指称语义可表达程序的失败和程序故障传播。

- 指称语义的应用：

[1] 程序设计语言的设计工具，为理解程序提供一相对严格的基础。无需上机即可考察程序的行为。

[2] 借助指称语义的数学性质，可作程序推理。证明程序的某些性质。

[3] 借助指称语义的严格形式表示，翻译为相近的函数式语言，可作语义原型。为渐近式开发程序设计语言提供快速上机验证的手段。

- 写语义方程时注意：

一般循环、迭代用递归函数表达。

语义等式本身虽没有作用域概念，但书写顺序如同一般程序设计语言，有隐含的作用域效应。

- 程序设计语言是规定语法标记的组合形式，按照这个形式组合的短语只有一部分是良定义的，上下文约束是判定短语是否良定义的规则。指称语义应反映这些约束。

- 语义原型就是该程序设计语言的解释器，但一般效率较低，只能为研究小语言或语言某些特征而用。

习题

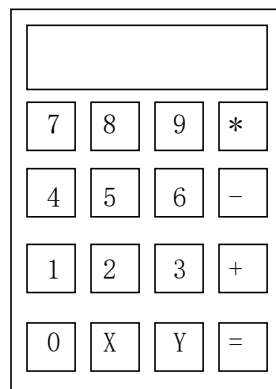
16.1 如果二进制数的语法不用(16.1)而用下式:

```
Number1a ::= 0
          | Numeral Numeral
```

虽然它依然可得到二进制数,但其语义(即术值)写法不同,试写出它的语义函数 valuation 的语义方程。

16.2 写出十进制数Decimal的抽象语法及语义等式。

16.3 右图示一简单计算器,它带有两个寄存器X, Y (可以存放多位数的结果值,掀按一下即可得到或显示寄存器中的值)。试写出它的抽象语法和指称语义。



16.4 按习题16.3 做出的结果执行命令9=X; X+7=Y; Y*Y。验证由语义等式推演出的值是否正确。

16.5 设正文文件是字符行的序列,有一正文编辑器可完成以下功能:将输入文件读入正文缓冲器;在正文缓冲器中按以下命令增、删、改该文件。然后将更新了的正文写入输出文件。用户可用命令如下(N, S代表数和字符串):

- mN 把当前处理移到第N行。
- m"S" 把当前处理移到有S子串的下一行。
- s"S1"S2" 在当前行以S2替换S1。
- d 删除当前行,当前处理移到下一行。
- i"S" 在当前行之前一行插入S。

试写出该正文编辑器的抽象语法和指称语义。

[提示]抽象语法:

```
Script  ::= Command
Command ::= m Numeral
          | m "String"
          | s" String" String"
          | d
          | i "String"
          | Command Command
```

语义域和辅助函数:

```
String = Character*
Text = String*
substitute : String × String ×String →String
Location  = 1, 2, 3, ...
Buffer = Location → (stored String + unused)
```

```

update: Buffer × Location × String → Buffer
fetch : Buffer × Location → String
shift-up : Buffer × Location → Buffer
shift-down : Buffer × Location → Buffer
stroe : Text → Buffer
unstore : Buffer → text
next-location: String × Buffer × Location → Location

```

注: Buffer相当于Store, 以location作行号, buffer 若包含n行, 则有1, 2, …n个location, 其余未使用。update和fetch相当于存储函数。

语义函数:

```

edit : Script → (Text → Text)
execute : Command → (Buffer × Location → Buffer × Location)

```

16.6 顺序声明是D1; D2, 并行声明是D1, D2。试写出它们的对比的指称语义。

16.7 将16.1.4 中定义的IMP语言增加以下控制结构, 并写出相应的语义方程。

[1] case E of N1: C1; ... Nn: Cn end 。按整表达式E求值与Ni匹配 则执行Ci子句, 执行完跳出。

[2] repeat C until E. 先执行C然后对E求值, 若E为false重复, 否则跳出。

[3] for I in E1.. E2 do C. 根据E1和E2求出的整数值, I在其范围内每个整数执行一次C。若E1..E2为空(零或负)则C不执行。

16.8 从具有函数声明的IMP语言出发。扩充该语言使之有动态束定, 即函数体在函数调用的环境中求值, 这个环境可以看作是增加了变元的环境, 则函数域是:

Function = Environ → Argument → Value

试写出函数声明和函数调用的语义等式。

16.9 扩充具有程序抽象的IMP语言, 其参数机制为引用机制。写出引用参数机制的语义。

16.10 扩充IMP语言使之具有表达同构表的能力, 提供对表的组合和分解操作: 表聚集、连接(连接符为@)、取表头(hd)、选表尾(tl), 表变量只能全部更新。抽象语法增补如下:

```

Expression ::= ....
              | nil // 空表
              | [List_Aggregate]
              | Expression @ Expression
              | hd Expression
              | tl Expression
List_Aggregate ::= Expression
                  | Expression, List_Aggregate
Type_denotes ::= ....
                  | list of Type_denoter

```

相应增补域:

List = Value*

写出增补的语义等式。

16.11 有表达式 E1 + E2 写出三种语义等式:

[1] E1, E2求值先后不影响求值结果。

[2] E1, E2均有副作用, 自左至右求值。

[3] 同(2), 自右到左求值。

16.12 表达式语言是命令和表达式不加区分的语言(如Algol-68, ML, C)。将16.1.4 节给出的IMP语言改为表达式语言, 并写出它的指称语义。

显然, 我们要以赋值表达式I:=E代替赋值命令。其语义是得到存入I中的表达式E的值。以顺序表达式E1; E2代替顺序命令C1; C2。其语义是整个顺序表达式要得到的值。先

对E1求值，但只是副作用部分有效。E2在其副作用影响下的求值即为整个顺序表达式的值。

在声明中以变量定义`var Identifier:=Expression`代替变量声明`var Identifier: Type_denoter`。其语义是分配并以表达式`Expression`初始化一变量，再束定于标识符`Identifier`。

16.13 试用指称语义证明IMP语言的命令：

`x:=1; y:=2 和 y:=2; x:=1`

是等价的。即证明，当`loc ≠ loc'`时，有

`update (update(sto, loc, val), loc', val')=`

`update (update (sto, loc', val'), loc, val)`

16.14 试用指称语义证明：

`repeat C until E 和 C; while not E do C`是等价的。

16.15 用ML语言语法定义16.5.3 (3)节中语义域`Environ`，`Store`和相应的辅助函数。并运行IMP语义原型的ML程序。

16.16 扩充ESMP 语言使之具有程序包机制，程序包实质是一环境，增加的抽象语法是：

`Declaration ::=`

`| package Identifier is`

`Declaration where Declaration end`

//前一声明为输出部分后一声明为包内可见。

`Name:: = Identifier`

`| Name.Identifier`

增加的语义域：

`Package = Environ`

`Bindable = ... + package Package`

增加的语义函数：

`elaborate : Declaration → (Environ → Store → Environ × Store)`

`identify : Name → (Environ → Bindable)`

写出程序包声明，识别标识符的语义等式。

第 17 章 代数语义学

指称语义学用数学函数的指称或描述程序的行为和状态。它是面向值(域)的。以最终值刻划程序的效应。用语义函数得到各种域上值来刻划程序的行为。当然, 这些效应和行为都是在既定模型上的值。代数语义学是用代数的方法来处理满足一计算逻辑的各种模型。把模型的集合看作是代数结构。代数语义学的公理规定了算子的组合规则和约束。算子集和域上值集的关系正好是代数系统研究的范畴。正是由于域上值都可以用算子生成, 算子集及其约束就成了论域的语法。同样, 域上值反映了语义。因此, 代数规格说明成为语法、语义一体化描述的形式基础。在程序语言的类型检查、类型多态、抽象数据类型、正确性证明、面向对象中得到广泛应用。

本章讨论语义学的代数途径, 17.1 节先复习抽象代数中的基本理论继而介绍代数和字代数、商代数。17.2 节介绍以代数模型与代数规格说明中的问题, 17.3 节是类型的代数规格说明的写法, 17.4 给出 λ 演算的代数规格说明实例。

17.1 代数基础

代数学是研究数的运算规则, 和数学符号在满足这些规则中的结构。计算机中的数据天然具有代数性质; 离散数据对应的符号运算。所以, 代数是精确描述离散数据和类型的数学工具。

17.1.1

我们先复习《抽象代数》中的基本概念和定义:

定义 17.1(代数)

代数是形如 (A, OP) 的对偶, 其中 A 是承载子(carrier)集合, OP 代表了操作符的有限集。

其中每个 OP_i 是以 A 的元素操作数并返回某个 A 的元素的算子(操作符), 即

$$OP_i(a_1, a_2, \dots, a_n) = a_s: \quad A \rightarrow A(a_i, a_s \in A, i = 1 \dots n)$$

其中 OP_i 的变元 $a_1 \dots a_n$ 和 a_s 都是承载子集合 A 中的离散数据。可以在不同的抽象层次上研究代数, 我们给出:

$(\{\text{true}, \text{false}\}, \{\wedge, \vee, \neg\})$ //布尔代数

$(N, \{+, *\})$ //整数代数

$(S, \{\text{gcd}, \text{lcm}\})$ //S-代数

都是具体代数, 即指定具体的值集和操作集, 第一行是布尔数据类型的完整模型。第二行, N 是整数集, 所指定的操作是说明只研究整数的加、乘运算的代数。第三行 S 是整数或实数集, gcd 是求最大公约数, lcm 求最小公倍数。我们就叫它 S -代数, 在 S 集上研究这两种运算的代数。

我们不能随意指定操作如:

$(N, \{+, <\})$

就不是一个代数, 因为 ' $<$ ' 操作, 当有两个 N 的元素比较时结果值是真假值, 超出了 N 的范围。反过来, 我们倒可以把操作集看作承载子集的构造子, 它按照算子 OP_i 规定的规则生成。

抽象代数从更高的层次上研究构造子和承载子之间的关系, 它不规定具体的值集和操作集, 只给出一抽象的 A 集合和(组合)算子 $\{o\}$, 以及在构造中某些必需满足的公理、定理。《抽象代数》中对构造子不同的约定(即应满足的性质)得到不同的抽象代数:

群: $(A, \{o\})$ // o 不满足任何定理

半群: $(A, \{o\})$ // o 必需满足结合律: $x o (y o z) = (x o y) o z$

独异半群: $(A, \{o, i\})$ //其中 $(A, \{o\})$ 是一个半群, i 是恒等操作(函数)

独导半群满足恒等定律:

$$x \circ (i(a)) = x = (i(a)) \circ x$$

$i(a)$ 为 A 的单位元。若 o 是 $+$, A 是整数集, $i(a) = 0$ 。同样若 o 是 $*$, $i(a) = 1$ 。单位元是相对 o 而言的。

每一群 $(A, \{o, i\})$ 中都有一逆操作 i^{-1} 的独异 $(A, \{o, i^{-1}\})$ 满足逆定律:

$$x \circ i^{-1} = i(a) = i^{-1} \circ x$$

在具体代数中我们研究数据对象和操作的代数性质, 并以此作为程序语言的模型。在抽象代数中, 我们证明这些代数的存在性。这样才可避免模型的深层错误。只有抽象代数是不矛盾的, 它的实例即具体代数才是可靠的。因此, 在处理复杂的软件之中, 抽象代数简化了复杂系统的处理。因为它只注重操作的性质, 也就是计算的语义。

更为抽象的是泛代数(universal algebra), 它把具体代数看作是具有某种操作性质的“对象”去研究各“对象”的“关系”。这些“关系”被抽象为态射(morphism)。通过态射可以知道两代数是否同态、同构、和等价。程序是数据(对象)集和该数据集上的操作集构成, 从代数的意义上写一个程序就是构造一具体代数。抽象的语法就是抽象的字代数, 将它态射到等价的有“值”的代数上就得到它的语义解释。

定义 17-2(子代数)

设 (A, OP) 是一个代数, (B, OP) 也是一代数且 $\subset(A, OP)$, 则称 (B, OP) 是 (A, OP) 的子代数, 写为 $(B, OP) \leq (A, OP)$ 。

定义 17-3 (范畴 category)

范畴 C 是 $(ob(C), morph(C))$ 的二元组。其中 $ob(C)$ 为集合对象 X, Y, Z, \dots 等的象元集合, $morph(C)$ 为 $C(X, Y), C(Y, Z), C(X, Z), \dots$ 组成的态射集合。 $C(X, Y)$ 为 X 到 Y 的态射(morphism)集合, 也可以写作 $f: X \rightarrow Y, f \in C(X, Y)$ 。 X 为态射函子 f (function) 的域(domain), Y 为 f 的协域(codomain)。公理保证这种映射总是有效。

对于每个态射函数的对偶 (f, g) , 若一态射函数的域是另一态射函数的协域, 即 $f: X \rightarrow Y; g: Y \rightarrow Z$, 则可利用组合算子 \circ 形成新的态射 $f \circ g: X \rightarrow Z$ 。组合算子服从结合律。若 $f: X \rightarrow Y, g: Y \rightarrow Z, h: Z \rightarrow W$, 则有:

$$(h \circ g) \circ f = h \circ (g \circ f): X \rightarrow W$$

对于范畴每一对象 X 均存在着恒等(identity)态射 $id_x: X \rightarrow X$ 。因此, 对任何态射有:

$$id_x \circ f: (X \rightarrow X) \circ (X \rightarrow Y) = X \rightarrow Y: f$$

$$g \circ id_y: (Y \rightarrow Z) \circ (Y \rightarrow Y) = Y \rightarrow Z: g$$

id_x 即单位态射, 很容易证明 id_x 是唯一的。单位态射在抽象代数的证明体系中, 十分重要。

态射表达两代数的结构相似性。

定义 17-4 (单射, 满射, 双射)

若有态射函子 $f: A \rightarrow B$, 对于任意两对象 $a_1, a_2 \in A$, 且 $a_1 \neq a_2$, 都有 $f(a_1) \neq f(a_2)$, $(f(a_1), f(a_2) \in B)$, 则 f 称为单射(injective)函子。

对于任意 $b \in B$ 都可以找到一个 $a \in A$, 使得 $b = f(a)$, 则 f 称为满射(surjective) 函子。

若 $f: A \rightarrow B$ 的 f 既是单射又是满射, 则 f 是可逆的, 即存在 $f^{-1}: B \rightarrow A$ 。 f 称为双射(bijjective) 函子。

定义 17-5(同态映射 homomorphism)

若态射函子 $f: A \rightarrow B$ 是从代数 (A, OP) 到 (B, OP') 的映射。如果对任意 $op \in OP$, $a_1, a_2, \dots, a_n \in A$ 有:

$$f(op(a_1, a_2, \dots, a_n)) = op'(f(a_1), f(a_2), \dots, f(a_n)) \quad (17-1)$$

其中 $op' \in OP'$, $f(a_1), f(a_2), \dots, f(a_n) \in B$, $n = 1 \dots k$ 。意即代数 A 中某 k 目操作 op , 若将其 k 个变元先映射到代数 B 中, 总可以找到同目的操作 op' , 以映射后的变元作变元, 其结果和 op 运算后再映射的结果一致。 (A, OP) , (B, OP') 是同态的。

同理。若 $f: A \rightarrow B$ 中 f 是单射的且满足(17-1), 则称单同态(monomorphism)。

若 f 是满射的且满足式(17-1), 则称满同态(epimorphism)。

若 f 是双射的且满足式(17-1), 则称同构(isomorphism)。即 f 既是单同态也是满同态。当然也存在 $f^{-1}: B \rightarrow A$, f^{-1} 也是同构映射。

同态保持两代数结构的相似性, 同构即两代数结构相等, 仅管其中值集不相同。

例 17-1 代数间的同态, 同构映射

BOOLEAN = ({true, false}, not)

not(true) = false

not(false) = true

A = ({0,1}, flip)

flip(0) = 1

flip(1) = 0

B = ({yes,no, maybe}, change)

change(yes) = no

change(no) = yes

change(maybe) = maybe

C = ({any}, same)

same(any) = any

若有态射函子 $h: \text{BOOLEAN} \rightarrow A$ 。具体定义是:

$h(\text{true}) = 1, h(\text{false}) = 0$

我们验证(17-1)式, 先看右侧:

$h(\text{not}(\text{true})) = h(\text{false}) = 0$

再看右侧:

$\text{flip}(h(\text{true})) = \text{flip}(1) = 0$

因此, 代数 BOOLEAN 和代数 A 是同态的, 且对于 0, 1 均有映射(满射且直射), 故 BOOLEAN 和 A 同构。 $h^{-1}(1) = \text{true}$, $h^{-1}(0) = \text{false}$ 成立。

若有态射函子 $h: \text{BOOLEAN} \rightarrow B$ 。同样有:

$h(\text{true}) = \text{yes}, h(\text{false}) = \text{no}$

验证(17-1)式可知 BOOLEAN, B 是同态的。但由于非满射(maybe 无对应), 故非同构。

同样, 若有 $h: \text{BOOLEAN} \rightarrow C$ 。同样有:

$h(\text{true}) = \text{any}, h(\text{false}) = \text{any}$

验证(17-1)式:

$h(\text{not}(\text{true})) = h(\text{false}) = \text{any}$

$\text{same}(h(\text{true})) = \text{same}(\text{any}) = \text{any}$

它们依然同态, 但由于非直射(非一对一), 故非同构。以上仅仅是为说明概念的非常简单的例子。

为了清晰表明代数间映射关系，常用交换图(commuting diagram)。

上例同构，同态映射如下图：

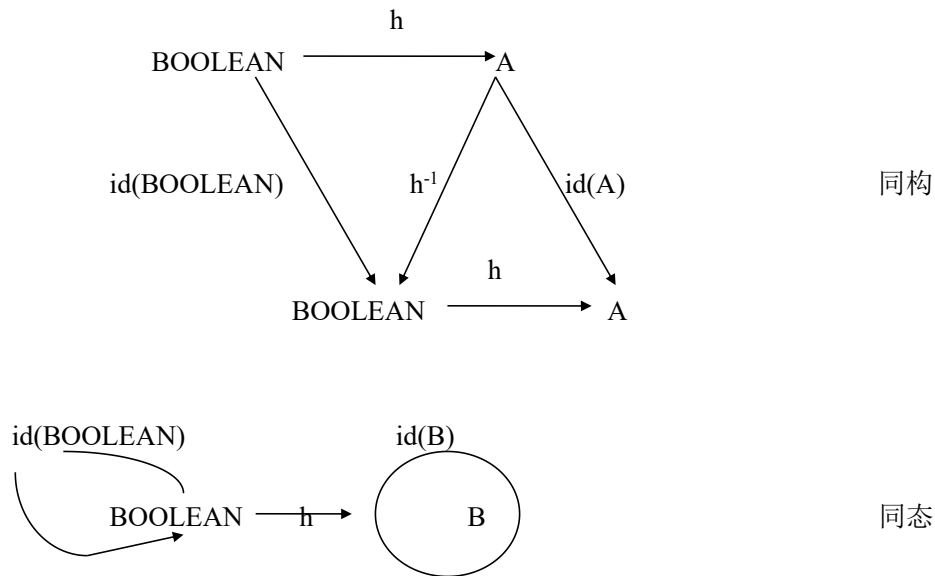


图 17-1 态射的交换图

其中 id 为恒等函数，是单位元操作。

17.1.2 Σ -代数

以上我们复习了代数的一般性质。程序员在设计程序时如能构造抽象代数，把它写成规格说明，即 S_p 代数，再通过中间形式变为实现，可以看作是同态映射变成不同的代数。这就成为公理化自动程序设计的模型。学术性语言: Alphard, CLU, Edis, Hop, ML, POLY, Amber, Quest, Pebble 就是力图按这样做的。商用语言 Modula-2, Ada, Pascal Plus 部分采用这种办法，特别是在抽象数据类型的描述中。为此，我们还要考察 S_p -代数的具体模型。先看 Σ -代数。

定义 17-6(型构 Signature)

型构是表示操作的符号(有限或无限)集。例如，我们在自然数集上指定四个函数符 $\{zero, succ, pred, plus\}$ ，我们就指明了一个代数结构 (N, Σ_n) 。 Σ_n 是这四个函数符的统称叫型构。

定义 17-7(目 Arity)

目是每一函数符所要求的参数个数。对于 Σ 中的每一函数符 σ ，均有一求目的函数：

$$arity(\sigma): \Sigma \rightarrow N$$

例如，以上四函数符的型构，其目数是：

```
arity(zero) = 0      // 不带参数 zero 为常(函)数，零目算子。
arity(succ) = 1
arity(pred) = 1
```

$$\text{arity}(\text{plus}) = 2$$

我们注意的是 0 目函数符，它是 Σ 中的常量，它的作用是为代数提供常值。

定义 17-8(Σ -代数)

若一代数其承载子集合 A 仅由 Σ 操作，则称 (A, Σ_A) 为 Σ -代数。

Σ -代数可以看作是代数的结构，即 A 中的元素仅仅是 Σ 中函数符可操纵的元素。 Σ -代数也是对型构 Σ 的解释，即每一函数符的参数是什么值，返回的是什么值。

例 17-2 设 $\Sigma = \{\text{zero}, \text{succ}, \text{pred}, \text{plus}\}$ ，其函数符目度如定义 17-7。若 N 是一自然数集(N, Σ_N)是一 Σ -代数，其中：

$$\begin{array}{ll} \text{zero}_N: \rightarrow N & \text{具体定义为 } \text{zero}_N = 0 \\ \text{succ}_N: N \rightarrow N & \text{succ}_N(n) = n + 1 \\ \text{pred}_N: N \rightarrow N & \text{pred}_N(0) = 0 \\ & \text{pred}_N(n+1) = n \\ \text{plus}_N: N \times N \rightarrow N & \text{plus}_N(n, m) = n + m \end{array}$$

若 Z 是一整数集， (Z, Σ_Z) 是一 Σ -代数，其中：

$$\begin{array}{ll} \text{zero}_Z: \rightarrow Z & \text{具体定义为 } \text{zero}_Z = 0 \\ \text{succ}_Z: Z \rightarrow Z & \text{succ}_Z(n) = n + 1 \\ \text{pred}_Z: Z \rightarrow Z & \text{pred}_Z(n) = n - 1 \\ \text{plus}_Z: Z \times Z \rightarrow Z & \text{plus}_Z(n, m) = n + m \end{array}$$

再如，若 E 是偶自然数集 $\{0, 2, 4, 6, \dots\}$ ，则 (E, Σ_E) 也是一 Σ -代数，其中：

$$\begin{array}{ll} \text{zero}_E: \rightarrow E & \text{具体定义为 } \text{zero}_E = 0 \\ \text{succ}_E: E \rightarrow E & \text{succ}_E(n) = n + 2 \\ \text{pred}_E: E \rightarrow E & \text{pred}_E(0) = 0 \\ & \text{pred}_E(n+2) = n \\ \text{plus}_E: E \times E \rightarrow E & \text{plus}_E(n, m) = n + m \end{array}$$

我们将同一 Σ 施加于三种承载子集合上，分别得到 (N, Σ_N) ， (Z, Σ_Z) ， (E, Σ_E) 三个 Σ -代数。然而，我们最感兴趣的是承载子元素均可由 Σ 生成的项代数。

定义 17-9 (Σ -项， Σ -项集，项-代数)

若 Σ -代数 (A, Σ_A) 中承载子集合 A 中的每一元素 $a_i \in A (i=1, \dots, n)$ 均可用 Σ 中的函数符及其复合表示，则每一用函数符号串表示的项称为 Σ -项。

[1] 若 $\sigma \in \Sigma$ ，且为 0 目函数符，则 σ 即为 Σ -项。记为 $\sigma_0 = C$ 。

[2] 若 $\sigma \in \Sigma$ ，且为 $k > 0$ 目的函数符，则形如 $\sigma(t_1, \dots, t_k)$ 的串是 Σ -项，其中 t_1, \dots, t_k 也是 Σ -项。记 Σ -项的集合为 T_Σ ，为满足上述规则的最小项集。

[3] 若 T_Σ 中没有 0 目 σ ，则 $T_\Sigma = \emptyset$ 。

[4] 若 $T_\Sigma \neq \emptyset$ 则 (T_Σ, Σ) 即为项-代数。

例 17-3 设 Σ 定义如前例， T_Σ 包括以下项：

$$\begin{array}{l} \text{zero}, \text{succ}(\text{zero}), \text{succ}(\text{succ}(\text{zero})), \dots \\ \text{pred}(\text{zero}), \text{pred}(\text{pred}(\text{zero})), \dots \\ \text{succ}(\text{pred}(\text{zero})), \text{pred}(\text{succ}(\text{zero})), \dots \\ \text{plus}(\text{zero}, \text{zero}), \text{plus}(\text{succ}(\text{zero}), \text{zero}), \dots \end{array}$$

显然，项代数成了承载子集生成语法规则。程序设计语言中的数据和操作可按某给定的 Σ 型构以项代数构造。项 _{Σ} 代数是特殊的 Σ _{Σ} 代数。

按上述项代数定义的承载子集 T_Σ 是归纳性的，即归纳出常量符号和 Σ 中每个 σ 对这些符号反复操作的最小串的集合。 T_Σ 的归纳性质为导出项的各种特性提供了强有力的证明方法。例如，为了证明 T_Σ 中所有项均具有 P 性质，只要证明以下两点就是充分的。

[1] 证明 Σ 中所有常量符号均具有性质 P 。

[2] 假定项 t_1, \dots, t_k 具有性质 P ，对于 Σ 中所有 $k > 0$ 目的 σ ，证明项 $\sigma(t_1, \dots, t_k)$ 也具有性质 P 。

这就是所谓结构归纳法。它在项的结构上归纳。我们也可以用结构归纳法定义 T_Σ 上的关系和函数。例如，如欲在 T_Σ 上定义函数 g ，满足以下两个条件就是充分的：

[1] 定义将 g 应用于常量函数符的结果。

[2] 对于 Σ 中每个 $k > 0$ 目的 σ ，通过 $g(t_1), \dots, g(t_k)$ 来定义 g 应用于 $\sigma(t_1, \dots, t_k)$ 的结果。这两个条件可归并为：

假定将 g 应用于项 t_1, \dots, t_k 的结果成立，则对 Σ 中每个 $k > 0$ 目的 σ ，可定义 g 应用于 $\sigma(t_1, \dots, t_k)$ 的结果。

有了结构归纳法我们继续考察项代数的性质。

定义 17-10 (Σ _{Σ} 同态, Σ _{Σ} 同构)

设 (A, Σ_A) , (B, Σ_B) 是两 Σ _{Σ} 代数， $h: A \rightarrow B$ 为映射函数，仅当 Σ 中每个 k 目的 σ ，有：

$$h(\sigma_A(a_1, \dots, a_k)) = \sigma_B(h(a_1), \dots, h(a_k)) \quad (17-2)$$

则两代数 Σ _{Σ} 同态， h 为同态映射。

通俗地说 Σ _{Σ} 同态为你有的承载子集及其服从的操作性质我也有，反过来则未必。当然具体值集可以不一样。读者可按此定义验证例 17-2 中 (N, Σ_N) 和 (Z, Σ_Z) 非同态，而 (N, Σ_N) 和 (E, Σ_E) 是同态的。

若 h 为双射的则 Σ _{Σ} 同态 $h: A \rightarrow B$ 即为同构。同构则表明 Σ 中任何函数符若作用于 A 的承载子集上为真，作用于 B 的承载子上亦为真。反之亦然。同样，两代数值集可以不一样。如 $A = \{\text{true}, (\wedge, \vee)\}$, $B = \{1, 0, (+, *)\}$ 。

请注意 Σ _{Σ} 同态和一般代数同态是一个意思，只是它仅限于 Σ 集，更容易满足。

定理 17-1 (Σ _{Σ} 同态的唯一性、存在性)

对于每个 Σ _{Σ} 代数 (A, Σ_A) 都存在唯一的 Σ _{Σ} 同态映射

$$i_A: T_\Sigma \rightarrow A \quad (17-3)$$

证明

[1] 先证同态存在性。

对于 Σ 中某个 $k > 0$ 目的 σ ，形如 $\sigma(t_1, \dots, t_k)$ 的项是 T_Σ 的项。按结构归纳法。常量 T_Σ 项的 $i_A(t_1) \dots, i_A(t_k)$ 已经定义。则 $i_A(\sigma_A(t_1, \dots, t_k))$ 可定义为 $\sigma_A(i_A(t_1), \dots, i_A(t_k))$ 。这样， T_Σ 中的每一元素都作了 i_A 定义。再检查 i_A 是否同态的：

设 Σ 中的 σ 是 k 目的，则有：

$$\begin{aligned} i_A(\sigma_{T_\Sigma}(t_1, \dots, t_k)) &= i_A(\sigma_A(t_1, \dots, t_k)) && // \text{按 } \sigma_{T_\Sigma} \text{ 定义} \\ &= \sigma_A(i_A(t_1), \dots, i_A(t_k)) && // \text{按 } i_A \text{ 定义} \end{aligned}$$

其中 $\sigma_{T_\Sigma}: T_\Sigma \rightarrow T_\Sigma$, 即将项元组 $\langle t_1, \dots, t_k \rangle$ 映射为项 $\sigma(t_1, \dots, t_k)$ 。此证同态。 \square

[2] 再证唯一性

设 h 是从 T_Σ 到 A 的某个同态映射, 只要证明 T_Σ 中的每个 t 都有 $i_A(t) = h(t)$, 即 i_A, h 重合。不失一般性, 设 σ 是 k 目的, 有:

$$\begin{aligned}
 i_A(\sigma(t_1, \dots, t_k)) &= \sigma(i_A(t_1), \dots, i_A(t_k)) && // \text{按 } i \text{ 定义} \\
 &= \sigma_A(h(t_1), \dots, h(t_k)) && // \text{按结构归纳} \\
 &= h(\sigma_{T_\Sigma}(t_1, \dots, t_k)) && // \text{由于 } h \text{ 是 } \Sigma \text{ 同态} \\
 &= h(\sigma(t_1, \dots, t_k)) && // \text{按 } \sigma_{T_\Sigma} \text{ 定义} \\
 \therefore i_A &= h && \text{此证唯一。} \quad \square
 \end{aligned}$$

如果我们把 T_Σ 看作程序设计语言的语法, Σ _代数 (A, Σ_A) 看作是语义域或解释。则本定理说明语言中的每一表达式或项, 在 (A, Σ_A) 中都对应唯一的含义, 即在语义域中只有一个解释。本定理的另一层意图是试图说明 T_Σ 是“最小”的 Σ _代数。一般说来, Σ _代数 (A, Σ_A) 有许多项是相等的。如 (N, Σ_N) 的 $\text{pred}(\text{succ}(\text{zero}))$ 和 zero ; (E, Σ_E) 的 $\text{succ}(\text{pred}(\text{zero}))$ 和 zero 。既然存在唯一的同态 $i_A: T_\Sigma \rightarrow A$, 若 (A, Σ_A) 中有 $i_A(t_1) = i_A(t_2)$, 则 T_Σ 中 t_1 和 t_2 必然相等。这样, T_Σ 就应该是“最小”的 Σ _代数了。事实上是做不到的。为了证明这一点, 设 A 等于 T_Σ , 那么存在的唯一同态就是 T_Σ 上的恒等函数 id_{T_Σ} 。只有在 t_1 和 t_2 在语法上也完全一致时, $\text{id}_{T_\Sigma}(t_1) = \text{id}_{T_\Sigma}(t_2)$ 才是正确的。

17.1.3 全等类

在讨论全等类之前我们先作几个定义。

定义 17-11(Σ -代数类)

具有 Σ 操作的代数集合称 Σ _代数类, 记为 C 。

定义 17-12(初始代数 Initial algebra)

若代数类 C 中 Σ _代数 I 是初始代数, 仅当对 C 中每一 Σ _代数 J 都存在着从 I 到 J 的唯一 Σ _同态。

由定理 17-1, 项代数 T_Σ 在所有 Σ _代数的类中是初始的。这就意味着 T_Σ 到任何 Σ _代数的值都存在着唯一的项映射。这是很强的概念。人们只要标识出某个有“意义”的 Σ _代数, 即可将项映射到该代数的元素上。以此定义项的语义。

定理 17-2

若 Σ _代数类 C 中代数 A, B 均为初始代数, 则它们必为同构的。

证: 若 A 为初始的, B 为一般 Σ _代数, 按定义 17-12 它们必存在唯一 Σ _同态 $i_1: A \rightarrow B$ 。同样, 若 B 为初始的, A 为一般 Σ _代数, 也存在唯一 Σ 同态 $i_2: B \rightarrow A$ 。它们的复合

$$i_1 \circ i_2 = A \rightarrow A = \text{id}_A$$

同理, $i_2 \circ i_1 = B \rightarrow B = \text{id}_B$

所以, 它们是同构的。

初始代数只在符号形式上区别初始项，只要符号不同就是不同的值。例如，有 Σ _项代数

$$\text{Bool} = \{\text{true}, \text{false}, \text{not}\}$$

其项集是：

$$T_{\Sigma} = \{\text{true}, \text{not}(\text{true}), \text{not}(\text{not}(\text{true})), \dots, \text{false}, \text{not}(\text{false}), \text{not}(\text{not}(\text{false})), \dots\}$$

事实上我们知道 $\{\text{true}, \text{not}(\text{false}), \text{not}(\text{not}(\text{true})), \dots\}$ ，和 $\{\text{false}, \text{not}(\text{true}), \text{not}(\text{not}(\text{false})), \dots\}$ 是语义等价的两个类我们记为 $\{[\text{true}], [\text{false}]\}$ 。

定义 17-13 (Σ _全等 congruence)

在 Σ _代数 (A, Σ_A) 中， A 上的关系 R 是 Σ _全等关系，若 $\langle a_i, a_i' \rangle \in R$ ($0 \leq i \leq k$, $a_i, a_i' \in A$) 成立，仅当对 Σ 中每个 k 目的 $\langle \sigma_A(a_1, \dots, a_k), \sigma_A(a_1', \dots, a_k') \rangle \in R$ 也成立。

按上例， $(\text{ture}, \text{not}(\text{false})) \in R$ ，则有 $(\text{not}(\text{true}), \text{not}(\text{not}(\text{false}))) \in R$ 。全等关系若以 '=' 符号直接表示两个项是全等的。以上定义是：

若 $\text{ture} \equiv \text{not}(\text{false})$ 则有

$$\text{not}(\text{ture}) \equiv \text{not}(\text{not}(\text{false}))$$

定义 17-14 (商代数 Quotient Algebra)

对于 Σ _代数 (A, Σ_A) 中的承载子 $a \in A$ ，按全等关系 R 归于 $[a]_R$ 则称商化(quotienting)。商化的结果得到全等类集合 $A/R = \{[a]_R \mid a \in A\}$ ，且在 A/R 上对 Σ 中的每个 σ 可定义以下映射：

$$\sigma_{A/R}([a_1]_R, \dots, [a_k]_R) = [\sigma_A(a_1, \dots, a_k)]_R$$

其中 $\sigma_{A/R} \in \Sigma_{A/R}$ ， $[]$ 表示...的全等类。则称 $(A/R, \Sigma_{A/R})$ 为商代数。

可以推论：[1] $(A/R, \Sigma_{A/R})$ 是 Σ _代数。

[2] 由于存在 $A \rightarrow A/R$ 直射， $h(a) = [a]_R$ 也是 Σ _同态。

证明从略。

我们最感兴趣的是在项集 T_{Σ} 上的 Σ _全等。如果所有项对偶 $\langle t, t' \rangle \in R$ ，根据 T_{Σ} 的初始性有 $i_A: T_{\Sigma} \rightarrow A$ ，则 $i_A(t) = i_A(t')$ 即 Σ _代数 A 也具有等价关系 R 。设 $C(R)$ 是所有具有 R 性质的 Σ _代数类。

定理 17-3 (全等的初始性)

在具有性质 R 的代数类中 Σ _代数 T_{Σ}/R 是初始代数。

证：

[1] 先证 T_{Σ}/R 在 $C(R)$ 之中。 T_{Σ}/R 是商化的 T_{Σ} ，它存在自然直射 $\text{in}: T_{\Sigma} \rightarrow T_{\Sigma}/R$ ，是 Σ _同态映射。由于 T_{Σ} 在所有 Σ _代数类中的初始性， in 和 $i_{T_{\Sigma}/R}$ 应一致且唯一。若 $\langle t, t' \rangle \in R$ ，($t, t' \in T_{\Sigma}$) 则有：

$$\begin{aligned} i_{T_{\Sigma}/R}(t) &= [t]_R \\ &= [t']_R \quad \text{由 } \langle t, t' \rangle \in R \\ &= i_{T_{\Sigma}/R}(t') \end{aligned}$$

故 $[t], [t'] \in R$ ， T_{Σ}/R 在 $C(R)$ 之中。

[2] 设任意 Σ _代数 $A \in C(R)$ ，证明 $h: T_{\Sigma}/R \rightarrow A$ 是同态映射。即证明 $h(\sigma_{T_{\Sigma}/R}([t]_R)) = \sigma_A(h([t]_R))$ 且 $t, t' \in [t]_R$ ，有：

$$h(\sigma_{T_{\Sigma}/R}([t]_R)) = h([\sigma(t)]_R) = i_A(\sigma(t)) = \sigma(i_A(t)) = \sigma_A(h([t]_R))$$

[3] 再证唯一性

设任一同态映射 $g: T_{\Sigma/R} \rightarrow A$ 。由于设 $i_A'(t) = g[t]_R$ ，且根据定义 $i_A': T_{\Sigma} \rightarrow A$ 。则有：

$$i_A(\sigma_{T_{\Sigma/R}}([t])) = g([\sigma(t)]_R) = g(\sigma_{T_{\Sigma/R}}(t)) = \sigma_A(g[t]_R) = \sigma_A(i_A'(t))$$

i_A' 是 Σ -同态映射。又由于 T_{Σ} 的初始性，则 i_A' 和 i_A 必须一致。再按

$$g([t]_R) = i_A(T) = i_A(t) = h([t]_R)$$

则 g 和 h 是一致的，唯一得证。 \square

17.1.4 泛同构映射

给定一操作集，我们可构造所有可能的表达式，也就是对应于 Σ 的所有可能值集的外延。在这个值集上操作的代数则称字代数。所有前述 Σ -代数(例 17-2)均为字代数。字代数必须以操作符号(字)表达计算。 $3+5$ 按前述自然数集上代数 (N, Σ_N) 是 Σ -代数，但不是字代数，虽然代数是处理‘3’和‘5’，‘pred’，‘succ’这些符号，但它们是有实指对象的，字代数纯粹是符号，只解决代数结构的语法没有恒定对象。例如代数式：

$$ss0$$

‘s’可以是‘后继’，‘0’可以是零，它是自然数集上的代数。‘s’可以是‘not’，‘0’可以是‘true’，它是布尔代数。我们前述的代数 (A, OP) 是有实指代数对象的抽象，与之对应的操作符号集 OPS ，以及由字‘变量’表征的重载子集 $E(X)$ ，则 $(E(X), OPS)$ 称字代数。正是字代数的抽象可以代表任何同构的代数，也是程序设计的工具(操作符号集)。以公理的办法限定 $(E(X), OPS)$ 的同构映射，则程序得出语义。

从商代数出发(字代数也是商代数之一)寻找同态映射，直至找到同构。字代数描述了语法，同构的某个代数是语义解释，如‘ $3+5=8$ ’是自然数表达式表达的语义。

定义 17-15(泛同构 Universal Isomorphism)

给定一代数，从它的商代数出发可以找到许多同态映射，直到找出同构。则称为泛同构。

从以上可知，商代数可以看作是简化了的字代数，它按全等关系归类，是全等关系上的字代数。

定义 17-16(自由字代数 Free word algebra)

设 $Y = \{y_1, \dots, y_m\}$ 是 m 个可区分的变量符号集合， $\xi_i(y_1, \dots, y_m) \in E(Y)$ 是由 Y 组成的表达式集合。 (A, OP) 是一个代数 $(E(Y), OPS)$ 是与之对应的字代数 $f \in OPS, op \in OP$ ，仅当下式成立：

$$[1] (\xi(y_1, \dots, y_m) = y_1) \supset (\xi(a_1, \dots, a_m) = a_1)$$

$$[2] (\xi(y_1, \dots, y_n) = f(\xi_1(y_1, \dots, y_m), \dots, \xi_n(y_1, \dots, y_m)))$$

$$\supset (\xi(a_1, \dots, a_m) = a_1) = op(\xi_1(a_1, \dots, a_m), \dots, \xi_n(a_1, \dots, a_m))$$

自由字代数是每个项均可变为变量的字代数。因为按泛同构理论，从商代数寻找同构，把字代数因素化要更方便。 $E(Y)$ 是有变量并以表达式形式表达的承载子集合。

例 17-4 Σ -字代数的项集

设 $Y = \{x, y, z\}$ ， $\Sigma = \{+, *\}$ 对于 Σ -字代数 $(E(Y), \Sigma)$ 可能的项集是：

$$\{x, y, z, x*y, z+x, x*(y+x*z), \dots\}$$

Σ -字代数的项，可描述为 $terms(\Sigma, S)$ 当上下文清晰时 $terms(S)$ ，即 S 类别上的 Σ -项。

以上代数理论可直接应用于描述数据类型。(因为数据类型定义了值集和操作集, 本身即为一代数), 我们叫它规格说明代数。

定义 17-17(Sp-代数)

我们称 (Σ_0, Σ, E) 为 Sp-代数, 其中 Σ_0 为常量算子集, Σ 为算子集, E 为公理集。

公理集 E 是由 Σ_0 项表达的全等关系。在公理的约束下, Σ 中的任意 σ 的复合, 即表达了该类型值的生成规则和语义解释。下两节我们进一步介绍如何保证写出正确公理, 直至给出完整的代数规格说明。

17.2 代数规格说明

从上节代数理论中我们以型构描述数据类型是建立 Σ_0 代数的同构类。数据类型可以以下述等价方式描述:

- 字代数上的某个全等关系。
- 字代数的某个商代数。

如前所述, 字代数也是项代数。而商代数每个元素都是全等类, 直接给出有些麻烦。而它们本质是一回事。故一般代数规格说明中以代数公理给出全等关系。商代数在证明系统中更为方便。

代数公理是表征两个代数项全等的等式集合。若 x, y 是同一类别的两个项, 则

$$x =_R y \text{ (上下文清晰时略去下标 } R \text{)}$$

即为一简单公理。根据公理置换型构中的操作符, 即可生成全等类。置换中遵循全等性质:

$$\text{自反性 } x =_R y, y =_R x \text{ 或 } x \equiv y \quad //x, y \text{ 是同一项}$$

$$\text{对称性 } y =_R x$$

$$\text{传递性 } x =_R y, y =_R z \text{ 则 } x =_R z$$

全等性 若 $x_i =_R y_i$, 有某操作 σ 则

$$x_i \equiv \sigma(x_1, \dots, x_m); y_i \equiv \sigma(y_1, \dots, y_m) \quad 0 \leq i \leq n$$

$$\text{转换性 } x =_R y; f(x) =_R f(y)$$

例 17-5 简单公理的全等类

若有一最简单型构:

$$0: \rightarrow N$$

$$S: N \rightarrow N$$

和以下公理集:

$$R_0 = \{ \}$$

$$R_1 = \{ 0 = 0 \}$$

$$R_2 = \{ 0 = S0 \}$$

$$R_3 = \{ 0 = SS0 \}$$

$$R_4 = \{ S0 = SS0 \}$$

由这五个公理生成的全等类是:

$$C/R_0, R_1: \{0\}, \{S0\}, \{SS0\}, \dots$$

对于 R_0 因无公理所以算子可任意组合, 每个 $\{\}$ 号内为一全等类。对于 R_1 只有 $0 = 0$ 两边同施 S 算子任意次, 每次都是不同的全等类, 故如上述。如果 S 的语义是“后继”则 $C/R_0, R_1$ 为自然数集。

$$C/R_2: \{0, S0, SS0, \dots\}$$

由于满足 R_2 公理 $0 = S0$, 它们是同一全等类, 两边同施 S , 即 $S0 = SS0$ 。根据传递关系它们仍为同一全等类…。即为所有项均全等的小代数。

$$C/R_3: \{0, SS0, SSSS0, \dots\}, \{S0, SSS0, SSSSS0, \dots\}$$

是两个全等类。因为对 $0 = SS0$ 两边施 S 得 $S0 = SSS0$, 再施 S 得 $SS0 = SSSS0$ …。于是, 按传递性得以上两类, 奇数个 S 和 0 一类, 偶数个 S 和 0 为另一类。可以看做布尔代数值集 $\{[true], [false]\}$ 。

$$C/R_4: \{0\}, \{S0, SS0, SSS0, \dots\}$$

由于公理 R_4 除 $\{0\}$ 外都是同一全等类, 那么 S 必为常量算子, 联系到具体代数, 为二值代数, S 为“补”算子。

由此看出, 一个简单的常量和一个单目算子的型构, 用一个简单公理即可定义许多不同的代数。再次说明, 以上 $(C/R_i, \Sigma)$ 都是 Σ -字代数, 由于 Σ -全等的关系不同, 同态映射为自然数、小代数、布尔代数、二值代数。也说明 Σ -字代数的初始性。每一代数都是一简单数据类型的模型。实际的数据类型要比这复杂得多, 其型构不仅多且往往是多类别的。这就要把前述的理论扩充到多类别 (multi sort) 代数上。这并不十分难, 因为, 任何一 k 目操作 $\sigma_{T_\Sigma}: T_\Sigma^k \rightarrow T_\Sigma$ 都是将 k 元元组 $\langle t_1, t_2, \dots, t_k \rangle$ 映射为项 $\sigma(t_1, t_2, \dots, t_k)$ 只需记

$$\sigma(t_1: s_1, t_2: s_2, \dots, t_k: s_k)$$

17.2.1 Sp 代数公理

Sp-代数公理更强的一般形式是带变量的公理

定义 17-18 (Sp-代数公理)

公理带有项变量的等式。形如:

$$r(v_1, \dots, v_n) = s(v_1, \dots, v_n)$$

当变量 v_i 以项 t_i 置换后

$$r(t_1, \dots, t_n) = s(t_1, \dots, t_n)$$

就是全等项。当变量个数为零时即简单公理。

例 17-6 带变量的公理

设有简单型构

$$0: \rightarrow N$$

$$S: N \rightarrow N$$

$$+: N \times N \rightarrow N$$

和公理

$$R5: \{x + 0 = x; \quad x + Sy = S(x + y)\}$$

$$R6: \{SSx = x; x + 0 = 0; x + S0 = x\}$$

满足它们的全等项:

$$C/R5 : \{0\}, \{S0\}, \{SS0\} \cdots$$

若数字 0 是 0, S 是“后继”, + 是“加”, 它映射为自然数集。

$$C/R6 : \{0\}, \{S0\}$$

若 0 是 false, S 是 not, + 是 \wedge , 它映射为布尔数集(S0 为 true)。

为证明 R5 描述的是自然数集上的全等。我们证明 $2+1 \equiv 3$, 即 $SS0+S0 \equiv SSS0$;
先证左端表达式, 与 R5(2)(即第 2 条公理)的左端匹配 $x = SS0, y = 0$ 置换右端得:

$$\begin{aligned} SS0+S0 &= S(SS0+0) && \text{由公理 2} \\ &= S(SS0) && \text{由公理 1} \\ &= SSS0 && \text{去括号得右端} \end{aligned}$$

同理, 为证明 R6 描述的是布尔数集上的全等。我们证明:

$$\text{not true} \wedge \text{not false} \equiv \text{false}$$

即 $SS0 + S0 \equiv S0$ 。也是先看左端

$$\begin{aligned} SS0 + S0 &= S0 + 0 && \text{由 R6(3)} \\ &= 0 && \text{得证} \end{aligned}$$

再证 $\text{not}(\text{not}(\text{not}(\text{not}(\text{not}(\text{not}(\text{false})))))) \equiv \text{false}$, 即 $SSSSSS0 = 0$ 。我们有

$$\begin{aligned} SSSSSS0 &= SS(SSSS0) &= SSSS0 && \text{由 R6(1), } x \text{ 与 } SSSS0 \text{ 匹配} \\ &= SS(SS0) &= SS0 && \text{由 R6(1), } x \text{ 与 } SS0 \text{ 匹配} \\ &= SS(0) &= 0 && \text{由 R6(1), } x \text{ 与 } 0 \text{ 匹配} \end{aligned}$$

17.2.2 隐式算子与条件公理

现在要问一个问题: 是否有限个公理就足以描述所有的数据类型? Majster 首先研究了这个问题。结论是不行(1979)。Thatcher 等人举出一个玩具堆栈的例子。引入以下算子:

$$\Sigma = \{0, E, P, D\}$$

0: $\rightarrow T$ (空栈);

E: $\rightarrow T$ (错误);

P: $T \rightarrow T$ (压入);

D: $T \rightarrow T$ (弹出)。

公理 R7 规定了错误条件: 弹出错误, 压入错误, 先弹后压入空栈是错误, 弹出次数大于压入次数是错误。

$$R7: \{DE = E, PE = E, PDx = E\} \cup \{D^n P^k 0 = E \mid n > k\}$$

D^n 是 n 个 D 的缩写, P^k 是 k 个 P 的缩写。

由 R7 生成的等价类是:

$$C/R7: \{ D^n P^k 0 \mid n \leq k \} \cup \{ E, D0, DE, PE, \dots \}$$

即所有有效操作和无效操作。Thatcher 证明 R7 是一无限公理集。 n, k 个数不定, 很难写栈满操作。

如果在型构上加上一个隐藏(对用户)的算子, 则用有限个公理即可描述了:

```

0:  $\rightarrow T$ 
E:  $\rightarrow T$ 
P:  $T \rightarrow T$ 
D:  $T \rightarrow T$ 
*H:  $T \rightarrow T$           /* 为隐式算子
x: T                  //变量
R8 : { DE = E, PE = E, HE = E, D0 = E,
      PDx = E, PHx = E,
      DHx = HDx,
      DP0 = H0,
      DPPx = HPx }
```

显然, 任何 $D^n P^k 0$ 均可用后三条公理置换出。

隐式算子仅仅是为了写规格说明方便。 $p = \text{if } q \text{ then } r \text{ else } s$ 是条件公理, 其中 if, then, else 既非项也非操作符, 为了一致性, 条件公理也可以说是增加了一个 ifthenelse 算子, 相当于:

$$p = \text{ifthenelse}(q, r, s)$$

隐式算子用户不能用(于证明系统), 因此全等类依然保持不变。可以认为前者是后者的易读写法, 可出现在等式集之中。尽管 $\Sigma' = \Sigma + \{\text{ifthenelse}\}$ 增大了

$$A = \{[t] \mid t \in \text{terms}(\Sigma, s)\}$$

而不是:

$$A' = \{[t] \mid t \in \text{terms}(\Sigma', s)\}$$

即 A' 比 A 中只多出一个子全等类, 其中每项至少有一个隐式算子。这种假定因多余的字全等类不可达, 是可行的。

Guttag 1980 年证明: 任何可计算数据类型都可以用有限个带隐式算子的公理定义。

17.2.3 保留, 完整性和一致性

数据类型的描述, 往往是将描述过的类型为以后的用, 在原有规格说明 S 上作扩充(增加类别、算子、公理), 或增强(只增加算子和公理)为 S' 。 S 的所有类别、算子、公理为 S 继承, 称之为保留(protection)。显然, 在扩充为新类型的规格说明时, 不对老类型的完整性和一致性有所影响。

定义 17-19($S_{\text{完整}}$)

设 T 具有型构 Σ 和公理 E 的规格说明 (Σ, E) , T' 是规格说明 (Σ', E') 且 $\Sigma \subset \Sigma'$, $E \subset E'$ 。对于某个类别 s , 由 E, E' 产生的全等关系 R, R' 。若 R' 中的每一个全等类都包含了 R 的全等类的一个项, 则称 T' 对 T 有 $S_{\text{完整}}$ 。

直觉上从 T 扩充到 T' , s 类别并未得到新值。

定义 17-20($S_{\text{一致}}$)

设 T, T' 定义如定义 17-19。若在 R 中找到非全等项 x 和 y , 在 R' 中也不全等, 则称 T' 对 T 是 S 一致。

直觉上, 在 T 中可区分的值在 T' 中仍然是可区分的。

定义 17-21(完整, 一致, 保留)

设 T , T' 定义如 17-19。若 T 中所有类别 S 在 T' 中均 S _完整, 则称 T' 对 T 有 T _完整, 也叫充分完整(Wand 1979 称 \wedge_full)。

若 T 中所有类别 S 对 T' 的所有 S 是 S _一致的, 则称 T' 对 T 一致(Wand 1979 称 $\wedge_faithful$)。

完整性保证扩充时不为老类型带来新值。一致性保证扩充时原有单独的值不会合并为新值。不完整性意味着有较多的公理(过强), 而不一致性意味着我们需要较少的公理(较弱)。

17.3 数据类型的代数规格说明

一个代数规格说明包括以下五个部分

- 规格说明的名字(扩充 | 增强规格说明名字表)
- 类别列表
- 型构
- 变量列表
- 公理列表

每个部分以关键字开头, 如 **specification**, **sort(s)**, **operators**, **variables**, **equations**。以下是 TRUTH_VALUES 数据类型的代数规格说明。

specification TURTH-VALUES

sort Truth_Value

operations

ture : Truth_Value

false: Truth_Value

not_ : Truth_Value \rightarrow Truth_Value

$_ \wedge _$: Truth_Value , Truth_Value \rightarrow Truth_Value

$_ \vee _$: Truth_Value , Truth_Value \rightarrow Truth_Value

$_ = > _$: Truth_Value , Truth_Value \rightarrow Truth_Value

variables t, u: Truth_Value

equations

not true = false (17.5-a)

not flase = true (17.5-b)

$t \wedge \text{true} = t$ (17.5-c)

$t \wedge \text{false} = \text{false}$ (17.5-d)

$t \wedge u = u \wedge t$ (17.5-e)

$t \vee \text{true} = \text{true}$ (17.5-f)

$t \vee \text{false} = t$ (17.5-g)

$t \vee u = u \vee t$ (17.5-h)

$t = > u = (\text{not } t) \vee u$ (17.5-i)

end specification

其中 **sort** 引入类型的名字(故叫类别), **operators** 引入操作集(即 Σ)其中占位符'_'表明每个操作的目度, 零目的常量无占位符。变量为描述公理而设。公理用一组等式描述, 即 Truth_Value 类型的理论。等式定义了项的语义, 即 \models 两边当用项置换变量后相等, 符合一般推理风范。

这个简单类型没有带扩充/增强规格说明名字表。一般情况下像这种单一类别是很少的, 即使

最简单的基本类型 NATURAL，当两自然数比较时，结果是 Truth_Value 类别。所以代数规格说明一般是多类别代数。以 **include** 关键字引入老类别(包括了老类别定义的所有类别、操作、变量、等式)。它称之为保留(protection)和 OO 中继承概念一样。对于新写的规格说明，如果缺类别，则称此规格说明为保留规格的增强。否则称为扩充。**variables** 部分如上下文可明显看出则可省去。

17.3.1 简单类型的代数规格说明

真值类型的规格说明已如上述。该类型的值域通过给定的操作和等式出现，多次复合即可获得。凭我们直觉它是 {true, false} 的有限集。操作集描述了这些值具有的性质。

下面我们给出自然数类型的代数规格说明：

specification NATURALS

include TRUTH_VALUE

sort Natural

operations

0 : Natural
succ : Natural \rightarrow Natural
pred : Natural \rightarrow Natural
<_ : Natural, Natural \rightarrow Truth_Value
>_ : Natural, Natural \rightarrow Truth_Value
is_ : Natural, Natural \rightarrow Truth_Value
+_ : Natural, Natural \rightarrow Natural
*_ : Natural, Natural \rightarrow Natural

variables n, m: Natural

equations

pred succ n = n (17.6.-a)
pred 0 = 0 (17.6.-b)
0 < 0 = false (17.6.-c)
0 < succ n = true (17.6.-d)
succ n < 0 = false (17.6.-e)
succ n < succ m = n < m (17.6.-f)
n > m = m < n (17.6.-g)
0 is 0 = true (17.6.-h)
0 is succ n = false (17.6.-i)
succ n is succ m = n is m (17.6.-j)
n is m = m is n (17.6.-k)
0+n = n (17.6.-l)
(succ n) + m = (succ (n + m)) (17.6.-m)
n + m = m + n (17.6.-n)
0 * n = 0 (17.6.-o)
(succ n) * m = m + (n * m) (17.6.-p)
n * m = m * n (17.6.-q)

end specification

因为 **include** 保留了 TRUTH_VALUES 规格说明，本定义中可用 **sort** Truth_Value。自然数是无限集，故在操作中定义 succ, pred(后继和前导)函数。除了+, *运算还定义了三个比较操作，注

意'is'是判定相等的比较，为避开公理等式中的'='号而设。

每一操作的意义由公理等式限定。等号左边一个操作名往往要出现数次，如，`pred` 两次，`'<'` 四次，`'is'` 三次。而右边的表达式虽然也是项，但其中可以出现上文定义过的操作符号。公理等式描述的是操作符的代数性质，而不是对左边表达式的定义。例如：

```
n + m = m + n      //指出 '+' 算子的可交换性。
0 * n = 0           //指出等式两边表表达式的等价性。
```

只要满足等式公理的项可任意复合和置换，从而证明这个规格说明描述的项所具有性质。本规格说明并未显式给出自然数集 $\{0, 1, 2, 3, \dots\}$ ，但已给出同构的项集 $\{0, \text{succ}0, \text{succ succ}0, \dots\}$ ，它与以阿拉伯数字表示的自然数集具有完全一致的代数性质。例如， $\text{pred}(3) = 2$ ，我们有：

```
pred succ succ succ0 = succ succ 0      //按公理(17.6-a)
```

再如， $1 * n = n$ ，我们有：

```
(succ 0)*n ≡ n+(0*n)      //按公理(17.6-p)
           ≡ n+0           //按公理(17.6-n)
           ≡ 0+n           //按公理(17.6-o)
           ≡ n
```

请注意，在用公理作推理时，我们用'≡'表示两边是等价的。例如，若证明 $2 > 1$ ，在证明中应得出真假值。我们有：

```
succ succ 0 > succ 0 ≡ succ 0 < succ succ 0  //按(17.6-g)
                    ≡ 0 < succ 0             //按(17.6-f)
                    ≡ true                    //按(17.6-d)
```

本规格说明中有 $\text{pred}(0) = 0$ (17.6-b) 这条公理。它在自然数中并未定义，本处明确定义的目的是令 $\text{pred pred} \dots \text{pred}0 = 0$ ，则限定项集不是 0 就是 0 的多次后继。

有时定义公理时要用条件等式，即在公理等式后加上条件表达式，仅当条件表达式为 `true` 时等号右侧子表达式才成立。例如，我们为本规格说明加上 `'/'` 算子，则必须加一条公理：

```
n/n = succ 0    if not (n is 0)
```

则不会出现当 n 以置换时 $0/0 = \text{succ } 0$ ，因为 $\text{not}(0 \text{ is } 0)$ 为假。

17.3.2 参数化规格说明

有了以上约定我们就可以为较复杂的数据类型写规格说明了。例如，一个自然数的表：

```
specification NATURALS_LIST
```

```
  include NATURALS
```

```
  sort List
```

```
  operations
```

```
    empty_list:    List
```

```
    conr(, _):     Natural , List → List
```

```
    head_of_:      List → List
```

```
    tail_of_:      List → List
```

length_of: List \rightarrow Natural

variables

c: Natural

l: List

equations

head_of_cons(c, l) = c (17.7-a)

tail_of_cons(c, l) = l (17.7-b)

tail_of_empty_list = empty_list (17.7-c)

length_of_empty_list = 0 (17.7-d)

length_of_cons(c, l) = succ(length_of l) (17.7-e)

end specification

定义了一个 List 类别，其运算性质不外乎加一元素、取表头、取表尾、求表长。还必需定义表常量(empty_list)，这样，任何自然数表是空表与自然数的多次组合: cons(cj, ...cons(ci, cons(cl, empty_list))...), 其中 cj, ci, cl 为自然数。

(1) 参数化类别

除了个别缺陷而外(如 head_of_empty_list 应返回 error, 下文还要讲), 本规格说明描述了自然数表的性质。但是作为表其元素可以是任何类型的元素, 为了使一个规格说明通用则采用参数化类别、参数化操作。以下是参数化的表的规格说明:

specification LISTS

include NATURALS

formal sort Component

sort List

operators

empty_list: List

cons(,): Component, List \rightarrow List

head_of_: List \rightarrow Component

tail_of: List \rightarrow List

length_of: List \rightarrow Natural

variables

c: Component

l: List

equations

head_of_cons(c, l) = c

tail_of_cons(c, l) = l

tail_of_empty_list = empty_list

length_of_empty_list = 0

length_of_cons(c, l) = succ(length_of l)

end specification

这个表将表元素参数化为 Component 类别, 并指出它是形式类别。有了它可以定义一系列表, 类似于 Ada 的参数化设例。

```

specification NATURALS_LISTS
  include instantiation of LISTS by NATURALS
  using Natural for Component
end specification

```

指出以 NATURALS 为 LISTS 设例，LISTS 中的 Component 均以 NATURALS 中的 Natural 类别置换。显然换后的规格说明和前述 NATURALS_LISTS 是完全一致的。甚至还可以在设例时换名：

```

specification TRUTH_VALUE_LISTS
  include instantiation of LISTS by TRUTH_VALUES
  using Truth_Value for Component
  renamed using Truth_Value_List for List
end specification

```

这样就定义了元素为 Truth_Value 类别的表，因为参数化的 LISTS 包容了 NATURALS，而 NATURALS 中又包容了 TRUTH_VALUES，则本例是可行的。换名使凡是 List 出现之处均以 Truth_Value_List 代。

(2) 操作参数化

我们再写一个较为复杂的数组类型的规格说明。我们不仅使数组元素参数化，数组的长度也参数化，由于 Σ 代数中常数只能是常函数所以它是操作的参数化。

只定义 3 个数组的操作：构造一个空数组；在给定制索引处修改成分值；选取给定制索引处的数组元素。

```

specification ARRAYS
  include NATURALS
  formal sort Component
  formal operation maxsize: Natural
  sort Array
  operations
    empty_array : Array
    modify(, , ): Natural , Component , Array  $\rightarrow$  Array
    component_of: Natural , Array  $\rightarrow$  Component
  variables c : Component
    j , j : Natural
    a : Array
  equations
    component i of modify(j , c , a) = c
    if i is j  $\wedge$  i < maxsize (17.8-a)
    component i of modify(j , c , a) = component i of a
    if not (i is j) (17.8-b)
    modify(i , c , a) = a
    if not (i < maxsize) (17.8-c)
end specification

```

作为实例化的一个例子，请看以下的最大长度为 6 的真假值数组的规格说明：

specification TRUTH_VALUE ARRAYS**include instantiation of ARRAYS by TRUTH_VALUES****using Truth_Value of Component**succ succ succ succ succ succ 0 **for** maxsize**renamed using Truth_Value_Array for Array****end specification**

请注意，数字 6 以六个 succ 后接 0 的表示法是项的形式。它置换 maxsize。最后的换名换的是本规格说明中 sort 的名(Array)，而不是整个规格说明的名字。

设例后我们得到六元素真假值数组 TRUTH_VALUE_ARRAY。

这几个公理似乎有些难懂，但能反映数组元素运算特性。我们先解释定理：

$$\text{component_I_of_modify}(i, y, \text{modify}(i, x, a)) \equiv y$$

意思是找出数组第 i 个成分的值，这个数组是用 $\text{modify}(i, y, \text{modify}(i, x, a))$ 表达的。内面的 $\text{modify}(i, x, a)$ 是说数组 a 的第 i 个元素已改为 x 值。就这个数组，仍在第 i 个元素上改为 y，故整个式子取第 i 个位置元素，它应为 y。同理，可写出定理：

$$\text{component_I_of_modify}(i, x, \text{modify}(i, y, a)) \equiv x$$

它按修改次序给出正确的值。但如果不选取修改处成分，则其修改先后次序无关紧要。例如，按给出的公理可以证明以下定理：

$$\begin{aligned} & \text{component_I_of_modify}(j, x, \text{modify}(j, y, a)) \\ & \equiv \text{component_I_of_modify}(j, y, \text{modify}(j, x, a)) \quad \text{if not } (i \text{ is } j) \end{aligned}$$

可如下证明这个定理。在等式左边

$$\begin{aligned} & \text{component_I_of_modify}(j, x, \text{modify}(j, y, a)) \\ & \equiv \text{component_I_of_modify}(j, y, \text{modify}(j, x, a)) \quad \text{由(17.8-a)} \\ & \equiv \text{component_I_of_a} \quad \text{由(17.8-a)} \end{aligned}$$

在等式右边：

$$\begin{aligned} & \text{component_I_of_modify}(j, x, \text{modify}(j, y, a)) \\ & \equiv \text{component_I_of_modify}(j, x, a) \quad \text{由(17.8-c)} \\ & \equiv \text{component_I_of_a} \quad \text{由(17.8-a)} \end{aligned}$$

下面的一点是十分重要的，如果 $i \neq j$ ，就可以证明：

$$\begin{aligned} & \text{component_I_of_modify}(i, x, \text{modify}(j, y, a)) \\ & \equiv \text{component_I_of_modify}(j, y, \text{modify}(i, x, a)) \end{aligned}$$

但不能证明:

$$\text{modify}(i, x, \text{modify}(j, y, a)) \equiv \text{modify}(j, x, \text{modify}(i, x, a))$$

因为没有证明它的公理, 如果加上一条:

$$\text{modify}(i, x, \text{modify}(j, c', a)) \equiv \text{modify}(j, c', \text{modify}(i, c, a)) \quad \text{if not}(i \text{ is } j)$$

就可以证明了。公理就是在不断改进中完善的。

(3) 更多的参数化

除了类别、型构可参数化而外, 为了表达简洁, 规格说明其它部分均可参数化。如变量、公理的等式。这样, 就可描述更加复杂的类型。以下以映射的代数规格说明为例, 说明这个问题。

数组、函数都是映射, 它从自变量(对于数组是下标)域(Domain)映射为函数域(Range)。为了通用把它们定为形式类别。判定同一域上的两个自变量是否相等, 也可以作为形式操作。为了拿公理说明同一域内两元素相等的代数性质, 公理也参数化。相应用到的变量也参数化。公理无非说明它是对称、自反、传递的。显然, 映射代数规格说明中有一类别 Mapping(可理解为函数或数组索引)。其操作是设空映射; 自变量通过函数映射为映象(函数域的值); 以及修改函数, 即在自变量下得到新映象, 也就是修改了映射。modify 操作后返回值仍是映射。在公理中将它代入求映象函数, 并描述在条件下得到的结果。映射的代数规格说明如下:

specification MAPPINGS

include TRUTH_VALUES

formal sorts Domain, Range

formal operation

equals : Domain , Domain \rightarrow Truth_Value

formal variables a , b , c: Domain

formal equations

$$a \text{ equals } b = b \text{ equals } a \quad (17.9-a)$$

$$a \text{ equals } a = \text{true} \quad (17.9-b)$$

$$(a \text{ equals } b \wedge b \text{ equals } c) = > a \text{ equals } c = \text{true} \quad (17.9-c)$$

sort Mapping

operations

empty_mapping: Mapping

modify(_, _, _): Domain , Range , Mapping \rightarrow Mapping

image of _in_: Domain , Mapping \rightarrow Range

variables d , d' : Domain

r : Range

m : Mappings

equations

$$\text{image_of_d_in_modify}(d', r, m) = r \quad \text{if } d \text{ equals } d' \quad (17.9-d)$$

$$\text{image_of_d_in_modify}(d', r, m) = \text{image of } d \text{ in } m \quad \text{if not}(d \text{ equals } d') \quad (17.9-e)$$

end specification

如果是 NATURAL 到 TRUTH_VALUE 的映射，我们可设例如下：

```

specification NATURAL_TRUTH_VALUE_MAPINGS
  include instantiation of MAPPINGS  by NATURALS, TRUTH_VALUES
    using Truth_Value for Range,
      Natural for Domain,
    is for equals
end specification

```

这里把 NATURALS 中的 is 转换形式操作 equals。那么，NATURALS 中应有与(17.9-a)，(17.9-b) (17.9-c)对应的实际公理等式，但 NATURALS 中只有(17.6-k)与(17.8-a)对应，而(17.9-b)，(17.9-c)却没有直接对应。如果由公理(17.9-a)，(17.9-c)产生的所有理论，以 NATURALS 的公理均可证明，这是允许的。例如，利用归纳法可以证明(在 NATURALS 中)：

$$\text{succ } 0 \text{ is succ } 0 \equiv \text{true}$$

而公理(17.9-b)也可以产生理论：

$$\text{succ } 0 \text{ equals succ } 0 \equiv \text{true}$$

转换后一样。同理，有了以上证明，则有：

$$\begin{aligned}
 &(\text{succ } 0 \text{ is succ } 0) \wedge (\text{succ } 0 \text{ is succ } 0) = > (\text{succ } 0 \text{ is succ } 0) \\
 &\text{true} \wedge \text{true} \Rightarrow \text{true} && \text{由以上证明} \\
 &\text{true} \Rightarrow \text{true} && \text{由(17.5-c)} \\
 &(\text{not true}) \vee \text{true} && \text{由(17.5-f)} \\
 &\equiv \text{true}
 \end{aligned}$$

如果在本例中把自变量域和函数映倒过来，从 TRUTH_VALUES 映射为 NATURALS，利用这个规格说明就不对了：

```

Instantiation of MAPPINGS by TRUTH_VALUES, NATURALS
  using Natural for Range,
    Truth_Value for Domain,
  ∨ for equals

```

该实例是不正确的，因为它不满足由(17.9-b)和(17.9-c)产生的等式。例如，由 TRUTH_VALUES 中的等式(17.2-g)可得：

$$\text{false} \vee \text{false} \equiv \text{false}$$

它与(17.9-b)式相矛盾。

17.4 λ 演算的代数规格说明

λ 演算是个小语言，常用以研究数据类型，把它作为代数规格说明的例子，更能说明代数规

格说明的表达能力。

我们在第 11 章介绍过 λ 演算的语法。

$$\begin{aligned}\lambda_表达式 &::= \text{标识符} \\ &\quad | \lambda_标识符.表达式 \\ &\quad | 表达式 \ 表达式 \\ &\quad | (\lambda_表达式)\end{aligned}$$

此外，还介绍了 λ 演算的 α ， β ， η 三种归约。如果我们定义一个置换函数 $\text{sub}(M, a, b)$ 表示 a 在表达式 M 中所有自由出现均以 b 置换，则三种归约描述为：

$$\begin{aligned}\alpha \quad & \text{if } w \text{ is not free in } M \text{ then } (\lambda u. M) = (\lambda w. \text{sub}(M, u, w)) \\ \beta \quad & ((\lambda x. M)N) = \text{sub}(M, x, N) \\ \eta \quad & \text{if } x \text{ is not free in } M \text{ then } (\lambda x. (Mx)) = M\end{aligned}$$

我们先定义 λ 演算的类别：1d(标识符)、Expr(表达式)、Truth_Value(由 TRUTH_VALUES 扩充) 其中， λx 和 x 都是标识符， λ 不显式出现。由操作 `firstid` 和 `nextid(x)` 区分。

再定义操作：`equals` 比较两标识符是否同样出现；`var` 将标识符转换为表达式类别，`app` 为构造 λ 应用表达式；`abs` 为构造 λ 函数表达式；此外，还有两个隐式算子：`sub` 转换；`notfree` 查询标识符是否是自由出现。有了这些操作在限定公理下可定义所有 λ 演算的项集及其行为。其代数规格说明如下：

specification LAMBDA_CLACULUS

include TRUTH_VALUES

sorts Expr, 1d

operations

$$\begin{aligned}\text{firstid} & \rightarrow 1d \\ \text{nextid_} : 1d & \rightarrow 1d \\ \text{equals}(_,_): 1d, 1d & \rightarrow \text{Truth_Value} \\ \text{var_} : 1d & \rightarrow \text{Expr} \\ \text{ap}(_,_): \text{Expr}, \text{Expr} & \rightarrow \text{Expr} \\ \text{abs}(_,_): 1d, \text{Expr} & \rightarrow \text{Expr} \\ * \text{sub}(_,_): \text{Expr}, 1d, \text{Expr} & \rightarrow \text{Expr} \\ * \text{notfree}(_,_): 1d, \text{Expr} & \rightarrow \text{Truth_Value}\end{aligned}$$

variables

$v, w, x, y: 1d$

$M, N, E: \text{Expr}$

equations

$$\begin{aligned}\text{equals}(x, x) &= \text{true} \\ \text{equals}(\text{firstid}, \text{nextid}(x)) &= \text{false} \\ \text{equals}(\text{nextid}(x), \text{firstid}) &= \text{false} \\ \text{equals}(\text{nextid}(x), \text{nextid}(y)) &= \text{equals}(x, y) \\ \text{notfree}(x, \text{var}(y)) &= \text{not equals}(x, y) \\ \text{notfree}(x, \text{app}(M, N)) &= \text{notfree}(x, M) \wedge \text{notfree}(x, N) \\ \text{notfree}(x, \text{abs}(y, M)) &= \text{equals}(x, y) \vee \text{not free}(x, M) \\ \text{abs}(v, M) &= \text{if notfree}(w, M) \\ &\quad \text{then abs}(w, \text{sub}(M, v, \text{var}(w))) \\ &\quad \text{else abs}(v, M) && // \alpha \text{ 归约} \\ \text{app}(\text{abs}(_, x, M), N) &= \text{sub}(M, x, N) && // \beta \text{ 归约}\end{aligned}$$

```

abs(x , (app(M , var(x)))) = if notfree(x , M)
                               then M
                               else abs(x , app(M , var(x)))           // η 归约
sub(var(y) , x , E)      =   if equals(x , y)
                               then E else var(y)
sub(app(M , N) , x , E) = app(sub(M , x , E) , sub (N , x , E))
sub(abs(y , M) , x , E) =   if equals(x , y)
                               then abs(y , M)
                               else if notfree(y , E)
                                     then abs(y , sub(M , x , E))
                                     else sub(abs(y , M) , x , E)

```

end specification

17.5 小结

本章我们介绍了代数语义学，目标是以代数理论建立描述程序规格说明的代数模型，并介绍具体的定义方法。

- 代数按不同的抽象层次有具体代数(定义值集，操作集)、抽象代数(值和操作关系)和泛代数(代数间关系)。
- 代数间关系最重要是同态映射，甲代数对其承载子操作后可映射为乙代数某个操作对先映射的承载子操作。同构是两代数有对等的同态映射。同构一般认为一致。
- 交换图可清晰描述范畴中代数对象的映射关系。
- Σ _代数是给定型构 Σ 上的代数，即在 Σ 前题下研究它们的映射关系。
- Σ _项代数是所有承载子均由 Σ 中操作 σ 复合的 Σ _代数。正因为如此，它构成描述的数据类型的语法。
- Σ _同态、同构指项代数上的。可以证明它的存在性和唯一性。因此，若用一字代数描述的 Σ 项代数，则它是其它某代数的唯一同态。即可得出无歧义的语义解释。
- 给定一 Σ 必然得到一类代数，初始代数即对代数类中每一 Σ _代数都有唯一 Σ 同态的代数。初始代数也是项代数。
- 初始代数就符号意义是不重复的，但其实际值可能相同，因此引出全等类概念。 Σ _全等工具是说 Σ 代数的承载子有等价关系 R ，按 R 归类即得全等类。
- 按全等的等价关系 R 归类过程称商化，商化结果得的值集构成商代数，商代数可以看作是简化了的字代数，并包含全等概念。
- 泛同构映射指商代数寻求同构的映射。
- Sp _代数可以是字代数上某个全等关系，或字代数上某个商代数描述。目的是前者，以等式集合给出全等类的公理定义。
- 全等关系具有自反，对称，传递性，置换性，按公理集可生成所有类型值的表达式。满足此公理集不止一种代数，但它们泛同构，语义一致。
- 隐式算子是规格说明中为方便简化描述设定的操作。由于不可达不影响原 Sp _代数描述。
- 为继承已有规格说明，新规格说明在原有的上面扩充和增强。新增类别及操作不得影响原有类别与操作，引出 S _完整， S _一致，充分完整，一致和保留的严格定义。