国 际 电 信 联 盟

CCITT
国际电报电话
咨询委员会

ISO
国际标准化组织

IEC
国际电工技术
委员会

蓝 皮 书

卷 X.6

CCITT 高级语言(CHILL)

国际标准
ISO/IEC 9496

建议 Z.200

第 九 次 全 体 会 议
1988年11月14—25日 墨尔本

文献号
ISO/IEC 9496: 1989 (E)

1989年 日内瓦

国 际 电 信 联 盟

CCITT
国际电报电话
咨询委员会

ISO
国际标准化组织

IEC
国际电工技术
委员会

蓝 皮 书

卷 X.6

CCITT 高级语言(CHILL)

建议 Z.200

国际标准
ISO/IEC 9496

第 九 次 全 体 会 议
1988年11月14—25日 墨尔本

文献号
ISO/IEC 9496: 1989 (E)

1989年 日内瓦

ISBN 92-61-03805-0

中国印刷

CCITT 图书目录
第九次全体会议(1988 年)

蓝 皮 书

卷 I

卷 I.1 — 全会会议记录和报告
研究组及研究课题一览表

卷 I.2 — 意见和决议
关于 CCITT 的组织和工作程序的建议(A 系列)

卷 I.3 — 术语和定义 缩略语和首字母缩写词 关于措词含义的建议(B 系列)和综合电信统计的建议(C 系列)

卷 I.4 — 蓝皮书索引

卷 II

卷 II.1 — 一般资费原则 — 国际电信业务的资费和帐务 D 系列建议(第 III 研究组)

卷 II.2 — 电话网和 ISDN — 运营、编号、选路和移动业务 建议 E.100-E.333(第 II 研究组)

卷 II.3 — 电话网和 ISDN — 服务质量、网络管理和话务工程 建议 E.401-E.880(第 II 研究组)

卷 II.4 — 电报业务和移动业务 — 运营和服务质量 建议 F.1-F.140(第 I 研究组)

卷 II.5 — 远程信息处理业务、数据传输业务和会议电信业务 — 运营和服务质量 建议F.160-F.353、F.600、F.601、F.710-F.730(第 I 研究组)

卷 II.6 — 报文处理和查号业务 — 运营和服务的限定 建议 F.400-F.422、F.500(第 I 研究组)

卷 III

卷 III.1 — 国际电话接续和电路的一般特性 建议 G.100-G.181(第 XII 和 XV 研究组)

卷Ⅲ.2 — 国际模拟载波系统　建议 G.211-G.544(第 XV 研究组)

卷Ⅲ.3 — 传输媒质 — 特性　建议 G.601-G.654(第 XV 研究组)

卷Ⅲ.4 — 数字传输系统的概况;终端设备　建议 G.700-G.795(第 XV 和第 XⅧ研究组)

卷Ⅲ.5 — 数字网、数字段和数字线路系统　建议 G.801-G.961(第 XV 和第 XⅧ研究组)

卷Ⅲ.6 — 非话信号的线路传输　声音节目和电视信号的传输　H 和 J 系列建议(第 XV 研究组)

卷Ⅲ.7 — 综合业务数字网(ISDN) — 一般结构和服务能力　建议 I.110-I.257(第 XⅧ研究组)

卷Ⅲ.8 — 综合业务数字网(ISDN) — 全网概貌和功能、ISDN 用户-网络接口　建议 I.310-I.470(第 XⅧ研究组)

卷Ⅲ.9 — 综合业务数字网(ISDN) — 网间接口和维护原则　建议 I.500-I.605(第 XⅧ研究组)

卷Ⅳ

卷Ⅳ.1 — 一般维护原则:国际传输系统和电话电路的维护　建议 M.10-M.782(第Ⅳ研究组)

卷Ⅳ.2 — 国际电报、相片传真和租用电路的维护　国际公用电话网的维护　海事卫星和数据传输系统的维护　建议 M.800-M.1375(第Ⅳ研究组)

卷Ⅳ.3 — 国际声音节目和电视传输电路的维护　N 系列建议(第Ⅳ研究组)

卷Ⅳ.4 — 测量设备技术规程　O 系列建议(第Ⅳ研究组)

卷 V — 电话传输质量　P 系列建议(第 XⅡ研究组)

卷Ⅵ

卷Ⅵ.1 — 电话交换和信令的一般建议　ISDN 中服务的功能和信息流　增补　建议 Q.1-Q.118(乙)(第Ⅺ研究组)

卷Ⅵ.2 — 四号和五号信令系统技术规程　建议 Q.120-Q.180(第Ⅺ研究组)

卷Ⅵ.3 — 六号信令系统技术规程　建议 Q.251-Q.300(第Ⅺ研究组)

卷Ⅵ.4 — R1和 R2信令系统技术规程　建议 Q.310-Q.490(第Ⅺ研究组)

卷Ⅵ.5 — 综合数字网和模拟—数字混合网中的数字本地、转接、组合交换机和国际交换机　增补　建议 Q.500-Q.554(第Ⅺ研究组)

卷Ⅵ.6 — 各信令系统之间的配合　建议 Q.601-Q.699(第Ⅺ研究组)

卷Ⅵ.7 — 七号信令系统技术规程　建议 Q.700-Q.716(第Ⅺ研究组)

卷Ⅵ.8 — 七号信令系统技术规程　建议 Q.721-Q.766(第Ⅺ研究组)

卷Ⅵ.9 — 七号信令系统技术规程　建议 Q.771-Q.795(第Ⅺ研究组)

卷Ⅵ.10 — 一号数字用户信令系统(DSS 1)　数据链路层　建议 Q.920-Q.921(第Ⅺ研究组)

卷Ⅵ.11 — 一号数字用户信令系统(DSS 1)　网络层、用户—网络管理　建议 Q.930-Q.940(第Ⅺ研究组)

卷Ⅵ.12 — 公用陆地移动网 与ISDN和PSTN的互通 建议Q.1000-Q.1032(第Ⅺ研究组)

卷Ⅵ.13 — 公用陆地移动网 移动应用部分和接口 建议Q.1051-Q.1063(第Ⅺ研究组)

卷Ⅵ.14 — 其它系统与卫星移动通信系统的互通 建议Q.1100-Q.1152(第Ⅺ研究组)

卷Ⅶ

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

卷Ⅶ.2 — 电报交换 U系列建议(第Ⅸ研究组)

卷Ⅶ.3 — 远程信息处理业务的终端设备和协议 建议T.0-T.63(第Ⅷ研究组)

卷Ⅶ.4 — 智能用户电报各建议中的一致性测试规程 建议T.64(第Ⅷ研究组)

卷Ⅶ.5 — 远程信息处理业务的终端设备和协议 建议T.65-T.101,T.150-T.390(第Ⅷ研究组)

卷Ⅶ.6 — 远程信息处理业务的终端设备和协议 建议T.400-T.418(第Ⅷ研究组)

卷Ⅶ.7 — 远程信息处理业务的终端设备和协议 建议T.431-T.564(第Ⅷ研究组)

卷Ⅷ

卷Ⅷ.1 — 电话网上的数据通信 V系列建议(第XⅦ研究组)

卷Ⅷ.2 — 数据通信网:业务和设施,接口 建议X.1-X.32(第Ⅶ研究组)

卷Ⅷ.3 — 数据通信网:传输,信令和交换,网络概貌,维护和管理安排 建议X.40-X.181(第Ⅶ研究组)

卷Ⅷ.4 — 数据通信网:开放系统互连(OSI) — 模型和记法表示,服务限定 建议X.200-X.219(第Ⅶ研究组)

卷Ⅷ.5 — 数据通信网:开放系统互连(OSI) — 协议技术规程,一致性测试 建议X.220-X.290(第Ⅶ研究组)

卷Ⅷ.6 — 数据通信网:网间互通,移动数据传输系统,网际管理 建议X.300-X.370(第Ⅶ研究组)

卷Ⅷ.7 — 数据通信网:报文处理系统 建议X.400-X.420(第Ⅶ研究组)

卷Ⅷ.8 — 数据通信网:查号 建议X.500-X.521(第Ⅶ研究组)

卷Ⅸ — 干扰的防护 K系列建议(第Ⅴ研究组) 电缆及外线设备的其它部件的结构、安装和防护 L系列建议(第Ⅵ研究组)

卷X

卷X.1 — 功能规格和描述语言(SDL) 使用形式描述方法(FDT)的标准 建议Z.100和附件A、B、C和E,建议Z.110(第X研究组)

卷X.2 — 建议Z.100的附件D:SDL用户指南(第X研究组)

卷X.3 — 建议Z.100的附件F.1:SDL形式定义 介绍(第X研究组)

卷X.4 — 建议Z.100的附件F.2:SDL形式定义 静态语义学(第X研究组)

卷X.5 — 建议Z.100的附件F.3:SDL形式定义 动态语义学(第X研究组)

卷X.6 — CCITT高级语言(CHILL) 建议Z.200(第X研究组)

卷X.7 — 人机语言(MML) 建议Z.301-Z.341(第X研究组)

CCITT 高级语言（CHILL）

（日内瓦，1988）

目 录

1 绪论

本建议定义了 CCITT 高级程序设计语言 CHILL。CHILL 是 CCITT High Level Language 的缩写。

本章下面各节介绍语言设计的某些背景动机，并给出语言特色的概貌。

有关 CHILL 语言知识的另一种入门性简介和 CHILL 培训资料，建议读者分别参阅 CCITT 手册、“CHILL 入门”和“CHILL 用户手册”。

题为“CHILL 形式定义”的 CCITT 手册给出了 CHILL 的另一种定义，它是以基于 VDM（维也纳定义方法）的严格数学形式定义的。

1.1 概述

CHILL 是一种强类型、分程序结构化语言，主要设计目标是用来实现大型、复杂的嵌入式系统。

在设计 CHILL 时，考虑了下列要求：

- 借助充分的编译时刻检验以提高可靠性和运行效率；
- 使用足够灵活，表达能力足够强，以覆盖所要求的应用领域，并充分发挥不同硬件的作用；
- 提供有利于大型系统分块和模块化开发的设施；
- 由提供内在的并发和时钟监控原语来适应实时程序设计的需要；
- 能生成高效目标代码；
- 易学易用。

语言设计所提供的内在表达能力使得 CHILL 程序员能从一组丰富的设施中选择适当的成份，以便结果实现能更准确地与最初的设计描述相匹配。

由于 CHILL 严格区分静态对象和动态对象，故几乎所有的语义检查都能在编译时刻得以完成。这可大大缩短运行时间。违反 CHILL 动态规则将引发运行时刻异常，而这些异常可以由某个合适的异常处理程序处理（然而，是否生成这种隐式检验的代码是任选的，除非显式指明了某个用户定义的处理程序）。

CHILL 允许以与机器无关的方式编写程序。语言本身是独立于机器的，但特定的编译系统可能要求提供某些实现定义的对象。注意：包含这种对象的程序往往是不可移植的。

1.2 语言总貌

CHILL 程序基本上由三个部分组成：

- 数据对象的描述；
- 在数据对象上实施的动作的描述；
- 程序结构的描述。

数据对象由数据语句（说明语句和定义语句）描述，动作由动作语句描述，而程序结构由程序结构语句确定。

CHILL 能处理的数据对象是值和可用来存储值的单元。动作定义作用于数据对象之上的运算，并定义了向单元内存储值及从单元中取出值的次序。程序结构确定数据对象的生存期和可见性。

对于在某个给定上下文内数据对象的使用，CHILL 提供了充分的静态检验。

以下各节概述了各种 CHILL 概念。每一节都是对与该节同名的一章内容的简介，有关概念的详述见同名章节。

1.3 模式和类

每个单元具有一个附着其上的模式。单元的模式定义了可以存放于该单元内的值的集合，也定义了与该单元有关的其它性质（注意：单元的所有性质并不是都由其模式单独确定的）。单元的性质包括大小、内

部结构、只读性及引用性等。值的性质包括内部表示、排序和可应用的运算等。

每个值具有一个附着其上的类。值的类确定了可存放该值的单元的模式。

CHILL 提供了下列诸种模式：

离散模式	整数模式、字符模式、布尔模式、(符号) 集合模式以及基于这些模式的范围模式；
幂集模式	某种离散模式的元素的集合；
引用模式	用于引用单元的受限引用模式、自由引用模式及行模式；
组合模式	串模式、数组模式和结构模式；
过程模式	过程被视为可加工的数据对象；
实例模式	用于进程的标识；
同步模式	用于进程同步与通信的事件模式和缓冲区模式；
输入输出模式	用于输入输出操作的结合模式、访问模式及文本模式；
计时模式	用于时钟监控的时延模式和绝对时钟模式。

CHILL 给出了一组标准模式的表示。借助模式定义可以引进由程序定义的模式。某些语言成份具有一个附着其上的、被称为动态模式的模式。动态模式是指其某些性质只能动态确定的模式。动态模式都是带有运行时刻参数的参数化模式。非动态模式的模式称为静态模式。

在 CHILL 中不存在类的表示。它们仅在元语言中出现，用来描述静态和动态上下文条件。

1.4 单元及其访问

单元是（抽象的）位置，可将值存进去，也可从中把值取出来。为了存放和得到一个值，必须访问一个单元。

说明语句定义用于访问一个单元的名字，有两种格式的说明语句：

1. 单元说明；
2. 单元等同说明。

第一种创建了单元，并确定了对新创建的单元的访问名字。后一种为其它处已创建的单元建立新的访问名字。

除单元说明外，借助 *GETSTACK* 或 *ALLOCATE* 内部子程序调用也能创建新单元，这两个内部子程序调用均产生一个引用它所创建的单元的引用值（见下）。

一个单元可以是**可引用的**。这意味着对该单元存在一个相应的引用值。对该**可引用**单元实施引用操作的结果就得到该引用值。通过对某个引用值实施间接引用运算就可获得该值所引用的单元。CHILL 要求某些单元是**可引用的**，而另一些单元是不**可引用的**。对其它单元而言，它们是不是**可引用的**要由实现确定。引用性应该是单元的一种可静态确定的性质。

一个单元可以具有**只读**模式，这意味着只能通过访问该单元才能从中取出一个值，而不能将一个新值存入该单元（赋初值除外）。

一个单元可以是组合的，这意味着它具有能被独立访问的子单元。子单元不一定是**可引用的**。如果一个单元至少包含一个**只读**子单元，则称该单元具有**只读性质**。提供子单元（或子值）的访问方法分别为：对串或数组而言是取下标和切片；对结构而言是选择。

每个单元附有一个模式。若该模式是动态的，则该单元被称为动态模式单元。

单元的下列性质虽然可以静态地确定，但它们并不属于模式：

引用性：对该单元是否存在一个引用值；

存储种类：该单元是否被静态地分配存储空间；

区域性：该单元是不是在某个区域内部说明的。

1.5 值及其运算

值是在其上定义了特殊运算的基本对象。一个值要么是一个 (CHILL) 有定义值，要么是一个 (在 CHILL

意义下的）**未定义**值。在已指明的上下文中使用**未定义**值将导致（在 CHILL 意义下的）未定义的情况出现，此时认为该程序是错误的。

CHILL 允许要求出现值的上下文处使用单元。在这种情况下，对该单元的访问是取出其内存放的值。

每个值附有一个类。除了附有类外还附有模式的值称为**强**值。在这种情况下，该值总是由该模式所定义的值之一。**强**值的类用于相容性检查，而**强**值的模式用来描述该值的性质。某些上下文要求值的这些性质是已知的，此时就要求出现**强**值。

一个值可以是**字面值**，此时它表示一个与实现无关的、编译时刻已知的离散值。一个值可以是**常数**，此时它总是传递同一个值，即它只需计算一次。当上下文要求一个**字面值**或**常数**值时，该值被假定是在程序运行前计算的，因而不会产生运行时刻异常。一个值可以是**区域内的**，此时它能以某种方式与在某个区域内部说明的单元有联系。一个值可以是组合的，即它含有子值。

同义词定义语句建立表示**常数**值的新名字。

1.6 动作

动作构成 CHILL 程序的算法部分。

赋值动作把一个（计算好的）值存入一个或多个单元。过程调用调用一个过程，而内部子程序调用调用一个内部子程序（内部子程序是这样一种过程，其定义不必用 CHILL 编写，且它具有更为通用的参数传递机制和结果返送机制）。为了从过程调用返回和/或建立过程调用的结果，要使用返回动作和结果动作。

为了控制顺序动作流，CHILL 提供了下列控制动作流：

条件动作　　用于两叉分支；

情况动作　　用于多叉分支。与判别表的情形类似，分支的选择可能以多个值为基础；

循环动作　　用于迭代或加括号；

出口动作　　用于以结构化方式退出某个带括号动作或某个模块；

引发动作　　引发某个特定的异常；

转向动作　　用于无条件转移到某个加了标号的程序点。

动作和数据语句可以组合到一起形成一个模块或 begin-end 分程序。后者构成一个（复合）动作。

为了控制并发动作流，CHILL 提供了启动、停止、延迟、继续、发送、延迟情况和接收情况等动作以及接收表达式和启动表达式。

1.7 输入和输出

CHILL 的输入输出设施提供 CHILL 程序与外界各种设备之间通信的手段。

输入输出参考模型识别三种状态。在自由状态下，CHILL 程序与外界不存在任何交互关系。

通过 *ASSOCIATE* 操作进入文件处理状态。在文件处理状态下，存在表示外界对象的结合模式单元。可以通过内部子程序读或修改语言定义的结合属性，即**存在的**、**可读的**、**可写的**、**可求下标的**、**顺序的**和**可变的**属性。文件的建立和删除也在文件处理状态下完成。

通过 *CONNECT* 操作将某个访问模式单元连接到某个结合模式单元之上，并进入数据传送状态。*CONNECT* 允许在一个文件中设置**基本**下标。在数据传送状态下，可以检测访问模式单元的各种属性，也能调用数据传送操作 *READRECORD* 和 *WRITERECORD*。

通过文本传送操作，可将 CHILL 值从某个文件输入到某个 CHILL 单元，或从某个 CHILL 单元输出至某个文件，这时可将该 CHILL 值表示成人类可读的格式。

1.8 异常处理

CHILL 的动态语义条件是那些一般说来不能被静态确定的（非上下文无关的）条件（除非已显式指明了合适的处理程序，是否生成运行时刻检测动态条件的代码由实现系统确定）。违反动态语义规则会引发运行

时刻异常。然而，如果某种实现系统能静态地确定出一个程序将违反某种动态条件的话，它可以拒绝该程序。

异常也能因执行引发动作而引发，或有条件地因执行断言动作而引发。在某个给定的程序点，当出现某个异常时，如果它是可指明的（即它有一个名字），且已指明了的话，控制就转向该异常有关的处理程序。可以静态地确定在给定程序点处是否为一个异常指明处理程序。如果未显式指明处理程序，则控制可以转向一个实现定义的异常处理程序。

异常都有名字，这个名字要么是一个 CHILL 定义的异常名字或一个实现定义的异常名字，要么是一个程序定义的异常名字。注意：当为某个异常名字指明异常处理程序时，必须检测与之相应的动态条件。

1.9 时钟监控

CHILL 时钟监控设施提供感知外界时钟消逝的方法。只有在某些精确的**可超时的**程序点处，执行期间的 CHILL 进程才能被中断。当这种情况发生时，控制转移到某个合适的（超时）处理程序上。

程序可以检测一段时间的消逝，也能与某个绝对时钟点同步，或在无累积误差的精确间隙内与外界时钟同步。所提供的时钟内部子程序将用来把时钟值和时延值转换为整数值、使某个进程进入等待状态以及检测时钟监控是否到期。

1.10 程序结构

程序结构语句包括 begin-end 分程序、模块、过程、进程和区域。程序结构语句提供控制单元的生存期和名字的可见性的手段。

单元的生存期是指单元在程序内存在的那一段时间。单元能（通过一个单元说明）显式地说明或（通过调用内部子程序 *GETSTACK* 或 *ALLOCATE*）显式地生成，也能作为某些语言成份的使用结果而隐式地说明或生成。

如果一个名字在程序内某点处可以使用，则称该名字在该点是**可见的**。一个名字的作用域包括了该名字的所有可见点，即在一个名字的作用域内，由它表示的对象是由该名字标识的。

begin-end 分程序既确定名字的可见性，又确定单元的生存期。

模块用于限制名字的可见性，谨防它们被非法使用。借助可见性语句，可以精确控制名字在不同程序部分内的可见性。

过程是一个（可能参数化了的）子程序，它能在程序内部的不同位置被引用（调用）。过程可以返送一个值（值过程）或一个单元（单元过程），也可以不返送结果。在后一种情况下，该过程只能在过程调用动作中被调用。

进程和区域提供构造并发执行程序的手段。

一个完整的 CHILL 程序是一串模块或区域，并认为被一个（假想的）进程定义包围。这个最外层进程由控制该程序执行的系统启动。

CHILL 提供了有助于按若干不同的方式分块开发程序的构造。说明模块和说明区域用来定义某个程序块的静态性质，上下文用来定义各移入名字的静态性质。此外，还可以通过远程设施指出某程序块的正文在何处能找到。

1.11 并发执行

CHILL 允许程序单位间的并发执行。进程是并发执行的程序单位。启动表达式的求值创建了一个指定的进程定义的新进程。此后该进程被视为同其启动进程并发执行。CHILL 允许一个或多个具有相同或不同进程定义的进程同时都处于活跃状态。一个进程执行停止动作后该进程执行终止。

进程总是处于活跃的或被延迟的这两种状态之一。从活跃态变为被延迟态称为进程的延迟，而从被延迟态变回活跃态称为进程的重新激活。执行定义在事件单元之上的延迟动作、定义在缓冲区单元或信号之

上的接收动作以及定义在缓冲区单元之上的发送动作都能使该执行进程变成被延迟态。执行定义在事件单元之上的继续动作、定义在缓冲区单元或信号之上的发送动作以及定义在缓冲区单元之上的接收动作都能使某个被延迟的进程再次成为活跃的。

缓冲区（单元）和事件（单元）都是单元，它们的使用是有限制的。发送、接收及接收情况操作是定义在缓冲区之上的，而延迟、延迟情况及继续操作是定义在事件之上的。缓冲区（单元）是进程间同步和传递信息的手段。事件（单元）只用于进程间的同步。信号由信号定义语句定义。信号表示进程间传递的值表的组合与分解功能。发送信号动作和接收信号情况动作用于进程间传送值表和取得同步。

区域是一种特殊的模块。它用来对由几个进程所共享的数据结构实施互斥访问。

1.12 总的语义性质

CHILL 的（非上下文无关的）语义条件是模式与类的相容性条件（模式检验）和可见性条件（作用域检查）。模式检验规则确定如何使用名字，而作用域检查规则确定名字可以在哪儿使用。

模式检验规则用模式与模式之间、模式与类之间以及类与类之间的相容性要求加以描述。模式与类之间以及类与类之间的相容性要求是通过模式间的等价关系定义的。如果涉及动态模式，则模式检验之一部分是动态的。

作用域检查规则通过程序结构语句和显式的可见性语句确定名字的可见性。显式的可见性语句影响它所提到的名字的作用域，也影响那些名字所**隐含的**名字的作用域。程序中引入的每个名字都具有一个被定义或说明的程序点。该程序点称为该名字的定义性出现，而使用该名字的程序点称为该名字的应用性出现。名字约束规则把名字的唯一定义性出现与它的每个应用性出现联系起来。

1.13 实现任选

CHILL 允许有实现定义的整数模式、实现定义的内部子程序、实现定义的**进程**名字、实现定义的异常处理程序和实现定义的异常名字。

实现定义的整数模式必须由实现定义的**模式**名字表示。这种名字被认为是未在 CHILL 中说明过的、由一个新模式定义语句定义的。在 CHILL 的语法和语义规则的范围内允许将现有的 CHILL 定义的算术运算推广到实现定义的整数模式中去。实现定义的整数模式的例子有长整数和短整数。

实现定义的内部子程序是一种过程，其过程定义不必用 CHILL 编制，它可以具有比 CHILL 过程更通用的参数传递和结果返送策略。

实现定义的**进程**名字是一个不用 CHILL 语句定义的**进程**名字，它可以具有比 CHILL 进程更通用的参数传递机制。CHILL 进程可以与实现定义的进程协同运行，或启动这种进程。

实现定义的异常处理程序是附着于某个进程定义的处理程序。若一个异常出现后这种处理程序获得了控制，则实现系统决定执行哪些动作。如果违反某种实现定义的动态条件，则引发一个实现定义的异常。

2 预备知识

2.1 元语言

CHILL 的描述由两个部分组成：

- 上下文无关语法的描述；
- 语义条件的描述。

2.1.1 上下文无关语法描述

上下文无关语法用扩展的巴科斯-瑙尔范式描述。语法范畴由用尖括号（〈和〉）括起来的一组楷体汉字或英文词组指明，称之为一个非终结符。对每个非终结符而言，在相应语法部分都给出一条产生式规则。非终结符的产生式规则是这样组成的：在符号::=的左侧是该非终结符，在符号::=之右侧是一个或多个由非终结符与/或终结符组成的成分，这些成分之间用竖线（|）分开，并表示该非终结符的不同候选产生式。

有时，非终结符包括带下划线的部份。这种带下划线的部分并不是上下文无关语法描述的一部分，但它定义了一种语义子范畴（参见第 2.1.2 节）。

语法单位可以用花括号（{和}）组合在一起。被花括号括起来的一组语法单位的重复由星号（*）或加号（+）指明 。星号表示该组语法单位是任选的，也可以重复任意多次；加号表示该组语法单位必须存在，且可以再重复任意多次。例如，{A}*表示 A 的任意序列，其中包括不出现 A，而 $\{A\}^+$代表最少有一个 A 的序列。如果语法单位由方括号（[和]）组合起来，则表示这组语法单位是任选的。被花括号或方括号括起来的一组语法单位间可以包含一个或多个竖线，它们表示不同的候选语法单位。

语法有严格语法和导出语法之分，它们的区别在于：严格语法的语义条件是直接给出的，而导出语法被认为是严格语法的拓广，其语义通过相应严格语法的语义间接解释。

应当注意的是，本文献中上下文无关语法描述的方式是根据语义描述的需要选定的，并非为了适应任何特定的语法分析算法（例如，为了阐述清晰起见，语法描述中引入了一些上下文无关的二义性）。使用该语法单位的语义范畴可以解决这种二义性问题。

2.1.2 语义描述

每个语法范畴（非终结符）的语义描述在**语义**、**静态性质**、**动态性质**、**静态条件**和**动态条件**等部分给出。

语义部分描述由该语法范畴所表示的概念（即它的含义和行为）。

静态性质部分定义该语法范畴的可静态确定的语义性质。在使用该语法范畴的相应章节中，这些性质用来阐述其静态与/或动态条件。

动态性质部分定义该语法范畴的只能动态可知的那些语义性质。

静态条件部分描述与上下文有关的、可静态检查的那些语义条件，当使用该语法范畴时必须满足这些条件。利用带下划线部分可以在语法描述中表达某些静态条件（参见第 2.1.1 节）。这种用法要求该非终结符属于特定的语义子范畴。例如，〈布尔表达式〉从上下文无关语法描述的意义上说与〈表达式〉是相同的，但在语义上要求该表达式具有布尔类。

动态条件部分描述程序执行期间必须满足的、与上下文有关的那些语义条件。在某些情况下，如果不涉及动态模式，那些语义条件是静态的。此时，在**静态条件**部分提到这类条件，而在**动态条件**部分引用这些条件。在其它情况下，可以静态检查动态语义条件。某种实现可以将它作为违反静态条件情况处置。

在语义描述中，按下列约定使用不同的字体，即（不带〈和〉的）楷体用于指明语法对象；仿宋体的

相应术语表示相应的语义对象（如，*单元*表示单元）；黑体用于命名语义性质。有时一个性质既可通过语法来表达，也可通过语义来陈述（例如，句子“*表达式*是**常数**”与“*表达式*是一个*常数*表达式”的含义相同）。

除非已特别指明，在某个语法范畴的**语义**部分、**性质**部分和**条件**部分分述的语义、性质和条件均成立，而不管其它章节内可能出现的该语法范畴处的上下文。

除非已特别指明，具有形如 A::=B 的产生式的语法范畴 A 的性质与 B 的性质相同，其中 B 也是一个语法范畴。

2.1.3 例子

对大多数语法章节而言，都有一个**例子**部分，它给出所定义的语法范畴的一个或几个例子，这些例子选自附录 D 所含的一组程序实例。引文指明每个例子由哪个语法产生式规则产生，并指明取自附录 D 中哪个程序实例。例如，*6.20* (d+5) /5 (*1.2*) 指出它是一个终结符串 (d+5) /5，它是根据相应章节语法部分的产生式规则 (*1.2*) 产生的，取自附录 D 第六个程序实例的第 20 行。

2.1.4 元语言中的约束规则

有时语义描述提到 CHILL **专用**简单名字串（参见附录 C）。这些**专用**简单名字串在使用中总是带有其 CHILL 含义，因而不受实际 CHILL 程序的名字约束规则的影响。

2.2 词汇表

使用 CHILL 字符集（参见附录 A）表达程序。该字母表由语法范畴〈字符〉表示，从这一点上说，CHILL 字符集中任一个字符都可以作为一个终结产生式导出。

CHILL 的词法单位有：

- 专用符号；
- 简单名字串；
- 字面值。

除了上述词法单位外，CHILL 中还存在专用字符组合。附录 B 列出专用符号和专用字符组合。

简单名字串根据下列语法组成：

语法：

〈简单名字串〉::= (*1*)

〈字母〉{〈字母〉|〈数字〉|_}* (*1.1*)

〈字母〉::= (*2*)

A|*B*|*C*|*D*|*E*|*F*|*G*|*H*|*I*|*J*|*K*|*L*|*M* (*2.1*)

| *N*|*O*|*P*|*Q*|*R*|*S*|*T*|*U*|*V*|*W*|*X*|*Y*|*Z* (*2.2*)

| *a*|*b*|*c*|*d*|*e*|*f*|*g*|*h*|*i*|*j*|*k*|*l*|*m* (*2.3*)

| *n*|*o*|*p*|*q*|*r*|*s*|*t*|*u*|*v*|*w*|*x*|*y*|*z* (*2.4*)

〈数字〉::= (*3*)

0|*1*|*2*|*3*|*4*|*5*|*6*|*7*|*8*|*9* (*3.1*)

语义：下划线字符（_）是简单名字串的一部分。如简单名字串 life_time 与简单名字串 *lifetime* 是不同的。大写字母和小写字母是不同的，如 *Status* 和 *status* 是两个不同的简单名字串。

CHILL 语言有若干具有已预先确定其含义的**专用**简单名字串（参见附录 C）。其中有一些是**保留**简

单名字串，即它们不能派作其它用途。

在某个程序段中的**专用**简单名字串要么全用大写字母表示，要么全用小写字母表示。**保留**简单名字串只在选定的一种表示方式下才具有特定含义（例如，如果选用小写字母表示，则 **row** 是**保留**简单名字串，而 **ROW** 则不是）。

静态条件：简单名字串不能是某个**保留**简单名字串（参见附录 C.1）。

2.3 空格的使用

空格终止任何词法单位或专用字符组合。词法单位也终止于不可能是该词法单位之一部分的第一个字符。举例来说，*IFBTHEN* 被认为是一个简单名字串，而不是动作语句 **IF** *B* **THEN** 的开始；//* 被认为是并置运算符（//）后跟一个星号（*），而不是一个除号（/）后跟一个注释开括号（/*）。

2.4 注释

语法：

〈注释〉::= (1)

〈带括号注释〉 (1.1)

| 〈行尾注释〉 (1.2)

〈带括号注释〉::= (2)

/* 〈字符串〉 */ (2.1)

〈行尾注释〉::= (3)

-- 〈字符串〉〈行尾〉 (3.1)

〈字符串〉::= (4)

{ 〈字符〉}* (4.1)

注意：行尾表示该注释所在行的结束。

语义：注释向程序的阅读者传递信息。它对程序的语义没有影响。

只要是允许使用空格作为词法单位分隔符的地方，都可以插入注释。

带括号注释由出现的第一个专用序列 */ 而终止。行尾注释由该行的结束的第一次出现而终止。

例子：

4.1 / from collected algorithms from CACM no. 93 */* (2.1)

2.5 格式控制符

CHILL 字符集中的格式控制符 BS（退格）、CR（回车）、FF（换页）、HT（水平制表）、LF（换行）和 VT（垂直制表）（参见附录 A，它们位于 FE_0到 FE_5）在 CHILL 上下文无关语法描述中没有提到。但当使用它们时，它们具有与空格一样的分隔作用。除字符串字面值外，不能在词法单位内部使用空格和格式控制符。

2.6 编译程序指示

语法：

〈指示子句〉::= (1)

〈〉〈指示〉{,〈指示〉}* 〈〉 (1.1)

〈指示〉::= (2)

〈实现指示〉 (2.1)

语义：指示子句向编译程序传递信息。这类信息以实现定义的形式来指明。

实现指示不能影响程序的语义，即一个带实现指示的程序在CHILL意义下是正确的，当且仅当在去掉实现指示后它也是正确的时。

指示子句由出现的第一个指示结束符（〈〉）而终止。指示可以包含CHILL字符集（参见附录A）中任一字符.

静态性质：指示子句可以插在任何允许出现空格的地方，它具有空格一样的分隔作用。指示子句中使用的名字遵循实现定义的名字约束规则，而这种约束规则不影响CHILL名字约束规则（参见12.2节）。

2.7 名字及其定义性出现

语法：

〈名字〉::= (1)

〈名字串〉 (1.1)

〈名字串〉::= (2)

〈简单名字串〉 (2.1)

|〈带前缀名字串〉 (2.2)

〈带前缀名字串〉::= (3)

〈前缀〉!〈简单名字串〉 (3.1)

〈前缀〉::= (4)

〈简单前缀〉{!〈简单前缀〉}* (4.1)

〈简单前缀〉::= (5)

〈简单名字串〉 (5.1)

〈定义性出现〉::= (6)

〈简单名字串〉 (6.1)

〈定义性出现表〉::= (7)

〈定义性出现〉{,〈定义性出现〉}* (7.1)

〈域名字〉::= (8)

〈简单名字串〉 (8.1)

〈域名字定义性出现〉::= (9)

〈简单名字串〉 (9.1)

〈域名字定义性出现表〉::= (10)

〈域名字定义性出现〉{,〈域名字定义性出现〉}* (10.1)

〈异常名字〉::= (11)

〈简单名字串〉 (11.1)

|〈带前缀名字串〉 (11.2)

〈正文引用名字〉::= (12)

〈简单名字串〉 (12.1)

| 〈带前缀名字串〉 (12.2)

语义：程序内出现的名字表示对象。对程序中给定的一个名字的出现而言（严格说来是名字的终结产生式的出现），12.2节内定义的名字约束规则提供了（所出现的）该名字**被约束**到其上的一个或多个定义性出现（严格说来是定义性出现的终结产生式的出现）。因而，该名字表示了由该定义性出现所定义或说明的对象。对一个名字而言，只有它是集合元素名字或是具有**准**定义性出现的名字时，它才可能存在多于一个的定义性出现。称这个或这些定义性出现定义了该名字。当一个名字是由被**约束**到其上的定义性出现所创建的名字时，称它是该名字的一个应用性出现。该名字具有与其最右端的简单名字串同样的简单名字串。

类似地，域名字**被约束**到域名字定义性出现之上，并且表示由这些域名字定义性出现所定义的（某个结构模式的）那些域。

根据第八章陈述的规则，异常名字用来识别异常处理程序。

按10.10.1节所述的规则，正文引用名字以实现定义的方式用于标识某段源程序正文的描述。

当一个名字**被约束**到一个以上的定义性出现之上时，每个这种定义性出现都定义或说明了同一个对象（精确规则参见10.10节和12.2.2节）。

表示法的定义： 给定名字串 NS 和字符串 P，P 或者为前缀，或者为空，给 NS 置前缀 P（记为 P!NS）的结果定义如下：

- 如果 P 为空，则 P!NS 就是 NS；
- 否则，P!NS 是将 P 中所有字符、一个置前缀运算符以及 NS 中所有字符并置后得到的名字串。

例如，如果 P 是 "$q!r$"，而 NS 是 "$s!n$"，则 P!NS 是 "$q!r!s!n$"。

静态性质：每个简单名字串都附有一个**规范**名字串，它就是该简单名字串本身。每个名字串都附有一个**规范**名字串，其定义如下：

- 如果该名字串是一个简单名字串，则其**规范**名字串就是该简单名字串的**规范**名字串，即这个简单名字串本身；
- 如果该名字串是一个带前缀名字串，则其**规范**名字串是由置前缀运算符所隔开的该名字串中所有简单名字串按自左向右次序的并置，即去掉该名字串内起词法单位分隔作用的空格、注释和格式控制符（如果有的话）后得到的名字串。

在本文献余下部分中：

- 名字、异常名字或正文引用名字的名字串分别指相应项中的名字串的**规范**名字串；
- 定义性出现、域名字或域名字定义性出现的名字串分别指上述各项中简单名字串的**规范**名字串。

名字约束规则是诸如下列规定的一些规则：

- 具有简单名字串的名字**被约束**到具有相同名字串的定义性出现之上；
- 具有带前缀名字串的名字**被约束**到具有与该名字的带前缀名字串内最右端简单名字串一样的名字串的定义性出现之上；
- 域名字**被约束**到具有与该域名字相同的名字串的域名字定义性出现之上。

名字继承它**被约束**到其上的定义性出现所定义的名字所具有的所有静态性质。域名字继承它**被约束**到其上的域名字定义性出现所定义的域名字所具有的所有静态性质。

3 模式和类

3.1 概述

每个单元具有一个附着其上的模式，每个值具有一个附着其上的类。单元所附着的模式定义了可以存放在该单元内的值的集合、对该单元的访问方式以及能在这些值上施行的运算。值所附着的类是确定可以存放该值的单元的模式的一种手段。有些值是**强的**。每个**强**值既附着一个类又附着一个模式。如果某个值所在的上下文处需要模式信息，则该值应当是**强**值。

3.1.1 模式

CHILL 有静态模式（即其所有性质均可静态确定的模式）和动态模式（即其某些性质在运行时刻才能得知的模式）。动态模式总是带运行时刻参数的参数化模式。

静态模式是语法范畴*模式*的终结产生式。

在本文献中，引入虚拟**模式**名字来描述在程序正文中不能显式表示的那些模式。此时，这种**模式**名字前面冠以一个与符号（&）。

模式也可能由在程序正文中未显式指明的值参数化。

3.1.2 类

CHILL 没有类的表示。

类分为如下几种，且 CHILL 程序中的任一个值都属于这些类之一：

- 对模式 M 而言，可能存在 M 值类。所有具有此类的值，并且只有这些值是**强**值，这些值所附着的模式是 M；
- 对模式 M 而言，可能存在 M 导出类；
- 对任何模式 M 均存在 M 引用类；
- **null** 类；
- **all** 类。

最后两种类是恒定的类，即它们不依赖任何模式 M。当且仅当一个类是 M 值类、M 导出类或 M 引用类，且 M 是一个动态模式时，称该类是动态的。

3.1.3 模式与类的性质以及模式与类之间的关系

CHILL 中的模式具有若干性质。这些性质可能是继承性质，也可能是非继承性质。由某个定义模式定义的**模式**名字具有该定义模式的所有继承性质。下面给出对所有模式均适用的性质的摘要（除第一个性质外，其余性质均在 12.1 节内定义）：

- 模式具有**新颖性**（这在 3.2.2、3.2.3 和 3.3 节中定义）；
- 模式可能具有**只读性质**；
- 模式可能是**可参数化的**；
- 模式可能具有**引用性质**；
- 模式可能具有**带标志参数化性质**；
- 模式可能具有**非值性质**。

CHILL 中的类可能具有下列性质（它们在 12.1 节内定义）：

- 类可能具有**根**模式；

- 一个或多个类可能具有**结果类**；

CHILL 中的运算由单元的模式和值的类确定。通过在 12.1 节内定义的模式检验规则来叙述如何由单元的模式和值的类来确定 **CHILL** 运算，而模式检验规则是用模式与模式、模式与类以及类与类之间的若干关系来阐述的。模式与模式、模式与类以及类与类之间存在下列关系：

- 两个模式可能是**相似的**；
- 两个模式可能是**值等价的**；
- 两个模式可能是**等价的**；
- 两个模式可能是**单元等价的**；
- 两个模式可能是**类同的**；
- 两个模式可能是**新颖性约束的**；
- 两个模式可能是**读相容的**；
- 两个模式可能是**动态读相容的**；
- 两个模式可能是**动态等价的**；
- 一个模式对另一个模式而言可能是**可限制的**；
- 一个模式可能与一个类是**相容的**；
- 一个类可能与另一个类是**相容的**。

3.2 模式定义

3.2.1 概述

语法：

〈模式定义〉::= (1)

〈定义性出现表〉 = 〈定义模式〉 (1.1)

〈定义模式〉::= (2)

〈模式〉 (2.1)

导出语法：其定义性出现表中包含多于一个定义性出现的模式定义是从满足下列条件的多个模式定义导出的，即每个模式定义的定义性出现表仅含一个定义性出现；每个模式定义都具有相同的定义模式；每个模式定义之间用逗号隔开。例如，

NEWMODE *dollar*, *pound* = *INT*;

是从

NEWMODE *dollar* = *INT*, *pound* = *INT*;

导出的。

语义：模式定义定义了一个表示指定模式的名字。模式定义出现于同义模式定义语句和新模式定义语句中。同义模式与其定义模式**同义**。新模式与其定义模式不**同义**。这种区别通过模式的**新颖性**性质来反映，模式的**新颖性**用于模式检验（参见 12.1 节）。

静态性质：模式定义中的定义性出现定义了一个**模式**名字。

预定义的**模式**名字以及实现定义的整数**模式**名字（如果有的话，参见 3.4.2 节）也是**模式**名字。

模式名字具有一个**定义**模式，它就是出现于定义该**模式**名字的模式定义中出现的定义模式（对于预定义和实现定义的**模式**名字而言，其**定义**模式是一个虚拟模式）。**模式**名字的继承性质是其**定义**模式所具有的那些继承性质。

递归定义的集合是满足下列条件的模式定义或同义词定义（参见 5.1 节）的集合，即每个模式定义中的定义模式或每个同义词定义中的常数值或模式是（或者直接包含）由该集合内某个定义所定义

的一个**模式**名字或一个**同义词**名字。

递归模式定义的集合是仅含模式定义的递归定义的集合（任何递归定义的集合必须是一个递归模式定义的集合，参见 5.1 节）。

如果某个模式是一个**模式**名字或包含一个**模式**名字，而此**模式**名字又是在一个递归模式定义的集合中定义的，则称该模式表示一个递归模式。递归模式定义的集合中的一条路径是一个**模式**名字表，其中每个名字均带有一个下标作为标志，并且：

- 该路径中所有名字都具有不同的定义；
- 对每个名字来说，其后继是其定义模式或直接出现在其定义模式中（最后一个名字的后继是第一个名字）；
- 每个名字的标志唯一地指出了在其前驱的定义模式中该名字的位置（第一个名字的前驱是最后一个名字）。

（例如，**NEWMODE** M=**STRUCT** (i M, n **REF** M)；包含两条路径 $\{M_i\}$ 和 $\{M_n\}$）。

当且仅当某条路径的名字表中至少有一个名字在被标志的地方出现于一个引用模式、行模式或过程模式中时，该路径是**安全的**。

静态条件：对任何递归模式定义的集合而言，它的所有路径必须是**安全的**（上面例子中第一条路径就不是**安全的**）。

例子：

1.15 *operand_mode = INT* *(1.1)*
3.3 *complex* = **STRUCT** *(re,im INT)* *(1.1)*

3.2.2 同义模式定义

语法：

〈同义模式定义语句〉::= (1)
SYNMODE 〈模式定义〉{,〈模式定义〉}*; (1.1)

语义：同义模式定义语句定义了一些**模式**名字，这些**模式**名字与它们的定义模式**同义**。

静态性质：同义模式定义语句内的某个模式定义中的一个定义性出现定义了一个**同义模式**名字（它也是一个**模式**名字）。当且仅当下列条件之一成立时，称一个**同义模式**名字与一个模式 M **同义**（反过来说，也称模式 M 与该**同义模式**名字**同义**）：

- 要么 M 是该**同义模式**名字的**定义**模式；
- 要么该**同义模式**名字的**定义**模式本身就是与 M **同义**的**同义模式**名字。

同义模式名字的**新颖性**是其**定义**模式的**新颖性**。

如果某个**同义模式**名字的**定义**模式是一个范围模式，则该**同义模式**名字的**祖先**模式是其**定义**模式的**祖先**模式。如果某个**同义模式**名字的**定义**模式是一个**变长**串模式，则该**同义模式**名字的**组成**模式是其**定义**模式的**组成**模式。

例子：

6.3 **SYNMODE** *month* = **SET** *(jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec);* *(1.1)*

3.2.3 新模式定义

语法：

〈新模式定义语句〉::= (1)

NEWMODE〈模式定义〉{,〈模式定义〉}*; (1.1)

语义：新模式定义语句定义一些**模式**名字，这些**模式**名字与它们的定义模式不**同义**。

静态性质：在新模式定义语句内某个模式定义中的一个定义性出现定义了一个**新模式**名字（它也是一个**模式**名字）。

新模式名字的**新颖性**是定义它的定义性出现。如果某个**新模式**名字的**定义**模式是一个范围模式，则引入一个虚拟模式 & *名字*作为该**新模式**名字的**祖先**模式：& *名字*的**定义**模式是该范围模式的**祖先**模式，并且，& *名字*的**新颖性**是该**新模式**名字的**新颖性**。

如果某个**新模式**名字的**定义**模式是一个**变长**串模式，则引入一个虚拟模式 & *名字*作为该**新模式**名字的**组成**模式，即 & *名字*的**定义**模式是该**变长**串模式的**组成**模式，且 & *名字*的**新颖性**是该**新模式**名字的**新颖性**。

如果某个新模式定义语句内的模式定义中的定义性出现是一个**准**定义性出现，则该**新模式**名字的**新颖性**是**准新颖性**，否则是**实新颖性**。

静态条件：如果某个**新模式**名字的**新颖性**是**准新颖性**，则至多有一个**实新颖性**被**新颖性约束**到该**准新颖性**之上。

例子：

11.6 **NEWMODE** *line = INT (1:8);* *(1.1)*

11.12 **NEWMODE** *board =* **ARRAY** *(line)* **ARRAY** *(column) square;* *(1.1)*

3.3 模式分类

语法：

〈模式〉::= (1)

[**READ**]〈非组合模式〉 (1.1)

| [**READ**]〈组合模式〉 (1.2)

〈非组合模式〉::= (2)

〈离散模式〉 (2.1)

| 〈幂集模式〉 (2.2)

| 〈引用模式〉 (2.3)

| 〈过程模式〉 (2.4)

| 〈实例模式〉 (2.5)

| 〈同步模式〉 (2.6)

| 〈输入输出模式〉 (2.7)

| 〈计时模式〉 (2.8)

语义：模式定义了值的集合以及允许在这些值上实施的运算。模式可以是**只读**模式，它表示不能为把一个值存入具有该模式的单元而访问该单元。模式具有**新颖性**，它表示该模式是不是由某个新模式定义语句

引入的。

静态性质：模式具有下列继承性质：

- 当它是一个显式或隐式的**只读**模式时，它是一个**只读**模式；
- 满足下列条件之一时，它是显式的**只读**模式：(1) 指定了 **READ**；(2) 它是一个**参数化**数组模式、**参数化**串模式或**参数化**结构模式，而其内相应的**原始**数组模式名字、**原始**串模式名字或**原始变体**结构模式名字分别是一个**只读**模式；
- 当一个模式本身不是一个显式的**只读**模式，且满足下列条件之一时，它是隐式的**只读**模式：
  — 它是某个**只读**数组模式的**元素**模式（参见3.12.3节）；
  — 它是某个**只读**结构模式的一个**域**模式或它是某个**参数化**结构模式的**标志**域模式（参见3.12.4节）。

模式与其内所含的非组合模式或组合模式具有相同的性质。在以下各节中，定义了预定义**模式**名字和不是*模式名字*的模式的性质，而*模式名字*的性质在3.2节中定义。除了**只读性质**外（参见12.1.1.1节），**只读**模式与相应的非**只读**模式具有相同的性质。

模式具有下列非继承性质：

- 它具有**新颖性**。它要么是 **nil**，要么是新模式定义语句中模式定义内的定义性出现。既不是*模式名字*，又不是 **READ** *模式名字*的模式的**新颖性**定义如下：
  — 如果它是一个**参数化**串模式、**参数化**数组模式或**参数化**结构模式，则其**新颖性**分别是其**原始**串模式、**原始**数组模式或**原始变体**结构模式的**新颖性**；
  — 如果它是一个范围模式，则其**新颖性**是其**祖先**模式的**新颖性**；
  — 否则，其**新颖性**是 **nil**。

  如果一个模式是一个*模式名字*或 **READ** *模式名字*，则其**新颖性**在 3.2.2 和 3.2.3 节中定义。
- 它具有**大小**，即由 *SIZE* (*&M*) 所传递的值，其中 *&M* 是一个与该模式**同义**的虚拟**同义模式**名字。

3.4 离散模式

3.4.1 概述

语法：

〈离散模式〉::= (1)
〈整数模式〉 (1.1)
| 〈布尔模式〉 (1.2)
| 〈字符模式〉 (1.3)
| 〈集合模式〉 (1.4)
| 〈范围模式〉 (1.5)

语义：离散模式定义了良序值的集合及其子集。

3.4.2 整数模式

语法：

〈整数模式〉::= (1)
〈*整数模式*名字〉 (1.1)

预定义名字：名字 *INT* 被预定义为一个**整数模式**名字。

语义：整数模式定义了位于由实现定义的上、下界之间的带符号整数的集合，并在其上定义了通常的次序和算术运算（参见5.3节）。某种实现可以定义其它具有不同上、下界的整数模式（例如，*LONG_INT*，*SHORT_INT*，…），这些整数模式也可以作为范围模式的**祖先**模式（参见13.2节）。一个整数值的内部表示是该整数值自身。

静态性质：整数模式具有下列继承性质：

- 整数模式具有**上界**和**下界**，它们分别是表示由该整数模式定义的最大值和最小值的字面值。它们是实现定义的；
- 整数模式的**值的个数**是**上界**－**下界**＋1。

例子：

1.5 INT (*1.1*)

3.4.3 布尔模式

语法：

〈布尔模式〉::= (*1*)

〈布尔模式名字〉 (*1.1*)

预定义名字：名字 *BOOL* 被预定义为一个**布尔模式**名字。

语义：布尔模式定义了逻辑真值（*TRUE* 和 *FALSE*），在其上可实施通常的布尔运算（参见 5.3 节）。*FALSE* 和 *TRUE* 的内部表示分别是整数 0 和 1。这种内部表示定义了布尔值的次序。

静态性质：布尔模式具有下列继承性质：

- 布尔模式具有**上界和下界**，它们分别是 *TRUE* 和 *FALSE*；
- 布尔模式的**值的个数**是 2。

例子：

5.4 BOOL (*1.1*)

3.4.4 字符模式

语法：

〈字符模式〉::= (*1*)

〈字符模式名字〉 (*1.1*)

预定义名字：名字 *CHAR* 被预定义为一个**字符模式**名字。

语义：字符模式定义了字符值，这些字符值由 CHILL 字符集描述（参见附录 A）。该字母表定义了字符之间的次序以及作为其内部表示的整数值。

静态性质：字符模式具有下列继承性质：

- 字符模式具有**上界**和**下界**，它们分别是表示 *CHAR* 所定义的最大值和最小值的字符字面值。
- 字符模式的**值的个数**是 256。

例子：

8.4 CHAR (*1.1*)

3.4.5 集合模式

语法：

〈集合模式〉::= (*1*)

SET（〈集合表〉） (*1.1*)

|〈集合模式名字〉 (*1.2*)

〈集合表〉::= (*2*)

〈编号集合表〉 (*2.1*)

|〈未编号集合表〉 (*2.2*)

〈编号集合表〉::= (*3*)

〈编号集合元素〉{，〈编号集合元素〉}* (*3.1*)

〈编号集合元素〉::= (*4*)

〈定义性出现〉＝〈整数字面值表达式〉 (*4.1*)

〈未编号集合表〉::= (*5*)

〈集合元素〉{，〈集合元素〉}* (*5.1*)

〈集合元素〉::= (*6*)

〈定义性出现〉 (*6.1*)

语义：集合模式定义了命名值或未命名值的集合。命名值由集合表中定义性出现所定义的名字表示，而未命名值是集合模式所定义的其它值。命名值的内部表示是与该命名值有关的整数值，这种内部表示定义了命名值之间的次序。

静态性质：集合表中一个定义性出现定义了一个**集合元素**名字。**集合元素**名字具有一个**集合**模式，它就是定义该**集合元素**名字的集合模式。

集合模式具有下列继承性质：

- 集合模式具有一个**集合元素**名字的集合，它就是由其集合表中那些定义性出现所定义的名字组成的集合；
- 集合模式的每个**集合元素**名字都附有一个内部表示值。对编号集合元素而言，该值就是其内整数字面值表达式所传递的值；对集合元素而言，按它们在未编号集合表中出现的位置，该值分别是0、1、2等等之一。例如，在集合模式 **SET**（*a*，*b*）中，*a* 附有内部表示值0，而 *b* 附有内部表示值1。
- 集合模式具有**上界**和**下界**，它们分别是其附有最大内部表示值和最小内部表示值的**集合元素**名字；
- 集合模式的**值的个数**是所有**集合元素**名字所附有的内部表示值中的最大值加1；
- 如果其内的集合表是一个编号集合表，则集合模式是一个**编号**集合模式；否则是一个**未编号**集合模式。

静态条件：对集合表中的每对整数字面值表达式 e_1、e_2 而言，*NUM*（e_1）与 *NUM*（e_2）必须传递不同的非负值。

例子：

11.7 **SET**（*occupied*，*free*） (*1.1*)

6.3　　month　　(*1.2*)

3.4.6　范围模式

语法：

〈范围模式〉::=　　(*1*)

〈离散模式名字〉(〈字面值范围〉)　　(*1.1*)

| **RANGE** (〈字面值范围〉)　　(*1.2*)

| **BIN** (〈整数字面值表达式〉)　　(*1.3*)

| 〈范围模式名字〉　　(*1.4*)

〈字面值范围〉::=　　(*2*)

〈下界〉:〈上界〉　　(*2.1*)

〈下界〉::=　　(*3*)

〈离散字面值表达式〉　　(*3.1*)

〈上界〉::=　　(*4*)

〈离散字面值表达式〉　　(*4.1*)

导出语法： **BIN** (*n*) 表示法是从 *INT* (*0*:2^n-1) 导出的。如 **BIN** (*2*+*1*) 表示 *INT* (*0*:*7*)。

语义： 范围模式定义了位于由字面值范围指定的边界之内（含边界）的值的集合。范围模式所定义的值取自特定的**祖先**模式所定义的值集，该**祖先**模式确定了范围值上实施的运算以及范围值间的次序。

静态性质： 范围模式具有下列非继承性质，即它具有一个**祖先**模式。定义如下：

- 如果该范围模式形如：

  〈离散模式名字〉(〈字面值范围〉)

  那么，如果该离散模式名字不是一个范围模式，则该**祖先**模式就是这个离散模式名字；否则是这个离散模式名字的**祖先**模式；
- 如果该范围模式形如：

  RANGE (〈字面值范围〉)

  则该**祖先**模式是字面值范围内上界和下界的类的**结果类**的**根**模式；
- 如果该范围模式是一个范围模式名字，且它是一个**同义模式**名字，则其**祖先**模式是该**同义模式**名字的**定义**模式的**祖先**模式；否则，它是一个**新模式**名字，因而其**祖先**模式是虚拟引进的**祖先**模式（参见3.2.3节）。

范围模式具有如下继承性质：

- 范围模式具有**上界**和**下界**，它们分别是表示由字面值范围内上界和下界所传递的值的字面值；
- 范围模式的**值的个数**是（*NUM* (*U*) −*NUM* (*L*) +1）所传递的值，其中 *U* 和 *L* 分别表示该范围模式的**上界**和**下界**；
- 当一个范围模式的**祖先**模式是一个**编号**集合模式时，该范围模式是**编号**范围模式。

静态条件： 上界和下界的类必须是**相容的**，并且，如果指定了离散模式名字，则它们都必须与该离散模式名字是**相容的**。

下界所传递的值必须小于等于上界所传递的值，并且，如果指明了离散模式名字，则这两个值都必须属于该离散模式名字所定义的值集。

在 **BIN** 情况下，整数字面值表达式必须传递一个非负值。

例子：

9.5 *INT (2:max)* *(1.1)*

11.12 *line* *(1.4)*

3.5 幂集模式

语法：

〈幂集模式〉::= *(1)*

POWERSET〈成员模式〉 *(1.1)*

| 〈幂集模式名字〉 *(1.2)*

〈成员模式〉::= *(2)*

〈离散模式〉 *(2.1)*

语义：幂集模式定义了一组值，这些值是由其成员模式所定义的值组成的集合。幂集值的取值范围是其成员模式定义的值集的所有子集。在幂集值上定义了常见的集合论运算（参见5.3节）。

静态性质：幂集模式具有下列继承性质：

·它具有**成员**模式，即指定的成员模式。

例子：

8.4 **POWERSET** *CHAR* *(1.1)*

9.5 **POWERSET** *INT (2:max)* *(1.1)*

9.6 *number_list* *(1.2)*

3.6 引用模式

3.6.1 概述

语法：

〈引用模式〉::= *(1)*

〈受限引用模式〉 *(1.1)*

| 〈自由引用模式〉 *(1.2)*

| 〈行模式〉 *(1.3)*

语义：引用模式定义了对**可引用的**单元的引用值（或称地址、描述符）。按照定义，受限引用值只能引用具有给定静态模式的单元，自由引用值可以引用具有任何静态模式的单元，而行值只能引用具有动态模式的单元。

间接引用操作定义在引用值之上（参见4.2.3、4.2.4和4.2.5节），其结果是传递被引用的单元。

当且仅当两个引用值都引用同一个单元，或它们都不引用单元（即它们都是值 *NULL*）时，这两个引用值是相等的。

3.6.2 受限引用模式

语法：

〈受限引用模式〉::= (1)
REF〈被引用模式〉 (1.1)
|〈受限引用模式名字〉 (1.2)
〈被引用模式〉::= (2)
〈模式〉 (2.1)

语义：受限引用模式定义对具有指定的被引用模式的单元的引用值。

静态性质：受限引用模式具有下列继承性质：

- 它具有一个**被引用**模式，这就是指定的被引用模式。

例子：

10.42 **REF** *cell* (1.1)

3.6.3 自由引用模式

语法：

〈自由引用模式〉::= (1)
〈自由引用模式名字〉 (1.1)

预定义名字：名字 *PTR* 被预定义为一个**自由引用模式**名字。

语义：自由引用模式定义了对任意静态模式单元的引用值。

例子：

19.8 PTR (1.1)

3.6.4 行模式

语法：

〈行模式〉::= (1)
ROW〈串模式〉 (1.1)
| **ROW**〈数组模式〉 (1.2)
| **ROW**〈变体结构模式〉 (1.3)
|〈行模式名字〉 (1.4)

语义：行模式定义了对具有动态模式的单元的引用值（具有动态模式的单元是具有带不能静态确定其值的某些参数的参数化模式的单元）。

行值可以引用：

- 具有不能静态确定其**串长度**的串单元；
- 具有不能静态确定其**上界**的数组单元；
- 具有不能静态确定其参数的参数化结构单元。

静态性质：行模式具有下列继承性质：

- 它具有一个**被引用原始**模式，该模式分别是串模式、数组模式或变体结构模式。

静态条件：变体结构模式必须是**可参数化的**。

例子：

8.6 **ROW CHARS** (*max*) *(1.1)*

3.7 过程模式

语法：

〈过程模式〉::= *(1)*

PROC ([〈参数表〉]) [〈结果说明〉]

[**EXCEPTIONS** 〈异常表〉] [**RECURSIVE**] *(1.1)*

| 〈过程模式名字〉 *(1.2)*

〈参数表〉::= *(2)*

〈参数说明〉 {, 〈参数说明〉}* *(2.1)*

〈参数说明〉::= *(3)*

〈模式〉 [〈参数属性〉] *(3.1)*

〈参数属性〉::= *(4)*

IN|**OUT**|**INOUT**|**LOC** [**DYNAMIC**] *(4.1)*

〈结果说明〉::= *(5)*

RETURNS (〈模式〉 [〈结果属性〉]) *(5.1)*

〈结果属性〉::= *(6)*

[**NONREF**] **LOC** [**DYNAMIC**] *(6.1)*

〈异常表〉::= *(7)*

〈异常名字〉 {, 〈异常名字〉}* *(7.1)*

语义：过程模式定义（**通用的**）过程值，即由**通用过程**名字表示的对象。**通用过程**名字是在过程定义语句中定义的。在动态上下文中，过程值指示代码块。过程模式使程序员能够动态地处理过程，如将它作为一个参数传递给其它过程，将它作为信息值发往某个缓冲区，或将它存入某个单元等等。

过程值能够被调用（参见6.7节）。

当且仅当两个过程值在同一个动态上下文中表示同一个过程，或者它们都不表示过程（即它们都是值 *NULL*）时，这两个过程值是相等的。

静态性质：过程模式具有如下继承性质：

- 它具有一个**参数说明**表，每个**参数说明**由一个模式以及可能有的参数属性组成。**参数说明**由参数表定义；
- 它具有任选的**结果说明**，该**结果说明**由一个模式和任选的结果属性组成。**结果说明**由结果说明定义；
- 它具有一个可能为空的**异常**名字表，这些**异常**名字是在异常表中提到的名字；
- 它具有**递归性**。若指明了 **RECURSIVE**，则**递归性**是**递归的**，否则由实现定义的缺省值来指出它是**递归的**还是**非递归的**。

静态条件：在异常表中提到的所有名字必须彼此不同。

仅当在参数说明或结果说明中指定 **LOC** 时，其中的模式才能具有**非值性质**。

如果在参数说明或结果说明中指定了 **DYNAMIC**，则其中的模式必须是**可参数化的**。

3.8 实例模式

语法：

〈实例模式〉::= (1)

〈实例模式名字〉 (1.1)

预定义名字：名字 *INSTANCE* 被预定义为一个**实例模式**名字。

语义：实例模式定义用于标识进程的值。每当创建一个新进程（参见5.2.14、6.13和11.1节）时就产生一个实例值，它用来唯一地标识所创建的进程。

当且仅当两个实例值标识同一个进程，或都不标识进程（即它们都是值 *NULL*）时，这两个实例值是相等的。

例子：

15.39 INSTANCE (1.1)

3.9 同步模式

3.9.1 概述

语法：

〈同步模式〉::= (1)

〈事件模式〉 (1.1)

| 〈缓冲区模式〉 (1.2)

语义：同步模式提供进程之间通信和同步的手段（参见第十一章）。在 CHILL 中不存在表示由同步模式所定义的值的表达式。因此，也不存在定义在这些值之上的运算。

3.9.2 事件模式

语法：

〈事件模式〉::= (1)

EVENT [(〈事件长度〉)] (1.1)

| 〈事件模式名字〉 (1.2)

〈事件长度〉::= (2)

〈整数字面值表达式〉 (2.1)

语义：事件模式单元提供进程之间的同步手段。在事件模式单元之上定义的操作有继续动作、延迟动作和延迟情况动作，这些动作分别在6.15、6.16和6.17节中叙述。

事件长度指明在具有该事件模式的单元之上成为被延迟的进程的最大数目。如果未指明事件长度，则该数目是没有限制的。

静态性质：事件模式具有如下继承性质：

- 它具有任选的**事件长度**，即由事件长度所传递的值。

静态条件：事件长度必须传递正值。

例子：

14.10 **EVENT** *(1.1)*

3.9.3 缓冲区模式

语法：

〈缓冲区模式〉::= (1)

BUFFER [(〈缓冲区长度〉)] 〈缓冲区元素模式〉 (1.1)

| 〈缓冲区模式名字〉 (1.2)

〈缓冲区长度〉::= (2)

〈整数字面值表达式〉 (2.1)

〈缓冲区元素模式〉::= (3)

〈模式〉 (3.1)

语义：缓冲区模式单元提供进程之间同步与通信的手段。定义在缓冲区单元之上的操作有发送动作、接收情况动作以及接收表达式，它们分别在6.18、6.19和5.3.9节中叙述。

缓冲区长度指明可存入具有该缓冲区模式的单元的值的最大数目。如果未指明缓冲区长度，则该数目是没有限制的。

静态性质：缓冲区模式具有下列继承性质：

- 它具有任选的**缓冲区长度**，即由缓冲区长度传递的值；
- 它具有**缓冲区元素**模式，它就是指定的缓冲区元素模式。

静态条件：缓冲区长度必须传送一个非负的值。

缓冲区元素模式不能具有**非值性质**。

例子：

16.30 **BUFFER** *(1) user_messages* *(1.1)*

16.34 *user_buffers* *(1.2)*

3.10 输入输出模式

3.10.1 概述

语法：

〈输入输出模式〉::= (1)

〈结合模式〉 (1.1)

| 〈访问模式〉 (1.2)

| 〈文本模式〉 (1.3)

语义：输入输出模式为实现第七章内定义的输入输出操作提供了手段。在 CHILL 中不存在表示由输入输出

模式所定义的值的表达式。当然也就不存在定义在这些值之上的运算。

例子：

20.17 ASSOCIATION (1.1)

3.10.2 结合模式

语法：

〈结合模式〉::= (1)

〈结合模式名字〉 (1.1)

预定义名字：名字 *ASSOCIATION* 被预定义为一个**结合模式**名字。

语义：结合模式单元提供体现与某个外界实体之间的联系的手段。这种联系在 CHILL 中称之为结合。结合能由内部子程序 *ASSOCIATE* 创建，也能由内部子程序 *DISSOCIATE* 撤消。

3.10.3 访问模式

语法：

〈访问模式〉::= (1)

ACCESS [(〈下标模式〉)] [〈记录模式〉 [**DYNAMIC**]] (1.1)

| 〈访问模式名字〉 (1.2)

〈记录模式〉::= (2)

〈模式〉 (2.1)

〈下标模式〉::= (3)

〈离散模式〉 (3.1)

| 〈字面值范围〉 (3.2)

导出语法：下标模式的字面值范围表示法是从离散模式 **RANGE**（字面值范围）导出的。

语义：访问模式单元提供了对一个文件进行定位以及从 CHILL 程序向其外界环境中的一个文件传送值或从外界环境中的某个文件向 CHILL 程序传送值的手段。

访问模式可以定义一个记录模式，这个记录模式定义了下述值的类的**根**模式，即那些通过一个具有该访问模式的单元可把它们从 CHILL 程序传送给某个外界文件或从某个外界文件传送给 CHILL 程序的值。当该访问模式的表示中指定了属性 **DYNAMIC** 或记录模式是一个**变长**串模式时，所传送的值的模式可以是动态的，即读写的记录的**大小**可以发生变化。在后一种情况下不必指明 **DYNAMIC**。

访问模式还可以定义一个下标模式，这种下标模式定义了为相应文件（的一部分）而设置的“窗口”的大小。通过该窗口可以从这个文件内随机地读出记录或向该文件随机地写入记录。可以通过连接操作对一个（可求下标的）文件设置这种窗口。如果未指明下标模式，则只能顺序地双向传送记录。

静态性质：访问模式具有下列继承性质：

- 它具有任选的**记录**模式。如果指定了记录模式，则**记录**模式就是该记录模式。如果还指明了 **DYNAMIC** 或记录模式是一个**变长**串模式，则**记录**模式是**动态记录**模式，否则是**静态记录**模式；
- 它具有任选的**下标**模式，即指定的下标模式。

静态条件：任选的记录模式不能具有**非值性质**。

如果指明了 **DYNAMIC**，则**记录**模式必须是**可参数化的**，且不能是一个**无标志**结构模式。

下标模式既不能是一个**编号**集合模式，也不能是一个**编号**范围模式。

例子：

20.18 **ACCESS** *(index_set) record_type* *(1.1)*
22.20 **ACCESS** *string* **DYNAMIC** *(1.1)*
20.18 *record_type* *(2.1)*
20.18 *index_set* *(3.1)*

3.10.4 文本模式

语法：

〈文本模式〉::= (1)
TEXT (〈文本长度〉) [〈下标模式〉] [**DYNAMIC**] (1.1)

〈文本长度〉::= (2)
〈整数字面值表达式〉 (2.1)

语义：文本模式单元提供把以人类可读格式表示的值从某个 CHILL 程序写入外部环境中某个文件，或从这种文件将上述值读到 CHILL 程序中的手段。文本模式单元具有一个**文本记录**子单元和一个**访问**子单元。**文体记录**子单元被初始化为一个空串。

文件模式具有**文本长度**，它定义了可被传送的那些记录的最大长度。文本模式还可能具有**下标**模式，它与访问模式的下标模式具有同样的含义。

静态性质：文本模式具有下列继承性质：

- 它具有**文本长度**，即由文本长度传递的值；
- 它具有**文本记录**模式，即
  CHARS (〈文本长度〉) **VARYING**；
- 它具有一个**访问**模式，即 **ACCESS** [(〈下标模式〉)] **CHARS** (〈文本长度〉) [**DYNAMIC**] (仅当文本模式中指定了〈下标模式〉和 **DYNAMIC** 时，它们才是**访问**模式的组成部分)。

例子：

26.8 **TEXT** (*80*) **DYNAMIC** *(1.1)*

3.11 计时模式

3.11.1 概述

语法：

〈计时模式〉::= (1)
〈时延模式〉 (1.1)
| 〈绝对时钟模式〉 (1.2)

语义：计时模式提供对进程实施时钟监控的手段，这在第九章中叙述。

计时值由一组内部子程序创建。在计时值上定义了关系运算。

3.11.2 时延模式

语法：

〈时延模式〉::= (1)

〈时延模式名字〉 (1.1)

预定义名字：名字 *DURATION* 被预定义为一个**时延模式**名字。

语义：时延模式定义表示一段时间的值。时延模式所定义的值的集合由实现确定。某种实现可以选用精度和值对来表示时延值。不同时延值按直观方式排序。

3.11.3 绝对时钟模式

语法：

〈绝对时钟模式〉::= (1)

〈绝对时钟模式名字〉 (1.1)

预定义名字：名字 *TIME* 被预定义为一个**绝对时钟模式**名字。

语义：绝对时钟模式定义表示某一时刻的值。绝对时钟模式所定义的值的集合由实现确定。按直观方式对绝对时钟值排序。

3.12 组合模式

3.12.1 概述

语法：

〈组合模式〉::= (1)

〈串模式〉 (1.1)

| 〈数组模式〉 (1.2)

| 〈结构模式〉 (1.3)

语义：组合模式定义组合值。组合值由若干能够被单独访问或获取的子值组成（参见4.2.6～4.2.10节和5.2.6～5.2.10节）。

3.12.2 串模式

语法：

〈串模式〉::= (1)

〈串类型〉(〈串长度〉) [**VARYING**] (1.1)

| 〈参数化串模式〉 (1.2)

| 〈串模式名字〉 (1.3)

〈参数化串模式〉::= (2)

〈原始串模式名字〉(〈串长度〉) (2.1)

| 〈参数化串模式名字〉 (2.2)

〈原始串模式名字〉::= (3)

〈串模式名字〉 (3.1)

〈串类型〉::= (4)

BOOLS (4.1)

| **CHARS** (4.2)

〈串长度〉::= (5)

〈整数字面值表达式〉 (5.1)

语义：**定长**串模式定义具有由该串模式所指定或隐含的长度的位串值或字符串值。**变长**串模式定义了一些位串值或字符值，它们的**实际长度**可在0到**串长度**范围内动态变化。**变长**串值的长度只能在运行时刻根据属性**实际长度**的值得知。对**定长**串模式而言，**定长**串值的**实际长度**总是等于**串长度**。字符串是一串字符值，而位串是一串布尔值。

串值要么是空串，要么是若干具有从0开始向上编号的串元素组成的值。

根据分量值的次序和下列定义，某个给定的串模式所定义的串值是良序的。

当且仅当两个串值 s 和 t 满足下列条件之一时，$s=t$：(1) 它们都是空串；(2) 它们都具有长度 l，且对所有的 i ($0\leqslant i<l$) 而言，$s(i)=t(i)$ 均成立。

当下列条件之一成立时，$s<t$：

- 存在一个下标 j，使得 $s(j)<t(j)$ 且 $s(0:j-1)=t(0:j-1)$；或
- *LENGTH* (s) <*LENGTH* (t) 且 $s=t$ (0 **UP** *LENGTH* (s))。

在串值上定义了并置运算。在位串值之上定义了通常的逻辑运算，它们是在对应的串元素上实施的运算（参见5.3节）。

静态性质：串模式具有下列继承性质：

- 它具有**串长度**，即串长度所传递的值；
- 它具有**上界**和**下界**，它们分别是（**串长度**－1）传递的值和0；
- 它是一个**位**串模式或一个**字符**串模式，这取决于串类型是 **BOOLS** 还是 **CHARS**，或取决于原始串模式名字是**位**串模式还是**字符**串模式；
- 如果一个串模式内指明了 **VARYING** 或其原始串模式名字是一个**变长**串模式，则该串模式是一个**变长**串模式，否则是一个**定长**串模式。

当且仅当一个串模式是一个参数化串模式时，该串模式是**参数化**串模式。

参数化串模式具有一个**原始**串模式，它就是由原始串模式名字表示的模式。

变长串模式具有如下非继承性质，即它具有**组成**模式，其定义如下：

- 如果该**变长**串模式形如：

  〈串类型〉(〈串长度〉) **VARYING**

  则其**组成**模式是〈串类型〉(〈串长度〉)；
- 如果该**变长**串模式形如：

  〈原始串模式名字〉(〈串长度〉)

  则其**组成**模式是 & 名字 (串长度)，其中，& 名字是一个虚拟引入的**同义模式**名字，它与原始串模式名字的**组成**模式**同义**；
- 如果该**变长**串模式是一个串模式名字，且它是一个**同义模式**名字，则其**组成**模式是该**同义模式**名字的定义模式的**组成**模式；否则，该串模式名字是一个**新模式**名字，故其**组成**模式是虚拟引入的**组成**模式（参见3.2.3节）。

静态条件：串长度必须传递一个非负的值。

直接包含在某个参数化串模式内的串长度所传递的值必须小于等于其原始串模式名字 的**串长度**。

本条件仅对不是虚拟引入的**参数化**串模式适用。

例子：

7.51 **CHARS** *(20)* *(1.1)*
22.22 **CHARS** *(20)* **VARYING** *(1.1)*

3.12.3 数组模式

语法：

〈数组模式〉::= (1)
ARRAY (〈下标模式〉{,〈下标模式〉}*)
〈元素模式〉{〈元素布局〉}* (1.1)
| 〈参数化数组模式〉 (1.2)
| 〈数组模式名字〉 (1.3)

〈参数化数组模式〉::= (2)
〈原始数组模式名字〉(〈下标上界〉) (2.1)
| 〈参数化数组模式名字〉 (2.2)

〈原始数组模式名字〉::= (3)
〈数组模式名字〉 (3.1)

〈下标上界〉::= (4)
〈离散字面值表达式〉 (4.1)

〈元素模式〉::= (5)
〈模式〉 (5.1)

导出语法：具有多于一个下标模式的数组模式（表示一个多维数组）是具有元素模式（其中该元素模式是一个数组模式）的数组模式的导出语法。例如，

ARRAY *(1:20,1:10) INT*

是从

ARRAY *(***RANGE** *(1:20)) ***ARRAY** *(***RANGE** *(1:10)) INT*

导出的。

仅当使用这种导出语法时，才允许出现多于一个的元素布局。而且，元素布局的出现个数必须小于等于下标模式的出现个数。在这种情况下，最左边的元素布局对应于最内层的元素模式，余类推。

语义：数组模式定义组合值，这种组合值是由若干其元素模式所定义的值构成的表。元素布局可以控制数组单元或数组值的物理布局（参见3.12.5节）。当且仅当两个数组值的所有对应元素值都相等时，它们才相等。

静态性质：数组模式具有下列继承性质：

- 它具有**下标**模式。如果该数组模式不是一个参数化数组模式，则**下标**模式就是指定的下标模式，否则是如下构造的范围模式。

& 名字（下界：上界）

其中：& 名字是一个与原始数组模式名字的**下标**模式**同义**的虚拟**同义模式**名字；下界是原始数组模式名字的**下标**模式的**下界**，而上界就是指定的下标上界。

- 它具有**上界**和**下界**，它们分别是其**下标**模式的**上界**和**下界**。
- 它具有**元素**模式。**元素**模式要么是 *M*，要么是 **READ** *M*，其中 M 要么是指定的元素模式，要么是原始数组模式名字的**元素**模式。当且仅当 *M* 不是一个**只读**模式，而该数组模式是一个**只读**模式时，**元素**模式是 **READ** *M*。当**元素**模式是 **READ** *M* 时，它是一个隐式的**只读**模式。
- 它具有**元素布局**。如果该数组模式是一个参数化数组模式，则**元素布局**是其原始数组模式名字的**元素布局**；否则，**元素布局**要么是指定的元素布局，要么是由实现定义的缺省值。在后一种情况下，它是 **PACK** 或 **NOPACK**；
- 它具有**元素个数**，即由（*NUM*（上界）－*NUM*（下界）＋1）传递的值，其中上界和下界分别是其**下标**模式的**上界**和**下界**；
- 当且仅当一个数组模式指定了步长形式的元素布局时，它是一个**被映射的**模式。

当且仅当一个数组模式是参数化数组模式时，该数组模式是**参数化的**。

参数化数组模式具有**原始**数组模式，即由原始数组模式名字所表示的模式。

静态条件：下标上界的类必须与原始数组模式名字的**下标**模式**相容**，且它所传递的值必须位于**下标**模式所确定的界限之内。

例子：

5.29	**ARRAY** *(1:16)* **STRUCT** *(c4, c2, c1 BOOL)*	*(1.1)*
11.12	**ARRAY** *(line)* **ARRAY** *(column) square*	*(1.1)*
11.17	*board*	*(1.3)*

3.12.4 结构模式

语法：

〈结构模式〉 ::= (1)

STRUCT （〈域〉{，〈域〉}*） (1.1)

| 〈参数化结构模式〉 (1.2)

| 〈结构模式名字〉 (1.3)

〈域〉 ::= (2)

〈固定域〉 (2.1)

| 〈替换域〉 (2.2)

〈固定域〉 ::= (3)

〈域名字定义性出现表〉〈模式〉[〈域布局〉] (3.1)

〈替换域〉 ::= (4)

CASE [〈标志表〉] **OF**

〈变体选择项〉{，〈变体选择项〉}*

[**ELSE** [〈变体域〉{，〈变体域〉}*]] **ESAC** (4.1)

〈变体选择项〉 ::= (5)

[〈情况标号说明〉]：[〈变体域〉{，〈变体域〉}*] (5.1)

〈标志表〉 ::= (6)

〈标志域名字〉{，〈标志域名字〉}* (6.1)

〈变体域〉::= (7)

〈域名字定义性出现表〉〈模式〉[〈域布局〉] (7.1)

〈参数化结构模式〉::= (8)

〈原始变体结构模式名字〉(〈字面值表达式表〉) (8.1)

| 〈参数化结构模式名字〉 (8.2)

〈原始变体结构模式名字〉::= (9)

〈变体结构模式名字〉 (9.1)

〈字面值表达式表〉::= (10)

〈离散字面值表达式〉{，〈离散字面值表达式〉}* (10.1)

导出语法：其域名字定义性出现表包含多于一个域名字定义性出现的固定域出现或变体域出现，它们分别是只带一个域名字定义性出现的多个固定域出现或变体域出现的导出语法。其中每个出现都具有指定的模式及任选的域布局，并分别具有域名字定义性出现表中的一个域名字定义性出现。在指明域布局的情况下，该域布局不能是定位。例如：

STRUCT *(I,J BOOL* **PACK***)*

是从

STRUCT *(I BOOL* **PACK***, J BOOL* **PACK***)*

导出的。

语义：结构模式定义组合值，这种组合值由一组值构成，并可以通过分量名字来选取这些分量值。每个分量的值都由附着在相应分量名字之上的模式定义。结构值可以存入（组合的）结构单元中，结构单元的分量名字用来访问相应子单元。结构值或结构单元的分量称为域，它们的名字称为**域**名字。

结构模式有**固定**结构模式、**变体**结构模式和**参数化**结构模式之分。

固定结构模式仅由固定域组成，这种域总是存在的，并且不必经过任何动态检查就能访问这种域。

变体结构包含变体域。这种域并非总是存在的。对**带标志变体**结构模式而言，变体域存在与否只能在运行时刻根据某些与之有关的、被称为**标志**域的固定域的值来判定。**无标志变体**结构模式不含**标志**域。由于在运行时刻**变体**结构模式的组合可能会发生变化，故变体结构模式单元的**大小**由所含的那些变体选择项中最大的（最坏情况）决定。

在某个替换域中，所选的变体选择项是其情况标号说明内所给定的值与标志表相匹配的变体选择项。如果不存在这样的变体选择项，且该替换域指明了 **ELSE** 部分，则选择跟在 **ELSE** 后面的那些变体域。

参数化结构模式是从某个**变体**结构模式构造出来的，构造方式是通过字面值表达式表静态指出选取该**变体**结构模式中的哪些变体选择项。从**参数化**结构模式的生成开始，其域的组合就确定下来，在运行期间也不再改变。如果存在**标志**域，则这些**标志域**是**只读的**，并自动地被赋以指定的初值。对**参数化**结构模式单元而言，可以在其说明或生成处精确地分配存储。应注意，在 CHILL 中还存在动态**参数化**结构模式，其语义在3.13.4节中定义。

通过域布局说明（参见3.12.5节）可以控制结构单元或结构值的物理布局。

当且仅当两个结构值的所有相应分量的值相等时，这两个结构值也是相等的。然而，如果这两个结构值是**无标志变体**结构值时，则比较的结果由实现定义。

静态性质：

通用静态性质：

结构模式具有下列继承性质：

- 如果某个结构模式是一个*结构模式*，且其内不直接包含替换域出现，则它是一个**固定**结构模式；
- 如果某个结构模式是一个*结构模式*，且其内至少包含一个替换域出现，则它是一个**变体**结构模式；
- 如果某个结构模式是由一个*参数化结构模式*表示的模式，则它是一个**参数化**结构模式；
- 结构模式具有一个**域**名字集合。针对不同情况在下面非通用性质中定义该集合。当且仅当一个名字是在某个*结构模式*的*固定域*或*变体域*内的*域名字定义性出现*表中被定义时，该名字称为**域名字**。

  结构模式的每个*固定域*或*变体域*都附着一个**域**模式，因而每个域的**域**名字也都附着一个**域**模式。它要么是 *M* 要么是 **READ** *M*，其中 *M* 是该*固定域*或*变体域*内指定的*模式*。当 *M* 本身不是一个**只读**模式，并且，要么该结构模式是一个**只读**模式，要么该域是**参数化**结构模式的**标志**域时，**域**模式是 **READ** *M*。当一个**域**模式是 **READ** *M* 时，该**域**模式是一个隐式的**只读**模式。

  一个给定的结构模式的每个*固定域*或*变体域*都附着一个**域布局**，因而每个域的**域**名字也都附有一个**域布局**。如果该*固定域*或*变体域*内含有*域布局*的话，其**域布局**就是此*域布局*，否则是缺省的域布局，即 **PACK** 或 **NOPACK**。
- 如果一个结构模式的所有**域**名字都附有*定位形式*的*域布局*，则该结构模式是一个**被映射的**模式。

固定结构模式的特殊静态性质：

固定结构模式还具有如下继承性质：

- 它具有一个**域**名字集合，即由该**固定**结构模式所含的那些*固定域*中任何*域名字定义性出现*表所定义的名字组成的集合。这些**域**名字是**固定域**名字。

变体结构模式的特殊静态性质：

变体结构模式还具有下列继承性质：

- 它具有一个**域**名字集合，这个集合是如下两个名字集合之并集：其一是由该**变体**结构模式所含的那些*固定域*中的任何*域名字定义性出现*表所定义的名字组成的集合；其二是它所含的那些*替换域*中的任何*域名字定义性出现*表所定义的名字组成的集合。*固定域*中的*域名字定义性出现*表所定义的那些**域**名字是该**变体**结构模式的**固定域**名字，而其它**域**名字是该**变体**结构模式的**变体域**名字。

  当且仅当**变体**结构模式的某个**域**名字出现在它的某个*替换域*的*标志表*中时，该**域**名字是**标志域**名字。未指定*标志表*的*替换域*是**无标志***替换域*。
- 如果一个**变体**结构模式的所有*替换域*出现都是**无标志***替换域*，则该**变体**结构模式是**无标志变体**结构模式，否则是**带标志变体**结构模式。
- 如果某个**变体**结构模式是**带标志变体**结构模式，或者是**无标志变体**结构模式，并且，在后一种情况下，对其中出现的每个*替换域*内的所有变体选择项出现都给出了*情况标号说明*，则该**变体**结构模式是一个**可参数化的变体**结构模式。
- **可参数化的变体**结构模式附有一个类表，其确定方式如下：
  - 如果它是一个**带标志变体**结构模式，则上述类表是由 M_i 值类组成的表，其中 M_i 是其每个**标志域**名字的模式，它们的排列次序与相应**域**名字在*固定域*中定义的顺序相同；
  - 如果它是一个**无标志变体**结构模式，则上述类表是按*替换域*出现的次序把每个*替换域*出现的**结果类表**连接而成的。*替换域*出现的**结果类表**是其内出现的*情况标号说明*表的**结果类表**（参见 12.3 节）。

参数化结构模式的特殊静态性质：

参数化结构模式还具有下列继承性质：

- 它具有**原始变体**结构模式，那就是*原始变体结构模式名字*所表示的模式；
- 它具有一个**域**名字集合，该集合是如下两个名字集合之并集：其一是由其**原始变体**结构模式的**固定域**名字组成的集合；其二是由其**原始变体**结构模式中满足下列条件的**变体域**名字组成的集合；这

种**变体域**名字是在由字面值表达式表定义的值表所选定的变体选择项出现中被定义的。

参数化结构模式的**标志域**名字集合是其**原始变体**结构模式的**标志域**名字集合；

- 它附有一个值表，即由字面值表达式表定义的值表；
- 如果一个**参数化**结构模式的**原始变体**结构模式是一个**带标志变体**结构模式，则它是一个**带标志参数化**结构模式，否则，该**参数化**结构模式是**无标志的**。

3.13.4 节将定义动态**参数化**结构模式的静态性质。

静态条件：

通用静态条件：

结构模式的所有**域**名字必须各不相同。

如果任一个域具有定位形式的域布局，则所有域都必须具有域布局，并且它们都必须是定位形式的域布局。

变体结构模式的特殊静态条件：

每个**标志域**名字必须是一个**固定域**名字，并且在程序中必须在所有其标志表中提到该**标志域**名字的那些替换域出现之前定义该**固定域**名字（因此，**标志**域在所有依赖于它的**变体**域之前出现）。**标志域**名字的模式必须是一个离散模式。

变体域的模式既不能具有**非值性质**，也不能具有**带标志参数化性质**。

在某个**变体**结构模式中，所有替换域出现要么都是**带标志的**，要么都是**无标志的**。对**无标志**替换域而言，要么所有变体选择项出现都省略情况标号说明，要么每个变体选择项都指明情况标号说明。

对一个**无标志变体**结构模式而言，如果它的某个替换域给出了情况标号说明，则它的所有替换域都必须具有情况标号说明。

替换域必须满足情况选择条件（参见 12.3 节），而且，凡是情况动作应满足的完备性、一致性以及相容性条件，替换域也应满足（参见 6.4 节）。如果含标志表的话，标志表中的每个**标志域**名字用作具有 M 值类的情况选择器，其中 M 是该**标志域**名字的模式。在**无标志**替换域情况下，忽略涉及情况选择器的检查。

对一个**可参数化的变体**结构模式而言，它所附有的类表中的任一个类不能是 **all** 类（**带标志变体**结构模式自动满足这个条件）。

参数化结构模式的特殊静态条件：

原始变体结构模式名字必须是**可参数化的**。

字面值表达式表中的**字面值**表达式的个数必须与原始变体结构模式名字的类表中的类的个数相同，并且，每个**字面值**表达式的类必须与上述类表中按位置对应的类**相容**。如果后者是 M 值类，则由**字面值**表达式所传递的值必须是由 M 定义的诸值之一。

例子：

```
3.3     STRUCT (re, im INT)                                          (1.1)
11.7    STRUCT (status SET (occupied, free),
                CASE status OF
                  (occupied): p piece,
                  (free):
                ESAC)                                                (1.1)
2.6     fraction                                                     (1.3)
11.7    status SET (occupied, free)                                  (3.1)
11.8    status                                                       (6.1)
11.9    p piece                                                      (7.1)
```

3.12.5 数组模式和结构模式的布局描述

语法：

〈元素布局〉::= (1)

PACK|**NOPACK**|〈步长〉 (1.1)

〈域布局〉::= (2)

PACK|**NOPACK**|〈定位〉 (2.1)

〈步长〉::= (3)

STEP (〈定位〉[,〈步长大小〉]) (3.1)

〈定位〉::= (4)

POS (〈字〉,〈开始位〉,〈长度〉) (4.1)

| **POS** (〈字〉[,〈开始位〉[:〈结束位〉]]) (4.2)

〈字〉::= (5)

〈整数字面值表达式〉 (5.1)

〈步长大小〉::= (6)

〈整数字面值表达式〉 (6.1)

〈开始位〉::= (7)

〈整数字面值表达式〉 (7.1)

〈结束位〉::= (8)

〈整数字面值表达式〉 (8.1)

〈长度〉::= (9)

〈整数字面值表达式〉 (9.1)

语义：通过在数组模式或结构模式中给出压缩或映射信息，可以控制相应数组或结构的布局。压缩信息是 **PACK** 或 **NOPACK**。在数组模式情况下，映射信息是步长；在结构模式情况下，映射信息是定位。如果在数组或结构模式中不出现*元素布局*或*域布局*，则此时总解释为指明了压缩信息，即 **PACK** 或 **NOPACK**。

如果对一个数组的元素或一个结构的域指定了 **PACK**，则意味着对该数组的元素或该结构的域所占用的存储空间的使用是优化的，而 **NOPACK** 意味着对该数组的元素或该结构的域的访问时间是优化的。**NOPACK** 还意味着该数组的元素或该结构的域是**可引用的**。

PACK 和 **NOPACK** 信息仅用于一层，即它只能用于该数组的元素或该结构的域，但不能用于该数组元素或结构域本身可能有的分量。布局信息总附着于最近的可以使用该布局信息的模式之上，并且该模式尚未附着布局信息。例如，如果缺省压缩信息为 **NOPACK**，则

STRUCT *(f* **ARRAY** *(0:1) m* **PACK***)*

等价于：

STRUCT *(f* **ARRAY** *(0:1) m* **PACK NOPACK***)*

通过在数组或结构模式中给出其分量的定位信息也可以精确控制该数组或结构的布局。以下列方式给出定位信息：

- 对数组模式而言，定位信息是该数组模式后面的步长形式的*元素布局*，它对该数组的所有元素均适

用；

- 对结构模式而言，定位信息是其每个域模式后面的定位形式的域布局，必须为每个域单独给出定位信息。

带定位的映射信息以字和位区距的形式给出。

形如：

POS（〈字〉，〈开始位〉，〈长度〉）

的定位定义了其值为

NUM（字）$*$ *WIDTH*＋*NUM*（开始位）

的位区距以及其值为 *NUM*（长度）的长度。其中 *WIDTH* 是（实现定义的）一个字所占的位数，字是一个整数字面值表达式。

当域布局内指定定位时，它表示相应的域从具有该模式的每个单元的起始处第 b_0 位开始（b_0 是该定位定义的位区距），并且，该域占据的位数是该定位定义的长度。

形如：

STEP（〈定位〉，〈步长大小〉）

的步长定义了一个位区距 b_i 序列：

$b_i = i * NUM$（步长大小）

i 取值范围为0到 $n-1$，其中 n 是该数组模式的**元素个数**。

该数组的第 j 个元素从具有该数组模式的单元的起始处后面第 $(p+b_j)$ 位开始，其中 p 是定位所指定的位区距。每个数组元素占据的位数是定位中指定的长度。

缺省规则

表示法：

POS（〈字〉，〈开始位〉：〈结束位〉）

语义上等价于：

POS（〈字〉，〈开始位〉，*NUM*（〈结束位〉）－*NUM*（〈开始位〉）＋1）

表示法：

POS（〈字〉，〈开始位〉）

语义上等价于：

POS（〈字〉，〈开始位〉，*BSIZE*）

其中 *BSIZE* 是带该定位的分量所需占据的最小位数。

表示法：

POS（〈字〉）

语义上等价于：

POS（〈字〉，0，*BSIZE*）

表示法：

STEP（〈定位〉）

语义上等价于：

STEP（〈定位〉，*SSIZE*）

其中 *SSIZE* 是定位中指明的或根据上述缺省规则从定位推导出来的〈长度〉。

静态性质：对具有某种数组模式的任何单元而言，该模式的元素布局如下确定了该数组单元（包括数组元素和数组片）的引用性：

- 要么所有子单元都是**可引用的**，要么全都不是；
- 如果元素布局是 **NOPACK**，则所有子单元都是**可引用的**。

对具有某个结构模式的单元而言，某个**域**名字的域布局如下确定由该**域**名字选出的结构域的引用性：

· 如果域布局是 **NOPACK**，则该**域**名字是**可引用的**。

静态条件：如果某个给定的数组模式的**元素**模式，或某个给定的结构模式的**域**名字的**域**模式本身是一个数组模式或结构模式，那么，如果给定的数组或结构模式是**被映射的**，则它也必须是一个**被映射的**模式。

如果指定了*字*、*开始位*、*结束位*、*长度*或*步长大小*，则它们都必须传递一个非负的值。此外，*开始位*和*结束位*所传递的值必须小于 WIDTH，即实现定义的一个字的位数，且*开始位*所传递的值必须小于等于*结束位*所传递的值。

每种实现都为每种模式定义了它的值所需占据的最小位数，这称为最小位占据。对离散模式而言，它是不小于以2为底的、该模式的**值的个数**的对数值的任何位数。对数组模式而言，它是该数组的具有最大下标的元素的位区距加上该元素自身所占据的位数。对结构模式而言，它是所占据的最高位的位区距。

对每个*定位*而言，其内指定的*长度*不能小于相应域或数组元素的模式的最小位占据。

对每个**被映射的**数组模式而言，*步长大小*不能小于*定位*中给定的或隐含说明的*长度*。

一致性和可行性

一致性：

当为一个结构模式的某个域指明*域布局*时，不能使该域占用同一对象的其它域已占据的任何位，下列情况除外：这两个**域**名字都是在同一个*替换域*中被定义的**变体域**名字，且这两个**变体域**名字不能都在同一个*变体选择项*中被定义，也不能都跟在 **ELSE** 后面。

可行性：

除了能从下面的规则推导出来的可行性要求外，不存在语言定义的可行性要求。这条规则是：任何（**可引用的**或非**可引用的**）单元的子单元的引用性仅由相 应的（元素或域）布局确定，这是单元的模式的一个性质。这一点给本身具有**可引用的**分量的分量的映射加了某些限制。

例子：

17.5 **PACK** *(1.1)*
19.14 **POS** *(1,0:15)* *(4.2)*

3.13 动态模式

3.13.1 概述

动态模式是其某些性质在运行时刻才能得知的模式。动态模式总是带有一个或多个运行时刻参数的参数化模式。为描述方便起见，本建议书引入了动态模式的虚拟表示。这些虚拟表示的前面冠以与符号（&），以便把它们与可以出现于 CHILL 程序正文中的实际表示区分开来。

3.13.2 动态串模式

虚拟表示：&〈*原始串模式名字*〉（〈*整数表达式*〉）

语义：动态串模式是带有不能静态确定其长度的参数化串模式。

静态性质：除下面描述的动态性质外，动态串模式与串模式具有同样的性质。

动态性质：

- 动态串模式具有动态**串长度**，即由整数表达式所传递的值；
- 动态串模式具有**上界**和**下界**，它们分别是由（**串长度**－1）所传递的值和 0。

3.13.3 动态数组模式

虚拟表示：&〈原始数组模式名字〉（〈离散表达式〉）

语义：动态数组模式是带不能静态确定其**上界**的参数化数组模式。

静态性质：除下面描述的动态性质外，动态数组模式与数组模式具有相同的性质。

动态性质：

- 动态数组模式具有动态**上界**和动态**元素个数**。前者是离散表达式所传递的值，后者是（*NUM*（离散表达式）－*NUM*（下界）＋1）所传递的值，其中下界是原始数组模式名字的**下界**。

3.13.4 动态参数化结构模式

虚拟表示：&〈原始变体结构模式名字〉（〈表达式表〉）

语义：动态**参数化**结构模式是具有不能静态确定其参数的**参数化**结构模式。

静态性质：除如下性质外，动态**参数化**结构模式与静态**参数化**结构模式具有相同的静态性质：

- 动态**参数化**结构模式的**域**名字集合是其**原始变体**结构模式的**域**名字集合。

动态性质：

- 动态**参数化**结构模式附有一个值表，它是由表达式表中的诸表达式传递的值组成的表。

4 单元及其访问

4.1 说明

4.1.1 概述

语法：

〈说明语句〉::= (1)

DCL 〈说明〉{, 〈说明〉}*; (1.1)

〈说明〉::= (2)

〈单元说明〉 (2.1)

| 〈单元等同说明〉 (2.2)

语义：说明语句说明了一个或多个名字，这个或这些名字用于访问一个单元。

例子：

```
6.9     DCL j INT := julian_day_number,
        d, m, y INT;                                  (1.1)
11.36   starting_square LOC := b(m.lin_1)(m.col_1)    (2.2)
```

4.1.2 单元说明

语法：

〈单元说明〉::= (1)

〈定义性出现表〉〈模式〉[**STATIC**] [〈初始化〉] (1.1)

〈初始化〉::= (2)

〈可达区初始化〉 (2.1)

| 〈生存期初始化〉 (2.2)

〈可达区初始化〉::= (3)

〈赋值号〉〈值〉[〈处理程序〉] (3.1)

〈生存期初始化〉::= (4)

INIT 〈赋值号〉〈常数值〉 (4.1)

语义：单元说明创建了若干个单元，其数量与定义性出现表中指定的定义性出现个数相等。

对带可达区初始化的单元说明而言，每当进入该说明所在的可达区时，就计算值（参见10.2节），并将它所传递的值赋给这个或这些单元。在计算值之前，这个或这些单元中包含**未定义的**值。

对带**生存期初始化**的单元说明而言，由常数值产生的值在这个或这些单元的生存期开始处被一次性地赋给这个或这些单元（参见10.2和10.9节）。

未指明初始化的单元说明语义上与带指定了**未定义的**值的生存期初始化的单元说明等价（参见5.3.1节）。

对附着具有**带标志参数化性质**或**非值性质**的模式的单元而言，**未定义的**值作为初值的含义如下：

- **带标志参数化性质**：所创建的**标志**域子单元被初始化为相应的参数值；
- **非值性质**：
  — 所创建的事件与/或缓冲区（子）单元被初始化为“空”，即该事件或缓冲区之上未附着任何被延迟的进程，且该缓冲区中没有任何信息；
  — 所创建的结合（子）单元被初始化为“空”，即它们未包含结合；
  — 所创建的访问（子）单元被初始化为“空”，即它们未被连接到某个结合之上；
  — 所创建的文本（子）单元具有一个**文本记录**子单元和一个**访问**子单元，前者被初始化为空串，后者被初始化为“空”，即它未被连接到某个结合之上。

STATIC 和处理程序的语义分别在10.9节和第8章中给出。

静态性质：单元说明中的一个定义性出现定义了一个**单元**名字。附着在该**单元**名字之上的模式是该单元说明内指定的模式。**单元**名字是**可引用的**。

静态条件：值或常数值的类必须与模式是**相容的**，并且，它所传递的值应是由模式定义的诸值之一，或是**未定义的**值。

如果模式具有**只读性质**，则必须指明初始化。如果模式具有**非值性质**，则不能指明可达区初始化。

动态条件：在指定可达区初始化的情况下，值相对模式而言应满足赋值条件（参见6.2节）。

例子：

```
5.7     k2, x, w, t, s, r BOOL                    (1.1)
6.9     := julian_day_number                      (3.1)
8.4     INIT := ['A':'Z']                         (4.1)
```

4.1.3 单元等同说明

语法：

〈单元等同说明〉::= (1)

〈定义性出现表〉〈模式〉**LOC** [**DYNAMIC**]

〈赋值号〉〈单元〉[〈处理程序〉] (1.1)

语义：单元等同说明创建了若干个用来访问指定的单元的访问名字，其数量与定义性出现表内指定的定义性出现个数相同。仅当指定了 **DYNAMIC** 时，指定的单元的模式才能是动态的。

如果单元是动态求值的，则每当进入该单元等同说明所在的可达区时就求值一次。在这种情况下，在由某个被说明的名字表示的访问的生存期内第一次求值前，该名字表示一个**未定义的**单元（参见10.2节和10.9节）。

静态性质：单元等同说明内的一个定义性出现定义了一个**单元等同**名字。如果未指定 **DYNAMIC**，则附着到该**单元等同**名字之上的模式是在该单元等同说明中指定的模式，否则是基于上述模式的动态参数化模式，它与指定的单元的模式具有相同的参数。

当且仅当指定的单元是**可引用的**时，**单元等同**名字是**可引用的**。

静态条件：如果在单元等同说明中指定了 **DYNAMIC**，则模式必须是**可参数化的**。如果指定了 **DYNAMIC**，则模式必须与单元的模式是**动态读相容的**；否则，模式必须与单元的模式是**读相容的**。

单元不能是满足下列条件的串元素或串片，即其内指定的串单元的模式是一个**变长**串模式。

动态条件：如果指定了 **DYNAMIC**，且上面提到的**动态读相容**关系检查失败，则引发异常 *RANGEFAIL* 或 *TAGFAIL*。

例子：

11.36 starting square **LOC** := *b(m.lin_1)(m.col_1)* (1.1)

4.2 单元

4.2.1 概述

语法：

〈单元〉::= (1)
〈访问名字〉 (1.1)
| 〈间接引用的受限引用〉 (1.2)
| 〈间接引用的自由引用〉 (1.3)
| 〈间接引用行〉 (1.4)
| 〈串元素〉 (1.5)
| 〈串片〉 (1.6)
| 〈数组元素〉 (1.7)
| 〈数组片〉 (1.8)
| 〈结构域〉 (1.9)
| 〈单元过程调用〉 (1.10)
| 〈单元内部子程序调用〉 (1.11)
| 〈单元转换〉 (1.12)

语义：单元是能够包含（存储）值的对象。为了存入或取出一个值必须访问单元。

静态性质：单元具有下列性质：

- 它具有一个模式，这在适当的章节中定义。该模式要么是静态的，要么是动态的；
- 它是**静态的**或不是**静态的**（参见 10.9 节）；
- 它是**区域内的**或是**区域外的**（参见 11.2.2 节）；
- 它是**可引用的**或不是**可引用的**。语言定义要求某些单元是**可引用的**，而另一些单元不是**可引用的**，这些条件在适当的章节中定义。实现可以将引用性扩展到其它单元，除非已明确规定不允许这么做。

4.2.2 访问名字

语法：

〈访问名字〉::= (1)
〈单元名字〉 (1.1)
| 〈单元等同名字〉 (1.2)
| 〈单元枚举名字〉 (1.3)
| 〈单元开域名字〉 (1.4)

语义：访问名字传递一个单元。访问名字是下列诸种名字之一：

- **单元**名字，即在某个单元说明中显式说明的名字或在某个不带 **LOC** 属性的形参中隐式说明的名字；
- **单元等同**名字，即在某个单元等同说明中显式说明的名字或在某个具有 **LOC** 属性的形参中隐式说明的名字；
- **单元枚举**名字，即在某个单元枚举中的循环计数器；
- **单元开域**名字，即在某个具有开域部分的循环动作内用作直接访问的**域**名字。

如果由某个单元开域名字表示的单元是某个无标志变体结构单元的一个变体域，则其语义由实现定义。

静态性质：访问名字附着的（可能是动态的）模式分别是相应单元名字、单元等同名字、单元枚举名字或单元开域名字的模式。

当且仅当访问名字是单元名字、**可引用的**单元等同名字、**可引用的**单元枚举名字或**可引用的**单元开域名字时，它是**可引用的**。

动态条件：当通过单元等同名字访问一个单元时，它不能表示一个**未定义的**单元。

当通过单元等同名字访问一个单元，且该单元是一个**变体**域时，必须满足关于该单元的变体域访问条件（参见4.2.10节）。通过单元开域名字访问一个单元时，如果它所表示的单元是一个**变体**域，且不满足相应变体域访问条件，则引发异常 *TAGFAIL*。

例子：

4.12 a (1.1)
11.39 *starting* (1.2)
15.35 *each* (1.3)
5.10 *c1* (1.4)

4.2.3 间接引用的受限引用

语法：

〈间接引用的受限引用〉::= (1)
〈受限引用原值〉-〉[〈模式名字〉] (1.1)

语义：间接引用的受限引用传递由指定的受限引用值所引用的单元。

静态性质：如果指定了模式名字，则间接引用的受限引用附着的模式就是该模式名字，否则是指定的受限引用原值的模式的**被引用**模式。间接引用的受限引用是**可引用的**。

静态条件：受限引用原值必须是**强**值。如果指明了任选的模式名字，则它必须与受限引用原值的模式的**被引用**模式是**读相容的**。

动态条件：被引用的单元的生存期不能已经结束。

如果受限引用原值传递值 *NULL*，则引发异常 *EMPTY*。

如果被引用的单元是一个**变体**域，则必须满足关于该单元的变体域访问条件（参见4.2.10节）。

例子：

10.54 P-〉 *(1.1)*

4.2.4 间接引用的自由引用

语法：

〈间接引用的自由引用〉::= *(1)*

〈自由引用原值〉-〉〈模式名字〉 *(1.1)*

语义：间接引用的自由引用传递由指定的自由引用值所引用的单元。

静态性质：间接引用的自由引用附着的模式是模式名字。间接引用的自由引用是**可引用的**。

静态条件：自由引用原值必须是**强**值。

动态条件：被引用的单元的生存期不能已经结束。

如果自由引用原值传递值 *NULL*，则引发异常 *EMPTY*。

模式名字必须与被引用的单元的模式是**读相容的**。

如果被引用的单元是一个**变体**域，则必须满足关于该单元的变体域访问条件（参见4.2.10节）。

4.2.5 间接引用行

语法：

〈间接引用行〉::= *(1)*

〈行原值〉-〉 *(1.1)*

语义：间接引用行传递由指定的行值所引用的单元。

静态性质：间接引用行附着的动态模式构造如下：

& 原始模式名字（〈参数〉{，〈参数〉}*）

其中原始模式名字是一个与行原值的模式的**被引用原始**模式**同义**的虚拟**同义模式**名字；按上述**被引用原始**模式是何种模式，其相应的参数分别是：

- 动态**串长度**，在串模式情况下；
- 动态**上界**，在数组模式情况下；
- 与被引用的参数化结构单元的模式有关的值表，在**变体**结构模式的情况下。

间接引用行是**可引用的**。

静态条件：行原值必须是**强**值。

动态条件：被引用的单元的生存期不能已经结束 。

如果行原值传递值 *NULL*，则引发异常 *EMPTY*。

如果被引用的单元是一个**变体**域，则必须满足关于该单元的变体域访问条件（参见4.2.10节）。

例子：

8.11 input-〉 *(1.1)*

4.2.6　串元素

语法：

〈串元素〉::=　(1)

〈串单元〉(〈开始元素〉)　(1.1)

〈开始元素〉::=　(2)

〈整数表达式〉　(2.1)

语义：串元素传递一个（子）单元，它就是指定的串单元中由*开始元素*指明的元素。

静态性质：*串元素*附着的模式是 *BOOL* 或 *CHAR*，这取决于串单元的模式是**位**串模式还是**字符**串模式。

如果串单元的模式是一个**变长**串模式，则*串元素*不是**可引用的**。

动态条件：如果下列关系不成立，则引发 *RANGEFAIL* 异常：

$0 \leqslant NUM$（开始元素）$\leqslant L-1$

其中 L 是串单元的**实际长度**。

例子：

18.16　string-〉(*i*)　(1.1)

4.2.7　串片

语法：

〈串片〉::=　(1)

〈串单元〉(〈左元素〉:〈右元素〉)　(1.1)

|〈串单元〉(〈开始元素〉**UP**〈片大小〉)　(1.2)

〈左元素〉::=　(2)

〈整数表达式〉　(2.1)

〈右元素〉::=　(3)

〈整数表达式〉　(3.1)

〈片大小〉::=　(4)

〈整数表达式〉　(4.1)

语义：串片传递一个（可能是动态的）串单元，它就是指定的串单元的由*左元素*和*右元素*或由*开始元素*和*片大小*所指明的那部分子单元。串片的（可能是动态的）长度根据指定的诸表达式确定。

如果串片内的*右元素*传递的值小于串片内*左元素*传递的值或串片内的*片大小*传递一个非正值，则该串片表示一个空串。

静态性质：串片上附着的（可能是动态的）模式是如下构造出的**参数化**串模式：

&名字（串大小）

其中如果串单元的（可能是动态的）模式是一个**定长**串模式，则 &名字是一个与上述模式**同义**的虚拟**同义模式**名字，否则是与上述模式的**分量**模式**同义**的虚拟**同义模式**名字；串大小要么是

NUM（右元素）−*NUM*（左元素）+1，

要么是

NUM（片大小）

所传递的值。然而，如果串片表示一个空串，则串大小是0。如果某个串片的串大小是一个**字面值**，即其内左元素和右元素都是**字面值**或片大小是**字面值**的话，则该串片附着的模式是静态的，否则是动态的。

如果串单元的模式是一个**变长**串模式，则串片不是**可引用的**。

静态条件：下列关系式必须成立：

$$0 \leqslant NUM\text{（左元素）} \leqslant L-1$$
$$0 \leqslant NUM\text{（右元素）} \leqslant L-1$$
$$0 \leqslant NUM\text{（开始元素）} \leqslant L-1$$
$$NUM\text{（开始元素）} + NUM\text{（片大小）} \leqslant L$$

其中 L 是串单元的**实际长度**。如果 L 和所有整数表达式所传递的值都是静态可知的，则可以静态检查这些条件。

动态条件：如果上述关系的动态检查部分失败，则引发异常 *RANGEFAIL*。

例子：

18.26 blanks (count : 9) *(1.1)*
18.23 string ->(scanstart **UP** *10)* *(1.2)*

4.2.8 数组元素

语法：

〈数组元素〉::= (1)
〈数组单元〉(〈表达式表〉) (1.1)
〈表达式表〉::= (2)
〈表达式〉{,〈表达式〉}* (2.1)

导出语法：表示法：(〈表达式表〉) 是 (〈表达式〉){(〈表达式〉)}*的导出语法，其中带括号表达式的个数与表达式表中的表达式的个数相同。因而，在严格语法形式下，数组元素仅有一个（下标）表达式。

语义：数组元素传递一个（子）单元，它就是由表达式指明的指定的数组单元的元素。

静态性质：数组元素附着的模式是数组单元的模式的**元素**模式。

如果数组单元的模式的**元素布局**是 **NOPACK**，则数组元素是**可引用的**。

静态条件：表达式的类必须与数组单元的模式的**下标**模式是**相容的**。

动态条件：如果下列关系式不成立，则引发异常 *RANGEFAIL*：

$$L \leqslant \text{表达式} \leqslant U$$

其中 L 和 U 分别是数组单元的模式的**下界**和（可能是动态的）**上界**。

例子：

11.36　b(m.lin_1)(m.col_1)　　*(1.1)*

4.2.9　数组片

语法：

〈数组片〉::=　　*(1)*

〈数组单元〉(〈下元素〉:〈上元素〉)　　*(1.1)*

|〈数组单元〉(〈首元素〉**UP**〈片大小〉)　　*(1.2)*

〈下元素〉::=　　*(2)*

〈表达式〉　　*(2.1)*

〈上元素〉::=　　*(3)*

〈表达式〉　　*(3.1)*

〈首元素〉::=　　*(4)*

〈表达式〉　　*(4.1)*

语义：数组片传递一个（可能是动态的）数组单元，它就是由下元素和上元素或由首元素和片大小指明的指定的数组单元的那部分子单元。数组片的**下界**等于指定的数组单元的**下界**，而数组片的（可能是动态的）**上界**由指定的诸表达式确定。

静态性质：数组片附着的（可能是动态的）模式是如下构造出的**参数化**数组模式：

&名字（下标上界）

其中&名字是一个与数组单元的（可能是动态的）模式**同义**的虚拟**同义模式**名字，而下标上界要么是其类与下元素和上元素的类都是**相容的**表达式，且其值满足下列关系式：

NUM（下标上界）=*NUM*（*L*）+*NUM*（上元素）−*NUM*（下元素）

要么是其类与首元素的类是**相容的**表达式，且其值满足下列关系式：

NUM（下标上界）=*NUM*（*L*）+*NUM*（片大小）−1

上面的*L*是数组单元的模式的**下界**。

当下标上界是**字面值**，即下元素和上元素都是**字面值**或片大小是**字面值**时，数组片附着的模式是静态的，否则是动态的。

如果数组单元的模式的**元素布局**是**NOPACK**，则数组片是**可引用的**。

静态条件：下元素和上元素的类或首元素的类必须与数组单元的**下标**模式是**相容的**。

下列关系式必须成立：

$L \leqslant$下元素$\leqslant$上元素$\leqslant U$

$1 \leqslant NUM$（片大小）$\leqslant NUM$（U）$-NUM$（L）$+1$

NUM（L）$\leqslant NUM$（首元素）$\leqslant NUM$（首元素）$+NUM$（片大小）$-1 \leqslant NUM$（U）

其中*L*和*U*分别是数组单元的模式的**下界**和**上界**。如果*U*和所有表达式的值都是静态可知的，则可以静态检查上述关系式。

动态条件：如果上述关系式的动态检查部分失败，则引发异常*RANGEFAIL*。

例子：

17.27　res（*0*：*count*−*1*）　　*(1.1)*

4.2.10 结构域

语法：

〈结构域〉::= (1)

〈结构单元〉.〈域名字〉 (1.1)

语义：结构域传递一个（子）单元，它就是由域名字指明的指定的结构单元的那个域。如果结构单元具有**无标志变体**结构模式，且域名字是一个**变体域**名字，则这种结构域的语义由实现定义。

静态性质：结构域的模式是域名字的模式。

如果域名字的**域布局**是 **NOPACK**，则结构域是**可引用的**。

静态条件：域名字必须属于结构单元的模式的**域**名字集合。

动态条件：结构单元不能表示下列两种单元：

- 它是一个**带标志变体**结构模式单元，且其相应的**标志域**的值指出具有域名字的域不存在；
- 它是一个动态**参数化**结构模式单元，且其相应的值表指出具有域名字的域不存在。

上面提到的条件被称为关于单元的变体域访问条件（注意：这种条件不包括异常的出现）。如果指定的结构单元不满足上述条件，则引发异常 *TAGFAIL*。

例子：

10.57 last-〉.info (1.1)

4.2.11 单元过程调用

语法：

〈单元过程调用〉::= (1)

〈单元过程调用〉 (1.1)

语义：单元过程调用传递被调用的过程所返送的单元。

静态性质：若单元过程调用的**结果说明**内未指明 **DYNAMIC**，则单元过程调用附着的模式是单元过程调用的**结果说明**内指定的模式，否则是一个基于上述模式的动态参数化模式，该模式与所传递的单元的模式具有相同的参数。

如果单元过程调用的**结果说明**内未指明 **NONREF**，则单元过程调用是**可引用的**。

动态条件：单元过程调用不能传递一个**未定义的**单元，且所传递的单元的生存期不能已经结束。

4.2.12 单元内部子程序调用

语法：

〈单元内部子程序调用〉::= (1)

〈单元内部子程序调用〉 (1.1)

语义：单元内部子程序调用传递被调用的内部子程序所返送的单元。

静态性质：单元内部子程序调用附着的模式是单元内部子程序调用的**结果说明**内指定的模式。

动态条件：单元内部子程序调用不能传递一个**未定义的**单元，且所传递的单元的生存期不能已经结束。

4.2.13　单元转换

语法：

〈单元转换〉::=　　(1)

〈模式名字〉(〈静态模式单元〉)　　(1.1)

语义：单元转换传递由静态模式单元表示的单元。然而，它忽略 CHILL 模式检验和相容性规则，并显式地把一个模式附着到该单元之上。

单元转换的精确的动态语义由实现定义。

静态性质：单元转换的模式是模式名字。

单元转换是**可引用的**。

静态条件：静态模式单元必须是**可引用的**。

必须满足下列关系：

SIZE（模式名字）＝*SIZE*（静态模式单元）

5 值及其运算

5.1 同义词定义

语法：

〈同义词定义语句〉::= (1)

SYN 〈同义词定义〉{,〈同义词定义〉}*; (1.1)

〈同义词定义〉::= (2)

〈定义性出现表〉[〈模式〉] = 〈常数值〉 (2.1)

导出语法：其定义性出现表由多于一个定义性出现组成的同义词定义是从几个同义词定义出现导出的，其中每个同义词定义对应于定义性出现表中的一个定义性出现，并且它们具有相同的常数值和模式（如果有的话）。例如，**SYN** i，$j=3$；是从 **SYN** $i=3$，$j=3$；导出的。

语义：同义词定义定义了表示指定的**常数**值的名字。

静态性质：同义词定义中的一个定义性出现定义了一个**同义词**名字。

如果指明了模式，则**同义词**名字的类是 M 值类，其中 M 就是指定的模式；否则，同义词名字的类是常数值的类。

当且仅当常数值是未**定义的**值（参见5.3.1节）时，**同义词名字**是**未定义的**。

当且仅当常数值是**字面值**时，**同义词**名字是**字面值**。

静态条件：如果指明了模式，则它必须与常数值的类是**相容的**，并且，常数值所传递的值必须是模式所定义的值之一。

同义词定义不能通过其它同义词定义或模式定义形成递归或相互递归，即任何递归定义的集合不能包含同义词定义（参见3.2.1节）。

例子：

```
1.17    SYN neutral_for_add = 0,                    (1.1)
        neutral_for_mult = 1;
2.18    neutral_for_add fraction = [ 0,1 ]          (2.1)
```

5.2 原值

5.2.1 概述

语法：

〈原值〉::= (1)

〈单元内容〉 (1.1)

| 〈值名字〉 (1.2)

| 〈字面值〉 (1.3)

| 〈多元组〉 (1.4)

| 〈值串元素〉 (1.5)
| 〈值串片〉 (1.6)
| 〈值数组元素〉 (1.7)
| 〈值数组片〉 (1.8)
| 〈值结构域〉 (1.9)
| 〈表达式转换〉 (1.10)
| 〈值过程调用〉 (1.11)
| 〈值内部子程序调用〉 (1.12)
| 〈启动表达式〉 (1.13)
| 〈零目运算符〉 (1.14)
| 〈带括号表达式〉 (1.15)

语义：原值是表达式的基本组成部分。某些原值具有动态类，即基于动态模式的类。就这些原值而言，其相容性检查只能在运行时刻完成，故检查失败将导致出现异常 *TAGFAIL* 或 *RANGEFAIL*。

静态性质：原值的类分别是单元内容、值名字、…的类。

当且仅当一个原值是一个**常数**值名字、字面值、**常数**多元组、**常数**表达式转换、**常数**值内部子程序调用或**常数**带括号表达式时，该原值是**常数**。

当且仅当原值是一个**字面值**的值名字、**离散**字面值或**字面值**的值内部子程序调用时，它是**字面值**。

5.2.2 单元内容

语法：

〈单元内容〉 ::= (1)
〈单元〉 (1.1)

语义：单元内容传递包含在指定的单元内的值。对该单元的访问是取出其中存储的值。

静态性质：单元内容的类是 M 值类，其中 M 是单元的（可能是动态的）模式。

静态条件：单元的模式不能具有**非值性质**。

动态条件：所传递的值不能是**未定义的**值。

例子：

3.7 *c2.im* (1.1)

5.2.3 值名字

语法：

〈值名字〉 ::= (1)
〈同义词名字〉 (1.1)
| 〈值枚举名字〉 (1.2)
| 〈值开域名字〉 (1.3)
| 〈值接收名字〉 (1.4)

| 〈通用过程名字〉 (1.5)

语义：值名字传送一个值。值名字是下列名字之一：

- **同义词**名字，即在某个同义词定义语句中定义的一个名字；
- **值枚举**名字，即由某个值枚举内循环计数器定义的名字；
- **值开域**名字，即在某个带开域控制的循环动作中作为值名字引入的**域**名字；
- **值接收**名字，即在某个接收情况动作中引入的名字；
- **通用过程**名字（参见10.4节）。

如果由值开域名字表示的值是某个无标志变体结构值的一个变体域，则其语义由实现定义。

静态性质：值名字的类分别是同义词名字、值枚举名字、值开域名字、值接收名字的类或是M导出类，其中M是通用过程名字的模式。

当且仅当一个值名字是一个同义词名字，且它是**字面值**时，该值名字是**字面值**。

如果值名字是同义词名字或通用过程名字，且该通用过程名字表示附着到某个未被分程序包围的过程定义之上的**过程**名字，则它是**常数**。

静态条件：同义词名字不能是**未定义的**。

动态条件：如果值开域名字所表示的值是一个**变体**域、且不满足关于该值的变体域访问条件，则对其求值引发异常 *TAGFAIL*。

例子：

10.12 *max* (1.1)
8.8 *i* (1.2)
15.54 *this_counter* (1.4)

5.2.4 字面值

5.2.4.1 概述

语法：

〈字面值〉::= (1)
〈整数字面值〉 (1.1)
| 〈布尔字面值〉 (1.2)
| 〈字符字面值〉 (1.3)
| 〈集合字面值〉 (1.4)
| 〈空字面值〉 (1.5)
| 〈字符串字面值〉 (1.6)
| 〈位串字面值〉 (1.7)

语义：字面值传递一个**常数值**。

静态性质：字面值的类分别是整数字面值、布尔字面值、…的类。如果字面值是整数字面值、布尔字面值、字符字面值或集合字面值，则它是**离散的**。

字母以及后随的撇号是字面值限定符（引导符），它们表示整数字面值或位串字面值的开始。字面值限定符有 *B'*、*D'*、*H'*、*O'*、*b'*、*d'*、*h'* 及 *o'*。

5.2.4.2　整数字面值

语法：

〈整数字面值〉 ::= (1)
　〈十进制整数字面值〉 (1.1)
| 〈二进制整数字面值〉 (1.2)
| 〈八进制整数字面值〉 (1.3)
| 〈十六进制整数字面值〉 (1.4)

〈十进制整数字面值〉 ::= (2)
　[{*D*|*d*} *'*] { 〈数字〉 |_}$^+$ (2.1)

〈二进制整数字面值〉 ::= (3)
　{*B*|*b*} *'* {*0*|*1*|_}$^+$ (3.1)

〈八进制整数字面值〉 ::= (4)
　{*O*|*o*} *'* { 〈八进制数字〉 |_}$^+$ (4.1)

〈十六进制整数字面值〉 ::= (5)
　{*H*|*h*} *'* { 〈十六进制数字〉 |_}$^+$ (5.1)

〈十六进制数字〉 ::= (6)
　〈数字〉 |*A*|*B*|*C*|*D*|*E*|*F*|*a*|*b*|*c*|*d*|*e*|*f* (6.1)

〈八进制数字〉 ::= (7)
　0|*1*|*2*|*3*|*4*|*5*|*6*|*7* (7.1)

语义：整数字面值传递非负的整数值。CHILL 提供了通常的十进制（基数为10）以及二进制（基数为2）、八进制（基数为8）和十六进制（基数为16）表示法。下线符（_）并不重要，即它仅用来提高程序可读性，并不影响所表示的值。

静态性质：整数字面值的类是 *INT* 导出类。整数字面值是**常数**和**字面值**。

静态条件：跟在撇号（'）后面的串以及整个整数字面值不能仅由下线符组成。

例子：

6.11　*1_721_119* (1.1)
　　D'1_721_119 (1.1)
　　B'101011_110100 (1.2)
　　O'53_64 (1.3)
　　H'AF4 (1.4)

5.2.4.3　布尔字面值

语法：

〈布尔字面值〉 ::= (1)

〈布尔字面值名字〉 (1.1)

预定义名字：名字 *FALSE* 和 *TRUE* 被预定义为**布尔字面值**名字。

语义：布尔字面值传递布尔值。

静态性质：布尔字面值的类是 *BOOL* 导出类。布尔字面值是**常数**和**字面值**。

例子：

5.46 FALSE (1.1)

5.2.4.4 字符字面值

语法：

〈字符字面值〉::= (1)

'〈字符〉|〈控制序列〉' (1.1)

语义：字符字面值传递一个字符值。除了可打印的表示外，还可使用控制序列表示。

静态性质：字符字面值的类是 *CHAR* 导出类。字符字面值是**常数**和**字面值**。

静态条件：字符字面值中的控制序列仅能表示一个字符。

例子： *7.9 'M'* (1.1)

5.2.4.5 集合字面值

语法：

〈集合字面值〉::= (1)

〈集合元素名字〉 (1.1)

语义：集合字面值传递一个集合值。集合字面值是在某个集合模式中定义的名字。

静态性质：集合字面值的类是 M 导出类，其中 M 是集合元素名字附着的**集合**模式。集合字面值是**常数**和**字面值**。

例子：

6.51 dec (1)

11.78 king (1.1)

5.2.4.6 空字面值

语法：

〈空字面值〉::= (1)

〈空字面值名字〉 (1.1)

预定义名字：名字 *NULL* 被预定义为一个**空字面值**名字。

语义：空字面值传递一个空引用值、空过程值或空实例值。空引用值是不引用任何单元的值，空过程值是不指示任何过程的值，而空实例值是不标识任何进程的值。

静态性质：空字面值的类是 **null** 类。空字面值是**常数**。

例子：

10.43 NULL (*1.1*)

5.2.4.7 字符串字面值

语法：

〈字符串字面值〉::= (*1*)

”{〈非保留字符〉|〈引号〉|〈控制序列〉}*” (*1.1*)

〈引号〉::= (*2*)

”” (*2.1*)

〈控制序列〉::= (*3*)

^ (〈整数字面值表达式〉{,〈整数字面值表达式〉}*) (*3.1*)

|^ 〈非专用字符〉 (*3.2*)

|^ ^ (*3.3*)

语义：字符串字面值传递一个字符串值，其长度可以为 *0*。字符串字面值是一个值表，该值表中的每个值给出该字符串的相应元素的值，方式是按其下标递增顺序从左到右地给出每个元素的值。为了在字符串字面值中表示引号字符（”），必须将它写两次（””）。

除了可打印的表示外，还可使用控制序列表示。形如上箭头字符（^）后跟一个开圆括号的控制序列表示一串字符，而这些字符的表示就是其内的整数字面值表达式，形如 ^ ^ 的控制序列表示字符 ^ 本身；而形如 ^ 后跟非专用字符的控制序列就表示一个字符，该字符的表示是将其内指定的非专用字符的内部表示的 b7 位从逻辑意义上忽略后得到的（参见附录 A）。

静态性质：字符串字面值的**串长度**是非保留字符、引号以及由控制序列出现所表示的那些字符的总数。

字符串字面值的类是 **CHARS**（*n*）**导出类**，其中 *n* 是字符串字面值的**串长度**。字符串字面值是**常数**。

静态条件：由控制序列内的某个整数字面值表达式传递的值必须属于下列值范围内，即它是由在 CHILL 字符集（参见附录 A）中定义的所有字符的表示所定义的范围。

例子：

8.20 ”*A−B<ZAA9K’*” (*1.1*)

5.2.4.8 位串字面值

语法：

〈位串字面值〉::= (*1*)

〈二进制位串字面值〉 (*1.1*)

| 〈八进制位串字面值〉 (1.2)
| 〈十六进制位串字面值〉 (1.3)

〈二进制位串字面值〉::= (2)
{B|b}' {0|1|_}*' (2.1)

〈八进制位串字面值〉::= (3)
{O|o}' {〈八进制数字〉|_}*' (3.1)

〈十六进制位串字面值〉::= (4)
{H|h}' {〈十六进制数字〉|_}*' (4.1)

语义：位串字面值传递一个位串值，其长度可以是0。可以使用二进制、八进制或十六进制表示法。下线符(_)并不重要，即它仅用来改善程序可读性，并不影响所表示的值。

位串字面值是一个值表，它给出了该位串的每个串元素的值，方式是按其下标递增顺序从左到右地给出每个串元素的值。

静态性质：位串字面值的**串长度**要么是 B' 或 b' 后面出现的0和1的总数，要么是 O' 或 o' 后面出现的八进制数字的总数的三倍，要么是 H' 或 h' 后面出现的十六进制数字的总数的四倍。

位串字面值的类是 **BOOLS** (*n*) 导出类，其中 n 是位串字面值的**串长度**。位串字面值是**常数**。

例子：

B'101011_110100' (1.1)
O'53_64' (1.2)
H'AF4' (1.3)

5.2.5 多元组

语法：

〈多元组〉::= (1)
[〈模式名字〉](:{〈幂集多元组〉|
〈数组多元组〉|〈结构多元组〉}:) (1.1)

〈幂集多元组〉::= (2)
[{〈表达式〉|〈范围〉}{,{〈表达式〉|〈范围〉}}*] (2.1)

〈范围〉::= (3)
〈表达式〉:〈表达式〉 (3.1)

〈数组多元组〉::= (4)
〈无标号数组多元组〉 (4.1)
|〈有标号数组多元组〉 (4.2)

〈无标号数组多元组〉::= (5)
〈值〉{,〈值〉}* (5.1)

〈有标号数组多元组〉::= (6)
〈情况标号表〉:〈值〉{,〈情况标号表〉:〈值〉}* (6.1)

〈结构多元组〉::= (7)
〈无标号结构多元组〉 (7.1)
|〈有标号结构多元组〉 (7.2)

〈无标号结构多元组〉 ::= (8)

〈值〉 {, 〈值〉}* (8.1)

〈有标号结构多元组〉 ::= (9)

〈域名字表〉 : 〈值〉 {, 〈域名字表〉 : 〈值〉}* (9.1)

〈域名字表〉 ::= (10)

. 〈域名字〉 {,. 〈域名字〉}* (10.1)

导出语法：多元组的开、闭括号［和］分别是对应（:和:）的导出语法。这一点在语法中未表示出来，以免与作为元语言符号的方括号的用法混淆。

语义：多元组传递一个幂集值、数组值或结构值。

如果多元组是一个幂集值，则它由一个表达式和/或范围的表组成，范围或表达式表示在该幂集值中的若干或一个成员值。范围所表示的那些值是位于该范围内两个表达式所传递的值之间的所有值（包括上述两个表达式所传递的值）。如果第二个表达式所传递的值小于第一个表达式所传递的值，则该范围为空，即它不表示任何值。幂集多元组可以表示一个空幂集值。

如果多元组是一个数组值，则它是由关于该数组的每个数组元素的值组成的值表，这些值可能带有标号：在无标号数组多元组中，按其下标递增序给出每个数组元素的值；在有标号数组多元组中，在某个值前面出现的情况标号表中已指明其下标的那些数组元素被给予这个值。有标号数组多元组可被用作其内有许多相同值的较大的数组多元组的缩写手段。标号 **ELSE** 表示所有未被显式提到的下标值。标号 * 表示所有下标值（详见12.3节）。

如果多元组是一个结构值，则它是一个（可能带标号的）值的集合，其中每个值给出了该结构值的相应域的值。在无标号结构多元组中，按每个域在该结构值所附着的结构模式内指定的顺序依次给出它们的值。在有标号结构多元组中，某个域名字表中指出其域名字的那个域或那些域的值是该域名字表后面指出的值。

并未定义多元组中出现的诸表达式和值的求值次序；可以认为是以任意次序对它们求值的。

静态性质：多元组的类是 M 值类。如果指明了模式名字，则 M 就是该模式名字；否则，根据下面所列规则，M 依赖于该多元组出现处的上下文：

- 如果多元组是某个单元说明内初始化中的值或常数值，则 M 是该单元说明内指定的模式；
- 如果多元组是某个单赋值动作内赋值号右端的值，则 M 是赋值号左端的单元的（可能是动态的）模式；
- 如果多元组是某个其内指明了模式的同义词定义中的常数值，则 M 就是那个模式；
- 如果多元组是某个过程调用或启动表达式中的一个实参，且在相应参数说明中未指定 **DYNAMIC**，则 M 是相应参数说明内指定的模式；
- 如果多元组是某个返回动作或结果动作中指定的值，则 M 是该返回动作或结果动作的**过程**名字的**结果说明**的模式（参见6.8节）；
- 如果多元组是某个发送动作中指定的一个值，则 M 是在相应信号名字的信号定义中指定的相应模式，或是相应缓冲区单元的模式的**缓冲区元素**模式；
- 如果多元组是某个数组多元组内的一个表达式，则 M 是该数组多元组的模式的**元素**模式；
- 如果多元组是某个无标号结构多元组或某个有标号结构多元组内指定的一个表达式，同时有标号结构多元组内位于该多元组前面的域名字表仅含一个域名字，则 M 就是指定多元组为上述相应结构多元组中的域的模式；
- 如果多元组是某个 *GETSTACK* 或 *ALLOCATE* 内部子程序中指定的值，则 M 是相应模式变元所表示的模式。

当且仅当某个多元组中出现的每个值或表达式都是**常数**时，该多元组是**常数**。

静态条件：只有在上述上下文中才能省去任选的*模式名字*。据所指定的*多元组*类型（即是*幂集多元组*、*数组多元组*还是*结构多元组*），应满足相应的下列相容性要求：

a. *幂集多元组*

1. *多元组*的模式必须是一个幂集模式；
2. 每个*表达式*的类必须与*多元组*的模式的**成员**模式是**相容的**；
3. 对**常数**幂集多元组而言，其内每个*表达式*所传递的值必须是上述**成员**模式所定义的值之一。

b. *数组多元组*

1. *多元组*的模式必须是一个数组模式；
2. 每个*值*的类必须与*多元组*的模式的**元素**模式是**相容的**；
3. 在*无标号数组多元组*情况下，其内出现的*值*的个数必须与该*多元组*的数组模式的**元素个数**相等；
4. 在*有标号数组多元组*情况下，由*情况标号表*出现组成的表必须满足情况选择条件（参见 12.3 节）。上述表的**结果类**必须与该*多元组*的模式的**下标**模式是**相容的**。情况标号说明的表必须是**完备的**；
5. 在*有标号数组多元组*情况下，在某个*情况标号表*中由每个情况标号显式指出的值必须是该*多元组*的**下标**模式所定义的值；
6. 在*无标号数组多元组*中，至少有一个值出现是*表达式*；
7. 对其模式的**元素**模式是某个离散模式的**常数***数组多元组*而言，每个指定的*值*必须传递由那个**元素**模式所定义的值或**未定义的**值。

c. *结构多元组*

1. *多元组*的模式必须是一个结构模式；
2. 该模式不能是具有**不可见**的**域**名字的结构模式（参见 12.2.5 节）。

   在*无标号结构多元组*情况下：

   - 如果该*多元组*的模式既不是一个**变体**结构模式，也不是一个**参数化**结构模式，则：
     3. 其内出现的*值*的个数必须与该*多元组*的模式的**域**名字集合中**域**名字的个数相等；
     4. 每个*值*的类必须与该*多元组*的模式的（按位置）对应的**域**名字的模式是**相容的**。
   - 如果该*多元组*的模式是一个**带标志变体**结构模式或**带标志参数化**结构模式，则：
     5. 为某个**标志**域指定的*值*必须是一个*离散字面值表达式*；
     6. 其内出现的*值*的个数必须与由**标志**域所指定的诸*离散字面值表达式*传递的值指出存在的那些**域**名字的个数相等；
     7. 每个*值*的类必须与相应**域**名字的模式是**相容的**。
   - 如果该*多元组*的模式是一个**无标志变体**结构模式或**无标志参数化**结构模式，则：
     8. 不允许*无标号结构多元组*。

   在*有标号结构多元组*情况下：

   - 如果该*多元组*的模式既不是一个**变体**结构模式，也不是一个**参数化**结构模式，则：
     9. 该*多元组*的模式的**域**名字集合中的每个**域**名字必须在该*多元组*内的某个*域名字表*中出现一次且仅出现一次，而且，**域**名字出现的次序必须与它们在该*多元组*的模式内出现的次序相同；
     10. 每个*值*的类必须与位于该*值*前面的*域名字表*中指定的每个**域**名字的模式是**相容的**。
   - 如果该*多元组*的模式是一个**带标志变体**结构模式或**带标志参数化**结构模式，则：
     11. 为每个**标志**域指定的*值*必须是一个*离散字面值表达式*；
     12. 仅能指明对应于满足下列条件的域的**域**名字，即为诸**标志**域所指定的诸*离散字面值表达式*传递的值指出存在的那些域，并且，必须指明所有这样的**域**名字，还要求它们的出现次序与它们在该*多元组*的模式中出现的次序相同；
     13. 每个*值*的类必须与位于该*值*前面的*域名字表*中指定的任何**域**名字的模式都是**相容的**。

如果该多元组的模式是一个**无标志变体**结构模式或**无标志参数化**结构模式，则：

14. 在同一个替换域中定义的、且在域名字表内提到的那些**域**名字必须都是在同一个变体选择项中定义的，或都是在 **ELSE** 后面定义的。被选中的变体选择项的所有**域**名字或在 **ELSE** 后面定义的所有**域**名字必须按它们在该多元组的模式内出现的次序在该多元组中出现一次且仅出现一次；
15. 每个值的类必须与位于该值前面的域名字表中指定的任何**域**名字的模式都是**相容的**；

16. 如果该多元组的模式是一个**带标志参数化**结构模式，则由为所有**标志**域指定的诸离散字面值表达式所传递的值组成的值表必须与该多元组的模式的值表相同；
17. 对**常数**结构多元组而言，为每个具有离散模式的域指定的值必须传递由相应**域**模式定义的值或**未定义的**值；
18. 至少有一个值出现是表达式。

不允许多元组内出现两个这样的值，即其中之一是**区域外的**，而另一个是**区域内的**（参见 11.2.2 节）。

动态条件：在幂集多元组、数组多元组或 结构多元组情况下，其内指定的任何值分别相对其**成员**模式、**元素**模式或相应的**域**模式而言必须满足赋值条件（参见 6.2 节中所述的条件 a2、b2、c4、c7、c10、c13 及 c15）。

如果多元组具有动态数组模式，则当不满足条件 b3 和 b5 之一时引发异常 *RANGEFAIL*。

如果多元组具有动态**参数化**结构模式，则当不满足条件 c14 和 c16 之一时，引发异常 *TAGFAIL*。

多元组所传递的值不能是**未定义的**。

例子：

9.6	*number_list* [ ]	*(1.1)*
9.7	[*2:max*]	*(2.1)*
8.26	[(**'A'**):*3*,(**'B'**,**'K'**,**'Z'**):*1*,(**ELSE**):*0*]	*(6.1)*
17:5	[(***):' ']	*(6.1)*
12.35	(:*NULL*,*NULL*,*536*:)	*(7.1)*
11.18	[.*status:occupied*,.*p*:[*white*,*rook*]]	*(9.1)*

5.2.6 值串元素

语法：

〈值串元素〉::= *(1)*

〈串原值〉(〈开始元素〉) *(1.1)*

注意：如果串原值是串单元，则上述语法结构是二义的，此时将它解释为一个串元素（参见 4.2.6 节）。

语义：值串元素传递一个值，它就是由开始元素指出的指定的串值的元素。

静态性质：值串元素的类是 *BOOL* 值类或 *CHAR* 值类，这取决于串原值的模式是一个**位**串模式还是一个**字符**串模式。

动态条件：值串元素所传递的值不能是**未定义的**。

如果下面的关系式不成立，则引发异常 *RANGEAIL*：

$0 \leqslant NUM$（开始元素）$\leqslant L-1$

其中 L 是串原值的**实际长度**。

5.2.7 值串片

语法：

〈值串片〉::= (1)

〈串原值〉(〈左元素〉:〈右元素〉) (1.1)

| 〈串原值〉(〈开始元素〉**UP**〈片大小〉) (1.2)

注意：如果串原值是一个串单元，则该语法结构是二义的，此时将它解释为一个串片（参见4.2.7节）。

语义：值串片传递一个（可能是动态的）串值，它就是指定的串值中由左元素和右元素或由开始元素和片大小指示的那一部分。值串片的（可能是动态的）长度由指定的诸表达式确定。

满足下列条件的值串片表示一个空串：其内右元素所传递的值小于左元素所传递的值，或其内片大小传递一个小于等于0的值。

静态性质：如果串原值是**强**值，则值串片的（可能是动态的）类是M值类，否则是M导出类，其中M是如下构造出的**参数化**串模式：

& 名字（串大小）

其中，如果串原值的（可能是动态的）**根**模式是一个**定长**串模式，则 & 名字是一个与上述**根**模式**同义**的虚拟**同义模式**名字；否则，串原值的**根**模式是一个**变长**串模式，且 & 名字是一个与串原值的**根**模式的**分量**模式**同义**的虚拟**同义模式**名字。串大小要么是

NUM（右元素）$-$*NUM*（左元素）$+1$

要么是

NUM（片大小）。

然而，如果该值串片表示一个空串，则串大小是0。如果串大小是**字面值**，即左元素和右元素都是**字面值**或片大小是**字面值**，则值串片的类是静态的，否则是动态的。

静态条件：下列关系式必须成立：

$0 \leqslant NUM$（左元素）$\leqslant L-1$

$0 \leqslant NUM$（右元素）$\leqslant L-1$

$0 \leqslant NUM$（开始元素）$\leqslant L-1$

NUM（开始元素）$+$*NUM*（片大小）$\leqslant L$

其中 L 是串原值的**实际长度**。如果 L 和所有整数表达式的值都是静态可知的，则可以静态检查上述关系式。

动态条件：值串片所传递的值不能是**未定义的**。

如果上述关系式检查的动态部分失败，则引发 *RANGEFAIL* 异常。

5.2.8 值数组元素

语法：

〈值数组元素〉::= (1)

〈数组原值〉(〈表达式表〉) (1.1)

注意：如果数组原值是数组单元，则上述语法构造是二义的，此时将它解释为一个数组元素（参见4.2.8节）。

导出语法：参见4.2.8节。

语义：值数组元素传递一个值，它就是由表达式指出的指定的数组值的元素。

静态性质：值数组元素的类是M值类，其中M是数组原值的模式的**元素**模式。

静态条件：指定的表达式的类必须与数组原值的模式的**下标**模式是**相容的**。

动态条件：值数组元素所传递的值不能是**未定义的**。

如果下列关系式不成立，则引发异常 *RANGEFAIL*：

$L \leqslant$表达式$\leqslant U$

其中 L 和 U 分别是数组原值的模式的**下界**和（可能是动态的）**上界**。

5.2.9 值数组片

语法：

〈值数组片〉::= (1)

〈数组原值〉(〈下元素〉:〈上元素〉) (1.1)

| 〈数组原值〉(〈首元素〉**UP**〈片大小〉) (1.2)

注意：如果数组原值是一个数组单元，则上述语法构造是二义的，此时将它解释为一个数组片（参见4.2.9节）。

语义：值数组片传递一个（可能是动态的）数组值，它就是指定的数组值中由下元素和上元素或由首元素和片大小指明的那部分子值。值数组片的**下界**等于指定的数组值的**下界**，而值数组片的（可能是动态的）**上界**由指定的诸表达式确定。

静态性质：值数组片的（可能是动态的）类是M值类，其中M是一个如下构造出的**参数化**数组模式：

&名字（下标上界）

其中，&名字是一个与数组原值的（可能是动态的）模式**同义**的虚拟**同义模式**名字；下标上界要么是一个其类与下元素和上元素的类都是**相容的**一个表达式，且该表达式传递的值满足下列关系式：

NUM（下标上界）=*NUM*（*L*）+*NUM*（上元素）−*NUM*（下元素）

要么是其类与首元素的类是**相容的**一个表达式，且该表达式传递的值满足下列关系式：

NUM（下标上界）=*NUM*（*L*）+*NUM*（片大小）−1

其中 L 是数组原值的模式的**下界**。

如果下标上界是**字面值**，即下元素和上元素都是**字面值**或片大小是**字面值**，则值数组片的类是静态的，否则是动态的。

静态条件：下元素和上元素的类或首元素的类必须与数组原值的模式的**下标**模式是**相容的**。

下列关系式必须成立：

$L \leqslant$下元素$\leqslant$上元素$\leqslant U$

$1 \leqslant NUM$（片大小）$\leqslant NUM$（U）$-NUM$（L）$+1$

NUM（L）$\leqslant NUM$（首元素）$\leqslant NUM$（首元素）$+NUM$（片大小）$-1 \leqslant NUM$（U）

其中 L 和 U 分别是数组原值的模式的**下界**和**上界**。如果 U 和诸表达式的值都能静态得知，则可以静态检查这些关系式。

动态条件：值数组片所传送的值不能是**未定义的**。

如果上述关系式检查的动态部分失败，则引发异常 *RANGEFAIL*。

5.2.10 值结构域

语法：

〈值结构域〉::= (1)

〈结构原值〉.〈域名字〉 (1.1)

注意：如果结构原值是一个结构单元，则上述语法构造是二义的，此时将它解释为一个结构域（参见4.2.10节）。

语义：值结构域传送一个值，它就是指定的结构值中由域名字指出的域。如果结构原值具有**无标志变体**结构模式，且域名字是一个**变体域**名字，则这种值结构域的语义由实现定义。

静态性质：值结构域的类是 M 值类，其中 M 是域名字的模式。

静态条件：域名字必须是结构原值的模式的**域**名字集合中的一个名字。

动态条件：值结构域所传递的值不能是**未定义的**。

结构原值不能表示下列值：

- 它是一个具有**带标志变体**结构模式的值，而与之有关的**标志**域值指出该值结构域所表示的域不存在；
- 它是一个具有动态**参数化**结构模式的值，而与之有关的值表指出该值结构域所表示的域不存在。

上面提到的条件被称为关于值的变体域访问条件（注意：该条件不包含异常的出现）。如果结构原值不满足上述条件，则引发异常 *TAGFAIL*。

例子：

16.51 (**RECEIVE** *user_buffer*).*allocator* (1.1)

5.2.11 表达式转换

语法：

〈表达式转换〉::= (1)

〈模式名字〉(〈表达式〉) (1.1)

注意：如果表达式是一个静态模式单元，则上述语法构造是二义的，此时将它解释为一个单元转换（参见 4.2.13 节）。

语义：表达式转换忽略 CHILL 模式检验和相容性规则，它显式地把一个模式附着到指定的表达式之上。如果模式名字所表示的模式是一个离散模式，且表达式所传递的值的类也是离散的，则表达式转换所传递的值是满足下列关系式的值：

NUM（模式名字（表达式））＝*NUM*（表达式）

否则，表达式转换所传递的值由实现定义，并且与值的内部表示有关。

静态性质：表达式转换的类是 M 值类，其中 M 是模式名字。

当且仅当表达式是**常数**时，表达式转换是**常数**。

静态条件：模式名字不能具有**非值性质**。实现可以增加额外的静态条件。

动态条件：如果由表达式传递的值的类是离散的，且模式名字所表示的模式是一个离散模式，但它未定义具有与 *NUM*（表达式）相等的内部表示的值，则引发异常 *OVERFLOW*。实现可以增添额外的动态条件，违反这种动态条件时将引发实现定义的异常。

5.2.12 值过程调用

语法：

〈值过程调用〉::＝ *(1)*

〈值过程调用〉 *(1.1)*

语义：值过程调用传递由（被调用的）过程返送的值。

静态性质：值过程调用的类是 M 值类，其中 M 是值过程调用的**结果说明**的模式。

动态条件：值过程调用不能传递一个**未定义的**值（参见 5.3.1 节和 6.8 节）。

例子：

6.50 *julian_day_number([10,dec,1979])* *(1.1)*

11.63 *ok_bishop(b,m)* *(1.1)*

5.2.13 值内部子程序调用

语法：

〈值内部子程序调用〉::＝ *(1)*

〈值内部子程序调用〉 *(1.1)*

语义：值内部子程序调用传递（被调用的）内部子程序所返送的值。

静态性质：值内部子程序调用附着的类是值内部子程序调用的类。

动态条件：值内部子程序调用不能传递一个**未定义的**值（参见 5.3.1 和 6.8 节）。

5.2.14 *启动表达式*

语法：

〈*启动表达式*〉::= (1)

START〈进程名字〉([〈实参表〉]) (1.1)

语义：启动表达式的求值创建并激活其定义由进程名字指出的一个新进程（参见第十一章）。启动表达式传递一个唯一标识被创建的进程的实例值。参数传递与过程参数传递类似，但在启动表达式中可以给出额外的实参，它们具有实现定义的含义。

静态性质：*启动表达式*的类是 *INSTANCE* 导出类。

静态条件：实参表中出现的实参个数不能小于进程名字的进程定义的形参表中出现的形参个数。如果实参个数是 m，而上述形参个数是 n（$m \geq n$），则对前 n 个实参的相容性和**区域性**要求与对过程参数传递的相应要求相同（参见 6.7 节）。而针对其余额外实参的静态条件由实现定义。

动态条件：就参数传递而言，任何实参值相对应的形参的模式必须满足赋值条件（参见 6.7 节）。

如果存储需求得不到满足，则*启动表达式*引发异常 SPACEFAIL。

例子：

15.35 **START** *counter* () (1.1)

5.2.15 *零目运算符*

语法：

〈*零目运算符*〉::= (1)

THIS (1.1)

语义：零目运算符传递唯一标识执行它的进程的实例值。

静态性质：*零目运算符*的类是 *INSTANCE* 导出类。

5.2.16 *带括号表达式*

语法：

〈*带括号表达式*〉::= (1)

(〈表达式〉) (1.1)

语义：带括号表达式传递对指定的表达式求值后得到的值。

静态性质：*带括号表达式*的类是表达式的类。

当且仅当表达式是**常数（字面值）**时，*带括号表达式*是**常数（字面值）**。

例子：

5.10 (*a1* **OR** *b1*) (1.1)

5.3 值和表达式

5.3.1 概述

语法：

〈值〉::= (1)

〈表达式〉 (1.1)

| 〈未定义值〉 (1.2)

〈未定义值〉::= (2)

* (2.1)

| 〈未定义同义词名字〉 (2.2)

语义：值要么是一个**未定义的**值，要么是作为对一个表达式求值的结果而传递的（CHILL 定义的）值。

除了特别指明的情形外，并未定义一个表达式的成分及其子成分的求值次序，可以认为它们是按混杂次序被求值的。它们只需被求值到能唯一确定该表达式所传递的值处为止。如果上下文要求出现一个**常数**或**字面值**表达式，则其求值假定是在运行时刻之前完成的，且不会引发异常。实现将定义**字面值**和**常数**表达式的合法值的区间，并且，如果这种运行时刻之前的求值传递了一个超出实现定义的界限的值，则实现可以拒绝这种程序。

静态性质：值的类分别是表达式或未定义值的类。

如果未定义值是 *，则其类是 **all** 类，否则是未定义同义词名字的类。

当且仅当值是未定义值或一个为**常数**的表达式时，该值是**常数**。当且仅当值是一个为**字面值**的表达式时，该值是**字面值**。

动态性质：如果一个值是由未定义值表示的值或是在本建议书中被显式指明为**未定义的**，则称该值是**未定义的**。当且仅当一个组合值的所有子分量（即子串值、元素值或域值）是**未定义的**时，该组合值是**未定义的**。

例子：

6.40 *(146_097*c)/4+(1_461*y)/4*
+(153+m+c)/5+day+1_721_119 *(1.1)*

5.3.2 表达式

语法：

〈表达式〉::= (1)

〈运算数-0〉 (1.1)

| 〈条件表达式〉 (1.2)

〈条件表达式〉::= (2)

| **IF** 〈布尔表达式〉〈则选择项〉
〈否则选择项〉 **FI** (2.1)

| **CASE** 〈情况选择器表〉 **OF** { 〈值情况选择项〉}+

[**ELSE**〈子表达式〉] **ESAC** (2.2)

〈则选择项〉::= (3)

THEN〈子表达式〉 (3.1)

〈否则选择项〉::= (4)

ELSE〈子表达式〉 (4.1)

| **ELSIF**〈布尔表达式〉〈则选择项〉
〈否则选择项〉 (4.2)

〈子表达式〉::= (5)

〈表达式〉 (5.1)

〈值情况选择项〉::= (6)

〈情况标号说明〉:〈子表达式〉; (6.1)

语义：如果指定了 **IF**，则先对布尔表达式求值。如果它产生值 *TRUE*，则条件表达式所传递的值是由则选择项中子表达式所传递的值，否则是否则选择项所传递的值。

如果其内指明了 **ELSE**，则否则选择项所传递的值是 **ELSE** 后面的子表达式所传递的值；如果其内指明了 **ELSIF**，则先对 **ELSIF** 后面的布尔表达式求值，且如果产生值 *TRUE*，则否则选择项所传递的值是其内则选择项中子表达式所传递的值，否则是其内否则选择项所传递的值。

如果指明了 **CASE**，则先对情况选择器表内的诸子表达式求值。如果存在一个与之匹配的情况标号说明，则条件表达式所传递的值是具有该情况标号说明的值情况选择项内指定的子表达式所传递的值，否则是跟在 **ELSE** 后面的子表达式所传递的值（此时应出现 **ELSE** 成分）。

条件表达式中未用到的子表达式不被求值。

静态性质：如果表达式是运算数-0，则表达式的类是运算数-0的类。如果表达式是条件表达式，则表达式的类是 M 值类，其中模式 M 取决于该条件表达式出现处的程序正文，其确定规则与定义不带模式名字的多元组的类的模式的规则相同（参见5.2.5节）。

当且仅当一个表达式要么是为**常数（字面值）**的运算数-0，要么是条件表达式，且其内所有布尔表达式或情况选择器表以及所有子表达式都是**常数（字面值）**时，该表达式是**常数（字面值）**。

静态条件：如果表达式是条件表达式，则应满足下列条件：

- 条件表达式仅能出现在其前面不带模式名字的多元组所能出现的程序上下文中；
- 其内每个子表达式的类必须与从上述上下文导出的那个模式是**相容的**，推导规则与确定多元组的模式的规则相同。但是，仅对选中的子表达式实施相容性关系的动态检查；
- 如果指定了 **CASE**，则必须满足情况选择条件（参见12.3节），此外，如同情况动作一样，必须满足同样的完备性、一致性和相容性要求（参见6.4节）；
- 条件表达式内不能具有这样的两个子表达式，其一是**区域外的**，而另一个是**区域内的**（参见11.2.2节）。

动态条件：在条件表达式情况下，由选中的子表达式传递的值相对从出现该条件表达式的上下文中推导出来的模式 M 而言应满足赋值条件。

5.3.3 运算数-0

语法：

〈运算数-0〉::= (1)

〈运算数-1〉 (1.1)

| 〈子运算数-0〉 {**OR**|**ORIF**|**XOR**} 〈运算数-1〉 (1.2)

〈子运算数-0〉 ::= (2)

〈运算数-0〉 (2.1)

语义：如果指定了 **OR**、**ORIF** 或 **XOR**，且子运算数-0和运算数-1传递：

- 布尔值，则运算数-0传递一个布尔值。此时 **OR** 和 **XOR** 分别表示逻辑运算符“或”和“异或”。如果指定了 **ORIF**，且子运算数-0传递布尔值 *TRUE*，则运算数-0就传递值 *TRUE*，否则运算数-0所传递的值就是运算数-1所传递的值；
- 位串值，则运算数-0传递一个位串值。此时 **OR** 和 **XOR** 分别表示在那两个位串的元素上实施的逻辑运算符“按位或”和“按位异或”；
- 幂集值，则运算数-0传递一个幂集值，**OR** 表示那两个幂集值的“并”，而 **XOR** 表示由那些仅在指定的幂集值之一中出现的成员值组成的幂集值（即，*A* **XOR** *B*=*A*−*B* **OR** *B*−*A*）。

静态性质：如果运算数-0是运算数-1，则运算数-0的类就是运算数-1的类。如果指定了 **OR**、**ORIF** 或 **XOR**，则运算数-0的类是子运算数-0和运算数-1的类的**结果类**。

当且仅当运算数-0要么是为**常数（字面值）**的运算数-1，要么含有都是**常数（字面值）**的子运算数-0和运算数-1时，它是**常数（字面值）**。

静态条件：如果指明了 **OR**、**ORIF** 或 **XOR**，则子运算数-0的类必须与运算数-1的类是**相容的**。如果指定了 **ORIF**，则这两个类都必须具有布尔**根**模式，否则这两个类必须都具有布尔、幂集或位串**根**模式。在都具有位串**根**模式的情况下，子运算数-0和运算数-1的**实际长度**必须相等。如果子运算数-0和运算数-1的模式都是或它们之一是一个动态或**变长**串模式，则上述检查是动态的。

动态条件：在指定 **OR** 或 **XOR** 的情况下，如果两个运算数或其中之一具有动态类，且上面提到的相容性检查的动态部分失败，则引发异常 *RANGEFAIL*。

例子：

10.31 *i*<*min* (1.1)

10.31 *i*<*min* **OR** *i*>*max* (1.2)

5.3.4 运算数-1

语法：

〈运算数-1〉 ::= (1)

〈运算数-2〉 (1.1)

| 〈子运算数-1〉 {**AND**|**ANDIF**} 〈运算数-2〉 (1.2)

〈子运算数-1〉 ::= (2)

〈运算数-1〉 (2.1)

语义：如果指明了 **AND** 或 **ANDIF**，且子运算数-1和运算数-2传递：

- 布尔值，则运算数-1传递一个布尔值，此时 **AND** 表示逻辑“与”运算。如果指定 **ANDIF**，且子运算数-1传递布尔值 *FALSE*，则运算数-1所传递的值是 *FALSE*，否则是运算数-2所传递的值；
- 位串值，则运算数-1传递一个位串值，此时 **AND** 表示在那两个位串的每个元素上实施的逻辑“按位

与”运算；

- 幂集值，则运算数-1传递一个幂集值。此时 **AND** 表示幂集值的“交”运算。

静态性质：如果运算数-1是运算数-2，则运算数-1的类是运算数-2的类。

如果指明了 **AND** 或 **ANDIF**，则运算数-1的类是运算数-2和子运算数-1的类的**结果类**。

当且仅当运算数-1要么是为**常数**（**字面值**）的运算数-2，要么含有都是**常数**（**字面值**）的子运算数-1和运算数-2时，它是**常数**（**字面值**）。

静态条件：如果指定了 **AND** 或 **ANDIF**，则子运算数-1的类必须与运算数-2的类是**相容的**。如果指定 **ANDIF**，它们的类必须都具有布尔**根**模式，否则它们的类必须都具有布尔、幂集或**位**串**根**模式。在都具有**位**串**根**模式的情况下，子运算数-1和运算数-2的**实际长度**必须相等。如果这两个模式或两者之一是动态或**变长**串模式，则上述检查是动态的。

动态条件：在指定 **AND** 的情况下，如果两个运算数或其中之一具有动态类，并且上述相容性检查的动态部分失败，则引发异常 *RANGEFAIL*。

例子：

5.10 *(a1* **OR** *b1)* *(1.1)*
5.10 **NOT** *k2* **AND** *(a1* **OR** *b1)* *(1.2)*

5.3.5 运算数-2

语法：

〈运算数-2〉 ::= *(1)*
〈运算数-3〉 *(1.1)*
| 〈子运算数-2〉〈运算符-3〉〈运算数-3〉 *(1.2)*

〈子运算数-2〉 ::= *(2)*
〈运算数-2〉 *(2.1)*

〈运算符-3〉 ::= *(3)*
〈关系运算符〉 *(3.1)*
| 〈成员运算符〉 *(3.2)*
| 〈幂集缠含运算符〉 *(3.3)*

〈关系运算符〉 ::= *(4)*
=|/=|>|<|>=|<= *(4.1)*

〈成员运算符〉 ::= *(5)*
IN *(5.1)*

〈幂集缠含运算符〉 ::= *(6)*
<=|>=|<|> *(6.1)*

语义：等于（=）和不等（/=）运算符被定义在某个给定的模式的任意两个值之间。其它关系运算符（小于（<）、小于等于（<=）、大于（>）及大于等于（>=））被定义在某个给定的离散模式、计时模式或串模式的任意两个值之间。所有关系运算都传递一个布尔值作为结果。

成员运算符被定义在一个成员值与一个幂集值之间。如果该成员值属于该幂集值，则此运算传递布尔值 *TRUE*，否则传递布尔值 *FALSE*。

幂集蕴含运算符被定义在两个幂集值之间，它们用来测试一个幂集值是否被包含在（<=）或被真包含在（<）另一个幂集值中，或是否包含（>=）或真包含（>）另一个幂集值。幂集蕴含运算传递一个布尔值作为运算结果。

静态性质：如果运算数-2是运算数-3，则运算数-2的类就是运算数-3的类。如果指定了运算符-3，则运算数-2的类是 *BOOL* 导出类。

当且仅当运算数-2要么是为**常数（字面值）**的运算数-3，要么包含均为**常数（字面值）**的子运算数-2和运算数-3时，它是**常数（字面值）**。

静态条件：如果指定了运算符-3，则子运算数-2的类与运算数-3的类必须满足下列相容性要求：

- 如果运算符-3是=或/=，则两者必须是**相容的**；
- 如果运算符-3是非=和/=的其它关系运算符，则两者必须是**相容的**，且都具有离散、计时或串**根**模式；
- 如果运算符-3是成员运算符，则运算数-3的类必须具有幂集**根**模式，且子运算符-2的类必须与那个**根**模式的**成员**模式是**相容的**；
- 如果运算符-3是幂集蕴含运算符，则两者必须是**相容的**，且都具有幂集**根**模式。

动态条件：在指定关系运算符的情况下，如果两个运算数或其中之一的类是动态类，并且上述相容性检查的动态部分失败，则引发异常 *RANGEFAIL* 或 *TAGFAIL*。当且仅当其中某个动态类是基于某个动态**参数化**结构模式时，引发异常 *TAGFAIL*。

例子：

10.50 NULL (1.1)
10.50 last=NULL (1.2)

5.3.6 运算数-3

语法：

〈运算数-3〉::= (1)
〈运算数-4〉 (1.1)
| 〈子运算数-3〉〈运算符-4〉〈运算数-4〉 (1.2)

〈子运算数-3〉::= (2)
〈运算数-3〉 (2.1)

〈运算符-4〉::= (3)
〈算术加减运算符〉 (3.1)
| 〈串并置运算符〉 (3.2)
| 〈幂集求差运算符〉 (3.3)

〈算术加减运算符〉::= (4)
+|− (4.1)

〈串并置运算符〉::= (5)
// (5.1)

〈幂集求差运算符〉::= (6)
− (6.1)

语义：如果运算符-4是算术加减运算符，则两个运算数都传递整数值，且结果整数值是这两个值的和（+）或差（-）。

如果运算符-4是串并置运算符，则两个运算数要么都传递位串值，要么都传递字符串值，且运算结果值是由这两个值的并置而成的串值。还允许两个运算数传递布尔（字符）值，它们被视为是长度为1的位（字符）串值。

如果运算符-4是幂集求差运算符，则两个运算数都传递幂集值，且结果值是一个幂集值，其内的成员值属于子运算数-3所传递的幂集值，但不属于运算数-4所传递的幂集值。

静态性质：如果运算数-3是运算数-4，则运算数-3的类是运算数-4的类。如果指定了运算符-4，则运算数-3的类由运算符-4确定如下：

- 如果运算符-4是串并置运算符，则运算数-3的类依赖于运算数-4和子运算数-3的类。如果两个运算数之一是布尔值或字符值，则它们分别被视为其类为 **BOOLS**（*1*）导出类或 **CHARS**（*1*）导出类的串值：
  - — 如果两个运算数都不是**强**值，则根据这两个运算数是位串还是字符串，运算数-3的类是 **BOOLS**（*n*）导出类或 **CHARS**（*n*）导出类。其中 *n* 是运算数-4和子运算数-3的类的**根**模式的**串长度**之和；
  - — 否则，运算数-3 的类是 & *名字*（*n*）值类。其中 & *名字*是一个与两个运算数的类的**结果类**的**根**模式**同义**的虚拟**同义模式**名字，而 *n* 表示两个运算数的**根**模式的**串长度**之和。

  （如果两个运算数或其中之一具有动态类，则运算数-3的类是动态的）。
- 如果运算符-4是算术加减运算符或幂集求差运算符，则运算数-3的类是运算数-4和子运算数-3的类的**结果类**。

当且仅当一个运算数-3要么是为**常数**（**字面值**）的运算数-4，要么含有都是**常数**（**字面值**）的运算数-4和子运算数-3，且运算符-4是算术加减运算符或幂集求差运算符时，该运算数-3是**常数**（**字面值**）。

如果一个运算数-3内的运算符-4是串并置运算符，且其内运算数-4和子运算数-3都是**常数**，则该运算数-3是**常数**。

静态条件：如果指定了运算符-4，则必须满足下列相容性要求：

- 如果运算符-4是算术加减运算符，则两个运算数的类必须是**相容的**，且都具有整数**根**模式；
- 如果运算符-4是串并置运算符，则
  - — 两个运算数的类必须是**相容的**，且都具有**位串根**模式或**字符串根**模式；或
  - — 两个运算数的类都必须与 *BOOL* 或 *CHAR* 模式是**相容的**；或
  - — 其中一个运算数的类必须具有**位**（**字符**）串**根**模式，而另一个运算数的类必须与 *BOOL*（*CHAR*）模式是**相容的**。
- 如果运算符-4是幂集求差运算符，则两个运算数的类必须是**相容的**，且必须都具有幂集**根**模式。

动态条件：在运算数-3不是**常数**的情况下，如果由加运算（+）或减运算（-）产生一个不是由运算数-3的类的**根**模式所定义的值，则引发异常 *OVERFLOW*。

例子：

1.6 j (*1.1*)

1.6 i+j (*1.2*)

5.3.7 运算数-4

语法：

〈运算数-4〉 ::= (1)

〈运算数-5〉 (1.1)

| 〈子运算数-4〉〈算术乘除运算符〉〈运算数-5〉 (1.2)

〈子运算数-4〉 ::= (2)

〈运算数-4〉 (2.1)

〈算术乘除运算符〉 ::= (3)

* | / | **MOD** | **REM** (3.1)

语义：如果指定了算术乘除运算符，则子运算数-4和运算数-5都传递整数值，且计算结果传递的整数值是那两个值的积（*）、商（/）、模（**MOD**）或余数（**REM**）。

取模运算定义如下：i **MOD** j 传递唯一的整数值 k（$0 \leqslant k < j$），它使得存在一个整数值 n 满足 $i = n * j + k$；j 必须大于0。

商运算定义如下：对任意整数值 x 和 y 而言，下列关系运算都产生值 *TRUE*：

$$ABS\ (x/y) = ABS\ (x)\ /ABS\ (y)$$
$$sign\ (x/y) = sign\ (x)\ /sign\ (y)$$
$$ABS\ (x) - (ABS\ (x)\ /ABS\ (y))\ * ABS\ (y) = ABS\ (x)\ \textbf{MOD}\ ABS\ (y)$$

其中函数 *sign* 定义如下：如果 $x<0$，则 $sign\ (x) = -1$，否则 $sign\ (x) = 1$。

取余运算定义如下：对任意整数值 x 和 y 而言，关系运算 x ***REM*** $y = x - (x/y)\ * y$ 都产生值 *TRUE*。

静态性质：如果运算数-4是运算数-5，则运算数-4的类就是运算数-5的类，否则是子运算数-4和运算数-5的类的**结果类**。

当且仅当运算数-4要么是为**常数（字面值）**的运算数-5，要么包含都是**常数（字面值）**的子运算数-4和运算数-5时，它是**常数（字面值）**。

静态条件：如果指定了算术乘除运算符，则运算数-5与子运算数-4的类必须是**相容的**，且它们都具有整数**根**模式。

动态条件：在运算数-4不是**常数**的情况下，如果乘（*）、除（/）、取模（**MOD**）或取余（**REM**）运算产生一个不是运算数-4的类的**根**模式所定义的值，或者该运算在某个运算数的值上无定义，则引发异常 *OVERFLOW*。后一种情况是指在除或取余运算中运算数-5传递一个0值，或在取模运算中运算数-5传递一个非正整数值。

例子：

6.15 1_461 (1.1)

*6.15 (4 * d + 3) / 1_461* (1.2)

5.3.8 运算数-5

语法：

〈运算数-5〉 ::= (1)

[〈单目运算符〉]〈运算数-6〉 (1.1)

〈单目运算符〉::= (2)

— |**NOT** (2.1)

| 〈串重复运算符〉 (2.2)

〈串重复运算符〉::= (3)

(〈整数字面值表达式〉) (3.1)

语义：如果单目运算符是变号运算符（—），则运算数-6传递一个整数值，且运算结果是将上面那个整数值符号取反后得到的整数值。

如果单目运算符是**NOT**，则运算数-6传递一个布尔值、位串值或幂集值。在前两种情况下，运算结果是上述布尔值或位串值的所有元素的逻辑非值。在最后一种情况下，运算结果是上述幂集值的补值，即由不属于运算数幂集值的那些成员值组成的幂集值。

如果单目运算符是串重复运算符，则运算数-6是字符串字面值-或位串字面值。如果整数字面值表达式传递一个0值，则运算 结果是空串值，否则运算结果是下列串值，即由运算数-6传递的串值与该串值自己连续并置（n—1）次后得到的串值，其中n是由整数字面值表达式传递的值。

静态性质：如果运算数-5是运算数-6，则运算数-5的类就是运算数-6的类。

如果指定了单目运算符，则运算数-5的类分别是：

- 如果单目运算符是—或**NOT**，则它是运算数-6的**结果类**；
- 如果单目运算符是串重复运算符，则它是**CHARS**（n）导出类或**BOOLS**（n）导出类（这取决于运算数-6是字符串字面值还是位串字面值），其中$n=r*l$，而r是由整数字面值表达式传递的值，l是运算数-6所传递的串字面值的**串长度**。

当且仅当运算数-6是**常数**时，运算数-5是**常数**。当且仅当运算数-6是**字面值**，且单目运算符是—或**NOT**时，运算数-5是**字面值**。

静态条件：如果单目运算符是—，则运算数-6的类必须具有整数**根**模式。

如果单目运算符是**NOT**，则运算数-6的类必须具有布尔、**位**串或幂集**根**模式。

如果单目运算符是串重复运算符，则运算数-6必须是字符串字面值或位串字面值。整数字面值表达式必须传递一个非负的整数值。

动态条件：如果运算数-5不是**常数**，那么，当变号运算（—）产生一个不是由运算数-5的类的**根**模式定义的值时，引发异常*OVERFLOW*。

例子：

5.10 **NOT** *k2* (1.1)

7.54 *(6)''* (1.1)

7.54 *(6)* (2.2)

5.3.9 运算数-6

语法：

〈运算数-6〉::= (1)

〈被引用单元〉 (1.1)

| 〈接收表达式〉 (1.2)

| 〈原值〉 (1.3)

〈被引用单元〉 ::= (2)

-〉〈单元〉 (2.1)

〈接收表达式〉 ::= (3)

RECEIVE〈缓冲区单元〉 (3.1)

语义：被引用单元传递一个对指定的单元的引用值。

接收表达式从缓冲区单元处接收一个值。执行进程可能成为被延迟的，也可能重新激活另一个因向指定的缓冲区单元发送信息而被延迟的进程（细节详见6.19.3节）。

静态性质：运算数-6的类分别是被引用单元、接收表达式或原值的类。被引用单元的类是M引用类，其中M是单元的模式。接收表达式的类是M值类，其中M是缓冲区单元的模式的**缓冲区元素**模式。

当且仅当原值是**常数**或被引用单元是**常数**时，运算数-6是**常数**。当且仅当单元是**静态的**时，被引用单元是**常数**。当且仅当原值是**字面值**时，运算数-6是**字面值**。

静态条件：单元必须是**可引用的**。

动态条件：在执行接收表达式的进程被延迟在缓冲区单元之上期间，缓冲区单元的生存期不能结束。

例子：

8.25 -> *c* (2.1)

16.51 **RECEIVE** *user_buffer* (3.1)

6 动作

6.1 概述

语法：

〈动作语句〉 ::= (1)
　[〈定义性出现〉:]〈动作〉[〈处理程序〉][〈简单名字串〉]; (1.1)
　| 〈模块〉 (1.2)
　| 〈说明模块〉 (1.3)
　| 〈上下文模块〉 (1.4)

〈动作〉 ::= (2)
　〈带括号动作〉 (2.1)
　| 〈赋值动作〉 (2.2)
　| 〈调用动作〉 (2.3)
　| 〈出口动作〉 (2.4)
　| 〈返回动作〉 (2.5)
　| 〈结果动作〉 (2.6)
　| 〈转向动作〉 (2.7)
　| 〈断言动作〉 (2.8)
　| 〈空动作〉 (2.9)
　| 〈启动动作〉 (2.10)
　| 〈停止动作〉 (2.11)
　| 〈延迟动作〉 (2.12)
　| 〈继续动作〉 (2.13)
　| 〈发送动作〉 (2.14)
　| 〈引发动作〉 (2.15)

〈带括号动作〉 ::= (3)
　〈条件动作〉 (3.1)
　| 〈情况动作〉 (3.2)
　| 〈循环动作〉 (3.3)
　| 〈begin-end 分程序〉 (3.4)
　| 〈延迟情况动作〉 (3.5)
　| 〈接收情况动作〉 (3.6)
　| 〈计时动作〉 (3.7)

语义：动作语句构成 CHILL 程序的算法部分。任何动作语句都可以带标号。那些绝不可能引发异常的动作不能附加处理程序。

静态性质：动作语句中的定义性出现定义了一个**标号**名字。

静态条件：只能在带括号动作的动作或指明了处理程序的动作后面给出简单名字串，且仅当指明了定义性出现时，才能给出简单名字串。简单名字串必须与定义性出现具有相同的名字串。

6.2 赋值动作

语法：

〈赋值动作〉::= (1)

〈单赋值动作〉 (1.1)

| 〈多重赋值动作〉 (1.2)

〈单赋值动作〉::= (2)

〈单元〉〈赋值符号〉〈值〉 (2.1)

| 〈单元〉〈赋值运算符〉〈表达式〉 (2.2)

〈多重赋值动作〉::= (3)

〈单元〉{,〈单元〉}$^{+}$〈赋值符号〉〈值〉 (3.1)

〈赋值运算符〉::= (4)

〈封闭的二目运算符〉〈赋值符号〉 (4.1)

〈封闭的二目运算符〉::= (5)

OR|**XOR**|**AND** (5.1)

| 〈幂集求差运算符〉 (5.2)

| 〈算术加减运算符〉 (5.3)

| 〈算术乘除运算符〉 (5.4)

| 〈串并置运算符〉 (5.5)

〈赋值符号〉::= (6)

:= (6.1)

语义：赋值动作把一个值存入一个或多个单元。

如果使用了赋值符号，则赋值符号右端产生的值被存入赋值符号左端指定的单元。

如果使用了赋值运算符，则根据给定的封闭的二目运算符的语义将赋值符号左端单元中存储的值与赋值符号右端的值结合（就以这个次序），且运算结果被存回左端的单元。

赋值符号左端单元的求值、右端值的计算以及赋值运算符本身所含的计算可能以一种未指定的混合的次序完成。一旦求出值和单元，即可完成赋值。

如果指定的单元（或指定的诸单元之一）是某个变体结构的**标志**域，则对依赖于该**标志**域的变体域赋值的语义由实现定义。

静态条件：所有出现的单元的模式必须是**等价的**，且它们既没有**只读性质**，也没有**非值性质**。每个模式都必须与值的类是**相容的**。在涉及动态模式单元与/或具有动态类的值的情况下，上述相容性检查是动态的。

相对指定的每个单元而言，值必须是**区域安全的**（参见11.2.2节）。

如果任一个单元具有**定长**串模式，则该模式的**串长度**和值的**实际长度**必须相等；否则，如果该单元具有**变长**串模式，则该模式的**串长度**不能小于该值的**实际长度**。如果单元和值的模式都是或两者之一是动态的或是**变长**串模式，则上述条件的检查是动态的。上述条件被称为串赋值条件。

动态条件：如果赋值符号左端单元的模式和/或赋值符号右端值的模式是动态模式，且上面提到的相容性检查的动态部分失败，则引发 *RANGEFAIL* 或 *TAGFAIL* 异常。

如果赋值符号左端单元的模式和/或赋值符号右端值的模式是**变长**串模式，且上面提到的相容性检查的动态部分失败，则引发异常 *RANGEFAIL*。

如果赋值符号左端任一个单元具有一个范围模式，且赋值符号右端值的求值所传递的值既不是由

该范围模式定义的值之一，也不是**未定义的**值，则引发异常 *RANGEFAIL*。

上面叙述的动态条件以及串赋值条件称为值关于模式的赋值条件。

在指定赋值运算符的情况下，如果对下列表达式求值

〈单元〉〈封闭的二目运算符〉(〈表达式〉)

，并把它所传递的值存入指定的单元时引发了异常，则相应赋值动作也引发同样的异常（注意：该单元仅被求值一次）。

例子：

```
4.12    a := b+c                                        (1.1)
10.25   stackindex- := 1                                (2.1)
19.19   x->.prex, x->.next := NULL                      (3.1)
10.25   - :=                                            (4.1)
```

6.3 条件动作

语法：

〈条件动作〉::= (1)

IF 〈布尔表达式〉〈则子句〉[〈否则子句〉] **FI** (1.1)

〈则子句〉::= (2)

THEN 〈动作语句表〉 (2.1)

〈否则子句〉::= (3)

ELSE 〈动作语句表〉 (3.1)

|**ELSIF** 〈布尔表达式〉〈则子句〉[〈否则子句〉] (3.2)

导出语法：表示法：

ELSIF 〈布尔表达式〉〈则子句〉[〈否则子句〉]

是关于：

ELSE IF 〈布尔表达式〉〈则子句〉[〈否则子句〉] **FI**;

的导出语法。

语义：条件动作是两路分支的条件转移。若布尔表达式产生值 *TRUE*，则进入跟在 **THEN** 后面的动作语句表；否则进入跟在 **ELSE** 后面的动作语句表（如果存在的话）。

动态条件：如果存储需求得不到满足，则引发异常 *SPACEFAIL*。

例子：

```
7.22    IF n >= 50 THEN rn(r) := 'L';
            n- := 50;
            r+ := 1;
        FI                                              (1.1)
10.50   IF last = NULL
            THEN first,last := p;
            ELSE last->.succ := p;
            p->.pred := last;
            last := p;
        FI                                              (1.1)
```

6.4 情况动作

语法：

〈情况动作〉::= (1)

CASE 〈情况选择器表〉 **OF** [〈范围表〉;] {〈情况选择项〉}+

[**ELSE** 〈动作语句表〉] **ESAC** (1.1)

〈情况选择器表〉::= (2)

〈离散表达式〉{,〈离散表达式〉}* (2.1)

〈范围表〉::= (3)

〈离散模式名字〉{,〈离散模式名字〉}* (3.1)

〈情况选择项〉::= (4)

〈情况标号说明〉:〈动作语句表〉 (4.1)

语义：情况动作是多路分支的条件转移。它由一个或多个离散表达式的说明（情况选择器表）和若干带标志的动作语句表（情况选择项）组成。每个动作语句表前面都有一个情况标号说明，情况标号说明由一串情况标号表说明组成（每个情况标号表说明对应一个情况选择器）。每个情况标号表都定义了一个值集。在情况选择器表中，离散表达式表的使用使得选择能以多路条件分支为基础进行。当某个情况选择项的情况标号说明中给定的（诸）值与情况选择器表中（诸）值相匹配时，该情况动作就进入该情况选择项内的动作语句表。如果不存在匹配的值，且指明了 **ELSE**，则进入跟在 **ELSE** 后面的动作语句表。

未定义情况选择器表中的诸表达式的求值次序，它们可能是以混杂的次序被求值的。上述求值只进行到能唯一确定选择哪个情况选择项时为止。

静态条件：对出现的情况标号说明的表而言，应满足情况选择条件（参见12.3节）。

情况选择器表中出现的离散表达式的个数必须等于由情况标号表出现组成的表的**结果类表**中类的个数，并且，如果存在范围表的话，还必须等于范围表中离散模式名字出现的个数。

情况选择器表中任何离散表达式的类必须与相应诸情况标号表出现的**结果类表**中（按位置）对应的类是**相容的**，而且，如果存在范围表的话，还必须与范围表中（按位置）对应的离散模式名字是**相容的**。范围表中出现的模式也必须与上述**结果类表**中（按位置）对应的类是**相容的**。

在一个情况标号内由离散字面值表达式传递的值，或由字面值范围或可散模式名字定义的任何值（参见12.3节）都必须位于范围表内对应的离散模式名字所定义的值界限之内（如果存在范围表的话），同时，也必须位于由情况选择器表中对应的离散表达式的模式定义的值界限之内，如果它是一个**强**离散表达式的话。在后一种情况下，由范围表（如果存在的话）内某个离散模式名字定义的值也必须位于情况选择器表中对应的离散表达式的模式所定义的值界限之内。

如果情况标号表出现的表是**完备的**（参见12.3节），则此时才能省略从语法上说来是任选的 **ELSE** 部分。

动态条件：如果指明了范围表，且情况选择器表中某个离散表达式所传递的值不处于由范围表中对应的离散模式名字定义的值界限之内，则引发异常 *RANGEFAIL*。

如果存储需求得不到满足，则引发异常 *SPACEFAIL*。

例子：

```
4.11    CASE order OF
            (1): a := b+c;
                 RETURN;
            (2): d := 0;
            (ELSE): d := 1;
        ESAC                                                    (1.1)
11.43   starting.p.kind, starting.p.color                       (2.1)
11.58   (rook),(*):
            IF NOT ok_rook(b,m)
                THEN
                  CAUSE illegal;
            FI;                                                 (4.1)
```

6.5 循环动作

6.5.1 概述

语法：

〈循环动作〉::= (1)

DO [〈控制部分〉;]〈动作语句表〉OD (1.1)

〈控制部分〉::= (2)

〈步长型控制〉[〈当型控制〉] (2.1)

| 〈当型控制〉 (2.2)

| 〈开域部分〉 (2.3)

语法：循环动作具有三种不同的格式之一：do-for 和 do-while 格式用于循环；do-with 格式是以有效的方式访问结构域的方便的缩写手段。如果未指明控制部分，每当进入该循环动作时，仅进入指定的动作语句表一次。

当合用 do-for 和 do-while 格式时，在实施步长型控制后，且仅当该循环动作未被步长型控制终止时，才计算当型控制。

如果指定的控制部分是步长型控制和/或当型控制，则只要程序控制仍处于该循环动作的可达区内部，根据指定的控制部分的语义，可多次进入指定的动作语句表，但每次执行指定的动作语句表时不再重新进入该循环可达区。

动态条件：如果存储需求得不到满足，则引发异常 *SPACEFAIL*。

例子：

```
4.17    DO FOR i := 1 TO c;
            op(a,b,d,order-1);
            d := a;
        OD                                                      (1.1)
```

```
15.58   DO WITH each;
            IF this_counter = counter
                THEN
                  status := idle;
                  EXIT find_counter;
            FI;
        OD                                                    (1.1)
```

6.5.2 步长型控制

语法：

〈步长型控制〉::= (1)

FOR {〈迭代〉{,〈迭代〉}* |**EVER**} (1.1)

〈迭代〉::= (2)

〈值枚举〉 (2.1)

|〈单元枚举〉 (2.2)

〈值枚举〉::= (3)

〈步长枚举〉 (3.1)

|〈范围枚举〉 (3.2)

|〈幂集枚举〉 (3.3)

〈步长枚举〉::= (4)

〈循环计数器〉〈赋值符号〉〈开始值〉

[〈步长值〉][**DOWN**]〈结束值〉 (4.1)

〈循环计数器〉::= (5)

〈定义性出现〉 (5.1)

〈开始值〉::= (6)

〈离散表达式〉 (6.1)

〈步长值〉::= (7)

BY〈离散表达式〉 (7.1)

〈结束值〉::= (8)

TO〈离散表达式〉 (8.1)

〈范围枚举〉::= (9)

〈循环计数器〉[**DOWN**] **IN**〈离散模式名字〉 (9.1)

〈幂集枚举〉::= (10)

〈循环计数器〉[**DOWN**] **IN**〈幂集表达式〉 (10.1)

〈单元枚举〉::= (11)

〈循环计数器〉[**DOWN**] **IN**〈部分组合对象〉 (11.1)

〈部分组合对象〉::= (12)

〈数组单元〉 (12.1)

|〈数组表达式〉 (12.2)

|〈串单元〉 (12.3)

|〈串表达式〉 (12.4)

注意：如果部分组合对象是（串或数组）单元，则上述语法是二义的，此时将主解释为一个单元而不是表达式。

语义：步长型控制可能提到几个循环计数器。这些循环计数器在每次进入指定的动作语句表之前按未指定的次序计算，且只要进行到能确定终止该循环动作的那一点为止。如果至少有一个循环计数器指明终止该循环动作，则该循环动作终止。

1. **DO FOR EVER**：

它表示无限制地重复执行指定的动作语句表。该循环动作只能靠将控制转出该动作的转移而终止。

2. **值枚举**：

对循环计数器所指定的值集而言它表示反复进入指定的动作语句表。该值集要么由一个离散模式名字确定（对范围枚举而言），要么由一个幂集值确定（对幂集枚举而言），或由开始值、步长值和结束值确定（对步长枚举而言）。

循环计数器隐式定义了一个在指定的动作语句表内部表示其值或单元的名字。

范围枚举：

在不带（带）**DOWN** 说明的范围枚举的情况下，循环计数器的初值是由离散模式名字定义的值集中的最小（最大）的值。在下一次执行指定的动作语句表中，循环计数器的下一个值是：

SUCC（前一个值）（*PRED*（前一个值））

所传递的值。如果对离散模式名字所定义的最大（最小）值已执行完指定的动作语句表，则该循环动作终止。

幂集枚举：

在不带（带）**DOWN** 说明的幂集枚举中，循环计数器的初值是指定的幂集中最小（最大）的成员值。若该幂集值为空幂集值，则不执行指定的动作语句表。在其后执行指定的动作语句表中，循环计数器的下一个值将是该幂集值中下一个较大的（较小的）成员值。如果对指定的幂集值中最大（最小）的成员值已执行完指定的动作语句表，则该循环动作终止。当执行该循环动作时，仅计算一次幂集表达式。

步长枚举：

在不带（带）**DOWN** 说明的步长枚举情况下，循环计数器的值集根据指定的开始值、结束值以及任选的步长值确定。当执行该循环动作时，这些离散表达式仅求值一次，且求值次序是未定义的，可能是以任意次序进行的。步长值总是正的。在每次执行指定的动作语句表之前测试是否终止该循环动作。最初检查循环计数器的开始值是否大于（小于）结束值。在后来执行指定的动作语句表期间，在指定步长值的情况下，循环计数器的下一个值是：

前一个值＋步长值（前一个值－步长值）

所传递的值，否则是：

SUCC（前一个值）（*PRED*（前一个值））

所传递的值。

如果上述计算得到的值大于（小于）指定的结束值，或引发了异常 *OVERFLOW*，则该循环动作终止。

3. **单元枚举**

在带（不带）**DOWN** 说明的单元枚举情况下，对下列单元集合中的每个单元执行一次指定的动作语句表。该单元集合是由数组单元所表示的数组单元的所有数组元素组成的集合，或是由串单

元所表示的串单元的所有串元素组成的集合。如果指定了数组表达式或串表达式，而它又不是一个单元，则隐式创建一个包含指定的值的单元。所创建的单元的生存期就是该循环动作。如果指定的值具有动态类，则所创建的单元的模式是动态的。带单元枚举的循环动作的语义就如同在每次执行指定的动作语句表之前遇到了一个下列单元等同说明语句：

DCL 〈循环计数器〉〈模式〉 **LOC** ：＝ 〈部分组合对象〉（〈下标〉）；

其中模式是指定的数组单元的**元素**模式或 & *名字* (*1*)。如果指定的串单元的模式是一个**定长**串模式，则 & *名字*是一个与该串单元的模式**同义**的虚拟**同义模式**名字，否则 & *名字*是一个与该串单元的模式的**分量**模式**同义**的虚拟**同义模式**名字；下标最初被置为该单元的模式的**下界**（**上界**），在其后每次执行动作语句表之前被置为 *SUCC*（下标）(*PRED*（下标））。如果串单元的**实际长度**等于 0，则不执行指定的动作语句表。如果下标在某次执行指定的动作语句表之后恰好等于指定的单元的模式的**上界**（**下界**），则该循环动作终止。当执行该循环动作时，只对部分组合单元求值一次。

静态性质：循环计数器附有一个名字串，它就是其定义性出现的名字串。

值枚举：

由循环计数器定义的名字是一个**值枚举**名字。

步长枚举：

由循环计数器定义的名字的类是开始值、步长值（如果存在的话）以及结束值的类的**结果类**。

范围枚举：

由循环计数器定义的名字的类是 M 值类，其中 M 是离散模式名字。

幂集枚举：

循环计数器所定义的名字的类是 M 值类，其中 M 是（**强**）幂集表达式的模式的**成员**模式。

单元枚举：

循环计数器所定义的名字是一个**单元枚举**名字。其模式是数组单元或数组表达式的模式的**元素**模式，或是串模式 & *名字* (*1*)，其中 & *名字*是一个与串单元的模式或串表达式的类的**根**模式**同义**的虚拟**同义模式**名字。

如果数组单元的模式的**元素布局**是 **NOPACK**，则**单元枚举**名字是**可引用的**。

静态条件：开始值、结束值和步长值（如果出现的话）的类必须是两两**相容的**。

值枚举中循环计数器的类的**根**模式不能是一个**编号**集合模式。

动态条件：如果步长值所传递的值不大于 0，则引发异常 *RANGEFAIL*。该异常在该循环动作的分程序之外出现。

例子：

```
4.17    FOR i := 1 TO c                                  (1.1)
15.37   FOR EVER                                         (1.1)
4.17    i := 1 TO c                                      (3.1)
9.12    'j := MIN (sieve) BY MIN (sieve) TO max          (3.1)
14.28   i IN INT (1:100)                                 (3.2)
```

6.5.3 当型控制

语法：

〈当型控制〉::= (1)

WHILE〈布尔表达式〉 (1.1)

语义：恰好在进入指定的动作语句表之前对指定的布尔表达式求值（如果还指定了步长型控制，则在计算步长型控制之后进行）。如果它传递值 TRUE，则进入指定的动作语句表，否则该循环动作终止。

例子：

7.35 **WHILE** *n>=1* (1.1)

6.5.4 开域部分

语法：

〈开域部分〉::= (1)

WITH〈开域控制〉{,〈开域控制〉}* (1.1)

〈开域控制〉::= (2)

〈结构单元〉 (2.1)

| 〈结构原值〉 (2.2)

注意：如果开域控制是结构单元，则通过把它解释为一个单元而不是一个原值来解决语法二义性问题。

语义：在每个开域控制内指定的结构单元或结构值的模式的（**可见的**）域名字可以用来直接访问这些域。

可见性规则如同是针对附着于指定的结构单元或结构原值的模式之上的每个**域**名字引入了一个域名字定义性出现，并且它与该**域**名字具有相同的名字串一样。

如果指定了一个结构单元，则隐式地说明了若干个访问名字，它们分别与该结构单元的模式的那些域名字具有相同的名字串，且分别表示该结构单元的那些子单元。

如果指定了一个结构原值，则隐式地定义了若干值名字，它们分别与该（**强**）结构原值的模式的那些域名字具有相同的名字串，且分别表示该结构值的那些子值。

当进入该循环动作时，就在进入时计算一次指定的诸结构单元和/或结构值，计算次序是未定义的，可能是以混杂的次序进行的。

静态性质：为每个**域**名字引入的（虚拟）定义性出现与该**域**名字的域名字定义性出现具有相同的名字串。

如果指定了一个结构原值，则在带该开域部分的循环动作中的一个（虚拟）定义性出现定义了一个**值开域**名字。其类是 M 值类，其中 M 是该结构原值的结构模式中被用作**值开域**名字的那个**域**名字的模式。

如果指定了一个结构单元，则在带该开域部分的循环动作中的一个（虚拟）定义性出现定义了一个**单元开域**名字。其模式是该结构单元的模式中被用作**单元开域**名字的那个**域**名字的模式。如果与之相关的**域**名字的**域布局**是 **NOPACK**，则**单元开域**名字是**可引用的**。

例子：

15.58 **WITH** *each* (1.1)

6.6 出口动作

语法：

〈出口动作〉::= (1)

EXIT 〈标号名字〉 (1.1)

语义：出口动作用于退出一个带括号动作语句或一个模块。程序从紧跟在带标号名字的最内层带括号动作语句或模块之后的语句处恢复执行。

静态条件：出口动作必须位于带括号动作语句或模块内部，且在这些语句或模块前面的定义性出现与标号名字具有相同的名字串。

如果出口动作位于某个进程定义或过程定义内部，则被退出的带括号动作语句或模块也必须位于该进程定义或过程定义内部（即出口动作不能用来退出进程或过程）。

出口动作不能附着处理程序。

例子：

*15.62 **EXIT** find_counter* (1.1)

6.7 调用动作

语法：

〈调用动作〉::= (1)

〈过程调用〉 (1.1)

| 〈内部子程序调用〉 (1.2)

〈过程调用〉::= (2)

{ 〈过程名字〉 | 〈过程原值〉}

([〈实参表〉]) (2.1)

〈实参表〉::= (3)

〈实参〉{, 〈实参〉}* (3.1)

〈实参〉::= (4)

〈值〉 (4.1)

| 〈单元〉 (4.2)

〈内部子程序调用〉::= (5)

〈内部子程序名字〉([〈内部子程序参数表〉]) (5.1)

〈内部子程序参数表〉::= (6)

〈内部子程序参数〉{, 〈内部子程序参数〉}* (6.1)

〈内部子程序参数〉::= (7)

〈值〉 (7.1)

| 〈单元〉 (7.2)

| 〈非保留名字〉[(〈内部子程序参数表〉)] (7.3)

注意：如果实参或内部子程序参数是单元，则通过将它解释为一个单元而不是一个值来解决语法二义性问题。

语义：调用动作引起一个过程或一个内部子程序的调用。过程调用引起对由过程原值传递的值所指示的**通用**过程或由过程名字指示的过程的调用。实参表中指定的实际值和单元被传递给被调用的过程。

内部子程序调用要么是CHILL内部子程序调用，要么是实现内部子程序调用（分别参见6.20和13.1节）。

值、单元或任何一个不是**保留**简单名字串的程序定义的名字都可以作为内部子程序参数传递。内部子程序调用可以返送一个值或一个单元。

内部子程序可能是类属的，即其类（若它是一个**值**内部子程序调用）或其模式（若它是一个**单元**内部子程序调用）可能不仅依赖于内部子程序名字，而且依赖于所传递的实参的静态性质以及调用处的静态上下文。

静态性质：过程调用附着下列性质：**参数说明**表、可能具有的**结果说明**、可能为空的异常名字集合、**通用性**、**递归性**以及它可能是**区域内的**（指定过程名字的过程调用才会具有此性质，参见11.2.2节）。这些性质是从过程名字或从与过程原值的类是**相容的**任何模式继承来的（在后一种情况下，**通用性**总是**通用的**）。

当且仅当某个具有**结果说明**的过程调用的**结果说明**中指明了**LOC**时，它是单元过程调用，否则是值过程调用。

内部子程序名字是一个CHILL或实现定义的名字，这种名字被认为是在假想的最外层进程定义的可达区内或在任何上下文内定义的（参见10.8节）。

如果内部子程序调用传递一个单元，则它是一个**单元**内部子程序调用。如果内部子程序调用传递一个值，则它是一个**值**内部子程序调用。

静态条件：出现在过程调用中的实参个数必须与其**参数说明**的个数相等。过程调用中的实参与其（按位置）对应的**参数说明**的相容性要求是：

- 如果参数说明具有属性**IN**（缺省的参数属性是**IN**），则实参必须是值，且其类与对应的参数说明内指定的模式是**相容的**。参数说明中指定的模式不能具有**非值性质**。相对该过程调用而言实参必须是**区域安全的**值；
- 如果参数说明具有**INOUT**或**OUT**属性，则实参必须是单元，且其模式必须与M值类是**相容的**，其中M是相应参数说明内指定的模式。（实参）单元的模式必须是静态的，且既没有**只读性质**，又没有**非值性质**。实参是单元，能把它视为这样的值，该值相对过程调用而言必须是**区域安全的**；
- 如果参数说明具有属性**INOUT**，则参数说明中指定的模式必须与M值类是**相容的**，其中M是实参单元的模式；
- 如果参数说明具有不带**DYNAMIC**说明的**LOC**属性，则实参要么是满足下面两个条件的单元，即该单元是可引用的，且参数说明中指定的模式与该实参单元的模式是**读相容的**；要么是满足下列条件的值，即该值不是单元，但其类与参数说明中指定的模式是**相容的**；
- 如果参数说明具有带**DYNAMIC**说明的**LOC**属性，则实参要么是满足下面两个条件的单元，即该单元是**可引用的**，且参数说明中指定的模式与该实参单元的模式是**动态读相容的**；要么是满足下列条件的值，即该值不是单元，但其类与参数说明内指定的模式的某种参数化模式是**相容的**；
- 如果参数说明具有**LOC**属性，则：
  - — 如果实参是单元，则它必须与该过程调用具有相同的**区域性**；
  - — 如果实参是值，则它相对该过程调用而言必须是**区域安全的**。

动态条件：过程调用或内部子程序调用可能引发它所附着的异常名字集合中的任一种异常。如果过程原值传递值*NULL*，则过程调用引发异常*EMPTY*。如果存储需求得不到满足，则过程调用引发异常*SPACE-FAIL*。如果被调用的过程的**递归性**是**非递归的**，则该过程不能直接或间接地调用自己。

参数传递可能引发下列异常：

- 如果参数说明具有**IN**、**INOUT**属性，则相对参数说明内指定的模式而言调用点（实参）值应满足

赋值条件（参见 6.2 节），并且在该过程被调用之前引发可能的异常；

- 如果参数说明具有 **INOUT** 或 **OUT** 属性，则相对（实参）单元的模式而言过程返回点形参的局部值应满足赋值条件（参见 6.2 节），并且在该过程已返回后引发可能的异常；
- 如果参数说明具有 **LOC** 属性，且实参是一个非单元的值，则相对参数说明的模式而言调用点（实参）值应满足赋值条件，并且在该过程被调用之前引发可能的异常（参见 6.2 节）。

过程原值不能传递下述过程值：该过程是在某个进程定义内部被定义的，且执行该过程调用的进程的激活与该进程定义的激活不同（假想的最外层进程除外）。此外，它所表示的过程的生存期不能已经结束。

例子：

4.18 op (a, b, d, order-1) *(1.1)*

6.8 结果和返回动作

语法：

〈返回动作〉::= *(1)*

RETURN [〈结果〉] *(1.1)*

〈结果动作〉::= *(2)*

RESULT 〈结果〉 *(2.1)*

〈结果〉::= *(3)*

〈值〉 *(3.1)*

| 〈单元〉 *(3.2)*

导出语法： 带结果的返回动作是从 **DO RESULT** 〈结果〉; **RETURN**; **OD** 导出的。

语义： 结果动作用来建立一个过程调用所传送的结果。该结果可以是一个单元，也可以是一个值。返回动作使得程序控制从返回动作所在的过程定义的某次过程调用中返回。如果被调用的过程返送一个结果，则该结果是由最后执行的那个结果动作建立的。如果未执行到任何结果动作，则该过程调用分别传递**未定义的**单元或**未定义的**值。

静态性质： 结果动作和返回动作附有一个**过程**名字，它就是最内层包围它的过程定义的名字。

静态条件： 返回动作和结果动作必须出现在某个过程定义内部。仅当其**过程**名字具有**结果说明**时，该过程的过程体中才能出现结果动作。

处理程序不能附加到一个没有结果的返回动作之上。

如果一个结果动作的**过程**名字的**结果说明**中指定了 **LOC**（**LOC DYNAMIC**），则结果必须是满足下列条件的单元，即**结果说明**内指定的模式与该单元的模式是**读相容的**（**动态读相容的**）。如果**结果说明**内未指明 **NONREF**，则该单元必须是**可引用的**。结果是满足下列条件的单元，即它与附着在该结果动作之上的**过程**名字具有相同的**区域性**。

如果在一个结果动作的**过程**名字的**结果说明**内未指明 **LOC**，则结果必须是值，且其类与**结果说明**中指定的模式是**相容的**。结果是满足下列条件的值，即它相对附着在该结果动作之上的**过程**名字而言是**区域安全的**。

动态条件： 如果在其**过程**名字的**结果说明**中未指明 **LOC**，则相对其**过程**名字的**结果说明**中指定的模式而言结果动作中的值必须满足赋值条件。

例子：

4.21 **RETURN** *(1.1)*
1.6 **RESULT** *i+j* *(2.1)*
5.19 *c* *(3.1)*

6.9 转向动作

语法：

〈转向动作〉::= (1)
GOTO〈标号名字〉 (1.1)

语义：转向动作导致控制转移。程序从带标号名字的动作语句处恢复执行。

静态条件：如果转向动作位于某个过程定义或进程定义内部，则由标号名字指出的标号也必须在该定义内部被定义（即转向动作不能跳出一个过程或进程调用）。

转向动作不能附加处理程序。

6.10 断言动作

语法：

〈断言动作〉::= (1)
ASSERT〈布尔表达式〉 (1.1)

语义：断言动作提供测试一个条件的手段。

动态条件：如果布尔表达式传递值 *FALSE*，则引发异常 *ASSERTFAIL*。

例子：

4.7 **ASSERT** *b*>*0* **AND** *c*>*0* **AND** *order*>*0* (1.1)

6.11 空动作

语法：

〈空动作〉::= (1)
〈空〉 (1.1)
〈空〉::= (2)

语义：空动作不执行任何动作。

静态条件：空动作不能附加处理程序。

6.12 引发动作

语法：

〈引发动作〉::= (1)

CAUSE〈异常名字〉 (1.1)

语义：引发动作引发一个异常，该异常的名字由异常名字指出。

静态条件：引发动作不能附加处理程序。

例子：

4.9 **CAUSE** *wrong_input* (*1.1*)

6.13 启动动作

语法：

〈启动动作〉::= (*1*)

〈启动表达式〉 (*1.1*)

语义：启动动作对指定的启动表达式求值（参见 5.2.14 节），但并不使用求得的结果实例值。

例子：

14.45 **START** *call_distributor* () (*1.1*)

6.14 停止动作

语法：

〈停止动作〉::= (*1*)

STOP (*1.1*)

语义：停止动作终止执行它的进程（参见 11.1 节）。

静态条件：停止动作不能附加处理程序。

6.15 继续动作

语法：

〈继续动作〉::= (*1*)

CONTINUE〈事件单元〉 (*1.1*)

语义：继续动作对事件单元求值。

如果指定的事件单元之上附着一个非空的被延迟进程集合，则它们中的具有最高优先数的进程之一被重新激活。如果存在几个这样的进程，则按实现定义的方式选取其中之一。如果不存在这样的进程，则继续动作没有其它作用。

如果一个进程成为重新激活的，则把它从所有它是其成员之一的被延迟进程集合中移去。

例子：

13.25 **CONTINUE** *resource_freed* (*1.1*)

6.16 延迟动作

语法：

〈延迟动作〉::= (1)

DELAY 〈事件单元〉[〈优先数〉] (1.1)

〈优先数〉::= (2)

PRIORITY 〈整数字面值表达式〉 (2.1)

语义：延迟动作对事件单元求值。

然后引发异常 *DELAYFAIL*（见下）或执行它的进程成为被延迟的。

如果执行进程成为被延迟的，它就成为附着到指定的事件单元之上的被延迟进程集合中的一员，并带有一个优先数。如果指定了优先数，则上述优先数就是整数字面值表达式所传递的值，否则是 0（最低优先数）。

动态性质：当执行延迟动作的进程到达可能使它成为被延迟的执行点时，它成为**可超时的**。当它离开那个程序点时它不再是**可超时的**。

静态条件：整数字面值表达式不能传递一个负值。

动态条件：如果事件单元附着一个具有**事件长度**的模式，且该**事件长度**等于已被延迟在该事件单元之上的进程个数，则引发异常 *DELAYFAIL*。

在执行进程被延迟在事件单元之上期间，事件单元的生存期不能结束。

例子：

13.18 **DELAY** *resource_freed* (1.1)

6.17 延迟情况动作

语法：

〈延迟情况动作〉::= (1)

DELAY CASE [**SET** 〈实例单元〉[〈优先数〉]; | 〈优先数〉;]

{〈延迟选择项〉}$^{+}$**ESAC** (1.1)

〈延迟选择项〉::= (2)

(〈事件表〉):〈动作语句表〉 (2.1)

〈事件表〉::= (3)

〈事件单元〉{,〈事件单元〉}* (3.1)

语义：延迟情况动作对实例单元（如果出现的话）和其内任一延迟选择项中的所有事件单元求值，且这种求值是以一种未定义的，且可能是混杂的次序进行。

于是，出现异常 *DELAYFAIL*（见下）或执行它的进程成为被延迟的。

如果执行进程成为被延迟的，则它可成为附着于所有指明的事件单元之上的被延迟进程集合中的一员，并带有一个优先数。如果指定了优先数，则上述优先数就是整数字面值表达式所传递的值，否则是 0（最低优先数）。

如果该被延迟的进程因另一个进程在某个事件单元之上执行了一个继续动作而成为重新激活的，

则进入相应的动作语句表。如果几个延迟选择项指定同一个事件单元，则选择哪个延迟选择项是未定义的。在进入相应的动作语句表之前，如果指定了实例单元，则标识进程的实例值被存入该实例单元，其中该进程已执行了上述连续动作。

动态性质：当执行延迟情况动作的进程到达它可能成为被延迟的执行点时，它成为**可超时的**。当它离开那个程序点时不再是**可超时的**。

静态条件：实例单元的模式不能具有**只读性质**。优先数内的整数字面值表达式不能传递一个负值。

动态条件：如果任一事件单元附着一个具有**事件长度**的模式，且该**事件长度**等于已被延迟在该事件单元之上的进程个数时，则引发异常 *DELAYFAIL*。

在执行延迟情况动作的进程被延迟在指定的诸事件单元之上期间，这些事件单元的生存期都不能结束。

如果存储需求得不到满足，则引发异常 *SPACEFAIL*。

例子：

```
14.26   DELAY CASE
            (operator_is_ready): /* some actions */
            (switch_is_closed): DO FOR i IN INT (1:100);
                                CONTINUE operator_is_ready;
                                /* empty the queue */
                                OD;
        ESAC                                                    (1.1)
```

6.18 发送动作

6.18.1 概述

语法：

〈发送动作〉::= (1)

〈发送信号动作〉 (1.1)

| 〈发送缓冲区动作〉 (1.2)

语义：发送动作启动某个发送进程处同步信息的传送。详细的语义据同步对象是一个信号还是一个缓冲区而定。

6.18.2 发送信号动作

语法：

〈发送信号动作〉::= (1)

SEND 〈信号名字〉[(〈值〉{, 〈值〉}*)]

[**TO** 〈实例原值〉] [〈优先数〉] (1.1)

语义：发送信号动作对可能出现的值表及实例原值求值，且求值次序未定，可能是以混杂的次序进行的。

由信号名字指示的信号连同指定的诸值以及一个优先数一并传送。如果指定了优先数，则传送的优

先数是整数字面值表达式所传递的值，否则是0（最低优先数）。

如果**信号**名字附有一个**进程**名字，则只有具有该名字的进程才能接收该信号；如果指定了实例原值，则只有由它标识的进程才能接收该信号。否则，任何进程都能接收该信号。

如果该信号上附着一个非空的被延迟进程集合，而且其中有一个或多个被延迟进程能接收该信号，则它们之一将被重新激活。如果存在几个这样的进程，则将根据实现定义的方式选择其中之一成为重新激活的。如果不存在这样的进程，此信号成为挂起的。

如果一个进程成为重新激活的，则将它从所有它是其中一员的被延迟进程集合中移去。

静态条件：所出现的值的个数必须等于信号名字的模式的个数。每个值的类必须与信号名字的对应模式是**相容的**。所出现的值不能是**区域内的**（参见11.2.2节）。优先数中整数字面值表达式不能传递一个负值。

动态条件：相对信号名字的对应模式而言每个值应满足赋值条件。

如果实例原值传递值 *NULL*，则引发异常 *EMPTY*。

在执行发送信号动作的那一程序点，由实例原值传递的值所指示的进程的生存期不能已经结束。

如果信号名字附有一个**进程**名字，且该**进程**名字不是由实例原值传递的值所指明的进程的名字，则引发异常 SENDFAIL。

例子：

15.78 **SEND** *ready* **TO** *received_user* *(1.1)*
15.86 **SEND** *readout(count)* **TO** *user* *(1.1)*

6.18.3 发送缓冲区动作

语法：

〈发送缓冲区动作〉::= *(1)*
SEND 〈缓冲区单元〉(〈值〉)[〈优先数〉] *(1.1)*

语义：发送缓冲区动作以任意次序计算缓冲区单元和值。

如果该缓冲区单元之上附着了一个非空的被延迟进程集合，则这些进程之一将被重新激活。如果存在几个这样的进程，则按实现定义的方式选取其中之一。如果不存在这样的进程，且该缓冲区单元的容量已满，则执行该发送缓冲区动作的进程成为被延迟的，并带有一个优先数；否则，指定的值被存入该缓冲区单元，并带有一个优先数。如果指定了优先数，则上述优先数就是其内整数字面值表达式所传递的值，否则是0（最小优先数）。如果缓冲区单元附着一个具有**缓冲区长度**的模式，且该长度等于已存入该缓冲区单元内的值的个数，则相应缓冲区的容量已满。

如果执行进程成为被延迟的，则它可成为附着于指定的缓冲区单元之上的被延迟发送进程集合中的一员。如果一个进程成为重新激活的，则应将它从所有包含它的被延迟进程集合中移去。

动态性质：当执行发送缓冲区动作的进程到达它可能成为被延迟的执行点时，它成为**可超时的**。当它离开该程序点时就不再是**可超时的**。

静态条件：值的类必须与缓冲区单元的模式的**缓冲区元素**模式是**相容的**。值不能是**区域内的**（参见11.2.2节）。优先数中的整数字面值表达式不能传递一个负值。

动态条件：相对缓冲区单元模式的**缓冲区元素**模式而言值应满足赋值条件。在该进程成为被延迟之前引发可能的异常。

在执行发送缓冲区动作的进程被延迟在缓冲区单元之上期间，缓冲区单元的生存期不能结束。

例子：

16.119 **SEND** *user*—〉（[*ready*，—〉*counter_buffer*]） *(1.1)*

6.19 接收情况动作

6.19.1 概述

语法：

〈接收情况动作〉::= *(1)*

〈接收信号情况动作〉 *(1.1)*

|〈接收缓冲区情况动作〉 *(1.2)*

语义：接收情况动作接收由某个发送动作传送来的同步信息。详细语义取决于所使用的同步对象是信号还是缓冲区单元。进入接收情况动作不一定导致执行进程成为被延迟的（参见第11章）。

6.19.2 接收信号情况动作

语法：

〈接收信号情况动作〉::= *(1)*

RECEIVE CASE [**SET** 〈实例单元〉;]

{〈信号接收选择项〉}$^{+}$

[**ELSE** 〈动作语句表〉] **ESAC** *(1.1)*

〈信号接收选择项〉::= *(2)*

（〈信号名字〉[**IN** 〈定义性出现表〉]）:〈动作语句表〉 *(2.1)*

语义：接收信号情况动作对可能出现的实例单元求值。

然后，执行进程立即接收一个信号。如果不能接收一个信号且指定了 **ELSE**，则进入跟在 **ELSE** 后面的动作语句表；否则执行进程成为被延迟的。如果在其内所有信号接收选择项中指明的信号名字之一是挂起的，且该信号能被此进程接收，则执行进程立即接收该信号。如果能被接收的信号多于一个，则按实现定义的方式从中选择一个具有最高优先数的信号予以接收。

如果执行进程成为被延迟的，则它成为附着于每个指定的信号之上的被延迟进程集合之一员。如果由另一个进程执行某个发送信号动作而使得该被延迟进程成为重新激活的，则它接收相应信号。

如果执行进程收到一个信号，则进入相应的动作语句表。在进入之前，如果指定了实例单元，则标识发送已收到的信号的那个进程的实例值被存入该实例单元。如果已收到的信号的**信号**名字附着一个模式表，则应指明一个**值接收名字**表。该信号携带一个值表，且在被进入的动作语句表内这些**值接收**名字分别表示那个值表中的对应值。

静态性质：在信号接收选择项的定义性出现表中的一个定义性出现定义了一个**值接收**名字。其类是 M 值类，其中 M 是它前面的信号名字所附着的模式表中对应的模式。

动态性质：当执行接收信号情况动作的进程到达可能使它成为被延迟的执行点时，它成为**可超时的**。当它离开该程序点时不再是**可超时的**。

静态条件：实例单元的模式不能具有**只读性质**。

出现的所有信号名字必须是不同的。

当且仅当某个信号接收选择项中的信号名字具有非空的模式集合时，该信号接收选择项中必须指定任选的 **IN** 和定义性出现表。定义性出现表中名字的个数必须等于信号名字的模式的个数。

动态条件：如果存储需求得不到满足，则引发异常 *SPACEFAIL*。

例子：

15.83
```
RECEIVE CASE
    (advance): count + := 1;
    (terminate):
        SEND readout(count) TO user;
        EXIT work_loop;
ESAC                                                    (1.1)
```

6.19.3 接收缓冲区情况动作

语法：

〈接收缓冲区情况动作〉::= (1)

RECEIVE CASE [**SET** 〈实例单元〉;]
{〈缓冲区接收选择项〉}$^{+}$
[**ELSE** 〈动作语句表〉]
ESAC (1.1)

〈缓冲区接收选择项〉::= (2)

(〈缓冲区单元〉**IN** 〈定义性出现〉):〈动作语句表〉 (2.1)

语义：接收缓冲区情况动作对可能出现的实例单元和所有缓冲区接收选择项内指定的那些缓冲区单元求值。这种求值次序是未定义的，可能是以混杂的次序进行的。

然后，执行进程立即接收一个值。但如果不能接收值，且指定了 **ELSE**，则进入跟在 **ELSE** 后面的动作语句表，否则执行进程成为被延迟的。如果在指定的诸缓冲区单元之一内已存入一个值，或其上附着一个被延迟的发送进程，则执行进程立即接收一个值。如果能被接收的值多于一个，则按实现定义的方式从中选取一个带最高优先数的值予以接收。

如果执行进程成为被延迟的，则它成为附着于每个指定的缓冲区单元之上的被延迟进程集合之一员。如果该被延迟的进程因另一个进程执行了一个发送缓冲区动作而成为重新激活的，则它接收一个值。

如果执行进程收到了一个值，则进入相应的动作语句表。如果几个缓冲区接收选择项指定同一个缓冲区单元，则选取其中哪个缓冲区接收选择项是未定义的。在进入相应动作语句表之前，如果指定了实例单元，则用来标识已经发送出所收到的值的那个进程的实例值被存入该实例单元。在所进入的动作语句表内，指定的**值接收**名字表示已收到的值。

如果执行进程从某个缓冲区单元处接收了一个值，且在该缓冲区单元之上附着的被延迟发送进程集合非空，则另一个进程成为重新激活的。如果已收到的值曾存入指定的缓冲区单元，则该重新激活的进程是附着最高优先数的进程，否则是发送已收到的值的那个进程。在前一种情况下，由重新激活的进程所发送的值被存入指定的缓冲区单元（该缓冲区的容量仍然是满的）。如果可被重新激活的进程多于一个，则按实现定义的方式选取其中一个成为重新激活的。将重新激活的进程从该缓冲区单元之上附着的被延迟发送进程集合中移去。

静态性质：在缓冲区接收选择项内的定义性出现定义了一个**值接收**名字。其类是M值类，其中M是该缓冲区接收选择项前面的缓冲区单元的模式的**缓冲区元素**模式。

动态性质：当执行接收缓冲区情况动作的进程到达可能使得它成为被延迟的执行点时，它成为**可超时的**。当它离开该程序点时不再是**可超时的**。

静态条件：实例单元的模式不能具有**只读性质**。

动态条件：如果存储需求得不到满足，则引发异常SPACEFAIL。

在执行接收缓冲区情况动作的进程在指定的诸缓冲区单元之上被延迟期间，它们的生存期不能结束。

6.20 CHILL内部子程序调用

语法：

〈*CHILL* 内部子程序调用〉::= (*1*)
〈*CHILL* 一般内部子程序调用〉 (*1.1*)
| 〈*CHILL* 单元内部子程序调用〉 (*1.2*)
| 〈*CHILL* 值内部子程序调用〉 (*1.3*)

预定义名字：CHILL内部子程序名字被预定义为**内部子程序**名字（参见6.7节）。

语义：*CHILL* 内部子程序调用要么是不传递结果的 *CHILL* 一般内部子程序调用（参见6.20.1节），要么是传递一个单元的 *CHILL* 单元内部子程序调用（参见6.20.2节），要么是传递一个值的 *CHILL* 值内部子程序调用（参见6.20.3节）。

静态性质：如果 *CHILL* 内部子程序调用是 *CHILL* 单元内部子程序调用，则它是一个**单元**内部子程序调用；如果它是 *CHILL* 值内部子程序调用，则它是一个**值**内部子程序调用。

6.20.1 CHILL一般内部子程序调用

语法：

〈*CHILL* 一般内部子程序调用〉::= (*1*)
〈结束内部子程序调用〉 (*1.1*)
| 〈入出一般内部子程序调用〉 (*1.2*)
| 〈计时一般内部子程序调用〉 (*1.3*)

语义：*CHILL* 一般内部子程序调用是一个既不传递值，又不传递单元的内部子程序调用。第七章定义用于输入输出的一般内部子程序。第九章定义用于计时的一般内部子程序。

6.20.2 CHILL单元内部子程序调用

语法：

〈*CHILL* 单元内部子程序调用〉::= (*1*)
〈入出单元内部子程序调用〉 (*1.1*)

语义：*CHILL* 单元内部子程序调用是传递一个单元的内部子程序调用。第七章定义用于输入输出的单元内部子程序。

6.20.3 CHILL 值内部子程序调用

语法：

〈*CHILL* 值内部子程序调用〉::= (*1*)

NUM （〈离散表达式〉） (*1.1*)

| *PRED* （〈离散表达式〉） (*1.2*)

| *SUCC* （〈离散表达式〉） (*1.3*)

| *ABS* （〈整数表达式〉） (*1.4*)

| *CARD* （〈幂集表达式〉） (*1.5*)

| *MAX* （〈幂集表达式〉） (*1.6*)

| *MIN* （〈幂集表达式〉） (*1.7*)

| *SIZE* （｛〈单元〉|〈模式变元〉｝） (*1.8*)

| *UPPER* （〈上下变元〉） (*1.9*)

| *LOWER* （〈上下变元〉） (*1.10*)

| *LENGTH* （〈长度变元〉） (*1.11*)

|〈分配内部子程序调用〉 (*1.12*)

|〈入出值内部子程序调用〉 (*1.13*)

|〈时钟值内部子程序调用〉 (*1.14*)

〈模式变元〉::= (*2*)

〈模式名字〉 (*2.1*)

|〈数组模式名字〉（〈表达式〉） (*2.2*)

|〈串模式名字〉（〈整数表达式〉） (*2.3*)

|〈变体结构模式名字〉（〈表达式表〉） (*2.4*)

〈上下变元〉::= (*3*)

〈数组单元〉 (*3.1*)

|〈数组表达式〉 (*3.2*)

|〈数组模式名字〉 (*3.3*)

|〈串单元〉 (*3.4*)

|〈串表达式〉 (*3.5*)

|〈串模式名字〉 (*3.6*)

|〈离散单元〉 (*3.7*)

|〈离散表达式〉 (*3.8*)

|〈离散模式名字〉 (*3.9*)

〈长度变元〉::= (*4*)

〈串单元〉 (*4.1*)

|〈串表达式〉 (*4.2*)

注意：如果上下变元是（数组、串、离散）单元，则上述语法二义性问题的解决办法是把它解释为一个单元而不是一个表达式。如果长度变元是串单元，则上述语法是二义的，其解决办法是将它解释为一个单元而不是一个表达式。

语义：*CHILL* 值内部子程序调用是传递一个值的内部子程序调用。

NUM 传递一个整数值，该值与其变元所传递的值具有相同的内部表示。

PRED 和 *SUCC* 分别传递其变元的离散值的下一个较小或较大的离散值。

ABS 传递其变元的值的绝对值。

CARD、*MAX* 和 *MIN* 是在幂集值上定义的。*CARD* 传递在其变元所传递的幂集值中成员值的个数。

MAX 和 *MIN* 分别传递其变元所传递的幂集值中最大的成员值和最小的成员值。

SIZE 是在**可引用的**单元和（可能是动态的）模式之上定义的。在第一种情况下，它传递那个单元所占据的可寻址内存单位的数量；在第二种情况下，它传递具有那个模式的**可引用的**单元将占据的可寻址内存单位的数量。如果模式变元是模式名字，则该模式是静态的，否则是基于模式变元中指定的模式名字的某种动态参数化模式，它具有模式变元中指定的参数。在第一种情况下，单元在运行时刻不被求值。

UPPER 和 *LOWER* 分别定义在（可能是动态的）：

- 数组、串和离散单元之上，此时它们分别传递指定的单元的模式的**上界**和**下界**；
- 数组和串表达式之上，此时它们分别传递指定的值的类的模式的**上界**和**下界**；
- **强**离散表达式之上，此时它们分别传递指定的值的类的模式的**上界**和**下界**；
- 数组、串和离散**模式**名字之上，此时它们分别传递指定的模式的**上界**和**下界**。

LENGTH 传递其变元的**实际长度**。

静态性质：*NUM* 内部子程序调用的类是 *INT* 导出类。当且仅当其变元是**常数**或**字面值**时，*NUM* 内部子程序调用是**常数**。

PRED 或 *SUCC* 内部子程序调用的类是其变元的**结果类**。当且仅当其变元是**常数**（**字面值**）时，*PRED* 或 *SUCC* 内部子程序调用是**常数**（**字面值**）。

ABS 内部子程序调用的类是其变元的**结果类**。当且仅当其变元是**常数**（**字面值**）时，该内部子程序调用是**常数**（**字面值**）。

CARD 内部子程序调用的类是 *INT* 导出类。当且仅当其变元是**常数**时，*CARD* 内部子程序调用是**常数**。

MAX 和 *MIN* 内部子程序调用的类是 M 值类，其中 M 是幂集表达式的模式的**成员**模式。当且仅当其变元是**常数**时，*MAX* 和 *MIN* 内部子程序调用是**常数**。

SIZE 内部子程序调用的类是 *INT* 导出类。如果其变元的模式是静态的，则 *SIZE* 内部子程序调用是**常数**。

UPPER 和 *LOWER* 内部子程序调用的类定义如下：

- 如果上下变元是数组单元、数组表达式或数组模式名字，则它是 M 值类，其中 M 分别是数组单元、数组表达式或数组模式名字的**下标**模式；
- 如果上下变元是串单元、串表达式或串模式名字，则它是 INT 导出类；
- 如果上下变元是离散单元、离散表达式或离散模式名字，则它是 M 值类，其中 M 分别是离散单元、离散表达式或离散模式名字的模式。

满足下列条件之一时，*UPPER* 和 *LOWER* 内部子程序调用是**常数**：（1）上下变元是（数组、串或离散）模式名字；（2）上下变元是数组单元或串单元，且其模式是静态的；（3）上下变元是数组表达式或串表达式，且它具有静态类；（4）上下变元是离散表达式或离散单元。

LENGTH 内部子程序调用的类是 *INT* 导出类。

静态条件：如果 *PRED* 或 *SUCC* 内部子程序调用的变元是**常数**，则它不能是分别传递由该变元的类的**根**模式定义的最小离散值或最大离散值。*PRED* 和 *SUCC* 的离散表达式变元的**根**模式不能是一个**编号**集合模式。

如果 *MAX* 或 *MIN* 内部子程序调用的变元是**常数**，则它不能传递空幂集值。

SIZE 的单元变元必须是**可引用的**。

用作 *UPPER* 或 *LOWER* 变元的离散表达式必须是**强**值。

对非单个模式名字的模式变元而言，应满足下列相容性要求：

- 表达式的类必须与数组模式名字的**下标**模式是**相容的**；
- 变体结构模式名字必须是**可参数化的**，并且，表达式表中的表达式个数必须与变体结构模式名字的类表中的类的个数相等，且每个表达式的类必须与上述类表中相应的类是**相容的**。

动态条件：如果对由该变元的类的**根**模式所定义的最小或最大离散值实施 *PRED* 或 *SUCC* 内部子程序调用，则 *PRED* 或 *SUCC* 引发异常 *OVERFLOW*。

如果 *NUM* 和 *CARD* 内部子程序调用的结果值不属于由 *INT* 定义的值集，则它们引发异常 *OVERFLOW*。

如果对空幂集值实施 *MAX* 或 *MIN* 内部子程序调用，则它们引发异常 *EMPTY*。

如果 *ABS* 内部子程序调用的结果值超出由该变元的类的**根**模式所定义的边界，则它引发异常 *OVERFLOW*。

如果在模式变元中出现下列情况之一，则引发异常 *RANGEFAIL*：

- 表达式传递的值不属于由数组模式名字的**下标**模式定义的值集；
- 整数表达式传递一个负值或传递一个大于串模式名字的**串长度**的值；
- 表达式表中的任一表达式传递一个不属于模式 *M* 所定义的值集的值，且变体结构模式名字的类表中与该表达式对应的类是 *M* 值类（即该表达式是**强**值）。

例子：

9.12 MIN (sieve) (1.7)
11.47 PRED (col_1) (1.2)
11.47 SUCC (col_1) (1.3)

6.20.4 动态存储管理内部子程序

语法：

〈分配内部子程序调用〉::= (1)
GETSTACK （〈模式变元〉[，〈值〉]） (1.1)
|*ALLOCATE* （〈模式变元〉[，〈值〉]） (1.2)

〈结束内部子程序调用〉::= (2)
TERMINATE （〈引用原值〉） (2.1)

语义：*GETSTACK* 和 *ALLOCATE* 创建了一个具有指定的模式的单元，并传递一个对所创建的单元的引用值。*GETSTACK* 在栈上创建这个单元（参见10.9节）。*GETSTACK* 创建了一个单元，其模式是模式变元所表示的模式，并传递一个引用该单元的引用值。如果指定了值，则所创建的单元被初始化为值所传递的值；否则它被初始化为**未定义的**值（参见4.1.2节）。

TERMINATE 结束由引用原值传递的值所引用的单元的生存期。作为推论，实现可以释放该单元所占据的存储。

静态性质：*GETSTACK* 和 *ALLOCATE* 内部子程序调用的类是 M 引用类，其中 M 是模式变元所表示的模式。M 要么是指定的模式名字，要么分别是如下构造出的**参数化**模式：

&〈数组模式名字〉（〈表达式〉）或
&〈串模式名字〉（〈整数表达式〉）或
&〈变体结构模式名字〉（〈表达式表〉），

如果 *GETSTACK* 或 *ALLOCATE* 内部子程序调用被某个区域包围，则它是**区域内的**，否则是**区域外的**。

静态条件：如果在 *GETSTACK* 或 *ALLOCATE* 内部子程序调用中指定了值，则该值的类必须与模式变元表示的模式是**相容的**。当模式变元表示的模式是一个动态模式时，这种检查是动态的。

如果 *GETSTACK* 或 *ALLOCATE* 的第一个变元具有**只读性质**，则必须指定第二个变元。

如果在 *GETSTACK* 或 *ALLOCATE* 内部子程序调用中出现值，则该值相对所创建的单元必须是**区域安全的**。

动态性质：当且仅当一个引用值是由某个 *ALLOCATE* 内部子程序调用返送的值时，它是一个**已分配**引用值。

动态条件：如果存储需求得不到满足，则 *GETSTACK* 引发异常 *SPACEFAIL*。

如果存储需求得不到满足，则 *ALLOCATE* 引发异常 *ALLOCATEFAIL*。

对 *GETSTACK* 和 *ALLOCATE* 而言，由值传递的值相对模式变元所表示的模式而言应满足赋值条件。

如果引用原值传递值 *NULL*，则 *TERMINATE* 引发异常 *EMPTY*。

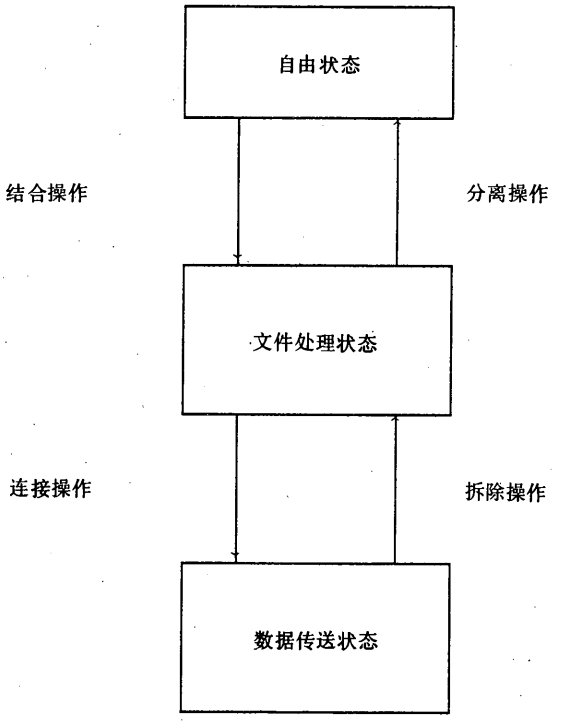
引用原值必须传递一个**已分配**引用值。它所引用的单元的生存期不能已经结束。

7 输入和输出

7.1 入/出参考模型

入/出参考模型以实现无关的方式用于描述输入/输出设施。对某个给定的结合单元而言，该模型区分三种状态，即自由状态、文件处理状态和数据传送状态。

下图显示了这三种状态以及可能发生的状态变迁。



在自由状态下，结合单元不包含值，它与CHILL 外部对象无关。

在文件处理状态下，结合单元点包含一个结合，允许实施类似创建文件、删除文件、改变文件性质的操作。

在数据传送状态下，某个访问单元（子单元）被连接到结合单元之上。可通过该访问单元对附着在结合单元中的结合之上的外部对象写入数据或从该外部对象中读出数据：即读和写操作。

该模型认为在外界（即 CHILL 程序的外部环境）中存在若干对象，这种对象在实现中常常被称为数据库、文件或设备。这类外界对象在该模型中称为文件。一个文件可能是一台物理设备、一条通信线路或就是某个文件管理系统中的一个文件。广义地说，文件是能产生或消耗数据的对象。

加工 CHILL 中的文件需要一个结合。结合操作创建一个结合，并且该结合标识一个文件。结合具有若干属性，这些属性描述已经或可能附着到该结合之上的一个文件的性质。

在自由状态下，CHILL 程序与外界对象之间不存在任何交互或联系。结合操作将该模型的状态从自由状态修改为文件处理状态。结合操作取一个结合单元作为一个变元，且为某个外界对象引入某种实现定义的表示，而该结合正是为此外界对象而创建的。结合操作的其它变元可被用来指出为该外部对象建立什么样的结合以及所创建的结合的属性的初值。某种特殊的结合也隐含着可以对附着于该结合之上的文件实施的（与实现有关的）一组操作。

在文件处理状态下，可以通过一个结合加工一个文件及其属性，假定该结合允许实施这种特殊的操作。一般说来，对改变一个文件的性质的诸操作而言，必须为该文件提供一个排它性结合。

该模型认为结合通常是排它性的，即对某个给定的外界对象而言，在某一时刻只存在一个结合。然而，实现可以允许为同一个外界对象创建多个结合，假定该对象能为不同的用户（程序）所共享，或为同一个程序中的不同结合所共享。在文件处理状态下，所有操作取某个结合作为一个变元。

分离操作用来终止为某个外界对象创建的一个结合，该操作引起从文件处理状态返回自由状态的状态变迁。

只有在数据传送状态下才能向一个文件写入或从一个文件读出数据。数据传送操作要求将一个**访问**单元连接到为该文件创建的某个结合之上。连接操作将一个访问单元连接到一个结合之上，并将该模型的状态修改为数据传送状态。连接操作取一个结合单元和一个访问单元作为变元，而该结合单元包括一个为下列文件创建的结合，即可以通过该访问单元向该文件内写入数据或自该文件读出数据。连接操作的其它变元表示该访问单元为哪种传送操作而必须被连接的以及该文件被定位在哪个记录处。在任一时刻，最多有一个访问单元能被连接到某个结合单元之上。

拆除操作取访问单元作为变元，并拆除该访问单元与先前所连接的结合之间的连接。该操作使本模型的状态从数据传送状态改回文件处理状态。

在数据传送状态下，传送操作必须取一个访问单元作为它的一个变元。CHILL 提供了两种传送操作，其一是将数据从某个文件传送到该程序的读操作；其二是将数据从该程序传送到某个文件的写操作。传送操作使用该访问单元的记录模式将 CHILL 值变换为该文件的记录，反之亦然。

在入/出参考模型中，一个文件被视为一个值数组，该数组的每个元素对应于该文件的一个记录。该数组的元素模式由连接操作确定为所连接的访问单元的记录模式。该文件的每个记录被赋予一个下标值，该下标值唯一地标识此文件的某个记录。在连接和传送操作的语义描述中将使用三个特殊的下标值，即**基本**下标、**当前**下标和**传送**下标。**基本**下标由连接操作设置，并且在下一个连接操作之前一直保持不变。**基本**下标用于计算传送操作中的**传送**下标和连接操作中的**当前**下标。**传送**下标表示该文件中被实施传送操作的记录的位置，而**当前**下标表示该文件当前被定位的记录。

7.2 结合值

7.2.1 概述

结合值反映了已附着在其上或能附着在其上的文件的性质。某个特殊的结合也隐含着能对可能附着在其上的文件实施的（与实现有关的）一组操作。

结合值在 CHILL 中没有相应的表示，但它被包含在结合模式单元中。不存在表示结合模式的值的表达式。只能通过取一个结合单元作为参数的内部子程序来加工结合值。

7.2.2 结合值的属性

结合值具有若干属性，这些属性描述该结合以及可能或能够附着在其上的文件的性质。

语言定义了结合值的下列属性：

- **存在的**：表示一个（可能为空的）文件是否被附着在该结合之上；
- **可读的**：表示当一个文件被附着到该结合之上时，对该文件是否可以实施读操作；
- **可写的**：表示当一个文件被附着到该结合之上时，对该文件是否可以实施写操作；
- **可求下标的**：表示当一个文件被附着到该结合之上时，是否允许对该文件的记录进行随机存取；
- **顺序的**：表示当一个文件被附着到该结合之上时，是否允许顺序存取该文件的记录；
- **可变的**：表示当一个文件被附着到该结合之上时，该文件的记录的**大小**在该文件内部是否可以变化。

上面这些属性具有布尔值。当创建该结合时初始化这些属性值，且在该结合上实施的某些特殊操作可以修改这些属性值。上面列出的属性只是语言定义的结合值的属性，而实现可以根据自身的需要增添其它属性。

7.3 访问值

7.3.1 概述

访问值被存入具有访问模式的单元中。为了从 CHILL 程序向其外界中的某个文件传送数据或自外界中的某个文件向 CHILL 程序传送数据，必须使用一个访问单元。

CHILL 中不存在访问值的表示，但它包含在访问模式单元中。不存在表示访问模式的值的表达式。只能通过取某个访问单元作为参数的内部子程序来加工访问值。

7.3.2 访问值的属性

访问值具有若干属性，这些属性描述访问值的动态性质、传送操作的语义以及可能引发异常的条件。

CHILL 定义了访问值的下列属性：

- **使用**：表示该访问单元被连接到某个结合之上以后可以实施哪种传送操作。该属性由连接操作设置；
- **超出文件**：表示最后一个读操作所计算的**传送**下标是否在该文件中。连接操作将该属性初始化为 *FALSE*，且其后的每个读操作都设置该属性。

7.4 输入输出内部子程序

7.4.1 概述

CHILL 定义了对结合单元和访问单元进行操作以及对结合值和访问值之属性进行检测或更改的语言定义的内部子程序。

下面各节将讨论这些内部子程序。

语法：

〈入出值内部子程序调用〉::= (1)
〈结合属性内部子程序调用〉 (1.1)
| 〈已结合内部子程序调用〉 (1.2)
| 〈访问属性内部子程序调用〉 (1.3)
| 〈读记录内部子程序调用〉 (1.4)
| 〈取文本内部子程序调用〉 (1.5)

〈入出一般内部子程序调用〉::= (2)
〈分离内部子程序调用〉 (2.1)
| 〈修改内部子程序调用〉 (2.2)
| 〈连接内部子程序调用〉 (2.3)
| 〈拆除内部子程序调用〉 (2.4)
| 〈写记录内部子程序调用〉 (2.5)
| 〈文本内部子程序调用〉 (2.6)
| 〈置文本内部子程序调用〉 (2.7)

〈入出单元内部子程序调用〉::= (3)
〈结合内部子程序调用〉 (3.1)

静态条件：如果某个入出内部子程序中的内部子程序参数是结合单元、访问单元或文本单元，则它必须是**可引用的**。

7.4.2 与外界对象的结合

语法：

〈结合内部子程序调用〉::= (1)

ASSOCIATE（〈结合单元〉[,〈结合参数表〉]) (1.1)

〈已结合内部子程序调用〉::= (2)

ISASSOCIATED（〈结合单元〉) (2.1)

〈结合参数表〉::= (3)

〈结合参数〉{,〈结合参数〉}* (3.1)

〈结合参数〉::= (4)

〈单元〉 (4.1)

| 〈值〉 (4.2)

语义： *ASSOCIATE* 创建了一个表示与某个外界对象之间联系的结合。它将结合单元初始化为所创建的结合，同时也对所创建的结合的属性进行初始化，并将该结合单元作为本调用的结果返送。所创建的特定的结合由结合参数表内出现的单元和/或值来确定，而这些单元（值）的模式（类）和语义由实现定义。

如果结合单元包含一个结合，则 *ISASSOCIATED* 返送 *TRUE*，否则返送 *FALSE*。

静态性质： *ISASSOCIATED* 内部子程序调用的类是 BOOL 导出类。*ASSOCIATE* 内部子程序调用的模式是结合单元的模式。

ASSOCIATE 内部子程序调用的**区域性**是结合单元的**区域性**。

静态条件： 每个结合参数的模式和类由实现定义。

动态条件： 如果结合单元已经包含了一个结合，或由于某种实现定义的原因不能创建该结合，则 *ASSOCIATE* 引发异常 *ASSOCIATEFAIL*。

例子：

20.21 ASSOCIATE (file_association,"DSK:RECORDS.DAT"); (1.1)

7.4.3 与外界对象的分离

语法：

〈分离内部子程序调用〉::= (1)

DISSOCIATE（〈结合单元〉) (1.1)

语义： *DISSOCIATE* 终止对某个外界对象的结合。在该结合被终止之前，拆除仍被连接在包含在指定的结合单元内的结合之上的访问单元。

动态条件： 如果结合单元未包含任何结合，则 *DISSOCIATE* 引发异常 *NOTASSOCIATED*。

例子：

22.38　DISSOCIATE (association);　　(1.1)

7.4.4　结合属性的存取

语法：

〈结合属性内部子程序调用〉::=　　(1)

EXISTING (〈结合单元〉)　　(1.1)

|*READABLE* (〈结合单元〉)　　(1.2)

|*WRITEABLE* (〈结合单元〉)　　(1.3)

|*INDEXABLE* (〈结合单元〉)　　(1.4)

|*SEQUENCIBLE* (〈结合单元〉)　　(1.5)

|*VARIABLE* (〈结合单元〉)　　(1.6)

语义：*EXISTING*、*READABLE*、*WRITEABLE*、*INDEXABLE*、*SEQUENCIBLE* 和 *VARIABLE* 分别返送结合单元中所含结合的**存在的**、**可读的**、**可写的**、**可求下标的**、**顺序的**和**可变的**属性的值。

静态性质：结合属性内部子程序调用的类是 *BOOL* 导出类。

动态条件：如果结合单元未包含任何结合，则结合属性内部子程序调用引发异常 *NOTASSOCIATED*。

7.4.5　结合属性的修改

语法：

〈修改内部子程序调用〉::=　　(1)

CREATE (〈结合单元〉)　　(1.1)

| *DELETE* (〈结合单元〉)　　(1.2)

| *MODIFY* (〈结合单元〉[,〈修改参数表〉])　　(1.3)

〈修改参数表〉::=　　(2)

〈修改参数〉{,〈修改参数〉}　　(2.1)

〈修改参数〉::=　　(3)

〈值〉　　(3.1)

|〈单元〉　　(3.2)

语义：*CREATE* 创建一个空文件，并将它附着到由结合单元所表示的结合之上。如果该操作成功，则指定的结合的**存在的**属性被置为 *TRUE*。

DELETE 将一个文件从结合单元所表示的结合上分离开来，并删除这个文件。如果该操作成功，则指定的结合的**存在的**属性被置为 *FALSE*。

MODIFY 提供了改变某个满足下列条件的外界对象的性质的手段：已为该外部对象创建了一个由结合单元表示的结合。修改参数表中出现的单元和/或值用来指明必须修改哪些性质。这些单元（值）的模式（类）和语义由实现定义。

动态条件：如果结合单元未包含任何结合，则 *CREATE*、*DELETE* 和 *MODIFY* 引发异常 *NOTASSOCIATED*。

如果出现下列情况之一，则 *CREATE* 引发异常 *CREATEFAIL*：

- 指定的结合的**存在的**属性是 *TRUE*；
- （实现定义的）建立文件失败。

如果出现下列情况之一，则 *DELETE* 引发异常 *DELETEFAIL*：

- 指定的结合的**存在的**属性是 *FALSE*；
- （实现定义的）删除文件失败。

如果由修改参数表定义的诸性质不允许或不能加以修改，则 *MODIFY* 引发异常 *MODIFYFAIL*。能导致引发该异常的具体条件由实现定义。

例子：

```
21.39    CREATE (outassoc);                                    (1.1)
21.69    DELETE (curassoc);                                    (1.2)
```

7.4.6 与访问单元的连接

语法：

```
〈连接内部子程序调用〉 ::=                                      (1)
    CONNECT (〈传送单元〉,〈结合单元〉,
    〈使用表达式〉[,〈定位表达式〉[,〈下标表达式〉]])            (1.1)

〈传送单元〉 ::=                                                (2)
    〈访问单元〉                                                (2.1)
  | 〈文本单元〉                                               (2.2)

〈使用表达式〉 ::=                                              (3)
    〈表达式〉                                                  (3.1)

〈位置表达式〉 ::=                                              (4)
    〈表达式〉                                                  (4.1)

〈下标表达式〉 ::=                                              (5)
    〈表达式〉                                                  (5.1)
```

预定义名字：为了控制由内部子程序 *CONNECT* 完成的连接操作，语言中预定义了两个**同义模式**名字，即 *USAGE* 和 *WHERE*。它们的定义模式分别是 **SET** (*READONLY*, *WRITEONLY*, *READWRITE*) 和 **SET** (*FIRST*, *SAME*, *LAST*)。

模式 *USAGE* 定义的诸值指出该访问单元被连接到一个结合之上后适用哪种传送操作，而模式 *WHERE* 定义的诸值反映该连接操作如何对附着在指定的结合之上的文件进行定位。

语义：*CONNECT* 将由传送单元表示的访问单元连接到包含在结合单元内的结合上去。指定的结合之上必须附着一个文件，即该结合的**存在的**属性必须是 *TRUE*。

如果传送单元是一个访问单元，则由传送单元表示的访问单元就是该访问单元，否则是文本单元的**访问**子单元。

由使用表达式传递的值指出指定的访问单元被连接到该文件之上后通过该访问单元可以实施何种传送操作。如果使用表达式传递值 *READONLY*，则通过该连接仅能实施读操作；如果使用表达式传递值 *WRITEONLY*，则对该连接仅能实施写操作；如果使用表达式传递值 *READWRITE*，则通过该连接既能实施读操作，又能实施写操作。

如果指定的访问单元具有**下标**模式，则所表示的结合的**可求下标的**属性必须是 *TRUE*，否则所表示的结合的**顺序的**属性必须是 *TRUE*。

CONNECT 对附着到指定的结合之上的文件（重新）定位，即它为该文件建立一个（新的）**基本**下标和**当前**下标。该（新的）**基本**下标依赖于位置表达式所传递的值。

- 如果位置表达式传递值 *FIRST*，或未指明位置表达式，则**基本**下标被置为0，即该文件被定位于第一个记录之前；
- 如果位置表达式传递值 *SAME*，则**基本**下标被置为该文件的**当前**下标，即不改变文件位置；
- 如果位置表达式传递值 *LAST*，则**基本**下标被置为 *N*，其中 *N* 表示该文件内记录个数，即该文件被定位于最后一个记录之后。

在设置**基本**下标后，*CONNECT* 再建立（新的）**当前**下标。此**当前**下标依赖于任选的下标表达式描述：

- 如果 *CONNECT* 操作中未指定下标表达式，则**当前**下标被置为（新的）**基本**下标；
- 如果指定了下标表达式，则**当前**下标被置为：

  基本下标 + *NUM* (*v*) − *NUM* (*l*)

  其中 *l* 表示指定的访问单元的**下标**模式的**下界**，而 *v* 表示下标表达式所传递的值。

如果指定的访问单元是为顺序写操作而被连接的（即该访问单元没有**下标**模式，且使用表达式传递值 *WRITEONLY*），则此文件内其下标比（新的）**当前**下标大的那些记录将从该文件中被删除，即 *CONNECT* 操作可能截断或清空文件。

没有**下标**模式的访问单元不能被如下连接到某个结合之上，即通过该连接既能实施读又能实施写操作。

在指定的结合被连接到由传送单元表示的访问单元之上之前，将隐式拆除先前已被连接到该结合之上的任何访问单元。

CONNECT 将指定的访问单元的**文件超出**属性初始化为 *FALSE*，并根据使用表达式所传递的值设置该访问单元的**使用**属性的值。

静态性质：传送单元附着的模式分别是指定的访问单元的模式或文本单元的**访问**模式。

静态条件：如果指定了下标表达式，则指定的传送单元的模式必须具有**下标**模式，下标表达式所传送的值的类必须与那个**下标**模式是**相容的**。传送单元与结合单元必须具有相同的**区域性**。

使用表达式所传递的值的类必须与 *USAGE* 导出类是**相容的**。

位置表达式所传送的值的类必须与 *WHERE* 导出类是**相容的**。

动态条件：如果结合单元未包含任何结合，则 *CONNECT* 引发异常 *NOTASSOCIATED*。

如果出现下列情况之一，则 *CONNECT* 引发异常 *CONNECTFAIL*：

- 指定的结合的**存在的**属性是 *FALSE*；
- 指定的结合的**可读的**属性是 *FALSE*，且使用表达式传送值 *READONLY* 或 *READWRITE*；
- 指定的结合的**可写的**属性是 *FALSE*，且使用表达式传送值 *WRITEONLY* 或 *READWRITE*；
- 指定的结合的**可求下标的**属性是 *FALSE*，且指定的访问单元具有**下标**模式；
- 指定的结合的**顺序的**属性是 *FALSE*，且指定的访问单元没有**下标**模式；
- 位置表达式传递值 *SAME*，而包含在结合单元内的结合未被连接到某个访问单元之上；
- 指定的结合的**可变的**属性是 *FALSE*，且指定的访问单元具有**动态记录**模式，而使用表达式传送值 *WRITEONLY* 或 *READWRITE*；
- 指定的结合的**可变的**属性是 *TRUE*，且指定的访问单元具有**静态记录**模式，而使用表达式传递值 *READONLY* 或 *READWRITE*；
- 指定的访问单元没有**下标**模式，而使用表达式传递值 *READWRITE*；
- 受实现定义的条件所限，包含在结合单元中的结合不能被连接到指定的访问单元之上。

如果指定的访问单元的**下标**模式是一个范围模式，且下标表达式传送超出该范围模式所定义的边界的一个值，则 *CONNECT* 引发异常 *RANGEFAIL*。

如果指定的文本单元的**访问引用**传递值 *NULL*，则 *CONNECT* 引发异常 *EMPTY*。

例子：

20.22 *CONNECT (record_file, file_association, READWRITE);* *(1.1)*
20.22 *READWRITE* *(3.1)*

7.4.7 与访问单元的拆除

语法：

〈拆除内部子程序调用〉::= *(1)*
DISCONNECT (〈传送单元〉) *(1.1)*

语义： *DISCONNECT* 将由传送单元表示的访问单元从先前被连接到其上的结合处拆除。

动态条件： 如果由传送单元表示的访问单元并未被连接到某个结合之上，则 *DISCONNECT* 引发异常 *NOTCONNECTED*。

7.4.8 访问单元属性的存取

语法：

〈访问属性内部子程序调用〉::= *(1)*
GETASSOCIATION (〈传送单元〉) *(1.1)*
|*GETUSAGE* (〈传送单元〉) *(1.2)*
|*OUTOFFILE* (〈传送单元〉) *(1.3)*

语义： 对由传送单元所表示的访问单元连接到其上的结合单元而言，*GETASSOCIATION* 返送引用值。如果上述访问单元未被连接到某个结合之上，则 *GETASSOCIATION* 返送值 *NULL*。

GETUSAGE 返送指定的访问单元的**使用**属性值，即如果该访问单元只为读(写)操作而被连接的话，则 *GETUSAGE* 返送值 *READONLY* (*WRITEONLY*)，否则返送值 *READWRITE*。

OUTOFFILE 返送指定的访问单元的**超出文件**属性值，即如果最后一个读操作计算出一个不在该文件之内的**传送**下标，则 *OUTOFFILE* 返送值 *TRUE*，否则返送值 *FALSE*。

静态性质： *GETASSOCIATION* 内部子程序调用的类是 *ASSOCIATION* 引用类。*GETASSOCIATION* 内部子程序调用的**区域性**是传送单元的**区域性**。

OUTOFFILE 内部子程序调用的类是 *BOOL* 导出类。

GETUSAGE 内部子程序调用的类是 *USAGE* 导出类。

动态条件： 如果指定的访问单元未被连接到某个结合之上，则 *GETUSAGE* 和 *OUTOFFILE* 引发异常 *NOTCONNECTED*。

例子：

21.47 *OUTOFFILE* (*infiles* (*FALSE*)) *(1.3)*

7.4.9 数据传送操作

语法：

〈读记录内部子程序调用〉::= (1)

READRECORD（〈访问单元〉[，〈下标表达式〉]

[，〈存储单元〉]) (1.1)

〈写记录内部子程序调用〉::= (2)

WRITERECORD（〈访问单元〉[，〈下标表达式〉]，〈写表达式〉) (2.1)

〈存储单元〉::= (3)

〈静态模式单元〉 (3.1)

〈写表达式〉::= (4)

〈表达式〉 (4.1)

注意：如果访问单元具有**下标**模式，则 *READRECORD* 的语法是二义的，解决办法是将第二个变元解释为下标表达式而不是存储单元。

语义：为了从 CHILL 程序向一个文件中传送数据，定义了内部子程序 *WRITERECORD*；为了从一个文件向 CHILL 程序中传送数据，定义了内部子程序 *READRECORD*。访问单元必须具有**记录**模式，并且该访问单元必须被连接到某个结合之上，以便向或自附着在该结合之上的那个文件传送数据。传送方向不能与访问单元的**使用**属性值发生矛盾。

在数据传送发生前，计算**传送**下标，即计算欲被传送的记录在该文件中的位置。如果访问单元没有**下标**模式，则**传送**下标是**当前**下标加*1*；否则如下计算**传送下**标：

传送下标：=**基本**下标+*NUM* (*v*) −*NUM* (*l*) +*1*

其中 *l* 是访问单元的**下标**模式的**下界**，而 *v* 表示由下标表达式传递的值。如果成功传送了具有经上述计算而得到的**传送**下标的记录，则**当前**下标变为该**传送**下标。

读操作：

READRECORD 将外界中的一个文件内的数据传送给 CHILL 程序。

如果计算得到的**传送**下标不在该文件内，则访问单元的**超出文件**属性被置为 TRUE；否则，该文件被定位于**传送**下标指定的位置，读出相应的记录，并将访问单元的**超出文件**属性置为 FALSE。

被读的记录不能传送**未定义的**值。如果根据访问单元的**记录**模式判明要从该文件内读出的记录不是一个合法的值，则该读操作的作用由实现定义。

如果指定了存储单元，则所读的记录的值被赋值给该单元。如果未指明存储单元，则上述值被赋值给一个隐式创建的单元。当访问单元被拆除或重新连接时，该隐式创建的单元的生存期就结束了。是仅由连接操作一次性创建上述隐式单元，还是每次读操作都创建一个这种单元，这一点没有定义。*READRECORD* 在上述两种情况下都返回一个引用值，该引用值引用被赋予所读出的值的那个（可能是动态模式的）单元。

作为 *READRECORD* 内部子程序调用的一个结果，如果访问单元的**超出文件**属性被置为 *TRUE*，则值 *NULL* 作为该调用的一个结果返送。

写操作：

WRITERECORD 从 CHILL 程序向外界中的某个文件传送数据。该文件被定位于计算所得的**传送**下标的记录处，并向该记录存入值。

写记录成功后，如果该文件中实际记录个数小于等于上述**传送**下标，则将**传送**下标置为该文件的记录个数。

由 *WRITERECORD* 写的记录是写表达式所传递的值。

静态性质：由 *READRECORD* 读出的值的类是 M 值类。如果访问单元具有**静态记录**模式，则 M 就是访问单元的**记录**模式；否则，访问单元具有**动态记录**模式，且 M 是基于该模式的某种动态参数化模式。这种动态参数化记录模式的参数分别是：

- 在访问单元的**动态记录**模式是一个串模式的情况下，它是读出的串值的动态**串长度**；
- 在访问单元的**动态记录**模式是一个数组模式的情况下，它是读出的数组值的动态**上界**；
- 在访问单元的**动态记录**模式是一个**变体**结构模式的情况下，它们是读出的与结构值的模式有关的（标志域）值表。

如果未指定存储单元，则 *READRECORD* 内部子程序调用的类是 M 引用类（M 如上定义），否则是 S 引用类，其中 S 是存储单元的模式。

如果指定了存储单元，则 *READRECORD* 内部子程序调用的**区域性**就是该存储单元的**区域性**，否则是访问单元的**区域性**。

静态条件：访问单元必须具有**记录**模式。

若访问单元没有**下标**模式，则不能指定下标表达式；但如果访问单元具有**下标**模式，则必须指定下标表达式。下标表达式所传递的值的类必须与那个**下标**模式是**相容的**。

存储单元必须是**可引用的**。

存储单元的模式不能具有**只读性质**。

如果指定了存储单元，那么，若访问单元具有**静态记录**模式或**变长串记录**模式，则存储单元的模式必须与访问单元的**记录**模式**等价**，否则存储单元的模式必须与基于访问单元的**记录**模式的某种动态参数化模式**等价**。这种动态参数化模式的参数是读出的值的那些参数。

如果访问单元具有**静态记录**模式或**变长串记录**模式，则由写表达式传递的值的类必须与访问单元的**记录**模式是**相容的**，否则必须存在一个基于上述**记录**模式的某种动态参数化模式，它与写表达式的类是**相容的**。写表达式的值相对上面提到的模式而言必须满足赋值条件。

动态条件：如果上面提到的相容性检验的动态部分失败，则引发异常 *RANGEFAIL* 或 *TAGFAIL*。

如果访问单元未被连接到某个结合之上，则 *READRECORD* 和 *WRITERECORD* 内部子程序调用引发异常 *NOTCONNECTED*。

如果访问单元的**下标**模式是一个范围模式，而下标表达式所传递的值位于该范围模式定义的界限之外，则 *READRECORD* 和 *WRITERECORD* 内部子程序调用引发异常 *RANGEFAIL*。

如果出现下列情况之一，则 *READRECORD* 内部子程序调用引发异常 *READFAIL*：

- 访问单元的**使用**属性的值是 *WRITEONLY*；
- 访问单元的**超出文件**属性的值是 *TRUE*，且访问单元是为顺序读操作而连接的；
- 由于外界原因，对具有计算所得的**传送**下标的记录读失败。

如果出现下列情况之一，则 *WRITERECORD* 内部子程序调用引发异常 *WRITEFAIL*：

- 访问单元的**使用**属性的值是 *READONLY*；
- 由于外界因素，对具有计算所得的**传送**下标的记录写失败。

如果出现异常 *RANGEFAIL* 或 *NOTCONNECTED*，则它们是在任何属性值被修改前以及该文件被定位前引发的。

例子：

```
20.24    READRECORD (record_file, curindex, record_buffer);    (1.1)
22.25    READRECORD (fileaccess);                              (1.1)
20.32    WRITERECORD (record_file, curindex, record_buffer);   (2.1)
21.61    WRITERECORD (outfile, buffers( flag ));               (2.1)
20.24    record_buffer                                         (3.1)
21.61    buffers( flag )                                       (4.1)
```

7.5 文本输入输出

7.5.1 概述

文本输出操作允许以人类可读的形式输出 CHILL 值的表示，而文本输入操作完成反向转换。

文本传送操作在基本 CHILL 输入/输出模型之上定义，并在可顺序或随机访问的文件之上实施。这种文件的记录长度可以是固定的，也可以是可变的。

（文本）入/出模型认为每个记录附有一个（可能为空的）定位信息，定位信息在实现中常被称作托架控制或托架控制字符。

在 CHILL 中，加工文本文件需要一个结合，从 CHILL 程序向文本文件传送数据或从文本文件向 CHILL 程序传送数据需要一个被连接到与该文件有关的某个结合之上的**文本**单元。

文本传送操作可以在下列 CHILL 值上实施，这些值可能成为某个文本文件中的记录，文本传送操作也可以在某些 CHILL 单元上实施，这种单元不必与该程序的任何入/出活动有关。

一般说来，不能保证将一段文本恢复成这样一个 CHILL 值，即该值与生成该文本的那个 CHILL 相同，但这取决于该文本所用的特殊表示。

文本值包含在具有文本模式的单元内。按人类可读的形式传送数据时需要一个文本单元。

在 CHILL 中不存在文本值的表示，但它们可包含在具有文本模式的单元内。不存在表示文本模式的值的表达式。只能通过取某个文本单元作为参数的内部子程序来加工文本值。

7.5.2 文本值的属性

文本值具有若干属性，这些属性用来描述该文本值的动态性质。语言定义了下列属性：

- **实际下标**：它指出将被读或写的**文本记录**的下一个字符位置。**实际下标**具有模式 *INT* (*0* : *L*)，其中 *L* 是该文本值的模式的**文本长度**。当创建一个文本单元时，**实际下标**被初始化为 0；
- **文本记录引用**：它指出对相应文本单元的**文本记录**子单元的引用值。它具有模式 **REF** M，其中 M 是该文本值的模式的**文本记录**模式；
- **访问引用**：它指出对相应文本单元的**访问**子单元的引用值。它具有模式 **REF** M，其中 M 是该文本值的模式的**访问**模式。

7.5.3 文本传送操作

语法：

〈文本内部子程序调用〉 ::= (1)

READTEXT （〈文本入出变元表〉） (1.1)

| *WRITETEXT* （〈文本入出变元表〉） (1.2)

〈文本入出变元表〉 ::= (2)

〈文本变元〉 [，〈下标表达式〉]，

〈格式变元〉 [，〈入出表〉] (2.1)

〈文本变元〉 ::= (3)

〈文本单元〉 (3.1)

| 〈字符串单元〉 (3.2)
| 〈字符串表达式〉 (3.3)

〈格式变元〉::= (4)
〈字符串表达式〉 (4.1)

〈入出表〉::= (5)
〈入出表元素〉{,〈入出表元素〉}* (5.1)

〈入出表元素〉::= (6)
〈值变元〉 (6.1)
| 〈单元变元〉 (6.2)

〈单元变元〉::= (7)
〈离散单元〉 (7.1)
| 〈串单元〉 (7.2)

〈值变元〉::= (8)
〈离散表达式〉 (8.1)
| 〈串表达式〉 (8.2)

注意：如果入出表元素是一个单元，则上述语法是二义的，解决方法是把该入出表元素解释为单元变元而不是值变元。

语义：*READTEXT* 对由文本变元表示的**文本记录**实施包含在格式变元之内的转换、编辑及入/出控制功能。这（可能）产生一串值，它们被分别赋值给入出表中相应的元素，且赋值次序与它们在入出表中指定的次序相同。*WRITETEXT* 完成相反的操作。不会执行隐式的入/出操作。

如果文本变元是字符串单元或字符串表达式，则实施转换和编辑功能，且与外界毫无关系。此时，**实际下标**表示在 *READTEXT* 内部子程序调用开始时隐式创建的一个单元，且被初始化为0。**文本记录**就是由字符串单元或字符串表达式表示的字符串，而**文本长度**就是其**串长度**。

入出表的元素可以是：

- 值变元和单元变元，或
- 下面叙述的**可变的**短语宽度。

格式变元与入出表间的关系

格式变元所传递的值必须具有格式控制串的形式（参见7.5.4节）。

在某个文本入出内部子程序调用的执行期间，从左到右地扫描由格式变元表示的格式控制串（参见7.5.4节）和入出表，解释出现的每个格式文本和格式说明，并采取如下合适的动作：

a. 格式文本：

在 *READTEXT* 内，**文本记录**在**实际下标**位置应包含一个串片，该串片与格式文本所传递的串相等。在 *WRITETEXT* 中，由格式文本传递的串被传送给**文本记录**。其语义就如同遇到了一个是*%C* 的格式说明，并遇到了一个传递与格式文本所传递的串值相等的串值的入出表元素。

b. 格式说明：

如果该格式说明包含重复因子，则它等价于一串格式元素，其内出现的格式元素的个数与由重复因子表示的数目相等。

如果该格式说明是格式短语，则它包含一个控制码。如果该控制码是转换短语，则从入出表中取出一个入出表元素，并对它实施由转换码、转换限定符和短语宽度选定的转换功能（参见7.5.5节）。如果该控制码是编辑短语或入出短语，则对指定的文本变元实施由编辑码或入出码和短语宽度选定的编辑或入出功能，此时不考虑指定的入出表（参见7.5.6和7.5.7节）。

如果该格式短语内所含的短语宽度是**可变的**，则从入出表中取出一个值，它表示转换或编辑控制

功能的**宽度**参数。

如果该格式说明是带括号短语，则扫描其内所含的格式控制串。

当已到达格式控制串所传递的串的尾部时，格式控制串的解释就终止。

按入出表元素在入出表中出现的次序扫描它们。

静态条件：如果文本变元是字符串单元，则其模式必须是一个**变长**串模式。

如果文本变元不是文本单元，或尽管文本变元是文本单元，但其**访问**模式没有**下标**模式，则不能指明下标表达式。如果文本变元是文本单元，且其**访问**模式具有**下标**模式，则必须指明下标表达式。由下标表达式传递的值的类必须与上述**下标**模式是**相容的**。

WRITETEXT 内部子程序调用内的文本变元必须是一个单元。

用作文本变元的字符串单元必须是**可引用的**。

动态条件：出现下列情况时引发 *TEXTFAIL* 异常：

- 由格式变元传递的串值不能作为格式控制串的终结产生式而导出；或
- 试图将一个小于 0 或大于**文本长度**的值赋值给**实际下标**；或
- 在解释期间，已到达格式控制串的尾部，但入出表尚未被扫描完；或入出表已没有新的入出表元素可取，但格式控制串还包含更多的转换码或**可变的**短语宽度；或
- 遇到一个入出短语，但文本变元不是一个文本单元；或
- 在 READTEXT 内遇到一个格式文本，且**文本记录**在**实际下标**位置并不包含一个与该格式文本所传送的串相同的串。

如果执行了某个入出控制功能，但违反了已定义的任何一个动态条件，则可能引发为 *READRECORD* 和 *WRITERECORD* 内部子程序调用所定义的任一异常。

例子：

26.18 WRITETEXT (output, "%B%/", 10) (1.2)

7.5.4 格式控制串

语法：

〈格式控制串〉::= (1)

[〈格式文本〉]{〈格式说明〉[〈格式文本〉]}* (1.1)

〈格式文本〉::= (2)

{〈非百分号字符〉|〈百分号〉} (2.1)

〈百分号〉::= (3)

% % (3.1)

〈格式说明〉::= (4)

% [〈重复因子〉]〈格式元素〉 (4.1)

〈重复因子〉::= (5)

{〈数字〉}+ (5.1)

〈格式元素〉::= (6)

〈格式短语〉 (6.1)

|〈带括号短语〉 (6.2)

〈格式短语〉::= (7)

〈控制码〉[%.] (7.1)

〈控制码〉::= (8)

〈转换短语〉 (8.1)

|〈编辑短语〉 (8.2)

|〈入出短语〉 (8.3)

〈带括号短语〉::= (9)

(〈格式控制串〉%) (9.1)

注意：格式说明由第一个不可能是格式元素之一部分的字符终止。在格式元素内部不能使用空格和格式控制符。圆号（·）可用于终止一个格式短语，它属于那个格式短语，只起分隔作用。为了在格式文本内部使用百分号字符（%），它必须写两次（%%）。

语义：格式控制串描述将被传送的值的外部形式以及在被传送的记录内部的数据布局。格式控制串由格式文本出现和格式说明出现组成。格式文本表示被传送的记录的固定部分，而格式说明表示 CHILL 值的外部表示。格式控制串允许对**文本记录**进行编辑或控制实际入出操作。

包含重复因子和格式短语的格式说明与多个相同的格式说明出现是等效的，其中每个格式说明出现仅含该格式短语，且这种格式说明出现的个数与重复因子所传递的值相等。重复因子可以是0，此时不考虑该格式说明。如“%3D4”等价于“%D4%D4%D4”。

假定重复因子中出现的数字取十进制表示形式。

根据重复因子的值反复扫描带括号短语内的格式控制串。如果未指定重复因子，则认为取缺省值1。

例子：

26.20 size=%C%/ (1.1)

7.5.5 转换

语法：

〈转换短语〉::= (1)

〈转换码〉{〈转换限定符〉}* [〈短语宽度〉] (1.1)

〈转换码〉::= (2)

B|O|H|C (2.1)

〈转换限定符〉::= (3)

L|E|P〈字符〉 (3.1)

〈短语宽度〉::= (4)

{〈数字〉}+|*V* (4.1)

导出语法：其内不出现短语宽度的转换短语是其内指定了其值为0的短语宽度的转换短语的导出语法。

语义：*READTEXT* 内部子程序调用内的转换功能是将一个是某种外部表示的串变换为一个 CHILL 值，而 *WRITETEXT* 内部子程序调用内的转换功能完成反向变换。转换码连同转换限定符一起指明转换的种类和诸如对齐、溢出处理或填充等所要求的操作细节。

这种外部表示是一个串，其长度通常取决于欲被转换的值。该串可能包含表示该 CHILL 值所必需的最小字符个数（对自由格式而言）或可能具有给定的长度（对固定格式而言）。

在固定格式下，根据由转换限定符选定的对齐和填充操作要求，从**实际下标**位置开始自**文本记录**中读出或向**文本记录**内写入其长度为**宽度**大小的那么一个串片，定义如下：

- 在 *READTEXT* 中：如果存在填充字符，则根据对齐要求向左或向右移去所有填充字符。然而，当读字

符或**定长**字符串时，被移去的填充字符的最大个数 N 是**宽度**$-L$，其中 L 分别是*1*和**串长度**。如果 N 小于*0*，则没有字符被移去。剩下的那些字符作为相应的外部表示；

- 在 *WRITETEXT* 中：如果相应外部表示的长度等于或小于**宽度**，则根据对齐要求在那个串片内向左或向右对齐字符。如果存在未用到的串元素，则它们被填入填充字符；否则，该串被截断（如果选择向右对齐，则截断左边的那些字符，否则截断右边的那些字符）。但如果指定了*转换限定符* E，则传送**宽度**那么多个“溢出”指示符（*）。截断功能在相应外部表示上实施，如果存在减号（－）的话，也包括该减号在内。

在自由格式下，满足下列要求：

- 在 *READTEXT* 中：如果存在填充字符，则跳过所有填充字符，除非是读一个字符或字符串，且未指定*转换限定符* P。因此，相应外部表示被当作从**实际下标**处开始的最长的字符串片，且它是由词法上可能属于该字符串片的所有后续字符组成的。满足上述条件的后续字符在下面定义；
- 在 *WRITETEXT* 中：将由转换所传 递的串从**实际下标**开始处插入**文本记录**。

在 *WRITETEXT* 中，把外部表示的串传送到**文本记录**中，并不考虑该串的**实际长度**。在传送操作完成后，**实际下标**自动增长到下一个可用的字符位置，且**实际长度**被置为介于**实际下标**与（老）**实际长度**区间内的最大值。

如果*短语宽度*是由*数字*组成的，则它是**恒定的**。假定*数字*采用十进制表示法；否则它是**可变的**。

如果**宽度**是*0*，则选择自由格式，否则**宽度**是固定格式的长度。

如果**宽度**小到不能包含指定的串时，则根据*转换限定符*采取适当的动作。

在 *READTEXT* 中，所读的外部表示是下面针对*单元变元*的模式而定义的那种外部表示。

在 *WRITETEXT* 中，所写的外部表示是下面针对模式 M 而定义的那种外部表示，其中 M 是*值变元*所传递的值的 M 值类或 M 导出类中的 M。

转换码

*转换码*以单个字母的形式表示。语言定义了下列*转换码*：

B：二进制表示；

O：八进制表示；

H：十六进制表示；

C：转换，用来指出 CHILL 值的缺省外部表示，这取决于欲被转换的值的模式（见下）。

外部表示取决于*转换码*和欲被 转换的值的模式。

转换限定符

转换**限定符**被表示为单个字母。语言定义了下列*转换限定符*：

L：左对齐。如果不存在**转换限定符**，则假定它是右对齐。在自由格式下该限定符不起作用；

E：溢出标记。在 *WRITETEXT* 中，选择了溢出指示。如果未指明该限定符，则执行截断功能。在 *READTEXT* 或自由格式下，该限定符不起作用；

P：填充。跟在该限定符之后的字符指出填充字符。如果不出现 *P*，则填充字符假定为缺省值空格。在 *READTEXT* 中，如果选定自由格式，则当限定符 *P* 之后出现空格或 HT（水平制表符）或由缺省约定填充字符是空格时，就跳过而忽略不计的目的而言，空格与 HT 被视为同一个字符。

外部表示

CHILL 值的外部表示定义如下：

a. 整数

整数值在词法上被表示为以缺省十进制形式出现的一个或多个数字，前面不带引导数字*0*；但如果它是负的，则前面带引导减号（负号）。在 *READTEXT* 中，忽略引导加号（正号）和引导数字*0*。可使用下列*转换码*：*B*、*O*、*C* 和 *H*。*转换码 C* 选择十进制表示。可能属于该外部表示的数字仅仅是由该转换码选定的那些数字。

b. 布尔

布尔值在词法上被表示为简单名字串，它们分别是 *TRUE* 和 *FALSE*（是大写（如 *TRUE*）还是小写（如 *true*）依赖于实现针对**专用**简单名字串选定的表示）。只能使用转换码 *C*。

c. 字符

字符值在词法上被表示为长度为 *1* 的串。只能使用转换码 *C*。

d. 集合

集合模式值在词法上被表示为简单名字串，这些简单名字串是该集合模式定义的集合字面值。只能使用转换码 *C*。

e. 范围

范围值与其**根**模式的值具有相同的表示。然而，仅仅是由某个范围模式定义的值的表示才属于与该范围模式有关的外部表示的集合。

f. 字符串

字符串值在词法上被表示为长度为 *L* 的字符串。在 *WRITETEXT* 中，*L* 是**实际长度**；但在 *READTEXT* 中，如果该字符串是一个**定长**串，则 *L* 是其**串长度**；否则，该字符串是一个**变长**串，且 *L* 是其**串长度**。但下列情况除外，即**文本记录**（的串片）在**实际下标**处没有那么多个字符可用，此时 *L* 是可用的字符个数。仅可使用转换码 *C*。

g. 位串

位串值在词法上被表示为二进制数字串。确定其内数字个数的规则与确定字符串值内字符个数的规则相同。仅能使用转换码 C。

动态性质：短语宽度具有**宽度**，它就是由那些数字所传递的值（对**恒定的**短语宽度而言）或是入出表中某个值所传递的值（对**可变的**短语宽度而言）。

动态条件：当出现下列情况之一时，引发异常 *TEXTFAIL*：

- 在 *READTEXT* 中，（在移去或跳过填充字符以后，见上）**文本记录**从**实际下标**处开始不包含如下的串片，即它能被解释为当前单元变元的模式所定义的值之一的外部表示（这包括当**实际下标**等于**实际长度**时试图从**文本记录**中读出某个非空的外部表示）；或
- 在 *WRITETEXT* 中，不能从**实际下标**处开始向**文本记录**写入一个串片，而该串片是当前值变元的外部表示；或
- 在 *READTEXT* 中，遇到一个转换码，且入出表中当前元素不是一个单元，或既使它是一个单元，但其模式具有**只读性质**；或
- 遇到一个**可变的**短语宽度，且入出表中相应的入出表元素的类不是整数类或其值小于 *0*。

例子：

26.21 CL6

7.5.6 编辑

语法：

〈编辑短语〉 ::= (1)

〈编辑码〉[〈短语宽度〉] (1.1)

〈编辑码〉 ::= (2)

X|〈|〉|*T* (2.1)

导出语法：如果编辑码不是 *T*，则其内不出现短语宽度的编辑短语是其内指定了为1的短语宽度的编辑短语的导出语法，否则是其内出现指定为0的短语宽度的编辑短语的导出语法。

语义：本语言定义了下列编辑功能：

X：空格 ： 插入或跳过**宽度**那么多个空格字符；

>：向右跳 ： 将**实际下标**向右移**宽度**那么多个位置；

<：向左跳 ： 将**实际下标**向左移**宽度**那么多个位置；

T：制表 ： 将**实际下标**移到**宽度**位置。

在 *WRITETEXT* 中，如果**实际下标**被移到某个大于**实际长度**的位置，则由 N 个空格字符组成的串被附加到**文本记录**之上，其中 N 是**实际下标**与（老）**实际长度**之差。**实际长度**被置为介于**实际下标**与（老）**实际长度**区间内的最大值。

动态条件：出现下列情况之一时引发异常 *TEXTFAIL*：

- **实际下标**被移到某个小于0或大于**文本长度**的位置；或
- 在 *READTEXT* 中，**实际下标**被移到某个大于**实际长度**的位置；或
- 在 *READTEXT* 中，虽然指定了编辑码 *X*，但是在**文本记录**中**实际下标**处不出现由**宽度**那么多个空格或 HT（水平制表符）组成的串。

例子：

26.22 X *(6.1)*

7.5.7 入/出控制

语法：

〈入出短语〉::= *(1)*

〈入出码〉 *(1.1)*

〈入出码〉::= *(2)*

/|－|＋|?|!|= *(2.1)*

语义：入/出控制功能（除%=外）完成某种入/出操作。它们允许对**文本记录**的传送实施精确的控制。在 *READTEXT* 中，所有入/出控制功能具有同样的作用，以便从文件中读出下一个记录，在 *WRITETEXT* 中，传送**文本记录**以及托架控制信息的恰当的表示。在连接文本单元的那个时刻，托架的初始位置是满足下列条件的位置，即第一个**文本记录**的第一个字符被打印在第一个未被占用的行的开始处（此时不考虑任何附着到**文本记录**之上的定位信息）。

利用下面定义在当前列、行和页（x，y，z）之上的抽象运算来描述托架的位置，并认为列是从左边界算起自0开始向上顺序编号，而行从顶端边界算起自0顺序向上编号：

nl（w）：托架向下移动 w 行，且位于该行的行首（新位置是（0，$(y+w)$ *MOD* p，$z+(y+w)/p$），其中 p 是每页的行数）。

np（w）：托架向下移动 w 页，且位于该页的页首（新位置是（0，0，$z+w$））。

本语言提供了下列入/出控制功能：

/： 下一个记录 ： 该记录被打印在下一行（nl（1），打印记录，nl（0））；

＋： 下一页 ： 该记录被打印在下一页的顶部（np（1），打印记录，nl（0））；

－： 当前行 ： 该记录被打印在当前行（打印记录，nl（0））；

?： 提示 ： 该记录被打印在下一行上。托架被留在该行的行尾（nl（1），打印记录）；

!： 仅打印 ： 不执行任何托架控制（打印记录）；

=： 结束页 ： 如果存在下一个记录，则将下一个记录的定位信息定义为下一页的顶端（这废除了在打印该记录之前执行的定位功能），但不执行任何入/出操作。

如下完成入/出传送：

- 在 *READTEXT* 中，其语义如同执行了一个 *READRECORD*（*A*，*I*，*R*），其中 *A* 是指定的文本单元的**访问**子单元，*I* 是下标表达式（如果存在的话），*R* 表示**文本记录**。在读传送操作完成之后，**实际下标**被置为0，且**实际长度**被置为所读出的串值的**串长度**；
- 在 *WRITETEXT* 中，其语义如同执行了一个 *WRITERECORD*（*A*，*I*，*R*），其中 *A* 是指定的文本单元的**访问**子单元，*I* 是下标表达式（如果存在的话），*R* 表示**文本记录**。与之有关的定位信息也一同被传送。如果该**访问**子单元的**记录**模式不是**动态的**，则**文本记录**的尾部被添入若干个空格字符，且其**实际长度**在写传送操作发生之前被置为**文本长度**。在写操作完成之后，**实际下标**和**实际长度**都被置为0。

例子：

26.21 (*1.1*)

7.5.8 文本单元属性的存取

语法：

〈取文本内部子程序调用〉::= (*1*)

GETTEXTRECORD（〈文本单元〉） (*1.1*)

|*GETTEXTINDEX*（〈文本单元〉） (*1.2*)

|*GETTEXTACCESS*（〈文本单元〉） (*1.3*)

|*EOLN*（〈文本单元〉） (*1.4*)

〈置文本内部子程序调用〉::= (*2*)

SETTEXTRECORD（〈文本单元〉，〈字符串单元〉） (*2.1*)

|*SETTEXTINDEX*（〈文本单元〉，〈整数表达式〉） (*2.2*)

|*SETTEXTACCESS*（〈文本单元〉，〈访问单元〉） (*2.3*)

语义： *GETTEXTRECORD* 返送文本单元的**文本记录引用**。

GETTEXTINDEX 返送文本单元的**实际下标**。

GETTEXTACCESS 返送文本单元的**访问引用**。

如果**文本记录**中没有更多的字符可用，（即如果**实际下标**等于**实际长度**），则 *EOLN* 传送 *TRUE*。

SETTEXTRECORD 将一个对由字符串单元传递的单元的引用值存入文本单元的**文本记录引用**中。

SETTEXTINDEX 与 *WRITETEXT* 内下列编辑短语具有相同的语义，其内编辑码是 *T*，短语宽度与指定的整数表达式传递相同的值，且该编辑短语是在由文本单元表示的**文本记录**之上实施的。

SETTEXTACCESS 将一个对由访问单元传递的单元的引用值存入文本单元的**访问引用**中。

静态性质： *GETTEXTRECORD* 内部子程序调用的类是 *M* 引用类，其中 *M* 是文本单元的**文本记录**模式。

GETTEXTINDEX 内部子程序调用的类是 *INT* 导出类。

GETTEXTACCESS 内部子程序调用的类是 *M* 引用类，其中 *M* 是文本单元的**访问**模式。

EOLN 内部子程序调用的类是 *BOOL* 导出类。

GETTEXTRECORD 和 *GETTEXTACCESS* 内部子程序与其内指定的文本单元具有相同的**区域性**。

静态条件： *SETTEXTRECORD* 的字符串单元变元的模式必须与文本单元的**文本记录**模式是**读相容的**。

SETTEXTACCESS 的访问单元变元的模式必须与文本单元的**访问**模式是**读相容的**。

SETTEXTRECORD 和 *SETTEXTACCESS* 内的单元变元必须与文本单元具有相同的**区域性**。

动态条件： 如果 *SETTEXTINDEX* 的整数表达式变元传递一个小于 *0* 或大于文本单元的**文本长度**的值，则引发异常 *TEXTFAIL*。

例子：

26.23 GETTEXTINDEX (output) (*1.2*)

8 异常处理

8.1 概述

异常可以是语言定义的异常，此时它具有语言定义的异常名字，也可以是用户定义的异常，还可以是实现定义的异常。动态违反某种动态语义条件将引发一个语言定义的异常。执行引发动作可以引发任何异常。

当一个异常被引发后，可以得到处理，即执行某个合适的处理程序中指定的动作语句表。

异常处理是这样定义的：在任何语句处静态可知它可能引发哪些异常（即静态可知哪些异常不会出现），也可以知道对哪些异常能找到合适的处理程序以及哪些异常能被传递给过程的调用点。如果出现了一个异常，但找不到相应的处理程序，则该程序是错误的。

当某个动作语句或说明语句处出现一个异常时，该语句执行到何处是不确定的，除非在语句的有关节中已明确指明该异常的被引发处。

8.2 处理程序

语法：

〈处理程序〉 ::= (1)

ON { 〈异常处理选择项〉}* [**ELSE** 〈动作语句表〉] **END** (1.1)

〈异常处理选择项〉 ::= (2)

(〈异常表〉): 〈动作语句表〉 (2.1)

语义：如果根据 8.3 节的定义，某个处理程序是关于异常 E 的合适的处理程序，则当出现异常 E 时进入该处理程序。如果该处理程序的某个异常处理选择项中的异常表内提到了 E，则进入该异常处理选择项内的动作语句表；否则，该处理程序内指定了 **ELSE**，并进入跟在 **ELSE** 后面的动作语句表。

当到达所选的动作语句表的尾部时，该处理程序以及附着该处理程序的语法成份被终止。

静态条件：一个处理程序内的所有异常表中出现的所有异常名字必须是不同的。

动态条件：如果进入处理程序内指定的某个动作语句表，但相应存储需求得不到满足，则引发异常 *SPACE-FAIL*。

例子：

```
10.47   ON
          (ALLOCATEFAIL): CAUSE overflow;
        END                                          (1.1)
```

8.3 处理程序的识别

当在动作或模块 A 或数据语句或区域 D 处出现异常 E 时，该异常可以由某个合适的处理程序处理，即执行该处理程序中某个指定的动作语句表，或将该异常传递到某个过程的调用点。如果不能采取前面两种处理，则该程序出错。

对任何动作或模块 A 或数据语句或区域 D 而言，可以静态确定下列事实，即对 A 或 D 处引发的一个给定的异常 E 能否找到合适的处理程序，或能否将 E 传递到过程的调用点。

对于 A 或 D 而言，如下确定关于具有异常名字 E 的异常的合适的处理程序：

1. 如果 A 或 D 上附加了一个处理程序或在 A 或 D 中包含了一个处理程序，且该处理程序内的某个异常表中提到了 E 或该处理程序指明了 **ELSE**，则该处理程序是关于 E 的合适的处理程序；
2. 否则，若 A 或 D 被某个带括号动作、模块或区域直接包围，则关于 E 的合适的处理程序（如果存在的话）就是该带括号动作、模块或区域的关于 E 的合适的处理程序；
3. 否则，如果 A 或 D 位于某个过程定义的可达区内部，那么：
   - 如果该过程定义上附加了一个处理程序，且其内某个异常表中提到了 E，或其内指明了 **ELSE**，则该处理程序是关于 E 的合适的处理程序；
   - 否则，如果该过程定义的异常表中提到了 E，则 E 在该过程的调用点被引发（即 E 被传递到过程的调用点）；
   - 否则，不存在关于 E 的合适的处理程序。
4. 否则，如果 A 或 D 位于某个进程定义的可达区内部，那么：
   - 如果该进程定义上附加了一个处理程序，且其内某个异常表中提到了 E，或其内指明了 **ELSE**，则该处理程序就是关于 E 的合适的处理程序；
   - 否则，不存在关于 E 的合适的处理程序。然而，在这种情况下，某个实现定义的处理程序可能是关于 E 的合适的处理程序（参见 13.4 节）。
5. 否则，如果 A 是一个处理程序 H 中指定的动作语句表之一内的一个动作，则关于 E 的合适的处理程序就是动作 A' 或数据语句或区域 D' 的关于 E 的合适的处理程序，而 H 正是附加到 A' 或 D' 之上的或是被 A' 或 D' 包含的。此时认为就好像 H 不存在一样。

如果引发了一个异常，且当控制转向关于该异常的合适的处理程序时即意味着退出分程序的话，则当退出该分程序时要释放该分程序所占用的局部存储。

9 时钟监控

9.1 概述

假定 CHILL 程序（系统）的外部存在着时钟概念。CHILL 虽未指 明时钟的确切性质，但提供了使得程序能与外部时钟观点交互的机制。

9.2 可超时的进程

为了识别程序执行期间可能出现的时钟中断的准确时刻，即为了识别时钟监控可能干预进程的正常执行的时刻，引入了**可超时的**进程的概念。

当一个进程在执行某些动作时到达某个良好定义的程序点时，该进程成为**可超时的**，CHILL 定义了一个进程在某些特殊动作的执行期间成为**可超时的**，而实现可以定义一个进程在更多的动作执行期间成为**可超时的**。

9.3 计时动作

语法：

〈计时动作〉::= (1)

〈相对计时动作〉 (1.1)

| 〈绝对计时动作〉 (1.2)

| 〈周期计时动作〉 (1.3)

语义：计时动作指出执行它的进程的时钟监控。可以启动时钟监控，时钟监控可能期满，也可能不复存在。由于周期计时动作和计时动作的嵌套，同一个进程可能与几个时种监控有关。

当一个进程是**可超时的**，且至少有一个与之有关的时钟监控已期满时，出现一个时钟中断。时钟中断的出现意味着该第一个期满的时钟监控不复存在。此外，时钟中断的出现还导致与在被监控进程内的那个时钟监控有关的控制的转移。如果被监控进程此刻是被延迟的，则它成为重新激活的。

当控制离开启动时钟监控的计时动作时，时钟监控也就不复存在。

9.3.1 相对计时动作

语法：

〈相对计时动作〉::= (1)

AFTER 〈时延原值〉[**DELAY**] **IN**

〈动作语句表〉〈超时处理程序〉 **END** (1.1)

〈超时处理程序〉::= (2)

TIMEOUT 〈动作语句表〉 (2.1)

语义：对时延原值求值，并启动一个时钟监控，然后进入动作语句表。

如果未指明 **DELAY**，则在进入动作语句表之前启动该时钟监控；否则，当执行进程执行到由动作语句表内那个动作语句指明的那个时刻成为**可超时的**时启动该时钟监控。

如果指明了 **DELAY**，那么，若该时钟监控已被启动，且执行进程不再是**可超时的**，则该时钟监控

不复存在。

如果自启动时刻起已过了指定的那么一段时间，且该时钟监控依然存在，则该时钟监控期满。

与该时钟监控有关的控制转移是转向超时处理程序内的动作语句表。

静态条件：如果指定了 **DELAY**，则动作语句表只能含一个动作语句，且该动作语句自己能导致执行进程成为**可超时的**。

动态条件：如果因某种实现定义的原因不能成功地启动该时钟监控，则引发异常 *TIMERFAIL*。

9.3.2 绝对计时动作

语法：

〈绝对计时动作〉::= (1)

AT 〈绝对时钟原值〉 **IN**

〈动作语句表〉〈超时处理程序〉 **END** (1.1)

语义：对绝对时钟原值求值，并启动一个时钟监控，然后进入动作语句表。

如果在所指定的时刻或在那个时刻之后该时钟监控依然存在，则该时钟监控期满。

与该时钟监控有关的控制转移是转向超时处理程序内的动作语句表。

动态条件：如果因某种实现定义的原因不能成功地启动该时钟监控，则引发异常 *TIMERFAIL*。

9.3.3 周期计时动作

语法：

〈周期计时动作〉::= (1)

CYCLE 〈时延原值〉 **IN**

〈动作语句表〉 **END** (1.1)

语义：周期计时动作用来确保执行进程不带累积误差地在精确的间隔内进入动作语句表（这意味着动作语句表的平均执行时间应小于指定的时延值）。对时延原值求值，并启动一个相对时钟监控，然后进入动作语句表。

如果自启动时刻起已过了指定的那么一段时间，而该时钟监控仍然存在，则该时钟监控期满。在本次期满的同时，又启动一个新的具有相同时延值的相对时钟监控。

与该时钟监控有关的控制转移是转向动作语句表的开始处。

注意：周期计时动作仅能由将控制转出该动作的控制转移终止。

动态性质：如果控制到达动作语句表的尾部，且仅在此时，执行进程成为**可超时的**。

动态条件：如果因某种实现定义的原因不能成功地启动某个相对时钟监控，则引发异常 *TIMERFAIL*。

9.4 时钟内部子程序

语法：

〈时钟值内部子程序调用〉::= (1)

〈时延内部子程序调用〉 (1.1)

| 〈绝对时钟内部子程序调用〉 (1.2)

语义：每种实现可以根据自己的需要对时钟值的精度和范围提出相当不同的要求和容量。下面定义的内部子程序用来以一种可移植的方式适应这些差异。

9.4.1 时延内部子程序

语法：

〈时延内部子程序调用〉 ::= (1)
MILLISECS (〈整数表达式〉) (1.1)
|*SECS* (〈整数表达式〉) (1.2)
|*MINUTES* (〈整数表达式〉) (1.3)
|*HOURS* (〈整数表达式〉) (1.4)
|*DAYS* (〈整数表达式〉) (1.5)

语义：时延内部子程序调用传递一个带实现定义的、可能具有不同精度的时延值（如 *MILLTSECS* (1000) 与 *SECS* (1) 可能传递不同的时延值）。在所选的精度下这个值是与指明的一段时间最接近的值。

静态性质：时延内部子程序调用的类是 *DURATION* 导出类。

动态条件：如果实现不能传递由指明的一段时间所表示的时延值，则引发异常 *RANGEFAIL*。

9.4.2 绝对时钟内部子程序

语法：

〈绝对时钟内部子程序调用〉 ::= (1)
ABSTIME ([[[[[[〈年表达式〉,] 〈月表达式〉,]
〈日表达式〉,] 〈时表达式〉,] 〈分表达式〉,] 〈秒表达式〉]) (1.1)

〈年表达式〉 ::= (2)
〈整数表达式〉 (2.1)

〈月表达式〉 ::= (3)
〈整数表达式〉 (3.1)

〈日表达式〉 ::= (4)
〈整数表达式〉 (4.1)

〈时表达式〉 ::= (5)
〈整数表达式〉 (5.1)

〈分表达式〉 ::= (6)
〈整数表达式〉 (6.1)

〈秒表达式〉 ::= (7)
〈整数表达式〉 (7.1)

语义：*ABSTIME* 内部子程序调用传递一个绝对时钟值，它表示在参数表内以格里历形式给出的某一时刻。当缺少较前面的诸参数时，所指明的时刻是与所出现的较后面的诸参数匹配的下一个时刻（例如，*ABSTIME* (*15*, *12*, *00*, *00*) 表示本月或下月 15 号正午）。

如果未指明任何参数，则传递表示当前时刻的那个绝对时钟值。

静态性质：绝对时钟内部子程序调用的类是 *TIME* 导出类。

动态条件：如果实现不能传递所指出的那个时刻的绝对时钟值，则引发异常 *RANGEFAIL*。

9.4.3 计时内部子程序调用

语法：

〈计时简单内部子程序调用〉::= (1)

WAIT () (1.1)

| *EXPIRED* () (1.2)

| *INTTIME* (〈绝对时钟原值〉,[[[[〈年单元〉〈月单元〉,]〈日单元〉,]〈时单元〉,]〈分单元〉,]〈秒单元〉) (1.3)

〈年单元〉::= (2)

〈整数单元〉 (2.1)

〈月单元〉::= (3)

〈整数单元〉 (3.1)

〈日单元〉::= (4)

〈整数单元〉 (4.1)

〈时单元〉::= (5)

〈整数单元〉 (5.1)

〈分单元〉::= (6)

〈整数单元〉 (6.1)

〈秒单元〉::= (7)

〈整数单元〉 (7.1)

语义：*WAIT* 无条件地使执行进程成为**可超时的**，其执行只能由某个时钟中断终止。

如果与执行进程有关的时钟监控之一已期满，则 *EXPIRED* 使得执行进程成为**可超时的**，否则 *EXPIRED* 不起任何作用。

INTTIME 将由绝对时钟原值指出的、以格里历形式给出的特定时刻的整数表示赋值给指定的那些单元。

静态条件：所有指定的整数单元都必须是**可引用的**，且它们的模式不能具有**只读性质**。

动态性质：*WAIT* 使得执行进程成为**可超时的**。

如果存在一个与执行进程有关的已期满的时钟监控，则 *EXPIRED* 使得执行进程成为**可超时的**。

10 程序结构

10.1 概述

条件动作、情况动作、循环动作、延迟情况动作、begin-end 分程序、模块、区域、说明模块、说明区域、上下文、接收情况动作、过程定义和进程定义确定了程序的结构，即它们确定了其内创建的名字的作用域和单元的生存期。

- 术语分程序将被用来表示：
  - 包含循环计数器和当型控制在内的循环动作中的动作语句表；
  - 条件动作内则子句中的动作语句表；
  - 情况动作内情况选择项中的动作语句表；
  - 延迟情况动作内延迟选择项中的动作语句表；
  - begin-end 分程序；
  - 去掉结果说明和形参表中所有形参的参数说明后的过程定义；
  - 去掉形参表中所有形参的参数说明后的进程定义；
  - 包含 **IN** 后面的定义性出现表中所有定义性出现在内的缓冲区接收选择项或信号接收选择项中的动作语句表；
  - 条件动作、情况动作、接收情况动作或处理程序内 **ELSE** 后面的动作语句表；
  - 处理程序中的异常处理选择项；
  - 相对计时动作、绝对计时动作、周期计时动作或超时处理程序中的动作语句表。
- 术语模体将被用来表示：
  - 去掉上下文表和定义性出现（假如有的话）后的模块或区域；
  - 去掉上下文表（假如有的话）后的说明模块或说明区域；
  - 上下文。
- 术语组块既可表示分程序，又可表示模体。
- 术语可达区或组块的可达区用来表示该组块中不为其内部组块所包围的那部分程序正文(参见10.2节)。

组块影响在其可达区内创建的每个名字的作用域。名字由定义性出现创建：

- 在某个说明、模式定义或同义词定义的定义性出现表中出现的一个定义性出现，或在某个信号定义中出现的定义性出现分别在该说明、模式定义、同义词定义或信号定义所在的可达区内创建了一个名字；
- 在集合模式中出现的一个定义性出现在直接包含该集合模式的可达区内创建了一个名字；
- 出现于形参表内定义性出现表中的一个定义性出现在相应过程定义或进程定义的可达区内创建了一个名字；
- 后面跟有一个动作、区域、过程定义或进程定义的冒号前面的定义性出现分别在该动作、区域、过程定义或进程定义所在的可达区内创建了一个名字；
- 由开域部分引入的（虚拟）定义性出现或作为循环计数器而引入的定义性出现在相应循环动作的分程序的可达区内创建了一个名字；
- 在缓冲区接收选择项或信号接收选择项内定义性出现表中出现的一个定义性出现分别在相应缓冲区接收选择项或信号接收选择项所在的分程序的可达区内创建了一个名字；
- 语言预定义的或实现定义的名字的（虚拟）定义性出现在假想的最外层进程（参见10.8节）的可达区内创建了一个名字。

使用一个名字的地方称为该名字的应用性出现。名字约束规则将该名字的定义性出现与该名字的每个

应用性出现联系起来（参见12.2.2节）。

名字具有一定的作用域，即该程序中可以看到该名字的定义或说明，因此，可以自由使用该名字的那部分程序正文。一个名字在其作用域内称为**可见的**。单元和过程具有一定的生存期，即在该程序中它们是存在的那部分程序正文。分程序确定了其内创建的名字的可见性和单元的生存期。模体仅确定可见性；在模体的可达区内创建的单元的生存期如同它们是在第一个包围该模体的分程序的可达区内创建的生存期一样。模体允许限制名字的可见性，例如，在某个模块的可达区内创建的单元名字在其内层或外层模块中不是自动**可见的**，尽管该单元的生存期可能包括该内层或外层模块。

10.2 可达区和嵌套

语法：

〈begin-end 分程序体〉::= (1)
〈数据语句表〉〈动作语句表〉 (1.1)

〈过程体〉::= (2)
〈数据语句表〉〈动作语句表〉 (2.1)

〈进程体〉::= (3)
〈数据语句表〉〈动作语句表〉 (3.1)

〈模块体〉::= (4)
{〈数据语句〉|〈可见性语句〉|〈区域〉|
〈说明区域〉}* 〈动作语句表〉} (4.1)

〈区域体〉::= (5)
{〈数据语句〉|〈可见性语句〉}* (5.1)

〈说明模块体〉::= (6)
{〈准数据语句〉|〈可见性语句〉|
〈说明模体〉|〈说明区域〉}* (6.1)

〈说明区域体〉::= (7)
{〈准数据语句〉|〈可见性语句〉}* (7.1)

〈上下文体〉::= (8)
{〈准数据语句〉|〈可见性语句〉|
〈说明模块〉|〈说明区域〉}* (8.1)

〈动作语句表〉::= (9)
{〈动作语句〉}* (9.1)

〈数据语句表〉::= (10)
{〈数据语句〉}* (10.1)

〈数据语句〉::= (11)
〈说明语句〉 (11.1)
|〈定义语句〉 (11.2)

〈定义语句〉::= (12)
〈同义模式定义语句〉 (12.1)
|〈新模式定义语句〉 (12.2)
|〈同义词定义语句〉 (12.3)
|〈过程定义语句〉 (12.4)

| 〈进程定义语句〉 (12.5)
| 〈信号定义语句〉 (12.6)
| 〈空〉; (12.7)

语义：当进入一个分程序的可达区时，就完成了进入该分程序时所创建的单元的生存期初始化。接着，按程序正文规定的先后顺序依次执行该分程序可达区中的单元的可达区初始化、单元等同说明中可能有的动态求值、所含区域内的单元可达区初始化以及动作语句。

当进入一个模体的可达区时，按程序正文规定的先后次序，顺序执行在该模体可达区内的单元可达区初始化、单元等同说明中可能有的动态求值、所含区域内的单元可达区初始化。如果该模体是一个模块，则还要执行其内指定的动作语句表。

当一个数据语句、动作、模块或区域是由本身完成或是由附着在其上的处理程序终止时，该数据语句、动作、模块或区域就被终止。

当一个可达区初始化、单元等同说明、动作、模块、区域、过程或进程被终止时，可根据语句或其终止的类型，如下恢复程序的执行：

- 如果该语句是因所附着的处理程序执行完毕而终止的，则程序从其后续语句处恢复执行；
- 否则，如果它是一个隐含控制转移的动作，则按该语句的语义规定恢复程序的执行(参见6.5、6.6、6.8和6.9节)；
- 否则，如果它是一个过程，则控制返回过程调 用点(参见10.4节)；
- 否则，如果它是一个进程，则该进程的执行结束，控制可能交给另一进程恢复运行。如果该进程是假想的最外层进程，则整个程序执行结束(参见11.1节)；
- 否则，控制转向其后续语句。

静态性质：任何一个可达区都被直接包围在零个或多个组块中，定义如下：

- 如果该可达区是某个循环动作、begin-end 分程序、过程定义或进程定义的可达区，则它被直接包围在其可达区内分别含有该循环动作、begin-end 分程序、过程定义或进程定义的组块中，且仅被直接包围在该组块中；
- 如果该可达区是某个计时动作或超时处理程序中的动作语句表，或它是某个条件动作、情况动作或延迟情况动作中的动作语句表之一，则它被直接包围在其可达区内包含该计时动作、超时处理程序、条件动作、情况动作或延迟情况动作的组块中，且仅被直接包围在该组块中；
- 如果该可达区是某个缓冲区接收选择项或信号接收选择项内的动作语句表，或者是某个接收缓冲区情况动作或接收信号情况动作内跟在 **ELSE** 后面的动作语句表，则它被直接包围在其可达区内含有该接收缓冲区情况动作或接收信号情况动作的组块中，且仅被直接包围在该组块中；
- 如果该可达区是某个异常处理选择项中的动作语句表，或者是某个处理程序内跟在 **ELSE** 后面的动作语句表，且该处理程序未附着到某个组块之上，则它被直接包围在其可达区内含有附着该处理程序的语句的组块中，并且仅被直接包围在该组块中；
- 如果该可达区是某个异常处理选择项中的动作语句表，或者是某个处理程序内跟在 **ELSE** 后面的动作语句表，且该处理程序附着到某个细块之上，则它被直接包围在附着该处理程序的组块中，且仅被直接包围在该组块中；
- 如果该可达区是一个模块、区域、说明模块或说明区域，则它被直接包围在该可达区所在的组块中，并且，如果该模块、区域、说明模块或说明区域之前有上下文的话，则它也被直接包围在该上下文中。这是仅有的一个可达区具有多于一个直接包围组块的情形；
- 如果该可达区是一个上下文，则它被直接包围在紧接在它前面的上下文中。如果没有这样的上下文，则它没有直接包围组块。

一个可达区具有零个、一个或多个直接包围可达区，它们就是其直接包围组块的可达区。每个语句具有一个唯一的直接包围组块，即该语句所在的组块。当且仅当一个可达区是某个组块（或可达区）的直接包围可达区时，称该可达区直接包围了该组块（或可达区）。

当且仅当一个组块是某个语句（或可达区）的直接包围组块，或该组块包围了该语句（或可达区）的直接包围可达区时，称该语句（或可达区）被该组块包围。

下列情况发生时称为进入一个可达区：

- 如果该可达区是模块可达区：该模块作为一个动作执行（例如，当转向动作将控制转移到在该模块内部定义的**标号名字**时，不能说进入该模块）；
- 如果该可达区是 begin-end 分程序可达区：该 begin-end 分程序作为一个动作执行；
- 如果该可达区是区域可达区：遇到该区域（例如，当它的某个**临界**过程被调用时，不能说进入该区域）；
- 如果该可达区是过程可达区：通过一个过程调用动作进入该过程；
- 如果该可达区是进程可达区：通过一个启动表达式的求值激活该进程；
- 如果该可达区是循环可达区：在控制部分中的表达式或单元被计算或访问之后，该循环动作作为动作执行；
- 如果该可达区是缓冲区接收选择项或信号接收选择项可达区：收到一个缓冲区值或信号后执行该选择项；
- 如果该可达区是异常处理可达区：在引发了该异常处理选择项的异常表中指定的异常之一后执行该异常处理选择项；
- 如果该可达区是其它分程序可达区：进入该动作语句表。

当且仅当程序控制从某个动作语句表之外转到它的第一个动作（如果存在的话）时，称进入该动作语句表。

如果一个可达区是某个说明模块、说明区域或上下文可达区，则称该可达区是**准**可达区，否则称之为**实**可达区。

如果一个定义性出现满足下列条件之一，则称之为**准**定义性出现，否则称之为**实**定义性出现：

- 它被一个上下文包围，但不被某个模块或区域包围；或
- 它被某个简单说明模块或简单说明区域包围；或
- 它不被上面提到的某种组块之一包围，但它被一个模块说明或区域说明包围，且它出现于一个准说明、准过程定义语句或准进程定义语句中，还要求它不是一个**集合元素**名字的定义性出现。

10.3 begin-end 分程序

语法：

〈begin-end 分程序〉 ::= (*1*)

BEGIN 〈begin-end 分程序体〉 **END** (*1.1*)

语义：begin-end 分程序是一个动作，可能包含某些局部说明和定义。它既确定了局部创建的名字的可见性，又确定了局部创建的单元的生存期（参见 10.9 和 12.2 节）。

动态条件：若 begin-end 分程序需要局部存储，而相应存储需求又得不到满足，则引发异常 *SPACEFAIL*。

例子：参见 *15.73～15.90*

10.4 过程定义

语法：

〈过程定义语句〉 ::= (*1*)

〈定义性出现〉:〈过程定义〉

[〈处理程序〉] [〈简单名字串〉]; (*1.1*)

〈过程定义〉 ::= (2)

PROC ([〈形参表〉]) [〈结果说明〉]

[**EXCEPTIONS** (〈异常表〉)] 〈过程属性表〉;

〈过程体〉 **END** (2.1)

〈形参表〉 ::= (3)

〈形参〉 {, 〈形参〉}* (3.1)

〈形参〉 ::= (4)

〈定义性出现表〉〈参数说明〉 (4.1)

〈过程属性表〉 ::= (5)

[〈通用性〉] [**RECURSIVE**] (5.1)

〈通用性〉 ::= (6)

GENERAL (6.1)

| **SIMPLE** (6.2)

| **INLINE** (6.3)

导出语法：其定义性出现表由多于一个定义性出现组成的形参是从几个用逗号隔开的形参出现中导出的，每个定义性出现都对应一个形参，且这些形参都具有相同的参数说明。例如，*i*, *j INT* **LOC** 是从 *i INT* **LOC**, *j INT* **LOC** 导出的。

语义：过程定义语句定义了（可能是）参数化的一串动作，在程序内不同地方可以调用这串动作。在出现下列情况之一时，过程被终止，且控制返回调用点：执行了一个返回动作；到达过程体的尾部；附着在该过程定义之上的处理程序已终止（失败出口）。可以如下指明过程的不同复杂程度：

a. **简单**过程（**SIMPLE**）是不能被动态加工的过程。它们并不作为值来看待，即不能把它们存入一个过程单元，也不能作为参数将它们传递给某个过程调用，还不能把它们作 为结果从某个过程调用返回；

b. **通用**过程（**GENERAL**）没有对**简单**过程的那些限制，并可以将它们作为过程值看待；

c. **直接插入**过程（**INLINE**）与**简单**过程具有同样的限制，同时，这些过程是不可**递归的**。它们与普通的过程具有同样的语义，但编译程序在调用点上直接插入生成的目标代码，而不是生成实际调用该过程的代码。

只有**简单**过程和**通用**过程可以被指定为（相互）**递归的**。如果未指明任何过程属性，则采用实现定义的缺省属性。

过程可以返送一个值，也可以返送一个单元（这由结果说明中 **LOC** 属性指出）

过程定义前面的定义性出现定义了该过程的名字。

参数传递：

基本上有两种参数传递机制，即“值传递”（**IN**、**OUT** 及 **INOUT**）和“单元传递”（**LOC**）。

值传递

在值传递参数传递机制中，值被作为参数传送给过程，并存入具有指定的参数模式的一个局部单元中。其作用就好像是在过程调用的开始处遇到下列单元说明语句（这些语句是为相应形参的定义性出现而设立的）：

DCL 〈定义性出现〉〈模式〉: = 〈实参〉;

但是只有在实参已求出值之后才能进入该过程。可用任选的关键字 **IN** 显式指明采用值传递机制。

如果指定了属性 **INOUT**，则上述被传递的值从一个实参单元中取出，且恰好在过程返回之前将相应形参的当前值回存到该实参单元中。

属性 OUT 的作用与属性 INOUT 相同，只不过在过程入口处不把实参单元的初值复制到上述形参单元中，故这种形参具有一个**未定义的**初值。如果该过程在调用点引发某个异常，则不必执行上述回存操作。

单元传递

在单元传递参数传递机制中，将一个（可能具有动态模式的）单元作为参数传递给过程体。只有**可引用的**单元能以这种方式传递给过程。其作用就好像在过程的入口处遇到下列单元等同说明语句（这些语句是为相应形参的定义性出现而设立的）：

DCL 〈定义性出现〉〈模式〉 **LOC** [**DYNAMIC**]：＝〈实参〉；

但是只有在实参求值之后才能进入过程。

如果指定的相应实参是一个值，且它不是一个单元，则隐式创建一个包含该值的单元，并在调用点传递该隐式单元。所创建的隐式单元的生存期是该过程调用。如果指定的值具有动态类，则所创建的单元的模式是动态的。

结果传送：

过程既可返送一个值，又可返送一个单元。在第一种情况下，在过程体内任何结果动作中指定一个值；而在后一种情况下，在过程体内任何结果动作中指定一个单元（参见6.8节）。如果在结果说明中未指明属性 **NONREF**，则这种单元必须是**可引用的**。过程所返送的值或单元由过程在返回前最后执行的结果动作所确定。如果一个带结果说明的过程没有执行结果动作就返回，则该过程返送一个**未定义的**值或**未定义的**单元。在这种情况下，该过程调用不能用作单元过程调用（参见4.2.11 节），也不能用作值过程调用（参见 5.2.12 节），而只能用作调用动作（参见 6.7 节）。

静态性质：过程定义语句中的定义性出现定义了一个**过程**名字。

每个**过程**名字附有一个过程定义，它就是定义该**过程**名字的过程定义语句内的过程定义。每个**过程**名字附有下列性质，它们是由其过程定义定义的：

- 它具有**参数说明**表，这些**参数说明**是由形参表中出现的诸参数说明定义的，每个参数由一个模式及可能有的参数属性组成；
- 它可能具有**结果说明**。**结果说明**由一个模式及任选的结果属性组成；
- 它具有一个可能为空的异常名字表，这些名字就是异常表中提到的那些名字；
- 它具有**通用性**。如果指定了通用性，则根据该通用性是 **GENERAL**、**SIMPLE**，还是 **INLINE** 来确定**通用性**分别是**通用的**、**简单的**或**直接插入的**；否则，由实现定义的缺省通用性指定为**通用的**或**简单的**。如果该**过程名字**是在某个区域内部定义的，则其**通用性**是**简单的**；
- 它具有**递归性**。如果指定了 **RECURSIVE**，则**递归性**是**递归的**，否则由实现定义的缺省值指定为**递归的**或**非递归的**。然而，如果该**过程**名字的**通用性**是**直接插入的**，或该**过程**名字是**临界的**（参见 11.2.1 节），则其**递归性**是**非递归的**。

如果一个**过程**名字的**通用性**是**通用的**，则称它是一个**通用过程**名字。**通用过程**名字附有如下构造的过程模式：

PROC （ [〈参数表〉]） [〈结果说明〉]
[**EXCEPTIONS** （ 〈异常表〉)] [**RECURSIVE**]

其中〈结果说明〉和〈异常表〉（如果有的话）与其过程定义中指定的相应项相同，而〈参数表〉是形参表内出现的那些〈参数说明〉组成的序列，〈参数说明〉之间用逗号分开。

当且仅当某个形参中的参数说明不含 **LOC** 属性时，由该形参的定义性出现表中每个定义性出现定义的名字是一个**单元名**字。如果它确实包括 **LOC** 属性，则上述名字是一个**单元等同**名字。任何这种**单元**名字或**单元等同**名字都是**可引用的**。

静态条件：如果一个**过程**是**区域内的**（参见11.2.2节），则其过程定义不能指明 **GENERAL**。

如果一个**过程**名字是**临界的**（参见11.2.1节），则其过程定义既不能指定 **GERERAL**，也不能指定 **RECURSIVE**。

任何过程定义都不能同时指定 **INLINE** 和 **RECURSIVE**。

如果某个过程定义语句尾部指定了简单名字串，则该简单名字串必须与过程定义前面的定义性出现的名字串相同。

仅当参数说明或结果说明内指明了 **LOC** 时，其内的模式才能具有**非值性质**。

异常表中提到的所有名字必须是不同的。

例子：

```
1.4     add:
        PROC (i,j INT) RETURNS (INT) EXCEPTIONS (OVERFLOW);
            RESULT i+j;
        END add;                                              (1.1)
```

10.5 进程定义

语法：

〈进程定义语句〉 ::= (1)

〈定义性出现〉:〈进程定义〉

[〈处理程序〉] [〈简单名字串〉]; (1.1)

〈进程定义〉 ::= (2)

PROCESS ([〈形参表〉]); 〈进程体〉 **END**; (2.1)

语义：进程定义语句定义了一个可能是参数化了的一串动作，在程序内不同程序点可以启动这种动作序列与其它进程并发执行（参见第十一章）。

静态性质：进程定义语句中的定义性出现定义了一个**进程**名字。

进程名字附有如下由其进程定义定义的性质：

- 它具有一个**参数说明**表，其中每个**参数说明**由形参表内出现的相应参数说明定义。每个参数由一个模式及可能有的参数属性组成。

静态条件：如果某个进程定义语句内指定了简单名字串的话，则它必须与进程定义前面的定义性出现具有相同的名字串。

进程定义语句不能被某个区域包围，也不能被非假想的最外层进程定义的分程序包围（参见10.8节）。

形参表中出现的那些参数属性不能是 **INOUT** 或 **OUT**。

仅当形参表内某个形参的参数说明中指定了 **LOC** 时，该参数说明内的模式才能具有**非值性质**。

例子：

```
14.13   PROCESS ();
          wait:
          PROC (x INT);
              /*some wait action*/
          END wait;
          DO FOR EVER;
              wait(10 /* seconds */ );
              CONTINUE operator_is_ready;
          OD;
        END                                                   (2.1)
```

10.6 模块

语法：

〈模块〉 ::= (1)

[〈上下文表〉] [〈定义性出现〉:]

MODULE [**BODY**] 〈模块体〉 **END**

[〈处理程序〉] [〈简单名字串〉]; (1.1)

| 〈远程模体〉 (1.2)

语义：模块是一个可能含有局部说明和定义的动作语句。模块是限制名字串的可见性的手段，但它不影响局部创建的单元的生存期。

关于模块的可见性细则在12.2节中给出。

静态性质：模块中的定义性出现定义了一个**模块**名字，并定义了一个**标号**名字。该名字附有该模块（将其视为一个模体，即去掉可能有的上下文表和定义性出现之后的程序正文）。

当且仅当一个模块指定了任选的上下文表时，该模块是分块开发的。

当且仅当一个模块指定了任选的 **BODY** 时，该模块是一个**实现模块**。

静态条件：如果一个模块的尾部指明了简单名字串，则它必须与前面指定的定义性出现具有相同的名字串。

模块语法定义中的远程模体必须引用某个模块。

例子：

```
7.48    MODULE
            SEIZE convert;
            DCL n INT INIT := 1979;
            DCL rn CHARS (20) INIT := (20)' ';
            GRANT n,rn;
            convert();
            ASSERT rn = "MDCCCCLXXVIIII"//(6)' ';
        END                                              (1.1)
```

10.7 区域

语法：

〈区域〉 ::= (1)

[〈上下文表〉] [〈定义性出现〉:]

REGION [**BODY**] 〈区域体〉 **END**

[〈处理程序〉] [〈简单名字串〉]; (1.1)

| 〈远程模体〉 (1.2)

语义：区域为进程的并发执行提供了互斥访问在该区域内部说明的局部数据对象的手段（参见第11章）。区域以和模块同样的方式确定局部创建的名字的可见性。

静态性质：区域中的定义性出现定义了一个**区域**名字。它附有该区域（将其视为一个模体，即去掉可能有的

上下文表和定义性出现之后的程序正文)。

当且仅当一个区域指明了任选的上下文表时，该区域是分块开发的。

当且仅当一个区域指明了任选的 **BODY** 时，该区域称为一个**实现区域**。

静态条件：如果一个区域的尾部指定了简单名字串，则该简单名字串必须与该区域开头处指定的定义性出现的名字串相同。

区域不能被一个非假想的最外层进程定义的分程序包围。

区域语法定义中的远程模体必须引用某个区域。

例子：见*13.1～13.28*

10.8 程序

语法：

〈程序〉::= (1)

{〈模块〉|〈说明模块〉|〈区域〉|〈说明区域〉}$^{+}$ (1.1)

语义：程序由一系列被假想的最外层进程定义包围的模块或区域组成。

就生存期而言，CHILL 预定义名字(参见附录 C.2)和实现定义的内部子程序名字以及实现定义的整数模式名字被认为是在该假想的最外层进程定义的可达区内定义的。有关它们的可见性见12.2节。

10.9 存储分配和生存期

单元或过程在其程序内存在的那一段时间称为其**生存期**。

单元是通过说明或执行 *GETSTACK* 或 *ALLOCATE* 内部子程序调用创建的。

在一个分程序的可达区内说明的单元的生存期是控制留在该分程序内或留在由该分程序所调用的过程内的那段时间，除非该单元的说明带有属性 **STATIC**。在一个模体的可达区内说明的单元的生存期就好像在该模体的最内层包围分程序内说明这些单元的生存期一样。其说明带有属性 **STATIC** 的单元的生存期与在假想的最外层进程定义的可达区中说明该单元的生存期相同。这就意味着，对带属性 **STATIC** 的单元说明而言，只分配一次存储，即在启动假想的最外层进程时分配。如果这种说明出现在过程定义或进程定义内部，则对于所有的调用或激活而言，只存在一个单元。

当 *GETSTACK* 内部子程序调用的直接包围分程序终止时，通过执行 *GETSTACK* 内部子程序调用所创建的单元的生存期就结束。

通过执行 *ALLOCATE* 内部子程序调用创建的单元的生存期是始于 *ALLOCATE* 调用时刻，终于该单元再也不能被任何 *CHILL* 程序访问的时刻的那段时间。后者总是指下列情况，即在引用该单元的**已分配**引用值上执行了 *TERMINATE* 内部子程序调用。

由单元等同说明所创建的**单元等同**名字的生存期是该单元等同说明的直接包围分程序。

过程的生存期是相应过程定义的直接包围分程序。

静态性质：当且仅当一个单元是满足下列条件之一的静态模式单元时，该单元称为**静态的**：

- 单元名字，其说明带属性 **STATIC**，或其定义不被非假想的最外层进程定义的分程序包围；
- 串元素或串片，其内的串单元是**静态的**，并且，要么指定的左元素和右元素是**常数**，要么指定的开始元素和片大小是**常数**；
- 数组元素，其内的数组单元是**静态的**，且指定的表达式**是常数**；
- 数组片，其内的数组单元是**静态的**，并且，要么指定的下元素和上元素是**常数**，要么指定的首元素和片大小是**常数**；

- 结构域，其内的结构单元是**静态的**；
- 单元转换，其内出现的单元是**静态的**。

10.10 分块程序设计设施

按分块程序设计方式开发出来的一个完整的 CHILL 程序的基本划分单位（程序块）是模块和区域，这种程序块的正文由远程设施指明（参见 10.10.1 节）。CHILL 定义完整的 CHILL 程序语法和语义时，将其内所有远程程序块的出现（虚拟）置换为它们所引用的程序正文。

10.10.1 远程程序块

语法：

〈远程模体〉 ::= (1)
　　[〈简单名字串〉:] **REMOTE** 〈程序块标志符〉; (1.1)

〈远程说明〉 ::= (2)
　　[〈简单名字串〉:] **SPEC REMOTE** 〈程序块标志符〉; (2.1)

〈远程上下文〉 ::= (3)
　　CONTEXT REMOTE 〈程序块标志符〉
　　[〈上下文体〉] **FOR** (3.1)

〈上下文模块〉 ::= (4)
　　CONTEXT MODULE REMOTE 〈程序块标志符〉; (4.1)

〈程序块标志符〉 ::= (5)
　　〈字符串字面值〉 (5.1)
　| 〈正文引用名字 (5.2)
　| 〈空〉 (5.3)

导出语法：表示法：

　　CONTEXT MODULE REMOTE 〈程序块标志符〉

是下列表示法的导出语法：

　　CONTEXT REMOTE 〈程序块标志符〉 **FOR**
　　MODULE SEIZE ALL; END;

注意：这种设施是冗余的，但可用于一致性检验。

语义：远程模体、远程说明、远程上下文和上下文模块是把程序的源正文表达为一组（具有内在联系的）文件的手段。

程序块标志符以实现定义的方式引用某段 CHILL 源程序正文之描述，定义如下：

- 如果该程序块标志符是空，则这段源程序正文从由其所在的程序的结构确定的位置处查找；
- 如果该程序块标志符是字符串字面值，则该字符串字面值用于查找相应的源程序正文；
- 如果该程序块标志符是正文引用名字，则以实现定义的方式来解释该正文引用名字，通过此名可查找相应的源程序正文。

带远程模体或远程说明的程序与用其程序块标志符所引用的 CHILL 正文程序段置换相应的远程模体或远程说明后得到的程序等价。

带远程上下文的程序与按下列方法构造出来的程序等价，即将原程序中的每个远程上下文用由其程序块标志符所引用的 CHILL 正文程序块替换，且当该程序块标志符所引用的上下文表内上下文体最后一次出现后立即将这些远程上下文中的上下文体虚拟地插入上述 CHILL 正文程序块。

如果由某个远程程序块标志的程序块还不能作为一段 CHILL 程序正文得到，则该远程程序块中的程序块标志符被认为引用一个虚拟引入的、与上述程序块等价的 CHILL 正文程序块。

尽管远程程序块的语义是以置换方式定义的，但 CHILL 并不要求真正实施任何正文置换。

静态条件：在1. 远程模体，2. 远程说明，3. 远程上下文，4. 上下文模块中的程序块标志符必须引用某段源程序的描述，而该段源程序必须是1. 非远程模体的模块或区域，2. 非远程说明的说明模块或说明区域，3.，4. 非远程上下文的上下文表的终结产生式。

当由在一个远程模体中的程序块标志符所引用的源程序正文以一个定义性出现打头时，该远程模体必须以一个简单名字串打头，且该简单名字串就是上述定义性出现的名字串。

当由在一个远程说明中的程序块标志符所引用的源程序正文以一个简单名字串打头时，该远程说明也必须以该简单名字串打头。

例子：

25.9 *stack:* **REMOTE** *"example 27 or 28";* *(1.1)*
25.9 *"example 27 or 28"* *(5.1)*

10.10.2 说明模块、说明区域及上下文

语法：

〈说明模块〉 ::= *(1)*
〈简单说明模块〉 *(1.1)*
| 〈模块说明〉 *(1.2)*
| 〈远程说明〉 *(1.3)*

〈简单说明模块〉 ::= *(2)*
[〈上下文表〉] [〈简单名字串〉 :] **SPEC MODULE**
〈说明模块体〉 **END** [〈简单名字串〉]; *(2.1)*

〈模块说明〉 ::= *(3)*
[〈上下文表〉] 〈简单名字串〉: **MODULE SPEC**
〈说明模块体〉 **END** [〈简单名字串〉]; *(3.1)*

〈说明区域〉 ::= *(4)*
〈简单说明区域〉 *(4.1)*
| 〈区域说明〉 *(4.2)*
| 〈远程说明〉 *(4.3)*

〈简单说明区域〉 ::= *(5)*
[〈上下文表〉] [〈简单名字串〉:] **SPEC REGION**
〈说明区域体〉 **END** [〈简单名字串〉]; *(5.1)*

〈区域说明〉 ::= *(6)*
[〈上下文表〉] 〈简单名字串〉: **REGION SPEC**
〈说明区域体〉 **END** [〈简单名字串〉]; *(6.1)*

〈上下文表〉 ::= *(7)*
〈上下文〉 { 〈上下文〉}* *(7.1)*
| 〈远程上下文〉 *(7.2)*

〈上下文〉 ::= (8)

CONTEXT 〈上下文体〉 **FOR** (8.1)

语义：简单说明模块、简单说明区域和上下文用于指出名字的静态性质。它们是冗余的，但可用于分块程序设计。

说明区域中的简单名字串不是名字。它们不**被约束**，也不存在相应的可见性规则。

在某个**实**可达区内的说明模块或说明区域指明一个或多个被分块编译的模块或区域的性质，并且可认为这些模块或区域被该**实**可达区包围。这种模块或区域的源程序正文由远程模体的出现指明。上下文表指出相应的包围可达区（注意：分块研制出来的模体的前面总含有一个上下文表）。

对满足下列条件的每个名字串 *OP*!*NS* 而言，认为要在相应的**实现模块**或**实现区域**的可达区内引入一个（虚拟的）带相同**老**名字串 *OP*!*NS* 和**新**名字串 *NP*!*NS* 的移出语句：

a. 该名字串在某个模块说明或区域说明的可达区内是**可见的**；

b. 该名字串在上述可达区内**被链接**到一个**准**定义性出现之上；

c. 该名字串作为 *NP*!*NS* 被移出到某个**实**可达区中。

静态条件：在说明模块或说明区域中，仅当 **SPEC** 前面出现任选的简单名字串时，**END** 后面才能出现任选的简单名字串。当它们都存在时，这两个简单名字串必须具有相同的名字串。

没有直接包围组块的上下文不能包含可见性语句。

一个包含说明模块或说明区域的**实**可达区也必须至少包含一个远程模体，反之亦然。

如果一个**实**可达区包含一个是**实现模块**的模块或是**实现区域**的区域，则它也必须包含一个满足下列条件的模块说明或区域说明，即该模块说明或区域说明前面的简单名字串与该**实现模块**或**实现区域**前面的简单名字串具有相同的名字串。称该模块说明或区域说明具有一个**对应的实现模块**或**对应的实现区域**。

在说明模块或说明区域语法定义中的远程说明必须引用一个说明模块或说明区域。

例子：

```
23.2    letter_count:
        SPEC MODULE
            SEIZE max;
            count: PROC (input ROW CHARS (max) IN,
              output ARRAY ('A':'Z') INT OUT) END;
            GRANT count;
        END letter_count;                                   (1.1)

24.1    CONTEXT
            count: PROC (ROW CHARS (max) IN,
              ARRAY ('A':'Z') INT OUT) END;
        FOR                                                 (8.1)
```

10.10.3 准语句

语法：

〈准数据语句〉 ::= (1)

〈准说明语句〉 (1.1)

| 〈准定义语句〉 (1.2)

〈准说明语句〉 ::= (2)

DCL 〈准说明〉 {, 〈准说明〉}*; (2.1)

〈准说明〉::= (3)
〈准单元说明〉 (3.1)
| 〈准单元等同说明〉 (3.2)

〈准单元说明〉::= (4)
〈定义性出现表〉〈模式〉[STATIC] (4.1)

〈准单元等同说明〉::= (5)
〈定义性出现表〉〈模式〉
LOC [NONREF] [DYNAMIC] (5.1)

〈准定义语句〉::= (6)
〈同义模式定义语句〉 (6.1)
| 〈新模式定义语句〉 (6.2)
| 〈同义词定义语句〉 (6.3)
| 〈准同义词定义语句〉 (6.4)
| 〈准过程定义语句〉 (6.5)
| 〈准进程定义语句〉 (6.6)
| 〈准信号定义语句〉 (6.7)
| 〈空〉; (6.8)

〈准同义词定义语句〉::= (7)
SYN 〈准同义词定义〉{,〈准同义词定义〉}* (7.1)

〈准同义词定义〉::= (8)
〈定义性出现表〉{〈模式〉= [〈常数值〉]
| [〈模式〉] = 〈字面值表达式〉} (8.1)

〈准过程定义语句〉::= (9)
〈定义性出现〉:PROC ([〈准形参表〉])
[〈结果说明〉] [EXCEPTIONS (〈异常表〉)]
〈过程属性表〉END [〈简单名字串〉]; (9.1)

〈准形参表〉::= (10)
〈准形参〉{,〈准形参〉}* (10.1)

〈准形参〉::= (11)
〈简单名字串〉{,〈简单名字串〉}*〈参数说明〉 (11.1)

〈准进程定义语句〉::= (12)
〈定义性出现〉: PROCESS ([〈准形参表〉])
END [〈简单名字串〉]; (12.1)

〈准信号定义语句〉::= (13)
SIGNAL 〈准信号定义〉{,〈准信号定义〉}*; (13.1)

〈准信号定义〉::= (14)
〈定义性出现〉[= (〈模式〉{,〈模式〉}*)] [TO] (14.1)

语义：准语句在说明模块、说明区域及上下文中用来说明名字的静态性质。这些说明是冗余的，但准语句能用于分块程序设计。

某个不能保证**准常数同义词**名字与相应**实常数同义词名字**等值的 CHILL 实现可以禁止准同义词定义语句中出现常数值。

注意：在 CHILL 中，对**标号**名字不存在**准**定义性出现。

静态性质：准语句是相应语句的限制形式，但它们具有相同的静态性质。

如果某个准单元等同说明中未指明 **NONREF**，则在该准单元等同说明内由一个定义性出现所定义的名字是**可引用的**。

静态条件：准语句是相应语句的限制形式，并遵守相应语句的静态条件。

准同义词定义语句只能被一个简单说明模块、简单说明区域或上下文直接包围。在准定义语句语法定义中的同义词定义语句只能被一个模块说明或区域说明直接包围。

10.10.4 准定义性出现与定义性出现间的匹配

如果两个定义性出现属于同一语义范畴，且满足下列条件，则称它们**匹配**：

- 如果它们都是**同义词**名字，则它们必须具有相同的**区域性**和值，它们的类的**根**模式必须是**类同的**，它们都具有 M 值类、M 导出类、M 引用类、**null** 类或 **all** 类。此外，如果其中之一是准**同义词**名字，且它是一个**字面值**，则另一个**同义词**名字也必须是一个**字面值**；
- 如果它们都是**集合元素**名字，则它们所附有的**集合**模式必须是**类同的**；
- 如果它们都是**新模式**名字或**同义模式**名字，则它们的模式必须是**类同的**；
- 如果它们都是**单元**名字或**单元等同**名字，则它们必须具有相同的**区域性**，它们都是或都不是**可引用的**，它们都是或都不是**静态的**，它们的模式必须是**类同的**；
- 如果它们都是**过程名字**，则它们必须具有相同的**区域性**和**通用性**，都是或都不是**临界的**，它们都必须满足如同过程模式应满足的**类同性**条件，形参表与准形参表中按位置对应的简单名字串必须相同；
- 如果它们都是**进程**名字，则它们的进程定义中的参数必须满足如同**过程**名字的参数应满足的匹配及类同性条件；
- 如果它们都是**信号名字**，则它们必须都指明或都不指明 **TO**，它们的模式表必须具有相同个数的模式，且对应的模式必须是**类同的**。

如果两个结构模式在可达区 R 中都是**新颖性约束的**，则它们在 R 中必须具有相同的**可见**域名字集合。

使用下列规则：

- 如果在某个可达区内一个名字串**被约束**到一个**准**定义性出现之上，且该可达区不是某个说明模块、说明区域或上下文的可达区，则该名字串也必须**被约束**到一个不是**准**定义性出现的定义性出现之上，且：
  - 令一个名字串**被约束**到某个**准**定义性出现 QD 之上，且它也**被约束**到可达区 R 中的一个**实**定义性出现 RD 之上，那么：
    1. QD 和 RD 必须**匹配**（如上定义），且
    2. RD 和 QD 必须都被包围在某个被 R 包围的组块内或都不被包围在 R 的组块中，或如果 R 是一个为**实现模块**或**实现区域**的模块或区域的可达区，则 QD 必须被包围在该**实现模块**或**实现区域**的**对应的**模块说明或区域说明的组块中，且 RD 必须被 R 的组块包围。
  - 如果在某个**实**可达区 R 中的一个名字串**被约束**到被包围在 R 的组块内的某个**准**定义性出现之上（即它被一个说明模体包围），则它也必须**被约束**到由某个模块或区域的组块所包围的某个**实**定义性出现之上，且该模块或区域由 R 直接包围的某个远程模体指明（非严格地说，如果接口移出，则实现也必须移出）。如果该**准**定义性出现被包围在某个模块说明或区域说明的组块内，则该**实**定义性出现必须被**对应的**模体的组块包围。
  - 如果在某个**实**可达区 R 中的一个名字串**被约束**到被某个模块或区域的组块包围的某个**实**定义性出现之上，且该模块或区域是由直接被 R 包围的某个远程模体指明的，则该名字串也必

须**被约束**到一个被 R 的组块包围的一个**准**定义性出现之上（即被一个说明模体包围。非严格地说，如果实现移出，则接口也必须移出。

— 对在被**实**可达区 R 直接包围的一个说明模块或说明区域的可达区 Q 中的每个名字串而言，如果这种名字串**被约束**到不被 Q 所包围的某个定义性出现之上，那么，在由被 R 直接包围的某个远程模体指明的模块或区域的可达区中必须存在一个相同的名字串**被约束**到同一个定义性出现之上（非严格地说，如果接口移入，则实现也必须移入）。

- 如果两个名字串都**被约束**到某个可达内的同一个**实**定义性出现或**准**定义性出现之上，则这两个名字串都必须**被约束**到同一个**准**定义性出现或**实**定义性出现之上，或者它们都不进一步**被约束**；
- 一个**实新颖性**不能**被新颖性约束**到任何可达区内的两个**准新颖性**之上。

设在一个可达区 R 中，**准新颖性** QN 与**实新颖性** RN 是被此**新颖性约束的**，那么 RN 和 QN 都必须被 R 的被包围组块包围或两者都不被 R 的组块包围，或如果 R 是为**实现模块**或**实现区域**的一个模块或区域的可达区，则 RN 必须被 R 的组块包围，而 QN 必须被**对应的**模块说明或区域说明的组块包围。

11 并发执行

11.1 进程及其定义

进程是一串语句的顺序执行。它可以与其它进程并发执行。一个进程的行为由相应的进程定义描述（参见10.5节），进程定义描述了某个进程的局部数据对象以及将要顺序执行的动作语句串。

进程可通过对启动表达式（参见5.2.14节）的求值来创建。一经创建，进程就成为活跃的（即处于执行中），并且被认为与其它进程并发执行。所创建的进程是由相应进程定义的**进程**名字所指出的定义的一次激活。由同一进程定义可以创建出若干个进程，这些进程具有相同的进程定义，其个数没有限制，并且它们也是并发执行的。每个进程由一个实例值唯一地标识，该实例值是作为相应的启动表达式的结果或作为THIS运算的计算结果而产生的。进程的创建引起了其局部说明的单元的创建，但具有属性STATIC的那些局部说明的单元除外（参见10.9节），同时也引起了局部定义的那些值和过程的创建。称这些局部说明或定义的单元、值和过程与它们所属的被创建的进程具有同一个激活 促想的最外层进程（参见10.8节）被认为是控制该程序执行的系统执行了某个启动表达式而创建的，该假想的最外层进程就是所执行的整个CHILL程序。在某个进程创建的那一时刻，如果它具有形参的话，则其每个形参表示相应启动表达式中对应的实参所传递的值或单元。

出现下列几种情况之一时，进程被终止：执行了一个停止动作；到达进程体的末尾；该进程的进程定义尾部指定的处理程序已终止（失败终止）。如果假想的最外层进程执行了一个停止动作或失败终止，当且仅当该程序内所有的其它进程都被终止时，它才完全终止。

在CHILL程序设计级上，进程总处于下列两种状态之一，即它要么是活跃的（即处于执行中），要么是被延迟的（即等待某个条件得到满足）。从活跃态到被延迟态的状态变迁称为进程的延迟，从被延迟态到活跃态的状态变迁称为进程的重新激活。

11.2 互斥和区域

11.2.1 概述

区域（参见10.7节）提供了多个进程对在该区域内部说明的单元进行互斥访问的手段。为确保（非假想的最外层进程的）进程只能通过调用在该区域内部定义并由该区域移出的过程来访问在该区域内部说明的单元，特制定了相应的静态上下文条件（参见11.2.2节）。

如果一个**过程**名字是在某个区域内部被定义的，并且由该区域移出，则称该**过程**名字表示一个**临界**过程（此时，它是一个**临界过程**名字）。

当且仅当程序控制要么不在一个区域的某个**临界**过程内部，要么因执行可达区初始化而处于该区域本身内部时，则称该区域是自由的。

如果出现下列情况之一，则区域被封锁（以防多个进程同时进入该区域）：

- 进入该区域（注意：由于区域不可能被某个分程序包围，所以不可能试图同时进入该区域）；
- 该区域的某个**临界**过程被调用；
- 在该区域之上被延迟的某个进程被重新激活。

在某个区域被封锁后，如果出现下列情况之一，则释放该区域，它重新成为自由的：

- 离开该区域；
- 被调用的**临界**过程返回；
- 被调用的**临界**过程执行了一个使得调用它的进程成为被延迟态的动作（参见11.3节）。在动态嵌套**临界**过程调用的情况下，仅释放最后被封锁的区域；

- 执行**临界**过程调用的进程终止。在动态嵌套**临界**过程调用的情况下，释放被该进程封锁的所有区域。

在一个区域被封锁期间，如果有某个进程试图调用其**临界**过程之一，或者在该区域之上被延迟的某个进程被重新激活，则该进程被挂起（在该区域之上），直到该区域被释放（注意：在 CHILL 意义上该进程仍然是活跃的）。

当一个区域被释放，且此刻存在多个进程因在该区域被封锁期间试图调用该区域的某个**临界**过程，或者因在该区域的某个**临界**过程内被重新激活而被挂起（在该区域之上）时，只允许按实现定义的调度算法从中选择一个进程封锁该区域。

11.2.2 区域性

为了使编译程序能静态检查只能通过调用某个区域的**临界**过程之一或为了执行可达区初始化而进入某个区域这两种方式来访问在该区域内部说明的单元，必须满足下列静态上下文条件：

- 必须满足有关章节（赋值动作、过程调用、发送动作、结果动作等）提到的**区域性**要求；
- **区域内的**过程不能是**通用的**（参见 10.4 节）；
- **临界**过程既不是**通用的**，也不是**递归的**（参见 10.4 节）。

单元和过程调用具有**区域性**，即要么是**区域内的**，要么是**区域外的**。值具有**区域性**，即要么是**区域内的**，要么是**区域外的**，要么是 **nil**。这些性质定义如下：

1. **单元**

当且仅当满足下列条件之一时，单元是**区域内的**：

- 它是一访问名字，该访问名字可为如下之一：
  - 在某个区域或某个说明区域内部说明的单元名字，且它不是在该区域或说明区域的某个**临界**过程的某个形参内定义的；
  - 单元等同名字，而其说明内指定的单元是**区域内的**，或者它是在某个**区域内的**过程的某个形参中定义的单元等同名字；
  - 单元枚举名字，而相应的循环动作中指定的数组单元或串单元是**区域内的**；
  - 单元开域名字，而相应的带开域控制的循环动作中指定的结构单元是**区域内的**。
- 它是间接引用的受限引用，而其内的受限引用原值是**区域内的**；
- 它是间接引用的自由引用，而其内的自由引用原值是**区域内的**；
- 它是间接引用行，而其内的行原值是**区域内的**；
- 它是数组元素或数组片，而其内的数组单元是**区域内的**；
- 它是串元素或串片，而其内的串单元是**区域内的**；
- 它是结构域，而其内的结构单元是**区域内的**；
- 它是单元过程调用，而其内指定了一个过程名字且该过程名字是**区域内的**；
- 它是单元内部子程序调用，而 CHILL 定义或实现指定它是**区域内的**；
- 它是单元转换，而其内的静态模式单元是**区域内的**。

非**区域内的**单元是**区域外的**。

2. **值**

值具有依赖于其类的**区域性**。如果它具有 M 导出类、**all** 类或 **null** 类，则它具有**区域性 nil**；否则，它具有 M 值类或 M 引用类，同时它也具有依赖于模式 M 的**区域性**，定义如下：

如果该值具有 M 值类，且 M 没有**引用性质**，则其**区域性**是 **nil**；否则，该值是运算数-6（并且 M 具有**引用性质**）或是条件表达式：

如果它是原值，则：

- 如果它是单元内容，且该单元内容是单元，则其**区域性**就是该单元的**区域性**；
- 如果它是值名字，则
  - 如果它是同义词名字，则其**区域性**是其定义中指定的常数值的**区域性**；

— 如果它是值开域名字，则其**区域性**是相应的带开域控制的循环动作中指定的结构原值的**区域性**；

— 如果它是值接收名字，则其**区域性**是**区域外的**。

- 如果它是多元组，那么，若其内出现的值之一具有非 **nil** 的**区域性**，则该多元组的**区域性**就是该值的**区域性**（选择其内哪个值都没关系，参见 5.2.5 节静态条件），否则该多元组的**区域性**是 **nil**；
- 如果它是值数组元素或值数组片，则其**区域性**是其内数组原值的**区域性**；
- 如果它是值结构域，则其**区域性**是其内的结构原值的**区域性**；
- 如果它是表达式转换，则其**区域性**是其内的表达式的**区域性**；
- 如果它是值过程调用，则其**区域性**是其内过程调用的**区域性**；
- 如果它是值内部子程序调用，则其**区域性**由 CHILL 定义或实现指定它为**区域内的**或**区域外的**。

如果它是被引用单元，则其**区域性**是其内指定的单元的**区域性**；

如果它是接收表达式，则其**区域性**是**区域外的**。

如果它是条件表达式，那么，如果其内出现的子表达式之一具有非 **nil** 的**区域性**，则该条件表达式的**区域性**就是该子表达式的**区域性**（选择哪个子表达式都没关系，参见 5.3.2 节静态条件）；否则，该条件表达式的**区域性**是 **nil**。

3. **过程名字**

当且仅当一个过程名字是在某个区域或说明区域内部定义的，且它不是一个**临界过程**名字（即它未被该区域移出）时，该过程名字是**区域内的**，否则是**区域外的**。

4. **过程调用**

如果一个过程调用包含一个是**区域内的**过程名字，则该过程调用是**区域内的**，否则是**区域外的**。

当且仅当下列条件之一成立时，一个值相对某个非终结符（仅指单元过程调用和过程名字）而言是**区域安全的**：

- 该非终结符是**区域外的**，而该值不是**区域内的**；
- 该非终结符是**区域内的**，而该值不是**区域外的**；
- 该非终结符具有**区域性 nil**。

11.3 进程的延迟

一个活跃的进程执行下列动作之一或对下列接收表达式之一求值后可能成为被延迟的：

- 延迟动作（参见 6.16 节）；
- 延迟情况动作（参见 6.17 节）；
- 接收表达式（参见 5.3.9 节）；
- 接收信号情况动作（参见 6.19.2 节）；
- 接收缓冲区情况动作（参见 6.19.3 节）；
- 发送缓冲区动作（参见 6.18.3 节）。

当一个进程成为被延迟的，而其控制处在某个**临界**过程内部时，将释放该**临界**过程所属的区域。该进程的动态上下文将被保存起来，直到该进程被重新激活。到那时该进程试图再次封锁该区域，这将可能导致该进程被挂起（在该区域之上）。

11.4 进程的重新激活

如果某个被延迟的进程被设置了时钟监控，且出现了一个时钟中断（参见第九章），则它可能成为被重新激活的。如果另一个进程执行了下列动作之一或对接收表达式求值，则被延迟的进程也可能被重新激

活：

- 继续动作（参见 6.15 节）；
- 发送信号动作（参见 6.18.2 节）；
- 发送缓冲区动作（参见 6.18.3 节）；
- 接收表达式（参见 5.3.9 节）；
- 接收缓冲区情况动作（参见 6.19.3 节）。

当一个进程已封锁了某个区域期间重新激活了另一个进程时，它仍然是活跃的，即它在那一时刻并不释放该区域。

11.5 信号定义语句

语法：

〈信号定义语句〉::= (1)

SIGNAL〈信号定义〉{,〈信号定义〉}*; (1.1)

〈信号定义〉::= (2)

〈定义性出现〉[=(〈模式〉{,〈模式〉}*)][**TO**〈进程名字〉] (2.1)

语义：信号定义定义了进程之间要传送的值的组合和分解功能。如果发送了一个信号，则传送指定的值表。如果没有一个进程因执行某个接收信号情况动作而等待该信号的话，则该值表中的那些值被保留下来，直到某个进程接收这些值为止。

静态性质：信号定义中的定义性出现定义了一个**信号**名字。

信号名字具有下列性质：

- 它附有一个任选的模式表，即由相应信号定义中提到的那些模式组成的表；
- 它附有一个任选的**进程**名字，即 **TO** 后面的进程名字。

静态条件：在信号定义中出现的任何模式都不能具有**非值性质**。

例子：

15.27 **SIGNAL** *initiate = (INSTANCE),*
terminate; *(1.1)*

12 总的语义性质

12.1 模式规则

12.1.1 模式和类的性质

12.1.1.1 只读性质

非严格描述

如果某个模式是一个**只读**模式或含有**只读**模式的成分或子成分，则它具有**只读性质**。

定义

当且仅当一个模式是下列模式之一时，该模式具有**只读性质**：

- **只读**模式；
- 数组模式，其内的**元素**模式具有**只读性质**；
- 结构模式，其内至少有一个具有**只读性质**的**域**模式，且具有该**域**模式的域不是具有**参数化**结构模式的隐式**只读**模式的**标志**域。

12.1.1.2 可参数化的模式

非严格描述

如果一个模式能被参数化，则它是**可参数化的**。

定义

当且仅当一个模式是下列模式之一时，它是**可参数化的**：

- 串模式；
- 数组模式；
- **可参数化的变体**结构模式。

12.1.1.3 引用性质

非严格描述

如果某个模式是一个引用模式或含有引用模式的成分或子成分，则它具有**引用性质**。

定义

当且仅当一个模式是下列模式之一时，它具有**引用性质**：

- 引用模式；
- 数组模式，其内的**元素**模式具有**引用性质**；
- 结构模式，其内至少有一个具有**引用性质**的**域**模式。

12.1.1.4 带标志参数化性质

非严格描述

如果某个模式是一个**带标志参数化**结构模式或含有**带标志参数化**结构模式的成分或子成分，则它具有**带标志参数化性质**。

定义

当且仅当一个模式是下列模式之一时，它具有**带标志参数化性质**：

- **带标志参数化**的结构模式；
- 数组模式，其**元素**模式具有**带标志参数化性质**；
- 结构模式，其内至少有一个具有**带标志参数化性质**的**域**模式。

12.1.1.5 非值性质

非严格描述

如果对一个模式而言不存在具有这种模式的表达式或原值的表示，则该模式具有**非值性质**。

定义

当且仅当一个模式是下列模式之一时，它具有**非值性质**：

- 事件模式、缓冲区模式、访问模式、结合模式或文本模式；
- 数组模式，其**元素**模式具有**非值性质**；
- 结构模式，其内至少有一个具有**非值性质**的**域**模式。

12.1.1.6 根模式

任何模式 M 具有如下定义的**根**模式：

- 如果 M 不是一个范围模式，则其**根**模式就是 M；
- 如果 M 是一个范围模式，则其**根**模式是 M 的**祖先**模式。

任何 M 值类或 M 导出类具有一个**根**模式，它就是模式 M 的**根**模式。

12.1.1.7 结果类

给定两个**相容的**类（参见 12.1.2.16 节），它们是 **all** 类、M 值类或 M 导出类，其中 M 是一个离散模式、幂集模式或串模式，则其**结果类**通过 M 和 N 的**结果**模式 R 以及 M 的**根**模式 P 的概念来定义。

给定两个**相似的**模式 M 和 N，M 和 N 的**结果**模式 R 定义如下：

- 如果 M 和 N 之一的**根**模式是一个**定长**串模式，而另一个的**根**模式是一个**变长**串模式，则 M 和 N 的**结果**模式是 M、N 中其**根**模式是**变长**串模式的那个模式的**根**模式；
- 否则，M 和 N 的**结果**模式是 **P**。

两个**相容的**类的**结果类**定义如下：

- M 值类和 N 值类的**结果类**是 R 值类；
- M 值类和 N 导出类或 **all** 类的**结果类**是 P 值类；
- M 导出类和 N 导出类的**结果类**是 R 导出类；
- M 导出类和 **all** 类的**结果类**是 P 导出类；
- **all** 类和 **all** 类的**结果类**是 **all** 类。

给定两两**相容的**类表 C_i（i=1，2，…，n），该类表的**结果类**被递归定义为：如果 n〉1，则它是类表 C_j（j=1，2，…，n−1）的**结果类**和类 C_n 的**结果类**；否则它是 C_1 和 C_1 的**结果类**。

12.1.2 模式与类的关系

12.1.2.1 概述

以下各小节定义了模式之间、类之间以及模式与类之间的相容性关系。本文献始终利用这些关系来定

义静态条件。

为了达到上述目的，相容性关系本身通过其它主要在本章内使用的关系来定义。

12.1.2.2 模式等价关系

非严格描述

下列等价关系在严格定义相容性关系中发挥作用：

- 如果两个模式是同一种类的，即它们具有相同的继承性质，则它们是**相似的**；
- 如果两个模式既是**相似的**，又具有相同的**新颖性**，则它们是 V **等价的**（**值等价的**）；
- 如果两个模式既是 V **等价的**，同时又考虑到存储中值的表示或最小存储大小方面的可能有的差异，则它们是**等价的**；
- 如果两个模式既是**等价的**，又具有相同的**只读**说明，则它们是 L **等价的**（**单元等价的**）；
- 如果两个模式是难以区别的，即在不考虑**新颖性**的前提下，能在一个模式的对象上实施的所有运算也都能在另一个模式的对象上实施，则这两个模式是**类同的**；
- 如果两个模式既是**类同的**，又具有相同的**新颖性**说明，则它们是**新颖性约束的**。

定义

在以下各小节中，将以关系集合（或其子集）的形式给出模式间的等价关系。通过求该关系集合的对称、自反和传递闭包获得完全的等价算法。这些关系中提到的模式可以是虚拟引进的或是动态的。在后一种情况下，完整的等价性检验只能在运行时刻实施。其动态检查部分失败将引发异常 RANGEFAIL 和 TAG-FAIL（参见有关章节）。

若要对任一等价关系的两个递归模式检查，则要求对定义它们的递归模式集合的相应的路径中与之有关的模式进行检查。如果没有发现矛盾，则这两个递归模式是等价的（作为推论，如果比较了先前已比较过的两个模式，则检查算法的一条路径就成功地停止）。

12.1.2.3 相似关系

当且仅当下列条件之一成立时，两个模式是**相似的**：

- 它们都是整数模式；
- 它们都是布尔模式；
- 它们都是字符模式；
- 它们都是满足下列条件的集合模式：
  1. 它们定义了相同的**值的个数**；
  2. 对由其中一个模式定义的每个**集合元素**名字而言，都存在由另一个模式所定义的、具有相同名字串和相同表示值的**集合元素**名字；
  3. 它们都是**编号**集合模式或都是**未编号**集合模式；
- 它们都是范围模式，而它们的**祖先**模式是**相似的**；
- 二者之一是范围模式，且其**祖先**模式与另一模式**相似**；
- 它们都是幂集模式，且它们的**成员**模式是**等价的**；
- 它们都是受限引用模式，且它们的**被引用**模式是**等价的**；
- 它们都是自由引用模式；
- 它们都是行模式，且它们的**被引用原始**模式是**等价的**；
- 它们都是过程模式，并且：
  1. 它们具有同样多个**参数说明**，且（按位置）对应的**参数说明**具有 L **等价的**模式并具有相同的参数属性（如果有的话）；
  2. 它们都具有或都没有**结果说明**，如果有**结果说明**，则它们必须具有 L **等价的**模式，且具有相同的属性（如果有的话）；

3. 它们具有相同的**异常**名字表；
4. 它们具有相同的**递归性**；

- 它们都是实例模式；
- 它们都是事件模式，且要么都没有**事件长度**，要么具有相同的**事件长度**；
- 它们都是缓冲区模式，并且：
  1. 它们都没有**缓冲区长度**，或具有相同的**缓冲区长度**；
  2. 它们的**缓冲区元素**模式是 L **等价的**；
- 它们都是结合模式；
- 它们都是访问模式，并且：
  1. 它们都没有**下标**模式或具有**等价的下标**模式；
  2. 至少有一个没有**记录模**式，或它们都具有 L **等价的记录**模式，并且，这两个**记录**模式要么都是**静态记录**模式，要么都是**动态记录**模式；
- 它们都是文本模式，且：
  1. 它们具有相同的**文本长度**；
  2. 它们具有 L **等价的文本记录**模式；
  3. 它们具有 L **等价的访问**模式；
- 它们都是时延模式；
- 它们都是绝对时钟模式；
- 它们都是串模式，并且都是**位**串模式或都是**字符串**模式；
- 它们都是数组模式，并且：
  1. 它们的**下标**模式是 V **等价的**；
  2. 它们的**元素**模式是**等价的**；
  3. 它们的**元素布局**是**等价的**；
  4. 它们具有相同的**元素个数**。如果其中一个模式是动态的或两者都是动态的，则这种检查是动态的。检查失败将引发异常 *RANGEFAIL*；
- 它们都是非**参数化**结构模式的结构模式，并且：
  1. 在严格语法形式下，它们具有同样多个域，且（按位置）对应的域是**等价的**；
  2. 如果它们是**可参数化的变体**结构模式，则它们的类表必须是**相容的**；
- 它们都是**参数化**结构模式，并且：
  1. 它们的**原始变体**结构模式是**相似的**；
  2. 它们的（按位置）对应的值是相同的。如果其中一个模式是动态的或两者都是动态的，则这种检查是动态的。检查失败将引发异常 *TAGFAIL*。

12.1.2.4 V 等价关系

当且仅当两个模式既是**相似的**，又具有相同的**新颖性**时，它们是 V **等价的**。

12.1.2.5 等价关系

当且仅当两个模式既是 V **等价的**，又满足下列条件之一时，它们是**等价的**：

- 如果其中一个模式是范围模式，则另一个模式也必须是范围模式，且两者的**上界**和**下界**分别相等；
- 如果其中一个模式是**定长**串模式，则另一个模式也必须是**定长**串模式，且它们必须具有相同的**串长度**。在其中之一或两者都是动态串模式时，这种检查是动态的。检查失败将引发异常 *RANGEFAIL*；
- 如果其中一个模式是**变长**串模式，则另一个模式也必须是**变长**串模式，且它们必须具有相同的**串**

长度。当其中之一或两者都是动态串模式时，这种检查是动态的。检查失败将引发异常 *RANGE-FAIL*。

12.1.2.6 L 等价关系

当且仅当两个模式既是**等价的**，且如果其中一个模式是**只读**模式，则另一个模式也必须是**只读**模式，此外，还满足下列条件之一时，这两个模式是 L **等价的**：

- 如果它们都是受限引用模式，则它们的**被引用**模式必须是 L **等价的**；
- 如果它们都是行模式，则它们的**被引用原始**模式必须是 L **等价的**；
- 如果它们都是数组模式，则它们的**元素**模式必须是 L **等价的**；
- 如果它们都是非**参数化**结构模式的结构模式，则在严格语法形式下，（按位置）对应的域必须是 L **等价的**；如果它们都是**参数化**结构模式，则它们的**原始变体**结构模式必须是 L **等价的**。

12.1.2.7 域的等价和 L 等价关系

当且仅当在两个给定的结构模式的上下文内指定的两个域满足下列条件时，这两个域是 1. **等价的**；2. L **等价的**：这两个域都是所属结构模式的固定域，且它们是 1. **等价的**；2. L **等价的**；或者这两个域都是所属结构模式的替换域，且它们是 1. **等价的**；2. L **等价的**。

分别按下列方式递归定义对应的固定域、变体域、替换域及变体选择项间的**等价**关系或 L **等价**关系：

- 固定域和变体域
  1. 这两个固定域或变体域必须具有**等价的域布局**；
  2. 这两个固定域或变体域的域模式必须是 1. **等价的**；2. L **等价的**。
- 替换域
  1. 这两个替换域都具有标志表或都没有标志表。在前一种情况下，两者的标志表必须具有同样多个**标志域**名字，且（按位置）对应的**标志域**名字必须表示对应的固定域；
  2. 它们必须具有同样多个变体选择项，且（按位置）对应的变体选择项必须是 1. **等价的**；2. L **等价的**；
  3. 它们必须都未指定或都指定了 **ELSE**。在后一种情况下，**ELSE** 后面必须跟有同样多个变体域，且（按位置）对应的变体域必须是 1. **等价的**；2. L **等价的**。
- 变体选择项
  1. 这两个变体选择项必须具有同样多个情况标号表，且（按位置）对应的情况标号表要么都是无关紧要，要么都定义了相同的值集；
  2. 它们必须具有同样多个变体域，且（按位置）对应的变体域必须是 1. **等价的**；2. L **等价的**。

12.1.2.8 布局间的等价关系

在本小节内，假定每个定位是形如

POS（〈字〉，〈开始位〉，〈长度〉）

的定位，每个步长是形如

STEP（〈定位〉，〈步长大小〉）

的步长。

第 3.12.5 节给出了将其它形式的定位或步长转换为上面所列形式的相应规则。

- 域布局

  如果两个**域布局**都是 **NOPACK**、**PACK** 或都是定位，则它们是**等价的**。在最后一种情况下，这两个定位必须是**等价的**（见下）。
- 元素布局

如果两个**元素布局**都是 NOPACK、PACK 或都是步长，则它们是**等价的**。在最后一种情况下，这两个步长中的定位必须是**等价的**（见下），且它们的步长大小必须传递相同的值。

- 定位

  当且仅当两个定位内出现的字、开始位及长度都分别传递相同的值时，这两个定位是**等价的**。

12.1.2.9 类同关系

当且仅当两个模式都是或都不是**只读**模式，它们都具有**新颖性** nil 或都具有相同的**新颖性**，而且满足下列条件之一时，它们是**类同的**：

- 它们都是整数模式；
- 它们都是布尔模式；
- 它们都是字符模式；
- 它们是**相似的**集合模式；
- 它们都是范围模式，且它们具有相等的**上界**和**下界**；
- 它们都是幂集模式，且它们的**成员**模式是**类同的**；
- 它们都是受限引用模式，且它们的**被引用**模式是**类同的**；
- 它们都是自由引用模式；
- 它们都是行模式，且它们的**被引用原始**模式是**类同的**；
- 它们都是过程模式，并且：
  1. 它们具有同样多个**参数说明**，且（按位置）对应的**参数说明**具有**类同的**模式和相同的参数属性（如果有的话）；
  2. 它们都具有或都没有**结果说明**，如果有**结果说明**，则它们的**结果说明**必须具有**类同的**模式及相同的属性（如果有的话）；
  3. 它们具有相同的**异常**名字表；
  4. 它们具有相同的**递归性**。
- 它们都是实例模式；
- 它们都是事件模式，且都没有**事件长度**，或具有相同的**事件长度**；
- 它们都是缓冲区模式，并且：
  1. 它们都没有**缓冲区长度**或具有相同的**缓冲区长度**；
  2. 它们的**缓冲区元素**模式是**类同的**。
- 它们都是结合模式；
- 它们都是访问模式，且
  1. 它们都没有**下标**模式或具有**类同的下标**模式；
  2. 至少有一个没有**记录**模式，或两者都具有**记录**模式。在后一种情况下，这两个**记录**模式必须是**类同的**，且要么都是**静态记录**模式，要么都是**动态记录**模式。
- 它们都是文本模式，且
  1. 它们具有相同的**文本长度**；
  2. 它们的**文本记录**模式是**类同的**；
  3. 它们的**访问**模式是**类同的**。
- 它们都是时延模式；
- 它们都是绝对时钟模式；
- 它们都是串模式，且
  1. 它们都是**位**串模式或都是**字符串**模式；
  2. 它们具有相同的**串长度**；
  3. 它们都是**定长**串模式或都是**变长**串模式。
- 它们都是数组模式，且

1. 它们的**下标**模式是**类同的**；
2. 它们的**元素**模式是类同的；
3. 它们的**元素布局**是**等价的**；
4. 它们具有相同的**元素个数**。

- 它们都是非**参数化**结构模式的结构模式，并且：
  1. 在严格语法形式下，它们的域的个数相同，且（按位置）对应的域是**类同的**；
  2. 如果它们都是**可参数化的变体**结构模式，则它们的类表必须是**相容的**。
- 它们都是**参数化**结构模式，且
  1. 它们的**原始变体**结构模式是**类同的**；
  2. 它们（按位置）对应的值相等。

12.1.2.10 域类同关系

当且仅当分属两个给定结构模式的两个域满足下列条件时，它们是**类同的**：它们都是相应结构模式的固定域或替换域，并且它们是**类同的**。

对应的固定域、变体域、替换域及变体选择项之间的**类同**关系分别递归地定义如下：

- 固定域和变体域
  1. 这两个固定域或变体域必须具有**等价的域布局**；
  2. 这两个域的**域**模式必须是**类同的**；
  3. 这两个固定域或变体域必须附有相同的名字串。
- 替换域
  1. 这两个替换域都具有或都没有标志表。在前一种情况下，这两个标志表必须具有同样多个**标志域**名字，且（按位置）对应的**标志域**名字必须表示对应的固定域；
  2. 它们必须具有同样多个变体选择项，且（按位置）对应的变体选择项必须是**类同的**；
  3. 它们都未指定或都指定了 **ELSE**。在后一种情况下，其后必须跟有同样多个变体域，且（按位置）对应的变体域必须是**类同的**。
- 变体选择项
  1. 这两个变体选择项必须具有同样多个情况标号表，且（按位置）对应的情况标号表要么都是无关紧要，要么都定义了相同的值集；
  2. 它们必须具有同样多个变体域，且（按位置）对应的变体域必须是**类同的**。

12.1.2.11 新颖性约束关系

非严格描述

在一个程序中，每个**准**新模式必须最多代表一个**实**新模式。如下确立这个要求：当一个*名字串*既**被约束**到一个**实**定义性出现之上，又**被约束**到一个**准**定义性出现之上时，所有涉及到的新模式被配对。于是在**新颖性**之间建立了**新颖性约束**关系。

定义

新颖性配对关系应用于两个模式和一个可达区之间。对在一个可达区 R 内既**被约束**到一个**实**定义性出现之上，又**被约束**到一个**准**定义性出现之上的每个*名字串*而言：

- 如果这两个定义性出现是**同义词**名字，则它们的类的**根**模式在 R 内是**新颖性配对的**；
- 如果这两个定义性出现是**集合元素**名字，则它们附着的**集合**模式的模式在 R 内是**新颖性配对的**；
- 如果这两个定义性出现是**单元**名字或**单元等同**名字，则它们的单元模式在 R 内是**新颖性配对的**；
- 如果这两个定义性出现是**过程名字**，那么，如果存在**参数说明**或**结果说明**，则它们的**参数说明**和**结果说明**的模式在 R 内是**新颖性配对的**；
- 如果这两个定义性出现是**进程**名字，那么，如果它们具有**参数说明**的话，则它们的**参数说明**的模

式在 R 内是**新颖性配对的**；

- 如果这两个定义性出现是**信号**名字，则它们的模式表内指定的模式在 R 内是**新颖性配对的**。

如果两个模式在某个可达区 R 内是**新颖性配对的**，则：

- 如果它们是幂集模式，则它们的**成员**模式在 R 内也是**新颖性配对的**；
- 如果它们是受限引用模式，则它们的**被引用**模式在 R 内也是**新颖性配对的**；
- 如果它们是行模式，则它们的**被引用原始**模式在 R 内也是**新颖性配对的**；
- 如果它们是过程模式，则它们的**参数说明**和**结果说明**的模式（如果有的话）在 R 内也是**新颖性配对的**；
- 如果它们是缓冲区模式，则它们的**缓冲区元素**模式在 R 内也是**新颖性配对的**；
- 如果它们是访问模式，则它们的**下标**模式和**记录**模式（如果有的话）在 R 内也是**新颖性配对的**；
- 如果它们是文本模式，则它们的**下标**模式（如果有的话）在 R 内也是**新颖性配对的**；
- 如果它们是数组模式，则它们的**下标**模式和**元素**模式在 R 内也是**新颖性配对的**；
- 如果它们是结构模式，则它们的**域**模式在 R 内也是**新颖性配对的**。

如果两个模式在某个可达区 R 内是**新颖性配对的**，且它们的**新颖性**不同，则这两个模式的相应**实新颖性**和**准新颖性**在 R 内是互为**新颖性约束的**。

如果两个**新颖性**满足下列条件之一，则它们被认为是相同的：

- 它们是同一个**实新颖性**；或
- 一个是**实新颖性**，另一个是**准新颖性**，且它们是**新颖性约束的**。

12.1.2.12 读相容关系

非严格描述

读相容关系是针对**等价的**模式而言的。如果模式 M 或其可能有的（子）成分与模式 N 或其可能有的（子）成分具有相同的或更严格的**只读**说明，并且，如果 M 和 N 都是引用模式，则它们都引用 L **等价的**模式，则称模式 M 与模式 N 是**读相容的**。因而这个关系是非对称的。

例如：

READ REF READ *CHAR* 与 **REF READ** *CHAR* 是**读相容的**。

定义

当且仅当模式 M 与模式 N 是等价的，并且，如果 N 是**只读**模式，则 M 也必须是**只读**模式，此外，还满足下列条件之一时，称 M 与 N 是**读相容的**（这是一种非对称关系）：

- 如果 M 和 N 都是受限引用模式，则 M 和 N 的**被引用**模式必须是 L **等价的**；
- 如果 M 和 N 都是行模式，则 M 和 N 的**被引用原始**模式必须是 L **等价的**；
- 如果 M 和 N 都是数组模式，则 M 的**元素**模式与 N 的**元素**模式必须是**读相容的**；
- 如果 M 和 N 都是非**参数化**结构模式的结构模式，则 M 的任何**域**模式与 N 的对应的**域**模式必须是**读相容的**。如果 M 和 N 都是**参数化**结构模式，则 M 的**原始变体**结构模式与 N 的**原始变体**结构模式必须是**读相容的**。

12.1.2.13 动态等价与动态读相容关系

非严格描述

1. **动态等价**，2. **动态读相容**关系是仅针对可能是动态的模式而言的（即对串模式、数组模式和**变体**结构模式而言的）。如果对**可参数化的**模式 M 而言，存在一个基于 M 的动态参数化模式与（可能是动态的）模式 N 是 1. **等价的**，2. **读相容的**，则称 M 与 N 是 1. **动态等价的**，2. **动态读相容的**。

定义

当且仅当满足下列条件之一时，模式 M 与模式 N 是 1. **动态等价的**，2. **动态读相容的**（这是一种非对称关系）：

- M 和 N 都是串模式，且 M（*p*）与 N 是 1. **等价的**，2. **读相容的**，其中 *p* 是 N 的（可能是动态的）长度。值 *p* 不能大于 M 的**串长度**。如果 N 是一个动态模式，则上述检查是动态的。检查失败将引发异常 *RANGEFAIL*；
- M 和 N 都是数组模式，且 M（*p*）与 N 是 1. **等价的**，2. **读相容的**，其中 *p* 是使得关系式 *NUM*（*p*）－*LOWER*（M）＋1＝N 的（可能是动态的）**元素个数**成立的一个值，且值 *p* 不能大于 M 的**上界**。如果 N 是一个动态模式，则上述检查是动态的。检查失败将引发异常 *RANGEFAIL*；
- M 是**可参数化的变体**结构模式，而 N 是**参数化**结构模式，且 M（p_1，…，p_n）与 N 是 1. **等价的**，2. **读相容的**，其中 p_1，…，p_n 表示 N 的值表。

12.1.2.14　可限制关系

非严格描述

可限制关系是针对具有**引用性质**的**等价**模式而言的。如果模式 M 或其可能有的（子）成分所引用的模式比模式 N 或其可能有的（子）成分所引用的模式具有相同或更严格的**只读**说明，则称模式 M 相对模式 N 是**可限制的**。故此关系是非对称的。

例如：

REF　READ　*INT*　相对　**REF**　*INT*　是**可限制的**

STRUCT（*P*　**REF　READ**　*BOOL*）相对　**STRUCT**（*Q*　**REF**　*BOOL*）是**可限制的**。

定义

当且仅当模式 M 与模式 N 是**等价的**，且满足下列条件之一时，M 相对 N 是**可限制的**（这是一种非对称关系）：

- 如果 M 和 N 都是受限引用模式，则 M 的**被引用**模式与 N 的**被引用**模式必须是**读相容的**；
- 如果 M 和 N 都是行模式，则 M 的**被引用原始**模式与 N 的**被引用原始**模式必须是**读相容的**；
- 如果 M 和 N 都是数组模式，则 M 的**元素**模式相对 N 的**元素**模式必须是**可限制的**；
- 如果 M 和 N 都是结构模式，则 M 的每个**域**模式相对 N 的对应的**域**模式必须是**可限制的**。

12.1.2.15　模式与类间的相容性

- 任何模式 M 与 **all** 类都是**相容的**；
- 当且仅当模式 M 是引用模式、过程模式或实例模式时，M 与 **null** 类是**相容的**；
- 当且仅当模式 M 是一个引用模式，且满足下列条件之一时，M 与 N 引用类是**相容的**：
  1. N 是一个静态模式，而 M 是一个受限引用模式，且其**被引用**模式与 N 是**读相容的**；
  2. N 是一个静态模式，而 M 是一个自由引用模式；
  3. M 是一个行模式，且其**被引用原始**模式与 N 是**动态读相容的**。
- 当且仅当模式 M 与模式 N 是**相似的**时，M 和 N 导出类是**相容的**；
- 当且仅当下列条件之一成立时，模式 M 和 N 值类是**相容的**：
  1. 如果 M 没有**引用性质**，则 M 和 N 必须是 V **等价的**；
  2. 如果 M 确有**引用性质**，则 M 相对 N 必须是**可限制的**。

12.1.2.16 类与类间的相容性

- 任何类与其本身都是**相容的**；
- **all** 类与任何其它的类都是**相容的**；
- **null** 类与任何 M 引用类都是**相容的**；
- 当且仅当模式 M 是一个引用模式、过程模式或实例模式时，**null** 类与 M 导出类或 M 值类是**相容的**；
- 当且仅当模式 M 与模式 N 是**等价的**时，M 引用类与 N 引用类是**相容的**。如果 M 与/或 N 是动态模式，则忽略等价性检查的动态部分，即不会出现异常；
- 当且仅当模式 N 是一个引用模式，且满足下列条件之一时，M 引用类与 N 值类是**相容的**：
  1. M 是一个静态模式，而 N 是一个受限引用模式，且其**被引用**模式与 M 是**等价的**；
  2. M 是一个静态模式，而 N 是一个自由引用模式；
  3. N 是一个行模式，且其**被引用原始**模式与 M 是**动态等价的**。
- 当且仅当模式 M 与模式 N 是**相似的**时，M 导出类与 N 导出类或 N 值类是**相容的**；
- 当且仅当模式 M 与模式 N 是 V **等价的**时，M 值类与 N 值类是**相容的**。

当且仅当两个类表具有同样多个类，且（按位置）对应的类是**相容的**时，这两个类表是**相容的**。

12.2 可见性和名字约束

可见性和名字约束的定义基于下列术语：

- *名字串*：表示一个终结串，它附着一个**规范**名字串（参见 2.7 节）和可见性性质；
- *名字*：表示一个简单*名字串*，它与创建该名字的定义性出现有关（参见 10.1 节）；
- *名字*：表示一个（可能具有带前缀名字串的）名字的应用性出现。

12.2.1 可见性等级

名字约束规则建立在程序可达区内*名字串*的可见性的基础之上。在一个可达区内部，每个*名字串*都具有表 1 所列的四种可见性等级之一。

如果一个*名字串*在某个可达区内是**直接强可见的**或**间接强可见的**，则称该*名字串*在该可达区内是**强可见的**。如果一个*名字串*在某个可达区内是**弱可见的**或**强可见的**，则称该*名字串*在该可达区内是**可见的**；否则称该*名字串*在该可达区内是**不可见的**。程序结构语句和可见性语句唯一确定了每个*名字串*属于哪一级可见性。

表 1 **可见性等级**

可见性	**性质**（非严格的定义）
直接强可见的	创建的*名字串*，移入、移出的*名字串*，从说明到体继承的*名字串*都是**可见的**。
间接强可见的	预定义的*名字串*或通过分程序嵌套继承的*名字串*。
弱可见的	由**强可见的***名字串***隐含的***名字串*
不可见的	不能使用的*名字串*

当一个*名字串*在某个可达区内是**可见的**时，它能**被直接链接**到另一个可达区中的另一个*名字串*之上，或

者**被直接链接**到该程序中的某个定义性出现之上。**直接链接**规则见 12.2.3 节。注意：对一个名字串而言，一条规则的任一次应用都引入了一次新的**直接链接**。

在**直接链接**的基础上如下定义（不必是**直接的**）**链接**的概念：

当且仅当满足下列条件之一时，称在可达区 R_1 中**可见的**名字串 N_1 **被链接**到可达区 R_2 中的名字串 N_2 或定义性出现 D 之上：

- R_1 中的 N_1 **被直接链接**到 R_2 中的 N_2 或 D 之上。然而，如果 N_1 **被直接链接**到 R_1 中的多个定义性出现之上，则除这些定义性出现之一外，所有其它的定义性出现是多余的，且 R_1 中的 N_1 **被链接**到那些定义性出现中的任意一个之上；
- R_1 中的 N_1 **被直接链接**到某个 R 中的某个 N 之上，而 R 中的 N **被链接**到 R_2 中的 N_2 或 D 之上。

12.2.2 可见性条件和名字约束

在一个程序的每个可达区内，必须满足下列条件：

- 如果某个名字串在某个可达区内是**强可见的**，并且具有多于一个的**直接链接**，那么：
  - — 它必须仅**被直接链接**到这些定义性出现之上，且这些定义性出现必须定义**相似的**集合模式的相同的集合元素，或
  - — 它必须确实**被链接**到一个**实**定义性出现和一个**准**定义性出现之上。

如果一个名字串在某个可达区内是**弱可见的**，并且在该可达区内它作为**弱可见的**名字串**被链接**到彼此**相似的**集合模式中定义的不同集合元素的那些定义性出现之上，则称该名字串在该可达区内具有**弱冲突**。

按下列规则，称在可达区 R 中**可见的**名字串 NS 在 R 中**被约束**到几个定义性出现之上：

- 如果 NS 在 R 中是**强可见的**，则 NS（作为**强可见的**名字串）**被约束**到在 R 中 NS **被链接**到的那些定义性出现之上。如果 NS 既**被约束**到一个**准**定义性出现之上，又**被约束**到一个**实**定义性出现之上，则该**准**定义性出现是冗余的，且在可见性和名字约束中不再起更多的作用（即它不能被移出、移入或继承，也不引入**隐含的**名字）；
- 否则，如果 NS 在 R 中是**弱可见的**，则 NS（作为**弱可见的**名字串）**被约束**到在 R 中 NS **被链接**到的那些定义性出现之上，假定 NS 在 R 中没有**弱冲突**（当在某个可达区内不存在具有带**弱冲突**的名字串的名字时，在该可达区内允许**弱冲突**）；
- 否则，NS 在 R 中不**被约束**。

静态条件：被直接包围在某个可达区内的每个名字所附着的名字串在该可达区内必须**被约束**。

名字的约束：在可达区 R 中带有附着名字串 NS 的名字 N **被约束**到在 R 中 NS **被约束**到的定义性出现之上。

12.2.3 可达区内的可见性

12.2.3.1 概述

按下列规则，一个名字串在一个可达区内是**直接强可见的**：

- 该名字串被移入该可达区（参见 12.2.3.5 节）；
- 该名字串被移出到该可达区（参见 12.2.3.4 节）；
- 在该可达区中存在一个具有该名字串的定义性出现。此时，在该可达区内该名字串**被直接链接**到该定义性出现之上（注意：该名字串在该可达区内可能**被直接链接**到几个定义性出现之上）。
- 该可达区是一个模块体或区域体，且该名字串在**对应的**说明模块或说明区域的可达区内是**直接强可见的**，则该名字串**被直接链接**到相应模块体或区域体的可达区中的名字串之上。

根据下列规则，在某个可达区内不是**直接强可见的**一个名字串在该可达区内是**间接强可见的**：

- 该可达区是一个分程序，且该名字串在该分程序的直接包围可达区内是**直接强可见的**。称该名字串由该分程序继承，且该名字串**被直接链接**到在该直接包围可达区内的相同名字串之上；
- 该可达区不是在其内该名字串被继承的一个分程序，且该名字串是一个语言定义的名字串（参见附录C.2）或实现定义的名字串。此时，根据其预定义含义，认为该名字串**被直接链接**到假想的最外层进程定义的可达区中的某个定义性出现之上。

当某个名字串是由在某个可达区内**强可见的**名字串**隐含的**名字串，且它在该可达区内不是**强可见的**时，称该名字串在该可达区内是**弱可见的**。该名字串在该可达区内**被直接链接**到某个**隐含的**定义性出现之上（参见12.2.4节）。

12.2.3.2 可见性语句

语法：

〈可见性语句〉::= (1)
〈移出语句〉 (1.1)
| 〈移入语句〉 (1.2)

语义：可见性语句只能出现于模体可达区中，它们控制其内提到的名字串的可见性以及隐式控制由这些名字所**隐含**的所有名字串的可见性。

静态性质：可见性语句附有一个或两个**源**可达区（参见10.2节）和一个或两个**目标**可达区，定义如下：

- 如果该可见性语句是移入语句，则其**目标**可达区是直接包围该移入语句的模体可达区，而其**源**可达区是直接包围上述模体可达区的可达区；
- 如果该可见性语句是移出语句，则其**源**可达区是直接包围该移出语句的模体可达区，而其**目标**可达区是直接包围上述模体可达区的可达区。

12.2.3.3 前缀更名子句

语法：

〈前缀更名子句〉::= (1)
(〈老前缀〉—〉〈新前缀〉)!〈后缀〉 (1.1)

〈老前缀〉::= (2)
〈前缀〉 (2.1)
| 〈空〉 (2.2)

〈新前缀〉::= (3)
〈前缀〉 (3.1)
| 〈空〉 (3.2)

〈后缀〉::= (4)
〈移入后缀〉{,〈移入后缀〉}* (4.1)
| 〈移出后缀〉{,〈移出后缀〉}* (4.2)

导出语法：其中后缀由多个移入后缀（或移出后缀）组成的前缀更名子句是几个前缀更名子句的导出语法，其中每个前缀更名子句之间用逗号隔开，且这些前缀更名子句具有相同的老前缀和新前缀，其后缀依次是相应的移入后缀（或移出后缀）。

例如：

GRANT $(p\text{—}\rangle\ q)!a, b;$

是

GRANT $(p\text{—}\rangle\ q)!a, (p\text{—}\rangle\ q)!b;$

的导出语法。

语义：前缀更名子句在可见性语句中用来表达被移出或被移入的带前缀名字串中的前缀的变更。（由于前缀更名子句能用于不改变前缀的情形 — 即*老前缀*和*新前缀*都为空 — ，因而它被作为可见性语句的语义基础）。

静态性质：*前缀更名子句*附有一个或两个**源**可达区，那就是该子句所在的*可见性语句*的**源**可达区。

*前缀更名子句*附有一个或两个**目标**可达区，那就是该子句所在的*可见性语句*的**目标**可达区。

*后缀*附有一个*名字串*集合，即由附着于其*移入后缀*或其*移出后缀*的*名字串*组成的集合。这些*名字串*是该*前缀更名子句*的后缀*名字串*。

*前缀更名子句*附有一个**老***名字串*集合和一个**新***名字串*集合。每个附着于该*前缀更名子句*的后缀*名字串*既给出了一个附着于该*前缀更名子句*的**老***名字串*，又给出了一个附着于该*前缀更名子句*的**新***名字串*，定义如下：该**新***名字串*是将该后缀*名字串*前缀以*新前缀*后得到的*名字串*；该**老***名字串*是将该后缀*名字串*前缀以*老前缀*后得到的*名字串*。

当一个**新***名字串*和一个**老***名字串*是由同一个后缀*名字串*生成的时，称该**老***名字串*是该**新***名字串*的源。

可见性规则：附着于某个*前缀更名子句*的那些**新***名字串*在该*前缀更名子句*的**目标**可达区内是**强可见的**，并且，在上述可达区内**被直接链接**到在该*前缀更名子句*的**源**可达区内的源之上。如果该*前缀更名子句*是某个*移入*（或*移出*）*语句*的一部分，则这些**新**名字串在该*前缀更名子句*的**目标**可达区内被移入（或被移出）。

当且仅当*名字串* NS 在可达区 R 内是**强可见的**，且在 R 中 NS 既未**被链接**到被可达区 R 直接包围的模体 M 的可达区内的任一*名字串*之上，也未**被直接链接**到某个预定义*名字串*的定义性出现之上时，称*名字串* NS 是可由模体 M **移入的**。

当且仅当*名字串* NS 在被可达区 R 直接包围的模体 M 的可达区内是**强可见的**，且 NS 在 M 的可达区内既未**被链接**到在 R 中的任何*名字串*之上，也未**被直接链接**到某个预定义*名字串*的定义性出现之上时，称*名字串* NS 是可由模体 M **移出的**。

静态条件：如果*前缀更名子句*出现于被直接包围在模体 M 的可达区内的某个*移入语句*中，则其每个**老***名字串*必须是：

- **被约束**到直接包围模体 M 的可达区的可达区内，且
- 可由 M **移入的**。

如果*前缀更名子句*出现于被直接包围在模体 M 的可达区内的某个*移出语句*中，则其每个**老***名字串*必须是：

- **被约束**到 M 的可达区内，且
- 可由 M **移出的**。

出现于某个*移出*（或*移入*）*语句*中的*前缀更名子句*必须包含后缀，且该后缀必须是*移出后缀*（或*移入后缀*）。

例子：

25.35 $(stack!int\text{—}\rangle\ stack)!$**ALL** *(1.1)*

12.2.3.4 移出语句

语法：

〈移出语句〉 ::= (1)

GRANT 〈前缀更名子句〉{，〈前缀更名子句〉}*； (1.1)

| **GRANT** 〈移出窗口〉[〈前缀子句〉]； (1.2)

〈移出窗口〉 ::= (2)

〈移出后缀〉{，〈移出后缀〉}* (2.1)

〈移出后缀〉 ::= (3)

〈名字串〉 (3.1)

| 〈新模式名字串〉〈禁止子句〉 (3.2)

| [〈前缀〉!] **ALL** (3.3)

〈前缀子句〉 ::= (4)

PREFIXED [〈前缀〉] (4.1)

〈禁止子句〉 ::= (5)

FORBID {〈禁止名字表〉|**ALL**} (5.1)

〈禁止名字表〉 ::= (6)

(〈域名字〉{，〈域名字〉}*) (6.1)

语义：移出语句是一种把在某个模体可达区内的名字串的可见性扩展到该模体可达区的直接包围可达区中的手段。**FORBID** 是只为结构模式的**新模式**名字指定的，这意味着具有该模式的所有单元和值在该移出模体内部可以选取这些域，但在该模体外部却不能选取禁止名字表中指明的那些域。

下列可见性规则适合使用：

- 如果移出语句包含前缀更名子句，则该移出语句具有其内出现的诸前缀更名子句的作用（参见12.2.3.3节）；
- 如果移出语句包含移出窗口，则它是一组如下构造出来的带前缀更名子句的移出语句的缩写表示法：
  - — 对移出窗口中每个移出后缀而言，都存在一个相应的移出语句；
  - — 这些移出语句中的前缀更名子句中的老前缀为空；
  - — 这些移出语句中的前缀更名子句中的新前缀是原移出语句内前缀子句所附着的前缀，当原移出语句中没有前缀子句时，上述新前缀为空；
  - — 这些移出语句中的前缀更名子句中的后缀是移出窗口内相应的移出后缀。
- 表示法 **FORBID ALL** 是禁止指定的**新模式**名字的所有域名字的缩写表示法（参见12.2.5节）；
- 如果移出语句中的前缀更名子句具有一个包含前缀和 **ALL** 的移出后缀，则该前缀更名子句是形如：

  (*OP*—〉*NP*)!*P*!**ALL**

  的前缀更名子句，其中，*OP* 和 *NP* 分别是可能为空的老前缀和新前缀，而 *P* 是该移出后缀中指定的前缀。故这种前缀更名子句是形如：

  (*OP*!*P*—〉*NP*!*P*)!**ALL**

  的前缀更名子句的缩写表示法。

静态性质：前缀子句附着一个前缀，定义如下：

- 如果该前缀子句包含前缀，则它所附着的前缀就是该前缀；

否则，所附着的前缀是一个其名字串如下确定的简单前缀：

— 如果直接包围该前缀的可达区是一个模块或区域，则该名字串与该模体的模体名字之一的名字串相同；

— 如果直接包围该前缀的可达区是一个说明区域或说明模块，则该名字串是 **SPEC** 前面的名字串。

*移出后缀*附着有一个名字串集合，定义如下：

- 如果该*移出后缀*是一个名字串，或包含一个*新模式*名字串，则该集合仅仅包含该名字串；
- 否则，令 *OP* 是该*移出后缀*所在的*前缀更名子句*中的（可能为空的）*老前缀*，那么，对任何满足下列条件的名字串 *N* 而言，该集合包含所有形如 *OP*!*N* 的名字串（即在 N 前面加上前缀 *OP* 后得到的名字串）：(1) .*OP*!*N* 在该*移出后缀*所在的模体的可达区内是**强可见的**；(2) .*OP*!*N* 是可由该模体**移出的**。

静态条件：带禁止子句的*新模式*名字串在该*移出语句*所在的模体可达区 R 内必须是**强可见的**。在 R 内该*新模式*名字串必须**被约束**到一个结构模式的**新模式**名字的定义性出现之上，且*禁止名字表*中提到的每个域名字必须是那个结构模式的域名字之一。该**新模式**定义性出现必须被 R 直接包围。*禁止名字表*中的所有域名字必须具有不同的名字串。

如果*移出语句*位于某个区域或说明区域的可达区内，则它不能移出满足下列条件的名字串：在上述可达区内该名字串**被约束**到一个定义性出现之上，且该定义性出现是：

- 某个**单元**名字的定义性出现；或
- 某个**单元等同**名字的定义性出现，而其说明中指定的单元是**区域内的**；或
- 某个其值是**区域内的同义词**名字的定义性出现。

*移出语句*中的*前缀更名子句*必须包含一个*移出后缀*。

如果*移出语句*包含不带前缀的*前缀子句*，则其直接包围模体不能是一个*上下文*，且：

- 如果其直接包围模体是一个模块或区域，则它必须被命名（即它必须以后面跟有一个冒号的定义性出现打头）；
- 如果其直接包围模体是一个说明模块或说明区域，则它必须以一个*简单名字串*打头。

例子：

```
25.7    GRANT (-> stack ! char) ! ALL;                    (1.1)
6.44    gregorian_date, julian_day_number                 (2.1)
```

12.2.3.5 *移入语句*

语法：

```
〈移入语句〉 ::=                                            (1)
    SEIZE 〈前缀更名子句〉 {, 〈前缀更名子句〉}*;            (1.1)
  | SEIZE 〈移入窗口〉 [ 〈前缀子句〉];                      (1.2)

〈移入窗口〉 ::=                                            (2)
    〈移入后缀〉 {, 〈移入后缀〉}*                           (2.1)

〈移入后缀〉 ::=                                            (3)
    〈名字串〉                                              (3.1)
  | [ 〈前缀〉!] ALL                                        (3.2)
```

语义：移入语句是把在一个组块可达区内的名字串的可见性扩展到被该组块可达区直接包围的模体的可达区内的方法。

下列可见性规则适合使用：

- 如果*移入语句*包含*前缀更名子句*，则该*移入语句*具有其内所含的*前缀更名子句*的作用（参见12.2.3.3节）；
- 如果*移入语句*包含*移入窗口*，则它是一组如下构造出来的带*前缀更名子句*的*移入语句*的缩写表示法：
  - — 对*移入窗口*中的每个*移入后缀*而言，都存在一个相应的*移入语句*；
  - — 这些*移入语句*中的*前缀更名子句*中的*老前缀*是附着于原*移入语句*内*前缀子句*之上的*前缀*，当原*移入语句*不带*前缀子句*时，上述*老前缀*为空；
  - — 这些*移入语句*中的*前缀更名子句*中的*新前缀*为空；
  - — 这些*移入语句*中的*前缀更名子句*中的*后缀*是*移入窗口*中相应的*移入后缀*。
- 如果*移入语句*中的*前缀更名子句*具有一个含有*前缀*和 **ALL** 的*移入后缀*，则该*前缀更名子句*是形如：

  *(OP—〉 NP)!P!***ALL**

  的*前缀更名子句*，其中 *OP* 和 *NP* 分别是可能为空的*老前缀*和*新前缀*，而 *P* 是该*移入后缀*中指定的*前缀*。故这种*前缀更名子句*是形如：

  *(OP!P—〉 NP!P)!***ALL**

  的*前缀更名子句*的缩写表示法。

静态性质：*移入后缀*附有一个*名字串*集合，定义如下：

- 如果该*移入后缀*是*名字串*，则上述集合仅包含该*名字串*；
- 否则，如果该*移入后缀*是 **ALL**，令 *OP* 是该*移入后缀*所在的*前缀更名子句*中的*老前缀*，则对满足下列条件的任一*名字串* *S* 而言，上述集合包含所有形如 *OP!S* 的*名字串*：(1) *.OP!S* 在直接包围相应*移入语句*所在的模体的可达区内是**强可见的**；(2) *.OP!S* 是可由该模体**移入的**。

静态条件：*移入语句*中的*前缀更名子句*必须具有一个*移入后缀*。

如果某个*移入语句*包含不带*前缀*的*前缀子句*，则其直接包围模体不能是一个上下文，且：

- 如果其直接包围模体是一个*模块*或*区域*，则该*模块*或*区域*必须被命名（即它必须以后跟一个冒号的*定义性出现*开始）；
- 如果其直接包围模体是一个*说明模块*或*说明区域*，则该*说明模块*或*说明区域*必须以一个*简单名字串*开始。

例子：

25.35 **SEIZE** *(stack ! int -> stack) !* **ALL**; *(1.1)*

12.2.4 隐含的名字串

在可达区 R 中每个**强可见的***名字串*具有一个由**隐含的***名字串*组成的集合，这些**隐含的***名字串*在 R 中可能是**弱可见的**。

如表 2 所列，每种*模式*在一个可达区内附有一个可能为空的**隐含的***定义性出现*集合。

表 2　在可达区 R 中模式所隐含的定义性出现

模式	隐含的定义性出现集合
INT, *BOOL*, *CHAR*, **RANGE** (…), **BIN** (*n*), *PTR*, *INSTANCE*, **EVENT**, *ASSOCIATION*, *TIME*, *DURATION*, **BOOLS** (*n*), **CHARS** (*n*)	空
模式名字	由其定义模式在 R 内所**隐含的**定义性出现集合
模式名字 (…)（参数化模式）	在 R 中由模式名字所**隐含的**定义性出现集合
M (*m*: *n*), **REF** *M*, **ROW** *M*, **READ** *M*, **POWERSET** *M*, **BUFFER** *M*, **TEXT** (…) *M*	在 R 中由 *M* 所**隐含的**定义性出现集合
SET (…)	该模式中**集合元素**定义性出现组成的集合
PROC (M_1, …M_n) (M_{n+1})	在 R 中由 M_1 到 M_{n+1}所**隐含的**定义性出现集合的并集
ARRAY (*M*) *N*, **ACCESS** (*M*) *N*	在 R 中由 *M* 和 *N* 所**隐含的**定义性出现集合的并集
STRUCT (N_1 M_1, …, N_n M_n)	在 R 中由 M_1 到 M_n 所**隐含的**定义性出现集合的并集。 对变体结构模式而言，它是由变体结构模式的域（该域在 R 中是**可见的**）的域模式在 R 中所**隐含的**定义性出现集合的并集。

在可达区 R 中**强可见的**每个名字串 NS 具有如下定义的**隐含的**定义性出现集合，其中 D 是在 R 内**约束** NS 到定义性出现上的那些定义性出现之一：

- 如果 D 定义了一个具有模式 M 的**访问**名字，则在 R 内 NS 的**隐含的**定义性出现是在 R 中的那些由 M 所**隐含的**定义性出现；
- 如果 D 定义了一个**模式**名字，则在 R 中 NS 的**隐含的**定义性出现是在 R 中由该**模式**名字所**隐含的**那些定义性出现；
- 如果 D 定义了一个**过程**名字，那么，如果该过程具有非空的**参数说明**表或**结果说明**，则在 R 中 NS 的**隐含的**定义性出现是在 R 中由这些**参数说明**和**结果说明**内指定的模式所**隐含的**那些定义性出现；
- 如果 D 定义了一个**进程**名字，那么，如果该进程具有非空的**参数说明**表，则在 R 中 NS 的**隐含的**定义性出现是在 R 中由这些**参数说明**内指定的模式所**隐含的**那些定义性出现；
- 如果 D 定义了一个**信号**名字，则在 R 中 NS 的**隐含的**定义性出现是在 R 中由附着在该信号之上的所有模式所**隐含的**那些定义性出现；
- 否则，在 R 中 NS 的**隐含的**定义性出现集合为空。

如果在可达区 R 内**强可见的**名字串 NS 具有**隐含的**定义性出现，则每个这种定义性出现都说明了在 R 内 NS 的一个**隐含的**名字串：令 D 是在 R 内由 NS 所**隐含的**一个定义性出现，N_i 是 D 的名字串，存在两种情况：

- NS 是一个简单名字串，那么 Ni 就是在 R 中 NS 的**隐含的**名字串；
- NS 是形如 *P*! *S* 的名字串，其中 *S* 是一个简单名字串，那么，*P*! *Ni* 就是在 R 中 NS 的**隐含的**名字串。

例子：

```
m: MODULE
      DCL x SET (on, off);
      GRANT x PREFIXED;
END;
/* m ! x visible here with implied m ! on, m ! off */
```

12.2.5 域名字的可见性

域名字只能出现在下列上下文中：

- 结构域和值结构域；
- 带标号结构多元组；
- 移出语句中的禁止子句。

在这些情形下，域名字的名字串能**被约束**到在模式 M 或 M 的定义模式中的某个域名字定义性出现之上，其中 M 分别是：

- 该结构域内指定的结构单元或该值结构域内指定的（**强**）结构原值的模式；
- 该结构多元组的模式；
- 在该禁止子句所在的可达区内指定的新模式名字串**被约束**到定义性出现之上的那些定义性出现的模式。

然而，如果 M 的**新颖性**是一个定义性出现，且它定义了一个**新模式**名字，而该**新模式**名字在某个模体中又作为一个带禁止子句的移出语句中的移出后缀而被移出的话，则在该禁止子句的禁止名字表内提到的域名字只在下列情形是**可见的**：

- 在该移出模体的组块内是**可见的**；
- 如果 M 的**新颖性被新颖性约束**到一个**准新颖性** N 之上，则在直接包围 N 的可达区的组块内是**可见的**；
- 如果该模体是一个模块说明或区域说明，则在**对应的**模体的可达区内是**可见的**。

在上述几种可达区之外，上述禁止名字表中提到的域名字是**不可见的**，也不能使用这些域名字。

12.3 情况选择

语法：

```
〈情况标号说明〉 ::=                                    (1)
    〈情况标号表〉 {, 〈情况标号表〉}*                    (1.1)

〈情况标号表〉 ::=                                      (2)
    ( 〈情况标号〉 {, 〈情况标号〉}* )                    (2.1)
  | 〈无关紧要〉                                        (2.2)

〈情况标号〉 ::=                                        (3)
    〈离散字面值表达式〉                                 (3.1)
  | 〈字面值范围〉                                      (3.2)
  | 〈离散模式名字〉                                    (3.3)
  | ELSE                                              (3.4)

〈无关紧要〉 ::=                                        (4)
    ( * )                                              (4.1)
```

语义：情况选择是一种从一组选择项中选择一个选择项的手段。这种选择基于指定的选择器值表。情况选择可以应用于下列情形：

- 替换域（参见3.12.4节），此时选择一串变体域；
- 带标号数组多元组（参见5.2.5节），此时选择一个数组元素值；

- 条件表达式（参见5.3.2节），此时选择一个表达式；
- 情况动作（参见6.4节），此时选择一个动作语句表。

在第一、第三及第四种情形中，每个选择项都以一个情况标号说明开始，而在带标号数组多元组中，每个值前面都有一个情况标号表。出于易于叙述起见，在本节中带标号数组多元组内的情况标号表被认为是只具有一个情况标号表出现的情况标号说明。

情况选择选出以与选择器值表匹配的情况标号说明开始的选择项（选择器值的个数总是与情况标号说明中出现的情况标号表的个数相同）。当且仅当一个值表内的每个值均与某个情况标号说明内（按位置）对应的情况标号表相匹配时，称该值表与该情况标号说明匹配。

当且仅当满足下列条件之一时，称一个值与一个情况标号表相匹配：

- 该情况标号表由若干情况标号组成，且该值是由这些情况标号之一显式指出的那些值之一或是由情况标号 **ELSE** 隐式指出的那些值之一；
- 该情况标号表由*无关紧要*组成。

由一个情况标号显式指出的值是相应*离散字面值*表达式所传递的值，或者是由相应字面值范围或*离散模式*名字所定义的那些值。由情况标号 **ELSE** 隐式指出的值是相应选择器所有可能取的值中未被任何情况标号说明里的任何对应情况标号表（即属于同一选择器值的情况标号表）显式指出的那些值。

静态性质：

- 带情况标号说明的替换域、带标号数组多元组、条件表达式或情况动作都附有一个情况标号说明表，该表分别是由每个变体选择项、值或情况选择项前面的情况标号说明组成的。
- 每个情况标号附有一个类。如果该情况标号是*离散字面值*表达式，则其类就是该*离散字面值*表达式的类；如果该情况标号是字面值范围，则其类就是该字面值范围中的两个*离散字面值*表达式的类的**结果类**；如果该情况标号是*离散模式*名字，则其类就是 **M** 值类的结果类，其中 **M** 是该离散模式名字；如果该情况标号是 **ELSE**，则其类是 **all** 类。
- 每个情况标号表附有一个类。如果它是*无关紧要*，则其类是 **all** 类，否则是该情况标号表中每个情况标号的类的**结果类**。
- 每个情况标号说明附有一个类表，它就是由该情况标号说明内诸情况标号表的类组成的类表。
- 情况标号说明表附有一个**结果类表**，该**结果类**表是如下构成的：对应于情况标号说明表中的每个位置都存在一个**结果类**，它就是每个情况标号说明所附着的类表中与该位置对应的类的**结果类**。

当且仅当某个情况标号说明表中对所有可能的选择器值表而言都存在一个情况标号说明与之匹配时，该情况标号说明表是**完备的**。所有可能的选择器值的集合由情况选择出现的上下文确定如下：

- 对**带标志变体**结构模式而言，它是相应的**标志**域的域模式所定义的值集；
- 对**无标志变体**结构模式而言，它是相应的**结果类**（这个类绝不可能是 **all** 类，参见3.12.4节）的**根模**式所定义的值集；
- 对数组多元组而言，它是该数组多元组的模式的**下标**模式所定义的值集；
- 对带范围表的情况动作而言，它是该范围表内相应的离散模式所定义的值集；
- 对不带范围表的情况动作或条件表达式而言，它是 M 所定义的值集，而相应选择器的类是 M 值类或 M 导出类。

静态条件：对每个情况标号说明而言，情况标号表出现的个数必须相等。

对任意两个情况标号说明而言，它们的类表必须是**相容的**。

由情况标号说明出现组成的表必须是**一致的**，即每个可能的选择器值表最多只能与一个情况标号说明匹配。

例子：

11.9 *(occupied)* *(2.1)*
11.58 *(rook),(*)* *(1.1)*
8.26 *(**ELSE**)* *(2.1)*

12.4 语义范畴的定义与摘要

本节给出所有语义范畴的概况，语义范畴由语法描述中带下划线的部分指明。如果在有关章节中未给出这些语义范畴的定义，则它们的定义在本节给出，否则仅提示所引用的有关章节。

12.4.1 名字

模式名字

绝对时钟模式名字：由绝对时钟模式定义的名字。
访问模式名字：由访问模式定义的名字。
数组模式名字：由数组模式定义的名字。
结合模式名字：由结合模式定义的名字。
布尔模式名字：由布尔模式定义的名字。
受限引用模式名字：由受限引用模式定义的名字。
缓冲区模式名字：由缓冲区模式定义的名字。
字符模式名字：由字符模式定义的名字。
离散模式名字：由离散模式定义的名字。
时延模式名字：由时延模式定义的名字。
事件模式名字：由事件模式定义的名字。
自由引用模式名字：由自由引用模式定义的名字。
实例模式名字：由实例模式定义的名字。
整数模式名字：由整数模式定义的名字。
模式名字：参见 3.2.1 节。
新模式名字：参见 3.2.3 节。
参数化数组模式名字：由**参数化**数组模式定义的名字。
参数化串模式名字：由**参数化**串模式定义的名字。
参数化结构模式名字：由**参数化**结构模式定义的名字。
幂集模式名字：由幂集模式定义的名字。
过程模式名字：由过程模式定义的名字。
范围模式名字：由范围模式定义的名字。
行模式名字：由行模式定义的名字。
集合模式名字：由集合模式定义的名字。
串模式名字：由串模式定义的名字。
结构模式名字：由结构模式定义的名字。
同义模式名字：参见 3.2.2 节。
变体结构模式名字：由**变体**结构模式定义的名字。

访问名字

单元名字：参见 4.1.2 节。
单元开域名字：参见 6.5.4 节。
单元枚举名字：参见 6.5.2 节。

单元等同名字：参见 4.1.3 节。

值名字

布尔字面值名字：参见 5.2.4.3 节。
空字面值名字：参见 5.2.4.6 节。
同义词名字：参见 5.1 节。
值开域名字：参见 6.5.4 节。
值枚举名字：参见 6.5.2 节。
值接收名字：参见 6.19.2、6.19.3 节。

其它名字

受限引用单元名字：具有受限引用模式的单元名字。
内部子程序名字：表示某个内部子程序的任何 CHILL 或实现定义的名字。
自由引用单元名字：具有自由引用模式的单元名字。
通用过程名字：其**通用性**为**通用的**过程名字。
标号名字：参见 6.1 和 10.6 节。
新模式名字串：**被约束**到某个**新模式**名字的定义性出现之上的名字串。
非保留名字：不是附录 C.1 中提到的诸保留简单名字串的名字。
过程名字：参见 10.4 节。
进程名字：参见 10.5 节。
集合元素名字：参见 3.4.5 节。
信号名字：参见 11.5 节。
标志域名字：参见 3.12.4 节。
未定义同义词名字：参见 5.1 节。

12.4.2 单元

访问单元：具有访问模式的单元。
数组单元：具有数组模式的单元。
结合单元：具有结合模式的单元。
字符串单元：具有**字符**串模式的单元。
缓冲区单元：具有缓冲区模式的单元。
离散单元：具有离散模式的单元。
事件单元：具有事件模式的单元。
实例单元：具有实例模式的单元。
静态模式单元：具有静态模式的单元。
串单元：具有串模式的单元。
结构单元：具有结构模式的单元。
文本单元：具有文本模式的单元。

12.4.3 表达式和值

绝对时钟原值：其类与某个绝对时钟模式**相容**的原值。
数组表达式：其类与某个数组模式**相容**的表达式。
数组原值：其类与某个数组模式**相容**的原值。
布尔表达式：其类与某个布尔模式**相容**的表达式。

受限引用原值：其类与某个受限引用模式**相容**的原值。
字符串表达式：其类与某个**字符**串模式**相容**的表达式。
常数值：是**常数**的值。
离散表达式：其类与某种离散模式**相容**的表达式。
离散字面值表达式：是**字面值**的离散表达式。
时延原值：其类与某个时延模式**相容**的原值。
自由引用原值：其类与某个自由引用模式**相容**的原值。
实例原值：其类与某个实例模式**相容**的原值。
整数表达式：其类与某个整数模式**相容**的表达式。
整数字面值表达式：是**字面值**的整数表达式。
幂集表达式：其类与某个幂集模式**相容**的表达式。
过程原值：其类与某个过程模式**相容**的原值。
引用原值：其类与某个受限引用模式、自由引用模式或行模式**相容**的原值。
行原值：其类与某个行模式**相容**的原值。
串表达式：其类与某个串模式**相容的**表达式。
串原值：其类与某个串模式**相容**的原值。
结构原值：其类与某个结构模式**相容**的原值。

12.4.4 其它语义范畴

数组模式：是被定义为数组模式的组合模式的模式。
离散模式：是被定义为离散模式的非组合模式的模式。
单元内部子程序调用：参见 6.7 节。
单元过程调用：参见 6.7 节。
非保留字符：既不是引号（"），又不是上箭头（^）的字符。
非专用字符：既不是上箭头（^），又不是开圆括号（（）的字符。
串模式：是被定义为串模式的组合模式的模式。
值内部子程序调用：参见 6.7 节。
值过程调用：参见 6.7 节。

13 实现任选

13.1 实现定义的内部子程序

语义：除语言定义的一组内部子程序外，实现也可以提供一组由实现定义的内部子程序。

实现定义的内部子程序调用的参数传递机制由实现规定。

预定义名字：实现定义的内部子程序的名字被预定义为**内部子程序**名字。

静态性质：**内部子程序**名字可能附有一个实现定义的异常名字集合。当且仅当实现指出对一个内部子程序调用的参数所选定的静态性质及该调用的给定的静态上下文而言，该内部子程序调用传递一个值（或单元）时，该内部子程序调用是**值**（或**单元**）内部子程序调用。

实现也指出**值**（或**单元**）内部子程序调用所返送的值（或单元）的**区域性**。

13.2 实现定义的整数模式

实现定义整数模式 *INT* 的**上界**和**下界**。实现可以定义非由 *INT* 定义的其它整数模式，如短整数、长整数、无符号整数等等。这些整数模式必须由实现定义的**整数**模式名字表示。这些名字被认为是与 *INT* **相似**的**新模式**名字。它们的取值范围由实现定义。这些整数模式可以被定义为相应类的**根**模式。

13.3 实现定义的进程名字

实现可以定义一组由实现定义的**进程**名字，即其定义不是用 CHILL 描述的**进程**名字。这种进程定义被认为是位于假想的最外层进程的可达区或任意程序上下文内的。可以启动具有这种名字的进程，也可以处理标志这种进程的实例值。

13.4 实现定义的处理程序

实现可以指定将一个实现定义的处理程序附着到一个进程定义之上。这种处理程序能处理任何异常。

13.5 实现定义的异常名字

实现可以定义一组异常名字。

13.6 其它实现定义的特性

- 动态条件的静态检查(参见 2.1.2 节)；
- 实现编译指示(参见 2.6 节)；
- 正文引用名字(参见 2.7 和 10.10.1 节)；
- 缺省的**递归性**和**通用性**(参见 3.7 和 10.4 节)；
- 延时模式的值集(参见 3.11.2 节)；
- 绝对时钟模式的值集(参见 3.11.3 节)；
- 缺省的**元素布局**(参见 3.12.3 节)；
- **无标志变体**结构值的比较(参见 3.12.4 节)；

- 一个字中的位数(参见 3.12.5 节);
- 最小位占据(参见 3.12.5 节);
- 其它**可引用的**(子)单元(参见 4.2.1 节);
- 是某个**无标志变体**结构单元的一个**变体**域的*单元开域*名字和*值开域*名字的语义(参见 4.2.2 和 5.2.3 节);
- **无标志变体**结构的**变体域**的语义(参见 4.2.10、5.2.13 和 6.2 节);
- *单元转换*的语义(参见 4.2.13 节);
- *表达式转换*的语义及附加条件(参见 5.2.11 节);
- *启动表达式*内额外的实参的语义(参见 5.2.14 节);
- **字面值**表达式和**常数**表达式的值范围(参见 5.3.1 节);
- (进程)调度算法(参见 6.15、6.18.2、6.18.3、6.19.2 及 6.19.3 节);
- *TERMINATE* 内所隐含的存储释放(参见 6.20.4 节);
- 文件的表示(参见 7.1 节);
- 结合上的操作(参见 7.1 和 7.2.1 节);
- 非排它性结合(参见 7.1 节);
- 结合值的附加属性(参见 7.2.2 节);
- *结合参数*的语义(参见 7.4.2 节);
- 引发异常 *ASSOCIATEFAIL* 的条件(参见 7.4.2 节);
- *修改参数*的语义(参见 7.4.5 节);
- 引发异常 *CREATEFAIL*、*DELETEFAIL*、*MODIFYFAIL* 的条件(参见 7.4.5 节);
- 引发异常 *CONNECTFAIL* 的条件(参见 7.4.6 节);
- 根据记录模式判定读出的记录是非法的值时读记录操作的语义(参见 7.4.9 节);
- 其它**可超时的**动作(参见 9.2 节);
- 引发异常 *TIMERFAIL* 的条件(参见 9.3.1、9.3.2 及 9.3.3 节);
- 时延值的精度(参见 9.4.1 和 9.4.2 节);
- *准同义词定义*中的**常数**值的指定(参见 10.10.3 节);
- 内部子程序的**区域性**(参见 11.2.2 节)。

附录 A：CHILL 字符集

CHILL 字符集是 CCITT 五号字母表（即国际引用版，建议 V3）的一种扩充。对其整数表示值大于 127 的那些字符未定义图形表示。

字符的整数表示是从位 b_8 到位 b_1 组成的二进制整数，其中 b_1 是最低有效位。

	$b_7b_6b_5$	000	001	010	011	100	101	110	111
$b_4b_3b_2b_1$		0	1	2	3	4	5	6	7
0000	0	NUL	TC_7 (DLE)	SP	0	@	P	‘	p
0001	1	TC_1 (SOH)	DC_1	!	1	A	Q	a	q
0010	2	TC_2 (STX)	DC_2	"	2	B	R	b	r
0011	3	TC_3 (ETX)	DC_3	#	3	C	S	c	s
0100	4	TC_4 (EOT)	DC_4	$	4	D	T	d	t
0101	5	TC_5 (ENQ)	TC_8 (NAK)	%	5	E	U	e	u
0110	6	TC_6 (ACK)	TC_9 (SYN)	&	6	F	V	f	v
0111	7	BEL	TC_{10} (ETB)	’	7	G	W	g	w
1000	8	FE_0 (BS)	CAN	(	8	H	X	h	x
1001	9	FE_1 (HT)	EM	)	9	I	Y	i	y
1010	10	FE_2 (LF)	SUB	*	:	J	Z	j	z
1011	11	FE_3 (VT)	ESC	+	;	K	[	k	{
1100	12	FE_4 (FF)	IS_4 (FS)	,	<	L	\	l	|
1101	13	FE_5 (CR)	IS_3 (GS)	-	=	M	]	m	}
1110	14	SO	IS_2 (RS)	.	>	N	^	n	~
1111	15	SI	IS_1 (US)	/	?	O	_	o	DEL

附录 B：专用符号和专用字符组合

	名字	用途
;	分号	语句终止符号等
,	逗号	若干语法成分的分隔符
(	左圆括号	若干成分的开圆括号
)	右圆括号	若干成分的闭圆括号
[	左方括号	多元组的开括号
]	右方括号	多元组的闭括号
(:	左多元组括号	多元组的开括号
:)	右多元组括号	多元组的闭括号
:	冒号	标号指示符，范围指示符
.	圆点	域选择符号
:=	赋值号	赋值、初始化
<	小于	关系运算符
<=	小于等于	同上
=	等于	关系运算符、赋值、初始化、定义指示符
/=	不等	关系运算符
>=	大于等于	同上
>	大于	同上
+	加号	加法运算符
-	减号	减法运算符
*	星号	重复运算符、未定义值、未命名值、无关紧要符号
/	斜杠	除法运算符
//	双斜杠	并置运算符
->	箭头	引用与间接引用，前缀更名
<>	方块	编译指示子句的开始或结束
/*	注释开括号	带括号注释的开始
*/	注释闭括号	带括号注释的结束
'	撇号	若干字面值的开始或结束符号
"	引号	字符串字面值的开始或结束符号
""	双引号	字符串字面值内部出现的引号
!	置前缀运算符	给名字加前缀
B'	字面值限定符	二进制字面值
D'	同上	十进制字面值
H'	同上	十六进制字面值
O'	同上	八进制字面值
--	行尾	行尾注释的行结束分隔符

附录 C：专用简单名字串

C.1 保留简单名字串

ACCESS
AFTER
ALL
AND
ANDIF
ARRAY
ASSERT
AT

BEGIN
BIN
BODY
BOOLS
BUFFER
BY

CASE
CAUSE
CHARS
CONTEXT
CONTINUE
CYCLE

DCL
DELAY
DO
DOWN
DYNAMIC

ELSE
ELSIF
END
ESAC
EVENT
EVER
EXCEPTIONS
EXIT

FI
FOR
FORBID

GENERAL
GOTO
GRANT

IF
IN
INIT
INLINE
INOUT

LOC

MOD
MODULE

NEWMODE
NONREF
NOPACK
NOT

OD
OF
ON
OR
ORIF
OUT

PACK
POS
POWERSET
PREFIXED
PRIORITY
PROC
PROCESS

RANGE
READ
RECEIVE
RECURSIVE
REF
REGION
REM
REMOTE
RESULT
RETURN
RETURNS
ROW

SEIZE
SEND
SET
SIGNAL
SIMPLE
SPEC
START
STATIC
STEP
STOP
STRUCT
SYN
SYNMODE

TEXT
THEN
THIS
TIMEOUT
TO

UP

VARYING

WHILE
WITH

XOR

C.2 预定义简单名字串

ABS
ABSTIME
ALLOCATE
ASSOCIATE
ASSOCIATION

BOOL

CARD
CHAR
CONNECT
CREATE

DAYS
DELETE
DISCONNECT
DISSOCIATE
DURATION

EOLN
EXISTING
EXPIRED

FALSE
FIRST

GETASSOCIATION
GETSTACK
GETTEXTACCESS
GETTEXTINDEX
GETTEXTRECORD
GETUSAGE

HOURS

INDEXABLE
INSTANCE
INT
INTTIME
ISASSOCIATED

LAST
LENGTH
LOWER

MAX
MILLISECS
MIN
MINUTES
MODIFY

NULL
NUM

OUTOFFILE

PRED
PTR

READABLE
READONLY
READRECORD
READTEXT
READWRITE

SAME
SECS
SEQUENCIBLE
SETTEXTACCESS
SETTEXTINDEX
SETTEXTRECORD
SIZE
SUCC

TERMINATE
TIME
TRUE

UPPER
USAGE

VARIABLE

WAIT
WHERE
WRITEABLE
WRITEONLY
WRITERECORD
WRITETEXT

C.3 异常名字

ALLOCATEFAIL
ASSERTFAIL
ASSOCIATEFAIL
CONNECTFAIL
CREATEFAIL
DELAYFAIL
DELETEFAIL
EMPTY
MODIFYFAIL
NOTCONNECTED
NOTASSOCIATED
OVERFLOW
RANGEFAIL
READFAIL
SENDFAIL
SPACEFAIL
TAGFAIL
TEXTFAIL
TIMERFAIL
WRITEFAIL

附录 D：程序实例

1. 整数运算

```
integer_operations:
MODULE

    add:
    PROC (i,j INT) RETURNS (INT) EXCEPTIONS (OVERFLOW);
            RESULT i+j;
    END add;

    mult:
    PROC (i,j INT) RETURNS (INT) EXCEPTIONS (OVERFLOW);
            RESULT i*j;
    END mult;

    GRANT add, mult;
    SYNMODE operand_mode=INT;
    GRANT operand_mode;
    SYN neutral_for_add=0,
        neutral_for_mult=1;
    GRANT neutral_for_add,
          neutral_for_mult;

END integer_operations;
```

2. 分数上的同类运算

```
fraction_operations:
MODULE
    NEWMODE fraction=STRUCT (num,denum INT);

    add:
    PROC (f1,f2 fraction) RETURNS (fraction) EXCEPTIONS (OVERFLOW);
            RETURN [f1.num*f2.denum+f2.num*f1.denum,f1.denum*f2.denum];
    END add;

    mult:
    PROC (f1,f2 fraction) RETURNS (fraction) EXCEPTIONS (OVERFLOW);
            RETURN [f1.num*f2.num,f2.denum*f1.denum];
    END mult;

    GRANT add, mult;
    SYNMODE operand_mode=fraction;
    GRANT operand_mode;
    SYN neutral_for_add fraction=[ 0,1 ],
        neutral_for_mult fraction=[ 1,1 ];
    GRANT neutral_for_add,
          neutral_for_mult;

END fraction_operations;
```

3. 复数上的同类运算

```
1   complex_operations:
2   MODULE
3       NEWMODE complex=STRUCT (re,im INT);
4
5       add:
6       PROC (c1,c2 complex) RETURNS (complex) EXCEPTIONS (OVERFLOW);
7               RETURN [c1.re+c2.re,c1.im+c2.im];
8       END add;
9
10      mult:
11      PROC (c1,c2 complex) RETURNS (complex) EXCEPTIONS (OVERFLOW);
12              RETURN [c1.re*c2.re-c1.im*c2.im,c1.re*c2.im+c1.im*c2.re];
13      END mult;
14
15      GRANT add, mult;
16      SYNMODE operand_mode=complex;
17      GRANT operand_mode;
18      SYN neutral_for_add=complex [ 0,0 ],
19          neutral_for_mult=complex [ 1,0 ];
20      GRANT neutral_for_add,
21              neutral_for_mult;
22
23  END complex_operations;
```

4. 广阶算术

```
1   general_order_arithmetic: /* from collected algorithms from CACM no. 93 */
2   MODULE
3       op:
4       PROC (a INT INOUT, b,c,order INT)
5               EXCEPTIONS (wrong_input) RECURSIVE;
6           DCL d INT;
7           ASSERT b>0 AND c>0 AND order>0
8               ON (ASSERTFAIL):
9                   CAUSE wrong_input;
10              END;
11          CASE order OF
12              (1):    a := b+c;
13                      RETURN;
14              (2):    d := 0;
15              (ELSE): d := 1;
16          ESAC;
17          DO FOR i := 1 TO c;
18              op (a,b,d,order-1);
19              d := a;
20          OD;
21          RETURN;
22      END op;
23
24      GRANT op;
25
26  END general_order_arithmetic;
```

5. 按位相加并检查其结果

```
1   add_bit_by_bit:
2   MODULE
3       adder:
4       PROC (a STRUCT (a2,a1 BOOL) IN, b STRUCT (b2,b1 BOOL) IN)
5               RETURNS (STRUCT (c4,c2,c1 BOOL));
6           DCL c STRUCT (c4,c2,c1 BOOL);
7           DCL k2,x,w,t,s,r BOOL;
8           DO WITH a,b,c;
9               k2 := a1 AND b1;
10              c1 := NOT k2 AND (a1 OR b1);
11              x := a2 AND b2 AND k2;
12              w := a2 OR b2 OR k2;
13              t := b2 AND k2;
14              s := a2 AND k2;
15              r := a2 AND b2;
16              c4 := r OR s OR t;
17              c2 := x OR (w AND NOT c4);
18          OD;
19          RETURN c;
20      END adder;
21      GRANT adder;
22  END add_bit_by_bit;
23
24  exhaustive_checker:
25  MODULE
26      SEIZE adder;
27      DCL a STRUCT (a2,a1 BOOL),
28          b STRUCT (b2,b1 BOOL);
29      SYNMODE res=ARRAY (1:16) STRUCT (c4,c2,c1 BOOL);
30      DCL r INT, results res;
31      DO WITH a,b;
32          r := 0;
33          DO FOR a2 IN BOOL;
34              DO FOR a1 IN BOOL;
35                  DO FOR b2 IN BOOL;
36                      DO FOR b1 IN BOOL;
37                          r+ := 1;
38                          results (r) := adder (a,b);
39                      OD;
40                  OD;
41              OD;
42          OD;
43      OD;
44      ASSERT
45          results=res [[FALSE,FALSE,FALSE ],[FALSE,FALSE,TRUE ],
46                       [FALSE,TRUE,FALSE ] ,[FALSE,TRUE,TRUE ],
47                       [FALSE,FALSE,TRUE ] ,[FALSE,TRUE,FALSE ],
48                       [FALSE,TRUE,TRUE ]  ,[TRUE,FALSE,FALSE ],
49                       [FALSE,TRUE,FALSE ] ,[FALSE,TRUE,TRUE ],
50                       [TRUE,FALSE,FALSE ] ,[TRUE,FALSE,TRUE ],
51                       [FALSE,TRUE,TRUE ]  ,[TRUE,FALSE,FALSE ],
52                       [TRUE,FALSE,TRUE ]  ,[TRUE,TRUE,FALSE ]];
53  END exhaustive_checker;
```

6. 日历游戏

```
playing_with_dates:
MODULE/* from collected algorithms from CACM no. 199 */
    SYNMODE month=SET (jan,feb,mar,apr,may,jun,
                jul,aug,sep,oct,nov,dec);
    NEWMODE date=STRUCT (day INT (1:31), mo month, year INT);

    gregorian_date:
    PROC (julian_day_number INT) RETURNS (date);
        DCL j INT := julian_day_number,
            d,m,y INT;
        j- := 1_721_119;
        y := (4 * j - 1) / 146_097;
        j := 4 * j - 1 - 146_097 * y;
        d := j / 4;
        j := (4 * d + 3) / 1_461;
        d := 4 * d + 3 - 1_461 * j;
        d := (d + 4) / 4;
        m := (5 * d - 3) / 153;
        d := 5 * d - 3 - 153 * m;
        d := (d + 5) / 5;
        y := 100 * y + j;
        IF m<100 THEN m + := 3;
                 ELSE m - := 9;
                      y + := 1;
        FI;
        RETURN [d,month (m+1), y];
    END gregorian_date;

    julian_day_number:
    PROC (d date) RETURNS (INT);
        DCL c,y,m INT;
        DO WITH d;
            m := NUM (mo)+1;
            IF m>2 THEN m - := 3;
                   ELSE m + := 9;
                        year - := 1;
            FI;
            c := year/100;
            y := year-100*c;
            RETURN (146_097*c)/4+(1_461*y)/4
                   +(153+m+c)/5+day+1_721_119;
        OD;
    END julian_day_number;
    GRANT gregorian_date, julian_day_number;
END playing_with_dates;

test:
MODULE
    SEIZE gregorian_date, julian_day_number;
    ASSERT julian_day_number ([ 10,dec,1979 ])=julian_day_number
           (gregorian_date(julian_day_number([ 10,dec,1979 ])));
END test;
```

7. 罗马数字

```
1   Roman:
2   MODULE
3       SEIZE n,rn;
4       GRANT convert;
5       convert:
6       PROC () EXCEPTIONS (string_too_small);
7           DCL r INT := 0;
8           DO WHILE n>=1_000;
9               rn(r) := 'M';
10              n - := 1_000;
11              r + := 1;
12          OD;
13          IF n>500 THEN rn(r) := 'D';
14                        n - := 500;
15                        r + := 1;
16          FI;
17          DO WHILE n>=100;
18              rn(r) := 'C';
19              n - := 100;
20              r + := 1;
21          OD;
22          IF n>=50 THEN rn(r) := 'L';
23                        n - := 50;
24                        r + := 1;
25          FI;
26          DO WHILE n>=10;
27              rn(r) := 'X';
28              n - := 10;
29              r + := 1;
30          OD;
31          IF n>=5 THEN rn(r) := 'V';
32                       n - := 5;
33                       r + := 1;
34          FI;
35          DO WHILE n>=1;
36              rn(r) := 'I';
37              n - := 1;
38              r + := 1;
39          OD;
40          RETURN;
41      END ON (RANGEFAIL): DO FOR i := 0 TO UPPER (rn);
42                              rn(i) := '.';
43                          OD;
44                          CAUSE string_too_small;
45      END convert;
46  END Roman;
47  test:
48  MODULE
49      SEIZE convert;
50      DCL n INT INIT := 1979;
51      DCL rn CHARS (20) INIT := (20)' ';
52      GRANT n,rn;
53      convert ();
54      ASSERT rn="MDCCCCLXXVIIII"//(6)' ';
55  END test;
```

8. 任意长字符串内的字母计数

```
letter_count:
MODULE
    SEIZE max;
    DCL letter POWERSET CHAR INIT := ['A' : 'Z'];
    count:
    PROC (input ROW CHARS (max) IN, output ARRAY ('A':'Z') INT OUT);
        output := [(ELSE) : 0];
        DO FOR i := 0 TO UPPER (input ->);
            IF input -> (i) IN letter
                THEN
                    output (input -> (i)) + := 1;
            FI;
        OD;
    END count;
    GRANT count;
END letter_count;
test:
MODULE
    SYNMODE results=ARRAY ('A':'Z')INT;
    DCL c CHARS (10) INIT := "A-B<ZAA9K' ";
    DCL output results;
    SYN max=10_000;
    GRANT max;
    SEIZE count;
    count (-> c,output);
    ASSERT output=results [('A') : 3,('B','K','Z') : 1, (ELSE) : 0];
END test;
```

9. 素数

```
prime:
MODULE

    SYN max = H'7FFF;
    NEWMODE number_list =POWERSET INT (2:max);
    SYN empty = number_list [ ];
    DCL sieve number_list INIT := [ 2:max ],
        primes number_list INIT := empty;
    GRANT primes;
    DO WHILE sieve/=empty;
        primes OR := [MIN (sieve)];
        DO FOR j := MIN (sieve) BY MIN (sieve) TO max;
            sieve - := [j];
        OD;
    OD;
END prime;
```

10. 用两种方式实现栈，对用户透明

```
stack: MODULE
    NEWMODE element =STRUCT (a INT, b BOOL);
    stacks_1:
    MODULE
```

```
5           SEIZE element;
6           SYN max=10_000,min=1;
7           DCL stack ARRAY (min : max) element,
8                 stackindex INT INIT := min;
9
10          push:
11          PROC (e element) EXCEPTIONS (overflow);
12                IF stackindex=max
13                   THEN CAUSE overflow;
14                FI;
15                stackindex + := 1;
16                stack (stackindex) := e;
17                RETURN;
18          END push;
19
20          pop:
21          PROC () EXCEPTIONS (underflow);
22                IF stackindex=min
23                   THEN CAUSE underflow;
24                FI;
25                stackindex - := 1;
26                RETURN;
27          END pop;
28
29          elem:
30          PROC (i INT) RETURNS (element LOC) EXCEPTIONS (bounds);
31                IF i<min OR i>max
32                   THEN CAUSE bounds;
33                FI;
34                RETURN stack (i);
35          END elem;
36
37          GRANT push,pop,elem;
38    END stacks_1;
39    stacks_2:
40    MODULE
41          SEIZE element;
42          NEWMODE cell=STRUCT (pred,succ REF cell,info element);
43          DCL p,last,first REF cell INIT := NULL;
44
45          push:
46          PROC (e element) EXCEPTIONS (overflow);
47              p := ALLOCATE (cell) ON
48                                   (ALLOCATEFAIL) : CAUSE overflow;
49                                   END;
50              IF last=NULL
51                 THEN first := p;
52                      last := p;
53                 ELSE last ->. succ := p;
54                      p ->. pred := last;
55                      last := p;
56              FI;
57              last ->. info := e;
58              RETURN;
59          END push;
60
61          pop:
62          PROC () EXCEPTIONS (underflow);
63              IF last=NULL
64                 THEN CAUSE underflow;
65              FI;
```

```
66              p := last;
67              last := last ->. pred;
68              IF last = NULL
69                  THEN first := NULL;
70                  ELSE last ->. succ := NULL;
71              FI;
72              TERMINATE(p);
73              RETURN;
74          END pop;
75
76          elem:
77          PROC (i INT) RETURNS (element LOC) EXCEPTIONS (bounds);
78              IF first=NULL
79                  THEN CAUSE bounds;
80              FI;
81              p := first;
82              DO FOR j := 2 TO i;
83                  IF p ->. succ=NULL
84                      THEN CAUSE bounds;
85                  FI;
86                  p := p ->. succ;
87              OD;
88              RETURN p ->. info;
89          END elem;
90
91          /* GRANT push,pop,elem; */
92      END stacks_2;
93  END stack;
```

11. 国际象棋游戏程序片断

```
1   chess_fragments:
2   MODULE
3       NEWMODE piece=STRUCT (color SET (white,black),
4                             kind SET (pawn,rook,knight,bishop,queen,king));
5       NEWMODE column=SET (a,b,c,d,e,f,g,h);
6       NEWMODE line=INT (1 : 8);
7       NEWMODE square=STRUCT (status SET (occupied,free),
8                              CASE status OF
9                              (occupied) : p piece,
10                             (free) :
11                             ESAC);
12      NEWMODE board=ARRAY (line) ARRAY (column) square;
13      NEWMODE move=STRUCT (lin_1,lin_2 line,
14                           col_1,col_2 column);
15
16      initialise:
17      PROC (bd board INOUT);
18          bd := [ (1): [(a,h):   [.status: occupied, .p : [white,rook]],
19                       (b,g):   [.status: occupied, .p : [white,knight]],
20                       (c,f):   [.status: occupied, .p : [white,bishop]],
21                       (d):     [.status: occupied, .p : [white,queen]],
22                       (e):     [.status: occupied, .p : [white,king]]],
23                  (2): [(ELSE):[.status: occupied, .p : [white,pawn]]],
24                  (3:6):[(ELSE):[.status: free]],
25                  (7): [(ELSE):[.status: occupied, .p : [black,pawn]]],
26                  (8): [(a,h):   [.status: occupied, .p : [black,rook]],
27                       (b,g):   [.status: occupied, .p : [black,knight]],
```

```
28                  (c,f):    [.status: occupied, .p : [black,bishop]],
29                  (d):      [.status: occupied, .p : [black,queen]],
30                  (e):      [.status: occupied, .p : [black,king]]]
31              ];
32          RETURN;
33      END initialise;
34      register_move:
35      PROC (b board LOC,m move) EXCEPTIONS (illegal);
36          DCL starting square LOC := b (m.lin_1)(m.col_1),
37              arriving square LOC := b (m.lin_2)(m.col_2);
38          DO WITH m;
39              IF starting.status=free THEN CAUSE illegal; FI;
40              IF arriving.status/=free THEN
41                  IF arriving.p.kind=king THEN CAUSE illegal; FI;
42              FI;
43              CASE starting.p.kind, starting.p.color OF
44                  (pawn),(white):
45                  IF col_1 = col_2 AND (arriving.status/=free
46                      OR NOT (lin_2=lin_1+1 OR lin_2=lin_1+2 AND lin_2=2))
47                      OR (col_2=PRED (col_1) OR col_2=SUCC (col_1))
48                      AND arriving.status=free THEN CAUSE illegal; FI;
49                      IF arriving.status/=free THEN
50                          IF arriving.p.color=white THEN CAUSE illegal; FI; FI;
51                  (pawn),(black):
52                  IF col_1=col_2 AND (arriving.status/=free
53                      OR NOT (lin_2=lin_1-1 OR lin_2=lin_1-2 AND lin_1=7))
54                      OR (col_2=PRED (col_1) OR col_2=SUCC (col_1))
55                      AND arriving.status=free THEN CAUSE illegal; FI;
56                      IF arriving.status/=free THEN
57                          IF arriving.p.color=black THEN CAUSE illegal; FI; FI;
58                  (rook),(*):
59                  IF NOT ok_rook (b,m)
60                      THEN CAUSE illegal;
61                  FI;
62                  (bishop),(*):
63                  IF NOT ok_bishop (b,m)
64                      THEN CAUSE illegal;
65                  FI;
66                  (queen),(*):
67                  IF NOT ok_rook (b,m) AND NOT ok_bishop (b,m)
68                      THEN CAUSE illegal;
69                  FI;
70                  (knight),(*):
71                  IF ABS (ABS (NUM (col_2)-NUM (col_1))
72                          -ABS (lin_2- lin_1)) /= 1
73                      OR ABS (NUM (col_2)-NUM (col_1))
74                          +ABS (lin_2- lin_1) =/ 3 THEN CAUSE illegal; FI;
75                  IF arriving.status/=free THEN
76                          IF arriving.p.color=starting.p.color THEN
77                          CAUSE illegal; FI; FI;
78                  (king),(*):
79                  IF ABS (NUM (col_2)-NUM (col_1)) > 1
80                      OR ABS (lin_2- lin_1) > 1
81                      OR lin_2=lin_1 AND col_2=col_1 THEN CAUSE illegal; FI;
82                  IF arriving.status/=free THEN
83                      IF arriving.p.color=starting.p.color THEN
84                      CAUSE illegal; FI; FI;/* checking king moving to check not implemented */
85              ESAC;
86          OD;
87          arriving := starting;
88          starting := [.status:free];
```

```
89          RETURN;
90      END register_move;
91      ok_rook:
92      PROC (b board,m move) RETURNS (BOOL);
93          DCL starting square := b (m.lin_1)(m.col_1),
94              arriving square := b (m.lin_2)(m.col_2);
95
96          DO WITH m;
97              IF NOT (col_2=col_1 OR lin_1=lin_2) THEN RETURN FALSE; FI;
98              IF arriving.status/=free THEN
99                  IF arriving.p.color=starting.p.color THEN;
100                 RETURN FALSE; FI; FI;
101             IF col_1=col_2
102                 THEN IF lin_1<lin_2
103                         THEN DO FOR lin := lin_1+1 TO lin_2-1;
104                                 IF b (lin)(col_1).status/=free
105                                     THEN RETURN FALSE;
106                                 FI;
107                             OD;
108                         ELSE DO FOR lin := lin_1-1 DOWN TO lin_2+1;
109                                 IF b (lin)(col_1).status/=free
110                                     THEN RETURN FALSE;
111                                 FI;
112                             OD;
113                     FI;
114                 ELSIF col_1<col_2
115                         THEN DO FOR col := SUCC (col_1) TO PRED (col_2);
116                                 IF b (lin_1)(col).status/=free
117                                     THEN RETURN FALSE;
118                                 FI;
119                             OD;
120                         ELSE DO FOR col := SUCC (col_2) DOWN TO PRED (col_1);
121                                 IF b (lin_1)(col).status/=free
122                                     THEN RETURN FALSE;
123                                 FI;
124                             OD;
125             FI;
126             RETURN TRUE;
127         OD;
128     END ok_rook;
129     ok_bishop:
130     PROC (b board,m move) RETURNS (BOOL);
131         DCL starting square := b (m.lin_1)(m.col_1),
132             arriving square := b (m.lin_2)(m.col_2),
133             col column;
134
135         DO WITH m;
136             CASE lin_2>lin_1,col_2>col_1 OF
137                 (TRUE),(TRUE): col := col_1;
138                             DO FOR lin := lin_1+1 TO lin_2-1;
139                                     col := SUCC (col);
140                                     IF b (lin)(col).status/=free
141                                         THEN RETURN FALSE;
142                                     FI;
143                             OD;
144                             IF SUCC (col)/=col_2
145                                     THEN RETURN FALSE;
146                             FI;
147                 (TRUE),(FALSE): col := col_1;
148                             DO FOR lin := lin_1+1 TO lin_2-1;
149                                     col := PRED (col);
```

```
150                                     IF b (lin)(col).status/=free
151                                         THEN RETURN FALSE;
152                                     FI;
153                                 OD;
154                                 IF PRED (col)/=col_2
155                                     THEN RETURN FALSE;
156                                 FI;
157                             (FALSE),(TRUE): col := col_1;
158                                 DO FOR lin := lin_1-1 DOWN TO lin_2+1;
159                                     col := SUCC (col);
160                                     IF b (lin)(col).status/=free
161                                         THEN RETURN FALSE;
162                                     FI;
163                                 OD;
164                                 IF SUCC (col)/=col_2
165                                     THEN RETURN FALSE;
166                                 FI;
167                             (FALSE),(FALSE): col := col_1;
168                                 DO FOR lin := lin_1-1 DOWN TO lin_2+1;
169                                     col := PRED (col);
170                                     IF b (lin)(col).status/=free
171                                         THEN RETURN FALSE;
172                                     FI;
173                                 OD;
174                                 IF PRED (col)/=col_2
175                                     THEN RETURN FALSE;
176                                 FI;
177                 ESAC;
178                 IF arriving.status=free THEN RETURN TRUE;
179                 ELSE RETURN arriving.p.color/=starting.p.color; FI;
180             OD;
181         END ok_bishop;
182         END chess_fragments;
```

12. 建立并加工一个循环链表

```
1   circular_list:
2   MODULE
3       handle_list:
4       MODULE
5           GRANT insert, remove, node;
6           NEWMODE node=STRUCT (pred, suc REF node, value INT);
7           DCL pool ARRAY (1:1000)node;
8           DCL head node := (: NULL,NULL,0 :);
9
10          insert: PROC (new node);
11              /* insert actions */
12          END insert;
13
14          remove: PROC ();
15              /* remove actions */
16          END remove;
17
18          initialize_list:
19          BEGIN
20              DCL last REF node := ->head;
21              DO FOR new IN pool;
22                  new.pred := last;
```

```
23                  last->.suc := ->new;
24                  last := ->new;
25                  new.value := 0;
26              OD;
27              head.pred := last;
28              last->.suc := ->head;
29          END initialize_list;
30
31      END handle_list;
32      manipulate:
33      MODULE
34          SEIZE node, remove, insert;
35          DCL node_a node := (: NULL,NULL,536 :);
36          remove();
37          remove();
38          insert(node_a);
39      END manipulate;
40  END circular_list;
```

13. 用于管理对共享资源竞争访问的一个区域

```
1   allocate_resources:
2   REGION
3       GRANT allocate, deallocate;
4       NEWMODE resource_set = INT (0:9);
5       DCL allocated ARRAY (resource_set)BOOL := (: (resource_set): FALSE :);
6       DCL resource_freed EVENT;
7
8       allocate:
9       PROC() RETURNS (resource_set);
10          DO FOR EVER;
11              DO FOR i IN resource_set;
12                  IF NOT allocated(i)
13                      THEN
14                          allocated(i) := TRUE;
15                          RETURN i;
16                  FI;
17              OD;
18              DELAY resource_freed;
19          OD;
20      END allocate;
21
22      deallocate:
23      PROC (i resource_set);
24          allocated(i) := FALSE;
25          CONTINUE resource_freed;
26      END deallocate;
27
28  END allocate_resources;
```

14. 对电话总机的呼叫进行排队

```
1   switchboard:
2   MODULE
3       /* This example illustrates a switchboard which queues incoming calls
4          and feeds them to the operator at an even rate. Every time the
```

```
5          operator is ready one and only one call is let through. This is
6          handled by a call distributor which lets calls through at fixed
7          intervals. If the operator is not ready or there are other calls
8          waiting, a new call must queue up to wait for its turn. */
9       DCL operator_is_ready,
10            switch_is_closed EVENT;
11
12      call_distributor:
13      PROCESS ();
14          wait:
15          PROC (x INT);
16              /*some wait action*/
17          END wait;
18          DO FOR EVER;
19              wait(10 /*seconds*/);
20              CONTINUE operator_is_ready;
21          OD;
22      END call_distributor;
23
24      call_process:
25      PROCESS ();
26          DELAY CASE
27          (operator_is_ready): /* some actions */ ;
28          (switch_is_closed): DO FOR i IN INT (1:100);
29                                  CONTINUE operator_is_ready;
30                                  /* empty the queue*/
31                              OD;
32          ESAC;
33      END call_process;
34
35      operator:
36      PROCESS ();
37          DCL time INT;
38          DO FOR EVER;
39              IF time = 1700
40                  THEN CONTINUE switch_is_closed;
41              FI;
42          OD;
43      END operator;
44
45      START call_distributor();
46      START operator();
47      DO FOR i IN INT (1:100);
48          START call_process();
49      OD;
50  END switchboard;
```

15. 分配与释放一组资源

```
definitions:
MODULE
SIGNAL
    acquire,
    release=(INSTANCE),
    congested,
    ready,
    advance,
    readout=(INT);
    GRANT ALL;
END definitions;
counter_manager:
MODULE
/* To illustrate the use of signals and the receive case, (buffers
    might have been used instead) we will look at an example where an
    allocator manages a set of resources, in this case a set of
    counters. The module is part of a larger system where there are
    users, that can request the services of the counter_manager. The
    module is made to consist of two process definitions, one for the
    allocation and one for the counters. Initiate and terminate
    are internal signals sent from the allocator
    to the counters. All the other signals are external, being sent
    from or to the users. */

    SEIZE/* external signals */
        acquire, release, congested,ready,advance,readout;
    SIGNAL initiate = (INSTANCE),
            terminate;
    allocator:
    PROCESS ();
        NEWMODE no_of_counters = INT (1:100);
    DCL counters ARRAY (no_of_counters)
                        STRUCT (counter INSTANCE,status SET (busy,idle));
    DO FOR each IN counters;
        each := (: START counter(), idle :);
    OD;
    DO FOR EVER;
    BEGIN
        DCL user INSTANCE;
        await_signals:
        RECEIVE CASE SET user;
        (acquire):
            DO FOR each IN counters;
                DO WITH each;
                    IF status = idle
                        THEN
                            status := busy;
                            SEND initiate (user) TO counter;
                            EXIT await_signals;
                    FI;
                OD;
            OD;
            SEND congested TO user;
        (release IN this_counter):
            SEND terminate TO this_counter;
            find_counter:
            DO FOR each IN counters;
```

```
58                  DO WITH each;
59                      IF this_counter = counter
60                          THEN
61                              status := idle;
62                              EXIT find_counter;
63                      FI;
64                  OD;
65              OD find_counter;
66          ESAC await_signals;
67      END;
68      OD;
69      END allocator;
70      counter:
71      PROCESS ();
72          DO FOR EVER;
73          BEGIN
74              DCL user INSTANCE,
75                  count INT := 0;
76              RECEIVE CASE
77                  (initiate IN received_user):
78                      SEND ready TO received_user;
79                      user := received_user;
80              ESAC;
81              work_loop:
82              DO FOR EVER;
83                  RECEIVE CASE
84                      (advance): count + := 1;
85                      (terminate):
86                          SEND readout(count) TO user;
87                          EXIT work_loop;
88                  ESAC;
89              OD work_loop;
90          END;
91          OD;
92      END counter;
93      START allocator();
94  END counter_manager;
```

16. 使用缓冲区实现对一组资源的分配与释放

```
1
2
3   user_world:
4   MODULE
5   /* This example is the same as no.15 except that buffers are
6       used for communication in stead of signals.
7       The main difference is that processes are now identified
8       by means of references to local message buffers rather than
9       by instance values. There is one message buffer declared
10      local to each process. There is one set of message types
11      for each process definition. When started each process must
12      identify its buffer address to the starting process.
13      The user_world module sketches some of the environment in
14      which the counter_manager is used. */
15
16  SEIZE allocator;
17  GRANT user_buffers,user_messages,
18            allocator_messages, allocator_buffers,
```

```
19          counter_messages, counters_buffers;
20  NEWMODE
21      user_messages =
22          STRUCT (type SET (congested, ready,
23                      readout, allocator_id),
24                              CASE type OF
25                                  (congested) : ,
26                                  (ready) : counter REF counters_buffers,
27                                  (readout) : count INT,
28                                  (allocator_id): allocator REF allocator_buffers
29                              ESAC),
30      user_buffers = BUFFER (1) user_messages,
31      allocator_messages =
32          STRUCT (type SET (acquire, release, counter_id),
33                      CASE type OF
34                          (acquire) : user REF user_buffers,
35                          (release,
36                          counter_id): counter REF counters_buffers
37                      ESAC),
38      allocator_buffers = BUFFER (1) allocator_messages,
39      counter_messages =
40          STRUCT (type SET (initiate, advance, terminate),
41                      CASE type OF
42                          (initiate) : user REF user_buffers,
43                          (advance,
44                          terminate):
45                      ESAC),
46      counters_buffers = BUFFER (1) counter_messages;
47  DCL user_buffer user_buffers,
48        allocator_buf REF allocator_buffers,
49        counter_buf REF counters_buffers;
50  START allocator(->user_buffer);
51  allocator_buf := (RECEIVE user_buffer).allocator;
52  END user_world;
53  counter_manager:
54  MODULE
55  SEIZE user_buffers,user_messages,
56          allocator_messages, allocator_buffers,
57          counter_messages, counters_buffers;
58  GRANT allocator;
59
60  allocator:
61  PROCESS (starter REF user_buffers);
62      DCL allocator_buffer allocator_buffers;
63      NEWMODE no_of_counters = INT (1:10);
64      DCL counters ARRAY (no_of_counters)
65                              STRUCT (counter REF counters_buffers,
66                                      status SET (busy, idle)),
67          message allocator_messages;
68      SEND starter->([allocator_id, ->allocator_buffer]);
69      DO FOR each IN counters;
70          START counter(->allocator_buffer);
71          each := [(RECEIVE allocator_buffer).counter, idle];
72      OD;
73      DO FOR EVER;
74          BEGIN
75          DCL user REF user_buffers;
76          message := RECEIVE allocator_buffer;
77          handle_messages:
78          CASE message.type OF
79          (acquire):
80              user := message.user;
```

```
81              DO FOR each IN counters;
82                  DO WITH each;
83                      IF status= idle
84                          THEN status := busy;
85                                  SEND counter->([initiate, user]);
86                                  EXIT handle_messages;
87                      FI;
88                  OD;
89              OD;
90              SEND user->([congested]);
91          (release):
92              SEND message.counter->([terminate]);
93              find_counter:
94              DO FOR each IN counters;
95                  DO WITH each;
96                      IF message.counter = counter
97                          THEN status := idle;
98                                  EXIT find_counter;
99                      FI;
100                 OD;
101             OD find_counter;
102         (counter_id): ;
103         ESAC handle_messages;
104         END;
105     OD;
106 END allocator;
107 counter:
108 PROCESS (starter REF allocator_buffers);
109     DCL counter_buffer counters_buffers;
110     SEND starter->([counter_id, ->counter_buffer]);
111     DO FOR EVER;
112         BEGIN
113             DCL user REF user_buffers,
114                 count INT := 0,
115                 message counter_messages;
116             message := RECEIVE counter_buffer;
117             CASE message.type OF
118                 (initiate): user := message.user;
119                         SENL user->([ready, ->counter_buffer]);
120                 ELSE/* some error action */
121             ESAC;
122         work_loop:
123         DO FOR EVER;
124             message := RECEIVE counter_buffer;
125             CASE message.type OF
126                 (advance) : count + := 1;
127                 (terminate):SEND user->([readout, count]);
128                         EXIT work_loop;
129                 ELSE/* some error action */
130             ESAC;
131         OD work_loop;
132     END;
133     OD;
134 END counter;
135 END counter_manager;
```

17. 串扫描程序之一

```
1   string_scanner1: /* This program implements strings by means
2                       of packed arrays of characters. */
3   MODULE
4       SYN
5           blanks ARRAY (0:9)CHAR PACK = [(*):' ' ], linelength = 132;
6       SYNMODE
7           stringptr = ROW ARRAY (lineindex)CHAR PACK,
8           lineindex = INT (0:linelength-1);
9
10  scanner:
11  PROC (string stringptr, scanstart lineindex INOUT,
12          scanstop lineindex, stopset POWERSET CHAR)
13          RETURNS (ARRAY (0:9)CHAR PACK);
14      DCL count INT := 0,
15           res ARRAY (0:9)CHAR PACK := blanks;
16      DO
17          FOR c IN string->(scanstart:scanstop)
18          WHILE NOT (c IN stopset);
19              count + := 1;
20      OD;
21      IF count>0
22          THEN
23              IF count>10
24                  THEN
25                      count := 10;
26              FI;
27              res(0:count-1) := string->(scanstart:scanstart+count-1);
28      FI;
29      RESULT res;
30      IF scanstart+count < scanstop
31          THEN
32              scanstart := scanstart+count+1;
33      FI;
34  END scanner;
35
36  GRANT scanner;
37
38  END string_scanner1;
```

18. 串扫描程序之二

```
string_scanner2: /* This example is the same as no.17 but it uses
                    character string instead of packed arrays */
MODULE
    SYN
        blanks = (10)' ', linelength = 132;
    SYNMODE
        stringptr = ROW CHARS (linelength),
        lineindex = INT (0:linelength-1);

    scanner:
        PROC (string stringptr, scanstart lineindex INOUT,
              scanstop lineindex, stopset POWERSET CHAR)
              RETURNS (CHARS (10));
            DCL count INT := 0;
            DO FOR i := scanstart TO scanstop
                WHILE NOT (string->(i) IN stopset);
                    count + := 1;
            OD;
            IF count>0
                THEN
                    IF count>=10
                        THEN
                            RESULT string->(scanstart UP 10);
                        ELSE
                            RESULT string->(scanstart:scanstart+count-1)
                                    //blanks(count:9);
                    FI;
                ELSE
                    RESULT blanks;
            FI;
            IF scanstart+count < scanstop
                THEN
                    scanstart := scanstart+count+1;
            FI;
        END scanner;

    GRANT scanner;

END string_scanner2;
```

19. 从双链表中删除一个节点

```
queue: MODULE
    SYNMODE info=INT;
    queue_removal:
    MODULE
        SEIZE info;
        GRANT remove;
        remove:
        PROC (p PTR) RETURNS (info) EXCEPTIONS (EMPTY);
            /* This procedure removes the item referred to
               by p from a queue and returns the information
               contents of that queue element */
            SYNMODE element = STRUCT (
                                i info POS (0,8:31),
                                prev PTR POS (1,0:15),
                                next PTR POS (1,16:31));
            DCL x REF element LOC := element(p), prev, next PTR;
            prev := x->.prev;
            next := x->.next;
            x->.prev, x->.next := NULL;
            RESULT x->.i;
            p := prev;
            x->.next := next;
            p := next;
            x->.prev := prev;
        END remove;
    END queue_removal;
END queue;
```

20. 加工文件的记录

```
read_modify_write:
MODULE

  /* this example indicates how the CHILL i/o concepts can be used */
  /* to write an application where a record of a random accessible  */
  /* file can be updated or added if not yet in use                 */

  NEWMODE
    index_set = INT (1:1000),
    record_type = STRUCT (
                          free  BOOL,
                          count INT,
                          name  CHARS (20));
```

```
14
15    DCL
16      curindex          index_set,
17      file_association  ASSOCIATION,
18      record_file       ACCESS (index_set) record_type,
19      record_buffer     record_type;
20
21    ASSOCIATE (file_association,"DSK:RECORDS.DAT");    /* create association */
22    CONNECT (record_file,file_association,READWRITE);  /* connect to file    */
23    curindex := 123;                                   /* position record    */
24    READRECORD (record_file,curindex,record_buffer);   /* read the record    */
25      IF record_buffer.free                            /* if record is free  */
26        THEN                                           /* the claim and      */
27          record_buffer.free := FALSE                  /* initialize it      */
28          record_buffer.count := 0;
29          record_buffer.name := "CHILL I/O concept   ";
30      FI;
31    record_buffer.count + := 1;                        /* increment its count*/
32    WRITERECORD (record_file, curindex, record_buffer); /* write the record   */
33    DISSOCIATE (file_association);                     /* end the association*/
34
35  END read_modify_write;
```

21. 合并两个文件

```
1   merge_sorted_files:
2   MODULE
3
4     /* this example shows how two sorted files can be merged into one */
5     /* new sorted file, where the field 'key' is used for sorting     */
6     /* the old sorted files are deleted after the merging has been done */
7
8     NEWMODE
9       record_type = STRUCT (
10                              key   INT,
11                              name CHARS (50));
12
13    DCL
14      flag       BOOL,
15      infiles    ARRAY (BOOL) ACCESS record_type,
16      outfile    ACCESS record_type,
17      buffers    ARRAY (BOOL) record_type,
18      innames    ARRAY (BOOL) CHARS (10) INIT := ["FILE.IN.1 ","FILE.IN.2 "],
19      outname    CHARS (10) INIT := "FILE.OUT  ",
20      inassocs   ARRAY (BOOL) ASSOCIATION,
21      outassoc   ASSOCIATION;
22
23    /* associate both sorted input files, connect an access to them for input */
24    /* and read their first record into a buffer                              */
25
26    DO
27      FOR curfile IN infiles,
28          curbuffer IN buffers,
29          curassoc IN inassocs,
30          curname IN innames;
31        CONNECT (curfile, ASSOCIATE (curassoc,curname), READONLY);
32        READRECORD (curfile, curbuffer);
33    OD;
```

```
34
35    /* associate the output file, create a file for the association */
36    /* and connect an access to it for output                      */
37
38    ASSOCIATE (outassoc,outname);
39    CREATE (outassoc);
40    CONNECT (outfile, outassoc, WRITEONLY);
41    merge_files:
42      DO FOR EVER
43
44        /* determine which file, if any at all, to process next*/
45        /* 'flag' indicates the file                           */
46
47        CASE OUTOFFILE (infiles(FALSE)),OUTOFFILE (infiles(TRUE)) OF
48          (TRUE),  (TRUE):                    /* both files are empty */
49                   EXIT merge_files;
50          (TRUE),  (FALSE):                   /* one file is empty    */
51                   flag := TRUE;
52          (FALSE), (TRUE):                    /* one file is empty    */
53                   flag := FALSE;
54          (FALSE), (FALSE):                   /* no file is empty     */
55                   flag := buffers(FALSE).key>buffers(TRUE).key;
56        ESAC;
57
58          /* output the buffer which currently contains a record with the */
59          /* smallest value for 'key', fill the buffer with a new record  */
60
61          WRITERECORD (outfile,buffers(flag));
62          READRECORD (infiles(flag), buffers(flag));
63    OD merge_files;
64
65    /* delete the input files and close the output file */
66
67    DO
68      FOR curassoc IN inassocs;
69        DELETE (curassoc);                    /* delete the file          */
70        DISSOCIATE (curassoc);                /* and terminate association */
71    OD;
72    DISSOCIATE (outassoc);                    /* disconnect and terminate */
73
74  END merge_sorted_files;
```

22. 读一个变长记录文件

```
1   variable_length_records:
2   MODULE
3
4     /* This example shows how a file which consists of variable length  */
5     /* records can be treated.                                          */
6     /* The file consists of a number of strings of varying length; the  */
7     /* algorithm will read a string, allocate an appropriate location   */
8     /* for it, and put the reference to this location into a push down list */
9
10    NEWMODE
11      string = CHARS (80),
12       link_record = STRUCT (
13                              next_record   REF link_record,
14                              string_row    ROW string);
```

```
15
16    DCL
17      pushdownlist    REF link_record INIT := NULL,
18      length          INT (1:80),
19      temporaryrow    ROW string,
20      fileaccess      ACCESS string DYNAMIC,
21      association     ASSOCIATION;
22      filename        CHARS (20) VARYING INIT := "INPUT.DATA";
23    ASSOCIATE (association,filename);                    /* associate the input file    */
24    CONNECT (fileaccess, association, READONLY);         /* connect access for input    */
25    temporaryrow := READRECORD (fileaccess);             /* read the first record       */
26    DO                                                   /* while not end-of-file       */
27      WHILE NOT(OUTOFFILE(fileaccess));
28        pushdownlist := ALLOCATE (link_record,           /* get a new link record       */
29                                  [pushdownlist,NULL ]); /* and initialize it           */
30        length := 1 + UPPER (temporaryrow->);            /* determine length of string  */
31        DO
32          WITH pushdowlist->;                            /* add new string to list      */
33            string_row := ALLOCATE (CHARS (length),      /* allocate space for string   */
34                                    temporaryrow->);     /* and fill it                 */
35        OD;
36        temporaryrow := READRECORD (fileaccess);         /* get next record in file     */
37    OD;
38    DISSOCIATE (association);                            /* end the association         */
39
40  END variable_length_records;
```

23. 说明模块的使用

```
1       /* The examples 23 and 24 are example 8 divided in two pieces. */
2   letter_count:
3   SPEC MODULE
4       /* This is a spec module for the corresponding module in example 8. */
5       SEIZE max;
6       count:
7       PROC (input ROW CHARS (max) IN, output ARRAY ('A':'Z') INT OUT) END;
8       GRANT count;
9   END letter_count;
10  letter_count: REMOTE "example 24";
11  test:
12  MODULE
13      /* This is the module 'test' from example 8.          */
14      /* It can now be piecewise compiled together with     */
15      /* the above spec module                              */
16      SYNMODE results = ARRAY ('A':'Z') INT;
17      DCL c CHARS (10) INIT := "A-B<ZAA9K' ";
18      DCL output results;
19      SYN max = 10_000;
20      GRANT max;
21      SEIZE count;
22      count (-> c, output);
23      ASSERT output = results [('A') : 3, ('B', 'K', 'Z') : 1, (ELSE) : 0 ];
24  END test;
```

24. 上下文的例子

```
CONTEXT
    /* This is a context for the module "letter_count" */
    /* as used in example 23, allowing the piecewise   */
    /* compilation of "letter_count"                   */
    SYN max = 10_000;
FOR
letter_count:
MODULE
    SEIZE max;
    DCL letter POWERSET CHAR INIT := ['A' : 'Z'];
    count:
    PROC (input ROW CHARS (max) IN, output ARRAY ('A':'Z') INT OUT);
        output := [(ELSE) : 0];
        DO FOR i := 0 TO UPPER (input ->);
            IF input -> (i) IN letter THEN
                output (input -> (i)) + := 1;
            FI;
        OD;
    END count;
    GRANT count;
END letter_count;
```

25. 加前缀和远程模块的使用

```
    /* This example uses the module 'stack' from example 27 or 28. */
    /* It shows how prefixes can be used to prevent name clashes.  */
    /* It uses the remote construct to share the source code.      */
char_stack:
MODULE
    SYNMODE element = CHAR;
    GRANT (-> stack ! char) ! ALL;
    stack: SPEC REMOTE "example 29";
    stack: REMOTE      "example 27 or 28";
END char_stack;

int_stack:
MODULE
    SYNMODE element = INT;
    GRANT (-> stack ! int) ! ALL;
    stack: SPEC REMOTE "example 29";
    stack: REMOTE      "example 27 or 28";
END int_stack;
    /* Here 'push', 'pop' and 'element' are visible but  */
    /* with prefixes 'stack ! char' and 'stack ! int' for */
    /* the implementations with element = CHAR and       */
    /* element = INT, respectively.                      */
    /* Below are some possibilities of using the granted */
    /* names inside modules.                             */
MODULE
    SEIZE ALL PREFIXED stack ;
    DCL c CHAR;
    int ! push (123) ;
    char ! push ('a') ;
    int ! pop ( ) ;
    c = char ! elem (1) ;
```

```
32  END;
33
34  MODULE
35      SEIZE (stack ! int -> stack) ! ALL;
36      stack ! push (345) ;
37      stack ! pop ( ) ;
38  END;
```

26. 文本输入输出的使用

```
1   textio:
2   MODULE
3
4       /* This example shows the use of the text i/o features. */
5
6       DCL
7           outfile   ASSOCIATION,
8           output    TEXT (80) DYNAMIC,
9           size      INT := 12345,
10          flag      BOOL := FALSE,
11          set       SET (a,b,c) := b,
12          s1        CHARS (5) := "CHILL",
13          s2        CHARS (5) DYNAMIC := "text";
14
15      ASSOCIATE (outfile,"OUTPUT.DATA");          -- associate the output file
16      CREATE (outfile);                           -- create it
17      CONNECT (output,outfile,WRITEONLY);         -- then connect text location
18      WRITETEXT (output,"%B%/",10);               -- 1010
19      WRITETEXT (output,"%C%/",set);              -- b
20      WRITETEXT (output,"size = %C%/",size);      -- size = 12345
21      WRITETEXT (output,"%CL6%C i/o%/",s1,s2);    -- CHILL text i/o
22      WRITETEXT (output,"flag =%X%C",flag);       -- flag = FALSE
23      size := GETTEXTINDEX (output);              -- 12
24      DISSOCIATE (outfile);
25  END textio;
```

27. 类属栈

```
1       /* This example implements a generic stack. Please */
2       /* note that the element mode has been left out.   */
3       /* The element mode is defined in the surroundings.*/
4       /* The context is a virtually introduced context,  */
5       /* and it has no source.                           */
6   CONTEXT REMOTE FOR
7   stack:
8   MODULE
9       SEIZE element;
10      NEWMODE cell = STRUCT (pred,succ REF cell,info element);
11      DCL p,last,first REF cell INIT := NULL;
12
13      push:
14      PROC (e element) EXCEPTIONS (overflow)
15          p := ALLOCATE (cell) ON (ALLOCATEFAIL): CAUSE overflow; END;
16          IF last = NULL THEN
17              first := p;
18              last := p;
```

```
19          ELSE
20              last -> .succ := p;
21              p -> .pred := last;
22              last := p;
23          FI;
24          last -> .info := e;
25          RETURN;
26      END push;
27
28      pop:
29      PROC () EXCEPTIONS (underflow)
30          IF last = NULL THEN
31              CAUSE underflow;
32          FI;
33          p:= last;
34          last := last -> .pred;
35          IF last = NULL THEN
36              first := NULL;
37          ELSE
38              last -> .succ := NULL;
39          FI;
40          TERMINATE (p);
41          RETURN;
42      END pop;
43
44      elem:
45      PROC (i INT) RETURNS (element LOC) EXCEPTIONS (bounds)
46          IF first = NULL THEN
47              CAUSE bounds;
48          FI;
49          p := first;
50          DO FOR j := 2 TO i;
51              IF p -> .succ = NULL THEN
52                  CAUSE bounds;
53              FI;
54              p := p -> .succ;
55          OD;
56          RETURN p -> .info;
57      END elem;
58
59      GRANT push,pop,elem;
60  END stack;
```

28. 一个抽象数据类型

```
1       /* This example implements the functionality of example 27 */
2       /* demonstrating how an abstract data type can be          */
3       /* implemented in two different ways in CHILL.             */
4   CONTEXT REMOTE FOR
5   stack:
6   MODULE
7       SEIZE element;
8       SYN max = 10_000, min = 1;
9       DCL  stack ARRAY (min : max) element,
10                stackindex INT INIT := min-1;
11      push:
12      PROC (e element) EXCEPTIONS (overflow)
13          IF stackindex = max THEN
```

```
14              CAUSE overflow;
15          FI;
16          stackindex +:= 1;
17          stack(stackindex) := e;
18          RETURN;
19      END push;
20      pop:
21      PROC () EXCEPTIONS (underflow)
22          IF stackindex = min THEN
23              CAUSE underflow;
24          FI;
25          stackindex-:= 1;
26          RETURN;
27      END pop;
28
29      elem:
30      PROC (i INT) RETURNS (element LOC) EXCEPTIONS (bounds)
31          IF i < min OR i > max THEN
32              CAUSE bounds;
33          FI;
34          RETURN stack(i);
35      END elem;
36
37      GRANT push,pop,elem;
38  END stacks;
```

29. 说明模块的例子

```
1       /*This SPEC MODULE defines the interface of example 27 and 28. */
2   stack: SPEC MODULE
3       SEIZE element;
4       push: PROC (e element) EXCEPTIONS (overflow) END;
5       pop: PROC () EXCEPTIONS (underflow) END;
6       elem: PROC (i INT) RETURNS (element LOC) EXCEPTIONS (bounds) END;
7       GRANT push,pop,elem;
8   END stack;
```

附录 E：本建议删去的语言成份

下面叙述的语言特色不是本建议 Z.200 定义的成分，但却是 1984 年批准的建议 Z.200(红皮书卷 Ⅵ.12) 所定义的语言成分。下面将给出那些语言成分的简短的描述，它们的完整定义请参见 Z.200 1984 版本的有关章节，本附录已给出那儿的节号。某个实现系统可以支持这些成分。

1. **释放编译指示**（参见 2.6 节）

释放编译指示用于释放*保留简单名字串*表中的那些**保留**简单名字串，以便能重新定义它们。

2. **整数模式语法**（参见 3.4.2 节）

BIN 是 *INT* 的导出语法。

3. **带孔的集合模式**（参见 3.4.5 节）

集合模式定义了若干命名值或未命名值的集合。当且仅当一个集合模式的**集合元素**名字的个数小于该集合模式的**值的个数**时，该集合模式是**带孔的**集合模式。

4. **过程模式语法**（参见 3.7 节）

不带任选的**保留**简单名字串 **RETURNS** 的*结果说明*是带 **RETURNS** 的*结果说明*的导出语法。

5. **数组模式语法**（参见 3.11.3 节）

保留简单名字串 **ARRAY** 是任选的。

6. **层次结构表示法**（参见 3.11.5 节）

*层次结构模式*是*嵌套结构模式*的导出语法。在层次结构表示法中，域前面带有一个层号。如果某个结构包括其自身是一个结构或结构数组的域，则构成层次型结构，且每个域应带有层号。作为书写嵌套结构模式的一种替代形式，在相应*层次结构模式*中允许在域名字前面写上层号。

7. **映射引用名字**（参见 3.11.6 节）

可使用映射引用名字以实现定义的方式来指明映射。

8. **有基说明**（参见 4.1.4 节）

不带*受限或自由引用单元名字*的有基说明是某个同义模式定义语句的导出语法。带*受限或自由引用单元名字*的有基说明定义了一个或多个访问名字。这些名字用作访问某个单元的替代形式，其方法是间接引用包含在指定的引用单元内的引用值。每当通过所说明的**有基**名字访问一个单元时就执行一次上述间接引用运算。

9. **字符串字面值**（参见 5.2.4.6 节）

字符串字面值由撇号字符分隔。除了可打印的表示外，还可使用十六进制表示。长度为 1 的字符串字面值用作字符字面值。

10. **ADDR 表示法**（参见 5.3.8 节）

ADDR（〈单元〉）是一〉〈单元〉的导出语法。

11. **赋值语法**（参见 6.2 节）

符号＝是符号：＝的导出语法。

12. **情况动作语法（参见 6.4 节）**

情况动作的范围表内可更一般地出现离散模式，而不仅仅是现在的离散模式名字。

13. **步长型循环动作语法（参见 6.5.2 节）**

步长型循环动作的范围枚举内的范围可以更一般地出现离散模式，而不仅仅是现在的离散模式名字。

14. **显式循环计数器（参见 6.5.2 节）**

如果在循环动作所在的可达区内有一个访问名字是**可见的**，且它与该循环动作的某个循环计数器所定义的名字同名，则该循环计数器是**显式的**，否则是**隐式的**。在前一种情况下，该循环计数器的值恰好在该循环非正常终止之前被存入所表示的单元。

制定了区分循环动作的**正常**和**非正常**终止的方法。如果至少有一个循环计数器的求值已指明终止该循环，则出现正常终止。如果当型条件的求值传递 FALSE，或如果因控制转出该循环动作而离开该循环动作，则出现非正常终止。

15. **调用动作语法（参见 6.7 节）**

保留简单名字串 **CALL** 是任选的。带 **CALL** 的调用动作是不带 **CALL** 的调用动作的导出语法。

16. **RECURSEFAIL 异常（参见 6.7 节）**

当某个**非递归的**过程递归地调用自己时引发 *RECURSEFAIL* 异常。

17. **启动动作语法（参见 6.13 节）**

带 **SET** 任选项的启动动作是下列单赋值动作的导出语法：

〈实例单元〉 ::= 〈启动表达式〉

18. **显式值接收名字（参见 6.19 节）**

接收信号情况动作和接收缓冲区情况动作可能引入**值接收**名字。如果在该接收信号情况动作或接收缓冲区情况动作所在的可达区内有一个名字是**可见的**，且它与 **IN** 后面引入的名字之一同名，则该**值接收**名字是**显式的**，否则是**隐式的**。在前一种情况下，所收到的值恰好在执行相应动作语句表之前被存入它所表示的单元。

19. **分程序（参见 8.1 节）**

条件动作、情况动作、循环动作和延迟情况动作未被定义为分程序。

20. **入口语句（参见 8.4 节）**

借助入口语句，过程可以具有多个入口点。这些入口语句被视为附加的过程定义。入口语句内的定义性出现在该入口语句所在的过程可达区内定义了入口点名字。入口点由入口语句的程序正文位置确定。

21. **寄存器名字（参见 8.4 节）**

过程的形参和结果说明内可以指定寄存器说明。在值传递情况下，它表示实参值存于指定的寄存器内；在单元传递情况下，它表示实参单元的（隐藏的）指针存于指定的寄存器中。如果结果说明内出现寄存器说明，则意味着该过程所返送的值或所返送的单元的（隐藏的）指针被存入指定的寄存器中。

22. **弱可见名字和可见性语句（参见 10.2.4.3 节）**

如果在可达区 R 内**弱可见的**名字串 NS 在 R 中**被链接**到一个定义性出现之上，且该定义性出现不为直接包围在 R 内的模体 M 的可达区所包围，则称名字串 NS 是可由模体 M **移入的**。如果在模体 M 的可达区 R

内**弱可见的***名字串* NS 在 R 中**被链接**到由 R 包围的某个*定义性出现*之上，则称 NS 是可由模体 M **移出的**。

23. 利用模体名字移入（参见 10.2.4.5 节）

如果某个*移入语句*内的*前缀更名子句*具有一个*移入后缀*，且该*移入后缀*包含模体*名字串*和 **ALL**，则该*移入语句*与一组*移入语句*等价，对在直接包围该*移入语句*所在的模体的可达区内**强可见的**、且可由上述模体**移入的**每个名字串而言，都存在一个*移入语句*，并且，这些名字串是由该模体*名字*附着的模体移出到直接包围该*移入语句*所在的模体的可达区内的。

24. 预定义简单名字串（参见 C.2 节）

AND、*NOT*、*OR*、*REM*、*MOD*、*THIS* 和 *XOR* 是预定义简单名字串。

附录 F：语法汇总

2 预备知识

〈简单名字串〉 ::=
〈字母〉{〈字母〉 | 〈数字〉 |-}*

〈字母〉 ::=
A|*B*|*C*|*D*|*E*|*F*|*G*|*H*|*I*|*J*|*K*|*L*|*M*
| *N*|*O*|*P*|*Q*|*R*|*S*|*T*|*U*|*V*|*W*|*X*|*Y*|*Z*
| *a*|*b*|*c*|*d*|*e*|*f*|*g*|*h*|*i*|*j*|*k*|*l*|*m*
| *n*|*o*|*p*|*q*|*r*|*s*|*t*|*u*|*v*|*w*|*x*|*y*|*z*

〈数字〉 ::=
0|*1*|*2*|*3*|*4*|*5*|*6*|*7*|*8*|*9*

〈注释〉 ::=
〈带括号注释〉
| 〈行尾注释〉

〈带括号注释〉 ::=
/* 〈字符串〉 */

〈行尾注释〉 ::=
--〈字符串〉〈行尾〉

〈字符串〉 ::=
{〈字符〉}*

〈指示子句〉 ::=
〈〉〈指示〉{,〈指示〉}* 〈〉

〈指示〉 ::=
〈实现指示〉

〈名字〉 ::=
〈名字串〉

〈名字串〉 ::=
〈简单名字串〉
| 〈带前缀名字串〉

〈带前缀名字串〉 ::=
〈前缀〉!〈简单名字串〉

〈前缀〉 ::=
〈简单前缀〉{!〈简单前缀〉}*

〈简单前缀〉 ::=
〈简单名字串〉

〈定义性出现〉 ::=
〈简单名字串〉

〈定义性出现表〉 ::=

〈定义性出现〉{，〈定义性出现〉}*

〈域名字〉::=

〈简单名字串〉

〈域名字定义性出现〉::=

〈简单名字串〉

〈域名字定义性出现表〉::=

〈域名字定义性出现〉{，〈域名字定义性出现〉}*

〈异常名字〉::=

〈简单名字串〉

|〈带前缀名字串〉

〈正文引用名字〉::=

〈简单名字串〉

|〈带前缀名字串〉

3 模式和类

〈模式定义〉::=

〈定义性出现表〉＝〈定义模式〉

〈定义模式〉::=

〈模式〉

〈同义模式定义语句〉::=

SYNMODE〈模式定义〉{，〈模式定义〉}*；

〈新模式定义语句〉::=

NEWMODE〈模式定义〉{，〈模式定义〉}*；

〈模式〉::=

[**READ**]〈非组合模式〉

|[**READ**]〈组合模式〉

〈非组合模式〉::=

〈离散模式〉

|〈幂集模式〉

|〈引用模式〉

|〈过程模式〉

|〈实例模式〉

|〈同步模式〉

|〈输入输出模式〉

|〈计时模式〉

〈离散模式〉::=

〈整数模式〉

|〈布尔模式〉

|〈字符模式〉

|〈集合模式〉

|〈范围模式〉

〈整数模式〉 ::=
〈整数模式名字〉

〈布尔模式〉 ::=
〈布尔模式名字〉

〈字符模式〉 ::=
〈字符模式名字〉

〈集合模式〉 ::=
SET（〈集合表〉）
| 〈集合模式名字〉

〈集合表〉 ::=
〈编号集合表〉
| 〈未编号集合表〉

〈编号集合表〉 ::=
〈编号集合元素〉{，〈编号集合元素〉}*

〈编号集合元素〉 ::=
〈定义性出现〉 = 〈整数字面值表达式〉

〈未编号集合表〉 ::=
〈集合元素〉{，〈集合元素〉}*

〈集合元素〉 ::=
〈定义性出现〉

〈范围模式〉 ::=
〈离散模式名字〉（〈字面值范围〉）
| **RANGE**（〈字面值范围〉）
| **BIN**（〈整数字面值表达式〉）
| 〈范围模式名字〉

〈字面值范围〉 ::=
〈下界〉：〈上界〉

〈下界〉 ::=
〈离散字面值表达式〉

〈上界〉 ::=
〈离散字面值表达式〉

〈幂集模式〉
POWERSET〈成员模式〉
| 〈幂集模式名字〉

〈成员模式〉 ::=
〈离散模式〉

〈引用模式〉 ::=
〈受限引用模式〉
| 〈自由引用模式〉
| 〈行模式〉

〈受限引用模式〉 ::=

REF〈被引用模式〉
| 〈受限引用模式名字〉

〈被引用模式〉::=
〈模式〉

〈自由引用模式〉::=
〈自由引用模式名字〉

〈行模式〉::=
ROW〈串模式〉
| **ROW**〈数组模式〉
| **ROW**〈变体结构模式〉
| 〈行模式名字〉

〈过程模式〉::=
PROC ([〈参数表〉]) [〈结果说明〉]
[**EXCEPTIONS** (〈异常表〉)] [**RECURSIVE**]
| 〈过程模式名字〉

〈参数表〉::=
〈参数说明〉{, 〈参数说明〉}*

〈参数说明〉::=
〈模式〉[〈参数属性〉]

〈参数属性〉::=
IN|**OUT**|**INOUT**|**LOC** [**DYNAMIC**]

〈结果说明〉::=
RETURNS (〈模式〉[〈结果属性〉])

〈结果属性〉::=
[**NONREF**] **LOC** [**DYNAMIC**]

〈异常表〉::=
〈异常名字〉{, 〈异常名字〉}*

〈实例模式〉::=
〈实例模式名字〉

〈同步模式〉::=
〈事件模式〉
| 〈缓冲区模式〉

〈事件模式〉::=
EVENT [(〈事件长度〉)]
| 〈事件模式名字〉

〈事件长度〉::=
〈整数字面值表达式〉

〈缓冲区模式〉::=
BUFFER [(〈缓冲区长度〉)]〈缓冲区元素模式〉
| 〈缓冲区模式名字〉

〈缓冲区长度〉::=

〈整数字面值表达式〉

〈缓冲区元素模式〉::=
〈模式〉

〈输入输出模式〉::=
〈结合模式〉
| 〈访问模式〉
| 〈文本模式〉

〈结合模式〉::=
〈结合模式名字〉

〈访问模式〉::=
ACCESS [(〈下标模式〉)] [〈记录模式〉[**DYNAMIC**]]
| 〈访问模式名字〉

〈记录模式〉::=
〈模式〉

〈下标模式〉::=
〈离散模式〉
| 〈字面值范围〉

〈文本模式〉::=
TEXT (〈文本长度〉) [〈下标模式〉] [**DYNAMIC**]

〈文本长度〉::=
〈整数字面值表达式〉

〈计时模式〉::=
〈时延模式〉
| 〈绝对时钟模式〉

〈时延模式〉::=
〈时延模式名字〉

〈绝对时钟模式〉::=
〈绝对时钟模式名字〉

〈组合模式〉::=
〈串模式〉
| 〈数组模式〉
| 〈结构模式〉

〈串模式〉::=
〈串类型〉(〈串长度〉) [**VARYING**]
| 〈参数化串模式〉
| 〈串模式名字〉

〈参数化串模式〉::=
〈原始串模式名字〉(〈串长度〉)
| 〈参数化串模式名字〉

〈原始串模式名字〉::=
〈串模式名字〉

〈串类型〉::=
BOOLS
| **CHARS**

〈串长度〉::=
〈整数字面值表达式〉

〈数组模式〉::=
ARRAY (〈下标模式〉{,〈下标模式〉}*)
〈元素模式〉{〈元素布局〉}*
| 〈参数化数组模式〉
| 〈数组模式名字〉

〈参数化数组模式〉::=
〈原始数组模式名字〉(〈下标上界〉)
| 〈参数化数组模式名字〉

〈原始数组模式名字〉::=
〈数组模式名字〉

〈下标上界〉
〈离散字面值表达式〉

〈元素模式〉::=
〈模式〉

〈结构模式〉::=
STRUCT (〈域〉{,〈域〉}*)
| 〈参数化结构模式〉
| 〈结构模式名字〉

〈域〉::=
〈固定域〉
| 〈替换域〉

〈固定域〉::=
〈域名字定义性出现表〉〈模式〉[〈域布局〉]

〈替换域〉::=
CASE [〈标志表〉] **OF** 〈变体选择项〉
{,〈变体选择项〉}*
[**ELSE** [〈变体域〉{,〈变体域〉}*]] **ESAC**

〈变体选择项〉::=
[〈情况标号说明〉]:[〈变体域〉{,〈变体域〉}*]

〈标志表〉::=
〈标志域名字〉{,〈标志域名字〉}*

〈变体域〉::=
〈域名字定义性出现表〉〈模式〉[〈域布局〉]

〈参数化结构模式〉::=
〈原始变体结构模式名字〉(〈字面值表达式表〉)
| 〈参数化结构模式名字〉

〈原始变体结构模式名字〉::=
〈变体结构模式名字〉

〈字面值表达式表〉::=
〈离散字面值表达式〉{,〈离散字面值表达式〉}*

〈元素布局〉::=
PACK|**NOPACK**|〈步长〉

〈域布局〉::=
PACK|**NOPACK**|〈定位〉

〈步长〉::=
STEP(〈定位〉[,〈步长大小〉])

〈定位〉::=
POS(〈字〉,〈开始位〉,〈长度〉)
| **POS**(〈字〉[,〈开始位〉[:〈结束位〉]])

〈字〉::=
〈整数字面值表达式〉

〈步长大小〉::=
〈整数字面值表达式〉

〈开始位〉::=
〈整数字面值表达式〉

〈结束位〉::=
〈整数字面值表达式〉

〈长度〉::=
〈整数字面值表达式〉

4 单元及其访问

〈说明语句〉::=
DCL〈说明〉{,〈说明〉}*;

〈说明〉::=
〈单元说明〉
|〈单元等同说明〉

〈单元说明〉::=
〈定义性出现表〉〈模式〉[**STATIC**][〈初始化〉]

〈初始化〉::=
〈可达区初始化〉
|〈生存期初始化〉

〈可达区初始化〉::=
〈赋值号〉〈值〉[〈处理程序〉]

〈生存期初始化〉::=
INIT〈赋值号〉〈常数值〉

〈单元等同说明〉::=

〈定义性出现表〉〈模式〉**LOC** [**DYNAMIC**]
〈赋值号〉〈单元〉[〈处理程序〉]

〈单元〉::=
〈访问名字〉
|〈间接引用的受限引用〉
|〈间接引用的自由引用〉
|〈间接引用行〉
|〈串元素〉
|〈串片〉
|〈数组元素〉
|〈数组片〉
|〈结构域〉
|〈单元过程调用〉
|〈单元内部子程序调用〉
|〈单元转换〉

〈访问名字〉::=
〈单元名字〉
|〈单元等同名字〉
|〈单元枚举名字〉
|〈单元开域名字〉

〈间接引用的受限引用〉::=
〈受限引用原值〉—〉[〈模式名字〉]

〈间接引用的自由引用〉::=
〈自由引用原值〉—〉〈模式名字〉

〈间接引用行〉::=
〈行原值〉—〉

〈串元素〉::=
〈串单元〉(〈开始元素〉)

〈开始元素〉::=
〈整数表达式〉

〈串片〉::=
〈串单元〉(〈左元素〉:〈右元素〉)
|〈串单元〉(〈开始元素〉**UP**〈片大小〉)

〈左元素〉::=
〈整数表达式〉

〈右元素〉::=
〈整数表达式〉

〈片大小〉::=
〈整数表达式〉

〈数组元素〉::=
〈数组单元〉(〈表达式表〉)

〈表达式表〉::=
〈表达式〉{,〈表达式〉}*

〈数组片〉::=
〈数组单元〉(〈下元素〉:〈上元素〉)
|〈数组单元〉(〈首元素〉**UP**〈片大小〉)

〈下元素〉::=
〈表达式〉

〈上元素〉::=
〈表达式〉

〈首元素〉::=
〈表达式〉

〈结构域〉::=
〈结构单元〉·〈域名字〉

〈单元过程调用〉::=
〈单元过程调用〉

〈单元内部子程序调用〉::=
〈单元内部子程序调用〉

〈单元转换〉::=
〈模式名字〉(〈静态模式单元〉)

5 值及其运算

〈同义词定义语句〉::=
SYN〈同义词定义〉{,〈同义词定义〉}*;

〈同义词定义〉::=
〈定义性出现表〉[〈模式〉]=〈常数值〉

〈原值〉::=
〈单元内容〉
|〈值名字〉
|〈字面值〉
|〈多元值〉
|〈值串元素〉
|〈值串片〉
|〈值数组元素〉
|〈值数组片〉
|〈值结构域〉
|〈表达式转换〉
|〈值过程调用〉
|〈值内部子程序调用〉
|〈启动表达式〉
|〈零目运算符〉
|〈带括号表达式〉

〈单元内容〉::=
〈单元〉

〈值名字〉::=
〈同义词名字〉
|〈值枚举名字〉
|〈值开域名字〉
|〈值接收名字〉
|〈通用过程名字〉

〈字面值〉::=
〈整数字面值〉
|〈布尔字面值〉
|〈字符字面值〉
|〈集合字面值〉
|〈空字面值〉
|〈字符串字面值〉
|〈位串字面值〉

〈整数字面值〉::=
〈十进制整数字面值〉
|〈二进制整数字面值〉
|〈八进制整数字面值〉
|〈十六进制整数字面值〉

〈十进制整数字面值〉::=
[{D|d}'] { 〈数字〉 |_}+

〈二进制整数字面值〉::=
{B|b}' {0|1|_}+

〈八进制整数字面值〉::=
{O|o} { 〈八进制数字〉 |_}+

〈十六进制整数字面值〉::=
{H|h}' { 〈十六进制数字〉 |_}+

〈十六进制数字〉::=
〈数字〉 |A|B|C|D|E|F|a|b|c|d|e|f

〈八进制数字〉::=
0|1|2|3|4|5|6|7

〈布尔字面值〉::=
〈布尔字面值名字〉

〈字符字面值〉::=
'〈字符〉|〈控制序列〉'

〈集合字面值〉::=
〈集合元素名字〉

〈空字面值〉::=
〈空字面值名字〉

〈字符串字面值〉::=

”{〈非保留字符〉|〈引号〉|〈控制序列〉}*”

〈引号〉::=
””

〈控制序列〉::=
^ (〈整数字面值表达式〉{,〈整数字面值表达式〉}*)
| ^ 〈非专用字符〉
| ^ ^

〈位串字面值〉::=
〈二进制位串字面值〉
|〈八进制位串字面值〉
|〈十六进制位串字面值〉

〈二进制位串字面值〉::=
{B|b}’{0|1|_}*’

〈八进制位串字面值〉::=
{O|o}’{〈八进制数字〉|_}*’

〈十六进制位串字面值〉::=
{H|h}’{〈十六进制数字〉|_}*’

〈多元组〉::=
[〈模式名字〉](:{〈幂集多元组〉
|〈数组多元组〉|〈结构多元组〉}:)

〈幂集多元组〉::=
[{〈表达式〉|〈范围〉}{,{〈表达式〉|〈范围〉}}*]

〈范围〉::=
〈表达式〉:〈表达式〉

〈数组多元组〉::=
〈无标号数组多元组〉
|〈有标号数组多元组〉

〈无标号数组多元组〉::=
〈值〉{,〈值〉}*

〈有标号数组多元组〉::=
〈情况标号表〉:〈值〉{,〈情况标号表〉:〈值〉}*

〈结构多元组〉::=
〈无标号结构多元组〉
|〈有标号结构多元组〉

〈无标号结构多元组〉::=
〈值〉{,〈值〉}*

〈有标号结构多元组〉::=
〈域名字表〉:〈值〉{,〈域名字表〉:〈值〉}*

〈域名字表〉::=
〈域名字〉{,·〈域名字〉}*

〈值串元素〉::=
〈串原值〉(〈开始元素〉)

〈值串片〉::=
〈串原值〉(〈左元素〉:〈右元素〉)
| 〈串原值〉(〈开始元素〉**UP**〈片大小〉)

〈值数组元素〉::=
〈数组原值〉(〈表达式表〉)

〈值数组片〉::=
〈数组原值〉(〈下元素〉:〈上元素〉)
| 〈数组原值〉(〈首元素〉**UP**〈片大小〉)

〈值结构域〉::=
〈结构原值〉·〈域名字〉

〈表达式转换〉::=
〈模式名字〉(〈表达式〉)

〈值过程调用〉::=
〈值过程调用〉

〈值内部子程序调用〉::=
〈值内部子程序调用〉

〈启动表达式〉::=
START〈进程名字〉([〈实参表〉])

〈零目运算符〉::=
THIS

〈带括号表达式〉::=
(〈表达式〉)

〈值〉::=
〈表达式〉
| 〈未定义值〉

〈未定义值〉::=
*
| 〈未定义同义词名字〉

〈表达式〉::=
〈运算数-0〉
| 〈条件表达式〉

〈条件表达式〉::=
IF〈布尔表达式〉〈则选择项〉〈否则选择项〉**FI**
| **CASE**〈情况选择器表〉**OF**{〈值情况选择项〉}$^{+}$
[**ELSE**〈子表达式〉] **ESAC**

〈则选择项〉::=
THEN〈子表达式〉

〈否则选择项〉::=
ELSE〈子表达式〉
| **ELSIF**〈布尔表达式〉〈则选择项〉
〈否则选择项〉

〈子表达式〉::=
〈表达式〉

〈值情况选择项〉::=
〈情况标号说明〉:〈子表达式〉;

〈运算数-0〉::=
〈运算数-1〉
|〈子运算数-0〉{**OR**|**ORIF**|**XOR**}〈运算数-1〉

〈子运算数-0〉::=
〈运算数-0〉

〈运算数-1〉::=
〈运算数-2〉
|〈子运算数-1〉{**AND**|**ANDIF**}〈运算数-2〉

〈子运算数-1〉::=
〈运算数-1〉

〈运算数-2〉::=
〈运算数-3〉
|〈子运算数-2〉〈运算符-3〉〈运算数-3〉

〈子运算数-2〉::=
〈运算数-2〉

〈运算符-3〉::=
〈关系运算符〉
|〈成员运算符〉
|〈幂集蕴含运算符〉

〈关系运算符〉::=
=|/=|>|>=|<|<=

〈成员运算符〉::=
IN

〈幂集蕴含运算符〉::=
<=|>=|<|>

〈运算数-3〉::=
〈运算数-4〉
|〈子运算数-3〉〈运算符-4〉〈运算数-4〉

〈子运算数-3〉::=
〈运算数-3〉

〈运算符-4〉::=
〈算术加减运算符〉
|〈串并置运算符〉
|〈幂集求差运算符〉

〈算术加减运算符〉::=
+|-

〈串并置运算符〉::=

//

〈幂集求差运算符〉::=

—

〈运算数-4〉::=

〈运算数-5〉

|〈子运算数-4〉〈算术乘除运算符〉〈运算数-5〉

〈子运算数-4〉::=

〈运算数-4〉

〈算术乘除运算符〉::=

*|/|**MOD**|**REM**

〈运算数-5〉::=

[〈单目运算符〉]〈运算数-6〉

〈单目运算符〉::=

—|**NOT**

|〈串重复运算符〉

〈串重复运算符〉::=

(〈整数字面值表达式〉)

〈运算数-6〉::=

〈被引用单元〉

|〈接收表达式〉

|〈原值〉

〈被引用单元〉::=

—〉〈单元〉

〈接收表达式〉::=

RECEIVE〈缓冲区单元〉

6 动作

〈动作语句〉::=

[〈定义性出现〉:]〈动作〉[〈处理程序〉][〈简单名字串〉];

|〈模块〉

|〈说明模块〉

|〈上下文模块〉

〈动作〉::=

〈带括号动作〉

|〈赋值动作〉

|〈调用动作〉

|〈出口动作〉

|〈返回动作〉

|〈结果动作〉

|〈转向动作〉

|〈断言动作〉

| 〈空动作〉
| 〈启动动作〉
| 〈停止动作〉
| 〈延迟动作〉
| 〈继续动作〉
| 〈发送动作〉
| 〈引发动作〉

〈带括号动作〉 ::=
〈条件动作〉
| 〈情况动作〉
| 〈循环动作〉
| 〈begin-end 分程序〉
| 〈延迟情况动作〉
| 〈接收情况动作〉
| 〈计时动作〉

〈赋值动作〉 ::=
〈单赋值动作〉
| 〈多重赋值动作〉

〈单赋值动作〉 ::=
〈单元〉〈赋值号〉〈值〉
| 〈单元〉〈赋值运算符〉〈表达式〉

〈多重赋值动作〉 ::=
〈单元〉{,〈单元〉}$^{+}$〈赋值号〉〈值〉

〈赋值运算符〉 ::=
〈封闭的二目运算符〉〈赋值号〉

〈封闭的二目运算符〉 ::=
OR|**XOR**|**AND**
| 〈幂集求差运算符〉
| 〈算术加减运算符〉
| 〈算术乘除运算符〉
| 〈串并置运算符〉

〈赋值号〉 ::=
:=

〈条件动作〉 ::=
IF 〈布尔表达式〉〈则子句〉[〈否则子句〉] **FI**

〈则子句〉 ::=
THEN 〈动作语句表〉

〈否则子句〉 ::=
ELSE 〈动作语句表〉
| **ELSIF** 〈布尔表达式〉〈则子句〉[〈否则子句〉]

〈情况动作〉 ::=
CASE 〈情况选择器表〉 **OF** [〈范围表〉;] {〈情况选择项〉}$^{+}$

[**ELSE**〈动作语句表〉] **ESAC**

〈情况选择器表〉::=
〈离散表达式〉{,〈离散表达式〉}*

〈范围表〉::=
〈离散模式名字〉{,〈离散模式名字〉}*

〈情况选择项〉::=
〈情况标号说明〉:〈动作语句表〉

〈循环动作〉::=
DO [〈控制部分〉;]〈动作语句表〉 **OD**

〈控制部分〉::=
〈步长型控制〉[〈当型控制〉]
|〈当型控制〉
|〈开域部分〉

〈步长型控制〉::=
FOR {〈迭代〉{,〈迭代〉}* |**EVER**}

〈迭代〉::=
〈值枚举〉
|〈单元枚举〉

〈值枚举〉::=
〈步长枚举〉
|〈范围枚举〉
|〈幂集枚举〉

〈步长枚举〉::=
〈循环计数器〉〈赋值号〉
〈开始值〉[〈步长值〉] [**DOWN**]〈结束值〉

〈循环计数器〉::=
〈定义性出现〉

〈开始值〉::=
〈离散表达式〉

〈步长值〉::=
BY〈整数表达式〉

〈结束值〉::=
TO〈离散表达式〉

〈范围枚举〉::=
〈循环计数器〉[**DOWN**] **IN**〈离散模式名字〉

〈幂集枚举〉::=
〈循环计数器〉[**DOWN**] **IN**〈幂集表达式〉

〈单元枚举〉::=
〈循环计数器〉[**DOWN**] **IN**〈部分组合对象〉

〈部分组合对象〉::=
〈数组单元〉

| 〈数组表达式〉
| 〈串单元〉
| 〈串表达式〉

〈当型控制〉::=
WHILE 〈布尔表达式〉

〈开域部分〉::=
WITH 〈开域控制〉{,〈开域控制〉}*

〈开域控制〉::=
〈结构单元〉
| 〈结构原值〉

〈出口动作〉::=
EXIT 〈标号名字〉

〈调用动作〉::=
〈过程调用〉
| 〈内部子程序调用〉

〈过程调用〉::=
{〈过程名字〉|〈过程原值〉}
([〈实参表〉])

〈实参表〉::=
〈实参〉{,〈实参〉}*

〈实参〉::=
〈值〉
| 〈单元〉

〈内部子程序调用〉::=
〈内部子程序名字〉([〈内部子程序参数表〉])

〈内部子程序参数表〉::=
〈内部子程序参数〉{,〈内部子程序参数〉}*

〈内部子程序参数〉::=
〈值〉
| 〈单元〉
| 〈非保留名字〉[(〈内部子程序参数表〉)]

〈返回动作〉::=
RETURN [〈结果〉]

〈结果动作〉::=
RESULT 〈结果〉

〈结果〉::=
〈值〉
| 〈单元〉

〈转向动作〉::=
GOTO 〈标号名字〉

〈断言动作〉::=

ASSERT〈布尔表达式〉

〈空动作〉::=

〈空〉

〈空〉::=

〈引发动作〉::=

CAUSE〈异常名字〉

〈启动动作〉::=

〈启动表达式〉

〈停止动作〉::=

STOP

〈继续动作〉::=

CONTINUE〈事件单元〉

〈延迟动作〉::=

DELAY〈事件单元〉[〈优先数〉]

〈优先数〉::=

PRIORITY〈整数字面值表达式〉

〈延迟情况动作〉::=

DELAY CASE [**SET**〈实例单元〉[〈优先数〉]; | 〈优先数〉;]

{〈延迟选择项〉}$^{+}$

ESAC

〈延迟选择项〉::=

(〈事件表〉):〈动作语句表〉

〈事件表〉::=

〈事件单元〉{,〈事件单元〉}*

〈发送动作〉::=

〈发送信号动作〉

| 〈发送缓冲区动作〉

〈发送信号动作〉::=

SEND〈信号名字〉[(〈值〉{,〈值〉}*)]

[**TO**〈实例原值〉][〈优先数〉]

〈发送缓冲区动作〉::=

SEND〈缓冲区单元〉(〈值〉)[〈优先数〉]

〈接收情况动作〉::=

〈接收信号情况动作〉

| 〈接收缓冲区情况动作〉

〈接收信号情况动作〉::=

RECEIVE CASE [**SET**〈实例单元〉;]

{〈信号接收选择项〉}$^{+}$

[**ELSE**〈动作语句表〉] **ESAC**

〈信号接收选择项〉::=

(〈信号名字〉[IN〈定义性出现表〉])〈动作语句表〉

〈接收缓冲区情况动作〉::=
RECEIVE CASE [SET〈实例单元〉;]
{〈缓冲区接收选择项〉}+
[ELSE〈动作语句表〉]
ESAC

〈缓冲区接收选择项〉::=
(〈缓冲区单元〉IN〈定义性出现〉):〈动作语句表〉

〈*CHILL* 内部子程序调用〉::=
〈*CHILL* 一般内部子程序调用〉
| 〈*CHILL* 单元内部子程序调用〉
| 〈*CHILL* 值内部子程序调用〉

〈*CHILL* 一般内部子程序调用〉::=
〈结束内部子程序调用〉
| 〈入出一般内部子程序调用〉
| 〈计时一般内部子程序调用〉

〈*CHILL* 单元内部子程序调用〉::=
〈入出单元内部子程序调用〉

〈*CHILL* 值内部子程序调用〉::=
NUM (〈离散表达式〉)
| (*PRED*〈离散表达式〉)
| *SUCC* (〈离散表达式〉)
| *ABS* (〈整数表达式〉)
| *CARD* (〈幂集表达式〉)
| *MAX* (〈幂集表达式〉)
| *MIN* (〈幂集表达式〉)
| *SIZE* ({〈单元〉|〈模式变元〉})
| *UPPER* (〈上下变元〉)
| *LOWER* (〈上下变元〉)
| *LENGTH* (〈长度变元〉)
| 〈分配内部子程序调用〉
| 〈入出值内部子程序调用〉
| 〈时钟值内部子程序调用〉

〈模式变元〉::=
〈模式名字〉
| 〈数组模式名字〉(〈表达式〉)
| 〈串模式名字〉(〈整数表达式〉)
| 〈变体结构模式名字〉(〈表达式表〉)

〈上下变元〉::=
〈数组单元〉
| 〈数组表达式〉
| 〈数组模式名字〉
| 〈串单元〉
| 〈串表达式〉

| 〈串模式名字〉
| 〈离散单元〉
| 〈离散表达式〉
| 〈离散模式名字〉

〈长度变元〉::=
〈串单元〉
| 〈串表达式〉

〈分配内部子程序调用〉::=
GETSTACK (〈模式变元〉[,〈值〉])
| *ALLOCATE* (〈模式变元〉[,〈值〉])

〈结束内部子程序调用〉::=
TERMINATE (〈引用原值〉)

7 输入和输出

〈入出值内部子程序调用〉::=
〈结合属性内部子程序调用〉
| 〈已结合内部子程序调用〉
| 〈访问属性内部子程序调用〉
| 〈读记录内部子程序调用〉
| 〈取文本内部子程序调用〉

〈入出一般内部子程序调用〉::=
〈分离内部子程序调用〉
| 〈修改内部子程序调用〉
| 〈连接内部子程序调用〉
| 〈拆除内部子程序调用〉
| 〈写记录内部子程序调用〉
| 〈文本内部子程序调用〉
| 〈置文本内部子程序调用〉

〈入出单元内部子程序调用〉::=
〈结合内部子程序调用〉
〈结合内部子程序调用〉::=
ASSOCIATE (〈结合单元〉[,〈结合参数表〉])

〈已结合内部子程序调用〉::=
ISASSOCIATED (〈结合单元〉)

〈结合参数表〉::=
〈结合参数〉{,〈结合参数〉}*
〈结合参数〉::=
〈单元〉
| 〈值〉

〈分离内部子程序调用〉::=
DISSOCIATE (〈结合单元〉)

〈结合属性内部子程序调用〉::=

EXISTING （〈结合单元〉）
| *READABLE* （〈结合单元〉）
| *WRITEABLE* （〈结合单元〉）
| *INDEXABLE* （〈结合单元〉）
| *SEQUENCIBLE* （〈结合单元〉）
| *VARIABLE* （〈结合单元〉）

〈修改内部子程序调用〉::=
CREATE （〈结合单元〉）
| *DELETE* （〈结合单元〉）
| *MODIFY* （〈结合单元〉[，〈修改参数表〉]）

〈修改参数表〉::=
〈修改参数〉{，〈修改参数〉}*

〈修改参数〉::=
〈值〉
|〈单元〉

〈连接内部子程序调用〉::=
CONNECT （〈传送单元〉，〈结合单元〉，
〈使用表达式〉[，〈定位表达式〉[，〈下标表达式〉]]）

〈传送单元〉::=
〈访问单元〉
|〈文本单元〉

〈使用表达式〉::=
〈表达式〉

〈位置表达式〉::=
〈表达式〉

〈下标表达式〉::=
〈表达式〉

〈拆除内部子程序调用〉::=
DISCONNECT （〈传送单元〉）

〈访问属性内部子程序调用〉::=
GETASSOCIATION （〈传送单元〉）
| *GETUSAGE* （〈传送单元〉）
| *OUTOFFILE* （〈传送单元〉）

〈读记录内部子程序调用〉::=
READRECORD （〈访问单元〉[，〈下标表达式〉]
[，〈存储单元〉]）

〈写记录内部子程序调用〉::=
WRITERECORD （〈访问单元〉[，〈下标表达式〉]，
〈写表达式〉）

〈存储单元〉::=
〈静态模式单元〉

〈写表达式〉 ::=
〈表达式〉

〈文本内部子程序调用〉 ::=
READTEXT (〈文本入出变元表〉)
| *WRITETEXT* (〈文本入出变元表〉)

〈文本入出变元表〉 ::=
〈文本变元〉[,〈下标表达式〉],
〈格式变元〉[,〈入出表〉]

〈文本变元〉 ::=
〈文本单元〉
| 〈字符串单元〉
| 〈字符串表达式〉

〈格式变元〉 ::=
〈字符串表达式〉

〈入出表〉 ::=
〈入出表元素〉{,〈入出表元素〉}*

〈入出表元素〉 ::=
〈值变元〉
| 〈单元变元〉

〈单元变元〉 ::=
〈离散单元〉
| 〈串单元〉

〈值变元〉 ::=
〈离散表达式〉
| 〈串表达式〉

〈格式控制串〉 ::=
[〈格式文本〉]{〈格式说明〉[〈格式文本〉]}*

〈格式文本〉 ::=
{〈非百分号字符〉 | 〈百分号〉}

〈百分号〉 ::=
%%

〈格式说明〉 ::=
% [〈重复因子〉]〈格式元素〉

〈重复因子〉 ::=
{〈数字〉}+

〈格式元素〉 ::=
〈格式短语〉
| 〈带括号短语〉

〈格式短语〉 ::=
〈控制码〉[%.]

〈控制码〉 ::=

〈转换短语〉
| 〈编辑短语〉
| 〈入出短语〉

〈带括号短语〉::=
(〈格式控制串〉%)

〈转换短语〉::=
〈转换码〉{〈转换限定符〉}*
[〈短语宽度〉]

〈转换码〉::=
B|*O*|*H*|*C*

〈转换限定符〉::=
L|*E*|*P*〈字符〉

〈短语宽度〉::=
{〈数字〉}+|*V*

〈编辑短语〉::=
〈编辑码〉[〈短语宽度〉]
〈编辑码〉::=
X|<|>|*T*

〈入出短语〉::=
〈入出码〉

〈入出码〉::=
/|−|+|?|!|=

〈取文本内部子程序调用〉::=
GETTEXTRECORD(〈文本单元〉)
| *GETTEXTINDEX*(〈文本单元〉)
| *GETTEXTACCESS*(〈文本单元〉)
| *EOLN*(〈文本单元〉)

〈置文本内部子程序调用〉::=
SETTEXTRECORD(〈文本单元〉,〈字符串单元〉)
| *SETTEXTINDEX*(〈文本单元〉,〈整数表达式〉)
| *SETTEXTACCESS*(〈文本单元〉,〈访问单元〉)

8 异常处理

〈处理程序〉::=
ON {〈异常处理选择项〉}* [**ELSE**〈动作语句表〉] **END**

〈异常处理选择项〉::=
(〈异常表〉):〈动作语句表〉

9 时钟监控

〈计时动作〉::=

〈相对计时动作〉
| 〈绝对计时动作〉
| 〈周期计时动作〉

〈相对计时动作〉::=
AFTER 〈时延原值〉[**DELAY**] **IN**
〈动作语句表〉〈超时处理程序〉 **END**

〈超时处理程序〉::=
TIMEOUT 〈动作语句表〉

〈绝对计时动作〉::=
AT 〈绝对时钟原值〉 **IN** 〈动作语句表〉
〈超时处理程序〉 **END**

〈周期计时动作〉::=
CYCLE 〈时延原值〉 **IN**
〈动作语句表〉 **END**

〈时钟值内部子程序调用〉::=
〈时延内部子程序调用〉
| 〈绝对时钟内部子程序调用〉

〈时延内部子程序调用〉::=
MILLISECS (〈整数表达式〉)
| *SECS* (〈整数表达式〉)
| *MINUTES* (〈整数表达式〉)
| *HOURS* (〈整数表达式〉)
| *DAYS* (〈整数表达式〉)

〈绝对时钟内部子程序调用〉::=
ABSTIME ([[[[[[〈年表达式〉,]〈月表达式〉,]
〈日表达式〉,]〈时表达式〉,]〈分表达式〉,]〈秒表达式〉])

〈年表达式〉::=
〈整数表达式〉

〈月表达式〉::=
〈整数表达式〉

〈日表达式〉::=
〈整数表达式〉

〈时表达式〉::=
〈整数表达式〉

〈分表达式〉::=
〈整数表达式〉

〈秒表达式〉::=
〈整数表达式〉

〈计时简单内部子程序调用〉::=
WAIT ()
| *EXPIRED* ()

| *INTTIME* (〈绝对时钟原值〉, [[[[〈年单元〉
〈月单元〉,] 〈日单元〉,] 〈时单元〉,]
〈分单元〉,] 〈秒单元〉)

〈年单元〉 ::=
〈整数单元〉

〈月单元〉 ::=
〈整数单元〉

〈日单元〉 ::=
〈整数单元〉

〈时单元〉 ::=
〈整数单元〉

〈分单元〉 ::=
〈整数单元〉

〈秒单元〉 ::=
〈整数单元〉

10 程序结构

〈begin-end 分程序体〉 ::=
〈数据语句表〉〈动作语句表〉

〈过程体〉 ::=
〈数据语句表〉〈动作语句表〉

〈进程体〉 ::=
〈数据语句表〉〈动作语句表〉

〈模块体〉 ::=
{ 〈数据语句〉 | 〈可见性语句〉 | 〈区域〉 |
〈说明区域〉}* 〈动作语句表〉

〈区域体〉 ::=
{ 〈数据语句〉 | 〈可见性语句〉}*

〈说明模块体〉 ::=
{ 〈准数据语句〉 | 〈可见性语句〉 |
〈说明模体〉 | 〈说明区域〉}*

〈说明区域体〉 ::=
{ 〈准数据语句〉 | 〈可见性语句〉}*

〈上下文体〉 ::=
{ 〈准数据语句〉 | 〈可见性语句〉 |
〈说明模块〉 | 〈说明区域〉}*

〈动作语句表〉 ::=
{ 〈动作语句〉}*

〈数据语句表〉 ::=
{ 〈数据语句〉}*

〈数据语句〉 ::=

〈说明语句〉
| 〈定义语句〉

〈定义语句〉::=
〈同义模式定义语句〉
| 〈新模式定义语句〉
| 〈同义词定义语句〉
| 〈过程定义语句〉
| 〈进程定义语句〉
| 〈信号定义语句〉
| 〈空〉;

〈begin-end 分程序〉::=
BEGIN 〈begin-end 分程序体〉 **END**

〈过程定义语句〉::=
〈定义性出现〉:〈过程定义〉
[〈处理程序〉][〈简单名字串〉];

〈过程定义〉::=
PROC ([〈形参表〉]) [〈结果说明〉]
[**EXCEPTIONS** (〈异常表〉)]〈过程属性表〉;
〈过程体〉 **END**

〈形参表〉::=
〈形参〉{,〈形参〉}*

〈形参〉::=
〈定义性出现表〉〈参数说明〉

〈过程属性表〉::=
[〈通用性〉][**RECURSIVE**]

〈通用性〉::=
GENERAL
| **SIMPLE**
| **INLINE**

〈进程定义语句〉::=
〈定义性出现〉:〈进程定义〉
[〈处理程序〉][〈简单名字串〉];

〈进程定义〉::=
PROCESS ([〈形参表〉]);〈进程体〉 **END**

〈模块〉::=
[〈上下文表〉][〈定义性出现〉:]
MODULE [**BODY**]〈模块体〉 **END**
[〈处理程序〉][〈简单名字串〉];
| 〈远程模体〉

〈区域〉::=
[〈上下文表〉][〈定义性出现〉:]
REGION [**BODY**]〈区域体〉 **END**

[〈处理程序〉][〈简单名字串〉];
|〈远程模体〉

〈程序〉::=
{〈模块〉|〈说明模块〉|〈区域〉|〈说明区域〉}+

〈远程模体〉::=
[〈简单名字串〉:] **REMOTE**〈程序块标志符〉;

〈远程说明〉::=
[〈简单名字串〉:] **SPEC REMOTE**〈程序块标志符〉;

〈远程上下文〉::=
CONTEXT REMOTE〈程序块标志符〉
[〈上下文体〉] **FOR**

〈上下文模块〉::=
CONTEXT MODULE REMOTE〈程序块标志符〉;

〈程序块标志符〉::=
〈字符串字面值〉
|〈正文引用名字〉
|〈空〉

〈说明模块〉::=
〈简单说明模块〉
|〈模块说明〉
|〈远程说明〉

〈简单说明模块〉::=
[〈上下文表〉][〈简单名字串〉:] **SPEC MODULE**
〈说明模块体〉 **END** [〈简单名字串〉];

〈模块说明〉::=
[〈上下文表〉]〈简单名字串〉:**MODULE SPEC**
〈说明模块体〉 **END** [〈简单名字串〉];

〈说明区域〉::=
〈简单说明区域〉
|〈区域说明〉
|〈远程说明〉

〈简单说明区域〉::=
[〈上下文表〉][〈简单名字串〉:] **SPEC REGION**
〈说明区域体〉 **END** [〈简单名字串〉];

〈区域说明〉::=
[〈上下文表〉]〈简单名字串〉:**REGION SPEC**
〈说明区域体〉 **END** [〈简单名字串〉];

〈上下文表〉::=
〈上下文〉{〈上下文〉}*
|〈远程上下文〉

〈上下文〉::=

CONTEXT〈上下文体〉FOR

〈准数据语句〉::=
〈准说明语句〉
|〈准定义语句〉

〈准说明语句〉::=
DCL〈准说明〉{,〈准说明〉}*;

〈准说明〉::=
〈准单元说明〉
|〈准单元等同说明〉

〈准单元说明〉::=
〈定义性出现表〉〈模式〉[STATIC]

〈准单元等同说明〉::=
〈定义性出现表〉〈模式〉
LOC [NONREF] [DYNAMIC]

〈准定义语句〉::=
〈同义模式定义语句〉
|〈新模式定义语句〉
|〈同义词定义语句〉
|〈准同义词定义语句〉
|〈准过程定义语句〉
|〈准进程定义语句〉
|〈准信号定义语句〉
|〈空〉;

〈准同义词定义语句〉::=
SYN〈准同义词定义〉{,〈准同义词定义〉}*;

〈准同义词定义〉::=
〈定义性出现表〉{〈模式〉=[〈常数值〉]
|[〈模式〉]=〈字面值表达式〉}

〈准过程定义语句〉::=
〈定义性出现〉:PROC([〈准形参表〉])
[〈结果说明〉][EXCEPTIONS(〈异常表〉)]
〈过程属性表〉END[〈简单名字串〉];

〈准形参表〉::=
〈准形参〉{,〈准形参〉}*

〈准形参〉::=
〈简单名字串〉{,〈简单名字串〉}*〈参数说明〉

〈准进程定义语句〉::=
〈定义性出现〉:PROCESS([〈准形参表〉])
END[〈简单名字串〉];

〈准信号定义语句〉::=
SIGNAL〈准信号定义〉{,〈准信号定义〉}*;

〈准信号定义〉::=
〈定义性出现〉[=(〈模式〉{,〈模式〉}*)][TO]

11 并发执行

〈信号定义语句〉::=
SIGNAL〈信号定义〉{,〈信号定义〉}*;

〈信号定义〉::=
〈定义性出现〉[=(〈模式〉{,〈模式〉}*)][TO〈进程名字〉]

12 总的语义性质

〈可见性语句〉::=
〈移出语句〉
|〈移入语句〉

〈前缀更名子句〉::=
(〈老前缀〉—〉〈新前缀〉)!〈后缀〉

〈老前缀〉::=
〈前缀〉
|〈空〉

〈新前缀〉::=
〈前缀〉
|〈空〉

〈后缀〉::=
〈移入后缀〉{,〈移入后缀〉}*
|〈移出后缀〉{,〈移出后缀〉}*

〈移出语句〉::=
GRANT〈前缀更名子句〉{,〈前缀更名子句〉}*;
|GRANT〈移出窗口〉[〈前缀子句〉];

〈移出窗口〉::=
〈移出后缀〉{,〈移出后缀〉}*

〈移出后缀〉::=
〈名字串〉
|〈新模式名字串〉〈禁止子句〉
|[〈前缀〉!]ALL

〈前缀子句〉::=
PREFIXED[〈前缀〉]

〈禁止子句〉::=
FORBID{〈禁止名字表〉|ALL}

〈禁止名字表〉::=
(〈域名字〉{,〈域名字〉}*)

〈移入语句〉::=

SEIZE〈前缀更名子句〉{,〈前缀更名子句〉}*;
| **SEIZE**〈移入窗口〉[〈前缀子句〉];

〈移入窗口〉::=
〈移入后缀〉{,〈移入后缀〉}*

〈移入后缀〉::=
〈名字串〉
| [〈前缀〉!] **ALL**

〈情况标号说明〉::=
〈情况标号表〉{,〈情况标号表〉}*

〈情况标号表〉::=
(〈情况标号〉{,〈情况标号〉}*)
|〈无关紧要〉

〈情况标号〉::=
〈离散字面值表达式〉
|〈字面值范围〉
|〈离散模式名字〉
| **ELSE**

〈无关紧要〉::=
(*)

附录 G：产生式规则索引

非终结符	定义处 节 号	定义处 页 号	使用处 页 号
〈八进制整数字面值〉	5.2.4.2		
〈老前缀〉	12.2.3.3		
〈异常处理选择项〉	8.2		
〈运算数-0〉	5.3.3		
〈运算数-1〉	5.3.4		
〈运算数-2〉	5.3.5		
〈运算数-3〉	5.3.6		
〈运算数-4〉	5.3.7		
〈运算数-5〉	5.3.8		
〈运算数-6〉	5.3.9		
〈运算符-3〉	5.3.5		
〈运算符-4〉	5.3.6		
〈原始数组模式名字〉	3.12.3		
〈原始串模式名字〉	3.12.2		
〈原始变体结构模式名字〉	3.12.4		
〈参数属性〉	3.7		
〈参数化数组模式〉	3.12.3		
〈参数化串模式〉	3.12.2		
〈参数化结构模式〉	3.12.4		
〈参数表〉	3.7		
〈参数说明〉	3.7		
〈带括号短语〉	7.5.4		
〈带括号表达式〉	5.2.16		
〈百分号〉	7.5.4		
〈程序块标志符〉	10.10.1		
〈定位〉	3.12.5		
〈后缀〉	12.2.3.3		
〈幂集求差运算符〉	5.3.6		
〈幂集枚举〉	6.5.2		
〈幂集蕴含运算符〉	5.3.5		
〈幂集模式〉	3.5		
〈幂集多元组〉	5.2.5		
〈前缀〉	2.7		
〈前缀子句〉	12.2.3.4		
〈带前缀名字串〉	2.7		
〈前缀更名子句〉	12.2.3.3		
〈原值〉	5.2.1		
〈优先数〉	6.16		
〈过程体〉	10.2		
〈过程属性表〉	10.4		
〈过程调用〉	6.7		
〈过程定义〉	10.4		
〈过程定义语句〉	10.4		
〈过程模式〉	3.7		
〈进程体〉	10.2		
〈进程定义〉	10.5		
〈进程定义语句〉	10.5		
〈程序〉	10.8		

非终结符	定义处 节 号	定义处 页 号	使用处 页 号
〈准数据语句〉	10.10.3		
〈准说明〉	10.10.3		
〈准说明语句〉	10.10.3		
〈准定义语句〉	10.10.3		
〈准形参〉	10.10.3		
〈准形参表〉	10.10.3		
〈准单元说明〉	10.10.3		
〈准单元等同说明〉	10.10.3		
〈准过程定义语句〉	10.10.3		
〈准进程定义语句〉	10.10.3		
〈准信号定义〉	10.10.3		
〈准信号定义语句〉	10.10.3		
〈准同义词定义〉	10.10.3		
〈准同义词定义语句〉	10.10.3		
〈引号〉	5.2.4.7		
〈范围〉	5.2.5		
〈范围枚举〉	6.5.2		
〈范围表〉	6.4		
〈范围模式〉	3.4.6		
〈可达区初始化〉	4.1.2		
〈读记录内部子程序调用〉	7.4.9		
〈接收缓冲区情况动作〉	6.19.3		
〈接收情况动作〉	6.19.1		
〈接收表达式〉	5.3.9		
〈接收信号情况动作〉	6.19.2		
〈记录模式〉	3.10.3		
〈被引用单元〉	5.3.9		
〈被引用模式〉	3.6.2		
〈引用模式〉	3.6.1		
〈区域〉	10.7		
〈区域体〉	10.2		
〈区域说明〉	10.10.2		
〈关系运算符〉	5.3.5		
〈相对计时动作〉	9.3.1		
〈远程上下文〉	10.10.1		
〈远程模体〉	10.10.1		
〈远程说明〉	10.10.1		
〈重复因子〉	7.5.4		
〈结果〉	6.8		
〈结果动作〉	6.8		
〈结果属性〉	3.7		
〈结果说明〉	3.7		
〈返回动作〉	6.8		
〈右元素〉	4.2.7		
〈行模式〉	3.6.4		
〈秒表达式〉	9.4.2		

非终结符	定义处节号	定义处页号	使用处页号
〈秒单元〉	9.4.3		
〈移入后缀〉	12.2.3.5		
〈移入语句〉	12.2.3.5		
〈移入窗口〉	12.2.3.5		
〈发送动作〉	6.18.1		
〈发送缓冲区动作〉	6.18.3		
〈发送信号动作〉	6.18.2		
〈集合元素〉	3.4.5		
〈集合表〉	3.4.5		
〈集合字面值〉	5.2.4.5		
〈集合模式〉	3.4.5		
〈置正文内部子程序调用〉	7.5.8		
〈信号定义〉	11.5		
〈信号定义语句〉	11.5		
〈信号接收选择项〉	6.19.2		
〈简单名字串〉	2.2		
〈简单前缀〉	2.7		
〈简单说明模块〉	10.10.2		
〈简单说明区域〉	10.10.2		
〈单赋值动作〉	6.2		
〈片大小〉	4.2.7		
〈说明模块〉	10.10.2		
〈说明模块体〉	10.2		
〈说明区域〉	10.10.2		
〈说明区域体〉	10.2		
〈启动动作〉	6.13		
〈开始位〉	3.12.5		
〈开始元素〉	4.2.6		
〈启动表达式〉	5.2.14		
〈开始值〉	6.5.2		
〈步长〉	3.12.5		
〈步长枚举〉	6.5.2		
〈步长大小〉	3.12.5		
〈步长值〉	6.5.2		
〈停止动作〉	6.14		
〈存储单元〉	7.4.9		
〈串并置运算符〉	5.3.6		
〈串元素〉	4.2.6		
〈串长度〉	3.12.2		
〈串模式〉	3.12.2		
〈串重复运算符〉	5.3.8		
〈串片〉	4.2.7		
〈串类型〉	3.12.2		
〈结构域〉	4.2.10		
〈结构模式〉	3.12.4		
〈结构多元组〉	5.2.5		
〈子表达式〉	5.3.2		
〈子运算数-0〉	5.3.3		
〈子运算数-1〉	5.3.4		

非终结符	定义处 节 号	定义处 页 号	使用处 页 号
〈变体域〉	3.12.1		
〈可见性语句〉	12.2.3.2		
〈位置表达式〉	7.4.6		
〈当型控制〉	6.5.3		
〈开域控制〉	6.5.4		
〈开域部分〉	6.5.4		
〈字〉	3.12.5		
〈写表达式〉	7.4.9		
〈写记录内部子程序调用〉	7.4.9		
〈年表达式〉	9.4.2		
〈年单元〉	9.4.3		
〈零目运算符〉	5.2.15		

附录 H：索引

涉及一术语的定义性出现可参见该索引中以黑体表示的相应页；所索引术语的应用性出现参见正体表示的相应页。

注 — 下文页号均取自英文原版卷 X.6。

中国印刷 ISBN 92-61-03805-0