国 际 电 信 联 盟

CCITT

国际电报电话咨询委员会

红 皮 书

卷Ⅵ.10

功能规格和描述语言(SDL)

建议Z.100-Z.104

第 八 次 全 体 会 议

1984年10月8—19日 马拉加—托雷莫里诺斯

1986年 北京

国际电信联盟

CCITT

国际电报电话咨询委员会

红皮书

卷Ⅶ.10

功能规格和描述语言(SDL)

建议Z.100-Z.104

第八次全体会议

1984年10月8—19日　马拉加—托雷莫里诺斯

1986年北京

ISBN 92-61-02235-9

CCITT图书目录

适用于第八次全体会议（1984年）以后

红　　皮　　书

卷 Ⅵ.10 - 功能规格和描述语言（SDL）。建议Z.101-Z.104（第11研究组）。

卷 Ⅵ.11 - 功能规格和描述语言（SDL）。建议Z.101-Z.104的附件（第11研究组）。

卷 Ⅵ.12 - CCITT高级语言（CHILL）。建议Z.200（第11研究组）。

卷 Ⅵ.13 - 人机语言（MML）。建议Z.301-Z.341（第11研究组）。

卷Ⅶ - （3个分册，按册出售）

卷 Ⅶ.1 - 电报传输。R系列建议（第9研究组）。电报业务终端设备。S系列建议（第9研究组）。

卷 Ⅶ.2 - 电报交换。U系列建议（第9研究组）。

卷 Ⅶ.3 - 远程信息处理业务的终端设备与协议。T系列建议（第8研究组）。

卷Ⅷ - （7个分册，按册出售）

卷 Ⅷ.1 - 电话网上的数据通信。V系列建议（第17研究组）。

卷 Ⅷ.2 - 数据通信网：业务和设施。建议X.1-X.15（第7研究组）。

卷 Ⅷ.3 - 数据通信网：接口。建议X.20-X.32（第7研究组）。

卷 Ⅷ.4 - 数据通信网：传输、信号和交换；网路问题；维护和行政安排。建议X.40-X.181（第7研究组）。

卷 Ⅷ.5 - 数据通信网：开放系统的相互连接（OSI），系统描述技术。建议X.200-X.250（第7研究组）。

卷 Ⅷ.6 - 数据通信网：网路间的互通，移动数据传输系统。建议X.300-X.353（第7研究组）。

卷 Ⅷ.7 - 数据通信网：信息处理系统。建议X.400-X.430（第7研究组）。

卷Ⅸ - 干扰的防护。K系列建议（第5研究组）。电缆的建筑、安装和防护以及外线设备的其他组成部分。L系列建议（第6研究组）。

卷Ⅹ - （2个分册，按册出售）

卷 Ⅹ.1 - 术语和定义。

卷 Ⅹ.2 - 红皮书索引。

红 皮 书 卷 Ⅶ.10 目 录

建议Z.100至Z.104

功能规格和描述语言(SDL)

卷 首 说 明

1 在1985—1988研究期内委托给各研究小组的研究课题在给各研究小组的第1号文献中。

2 本卷中，为简便起见,“主管部门”一词既代表电信主管部门，也代表经认可的私营机构。

卷Ⅵ.10

建议Z.100至Z.104

功能规格和描述语言(SDL)

PAGE INTENTIONALLY LEFT BLANK

PAGE LAISSEE EN BLANC INTENTIONNELLEMENT

SDL 简 介

1 引言

本建议是对CCITT功能规格和描述语言（SDL）的一般性说明和介绍。该语言由建议Z.101至Z.104详细定义。

1.1 概述

推荐SDL的目的是为了提供一种用来对电信系统的性能制定确切的功能规格和描述的语言。用SDL作为功能规格和描述语言是为了在一定意义上能够对它们进行无二义地分析和解释。

系统功能规格包括对功能行为的规格说明和系统的一般参数。SDL的目标只是在于描述一个系统的功能行为；而表示诸如容量和重量之类的特性的一般参数，则应该用别的方法来描述。

功能规格和描述这一术语具有下述意义：

— 系统的功能规格是要求系统具有行为的描述，而

— 系统的描述是对系统的实际行为的描述。

SDL假定激励和响应都是离散实体，以激励/响应的方式描述系统的行为。此方法以扩展的有限态自动机概念为基础。

用SDL描述的系统行为是对于任何给定的激励序列所作出的响应序列，从系统的外部可以观察到这两个序列。从这种意义上来讲，用SDL描述的任何两个系统，如果具有相同的行为，就被称为功能等效。功能等效概念被用来进行各个系统间的比较，例如，一个所要求的系统的功能规格与一个已提供的系统的描述间的比较。附在建议中的SDL用户指南讨论了这种比较的准则和方法。

SDL也提供结构化的概念，即允许把一个系统分割为几个部分，以便分别对各个部分进行定义、开发和理解。

在最初规定系统功能时，需要独立地处理各个不同的方面，这时这些概念是很有用的。在后来对系统进行描述时，描述的结构应该与系统的结构相对应，这时这些概念也是很有用的。

1.2 目标

在定义SDL时，总的目标是要提供一种语言，这种语言

— 对于营运机构的需要来说，易于学习，便于使用和解释；

— 在订购和投标中提供明确而不含糊的功能规格和描述；

— 可以扩展以适于新的发展；

— 能够适应若干种系统功能规格和设计方法，而不突出其中任何一种。

1.3 应用范围

SDL主要是用来描述电信系统方面的行为。其应用包括：

— SPC交换系统中的呼叫处理过程（例如呼叫处理，电话信号，计次）；

— 一般电信系统中的维护和障碍处理（例如告警，障碍自动排除，例行测试）；

— 系统控制（例如过负荷控制，变更和扩充过程）；

— 数据通信协议。

SDL当然也可用来描述任何能用离散模式描述的行为。离散模式指的是能与其环境用离散信息进行通信的模式。

2 语言概述

下面的SDL概述是对建议Z．101—Z．104的简介。这里所给出的解释除了§2.1节以外都不是正规的定义，但却可作为指导性的说明。

2.1 一些基本定义

在建议Z．100—Z．104中，要用到一些一般的概念和约定。它们的定义如下：

类型、定义和实例

在这几个建议中，实体与其定义应严格区分。所使用的方案和术语定义如下，并参见图1/Z．100。

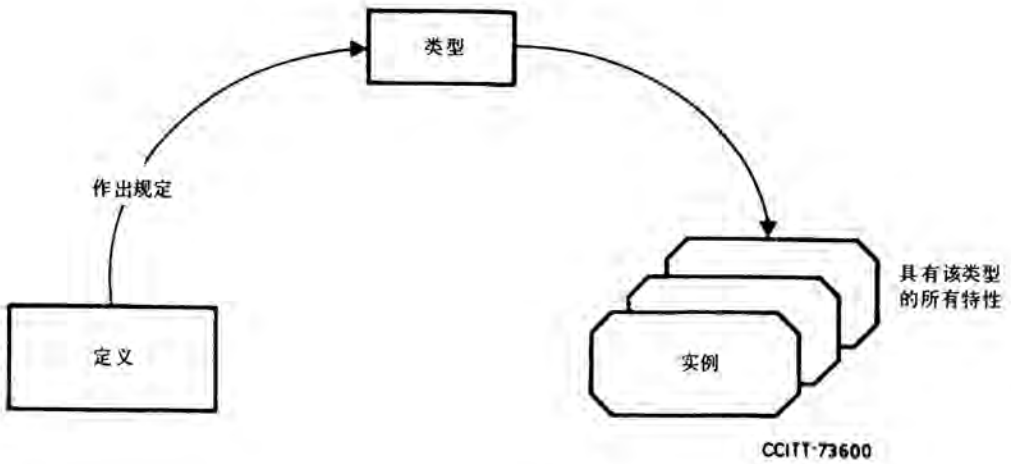


图 1/Z.100
类型的概念

类型由其定义规定，定义规定了与该类型有关的所有特性。一种类型可以有任意多个实例。某一特定类型的任何实例具有该类型的所有特性。

这一原则适用于若干SDL概念，例如，我们有系统定义和系统实例，进程定义和进程实例。

某种类型的实例可以是类型本身。根据分级应用的情况，与类型有关的特性相应地得到保留，例如，具体的系统定义的所有实例都保留了普通类型的系统的特性。

为避免繁琐，特约定：每当根据上下文可明显看出所指的是定义或类型的实例时，将略去定语。

名字

为了识别定义和实例，采用了下述命名约定和术语：

在说明中所用的所有标识符是唯一的。一个实体的识别标志由两部分组成：一是名字部分，一是限定部分：

根据定义了实体的完整的分级上下文以及根据实体的类型来得出限定部分。名字部分应该根据被命名实体的目的或效果来确定出一个有意义的名字。

当具体的语法中出现识别标志时，只允许省略明显的限定符，也就是说，在不会混淆的情况下，可以仅使用名字部分。

可见性规则

当在功能规格或描述中引入一个标识符时，在引入的那点，该标识符是可见的，以及在功能规格或描述分级结构中，紧挨着引入点下面的那些层看来，该标识符也是可见的。一个标识符是可见的最高层由该标识符的限定部分中所给出的最低层来指明。

功能规格错误

在SDL功能规格的特性存在前后矛盾或含糊不清之处，该功能规格就无效。对SDL功能规格的解释如果违反有效的功能规格的特性，则系统解释有错误。导致错误的系统解释意味着不可能从功能规格来预言系统今后的行为。

2.2 基本的SDL

一个SDL系统的动态行为由并发运行的进程实例所产生。一个进程可模拟为一台扩展有限态自动机。它仅根据外部的离散激励而动作，然后，产生离散响应回送给其环境。其他进程可以接受这些响应用作为激励。

在SDL中，一个进程将一直在某个状态等待直到它从其环境中收到一个有效的信号。然后，它将跃迁到另一状态。在跃迁过程中，它可以执行若干动作。这些动作既可以是对进程内部信息的处理，也可以是给其它进程或给系统的外部环境发送信号。

系统中的所有进程或在其环境中的所有进程都可以获得“绝对时间”，并可以进行时间度量和定时。

进程实例可被动态地创建或撤消。任何实例可由另一个实例创建，或者在系统初始化时就存在。一个进程实例仅能被其自身执行的明确的停止动作所撤消。

几个信号实例可能都在等待被某个进程接收。为了处理这种信号队列，这里为每个进程实例安排了一个输入端口。排队原则基本上是先进先出，但进程可以通过某状态的保存信号组自己来管理排队原则。

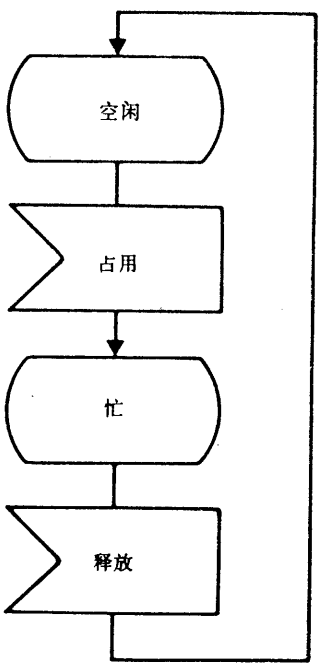


图 2/Z.100

进程的定义

信号实例可以带有数据值，接收信号的进程实例可使用该数据值。数据值可存贮在进程的局部变量中，也可以从变量中取出。

一个SDL系统的静态结构用系统、功能块和信道的概念来描述，如图3/Z.100所示。

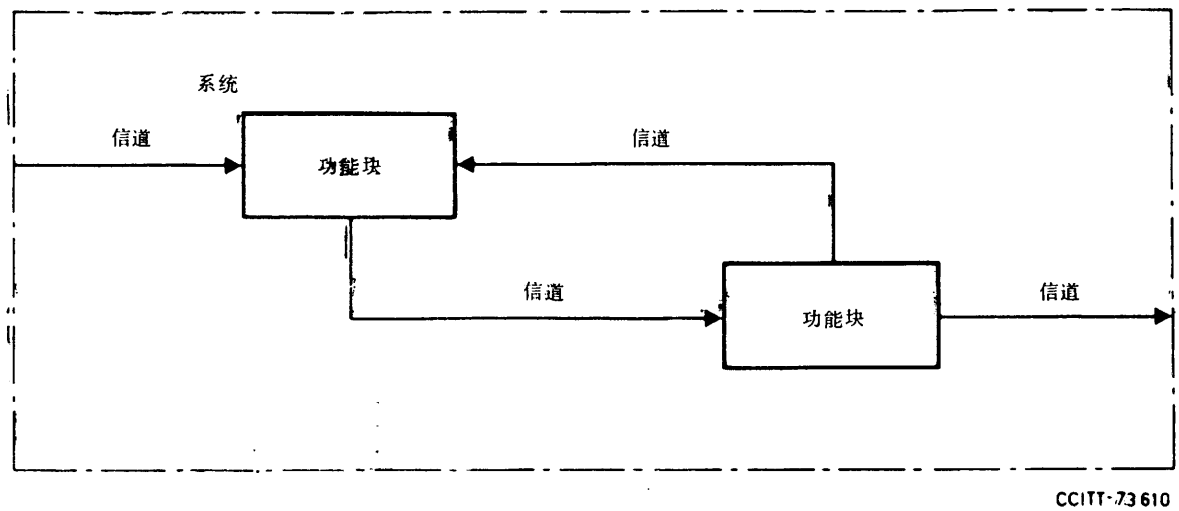


图 3/Z.100

SDL系统的静态结构

系统由多个功能块组成。功能块通过信道互相连接并通过信道连接到系统边界。系统与其环境用系统边界分开。环境是"以SDL方式工作"的，即它会给系统发送信号，并从系统接收信号形式的响应。

信道在功能块之间以及在功能块与系统边界之间充当传输信号的媒介。信道是单方向的。功能块含有进程，并用来构造系统。

2.3 SDL中的数据

SDL进程可以保留和处理数据值。数据值受进程的局部变量的约束。然而，借助于信号，数据值可以在进程之间传送，也可传向环境以及从环境传来。进程的局部变量由该进程定义给出定义。

被处理的数据是已经指定了类型的。在SDL中定义了一些预定义数据类型。此外用户还可定义新的数据类型。这些预定义数据类型是：

- Boolean（布尔） 具有值TRUE和FALSE，以及对这些值的正常的逻辑运算。
- Integer（整数） 具有正常的数学意义。
- Natural（自然数） 具有自然数的正常的数学意义。
- Real（实数） 具有实数的正常的数学意义。
- Character（字符） 具有正常的（如在编程语言中）意义，并且具有CCITT第5号字母表中的值和字形。
- Array（数组） 具有下标的项的集合，这些项具有相同的数据类型。这里对编程语言中所用的数组的通常意义作了扩充。
- Structure（结构） 若干数据类型的组合，这些类型可以不同。
- String（串） 具有任意数据类型的项的序列，其长度是任意的。
- Charstring（字符串） 是一个字符序列，长度是任意的。注意，一个字符串是用字符组成的串，由数据类型字符组成的数据类型串所构成。
- Powerset（幂集） 具有通常的数学意义，因此一个幂集的值是一个有序集合。
- PId（进程标识符） 用以标识进程实例。
- Time（时刻） 绝对时间。

— Duration（持续时间） 绝对时间中两个时刻间的间隔。

— Timer（定时器） 用于定时器的数据类型，该数据类型为 TIMER 变量规定了SET 操作和 RESET 操作。

STRUCT 概念允许进一步构造出包含组合值的数据类型，组合值由若干（可能不同的）数据类型组成。

用户定义的数据类型可以用预定义数据类型来定义，可以加一些限制，诸如整数的范围之类，或者，用户可以定义另外的抽象数据类型。抽象数据类型可用“公理的”方法来定义。

数据类型可以对一个系统通用，或者可以包含于功能块定义中、信道定义中或者进程定义中。在包含这些数据类型的对象的范围内，定义是可见的，因此也是可用的。例如，如果一个功能块中包含一个类型定义，则该类型的数据项可被该功能块中以及该功能块的子功能块中的所有进程使用。

2.4 SDL 中的结构化概念

为了便于描述复杂的和（或）大型的系统，在SDL 中提供了结构化的概念。这些概念是这样定义的，以使它们能够支持：

— 把大的描述划分成模块，以便能独立地处理和理解各个部分。

— 按抽象的若干层次来描述一个系统或描述系统的各个部分。

— 描述一个系统的实际结构。

当使用这些结构化概念时，必须搞清楚它们是代表所要求的结构或实际的结构，还是仅用来使表达更简便。

在基本的SDL 中，一个系统由功能块、信道和进程组成。这些组成部分可以进一步划分结构，即功能块划分为子功能块和子信道，信道划分成功能块和信道，以及进程划分为子进程。进程定义也可以通过使用过程和宏经过若干步骤逐步详细描述之。

当把功能块划分成为子功能块和子信道时，就得到一个系统的分级多层描述。较高层中包含的信息应当包含于较低层的界面中。所得到的结构可以用功能块树图来表示，如图 4 /Z .100所示。

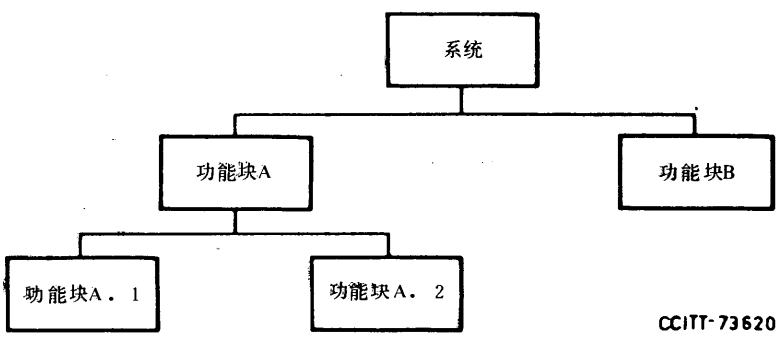


图 4/Z .100
功能块树图

它也可以用功能块互作用图作更详细的描述，如图5/Z .100所示。

后一个图还给出了连接功能块和子功能块的结构的信道和子信道。使用这种功能块划分法时，为了使描述有意义，只要求最细分的子功能块（功能块树图中的“叶功能块”）含有进程。

把信道划分成为信道和功能块也会得到分级的结构。在图6/Z .100中给出了一个简单例子。

把一个进程划分成为一组子进程与功能块的划分有关，因为一个进程的子进程必须总是处在包含该进程的功能块的子功能块之中。正如图7/Z .100中所示。

一个进程的所有子进程一起更详细地表达了进程的行为，它是进程所描述的行为的另一种更为详细的描述。因此当具有如图7/Z .100中那样的结构时，我们必须作出选择，是应该解释用子进程表示的详细的行为呢，还是应该解释用进程表示的不很详细的行为。在SDL 中，我们说这两种描述支持不同的抽象层次。

进程的划分可用进程树图来表示，如图8 /Z .100。

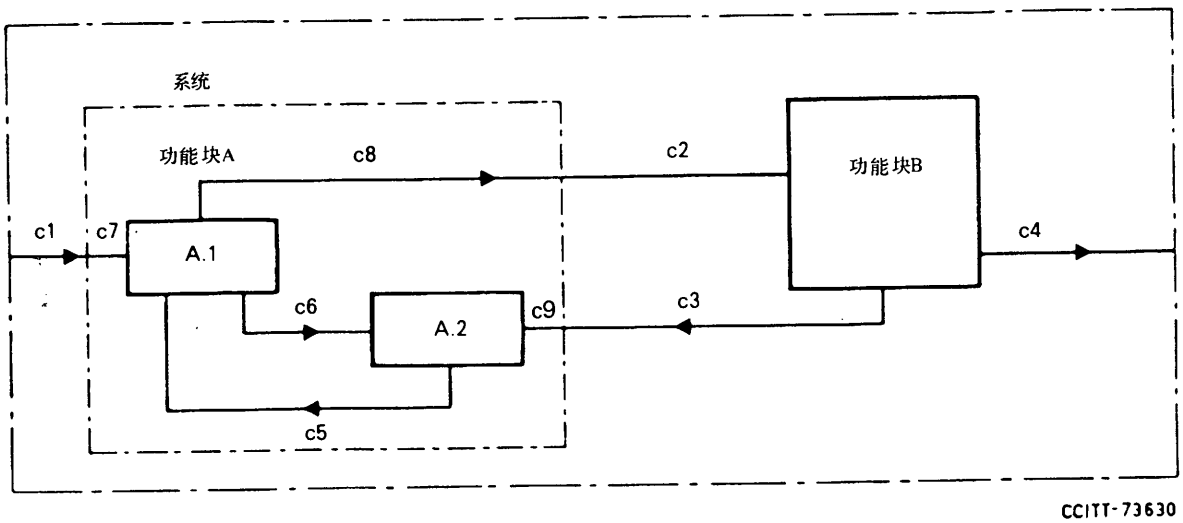


图 5/Z .100

功能块互作用图

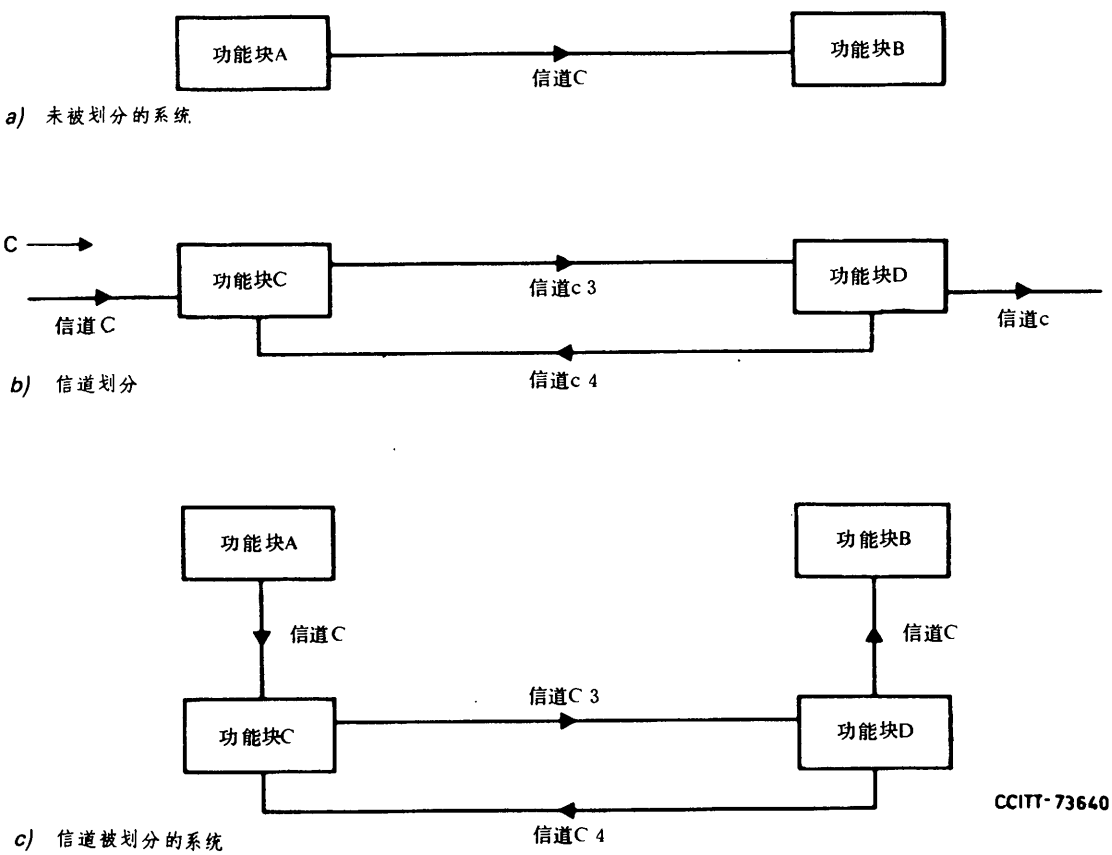


图 6/Z .100

信道划分

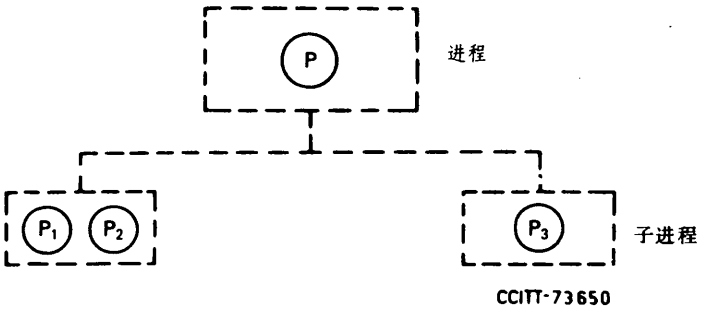


图 7/Z .100

进程划分

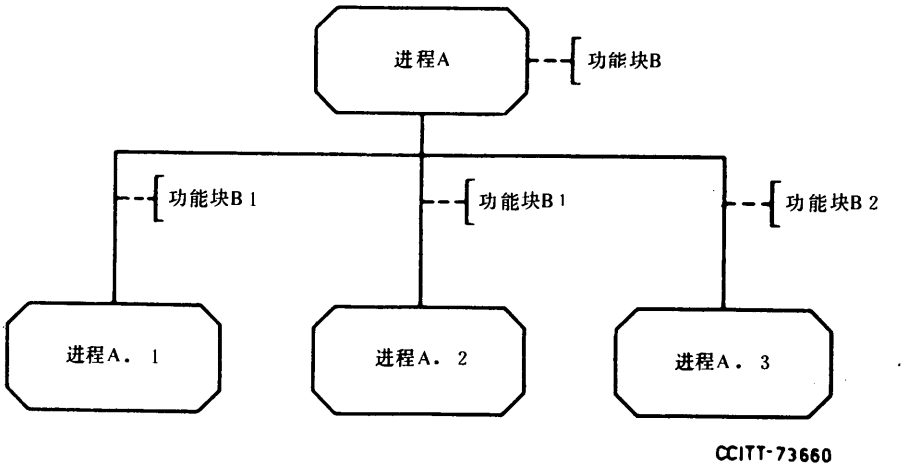


图 8/Z .100

进程树图

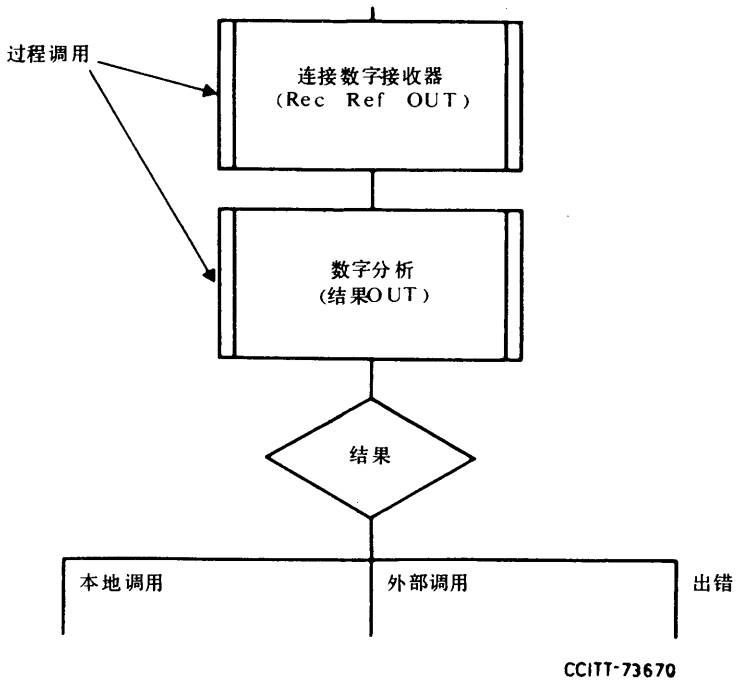


图 9/Z .100

在跃迁中使用过程的例子

进程也可用过程来构造，用过程来详述。SDL中的过程类似于编程语言中的过程。它表示参数化了的和预定义的行为。该行为可被能够访问其定义的任何进程所引用。过程可以在程序库中预定义或由用户自己定义。

与进程相似，过程可定义为动作的集合，并且可以包括状态。使用过程的例子见图9/Z.100。

SDL表示也可用宏来构造。宏是一种语法上的措施，它减轻了SDL表示中作图或书写的负担，而且便于理解SDL表示。宏是由用户定义的多个语法项的命名组合。每当在描述中引用一个宏时，可以理解为用所定义的项来替换它，即宏本身没有自己的语义。宏可用于任何SDL表示，例如用于进程表示，用于结构图等等。图10/Z.100中所示的是使用宏的一个例子。

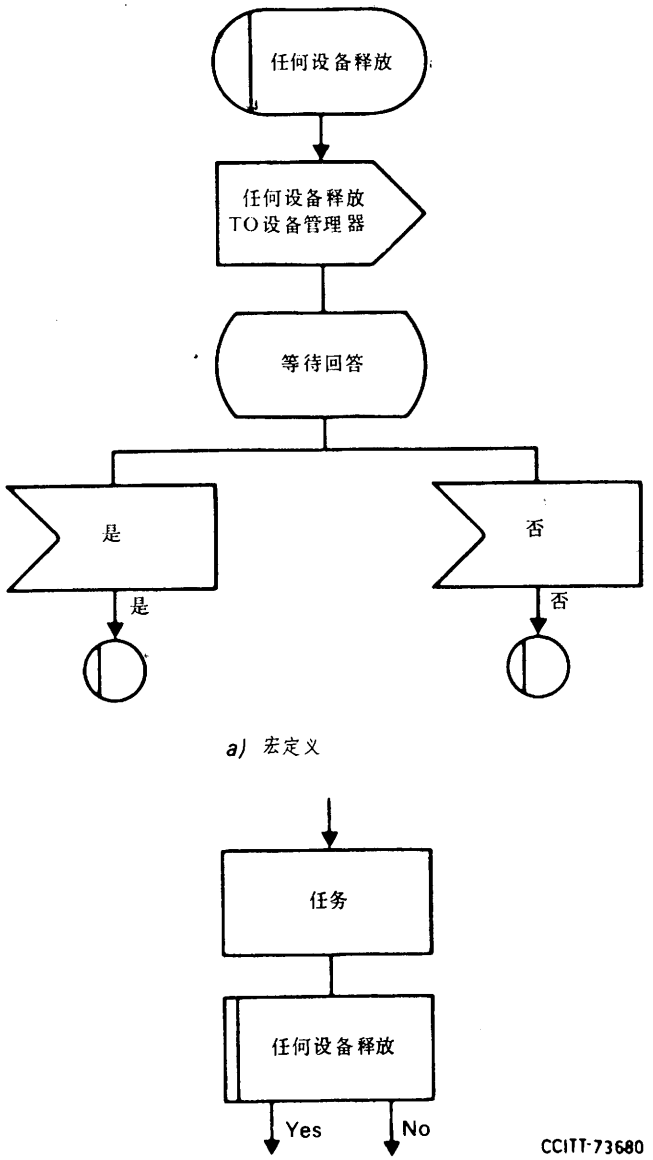


图 10/Z.100
使用宏的例子

所有SDL中的结构化概念当然可以组合起来使用。

2.5 组合操作

SDL中的组合操作是该语言提供的标准简化符号，目的是为了简化SDL进程的设计。组合操作用其他SDL操作来定义，并且应把组合操作看成是与那些操作等效的。

通常，组合操作意味着隐藏了状态以及与其他进程交换的信号，这就是为什么有时应注意副作用的原因。所提供的组合操作有：

数据值的进口或出口

是一个简写符号，用来表示通过隐含信号的交换访问属于其它进程的数据项的值。

允许条件

是一个简写符号，用来表示接收一个信号作为输入的能力或表示保存一个信号的能力，所依据的条件可能含有进口值。这可模拟为若干个状态把信号作为输入来接收，以及模拟为其它的状态把信号保存起来。在求出条件以后选定隐含的状态。此操作也可隐含信号交换。

连续信号

是一个简写符号，用来表示当可能使用进口值的一个条件为真时，离开一个状态并进入一个跃迁。一个连续信号的优先级比“普通”信号的要低，它可用于在外部数据值改变时激励一个跃迁的产生。此操作隐含了若干个状态和信号的互相配合。

使用组合操作，可以减少在进程定义中的状态数和跃迁次数。

2.6 SDL中任选的概念

在用SDL规定或描述若干类似的应用时，同一个进程定义如果略加修改常常可适用于若干个应用。任选(OPTION)这一概念使一个进程定义能够具有任选的部分。

而且，在功能规格中，允许表示从规定者看来是等价于可接受的行为。实际的系统将实现这些可替换的行为中的一个。

3 语言规格初步

SDL的语言规格包含在建议Z．101、Z．102、Z．103和Z．104之中。在下面的初步介绍中，解释了在定义此语言时所用的方法和策略，并且给出了如何阅读这些建议的指南。

3.1 语言规格中所用的策略

SDL给出了两种不同的语法形式可供在表达SDL描述时选择使用：图形表示(SDL/GR)，以及文字的短语表示(SDL/PR)。因为这两种表示都是对同一SDL语义的具体表示，故从语义学的观点来讲，它们是等效的。为了保证它们彼此等效并因而可互相转换，就要严格区分SDL的语义定义与不同的具体语法的定

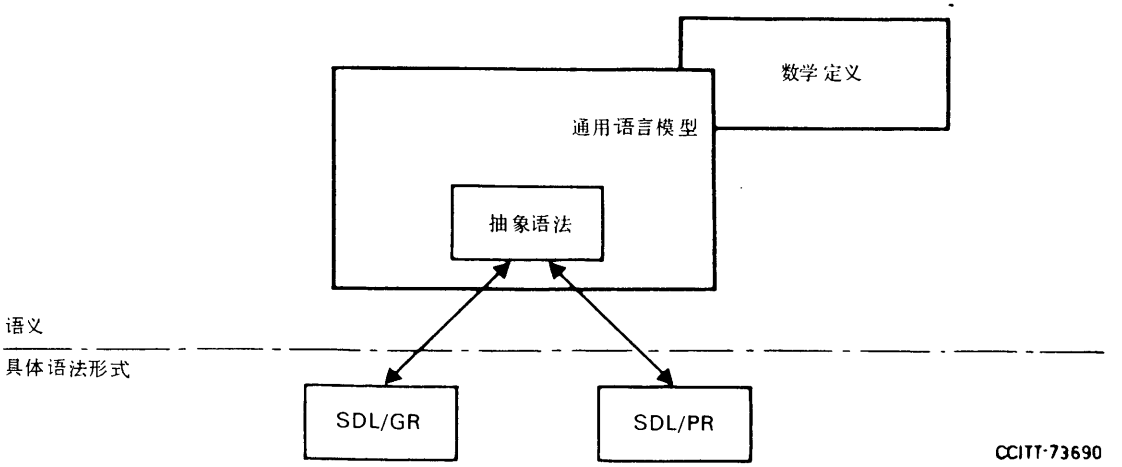


图 11/Z．100
SDL语言规格的结构

义。SDL的语义定义与具体的语法形式之间的关系可由图11/Z．100来描述。

SDL的语义使用抽象语法定义，不带有具体的表示，而带有好的格式规则和解释规则。这个定义称为“通用语言模型”。因此每个具体的语法形式都有自己的语法形式的定义以及与抽象语法的关系的定义（即如何转换成抽象形式）。使用这种方法，只有一种SDL的语义定义。每个具体的表示形式通过与抽象语法的关系继承语义。此方法也保证了具体的语法形式是等效的。由于可以进行两个方向的转换，因此任一种表示均可通过抽象语法转换成另一种形式。

通用语言模型的解释规则以操作的方式来定义，即此定义描述了SDL概念的实例如何把它们的定义解释为“抽象SDL机器”。此外，也提供了用符号语义写的SDL的数学定义（但不是本建议的一部分）。数学定义与抽象语法以及通用语言模型的结构有紧密的关系。

按照这一策略，对每个SDL概念，建议中首先给出抽象语法的定义和规则，然后给出如何解释此概念的规则。最后给出表达此概念的具体语法形式。

3.2 术语

对每个SDL概念，在整个建议中始终使用同一个术语。为了将一个术语指的是一个SDL概念时的情况与指的是一个较一般的意义时的情况相区别，所有的SDL概念的术语将用仿宋体印刷。

包含所有SDL的术语的词汇表作为附件A附在建议后面。词汇表包括对每一术语的简短解释并指出定义该术语的地方。

3.3 SDL/GR的定义

SDL/GR按以下的次序安排来定义：

— 首先定义符号的形状及含义。

— 如果合适的话，接着给出连接规则，即什么样的符号组合是允许的。

— 最后给出与通用语言模型中的抽象语法的关系。

3.4 SDL/PR的定义

SDL/PR用语法图及用自然语言表达的附加规则来定义。定义要遵循下列的次序安排：

— 首先用图和文字定义语法。

— 然后给出与普通语言模型中的抽象语法的关系。

3.4.1 语法图

语法图由终结符和非终结符组成，它们之间用流线连接起来。

终结符包含一个字符或一串字符，其产生规则是：当在通路中通过时，那些字符应出现在SDL/PR的正文中。

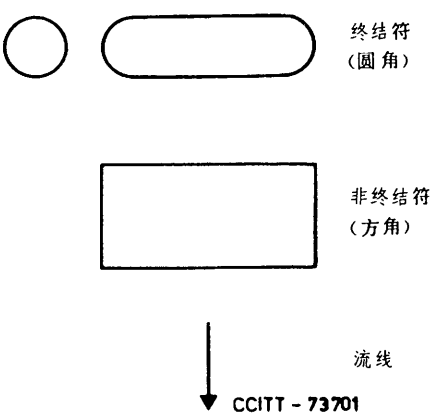


图 12/Z．100
语法图中的符号

一个非终结符是对另一个语法图的引用，该语法图具有在符号中出现的名字。其产生规则是：当非终结符在图中出现时，通路就进入所引用的图，并且此通路直到离开所引用的图时才离开该非终结符。

所有终结符、非终结符以及语法图都只有一条通向它们的流线，也只有一条流线离开它们。

所用的图形符号如图12/Z.100所示。

在"最高"层上，SDL/PR语法的定义由一个语法图SYSTEM（系统）组成。借助于非终结符，此图又进一步引用一组其它的图。从通向此图的流线开始到从此图离开的任何通路将在其通路上产生一SDL/PR文句。如果遵循了SDL语法规则，则该文句在语法上是正确的。

3.5 SDL/GR和SDL/PR的公共语法元素

对于某些语法元素，在SDL/GR和SDL/PR中两者都使用同一个具体的语法。

3.6 SDL建议的结构

本建议后有四个建议(Z.101至Z.104)。目的是指导用户选择一个适合于他们的应用和研究方法的SDL子集。

应用SDL的方式可以有许多种，用以支持不同的目的和方法。可供选用的最小SDL子集在建议Z.101中给出。

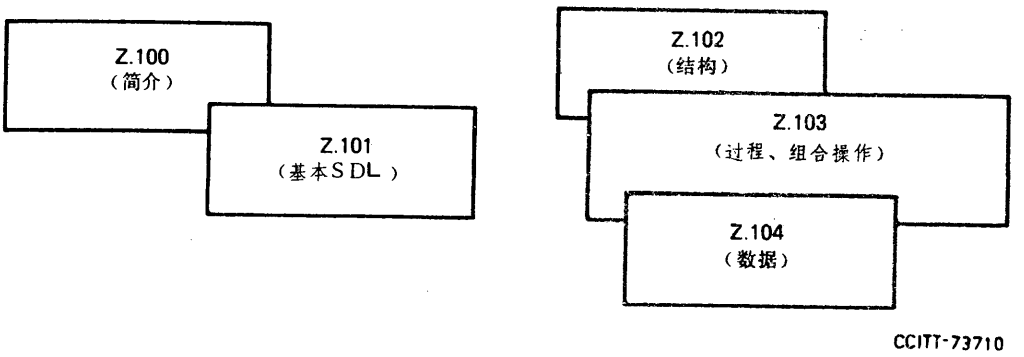


图 13/Z.100
SDL建议的结构

建议Z.102至Z.104包含对最小SDL子集的扩展，在适当时应该加以采用。多个扩展的内容也可以以任何组合一起应用。

在附件中包含了一些例子，使用了在建议中定义的概念。SDL用户指南中还包含了更容易理解的、使用SDL的例子。

后面的建议的内容可以简单地概括为：

Z.101 定义SDL的基本概念。该建议给出可应用的SDL的最小子集。要描述系统的行为，这个SDL子集已经足够了。

Z.102 定义另加的结构化概念，用来描述大型的和（或）复杂的系统。它们既能用来描述实际系统的结构，也能在若干抽象层次上描述此系统。

Z.103 定义过程概念、组合操作、宏概念、任选概念以及面向状态的图形扩展(SDL/PE)。这些概念被相互独立地定义，可以适用于任何组合。

Z.104 在Z.104中定义了一些数据概念，这些概念在Z.101至Z.103中被认为是已预先定义的。应当注意，在SDL中可以完全非形式地使用数据。

Z.104也包含对SDL抽象数据类型概念的定义。

除了这些建议以外，还有一组辅助的文件用来描述和解释此语言，它们没有以建议的形式出现。其中一些文件作为建议的附件。这些辅助文件是：

SDL的形式定义

此文件包含SDL语义的数学定义。定义用符号语义学（VDM，META IV）表达。不久就可以把它作为一份独立的文件提供。[1]

SDL术语词汇表

此文件包含全部SDL的术语。每个术语有简短的解释并给出在建议中定义它的地方。此文件作为建议的附件A，放在红皮书卷VI.11中。

SDL抽象语法摘要

此文件包含该语言的全部抽象语法的摘要。此抽象语法用简短的类巴科斯—诺尔（BNF-like）形式描述。此文件作为建议的附件B，放在红皮书卷VI.11中。

SDL具体语法摘要

此文件包含SDL的全部具体语法的摘要，即图形表示（SDL/GR），图形形式的面向状态的图形扩展（SDL/PE），以及文字的短语表示（SDL/PR）。此文件作为建议的附件C，放在红皮书卷VI.11中。

SDL用户指南

此文件举例说明和解释SDL的使用（不包含此语言的定义）。它包含一些例子和讨论，涉及SDL的不同用法。在用户指南中还讨论了在使用中需特别小心的一些概念，这些概念是建议中定义的那些。此文件作为建议的附件D，放在红皮书卷VI.11中。

SDL教程

此文件用来训练人们使用SDL。它作为独立的一个文件提供使用。[2]

最后，在本卷的封底内页中装有一个样板以供在画SDL的图形时使用。它们包含所有推荐的图形符号，符合建议的格式。

建 议 Z.101

基 本 的 SDL

1 引言

本建议定义基本的CCITT功能规格和描述语言（SDL）。SDL的基础是能进行通信的、称作为进程的有限态自动机的概念。一个SDL系统就是一组功能块。功能块与功能块之间以及功能块与环境之间通过信道连接。每个功能块内有一个或多个进程。这些进程通过信号互相通信，并且并发地执行。

在定义SDL时，首先定义一个通用的SDL模型是很有用的。此通用模型构成了具体语法的抽象基础，并把这些语法结合在一起。各个具体语法只不过是表达SDL概念的不同方法而已。目前有两种具体语法：SDL/GR和SDL/PR。既然这两种语法表达相同的SDL概念，因此有可能把一个采用某种SDL具体

1) SDL的形式定义可向此处索取：International Telecommunication Union, General Secretariat-Sales Section, Place des Nations, CH-1211 Genève 20 (Switzerland).

2) SDL教程可通过此处得到：International Sharing System for Training, ITU Secretary General-Technical Co-operation Department, Training Division, Place des Nations, CH-1211 Genève 20 (Switzerland).

语法定义的系统映射到另一种具体语法。

本建议这样来定义SDL语言：首先定义通用语言模型，然后定义SDL/GR和SDL/PR具体语法。

2 通用语言模型

2.1 通用模型介绍

在SDL中，一个系统由一些功能块组成。功能块与功能块之间以及功能块与系统的环境之间通过信道相互连接（参见图1/Z.101）。信道是单方向的。

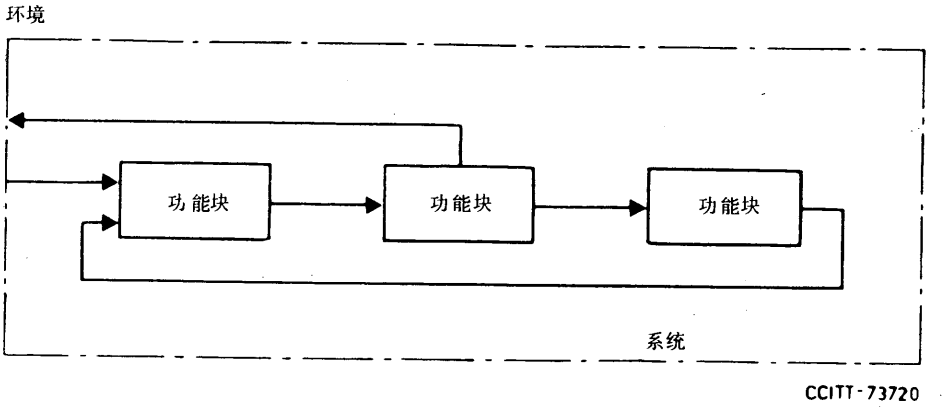


图 1/Z.101
SDL模型

每个功能块的行为用一个或多个进程来表示。进程由进程定义来进行定义。

进程借助于信号与其它进程或环境互相作用。一个信号就是进程之间传送数据的信息的一次流动。当一个进程输出一个信号时，该信号就被传送到它指向的进程。在同一个功能块中进程之间传递信号的传输原理与在不同功能块中的进程之间传递信号的传输原理是一样的。信道表示功能块之间互相交换信号的传输路径。

当一个信号到达它所指向的进程时，它被保留在进程外面直到该进程准备好可以接收该输入信号。当目的进程接收了这个信号后，该信号就被消耗掉了。

SDL可以模拟开放系统，其含义为系统可以与其环境互相作用。只有通过出入环境的信道传递信号，这种互相作用才会发生。环境应以类SDL方式工作，即环境可含有一个进程，该进程将信号输出到通向系统的信道以及接收从系统来的信道上的信号。

在进程的生命期间，它或者处于一个状态（等待接收一组信号中的某一个），或者处于一个跃迁之中（完成一系列的动作）。当进程处于某个状态时，它只能接收一组特定的信号。如果这些输入信号中的一个信号被保留在进程之外，它就会被进程接收。输入信号被接收将使信号中携带的数据可以被进程所使用，于是引起了一个跃迁。在跃迁过程中可以处理进程的数据，并可输出信号。此跃迁将以进程进入一个新的状态而告终，或者以一个停止而告终。

一个停止使进程不复存在。

绝对时间的概念是系统和它的环境所共有的。遍及整个系统和环境，绝对时间都是一样的。

2.2 抽象语法

系统定义

系统定义包含一个系统名、一个或多个功能块定义、一组信道定义以及一组信号定义。

系统定义包含有对每个信号名的信号定义。这些信号名包含在信道定义中的信号表中。

功能块定义

功能块定义包含一个功能块名、一个或多个进程定义，也可能包含信号定义。

每个功能块定义包含有对每个信号类型的信号定义，这些信号是在功能块内的进程之间传送的信号。

信道定义

信道定义包含一个信道名、一个源功能块定义标识符、一个目的功能块定义标识符以及一个信号表。信号表包含每个信号类型的标识符，这些信号是可通过信道传递的信号。

信道定义中的源功能块定义标识符和目的功能块定义标识符必须是不同的，每个标识符必须是在系统定义中的功能块定义的标识符，或者必须是环境。

与信道定义有关的信号表至少包含一个信号标识符。

信号定义

信号定义包含一个信号名，也可能包含一系列数据类型名。

进程定义

进程定义包含一个进程名、一对整数以及一个进程流图，也可能包含一形式参数表、多个变量定义以及视见定义。

进程流图

进程流图是一用有向弧连接其节点的图。进入节点的弧称为进入弧，离开节点的弧称为走出弧。有一条进入弧的节点跟随在以同一条弧作为走出弧的节点的后面。

有下列各种节点：

状态节点
输入节点
任务节点
输出节点
判定节点
起动节点
停止节点
创建请求节点

下面规定进程流图的连接规则：

1) 每个进程流图包含一个且只有一个起动节点。起动节点的后面是一个跃迁串。起动节点不跟在任何节点后面。
2) 跃迁串可以是下列中的一个：
   a) 空操作后面跟一个状态节点，或者跟一个停止节点；
   b) 一个动作串后面跟一个跃迁串；
   c) 一个判定节点。
3) 动作串可以是下列中的一个：
   a) 一个任务节点；
   b) 一个输出节点；
   c) 一个创建请求节点。
4) 判定节点后面跟两条或多条判定弧。
5) 判定弧是一条有名字的弧，后面跟一个跃迁串。
6) 状态节点后面跟一个或多个输入节点。
7) 一个输入节点跟在一个且只能跟在一个状态节点的后面。
8) 一个输入节点后面跟一个跃迁串。
9) 停止节点后面不跟有任何节点。
10) 每个进程流图最多只能有一个停止节点。
11) 从起动节点可到达每一个节点。

状态节点

一个状态节点包含一个状态名，也可包含一个保存信号组。一个进程流图中的所有状态节点各有不同的名字。

输入节点

输入节点包含一个信号标识符，也可以包含一组有序的变量标识符。

保存信号组

一个保存信号组包含多个信号标识符。

任务节点

任务节点或者包含一系列语句，或者包含非形式文句。

语句

语句可以是一个设置语句、复位语句或赋值语句。

设置语句

设置语句包含一个时间表达式以及一个定时器标识符。

复位语句

复位语句包含一个定时器标识符。

赋值语句

赋值语句包含一个变量标识符、一个赋值运算符以及一个表达式。

输出节点

输出节点包含一个信号标识符和一个目的地表达式，也可以包含一个实在参数表。

判定节点

判定节点包含一个问题，且至少有两条走出弧。每条走出弧对应于对该问题的一个或一组回答。对该问题的每个可能的回答应对应于一条弧且只能是一条弧。一个问题或者是一个表达式，或者是非形式文句。一个回答或者是值标识符，或者是非形式文句。

创建请求节点

创建请求节点包含一个进程定义标识符，也可以包含一个实在参数表。

数据类型

预定义的数据类型有：自然数、整数、实数、字符、字符串、布尔、时刻、持续时间、定时器、进程实例标识符。它们在建议Z．104中定义。

变量定义

变量定义包含一个变量名和一个数据类型标识符，还可能包含一个透露属性。

形式参数表

形式参数表是形式参数的一个有序集合。

形式参数

形式参数包含一个形式参数名和一个类型标识符。

实在参数表

实在参数表是实在参数的一个有序集合。

实在参数

实在参数是一个表达式。

视见定义

视见定义包含一个变量标识符、一个数据类型标识符以及一个进程定义标识符。

由进程定义标识符所指的进程定义中，该变量必须有一个透露属性。被提及的提供透露的进程定义必须与视见进程定义属于同一个功能块。

表达式

表达式或是一个值标识符，或是一个变量标识符，或是一个运算。

运算

运算或者包含一个视见表达式，或者包含一个运算符和一个或多个表达式。

视见表达式

视见表达式由视见运算符加上变量名和进程实例标识符组成。

2.3 解释规则

2.3.1 系统

系统是具体的实体，例如一个电话交换机，并且系统是系统类型的具体实例，系统类型由系统定义规定。系统边界将系统与其环境分开。系统含有一组功能块。系统与其环境之间的通信以及在系统内部功能块与功能块之间的通信只能通过信号才能进行。在系统内，这些信号在信道上传送。信道将功能块与其它功能块连接起来，或把功能块连到系统边界。

系统有一个不带参数的函数，叫做NOW，其类型为时刻，用以产生当前的时刻。当前的时刻可立即提供给整个系统和其环境使用。系统中任何一个进程的表达式中都可以使用NOW。

2.3.2 信道

在系统内，对应系统定义中的每个信道定义都有一条信道。信道是信号的传输路径。该路径是单方向的。信道的终端或是一个功能块，或是系统边界。至少要使信道的一个终端是在功能块上。如果两个终端都在功能块上，则这两个功能块必须不同。信道定义中要列出能在该信道上传递的所有信号。

当一个信号被输出到信道上时，该信号就被传送到目的功能块。输出到信道上的信号的次序与从信道上收到的信号的次序是一样的。如果两个或两个以上的信号同时到达信道，则可随意安排它们的次序。

2.3.3 功能块

在系统内，对应于系统定义中的每个功能块定义都有一个功能块。功能块的规模（大小）易于安排，能够对一个或多个进程进行解释。在同一个功能块内部进程之间的通信机制有两种：信号和共享值。

当信号从信道到达功能块时，此功能块就将信号送到某进程的输入端口，这个进程是由信号中的进程标识符指定的。

当一个信号由功能块中的一个进程输出时，就被送到由信号中的进程标识符所指定的进程。如果所指定的进程是在同一个功能块中，该功能块就把信号送到其输入端口。如果所指定的进程是在另一个功能块中，则该功能块（即信号所在功能块）就将信号送到能够传送此信号的信道上。

共享值允许一个进程视见另一个进程中的透露变量。只有透露进程才被允许改变变量的值。视见进程通过使用视见运算符接收透露变量的当前值。

2.3.4 信号

信号是在进程之间传送信息的数据的一次流动，它是由信号定义所规定的信号类型的具体实例。信号可由系统环境或进程发送，它总是被送往进程或环境。

每个信号包含信号定义中的信号标识符、源进程实例标识符以及目的进程实例标识符。此外，其它值可由信号中的变量传送。在一个信号中，对应每个列入信号定义中的数据类型的每个名字都有一个变量。

2.3.5 进程

进程是一个会进行通信的有限态自动机，它是由进程定义所规定的进程类型的具体实例。在一个功能块内，对每个进程定义可以有零个、一个或多个进程。进程可从创建系统时就存在，或者由创建请求动作来创建，进程可通过执行停止动作而撤消。系统中的进程是相互独立的，但又是并发地执行的。

系统中的所有进程具有四种预定义变量，其进程标识符类型称为：SELF，PARENT，OFFSPRING和SENDER。这些变量是下列进程的进程实例标识符：

— 本进程（SELF）；

— 进行创建工作的进程（PARENT）；

— 最新创建的进程（OFFSPRING）；

— 来自所接收到的最后一个输入信号的进程（SENDER）。

这些变量能够用于表达式，但不能显式地对其赋值。对所有在系统初始化时出现的进程，给于PARENT同一个明显的值，此值与任何进程的SELF值是不同的。

传向此进程的信号叫做输入信号，从此进程传出来的信号叫做输出信号。输入信号是用来起动此进程并把信息传给进程的实体。输出信号被用来起动另一个进程并把信息传给它。

在进程流图的输入节点处附有一组信号标识符，用来表示对这个进程定义而言是有效的一组输入信号标识符。对每个状态来说，所有的输入信号标识符或者出现在信号保存组中，或者出现在输入节点中。

进程定义中所包含的两个整数定义了建立系统时所创建的进程的实例个数以及具有相同进程类型的同时存在的实例的最大个数。当建立一个系统时，初始的进程以随机的次序创建，同时没有实在参数传给进程。

当创建一个进程时，给它提供一个空的输入端口，而且所建立的变量具有不定值，然后通过解释进程流图中的起动节点来起动此进程。

当一个有效输入信号到达一个进程时，它就被放入该进程的输入端口。每个进程只有一个输入端口。输入端口可以保留任意个输入信号，因此对此进程会有好几个输入信号排队等待。被保留的信号的集合在队列中按它们的到达时间排序。如果两个或多个信号同时到达，它们的次序就是任意的。在输入端口处可能附有一个可能为空的定时器集合。每个定时器包含时间类型的一个值。

一个进程或是在一状态中等待，或是正在活跃地执行一次跃迁。对每个状态都有一个信号保存组。当在某一状态中等待时，其标识符不在信号保存组中的第一个输入信号就从队列中被取出，并为进程所接受。如果有定时器，则输入端口不断地将NOW与一个定时器的值相比较，该值应是比零大的各定时器值中的最小值。当NOW的值大于或等于此定时器的值时，一个带有与此定时器相同名字的信号就被放入队列，然后给定时器置零。

当一个进程正在等待时，它总是处在由进程流图中相应的状态节点所代表的那个状态中。接收了输入信号后，该进程就被激活了，它把输入节点解释为具有与输入信号相同的名字，并执行一次跃迁，通向一个新的状态。

2.3.6 进程流图

一个起动节点被解释为一个起动动作。起动动作使得跟在起动节点后的那个节点被解释。

一个输入节点被解释为一个输入动作，此动作接收并消耗了给定的信号，然后解释跟在输入节点后的那个节点。信号的消耗就使得进程可以利用由信号传递来的信息。信号中的变量值就赋给输入节点中的相应的变量。对信号中的某个变量，如果在输入节点中没有对应的变量，此变量的值就被抛弃。正在接收的进程中的SENDER被赋以由信号传来的源进程实例标识符的值。

任务节点被解释为一个任务动作。任务动作是对一系列语句和非形式文句的解释。当该动作完成后，就对跟在任务节点后的那个节点进行解释。

设置语句将导致对该语句中给出的定时器执行一个复位语句，并将把给定的时间值赋给定时器。

复位语句把定时器的时间值置为零。如果有与定时器同名的任何信号出现在输入队列中，那么就要从输入队列中将它们移出并抛弃。所有在队列中的信号如果其标识符与定时器信号标识符相同，就将从队列中移出并抛弃。

赋值语句被非形式地解释为：把赋值语句中的表达式的值用作为赋值语句中变量的值。

输出节点被解释为一个输出动作，该动作产生一个信号并将它传送到功能块中。然后解释输出节点后面的那个节点。输出信号是由信号定义所规定的信号类型的一个具体实例，该信号定义由输出节点中的信号标识符指明。输出节点中的实在参数的值赋给信号中的变量。对于信号中的一个变量，如果输出节点中没有对应的实在参数，则此变量就具有不定值。把变量SELF的值赋给由该信号携带的源进程实例标识符。信号的目的进程实例标识符被赋以输出节点中包含的目的表达式的值。

判定节点被解释为回答某个问题的判定动作。它选择出与此问题的回答相适配的弧，然后解释这条弧后面的那个节点。

状态节点被解释为：通过给进程以一个新的状态作为一次跃迁的结束。新的状态由节点中包含的名字指定。该进程现在正在一个状态中等待这一情况通知给输入端口，同时向输入端口提供附于被解释的状态节点上的信号保存组。此后，该进程就等待，直到给出一个新的输入信号。

停止节点解释为进程的立即结束。这就意味着抛弃了那些保留在输入端口中的输入信号，同时由进程建立的变量、输入端口以及该过程本身也将不复存在。

创建请求节点被解释为一个创建请求动作。创建请求动作导致在本功能块内创建一个进程。该进程的定义处于本功能块中，并由创建请求节点中的进程标识符来标识。作为创建请求动作的一部分，把进行创建活动的进程的SELF变量的值赋给被创建的进程的PARENT变量。把同一个唯一的进程实例标识符的值既赋给被创建进程的SELF变量也赋给进行创建工作的进程的OFFSPRING变量。在新创建的进程中的形式参数被赋予创建请求节点中包含的实在参数值。

2.3.7 数据类型

预定义的数据类型按照正常的含义来使用。

2.3.8 变量

变量可被赋值。赋给变量的值的内容以后又可以从变量中取出。变量也有数据类型，此类型限制了能够赋给此变量的值的类型。

在一个进程中，对进程定义中的每个变量定义都有一个变量与之对应。变量的值只能由定义该变量的进程来修改。变量的值只能为定义该变量的进程所知，除非此变量具有透露属性。透露属性允许同一功能块中的其他进程去视见一个变量。变量与声明此变量的进程具有相同的寿命（即变量随着声明该变量的进程的创建而产生，随着声明该变量的进程的消失而消失）。

在一个信号中，对信号定义中的每个数据类型名的每次出现都有一个隐名变量。此变量的值只有在产生该信号的那个输出节点中赋值，并且此变量的值只有在接收并消耗此信号的输入节点中才能知道。这些变量的寿命以信号的寿命为界限。

2.3.9 表达式

表达式可非形式地解释为产生一个值。

2.3.10 视见定义

视见定义规定了一个变量名，它只可用于一个视见表达式来得到属于另一个进程的变量的值。

2.3.11 视见表达式

视见表达式的值就是视见表达式中由变量名和进程实例标识符所标识的变量的值。

3 SDL/GR

一个系统定义在SDL/GR语法中可表达为：

— 功能块互作用图，它包含系统名和信道定义，并且指明功能块定义。功能块互作用图还指明塑造每个功能块的行为的进程定义。功能块互作用图可表示：1)在同一功能块中进程间传递的信号表；2)功能块间通过信道传递的信号表；以及3)由其他进程实例创建新的进程实例；

— 进程图，它定义每个进程的行为，它是进程定义的图形描述。进程图可以包含变量定义、形式参数以及视见定义；

— 信号表，它列出通过信道传递的信号名单或列出在功能块的内部从一个进程传递到另一个进程的信号名单，可以使用信号表符号将这些信号表包括在功能块互作用图中。或者以任何觉得合适的形式把信号表表达为独立的表；

— 信号定义，对信号表中命名的每个信号给出数据类型和包含于信号中的数据值的次序。这些用SDL/PR语法来规定。

系统定义中可以使用名字和标识符。名字用SDL/PR语法来规定。标识符由一个名字连同限定符组成。限定符是与名字有关的类型实体。限定符表示了它所标识的实体的分级结构层次。不能让两个或两个以上的实体具有相同的标识符。在SDL/GR中，名字的限定符可以根据上下文得出，但无论什么时候使用它们，都必须用SDL/PR语法来确定它们。

3.1 功能块互作用图

一个系统的功能块互作用图包含一个系统名、一组功能块符号、环境符号以及一组信道符号和信号表符号。

3.1.1 符号

推荐的符号画在下图2/Z.101中。

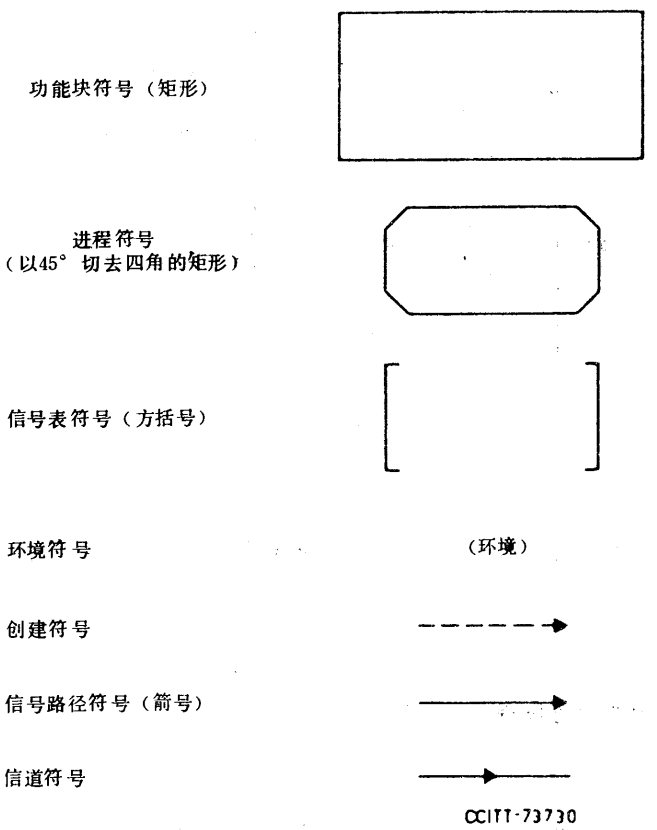


图 2/Z.101

功能块互作用图符号

3.1.2 SDL/GR功能块互作用图与抽象语法以及符号的使用之间的关系

— 环境符号表示系统环境，可以出现多次。

— 功能块符号包含有一个名字、进程符号的一个非空集合、信号路径符号，并可含有创建符号以及信号表符号。

— 进程符号包含有一个进程名，并可含有形式参数表符号。进程名与进程定义中含有的名字相同，而进程定义描述了该进程的行为。形式参数表符号包含了形式参数的名单，形式参数或者在创建系统时得到初始化，或者在动态地产生进程实例时得到初始化。

— 两个整数值与一个进程符号对应，第一个值表示创建系统时就存在的该进程的实例的数目，第二个值表示这个进程类型能够同时存在的实例的最大数目。这两个值放在进程符号的右上角。
第一个值的缺省值为1。第二个值的缺省值为无穷大。这两个整数用SDL/PR语法确定。当没有指定其中的一个整数时，就选择其缺省值作为它的值。

— 每个信道符号附有一个信道名。信道符号有一个源端连到某功能块符号，还有一个目的端连到另一功能块符号。或者源端或者目的端（但不能两者都）可以不连到一个功能块符号，而是连到一个环境符号。信号表符号可以放在信道符号旁边以标识通过该信道传输的信号。

— 功能块中的信号路径符号与信号表符号对应。信号路径符号或者是从一个进程通向另一个进程，或者是从某进程通向信道的源端（在功能块的边界上），或者是从信道的目的端（在功能块的边界上）通向某个进程。

— 功能块互作用图中的信号表是名字的罗列，全部围在信号表符号中（即在方括号中）。信号表本身可以有一个名字，写在该符号的上面。表中的各项（用逗号分隔，可以用列或行的格式书写）是各个信号定义的名字以及其它信号表的名字。在表中，为了区分信号表名与各个信号定义名，又用一对方括号把信号表名括起来。如果决定使用信号表符号把信号表包括在功能块互作用图中，则必须将全部信号表包括在图中。图3/Z.101给出了信号表的例子。

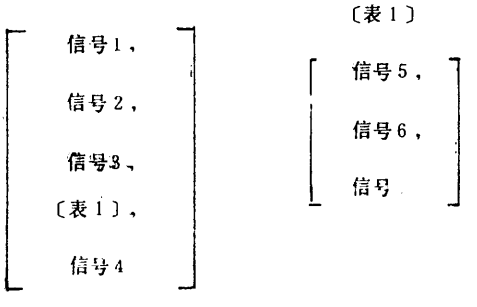


图 3/Z.101
功能块互作用图中的信号表举例

— 创建符号从一个进程符号引向另一个进程符号。前面的进程符号代表“进行创建的”进程，后面的进程符号代表“被创建的”进程。

3.1.3 绘图规则

— 信道符号连至与它们有关的功能块符号的边界。信道符号应该以90°角连接到功能块符号的边界。如果需要，信道符号可以有90°的折角。信道符号含有一个箭头以表明信号沿着该信道流动的方向。

— 信号路径符号连至与它们有关的功能块符号或进程符号的边界。信号路径符号以90°角连接到这些边界较为可取。如果需要，信号路径符号可含有90°角的折角。若干个信号路径符号可以在信道的源端汇合（在功能块的边界）。若干个信号路径符号可在功能块的边界从信道的目的端发散。信号路径符号有一个箭头画在一端，以表明信号流动的方向。

— 创建符号用虚线连至进程符号边界，虚线以90°角与进程符号边界相连。在这些连线上用箭头从“进行创建的”进程符号指向“被创建的”进程符号。
— 符号的最佳取向如图2/Z.101所示。
— 符号的尺寸可由用户选择。
— 符号边界不能重叠或交叉。信道符号和信号路径符号是例外，它们可以互相交叉。互相交叉的信道符号之间或信号路径符号之间没有逻辑上的关系。

3.2 信号表

与连到一个给定进程定义的信号路径符号有关的多个信号表中，其全部信号名的并集就组成该进程的有效输入信号名的集合。

与连到某信道源端的信号路径符号有关的信号表中，其全部信号名的并集就等于那个信道的信号路径表中的名字表，并等于信号路径表中全部信号名的并集，该信号路径表与连到那个信道的目的端的信号符号（在信道的目的功能块中）有关。

3.3 进程图

3.3.1 符号

进程的行为通过一个进程图以图形形式表示。进程图的名字与它所表示的进程定义中的进程名相同。图4/Z.101给出了SDL/GR进程图中所用的符号。

3.3.2 SDL/GR进程图与SDL抽象语法以及符号的使用之间的关系

列在图4/Z.101中的每个进程图符号代表抽象语法中进程流图的等效命名节点。连接符号的流线代表连接

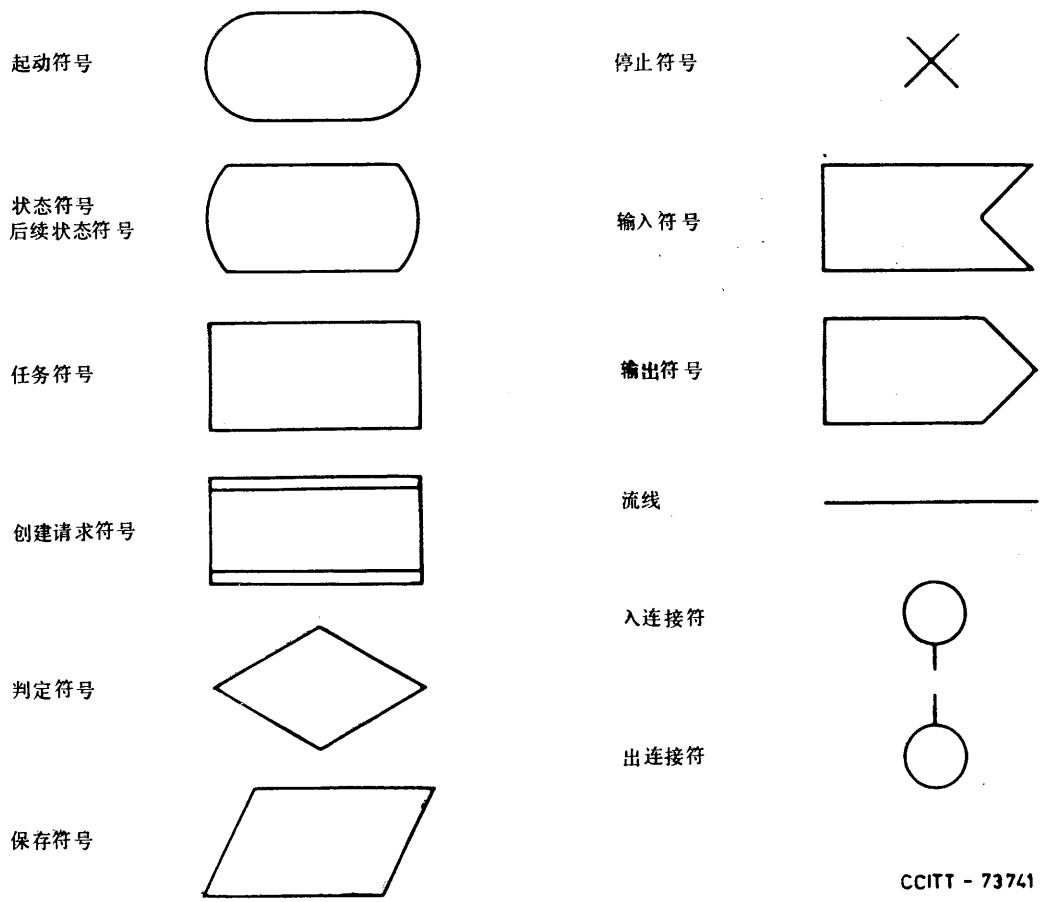


图 4/Z.101

SDL/GR进程图符号

节点的有向弧。在SDL-GR进程图中，用流线所作符号间允许的连接在图5/Z.101中示出。

形式参数、有效输入信号组、变量定义、视见定义、表达式以及视见表达式都用SDL/PR语法规定。

起动符号代表一个起动节点（见§ 2.2.3.3)。起动符号含有它所描述的进程的名字。

停止符号代表停止节点。

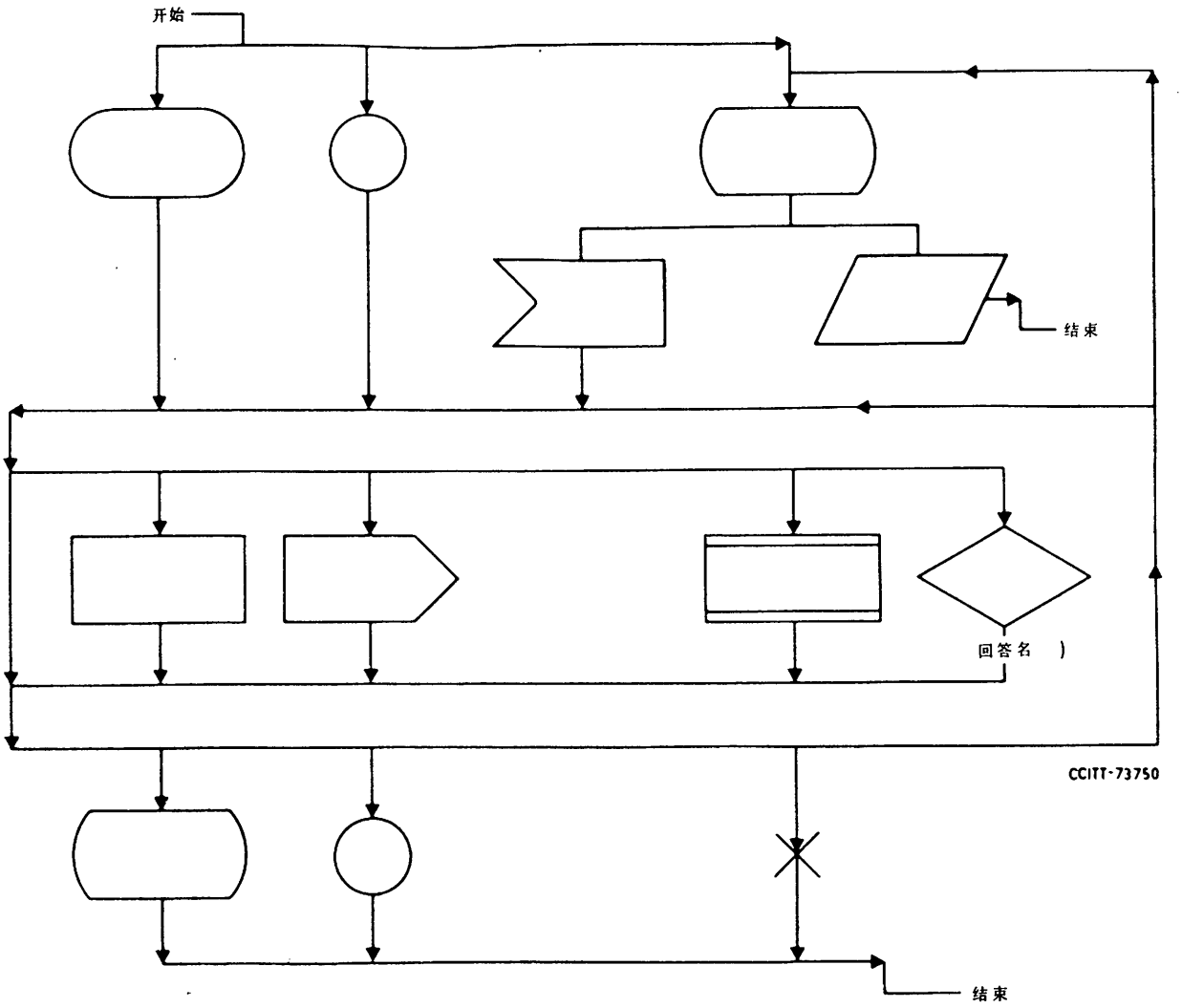


图 5/Z.101

在SDL/GR进程图中，用流线所作的符号间允许的连接

状态符号代表一个或多个状态节点，它含有一个或多个用逗号分开的状态名，或含有星号，或在星号后跟以用括号括起来的一串状态名。

保存符号表示附在状态节点上的一组保存信号。它含有一个或多个用逗号分开的信号名，或含有一个星号。

输入符号表示一个或多个输入节点，包含在输入符号中的信号名用逗号分隔。每一个信号名给出了这个输入符号所表示的一个输入节点的名字。

任务符号表示一个任务节点。任务符号含有任务名，并可含有一系列语句或非形式文句。

创建请求符号表示一个创建请求节点。它含有一个用SDL/PR语法规定的创建请求动作。

判定符号表示一个判定节点。它含有一个问题，并可含有一个判定名。两条或更多的流线从判定符号连向

其他符号；每根这样的流线上附有各自的回答名（即写在流线旁或插入断开的流线中）。ＥＬＳＥ回答意指没被任何其它的回答名包括在内的任何回答。

输出符号表示一个或多个输出节点。包含在输出符号中的信号名用逗号分隔。每个信号名给出了这个输出符号所表示的输出节点的名字。目的进程实例标识符能用ＳＤＬ/ＰＲ语法在输出符号中任选地给出。(亦即，关键字ＴＯ后面接类型进程实例标识符的一个表达式，关键字ＴＯ写在一串信号名之后)。当根据信号名或根据上下文还不足以唯一地决定一个信号的目的进程时，就需要进程实例标识符。

后续状态符号表示把跃迁串中最后一个节点连接到下一个状态节点的一条弧。

连接两个其它符号（代表节点）的流线表示连接相应节点的弧。

注释符号用来给任何其它的符号加注非形式文句。

入连接符含有一个标号，它表示从含有相同标号的相应的出连接符到该入连接符之间流线是连通的。

3.3.3 绘图约定

3.3.3.1 隐含的跃迁

对一个进程的每个状态，ＳＤＬ的抽象语法要求为该进程中有效输入信号组中的每一个信号规定一个保存或一个导致跃迁的输入。对于任何信号，如果既不明显地给出跃迁也不明显地给出保存，那么ＳＤＬ/ＧＲ将提供隐含的“空”跃迁。空跃迁等效于从一个状态符号连接到一个输入符号，然后又返回连接到原来的状态符号。于是对一个给定的状态，ＳＤＬ/ＧＲ就把该状态所没有明显地提及的任何信号舍弃。包含在这些信号中的数据也被舍弃。

如果ＳＤＬ/ＧＲ进程图中没出现起动符号，则假定为有一个隐含的起动符号。直接连到“进行起动的”状态符号，进行起动的状态可以从其名字隐含地识别，或通过一个注释加以识别。

3.3.3.2 流线和连接符

两个或多个符号后面接单个符号时，通向后面那个符号的流线就汇聚。当一条流线流入另一条时，或者当两个或多个出连接符与单个入连接符相连接时，或者当多根独立的流线进入相同的符号时，就会出现这种汇聚。

一个符号后面接两个或两个以上的其他符号时，从前面那个符号来的流线就会分叉发散成为两条或多条流线。

每当两条流线汇聚，以及每当一条流线进入一个出连接符或进入一个状态符号时，都需要画箭头。在进入输入符号的流线上禁止用箭头。在所有其他情况下，箭头可任选。

3.3.3.3 多重出现

每当多个状态符号，或者多个连接符符号以相同的名字出现时，就认为它们代表同一个语义实体。结果所得到的实体被认为具有从它的所有重复表示之处进入或走出的所有流线的并集。每当在同一个进程图中出现一个或多个停止符号时，它们全部代表停止节点。

3.3.3.4 简化符号

提供简化符号是为了允许在输入、保存或状态符号中引用全部的信号或状态，或引用全部其它的信号或状态。

附于一个状态的输入符号如果具有一个“＊”(星号)，则表明在该状态符号的重复出现处附有的输入符号或保存符号中没有再出现的所有的进入信号，都适用于下一次跃迁。对任何状态，只许可一个输入符号或一个保存符号具有一个星号“＊”。

附于一个状态的保存符号如果具有一个“＊”号，则表明在那个状态符号的重复出现处附有的输入符号中没有出现的所有信号，都应该保存。

状态符号中的“＊”号代表该进程中的全部状态，并表明在每一个状态中，都应解释后续的跃迁或保存。“＊”号后面紧接一串状态名（在括号内）表明除了所列出的这些状态以外，在其它的每个状态中都应解释后续的跃迁或保存。这样的状态符号绝不能有任何进入的流线。

后续状态符号中的“－”号表示下一个状态就是起动当前的跃迁的那个状态。跟在起动符号后的后续状态

符号中不允许有“－”号。

只有当后续状态符号和状态符号表示同一个状态符号时才能把它们合并。

3.3.3.5 其它

在任一图中相同类型的所有符号最好具有相同的尺寸。

符号的最佳定向是横向的，并且最佳的宽高比为2:1。

输入或输出符号的镜像（左右反置）是允许的。

流线应是水平的或垂直的，可具有折角。

交叉的流线没有逻辑关系。

与符号有关的文句应切合实际地放在该符号内。

3.3.3.6 SDL样板

适合于用手描画SDL符号的基本集的样板印在SDL用户指南中。

3.4 文句扩展符号

文句扩展符号可以连在所有SDL/GR的符号上。文句扩展符号中包含的文句将被认为是包含在文句扩展符号与之相连的符号之中。文句扩展符号如图6/Z.101所示。

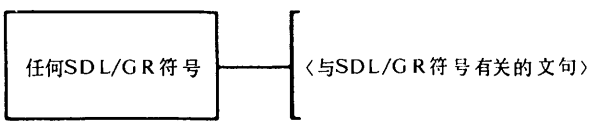


图 6/Z.101
文句扩展符号

3.5 SDL/GR中的注释

在所有SDL/GR图中，只要在用户觉得合适的地方都可插入注释。插入注释可以用SDL/GR注释符号（见图7/Z.101）或者用SDL/PR关于注释的语法（见§4.3.2,词法规则7）。

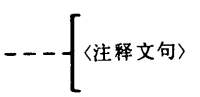


图 7/Z.101
注释符号

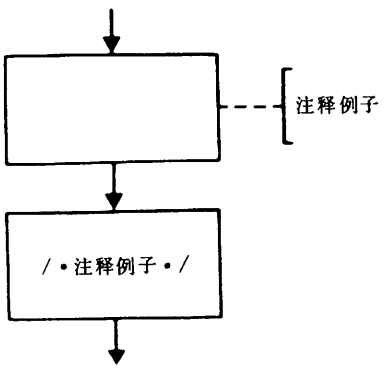


图 8/Z.101
SDL/GR中注释的例子

4 线性语法

4.1 概述

本节定义SDL/PR并建立它与通用语言模型的关系（见§2）。

在SDL/PR语法中系统定义用一系列语句来表示，以字SYSTEM开头，用字ENDSYSTEM结尾。

SDL/PR语法的详细规则包含在语法图中（见§4.3）。

在命名实体的定义以外引用该实体需要通过一个标识符。标识符由一个名字及任选的限定部分组成。当仅仅一个名字不能唯一地决定所指的项时，必须使用限定部分。

4.2 关键字

SDL/PR使用一些关键字来表达抽象语法中定义的SDL概念。有些关键字成对地使用以便把SDL的结构反映到SDL/PR中。

4.2.1 与定义有关的成对结构的关键字

对于这些成对的字使用“包围”一词以表明它们用作为定界符的作用。

SYSTEM ENDSYSTEM	包围系统定义的概念（一个系统的SDL/PR表示以关键字SYSTEM开始，以关键字ENDSYSTEM结束）。
BLOCK ENDBLOCK	包围一个功能块定义的概念。
PROCESS ENDPROCESS	包围一个进程定义的概念。

4.2.2 与定义有关的单个关键字

本节中的关键字用来表明这后面接一个定义。

DCL	引入对变量定义的描述。在DCL语句中使用关键字REVEALED以标明透露变量。
VIEWED	引入对视见定义的描述。
SIGNAL	引入对信号定义的描述。
CHANNEL	引入对信道定义的描述。在CHANNEL定义中使用关键字FROM表示信道的源功能块，使用关键字TO表示目的功能块。关键字ENV用来指定环境。关键字WITH后面接通过该信道传递的信号表。
FPAR	引入对形式参数定义的描述。
DURATION TIME TIMER NATURAL INTEGER REAL CHARSTRING CHARACTER BOOLEAN PID	表示预定义数据类型。
VIEW	引入对视见表达式的描述。它用在一个表达式内，用在使用声明为VIEWED的变量处。
SET	引入对设置语句的描述。
RESET	引入对复位语句的描述。

SYSTEM
BLOCK
PROCESS
} 引入标识符的限定部分。

4.2.3 在进程流图中与节点有关的关键字

用抽象语法描述的进程流图由多个节点组成，节点间用有向弧连接。

关键字的选择要与节点相对应，在抽象语法中连接节点的弧由关键字出现的次序来表达（见§4.2.5）。

START　表示起动节点。如果没有此关键字，则在PROCESS后的第一个STATE关键字就表示起动状态。

STATE　引入对一个或多个状态节点的描述。附于状态节点的保存信号组由关键字SAVE后接一个或多个信号标识符来表示。

INPUT　引入对一个或多个输入节点的描述。

TASK　引入对一个任务节点的描述。

OUTPUT　引入对一个或多个输出节点的描述。目的进程实例标识符可任选，由关键字TO后面接一个表达式给出，该表达式可产生一个进程实例标识符值。在信号的目的地不能被唯一地确定时，就要用关键字TO。

DECISION
ENDDECISION　包围一个判定节点的概念。关键字ELSE用来表示对所有没被明显地命名的情况的回答。

CREATE　引入对一个创建请求节点的描述。

STOP　表示一个停止节点。

4.2.4 与弧有关的关键字

JOIN　代表两个节点间的一条弧，这两个节点不是状态节点。一般，紧挨着关键字JOIN的前面那个关键字就代表第一个节点，而第二个节点总是具有同关键字JOIN相同的标号标识符以便识别。就第一个节点而言，这条一般性的解释有几个例外（见§4.2.5）。

如果第二个节点是另一个JOIN，此弧就与这个JOIN所指的节点相连。

如果具有连接标号的关键字是NEXTSTATE，第二个节点就是具有相同名字的状态（NEXTSTATE规则有效）。

NEXTSTATE代表一条弧。紧挨着关键字NEXTSTATE前面那个关键字就表示该弧的第一个节点。而第二个节点就是具有相同名字的状态。

4.2.5 SDL/PR中弧的表示

SDL/PR中一条弧的表示规则由关键字的次序给出。

正如在前一节所说的，在关键字为JOIN和NEXTSTATE的情况下，这个一般性的意义有所例外。

而且，当一个关键字（与一个节点或一条弧有关）紧接在一个回答的后面时，此弧的第一个节点就是前面的相配合的判定。

如果在一个节点或一条弧的关键字前面且紧挨着的那个关键字是ENDDECISION，那么，这些弧的第一个节点可由所有判定的跃迁串中的最后的关键字来表示，而这些判定是没有结束式语句的。

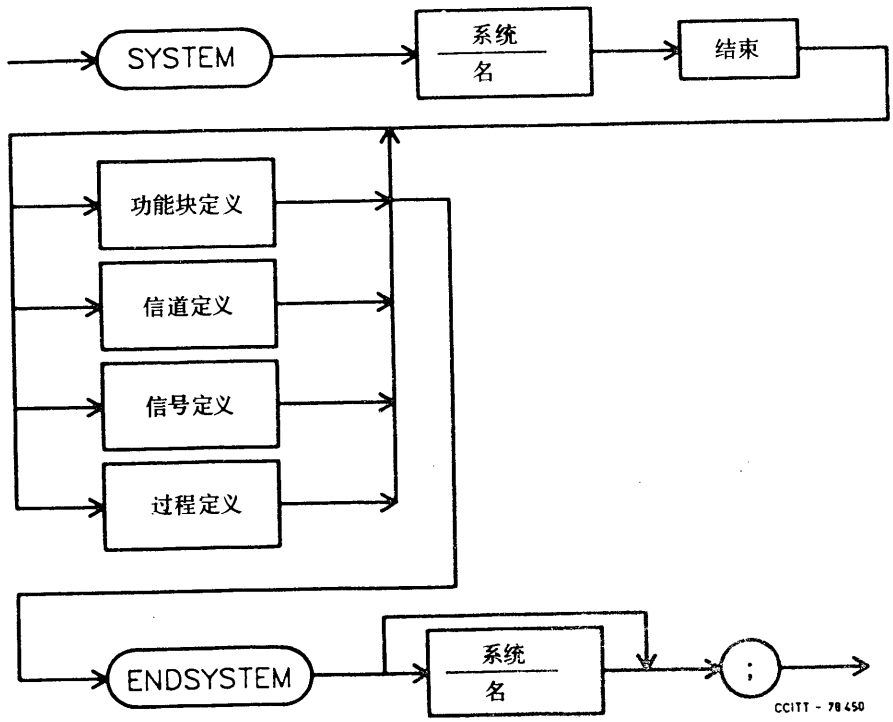
一个判定分支的最后的关键字表示一个节点，它并不与跟在下一个结果名字后的关键字连接，而与ENDDECISION后的关键字连接。如果一个判定分支的最后关键字是一个结束式语句，则此规则显然无效。

4.3 SDL/PR中的保留字

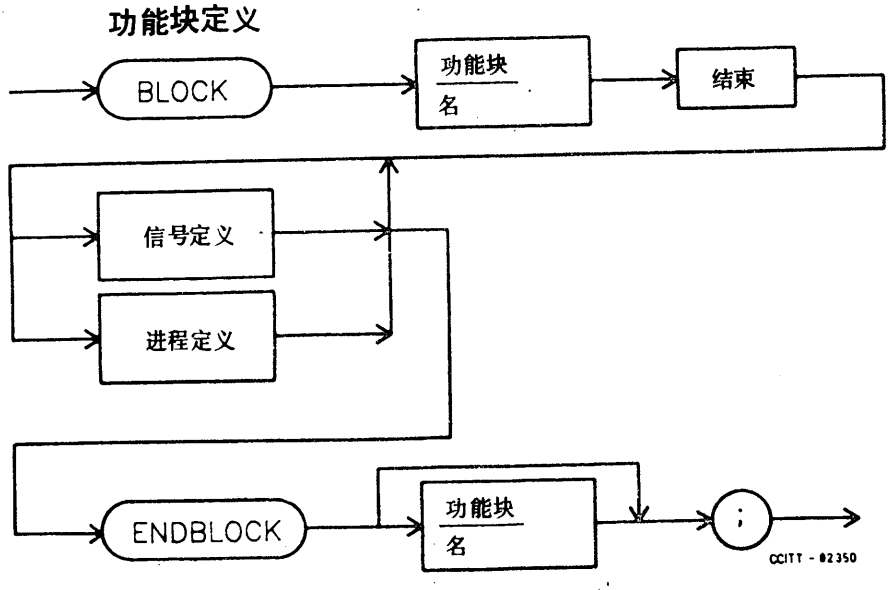
SDL/PR中保留了一些字，不能作为名字使用。保留字的列表可见建议Z.100至Z.104的附件C.2。

4.4 语法图

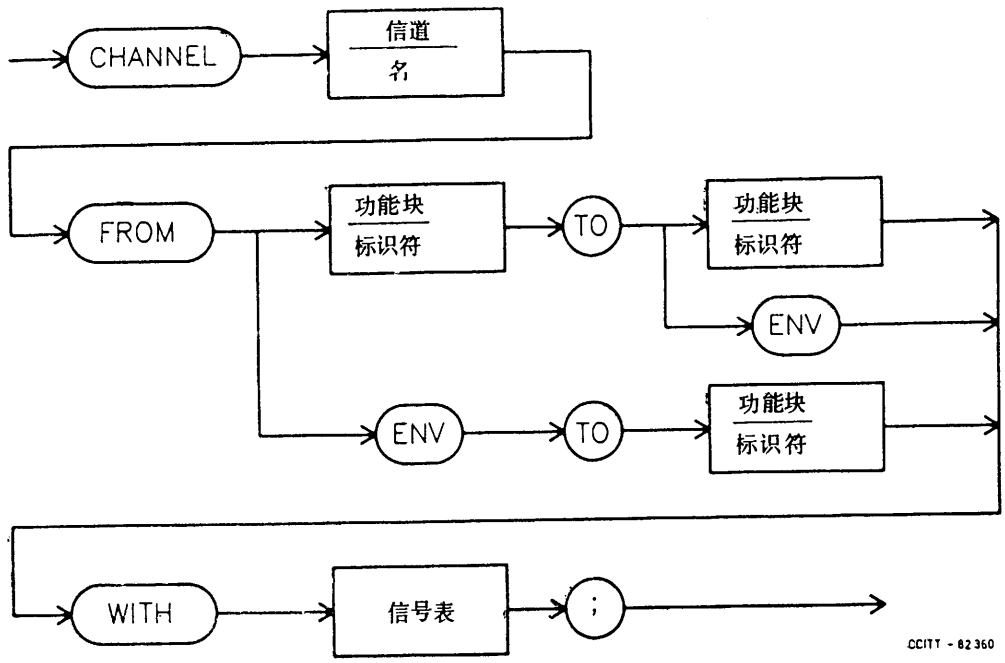
系统定义



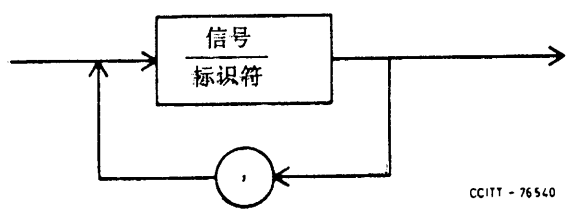
功能块定义



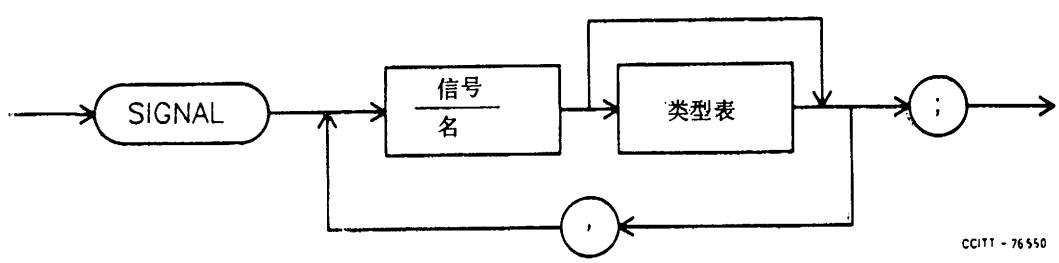
信道定义



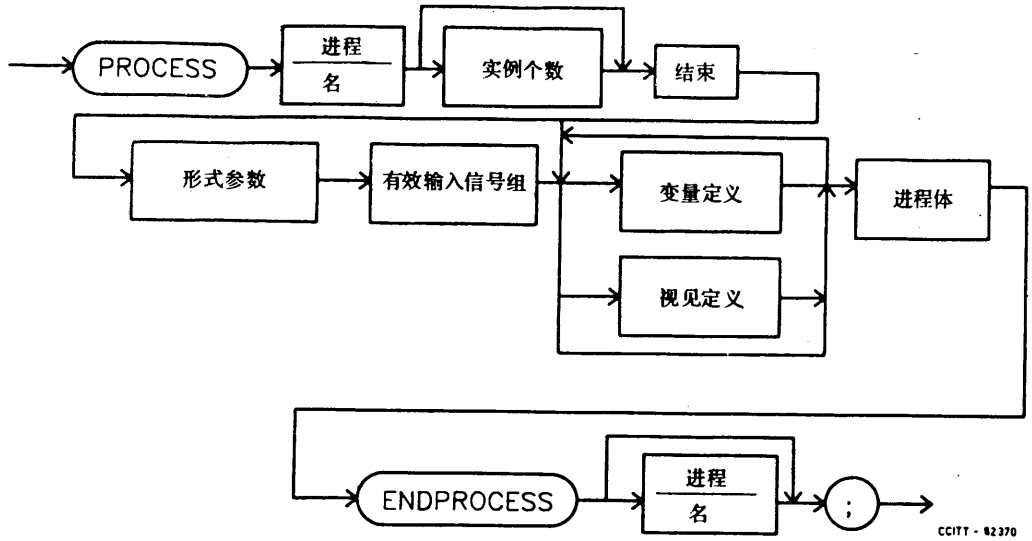
信号表



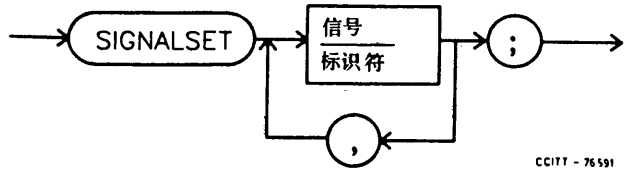
信号定义



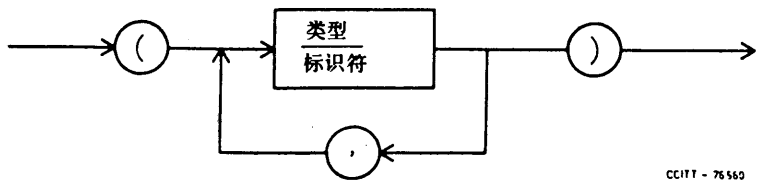
进程定义



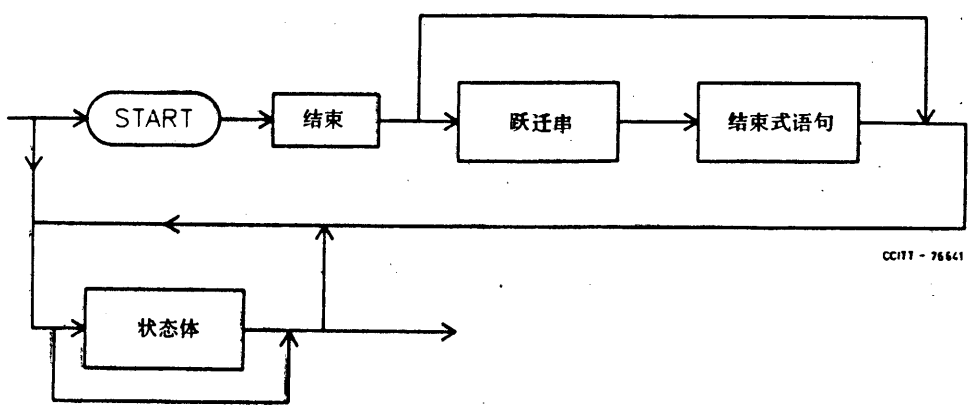
有效输入信号组



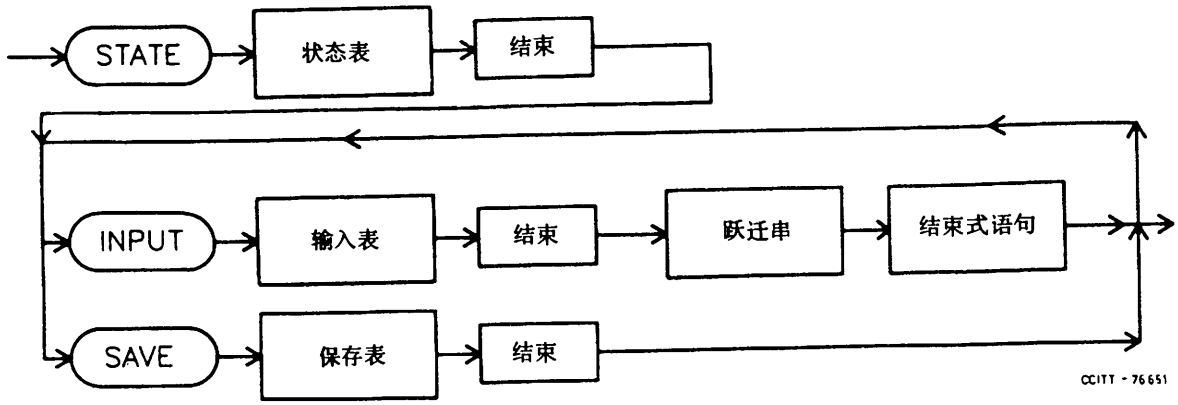
类型表



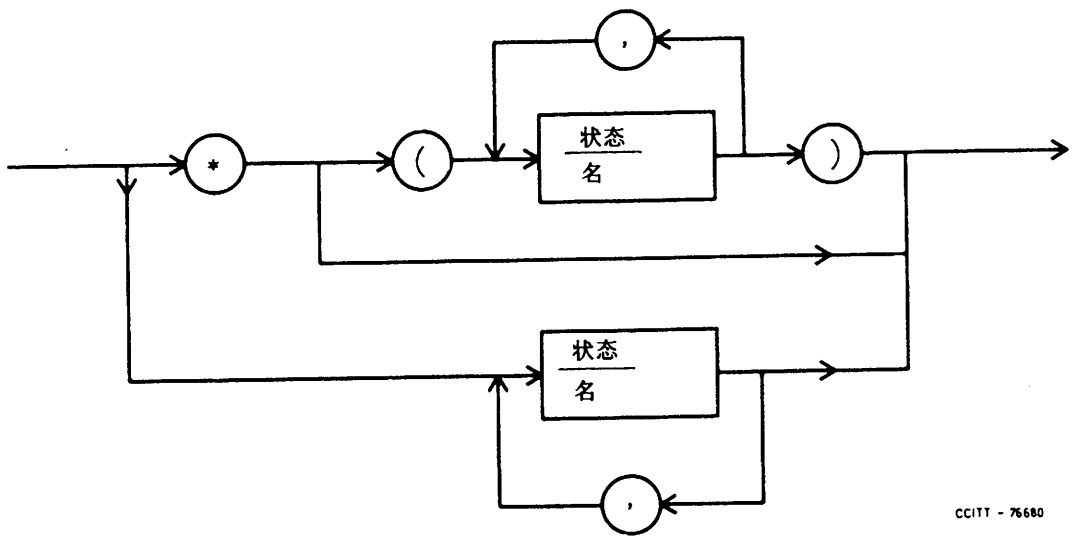
进程体



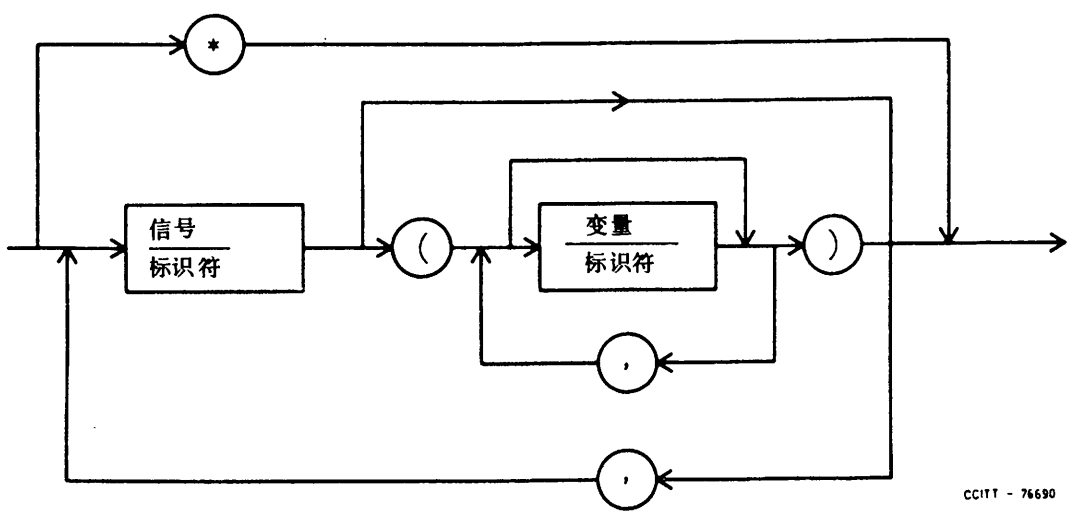
状态体


STATE
状态表
结束
INPUT
输入表
结束
跃迁串
结束式语句
SAVE
保存表
结束
CCITT - 76651

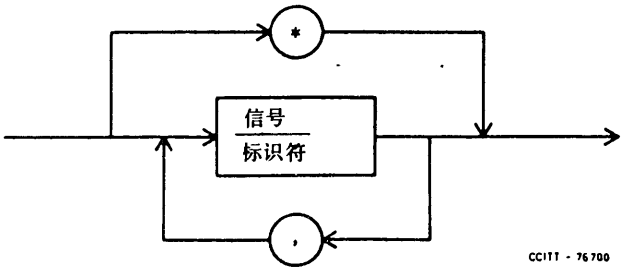
状态表


,
*
(
状态
名
)
状态
名
,
CCITT - 76680

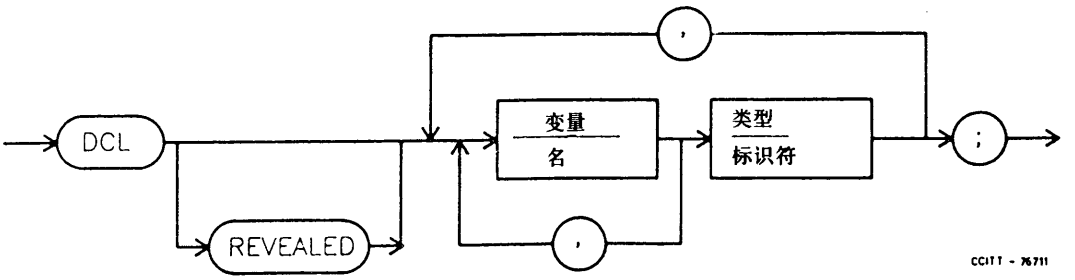
输入表


*
信号
标识符
(
变量
标识符
)
,
,
CCITT - 76690

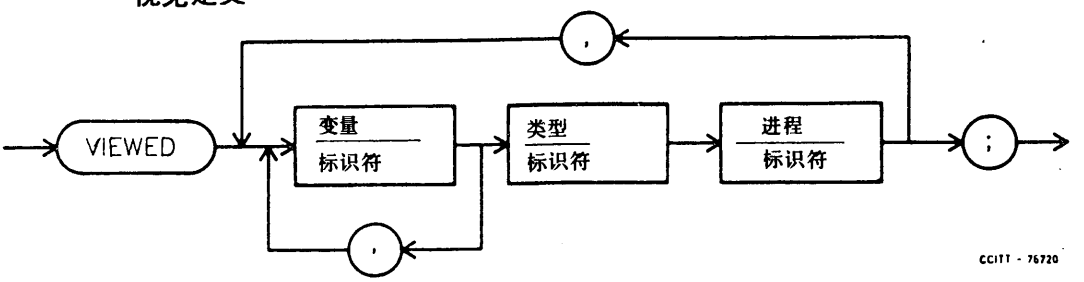
保存表



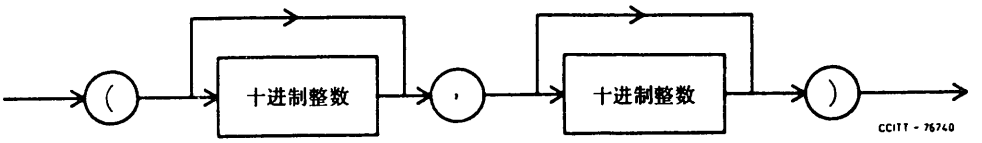
变量定义



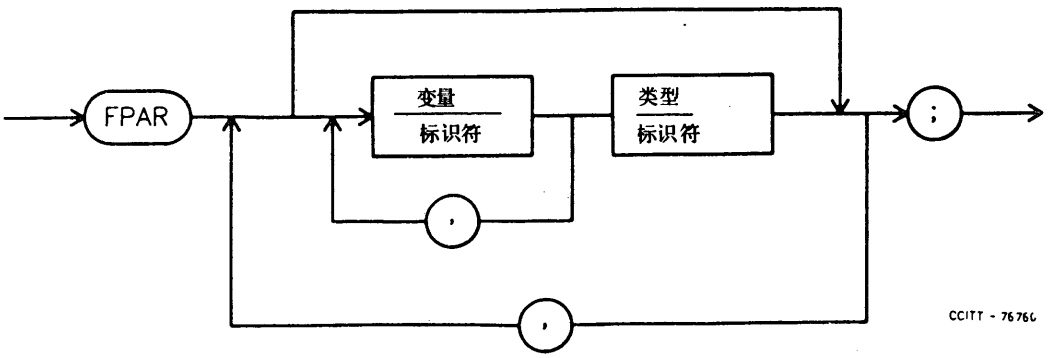
视见定义



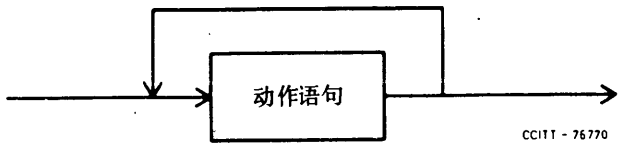
实例个数



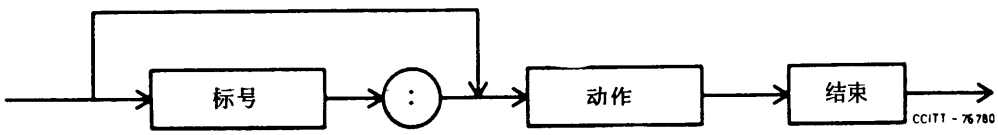
形式参数



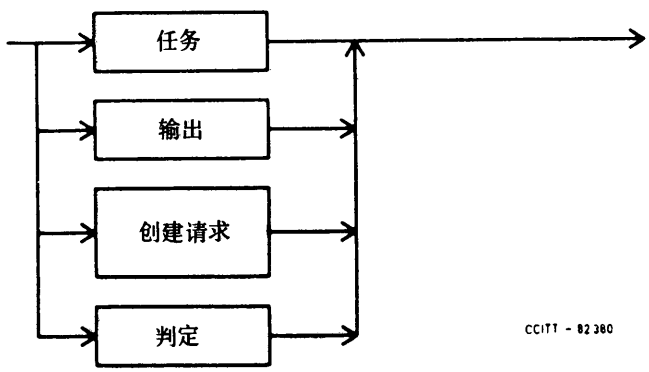
跃迁串



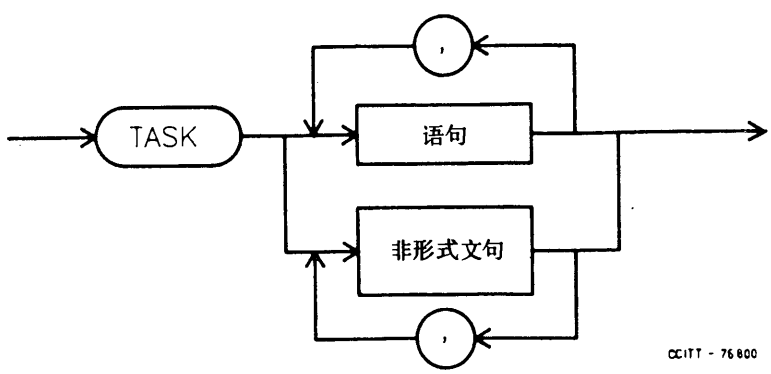
动作语句



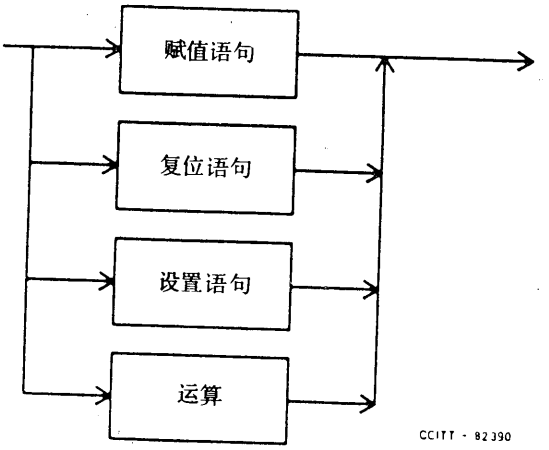
动作



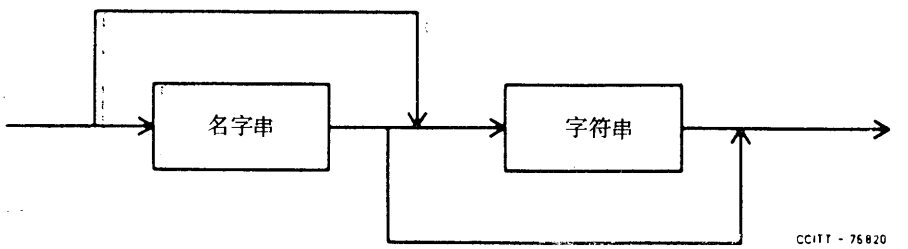
任务



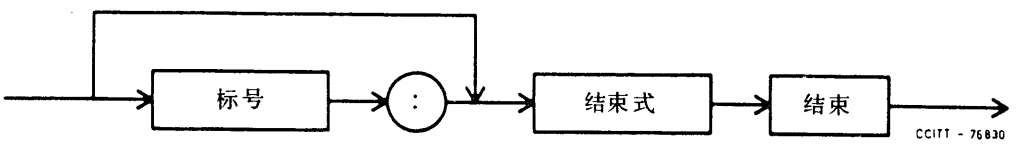
语句



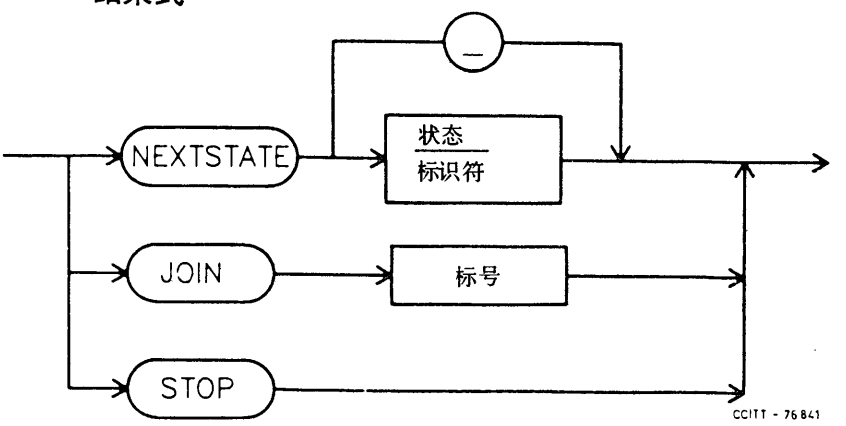
非形式文句



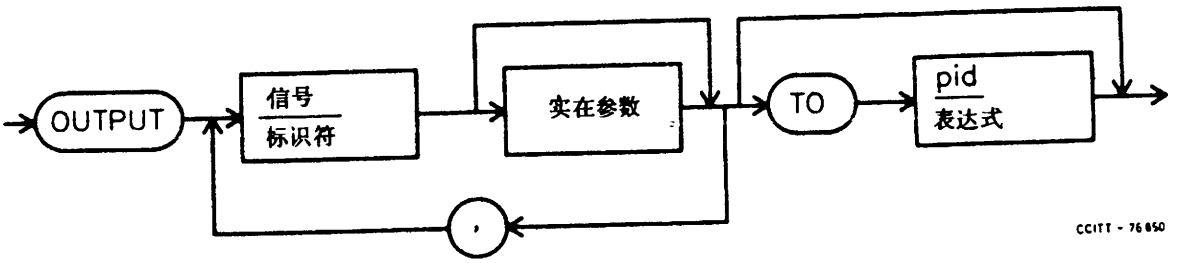
结束式语句



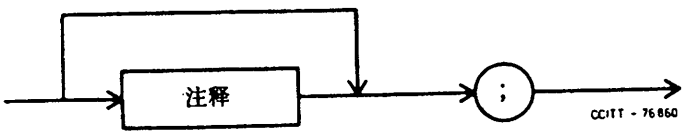
结束式



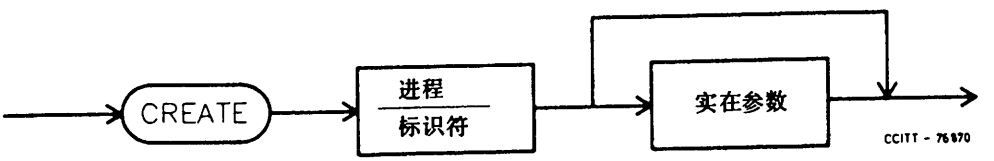
输出



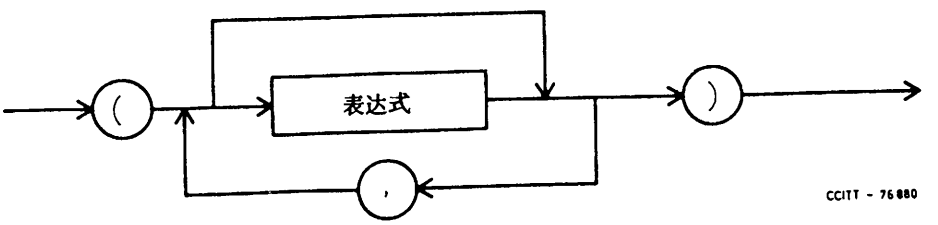
结束



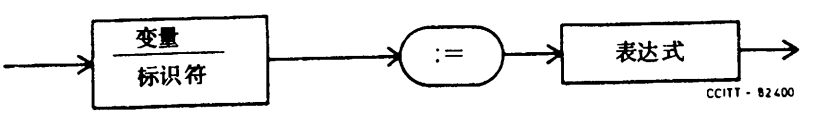
创建请求



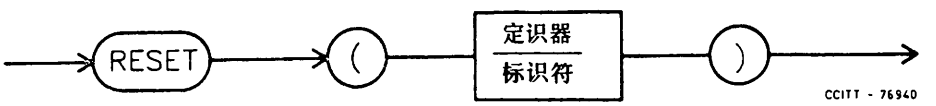
实在参数



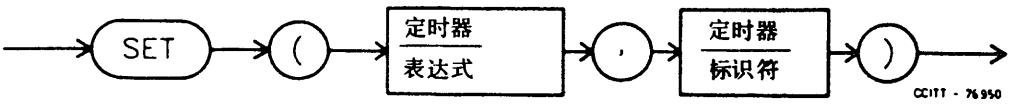
赋值语句



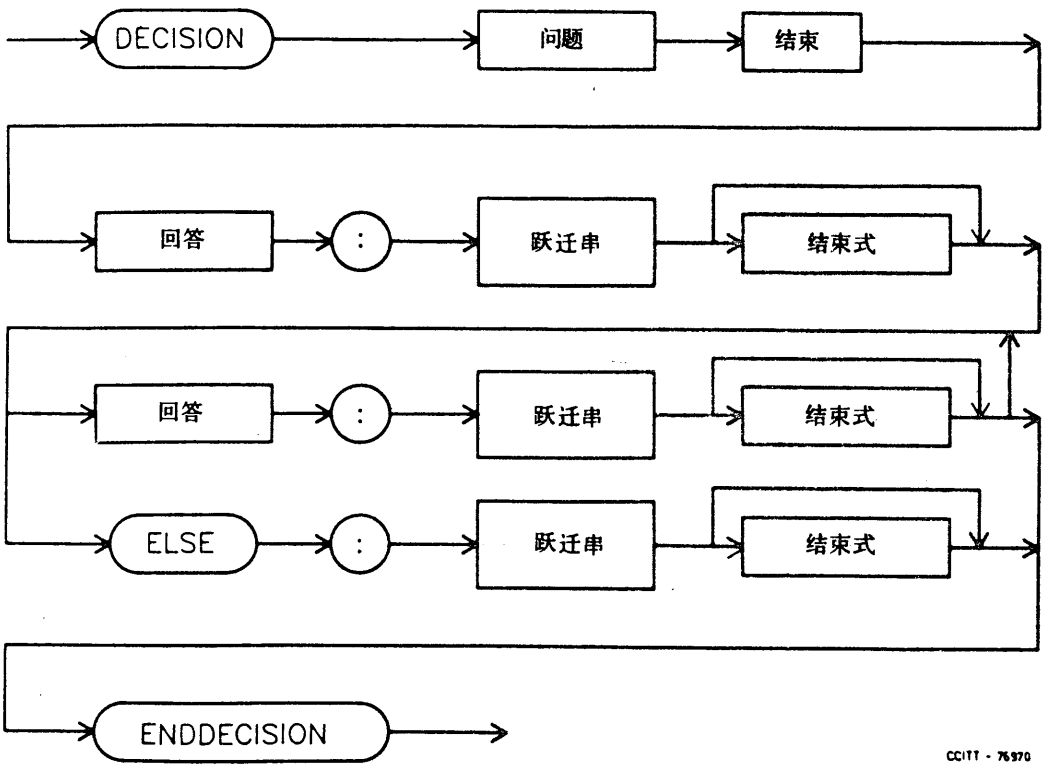
复位语句



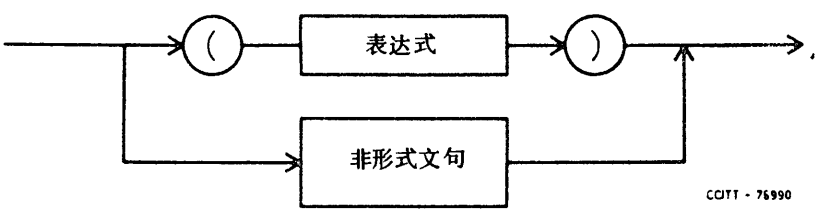
设置语句


SET
(
定时器
表达式
,
定时器
标识符
)
CCITT - 76950

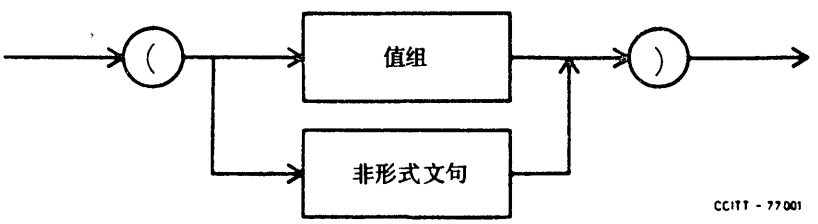
判定


DECISION
问题
结束
回答
:
跃迁串
结束式
回答
:
跃迁串
结束式
ELSE
:
跃迁串
结束式
ENDDECISION
CCITT - 76970

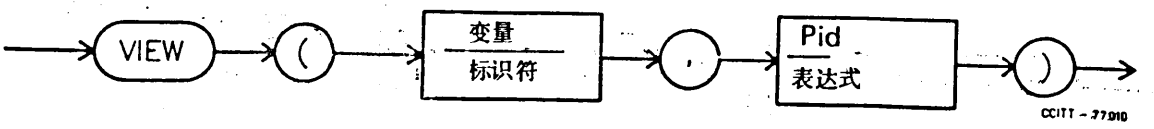
问题


(
表达式
)
非形式文句
CCITT - 76990

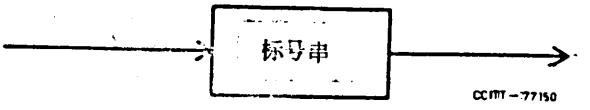
回答


(
值组
)
非形式文句
CCITT - 77001

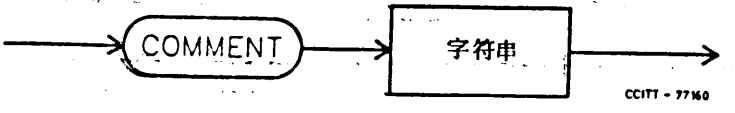
视见运算符



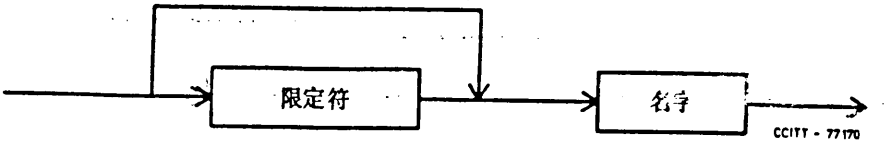
标号



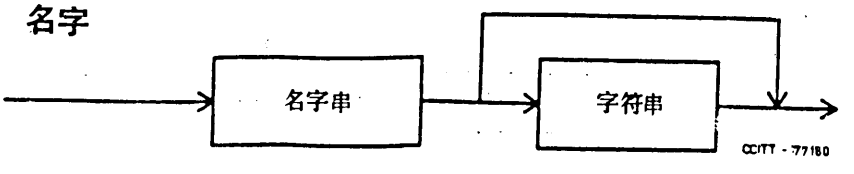
注释



标识符



名字



4.4.1 词法单元

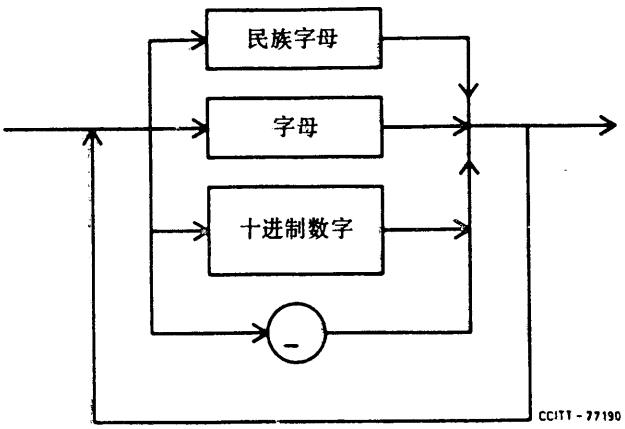
4.4.1.1 词法规则

- 所有标点符号［例如，．：’：！ =（ ）］以及运算符（例如＋、－、＊、<、>…）都是可以替换空格的词法单元。
- 两个词法单元必须用一个或多个空格分开。
- 关键字属于与名字串一样的词法种类，并且它们是保留字。
- 词法单元外的多个空格与一个空格“意义”相同。
- 表格字符（VT，HT，CR，BS…）可看作为空格。

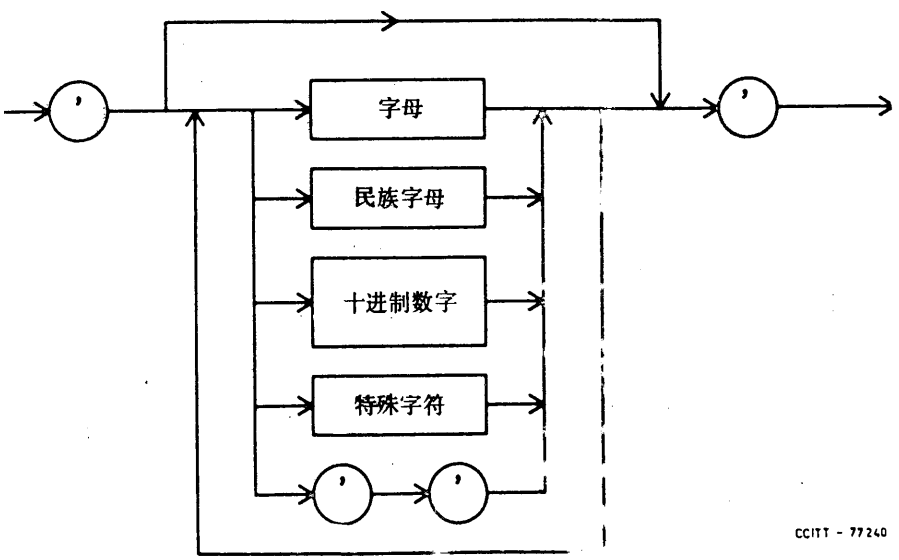
– 除字符串以外，所有字母和民族字母都被作为大写字母来解释。
– 在出现空格之处都可插入注释，用“/*”和“*/”来定界。这些注释与一个空格意义相同。注释中绝对不能含有“*/”这个特定的序列。

4.4.1.2 语法图

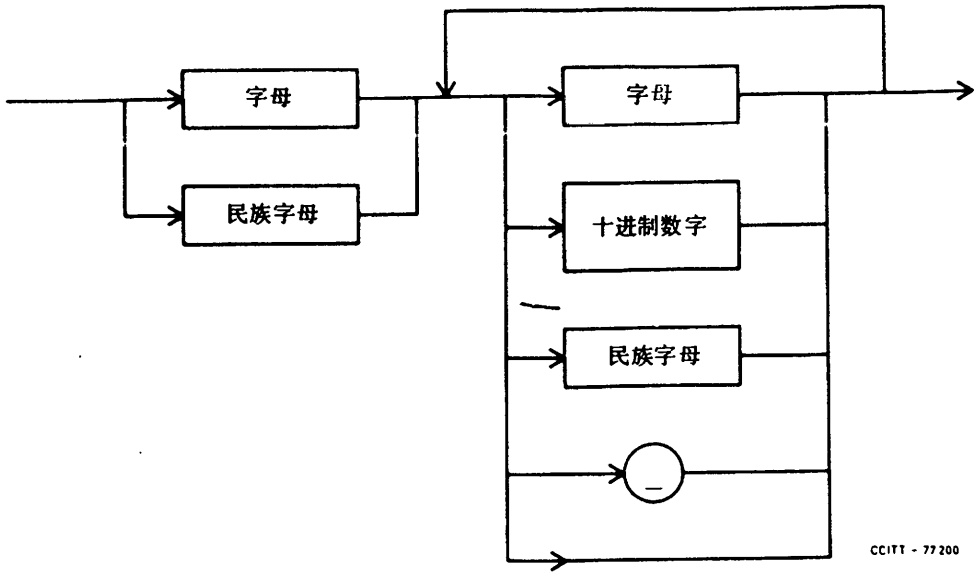
标号串



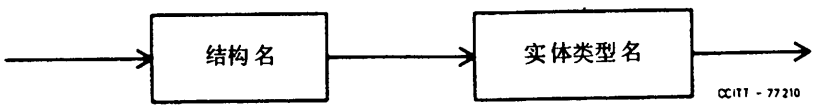
字符串



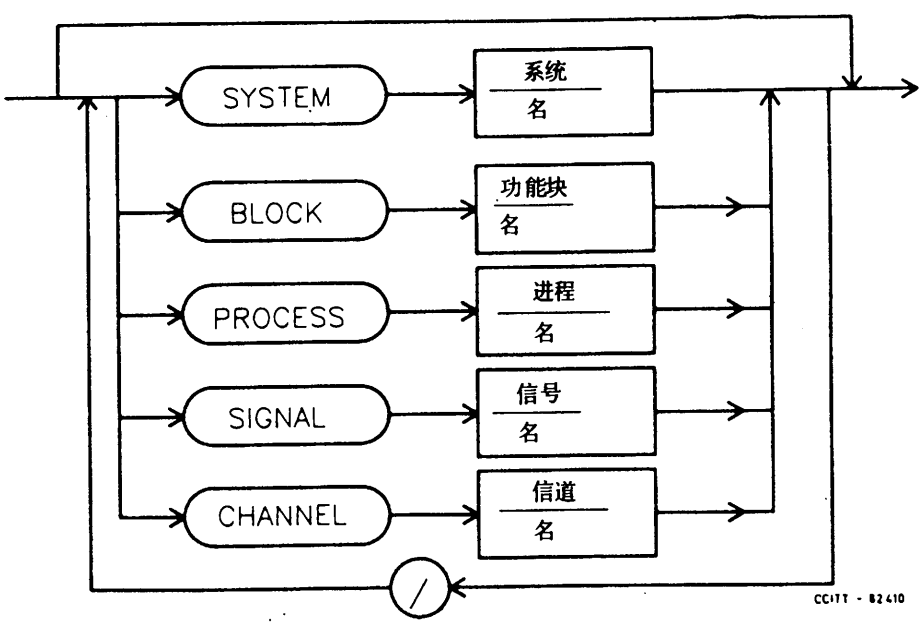
名字串


字母
民族字母
字母
十进制数字
民族字母
_
CCITT - 77200

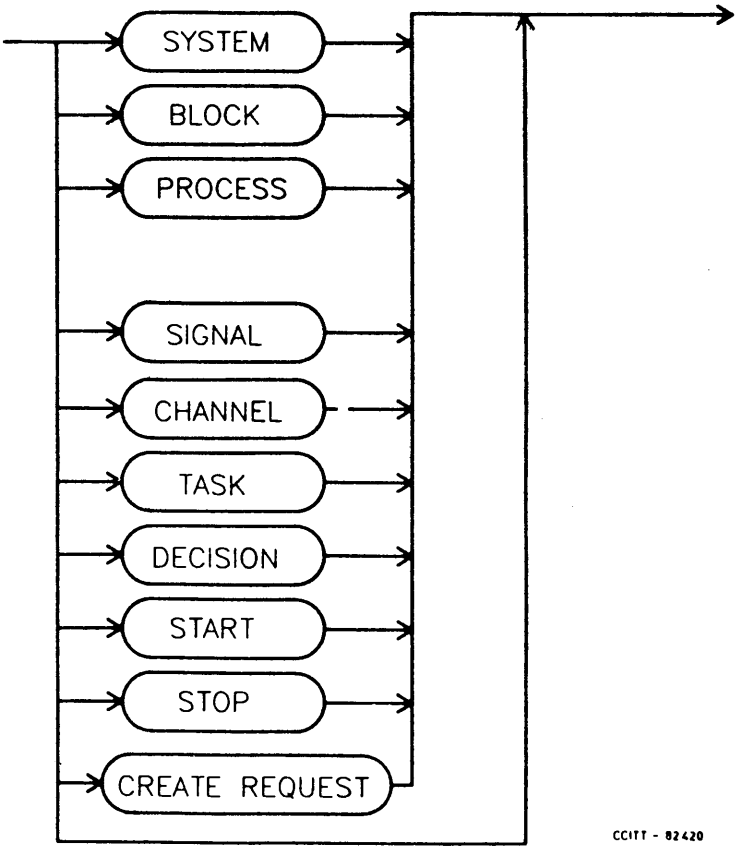
限定符


结构名
实体类型名
CCITT - 77210

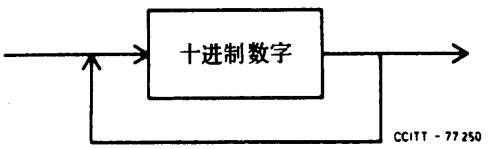
结构名


SYSTEM
系统
名
BLOCK
功能块
名
PROCESS
进程
名
SIGNAL
信号
名
CHANNEL
信道
名
/
CCITT - 82410

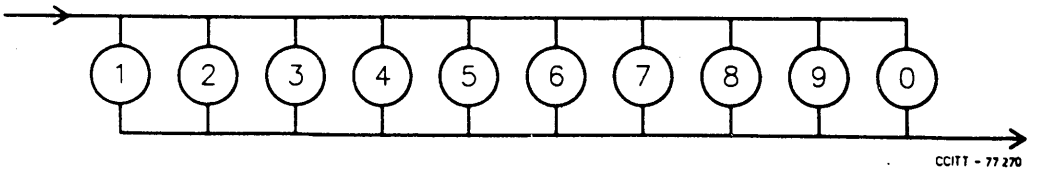
实体类型名



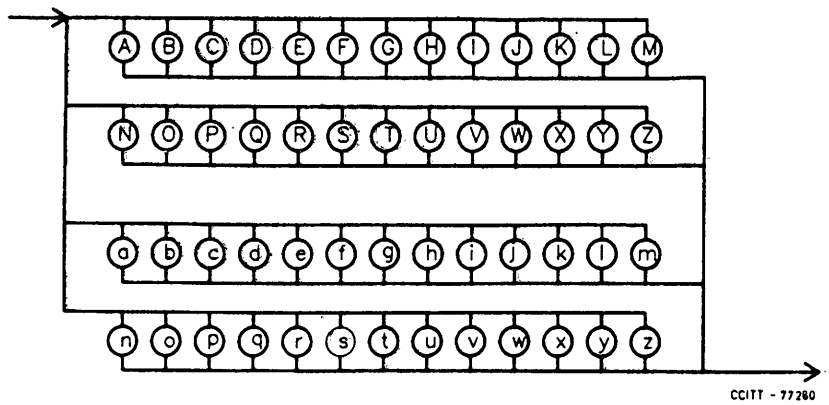
十进制整数



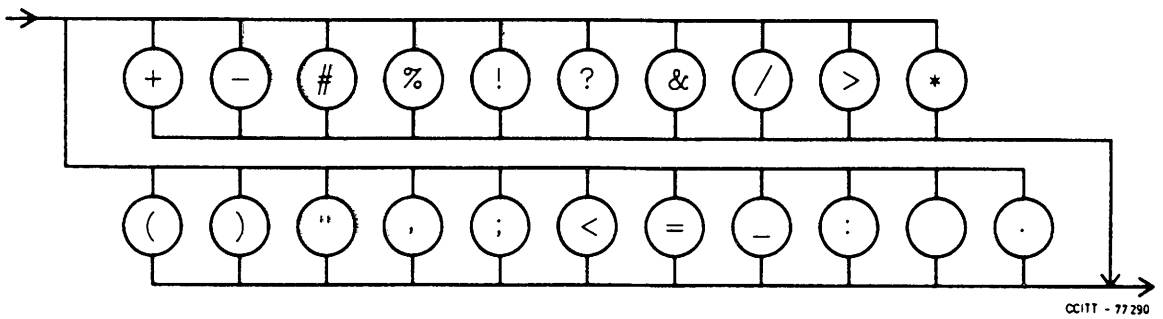
十进制数字



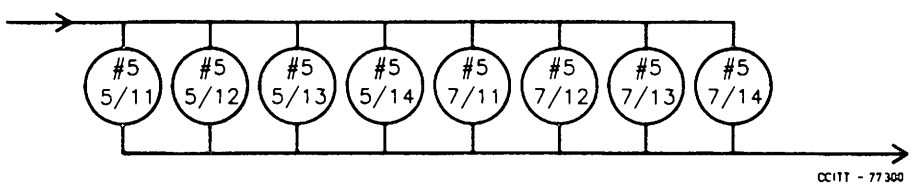
字母



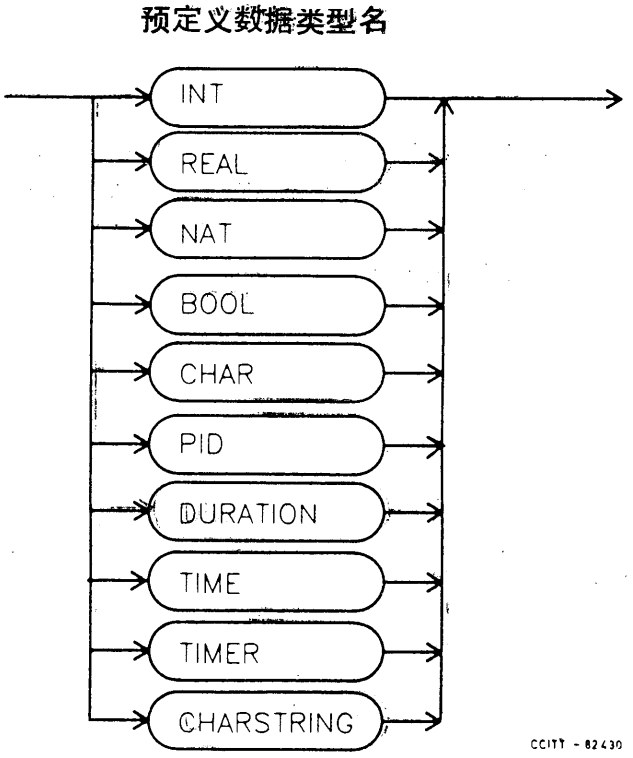
特殊字符



民族字母



上面涉及的位置可参见CCITT第5号国际字母表中为各国自己的需要而保留的位置



建 议 Z.102

SDL 中 的 结 构 化 概 念

1 引言

本建议定义SDL中处理分层结构所需的若干概念。这些概念的基础是建议Z.101中所定义的SDL，并且，所定义的概念就是对那些在建议Z.101中定义的概念的严谨补充。本建议中所包含的定义与那些在建议Z.103和Z.104中的定义毫无矛盾之处。

在本建议中引入这些概念的目的是为SDL用户描述大型的和（或）复杂的系统提供手段。在Z.101中定义的SDL适用于规定或描述比较小的系统，它可在用功能块组成的一个层内理解和处理。当要表达一个较大的或复杂的系统时，必须把系统规格或描述划分为易于处理的单元，分别单独地处理和理解。用若干步骤完成划分常常是合适的，最后形成一个用来表示系统的用单元组成的分级结构。

为了规定或描述所需要的系统结构或实际的系统结构，也需要使用结构化概念。

本建议定义的一些概念用于把：

功能块　　划分成子功能块、子信道和新的信道，

信道　　划分成功能块和信道，

进程　　划分成子进程。

这些概念使得最后所得到的分级结构能够代表系统，并且将给读者提供一系列的系统概貌，从中读者可以获得一个总体的理解，然后再转到更为详细的描述。这也意味着这些概念将支持“逐步求精”的设计技术，通

过若干步骤增加更为详细的信息。

2 通用语言模型

2.1 概述

在建议Z.101中，一个系统可描述为由一组功能块所组成。用单向信道把功能块与功能块以及把功能块与系统边界连接起来。此建议引入了一些概念，以便叙述如何把功能块、信道和进程划分成子成分。

系统中的每个功能块可划分成一个或多个子功能块。在此划分过程中引入了新的信道来连接各个子功能块。此外，终止于或起始于被划分的功能块的信道可以划分成子信道。

功能块A可划分为：

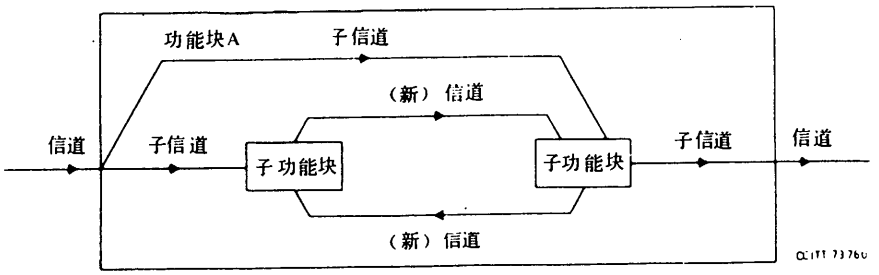


图 1/Z . 102
功能块的划分

子功能块本身又是一个功能块，并且仍可划分。此划分可重复任意次，最后得到功能块和其子功能块的一个分级结构。我们说，一个功能块的子功能块位于功能块树中的下一层中。

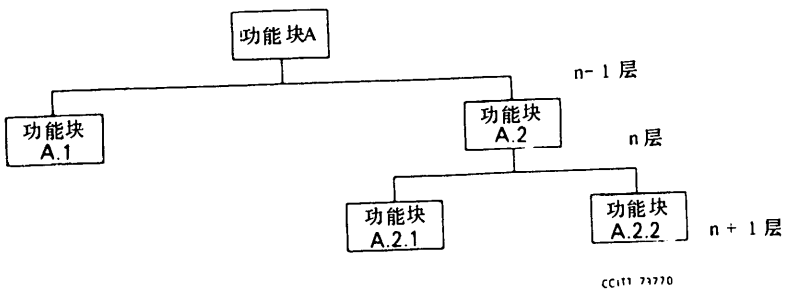


图 2/Z .102
功能块树

当一个功能块被划分成子功能块时，建议Z .101中一个功能块定义应该包含一个或多个进程定义这一规则是不严密的，它仅对于不被进一步划分的功能块成立（即描述系统的功能块树中，“叶”功能块必须包含进程）。然而，如果一个功能块定义包含进程定义，则子功能块定义至少必须包含由进程的划分产生的子进程定义。

由于划分功能块而产生的子信道是信道。

信道也可划分，产生一组新的功能块、新的信道、一个进入信道、一个走出信道：

C[1_1]

CHANNEL C 可划分成：

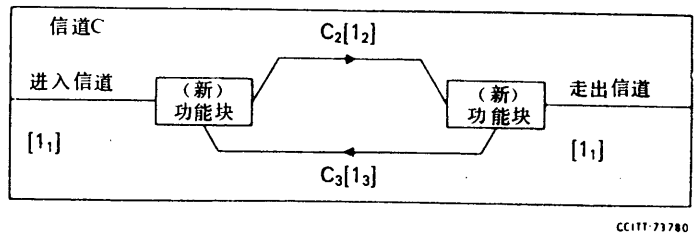


图 3/Z .102
信道的划分

注意，原来信道C的信号表要与进入信道和走出信道相联系。

功能块的划分和信道的划分都不改变被划分对象的界面。

一个进程定义可划分成一组子进程定义。当解释系统定义时，或者是该组子进程定义得到解释，或者是该进程定义得到解释，从这种意义上来讲，对这两种行为的描述是可以互换的。进程定义与相应的一组子进程定义是可以互换的。

子进程是一个进程，它本身又可被划分成子进程。所得到的进程的分级结构可用进程树表示。

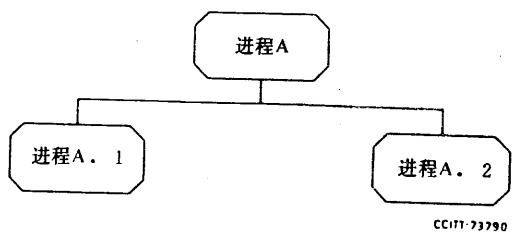


图 4/Z .102
进程-子进程的关系

由于子进程也是进程，由被划分的进程描述的行为可划分成一组并发的“子行为”。方式方法对保证划分正确是必需的，但不是SDL的一部分。

把一个功能块划分成子功能块，以及把它的进程划分成子进程可以同时进行。由于一个进程必须包含于一个功能块中，因此它的子进程必须包含于该功能块的子功能块中。该功能块就是包含该被划分进程的功能块。被划分的功能块和进程可看成是互相有关的分立结构：

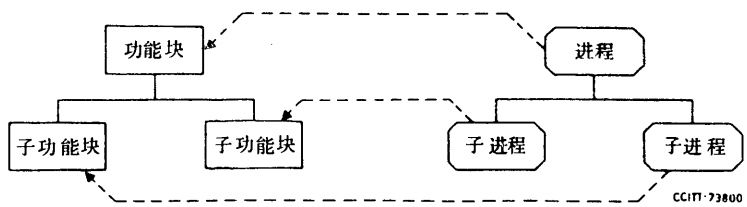


图 5/Z .102
功能块树与进程树间的关系

如果一个功能块被划分，那么它所包含的任一个进程也可被划分，在这种情况下，该进程的所有子进程必须出现在该功能块的子功能块中。如果该进程没有被划分，则它必须出现在其中一个子功能块中。为了进一步描述子功能块的行为，新的进程和信号也可包括在子功能块中，而不管原功能块中是否有进程。然而，“叶”功能块必须包含进程。

在系统的划分结构总体表示中，如果在多于一个的层次上出现进程定义，则可找到该表示的若干相容子集。一个相容子集是该总体表示中的功能块定义和进程定义的一种选择，使得：

a) 在功能块树中它包含最高的层次；

b) 如果它包含一个功能块，则它必须包含该功能块的父本功能块；

c) 如果它包含某个功能块的一个子功能块，它也必须包含该功能块的所有其它的子功能块；

d) 在得到的结构中所有“叶”功能块都包含有进程。

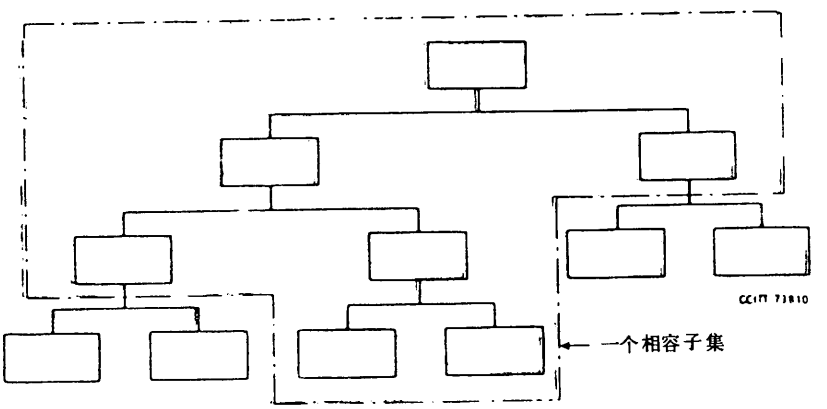


图 6/Z .102

系统表示的一个相容子集

如果在系统定义中起码有两层出现进程，则可找出该表示的若干个相容子集。这些子集表达了可互换的系统描述，描述的详细程度有所不同。它们可用来给读者既提供概述又提供详细的描述。可以选择它们以支持在不同抽象层次上对系统进行解释。如果存在若干个对系统行为的可互换的表示，则除了最详细的那个表示以外，其它所有的表示都将看成是该行为的概述。

2.2 抽象语法

本抽象语法以建议Z .101中给出的抽象语法为基础。此处仅对建议Z .101中的定义给以补充。

能块定义

功能块定义也可包含一个内部分块定义，如果这样，它就不必包含进程定义。

内部分块定义

内部分块定义包含一个功能块子结构定义；并且对于功能块定义中的每个进程定义，内部分块定义要包含一个进程子结构定义。它也可包含信号定义和数据类型定义。

功能块子结构定义

功能块子结构定义可包含一个或多个子功能块定义和一个或多个信道定义。

对包围的功能块的子信道的每个终止端点，必须至少有一个以该端点为起始端点的子信道定义，反之，对该包围的功能块的所有起始子信道端点也成立。子信道定义的信号表的并集必须与该功能块的信号表相同；此子信道定义与作为通向或来自该包围的功能块的子信道具有相同的端点。另外从终止端点发出的该子信道定义的信号表必须拆散。

在功能块子结构定义中包含的全部信道定义必须：要么把子功能块与该包围功能块的信道端点连接起来，要么把子功能块互相连接起来。

子功能块定义

子功能块定义就是一个功能块定义。

信道定义

信道定义也可包含一个信道子结构定义。

子信道定义

子信道定义就是一个信道定义。

信道子结构定义

信道子结构定义包含两个或多个信道定义、一个或多个功能块定义，并可包含信号定义。

在功能块子结构定义中包含的所有信道定义必须把子功能块互相连接起来，所有子信道定义必须把该包围功能块的信道端点与子功能块连接起来。

进程子结构定义

进程子结构定义与一个进程名对应，它包含一个或多个进程名，每一个进程名与一个子功能块名对应。

对应的子进程名必须是进程定义的名字，该进程定义包含于包围功能块定义中。所包含的进程名必须是子进程定义的名字，该子进程定义包含于具有有关的子功能块名的功能块定义中。

在有关进程的有效输入信号组中，每个信号名必须恰好出现在具有所含的子进程名的子进程定义的一个有效输入信号组中。附在有关进程的输出节点上的每个信号名必须附在子进程定义的至少一个输出节点上，这些子进程定义具有所包含的子进程名。

子进程定义

子进程定义就是一个进程定义。

2.3 解释

下面给出的解释规则定义是对建议Z.101中定义的相应规则集合的补充。

信道

如果一个信道定义包含一个信道子结构定义，那么，或者如建议Z.101中所定义的，对该信道进行解释，或者，对信道子结构定义进行解释。

如果对信道子结构定义进行解释，那么，送到该信道的起始端点的每个信号就被送到信道子结构，并且，由信道子结构传送的每个信号就被送到该信道的终止端点。

信道子结构

送到信道子结构的信号被送到进入信道，而由走出信道的终止端点传出的信号被送到包围的信道。

功能块

如果一个功能块定义包含一个或多个进程定义，并且也包含一个内部分块定义，那么，或者如建议Z.101中所定义的，对功能块予以解释，或者，对内部分块予以解释。如果该功能块不包含任何进程定义，则必须对内部分块进行解释。

如果内部分块被解释，则所有送到该功能块的信号将被送到内部分块，并且，所有由内部分块给出的信号将被进一步送往从该功能块出发的信道，就象从该功能块所包含的一个进程中把信号传来那样。

内部分块

包围功能块中的每个进程用一个进程子结构代替。

如果从包围功能块送往内部分块的一个信号递交给一个进程，则该信号就被送到进程子结构，或者被送到功能块子结构。

由功能块子结构给出的信号将送到该包围功能块。如果该信号从某个进程发出，此进程作为某进程子结构的一个子进程出现，那么，该信号的发送者属性就改为被替换进程的进程实例标识符。

功能块子结构

根据功能块子结构定义，一个功能块子结构包含功能块和信道，它们按照为功能块和信道而定义的规则进行解释。

进程子结构

进程子结构实例可替换所引用的进程定义的一个进程实例。它也表示一组子进程实例，对定义中的每个子进程名都有一个实例。每个子进程分配给具有相应功能块名的功能块。

送到进程子结构的每个信号将重新递交给在有效输入信号组中具有该信号名的子进程，并且被送到该包围功能块的功能块子结构。

3 图形语法

下面的图形语法是对建议Z.101中定义的语法的补充。此补充包括结构的表示和系统的划分。

系统结构的概观由功能块树图给出。在功能块互作用图和信道子结构图中表示了如何把功能块、进程和信道划分成为子元素。

描述系统的一套文件和图可能会很多。有必要通过互相参考和适当的标题使文件互相联系，但是，这种联系参考的语法手段并不成为SDL图形语法的一部分。

3.1 功能块树图

功能块树图意在给出系统结构的概观，即把系统划分成功能块的分级结构。在功能块互作用图中给出了如何用信道把功能块连接起来的细节。

此图的一部分也可用来给出如何把功能块划分成为子功能块的概况。在这种情况下，被划分的功能块用根框表示。

3.1.1 符号

功能块树图中使用的符号是一个框，用它来代表一个系统或一个功能块。所表示的对象的名字必须写在该框内。

把每个功能块（框）向下连接到它的子功能块(框)，从而形成一棵分级树，如下图7/Z.102中给出的例子：

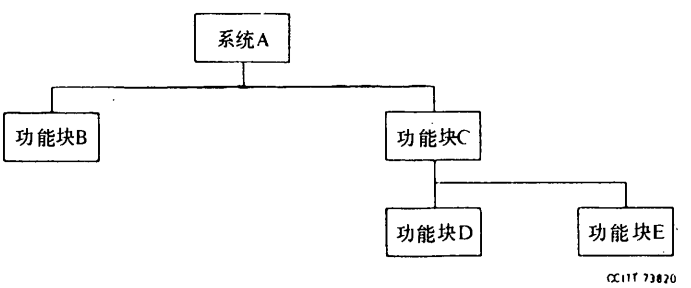


图 7/Z.102

功能块树图的例子

3.1.2 与SDL抽象语法的关系

所示的结构在系统定义中以及在系统中功能块的功能块子结构定义中，都具有它的等价物。

3.1.3 图形约定

树图最好这样来画，使得在同一层次的功能块在此图形表示中互相并排出现。

由于一个大系统的功能块树图也会很大，所以把此图分解成若干个图较为合适。分解应该这样进行：第一个图以系统作为根，此图被切断，使得一组需进一步划分的功能块好象并没有被划分。在随后的图中，这些功能块作为根出现。例如，在图8/Z.102中，把图7/Z.102中的图分解成为两个图。

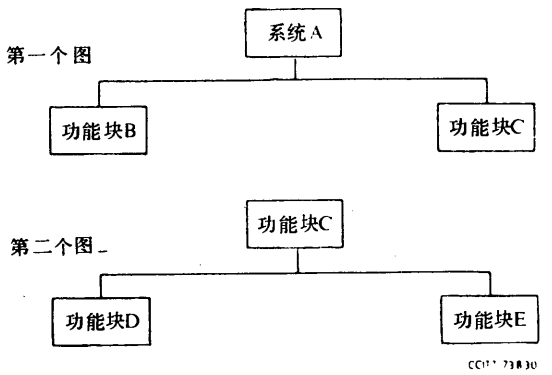


图 8/Z.102

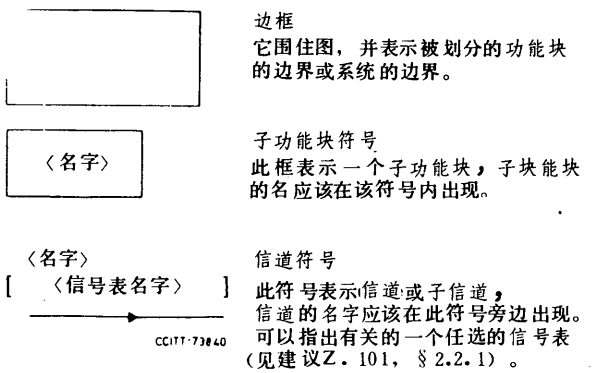
把一个功能块树图分解成若干个图的例子

3.2 功能块互作用图

此图表示了划分一个功能块为子功能块、子信道和（新的）信道的情况。此图基本上与建议Z.101中引入的功能块互作用图的形式相同，该图表示了如何把一个系统划分成功能块和信道。

3.2.1 符号

用于表示一个功能块互作用图的符号如下图9/Z.102中所示：



边框
它围住图，并表示被划分的功能块的边界或系统的边界。

子功能块符号
此框表示一个子功能块，子块能块的名应该在该符号内出现。

信道符号
此符号表示信道或子信道，信道的名字应该在此符号旁边出现。可以指出有关的一个任选的信号表（见建议Z.101，§2.2.1）。

图 9/Z.102

功能块互作用图中使用的符号

除了这些符号外，在图中可有使用SDL/PR的信号定义。

连接这些符号的规则与功能块互作用图中的一样（见建议Z.101），只有在图的标题上有例外，即应该标清

楚此图是某个功能块的功能块互作用图。图10/Z．102中给出了功能块互作用图的一个简单例子：

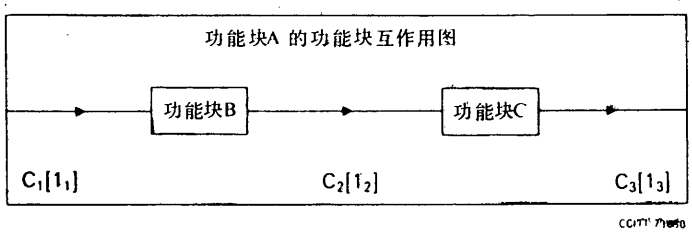


图10/Z．102
功能块互作用图的例子

3.2.2 与SDL抽象语法的关系

一个功能块互作用图代表一个功能块子结构定义。功能块子结构定义中所包含的定义用功能块以及信道符号连同用SDL/PR给出的信号定义来表示。

3.2.3 图形约定

建义Z．101中定义的、为互作用图而作的图形约定同样适用于功能块互作用图。

除此之外，在一个图内描述若干层次的功能块的划分常常是很有用的。它可以这样得到：在图中，把一个功能块符号用该功能块的功能块互作用图来代替。下图11/Z．102中给出了这样的例子：

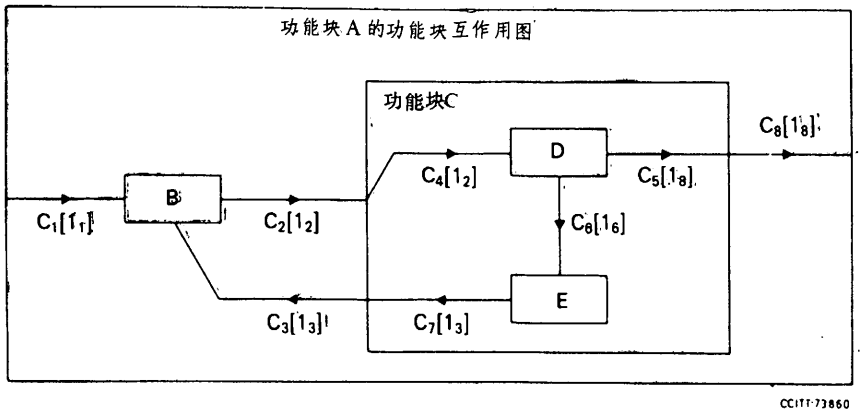


图11/Z．102
嵌套的功能块互作用图的例子

3.3 进程树图

进程树图描述把一个进程划分成为子进程的情况以及描述这些子进程配置在何处。

3.3.1 符号

用以组成一个进程树图的符号如下图12/Z．102所示：

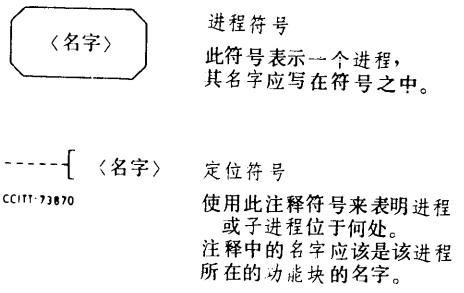


图12/Z．102

进程树图中使用的符号

每个进程向下与其子进程连接，形成一棵分级的树，如下图13/Z．102中所示：

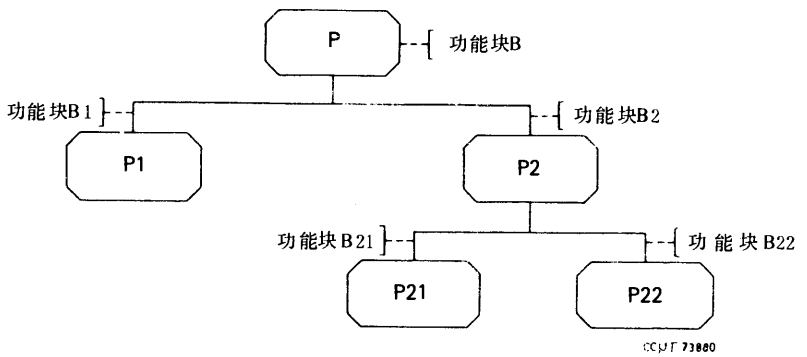


图13/Z．102

进程树图的例子

3.3.2　与SDL抽象语法的关系

此图表示了一个进程子结构定义。在根符号中的名字是有关的进程名，而叶符号中的名字是所包含的子进程名。定位符号中的功能块名是有关的子功能块名。

3.3.3　图形约定

最好这样来画树，使得属于同一划分层次的进程在图中并排地出现。

如果进程树图很大，把此图分解成若干个图较为合适。此分解应该这样进行：把第一个图切断，以使一些要进一步划分的进程好象没有被划分一样。在随后的图中这些进程作为根出现。例如，在图14/Z．102中，图13/Z．102中的图可分解成两个图。

3.4　信道子结构图

本图表示把一个信道划分成为子成分。由于这些成分是功能块和信道，所以本图类似于一个功能块互作用图。

3.4.1　符号

用于表示信道子结构图的符号如下图15/Z．102所示。

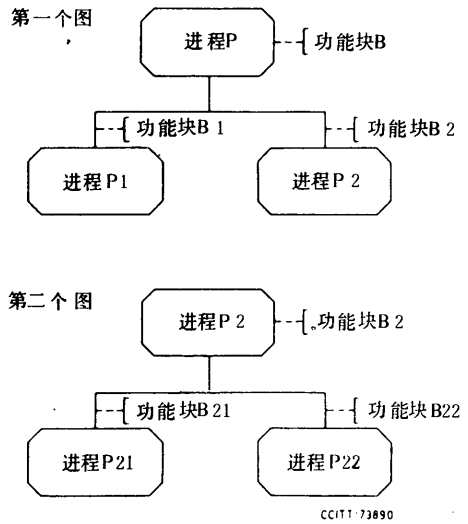


图14/Z . 102

把一个进程树图分解成若干个图的例子

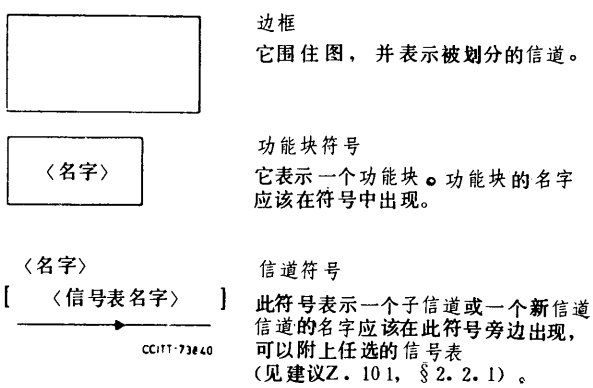


图15/Z . 102

信道子结构图中使用的符号

除了这些符号之外，图中也可有使用SDL/PR语法的信号定义。

连接此图的规则与连接互作用图（见建议Z . 101)的一样，只有此图的标题例外，即应该标清楚此图是一个信道子结构图。

下图16/Z . 102给出了信道子结构图的一个简单例子：

3.4.2 与SDL抽象语法的关系

信道子结构图表示了一个信道子结构定义。信道子结构定义中所含有的定义是用功能块和信道符号连同SDL/PR语法的定义一起来表达的。

从边框进入图内的信道表示进入信道，在边框终止的信道表示走出信道。

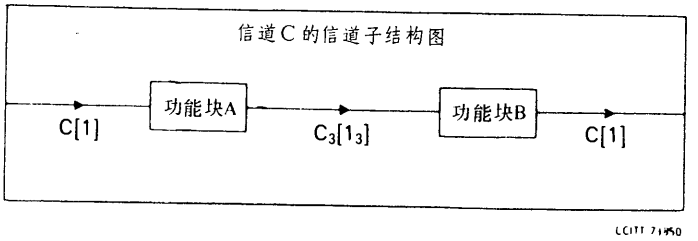


图16/Z．102
信道子结构图的例子

3.4.3 图形约定

建议Z．101中为互作用图而作的图形约定同样适用于信道子结构图。

4 SDL/PR

下面的SDL/PR语法是对建议Z.101中所定义的语法的补充。此补充包括系统的划分和系统结构的表示。

注意，在例子中，SDL/PR关键字用大写字母写出。

4.1 功能块定义

SDL/PR的功能块定义经扩展可任选地包括非终结的“功能块子结构定义”。

4.1.1 语法

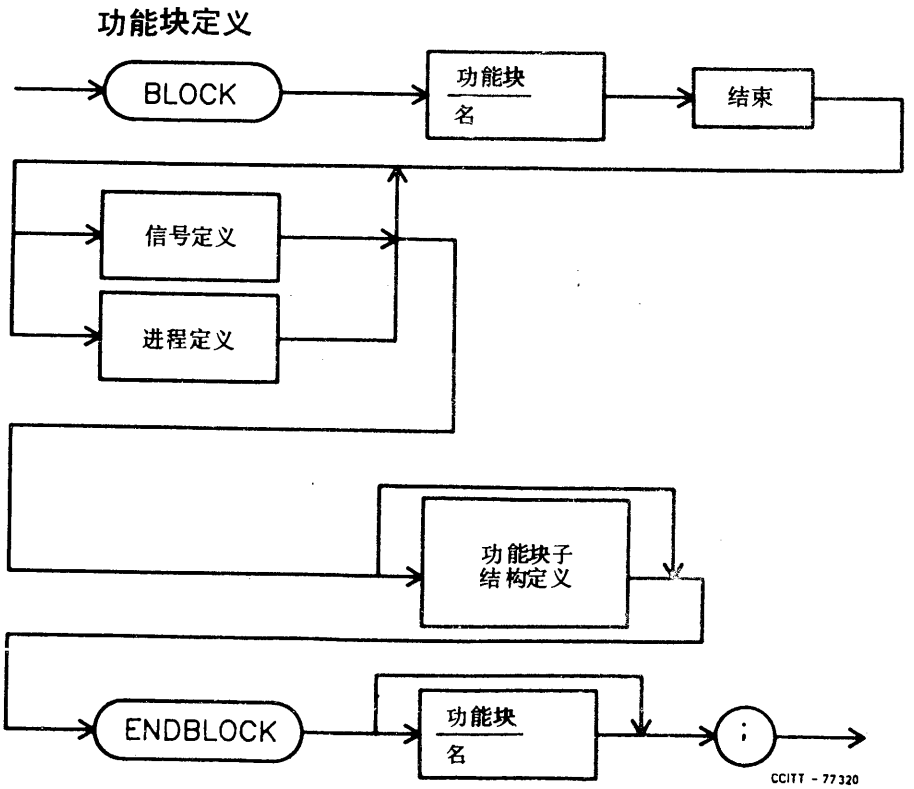


图17/Z．102
功能块的语法图

4.2 功能块子结构定义

功能块子结构定义表示划分一个功能块为子功能块、子信道和新的信道。

4.2.1 语法

功能块子结构定义

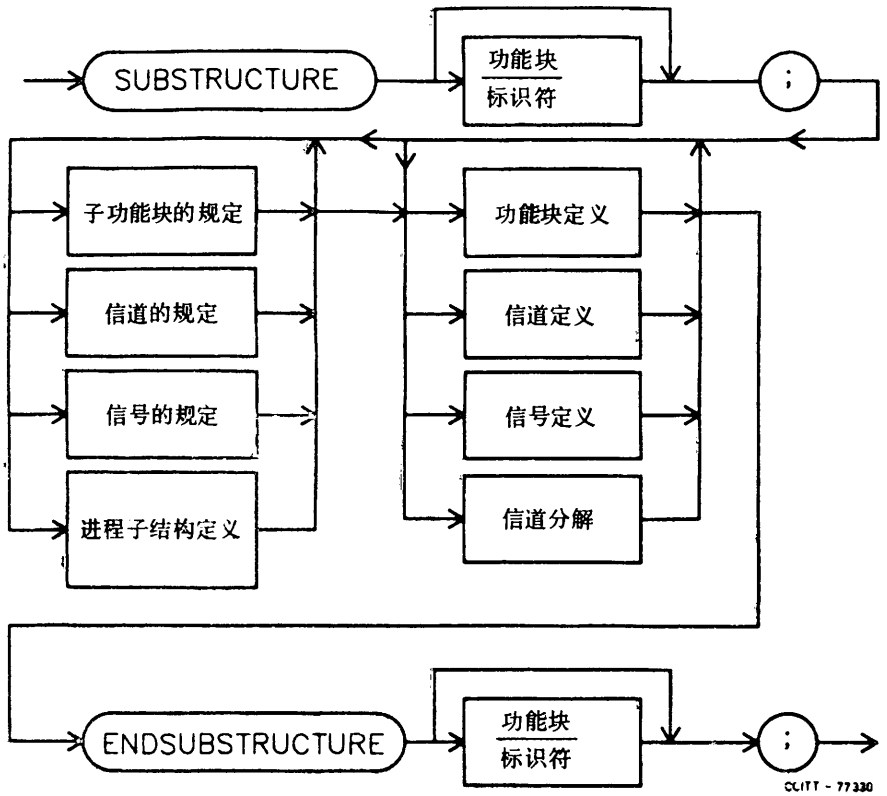
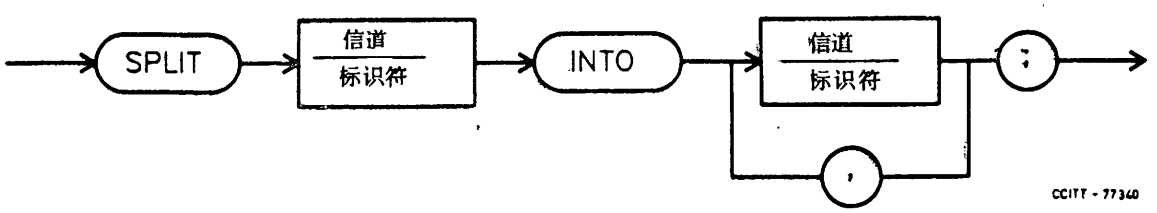
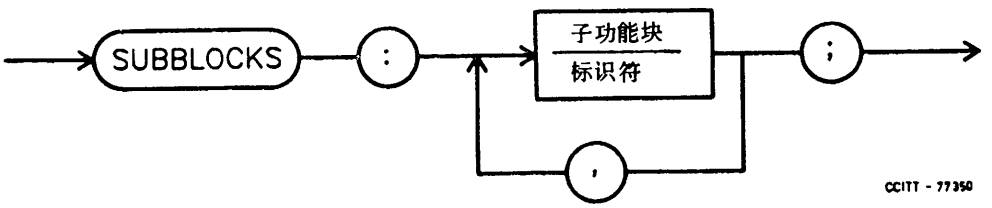


图18/Z.102
功能块子结构图的语法图

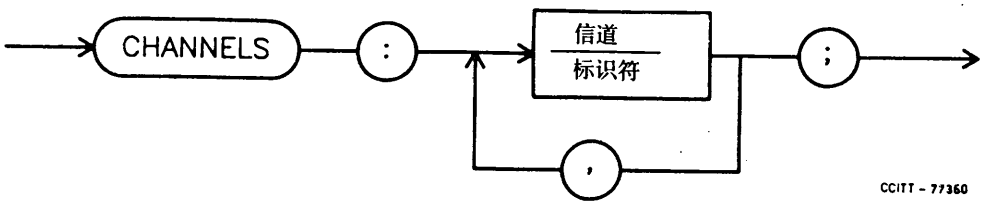
信道划分



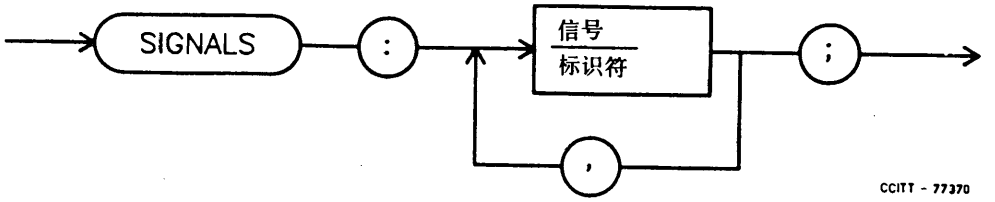
子功能块的规定



信道的规定



信号的规定



例子：

```
BLOCK b:

    ...
    SUBSTRUCTURE b;
        SUBBLOCKS   b1, b2, b3;
        CHANNELS    c1, c2, d1, d2, e;
        SIGNALS     s1, s2, s3;

    SPLIT c INTO c1, c2;
    SPLIT d INTO d1, d2;

    ...
        /* 信道定义*/
        * 功能块定义*
        /*信号定义*

    ENDSUBSTRUCTURE;

ENDBLOCK b;
```

4.2.2 与SDL抽象语法的关系

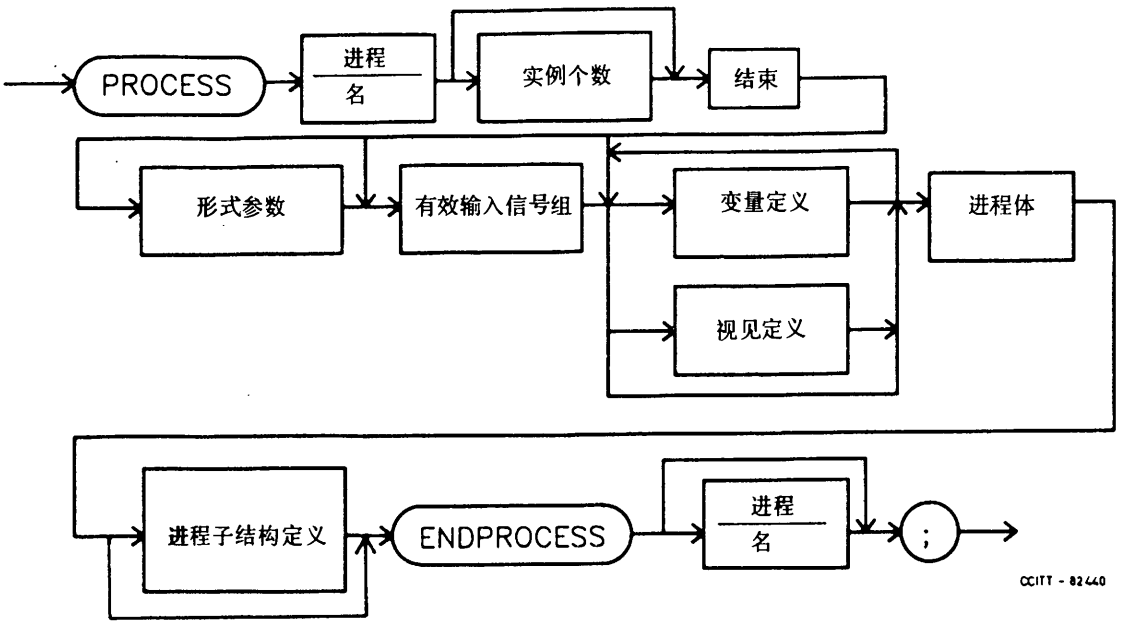
本语法表示功能块子结构定义以及抽象语法中的内部分块定义。功能块、信号及信道的名字参见所包含的功能块定义、信号定义及信道定义。

4.3 进程子结构

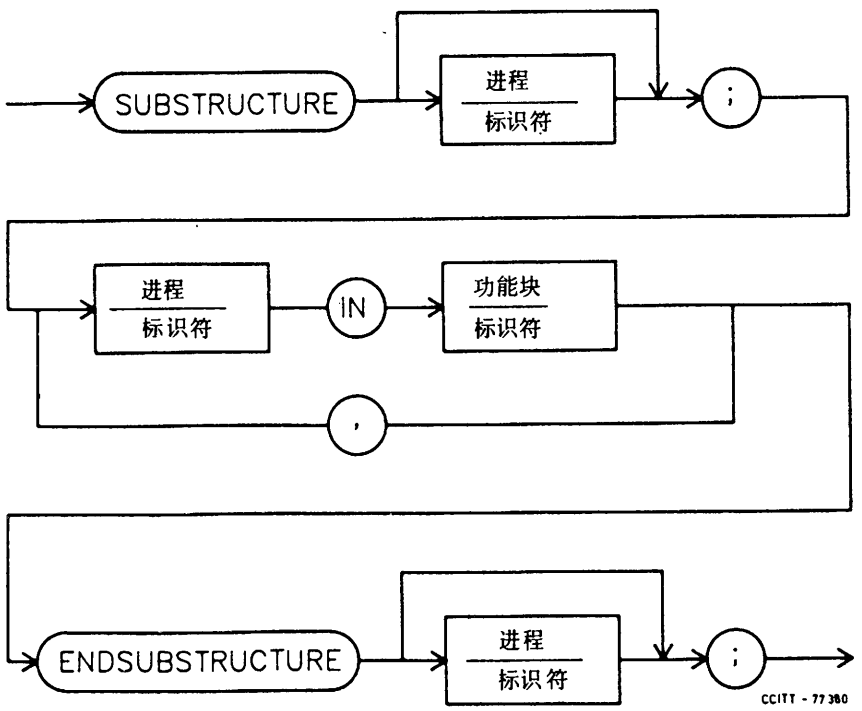
本语法表示把进程划分成为子进程，以及把这些子进程分配到子功能块中。此划分既可以在进程定义中也可以在功能块子结构定义中表示。

4.3.1 语法

进程定义扩展如下：



进程子结构定义如下：



下述例子表明进程子结构作为进程定义的一部分：

例子：

```
...
    SUBSTRUCTURE p1;
        p2 IN b2;
        p3 IN b1;
    ENDSUBSTRUCTURE;
...
ENDPROCESS p1;
```

下一个例子表示进程子结构作为功能块子结构定义的一部分：

例子：

```
SUBSTRUCTURE b;
    SUBSTRUCTURE p;
        p1 IN b2;
        p2 IN b1;
    ENDSUBSTRUCTURE
...
    ENDSUBSTRUCTURE;
ENDBLOCK b;
```

4.3.2 与SDL抽象语法的关系

本语法结构表示在抽象语法中的进程子结构定义。有关的进程名就是起动子句中的名字，所包含的一组进程名就是用关键字“IN”分隔的那些名字，它们中的每一个对应于一个功能块名。

4.4 信道子结构

本语法表示划分一个信道为一组信道和功能块。

4.4.1 语法

关于信道的语法经扩充后包含非终结的“信道子结构”作为任选部分。

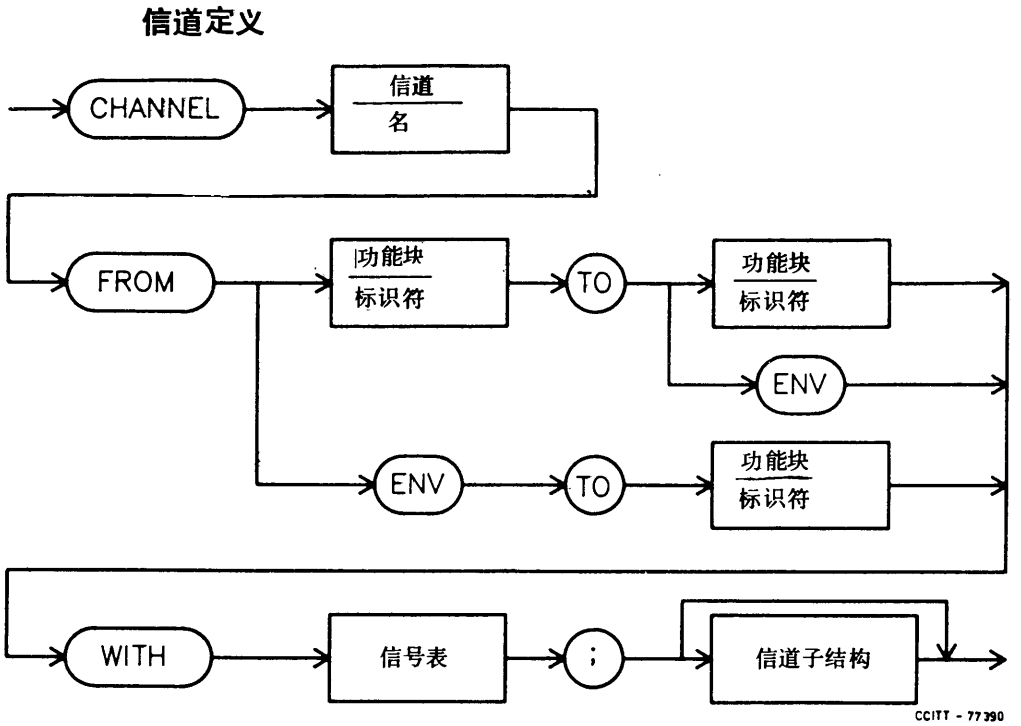


图19/Z.102(4中之1)
信道子结构的语法图

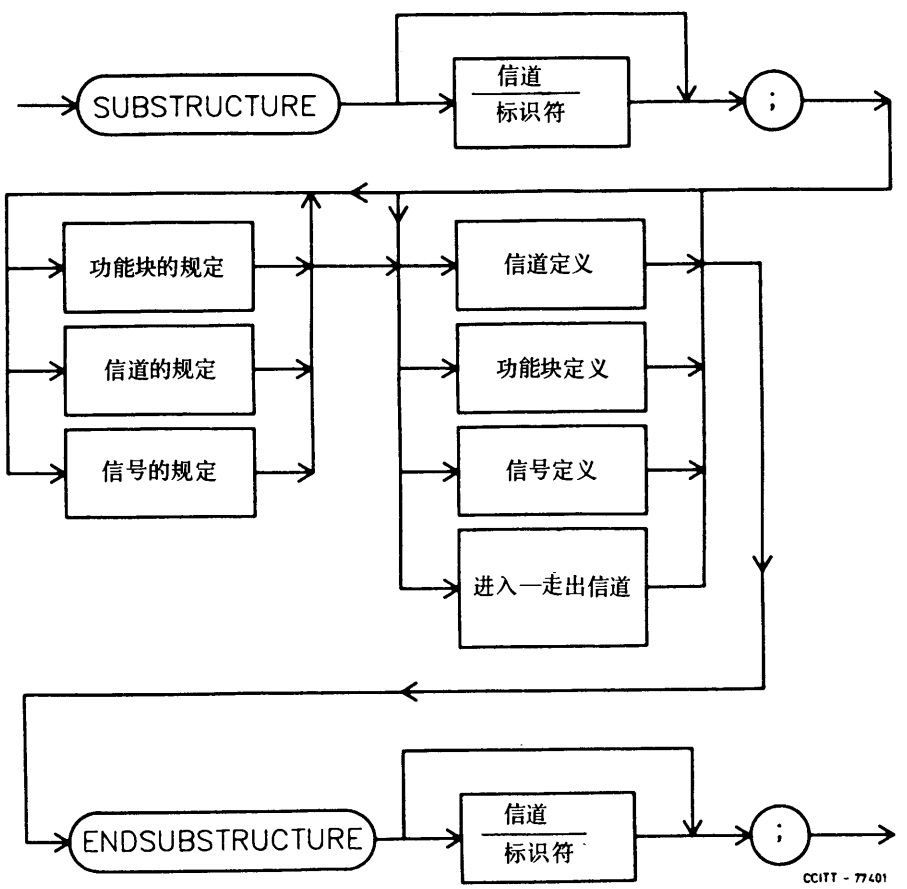


图19/Z.102(4中之2)
信道子结构的语法图

进入—走出信道

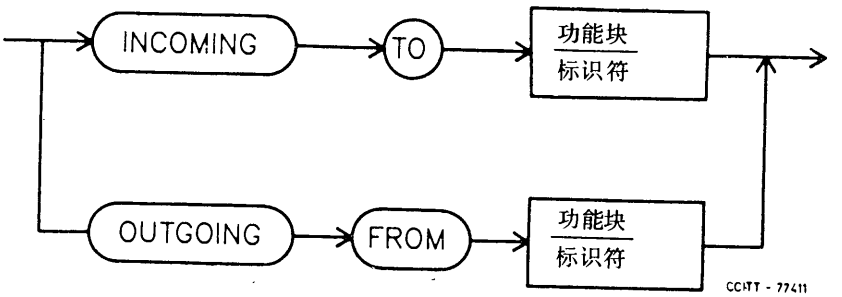


图19/Z.102(4中之3)
信道子结构的语法图

功能块的规定

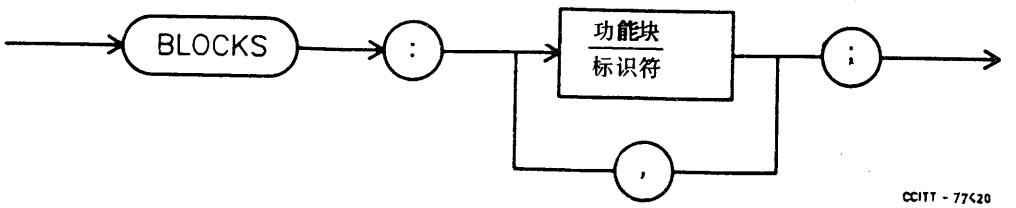


图19/Z.102(4中之4)
信道子结构的语法图

信道子结构是一组引用，引用信道的组成部分以及子结构的附加定义。

例子：

```
CHANNEL c FROM b TO d;
    SUBSTRUCTURE
        INCOMING TO e;
        OUTGOING FROM f;
        BLOCKS: e,f;
        CHANNEL c1 FROM e TO f WITH s1,s2,s3;
        BLOCK e;
                ...
        ENDBLOCK e;
        BLOCK f;
                ...
        ENDBLOCK f;
    ENDSUBSTRUCTURE;
```

4.4.2 与SDL抽象语法的关系

本语法表示抽象语法中的信道子结构定义。本语法中给出的功能块、信道、信号及数据标识符可以是对所包含的定义的引用。

建 议 Z.103

SDL 的 功 能 扩 展

1 引言

本建议定义了SDL中的一些补充概念和简写符号。这些补充的基础是建议Z.101中所定义的基本SDL。这些补充的目的是给SDL用户提供方便的概念和简写符号。

本建议中定义了下述概念和简写符号：

过程

提供了一种方法，用一个元素来表示进程流图的一部分，它可被多次引用。过程的详细行为可在别处定义，在进程流图内部或外部都可以。过程允许把一个进程流图分解为段的分级结构，用来支持使用SDL的结构化设计方法。此概念类似于普通编程语言中出现的过程概念。

值的进口和出口

这是信号的简写符号。当一些进程实例想共享其中某个进程拥有的一个变量的值时，在这些进程实例之间来往工作的信号的简写符号。

允许条件

它是当接收或保存一组信号附有条件时，为避免《状态爆炸》而作的简写符号。

连续信号

它是一个简写符号，当监测到进程外面的一个连续条件时，为表达信号的来往工作而作的简写符号。

宏

这是一种语法上的方法，给用户定义简写符号时采用。宏是复合语法元素的定义，它是以其它已经定义的语法元素来表示的复合语法元素。宏本身没有自己的语义。

任选

这是一种语法上的办法，用来表达一个图或线性正文中的若干个可替换的行为。在解释该图或线性正文之前，必须决定应该选择任选所提供的哪一个行为。

本建议还定义了一个SDL的扩展图形语法，它使用状态符号中的图形元素。

2 过程

过程是一种方法，它把一个名字给予多个项的组合，并用单个引用来代表此组合。过程的规则把纪律施加于对项的组合的选择方法上。过程的规则也限制数据项和信号的名字的作用范围。

过程意在：

a) 允许把一个进程流图构造成若干个具体的层次；

b) 为保持规格的紧凑性，而用单个项来表示复杂的多个项的一个组合，这个组合可看成是独立的；

c) 使经常使用的项的组合可以被预定义和重复使用。

过程用过程定义来定义。过程的调用可借助于引用过程定义的一次过程调用。一个过程调用有其相关的参数，这些参数既用来传递数值，也为过程的执行而控制数据和信号的范围。哪一个信号和数据项要受过程的解释的影响将由参数传递机制来控制。

2.1 通用语言模型

下述定义对建议Z.101中的定义作了严格的补充。

2.1.1 通用模型简介

过程是进程流图的一部分，它可看成是孤立的。过程具有单个入口点和单个出口点。在进程流图或过程流图中，过程调用可出现在一个任务可出现的任何地方。过程可含有多个状态。过程中所用的所有数据项必须在过程定义中予以定义。如果某个数据项被定义为一个形式参数，那么它使用提出调用的环境中的数据项的值，并且（或者）给提出调用的环境中的数据项提供数值，否则，该项是局部于过程本身的，并且必须在过程定义中予以定义。

一个过程只有当进程实例调用它时才加以解释，且是由该进程实例来解释。当一个过程调用得到解释时，必须先解释完过程流图才能继续解释出现该调用的跃迁。这意味着在解释该过程的同时，进行调用的进程实例的输入端口和有效输入信号组得到了使用。在某个过程中引用的所有信号必须命名为过程的形式参数。在进行调用的环境中的有效输入信号名字组中，所有在该过程中没有被引用的信号也必须被命名为该过程的参数，即附加保存组。这就允许过程自动地保存任何其它在输入端口中到达的信号。当这些信号在进程实例正在解释一个过程的时候到达时，如果不这么做，则引入一次从过程调用到包含一个状态的过程的跃迁时，就会隐藏丢失信号的副作用（由于隐含的《空跃迁》）。

过程定义由过程流图给出，并可出现在一个数据类型定义可出现的任何地方。

过程流图遵循与进程流图相同的规则，但有下述例外：

— 起动节点由过程起动节点代替；

— 停止节点由返回节点代替；

— 在解释过程时，只有在该过程中声明了的数据项、形式参数和信号名才对进程实例可见，亦即，一个过程不能访问在该过程外的任何数据项或信号，除非该数据项或信号被声明为一个形式参数（信号必须属于调用该过程的环境，并且在该过程中必须声明所有的信号是输入、保存还是输出）；

— 相同的信号名实在参数不能对应于一个以上的信号名形式参数；

— 在抽象语法中，要保存的信号组作为过程调用的参数传递。然而，在具体语法中，在过程中包含的所有状态节点具有一个隐含的保存信号组，它包含了在该过程中没被声明为形式参数的所有输入信号。

2.1.2 抽象语法

对建议Z.101中所定义的抽象语法，作了如下补充：

系统定义

系统定义也可包含一个或多个过程定义。

功能块定义

功能块定义也可包含一个或多个过程定义。

进程定义

进程定义也可包含一个或多个过程定义。

进程流图

进程流图也可包含一个过程调用节点。

跃迁串也可以是一个过程调用节点后面跟以一个跃迁串。

过程调用节点包含一个过程标识符和一个实在参数表。

过程定义

过程定义有一个过程名，并包含一个形式参数表、一个附加保存组及一个过程流图，并可包含多个过程定义和数据定义。

形式参数表可以是空的。表中的每个形式参数必须赋给一个属性IN、IN/OUT或SIGNAL。

赋有属性IN或IN/OUT的形式参数的名字不得用于过程定义的变量定义中；赋有属性SIGNAL的形式参数名字组必须包含过程流图中涉及的所有信号名。

过程流图

过程流图是一个其节点用有向弧连接的流图。进入节点的弧叫做进入弧，离开节点的弧叫做走出弧。以某条弧作为进入弧的节点跟在以这条弧作为走出弧的节点后面。

过程流图中允许有下述类型的节点：

状态节点

输入节点

任务节点

输出节点

判定节点

过程调用节点

过程起动节点

返回节点

创建请求节点

流图的每个节点有一个名字和一个类型。过程流图内的各个状态节点有不同的名字。

下面的规则规定了过程流图的连接规则：

— 每个过程流图包含一且仅一个过程起动节点。过程起动节点后跟一个跃迁串。起动节点不跟在任何其它节点的后面。

— 一个跃迁串可以是下列之一：

a) 空操作后面跟以一个状态节点或一个返回节点，

b) 一个动作串后面跟以一个跃迁串，

c) 一个判定节点。

— 一个动作串可以是下列之一：

a) 一个任务节点，

b) 一个输出节点，

c) 一个创建请求节点。

— 一个判定节点后跟两个或多个判定弧。

— 一条判定弧是一条已命名的弧后跟一个跃迁串。

— 一个状态节点后跟一个或多个输入节点。

— 一个输入节点后跟一个跃迁串。

— 返回节点后面不跟任何节点。

— 每个流图最多有一个返回节点。

— 从过程起动节点出发，每一个节点都是可达的。

一个调用节点含有一个过程标识符、一个附加保存组和一个实在参数表。在此表中，对所涉及的过程定义中的每个形式参数，都必须恰有一个对应的实在参数。每个实在参数都必须与相应的形式参数的类型相匹配。在实在参数表中，任何的信号实在参数名最多只能出现一次。

与具有属性IN/OUT的形式参数相应的每个实在参数都必须是一个变量名。

附加保存组是一组信号标识符，它可能是一个空集。

2.1.3 解释

对建议Z.101的解释规则作了如下补充：

进程

一个调用节点可解释为对过程定义的过程起动节点的解释，此过程定义由该节点的名字指出，然后解释跟在此调用节点后的节点。

附加保存组由所有信号标识符的值给出，这些信号标识符在进行调用的进程的有效输入信号组中，它们不作为该节点的实在参数出现。

过程

调用节点引用的过程定义包含有过程流图，当一个进程解释一个调用节点时，就解释了该过程流图。解释过程流图的节点与解释进程流图的等效节点的方法是相同的，但有下述例外及补充：

— 每个赋有属性IN的形式参数都表示一个类型已定的变量。这个变量属于过程本身，在解释过程起动节点时创建，在解释返回节点时消失。

— 每个赋有属性IN/OUT的形式参数都作为实在参数给出的变量的一个同义名字。当访问该变量的值时，或当给此变量赋一个新的值时，在解释过程流图的全过程中都使用这一同义名字。

过程起动节点

对过程起动节点的解释包括：

— 对每个赋有属性IN、具有形式参数的名字和数据类型的形式参数，要建立一个局部变量。把实在参数的值赋给此变量，此实在参数可以是未定义的。

— 在赋有属性IN/OUT的形式参数中给出的每个变量名作为对此变量名的一个同义名字使用，而该变量名作为实在参数给出。由同义名字表示的变量只在过程起动节点处计算一次，在该过程中每次使用形式参数时都不计算。

— 在赋有属性SIGNAL的形式参数中给出的每个信号名作为对信号标识符的一个同义名字使用，而此信号标识符作为实在参数给出。

— 解释跟在过程起动节点后的节点。

状态节点

状态节点的解释与进程流图中的方法相同，但有例外：送到输入端口的保存信号组是调用当前过程的进程或过程的附加保存组与保存信号组之并集。

2.2 SDL/GR

把过程概念补充到SDL中而使得在进程图中增加一个新的符号：调用符号。

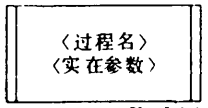
过程定义处于由过程图表示的图形语法中。除了过程起动符号和返回符号以外，这个图与进程图是类似的。

下面只给出对建议Z.101中所定义的图形语法的补充，这些补充是引入过程所需要的。

2.2.1 进程图

2.2.1.1 符号

下述符号用来表示过程调用：



可出现的任务符号进程图中可以出现此符号。

过程名和实在参数用SDL/PR语法给出

图 1/Z.103

过程调用符号

2.2.1.2 与SDL抽象语法的关系

用过程调用符号来表示进程流图中的过程调用节点。

附在过程调用节点上的实在参数用在此符号中给出的实在参数来表示。任何不作为实在参数的有效输入信号是过程调用节点的附加保存组的一员，这表示应该在过程执行的全过程都保存该信号。

返回节点

返回节点的解释包括：

— 由解释过程起动节点而建立的所有变量将消失；

— 由解释过程起动节点而建立的所有同义名字将消失；

— 对返回节点的解释完成了对过程起动节点的解释。

2.2.2 过程图

过程图类似于进程图，仅有的差别是过程起动符号代替了起动符号，返回符号代替了停止符号。

下面只定义补充的语法。

2.2.2.1 符号

在下图2/Z.103中，定义了两个补充的符号：

过程起动符号
过程名和形式参数
用SDL/PR语法给出。

返回符号

CCITT-73900

图 2/Z.103

过程图的补充符号

2.2.2.2 与SDL抽象语法的关系

SDL抽象语法中的过程定义用过程图来表示。

与进程图共用的符号，它们同抽象语法的关系与在进程图中出现时相同。补充的过程起动符号和返回符号分别表示过程起动节点和返回节点。

附于过程起动符号上的参数表示附于过程起动节点上的形式参数。

2.2.2.3 图形约定

一般说来，对进程图所作的图形约定这里同样适用。对过程起动符号和返回符号的约定分别与对进程起动符号和对停止符号的约定相同。

不论在何处出现一个返回符号，它都表示返回节点（当返回符号多次出现时）。

2.3 SDL/PR

对过程定义的SDL/PR语法在下面给出。在此语法中，凡是一个数据类型定义可以出现的地方就可出现过程定义。此语法与进程定义的语法相似。

下面只给出对建议Z.101中已定义的语法的补充。

2.3.1 系统

2.3.1.1 语法

系统的语法中加入了过程定义的语法：

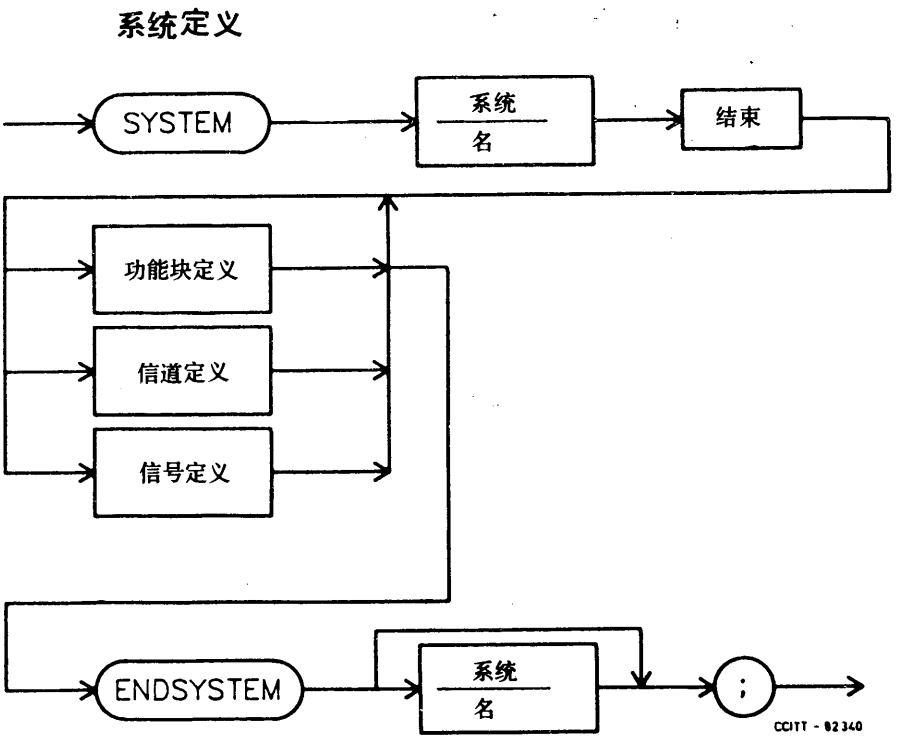


图 3/Z.103
系统的语法图

2.3.2 功能块

2.3.2.1 语法

功能块的语法中加入了过程定义的语法：

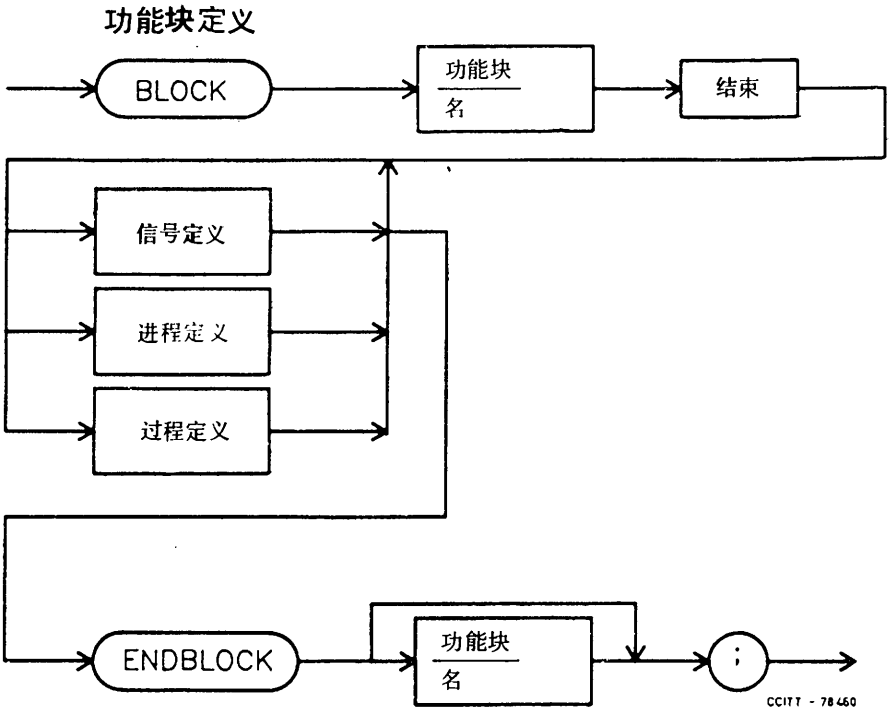


图 4/Z.103

功能块的语法图

2.3.3 进程

2.3.3.1 语法

进程的语法中加入了过程定义的语法，并且跃迁串的语法中加入了过程调用的语法：

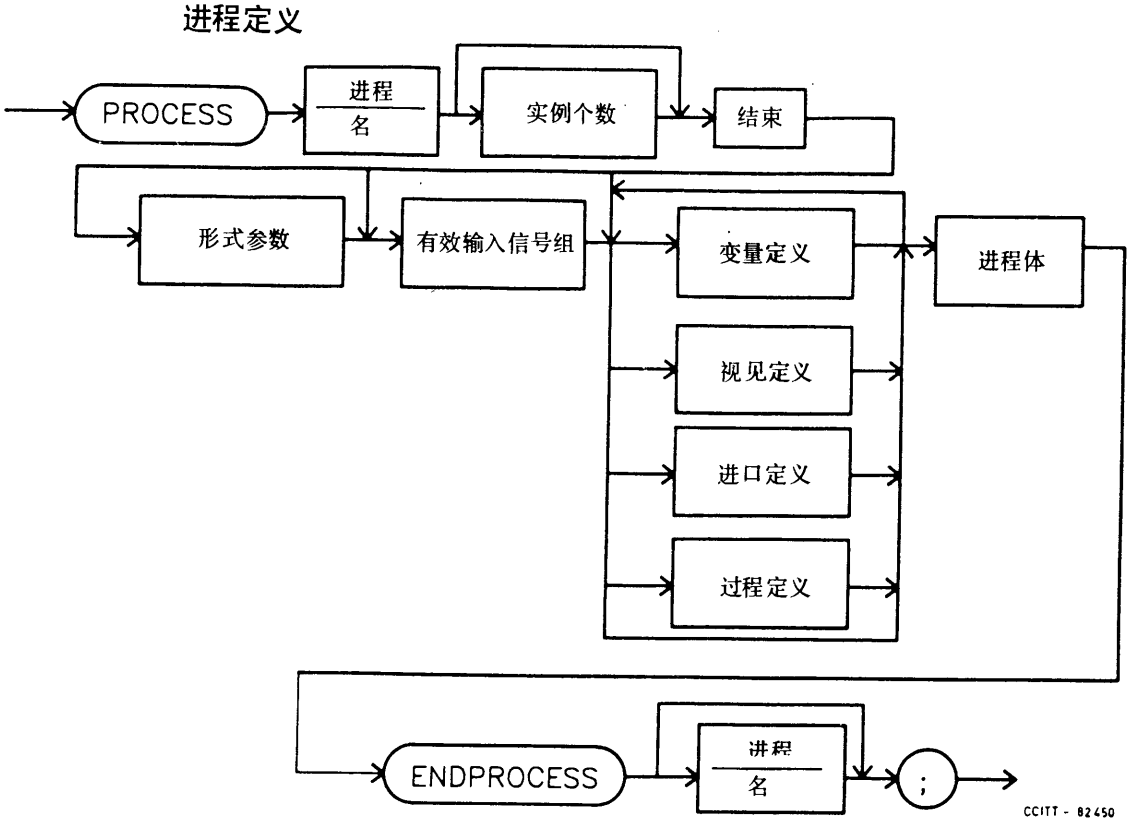


图 5/Z.103

进程的语法图

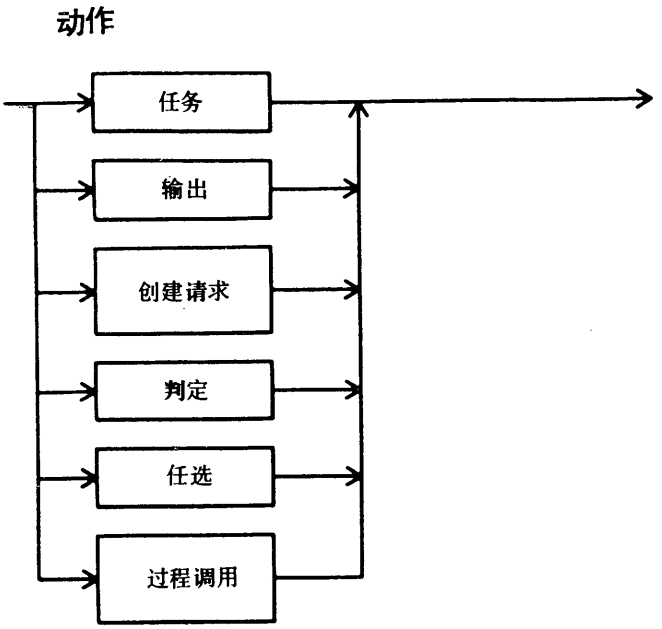


图 6/Z .103(2中之1)

动作的语法图

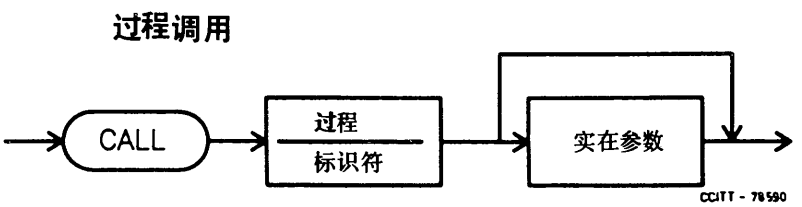


图 6/Z .103(2中之2)

过程调用的语法图

2.3.3.2　与SDL抽象语法的关系

一个跃迁内的过程调用表示抽象语法中的调用节点，而其实在参数表示附在调用节点上的实在参数。给定的每个实在参数必须与所涉及的过程定义中的相应形式参数的数据类型相符。

2.3.4　过程

2.3.4.1　语法

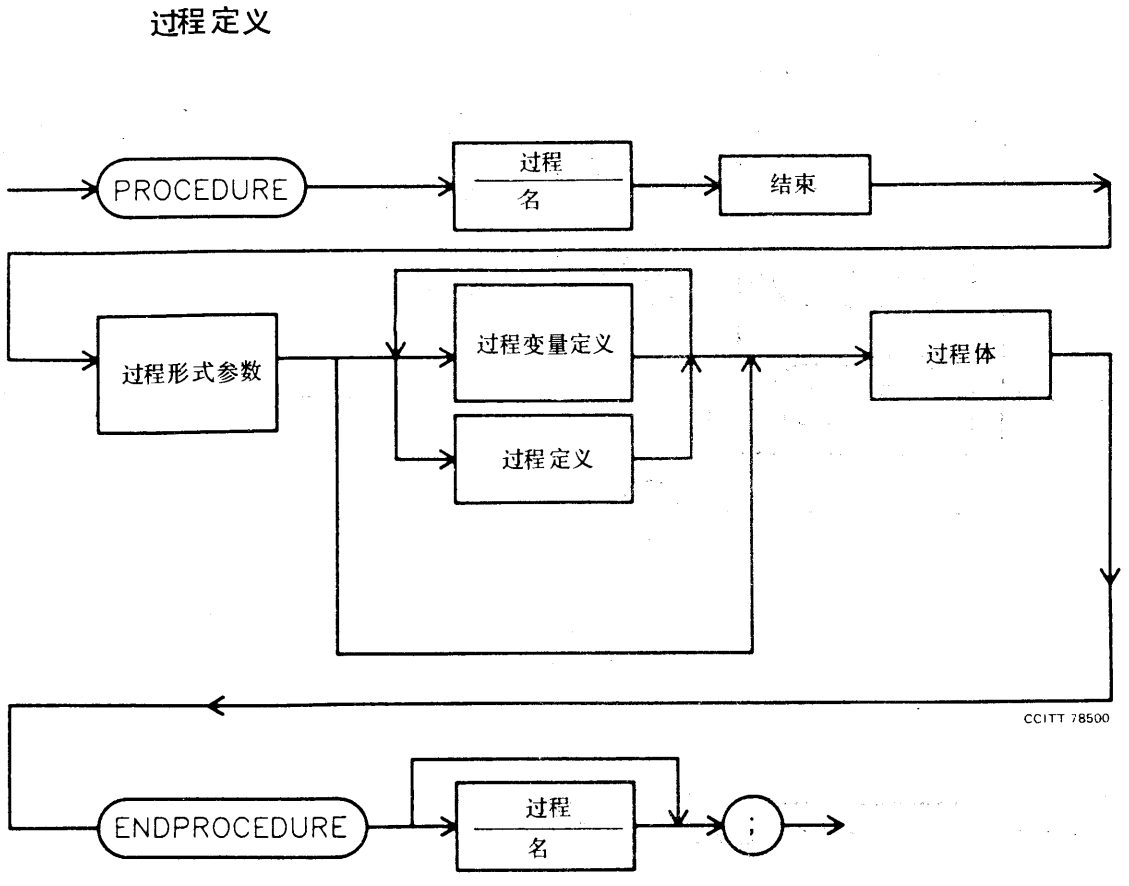


图 7/Z.103(5中之1)

过程的语法图

过程形式参数

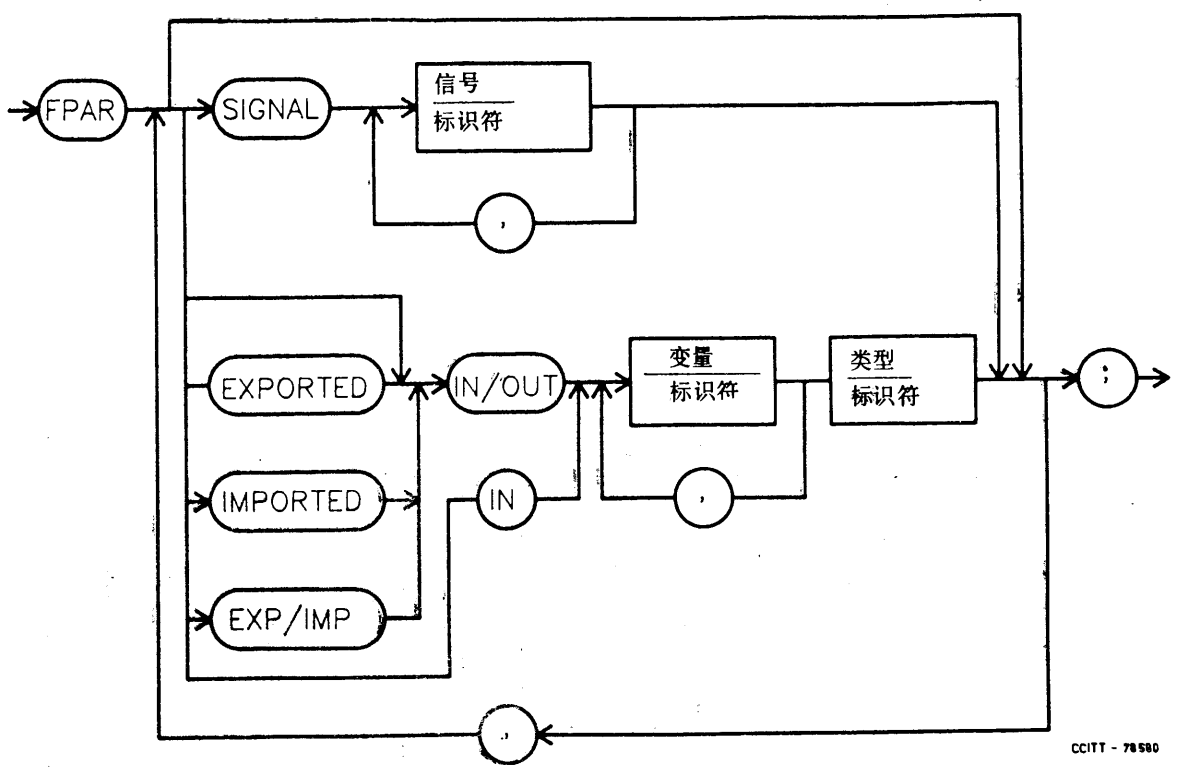


图 7/Z.103(5中之2)
过程的语法图

过程变量定义

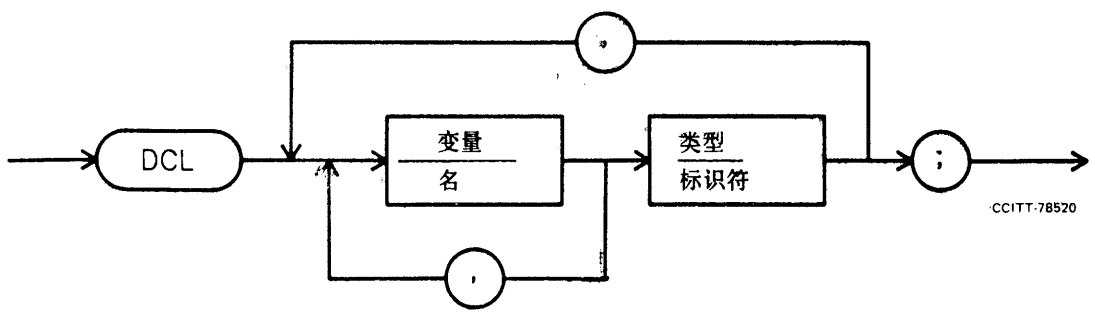


图 7/Z.103(5中之3)
过程的语法图

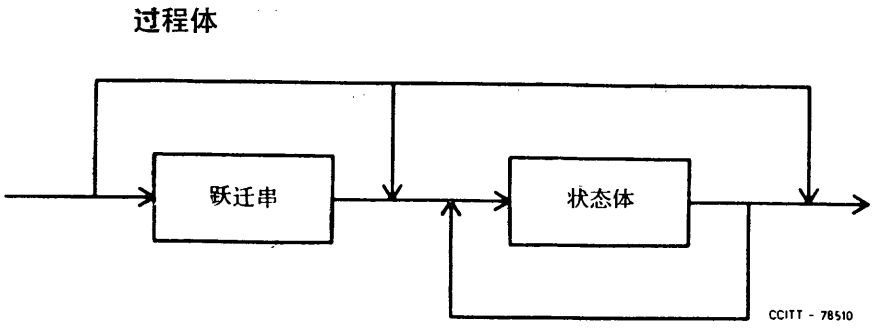


图 7/Z.103(5中之4)

过程的语法图

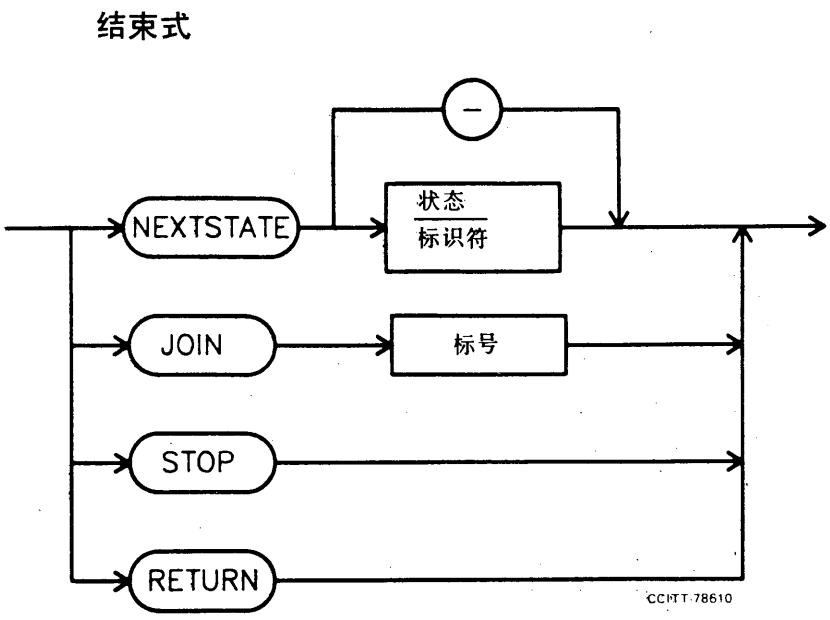


图 7/Z.103(5中之5)

过程的语法图

2.3.4.2 与SDL抽象语法的关系

过程语法图表示了抽象语法的过程定义。在此语法图中的形式参数表示了附在过程起动节点上的形式参数。

在过程内允许的语法结构与抽象语法的关系，就象此语法结构在进程内被允许时，它与抽象语法的关系那样。进程起动节点被隐含，后面跟该过程的第一个语法结构，并且，返回语句表示返回节点

3 SDL进程间通信的组合操作

组合操作是为SDL概念的组合而作的标准简写符号。组合操作用已有的SDL概念来定义，但并不加到

该语言的语义中。因此，在解释抽象语法时，它们并不改变抽象语法。引入它们是为了给SDL用户提供方便。

这里定义的组合操作提供了一些可互换的方法来进行SDL进程实例间的通信。尽管对用户来说它们与通常的SDL的信号交换机制明显地不同，但实际上它们是以通常的SDL语义为基础的。

组合操作提供了：

a) 即使进程处于不同的功能块中，也能共享数据项的值；

b) 有暂时允许或禁止接收特定信号的能力，而毋须用分立的状态精确地进行表示；

c) 连续信号，通过改变该信号的值可以产生跃迁。

对每个组合操作规定其自己的语法，并根据“通常的”SDL概念来定义这些组合操作。因此，组合操作替用户包含了暗指的即隐藏的状态、信号和过程。在§§3.1至3.3中，用通常的SDL符号的直观名字来定义组合操作。当在一个SDL表示中使用了任何组合操作时，所隐含的SDL符号具有唯一的隐含的名字。这些隐含的名字可以这样来选择，以使在SDL描述中不会产生组合操作的不同情况之间的冲突或与其它名字之间的冲突。这样，SDL用户可以自由地采用这些名字用于其它的目的，只要他愿意这样做。

由于组合操作用其它SDL术语来定义，故在某种情况下他们会产生副作用，这些将在SDL用户指南中讨论。

3.1 进口值和出口值

在SDL中，变量总是由一个进程实例所拥有，并且局部于该进程实例。虽然变量可以声明为一个共享值(见建议Z.101)，从而允许同一功能块中的其它进程实例可以视见这个变量的值，但是通常变量只对拥有它的那个进程实例是可见的。如果其它功能块中的一个进程实例要视见一个变量，则需要与拥有此变量的进程实例进行信号交换。

下面描述这样做的标准方法和简写符号。此技术称为进口值，因为通信是借助于复制该变量的值进行的，只有拥有此变量的进程实例才能访问变量本身。此技术也可用于把数值出口到同一功能块中的其它进程实例中，在这种情况下，它提供了使用共享值的另一种方法。当很可能要分解一个功能块使得所考虑的这些进程实例处于不同的子功能块中时，或者当在允许条件(见§3.3)中用到这些数值时，这种方法都是需要的。

有些变量的值可为其它进程实例使用，拥有这种变量的进程实例称为该变量的出口者。使用此变量的其它进程实例就是该变量的进口者。

通过信号的交换可以访问此变量的值。进口者给出口者发送一个信号，并等待答复。为了回答这个信号，当做完了最近的出口操作后，出口者就给进口者送回一个带有此变量的值的信号。

其值被进口和出口的变量在它们的变量定义中应按照情况规定进口或出口。这些变量也可用信道中的定义来标识，这些信道传送隐含的信号交换。

作为出口的变量的定义产生一个复制该变量的隐含的定义，该变量将在进口和出口操作中使用。

进程实例通过下列语句透露已定义为出口的变量的值：

EXPORT(x) 其中x是该变量的名字。

EXPORT(x)导致把X的当前值存入隐含的副本，并把信号送到等待允许条件(§3.2)或等待连续信号(§3.3)的进程实例中。

EXPORT操作可以与变量的计算一起进行或者独立地进行，例如：

EXPORT(x):=〈表达式〉;

或

EXPORT(x);

此结构只可出现在任务中。

来自另一个实例的访问只能通过下面的语法结构进行：

IMPORT(x, Pid) 其中x是变量名，Pid是对拥有此变量的进程实例的引用。

此结构可以出现在任务中和判定中。

3.1.1 定义

3.1.1.1 进口者的进口操作

在进口者进程流图中，进口操作由下面一系列节点组成：

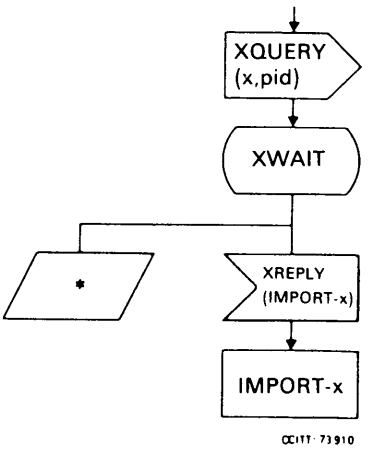
— 名字为XQUERY的一个输出节点，它引用一个变量名，此变量的值需要受到访问，并且它指出拥有该变量的进程实例（出口者）；

— 一个状态节点，它带有保存信号组，包含除了XREPLY以外的所有信号；

— 为接受信号XREPLY的输入节点，此信号返回所查询的变量的值；

— 把与信号XREPLY有关的值赋给与出口变量同类型的一个隐含变量，这个隐含的变量名代替了进口操作。每次出现构造操作时要使用一个新的隐含变量。

定义进口操作的一系列操作在下面用图形SDL语法来说明。IMPORT(x, pid)语句与下面的跃迁串相对应。



注 — 此图是说明性的。进口操作的使用者可只写：IMPORT (x, pid)。

图 8/Z.103
用SDL/GR说明进口操作

3.1.1.2 出口者的进口操作

由于进口操作隐含了由出口者完成的动作，因此在出口者的每个状态中（包括所有的隐含状态）加入了下列隐含的跃迁：

— 一个输入节点用以输入信号XQUERY；

— 一个名字为XREPLY的输出节点，把被查询的变量之值的隐含副本返回到进行查询的进程实例；

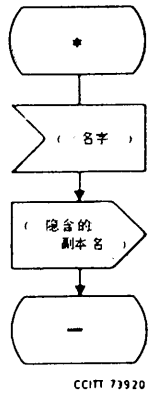
— 回到跃迁开始的那个状态。

（应该注意：如果在允许条件和（或）连续信号中使用出口值，出口者也可随其它跃迁而改变，参见§§3.2和§§3.3。）

图9/Z.103中示出了使用SDL图形语法的隐含跃迁。

3.1.1.3 出口操作

出口操作是一种手段，出口变量的拥有者通过它可以把该变量的当前值透露给进口者。作为出口操作的结果，变量的当前值就赋给了该变量的隐含副本。在没有允许条件或连续信号时，操作本身可模拟为一个任务节点，它把值赋给此变量。然而，出口操作与允许条件的模型及连续信号的模型互相有影响，参见§§3.2和§§3.3。



注—此图只是说明性的。因为跃迁是隐含的，用户什么也不用写。

图 9/Z.103
用SDL/GR说明出口者的进口操作

3.1.2 使用进口值的规则

— 所有进口值必须在进口者中定义为IMPORTED，而在出口者中定义为EXPORTED。变量定义中的出口属性使得此变量标识符在整个系统内都是可见的。
— 为了表示进口操作的源地和目的地，可以把进口值的名字放入一个信号表，该表附于一个信道上或附于一个功能块中的一个信号路径上。源地与目的地可以位于不同的功能块中，也可以位于同一个功能块中。
— 进口一个值的进程实例也可以出口数值，但绝不能进口它自己的变量值。
— 在任务、判定或允许条件中，一个进口值〈名字〉使用下面的语句来进口：
IMPORT (〈名字〉,〈pid〉)
〈名字〉是进口项的名字，而〈pid〉是拥有此项的进程实例的进程实例标识符。进程中关于〈名字〉的所有其它引用可解释为对〈名字〉的值的本地副本的引用。
— 通过在一任务中所包含的下列语句来透露出口变量的值：
EXPORT (〈名字〉) := 〈表达式〉
或
EXPORT (〈名字〉)

3.1.3 具体语法

在进程定义中有三处涉及进口值。在SDL/PR中和在SDL/GR中，关于这些值的语法是相同的。

3.1.3.1 在变量定义和进口定义中，关键字IMPORTED和EXPORTED指出了所声明的变量的用途。

变量定义

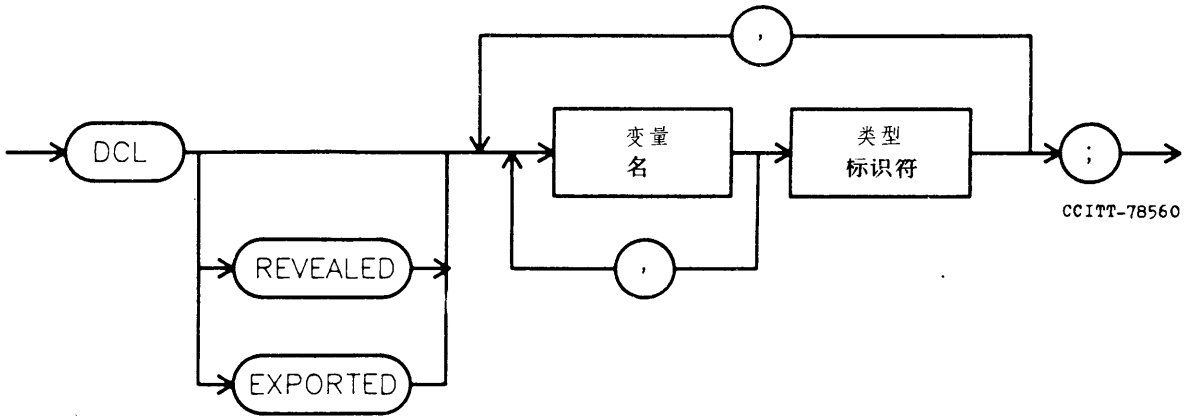


图 10/Z．103(2中之1)

变量定义和进口定义的语法图

进口定义

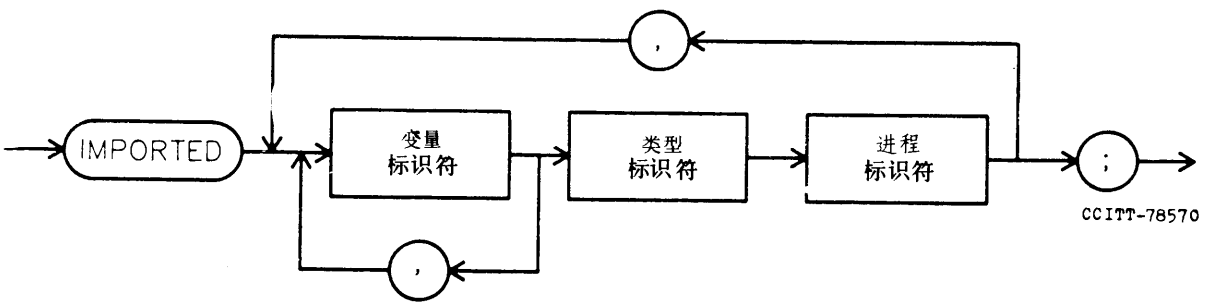


图 10/Z．103(2中之2)

变量定义和进口定义的语法图

3.1.3.2 在SDL/GR的信号表中，为了把进口值和信号区别开来，采用圆括号把进口值的名字括起来。

[信号名1
  (变量名)
  信号名2]

SDL/PR中的语法为：

信道定义

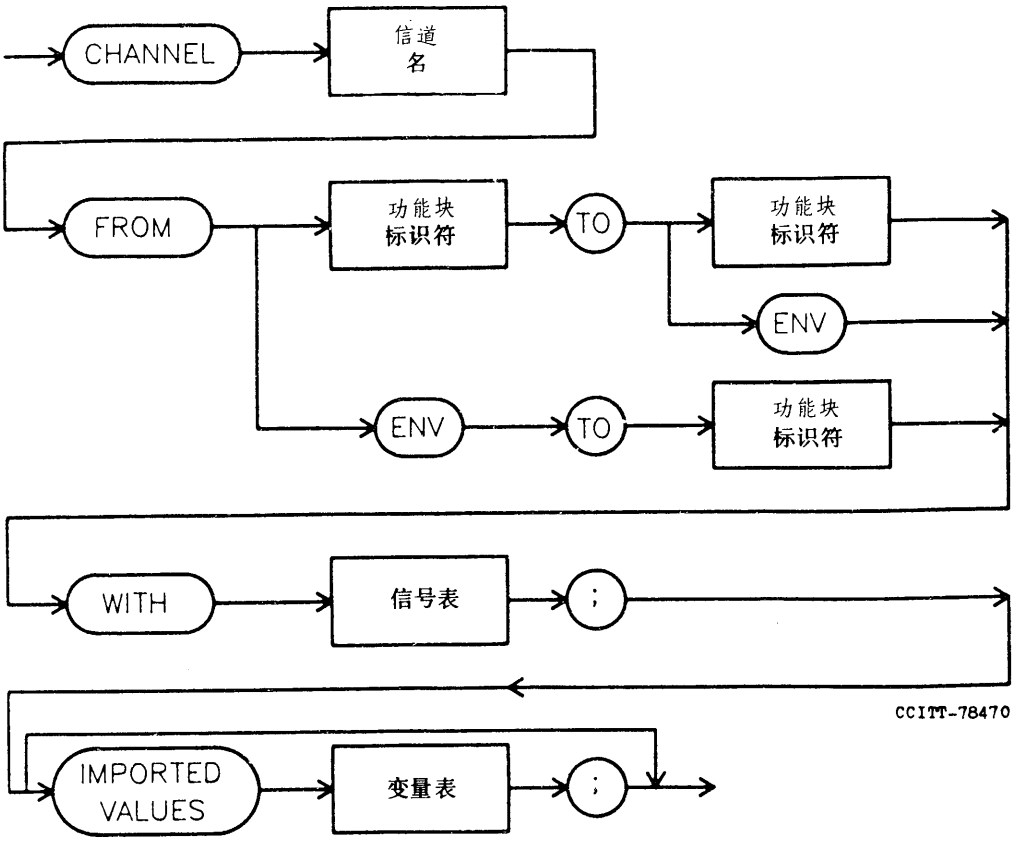


图 11/Z．103(2中之1)
信道定义和变量表的语法图

变量表

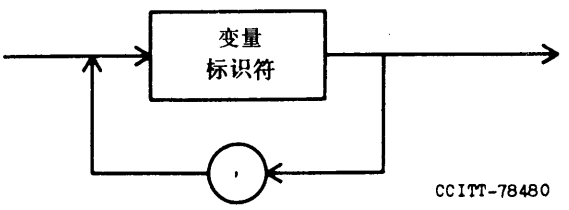


图 11/Z．103(2中之2)
信道定义和变量表的语法图

3.1.3.3　语句EXPORT(〈名字〉)可以在任务名中使用，类似地，语句IMPORT(〈名字〉，〈pid〉)可以在任务、判定和允许条件的名字中使用。

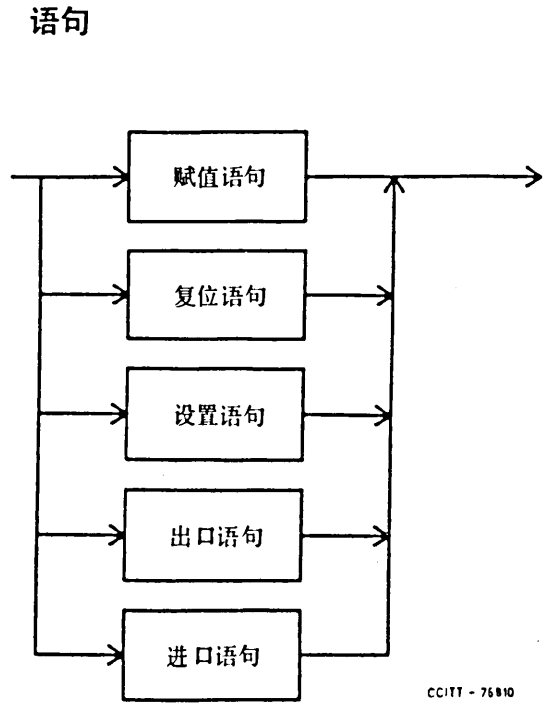


图 12/Z．103 （3中之1）
出口语句的语法图

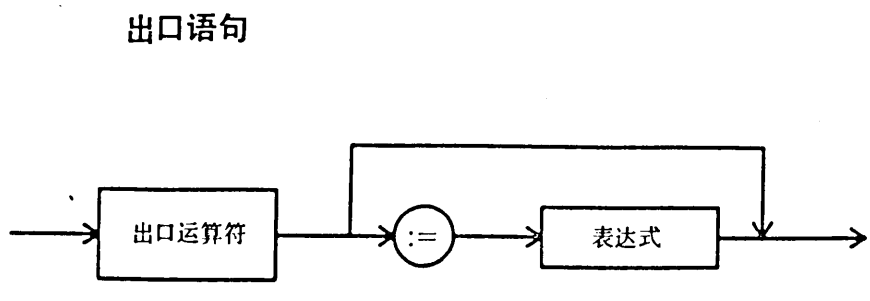


图 12/Z．103 （3中之2）
出口语句的语法图

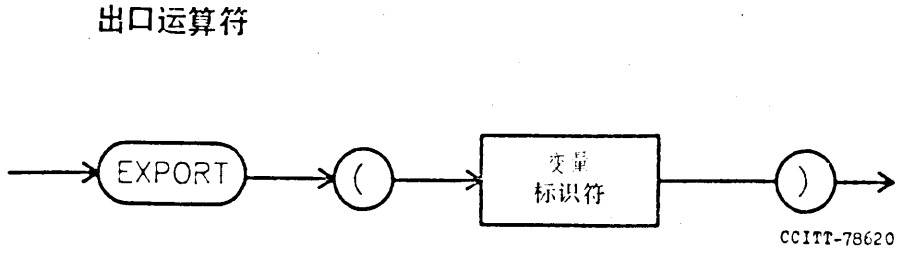


图 12/Z．103 （3中之3）
出口语句的语法图

进口运算符

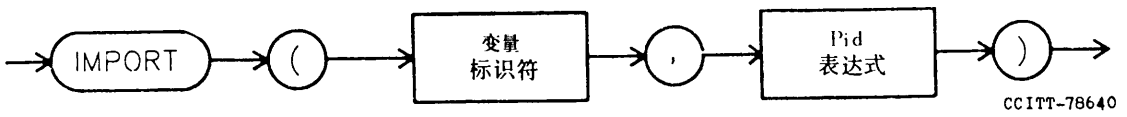


图 13/Z.103
进口语句的语法图

3.2 允许条件

为了减少进程流图中的状态个数，常可采用给跃迁的起始附加若干个条件的办法，这些条件是对变量的要求，称为允许条件。如果此条件为真，则保留有关的输入信号，并且跃迁正常往下进行。如果允许条件不真，那么跃迁就被禁止，而输入信号就被保存。

在普通的SDL中，这种跃迁的有条件执行需要对允许条件的每个值都建立一个单独的状态。允许条件所提供的简明的符号对简化进程流图是很有用的。

允许条件可以用局部值和（或）进口值来表达，绝对不能使用共享值。

允许条件用基本的SDL概念（见建议Z.101）来描述。允许条件用于进程流图中，附在一个输入符号或一个输入语句上。它是包含数据值的一个断言，此数据值可以是局部的或是进口的值。此条件必须是一个布尔表达式，故返回的是真(TRUE)或假(FALSE)值。如果一个状态后面跟着受允许条件控制的跃迁，那么这个状态实际上表示了一组状态，每个状态对应于此允许条件的一个可能的组合。先计算这些条件，然后选择这组状态中的一个合适成员作为下一个状态。

允许条件的解释要根据此条件中所用的数据值。如果一个允许条件只含有局部于进程的数值，则它只需计算一次。如果它包含多个进口值(见§ 3.1)，则每当有一个进口值改变时都必须重新计算此条件。引起这一计算的是：这些值的进口者与出口者之间隐含的信号交换。

用到进口值的允许条件使用更多的隐藏信号，除了§ 3.1中定义的XQUERY之外还使用了XATTACH和ZDETACH。出口者有一张表列出需要知道出口值发生变化的进程实例。XATTACH和ZDETACH请求出口者在列有进程实例的列表中加上或移去进口者。出口者给它出口的每个值维持这样一个隐含表XLIST，每当出口者对相应的数据项进行一次出口操作时，它就把另一个信号XWAKE送给该列表的每个成员。

3.2.1 定义

3.2.1.1 基于局部值的单个允许条件

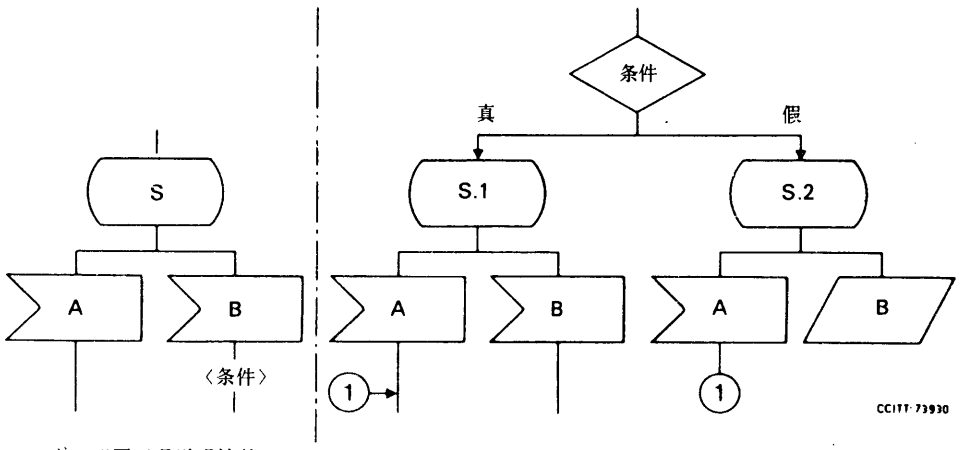
当跟在一个状态后的多个输入中只有一个输入附有允许条件，并且该条件只包含局部值时，它的解释如下：

此状态及后续的输入和保存可替换为：

— 一个判定节点，它计算允许条件；
— 判定为“真”的分支后面接一个状态节点，它上面带有与原来状态相同的保存信号组，并且后面接与原来状态相同的一组输入（附有允许条件的输入作为输入节点出现）；
— 判定为“假”的分支后面接一个状态节点，它使原状态的保存信号组扩充了附有允许条件的输入的信号名；
— 每个输入节点后面接与原流图中相同的跃迁。

在下图中，给出了使用SDL图形语法定义的只基于局部值的简单允许条件的定义：

注一　用户画这个图：



注—此图只是说明性的。

图 14/Z .103

用SDL/GR解释只用局部值的单个允许条件

3.2.1.2　包含进口值的单个允许条件

当跟在一个状态后的多个输入中只有一个输入附有允许条件时，且此条件至少包含一个进口值时，那么此解释就会产生与该进口值的拥有者交换的一个隐含的信号。

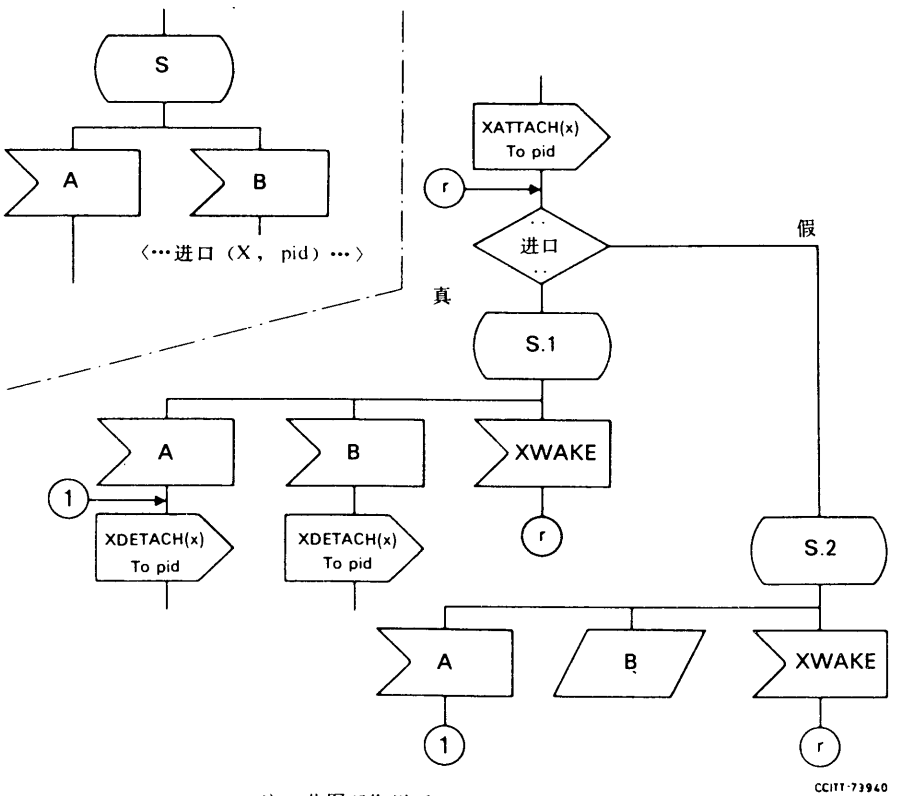
由于持有进口值的变量由其它并发工作的进程实例处理，所以当进程实例在某个状态中等待的同时，允许条件的值可能变动。对于任何这种变动应该重新计算此条件。进入允许条件结构时，就要告诉进口值的拥有者，如果此值发生任何变化就应通知这个实例。因此，当一个进口值变化时，还要给所有需要通知的进程发送一个信号，以使它们再次计算该条件。

定义如下：

含有允许条件的进程的有效输入信号组经隐含的信号 XWAKE 得到了扩展。该状态以及接在它后面的一组输入和保存可用下面的代替：

— 信号XATTACH的一系列输出节点（每一个对应于允许条件中使用的一个进口值）；
— 附加序列后面接一个判定节点和两个状态节点，如在§ 3.2.1.1中那样；
— 因为判定中包含了进口值，所以它自己得到扩充，如在§ 3.2.1.2中那样，这样它前面要安排进口操作，以进口它所使用的进口值；
— 除了§ 3.2.1.1中定义的扩充以外，这两个状态节点后面也接一个信号XWAKE的输入节点，随后的跃迁将返回到计算此条件的判定；
— 同样除了§ 3.2.1.1中定义的扩充外，除XWAKE以外的每个输入节点后面连接信号XDETACH的一系列输出节点（每一个对应于允许条件中使用的一个进口值），并在此序列后面连接与原来结构中相同的跃迁。

注一　用户画这个图：



注—此图只作说明用。

图 15/Z .103

用SDL/GR说明使用进口值的单个允许条件

3.2.1.3　*允许条件中使用进口值时出口者的动作*

当允许条件中使用出口值时，除了§ 3.1.1.2和3.1.1.3中定义的那些特性以外，出口者还有如下一些隐含的特性。

为了引起对使用进口值的允许条件的重新计算，出口者隐含地维持一个进程实例表XLIST，表中的进程实例在值发生变化时应得到通知。进程实例通过发送信号 XATTACH 而加入该表，或者通过发送信号XDETACH使自己退出此表。

无论什么时候借助于出口操作出口一个新的值时，都要通过信号XWAKE将此新的值告诉XLIST中的所有进程实例。

所补充的定义是：

对出口一些值的每个进程的扩充是通过：

a）在有效输入信号组中加上隐含的信号XATTACH和XDETACH；

b）给出口的每个值规定一个表XLIST．<名字>。这些表中含有进程实例标识符。

除了§ 3.1.1.2中定义的隐含的跃迁以外，进程的每个状态（包括所有隐含的状态）都增加了两个隐含的跃迁。其中的第一个跃迁包括：

— 信号XATTACH的输入节点；

— 一个任务，它把信号XATTACH的发送者的进程实例标识符加到与该信号中指定的变量相对应的

XLIST中,而这变量是指它的值被出口的那个变量;

— 返回到该跃迁开始前的状态。

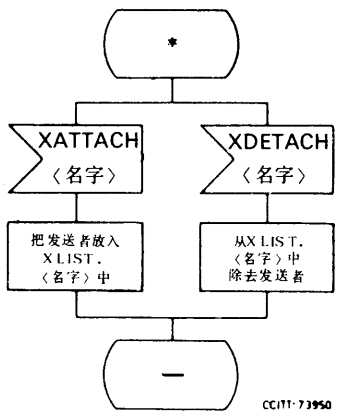
第二个隐含的跃迁包括:

— 信号XDETACH的输入节点;

— 一个任务，它把信号XDETACH的发送者的进程实例标识符从该信号所指定的XLIST中消除;

— 回到该跃迁出发以前的状态。

下面用SDL图形语法给出了出口者的每个状态的补充的隐含跃迁:



注一此图只作说明用。所有跃迁都是隐含的。

图 16/Z.103

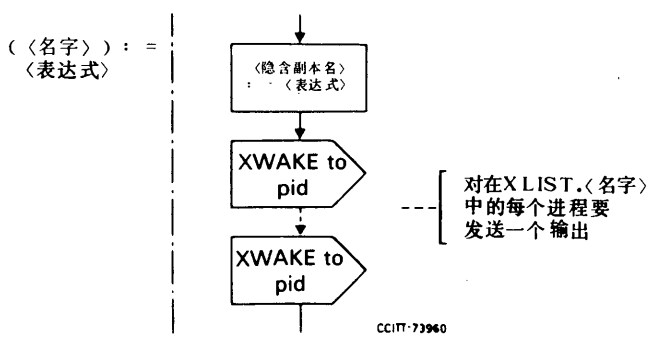
用SDL/GR说明出口进程的隐含跃迁

出口操作的定义(§ 3.1.1.3)中增加了下述隐含的跃迁:

— 包含出口操作的任务后面是信号XWAKE的一个输出节点序列，以传送被透露的值。对于透露数值的相应变量都有一个信号XWAKE送到当前XLIST中所包含的每个进程实例。

下面用SDL图形语法给出了出口操作的补充:

用户写:



注一此图只作说明用。

图17/Z.103

用SDL/GR说明出口操作的隐含动作

3.2.1.4　多重允许条件

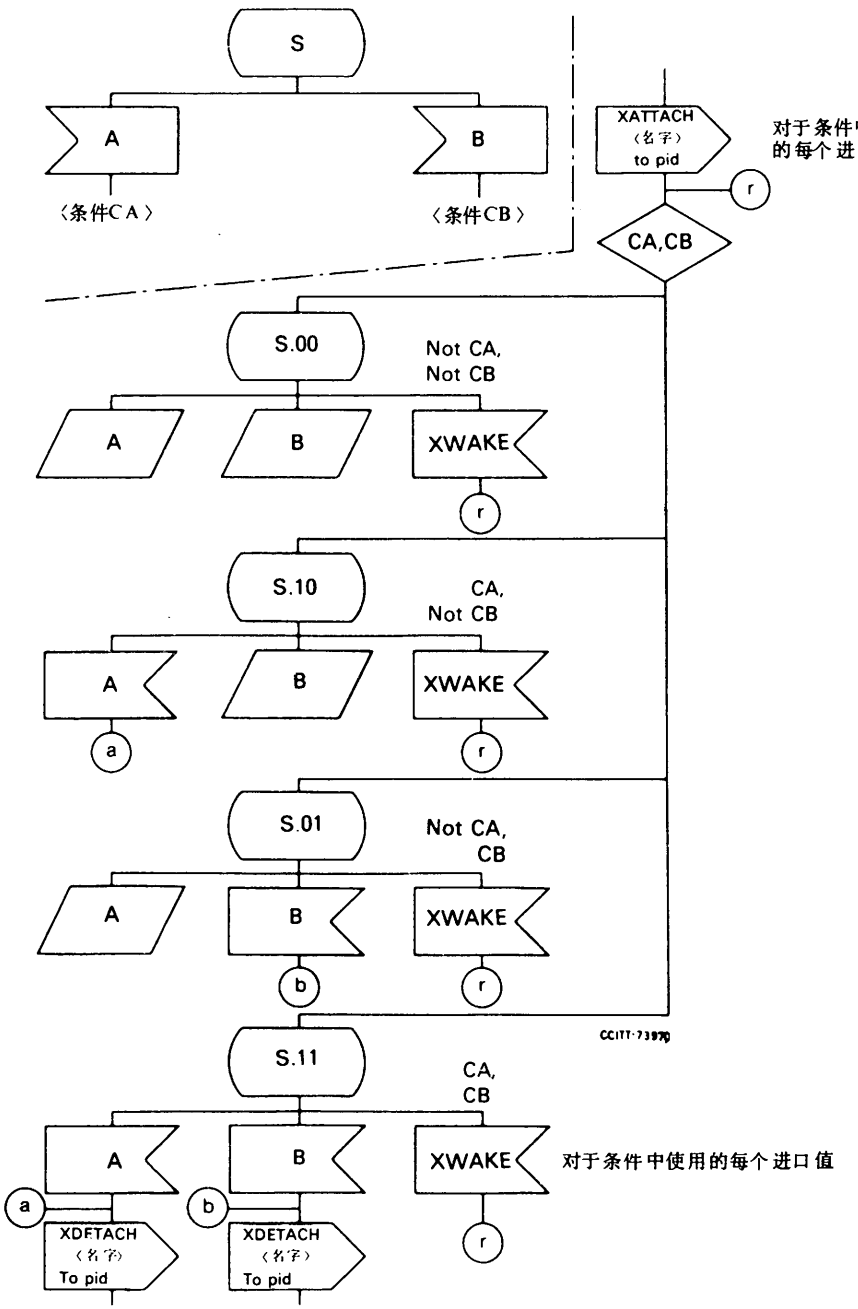
一个状态后有好几个允许条件就称为多重允许条件。

如果一个状态后的n个输入节点附有允许条件，则单个允许条件的定义可扩充为：

a）该状态及后面的输入和保存可用一个判定节点来表示。该判定节点具有一个复杂的条件，其值是一个n元组，其中每个元素是初等允许条件的值。判定节点后面跟以2** n个状态节点，每个状态节点后面是一个跃迁处理。此跃迁适合于该允许条件的特定值。就用此特定值作为进入该状态节点的弧的名字。

b）如果其中任何一个允许条件涉及进口值，则§ 3.2.1.2中的定义仍然适用。

用户画这个图：



注一此图只作说明用。

图 18/Z .103

用SDL/GR解释的多重允许条件

3.2.2　使用允许条件的规则

1）　允许条件可以附在任何输入信号上。

2）　在一个状态后面，含有一个给定输入信号的输入只能有一个，不管此输入是否附有允许条件。

3.2.3　具体语法

3.2.3.1　SDL/GR

在SDL/GR中，允许条件用一个输入符号后接一个允许条件符号来表示。其要素是一对尖括号括住一个布尔条件。图形语法在图19/Z.103中给出：

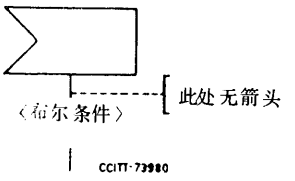


图 19/Z.103

允许条件的SDL/GR语法

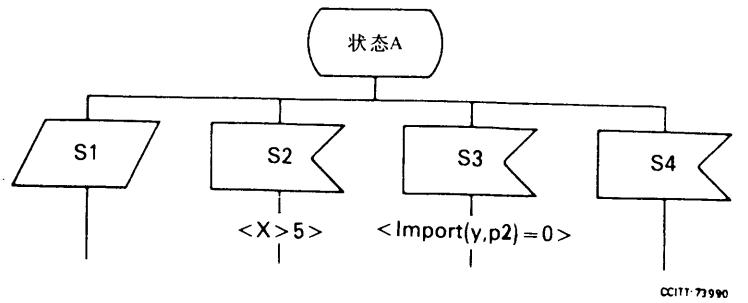


图 20/Z.103

使用SDL/GR允许条件语法的例子

3.2.3.2 SDL/PR

在SDL/PR中，通过增加PROVIDED短语来修改输入语句的语法。PROVIDED短语包含一个布尔表达式，经修改后的输入语句的语法图在图21/Z.103中给出：

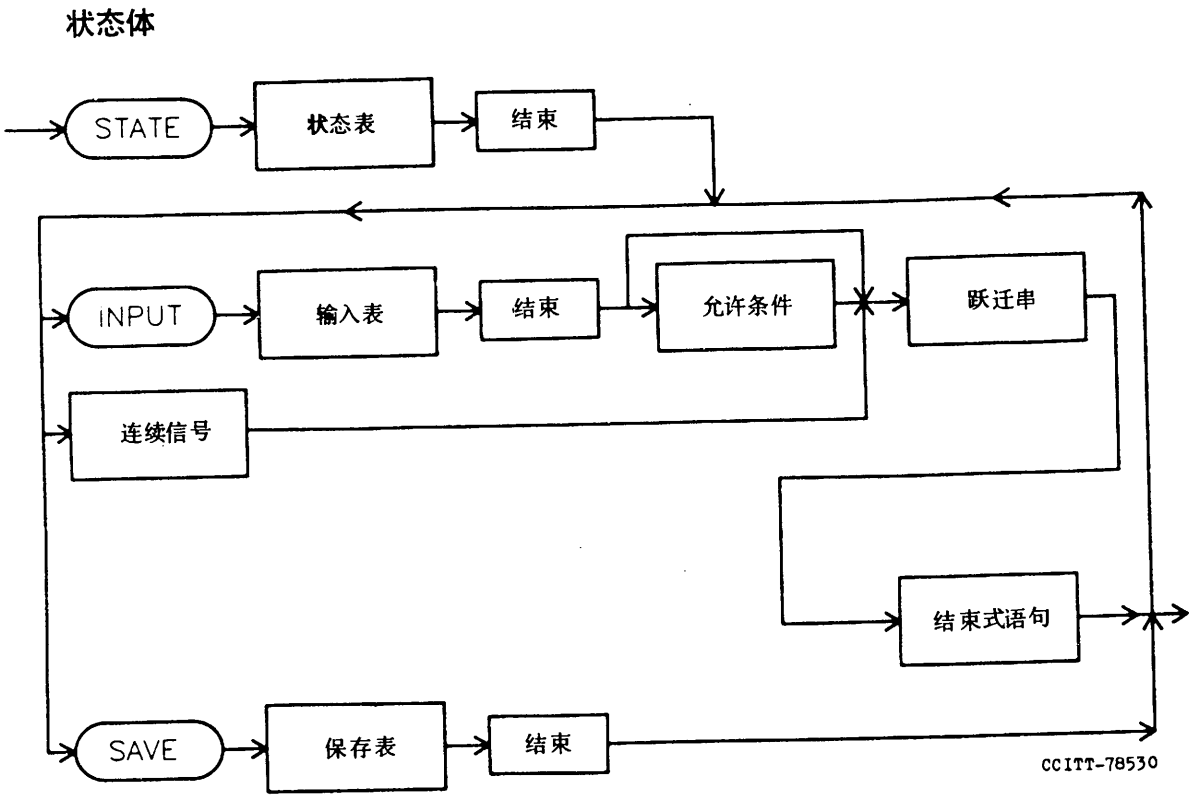


图 21/Z.103 (2中之1)

包括PROVIDED短语的输入语句的语法图

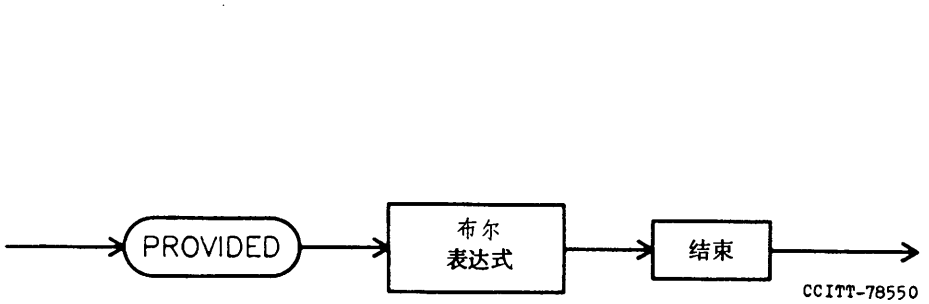


图 21/Z.103 (2中之2)

包括PROVIDED短语的输入语句的语法图

3.3 连续信号

在用SDL描述系统时，经常出现这种情况：用户想看一看由进程外部的变量的值的变化所引起的跃迁。例如，此值可能是一条导线上的高电压或低电压，或者是状态寄存器中的一个数。在SDL中要达到此目的，通常的方法是作一安排，使得当此值发生变化时就产生一个信号，并使该跃迁基于对该信号的接收。如果需要明确地定义、产生和接收这样的信号，将会使进程流图复杂化。被称为连续信号的组合操作允许一个条件的值的变化直接起动一个跃迁。

一个允许条件表示进入一个状态前的一次判定。当使用进口值时，允许条件借助于隐含的XWAKE信号提供离开一个状态的出路。当它是一个连续信号时，也可以不要有关的输入信号而使用它。在这种情况下，允许条件不再表示额外的、隐含的状态，而代之定义一种情形，在此情形中可以离开后面接有允许条件的状态。这种情形的优先权比保留信号低。

若干个连续信号可来自同一个状态，并且在同一时刻可有两个以上的条件为真。每个连续信号都与一个优先权有关。该优先权决定测试各连续信号的次序。

仅使用局部值的连续信号提供了一种手段，以便有条件地离开一个状态，如果在进入的时刻，输入端口中没有信号在等待的话，当该条件中使用的多个进口值中的一个有变化时，使用进口值的连续信号将增加重新计算该条件的能力。

3.3.1 定义

下面的定义基于值的进口和出口的定义（见§ 3.1)和允许条件的定义（见§ 3.2)。

后接连续信号的状态的解释可表示为一个一般模型，模型中有若干个连续信号，它们共同引用若干个进口值。如果没有使用进口值，则此模型可简化，把XATTACH和XDETACH输出及XWAKE输入消除即可。

状态和接在它后面的一组输入和保存，连同连续信号一起可替换为：

1) 信号XATTACH的一个输出节点序列，每一个输出节点对应于连续信号条件中所用的多个进口值中的一个；
2) 一个任务，产生一个唯一的值用在信号EMPTYQ中，EMPTYQ信号在3)中使用；
3) 信号EMPTYQ的一个输出节点，此信号送到发送者的进程实例个体，即送到它自己的输入端口；
4) 原进程流图或过程流图中的一个状态节点，后面接一组输入节点，它们包括原来的输入节点组以及两个其它的输入节点；
5) 每个输入节点后面接信号XDETACH的输出节点序列，每个输出节点对应于连续信号条件中所用的多个进口值中的一个。然后在这个序列后面接原来跟在输入节点后面的那个跃迁处理；
6) 4)中的状态节点后也接一个信号XWAKE的输入节点，此信号起动了7)中的跃迁；
7) 一系列任务节点，它们进口用于连续信号条件中的每个进口值；
8) 对应于每个连续信号条件的一系列判定节点，最先计算的判定是优先级最高的连续信号的判定（在具体语法中优先数最小)；
9) 每个判定的FALSE分支接到下一个优先级的连续信号条件的判定节点。优先级最低的连续信号判定的FALSE分支要接回到4)中的状态；
10) 每个判定的TRUE分支连到信号XDETACH的一系列输出节点，每个输出节点对应于连续信号条件中使用的多个进口值中的一个，后面接与判定中所测试的连续信号条件相对应的跃迁处理；
11) 4)中的状态节点后面还接一个信号EMPTYQ的输入节点，后面再接一个判定节点，它测试此信号是否携带有2)中赋给它的值，也即，它是3)中发送的同一个信号，还是更早的、未经处理的信号EMPTYQ。此判定的TRUE分支连至7)中处理的连续信号，而FALSE分支则回到4)的状态节点。

若进程包含使用进口值的连续信号，则它的有效输入信号组扩充了隐含的信号XWAKE。包含连续信号的任何进程的有效输入信号组用隐含的信号EMPTYQ来扩充。

用户画此图：

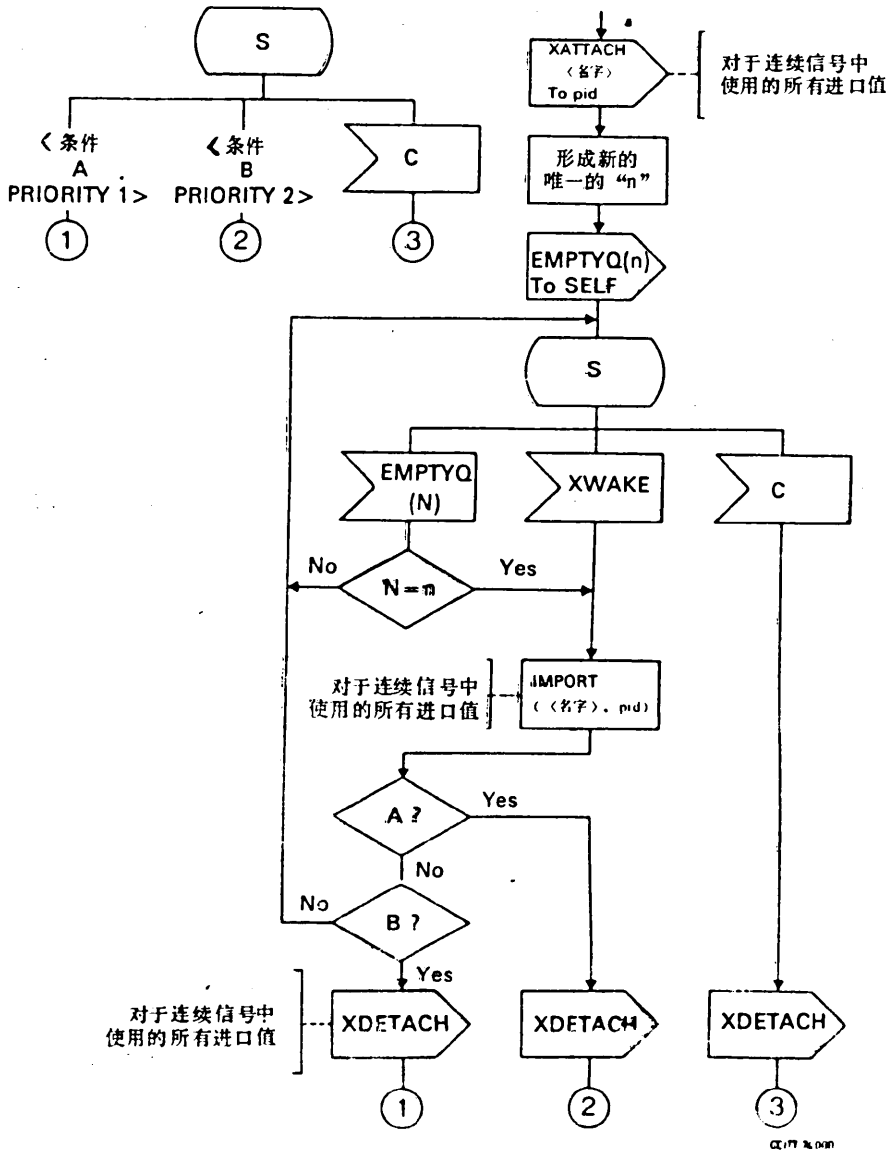


图 22/Z.103

用SDL/GR解释连续信号

3.3.2 使用连续信号的规则

1) 连续信号可接在任何状态后面。
2) 连续信号条件可以基于局部值和（或）进口值。
3) 接在同一状态后的两个连续信号不能有相同的优先数。

3.3.3 具体语法

3.3.3.1 SDL/GR

在SDL/GR中，一个连续信号可以用一个允许条件符号表示，该符号直接连在状态符号后面，即跃迁不以一个输入符号开头。此符号包含连续信号条件和后接一个优先数（自然数）的关键字PRIORITY。此数越小，连续信号的优先级就越高。

如果一个状态后面只有一个连续信号，PRIORITY子句就可省略。如果省略了此子句，则意味着优先数为"1"。

图 23/Z.103

连续信号的SDL/GR符号

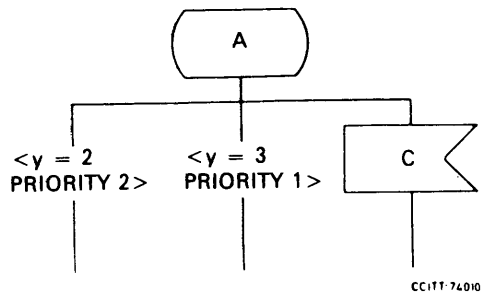


图 24/Z.103

使用SDL/GR连续信号的例子

3.3.3.2 SDL/PR

在SDL/PR中，一个PROVIDED语句后面接一个跃迁串表示连续信号。此语句包含一个PRIORITY子句。PRIORITY子句中的数越小，连续信号的优先级就越高。

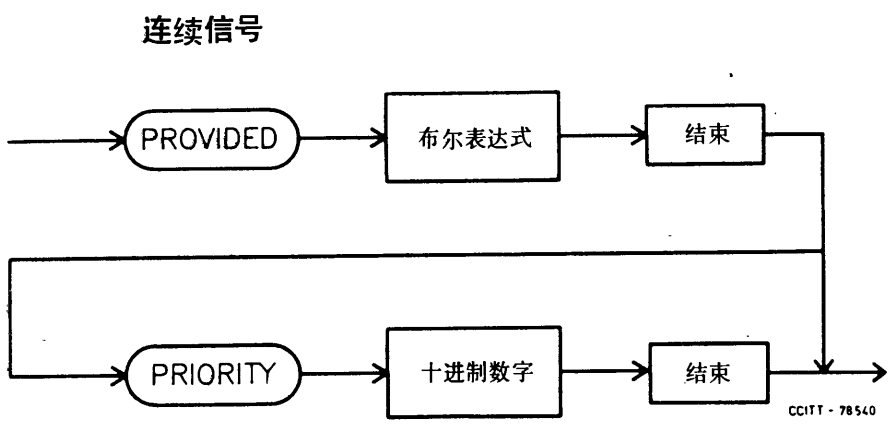


图 25/Z.103

连续信号的语法图

4 宏

宏是一个由用户定义的简写符号，它可以在一个系统的具体SDL表示的一处或多处出现。它表示引用在某处的文件中的一个定义。宏只是具体语法的一部分，为了解释出现宏的SDL表示，必须用宏的定义体来代替宏。

4.1 定义

宏可以表示语法项的任何集合，然而，很显然它不能是递归的（无限膨胀！）。

宏可以有零个或多个入口处，以及零个或多个出口处。在多于一个入口处的情况下，应该有一个标号附在与宏定义中的入口处标号相对应的每个入口处上，在只有一个入口处的情况下，标号可以省略。这同样适用于出口处。

没有任何范围或可见性与宏概念本身有关。对宏引用的解释只有当宏用其定义代替时才能得到。

4.2 具体语法

4.2.1 SDL/GR

4.2.1.1 语法

在SDL/GR中，对宏定义的引用用宏符号来表示。

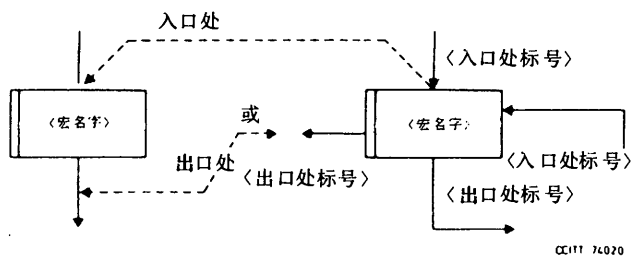


图 26/Z.103
SDL/GR的宏符号

连到宏的入口处和离开宏的出口处可以用连到或离开此符号的流线表示。标号可任选地附在流线上。

宏符号包含它所指的宏定义的名字，并且可把注释附在此符号上。

宏定义称为：

〈名字〉 MACRO DEFINITION

其中〈名字〉就是宏的名字，它用于宏符号中。

宏定义包含了图形表示，此图形表示在作解释前替换宏符号。连到此定义的入口处和离开此定义的出口处分别由流线连到和离开此定义中的符号来表示。入口处和出口处都可附有标号。

4.2.1.2 符号

图27/Z.103中定义了两个补充符号：

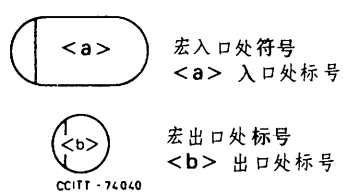


图 27/Z.103
宏的补充符号

4.2.1.3 在SDL/GR中使用宏的规则

宏符号可以插在图中的任何地方，可表示任何SDL/GR符号的组合。

宏符号中的入口处个数必须与所引用的宏定义中的入口处个数相同。宏符号中入口处上的标号必须与宏定

义中入口处上的标号相同。同样的规则适用于出口处和出口处标号。

入口处或出口处可以在符号的任何一边终止或开始。

由于宏的语义是通过用定义中的符号组合来代替而得到的，因此所有图形约定适用于对全部的宏作了替换以后的图。由于这些规则，可能会导致预料不到的后果，例如状态的多次出现。在SDL用户指南中对此作了进一步的讨论。

4.2.2 SDL/PR

4.2.2.1 语法

使用下述语法可把宏调用放在任何图中：MACRO 宏名字；

宏扩展是一段SDL/PR程序，以：

MACRO EXPANSION 宏名字

开头，并以：

ENDMACRO 宏名字

结束。在最后一个语句中宏名字不是非要不可的。

根据宏被引用的地方，这个定义必须紧接着放在SYSTEM、BLOCK或PROCESS结构的后面。宏定义可以包含任何字符，其正确性只有在它替换了宏语句后才能判断。

4.2.2.2 SDL/PR中使用宏的规则

在SDL/GR中使用宏时的规则和考虑同样也适用于SDL/PR。

5 任选

当用SDL来规定或描述好几个类似的应用时，通常，同一个进程定义只要略加修改就能用于不同的系统。SDL中的OPTION办法提供了一种手段来定义对几个应用都通用的进程，办法是引入描述的可替换的任选部分。

5.1 定义

任选是根据对任选表达式的判断来选择进程定义的可互换部分。在解释进程定义之前要作出此选择。

任选办法只是具体语法的一部分，应该被看成是一个简写符号，它给若干个应用提供一个通用的描述，而不是给每个应用提供一个专用的描述。

5.2 具体语法

5.2.1 SDL/GR

5.2.1.1 符号

在SDL/GR中使用下述符号来表示任选：

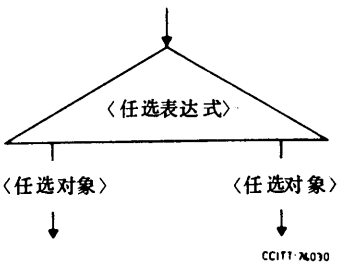


图 28/Z.103
SDL/GR 中的任选符号

5.2.1.2 SDL/GR中使用任选的规则

在进程图中，任选符号可以接在任务符号、判定符号的出口处、输出符号或过程符号的后面。任选符号后面可接状态符号、任务符号、判定符号、输出符号或过程符号。

符号中包含的〈任选表达式〉是这样的一个表达式，即经过计算后唯一地选定连在此符号后面的〈任选对象〉中的一个，并且在解释进程之前可以计算出此表达式。每个任选对象必须是与任选表达式同类型的一个值。当解释最终得到的进程时，该进程定义的不可达部分可认为是被删除了。

5.2.2 SDL/PR

5.2.2.1 语法

在SDL/PR中，任选由下列语法表示：

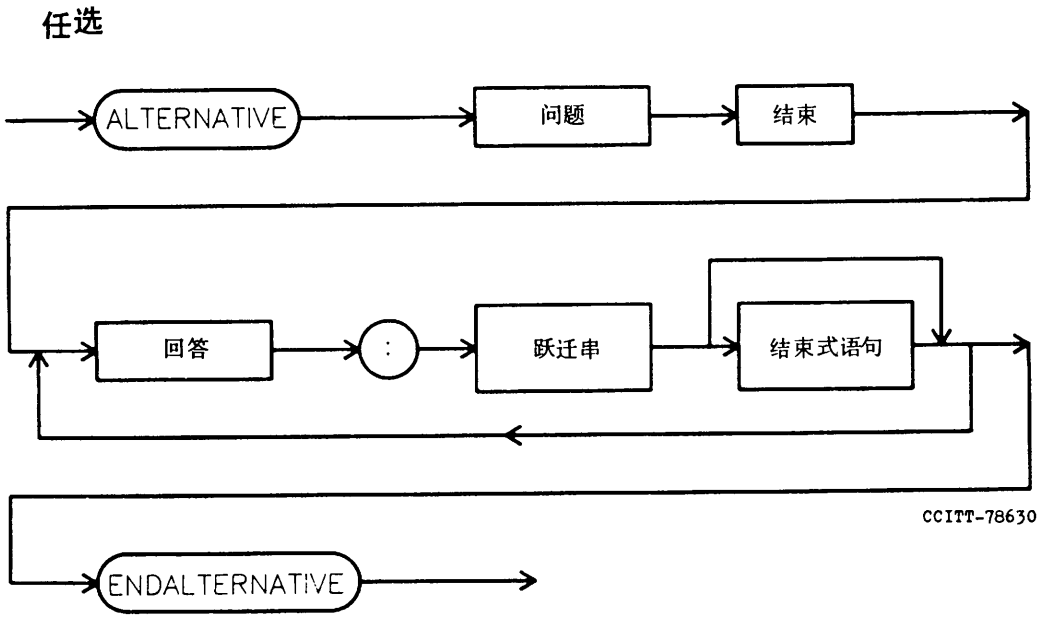


图 29/Z.103
任选的语法图

5.2.2.2 SDL/PR中使用任选的规则

语句中包含的〈任选表达式〉是这样一个表达式，即经计算后只能唯一地选择〈任选对象〉中的一个，并且此表达式在解释进程前算出。每个任选对象必须是与任选表达式同类型的一个值。当解释最终形成的进程时，该进程定义的不可达部分可看成是被删除了。

6 SDL/GR中的图形元素

当用SDL的图形语法来表示进程定义时，在状态符号内使用图形元素来构成一个状态图形是SDL/GR中可供选用的部分。

这种状态图形的优点是当用于某些系统定义时，可以得到更为紧凑而较少文字叙述的进程图。借助于图形元素和限定文句，状态图形可以描述进程在该状态时的实际状况。还有，状态图形中也可以描述对该进程的环境所设想的状况。使用状态图形时，所描述的状态的区别就隐含了状态间的跃迁所完成的动作。

使用图形元素时，其他在建议Z.101中定义的语法和语义仍然适用。然而，这些语义和语法被扩充了，下面就作出定义。

6.1 状态图形的语义

使用图形元素时，一个状态节点用一个状态符号表示。状态符号用其名字识别，并含有一个状态图形，它包括图形元素、变量值、输入变量和限定文句。

状态图形能够：

1) 通过采用图形元素和限定文句，表示该状态中与该进程有关的全部变量的一个经挑选的子集中的值。经挑选的子集可包含一些变量，它们纯碎作为与其它进程有关的变量的代表。这些变量“代表”携带有与其它进程有关的变量的值。这些值或者通过视见得到，或者通过进口得到。

2) 通过使用输入变量的值，表示对该状态关于有效输入信号的输入动作。

图形元素的数量原则上没有限制，因为可以创造新的图形元素来适应SDL的任何新的应用。然而，在电信交换和信号功能的应用中，下述的一组图形元素具有很大的通用性：

— 进程边界（左或右），
— 终端设备（多种），
— 信号接收器，
— 信号发送器，
— 信号收发器，
— 定时器监视进程，
— 交换通路（已连接、已保留），
— 交换模块，
— 正在计费，
— 控制元素，
— 不确定符号。

§ 6.3 中推荐了这些图形元素的标准符号。

6.2 解释规则

1) 输入变量是布尔变量，且每个输入变量对应于该进程的有效输入信号组中的一个信号。状态图形之间的一个输入变量之值的变化总是表示由进程的一个输入吸收了一个信号。因此，输入变量能用来表示一个进程的那些条件，这些条件如果发生变化，就会导致该进程去执行一个跃迁。输入变量的值可以与图形元素对应。

2) 一个状态图形中，图形元素的存在表示了当一进程在该状态中时变量子集的特定值。附加变量的值、特别是与这个初始子集有关的变量的值可以通过使用限定文句对图形元素作出限定来表示。限定文句不是一个输入变量；限定文句的改变并不表示该进程的输入动作吸收了一个输入信号。

3) 定位

a) 任何图形元素（不是进程边界）相对于进程边界的位置决定了此图形元素是进程的“内部”还是“外部”。内部的图形元素表示该进程所拥有的变量。外部的图形元素表示由另一个进程所拥有的变量，因此，要访问这些变量就必须要使用视见和进口这些手段。

b) 规则a)也适用于区分内部的还是外部的限定文句，只要用术语“限定文句”替换此规则中的图形元素即可。

4) 基本规则：

从一个状态转移到下一个状态所包含的全部处理工作就是为了改变状态图形，所需要的处理工作也包括在状态间的跃迁中出现的任何判定、输出或任务所指明的处理。于是：

a) 从一个状态中一个内部的图形元素的出现变化到下一个状态中该图形元素的消失，或反之，对应于某些变量值的变化，而这些变量能够用状态间的跃迁中的一个任务来等效地表示。

b) 从一个状态中一个外部的图形元素的出现变化到下一个状态中该图形元素的消失，或反之，对应于另一个进程所拥有的变量的变化。此变化可以用传往那个进程的输出信号来等效地表示，或者可以简单地用从那个进程传来的输入信号表示。

c) 规则a)和b)也适用于状态图形中限定文句的出现或消失，只要用术语“限定文句”替换在那些规则中的图形元素即可。

5) 对一个给定的进程图，特定的图形元素（或图形元素与限定文句的特定组合）唯一地放在状态图形内，这样，状态符号中该图形元素（或组合）的出现或消失能够通过此状态图形与进程图中的其它状态图形的比较而迅速决定。

6) 当状态图形中出现信号发送器时，其限定文句确定了一个信号。这个信号在本状态以前已经输出或者(在由该进程控制的连续信号的情况下)在本状态以前和在本状态期间已经输出。

6.3 推荐的图形元素符号

使用图形元素时，每个状态用一个状态符号表示，它包含一个状态图形，其格式见图30/Z.103：
用于SDL/GR的图形元素的一个基本集合已经定出标准，用于电信呼叫处理过程的系统描述，包括信号

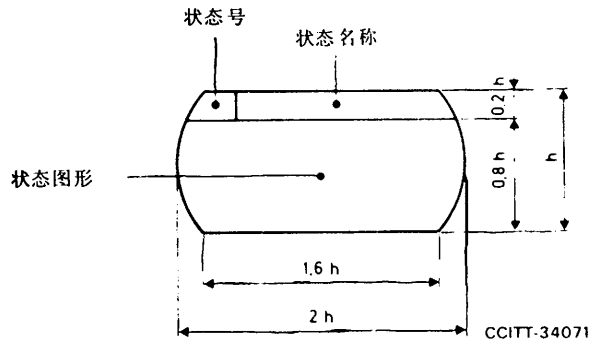


图 30/Z.103
推荐的带有状态图形的状态符号的格式

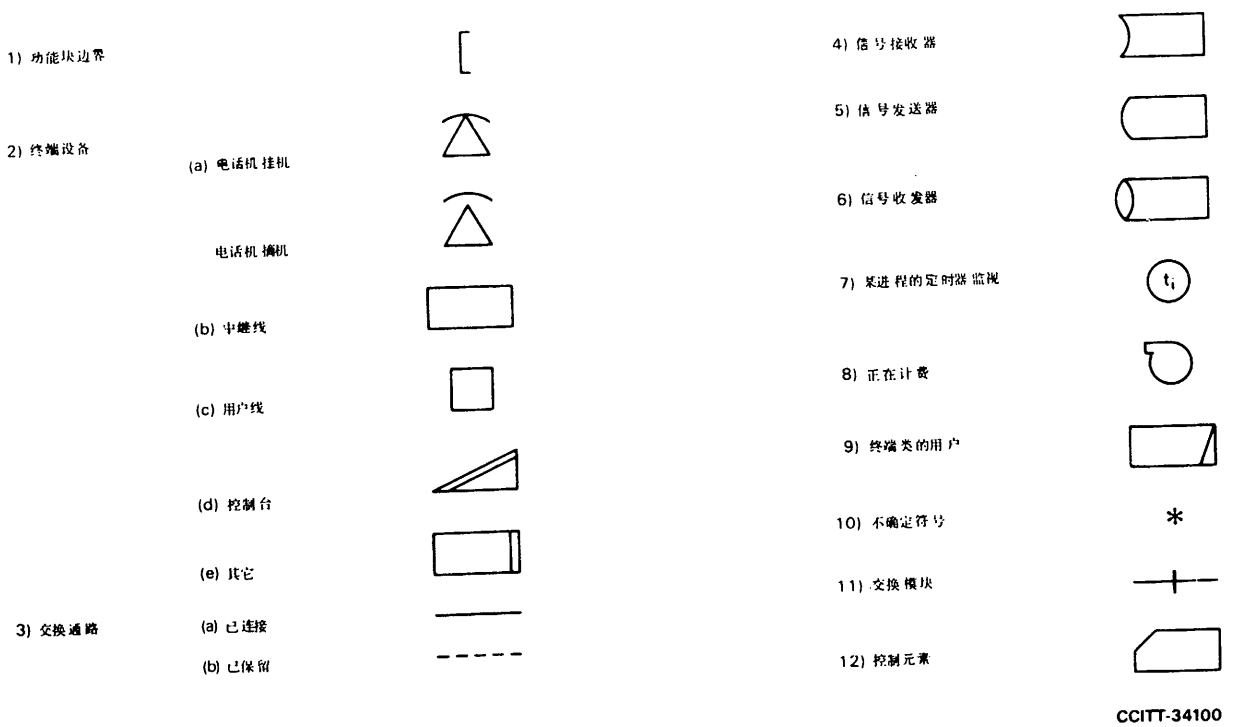


图 31/Z.103
推荐的图形元素概念的基本集合的符号

协议、网络服务及信号互相配合的过程。许多这些图形元素能够适用于呼叫处理过程以外的SDL/GR的应用，并且鼓励把它们应用到电信系统的合适的其他进程中。

推荐的图形元素基本集合的符号见图31/Z.103：

挑选图形元素的图形是以本建议附件A中给出的考虑和一般选择准则为基础的，为了扩大SDL/GR的应用而提出补充的图形元素之前应该参考此附件A。

推荐的图形元素符号的尺寸比例见本建议的附件B。

本分册的封底内侧装有一个样板，它便于手工画出SDL/GR符号的基本集合。在此基本集合中此样板包括了图31/Z.103中所示的图形元素符号。

6.4 面向状态的扩展SDL/GR中使用的专门的约定和解释

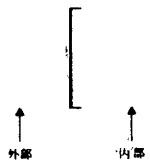
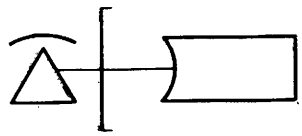
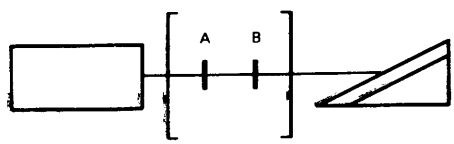
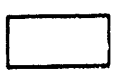
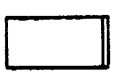
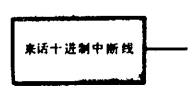
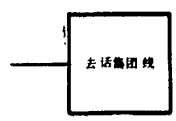
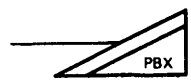
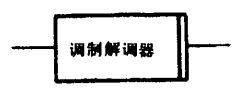
本节针对面向状态的扩展SDL/GR，定义了若干专门的约定和解释。包括：

- 根据所谓的基本规则（见§ 6.2，规则4），进程图所要求的专门的解释。
- 在状态图形内的图形元素（或图形元素和限定文句）的唯一的位置安排，这在使用图形元素（见§ 6.2，规则5）时是需要的。
- 由外部的图形元素和外部的限定文句表示的变量所需的专门的解释，这些变量作为与其它进程有关的其它变量的代表变量。

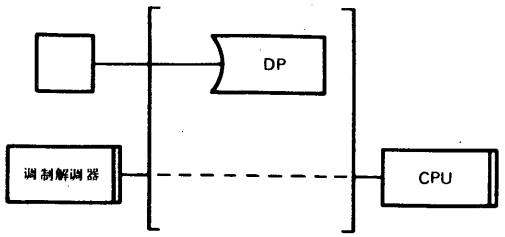
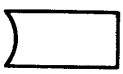
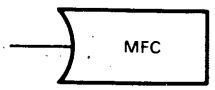
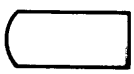
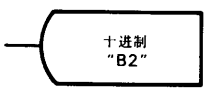
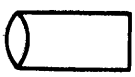
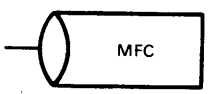
附 件 A

（属建议 Z.103）

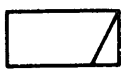
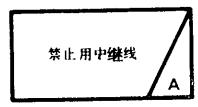
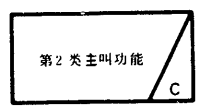
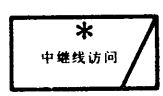
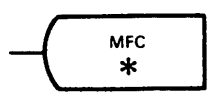
图形元素基本集的应用举例

序号	图形元素	注释	例
1.	功能块（FB）边界 	用来区别FB边界的内部元素和外部元素。 只有在边界内的元素的状态才能由本进程直接改变。	1.1 FB边界外的电话机连到FB边界内的数字接收器  1.2 FB边界外的中继线经两级交换单元连到FB边界外的控制台 
2.	终端设备 a) 电话机 挂机  摘机  b) 中继线  c) 用户线［除a)以外］  d) 控制台  e) 其它 	用来表示FB边界外的终端设备（例如电话机和控制台设备），帮助对处理工作的理解。	2.1 A挂机 $\bar{h}$  2.2 B摘机 h_B  2.3 进入十进制中继连接器（来自一空分交换机）  2.4 去集团线的用户线  2.5 PBX 控制台  2.6 调制解调器 

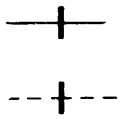
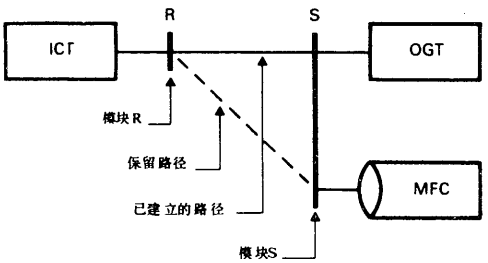
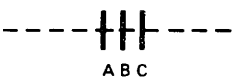
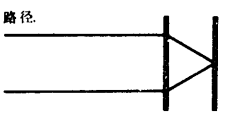
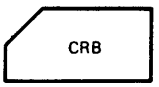
CCITT-20880

序号	图形元素	注释	例
3.	交换通路 a) 已连接 —————— b) 已保留 - - - - - -	表示进程中所涉及的终端设备与（或）信号设备间的连接。	3.1 用户线连到拨号脉冲数字接收器（DP）。 调制解调器用一条保留路径连到一中央处理单元（CPU） 
4.	信号接收器 	规定信号接收过程，并指明所接收信号的性质，特别是经过功能块边界的信号。	4.1 多频码信号接收器（MFC）  4.2 MFC 十进制信号接收器（MFC DEC） 
5.	信号发送器 	规定信号发送过程，并指明所发信号的性质，特别是要经过功能块边界的信号。	5.1 具有送出后向信号“B 2”功能的十进制信号发送器 
6.	混合信号收发器 	方便地把信号发送器与信号接收器的功能结合起来。	6.1 MFC收发器 
7.	监视进程定时器 	定时器影响进程的后续行为。 注：表明时间到的有关输入符号可表示为 $\bar{t}_i$。	7.1 定时器t_3正在计时  7.2 通用定时器t_s正在计时  s = 1，2，…n规定不同的服务台。

CCITT-20890

图形元素	注释	例
正在计费（以及被计费的是哪个用户） 	计费方法对主管部门、制造厂家及用户都很重要。	8.1 当前正在对用户A计费 
用户或终端类别（及识别信息）。 	对于多分机呼叫中的每个分机，用户或终端类别的变化将影响进程的行为	9.1 A分机禁止用中继线  9.2 分机C有第2类 
不确定符号 *	特用此符号表示不确定的信息，而在其它状态图形中可清楚地表示它。在某些情况下，用不确定符号可把两个或多个状态合并成一个，使图更易理解	10.1 挂机或摘机的电话机  10.2 进程在此状态时，用户类别可以是“禁止用中继线”或相反  10.3 在此状态下，送出一个未确定的MFC信号 

CCITT-20900

序号	图形元素	注释	例
11	交换模块  或 	表示在过程中涉及到哪些交换模块。 注：水平线是交换通路的图形元素，它可以是已连接或已保留。垂直线可用来表示一个完整的交换模块（当模块的内部结构并不需要时）或者在一个交换模块内的一个交换级。	11.1 通过一个交换模块连接的一条路径 LLN＝线路链接网络  11.2 经过两个交换模块连接一条路径及保留一条路径  ICT—入中继 OGT—出中继 MFC—多频码 注：此例中，ICT到OGT是连通的，但ICT与MFC收发器没有接通。 11.3 经过三级交换模块RSN接通的一条路径  11.4 经过三级交换模块ABC的一条保留路径  11.5 经过一个折叠网络接通的一条路径。 
12.	控制元素 （配给一进程） 	表示在进程中涉及的控制设备（尤其是必须标注尺寸的那些模块）。此符号也可用来表示已配给进程的特定的软件元素。	12.1 呼叫记发缓冲器（CRB） 

CCITT-20910

附 件 B

（属建议Z.103）

图形元素（PE）的选择准则

B.1 概述

PE符号的选择基于下述考虑和一般的选择准则，为了扩大SDL的应用而提出补充的PE符号以前应该参考这些考虑和准则。

B.2 典型的读者

阅读采用PE的SDL图的人员估计是下述范围内的技术人员和非技术人员：推销新设备、对新设备作规格说明、根据规格说明开发硬件和软件、管理研制项目、交换机的操作和维护、通信话务工程、电话技术方面的教育和培训课程。预计SDL图可作为共同使用的文件：

a) 在主管部门与制造厂家之间，

b) 在这些组织内部的不同部门之间，

c) 作为电话交换文件，以及

d) 在训练手册和教材中。

并不指望SDL图能用机器来阅读。但指望SDL（包括PE信息）的SDL/PR形式能由机器来阅读，然后由机器画出图来。（见§ B.3的b)和c)）

B.3 典型的画图方法

含有PE的SDL图常常要由技术人员，包括绘图员来画：

a) 用手工画，以样板作为画图工具，和（或）

b) 用电子装置在图形显示单元上显示此图，和（或）

c) 使用电控绘图机。

B.4 复制方法

典型的复制方法是：

a) 传统的方法，如染线或蓝图方法；

b) 用办公机器照相复制，包括照相缩版；

c) 一般的影印。

B.5 便于复制

为了能用染线或蓝图方法以及照相复制和影印方法方便地复制SDL图，PE符号的线条应该清晰而不带阴影。

B.6 便于画图

主要的画图方法是用样板由手工描绘，其次是在用电子装置控制的屏幕上显示一个图，然后用电控笔描绘。下述准则反映了这一点：

a) 每个PE符号应该容易用钢笔或铅笔徒手描绘或用绘图板描绘；

b) 所有PE符号应该用同样粗细的线条描绘；

c） 为了能用电子装置容易地产生PE符号，新创造的PE符号应该用简单的几何线条和曲线构成。

B.7 便于理解

这是所有考虑中最重要的一个，因为SDL文件的特性是其读者大大多于绘画员(或作者)。这一要求用下述关于PE符号的准则表达：

a） 适合性 -每个符号的形状应该适合于该符号所表示的概念。

b） 区别性 -选择符号的基本集时,应该仔细考虑,以使得每个符号与该集合中的其他符号可以明显地加以区别。

c） 相似性 -所表示的功能不同但又相关的PE形状，例如接收器和发送器，应该在某些方面有相似之处。

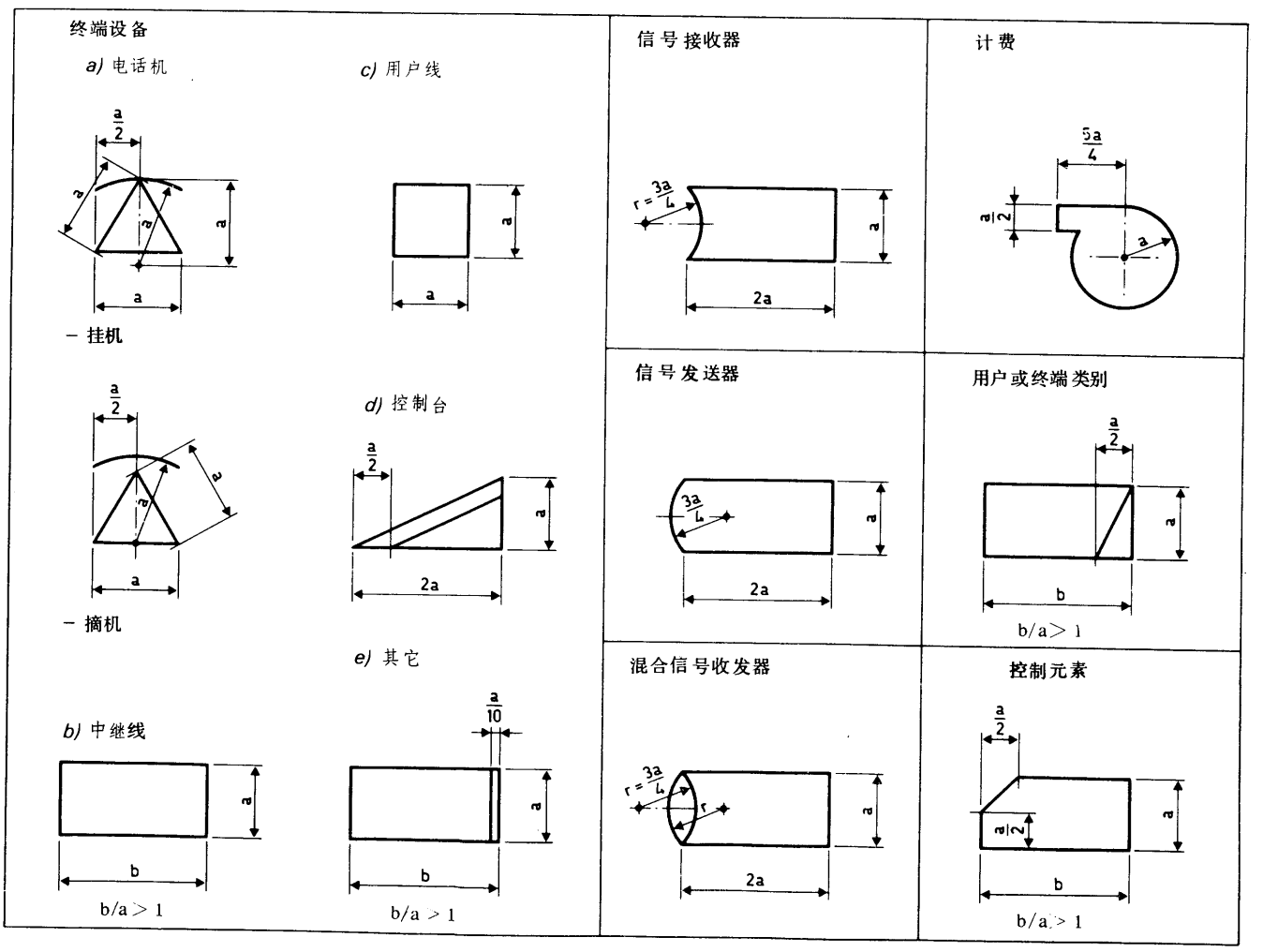
d） 缩写文字与符号的结合 -在有些情况下,为了表明PE的类别,希望把缩写文字与PE符号相结合；例如字母MFC与一个接收器符号结合就表示将接收多频码信号(MFC)。在这些情况下，PE应该有周围封闭的空白处，以允许填写少量字母和数字符号。

e） 有限集-为了能使此图形方法容易学习，应该使符号的总数尽量少。

附 件 C

(属建议Z.103)

推荐的图形元素基本集的线条比例



SDL 中 的 数 据

1 引 言

本建议定义SDL中的数据概念、SDL的数据术语、预定义数据类型的使用及定义新的数据类型的办法。

SDL中，数据主要出现在数据类型定义、表达式、以及运算符、变量、值、常量和字面值的应用中。

数据类型定义

SDL中的数据主要与数据类型有关。数据类型定义一组值、一组可作用于这些值的运算符、以及一组公理用来定义当这些运算符作用于那些值时的行为。值、运算符及公理一起定义了数据类型的特性。这些特性由数据类型定义确定。

SDL允许定义任何需要的数据类型，包括构成的机制(复合类型)，仅要求这些数据类型应该能够形式地定义。与编程语言相比，它有一些实现方面的考虑，要求这些可用的数据类型，特别是所提供的构成机制(数组、结构等)应该是有限制的。

表达式和运算符

表达式供计算值时使用(通过运用合适的运算符)，以便回送一个新的值。

变量

变量可以通过显式赋值或隐式赋值与一个值相关联。当访问变量时，就送回这个值。

值、常量和字母

所有数据类型至少有一个值。对于大多数数据类型，常有表示数据类型的常量值(例如整数)的字面的(语法的)形式。可用字面值来表示数值的数据类型称为具有可指定常量。可以有多于一个的字面值表示同一个常量值(例如12和H'C都表示同一个整数常量)，并且相同的字面值表示可用于多个数据类型。有些类型却没有可指定常量。例如，数据类型堆栈的值只能通过运算符才能得到，运用运算符可以得到回送的堆栈值。

在一个规格说明语言中，最重要的是允许人们能够根据数据类型的行为形式地描述数据类型，而不象在编程语言中那样，从提供的原始类型来组成它们。后一方法总是包含对数据类型的特定的实现，因而限制了实现者选择数据类型的合适表示的自由。SDL方法允许任何的具体实现，只要对SDL来说，它是可行的和正确的。

在SDL中，所有数据类型都是抽象数据类型。在§5中给出了这些类型的例子，那里定义了本语言的预定义数据类型。

尽管全部数据类型都是抽象的，并且预定义数据类型可由用户定义，在SDL中仍作了一番努力，以提供其行为和语法都是大家所熟悉的预定义数据类型。这些是：

Array(数组)，Boolean(布尔)，Character(字符)，Charstring(字符串)，Duration(持续时间)，Integer

(整数)，Natural(自然数)，Powerset(幂集)，PId，Real(实数)，String(串)，Time(时刻)，Timer(定时器)。

复合类型可使用结构数据类型来产生。

1.1 SDL数据体系

在SDL中，数据由类型代数模拟。一个类型代数是一组域、一个指派域、以及一组在域之间进行映射的功能。每个域是某个数据类型的所有可能值的集合。指派域是当前正在描述的数据类型。功能表示该数据类型的运算。

域和运算连同该数据类型的行为（由公理规定）构成该数据类型的特性。

同义类型的引入产生已定义类型的值的一个子集。新类型的引入创造了类别不同的新数据类型。新类型带有从父本那里继承下来的特性，但是对于这些特性具有不同的标识符。在具体语法中，这些名字不必互不相同，而这种多义性必须通过上下文来解决。

生成元类型是一个不完备的类型描述：在它呈现为一个数据类型的状态以前，它必须提供所缺少的信息，用实例来具体说明。

在某数据类型的域之间进行映射的功能通常划为两类。生成元功能映射到一个指派域，从而产生该数据类型的(可能是新的)值。所有其它功能都是语义功能，通过映射到其它已定义的类型，把意义赋给类型。语义功能包括映射到布尔域的谓词。

2 通用语言模型

2.1 概述

建议Z.104中使用的概念有变量、预定义数据类型、值、表达式和不正规的表达式。此建议严格地定义了变量、预定义数据类型、值和表达式，还对建议Z.101、Z.102和Z.103作了严格的扩充，允许引入新的数据类型。

2.2 抽象语法

本抽象语法对由建议Z.101、Z.102和Z.103所定义的语法作了扩充。

数据定义

数据定义是一个数据类型定义或一个同义定义。

系统定义

系统定义可包含数据定义。

功能块定义

功能块定义可包含数据定义。

内部分块定义

内部分块定义可包含数据类型定义。

信道子结构定义

信道子结构定义可包含数据定义。

进程定义

进程定义可包含数据定义和变量定义。

过程定义

过程定义可包含数据定义和变量定义。

输入节点

在输入节点中提到的变量必须在变量定义中定义，并且必须与信号定义中相应的数据类型有相同的数据类型。

数据类型定义

数据类型定义包含一个类型名以及一个新类型描述或者一个同义类型描述。

新类型描述

新类型描述包含一组（可能是零个）值名、一组（一个或多个）运算符指定以及一组（可能是零个）数据类型公理。

数据类型描述中的所有值名在该数据类型内必须是唯一的。

数据类型描述中的所有运算符名必须互不相同。

在同一上下文中的所有数据类型名必须是唯一的。

运算符指定

运算符指定或者是在通用运算符中指定一个（通用运算符允许连同任何数据类型一起指定），或者运算符定类一起指定一个运算符名。

运算符

运算符或者是带有数据类型标识符的一个通用运算符，或者是带有数据类型标识符的一个用户定义的运算符标识符。在两种情况下，数据类型标识符都允许建立数据类型定义，以便对运算符作出定义。

运算符定类

运算符定类包含关于该运算符的参数的数据类型标识符的列表以及运用此运算符所得的结果的数据类型标识符。

在该表中至少有一个数据类型个体必须是被定义的数据类型，或者，该结果必须是被定义的数据类型。

结果类型不能是同义类型。

公理

对于所定义的类型，在所有情况下都保持真的语句就是公理。它规定类型的行为。

赋值语句

赋值语句包含一个赋值运算符、一个变量个体和一个表达式。赋值运算符或者是用于赋值的通用运算符，用变量的数据类型的数据类型个体加以限制，或者是用户定义的插入运算符。

同义类型描述

同义类型描述含有一个同义类型名、父本数据类型个体和父本数据类型的一组值个体，该组值个体对同义数据类型适用。

表达式

表达式或者是一个初等量，或者是一次运算。

运算

运算包含一个运算符和一系列表达式(一个或多个)。在运算中包含的表达式个数与为该运算符的参数而定义的数据类型的数目一样多。

初等量

初等量是下列之一：

— 一个同义个体，
— 一个值个体，
— 一个变量个体，或
— 一个条件表达式。

条件表达式

条件表达式是一个布尔表达式，并且是相同数据类型的两个表达式的列表。

变量定义

变量定义包括一个变量名表和一个数据类型标识符。

通用运算

通用运算符或者是一个变量运算符，或者是一个比较符。

变量运算符是声明、赋值或访问。声明用来声明变量，赋值用来给变量赋值，而访问每当把一个变量个体解释为一个值的时候就要使用访问。

比较符或者是一个排序运算符，或者是一个等式运算符。排序运算符或者是小于，或者是大于。

所有通用运算符包括了与之相关的作为限定符的数据类型，这样，不同的数据类型引入不同的运算符。例如，数据类型平方和数据类型立方为赋值引入两个运算符个体：square!assign和cube!assign。

对于一个数据类型D，对应于比较符的运算符定类为：

D，D－＞Boolean

对于一个数据类型D，赋值运算符需要一个数据类型D的变量个体和数据类型D的值。

对于一个数据类型D，访问运算符需要一个数据类型D的变量标识符并给出数据类型D的一个值。

对于一个数据类型D，声明运算符需要一个变量标识符。

同义定义

同义定义包含一个同义名和一个常量表达式。

常量表达式

常量表达式或者是一个常量值，或者是一个运算，其参数全部是常量表达式。

常量值

常量值或者是一个值标识符，或者是一个同义标识符。同义字不能递归定义。

2.3 解释规则

2.3.1 进程

进程的实例出现在解释起动节点前，并使声明运算作用于在该进程定义的变量定义中出现的每个变量名。作为声明运算符的结果，所声明的变量把变量个体与一个初始值结合起来（该初始值是一个未定义值，除非在关于该变量的数据类型的公理中已有规定）。当产生一个进程实例时，所声明的变量个体包含变量名、进程实例个体和变量定义中的数据类型个体。

2.3.2 过程起动节点

与产生一个进程实例的方法类似，调用一个过程就产生一个由过程所创建的在该过程中定义的变量。

2.3.3 进程图

对一个输出节点的解释使该输出节点中的每个表达式按所规定的次序得到解释，并把所形成的值赋给与信号有关的无名隐含变量。当对输出节点进行解释时这些变量被认为是声明了的。这些变量中的每一个所具有的数据类型是同义数据类型，它与信号定义中的相应位置有关。赋给这些变量的值是由该信号传送的值。

对一个判定节点的解释使该判定节点中所包含的表达式得到解释，接着选择与表达式所提供的值相对应的那条弧。

创建请求

对创建请求节点的解释使该创建请求节点中的每个表达式按所规定的次序得到解释。

然后，产生被创建的进程实例，并声明该进程的形式参数，还有把该创建请求节点中每个表达式的相应的结果值，赋值给每个形式参数。然后，与其它进程分开地（但与之并发地）执行此创建的进程。

调用节点

调用节点使得用作为形式参数的表达式在过程的起动节点的解释之前解释，其中形式参数带有属性IN。每个表达式把它的值赋给相应的实在参数。

2.3.4 过程

带有属性IN/OUT的形式参数被解释为过程调用的上下文中与实在参数相应的变量标识符。带有属性SIGNAL的形式参数被解释为过程调用的上下文中与实在参数相应的信号标识符。

2.3.5 赋值语句

赋值语句被解释为把变量的旧值与一个表达式的值相结合，并使变量个体与这个新值结合起来。

赋值运算符决定了把变量的旧值与一个表达式的值相结合的规则。数据类型的数据类型公理中赋值运算符的使用决定了这些规则。如果赋值运算符是关于赋值的通用运算符，此变量与表达式的值则连结起来。

表达式的值必须是一个赋值运算符的变量的数据类型值。对于赋值的通用运算符，该参数的数据类型就是该变量的数据类型。

2.3.6 表达式和初等量

表达式解释为构成表达式的初等量。初等量或者是一个运算、一个同义字、一个值标识符，或者是一个变量标识符，并被照此解释。

2.3.6.1 运算

运算被解释为把运算符运用于值表中，这些值通过对表达式表的解释而得到。在数据类型的数据类型公理中，运算符的使用决定了运算的解释。

2.3.6.2 同义标识符

同义标识符被解释为同义定义中所定义的常量表达式。常量表达式按照与表达式相同的方法来解释。

2.3.6.3 值标识符

值标识符被解释为它所指的值。

在数据类型的数据类型公理中，值标识符的使用决定了值标识符所指的值的语义。

2.3.6.4 变量标识符

根据上下文，变量标识符按两种方法中的一种解释。在表达式内的变量标识符被解释为访问。在赋值语句的上下文中，变量标识符被解释为一个变量，它把变量标识符与一个值相连结。除了访问未定义值时解释为出错以外，访问一个变量都解释为与该变量相连结的值。

2.3.6.5 条件表达式

根据布尔表达式的解释是真还是假，条件表达式被解释为表中的第一个或第二个表达式。

2.3.7 数据类型定义

对这些不作解释。

3 SDL/GR

任务、判定等的行为（即这些节点的内部结构）的标准SDL规格说明是SDL/PR形式。故此，对于数据就没有特定的图形语法。

在包含有数据定义（对于数据类型、变量或同义标识符）的地方，它们应该与包含它们的图一起被定义，或者由包含它们的图所引用。

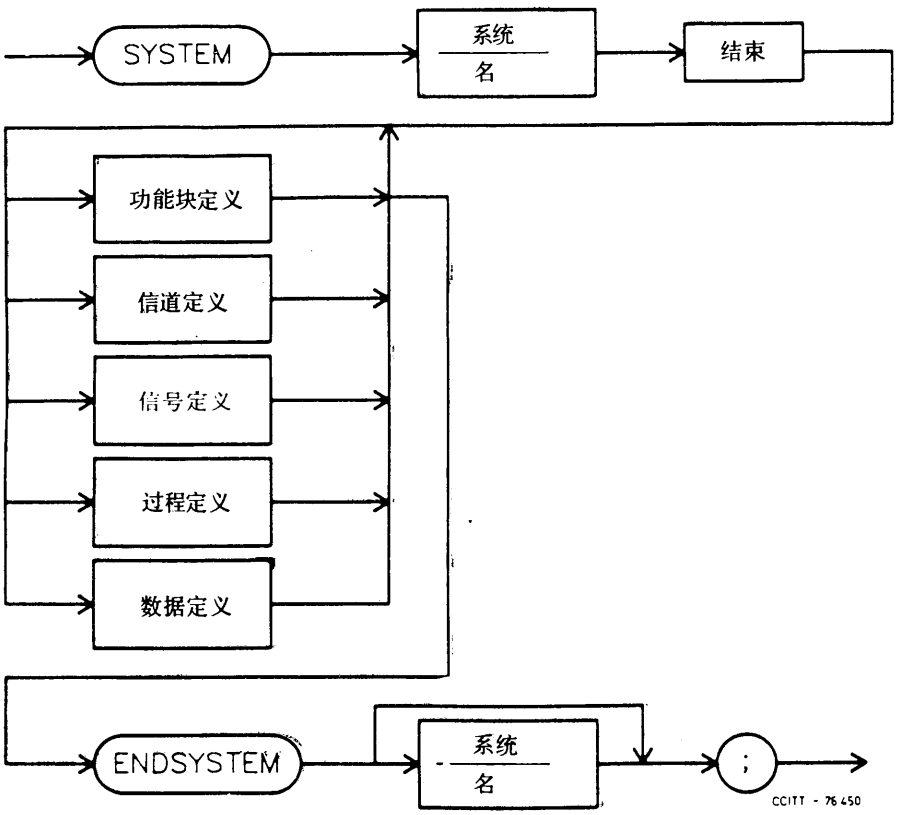
4 SDL/PR

4.1 语法补充

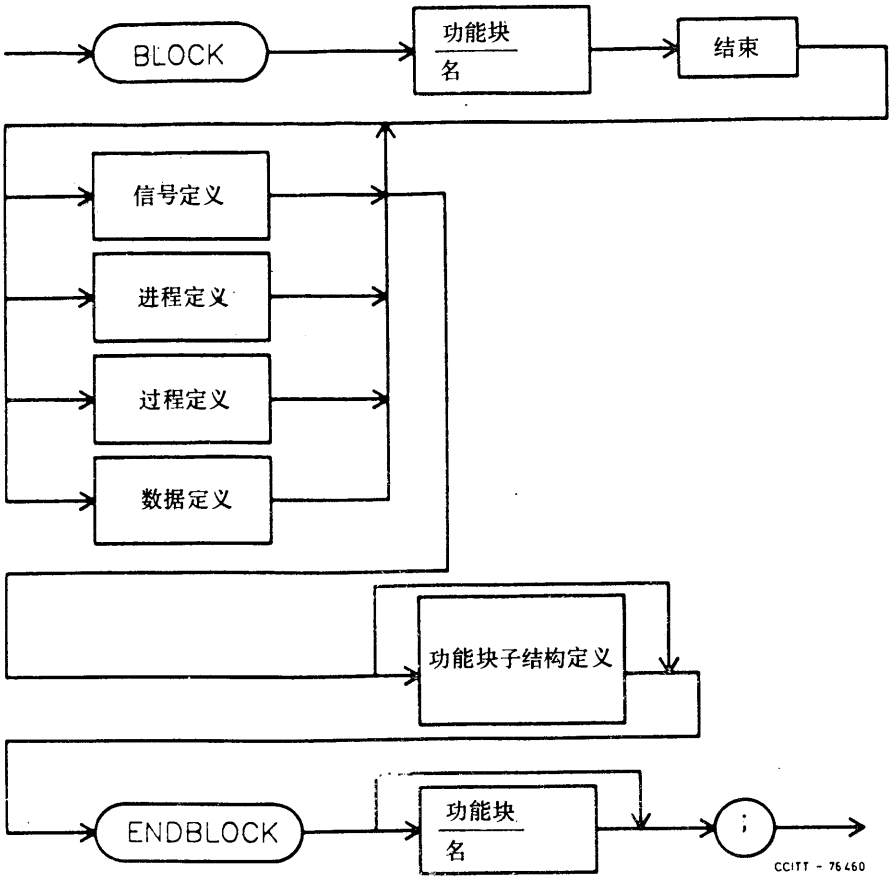
建议Z.101、Z.102和Z.103所定义的语法中增加了数据定义。

在系统定义、功能块定义、功能块子结构定义或信道子结构定义中，出现信号定义的地方也可以出现数据定义。在进程定义中，变量定义可出现的地方，和在过程定义中过程变量定义可出现的地方，数据定义也可以出现。

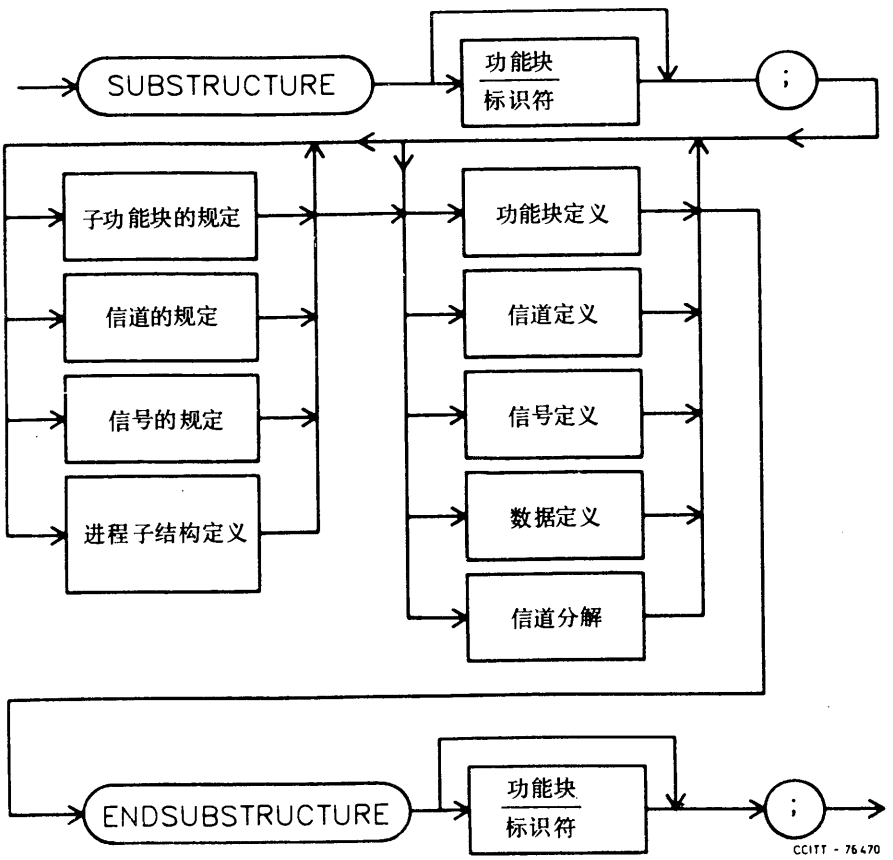
系统定义


SYSTEM
系统
名
结束
功能块定义
信道定义
信号定义
过程定义
数据定义
ENDSYSTEM
系统
名
;
CCITT - 76450

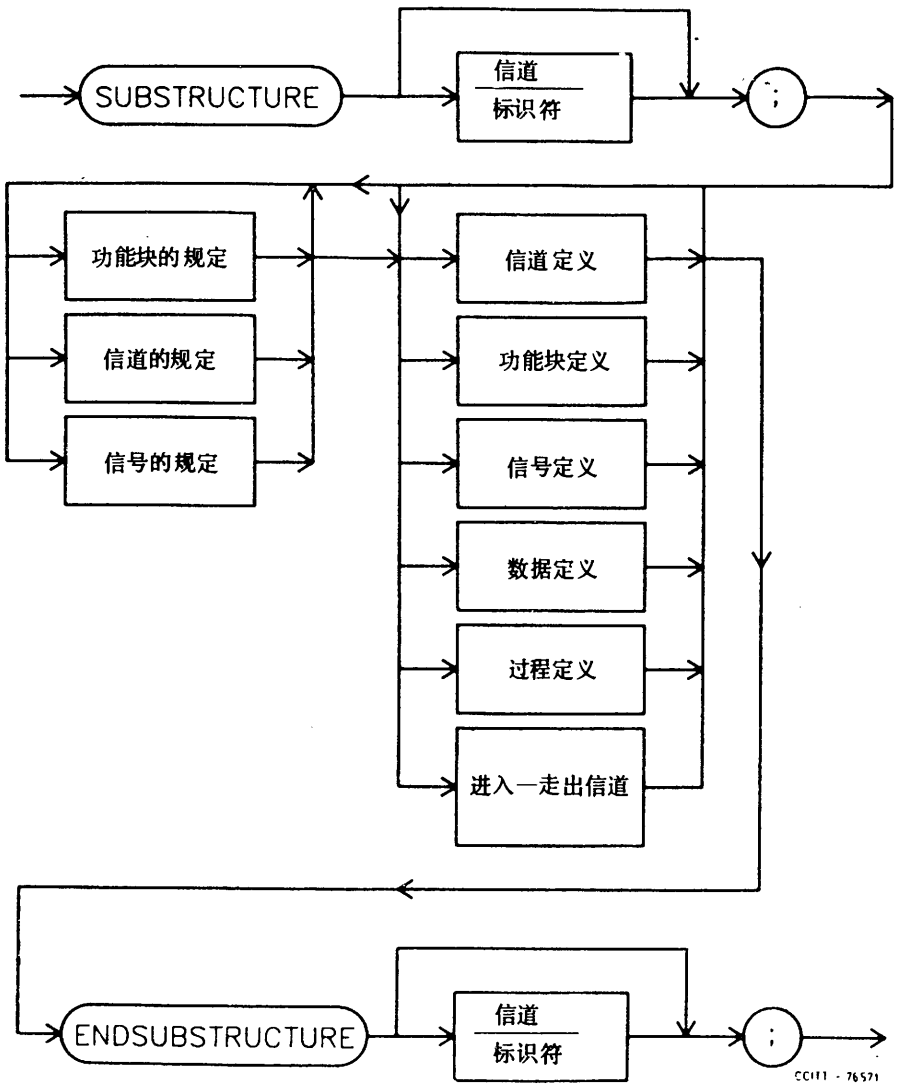
功能块定义

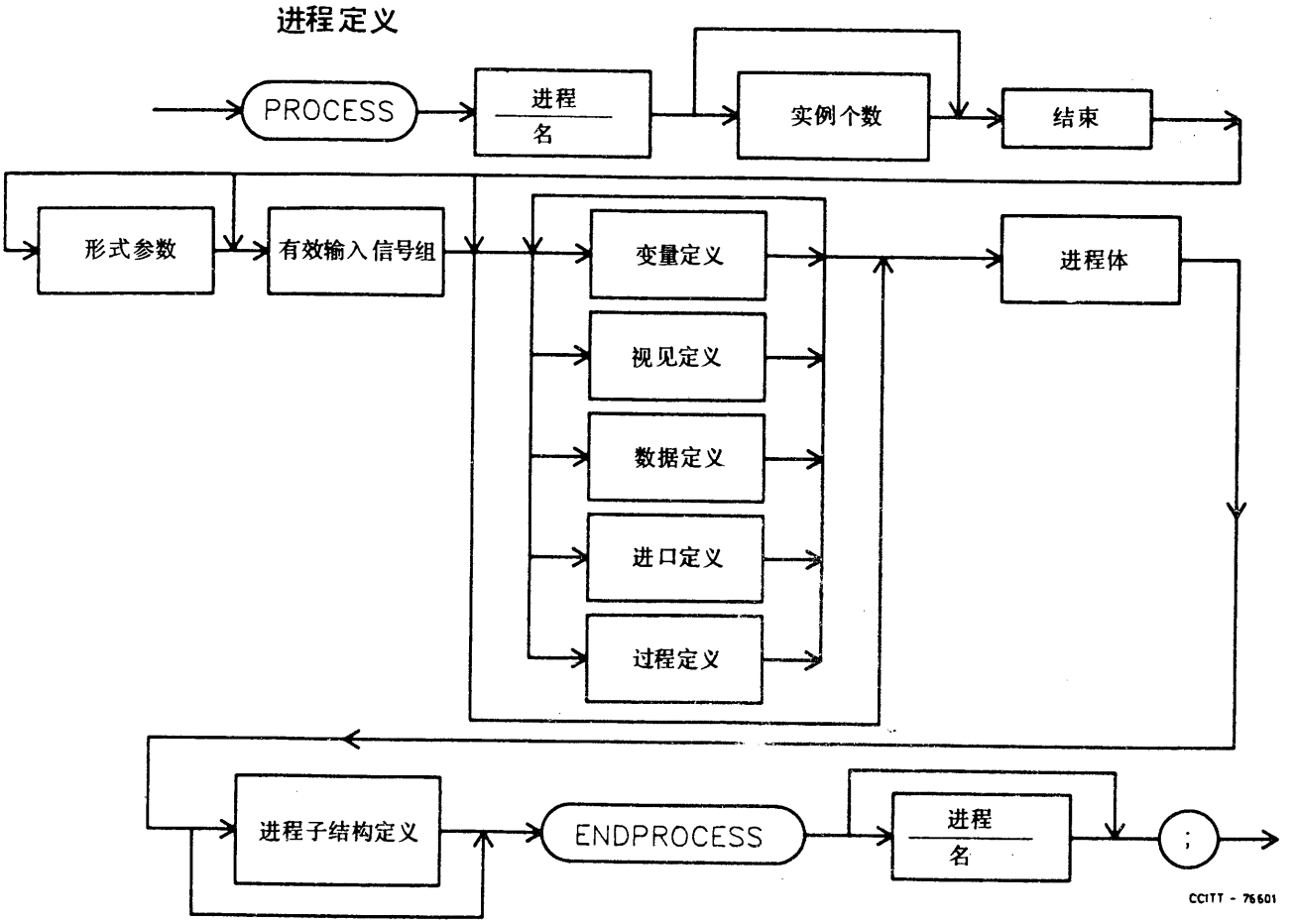


功能块子结构定义



信道子结构定义


SUBSTRUCTURE
信道
标识符
;
功能块的规定
信道的规定
信号的规定
信道定义
功能块定义
信号定义
数据定义
过程定义
进入—走出信道
ENDSUBSTRUCTURE
信道
标识符
;
CCITT - 76571

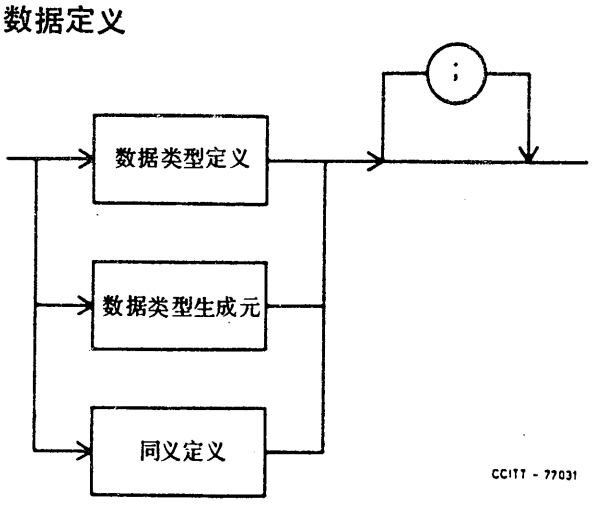


数据类型的语法经扩充可包括用户定义的数据类型标识符。

这里给出了赋值语句和表达式的语法。(注：建议Z.101中提到了表达式和赋值语句，但并未作出定义。)

4.2 数据定义

4.2.1 语法



4.2.2 语义

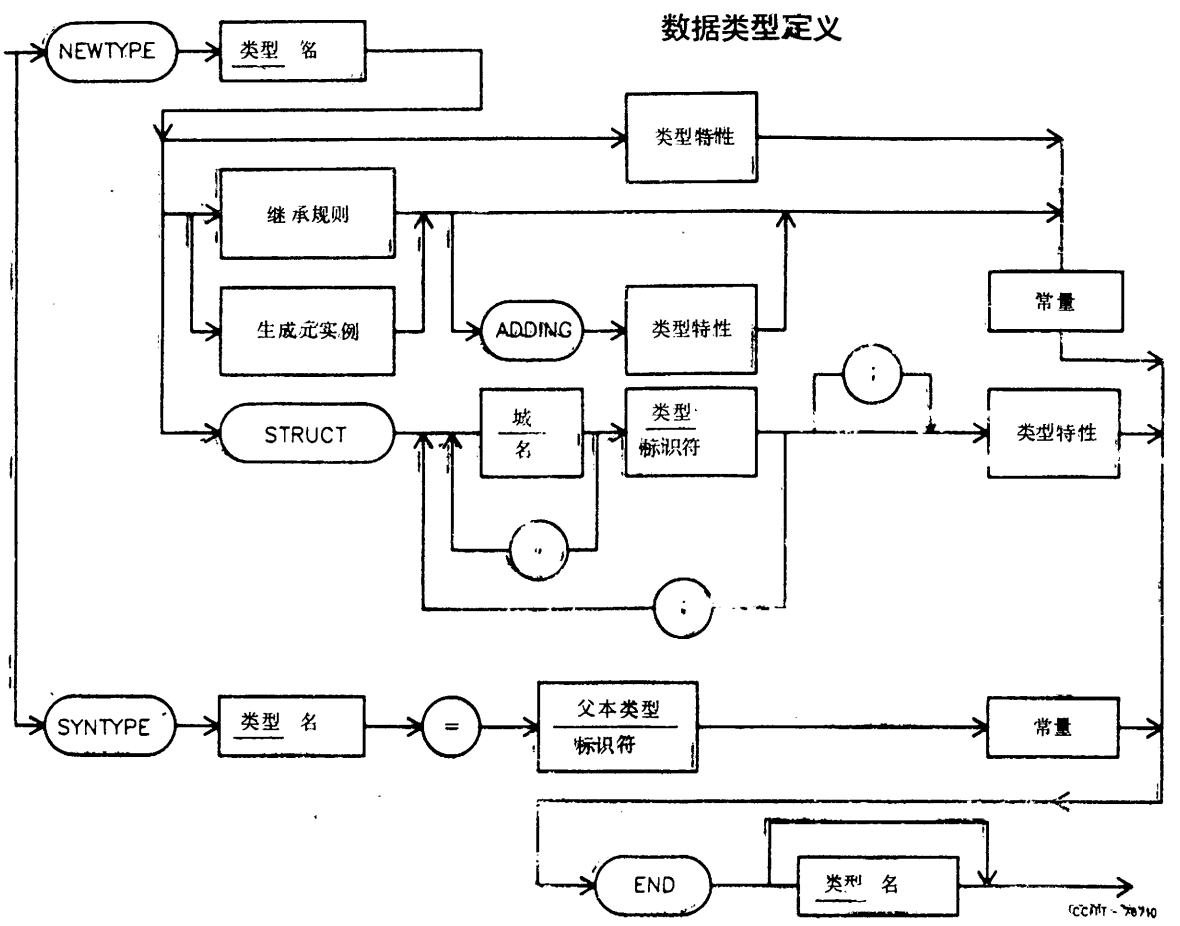
数据定义用来引入数据类型、数据类型生成元或同义字的名字和特性。

4.2.3 与SDL抽象语法的关系

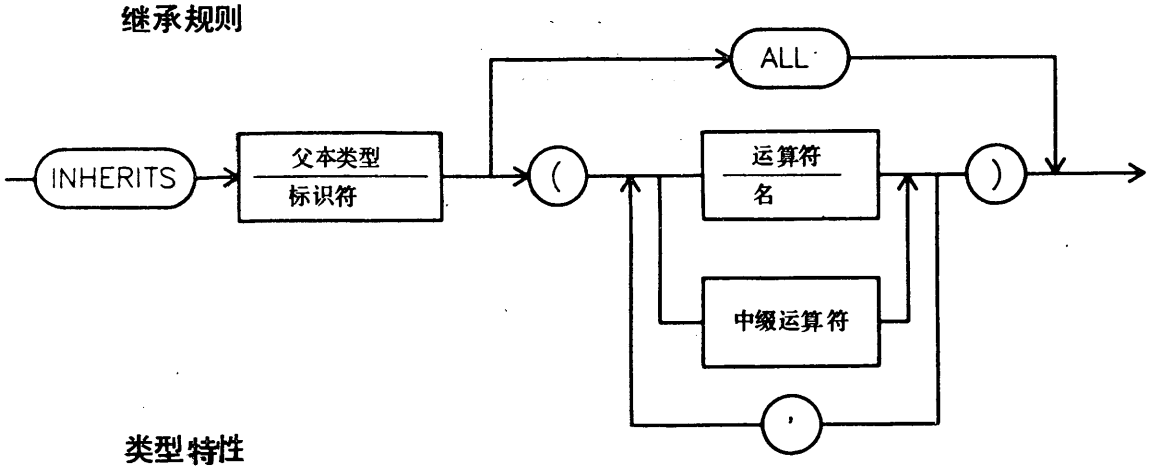
SDL/PR形式的数据定义表示抽象语法中的数据定义。如果数据定义是一个数据类型生成元的话，则与抽象语法没有直接的对应关系。由于数据类型生成元仅用来定义文句，该文句被认为是对生成元实例在文字上的扩展。

4.3 数据类型定义

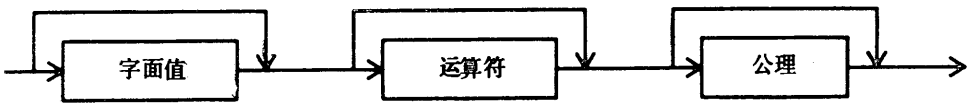
4.3.1 语法



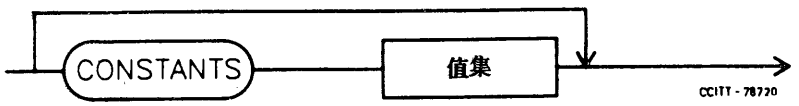
继承规则



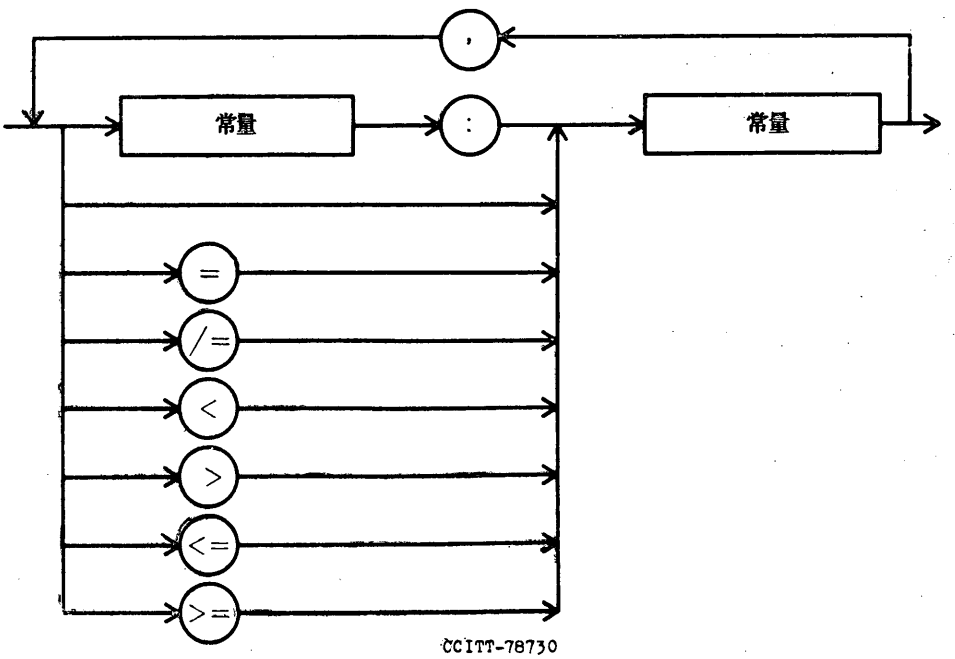
类型特性



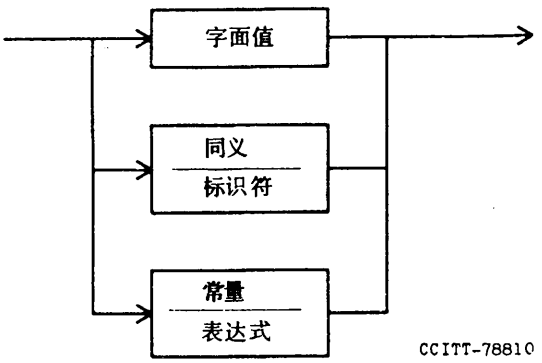
常量



值集



常量



4.3.2 语义

数据类型定义中给出的名字就是数据类型名。

4.3.2.1 NEWTYPE

NEWTYPE数据类型定义引入一个新数据类型。

如果不规定继承规则，则新数据类型不基于任何其他类型。该类型特性可用来引入该类型的任何字面值，引入可作用于该类型的运算符，以及通过叙述对该类型总是真的公理，引入（任选地）该类型的特性。

把NEWTYPE与继承规则结合使用，可以使一个新类型基于另一个类型。在此情况下，新数据类型的值集与父本类型的值集不相交。尽管新数据类型的值和运算符与父本数据类型的不同，该新类型的字面值及运算符名可以超载，也就是说它们可以与父本类型有相同的字面值和名字，并且，必须利用限定或上下文来决定一个字面值或名字是适合于新类型还是适合于父本数据类型。如果不能决定一个字面值或运算符名与一个类型的连结，那么此SDL规格说明就是二义的，因此也是非法的。当继承规则给出*ALL*时，则对于新数据类型，所有运算符名都是超载的。否则，继承规则中规定的运算符名必须是父本类型的运算符名，并为新类型定义这些名字。父本数据类型的公理集和字面值集仍被继承。

除了被继承的字面值、运算符名和公理以外，新数据类型还可以有在关键字ADDING后规定为类型特性的补充字面值、运算符名及公理。这些字面值、运算符名和公理绝不能与所继承的那些相冲突。

形式如下的数据定义：

```
NEWTYPE  X/*详细说明*/
CONSTANTS/*常量表*/
END  X;
```

等效于：

```
NEWTYPE  Anon/*详细说明*/
   END  Anon;
```

后接

```
SYNTYPE  X= Anon
   CONSTANTS/*常量表*/
   END  X;
```

对NEWTYPE使用常量限制就隐含地声明了一个无此限制的无名NEWTYPE（上面的Anon），然后用它作为带有此常量限制的SYNTYPE的父本。为了实施匿名，所确定的父本名字要与在该特定的隐含声明以外的SDL规格说明中的所有其他名字都不相同。

4.3.2.2 SYNTYPE

使用SYNTYPE，也可以定义一个数据类型，使之具有父本数据类型的值的一个子集。在此情况下，此值集或者在关键字CONSTANTS后规定，或者父本数据类型的所有值具有SYNTYPE中的相应值。带有SYNTYPE声明的变量只能赋以所规定的值。

访问一个带有SYNTYPE的变量产生父本数据类型的一个值。声明、赋值和访问的这些运算是SYNTYPE所允许的仅有的运算。

为SYNTYPE数据类型规定的值必须全都是该父本数据类型的值。一个同义类型的父本是同义类型定义中指定的第二个类型，只要该类型是一个新类型。否则，该父本就是所指定的类型的父本。

4.3.2.3 生成元实例

一个生成元实例等效于带有形式参数的生成元的文句，该形式参数在文字上由实在参数代替。在生成元的文句中使用生成元名的每个地方，都要用数据类型名或调用生成元实例的生成元代替。等效的文句必须完成一个有效的NEWTYPE数据类型定义。这个由文字展开而形成的数据类型定义将定义该数据类型名的特性。

4.3.2.4 结构

数据类型定义如果包含了一个结构就意味着对每个域名都有数据类型。对于一个给定的结构类型S，对于每个域名F_i和相应的类型标识符T_i，隐含地引入了下述公理（受增强的约束；见下面）：

— 单个公理：extract!(insert!(S, Fi, I)', Fi)= I

— 公理集：extract!(insert!(S, Fi, I)', Fj)
= Extract(S, Fj);
/*对所有不同的Fi, Fj*/

当在一个结构中有N个域名时，将有N*N个这种形式的公理被隐含地引入。为了保证无二义性，SDL要求在给定结构内的域名是唯一的。

与结构S有关的是一组类型，每个类型对应于一个域，每个类型带有类型名S!Fi、单个字面值Fi，而没有其他特性。这个类型就成为在Insert!和Extract!运算中使用的域名的载体。

一个结构定义的效果是创建（类似于编程语言）一个结构或记录，尽管结构定义可以包含补充的公理来增强该类型的行为。在由用户明显引入的补充公理与该结构的隐含的缺省公理集有冲突的地方，就舍弃隐含的公理来解决此矛盾。在一个结构中明显地引入公理需要特别小心。

4.3.3 与抽象语法的关系

数据类型定义表示抽象语法中的数据类型定义。类型名表示抽象语法中的类型名。

4.3.3.1 NEWTYPE

关键字NEWTYPE和END包围数据类型描述的抽象语法概念。抽象语法中的值名集、运算符指定和类型公理集表示如下：

a) 值名集

包括由类型特性中的字面值给定的字面值集合，如果规定了INHERITS的话还要包括父本数据类型的字面值集合。

b) 运算符指定集

包括由类型特性中的运算符给定的运算符集合，如果规定了INHERITS的话还要包括父本数据类型的运算符集合。

c) 类型公理集

包括由数据类型特性中的公理给定的公理集合，如果规定了INHERITS的话也要包括父本数据类型的公理集合。如果省略了公理集，或公理集不完备，那么，至少有一些运算只能非形式地解释。

4.3.3.2 SYNTYPE

关键字SYNTYPE和END包围同义类型描述的抽象语法概念。父本类型标识符表示抽象语法中的父本数据类型标识符。值集表示抽象语法中的值集。

4.3.3.3 生成元实例

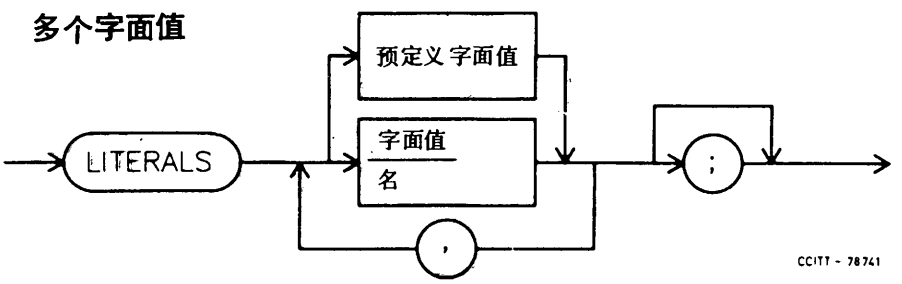
一个生成元实例表示（象在§ 4.3.2.3中所介绍的那样）通过在文字上扩展生成元而得到的文句。这样，它同抽象语法的关系与等效文句同抽象语法的关系相同。

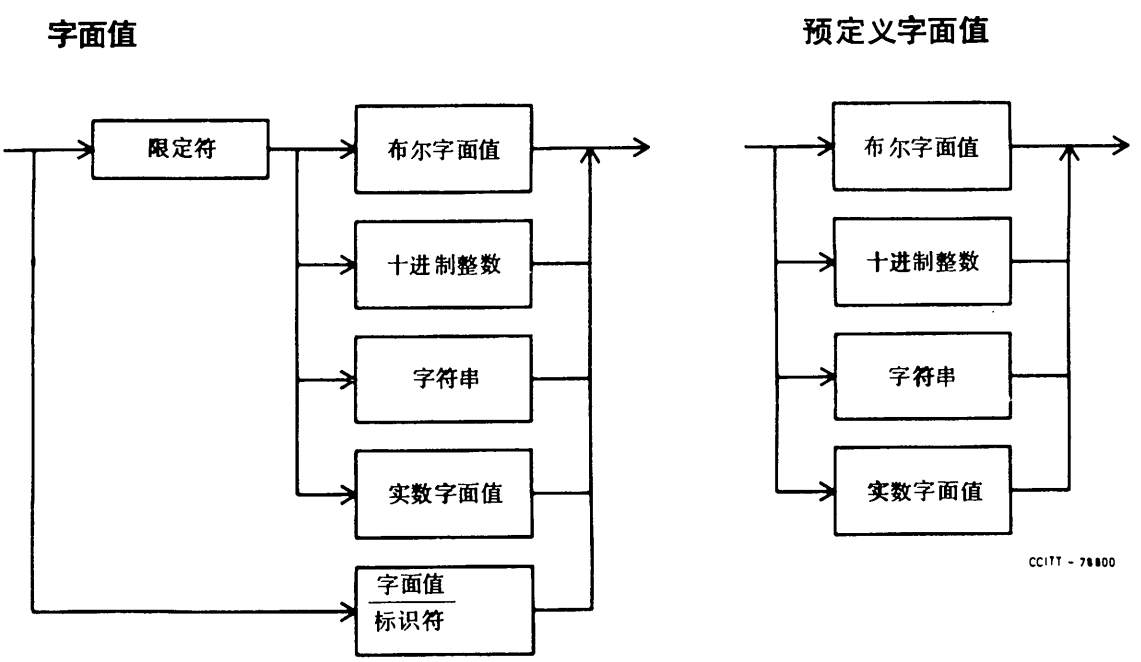
4.3.3.4 STRUCT

结构（Struct）表示通过明显地指定所有有关的特性而得到的文句，因此（象生成元那样），它同抽象语法的关系与此文句同抽象语法的关系相同。

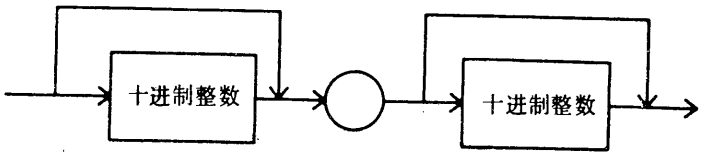
4.4 LITERALS

4.4.1 语法

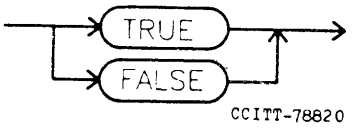




实数字面值



布尔字面值



4.4.2 语义

字面值被用来表示一个数据类型的值。字面值或者被预定义（对于预定义数据类型或基于预定义数据类型的数据类型），或者在关键字LITERALS后面指定一系列的数据类型的字面值来引入字面值。当一个类型包含Ordering!运算时，按约定这些字面值应该按上升的次序来指定。

4.4.3 与抽象语法的关系

由数据类型特性的字面值部分引入的字面值名，表示抽象语法中数据类型描述的值名。

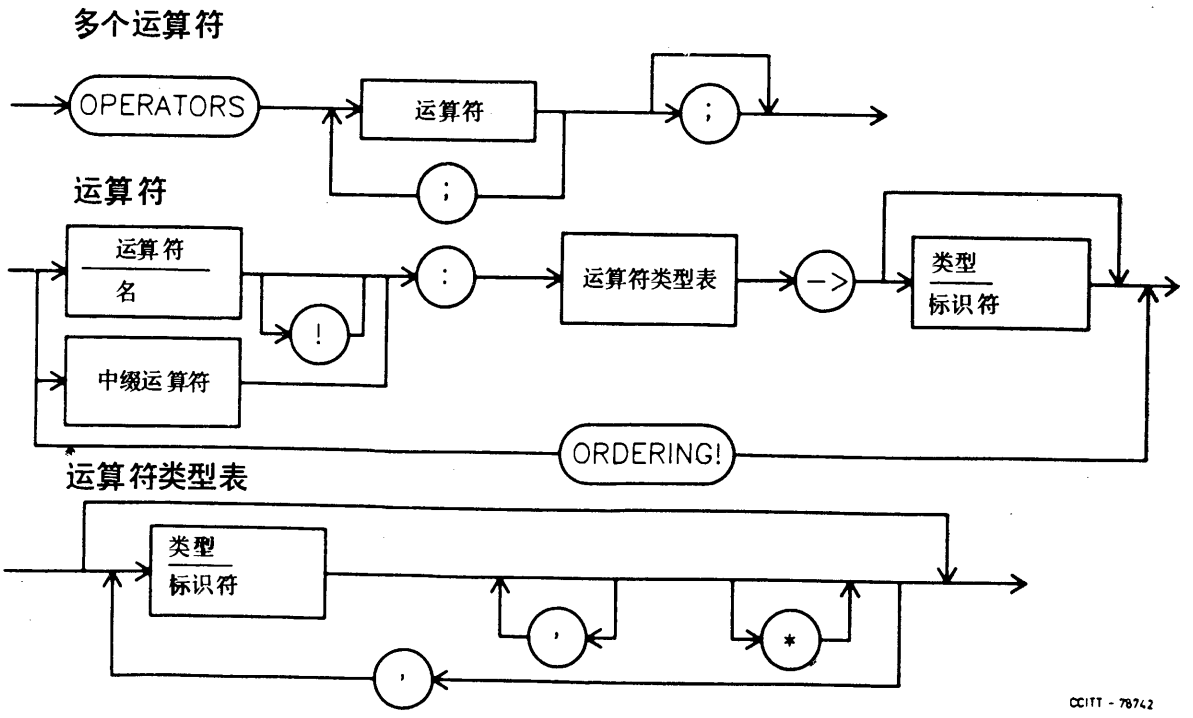
4.5 运算符

4.5.1 语法

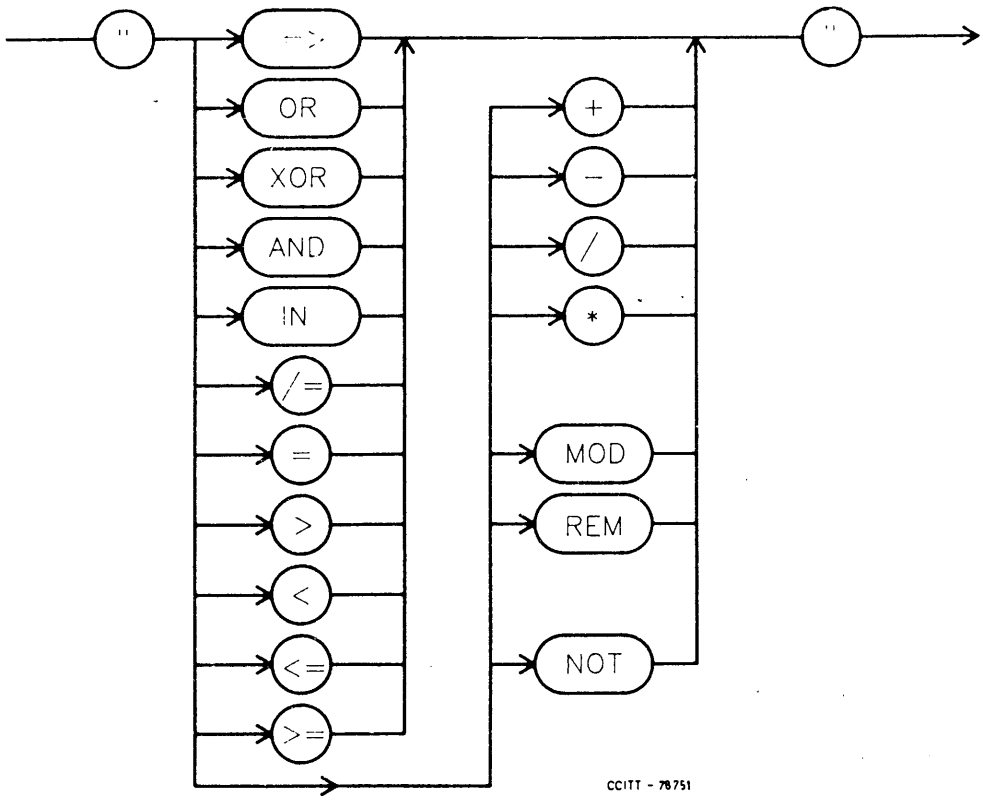
多个运算符

运算符

运算符类型表



中缀运算符



4.5.2 语义

数据类型特性的运算符引入运算符的名字和这些运算符的参数化方法。参数化方法决定了所需参数的个数和每个参数的数据类型，也决定了任何回送值的数据类型。

4.5.2.1 运算符名

通过在运算符中加入Ordering! 来规定排序运算符。这是一个简写符号，用来引入下列用于数据类型D的运算符。

“<”：D，D->Boolean

“>”：D，D->Boolean

“<=”：D，D->Boolean

“>=”：D，D->Boolean

在运算符中，中缀运算符的名字如+、AND、OR用引号括住。该带引号的形式可以用作为前缀运算符，即：

“+”(a, 2)

等效于a+2。

运算符后面可接（也可不接）一个惊叹号，它表示该运算符个体仅能在数据类型定义中被直接地引用。惊叹号构成了运算符名的一部分，因此使用该运算符时总是要写上惊叹号。

对于带有数据类型D的隐含定类的所有数据类型，名字Assign!、Declare!、Access!、“=”和“/=”隐含地定义为运算符。

Assign! : D′*, D −>
Declare! : D′*−>;
Access! : D′−>;
“=” : D, D −>Boolean;
“/=” : D′, D −>Boolean;

赋值运算符是中缀运算符“:=”。因为在声明中，声明运算符被隐含使用，所以没有声明运算符。无论什么时候在需要一个值的上下文中提及一个变量，都隐含了访问运算符。等于和不等于运算符分别是中缀运算符“=”和“/=”。

4.5.2.2 运算符定类

在冒号后和符号(−›) 前的数据类型标识符表称为运算符类型表。这个运算符类型表规定了该运算符所需值的数据类型。如果一个或多个数据类型标识符具有一个撇号属性，那么此运算符就是一个主动运算符，否则就是被动运算符。主动运算符能够改变与变量相关的值，而被动运算符纯粹是一种功能，因此不能改变与变量相关的值。

对于被动运算符，在类型表中的所有类型个体规定了该运算符的实在参数必须被解释为表达式。这些表达式中的每一个必须产生一个值，此值是数据类型表中相应位置的数据类型的值集的一员。

对于被动运算符，在符号(−›) 后必须有一个数据类型标识符。这个数据类型标识符规定：该运算符在作用时会产生此数据类型的一个值。对于主动运算符，在运算符类型表中的某些数据类型个体后面跟以撇号(')。

一个撇号表示：该运算符需要给定数据类型的一个变量作为参数，并且与此变量相关的值可以改变。在输入数据类型表中，两个数据类型标识符后面不能跟相同数量的撇号。

当一个主动运算用打撇号的办法来表示与一个变量相关的值时（此变量作为参数给出），则撇号的个数可使带撇号的参数互相区别开来（这只在公理中许可）。例如：

```
OPERATORS SwapAndAdd: Int′, Int″−›int
        /*数据类型定义中的一部分*/
AXIOM SwapAndAdd (a, b)′ =b;
        /*该公理叙述第一个参数接收第二个参数的值*/
SwapAndAdd (a, b)″ =a;
        /*第二个参数的公理*/
SwapAndAdd (a, b) =a+b;
        /*结果公理*/
```

如果一个带撇号的参数后面跟一个星号，那么，当使用此运算时就不访问该变量的初始值。(注意公理必须与此一致，否则该SDL规格就是非法的。)

对于主动运算符，如果省略符号(−›) 后的数据类型标识符，则在表达式内不能使用该运算符。在一个输入数据类型表中，可以既有带撇号参数又有无撇号参数。

在一个名字串中，后接带引号的字符串的数据类型名与数据类型表中一个带撇号的数据类型标识符之间，存在着语法二义性。在类型表中，当类型名后的一个撇号后面跟一个撇号、逗号或负号(−›中的一部分)时，就会产生这些二义性。在所有情况下，撇号被看作为数据类型标识符的撇号，而不是一个带引号字符串的开始。

4.5.2.3 Insert! 和Extract! 运算符

为了允许对数组和结构作出公理的定义，有两个预定义运算符名，它们具有数据类型定义以外的专门的符号。这两个运算符就是Insert! 和Extract! 。Insert! 是一个主动运算符，它必须与数据类型个体一起定义，以使第一个数据类型是带撇号的，而其他类型是无撇号的。

为了在数据类型定义外运用Insert!，先写第一个参数（必须是一个变量），后面跟左括弧、再跟除最后一个参数以外的其余参数，然后是右括弧和“:=”，末尾是最后一个参数。

这样，设A是一个变量，其数据类型可用已定义的Insert!，且对于这个数据类型i1、i2及e 是适合于Insert! 的表达式。

Insert! (A, i1, i2, e)

就写为

A(i1, i2) :=e

因为能够为不止一个的数据类型定义Insert!，所以要从变量的类型来决定合适的Insert!。Insert! 必须与至少两个参数一起加以定义，并且必须回送与第一个参数相同的数据类型。

Extract! 是一个被动运算符。出于语义原因，它要求一个变量作为第一个参数。要运用Extract! 时，应先写出第一个参数，后面跟以括弧内的所有其他参数。

象这样

Extract! (A, i1, i2)

写成为

A(i1, i2)

Extract! 的使用由变量的类型决定。

4.5.3 与抽象语法的关系

对于被动运算符，一个运算符名或中缀运算符表示抽象语法中的一个运算符名。对于被动运算符，输入类型表表示抽象语法中参数的类型个体表。符号(->)后面的数据类型标识符表示运用运算符的结果的数据类型个体。

通过重新写入到被动运算符中，把主动运算符与抽象语法联系起来。这也就定义了运算的先后次序的行为。对于每个带撇号的参数，都有一个隐含的被动运算符，它回送公理集所需的值，对于每个不带星号的带撇号参数，在使用该运算符的每个进程实例中都有一个隐含变量，它与接收该参数的初始值的参数有相同的数据类型。

对于每个隐含被动运算符，抽象语法中参数的数据类型个体表用输入数据类型表来表示，忽略任何带星号的参数。隐含被动运算符与输入数据类型表中的带撇参数一样多，且每个带撇参数数据类型用作为数据类型个体，而它是运用这些运算符中的其中一个的结果。例如：

```
OPERATORS
    complex: Integer', Integer'', Integer, Integer''' *-> Bool
        /*交换最前两个参数，把最前三个参数的和放入第四个参数，并且，如果第二个参数和第三个
        参数相等，则回送真*/
AXIOMS
    complex (a,b,c,d)' = b;
    complex (a,b,c,d)'' = a;
    complex (a,b,c,d)''' = a + b + c;
    complex (a,b,c,d) · = (b + c);
        /*参见§ 4.9.2公理中撇号的使用*/
```

隐含的运算符是：

implied1!: Integer, Integer, Integer -> Integer

implied2!: Integer, Integer, Integer -> Integer

implied3!: Integer, Integer, Integer -> Integer

implied4!: Integer, Integer, Integer -> Boolean

如果隐含变量是v1和v2，则在一个语句中应用 complex就等效于在：

V1:=a; V2:=b;

后面接此语句，但是其中V1代替a，V2代替 b．以及implied4!代替complex，后面再接：

a:=implied1! (V1,V2,c);

b:=implied2! (V1,V2,c);

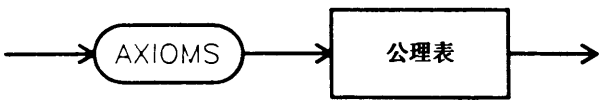
d:=implied3! (V1,V2,c);

在一个语句中出现了不止一个的主动运算符时，应按解释它们的顺序替换它们。

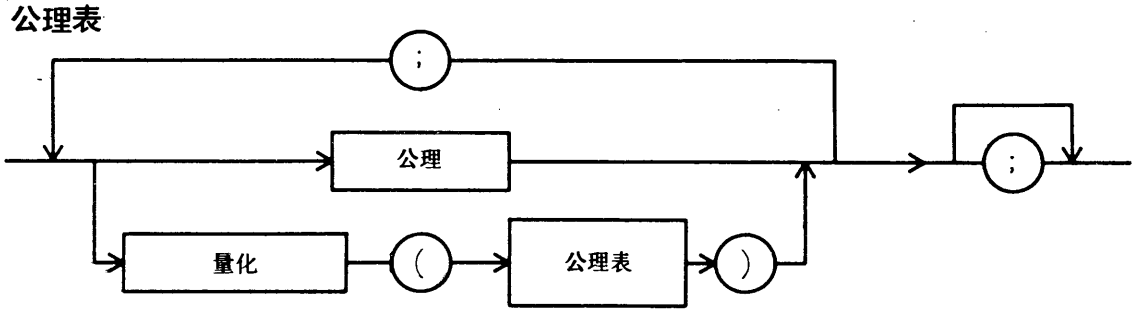
4.6 公理

4.6.1 语法

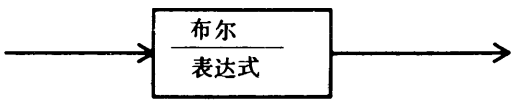
多个公理



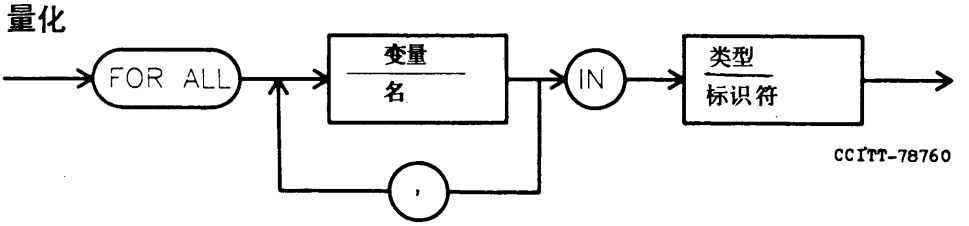
公理表



公理



量化



4.6.2 语义

公理是一组布尔表达式，它对于公理中的变量的所有值都保持真。

在一个公理内，一个“变量”决不是指一个进程或过程变量的名字，进程或过程变量有一个与它自己相关联的值，但是公理内的变量用来表示：一个特定类型的所有值可以替换为该变量而此公理仍为真。当考虑这个替换情况时，一个给定的变量名总是表示一个公理中的同一个值。例如在：

```
OPERATORS  even:Integer ->Boolean
AXIOMS  even(0)=True;                /*公理1*/
        even(1)=NOT even(i+1);/*公理2*/
```

对 i = 2、i = 3、i = 4 等等，公理 2 必须成立，即：

```
even(2)=NOT  even(2+1)
even(3)=NOT  even(3+1)
even(4)=NOT  even(4+1)
```

通常，公理中一个变量的数据类型可以由上下文决定，例如在上面的例子中，运算符语法要求此变量的数据类型为int。

有时，由于SDL中名字和符号（如“+”)的超载，不可能由上下文决定一个公理变量的数据类型，故此需要任选的量化。量化强制某变量具有某具体的类型。如果在一个公理内某一变量的使用具有二义性或者是矛盾的，则此SDL/PR就是无效的。

量化也允许一个变量名表示多个公理中的同一个替换。

为公理中的变量所选的变量名必须同适合于上下文（使用该变量的上下文）的字面值不同。

因为对于所有数据类型都隐含了运算符“=”和“/=”,所以总是隐含有下述公理：

“/=”(a,b)=NOT (“=”(a,b))

“=”(a,a);

“=”(a,b)AND “=”(b,c)=>“=”(a,c);

“=”(a,b)=>“=”(b,a);

无论什么时候规定了Ordering!，都隐含有下列公理：

“<”(a,b)=>NOT “>”(a,b);

“>”(a,b)=>NOT “<”(a,b);

“<”(a,b)AND “<”(b,c)=>“<”(a,c);

NOT “<”(a,a);

“<=”(a,b)=>“<”(a,b)OR “=”(a,b)

“>=”(a,b)=>“>”(a,b)OR “=”(a,b)

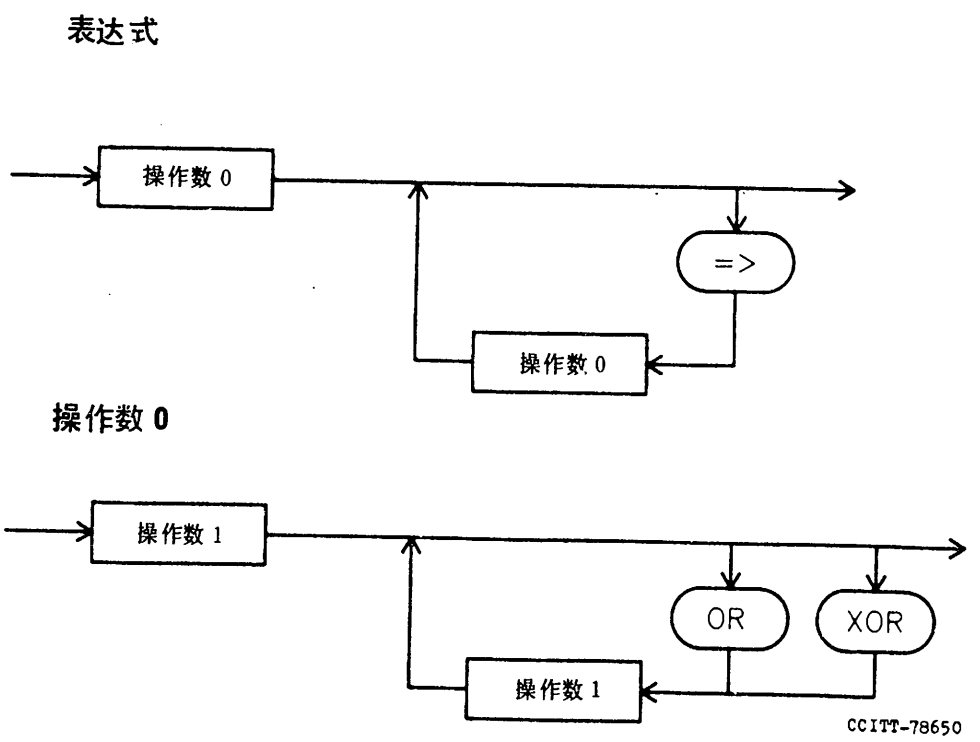
4.6.3 与抽象语法的关系

多个公理表示抽象语法中的数据类型的多个公理。每个公理表示抽象语法中的一个公理。

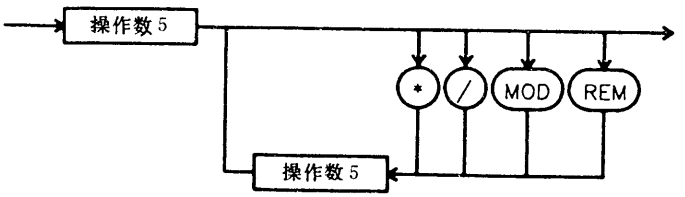
量化表示抽象语法中的量化，除非对于其数据类型由上下文决定的所有公理变量，具体语法中有隐含的量化。

4.7 表达式

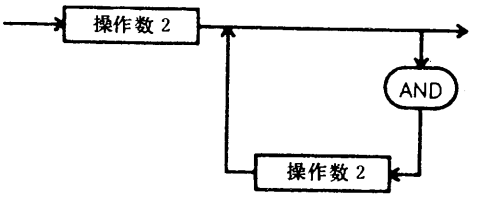
4.7.1 语法



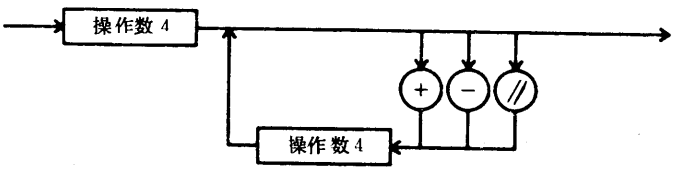
操作数 4



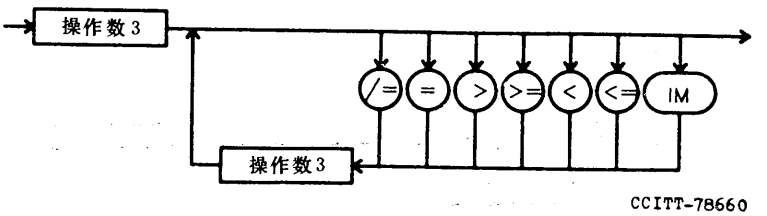
操作数 1



操作数 3

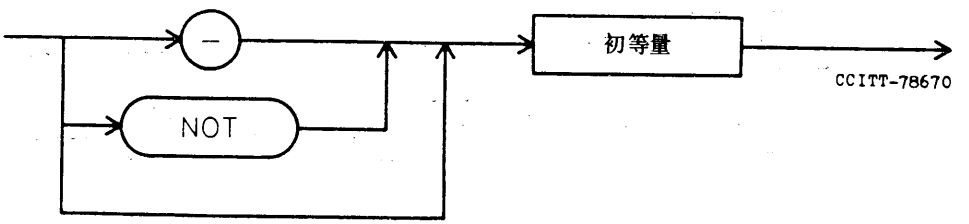


操作数 2



CCITT-78660

操作数 5



CCITT-78670

4.7.2 语义

表达式或者是一个初等量，或者是若干“中缀”运算符的应用。

运算符的作用顺序由语法中它们出现的情况来决定，这与编程语言（如CHILL，见建议Z.200）的方法类似。但是，SDL也允许布尔隐含运算符（=>），它的优先级比任何其他运算符低。如果左操作数为TRUE并且右操作数为FALSE，则它的值为FALSE。否则，对于布尔操作数，此隐含运算的值为TRUE。

通常，所有的运算符都与编程语言中所定义的具有相同的特性和有效性，但应该注意，在SDL中，用户可以通过把这些运算符包括在数据类型定义内，而为它们规定新的意义。然而，“中缀”运算符的优先级不能被改变。

一个有效运算所产生的值，由数据类型定义中的公理决定。

4.7.3 与抽象语法的关系

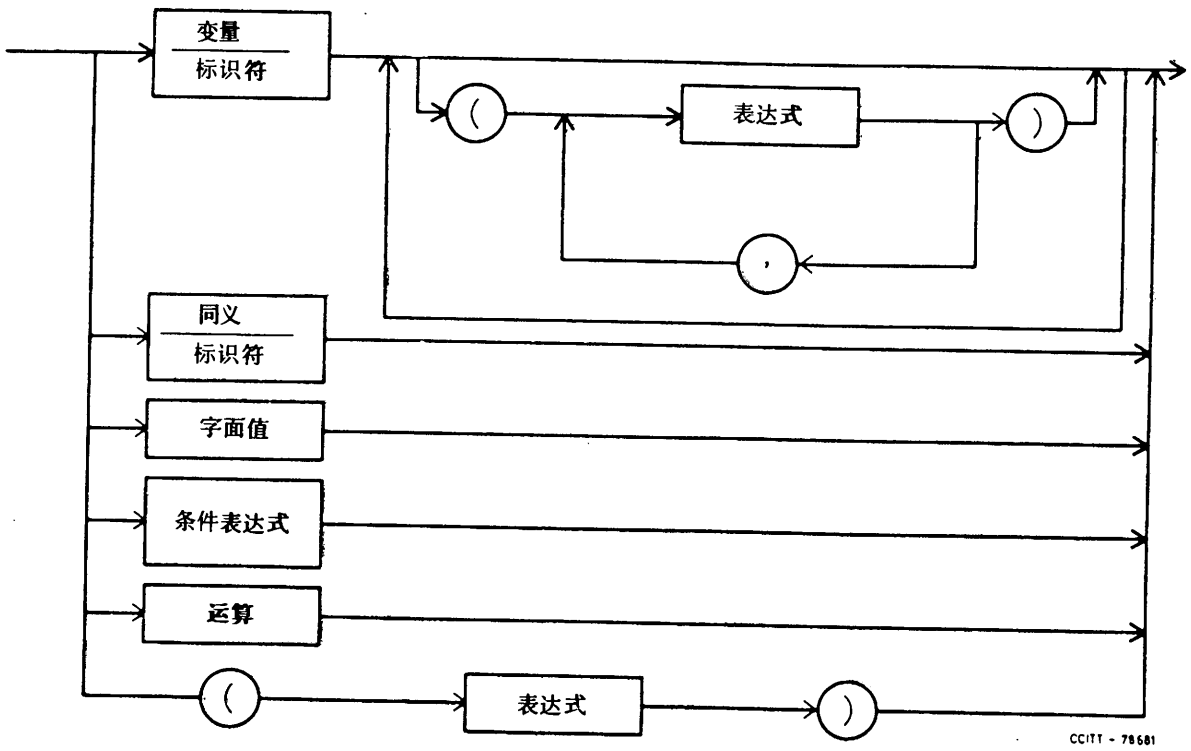
表达式表示抽象语法中的表达式。

“中缀”运算符能够超载，并可表示抽象语法中若干运算符中的任何一个。解决超载有两种方法：一种是，由参数的个数和类型解决，另一种是，在参数本身超载的情况下，由使用该运算的上下文中所需的数据类型解决。

4.8 初等量

4.8.1 语法

初等量



4.8.2 语义

初等量或者是一个变量个体，或者是一个字面值、一个同义个体、一个条件表达式、一个运算或者一个带括号的表达式。

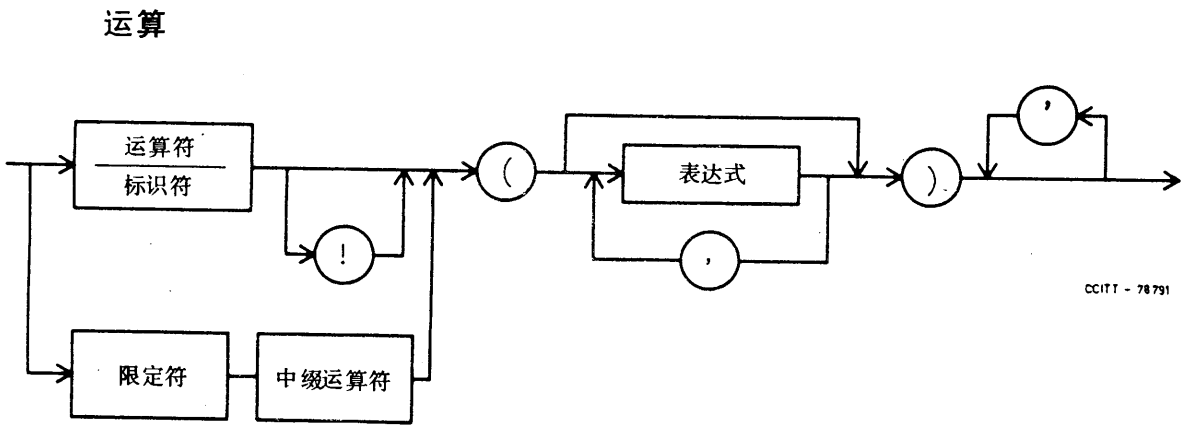
4.8.3 与抽象语法的关系

变量个体、字面值、同义个体、条件表达式、或带括号的表达式分别表示抽象语法中的变量个体访问、值个体、同义个体、条件表达式或表达式。

当一个公理中提到一个变量个体时，它表示一个公理变量，而不是指对进程或过程中所声明的一个变量的访问。公理变量的数据类型由上下文决定。

4.9 运算

4.9.1 语法



4.9.2 语义

运算就是在类型定义中定义了的运算符的应用。用作为实在参数的表达式的个数和类型必须与运算符个体的定义相一致。如果运算符名超载，则这些参数可以用来决定要运用的是哪一个运算符。

如果，在一个公理中要运用运算符，那么，当名字与一个惊叹号一起定义时，则在该公理中此惊叹号必须重复。

与一个惊叹号一起定义的运算符不能在类型定义外使用。

运算符的右括号后面的几个撇号只能用在公理中，用来表示该运算具有与撇号个数对应的参数的结果值。例如：

```
OPERATORS
    example: t1', t2'' -> t
AXIOMS
    example: (v1, v2)'' == v1
    /* v1的值放入v2 */
```

当运算符的参数的定类与撇号一起规定时，除了在公理的上下文内以外，实在参数必须是一个变量。

4.9.3 与抽象语法的关系

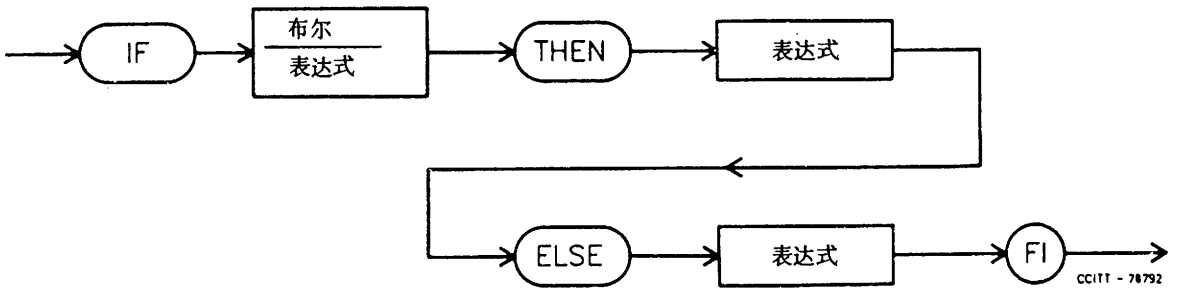
被动运算符标识符表示抽象语法中的一个运算符。对于被动运算，表达式表就表示抽象语法中对应于该运算的表达式表。

主动运算表示给隐含变量赋值，这些变量然后用作为隐含被动运算的变元，它们的值回过来又赋给作为实在参数而给出的变量（见§4.5.3）。在一个公理内，主动运算表示运用合适的隐含被动运算，此被动运算由附在该运算上的撇号的个数来决定。

4.10　条件表达式

4.10.1　语法

条件表达式



4.10.2　语义

对条件表达式的解释是这样的：如果布尔表达式为真，则对THEN后的表达式进行解释，否则，就对ELSE后的表达式求值。

在一个公理内，条件表达式的每个支路只须对选择该支路的条件有效。例如，由于对负数log(x)无定义，在下式中

IF　r>0 THEN　log(r)　ELSE　0.0　FI

如果r<=0，则log(r)无定义，这一点也没有关系。

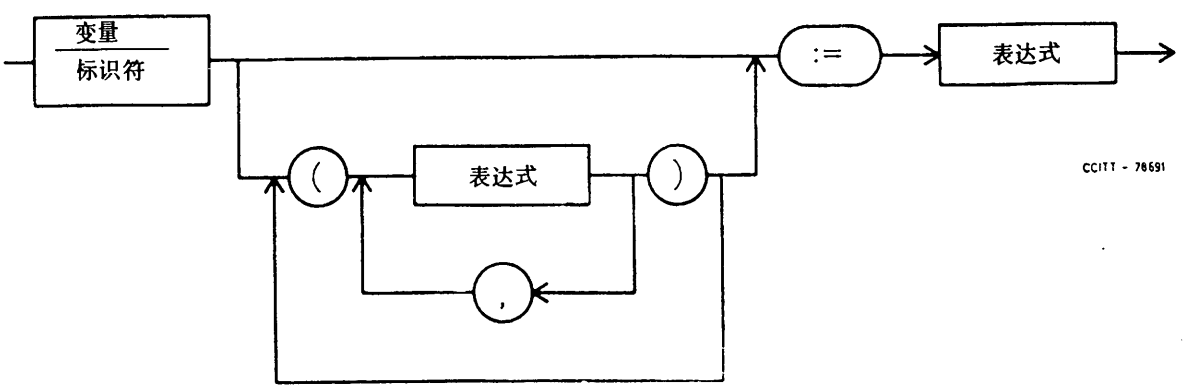
4.10.3　与抽象语法的关系

条件表达式表示抽象语法中的条件表达式。

4.11　赋值语句

4.11.1　语法

赋值语句



4.11.2　语义

赋值语句使一个值与一个变量相关联。

由右边表达式产生的值赋给左边的变量或变量的一个元素。

4.11.3　与抽象语法的关系

赋值语句表示运算符Assign! 的应用，或作为运算符Insert! 的应用。

赋值的语法与合适的运算符的应用之间有一个匹配关系。

如果在赋值的左边没用括号，则该赋值表示使用Assign!，因此

V:= e表示 Assign! (V, e)

当只使用一对括号时，这样的赋值表示Insert!，使得

a(i):= e表示 Insert! (a, i, e)

和

a(i, j):= e表示 Insert! (a, i, j, e)

当使用多个括号时，表示赋值是递归替换的，因此

a(i)(j):= e

表示

vi:= a(i);

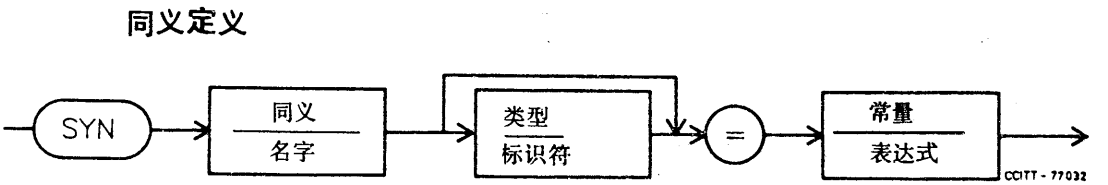
Insert! (vi, j, e);

a(i):= vi;

其中vi是一个隐含变量，其类型与a(i)相同。隐含的Insert! 运算是主动运算，在抽象语法中按对主动运算符的正常方法来表示它（见§4.9.3）。

4.12　同义定义

4.12.1　语法



4.12.2　语义

同义字等效于常量表达式。如果表达式的类型不能由常量或同义定义的上下文决定，那么必须规定一个类型，否则，常量表达式的值和类型（从而同义字的值）要由给出同义定义的上下文来决定。

4.12.3　与抽象语法的关系

同义字表示抽象语法中的同义定义。如果省略了数据类型，那么就由上下文隐含此数据类型。

4.13　数据类型生成元

4.13.1　语法

4.13.2　语义

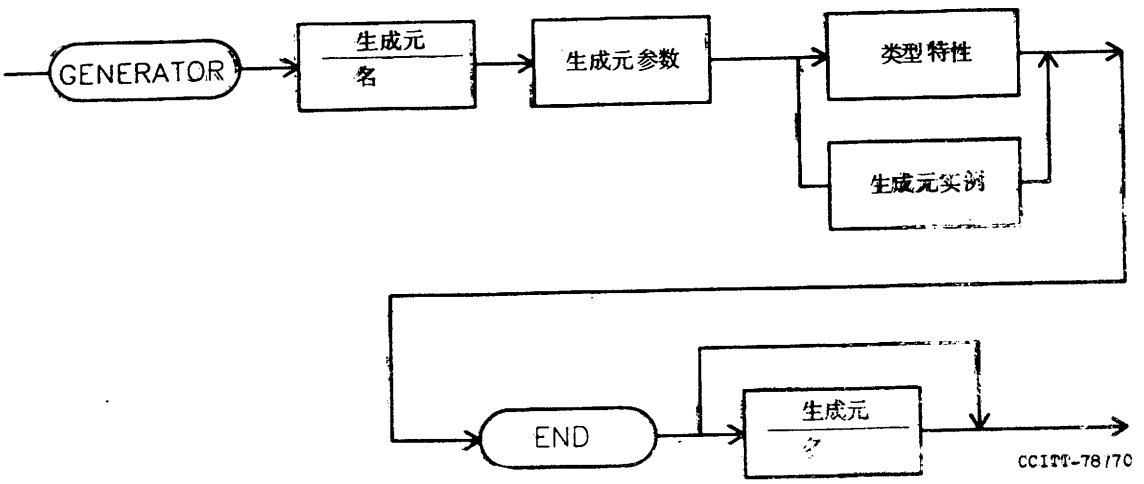
数据类型生成元中给定的名字是数据类型生成元名。生成元中所提供的特性构成数据类型的部分规格。当在一个类型定义中或在另一个数据类型生成元中使用某个生成元时，则在该生成元定义中要在文字上替代关于该生成元的参数，并且（连同在类型定义中所加入的任何特性一起）由此必须构成一个完整的类型定义或构成另一个数据类型生成元。在用一个生成元实例来定义一个数据类型生成元时，生成元参数可以与提供给具体实例的那些参数具有相同的名字。这样的一个生成元是一个部分实例。例如：

```
GENERATOR Stack(TYPE Component, CONSTANT Maxsize)
    /*详细说明*/
    END Stack;
GENERATOR IStack(CONSTANT Max) Stack(Integer, Max)
    END IStack;
    /*由整数组成的栈，不规定最大长度*/
```

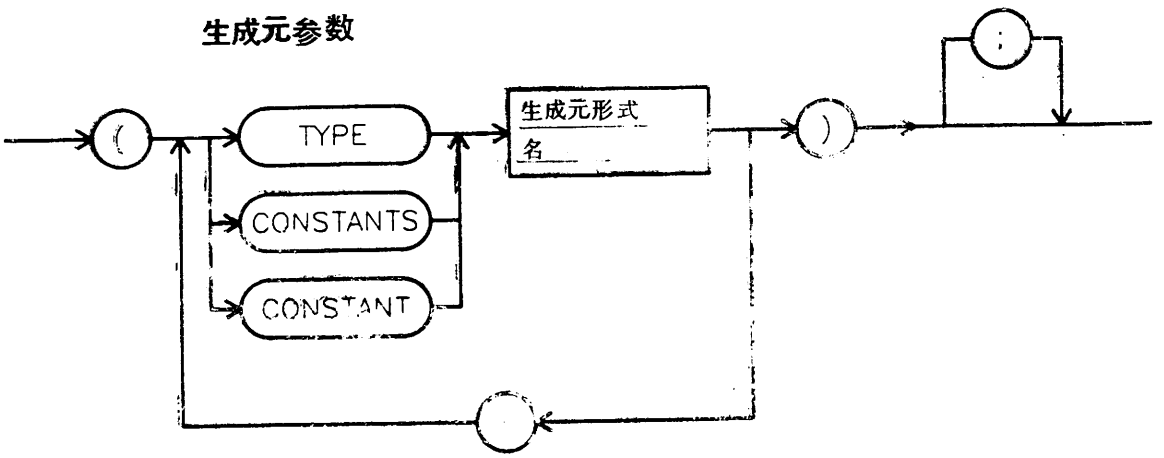
4.13.3 与抽象语法的关系

数据类型生成元在抽象语法中没有对应物。在数据类型定义中生成元实例的使用将表示由参数替代而形成的文句。

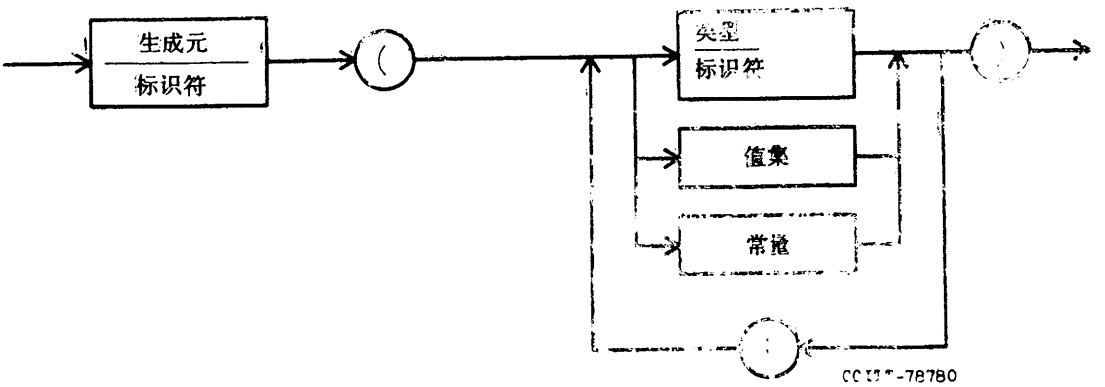
数据类型生成元



生成元参数



生成元实例



5 预定义数据类型

5.1 *Integer*（整数）

```
NEWTYPE Integer
        /* 按照整数字面值语法的字面值* /

    OPERATORS
        "+": Integer, Integer -> Integer;
        "-": Integer, Integer -> Integer;
        "-": Integer, Integer -> Integer;
        "/" Integer, Integer -> Integer;
        Float: Integer -> Real;
        Fix: Real -> Integer;

    AXIOMS
        /* 继承数学的整数性质，再加上：* /
        Fix(Float(r)) = r;
        r - 1 < Float(Fix(r)) < = r;
        i/j = Fix(Float(i)/Float(j))

    END Integer;
```

5.2 *Real*（实数）

```
NEWTYPE Real
        /* 具有在'实字面值'中规定的形式的字面值*/

    OPERATORS
        "-": Real, Real -> Real;
        "+": Real, Real -> Real;
        "*": Real, Real -> Real;
        "/": Real, Real -> Real;
        "-": Real -> Real;

        /* 继承数学的实数公理；这里,此工作需要进一步研究以便作出更形式化的规定* /

    END Real;
```

5.3 *Array*（数组）

```
GENERATOR Array(TypeIndex, TYPEItem);

    OPERATORS
        Insert!: Array', Index, Item -> ;
        Extract!: Array, Index -> Item;

    AXIOMS
        Extract! (Declare!(V)', I) = Error!;
        Extract!(Insert!(A, IPos, It), EPos =
          If Epos = Ipos Then It Else Extract(A, EPos) FI;

    End Array;
```

5.4 *Boolean*（布尔）

```
NEWTYPE Boolean;
    LITERALS True, False;
    OPERATORS
        "NOT": Boolean -' Boolean;
        "AND": Boolean, Boolean - > Boolean;
        "OR": Boolean, Boolean - > Boolean;
        "=": Boolean, Boolean - > Boolean;
AXIOMS
    "NOT"(True) = False;
    "NOT"(False) = True;
    "AND" (A, B) = If A = False Then False Else B;
    "OR"(A, B) = If A Then True Else B;
    "= >"(A, B) = If A = True And B = False Then False Else True;
END Boolean;
```

5.5 *Character*（字符）

```
NEWTYPE Character
    /* 长度为1的字符串，其中的字符是CCITT 第5号字母表中的那些字符*/
    OPERATORS
        Ordering!;
    END Character;
```

5.6 *Natural*（自然数）

```
SYNTYPE Natural Integer Constants > = 0 END Natural;
```

5.7 *Powerset*（幂集）

```
GENERATOR Powerset(TYPE Item);
    LITERALS Empty;
    OPERATORS
        "IN": Item, Powerset - > Boolean;
        Incl: Item, Powerset' - >;
        Del: Item, Powerset' - >;
        Ordering!;
    AXIOMS;
        Declare(v)' = Empty;
        I IN Empty = False;
        I IN Incl(I2, S) = IF I = I2 THEN True ELSE I IN S FI;
        Del(I, Incl(I2, S)) = IF I = I2 THEN Del(I,S) ELSE Incl(I2, Del(I,S)) FI;
        For all S1, S2 in powerset (for all I in Item (S1 < S2 = > (I IN S1 = > I IN S2))
        NOT(I in S) = > Del(I, S) = S;
    END Powerset;
```

5.8 *Pid*

```
NEWTYPE PId
    LITERALS Null;
    OPERATORS
        Create! : -> PId;
    AXIOMS;
        Declare!(v)' = Null;
        Create! /= Create!;
        /* 叙述所有创建都产生不同的唯一值的一种软方法*/
        Create! /= Null;

        /* 解释时创建请求节点回送PId值（它们被表示为由上述Creat！功能产生）。
        每次创建请求的解释产生一个非零的唯一的PId值。每个进程实例隐含地声明
        了三个PId 变量、称为“Parent”、“Self” 和“Offspring”。

    创建请求节点的解释产生一个新的和唯一的PId 值，并把它赋给正在创建的任务的Offspring。
    被创建的任务的Self 标识符被赋给这同一个值.而被创建的任务的Parent被赋给创建者本身的值。
    被创建的任务中的Offspring的长度被置为零。


    */ END PId;
```

5.9 *String*（串）

```
        GENERATOR String (TYPE Item);
            /* 由字符串字面值所规定的字面值*/
            /* 仅当用字符来作为生成元实例时*/

        OPERATORS
            Declare!(v)' : -> String;
            "//"         : String, String -> String;
            Length       : String -> Natural;
            First        : String -> Item;
            Last         : String -> Item;
            Extract!     : String, Natural -> Item;
            Insert       : String', Natural, String ->;
            Insert!      : String', Natural, Item ->;

        AXIOMS
            Length(Declare!(v)')=0;
            Length (S1 // S2) = Length(S1) + Length(S2);
            First(S1) = Extract!(S1, 1);
            Last(S1) = Extract!(S1, Length(S1));
            Extract!(Insert(S1,I,S2),j) =
              IF j< 1 THEN Error! ELSE
                IF j<= I THEN Extract!(S1,j) ELSE
                  IF j<= Length(S2) + I THEN Extract!(S2,j-I)) ELSE
                    IF j<= Length (S1) + Length(S2) THEN
                            Extract!(S1,J – Length(S2)) ELSE Error!
                    FI
                  FI
                FI
              FI;
            S1 // S2 = Insert(S1, Length(S1), S2);
            Extract!(Insert!(S1, I, It)', J) =
              IF I = J Then It Else Extract!(S1, J) FI;

        END String;
```

5.10 *Time*（时刻）

```
NEWTYPE Time Inherits Real

ADDING
    Operators
        Now: -> Time /*‘实’时*/;

END Time;
```

5.11 *Duration*（持续时间）

```
NEWTYPE Duration Inherits Real ("+", "-", "*", "/")

ADDING
    Operators
        "+": Time, Duration -> Time;
        "+": Duration, Time -> Time;
        "-": Time, Duration -> Time;

CONSTANTS > = 0.0

END Duration;
```

5.12 *Timer*（定时器）

```
NEWTYPE Timer

    OPERATORS
        Set: Time, Timer' *->;
        Reset: Timer' ->;
        Active: Timer -> Boolean;

    AXIOMS
        Active ( Set(Tm, Tmr)') = Tm > Now ();
        Active( Reset(Tmr) ) = False;
        /* 注意：活跃的定时器在它变为不活跃时，要给进程发送一个信号。此信号的命名应与Set 调用中的
        定时器变量名相同* /

    END Timer;
```

5.13 *Charstring*（字符串）

```
NEWTYPE Charstring String (Character)
    Adding LITERALS /* 字符串字面值* /
    END Charstring;
```

中国印刷-ISBN 92-61-02235-9
统一书号：15045·总3380-有5509