第一章　软件规约方法和语言

软件规约是对软件所应满足之需求的描述，其主要特征是软件需求描述的清晰性而非其功效性。根据描述软件规约所采用的语言，软件规约可分为非形式软件规约和形式软件规约。

非形式软件规约通常指用自然语言书写的软件规约，主要用于描述软件开发者与软件用户之间的协议和系统开发文档。非形式规约具有易书写、易理解和易使用等特点，但其语义的歧义性和描述的不完备性不便于软件的自动生成。

形式软件规约是用形式语言书写的软件规约。形式语言是指其语法和语义均为显示和精确定义的语言，例如，一阶谓词就是一种形式语言。形式语言避免了自然语言的歧义性和描述的不完备性，奠定了能保证程序正确性的各种形式化方法，如 E.W.Dijkstra 的 WP方法、C.B.Jones 的 VDM方法等的基础。形式软件规约具有良好的数学基础，易于研究软件规约的性质，如规约的一致性、完备性和规约之间的等价性等。形式软件规约是软件开发的基础，并且有利于软件的自动生成。

鉴于软件规约的重要性和难度，国际上对软件规约方法和语言进行了广泛的研究。在软件规约方法方面，代表性的工作有基于逻辑演算的前后断言方法、基于控制公理的高阶方法（HOS 方法）、基于异调代数的代数方法和基于抽象模型的方法等；在规约语言方面，代表性的工作有 Z 语言、LARCH语言、CIP－L语言和 AXES 语言等。

本章主要介绍几种典型的软件规约方法。

1.1 基础知识

1.1.1 层次级别与描述手段

按软件开发过程的不同阶段划分，软件规约分为需求规约、功能和性能规约以及设计规约。软件需求规约从用户的角度描述了对所需软件的各种需求，如功能需求和性能需求等，用以书写软件需求规约的语言称为需求规约语言。软件功能和性能规约通常基于抽象的数学模型给出了软件“做什么”的功能以及在速度和资源使用等方面的限制，用以书写软件功能和性能规约的语言称为软件功能规约语言和性能规约语言。软件设计规约则从可执行的角度刻划了软件的数据结构和抽象算法，用以书写软件设计的语言称为软件设计规约语言。

根据上述分类，软件自动生成可分为三个层次。第一层次是从软件需求规约到软件功能和性能规约的自动转换，其主要问题是从“非形式”到“形式”的过渡 第二层次是从软件功能和性能规约到软件设计规约的自动转换,其主要问题是从“做什么”到“如何做”的过渡；第三层次是软件设计规约到高级语言的自动转换，其主要问题是如何高效实现。

R.Balzer等人对从“非形式”到“形式”的自动转换问题进行了初步探讨,构造了 SAFE 系统，该系统旨在从非形式的规约中获取形式的规约，提供了解决非形式规约中歧义性的机制，当歧义不能自动消解时，由用户进行干预。SAFE系统所能接受的非形式化语言是一种受限的自然语言。从“非形式”到“形式”的自动转换难度颇大，受囿于自然语言理解等技术，目前仅限于具体领域的受限自然语言。自动转换的主要工作集中于形式软件功能规约语言和软件设计规约语言的研究，以及从功能规约到设计规约的自动转换。

代表性的形式软件规约语言有 AXES语言、LARCH 语言、CIP-L语言、Z 语言等，主要的实现途径包括演绎综合、程序转换、归

纳综合和过程实现。

1.1.2 过程抽象和数据抽象

过程抽象和数据抽象是软件开发中十分重要的两类抽象。过程抽象是指忽略任务具体完成的过程，而只精确描述该任务所要完成的功能，即指从输入值集到输出值的映射，其定义域和值域均由数据抽象刻划。数据抽象是用数据集上的运算来表示数据，体现了表示上的抽象,即利用抽象的数学结构,如集合、关系、函数、序列和多重集合等，进行功能性描述，而不关心这些抽象数学结构在计算机中是如何具体表示和实现的。这样就使软件开发者能针对问题选用合适的表示结构，而不是在问题尚未考虑成熟之前就用计算机语言中的具体的数据结构加以表示。

过程抽象是对传统高级程序语言中过程和函数的抽象。值得注意的是，尽管高级语言程序中的某些函数或过程存在副作用（即在不同的上下文，对同一输入会产生不同的输出），但仍然可将其看作映射。这是因为它们含有隐式的输入输出的缘故。如果将过程或函数的所有输入输出均以显式刻划，则所有过程或函数均可用过程抽象加以刻划。例如：随机数生成器含有一状态变量 $seed$,每产生一随机数后,便修改 $seed$,以产生下一个输出,如果将 $seed$ 的隐式输入输出加以显式表示，则随机数生成器可以表示成这样的映射：$seed \rightarrow seed \times value$.

基于过程抽象的规约方法之主要问题是如何定义输入值集到输出值的映射，即如何刻划过程抽象的功能，不同的刻划方法决定了相应的不同层次级别。主要级别有二，即功能级和设计级。功能级规约方法刻划了相应过程抽象“做什么”的功能而不涉及“如何做’的算法,而设计级规约方法则刻划了相应过程抽象‘如何做”的算法而不涉及实现的功效。

数据抽象通常由一个或多个相关的抽象数据类型构成。

抽象数据类型是模块化程序设计中十分重要的概念，它是封装原理和信息隐蔽原理的集中体现。一方面，抽象数据类型将数据及其上的运算视为一个整体，即以数据类型作为模块化的基本单位来直接描述现实世界中的对象；另一方面，它要求严格区分数据类型的内外性态。外部可见的只是数据类型上的操作及其性态，而数据的内部表示则是隐蔽的，从而达到抽象之目的。

高级程序设计语言如 Pascal、Algol 等仅提供了一些标准的数据类型(如整型、实型、字符型等)及一些固定的结构类型(如数组型、记录型等),而对问题领域中出现上述类型以外的对象,则必须用上述标准的数据类型和固定的结构类型加以模拟，无法直接定义反映问题性质的新对象。例如，编译程序中常用的符号表类型和栈类型，常被实现成记录数组，而符号表类型和栈类型中固有的运算被实现成记录或数组上的运算且常常分布于程序的各个部分，从而使程序中的类型与问题领域中的对象缺乏直接的对应关系。其结果是使程序结构复杂，难以理解，难以验证和难以维护。因此，有必要研究基于抽象数据类型的各种规约方法。

基于抽象数据类型的规约方法之关键问题是如何体现抽象数据类型与表示无关的特征，主要途径有公理化途径和模型化途径。公理化途径用公理方法来刻划抽象数据类型的性质而与具体表示无关；模型化途径通过给出定义抽象数据类型的抽象模型而达到与具体表示无关之目的。

1.2 规约方法

过程抽象和数据抽象是软件说明中的两个主要抽象，本节主要讨论基于过程抽象和数据抽象的几种有代表性的形式规约方法。

基于过程抽象的规约由两部分构成：第一部分是接口描述，它定义了过程抽象的名、定义域和值域 第二部分是映射描述,它

定义了从输入值集到输出值集的映射，两种有代表性的刻划方法是前后断言方法和 HOS方法。

基于数据抽象（或抽象数据类型）的规约方法主要有两种：一种是代数方法，它将抽象数据类型看作异调代数，用抽象代数中的公理化方法来刻划抽象数据类型中运算的性质而不涉及数据的具体表示；另一种是模型方法，它基于某些已知的抽象数据类型来给出待定义抽象数据类型的抽象模型，用抽象模型来刻划待定义的抽象数据类型中运算的功能。

下面分别介绍基于过程抽象的前后断言方法、HOS 方法和基于数据抽象的代数方法、模型方法。

1.2.1 前后断言方法

前后断言方法通过给出基于一阶谓词演算的前后断言，来刻划过程抽象输入输出之间的关系。由于前后断言方法只涉及过程抽象的输入与输出的性质而与具体算法无关。因此，它是功能级的。

前后断言方法源于 P.Naur、R.W.Floyd、C.A.R.Hoare 和 E.W.Dijkstra 等人的工作，C.A.R.Hoare 在 P.Naur 和 R.W.Floyd 等人的工作基础上，引入了前后断言方法，提出了定义程序设计语言语义的公理化方法，奠定了程序正确性形式证明的逻辑基础。其工作的主要不足在于：

(1) 只能保证程序的部分正确性

(2) 程序的设计和验证是分离的

E.W.Dijkstra 提出了能够保证程序完全正确性的最弱前置条件的概念，以及相应的程序设计演算，使程序设计和程序正确性验证可同时进行。从而使得前后断言方法不仅可作为程序正确性验证的基础，同时也可作为软件开发的依据。

采用前后断言方法,过程抽象的功能规约用二元组表示,即表示成 $P(F) = < IE_F, SP_F >$,其中 IE_F 是过程抽象 F 的接口描述，它刻划过程抽象与外界的接口,SP_F 是过程抽象的功能描述，它刻划了过程抽象‘做什么’的功能。

接口描述 $IE_F = < Y = F(X), DO_F, RA_F >$,其中 F 是过程抽象的名,X 是输入变元组,Y 是输出变元组,即 $X = < x_1, \cdots x_m >$, $Y = < y_1, \cdots y_n > (m, n \geqslant 1)$。$DO_F$ 的一般形式是 $D_1 \times \cdots \times D_m$, 每一 $D_i (1 \leqslant i \leqslant m)$是抽象数据类型的值集,用以刻划相应的输入变元 x_i; RA_F 的一般形式是 $R_1 \times \cdots \times R_n$,每一 $R_j (1 \leqslant j \leqslant n)$ 是抽象数据类型的值集，用以刻划相应的输出变元 y_j。

功能描述 $SP_F = < IC_F(X), OC_F(X, Y) >$,其中 $IC_F(X)$是一阶谓词,表示 DO_F 上的关系,称为前断言,DO_F 中凡满足前断言的输入值称为合法输人；$OC_F(X, Y)$也是一阶谓词,表示 $DO_F \times RA_F$ 上的关系，称为后断言。给定合法输人 X', RA_F 中凡满足 $OC_F(X, Y)$的 Y'称为关于 X'的合法输出。

过程抽象的功能规约刻划了这样的映射：对于任一合法输人，此映射所产生的输出均为合法。而对于任一非合法的输入，对其输出不作规定。值得注意的是，由于引入了前断言的概念，从而使得前后断言方法具有显式刻划部分映射的能力。

例：求整数集合最大值的函数的规约可定义如下：

$$P(max) = (IE_{max}, SP_{max})$$

其中,$IE_{max} = < y = max(s), \text{SET OF INTEGER}, \text{INTEGER} >$

$$SP_{rmax} = < s \neq \phi, y \in s \wedge y \geqslant \text{ALL}(s) >$$

接口描述 IE_{max}定义了函数的使用接口，指出其参数类型和结果类型分别是整数集合型和整型。功能描述 SP_{max}刻划了函数的功能。前断言 $s \neq \phi$ 表示 max 是部分函数，它仅作用于非空的整数集合；后断言说明此函数的输出属于 s 并且大于或等于 s 中的所有元

素。由于前后断言方法是功能级的，因此，其规约结构通常与最终算法的结构无直接对应关系。例如，$y \in s \wedge y \geqslant \mathrm{ALL}(s)$通常不能简单地分解成两个部分来求解，这是因为对任何确定的 s,满足 $y \geqslant \mathrm{ALL}(s)$ 的 y 构成一无穷集合。

基于前后断言方法的过程抽象功能规约的定义给出了前后断言方法的主要内容，但要注意的是，在不同的场合下，其具体表示方法可能会有所不同。例如，Z.Manna 和 R.Waldinger 在其程序综合方法中采用如下的表示形式：

$$max \leftarrow \textbf{find}\ y \bullet \text{such that}\ y \in s \wedge y \geqslant \mathrm{ALL}(s)$$
$$\textbf{where}\ s \neq \phi$$

在 VDM 开发方法中，采用如下形式：

$$max(s: \mathrm{SET\ OF\ INTEGER})\ y: \mathrm{INTEGER}$$
$$\mathrm{PRE}\ s \neq \phi$$
$$\mathrm{POST}\ y \in s \wedge y \geqslant \mathrm{ALL}(s)$$

可以看出,各种表示只是形式不同,并无本质差别。

归结起来，前后断言方法用作过程抽象的规约具有以下特点：

第一，从功能级上对过程抽象加以刻划，其描述方式接近人的思维方式，便于用户的书写和理解。例如求平方根函数的用户往往只关心最后结果的性质而不关心具体的算法。因此，该规约可用前后断言方法直接描述如下：

$$P(root) = (IE_{root}, SP_{root})$$
$$IE_{root} = < y = root(x), \mathrm{REAL}, \mathrm{REAL} >$$
$$SP_{root} = < x \geqslant 0, y^2 = x >$$

第二，能够显式表达过程抽象的前提条件。如上例中，前断言 $x \geqslant 0$ 显式刻划了求平方根函数只能对大于或等于零的输入求值。

第三，规约的后断言常常给出的是输出值集而非具体的值，从而体现出一种非确定性。例如：在求平方根函数的规约中，给定 $x = 4$,那么 ± 2 均满足后断言。如果用户只想要一个确定的输出,则需在后断言中附加约束条件,如 $y > 0$.

第四，在用谓词公式表达所希望的输出的性质时，应注意相应输出的存在性。如上例中当 $x = 2$ 时,在机器允许的范围内,不存在精确的解,因此,其后断言应适当放宽,如变为 $| y^2 - x | < 10^{-3}$。

第五，前后断言方法是基于一阶谓词演算的形式规约方法，易于研究功能规约的性质，如规约的一致性、完备性、规约间的等价性、可满足性以及规约与其解之间的关系。

正是由于前后断言方法的上述特点，它构成了各种软件规约语言、形式软件开发方法、以及软件自动生成方法的基础。需要指出的是，前后断言方法也有其局限性，它受限于所能使用的谓词，对于特例，有些问题本身的定义是算法性的，如阶乘函数的定义是：

$$f(n) = n \times f(n-1)$$
$$f(0) = 1$$

对于这类问题，前后断言方法就显得不太合适，有关问题及其解决方法将在其后讨论。

1.2.2 HOS 方法

早在 1960 年代,J.McCarthy 就已讨论了基于过程抽象的设计规约。他所用的形式语言十分简单，仅使用了条件和递归表达式。例如，求最大公约数问题的设计规约可表示如下：

interface: gcd(**integer**, **integer**) **returns integer**
Behavior: $gcd(x, y)$ = **if** $x \leqslant 0 \vee y \leqslant 0$
then error("unexpected input")

else *search* *from* $(x, y, \min(x,y))$

Abbreviations : *search* *from* $(x,y,z) =$

if $\mathbf{mod}(x, z) = 0$ & $\mathbf{mod}(y, z) = 0$ **then** z

else *search* *from* $(x,y,z-1)$

上述规约方法有两点不足，其一是难以描述较大问题的规约，因为这一方法未曾涉及大型软件规约中的基本问题，特别是接口的正确性问题；其二是这一方法的描述方式是纯正文形式的，不易于理解。HOS 方法在这两方面均颇具特色。

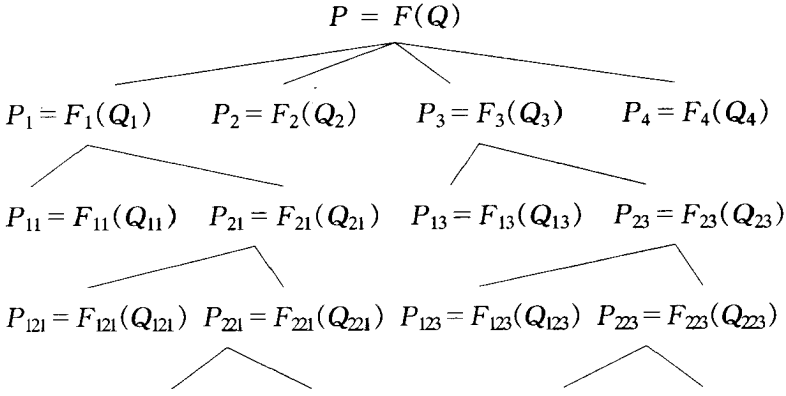
HOS方法也是基于过程抽象的设计规约方法，其主要特征是以数学函数作为过程抽象的数学模型，用树形结构对函数加以刻划。它能清晰地表达函数“如何做”的算法而不涉及与机器有关的实现细节，其描述方式符合结构程序设计的思想：层次分明，形象直观。特别是，其所用控制结构基于一组基本的形式规则，且这组形式规则对用户透明，从而既未给用户增加过重负担，又可避免软件开发中常见的逻辑接口错误。下面对此作详细介绍。

上述的形式规则又可称为公理，因此 HOS 方法也可以称之为软件规约的一种公理化方法。可以将这一方法与集合论中的公理化方法作一类比。众所周知，集合论的发展在本世纪初遇到了很大困难，即发现了“悖论”。以后不久产生了公理集合论。在公理集合论中，已知的“悖论”(如 Russell 悖论等)不会出现。因此，集合论的公理方法实际上是一种排除公理系统中某些类型的逻辑错误的方法。HOS 方法学中的公理方法则是用来排除软件规约中某些类型的逻辑错误的方法。

首先定义一个控制系统，在其中将控制的各种逻辑可能性表示成一个树结构。一个函数 $F: Q \rightarrow P$ (或 $P = F(Q)$)是输入集 Q 到输出集 P 的一种映射。为了在逻辑上实现一个函数的计算，必须有一个控制元。每一控制元在树结构中存在于所控制的函数的高一级的结点上，它仅控制比它低一级的函数。控制元和函数的定义是

相对的,较高一级的函数对较低一级的函数而言构成控制元。

定义(层次式控制系统)函数按树型分解的形式化控制系统,是如下的一个层次式控制系统。



其中每个控制元有唯一的标识 Sn_im_i:

$$Sn_im_i \equiv [Pn_im_i = Fn_im_i(Qn_im_i)]$$

n_im_i定义了一个特定的控制层,i 是该控制元的嵌套层次。$i = 1$ 是指仅低于顶层的控制层。n_i 是相对它的紧高结点 m_i 的结点位置(从左向右由 1 开始编号。若 $i > 2$, m_i 就按如下递归关系得出:

$$m_i = n_{i-1}m_{i-1}$$

如果 $i = 2$,则 $m_i = m_{i-1}$;如果 $i = 1$,则 $n_im_i = n_i$。

公理 1 控制元 Sn_im_i 控制且仅控制紧低层的有效函数集合 $\{Fn_{i+1}n_im_i\}$的调用。

公理 2 控制元 Sn_im_i 对且只对输出空间 Pn_im_i 中元素的生成进行控制,使得 $Fn_im_i(Qn_im_i)$的结果就是 Pn_im_i。

公理 3 控制元 Sn_im_i 对变量集合 $\{Yn_{i+1}n_im_i\}$ 的存取权进行

控制，该变量集合的值定义且只定义了紧低层函数的输出空间元素。

公理 4 控制元 Sn_im_i 对变量集合 $\{Xn_{i+1}n_im_i\}$ 的存取权进行控制，该变量集合的值定义且只定义了紧低层函数的输入空间元素。

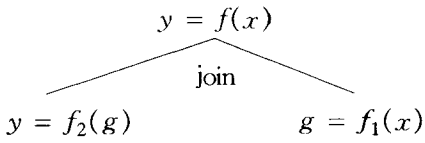
公理 5 控制元 Sn_im_i 只对它自己输入集合 Qn_im_i 里无效元素(即在紧低层中不出现的元素)的拒收进行控制。

公理 6 控制元 Sn_im_i 对紧低层的每棵子树 $\{Tn_{i+1}n_im_i\}$ 的次序关系进行控制。

满足上述六条公理的控制系统具有良好的性质。例如，转移语句 goto 与公理 1 是不一致的。如果有 C goto D 存在，那么 C 便失去控制，也就是 C 能控制它自己终结，与公理 1 不符。

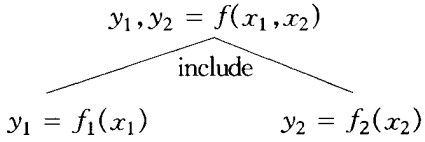
M. Hamilton 和 S. Zeldin 根据自己多年来从事软件开发的经验，导出了满足上述六条公理的三个基本控制结构："join"结构、"include"结构和'or'结构，并设计了软件设计规约语言 AXES。

(1) join 结构如下：



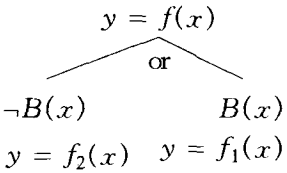
其含义是子函数 f_1 的输入恒同于父函数 f 的输入，子函数 f_2 的输入恒同于子函数 f_1 的输出；子函数 f_2 的输出恒同于父函数 f 的输出；其执行过程是先执行 f_1，再执行 f_2.

(2) include 结构如下：



其含义是子函数 f_1 的输入与输出分别恒同于父函数 f 的输入与输出的第一部分 子函数 f_2 的输入与输出分别恒同于父函数 f 输入与输出的第二部分；f_1 和 f_2 均需执行。

(3) or 结构如下：



其含义是,子函数 f_1 和 f_2 的输入均恒同于父函数 f 的输入,子函数 f_1 和 f_2 的输出均恒同于父函数 f 的输出；其执行是根据 $B(x)$ 的值从和中择一执行。

用这三个基本控制结构进行的函数功能分解具有以下性质：

(1) 等价分解性质：每个子函数都是必须的，并且所有子函数在该控制系统中作用的总和恰好完成父函数的功能。

(2) 单一赋值性质：每一函数在计算中对同一个变量只能赋值一次。

(3) 单一引用性质：每个子函数对父函数输入变量只能引用一次。

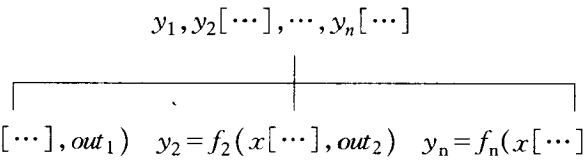
(4) 引用与赋值的互斥性质：同一函数不能对某变量既引用又赋值。

但是，由于这三个结构对子函数间的数据依赖关系限制较严，用起来较为不便。M.Hamilton 和 S.Zeldin 在上述基本结构的基础上给出了三个导出结构："cojoin"结构、"coinclude"结构和"coor"结构，从而使得变量的存取和重复引用等方面具有较大的灵活性，使用起来也较为方便。

AXES 语言的设计规约机制的主要不足在于，它对于函数间的数据传递和通信方式过严，使用欠方便；而且，所基于的二叉树形

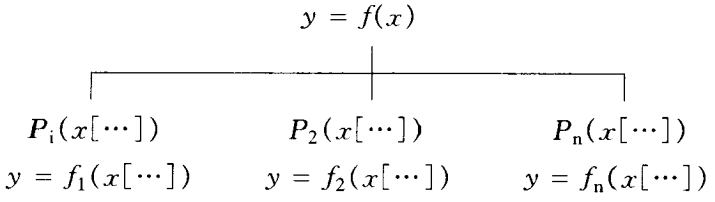
结构对较大软件的分解并不合适。基于 HOS 方法学的六条公理，可以定义两个功能颇强的基本控制结构，它们分别称为“PARTITION”结构和“CASE”结构。

设 $s[\ \wedge\ ]$ 表示数据集 s 的某个子集（可以为空集）。“PARTITION”结构的一般形式如下：



其中，父函数中的 $y_1, y_2[\cdots], \cdots, y_n[\cdots]$ 表示函数 f 的输出结果，$out_i(1 \leqslant i \leqslant n-1)$ 表示集合 $\mathop{\mathbf{Y}}_{j=i+1} yi$ 的子集(可以为空集)。该控制结构的含义是：函数 f 可以分解为 n 个子函数 $f_i(1 \leqslant i \leqslant n)$，每个子函数可以以 f 的输入和处于其右的子函数的输出作为自己的输入，并将各自的输出提供给 f 作为其输出的一部分，或者提供给位于其左的子函数作为输入。这样，n 个子函数协同作用的结果将等价于函数 f。显然，该控制结构隐含了有数据依赖关系的子函数从右到左的执行顺序，否则，f 的功能分解与子函数间的次序无关。

“CASE”控制结构的一般形式为：



其中控制条件 $P_i(x[\cdots])$（$1 \leqslant i \leqslant n$）两两互斥，并且任一时刻必定有一个取真值。它的含义是：在条件 $P_i(x[\cdots])$ 成立的情况下，函数

f 的功能等价于子函数 $f_i(x[\cdots])$的执行。此外，条件 $P_n(x[\cdots])$还可以用 OTHERS 来取代，它表示其他条件均不成立的情况。

基于上述结构，过程抽象设计规约可定义如下：

过程抽象设计规约 $A(F)$是一二元组，即 $A(F) = < IE_F, SD_F >$，其中 IE_F是过程抽象 F 的接口描述，它刻划了过程与外界的接口；SD_F 是过程抽象 F 的算法描述，它刻划了“如何做”的具体算法。

$IE_F = < Y = F(X), DO_F, RA_F >$，其含义同函数功能规约接口描述；

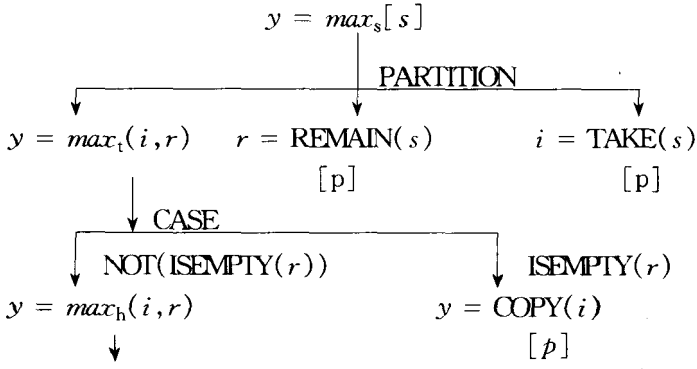
SD_F 可递归定义如下：

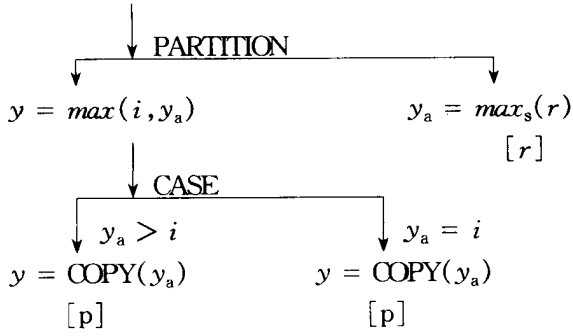
(a) SD_F 是基本运算标记 p，或

(b) SD_F 是递归标记 r，或

(c) SD_F 是 PARTITION$(A(F_1), \cdots, A(F_n))$或 CASE $(P_1, \cdots, P_n)$ $(A(F_1), \cdots, A(F_n))$；其中 $n \geqslant 2$并且每一 $P_i(1 \leqslant i \leqslant n)$是布尔表达式以及每一 $A(F_i)$是子过程抽象 F_i 的设计规约。另外，$A(F)$对输入 X 的作用记为 $A(F)(X)$。

为使过程抽象设计规约易于理解，可将之用树形结构表示。例如，求整数集合最大值函数的设计规约可表示如下：





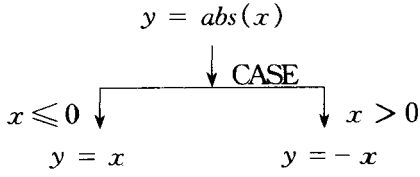
其中，i, y：INT；r, s：SET(INT)。

与前后断言方法相比，HOS方法的不足是算法仍需由人设计，机器只能检查软件规约的接口正确性，然而HOS方法在权衡软件规约的理论基础和易使用性方面进行了有益的探讨，它既为相应规约提供了一定的理论基础，又将之对一般的程序人员隐蔽，从而较好地满足了软件自动化的要求。

但是，在使用基于HOS方法的过程抽象设计规约时，有两个问题值得注意：

第一，尽管设计规约表达了过程抽象‘做什么’的过程，但是它与传统的高级语言程序却有本质区别，即注重清晰性而非功效性。

第二，HOS方法学中的六条公理只能保证相应软件规约的接口正确性，并不能保证其语义的正确性。例如，下述求整数绝对值的规约：



尽管此步分解满足HOS方法学的六条公理，但显然CASE($y = x$ |

$x<0, y=-x \mid x>0$)并不满足 $y=abs(x)$的语义功能。

1.2.3 代数方法

基于抽象数据类型描述的代数方法能够完整地刻划抽象类型的性质而与表示完全无关，从而可较好地体现抽象数据类型的特点。

抽象数据类型描述的代数方法基于 G.Birkhoff、J.D.Lipson 等人的异调代数理论,经 S.Zills、J.A.Goguen 和 J.V.Guttag 等人的发展，其理论基础日趋完备，并逐步应用于软件工程实践，成为有代表性的抽象数据类型的规约方法之一。鉴于此，许多著名的软件规约语言,如 CIP-L、LARCH 等均采用代数方法作为软件规约的手段。

抽象数据类型描述的代数方法基于下述观点：

(1) 数据类型是异调代数

同调代数是两元组(C, F),其中 C是非空值集,F 是运算 $F_{j,n}$ - $C^n \rightarrow C$ 的有限集合。

异调代数是同调代数概念的拓广，它是两元组(V, F),其中 V 是由非空集合 V_i 构成的集合族，F 是运算 $F_{j,n}$: $Vi_1 \times Vi_2 \times \cdots \times Vi_n \rightarrow V_k$(其中 $\forall 1 \leqslant h \leqslant n(Vi_h \in V$ 且 $V_k \in V)$的有限集合。一般说来，数据类型可定义成值集及其上运算的集合。因此，可自然地将之看成代数。

例：integer 类型的值集为 $\{0, \pm 1, \pm 2, \cdots\}$,其运算符为：

Add: **Integer** × **Integer** → **Integer**

Sub: **Integer** × **Integer** → **Integer**

Equal: **Integer** × **Integer** → **Boolean**

尽管所感兴趣的值集是 $\{0, \pm 1, \pm 2, \cdots\}$,但在其上所定义的运算的值域可能超出上述值集。因此，我们可把类型 Integer 处理

成异调代数。

(2) 抽象数据类型是一类代数

进一步，为了刻划抽象数据类型与表示无关的特性，可将抽象数据类型看作一类异调代数。在此类异调代数中，尽管各个异调代数可以有不同的值的表示，但其外部可见的运算却具有共同的抽象性质。这类似于近世代数中的抽象代数系统。例如，群就是一类代数系统，其具体表示各不相同，但其运算均需满足群公理。因此，从抽象数据类型规约的角度，运算就有了更加本质的作用，即可通过对各类运算性质的刻划来达到隐蔽数据内部表示的目的。

1.2.3.1 代数规约的构成

代数方法中抽象数据类型的规约 ADT-SP 由两部分组成：一是语法部分 Syntax，二是公理部分 Axiom。

语法部分给出了所定义抽象数据的名及其上所有运算的定义域和值域；同时，它也给出了与之相关的其他类型名。形式上，Syntax = (S, OP)。其中，S 是类型名的非空有限集合；OP 是常量符和运算符的有限集合，具体说来，OP 是下述子集的并集。

$$K_s: \{k: \rightarrow s \mid s \in S\}$$

$$OP_{w,s}: \{op: w \rightarrow s \mid w \in S^+, s \in S\}$$

例如，整型栈 I_{stack} 的语法可表示如下：

Type I_{stack};
Based on Boolean, Integer;
Introduce

New: $\rightarrow I_{stack}$;
$Push$: $I_{stack} \times$ **Integer** $\rightarrow I_{stack}$;
Pop: $I_{stack} \rightarrow I_{stack}$;
$Read$: $I_{stack} \rightarrow$ **Integer** $\cup \{$**error**$\}$;