Visual Prolog 7.5
语言参考手册

2015.03

前言

这个文档，是在译者以前翻译完成的《Visual Prolog 7.4语言参考手册》基础上，根据新的Visual Prolog 7.5 Language Reference补充和修改的。

修改过程中，得到了Thomas Linder Puls先生的有益帮助，在此向他表示诚挚的感谢！

乙丁 its2u@qq.com

2015.03

目录

Visual Prolog 7.5语言参考手册

本文档描述 Visual Prolog 编程语言的语法与语义。
Visual Prolog 是面向对象的强类型编程语言，它的基础是逻辑编程语言 Prolog。
一个 Visual Prolog 程序是由一个 goal（目标）及以下一些部件构成的：

- interfaces（接口）
- Class declarations（类声明）和
- Class implementations（类实现）

上述部件包含有下面这些 Prolog 实体的声明与定义：

- Domains （域）
- Constants （常数）
- Predicates （谓词）
- Properties （属性）
- Fact databases （事实数据库）。

Visual Prolog的实际代码是clauses（子句），它在类实现当中，谓词是由它实现的。

1. 基本概念

类型和子类型

类型

Visual Prolog 的类型分为对象类型与值类型。对象具有可变的状态而值是不可变的。

对象的类型是由 interface（接口）定义的。

值的类型包括numerical types（数值类型）、strings（串）、character types（字符类型）和compound domains（复合域）。复合域又称为代数数据类型，它的简单形式有结构及枚举类型，而其复杂形式会呈现为树的结构。

子类型

类型是由子类型体系构成的。子类型用来引入多态包容性：任何期望某类型值的语境也能很好地接受其任意子类型的值。或者反过来说：需要时某个类型的值可以自动地转换成其父类型加以应用，而不必进行显式的类型转换。

除代数数据类型而外，其它值类型都可派生出子类型。代数数据类型派生出的类型是同义类型而非子类型，也就是说派生出的是同一类型。

子类型和子集在概念上有联系，但是一定要注意，即便一个类型“算术上”是另一个类型的子集它也未必就是一个子类型。只有声明了之后一个类型才可以是另一个类型的子类型。

```
domains
  t1 = [1..17].
  t2 = [5..13].
  t3 = t1 [5..13].
```

t1 是整数类型，值从 **1** 到 **17**（含）。而 **t2** 的值是 **5** 到 **13**，所以，**t2** 是 **t1** 的一个子集，但它并非 **t1** 的一个子类型。另一方面，**t3**（它也含有与 **t2** 一样的值）则是 **t1** 的一个子类型，因为其声明如此。

本语言中有几个隐含的子类型关系，但一般而言子类型关系都必须在类型定义中显式地说明。

对象类型也是由子类型体系构成的，根是预定义的对象类型 ***Object***，也就是说任意对象类型都是 ***Object*** 的子类型。对象子类型的定义是通过声明说一个接口支持(supports)另一个而实现的。如果一个对象具有的接口/对象类型支持某个其它接口，那么该对象就有了那种类型，可以无冲突地当作那种对象来使用。

参看：Universal and Root Types（通用类型和根类型）

对象系统

外视图

这一节的描述不是对类的一般性介绍，而是对 Visual Prolog 中类的概念说明，希望读者已经熟悉通常的类概念。描述完全是纯概念意义上的，不涉及语法、实现等，也不考虑操作或编程方面的问题。尽管引入类、对象的主要原因是程序自然需要，但先不考虑程序上的原因而只描述基本概念还是很有价值的。

Visual Prolog 中类的概念以如下三个语义实体为基础：

- objects
- interfaces
- classes

Object

一个对象，就是一套命名了的对象成员谓词及一套支持的接口。对象也有状态，但这个状态只能通过成员谓词才能改变和观察。我们称之为状态封装在对象中。

这里的**封装**，意味着对象能隐藏其内部数据和方法，只提供需要的部份让程序访问。封装和模块化的重要性是众所周知的。封装有助于构建更加结构化和易读的程序，因为对象可以看作为一个黑盒子。对待一个复杂问题，先找出可以下断言和描述的一部分，把它封装在一个对象里，构造一个接口，接着再往下，直到声明了所有子问题。封装了问题对象后，只要保证它们工作正常，就可以从对象中抽取数据。

Interface

接口是一个对象类型，它有一个名字并定义一组命名了的对象谓词。

接口按 supports 体系构建（这个体系是半格状的，根是接口 objects）。如果一个对象具有一个接口所表示的类型，它也就有了所有被支持接口的类型。因此，supports 体系也是类型体系。一个接口是所有被它支持接口的一个子类型。也可以说对象支持接口，如果接口名为 *X*，我们可以说这个对象是一个 *X*，或是一个 *X* 对象。

Class

类是命名了的对象工厂，它创建相应于某个接口的对象。任何对象都是由类创建的，如果一个类是用接口**C**构造对象的，那么这个类创建的所有对象都叫C对象。

由某个类构造的所有对象共享相同的对象成员谓词定义，但每个对象有其自身的状态。因此，对象成员谓词实际上是类的一部分，而对象状态则是对象自身的一部分。

类还含有另一组命名了的谓词和封装的状态，分别称之为类成员和类状态。类成员与类状态存在于各个类中，而对象成员与对象状态存在于各个对象中。类成员和对象成员都可以访问类状态。

注意！一个类所定义的对象成员谓词是该类各接口中声明的谓词整体。更明确地说，如果在不同的两个接口中声明了同一个谓词，类只会提供该谓词的一个定义。因此，类只是检测实际意义，也就是说只是看这两个继承谓词的预期语义是否一致。

要注意，接口支持必须明确地指定。某个类提供相应某个接口的谓词并不意味着这个类支持这个接口。

模块

事实上，类并非必须产生对象。不产生对象的类只有类成员和类状态，而这样的类当模块看待更恰当。

标识

每个对象都是唯一的：对象具有可变的状态，而且对象的状态只能通过它们的成员谓词观察到，因而对象只与它自己是同一的。这就是说，即使有两个对象的状态是一样的，它们也不是同一的，因为我们可以改变一个对象的状态而不影响另一个对象的状态。

我们无法直接访问一个对象的状态，而总是通过对象的引用来访问对象的状态。尽管对象仅与自身是同一的，但可以对同一对象有多个引用。因此，一个对象可以由许多不同的引用来访问。

类和接口也是唯一的，用它们的名字来标识。在一个程序中，两个接口或两个类不能用相同的名字。只有当某个类构造某个接口的对象时，这个类和这个接口才可以用相同的名字。

总之，结构的相同并不意味着同一，对象、类和接口都是这样。

内视图

前一节描述了对象、类及接口的外在表现，这一节描述内部情况。这些内部问题更具程序特性，可以把它们分成类的声明与实现两个部分来考虑。

从程序的观点，类是核心：代码是包含在类中的。

接口主要地是静态特性，实事上，接口只存在于一个程序的文本表示中，运行时没有一个（直接的）代表接口的东西。而另一方面，对象主要地是动态特性，它在程序中并不是直接可见的，直到程序实际运行时它才存在。

类由声明与实现构成。声明通告了类的 *public*（公共）可访问的部分及它生成的对象。而实现则 *defines*（定义）在类声明中声明了的实体。 谓词的基本实现当然是子句，但谓词也可以由继承定义，或是由外部的库来确定。

在 Visual Prolog 中，类的声明纯粹是说明，它只是陈述有哪些实体可以访问，但并不涉及如何访问及实体在哪儿实现。

类实现可以声明和定义更多的实体（域、谓词等），它们只在类的内部是可见的，也就是说，它们是私有的（*private*）。

对象的状态是存储在对象本身的事实中的，这些事实是在类的实现中普通事实（数据库）段中声明的。事实对每个对象都是局部的（就像其它的对象实体），而类事实则由同类的所有对象共享。

事实只能在类的实现中声明，因而也就不能从类的外部（直接）访问。

类的实现可以声明支持类声明中提到的接口以外的其它接口。但这个消息也仅只在实现本身内部是可见的，因而还是私有的。

代码继承

Visual Prolog 的代码继承只能在类的实现中进行，可以有多重继承。通过在类实现中的 *inherits* 段指明某个类来得到该类的继承。被继承的类称为父类（*parent* classes）或超类（*super*-classes），子类（*Child* class）或亚类（*sub*-class）与父类是成对的，我们也说子类继承自父类。子类只能通过父类的公共接口访问它的父类，也就是说它与其它使用该父类的类一样，并没有接受额外的特权。

Scoping & Visibility（范围与可见性）

名称的类别

Visual Prolog 中的所有名称（标识）按语法分成两个大组：

- 常量名称（以小写字母开头）
- 变量名称（以大字字母或下划线开头）

常量名称又分为如下几类：

- 类型名（如域和接口）；
- 域载体（如类和接口）；

- 不带圆括号的名称（如常数、非函数类型的事实变量及零函子）；
- 返回值的元维为 N 的名称（如函数、函子和函数类型的事实变量）；
- 不返回值的元维为 N 的名称（如谓词、事实及谓词类型的事实变量）。

Visual Prolog 要求在声明阶段名称就不能有冲突，因为在使用阶段它不可能解决这样的冲突。只有在同一范围的名称才可能会有冲突，而不同的范围因为范围的限定可以解决冲突。某个类别的名称决不会与另一个类别中的名称有冲突，但是我们就要看到在单个声明中可以把一个名字放在几个类别中。

Packages（包）

包，是 Visual Prolog 中一般公认的代码组织基本单位，我们用包来组建东西。包的使用，保证了不同工程间构造原则的一致性。包为工程间构造和共享代码的工具确定了标准。

包是一些接口和类的集合体，它为所有这些接口和类提供了某个共同的名称。包里每个接口的各个声明或实现是放在独立的文件中的，这些文件的文件名与在该文件中声明的或实现的类或接口的名称相对应。所有包文件存放在一个独立的包目录下（如果一个包有子包，则子包会放在包目录的子目录中）。

包的概念，用来把一些相关的接口与类组合在一起，它的作用类似于类库的角色。在程序中包可以把使用的接口和类集中起来，而不是直接把这些接口和类放在程序里。

在帮助文件中的VDE部分，描述了Visual Prolog中可用的包结构及如何将包放到工程中。（参看 **Creating a Package** in **Creating New Project Items**）

可见性、屏蔽及限定符

大多数范围规则上面都说了，这一节补充最后的内容。

接口定义，类声明以及类实现都是范围（范围不能嵌套）。实现（悄悄地）扩展了相应的类声明的范围。在一个范围内，可见性到处都是一样的。这尤其是指在一个范围内不论某个东西是在哪儿声明的，它在整个范围内都是可见的。

来自被支持的接口和超类的公用名称，如果它出自何处不模糊，在一个范围内是直接可用的（也就是不需要限定符）。谓词调用中的模糊问题，可以用带类名的谓词名的修饰方法（如 **cc::p**）来解决。

这样的限定符，也用于修饰 *当前* 对象对超类的对象成员谓词的调用中。

Visual Prolog 有两个屏蔽层级：

- 局部的
- 超类及开放范围（opened scopes）

开放范围的情况与超类的一样，所以我们下面只说超类。

层级的意思是，局部声明会屏蔽超类声明，但两个超类间不会有屏蔽，所有超类都有同样的优选权。如果两个或更多个超类包含有冲突的声明，那这些声明就只能通过限定符来访问。

例 设有接口 **aa** 及类 **aa_class**：

```
interface aa
  predicates
    p1 : () procedure ().
    p2 : () procedure ().
    p3 : () procedure ().
end interface
class aa_class : aa
end class
```

再假设有 **bb_class** 类：

```
class bb_class
  predicates
    p3 : () procedure ().
    p4 : () procedure ().
end class bb_class
```

在上述情况下，来看看类 **cc_class** 的实现：

```
implement cc_class inherits aa_class
  open bb_class
  predicates
    p2 : () procedure ().
    p5 : () procedure ().
  clauses
    new() :-
      p1(),              % aa_class::p1
      p2(),              % cc::p2 (屏蔽了aa_class::p2)
      aa_class::p2(),    % aa_class::p2
      p3(),              % 错误的模糊调用：可以是aa_class::p3或bb_class::p3
      aa_class::p3(),    % aa_class::p3
      bb_class::p3(),    % bb_class::p3
      p4(),              % bb_class::p4
      p5(),              % cc::p5
end implement cc_class
```

2. 词汇元素

源文件要经过 Visual Prolog 编译器的编译。一个源文件可以包含其它的源文件，从概念上来讲插入的这些文件与包含它们的原始文件一道构成了一个**编译单元**。一个编译单元的编译要经过两个步骤：

- 首先，输入被转换成记号的序列；
- 其次，对这些记号进行语法分析并转换成可执行代码。

对程序的词汇分析把编译单元 ***CompilationUnit*** 再细分成输入元素 ***InputElement*** 的表：

```
CompilationUnit:
  InputElement-list
```

```
InputElement:
  Comment
  WhiteSpace
  Token
```

只有记号（Token）对后面的语法分析有重要意义。

注释

Visual Prolog 的注释可以按以下方法编写：

- 以/*（斜杠，星）开头，后面是任意字符序列（包括换行符），结束用*/（星，斜杠）。这样的注释可以有多行，可以嵌套。
- 以%（百分号）开头，后面是任意字符序列。这样的注释在一行的结尾就结束了，因此通常称之为单行注释。

看一下下面的注释：

```
/* 注释1的开头
% 嵌套了注释2 */ 这个标记不会结束多行注释
因为它插入了一个单行注释
这个才是注释1实际结束的标记 */
```

空白

```
WhiteSpace:
  Space
  Tab
  NewLine
```

这里的 ***Space*** 是空格字符，***Tab*** 是制表字符而 ***NewLine*** 是换行字符。

Tokens（记号）

```
Token:
  Identifier（标识）
  Keyword（关键字）
  Punctuator（标点）
  Operator（操作符）
```

```
Literal（文字）
```

标识

```
Identifier:
  LowercaseIdentifier（小写标识）
  UppercaseIdentifier（大写标识）
  AnonymousIdentifier（匿名标识）
  Ellipsis（省略号）
```

*小写标识*是一个以小写字母开头的字母、数字及下划线组成的序列；*大写标识*是一个以大写字母或下划线开头的字母、数字及下划线组成的序列。*匿名标识*是一个下划线：“_”；而*省略号*则是三个句点：“...”。

关键字

关键字分为主关键字和次关键字两类，但这只是表面上的分法，主次关键字并没有形式上的差异。不过我们下面还是用不同颜色区分它们：

```
Keyword :
  MajorKeyword（主关键字）
  MinorKeyword（次关键字）
```

```
MajorKeyword : one of
  class clauses constants constructors
  delegate domains
  end
  facts
  goal guards
  implement inherits interface
  monitor
  namespace
  open
  predicates
  properties
  resolve
  supports
```

```
MinorKeyword : one of
  align and anyflow as
  bitsize
  catch
  determ digits div do
  else elseif erroneous externally
  failure finally foreach from
  guards
  if in
  language
  mod multi
  nondeterm
  or orelse
  procedure
  quot
  rem
  single
  then to try
```

所有关键字除了 **as** 和 **language** 都是保留字。

end总是与其它关键字结合使用：

```
end class
end implement
end interface
end if
end foreach
end try
```

标点符号

标点符号在 Visual Prolog 中对编译器既有语法意义又有语义意义，但它们本身不是能产生值的操作运算。有的标点符号，单独或组合起来也可以是 Visual Prolog 的操作符。

标点符号有：

```
PunctuationMarks: one of
  ;   !   ,   .   #   [   ]   |   (   )   {   }   :-   :   ::
```

操作符

操作符规定了对相关的操作数要进行的运算操作。

```
Operators: one of
  ^
  /   *   div   mod   quot   rem
  +   -
  =   <   >   <>   ><   <=   >=   :=
```

所有操作符都是二元的，但 **-** 和 **+** 也可以是一元的操作符。

div、**mod**、**quot** 和 **rem** 是保留字。

文字

文字有以下类型：整数、字符、浮点数、串、二进制数字和表：

```
Literal:
  IntegerLiteral
  RealLiteral
  CharacterLiteral
  StringLiteral
  BinaryLiteral
  ListLiteral
  CompoundDomainLiteral
```

整数文字

```
IntegerLiteral:
  UnaryPlus-opt    DecimalDigit-list
  UnaryMinus-opt    DecimalDigit-list
  UnaryPlus-opt    OctalPrefix OctalDigit-list
```

```
    UnaryMinus-opt    OctalPrefix OctalDigit-list
    UnaryPlus-opt    HexadecimalPrefix HexadecimalDigit-list
    UnaryMinus-opt    HexadecimalPrefix HexadecimalDigit-list
  UnaryPlus:
    +
  UnaryMinus:
    -
  OctalPrefix:
    0o
  OctalDigit: one of
    0 1 2 3 4 5 6 7
  DecimalDigit: one of
    0 1 2 3 4 5 6 7 8 9
  HexadecimalPrefix:
    0x
  HexadecimalDigit: one of
    0 1 2 3 4 5 6 7 8 9 A a B b C c D d E e F f
```

一个整数文字可以属于整数或无符号数域，它不能超过整数或无符号数的最大和最小值。

实数文字

```
  RealLiteral:
    UnaryMinus-opt DecimalDigit-list FractionOfFloat-opt Exponent-opt
  FractionOfFloat:
    . DecimalDigit-list
  Exponent:
    ExponentSymbol ExponentSign-opt DecimalDigit-list
  ExponentSymbol: one of
    e E
  ExponentSign: one of
    - +
```

浮点数文字表示的值也应不超过实数所能表示的最大最小值。

字符文字

```
  CharacterLiteral:
    ' CharacterValue '
```

上面的 ***CharacterValue*** 可以是任意可打印字符或一个换码序列：

- **** 代表 ****
- **\t** 代表制表符
- **\n** 代表换行符
- **\r** 代表回车
- **\'** 表单引号
- **\"** 代表双引号
- **\u*XXXX***，这里的 ***XXXX*** 应该是四位 *十六进制数*，**\u*XXXX***代表相应这个数的Unicode字符。

串文字

```
  StringLiteral:
```

```
        StringLiteralPart-list

    StringLiteralPart:
      ' <CharacterValue>-list-opt '
      " <CharacterValue>-list-opt "
      @AtOpenChar AnyCharacter-list-opt AtCloseChar
```

串文字是由一个或多个 ***StringLiteralPart（串文字部件）*** 连接构成的。

前两种形式（' 和 " ）使用换码序列来表示特定字符：

- **** 代表 ****
- **\t** 代表制表符
- **\n** 代表换行符
- **\r** 代表回车
- **\"** 代表双引号
- **\'** 代表单引号
- **\u*XXXX***，这里的 ***XXXX*** 应该是四位*十六进制数*，代表相应这个数的Unicode字符。

在单引号串中可以选用双引号换码序列；同样，在双引号串中可以选用单引号换码序列。

单引号串必须至少包含两个字符，否则就被视为是字符文字（而不是串）。

以@开头的串文字比使用换码序列的串好懂。它以@开头，后跟若干非字母字符 ***AtOpenChar***，而以 ***AtCloseChar*** 结尾。对大多数字符来说，开始字符 ***AtOpenChar*** 与结尾字符 ***AtCloseChar*** 是一样的，但对于配对字符来说结尾字符要用与之配对的那个字符，如下表：

开始	结尾	开始	结尾
@(	)	@)	(
@[	]	@]	[
@{	}	@}	{
@<	>	@>	<

对于非配对字符，开始字符与结尾字符相同，比如说：@" is closed by "。

在所有的以@开头的串中，连续出现两次的结尾字符不表示串的结束，而是表示串中出现的结尾字符本身。

下面的例子是用@[开头和]结尾的，串中间使用了串字符"和'：

```
constants
    html : string = @[<span onclick="click('wer')">x</span>].
```

二进制数文字

```
    BinaryLiteral:
      $[ ElementValue-comma-sep-list-opt ]
    ElementValue:
      IntegerLiteral
```

ElementValue 可以是任意整数算术表达式（如常数），在编译时应该是可计算的，值的范围是 **0** 到 **255**。

表文字

一个表中的所有元素必须属于同一个域（或兼容域）。域可以是任何内建（built-in）的或用户定义域

（user defined domain），例如，可以是整数、字符、二进制、复合域等等。

```
ListLiteral:
  [ SimpleLiteral-comma-sep-list-opt ]
  [ SimpleLiteral-comma-sep-list | ListLiteral ]
```

这里的 ***SimpleLiteral（简单文字）*** 可以是：

```
SimpleLiteral:
  IntegerLiteral
  RealLiteral
  CharacterLiteral
  StringLiteral
  BinaryLiteral
  ListLiteral
  CompoundDomainLiteral
```

例：

```
[]                      % 空表
[1,2,3]                 % 整数表
["abc", "defg"]         % 串表
```

下面这个表是非法的，因为表中所有元素必须是同一类型的：

```
[1,"abc"]               % 这个表不合规定。
```

复合域文字

用户定义复合域所有的参数如果都是文字的则也可以当文字看待。

3. 编译单元

一个程序由若干编译单元构成，编译器对每个编译单元分别进行编译，一个编译结果就是一个目标文件。这些目标文件（可能还有其它文件）链接在一起，就产生了工程的结果。一个程序必须有一个、也只能有一个目标段（goalSection），它是整个程序的入口。

一个编译单元所有的引用名必须是已经在本单元中声明或定义过的，在这个意义上说一个编译单元必须是自完备的。接口定义与类声明可以包含在数个编译单元中（定义/声明在其所在的包含单元中必须是一致的），而类实现（定义）只能定义在单一的单元中。工程中也必须定义每个声明项，但有些定义也可以在库中，也就是说不一定非要原文定义。

一个编译单元（其中可能含有**#include**指令）是若干编译项的一个序列。

```
CompilationUnit：
  CompilationItem-list-opt
```

一个编译项可以是一个接口、一个类声明、一个类实现、目标段或是一个条件编译项（这个项将在条件编译中介绍）。

```
CompilationItem：
  Directive
  NamespaceEntrance
  ConditionalItem
  InterfaceDefinition
  ClassDeclaration
  ClassImplementation
  GoalSection
```

请参看：

- Interfaces接口
- Classes类
- Implementations实现
- Namespace 名称空间
- Goal Section目标段
- Directives指令
- Conditional Item条件项

4. Interfaces（接口）

一个接口定义规定了一个已命名对象的类型。接口可以支持其它的接口，详细内容请参看Supports限定符。

在一个接口中声明的所有谓词，都是该接口类型对象中的对象成员。

接口也是一个可以定义常数和域的全局范围。因此，在一个接口中定义的常数和域，并不是该接口所声明类型的一部份（或是说并非那种类型对象的一部份）。这样的域和常数可以用该接口名限定符**interfaceName::constant**，或是使用open限定符（参看Open限定符）在其它范围中引用。

```
InterfaceDeclaration :
  interface IinterfaceName
   ScopeQualifications
   Sections
  end interface IinterfaceName-opt
```

```
InterfaceName :
  LowerCaseIdentifier
```

参看Generic Interfaces及Monitors。

在上述结构中，结尾处的接口名（***InterfaceName***）要是用的话，一定要和结构开始处的相一致。

ScopeQualifications 的类型必须是：

- Supports限定符
- Open限定符

Sections 的类型必须是：

- ConstantsSection常数段
- DomainsSection域段
- PredicatesSection谓词段
- PredicatesFromInterface来自接口的谓词
- PropertiesSection属性段
- ConditionalSection条件段

在条件段中包含的所有段也必须是这些类型的。

The interface: object（object接口）

如果一个接口没有显式地支持任何接口，它意味着支持的只有内建接口 *object*。

object 是一个空接口，也就是说，它不含有谓词等。其主要用途是作为所有对象的通用基础类型。

Open限定符

Open 限定符用来使对类级实体的引用更便捷。Open 段使一个范围内的名称进入到另一个范围，因而不用限定符就可以引用。

Open 对对象成员的名称没有影响，因为它们只能通过对象来访问。但是对类成员、域、函子及常数的名称，访问时不需要限定符。

用这样的方式把名称带入到一个范围时，可能会引起一些名称的混淆（参看Scoping）。

Open 段只在它们出现的范围内有影响，这尤其是指在一个类声明中的 open 段不会影响到类的实现。

```
OpenQualification：
  open ScopeName-comma-sep-list
```

Supports限定符

Supports限定符只能用在接口定义（interfaceDefinition）与类实现（classImplementation）中。它有两个作用：

- 指定用一个接口 **A** 扩展另一个接口 **B**，因而使 **A** 类型的对象是 **B** 类型对象的一个子类
- 声明某个类的对象相对于原来构造类型规定的来说，“私下地”有了更多的对象类型

supports 是可传递的：如果接口 **A supports** 接口 **B**，而 **B** 又 **supports C**，则 **A** 也 **supports C**。

如果一个接口没有显式地支持任何接口，则意味着它是支持预定义的接口 *object*。

从功能上说，一个接口支持某个接口一次或多次（直接间接都算）没有什么差别，但可能会造成对象表示上的差别。

在类的实现中使用 **supports** 时，会导致 **This** 不仅可以与构造类型一道使用，也可以用于任意私有支持的对象类型。

```
SupportsQualification：
  supports InterfaceName-comma-sep-list
```

再强调一下，Supports 限定符只能用在接口定义和类实现中。

注意，如果各接口间有相互冲突的谓词，这样的接口就不能和 **supports** 限定符一起使用。谓词冲突，是指来自不同**本原接口**的谓词有相同的名称及变元数。而本原接口，是指谓词被原文声明的接口，它是相对于用 **supports** 限定符间接声明的接口而言的。因此，如果一个接口在 **supports** 链中出现了两次或多次，一定小心不要引起冲突。

例 来看一下下面的定义与声明：

```
interface aaa
predicates
  insert：(integer X).
end interface

interface bbb
  supports aaa
predicates
  insert：(integer X, string Comment).
end interface

interface cc
  supports aaa
predicates
  extract：() -> integer.
end interface

interface dd
  supports aaa, bbb, cc
predicates
  extract：(string Comment) -> integer.
end interface
```

下面是在 **dd** 中找到的全部谓词的表（按深度遍历）：

```
predicates
  insert : (integer).              % dd -> aaa
  insert : (integer).              % dd -> bbb -> aaa
  insert : (integer, string).      % dd -> bbb
  insert : (integer).              % dd -> cc -> aaa
  extract : () -> integer.         % dd -> cc
  extract: : (string) -> integer. % dd
```

这里有些谓词是相同的，因此实际上 **dd** 中含有的成员如下：

```
predicates
  insert : (integer).              % 本原接口: aaa
  insert : (integer, string).      % 本原接口: bbb
  extract : () -> integer.         % 本原接口: cc
  extract : (string) -> integer. % 本原接口: dd
```

例 考虑如下接口：

```
interface aaa
predicates
  insert : (integer X).
end interface

interface bbb
predicates
  insert : (integer X).
end interface

interface cc
  supports aaa, bbb          % 有冲突的接口
end interface
```

接口 cc 是不合法的，因为在 aaa 中支持的 insert/1 以 aaa 为本原接口，而在 bbb 中支持的 insert/1 以 bbb 为本原接口。

5. Classes（类）

类声明向外界定义了类的外观，外界可以看到和准确地使用类声明中记述过的实体。我们说，类声明规定了类的公有部分。

类声明可以含有常数和域定义以及谓词声明。

如果一个类说明了一个构造类型 ***ConstructionType***，则它可以构造此种类型的对象。可以构造对象的类至少有一个构造器，但也可以声明多个。没有显式声明任何构造器的类，都自动地拥有缺省构造器（即 **new/0**）。

对象是通过调用类的构造器构造出来的。在初始化继承的类时也要用构造器。

类声明中所提到的一切是属于类的，而不是属于它所构造的对象的。与对象有关的一切，都要在这个类所构造的对象的构造类型中声明。

任何类声明***ClassDeclaration***都必须伴有类实现classImplementation。在类声明中声明的谓词的定义/实现是由类实现提供的。同样，由类构造的对象所支持的谓词定义也是由类实现提供的。这两种谓词都可以由子句实现，但对象谓词还可从其它类继承来实现。

需要特别指出，类声明对代码继承没有作任何说明。代码继承完全是“私事儿”，只能在类实现中说明（这与许多其它面向对象的语言不同，是要在 *implementation* 中隐藏所有的实现细节）。

如果类没有说明构造类型 ***ConstructionType***，则这个类不能产生任何对象；此时这个类的角色与其说是“类”还不如说是个模块更恰当。

```
ClassDeclaration：
  class ClassName ConstructionType-opt
      ScopeQualifications
      Sections
  end class ClassName-opt

ConstructionType：
  ：InterfaceName

ClassName：
  LowerCaseIdentifier
```

参看Generic Interfaces及Monitors。

类声明结尾处的***ClassName***（如果要写的话）必须与开头的那个完全一样。

允许用类的名字 ***ClassName*** 指定该类构造类型的接口名字 ***ConstructionType***，就是说可以这样：

```
class interfaceAndClassName：interfaceAndClassName
```

要注意，类和接口都可以声明域和常数，它们之间必须没有冲突，因为它们都是在相同的名称空间的（它们只能由相同的接口或类的名称来限定了）。

ScopeQualifications只能用open限定符。

Sections 必须是下面的种类：

- constants段
- domains段
- predicates段
- constructors段

- propertiesSection属性段
- conditional段

只有当类中说明了***ConstructionType***时，constructors段才是合法的。
在条件（conditional）段中的所有段也必须是这些种类的。

6. Implementations （实现）

一个类的实现，用于提供在该类声明中声明的谓词和构造器的定义，以及由这个类构造的对象所支持的谓词定义。

类可以私有地（即在其实现中）声明和定义比类声明中提到的更多的实体。特别是，在实现里可以声明事实数据库，而事实数据库可以用于承载类和对象的状态。

实现是一个混合的范围，它既包含了类的实现，也包含了类产生的对象的实现。类的类部分是由类的所有对象共享的，而对象部分，则仅属于各个对象。类部分和对象部分都可以含有事实与谓词，而域、函子和常数则总是属于类部分，即，它们不是属于哪个对象个体的。

缺省时，所有在一个类实现中声明的谓词和事实成员是对象成员。要声明一个类成员，必须在段关键字（如 **predicates**、**facts**）前加关键字 **class**。在这样的段中声明的所有成员是类成员。

类成员可以引用类的类部分，但**不能**引用对象部分。

而另一方面，对象成员既可以访问类的类部分也可以访问类的对象部分。

在实现的代码中，所有者对象为全体对象谓词所包容。被包容的所有者对象，可以直接由特殊的变量This来访问。

```
ClassImplementation :
  implement ClassName
      ScopeQualifications
      Sections
  end implement ClassName-opt
```

类实现结尾处的 ***ClassName***（如果要写的话）必须与类实现开头的一致。

ScopeQualifications 必须是如下的种类：

- supports限定符
- open限定符
- inherit限定符
- delegate限定符

supports 限定符后列出一个接口表，表示由该类实现私有支持的接口。

delegate 限定符委派来自接口的对象谓词功能给来自对象的谓词，它可以如同事实变量一样地存储。

Sections 必须是如下种类：

- constants段
- resolve限定符
- delegate限定符
- domains段
- predicates段
- properties属性段
- facts段
- clauses段
- conditional段

只有当类说明了构造器类型时，constructors段才是合法的。说明构造器类型的类同时也是对象的构造器，它按规定的构造类型构造对象。

例 下面这个例子，要说明类事实在类的对象间共享，而对象事实则不是共享的。
设有接口 **aa** 和类 **aa_class**：

```
interface aa
  predicates
    setClassFact : (integer Value).
    getClassFact : () -> integer.
    setObjectFact : (integer Value).
    getObjectFact : () -> integer.
end interface
class aa_class : aa
end class
```

这些谓词分别从类事实与对象事实中存取值：

```
implement aa_class
  class facts
    classFact : integer := 0.
  facts
    objectFact : integer := 0.
  clauses
    setClassFact(Value) :-
      classFact := Value.
    getClassFact() = classFact.
  clauses
    setObjectFact(Value) :-
      objectFact := Value.
    getObjectFact() = objectFact.
end implement aa_class
```

设有目标为：

```
goal
  A1 = aa_class::new(),
  A2 = aa_class::new(),
  A1:setClassFact(1),
  A1:setObjectFact(2),
  ClassFact = A2:getClassFact(),
  ObjectFact = A2:getObjectFact().
```

类事实是由所有对象共享的，所以用 **A1** 设置类事实会影响到由 **A2** 取得的值，因而，**ClassFact** 的值是 1，用 **A1** 设置的那个值。
与此不同的是，对象事实只属于各个对象。因此，在 **A1** 中设置的对象事实不会影响存储在 **A2** 中的值。这样，**ObjectFact** 的值是 0，这个值是 **A2** 初始化时的值。

Construction（构造）

本节描述对象构造，因而只涉及那些产生对象的类。
对象是通过调用构造器（*constructor*）构造出来的。而构造器则是在类的声明和实现中的 **constructors**段显式地声明的（参见缺省构造器）。
一个构造器有两个关联的谓词：

- 一个类函数，它返回新构造的对象

- 一个对象谓词，它用于初始化继承对象

相关的对象谓词用于进行对象的初始化，这个谓词只能由类本身的构造器和继承类的构造器调用（也就是基于类的初始化）。

相关的类函数是隐含定义的，也就是说在哪儿都没有它的子句。这个类函数分配内存来存放对象、进行对象内部的初始化并调用对象构造器来创建对象。最后，作为构造器的执行结果，返回构造的对象。因此，在构造器的子句被调用之前：

- 所有具有初始化表达式的对象事实变量要被初始化
- 所有具有子句的对象事实要从这些子句开始被初始化

在构造器的子句被调用之前，这个初始化也会传递地进行到所有继承的子对象。

构造器子句必须要：

- 初始化那些在进入之前没有初始化的 single 对象事实及对象事实变量
- 初始化所有子对象

构造器子句还可以做些其它的事，但它必须完成上述初始化工作，以确保对象构造后是有效的。

注意：在构造期间对象可能是无效的，要避免访问未初始化的对象部分（参看构造对象的规则）。

缺省构造器

缺省构造器不需要输入参数，名为 **new/0**。如果构造对象的类在其类声明中没有声明任何构造器，则意味着在类声明部分隐含地声明了缺省的构造器（即 **new/0**）。这也意味着每个类至少有一个构造器，因此，这样写：

```
class aaa
end class aaa
```

与下面的写法，结果是一样的：

```
class aaa
  constructors
    new : ().
end class aaa
```

显式地重新声明缺省构造器也是合法的。

不必定义（也就是说，实现）缺省构造器，如果不定义，则隐含地假设有一个没有什么影响的定义。因此，如果有一个构造器**aaa**，写：

```
implement aaa
end implement aaa
```

与写成下面这样是一样的：

```
implement aaa
  clauses
    new().
end implement aaa
```

注意，只是在下面这样的情况中，类才有一个缺省构造器：

- 这个类没有公开地声明任何构造器
- 或是声明了一个 **new/0** 作为构造器

也可以反过来说，如果是下述情况，一个类就没有缺省的构造器：

- 这个类公开地声明了构造器
- 它声明的不是以 **new/0** 作为构造器

例 设有接口 **aa**，考查下面的代码：

```
class aa_class : aa
end class
```

类 **aa_class** 没有声明构造器，所以，它隐含地声明了缺省构造器。因而，可以这样创建一个 **aa_class** 类的对象：

```
goal
  _A = aa_class::new().        % 隐含声明的缺省构造器
```

例 实现隐含声明的 **aa_class** 类的缺省构造器是合法的：

```
implement aa_class
  clauses
    new() :-
    ...
end implement
```

例 **bb_class** 类显式地声明了一个构造器，因为它不是缺省构造器，这个类就没有缺省构造器。

```
class bb_class : aa
  constructors
    newFromFile : (file File).
end class
```

例 **cc_class** 类声明了构造器 **newFromFile/1**，但同时也声明了缺省构造器 **new/0**，所以显然它有缺省构造器 **new/0**：

```
class cc_class : aa
  constructors
    new : ().                  % 缺省构造器
    newFromFile : (file File).
end class
```

私有构造器

在类实现里，还可以声明“私有的”构造器。在下面的情况中，可能是有这个需要的：

1. 当某个谓词要返回一个 *construction type* 的对象时，在类实现中就可以声明、实现并调用“私有的”构造器来创建这样的对象
2. 当某个类里声明了几个“public” 的构造器、而这些构造器中有很多相同部分时，用一个在类实现中声明的“私有的”构造器来实现这个“相同部分”也是合理的。所有这些公共的构造器的子句只需要调用这个私有的构造器来实现“相同部分”。

注意，在一个可以构造对象的类中，如果在类声明中没有声明任何构造器，则意味着隐含地声明了缺省构造器 **new/0**，这和在类实现中是否声明私有构造器无关。这样一来，可以这样做：

```
interface aa
end interface
```

```
class aa : aa
end class

implement aa
  constructors
    myCreate : ().
  clauses
    myCreate() :-
    ...
end implement
% 程序代码
  ...
  Obj = aa::new(), % 这是隐含声明的缺省的类构造器
  ...
```

子对象的构造

所有构造器都要负责把构造的对象初始化成有效的状态。为要得到这样的有效状态，所有子对象（即继承的类）也必须初始化。

子对象的初始化可以用下述二种方法之一进行：程序员调用被继承类的构造器，或是自动地调用缺省构造器。后者要求被继承类确实有一个缺省构造器，这个缺省构造器是显式声明或是隐含声明的无关紧要。

如果被继承类没有缺省构造器，就**必须**显式地调用别的构造器。被继承类构造器的缺省调用时刻是紧接着初始化事实变量和事实的子句之后，在进入到构造器子句之前。

被继承类的构造器调用是按不返回值的模式进行的。如果按返回值的版本来做调用，实际上就创建了一个对象，而不是对 This 调用构造器（参看下面的例子）。

例 这个 **bb_class** 类的实现继承自类 **aa_class**，**bb_class** 类的缺省构造器调用 **aa_class** 类的一个构造器，创建一个新的 **cc_class** 类的对象：

```
implement bb_class inherits aa_class
  clauses
    new() :-
      C = cc_class::new(),      % 创建一个cc_class的对象
      aa_class::newC(C).        % 对继承的子对象调用构造器
      ...
end implement
```

例 如果基类不是显式构造的，那它就是用缺省构造器隐含构造的，这样的话，写：

```
implement bbb
  inherits aaa
clauses
  myNew() :-
    doSomething().
end implement bbb
```

和下面的写法是一样的：

```
implement bbb
  inherits aaa
clauses
  myNew() :-
    aaa::new(),
```

```
        doSomething().
    end implement bbb
```

如果**aaa**没有缺省构造器，这当然就会出错。

注意，这个规则当然也可以和前面在缺省构造器讨论过的规则结合起来，写

```
    implement bbb
      inherits aaa
    end implement bbb
```

与写成下面的样子（根据前面缺省构造器在中讨论过的规则）是完全相同的：

```
    implement bbb
      inherits aaa
    clauses
      new().
    end implement bbb
```

与写成下面的样子（按刚讨论过的规则）也是相同的：

```
    implement bbb
      inherits aaa
    clauses
      new() :-
          aaa::new().
    end implement bbb
```

single（对象）事实的初始化

如同所有构造器必须要初始化/构造子对象一样，它也必须在第一次引用之前初始化一个对象的所有的 single 事实。

注意，single 类事实只能由子句初始化，因为它们不是与哪个对象关联的，类事实在第一个对象创建之前就可以访问了。

例 这个例子表明（1）如何用表达式初始化一个事实变量；（2）如何用子句初始化一个 single 事实（point）；（3）如何在构造器里初始化一个 single 事实；（4）在哪儿调用被继承类的缺省构造器：

```
    implement bb_class inherits aa_class
      facts
        counter : integer := 0.
        point : (integer X, integer Y) single.
        c : (cc C) single.
      clauses
        point(0, 1).
    % 创建了对象，counter和point在进入之前初始化了
    % 缺省构造器aa_class::new/0在进入之前也被调用了
        new() :-
          C = cc_class::new(),
          assert(c(C)).
        ...
    end implement
```

委托构造

除了直接在构造器中构造一个对象，这个工作也可以委托给同一个类的其它构造器。这只需要调用其它的构造器（不返回值的版本）就可以了。委托构造时，必须要确保对象真的被构造了，而且还不能重复构造。single 事实可以多次赋值，所以不会构造重复；而继承的类，只能在对象构造时进行一次初始化。

例 这个例子表明了委托构造的一个典型应用，构造器 **new/0** 用缺省值调用另一个构造器 **newFromC/1**。

```
implement aa_class
  facts
    c : (cc C) single.
  clauses
    new() :-
      C = cc_class::new(),
      newFromC(C).
    newFromC(C) :-
      assert(c(C)).
      ...
end implement
```

构造对象的规则

程序员必须保证：

- 所有子对象只被初始化/构造了一次
- 所有 single 事实被初始化了（至少一次）
- 在初始化/构造之前没有子对象被引用
- 在初始化之前没有 single 事实被使用

在编译时编译器对这些问题的检测是没有什么保证的。编译器可能会给出一个运行时确认，也可能会**不安全地**忽略这样的运行时确认。

This

对象谓词总是被应用在一个对象上，这个对象带有对象事实而且就包含在对象谓词的实现中。对象谓词可以访问到这个暗含的对象，我们称这个对象为 **This**。可以**隐式**和**显式**两种方式访问 **This**。

显式This

在每个对象谓词的每个子句中，都隐含定义了变量 **This** 并绑定于“**This**”，也就是其成员谓词正在执行的那个对象。

隐式This

在一个对象成员谓词的子句里，可以直接调用其它对象成员谓词，因为隐含地设定了**This**来进行这种操作。超类的成员也可以直接被调用，只需明确调用的方法就行（参看范围与可见性）。同样，也可以访问存储在**This**中的对象事实。

This与继承

*[**注意** 这一段的内容有可能发生变化，因为关于语言的这个问题正打算作更动。]*

This 指的对象，总是属于在使用 **This** 的那个类的，如果这个类是被其它的类继承的，仍然如此。

设接口 **aa** 声明如下：

```
interface aa
  predicates
    action : ().
    doAction : ().
end interface
```

而类 **aa_class** 声明如下：

```
class aa_class : aa
end class
```

其实现是：

```
implement aa_class
  clauses
    action() :-
      doAction(),
      This:doAction().
    doAction() :-
      write("aa_class::doAction"), nl.
end implement
```

则如下目标：

```
goal
  A = aa_class::new(),
  A:action().
```

写出来的是：

```
aa_class::doAction
aa_class::doAction
```

再来考虑如下声明的类 **bb_class**：

```
class bb_class : aa
end class
```

其实现是：

```
implement bb_class inherits aa_class
  clauses
    doAction() :-
      write("bb_class::doAction"), nl.
end implement
```

这样的目标：

```
goal
  B = bb_class::new(),
  B:action().
```

写出来的也是：

```
aa_class::doAction
aa_class::doAction
```

因为在类 **aa_class** 中（显式和隐式）的 **This** 都指的是类 **aa_class** 的对象。

现在来看另一个例子：

```
interface iName
  predicates
    className : () -> string Class.
    name: () -> string Class.
    nameThis: () -> string Class.
end interface iName

class aaa : iName
end class aaa

implement aaa
  clauses
    className() = "aaa".
  clauses
    name() = className().
  clauses
    nameThis() = This:className().
end implement aaa

class bbb : iName
end class bbb

implement bbb
  inherits aaa
  clauses
    className() = "bbb".
end implement bbb

goal
  OOO = bbb::new(),
  Class1 = OOO:name(),
  Class2 = OOO:nameThis().
```

bbb类从aaa继承了name和nameThis的定义，但再实现了className。在目标中我们创建了一个bbb对象，在它的基础上我们调用name和nameThis。

name谓词直接调用className，而nameThis调用**This**:className，它们的效果是不同的。name谓词调用的className是在aaa中定义的。但nameThis调用的className则是定义在**This**上的，其实就是bbb::className，因为**This**是一个bbb对象。

总之，**Class1**的绑定值是“**aaa**”， **Class2**而则是“**bbb**”。

像nameThis那样在父类中调用子类，是在共享的父类中实现通用功能的常用方法。而通用功能则要依靠子类的对非通用功能的提炼区分。

再看一下这个类：

```
interface iTest
  predicates
    test : () -> string Name.
end interface iTest

class ccc : iTest
end class ccc

implement ccc
  inherits aaa
  clauses
    test() = aaa::nameThis().
end implement ccc
```

ccc也继承于aaa，因此这也就意味着ccc支持接口iName（私有地）。当aaa调用**This:**className时，它调用的是ccc提供的东西，而这刚好又是继承自aaa的。

再看下面这个例子：

```
class ddd : name
end class ddd

implement ddd
  inherits bbb
  clauses
    className() = "ddd".
end implement ddd

goal
  OOO = ddd::new(),
  Class1 = OOO:name(),
  Class2 = OOO:nameThis().
```

ddd继承自bbb，而后者是继承自aaa的。当调用nameThis时其实调用的是aaa::nameThis，而它调用的是**This:**className。在这里，**This**是ddd对象，因而实际的调用成了ddd::className。

总之，对**This:**className的调用指向支持含有className的接口最终的子类。

Inherits限定符

这个限定符用于说明一个实现是继承自一个或多个类的。继承只对类的对象部分有影响。

由其它的类继承的目的，是要从那些类中继承**行为**。当**cc**类继承**aa**类，意味着**cc**类的实现自动隐含地（私有的）支持**aa**类的构造类型（接口）。当然，如果类**aa**的实现已经显式地支持了**cc**的构造类型，它们之间就没什么分别了。

因此，要注意同一个谓词，比如说**p**，不能在**cc**类的构造类型接口中声明又在被继承的**aa**类构造类型接口中声明（编译器能检测出这个问题并产生错误报告说一个谓词在两个地方作了声明）。对这个问题再稍微仔细地讨论一下。假设**cc**类有一个构造类型接口**cci**，而另一个**aa**类有一个构造类型接口**aai**。在**aai**和**cci**接口中都声明同一个谓词**p**，只要**aa**类和**cc**类是独立的，编译器就不会检测出问题。但一旦声明**cc**类继承**aa**类，则**cc**类也开始支持**aai**接口。这样一来，**cc**类就遇到了两个**p**谓词的声明，这个问题就会产生编译时的错误。要避免谓词**p**声明的这种不确定性，只能在**cci**接口声明中使用Predicates from Interface段，如下：

```
interface cci
  predicates from aai
    p(), ...
end interface cci
```

对象谓词可以被继承：如果一个类没有实现某个它的对象谓词，但它所继承的一个类实现了这个谓词，则这个谓词就可以用于这个类。

继承的类对被继承的类没有什么特权，它也只能通过该构造类型接口访问其中的对象。

继承必须是明确的。如果一个类自己定义了谓词，这不会含糊，因为这个谓词定义是显而易见的。如果只有一个被继承类支持这个谓词，这也不会有问题。但是，如果有两个或多个类支持这个谓词，是哪个类提供的定义就模糊了。此时，必须用一个resolve限定符来解决这个模糊问题（参看Resolve限定符）。

在当前类的对象谓词中可以直接调用来自被继承类的对象谓词，因为隐含使用了作为谓词所有者的内建子对象。类限定符可解决来自被继承类的对象谓词调用模糊问题。

```
InheritsQualification :
  inherits ClassName-comma-sep-list
```

Resolve限定符

已经提到过关于调用谓词的模糊问题，可以通过使用指定名称来避免。但当有继承情况出现时，这也还有问题。看下面这个例子：

```
interface aa
  predicates
    p : () procedure ().
    ...
end interface

class bb_class : aa
end class

class cc_class : aa
end class

class dd_class : aa
end class

implement dd_class inherits bb_class, cc_class
end implement
```

在这种情况下，是类**bb_class**还是类**cc_class**来给类**dd_class**提供**aa**的实现是模糊的（注意，我们说一个类实现一个接口，就意味着它对在接口里声明的谓词提供定义）。

当然可以在**dd_class**的实现中添加子句，这可以有效地解决问题。比如，下面的子句，它会从**bb_class**中“输出”谓词*p*：

```
clauses
  p() :- bb_class::p().
```

但这样一来，就并非从**bb**继承了，其实是我们的类 ***委托*** 任务给**bb_class**。

为要解决这样的模糊（真的是继承而非委托），可以使用一个**resolve**段，这个段包含一些解决方案：

```
ResolveQualification :
  resolve Resolution-comma-sep-list
```

```
Resolution :
  InterfaceResolution
  PredicateFromClassResolution
  PredicateRenameResolution
  PpredicateExternallyResolution
```

限定符resolve用于从指定的源中提供实现。

谓词解决方案

```
PredicateFromClassResolution :
  PredicateNameWithArity from ClassName
```

来自类的谓词解决方案，由指定的类提供该谓词的实现。

对一个类的谓词解决方案来说：

- 该类必须实现了要解决的谓词，这也意味着该谓词必须要与被继承的那个谓词源于同样的接口
- 该类已经出现在**inherits**段

谓词重命名解决方案

谓词重命名解决方案，是指由另一个不同名称的谓词提供该谓词的实现。这个谓词必须来自一个继承的类，其类型、模式及流样式必须要完全匹配。

```
PredicateRenameResolution :
  PredicateNameWithArity from ClassName :: PredicateName
```

接口解决方案

接口解决方案用来从一个继承的类中提供完整的接口。因而，这个解决方案就是简单地说该接口中的所有谓词应该由相应的类来解决。

```
InterfaceResolution :
  interface InterfaceName from ClassName
```

该类必须公开地支持提供解决方案的接口。

如果谓词解决方案与接口解决方案都覆盖了某个谓词名，则会使用谓词解决方案，也就是说精细度高的解决方案优先。当数个接口覆盖一个谓词时，只要它们都是由相同的类提供谓词的解决方案的，也是这样。另一方面，如果谓词的解决方案来自不同的类，就必须要用谓词解决方案来克服模糊问题。

注意，解决方案的语法不能解决谓词对不同类的不同重载问题。

例 我们可以用接口解决方案来解决前面例子中的问题。在这种情况下，我们选择aa_class的实现继承cc_class，但从bb_class继承p。

```
implement dd_class
  inherits bb_class, cc_class
  resolve
    interface aa from cc_class
```

```
        p from bb_class
    end implement
```

外部解决方案

谓词的外部解决方案，是指谓词根本就不在类本身实现，而是使用外部的库。外部解决方案只适用于类谓词，对象谓词不能用这个解决方案。

重要的是，调用协议、链接名及参数类型要符合实现库的要求。

私有与公用谓词都可以使用外部解决方案。

```
    PredicateExternallyResolution : PredicateNameWithArity externally
```

动态外部解决方案

谓词的外部解决方案也提供了动态地从DLL中加载私有和公用类谓词的语法，其为：

```
    PredicateExternallyResolutionFromDLL :
      PredicateNameWithArity externally from DllNameWithPath

    DllNameWithPath :
      StringLiteral
```

如果在DLL ***DllNameWithPath***中没有谓词predicateNameWithArity，动态加载会让程序运行下去，直到程序实际调用这个没有找到的谓词，这时就会出现运行时错误。***DllNameWithPath***是程序运行的机器上的DLL的路径，它可以是绝对的也可以是相对的。例如，如果需要的dll是在应用程序所在位置的上一级目录中，就可以用“**../DllName**”。参看MSDN中的动态库搜索次序。

终结

对象一旦无法为程序所用，就可以被终结了。在本语言的语义中没有准确地说对象什么时候被终结，能保证的是：只要程序还能访问某个对象，它就不会被终结。实际上对象的终结，是当它被垃圾收集器废掉的时候。终结是构造的对立面，是把对象从内存中清除出去。

类也可以实现一个终结器，这是一个谓词，一个对象终结时调用它（在对象被清出内存之前）。

终结器是一个过程，没有参数，没有返回值，其名称是**finalize**。这个谓词是隐含声明的，不能在程序中直接调用。

终结器的主要目的是为了能释放外部资源，但是并没有限制它能够做什么。使用终结器要小心，前面说过，调用它们的时间不能完全确定，因而也就很难预测调用它时整个程序的状态。

注意，不可能从在终结器中的对象撤消对象事实，因为终结过程是自动进行的。

程序终止前，所有对象都要被终结（除非出现像电源故障这样的异常）。

例 这个例子中，用一个终结器来保证正确地关闭数据库的连接：

```
    implement aa_class
      facts
        connection : databaseConnection.
      clauses
        finalize() :-
          connection:close().
      ...
```

```
end implement aa_class
```

Delegate限定符

delegate段包含有一些委托项：

```
DelegateQualification :
  delegate Delegation-comma-sep-list
```

```
Delegation :
  PredicateDelegation
  InterfaceDelegation
```

委托限定符用于将对象谓词的实现委托给指定的源。

委托限定符分为两类，Predicate Delegation（谓词委托）和Interface Delegation（接口委托）。接口委托用于将一个接口中声明的全体对象谓词的实现委托给另一个对象的实现，存储为事实变量。这就是说，接口委托就是将该接口声明的所有谓词的实现委托给另一个对象的实现，存储在事实变量中。

委托段与相应的（谓词/接口）解决方案段相似，只不过需要委托事实变量来保存构造的类的对象，而不是从类中继承。

谓词委托

对象谓词的委托，是指该谓词的功能委托给由事实变量***FactVariable_of_InterfaceType***中指定的对象中的谓词。

```
PredicateDelegation :
  PredicateNameWithArity to FactVariable_of_InterfaceType
```

要委托一个谓词给一个由事实变量传递的对象：

- 事实变量***FactVariable_of_InterfaceType***必须具有声明谓词predicateNameWithArity的接口的类型（或子类型）
- 对象支持的接口必须已经构造好了，并且已经与***FactVariable_of_InterfaceType***事实变量相关联。

看一下下面的例子：

```
interface a
  predicates
    p1 : ().
    p2 : ().
end interface

interface aa
  supports a
end interface

class bb_class : a
end class

class cc_class : a
end class

class dd_class : aa
```

```
  constructors
    new : (a First, a Second).
end class

implement dd_class
  delegate p1/0 to fv1, p2/0 to fv2
  facts
    fv1 : a.
    fv2 : a.
  clauses
    new(I,J):-
    fv1 := I,
    fv2 := J.
end implement
```

后面，它就可以构造a类型的对象，并将其赋予事实变量fv1和fv2，来定义对象，这个对象的类就是我们实际要委托定义p1和p2功能的类。来看：

```
goal
  O_bb = bb_class::new(),
  O_cc = cc_class::new(),
  O_dd = dd_class::new(O_bb, O_cc),
  O_dd : p1(),                          % 这个p1来自O_bb对象
  O_dd : p2().                          % 这个p2来自O_cc对象
```

其实，Visual Prolog的委托与在dd_class的实现中添加子句明确地指出要从哪个类的对象“输出”谓词的功能，效果是一样的。对上面这个例子，也就是在dd_class的实现中添加下面的子句：

```
clauses
  p1() :- fv1:p1().
```

接口委托

如果需要指定在***InterfaceName***接口中声明的所有谓词的功能都委托给来自相同继承类的对象谓词，就可以用接口委托：

```
InterfaceDelegation :
  interface InterfaceName to FactVariable_of_InterfaceType
```

接口委托就是将在接口***InterfaceName***中声明的所有谓词的功能委托给存储为事实变量***FactVariable_of_InterfaceType***的对象。对象应该已经与***FactVariable_of_InterfaceType***事实变量相关联，而这个事实变量的类型应该是***InterfaceName***的（或是其子类型）。

要将接口委托给一个由事实变量传递的对象：

- 事实变量***FactVariable_of_InterfaceType***必须是接口***InterfaceName***的类型（或是其子类型）
- 对象支持的接口必须已经构造，并且已经与事实变量***FactVariable_of_InterfaceType***相关联。

谓词委托的优先级高于接口委托。如果一个谓词两个委托都做了，也就是做了谓词委托，而它又是在一个接口中声明的，这个接口又做了委托，这时，将实现高优先级的谓词委托。

7. 通用接口和类

接口和类可以用类型参数进行参数化，以便应用在不同关联的不同实例中。
这一节的内容，应该视为以下各节内容的扩展：

- Interfaces
- Classes
- Implementations

从编程上看，使用通用接口和类就是声明**参数化**的对象事实并实现对这些事实的操作。请看下面这个队列的例子。

例：Queue

考虑如下的接口：

```
interface queue_integer
   predicates
        insert : (integer Value).
        tryGet : () -> integer Value.
end interface queue_integer
```

一个此种类型的对象就是一个整数队列，如果把上面的integer替换成sting，则将得到串队列类型的描述。
通用接口可以用来描述所有这类的接口，而且只需要单一的接口定义：

```
interface queue{@Elem}
    predicates
        insert : (@Elem Value).
        tryGet : () -> @Elem Value determ.
end interface queue
```

@Elem是一个范围类型变量（scope type variable），@来区分于本地类型变量。
queue{integer}就表示整数队列，而queue{string}就表示串队列，以此类推。
可以这样声明通用的queue类：

```
class queueClass{@Elem} : queue{@Elem}
end class queueClass
```

queueClass{@Elem}可以为任意的@Elem实例构造queue{@Elem}类型的对象。
其实现可以是这样的：

```
implement queueClass{@Elem}
   facts
        queue_fact : (@Elem Value).
   clauses
        insert(Value) :-
            assert(queue_fact(Value)).
```

```
        clauses
            tryGet() = Value :-
                retract(queue_fact(Value)),
                !.
    end implement queueClass
```

下面的代码片段说明了如何创建一个整数队列并在其中插入一个元素：

```
    ..., Q = queueClass{integer}::new(), Q:insert(17), ...
```

显式地指明类型并不是必需的，其实编译器会从该关联中推得它的类型：

```
    ..., Q = queueClass::new(), Q:insert(17), ...
```

因为要把17插入到其中去，编译器就会明白**Q**必定是个整数队列。

通用接口（Generic Interfaces）

语法

通用接口有一个类型参数的表：

```
InterfaceDeclaration :
    interface InterfaceName { ScopeTypeVariable-comma-sep-list-opt }
        ScopeQualifications
    Sections
    end interface InterfaceName-opt
ScopeTypeVariable :
    @ UppercaseName
```

在接口的任意声明/定义中都可以使用范围类型参数。

语义

一个通用接口，定义了以实际类型实例化类型参数时所能得到的所有接口。起始处的范围类型参数是约束整个接口的。

限制

接口结尾的名称后不要再带参数：

```
    interface xxx{@A}
        ...
    end interface xxx      % 这里不再带参数
```

对不同元维的接口（在同一个名称空间中）使用相同的名称是非法的：

```
    interface xxx % xxx/0
        ...
    end interface xxx
```

```
interface xxx{@A, @B} % 错误：若干类、接口和/或名称空间用了同样的名称“xxx”
    ...
end interface xxx
```

参数可以用在支持的接口中：

```
interface xxx{@P} supports yyy{@P}        % 合法： @P 是绑定的
    ...
end interface xxx

interface xxx supports yyy{@P}            % 非法： @P 必须是绑定的
    ...
end interface xxx
```

支持的接口可以用任意类型表达式实例化（只要参数是绑定的）：

```
interface xxx{@A} supports yyy{integer, @A*}
    ...
```

通用类

语法

通用类有一个类型参数的表，接口类型的构造对象就使用这些参数。

ClassDeclaration :
    class *ClassName* **{** *ScopeTypeVariable*-**comma-sep-list-opt** **}** *ConstructionType*
        ScopeQualifications
    Sections
    end class *ClassName*-**opt**

ScopeTypeVariable :
    @ *UppercaseName*

ConstructionType :
    : *TypeExpression*

构造类型必须是通用型的（也就是一个通用接口）。

语义

通用类声明了一个带有通用构造器的类，它所构造对象的类型由所用的构造器确定。

限制

类结尾处的名称后不要再带参数：

```
class xxx{@A} : xxx{@AAA}
    ...
end class xxx          % 这里不要有参数
```

对不同元维的类（在同一个名称空间中）使用相同的名称是非法的：

```
class xxx % xxx/0
    ...
end class xxx

class xxx{@A} : yyy{@A} %错误：若干类、接口和/或名称空间用了同样的名称“xxx”
    ...
end class xxx
```

若类与接口用了相同的名称，则该类必须构造那个接口的对象。

```
interface xxx{@A}
    ...
end interface xxx

class xxx{@Q, @P} : object %错误：若干类、接口和/或名称空间用了同样的名称“xxx”
    ...
end class xxx
```

构造类型中的参数等必须绑定：

```
class bbb : xxx{@Q} % 错误：在类型表达式中使用了自由的参数@Q
    ...
```

来自该类的所有参数必须在构造类型中都得到应用：

```
class xxx{@P} : object % 错误：未使用的类型参数@P
    ...
```

在类声明中，范围参数只能用于构造器、域及常数中：

```
class xxx{@P} : xxx{@P}
    domains
        list = @P*. % 合法
    constants
        empty : @P* = []. % 合法
    constructors
        new   : (@P Init).   % 合法
    predicates
        p : (@P X). % 错误：类实体中使用了范围参数@P
end class xxx
```

通用实现

语法

通用实现有一个类型参数的表：

ClassImplementation :
    implement *ClassName* **{** *ScopeTypeVariable*-**comma-sep-list-opt** **}**
        ScopeQualifications
    Sections
    end implement *ClassName*-**opt**

ScopeTypeVariable :
    @ *UppercaseName*

语义

通用类声明了一个带有通用构造器的类，它所构造对象的类型由所用的构造器确定。

限制

结尾处的名称后不要再带参数：

```
implement xxx{@A}
    ...
end implement xxx       % 这里不要再带参数
```

所带参数必须与相应的类声明的参数相同并且顺序也要相同：

```
class xxx{@A} : aaa{@A}
    ...
end class xxx

implement xxx{@B} % 错误：参数@B与声明中的@A不一样
    ...
end implement xxx
```

在类实现中，范围参数可以用在构造器、域、常数及对象实体中（如对象事实、对象谓词和对象属性）。

```
implement xxx{@P}
    domains
        list = @P*. % 合法
    constants
        empty : @P* = []. % 合法
    constructors
        new   : (@P Init).   % 合法
    predicates
        op : (@P X). % 合法
    class predicates
        cp : (@P X). % 错误：在类实体中使用了范围参数@P
    facts
        ofct : (@P). % 合法
    class facts
        cfct : (@P). %错误：在类实体中使用了范围参数@P
    ...
end implement xxx
```

8. Monitors（监控程序）

监控程序是一种同步两个或多个线程的语言构造，这些线程使用共享的资源（通常是硬件设备或一组变量）。编译器会透明插入锁定和解锁代码到精心设计的过程中，而不需要程序员来考虑同步问题。

Visual Prolog的监控程序入口可由guard谓词（条件）来控制。

语法

监控程序的接口和监控程序的类都是范围：

```
Scope : one of
    ...
    MonitorInterface
    MonitorClass
    ...
```

监控程序的接口是通过在常规的接口定义前加关键字monitor来定义的：

```
MonitorInterface :
    monitor IntertfaceDefinition
```

而监控程序的类是通过在常规的类声明前加关键字monitor来声明的：

```
MonitorClass :
    monitor ClassDeclaration
```

监控程序的类与接口不能声明multi和nondeterm的谓词成员。

限制

- 常规的接口不能支持监控程序接口；
- 监控程序的类不能构造对象；
- 继承监控程序（如继承一个实现监控程序接口的类）是不合法的。

语义

在监控程序中声明的谓词和特性参数（properties）是该监控程序的入口。线程通过入口进入监控程序直到从那个入口离开一直在监控之中。一个时刻只允许有一个线程在监控程序中，所以每个入口都是一个排它性的保护。

其语义最简单的理解就是把它视为一个程序转换（它也正是这样实现的）。来看下面这个理论上的例子：

```
monitor class mmmm
predicates
    e1 : (a1 A1).
    e2 : (a2 A2).
    ...
```

```
        en : (an An).
    end class mmmm
    %--------------------------------------
    implement mmmm
    clauses
        e1(A1) :- <B1>.
    clauses
        e2(A2) :- <B2>.
    ...
    clauses
        en(An) :- <Bn>.
    end implement mmmm
```

其中<B1>、<B2>、…<Bn>是子句体。这个代码相当于下面的“普通”代码：

```
    class mmmm
    predicates
        e1 : (a1 A1).
        e2 : (a2 A2).
        ...
        en : (an An).
    end class mmmm
    %--------------------------------------
    implement mmmm
    class facts
        monitorRegion : mutex := mutex::create(false).
    clauses
        e1(A1) :-
         _W = monitorRegion:wait(),
         try
               <B1>
         finally
              monitorRegion:release()
         end try.
    clauses
        e2(A2) :-
         _W = monitorRegion:wait(),
         try
               <B2>
         finally
               monitorRegion:release()
         end try.
    ...
    clauses
        en(An) :-
         _W = monitorRegion:wait(),
         try
               <Bn>
         finally
               monitorRegion:release()
         end try.
    end implement mmmm
```

这样，每个监控程序类都由一个互斥体扩展了，而互斥体用来创建包围各入口体的排它区域。监控程序对象的代码也与此类似，只不过互斥对象是属于该对象的。

Guards（看守）

设想一个受监控的队列：某些线程（生产者）在队列中插入元素而另外的线程（消费者）从队列中拣取元素。当然，如果队列是空的，就不能从队列中取出了。

我们可以用监控程序来实现这个队列：取出的入口可以是确定的，如果队列空就失败。但消费者会一直检测队列直到有一个元素可以取出。这样的检测耗用系统资源，因而通常期望避免进行检测。这样的问题就可以用guard谓词来解决。

在实现中每个入口可以与一个guard联系起来，guard是作为一个特殊的guard子句添加在入口的其它子句之前。

```
Clause : one of
    ...
    GuardClause.
```

```
GuardClause : one of
    LowerCaseIdentifier guard LowerCaseIdentifier .
    LowerCaseIdentifier guard AnonymousPredicate .
```

下面的例子中guard是谓词的名字：

```
clauses
    remove guard remove_guard.
    remove() = ...
```

guard也可以是匿名谓词，如下例：

```
clauses
    remove guard { :- element_fact(_), ! }.
    remove() = ...
```

创建监控程序时会对guard谓词求值，对监控程序类来说这是在程序开始之时，对对象谓词来说则是紧随对象构建之后。当一个线程离开监控程序时也会对guard谓词求值，除了上述时刻都不会再对guard谓词求值了。

如果某个guard谓词成功，则相应的入口就打开，失败则该入口就是关闭的。只能从打开的入口进入。

下面的例子，在队列类的remove操作上用guard谓词解决取出问题：

```
monitor class queue
predicates
    insert : (integer Element).
predicates
    remove : () -> integer Element.
end class queue
%----------------------------------------
implement queue
class facts
    element_fact : (integer Element) nondeterm.
clauses
    insert(Element) :-
        assert(element_fact(Element)).
clauses
    remove guard remove_guard.
    remove() = Element :-
        retract(element_fact(Element)),
```

```
                !;
                common_exception::raise_error(common_exception::classInfo,
                predicate_name(),"guard要确保队列不是空的。").
        predicates
            remove_quard : () determ.
        clauses
            remove_guard() :-
                element_fact(_),
                !.
        end implement queue
```

注意，remove是procedure的，因为调用remove的线程会一直等待直到有元素给它。当队列中有元素时，看守谓词remove_guard会成功。

这样，每当一个线程离开监控程序时就会对remove_guard求值，而element_fact事实数据库也只能由在监控程序中的线程修改。因此，guard值在所有时间（例如，在监控程序中没有线程的时候）都保持可察觉。对看守来说，保证这种“保持可察觉”是很重要的。

guard谓词是在前面提到过的转换中处理的。队列的例子对下面这个没有监控程序（monitor-free）的代码同样有效：

```
class queue
predicates
    insert : (integer Element).
predicates
    remove : () -> integer Element.
end class queue
%----------------------------------------
implement queue
class facts
    monitorRegion : mutex := mutex::create(false).
    remove_guard_event : event := event::create(true, toBoolean(remove_guard())).
    element_fact : (integer Element) nondeterm.
clauses
    insert(Element) :-
      _W = monitorRegion:wait(),
      try
            assert(element_fact(Element))
      finally
            setGuardEvents(),
            monitorRegion:release()
      end try.
clauses
    remove() = Element :-
      _W = syncObject::waitAll([monitorRegion, remove_guard_event]),
      try
            retract(element_fact(Element)),
            !;
             common_exception::raise_error(common_exception::classInfo,
             predicate_name(),"guard要确保队列不是空的。")
      finally
            setGuardEvents(),
            monitorRegion:release()
      end try.
class predicates
    remove_guard : () determ.
clauses
    remove_guard() :-
        element_fact(_),
```

```
            !.
    class predicates
        setGuardEvents : ().
    clauses
        setGuardEvents() :-
            remove_guard_event:setSignaled(toBoolean(remove_guard())).
    end implement queue
```

为每个guard谓词都创建了一个事件，事件在guard谓词成功时被置为*signaled*。前面已经说过，这是在监控程序创建的时候以及一个谓词离开监控程序的时候（在离开排它区域之前）。

在进入一个入口时线程既要等待**monitorRegion**又要等待guard事件已是signalled状态。

在上面的代码中，类本身的初始化及guard事件的初始化进行的顺序是不确定的，不过实际上所有类/对象初始化进行完了之后再进行guard事件的初始化是有保证的。

实际用例

这一节给出几个例子，可以看到在这些情况下使用监控程序是很便利的。

写日志文件

若干线程要把登录信息写到同一个日志文件中去。

```
monitor class log
properties
    logStream : outputStream.
predicates
    write : (...).
end class log
%----------------------------------------
implement log
class facts
    logStream : outputStream := erroneous.
clauses
    write(...) :-
        logStream:write(time::new():formatShortDate(), ": "),
        logStream:write(...),
        logStream:nl().
end implement log
```

这个monitor确保了各线程写入的登录信息行不会混在一起，流的变更只发生在写完各信息行之后。

共享输出流

这个监控程序可以用于线程保护输出流的操作：

```
monitor interface outputStream_sync
   supports outputStream
end interface outputStream_sync
%----------------------------------------
class outputStream_sync : outputStream_sync
constructors
    new : (outputStream Stream).
end class outputStream_sync

%----------------------------------------
```

```
implement outputStream_sync
   delegate interface outputStream to stream
facts
    stream : outputStream.
clauses
    new(Stream) :- stream := Stream.
end implement outputStream_sync
```

不过，要知道像下面这样的代码：

```
clauses
    write(...) :-
        logStream:write(time::new():formatShortDate(), ": "),
        logStream:write(...),
        logStream:nl().
```

是由三个分开的操作构成的，所以实际情况仍有可能是两个线程先写时间而一个线程在写“...”，等等。

队列

前面的队列很好，不过实际上创建一个队列对象可能要更好。用通用接口可以创建一个非常通用的队列：

```
monitor interface queue{@Elem}
predicates
    enqueue : (@Elem Value).
predicates
    dequeue : () -> @Elem Value.
end interface queue
%----------------------------------------
class queue{@Elem} : queue{@Elem}
end class queue
%----------------------------------------
implement queue{@Elem}
facts
    value_fact : (@Elem Value).
clauses
    enqueue(V) :-
        assert(value_fact(V)).
clauses
    dequeue guard { value_fact(_), ! }.
    dequeue() = V :-
        value_fact(V),
        !.
    dequeue() = V :-
        common_exception::raise_error(....).
end implement queue
```

PFC中已经包含了一个很类似的类**monitorQueue**。

参考内容

- wikipedia:Monitor (synchronization)

9. Namespaces（名称空间）

名称空间用于无需使用长的冷僻名称就可避免名称冲突。在不同名称空间的名字是不会产生冲突的，但有的时候可能需要指定名称空间以解决引用模糊问题。

名称空间的声明与定义隐含地使用***NamespaceEntrance***：

```
NamespaceEntrance:
  namespace NamespaceIdentifier
```

```
NamespaceIdentifier: one of
  LowercaseIdentifier
  LowercaseIdentifier \ NamespaceIdentifier
```

简单地说，***NamespaceIdentifier***是一系列用反斜杠分隔的小写标识。

Namespace的入口与区域

名称空间入口把源文件分割成一些namespace regions（名称空间区域）。一个名称空间入口标志着一个名称空间区域的起始，这个区域结束于下一个名称空间入口或是文件的结尾。每个文件开始于root namespace（根名称空间）。

名称空间区域不会受**#include**指令的影响，也就是说：

- 在一个**#include**-文件中的名称空间入口不会改变包括文件的名称空间区域。
- 任何文件都起始于根名称空间（即使它是被包括在另一个文件中的内部名称空间也是这样）。

任何在一个名称空间区域中汇合的接口、类和实现，属于那个名称空间。

例

```
class aaa
end class aaa

namespace xxx

class bbb
end class bbb

namespace xxx\yyy

class ccc
end class ccc
```

这个文件分成了三个区域（假设这是一个完整的文件）。第一个区域是根名称空间（\），第二个区域是属于xxx名称空间的，而第三个区域是属于xxx\yyy名称空间的。相应地，类aaa属于根名称空间，类bbb属于xxx名称空间而类ccc属于xxx\yyy名称空间。

namespaces中的引用名

如果ccc是在名称空间xxx\yyy中的一个类，那么ccc的全称（full name）就是\xxx\yyy\ccc。最前

面的反斜杠表示是从根名称空间开始的。类/接口总是可以用全称来唯一性地引用。

Open namespaces

全称有时并不方便，我们可以用开放的名称空间来使用简短的名称。

```
ScopeQualification: one of
  OpenQualification
  ...
OpenQualification: one of
  open NamespaceIdentifier \
  ...
```

开放的名称空间是由结尾的反斜杠与开放的类/接口相区别的。

例

```
class aaa
  open xxx\yyy\
  ...
end class aaa
```

名称空间xxx\yyy在aaa中是开放的。

当一个名称空间是开放的时，其全称部分就可以省掉。如，全称是\xxx\yyy\zzz\ccc::ddd的一个域，可以在aaa中用zzz\ccc::ddd来引用，因为xxx\yyy是开放的。

注意，略写的名称不是用反斜杠开头的，用反斜杠开头的名称总是全称。

属于某个范围（如接口/类/实现）的名称空间，在那个范围内部是（隐含地）开放的。

10. 程序段

以下各段用于在范围内声明和定义实体。

```
Section：
  ConstantsSection      常数段
  DomainsSection        域段
  PredicatesSection     谓词段
  ConstructorsSection   构造器段
  PropertiesSection     属性段
  FactsSection          事实段
  ClausesSection        子句段
  ConditionalSection    条件段
```

上面的段并不会都出现在所有类型的范围内，详细情况请参考接口、类声明和类实现。
条件段在条件编译一节中叙述。

11. Domains（域）

Domains段

域段在当前范围中定义一组域（参看接口、类声明和类实现）。

```
DomainsSection:
  domains DomainDefinition-dot-term-list-opt
```

Domain定义

域定义在当前范围中对一个命名了的域作详细说明。

```
DomainDefinition:
  DomainName FormalTypeParameterList-opt = TypeExpression
```

如果右边的域表示一个接口或一个复合域，则定义的域与类型表达式是同义的（等价的）。否则，定义的域就是类型表达式表示的域的子类型。这里，域名***DomainName***必须是小写标识。

有的地方必须用域名而不是类型表达式：

- 作为形式参数类型的声明时；
- 作为常数或事实变量的类型时；
- 作为表域的类型时。

域表达式

域表达式用于在域定义中说明域的类型：

```
DomainExpression:
  TypeName
  CompoundDomain
  ListDomain
  PredicateDomain
  IntegralDomain
  RealDomain
  TypeVariable
  ScopeTypeVariable
  TypeApplication
```

类型表达式

全套的***DomainExpression***只能用于域定义中。***TypeExpression***则是这个全套表达式的一个子集，可以用在许多其它的场合。

```
TypeExpression:
  TypeName
  ListDomain
  TypeVariable
  ScopeTypeVariable
  TypeApplication
```

类型名

类型名是一个 *接口* 的名字或是一个 *值域* 的名字。*值域* 是指其元素是非易变的（不可更改的）域。这里我们可以说，属于相应 *接口* 名的域的对象具有易变的状态，而其它的域则是非易变的。因此，实际上值类型就是除了对象类型而外所有其它的东西。显而易见，类型名是按已有域名称来表示类型的。

```
TypeName:
  InterfaceName
  DomainName
  ClassQualifiedDomainName

InterfaceName:
  LowercaseIdentifier

DomainName:
  LowercaseIdentifier

ClassQualifiedDomainName:
  ClassName::DomainName

ClassName:
  LowercaseIdentifier
```

这里，***InterfaceName***是接口名，***DomainName***是值域名而***ClassName***是类名。

例

```
domains
 newDomain1 = existingDomain.
 newDomain2 = myInterface.
```

在上面这个例子中，分别用域名**existingDomain**和接口名**myInterface**定义了两个新的域。

复合域

复合域（也称为代数数据类型）用于表示表、树及其它树状结构的值。它的简单形式常用于表示结构和枚举值。复合域可以递归定义，可以互递归或间接递归。

```
CompoundDomain:
  Alignment-opt FunctorAlternative-semicolon-sep-list

Alignment:
  align IntegralConstantExpression
```

这里的***IntegralConstantExpression***是一个表达式，在编译时必须能计算得出整数值。

复合域的声明宣布了一个函子选项表，还带有可选的对齐因子（alignment），它必须是**1**、**2**或**4**。

如果复合域包含一个函子选项，则可以视其为结构，它的表示方法与C语言的相应结构是二进制兼容的。

```
FunctorAlternative:
  FunctorName FunctorName ( FormalArgument-comma-sep-list-opt )
```

这里的***FunctorName***是函子选项名称，应该用小写标识。

```
FormalArgument:
  TypeExpression ArgumentName-opt
```

这里的***ArgumentName***可以是任意大写标识。编译器会忽略它。

复合域对其它任何域都没有子类型的关系。如果一个域定义和一个复合域的相同，则这两个域是同义类型的而不是子类型。也就是说，这两个域是不同名称相同类型的。

例

```
domains
  t1 = ff(); gg(integer, t1).
```

t1是一个有两个选项的复合域，第一个选项是零元函子**ff**，而第二个选项是二元函子**gg**，它带有一个integer和一个域项**t1**本身作为参数。因此，域**t1**是递归定义的。

下面这些表达式都是域**t1**的项：

```
ff()
gg(77, ff())
gg(33, gg(44, gg(55, ff())))
```

例

```
domains
  t1 = ff(); gg(t2).
  t2 = hh(t1, t1).
```

t1是有两个选项的复合域，第一个选项是零元函子**ff**，而第二个选项是一元函子**gg**，它以域项**t2**为参数。**t2**也是一个复合域，有一个选项函子**hh**，这个函子以两个**t1**项为参数。因而，域**t1**和**t2**是互递归的。下面这些项都是域**t1**的：

```
ff()
gg(hh(ff(), ff()))
gg(hh(gg(hh(ff(), ff())), ff()))
ggg(hh(ff(), g(hh(ff(), ff()))))
gg(hh(gg(hh(ff(), ff())), gg(hh(ff(), ff()))))
```

例 这个例子中，**t1**和**t2**是同义类型。

```
domains
  t1 = f(); g(integer).
  t2 = t1.
```

通常没有必要在零元函子后使用空的圆括号。但在域定义中，如果仅有一个零元函子，就必须用空的圆括号来与同义/子类型定义相区分。

例 **t1**是一个仅有一个零元函子的复合域，而**t2**是定义为与**t1**同义的。

```
domains
  t1 = f().
  t2 = t1.
```

List Domains（表域）

表域表示某个域中一序列的值。因此，所有在表**T**中的元素都必须是**T**类型的。

```
ListDomain:
  TypeName *
```

T* 是**T**元素表的类型。

下面一些语法应用于表：

```
ListExpression:
  [ Term-comma-sep-list-opt ]
  [ Term-comma-sep-list | Tail ]
```

```
Tail:
  Term
```

这里***Tail***是其值为***ListDomain***类型的项。每个***Term***都应该是typeName类型的。

实际上，表只不过是有两个函子的复合域：表示空表的[]和表示表头**HD**与表尾**TL**的混合固定函子**[HD|TL]**。表头必须是基本元素类型，而表尾必须是相应类型的表。

表不过是语法便利化的结果。

[E1, E2, E3, ..., En | L]是**[E1 |[E2 |[...[En | L]...]]]**的简化。

[E1, E2, E3, ..., En]是**[E1, E2, E3, ..., En |[]]**的简化，而后者又是**[E1 |[E2 |[...[En | []]...]]]**的简化。

Predicate Domains（谓词域）

谓词域的值是带有相同特征的谓词，而“相同特征”指的是相同的参数与返回类型、相同的流样式和相同（或更强的）谓词模式。

返回值的谓词称为***function***（函数），而不返回值的谓词有时也称为**常规**谓词，以强调其不是一个函数。

```
PredicateDomain:
  ( FormalArgument-comma-sep-list-opt ) ReturnArgument-opt
  PredicateModeAndFlow-list-opt CallingConvention-opt
```

```
FormalArgument:
  PredicateArgumentType VariableName-opt
  Ellipsis
```

```
ReturnArgument:
  -> FormalArgument
```

```
VariableName:
  UpperCaseIdentifier
```

谓词域可以用***Ellipsis***（省略号）参数作为用逗号分隔的形式参数表中最后的形式参数。

谓词域可以用***AnonymousIdentifier***作为一个predicateArgumentType，表示这个参数可以是

任意类型的。

目前，带省略号的谓词域只能用在谓词声明中。

在域定义中的谓词域最多只能声明一个流样式。

```
PredicateModeAndFlow:
  PredicateMode-opt
  FlowPattern-list-opt
```

Predicate Mode（谓词模式）

规定的谓词模式适用于其后流样式表中的各个成员。

```
PredicateMode: one of
  erroneous
  failure
  procedure
  determ
  multi
  nondeterm
```

谓词模式可以用以下集合来描述：

```
erroneous = {}
failure = {Fail}
procedure = {Succeed}
determ = {Fail, Succeed}
multi = {Succeed, BacktrackPoint}
nondeterm = {Fail, Succeed, BacktrackPoint}
```

集合中有 *Fail* 就意味着这个谓词可以失败，有 *succeed* 意味着可以成功，有 *BacktrackPoint* 意味着可以由有效的回溯点返回。

如果一个集合（比如说**failure**）是另一个集合（比如说**nondeterm**）的子集，则称其模式比另一个强，也就是**failure**强于**nondeterm**。

谓词域实际上包含了带有适当类型和流样式谓词的指定的及更强的所有模式。

对构造器声明谓词模式是非法的，构造器的模式永远是**procedure**。

缺省的谓词模式是**procedure**。

Flow Pattern（流样式）

流样式定义参数的输入/输出流向。参数可以与函子域相结合，使得单个的参数一部分是输入而另一部分是输出。

流样式由一个流序列构成，每个流对应于一个参数（第一个流对应第一个参数，...等等）。

```
FlowPattern:
  ( Flow-comma-sep-list-opt ) AnyFlow
```

```
Flow: one of
  i
  o
  FunctorFlow
  ListFlow
  Ellipsis
```

省略流必须与省略参数相对应，因而它只能是流样式中的最后一个流。

```
Ellipsis:
  ...
```

函子流***FunctorFlow***说明一个函子及其各成分的流。当然，函子必须在相应参数的域中。

```
FunctorFlow:
  FunctorName ( Flow-comma-sep-list-opt )
```

函子流声明中不能含有省略流。

表流与函子流相似，只不过是采用了与表域相同的语法加工。

```
ListFlow:
  [ Flow-comma-sep-list-opt ListFlowTail-opt]
```

```
ListFlowTail:
  | Flow
```

表流也不能含有省略流。

声明谓词时流样式可以缺省。无论是局部谓词（实现内部声明的）还是公有谓词（接口或类中声明的），缺省的流样式均表示所有参数都是输入。

特殊流样式**anyflow**只能在局部谓词的声明（即在类实现内部的谓词声明）中使用。它表示准确的流样式在编译时确定。如果其前可选的谓词模式省略了，表示谓词模式是**procedure**。

例

```
domains
  pp1 = (integer Argument1).
```

pp1是一个谓词域。这个具有**pp1**类型的谓词有一个integer参数。因为没有写出流样式，表示这个参数是输入，又因为没有写出谓词模式，表示谓词是**procedure**。

例

```
domains
  pp2 = (integer Argument1) -> integer ReturnType.
```

pp2类型的谓词有一个integer参数并返回integer类型的值。因而，**pp2**是一个函数域，具有**pp2**类型的谓词实际上是函数。因为没有说明流样式，所以参数是输入的，谓词模式也没提，故谓词是**procedure**的。

例

```
predicates
  ppp : (integer Argument1, integer Argument2) determ (o,i) (i,o) nondeterm (o,o).
```

谓词**ppp**有两个integer参数，它有三种不同的流样式：**(o,i)**和**(i,o)**是**determ**的，**(o,o)**是

nondeterm的。

调用约定

调用约定确定了参数等如何传递给谓词，它也确定了如何由谓词名得到链接名。

```
CallingConvention:
  language CallingConventionKind

CallingConventionKind: one of
  c thiscall stdcall apicall prolog
```

如果没有说明调用约定，则使用**prolog**约定，它是Prolog谓词使用的标准约定。

c调用约定遵从C/C++的标准调用约定。谓词的链接名是谓词名前加下划线（_）。

thiscall调用约定遵从C++的标准调用约定，用于虚拟函数。它也使用**c**链接名策略，但有时可能会使用不同的参数与栈处理规则。这个调用约定只能用于对象谓词。

stdcall调用约定使用**c**的链接名策略，但参数与栈的处理规则有不同。下表说明了**stdcall**调用约定的实现：

特征	实现
参数传递顺序	从右到左
参数传递约定	除复合域项传递，都是传递的值。因此，它不能应用于参数数量可变的谓词
栈的维护职责	被调用的谓词从栈中弹出自己的参数
名字的装饰	在谓词名前的一个下划线（_）
大小写转换约定	不做谓词名的大小写转换

apicall调用约定使用与**stdcall**一样的参数和栈处理规则，但为了方便MS Windows API函数的调用使用了大多数MS Windows API函数调用所用的命名约定。根据该约定，谓词的链接名构造方法如下：

- 在谓词名前加下划线（_）；
- 谓词名的首字母改为大写；
- 如果参数和返回类型是ANSI的，加后缀“**A**”，如果是Unicode的，加后缀“**W**”，不确定的谓词什么都不加；
- 加“**@**”后缀；
- 后面附加推入栈中的字节数。

例

```
predicates
  predicateName : (integer, string) language apicall
```

这个谓词的参数类型表明它是一个Unicode谓词（因为string是Unicode串域的）。一个integer和一个string各占调用栈的4字节，因而，它的链接名就是：

```
_PredicateNameW@8
```

如果**apicall**与“**as**”结构一并使用，在“**as**”结构中说明的名称使用与之相同的方法。

apicall只能直接使用在谓词声明中，不能用在谓词域定义中。谓词域定义中必须替代使用**stdcall**。以**apicall**调用约定声明的谓词不能有子句，也不能是没有明确DLL名称的外部解决方案。

下表比较了**c**、**thiscall**、**apicall**及**stdcall**调用约定的实现（这里没有讨论**prolog**调用约定的特殊实现）：

类型	清栈职责	谓词名大小写转换	链接名构造方法
c	**调用**谓词从栈中弹出参数	不转换	谓词名前加下划线（_）
thiscall	除了用寄存器传递的隐式**this**参数外，**调用**谓词从栈中弹出参数	不转换	使用**c**链接名策略
stdcall	**被调用**谓词从栈中弹出自己的参数	不转换	谓词名前加下划线（_）
apicall	**被调用**谓词从栈中弹出自己的参数	谓词名首字母转换为大写	谓词名前加下划线（_），首字母换为大写，加“**A**”或“**W**”后缀，加“**@**”后缀，加参数字节数（十进制）

Visaul Prolog谓词域的概念覆盖了类和对象的成员。类成员的处理是直接向前的，但对象成员的处理需要注意。对象谓词的调用将“回退”到拥有该成员的对象关联中。

例

设有如下声明：

```
interface actionEventSource
  domains
    actionListener = (actionEventSource Source) procedure (i).
  predicates
    addActionListener : (actionListener Listener) procedure (i).
  ...
end interface
```

再假设有一个**button_class**类支持**actionEventSource**。点击按钮时传递事件。在**myDialog_class**类中实现一个对话框，创建一个按钮并监听它的动作事件以作出反应：

```
implement myDialog_class
  clauses
    new() :-
      OkButton = button_class::new(...),
      OkButton:addActionListener(onOk),
      ...
  facts
    okPressed : () determ.
  predicates
    onOk : actionListener.
  clauses
    onOk(Source) :-
      assert(okPressed()).
end implement
```

在这个例子中，关键在**onOk**是一个对象成员，当按了按钮后，注册了的**onOk**的调用会将我们带回到拥有**onOk**的对象。也就是说，我们能访问对象事实**okPressed**，这样我们才能插入事实。

Format Strings（格式串）

对谓词的格式参数，可以用**formatstring**属性标记成格式串。格式串中可以有原态字符，这样的字符是不加修饰地原样打印的；还有以百分号（%）开头的格式域。如果%后面的是未知字符（不是格式符），则百分号和这个字符也会原样打印。

例

```
predicates
  writef : (string Format [formatstring], ...).
```

格式域的构成是：

[-][0][width][.precision][type]

上述所有组份都是可选的。

连字符[-]表示这个域是右对齐的；缺省是左对齐。如果宽度值[width]没有设置，或是实际字符数比宽度值还要大，则没有效果。

宽度前的零[0]表示对值要添加零直到达到最小宽度。如果零和连字符一同出现，零就被忽略了。

宽度[width]是正的十进制数，表示域段的最小宽度。如果实际字符数小于这个值，就要在值的前面（若设置了‘-’则是在后面）添加空格字符。如果实际字符数比这个值大，就没有什么改变。

点之后带无符号十进制数[.precision]表示浮点数精度，也可以表示从一个串里最多打印出的字符数。

[type]规定了缺省之外其它一些格式。如，在该域可以用一个规定符表示一个整数要格式化为一个无符号数。可用值为：

f	将实数格式化为定点十进制数，如123.4，0.004321。这是实数的缺省格式。
e	将实数格式化为指数表示法，如1.234e+002，4.321e-003。
g	将实数格式化为f和e格式中较短的那种，但当指数值小于-4或大于等于精度时总是用e格式。尾零都要被截去。
d或D	格式化为有符号十进制数。
u或U	格式化为无符号整数。
x或X	格式化为十六进制数。
o或O	格式化为八进制数。
c	格式化为一个字符。
B	格式化为Visual Prolog的二进制类型。
R	格式化为数据库索引数。
P	格式化为过程参量。
s	格式化为串。

Integral Domains（整数域）

整数域用于表示整数，它分成两个主要类型：有符号数和无符号数。整数域也有不同的表示范围，预定义的域integer和unsigned表示有符号的和无符号的整数，是处理器架构（如32位机器上的32位，等）的固定表示长度。

```
IntegralDomain:
  DomainName-opt IntegralDomainProperties
```

如果在***IntegralDomainPropertie***前面有***DomainName***，那么它本身必须是整数域的，接下

来的域是这个域的一个子类型。此时，***IntegralDomainProperties***不能违反作为一个子类型的可能性，也就是说，数值表示不能超范围，不能改变比特位数。

```
IntegralDomainProperties:
  IntegralSizeDescription IntegralRangeDescription-opt
  IntegralRangeDescription IntegralSizeDescription-opt

IntegralSizeDescription:
  bitsize DomainSize

DomainSize:
  IntegralConstantExpression
```

IntegralSizeDescription（整数尺度描述）说明该整数域的***DomainSize***大小，以比特为单位。编译器会以不小于这个规定的比特位数实现这个整数域。***DomainSize***的值应该是正数并且不能超过编译器支持的最大值。

如果省略了整数尺度描述，就使用和父域一样的尺度。如果没有父域，则使用处理器的本身的尺度。

```
IntegralRangeDescription:
  [ MinimalBoundary-opt .. MaximalBoundary-opt ]

MinimalBoundary:
  IntegralConstantExpression

MaximalBoundary:
  IntegralConstantExpression
```

IntegralRangeDescription（整数界限描述）说明该整数域最小值***MinimalBoundary***和最大值***MaximalBoundary***。如果省略了，就使用父域的界限。如果没有父域，就使用***DomainSize***来确定相应的最小值和最大值。

注意，最小值不能大于最大值，也就是：

```
MinimalBoundary <= MaximalBoundary
```

还有，最小值***MinimalBoundary***和最大值***MaximalBoundary***还需要满足规定的**bitsize**隐含的限制。

域的比特尺度***DomainSize***的值以及最小值***MinimalBoundary***和最大值***MaximalBoundary***的值在编译时必须是可以求值的。

Real Domains（实数域）

实数域用于表示带有小数的数（即浮点数），它可以表示很大和很小的数。内建的域*real*具有处理器架构的本征精度（或是编译器给出的精度）。

```
RealDomain:
  DomainName-opt RealDomainProperties
```

如果在***RealDomainProperties***前面有***DomainName***，那么它本身必须是实数域的，接下来的域是这个域的一个子类型。此时，***RealDomainProperties***不能违反作为一个子类型的可能性，也就是说，数值表示不能超范围，不能增加精度。

```
RealDomainProperties: one of
  RealPrecisionDescription RealRangeDescription-opt
  RealRangeDescription RealPrecisionDescription

RealPrecisionDescription:
  digits IntegralConstantExpression
```

RealPrecisionDescription（实数精度描述）说明该实数域的精度，以十进制位数为单位。如果省略了它，就使用和父域一样的精度。如果没有父域，则使用处理器的本征精度或是编译器给定的精度（在Visual Prolog v.6编译器中是15位数）。精度有由编译器确定的上下界，如果使用超出这个界限的数，只能得到处理器（编译器）提供的精度。

```
RealRangeDescription:
  [ MinimalRealBoundary-opt .. MaximalRealBoundary-opt ]

MinimalRealBoundary:
 RealConstantExpression

MaximalRealBoundary:
 RealConstantExpression
```

这里的***RealConstantExpression***是一个表达式，在编译时它被计算出一个浮点值。也就是说，在编译时该实数域的精度和界限必须是可计算的。

RealRangeDescription（实数界限描述）说明该实数域最小值和最大值。如果省略了，就使用父域的界限。如果没有父域，就使用最大可用精度。

注意，最小值不能大于最大值，也就是：

```
MinimalBoundary <= MaximalBoundary
```

Generic Domains（通用域）

这节内容是通用域的形式化语法，对通用域更完整的介绍请参看本书中《通用接口与类》及网站资料Objects and Polymorphism（对象与多态性）。

```
FormalTypeParameterList:
  TypeVariable-comma-sep-list-opt
```

*FormalTypeParameterList*是一个类型变量的表：

```
TypeVariable:
  UpperCaseIdentifier
```

TypeVariable是一个大写标识。在域声明中，类型变量必须在***FormalTypeParameterList***中域定义的左边绑定。

在谓词声明中，所有自由的类型变量都会被隐含地绑定并限定范围于该谓词声明。

```
TypeApplication:
  TypeName {TypeExpression-comma-sep-list-opt }
```

TypeApplication是将一个类型名应用于一个类型的表。类型名必须是generic的，形式化的类型参数数量必须与类型表达式数量相匹配。

通用类型和根类型

Visual Prolog使用一些内部的类型，称为通用类型和根类型。

通用类型

一个文字的数字**1**没有任何特殊的类型，可以是任意值为的**1**类型，包括实数类型。我们称这样的**1**是通用类型的。通用类型表示可以是任意的能表示其值的类型。

算术运算返回的是通用类型。

根类型

算术运算所使用的操作数是非常自由的：可以把两个任意整数域的整数相加。

我们说作为参数的算术操作数具有根类型。整数的根类型是任意整数类型的超类型（不管声明中说不说）。因此，任何整数类型都可以转换成整数根类型，而且由于算术运算是按根类型进行的，这也就意味着它们可以使用任意整数域。

根类型的实际数值及有什么样的操作数是工具库的事情了，不在本文档中讨论。

12. 常数

Constants段

constants段在当前范围内定义一组常数。

```
ConstantsSection :
  constants ConstantDefinition-dot-term-list-opt
```

常数定义

常数定义规定命名了的常数，它的类型，它的值。

```
ConstantDefinition: one of
  ConstantName = ConstantValue
  ConstantName : TypeName = ConstantValue
```

```
ConstantName:
  LowerCaseIdentifier
```

ConstantValue应该为表达式，在编译时应该能计算得到值并且具有相应域的类型。***ConstantName***应该是小写标识。

只有对下列内建的域才可以省略类型名：

1. 数值（即整数或实数）常数。在这种情况下，对常数使用相应匿名数字域（详细请参见数字域）。
2. 二进制常数。
3. 串常数。
4. 字符常数。

例

```
constants
  my_char = 'a'.
  true_const : boolean = true.
  binaryFileName = "mybin".
  myBinary = #bininclude(binaryFileName).
```

13. Predicates（谓词）

Predicates段

谓词段在当前范围内声明一组对象或类谓词。

```
PredicatesSection：
  class-opt predicates PredicateDeclaration-dot-term-list-opt
```

关键字**class**只能用在类实现中。对于谓词，在接口中声明的总是对象谓词，而在类声明中声明的，总是类谓词。

Predicate Declarations（谓词声明）

谓词声明用于在一个范围内声明谓词，在这个范围内，该谓词是可见的。当谓词在接口定义中声明时，就意味着相应类型的对象必须支持这些谓词。当谓词在类声明中声明时，就意味着该类公有地支持所声明的谓词。而如果谓词是在类实现中声明的，则意味着谓词是局部的。不管在哪种情况下，必须有谓词的相应定义。

```
PredicateDeclaration：
  PredicateName：PredicateDomain LinkName-opt
  PredicateName：PredicateDomainName LinkName-opt
```

```
LinkName：
  as StringLiteral
```

```
PredicateName：
  LowerCaseIdentifier
```

这里，predicateDomainName（谓词域名）是在域段中声明的谓词域的名称。

谓词声明规定了谓词的名称、它的类型、模式、流（参见谓词域）及可选的链接名。

只有类谓词才可以有链接名，如果没有说明链接名则会依据调用约定由谓词名产生相应的链接名。如果调用约定是apicall，则**as**子句说明的链接名也要加修饰。如果这种修饰不可用，就使用stdcall调用约定。

链接名的缀饰加工

有时链接名必须有_...@N的缀饰，但apicall给的缺省名是错的。**decoratedA**和**decoratedW**可用于控制这类缀饰加工问题：

```
predicates
  myPredicate：(string X) language stdcall as decorated.
```

在这种情况下，链接名会是“**_MyPredicate@4**”，而apicall会使它成为“**_MyPredicateW@4**”。

```
predicates
  myPredicate：(string X) language stdcall as decoratedA.
```

在这种情况下，链接名会是“**_MyPredicateA@4**”，而apicall会使它成为“**_MyPredicate@4**”。

```
predicates
  myPredicate : (string X) language stdcall as decoratedW.
```

在这种情况下，链接名会是“**_MyPredicateW@4**”，而apicall会使它成为“**_MyPredicate@4**”。

上述所有情况中，都是把名称由xxxx变为_Xxxx，并加上后缀@N。第一种情况不用后缀的字母，第二种情况总是后缀A，而第三种情况总是后缀W。这就是说，编程者负责确定使用需要的后缀，不过不需要操心计算参数的字节数和起始的“_X”（加下划线和把首字母改为大写）。

Constructors段

constructors段声明一组构造器。构造器属于**constructors**段所在的范围（参见类声明和类实现）。

```
ConstructorsSection :
  constructors ConstructorDeclaration-dot-term-list-opt
```

构造器段只能在构造对象的类的声明和实现中出现。

Constructor声明

构造器声明说明类的命名了的构造器。
一个构造器实际上有两个相关的谓词：

- 一个类函数，这个函数能返回一个新构造的对象，
- 一个对象谓词，该谓词用来初始化继承的对象。

相关的构造器对象谓词用于进行对象的初始化。这个谓词只能在该类本身由该构造器调用，或是由继承这个类的一个类的构造器调用（即基类初始化）。

```
ConstructorDeclaration :
  ConstructorName : PredicateDomain
```

不能对构造器说明谓词模式，构造器总是**procedure**模式的。

例 看一下如下的类：

```
class test_class : test
  constructors
    new : (integer Argument).
end class test_class
```

相关的类谓词有如下特征：

```
class predicates
  new : (integer) -> test.
```

而相关的对象谓词特征如下：

```
        predicates
          new : (integer).
```

再来看一下如下的实现：

```
        implement test2_class inherits test_class
          clauses
            new() :-
              test_class::new(7),          %在"This"上调用基类构造器
              p(test_class::new(8)).       %  创建基类一个新对象并把它传递给p(...)
          ...
```

对**test_class::new**的第一个调用并不返回一个值，因此它是一个对构造器的非函数**对象**版本的调用，也就是说，它是在**This**上的基类构造器的一个调用。

而第二个调用则**确实**返回一个值，因此它是对构造器**类**函数版本的一个调用，也就是说，它创建一个新的对象。

Predicates from Interface（来自接口的谓词）

一个接口可以支持其它接口的一个子集，这是通过在predicates from段声明谓词实现的。**predicates from**段列出接口及所有支持的谓词。对谓词的说明可以是谓词的名称或名称加元维。

一个接口支持另外接口的子集时，它既不是这个另外的接口的子类，也不是它的超类。

对**predicates from**段来说，重要的一点是：所有列出的谓词都保留在原来的接口中，因此：

- 来自原来接口的谓词不会有支持冲突；
- 它们可以作为谓词由原来的接口中继承。

```
        PredicatesFromInterface :
          predicates from InterfaceName PredicateNameWithArity-comma-sep-list-opt
```

PredicatesFromInterface只能用在接口定义中。

例

```
        interface aaa
          predicates
            ppp : ().
            qqq : ().
        end interface aaa

        interface bbb
          predicates from aaa
            ppp
          predicates
            rrr : ().
        end interface bbb

        interface ccc supports aaa, bbb
        end interface ccc
```

尽管**aaa**和**bbb**都各声明了一个谓词**ppp**，**ccc**还是可以无冲突地支持这两者，因为在所有情况下**ppp**都有**aaa**作为源接口。

例

```
interface aaa
  predicates
    ppp : ().
    qqq : ().
end interface aaa

interface bbb
  predicates from aaa
    ppp
  predicates
    rrr : ().
end interface bbb

class aaa_class : aaa
end class aaa_class

class bbb_class : bbb
end class bbb_class

implement aaa_class inherits bbb_class
  clauses
    qqq().
end implement aaa_class
```

aaa_class可以由**bbb_class**类继承**ppp**，因为**ppp**在两个类中有**aaa**作为源接口。

Arity（元维）

一个谓词带有N个参数时，就称其是N元（*N-ary*）的，或说元维是N（arity *N*）的。不同元维的谓词是不同的谓词，即使它们的名称相同也是这样。

在大多数情况下，谓词的元维是清楚的，可以从上下文中看出来。但有些时候就不明显了，比如在谓词的from段里以及在使用resolve限定符时就可能会出现这样的问题。

为了能够区分谓词的不同元维，在**predicates from**段里及使用resolve限定符时，可以选用谓词名称加元维的方法来说明。可以使用如下的方法：

- ***Name/N***表示元维是N的本原谓词*Name*（即不是一个函数），
- ***Name/N->***表示元维是N的函数*Name*，
- ***Name/N...***表示带有N个参数后跟一个***Ellipsis***（省略符）参数（即参数数量可变）的本原谓词*Name*。（省略号可以在谓词及谓词域的声明中作为最后一个形式参数使用，此时它表示所声明的谓词（谓词域）有可变数量的参数。省略流必须与省略参数相匹配，故此也只能是流样式中的最后一个流。）
- ***Name/N...->***表示一个带有N个参数后跟一个省略参数的函数*Name*。

```
PredicateNameWithArity :
  PredicateName Arity-opt
```

```
Arity : one of
  / IntegerLiteral Ellipsis-opt
  / IntegerLiteral Ellipsis-opt ->
```

在*Name/0...*和*Name/0...->*的情况中，零是可以省去的，因而也可以分别写成*Name/...*和

Name/...->。

programPoint（程序点）

程序点是表示子句特定点的一个值，它包含有类的名称、谓词的名称、行号及行中的位置。程序点的域是在core类中定义的。

程序点主要是由异常机制使用的，表示异常产生点及程序继续进行的位置。但它的使用并不仅限于此。

编译器通过programPoint属性来支持程序点。可以按如下方式加在谓词声明中：

```
predicates
    raiseAnException : (integer X) [programPoint].
```

添加这个属性其实是声明了两个谓词，一个在上面提到了，还有一个是**raiseAnException_explicit**，它带有两个参数，除了**raiseAnException**的参数外还将**programPoint**作为第一个参数：

```
predicates
  raiseAnException : (integer X).
predicates
  raiseAnException_explicit : (programPoint ProgramPoint, integer X).
```

调用raiseAnException时，编译器会创建程序点并实际调用raiseAnException_explicit。

例

```
clauses
    test() :-
        raiseAnException(17).
```

其实际相当于：

```
clauses
    test() :-
        raiseAnException_explicit(programPoint(...), 17).
```

其中的程序点对应于**test**谓词中调用**raiseAnException**的点。

如果知道程序点，就可以直接使用程序点调用显式谓词。

例

```
clauses
    raiseAnExceptio17_explicit(ProgramPoint) :-
        raiseAnException_explicit(ProgramPoint, 17).
```

通常情况下，如同在上面这个例子中一样，显式谓词会用得到的程序点再嵌套地调用另一个显式谓词以使用原态的调用形式。

这样的代码情形也是通常形式，就是说，调用**raiseAnException**或**raiseAnException_explicit**都将导致调用**raiseAnException_explicit**。所以，只有后者才是需要实现的谓词。而事实上，为那个非显式的、永远不会被调用的谓词声明子句也是不合法的。

还有一个内建的谓词**programPoint/0->**，它会返回对应于它被调用点的**programPoint**。

小结一下：

- 用programPoint属性的谓词声明实际上会声明两个谓词，一个是非显式的，一个是显式的；
- 对非显式谓词的调用会导致以调用点为附加参数的显式谓词的调用；
- 只有显式谓词需要实现。

程序点的引入，大大简化了Visual Prolog 7.3及以前版本中的异常处理机制。比如， **classInfo**谓词不需要了（尽管它在Visual Prolog 7.4及7.5中还是可以用的，但也很容易迁移）。

14. Properties（属性）

属性是关联于类和对象的命名了的值。其实，它不过是对属性值进行get/set的谓词的语法便利工具。也就是在这个意义上说，属性是某种常用编程样式的一种语言具体化。

属性段

属性段声明在当前范围内对象或类的属性。

```
PropertiesSection :
    class-opt properties PropertyDeclaration-dot-term-list-opt
```

class关键字只能在类实现内使用，因为：

- 在接口中的声明的属性总是对象的属性，而
- 在类声明中声明的属性总是类属性。

声明

```
PropertyDeclaration :
  PropertyName : PropertyType FlowPattern-list-opt.

FlowPattern: one of
  (i)
  (o)
```

如果具有(o)流模式，就可以get这个属性的值；而如果具有(i)流模式，就可以set这个属性的值。如果没有声明流模式，则假定其既有(i)流模式又有(o)流模式，因而对这样的属性既可以set值又可以get值。

尽管同时声明(i)和(o)是合法的，但在实际使用中还是建议对set+get的情况省略流模式的声明。比如：

```
properties
    durationO : real (o). % 仅get属性
    durationI : real (i). % 仅set属性
    durationIO : real (i) (o). % 全属性，既可set又可get。
    duration : real. % 同上，这样做更好些。
```

以后，我们就使用如下方式：

```
interface ip
  properties
    duration : real.
    initialized : boolean(o).
    scale : real.
end interface
```

duration和scale可以get+set属性，而initialized只能get属性。

属性与事实变量的使用相同。可以用范围名称或对象来修饰限制属性。

设**X**是一个支持接口ip的对象：

```
X:duration := 5,
if true = X:initialized then ... else ... end if,
X:scale := 2.56,
....
```

在支持ip的类的实现内部，可以如同它们是事实一样来访问它们：

```
duration := 5,
if true = initialized then ... else ... end if,
scale := 2.56,
....
```

Implementation（实现）

属性是通过定义一个获取值的函数及一个设置值的谓词来实现的。
如：

```
clauses
 % 获取属性duration值的函数的实现
  duration() = duration_fact.

clauses
 % 设置属性duration值的谓词的实现
  duration(D):-
      duration_fact := D / scale.
```

还有一种实现方法是把一个相同名字的事实变量用作属性：

```
Facts
 % 用事实变量实现属性初始化
  initialized : boolean := false.
 % scale属性是由事实变量实现的
  scale : real := 1.2
```

在这种情况下，编译器会为get和set谓词提供隐含的子句实现。对上例的两个事实变量，提供的子句相当于这样：

```
clauses
  % 对initialized属性的隐含的get子句
  initialized() = initialized.
clauses
  % 对scale属性的隐含的get子句
  scale() = scale.
clauses
  % 对scale属性的隐含的set子句
  scale(V) :-
    scale := V.
```

但在程序中声明这样的子句是不合法的，后面还会说到。
不能既有设置和获取谓词又有相同名称的事实，这意味着一个属性要么是由编程者提供子句来实现要么是用事实来实现，不能混搭。例如，我们想在属性duration的值发生变化时发生changed事件。我们

必须用谓词来实现这个属性，而给事实变量另起一个名字来存放这个值：

```
properties
  duration : integer.
facts
  duration_fact : integer.
clauses
  duration() = duration_fact.
clauses
  duration(D) :-
    OldDuration = duration_fact,
    duration_fact := D,
    OldDuration <> D,
    !,
    sendChanged().
  duration(_D).
```

不能把duration当谓词来用（这不奇怪，因为它们不是声明为谓词而是声明为属性的；它只是实现属性的存和取的方法的）。但是，谓词的名称它占用了，所以不能用duration\1或duration\0->再声明谓词。

如上所述，属性总是由get/set谓词实现的，即使程序现实是用事实变量也是这样。

实现的细节

当访问一个在接口中声明的属性时，关于它的实现不能做任何假设，因而也就不得不认为它是用谓词实现的。如果程序员是把这个属性当事实定义的，编译器就必须产生适当的谓词，当通过一个接口来访问这个属性时，就使用它们。而在实现内部，定义为事实的属性就可以当成事实来访问。

限制读写

有的时候，想让一个属性只能读或只能写。假设我们用i/o范式声明它们：

```
duration : integer (o).          % 一个只读的属性
duration : integer (i).          % 一个只写的属性
duration : integer (o) (i).      % 一个正常的属性，可以读也可以写，与什么都不说明一样
duration : integer.              % 与上面声明的一样。.
```

如果属性声明为只读，就可以只声明取（get）谓词，同样，如果是声明为只写就可以只声明设置（set）谓词。

当然，如果属性是当成事实来实现的，在实现内部就可以把它当成一个普通的事实来使用。

Properties from Interface（来自接口的属性）

一个接口可以支持另一个接口的子集，这是通过在properties from段声明这些属性实现的。**properties from**段列出接口及所支持的属性。

一个接口支持另外接口的子集时，它既不是这个另外的接口的子类，也不是它的超类。

对**properties from**段来说，重要的一点是：所有列出的属性都保留在原来的接口中，因此：

- 来自原接口的属性不存在support的冲突，
- 它们可以作为属性由原来的接口中继承。

```
PropertiesFromInterface :
  properties from InterfaceName PpropertyName-comma-sep-list-opt
```

PropertiesFromInterface只能用在接口定义中。

例

```
interface aaa
  properties
    pp : integer.
    qq : boolean.
end interface aaa

interface bbb
  properties from aaa
    pp
  properties
    rr : string.
end interface bbb

interface ccc supports aaa, bbb
end interface ccc
```

尽管**aaa**和**bbb**都各声明了一个属性**pp**，**ccc**还是可以无冲突地支持这两者，因为在所有情况下**pp**都有**aaa**作为原接口。

例

```
interface aaa
  properties
    pp : integer.
    qq : boolean.
end interface aaa

interface bbb
  properties from aaa
    pp
  properties
    rr : string.
end interface bbb

class aaa_class : aaa
end class aaa_class

class bbb_class : bbb
end class bbb_class

implement aaa_class inherits bbb_class
  facts
    pp_fact(): integer.
  clauses
    pp()= pp_fact-3.
  clauses
    pp(D):- pp_fact:=D+3.
end implement aaa_class
```

aaa_class可以由**bbb_class**类继承**pp**，因为**pp**在两个类中有**aaa**作为原接口。

15. Facts（事实）

Fact段

facts段声明一个事实数据库，它由一些事实构成。事实数据库和事实属于当前范围。

事实数据库可是是类一级的，也可以是对象一级的。

事实段只能在类实现中声明。

如果事实数据库有名称，也就隐含地附加定义了一个复合域。这个域具有与事实段相同的名称，并具有与事实段中的事实相对应的函子。

如果**facts**段有名称，这个名字就表示了内建域**factDB**的一个值。**save**和**consult**谓词可以接纳这个域的值。

```
FactsSection :
  class-opt facts FactsSectionName-opt FactDeclaration-dot-term-list-opt

FactsSectionName :
  - LowerCaseIdentifier
```

Fact声明

事实声明说明一个事实数据库的事实。事实声明可以是声明的一个事实变量，也可以是一个函子事实。

```
FactDeclaration :
  FactVariableDeclaration
  FactFunctorDeclaration

FactFunctorDeclaration :
  FactName : ( Argument-comma-sep-list-opt ) FactMode-opt

FactName :
  LowerCaseIdentifier
```

事实函子的声明，缺省时具有**nondeterm**的事实模式。

事实函子可以经子句段初始化。这种情况下，子句中的值应该是表达式，在编译时应能得到确切的值。

```
FactMode : one of
  determ nondeterm single
```

如果模式是**single**，则这个事实总是有一个且只能有一个值，***assert***谓词会把旧值以新值替代掉。谓词***retract***不适用于single事实。

如果模式是**nondeterm**，则该事实可以有0、1或任意个值。如果模式是**determ**，则事实只能有0或1个值。如果事实有0个值，则任何对它的读访问都会失败。

事实变量的声明

一个事实变量，就如同只有一个参数的**single**函子事实。不过，语法上它是一个可变量（即可以赋值）。

```
FactVariableDeclaration :
  FactVariableName : Domain InitialValue-opt
```

```
InitialValue :
  := ConstantValue
  := erroneous
```

```
FactVariableName :
  LowerCaseIdentifier
```

常数值***ConstantValue***应该是（***Domain***类型的）一个项，它在编译时应该能得到确切的值。只有当事实变量在一个构造器里被初始化时，常数值才可以省去。而类事实变量总是要有初始常数值。注意，关键字**erroneous**可以当作一个值赋给一个事实变量。下面两种情况都是允许的：

```
facts
  thisWin : vpiDomains::windowHandle := erroneous.
clauses
  p() :- thisWin := erroneous.
```

赋值成**erroneous**，是想要在发生了代码错误地使用了没有初始化的事实变量时，给出明确的运行时错误。

Facts

事实只能在一个类的实现中声明，因而，也只能在这个实现中引用。因此，事实的范围是它的声明所在的实现。但是，对象事实的生命期是它所属的对象的生命期，同样，类事实的生命期是从程序的开始一直到程序结束。

例 下面的类声明了一个对象事实**objectFact**和一个类事实**classFact**：

```
implement aaa_class
  facts
    objectFact : (integer Value) determ.
  class facts
    classFact : (integer Value) determ.
  ...
end implement aaa_class
```

16. Clauses（子句）

Clauses段

clauses段由一些子句构成，它包含了谓词的实现或事实的初始化值。

一个子句段可以包含多个谓词及事实的子句。另一方面，一个（相同名称及元维）谓词/事实的所有子句在一个子句段中必须集中在一起，不能间隔有别的谓词/事实的子句。

```
ClausesSection :
  clauses Clause-dot-term-list-opt
```

参见**Monitor**中的**Guard**。

Clauses

子句用于定义谓词。一个谓词由一组子句来定义，每个子句依次执行，直到它们中的一个***succeeds***（成功），或是直到再也没有其它剩下的子句。如果没有子句成功，那么这个谓词就***fails***（失败）了。

如果一个子句**成功**了，而此时在该谓词中还剩有其它相关的子句，程序控制就能在以后再***backtrack***（回溯）到这个谓词的这些子句上以寻求其它的解。因此，一个谓词可以**失败**，可以**成功**，甚至可以**成功**多次。

一个子句有头部及可选的体部。

当一个谓词被调用时，它的子句依次（从上到下）被检测。对每个子句来说，头部要与调用参数进行合一，如果合一成功，有体部时才执行体部。只有在头部和体部都**成功**时，子句才**成功**，否则就**失败**了。

子句是由头部及一个可选的体部构成的。

```
Clause :
  ClauseHead ReturnValue-opt ClauseBody-opt .

ClauseHead :
  LowercaseIdentifier ( Term-comma-sep-list-opt )

ReturnValue :
  = Term

ClauseBody :
  :- Term
```

参见**Monitor**中的**Guard**。

Goal Section（目标段）

goal段是程序的入口。程序开始时就从**goal**开始执行，**goal**执行完，程序就退出了。

```
GoalSection :
  goal Term.
```

goal段是由子句体构成的，这个段定义了自己的范围，因此所有的调用都应包括类限定符。

目标必须是**procedure**模式的。

17. Terms（项）

本节描述项以及项和子句的执行/计算是如何进行的。

从语义上说，有两类项：**公式（*formulas*）**和**表达式（*expressions*）**。

- 表达式代表值，如数字7。
- 公式代表逻辑说明，如“数字7大于数字3”。

而依语法来看，这两类其实有很大的重叠，因此语法上把这两类合并成项（*terms*）。

下面对***Term***的定义是简化了的，因为按它包含的语法构造来讲有些是不合法的。比如，**! + !**就是不合法的写法。不过，这种简洁的语法描述与对语言概念、类型系统及下面描述的算子层次的直觉理解相结合，还是非常有用的。

```
Term:
  ( Term )
  Literal
  Variable
  Identifier
  MemberAccess
  PredicateCall
  PredicateExpression
  UnaryOperator Term
  Term Operator Term
  Cut
  Ellipsis
  FactvariableAssignment
```

Backtracking（回溯）

Prolog程序的求值就是对目标的一个解的搜寻。对解的搜索的每一步都有可能**成功**或**失败**。程序执行过程中的某个点上，对搜索解会有多种可能，在这种选择点上就会创建一个回溯点（***backtrack point***）。回溯点是一种记录，它包含有程序的状态及还未执行的选择的指针。如果先前的一个选择没有能够得到解（也就是失败了），程序就会**回溯**到记录的回溯点上来，因而可以恢复程序的状态再追寻其它的选择。后面会要描述这个机理并用例子来详细说明。

Literals（文字）

文字具有通用类型。

```
Literal:
  IntegerLiteral
  RealLiteral
  CharacterLiteral
  StringLiteral
  BinaryLiteral
  ListLiteral
  CompoundDomainLiteral
```

参看**词汇元素**中的文字。

Variables（变量）

Visual Prolog中的变量其实是不可变的：一经绑定为一个值时它们就一直保有这个值，但回溯可以为变量解除绑定以恢复程序的先前状态。

一个变量（在合一和匹配时）可以被绑定，如果已经被绑定了，就用它绑定的值进行解算。

变量名以一个大写字母或一个下划线开头，后跟字母（可以是大写也可以是小写）、数字和下划线序列（称为大写标识）：

```
Variable:
  UppercaseIdentifer
```

下面这些都是合法的变量名：

```
My_first_correct_variable_name
_
_Sales_10_11_86
```

而下面这些则是不合法的：

```
1stattempt
second_attempt
```

名字是一个下划线（即_）的变量称为**匿名**变量，它用于形式上需要但具体的值我们并不关心的场合。每次出现的匿名变量都是独立的，也就是说，哪怕在一个子句中使用了多次匿名变量，它们之间也是没有关系的。

名字用一个下划线开头的变量并不是匿名变量，但它的值仍是我们不关心而忽略的。如果它的值实际上没有被忽略，编译器会给出一个警告。

Prolog的变量是局限于它所在的子句的，即：如果两个子句都含有变量**X**，这两个**X**变量是独立不相关的。

当一个变量还没有与一个项联系在一起时，称这个变量是**自由**的；而当它与一个项合一后，则称其为**绑定**的或实例化了的。

Visual Prolog编译器对名称不区分除首字母而外的大小写，**SourceCode**和**SOURCECODE**这两个变量名指的是同一个变量。

Identifier（标识）

```
Identifier:
  MemberName
  GlobalScopeMembername
  ScopeQualifiedMemberName

MemberName:
  LowerCaseIdentifier
```

标识用于指称命名了的实体（如类、接口、常数、域、谓词、事实，等等）。

标识只能是小写标识（也就是一个小写字母后跟字母、数字和下划线字符等的序列）。

多个实体名称可以相同。因此，可能需要用指定范围来限定小写标识名，或者说明名称是全局的。

例 下面这些是无限定的标识：

```
integer
```

```
mainExe
myPredicate
```

全局实体的访问

在Visual Prolog中，全局实体只有内建的域、谓词和常数。全局名称在任何范围内都是可以直接访问的，但也存在全局名称被一个局部名称或引入名称屏蔽的情况。此时，全局实体可以用一个双冒号“::”(不需要前缀类/接口名称)来修饰。双冒号可以用于任何实体，但最重要的使用场合是用接口名称说明形式参数的类型。

```
GlobalScopeMemberName:
  :: MemberName
```

例 内建域integer是定义为全局范围的，但为避免含糊或强调这个特殊的域，也可以使用其全局范围成员名称：

```
::integer
```

类/接口成员的访问

类与接口的静态成员的访问是通过带类名称（及可选的名称范围前缀）的限定符进行的：

```
ScopeQualifiedMemberName
  NamespacePrefix-opt ScopeName :: MemberName

NamespacePrefix:
  NamespaceIdentifier-opt \

ScopeName:
  LowercaseIdentifier
```

ScopeName是定义/声明该名称的类或接口的名字。
Namespace的前缀在名称空间中的引用名中作了说明。
有些名称的访问可以不要限定符，参看范围与可见性。

Predicate Call（谓词调用）

谓词调用的形式是：

```
PredicateCall:
  Term ( Term-comma-sep-list-opt )
```

第一项必须是一个可以求得谓词类型的值的表达式。通常，它是一个类中一个谓词的名称，或是对一个对象的谓词成员求值的表达式。

要注意，有的谓词是返回值的，而有些则不会返回值。返回值的谓词是一个表达式，这样的谓词调用常称为一个**函数**调用。常规地，谓词是返回值的。

谓词的调用是通过应用参数于该谓词来进行的。谓词必须具有与参数自由/绑定状态相匹配的流样式。

大多数谓词是由一组子句定义的，但有的谓词是由语言内建的，还有的是在外部的DLL中定义的（可能还是另外的编程语言）。

当调用谓词时，各个子句依次执行直到其中的一个**成功**，或是再没有可执行的子句。如果没有子句成

功则谓词就**失败**了。

如果一个子句成功了而还剩有子句，就会有一个**回溯点**创建在下一个相关的子句上。

这样，谓词可以**失败**，**成功**，甚至成功多次。

每个子句有头部及可选的体部。

调用谓词时，子句被依次（从上到下）求值。每个子句在头部的参数与调用的参数进行合一。如果合一成功，有体部时才会执行体部。只有头部**成功**且体部也**成功**，子句才**成功**，否则就**失败**了。

例

```
clauses
  ppp() :- qqq(X), write(X), fail.

  qqq(1).
  qqq(2).
  qqq(3).
```

调用**ppp**时它接着会调用**qqq**。当**qqq**被调用时，它首先在第二个子句上创建一个回溯点，然后执行第一个子句。这时**ppp**中的自由变量**X**与数字**1**匹配，这样就将**X**绑定为**1**。

接着，**ppp**的**X**（也就是**1**）被写出，然后**fail**导致回溯到回溯点。因而程序控制会转到**qqq**的第二个子句，而程序的状态也会回到第一次进入**qqq**时的状态，即**ppp**中的**X**又是未绑定的了。

在实际执行**qqq**中的第二个子句之前，会先在第三个子句上创建一个回溯点。接下来的过程与**1**的一样。

Unification（合一）

当调用一个谓词时，调用的参数会与各子句的头部中的项进行**合一**。

所谓合一，就是变量绑定的过程，它使两个项相等，并且尽量少做绑定（也就是尽可能多地留下以后绑定的余地）。

变量可以绑定于任何种类的项，包括变量或含有变量的项。而合一可以是可能的，也可以是不可能的，也就是说，可以**成功**，或是**失败**。

变量及其要绑定的项是有类型的，一个变量只能和与它同类型的或是其子类型的项做绑定。如果两个变量是互相绑定的，则它们必须是相同的类型。

合一出现在一个谓词调用和子句的头部间。当两个项进行相等比较时也会出现合一。

例 考虑两个项（同一类型的）：

```
T1 = f1(g(), X, 17, Y, 17)
T2 = f1(Z, Z, V, U, 43)
```

我们来试试把这两个项从左到右进行合一。**T1**和**T2**都是**f1/5**项，是匹配的。因此，我们来试着由**T1**的参数合一**T2**相应的参数。首先要合一**Z**和**g()**，这只要把**Z**绑定为**g()**就可以了。目前为止还不错，我们得到了第一个绑定：

```
Z = g()
```

接下来第二个参数是**X**和**Z**，而后者已经绑定为**g()**了。只要让**X**绑定为**g()**，这两个参数也是可以合一的，这样我们有了第二个绑定：

```
X = Z = g()
```

再接下来，必须把**V**绑定为**17**，还要把**Y**绑定为**U**。现在已经绑定的有：

```
X = Z = g()
V = 17
Y = U
```

两个合一的项现在等价于下面的项：

```
T1 = f1(g(), g(), 17, Y, 17)
T2 = f1(g(), g(), 17, Y, 43)
```

但是我们还没有合一最后两个参数，这两个参数是**17**和**43**。没有什么变量绑定能使这两个项相等，所以最终合一**失败**了。**T1**和**T2**不能合一。

上面这个例子中，**T1**可以是一个谓词调用而**T2**可以是一个子句的头部。当然，它们也可以是两个用“＝”比较的项。

Matching（匹配）

匹配只有一点与合一不同，这就是匹配时变量只能绑定于根基（***grounded***）项。所谓**根基**项，就是不含任何未绑定变量的项。

正是说明了谓词的流样式，使得我们可以用匹配而不用全面的合一成为可能。

例

```
clauses
  ppp(Z, Z, 17).
  qqq() :-
    ppp(g(), X, 17).
```

调用**ppp**时合一**ppp**的子句，只要使用下面的绑定，是可行的：

```
Z = X = g()
```

如果**ppp**的流是**(i,o,i)**，合一就只是一个匹配：

- **g()**是个输入，作为第一个参数，它绑定于**Z**，
- 子句中第二个参数因此也可以绑定并可以输出给**X**，这样**X**就变成绑定于***g()***，
- 最后，第三个参数是**17**，它是一个输入，这个数只是简单地与子句中的第三个参数作比较。

可以看到，正是有了流样式，我们才能预知子句而并不需要实际进行合一。

嵌套的函数调用

要合一或匹配的项可以包含子项，而子项是表达式或函数调用，它们需要在合一/匹配完成之前进行求值。

这样的子项求值是按需要进行的。

在谓词调用时，所有输入参数在谓词调用前做求值，所有输出参数是变量，不需要求值。

子句头部也可以包含需要求值的项，在合一/匹配前确定下来。

- 在求值进行之前，先做不需要做求值处理的合一/匹配；
- 然后相应于输入的参数从左到右逐个求值。求值一个就与对应的输入比较一个；
- 接着对子句的体求值；

- 再从左到右对输出参数求值；
- 最后，如果谓词是函数，就对返回值求值。

如果上述过程中出现任何**失败**，就不进行其它的求值。

总之，基本原则是：

- 输入求值先于体部求值；
- 体部求值之后是输出求值；
- 求值从左向右进行。

匿名谓词

匿名谓词是对一个谓词的值进行计算的表达式，这个值可以绑定给一个变量，可以作为参数传递或作为结果返回，但这个值没有任何类、接口或实现的名称。

匿名谓词可以在表达式出现的关联中获取值，可以用它来避免一些冗长难懂的代码。

语法

匿名谓词是项：

```
Term : one of
  ...
  AnonymousPredicate
  ...
```

匿名谓词是一个在大括号中的无名子句，某些部分是可选的，形式如下：

```
AnonymousPredicate : one of
  { ( Arg-comma-sep-list ) = Term }
  { ( Arg-comma-sep-list ) = Term :- Term }
  { ( Arg-comma-sep-list ) :- Term }
  { = Term }
  { = Term :- Term }
  { :- Term }
```

省略参数表意味着“需要数量的参数”，用在不需要实际使用参数时。

语义

匿名谓词表达式对一个谓词求值。看下面的代码：

```
clauses
  run() :-
    Inc = { (X) = X+1 },
    A = Inc(4),
    B = Inc(23),
    stdio::writef("A = %, B = %", A, B).
```

Inc成了一个增量谓词，程序的结果是：

```
A = 5, B = 24
```

上面的代码相当于这样：

```
clauses
  run() :-
    Inc = inc,
    A = Inc(4),
    B = Inc(23),
    stdio::writef("A = %, B = %", A, B).

class predicates
  inc : (integer X) -> integer R.

clauses
  inc(X) = X+1.
```

这里，最后一行可以看到子句**(X) = X+1**，在这里它是一个命名了的谓词。

在匿名谓词之外（即匿名谓词出现之前）绑定的变量可以在匿名谓词中使用，变量的值将被匿名谓词获取。而在匿名谓词中绑定的变量是该匿名谓词的局部变量。

获取关联

匿名谓词可以获取关联，也就是说它可以引用关联中定义的东西，尤其是从子句中引用事实和变量。

获取变量

匿名谓词是出现在子句中的，而这个子句可能含有变量。对那些在匿名谓词出现之前绑定的变量，在匿名谓词中可以使用。下面的代码说明了如何获取变量的：

```
domains
  pred = (integer) -> integer.

class predicates
  createAdder : (integer A) -> pred Adder.

clauses
  createAdder(A) = { (X) = X+A }.

clauses
  run() :-
    Add17 = createAdder(17),
    A = Add17(4),
    B = Add17(20),
    stdio::writef("A = %, B = %", A, B).
```

用**17**做参数调用**createAdder**，这时**createAdder**子句中的**A**是**17**，因而结果是**{ (X) = X+17 }**。我们说匿名谓词获取了变量**A**。**Add17**是一个把**17**加到自身参数的谓词，输出的结果就是：

```
A = 21, B = 37
```

获取省略符

匿名谓词可以获取省略符变量：

```
clauses
  ppp(...) :-
    W = { () :- stdio::write(...) },
    qqq(W).
```

W获取了省略符变量。**qqq**收到一个0元维的谓词，当这个谓词被调用时，获取的省略符变量就会写到标准输出设备上去。

获取事实

匿名谓词可以获取事实。如果匿名变量是由类谓词创建的，它就可以访问类事实；如果是由对象谓词创建的，它就既可以访问对象事实又可以访问类事实。下面的代码获取的是类事实：

```
class facts
  count : integer := 0.

clauses
  seq() = { () = count :- count := count+1 }.

clauses
  run() :-
    A = seq(),
    B = seq(),
    stdio::writef("A1 = %, ", A()),
    stdio::writef("B1 = %, ", B()),
    stdio::writef("A2 = %, ", A()),
    stdio::writef("B2 = %", B()).
```

A和**B**都会使类事实计数增加，所以结果是：

```
A1 = 1, B1 = 2, A2 = 3, B2 = 4
```

对象谓词中可以获取对象事实。假设**seq**是**myClass**中的对象谓词，下面的代码说明了获取对象事实：

```
facts
  count : integer := 0.

clauses
  seq() = { () = count :- count := count+1 }.

clauses
  run() :-
    A = myClass::new():seq(),
    B = myClass::new():seq(),
    stdio::writef("A1 = %, ", A()),
    stdio::writef("B1 = %, ", B()),
    stdio::writef("A2 = %, ", A()),
    stdio::writef("B2 = %", B()).
```

这里，**A**和**B**来自不同的对象，它们有各自的计数，所以输出是：

```
A1 = 1, B1 = 1, A2 = 2, B2 = 2
```

技术上来说，类版本的匿名谓词其实获取不了什么东西，它很少用来访问事实。同样地，对象版的其实也没有获取事实，它获取的是This并通过This访问了对象事实。

获取This

如上所述，可以获取**This**并取得对对象事实的访问权。同样的机制可以调用对象谓词：

```
clauses
  seq() = { () = count :- inc() }.

clauses
  inc() :- count := count+1.
```

This也可以直接使用：

```
clauses
  ppp() = { () = aaa::rrr(This) }.
```

嵌套

匿名谓词可以嵌套：

```
clauses
  run() :-
    P = { (A) = { (B) = A+B } },
    Q = P(3300),
    R = P(2200),
    stdio::writef("Q(11) = %, ", Q(11)),
    stdio::writef("R(11) = %", R(11)).
```

要得到**Q**用**3300**调用**P**，这样**A**是**3300**而**Q**就是**{ (B) = 3300+B} }**，同样**R**是**{ (B) = 2200+B } }**，结果是：

```
Q(11) = 3311, R(11) = 2211
```

Syntactic Sugar（便利化的语法）

如果不需要参数，可以把它们略过。下面这个代码片断：

```
P = { (_) :- succeed },
Q = { (_, _) = 0 },
R = { (_, _, _) = _ :- fail }
```

可以简化成这样：

```
P = { :- succeed },
Q = { = 0 },
R = { = _ :- fail }
```

注意，简化时参数部分完全被略去了。如果写成**()**则意味着是零参数，而略去所有参数意思是“有那么些”参数。

实际用例

这节内容显示了匿名谓词的便利性。例子中假定PFC中的core、std、stdio、list和string等范围是开放的。

哑谓词

用匿名谓词来创建哑谓词值是很方便的：

```
ppp( { = true } ),      % 不过滤 (boolean)
qqq( { :- succeed } ), % 不过滤 (determ)
rrr( { = 17 } ),        % 所有行高必须为 17
```

适配

有的时候要用一个谓词，而已经有了一个差不多合适的，这时可以用匿名谓词来配合以满足需要。

适配索引

看一下谓词write3：

```
class predicates
  write3 : (function{integer, string} Indexer).
clauses
  write3(Indexer) :-
    foreach I = std::fromTo(0,2) do
      write(Indexer(I), "\n")
    end foreach.
```

Indexer提供了一个串的“数组”，**write3**可以写出以索引号为**0**、**1**和**2**的三个串。因此，**write3**设定“数组”索引号是从0开始的。不过，我们想用一个索引号从1开始的“数组”：

```
class predicates
  myArray : (integer N) -> string Value.
clauses
  myArray(1) = "First" :- !.
  myArray(2) = "Second" :- !.
  myArray(3) = "Third" :- !.
  myArray(_) = _ :-
    raiseError().
```

用一个匿名谓词，就可以轻而易举地使索引号从1开始的数组满足索引号从0开始的使用要求：

```
% myArray is 0-based, write3 requires 1-based
Arr = { (N) = myArray(N+1) },
write3(Arr)
```

这样，我们就可以得到期望的输出：

```
First
Second
Third
```

参数适配

在下面的代码中，**listChildren**将为每一个“**C**是**P**的孩子”数据对调用谓词**ChildWriter**：

```
class predicates
  listChildren :(predicate{string,string} ChildWriter).
clauses
  listChildren(CW) :-
    CW("Son1", "Father"),
    CW("Son2", "Father").
```

不过，现在我们想用谓词**wParent**写出“**P**是**C**的父母”的列表来：

```
class predicates
  wParent : (string Parent, string Child).
clauses
  wParent(P, C) :-
    writef("% is the parent of %\n", P, C).
```

wParent所用的参数刚好顺序是相反的，可以很容易地用一个匿名谓词来调整它：

```
Swap = { (A,B) :- wParent(B,A) },
listChildren(Swap)
```

输出就是我们想要的了：

```
Father is the parent of Son1
Father is the parent of Son2
```

我们还可以丢掉不需要的参数，比如调用谓词时我们只想要“**Child**”：

```
class predicates
  wKnowParent : (string Child).
clauses
  wKnowParent(C) :-
    writef("We know a parent of %\n", C).
```

可以这样来做需要的适配：

```
Fewer = { (C,P) :- wKnowParent(C) },
listChildren(Fewer)
```

输出会是这样：

```
We know a parent of Son1
We know a parent of Son2
```

我们还可以使用哑参数：

```
More = { (_,P) :- addChildren(P, 1) }
listChildren(More)
```

这里的**addChildren**会“把孩子数加到**P**中”，因为对应每个孩子的调用**addChild**都提供了一个“哑参数”**1**。More这个适配器提供一个哑参数并丢弃一个我们不需要的参数。

筛选

设有谓词：

```
class predicates
  writeFiltered :(string L, filterPredicate{integer} Filter).
clauses
  writeFiltered(Label, Filter) :-
    List = [1,2,3,4,5,6,7,8,9],
    FilteredList = filter(List, Filter),
    writef("%\t%\n", Label, FilteredList).
```

过滤器Filter用来筛选表**[1,2,3,4,5,6,7,8,9]**，筛选出来的表和标签Label写到标准输出设备上。我们先用个全选过滤器：

```
All = { :- succeed },
writeFiltered("All", All)
```

这个过滤器对任何元素都成功，所以输出是全表：

```
All [1,2,3,4,5,6,7,8,9]
```

创建一个过滤器对任何元素都失败也很简单：

```
None = { :- fail },
writeFiltered("None", None)
```

输出就是个空表：

```
None []
```

也可以创建一个过滤器选出大于**3**的元素和能被**3**整除的元素：

```
GreaterThan3 = { (X) :- X > 3 },
writeFiltered("> 3", GreaterThan3),
Rem3 = { (X) :- 0 = X rem 3 },
writeFiltered("Rem3", Rem3)
```

输出是：

```
> 3 [4,5,6,7,8,9]
Rem3 [3,6,9]
```

排序

表包里有排序谓词，不过有时缺省的顺序不能满足要求，所以表包里还有一个sortBy谓词，可以按程序员定义的比较操作对元素排序。我们先用这个谓词来考虑一下串的排序：

```
class predicates
  writeStringsSorted :
```

```
    (string Label, comparator{string} Comp).
  clauses
    writeStringsSorted(Label, C) :-
      List = ["John Wayne", "Uma Thurman",
              "Harrison Ford", "Nicolas Cage",
              "Elizabeth Taylor", "Cary Grant",
              "Jerry Lewis", "Robert De Niro"],
      Sorted = sortBy(C, List),
      write(Label, "\n"),
      foreach S = list::getMember_nd(Sorted) do
        writef("   %\n", S)
      end foreach.
```

我们可以用“normal”比较符调用谓词，还可以用一个匿名谓词很方便地实现降序排序：

```
  Normal = compare,
  writeStringsSorted("Normal", Normal),
  Descending = { (A,B) = compare(B,A) },
  writeStringsSorted("Descending", Descending)
```

输出是这样的：

```
Normal
  Cary Grant
  Elizabeth Taylor
  Harrison Ford
  Jerry Lewis
  John Wayne
  Nicolas Cage
  Robert De Niro
  Uma Thurman
Descending
  Uma Thurman
  Robert De Niro
  Nicolas Cage
  John Wayne
  Jerry Lewis
  Harrison Ford
  Elizabeth Taylor
  Cary Grant
```

我们再来排序一些更复杂的元素。下面的一些人名使用这样的域：

```
  domains
    person = p(string First, string Last).
```

作为演示，我们用这个谓词：

```
  class predicates
    writePersonsSorted :
    (string Label, comparator{person} Comparator).
  clauses
    writePersonsSorted(Label, C) :-
      List = [p("John","Wayne"),
              p("Uma","Thurman"),
              p("Harrison","Ford"),
```

```
            p("Nicolas","Cage"),
            p("Elizabeth","Taylor"),
            p("Cary","Grant"),
            p("Jerry","Lewis"),
            p("Robert","De Niro")],
    Sorted = sortBy(C, List),
    write(Label, "\n"),
    foreach p(F,L) = list::getMember_nd(Sorted) do
      writef(" % %\n", F, L)
    end foreach.
```

同样，我们也可以用正常和降序比较符：

```
    Normal = compare,
    writePersonsSorted("Normal", Normal),
    Descending = { (A,B) = compare(B,A) },
    writePersonsSorted("Descending", Descending)
```

因为比较谓词对函子使用从左到右的词典顺序，结果与前面的一样：

```
    Normal
      Cary Grant
      Elizabeth Taylor
      Harrison Ford
      Jerry Lewis
      John Wayne
      Nicolas Cage
      Robert De Niro
      Uma Thurman
    Descending
      Uma Thurman
      Robert De Niro
      Nicolas Cage
      John Wayne
      Jerry Lewis
      Harrison Ford
      Elizabeth Taylor
      Cary Grant
```

但用更复杂些的域我们还可以创建一个比较符，不是按名而是按姓来排序：

```
    LN = { (p(_,L1), p(_, L2)) = compare(L1,L2) },
    writePersonsSorted("LastName", LN)
```

这正是我们期望的结果：

```
    LastName
      Nicolas Cage
      Robert De Niro
      Harrison Ford
      Cary Grant
      Jerry Lewis
      Elizabeth Taylor
      Uma Thurman
      John Wayne
```

获取关联

前面说过，匿名谓词一个很有用的特性是获取关联的能力。下面的例子说明如何使用这个特性。

Background threads（后台线程）

启动线程的例程有一个空元谓词，会在一个新线程中运行它。不过我们常常需要传递一些输入数据给新线程中的作业。可以有多种办法，但最简单的办法是用匿名谓词。Visual Prolog例集（可以在IDE中安装）中的**bgDemo**工程用的就是这个办法。这个工程中有一个表格，可以启动后台作业，并用表格中一个**jobControl**控件显示作业的状态信息。后台作业是一个谓词，它接收**jobLog**，这个控件可以用来报告状态和进度：

```
domains
  job = (jobLog Log).
```

jobLog是这样的：

```
interface jobLog
  properties
    completion : real (i).
  properties
    status : string (i).
end interface jobLog
```

作业通过设置完成属性（**0**到**1**）来报告完成进度。同样，状态属性可以用来反映作业的当前状态。状态和完成进度用一个作业名显示在表格里。作业的启动是通过调用表格的**addJob**谓词：

```
clauses
  addJob(JobName, Job) :-
    JobCtrl = jobControl::new(This),
    JobCtrl:name := JobName,
    JobCtrl:show(),
    assert(jobCtrl_fact(JobCtrl)),
    arrange(),
    JobLog = jobLog::new(JobCtrl),
    Action = { :- Job(JobLog) },
    _ = thread::start(Action).
```

上面最后三行是我们关心的关联。**thread::start**用一个空元谓词作参数，但作业是一个以**jobLog**为参数的谓词。因此，我们用了一个匿名谓词**Action**，它没有参数但在**JobLog**上调用**Job**。匿名谓词从关联中既获取了**Job**又获取了**JobLog**，因而它们的值可以传递给新线程，尽管线程得到的是一个空元谓词。**bgDemo**工程中的作业差不多是个空作业，只是操控**jobLog**。作业是这个样子的：

```
clauses
  job(Log, From, To) :-
    Log:status := "Step 1",
    foreach N1 = std::fromTo(From, To) do
      Log:completion :=
      (N1-From) / (To-From) / 2,
      programControl::sleep(3)
    end foreach,
    Log:status := "Step 2",
```

```
        foreach N2 = std::fromTo(From, To) do
          Log:completion :=
          (N2-From) / (To-From) / 2 + 0.5,
          programControl::sleep(3)
        end foreach,
        Log:status := "finished".
```

从**From**到**To**这里有两个循环，计算完成进度并把它设置给**Log**。它还要在循环前，循环中和循环后设置状态文字。可以看到，这个作业没有固有的作业类型，因为一个特定的作业只有一个参数（**jobLog**），而这个作业有三个参数。还是匿名谓词帮了忙。

在表格中添加作业的代码是这样的：

```
predicates
  onFileNew : window::menuItemListener.
clauses
  onFileNew(_Source, _MenuTag) :-
    JF = jobForm::display(This),
    Job11 = {(L) :- job1::job(L, 1, 1000)},
    Job12 = {(L) :- job1::job(L, 200, 600)},
    Job13 = {(L) :- job1::job(L, 1200, 3000)},
    Job14 = {(L) :- job1::job(L, 1, 1000)},
    JF:addJob("job1.1", Job11),
    JF:addJob("job1.2", Job12),
    JF:addJob("job1.3", Job13),
    JF:addJob("job1.4", Job14),
    ...
```

在实际的程序中，**From**和**To**不会是常数，它会是从别的地方传递来的参数。在这种情况下，匿名谓词也会从关联中获取变量。在**bgDemo**中的**jobLog**说明了匿名谓词又一种使用方法，它传递完成进度和状态信息给一个控件**jobControl**。**jobControl**是**jobForm**中的一个GUI控件，它以适当的形式表现得到的信息。不过，这里有个同步的问题，因为GUI控件在线程级并不稳定，而在这里我们是从一个后台线程中更新控件的。这有可能导致冲突，因为是主线程在画该控件。解决的办法是把控件的更新交给GUI线程。可以通过把动作传递给控件来这样做。状态更新的实现是这样的：

```
clauses
  status(Status) :-
    Action = { :- jobCtrl:status := Status },
    jobCtrl:postAction(Action).
```

Action是一个空元谓词，它会设置**jobCtrl**中的状态。我们把这个动作传递给**jobCtrl**，当它接收到action时就会调用**Action**完成更新。这样，控件的实际更新是由GUI线程完成的，这个匿名谓词不仅获取**Status**变量而且获取了**jobCtrl**事实。

匿名回叫

考虑一下给远端服务发送命令的情况。命令的执行是异步的，所以执行一个命令也还需要一个回叫动作，执行完了命令就调用这个动作。也就是说，为执行命令需要调用这样的谓词：

```
predicates
  executeCommand :
      (command Cmd, predicate{} OnDone).
```

基于这个谓词，还可以创建一个可以执行一个命令表的谓词，前一个命令执行完了接着执行下一个命令。执行命令表也是异步的，所以命令脚本都执行完了也需要调用一个动作。脚本执行的形式是这样的：

```
predicates
  executeScript :
      (command* Script, predicate{} OnDone).
```

如果脚本空了，就调用**OnDone**。若脚本有一个命令**H**及剩下的脚本**T**，必须先执行**H**，完成之后执行剩下的脚本**T**。所以我们在**H**执行时提供的**OnDone**动作还需要执行**T**。总之，实现会是这样的：

```
clauses
  executeScript([], OnDone) :-
    OnDone().
  executeScript([H|T], OnDone) :-
    DoneH = { :- executeScript(T, OnDone) },
    executeCommand(H, DoneH).
```

我们用了一个匿名谓词来完成剩余脚本的执行，这个谓词获取**T**和**OnDone**。

事实变量的赋值

赋值算子**:=**用于给事实变量***FactVariable***赋予新值。***Term***必须能求值得到适当的类型（与事实变量相同的类型或是其子类型）。

```
FactVariableAssignment:
  FactVariable := Term
```

事实

事实数据库包含有若干完全实例化的（根基）谓词头部的相应事实，这些事实是在事实段声明的。事实可以由谓词调用来访问，只要把事实名当谓词名就行了。各个事实依次与谓词调用进行匹配，每次谓词调用与事实匹配了则成功，可能还会带有对下一个事实的回溯点；当事实数据库中再也没有其它的事实时这个谓词调用就失败了。

可以用谓词assert/1、asserta/1和assertz/1**插入**新的事实。assert/1和assertz/1是一样的，把新事实插入到事实列表的尾部，而asserta/1则把新事实插入到列表的开头。

已有的事实可以用谓词retract/1和retractAll/1**撤消**。retract/1撤消第一个与参数匹配的事实，并且可以通过回溯撤消更多的事实。retractAll/1能撤消所有与参数匹配的事实，它总能成功。

运算符

运算符（算子）按优先级分组，下面各组的算子在组内具有相同的优先级，而在前面的组优先级高于后面组的优先级。如，一元加减号（也就是正负号）优先级高于幂算子，而幂算子的优先级又高于乘法算子，等等。圆括号可以用来标示优先级使之更清晰。

```
BinaryOperator: one of
  PowerOperator
  UnaryOperator
  MultiplicationOperator
  AdditionOperator
  RelationOperator
```

```
        MustUnifyOperator
        InOperator
        AndOperator
        OrOperator

    UnaryOperator: one of
      - +
```

所有算子除***UnaryOperator***外都是二元的。只有幂算子是右结合的，其它所有算子都是左结合的。***RelationOperator***、***MustUnifyOperator*** 和 ***InOperator***的优先级相同。

注意：***UnaryOperator***的优先级位置与数学中的并不一样，在数学中它与***AdditionalOperator***的优先级相同。这个差别对于值的计算没有影响，只是对书写方式有影响。比如，程序中可以写**2*-2**，但在数学中就要求对第二个算子加括号，写成**2*(-2)**。再比如，程序中的**-2*2**意思是(-2)*2，而在数学上则是**-(2*2)**（尽管值是一样的）。

例 **-2^2**与**-(2^2)**是相同的，因为 ^ 的优先级比一元减号的高。

例 **3^2^2**与**3^(2^2)**是相同的，因为 ^ 是右结合的。

例 **-2*-3^-4+5**与**((-2) * (-(3 ^(-4)))) + 5**是一个意思。

例 下面的项：

```
    7 + 3 * 5 * 13 + 4 + 3 = X / 6 ; A < 7, p(X)
```

与下面的项是一样的：

```
    ((((7 + ((3 * 5) * 13)) + 4) + 3) = (X / 6)) ; ((A < 7) , p(X))
```

也就是说，最外层是两个“**或**”关系的项，其中第一个项是一个“**等号**”关系项而第二个是一个“**与**”项。

算术运算符

算术运算符用于对数进行算术运算。它们是表达式，并以表达式为参数。参数是根类型的，而返回的结果具有通用类型（参看通用类型和根类型）。

```
    PowerOperator:
      ^

    MultiplicationOperator: one of
      * / div mod quot rem

    AdditionOperator: one of
      + -
```

关系运算符

关系运算符是公式，以表达式为参数。由于这个特性，所以它们是非结合的。

```
RelationOperator: one of
  = > < >= <= <> ><
```

先对左边的项做计算，而后是右边的，最后将结果做比较。
注意，<>并非双算子的**=，<>**是比较两个值，而**=**是要合一两个项（至少通常情况下是这样）。
对偶的表达式**A = B**与**(A = B)**也是有差别的。

必须合一运算符

必须合一运算符是一个以表达式为参数的过程，这个运算符是非结合的。

```
MustUnifyOperator:
  ==
```

A == B 使**A**与**B**合一。如果合一过程失败则会产生异常，否则谓词就成功。因此，**A == B**总是成功的。
例：

```
clauses
  p(L) :-
    [H|T] == L,
    ...
```

p是一个过程。如果它调用时带着空表就会出现异常，因为此时**[H|T]**和**L**不能合一。

in

```
InOperator:
  in
```

这个算子用于测试某个集合（例如表）的成员关系和非确定性地生成某个集合的成员。

例

```
predicates
  p : (Type X, Type* L).

clauses
  p(X, L) :-
    if X in L then
      write("X is in L\n")
    end if.
```

p是一个以值**X**和表**L**为参数的过程。如果**X**在表**L**中，就会写出**"X is in L"**。上述例子中in算子用于测试（成员关系的测试）。

例

```
predicates
    q : (Type* L).
clauses
  q(L) :-
    foreach X in L do
        writef("% is in L\n", X)
    end foreach.
```

p是一个以表**L**为参数的过程。in算子用于非确定性的返回L的成员，这样所有成员都会被打印出来。

用**in_test**和**in_iterate**属性，in算子可以被定义用于任意域和接口。

属性**in_test(<predicate name>)**定义的谓词在**in-test**时用于某个域或接口。而**in_iterate**属性定义的谓词是在**in_iterate**时用于某个域或接口。

例

```
domains
    tree{E} = empty; node(tree{E} Left, E Value, tree{E} Right)
                [in_test(isMemberTree), in_iterate(getAll_nd)]
```

如果一个程序中有**A in B**，而且**A**是绑定的，**B**是一个树**tree{E}**，则会调用**isMemberTree**。这种情况下**A in B**就相当于**isMemberTree(A, B)**。如果此时**A**是自由的，调用就会是**A = getAll_nd(B)**。

对于一个域（集合），谓词必须具有这样的类型：

```
predicates
    <predicate> : (<some-type> Elem, <collection> Collection) determ.
```

例

```
interface collection [in_test(contains), in_test(getAll_nd)]
...
end interface collection
```

如果一个程序中有**A in B**，而且**A**是绑定的，**B**是一个集合**collection**，则会调用**contains**。这种情况下**A in B**就相当于**B:contains(A)**。如果此时**A**是自由的，调用就会是**A = B:getAll_nd()**。

对于域**<collection>**，**in_test**及**in_iterate**谓词必须具有如下形式的声明：

```
domains
    <collection> = ... [in_test(<in_test>), in_iterate(<in_iterate>)].
class predicates
    <in_test> : (<some-type> Elem, <collection> Collection) determ.
    <in_iterate : (<collection> Collection) -> <some-type> Elem nondeterm.
```

而对于接口**<collection>**，**in_test**及**in_iterate**谓词必须具有如下形式的声明：

```
interface <collection> [in_test(<in_test>), in_iterate(<in_iterate>)].
predicates
  <in_test> : (<some-type> Elem) determ.
  <in_iterate : () -> <some-type> Elem nondeterm.
...
end interface <collection>
```

对于表域来说，in算子是预定义的，而且在PFC中该集合有可用的属性。
例

```
clauses
  p() :-
    foreach X in [1, 2, 3, 4] do % in_iterate
      if X in [2, 4] then % in_test
        ...
      end if
    end foreach.
```

例子中第一个in是表域中预定义的**in_iterate**，而第二个是是预定义的**in_test**。
再如：

```
clauses
  q() :-
    M1 = setM_redBlack::new(),
    M1:inset("a"),
    M1:inset("b"),
    M2 = setM_redBlack::new(),
    M2:inset("b"),
    foreach X in M1 do % in_iterate
      if X in M2 then % in_test
        ...
      end if
    end foreach.
```

对于集合来说，in算子决定了**contains**和**getAll_nd**：

```
interface collection{@Type}
    [in_test(contains), in_iterate(getAll_nd)]
predicates
  contains : (@Type Value) determ.
  % @short 如果该集合含有@Type类型的值则成功
  % @end
```

```
predicates
  getAll_nd : () -> @Type Value nondeterm.
  % @short @Type is nondeterministic iteration of the elements in the collection.
  % @end
...
end interface collection
```

逻辑运算符

“与”算子（***AndOperator***）和“或”算子（***OrOperator***）是公式，以公式为参数，它们都是左结合的。“**,**”和“**and**”是一回事，“**;**”和“**or**”也是一回事。

```
AndOperator: one of
  , and

OrOperator: one of
  ; or
```

and (,)

一个“**and**”项“**A, B**”的处理过程如下：先对左子项**A**求值，如果求值失败，整个“**and**”项就失败了。如果**A**成功了，再对右子项**B**求值，如果失败，“**and**”项失败。否则“**and**”项就成功了。因此，对第二个子项**B**的求值只有当子项**A**成功之后才会进行。

例

```
clauses
  ppp() :-
    qqq(), rrr().
```

当调用**ppp**时，首先会调用**qqq**，如果**qqq**成功了，则会调用**rrr**。如果**rrr**成功，则“**and**”项成功，进而整个子句成功。

or (;)

一个“**or**”项“**A**; **B**”的处理过程如下：首先在第二项**B**上创建一个回溯点，然后对第一项**A**求值。如果求值成功，则整个“**or**”项成功，并留有一个第二项**B**的回溯点。如果第一项求值失败了，第二项上的回溯点就被激活。激活后（无论是第一项失败而激活的还是以后调用过程中激活的），就对第二项**B**求值，如果**B**成功则整个“**or**”项成功。因此，一个“**or**”项可以成功并带有一个回溯点，而第二个子项**B**仅在回溯时才会被求值。

例

```
clauses
  ppp() :-
      (V = 3 or V = 7), write(V), fail.
```

这个例子中我们用了关键字“**or**”，但也可以使用分号“**;**”。
调用**ppp**时首先会先**V = 7**上创建一个回溯点，再计算**V = 3**。这时，**V**会绑定为**3**，写**V**（即**3**）然

后碰到**fail**，**fail**总是失败，这样就又会回到回溯点**V = 7**。回溯会解除创建回溯点以来所有的绑定，所以前面对**V**的绑定解除了。然后计算**V = 7**， **V**被绑定为**7**并写出来。又碰到**fail**，此时在**ppp**中已经没有回溯点了，所以**ppp**失败。

可以在子句中用圆括号深度嵌套“**or**”。

```
clauses
  p(X) = Y :-
    (X = 1, !, Z = 3 or Z = 7), Y = 2*Z.
```

这里使用了关键字“**or**”推荐使用“**or**”。它主要是用在条件测试中：

```
clauses
  isOutside(X) :-
    (X < 10 or X > 90),!.
```

“**or**”是一个非确定性的结构，而“**orelse**”则是一个与之对应的确定性结构：

```
clauses
  isOutside(X) :-
    X < 10 orelse X > 90.
```

orelse

orelse是一个与非确定性的**or**相对应的确定性结构。如果**A**成功或者**B**成功则整个**A orelse B**结构就成功；若是**A**成功**不会**在**B**上留下回溯点。

一个**orelse**项**A orelse B**的求值过程如下：首先在第二项**B**上创建一个回溯点，而后对**A**求值。如果该求值成功了，则第二项**B**上的回溯点（包括其内部的所有回溯点）都会被消除，整个项成功。如果**A**项的求值不成功，则回溯到第二项**B**求值。因此，**orelse**项不会留下回溯点。

例

```
clauses
  ppp(V) :-
    (V = 3 orelse V = 7), write(V).
```

调用**ppp**时首先会先**V = 7**上创建一个回溯点，再计算**V = 3**。如果成功，则会消去在**V = 7**上创建的回溯点并执行**write(V)**；如果**V = 3**失败则回溯到**V = 7**求值。如果成功就接着执行**write(V)**；如果**V = 7**失败则整个**ppp**谓词就失败了。

Not

not/1用一个项作参数。**not(A)**的计算，首先计算**A**，如果**A成功**，则**not(A)失败**，如果**A失败**，则**not(A)成功**。

注意，**not(A)**不会绑定任何变量。因为如果**not(A)**成功则**A**会**失败**，而失败的项不会绑定任何东西；而如果**not(A)**失败，也不会绑定任何变量，因为该项本身是失败的。还要注意，**not(A)**不会在带有回溯点的情况下成功，因为如果**not(A)**成功则**A**是失败的，而失败的项不可能还有回溯点。这也就意味着，所有**A**成功的可能都已经穷尽了。

fail/0和succeed/0

fail/0和succeed/0是两个内建的空元谓词，前者总是失败而后者总是成功，此外它们没有别的效用。

Cut（截断）

截断“!”将进入当前谓词以来创建的所有回溯点都消除掉，这包括对后继子句的所有回溯点，以及谓词调用在当前子句 “!”之前建立的回溯点。

```
Cut:
  !
```

例

```
clauses
  ppp(X) :-
    X > 7,
    !,
    write("Greater than seven").
  ppp(_X) :-
    write("Not greater than seven").
```

执行**ppp**时，首先在第二个子句上创建一个回溯点，然后执行第一个子句。如果“**X > 7**”成功了，就碰到了截断“!”，它会消除第二个子句上的回溯点。

例

```
clauses
  ppp() :-
    qqq(X),
    X > 7,
    !,
    write("Found one").
  ppp() :-
    write("Did not find one").
clauses
  qqq(3).
  qqq(12).
  qqq(13).
```

当执行**ppp**时首先在**ppp**第二个子句创建一个回溯点，接着调用**qqq**。而**qqq**会在它的第二个子句上创建一个回溯点并执行第一个子句，返回值**3**。这样，在**ppp**中的**X**就绑定为这个值并与**7**相比较，它失败了，控制就会回退到**qqq**的第二个子句上。在执行**qqq**的第二个子句之前，会在它的第三个子句上创建一个回溯点，执行第二个子句返回的值是**12**。这次与**7**的比较成功了，执行截断，它会把在**qqq**上剩下的回溯点以及在**ppp**第二个子句上的回溯点都取消。

Cut范围

cut范围是指**截断**的作用域，其意义是：一个**cut**，只能撤消在其作用范围内的回溯点，而在这个范围之外的回溯点则不受影响，仍然保留。

谓词的子句是一个cut范围，也就是说，一个**cut**（至多）会撤消进入这个谓词之后创建的回溯点，在

进入这个谓词之前创建的那些回溯点将保留下来。

例

```
clauses
  aaa() :- p1_nd(), qqq().
  qqq() :- p2_nd(), !.
```

aaa调用**p1_nd**，会留下一个回溯点，然后又调用**qqq**；**qqq**又调用**p2_nd**，也会留下一个回溯点，接着就碰到一个**截断（cut）**。这个截断是在**qqq**谓词的截断范围内的，所以它只能废掉**p2_nd**上的那个回溯点，而**p1_nd**上的仍然保留。

有几个项涉及到截断范围（参看list comprehension、if-thenelse、foreach）。这里以**if-then-else**为例来说明一下截断范围的影响。来看下面这个示例性的**if-then-else**项：

```
if Cond then T1 else T2 end if
```

条件**Cond**是一个截断范围，在**Cond**中的**cut**只在**Cond**中起作用。另一方面，在**T1**和**T2**中的**Cut**，其作用则会超出**if-then-else**的语句。看下面的代码片断：

```
X = getMember_nd([3,1,2]),
if X = getMember_nd([3,3]), ! then
  write(X)
  else
  !
end if,
fail
```

getMember_nd是一个非确定谓词。这段代码的计算过程如下：首先**X**绑定为**3**，**getMember_nd**留下一个回溯点（这样以后**X**还可以为**1**再为**2**）。接着，要计算**if-then-else**项的条件，这个条件的第一部分会成功，因为**3**是**[3,3]**的一个成员。第一部分也一样会留下一个回溯点，这样它就能多次检查**X**是否为一个成员。接下来，碰到一个**截断**，这个截断是在**if-then-else**语句条件部分中的，故只在这个范围中起作用，也就是说它只废掉了第二个**getMember_nd**上的回溯点，而第一个**getMember_nd**谓词上的回溯点仍然保留。整个条件部分成功，现在进入到then-部分，写出“**3**”。**if-then-else**之后我们遇到了**fail**，这样就又回溯到第一个**getMember_nd**，它又绑定**X**为**1**，留下一个回溯点（以后再回溯时可以使**X**为**2**）。再次进行**if-then-else**的条件部分的计算，条件的第一部分失败了，因为**1**不是**[3,3]**的成员。这样就进入到**else**-部分，在这儿遇到一个截断，这个**cut**是在选择项的**else**-部分的，它的作用会超越**if-then-else**项，因此第一个**getMember_nd**上的回溯点被撤消了。在**if-then-else**项之后遇到了**fail**，而此时代码中已经没有回溯点，这段代码就失败了，而**X**总也不会为**2**。

List Comprehension（述求表）

```
ListComprehensionTerm :
  [ Term || Term ]
```

述求表项是一个表的表达式。看下面的示例项：

```
[ Exp || Gen ]
```

通常**Gen**是一个**nondeterm**项，而**Exp**是收集**Gen**的各个解计算得到的。它形成了一个表，相应于**Gen**的第一个解的是**Exp**的表中的第一个元素，等等。这个表，是述求表项的结果。**Exp**必须是procedure（或erroneous）。**Exp**和**Gen**都是cut范围。

述求表一般读为：如**Gen**所述，那样的**Exp**的表。

例

```
[ X || X = getMember_nd(L), X div 2 = 0 ]
```

它可以读为：**X**在**L**中并且**X**是偶数的**X**的表。所以，这个表达式是**L**中的偶数成员。

例

```
[ X + 1 || X = getMember_nd(L), X div 2 = 0 ]
```

这里的收集表达式更复杂了，要说这个项也更麻烦了：这样的（X+1）的表...。上面这个表达式还是找**L**中的偶数成员，但结果表则是把它们各加1。因而，它与下面这个表达式是完全等价的：

```
[ Y || X = getMember_nd(L), X div 2 = 0 , Y = X+1 ]
```

这个项就比较易于说：**X**则在**L**中并且**X**是偶数，**Y**是**X+1**，那样的**Y**的表。

访问对象成员

无论什么时候，只要我们引用了一个对象，我们就可以访问这个对象的对象成员谓词。

```
MemberAccess:
  Term : Identifier
```

（目前term（项）只能是一个变量或一个事实变量）。
标识必须是项的类型。
在实现内部，调用对象成员谓词时不必引用对象，因为包含了“**This**”，参看范围。

访问域、函子和常数

对域、函子和常数，可以将它们视为类成员来访问，即便是在接口中声明的也一样。这也意味着，如要对它们做限定，总是用类/接口名加双冒号。

foreach

```
ForeachTerm :
  foreach Term do Term end foreach
```

看下面的**foreach**示例项：

```
foreach Gen do Body end foreach
```

通常**Gen**是一个**nondeterm**项。**Body**对每一个**Gen**的解求值。当**Gen**失败则**foreach**项成功，不需要进行**Body**计算。**Body**必须是**procedure**（或**erroneous**）。**Gen**和**Body**都是cut范围。
foreach项的机理，类似于下面的fail循环：

```
Gen,
  Body,
fail
```

两者的主要差别是，重复后**foreach**项成功，而fail循环是失败。因此，**foreach**项可以后跟其它的项，还可以嵌套。

例

```
foreach L = list::getMember_nd(LL) do
  write("<< "),
  foreach X = list::getMember_nd(L) do
    write(" ", X)
  end foreach,
  write(" >>\n")
end foreach.
```

这里的**LL**假设是一个表的表。外层的**foreach**循环用**L**遍访这个表，所以**L**是一个表。内层的**foreach**循环用**X**遍访**L**中的元素。

有几点需要注意：

- 在关键字**do**和**end foreach**之前都没有逗号；
- **foreach**项的**Body**是一个cut范围，所以在体中的**cut**不会影响重复；
- **foreach**项总是成功(或引起异常)，在它求值完成之后，没有额外的变量被绑定。因此，**foreach**项只能使用其辅助功能。

例

```
clauses
  p(L) :-
    foreach X = list::getMember_nd(L) do
      stdio::write(X, "\n")
    end foreach,
    stdio::write("Finished\n").
```

Foreach可以用来替代传统的fail循环：

```
clauses
  p(L) :-
    X = list::getMember_nd(L),
    stdio::write(X, "\n"),
    fail.
  p(_L) :-
    stdio::write("Finished\n").
```

在这种情况下，体的过程性是很有好处的，因为在fail循环中就怕在到达循环的fail之前的偶尔失败。还有一个便利之处是，foreach循环在循环结束时是成功的，不像fail循环是失败，所以可以在同一个子句内继续进行下去。

foreach还可以嵌套：

```
clauses
  p(LL) :-
    foreach L = list::getMember_nd(LL) do
      foreach X = list::getMember_nd(L) do
        stdio::write(X, "\n")
      end foreach,
```

```
      stdio::write("Finished a list\n")
    end foreach,
    stdio::write("Finished all lists\n").
```

if-then-else

if-then-else既可作为一个语句也可作为一个表达式来使用。

if-then-else (语句)

根据条件执行一组语句。

```
IfThenElseTerm:
  if Condition then Term Elseif-list-opt Else-opt end if

Elseif:
  elseif Condition then Term

Else:
  else Term
```

下面两个项是等价的：

```
if Cond1 then T1 elseif Cond2 then T2 else T3 end if
```

```
if Cond1 then T1 else
  if Cond2 then T2 else T3 end if
end if
```

来看一下**if then else**项的机理：

```
if Cond then T1 else T2 end if
```

首先对**Cond**求值，如果成功则对**T1**求值，否则对**T2**求值。
Cond隐含地包含有一个截断，这就意味着：

- 它会把**nondeterm**条件转换成条件**determ**
- 并会把**multi**条件转换成**procedure**条件。

Cond是一个cut范围（参看cut范围）。

例

```
clauses
  w(X) :-
    if X div 2 = 0 and X > 3 then
      write("X is good")
    else
      write("X is bad"),
      nl
    end if,
    nl.
```

这个例子中，要注意几点：

- 在所有三个子项中，可以使用**and**和**or**逻辑操作符以及其它“组合”项；
- 在关键字**then**、**elseif**、**else**和**end if**之前没有逗号。

为有良好的可读性，推荐用“**or**”替代“**;**”。同样，当表示一个逻辑条件时（如上例中的情况），推荐用“**and**”替代“**,**”。

省去**else**部分是“**else** succeed”的简化形式，即：

```
if Cond then Term end if
```

是下面的简写形式：

```
if Cond then Term else succeed end if
```

if-then-else (表达式)

if-then-else表达式依据条件对表达式求值。

从语法上来看它和作为语句的if-then-else没有什么不同，不过它是用在要求项的分支必须是表达式而且要对整个if-then-else表达式求值的情况中。在这种应用中，缺失else部分是没有意义的。

例

```
clauses
  w(X, Y) :-
    Min = if X < Y then X else Y end if,
    writef("The minimum is : %\n", Min).
```

上面的例子中，当**X**比**Y**小时对**X**求值，反之则对**Y**求值，而**Min**则绑定于结果值。

try-catch-finally

try-catch-finally语句提供了一种处理给定程序代码块中可能出现异常的方法。

```
TryCatchTerm:
  try Term CatchFinally-list end try

CatchFinally: one of
  catch Variable do Trap-Handler
  finally Finally-Handler

Handler:
  Term
```

try结构有一个项、一个捕获表及最终异常处理过程。项和异常处理过程都不能留有回溯点（即不能是**multi**或**nondeterm**）。

带有多个异常处理过程的try结构等效于单个异常处理过程的try结构的嵌套，下面的项：

```
try
  Body
catch Var1 do
  Handler1
finally
  Handler2
catch Var2 do
  Handler3
```

```
        end try
```

与下面的嵌套项是等价的：

```
        try
          try
            try
              Body
            catch Var1 do
              Handler1
            end try
            finally
              Handler2
          end try
          catch Var2 do
            Handler3
        end try
```

try-catch

看下面这个try-catch结构：

```
        try
          Body
        catch Var do
          Handler
        end try
```

首先是对***Body***求值。如果***Body***失败或成功，则整个try结构失败或成功。即：若***Body***并未因一个异常而终止，则try结构就相当于对***Body***求值。

如果由于一个异常而使***Body***终止了，这个异常就被捕获，意思是***Var***首先被绑定为该异常（在PFC关联中这个异常是一个**traceId**），接着对***Handler***求值。这种情况下该结构就是带有绑定于捕获异常的***Var***对***Handler***的求值。

注意，如果***Body***因异常而终止了，它本身并不会绑定什么东西。

例 处理一个文件不能读的情况：

```
        clauses
          read(Filename) = Contents :-
            try
              Contents = file::readFile(Filename)
            catch TraceId do
              stdio::writef("Could not read the file: %\n", Filename),
              exceptionDump::dumpToStdio(TraceId),
              Contents = ""
            end try.
```

首先，求解**Contents = file::readFile(Filename)**，如果它并未因为**read**异常而终止，就会返回文件的内容。

如果出现了异常，就会带着绑定于该异常的**TraceId**对异常处理过程求解。此时，会将一个消息写到**stdio**并把异常输出到**stdio**，最后把**Contents**设为空串作为结果返回。

try-finally

看下面的try-finally结构：

```
try
  Body
finally
  Handler
end try
```

这个结构的目标是在***Body***之后对***Handler***求值，而不管***Body***是如何结束的，即它是成功也好，失败也好，因异常而终止也好（这时不能留有回溯点），都要对***Handler***求值。

求值过程是这样的：

先对***Body***求值，

- 若***Body***成功，再执行***Handler***，try-finally结构成功；
- 若***Body***失败，也执行***Handler***，try-finally结构失败；
- 若***Body***出现异常***Exn***，还是执行***Handler***，try-finally结构带着***Exn***异常终止。

例 确保在使用过后**odbcStatement**是自由的：

```
clauses
  tryGetName(Connection, PersonId) = Name :-
    Stmt = odbcStatement::new(Connection),
    try
      Stmt:setParameterValue(1, core::integer(PersonId)),
      Stmt:execDirect("select name from person where id=?"),
      Stmt:fetch(),
      Name = Stmt:getParameterValue_string(1)
    finally
      Stmt:free()
    end try.
```

即使**fetch**失败或是其它什么原因出现异常了，**Stmt**也是自由的。

18. 转换

视类型与构造类型

接口定义对象的类型，支持某个接口的所有对象都具有这个接口定义的类型。以后，我们把**对象类型**与**接口定义的类型**视为同义语。

接口定义了类型，这些类型就可以在谓词及事实声明和域定义中用作正式的类型说明符。

对象具有任意它所支持接口的对象类型，因而它也就可以当作任意这些类型的对象来使用。也就是说，对象类型被转换为任意支持的对象类型。下面要说明，这种转换多数情况下是自动进行的。

因此，同一个对象在不同关联中可以视为具有不同的类型。这样观察的对象类型被称为**视类型**，而构造对象的类的类型被称作**构造类型**或**定义类型**。构造类型也是一个视类型。

类型转换

如上所述，对象可以当成任意它所支持的接口的类型来使用。这一节要描述各种支持类型的转换是怎么进行的。

Conversion Upwards（向上转换）

如果某个项静态地归类于某个类型**T1**，而**T1**又声明支持**T2**，那么很明显该变量引用的任何对象肯定是支持的**T2**。因此，向上支持的消息是静态获取的。因而，在支持层级上的所有向上转换是自动进行的。

例 假设有接口**bb**，它支持接口**aa**及构造类型为**bb**的类**bb_class**。来看下面的代码：

```
implement ...
  predicates
    ppp : (aa AA).
  clauses
    ... :-
      BB = bb_class::new(), % (1)
      ppp(B). % (2)
  clauses
    ppp(AA) :- % (3)
      ...
      BB = convert(bb,AA), % 可以转换，因为AA的定义类型是bb
      ...
```

在（1）那行我们创建了一个**bb_class**对象，该对象具有构造类型**bb**。变量**BB**是对这个新对象的引用，它提供了对该对象的视类型**bb**。在（2）那一行该对象作为一个**aa**对象传递给了**ppp**，从视类型**bb**到视类型**aa**的转换是隐含进行的。到了（3）那行对象的视类型就是**aa**了，尽管此时构造类型仍是**bb**。

显式转换

显式转换是通过调用转换谓词来完成的，可用的转换谓词有若干个。

Checked Conversion

谓词convert/2->和tryConvert/2->用于安全地完成从一个类型到另一个类型的转换。这两个谓词都给不出真实的声明，伪声明如下：

```
predicates
  convert : ("type" Type, _ Value) -> Type ConvertedValue.
  tryConvert : ("type" Type, _ Value) -> Type ConvertedValue determ.
```

convert/2->和tryConvert/2->这两个谓词都用“type”作第一个参数，以任意类型的一个值做第二个参数，并把转换的值作为结果返回。

如果不能进行转换，convert/2->会产生一个异常，而tryConvert/2->只会失败。

注意，如果源类型是目标类型的子类型，使用convert/2->和tryConvert/2->就总是多余的，因为此时会隐含地进行转换。

这两个谓词可以用在以下场合：

- 从一个数域到另一个数域的转换；
- 将一个对象转换到另一个类型。

编译器如果能确定一个转换肯定不会成功（比如要转换的数域间根本没有交集），会有（但并非总有）报告。

Conversion Downward（向下转换）

当一个对象转换成超类型（如支持的接口）时，这个对象的信息就“不记得”了。当然能力并不是真的丢失了，而只是在这种弱化能力接口的关联中来看对象时看不到了。在很多情况下需要恢复对象的实际能力，因此需要对它们进行向下及向上的转换。

向下转换（一般）无法静态地验证，因此，当恢复“丢失”的接口时需要显式地转换。

例 尽管要弄个在支持层级中向上转换类型的有意义的说明例子很简单，但说明有意义的向下转换的用法还是需要一个“真实”的例子，这里我们给出一个较“真实”的例子。

假设我们要实现一组相似类型的对象，比如一组支持某个视类型的对象，我们需要这样的集合用于若干对象类型。因此，我们打算这么办：既要使不同类型的对象能接受，又要保持这些对象的同质性。

方法是标准化的：我们使集合的实际实现基于对象的类型，这个类型是所有对象都支持的。然后用一个在实际类型和对象间转换的薄层来构造集合的特殊版本。没有对象集合的实际实现，我们只需要假设它形式上存在于下面的类和接口中：

```
interface objectSet
  predicates
    insert : (object Elem).
    getSomeElem : () -> object determ.
  ...
end interface
class objectSet_class : objectSet
end class
```

假设有某个对象类型**myObject**而我们想要创建相应的“set”类**myObjectSet_class**。这样声明**myObjectSet_class**：

```
interface myObjectSet
  predicates
    insert : (myObject Elem).
```

```
    getSomeElem : () -> myObject determ.
  ...
end interface
class myObjectSet_class : myObjectSet
end class
```

即，**myObjectSet**有**objectSet**的所有谓词，但每当**object**出现时就替换成**myObject**。**myObjectSet_class**的实现继承自**objectSet_class**，这个内置/继承的**objectSet**将取得该集合的成员。该实现会执行如下的不变性：内置的**objectSet**将只包含**myObject**类型的对象（尽管从“技术”上说它们具有**object**类型）。

实现如下：

```
implement myObjectSet_class
  inherit objectSet_class
  clauses
    insert(Elem) :-
      objectSet_class::insert(Elem). % (1)
    getSomeElem() = Some :-
      SomeObject = objectSet_class::getSomeElem(), % (2)
      Some = convert(myObject, SomeObject). % (3)
  ...
end implement
```

在（1）行，Elem自动从类型**myObject**转换为**object**。在（2）那行，从内置的**object**集中取回一个对象。理论上说这个对象具有**object**类型，但从不变性上看这个对象也支持**myObject**。因而我们可以安全地恢复**myObject**接口，在（3）中显式地这样做了。

Private类型和Public类型

由构造器创建的对象，返回时是构造类型。这样的对象可以自动转换成任意所支持的接口类型并又被显式地返回。

即使实现对象的类已经声明所支持的接口是私有的，也可以把“公用”对象转换成这些私有的类型。

无论怎样，在实现内部，对象是可以用任意私有支持的类型访问的。而且，“This”可以用任意这些私有支持的类型递交到实现之外。

对象这样的“私有”版本还可以在其层级上隐含地向上转换并显式地向下转换回来。事实上，这样的“私有”对象可以显式地转换成任意公用或私有支持的接口。

所以，一个对象会有两面，公用的一面和私有的一面，而私有的一面包括了公用类型。对象不能从一面转换到另一面，但因为私有的一面包括了公用类型，所以私有的一面可以转换成任意支持的类型。

Unchecked Conversion

谓词uncheckedConvert/2用于进行基于内存表示法的非安全性转换。这个谓词并不对内存做任何修改，它只是强迫编译器对所指的存储片断解释为另外的类型。

注意，这个谓词是很不安全的，使用时要小心。

这个谓词主要用于与其它语言的接口，来解释其它语言对内存映象的使用方法。它只能使用在大小完全一样的内存片上。但不管怎么说，有很多种数据是用指针代表的，这样的数据有相同的比特规模。

```
predicates
  uncheckedConvert : ("type" Type, _ Value) -> Type ConvertedValue.
```

例

```
predicates
  interpretBufferAsString : (pointer BufferPointer) -> string Value.
clauses
  interpretBufferAsString(BufferPointer) = uncheckedConvert(string,BufferPointer).
```

这个谓词将把一个指针表示的缓冲区解释（转换）为一个串。只有这个内存块真的可以正确地表示成一个串，它才是有意义的。

19. 异常处理

本节描述Visual Prolog中异常处理的低级结构。PFC在这个低级机制之上还有一个较高级的层次（参看异常处理指南）。

异常处理的基础是基于内建谓词errorExit/1及try-catch语言结构的。

- errorExit/1产生一个异常
- try-catch为某个计算设置异常处理程序。

当errorExit/1调用时，当前激活的异常处理程序被调用，这个异常处理程序是按原来的关联执行的，这个关联是原来设置的而不是发生异常那里的关联。调用时的参数传递给了异常处理程序，这个参数必须以某种方式提供需要的异常描述。

与其它运行时例程一道，在这个系统之上就可以建立起高级的异常处理机制了。不过运行时系统访问程序及其如何处理异常等问题已经超出本文档的范围，就不介绍了。

try-catch的第一个参数是与新异常处理程序一道执行的项；第二个参数必须是一个变量。如果异常处理程序被激活，这个变量将绑定于errorExit/1被调用时的值。第三个参数是异常处理程序，如果errorExit/1被调用 则会激活这个异常处理程序。

异常处理程序可以访问第二个参数说明的变量，因而可以检查发生了什么异常。

例

```
clauses
  p(X) :-
    try
      dangerous(X)
    catch Exception do
      handleDangerous(Exception)
    end try.
```

如果在执行**dangerous**时出现了异常，**Exception**就会绑定为异常值，try-catch控制将转到的第三个参数，在这里就是把**Exception**传递给**handleDangerous**。

20. 内建实体

Visual Prolog有一个内含的类，提供了所有内建的常数、域及谓词的声明和实现。这些内建的常数、域及谓词既可以用在编译时（如在**#if** ...结构中）也可以用在实现中（由运行时支持）。每个编译单元隐含地包括了这个内含类的声明。可以在内建项的名称前使用“**::**”以与其它同名的实体相区分。

注意，子句变量**This**是在对象谓词子句中自动定义的。

运算符

运算符	描述	备注
^	幂	64位整数不适用
-（一元）	一元减（负号）	
*, /	乘、除法	
div, mod	整数除法取商和取余（向负无限大舍入）	实数不适用
quot, rem	整数除法取商和取余（向零舍入）	实数不适用
+, -	加、减法	

- 上面的运算符是按优先级由高到低排列的；
- 所有的乘法和除法运算符优先级是一样的；
- 幂运算符是右结合的；
- 所有其它运算符都是左结合的。

所有二元运算符都有两个参数，这两个参数基类型相同，返回的值也是同样的基类型。操作数和结果的类型可以应用普通子类型规则进行转换。

整数除法

div和**quot**是不同的整数除法算子：

- **div**向负无限大舍入，**mod**是对应于**div**的余数，
- **quot**是向零舍入，**rem**是对应于**quot**的余数。

当计算结果是正数时，**div**与**quot**没有什么区别；但当结果是负值时，可以从下面的表中观察到它们的差别：

A	B	A div B	A mod B	A quot B	A rem B
15	7	2	1	2	1
-15	7	-3	6	-2	-1
15	-7	-3	-6	-2	1
-15	-7	2	-1	2	-1

常数

compilation_date	编译日期
compilation_time	编译时间
compiler_buildDate	编译器的构建日期
compiler_version	编译器版本
maxFloatDigits	定义编译器支持的数字最大值
null	缺省的空指针
nullHandle	值为零的句柄类型的一个特殊常数
invalidHandle	表示非法的（值为-1）句柄类型的一个特殊常数
platform_bits	定义编译平台的数字能力
platform_name	定义目标平台名称

编译日期

```
compilation_date : string = "YYYY-MM-DD".
```

这里的**YYYY**是代表年的数字，**MM**是代表月份的数字，**DD**是代表日的数字。

编译时间

```
compilation_time : string = "HH-MM-SS".
```

这里的**HH**代表小时，**MM**代表分钟，**SS**代表秒。

编译器构建日期

```
compiler_buildDate : string = "YYYY-MM-DD HH-MM-SS".
```

编译器的构建日期。

编译器版本

编译器的版本，这个数值与编译器的版本相关。

```
compiler_version = 6003.
```

maxFloatDigits

定义编译器支持的数字最大值。

```
maxFloatDigits = 16.
```

null

```
null : pointer = uncheckedConvert(pointer, 0).
```

值为零的指针类型的一个特殊常数。

nullHandle

值为零的句柄类型的一个特殊常数。

```
nullHandle : handle = uncheckedConvert(handle, 0).
```

invalidHandle

其值（-1）表示非法的句柄类型的一个特殊常数。

```
invalidHandle : handle = uncheckedConvert(handle, -1).
```

platform_bits

编译平台的数字能力。

```
platform_bits = 32.
或
platform_bits = 64.
```

platform_name

目标编译平台名称。

```
platform_name : string = "Windows 32bits".
或
platform_name : string = "Windows 64bits".
```

域

any	通用项类型
char	宽（两字节）字符
string	以两个零字节结尾的宽字符序列
string8	以一个零字节结尾的ASCII (单字节)字符序列
symbol	以两个零字节结尾的宽字符序列
binary	字节序列
binaryNonAtomic	字节序列
integer	有符号整数
integer64	有符号整数
integerNative	有符号整数
unsigned	无符号整数
unsigned64	无符号整数
unsignedNative	无符号整数
real	浮点数

real32	浮点数
pointer	指向内存地址的指针
handle	句柄（例如本生文件和窗口的句柄）
boolean	布尔值
factDB	命名了的内部数据库描述符
compareResult	比较结果的值

any

通用项类型。

```
any
```

这个域的值是任意的项。这样的值包含有对项类型库的引用以及项本身。

char

宽字符。

```
char
```

这个域的值是UNICODE字符，由两个字节无符号数实现。
对这个域的值的操作，只能有赋值和（字典意义上的）比较。字符映象语法如下：

```
Char_image :
  ' Char_value '
Char_value :
  Letter
  Digit
  Graphical_symbol
  \ Escape_seq
Escape_seq:
  t
  n
  r
  \
  '
  "
  u <HHHH>
```

上面的***HHHH***对应于4个十六进制数。还有，反斜杠和单引号只能由换码序列来表示。

compareResult

这是一个内建的域，用来定义一个比较结果。内建谓词compare/2->的结果就是这个域的。

```
domains
  compareResult = less; equal; greater.
```

string

以宽零结尾的宽字符序列。

```
string
```

串是UNICODE字符序列。它是由一个指向以宽零结尾的宽字符序列的指针实现的。这个域的值只能进行赋值与（字典意义上的）比较操作。在源代码中，可以用以双引号括起的字符序列表示一个字符串。

```
StringLiteral:
  StringLiteralPart-list

StringLiteralPart :
  @" AnyCharacter-list-opt "
  " CharacterValue-list-opt "
```

字符串由一个或多个连接起来的***StringLiteralPart***组成。带有@的***StringLiteralPart***不使用换码序列，而不带@的***StringLiteralPart***则使用以下的换码序列：

- ****表示****
- **\t**表示制表符
- **\n**表示换行符
- **\r**表示回车
- **\'**表示单引号
- **\"**表示双引号
- **\u**后跟四个十六进制数表示相应的Unicode字符。
-

串中的双引号只能由换码序列来表示，而单引号可以用换码序列或图形符号两种方法来表示。

string8

内建的**string8**域这个项是一个ASCII字符序列。它是由一个指向零结尾的ASCII字符序列的指针实现的。这个域的值只能进行赋值与（字典意义上的）相等比较操作。目前，对这个域来说不能使用文字*。

symbol

以宽零结尾的宽字符序列。

```
symbol
```

与串一样，symbol也是一个UNICODE字符序列。它是由一个指向包含串的symbol表的入口指针实现的。对它可以做的操作与串一样。

symbol的映象是由串文字（用双引号包围的串）来表示的。它与串在很大程度上是可以互换的，但它们的存储方式不相同。symbol被放在一个查找表中，它们的地址（而不是symbol本身）存储起来用于代表对象。这意味着symbol可以快速地匹配，而且如果它重复地出现在程序中时占用存储空间很小。串

* **译注：**根据PDC的Thomas Linder Puls的说明，“不能使用文字”是指不能使用文字串而只能使用ASCII代码。这是因为不同的ASCII代码页对应的文字是不同的。

则不是放在查找表中的，进行匹配时Visual Prolog要逐个字符地进行检查。

binary

N个字节的序列。

```
binary
```

这个域的值用于保存二进制数。它是由指向这个字节序列的一个指针实现的。这个项的长度放在字节序列的前面，占**四个字节**。这四个字节的值为：

```
TotalNumberOfBytesOccupiedByBinary = ByteLen + 4
```

ByteLen是二进制项本身的长度。
对这个域的值只能进行赋值和比较操作。比较操作的方法如下：

- 如果两个项长度不一，长度大的那个项大；
- 如果两个项长度一样，就按无符号数逐字节地比较，比较到不一样的字节时就停止，那个字节的比较结果就是这两个项的比较结果。如果字节都一样，那就是相等。

二进制映象的内容语法由***Binary***规则确定：

```
Binary :
  $ [ Byte_value-comma-sep-list-opt ]
Byte_value :
  Expression
```

各个表达式的值应是在编译时可计算得到的，且值的范围在**0**到**255**之间。

binaryNonAtomic

N个字节的序列。

```
binaryNonAtomic
```

同binary，但可以包含指针。

integer

有符号整数。

```
integer
```

这个域的值占四个字节。可以进行算术运算（**+**, **-**, **/**, *****, **^**）、比较、赋值、div、mod、quot及rem操作。值的范围是**-2147483648**到**2147483647**。
它的文字表示方法由***Integer***规则确定：

```
Integer :
  Add_operation-opt 0o Oct_number
  Add_operation-opt Dec_number
  Add_operation-opt 0x Hex_number
Add_operation :
```

```
    +
    -
Oct_number :
  Oct_digit-list
Oct_digit : one of
  0 1 2 3 4 5 6 7
Dec_number :
  Dec_digit-list
Dec_digit : one of
  Oct_digit 8 9
Hex_number :
  Hex_digit-list
Hex_digit : one of
  Dec_digit a b c d e f A B C D E F
```

integer64

有符号整数。

```
integer64
```

这个域的值占八个字节。值的范围是**-2^63 = -9,223,372,036,854,775,808**到**2^63-1 = 9,223,372,036,854,775,807**。其文字表示方法及可以进行的操作与**integer**相同。

integerNative

有符号整数。

```
integerNative
```

这个域的值所占用的字节数与平台有关，32位平台的等同于integer，64位平台的等同于integer64。

unsigned

无符号整数。

```
unsigned
```

这个域的值占四个字节。可以进行算术运算（**+**, **-**, **/**, *****, **^**）、比较、赋值、div、mod、quot及rem操作。值的范围是**0**到**4294967295**。它的表示方法同**integer**，只是不能使用负号。

unsigned64

无符号整数。

```
unsigned64
```

这个域的值占八个字节。值的范围是**0**到**2^64-1 = 18,446,744,073,709,551,615**。它的表示方法同**integer64**，只是不能使用负号。可以进行的操作同**unsigned**。

unsignedNative

无符号整数。

```
unsignedNative
```

这个域的值所占用的字节数与平台有关，32位平台的等同于unsinged，64位平台的等同于unsigned64。

real

浮点数（实数）。

```
real
```

这个域的值占八个字节。这个域只是为了方便用户引入的，可以进行所有的算术、赋值及比较操作。它的值范围是**-1.7e+308**到**1.7e+308**。需要的时候，整数域的值会自动地转换成实数域的值。它的文字表示方法由***Real***规则确定：

```
Real :
  Add_operation-opt Fraction Exponent-opt
Fraction :
  Dec_number Fractional_part-opt
Fractional_part :
  . Dec_number
Exponent :
  Exp Add_operation-opt Dec_number
Exp :
  e
  E
Add_operation :
  +
  -
Dec_number :
  Dec_digit-list
Dec_digit : one of
  0 1 2 3 4 5 6 7 8 9
```

real32

浮点数。

```
real32
```

这个域的值占四个字节。它只是为了用户的方便而引入的，可以进行所有的算术、赋值及比较操作。它的值范围是**-3.4e+38**到**3.4e+38**。表示方法同**real**。

pointer

内存地址的指针。

```
pointer
```

指针直接对应于内存地址，只能进行相等操作。这个类型中有一个内建的常数**null**。

handle

handle（句柄）用于Windows API类函数调用。该域的值与指针一样：在32位的平台上占四个字节；64位平台占8个字节。不能对它进行什么操作，也不能从别的域转换来或转换到别的域（除非是uncheckedConvert）。这个类型中有内建的常数nullHandle及invalidHandle。

boolean

布尔值。

```
boolean
```

这个域只是为了用户方便而引入的，它可以看成如下定义的复合域：

```
domains
  boolean = false(); true().
```

factDB

命名了的内部数据库描述符。

```
factDB
```

这个域有如下隐含的元声明：

```
domains
  factDB = struct @factdb( named_internal_database_domain, object ).
```

事实段中所有用户定义的名称都是这个域的常数。需要时，编译器自动地从这些常数中构建相应的复合项。在运行时，这个结构的第一项含有相应域描述符的地址而第二项或是零（对类事实段）或是一个对象的指针（即 **This**，对对象事实段）。

谓词

and/2 ,/2	项“and”
assert/1	在相应内部数据库的末尾插入指定的事实
asserta/1	在相应内部数据库的最前面插入指定的事实
assertz/1	在相应内部数据库的末尾插入指定的事实
bound/1 **determ**	测试指定变量是否已经绑定
class_Name/0->	这个编译时的谓词返回代表当前接口或类的名称的串
compare/2->	返回变量比较的结果
convert/2->	有核查的项转换

digitsOf/1->	返回指定实数域的精度
errorExit/1 **erroneous**	以指定的返回代码ErrorNumber完成一次运行时错误，并设置内部错误消息
fail/0 **failure**	引发回溯
free/1 **determ**	检查一个变量是否是自由的
fromEllipsis/1->	用EllipsisBlock中的通用类型any的项创建表
hasDomain/2 hasDomain/2->	检查一个变量是否是指定域的
in/2 **determ** in/2 **nondeterm**	中辍算子“in”(in-test和in-iterator)
isErroneous/1 **determ**	检查事实变量是否为erroneous的
lowerBound/1->	返回指定数字域的下界值
maxDigits/1->	获取相应实数域的精度值
not/1 **determ**	子目标结果(success/fail)取非
or/2 ;/2	非确定性的项“or”
orelse	确定性的项“or”
predicate_fullname/1->	这个编译时谓词返回谓词全名的串，这个谓词名表示的是其在子句体中调用谓词predicate_name的那个谓词。返回的谓词名带有范围名称。
predicate_name/0->	这个编译时谓词返回谓词名的串，这个谓词名表示的是其在子句体中调用谓词predicate_name的那个谓词。返回的谓词名不带范围名称的修饰。
programPoint/0->	这个编译时谓词返回它被调用的位置的程序点
retract/1 **nondeterm**	由相应内部数据库中删除相应的事实
retractall/1	由相应内部数据库中删除所有相应的事实
retractFactDb/1	由相应内部数据库中删除所有事实
sizeBitsOf/1->	获取指定域的实体所占内存的比特数
sizeOf/1->	获取指定项所占内存的字节数
sizeOfDomain/1->	获取指定域的实体所占内存的字节数
sourcefile lineno/0->	返回编译器处理的源文件的当前行号
sourcefile_name/0->	返回编译器处理的源文件名
sourcefile_timestamp/0->	返回表示编译器处理的源文件时间的串
succeed/0	该谓词总是成功
toAny/1->	将指定的项转换成通用项类型**any**的值
toBinary/1->	将指定的项转换成二进制表示
toBoolean/1->	这个元谓词的目标是要把（对一个谓词或事实的）确定性的调用转换成procedure的，返回**boolean**域的值。
toEllipsis/1->	由**any**类型值的表创建EllipsisBlock
toString/1->	将指定的项转换成串表示
toTerm/1-> toTerm/2->	将指定的串/二进制表示项转换为相应域的返回值变量的Prolog的项
tryToTerm/1-> **determ** tryToTerm/2-> **determ**	将指定的串/二进制表示项转换为相应域的返回值变量的Prolog的项
tryConvert/2-> **determ**	核查输入项能否被严格地转换为指定域的项，返回转换后的项

uncheckedConvert/2->	没有核查的域转换
upperBound/1->	返回指定数字域的上界值

下面的谓词已经不再使用：

finally/2	用**try ... finally ... end try**替代
findall/3	用述求表**[... || ...]**替代
trap/3 **determ**	用**try ... catch V do ... end try**替代

and

参见and (,)。

assert

```
assert : (<fact-term> FactTerm).
```

assert(Fact)把指定的事实**Fact**插入到内部事实数据库相应的数据库谓词已有的其它事实的最后面。**Fact**必须是属于内部事实数据库域的一个项。**assert/1**把一个事实加入到其已有的实例当中从而改变了它的整体情况。它与assertz/1效果是一样的。还可以参看asserta/1。

注意，retract/1和**assert/1**两者的如下组合会导致死循环：

```
loop() :-
  retract(fct(X)),
    ... % 由X建立Y
    assert(fct(Y)),
  fail.
```

问题在于，在第一行的retract总是要删除插入到最后一行的事实，因为在事实链中它是**最后**插入的。

异常

插入一个声明为determ的事实，而那个事实实例已经存在了。

asserta

```
asserta : (<fact-term> FactTerm).
```

插入一个事实到相应内部数据库的最前面。

asserta(Fact)把指定的事实**Fact**入到内部事实数据库相应谓词已有的其它事实的最前面。**Fact**必须是属于内部事实数据库域的一个项。**asserta/1**把一个事实加入到其已有的实例当中从而改变了它的整体情况。可以参看assert/1和assertz/1。

异常

插入一个声明为determ的事实，而那个事实实例已经存在了。

assertz

```
assertz : (<fact-term> FactTerm).
```

assertz与assert/1谓词完全一样。

bound

```
bound : (<variable> Variable) determ.
```

测试指定的变量是否已经绑定了一个值。

如果**Variable**已经被绑定，**bound(Variable)**成功，反之则失败。该谓词用于控制流样式以及检查引用的变量绑定情况，如果**Variable**的任意一部分已经实例化了，都将视为已经绑定

参看free/1。

class_Name

```
class_Name : () -> string ClassName.
```

这个编译时谓词返回一个串，这个串是当前接口或类的名称。

compare

```
compare : (A Left, A Right) -> compareResult CompareResult.
```

对相同域的两个项进行比较。返回的值***CompareResult***属于compareResult域。

例

```
CompareResult = compare("bar ","foo")
```

convert

```
convert : (<type> Type, Term) -> <type> Converted.
```

有核查的项转换。

这个函数的调用模板是：

```
ReturnTerm = convert(returnDomain, InputTerm)
```

- **returnDomain**：指定要将**InputTerm**转换到该域，它必须是内建的Visual Prolog域或接口域的名称，或是用户定义的与内建的Visual Prolog域、数字域、二进制域、指针域同义的名称。这个名称在编译时必须是确定的，也就是说，不能用变量来表示。
- ***InputTerm***：指定要转换的值，它可以是任意Prolog项或表达式。如果是表达式，在转换时应有明确的计算结果。
- **ReturnTerm**：返回参数**ReturnTerm**是*returnDomain*类型的。

谓词**convert**把**InputTerm**彻底转换到指定的*returnDomain*域，返回新的项**ReturnTerm**。如果它不能完成要求的转换，就会产生一个错误。还有一个功能上相似的谓词tryConvert/2->，但这个谓词如果完成不了指定的转换就只是失败，不会产生任何运行时错误。

允许的转换：

- 数字域间；

- 接口类型间；
- string和symbol间；
- 从binary转换到pointer；
- 上述各项的同义域间；
- reference域间及相应的non-reference域间。

与谓词uncheckedConvert/2->对比一下，uncheckedConvert/2->可以对两个项进行非核查的转换，对域的类型没有规定，只要这两个域的比特位长短一样就行。

当编译时对源及目标域静态已知的情况下，**convert/2->**（或tryConvert/2->）谓词进行一个有核查的显式的转换。这种显式转换的结果是下列情况之一：

- **ok**－成功转换到目标域；
- **run-time-check**－转换到目标域，但带有一个运行时兼容性的核查；
- **error**－不可能转换，产生错误。

带核查的显式转换规则：

- 同义域使用相应的域本身转换的规则；
- 数字域只能转换为数字域；
- 匿名整数域**[const .. const]**由整数常数代表；
- 匿名实数域**digits dig [const.. const]** 由实数常数代表，这里的是尾数中去掉无意义的零后的数字位数；
- symbol域和string域的值可以相互转换；
- binary域的值可以转换为pointer域；
- 对接口隐含引入的域只能按下面要介绍的规则转换到该接口域；
- 其它的域间不能转换。

数字域间的转换：

这种转换首先要考虑数值的范围，如果源的范围与目标的不重合，就会产生错误；而如果只是部分重合，就会有运行时的核查。还有，如果一个是整数域一个是实数域，就要在比较前把整数范围转换成实数范围。

如果输入项是实数而输出项是整数，**convert/2->**及tryConvert/2->谓词就会把输入值截断成最接近零的近似整数值。

接口类型的转换

谓词**convert/2->**允许把任意对象转换成任意接口类型的，而这种转换的实际正确性在运行时作核查。当创建一个对象时，保存了它的类型，因而这个对象作为参数传递时它是知道自己原来的类型的，这个原来的类型可以用来核查允许的转换。例如：

```
interface x
  supports a, b
end interface x
```

如果对象是由实现接口**x**的类创建的，而后对象又以类型**a**的参数传递给某个谓词，那么就允许转换这个对象到类型**b**。

异常

- 核查范围出错；
- 不支持的接口类型。

digitsOf

```
digitsOf : (<real-domain> Domain) -> unsigned.
```

返回指定的实数域精度。
这个函数的调用模板是：

```
Precision = digitsof(domainName)
```

这个编译时谓词的输入参数*domainName*是一个实数域，编译时*domainName*应该是明确的（也就是说它不能来自变量）。谓词返回的***Precision***数值由该域声明中的**digits**属性确定。
编译器保证*domainName*域的值至少具有***Precision***个有效十进制位数。

errorExit

```
errorExit : (unsigned ErrorNumber) erroneous.
```

以指定的返回代码**ErrorNumber**完成一次运行时错误，错误代码可以用在try-catch-finally中。

fail

```
fail : () failure.
```

fail谓词强迫一个谓词失败，因而总是引起回溯。一个以**fail**结尾的子句，无法为子句绑定输出参数。

free

```
free : (<variableName> Variable) determ.
```

检查一个变量是否为自由的。
这个谓词的调用模板是：

```
free(Variable)
```

如果指定的变量**Variable**是自由的，**free**这个谓词就成功，而如果**Variable**是绑定的，则失败。如果**Variable**的任意部分实例化了，**free**谓词将认为它是绑定的。
参看bound/1。

fromEllipsis

```
fromEllipsis : (...) -> any* AnyTermList.
```

这个谓词由省略块**...**（也就是特定的可变量参数块）中创建**any**通用类型的项表。
这个谓词的调用模板是：

```
AnyTermList = fromEllipsis(EllipsisBlock )
```

参看 toEllipsis/1->。

hasDomain

hasDomain不是个实际的谓词而更像是个声明和限定。它有两种形式，一种是非函数式的，用于声明/限定变量的类型；另一种是函数式的，用于声明/限定值的类型。

非函数式的带有两个参数，一个是类型，另一个是变量：

```
hasDomain : (<type> Type, Type Variable).
```

这个调用仅有的效果就是**Variable**被限定为**Type**类型的。

变量（variable）可以是自由的、绑定的或某种混合流模式。绑定的变量不会有变化。

函数式的也带有两个参数，第一个是类型第二个是个值，返回相同的值。

```
hasDomain : (<type> Type, Type Variable) -> Type Value.
```

这个调用仅有的效果就是确保**Value**被限定为**Type**类型的。

lowerBound

```
lowerBound : (<numeric-domain> NumericDomain) -> <numeric-domain> LowerBound.
```

返回指定的**NumericDomain**域的下限值**LowerBound**。

这个谓词的调用模板是：

```
LowerBoundValue = lowerBound(domainName)
```

这是一个编译时谓词，它返回指定的数字域*domainName*的下限值**LowerBoundValue**，这个值也是*domainName*域的。域名*domainName*在编译时应该是确定的（也就是说不能来自一个变量）。

参看upperBound/1->。

如果指定的*domainName*域不是数字域，编译时会出错。

in

参见in/2。

isErroneous

```
isErroneous : (<fact-variable> FactVariable) determ.
```

如果指定的事实变量是**erroneous**的，这个谓词就成功。

这个谓词的调用模板是：

```
isErroneous(factVariableName)
```

如果指定的事实变量**factVariableName**具有**erroneous**的值，这个谓词就会成功，否则就失败。

maxDigits

```
maxDigits : (<real-domain> RealDomain) -> unsigned MaxDigits
```

获取RealDomain指定的相应实数域的精度值。
这个谓词的调用模板是：

```
MaxDigitsNumber = maxdigits(domainName)
```

返回对应*domainName*参数的最大位数**MaxDigitsNumber**。*domainName*应该是**real**域名。

not

参见not。

or

参见or (;)。

orelse

参见orelse。

predicate_fullname

```
predicate_fullname : () -> string PredicateFullName.
```

这个谓词返回调用它的谓词全称**PredicateFullName**，全称中带有范围名称的限定。
谓词**predicate_fullname**只能用在子句中，如果在其它的地方使用这个谓词，会在编译时出错。参看predicate_name。

predicate_name

```
predicate_name : () -> string PredicateName.
```

这个谓词返回调用它的谓词名称**PredicateName**。
谓词**predicate_name**只能用在子句中，如果在其它的地方使用这个谓词，会在编译时出错。参看predicate_fullname。

programPoint

```
programPoint : () -> programPoint ProgramPoint.
```

这个谓词返回相应于它被调用的地方的程序点名称。

retract

```
retract : (<fact-term> FactTerm) nondeterm anyfow.
```

从事实数据库中删除了第一个匹配的事实时成功，如果没有匹配的事实则失败。
这个谓词的调用模板是：

```
retract(FactTemplate)
```

这里的**FactTemplate**应该是一个事实项。**retract/1**谓词从相应的事实数据库中删除第一个匹配的**FactTemplate**；通过回溯还可以删除剩余的匹配事实。

注意，**FactTemplate**可以是实例的任意层级，它是与事实数据库中的事实匹配，这意味着所有自由变量在**retract/1**调用时都会被绑定。

FactTemplate可以有任意个匿名变量，也就是变量名可以仅是一个下划线或以一个下划线开头的变量名，只要是这个变量在子句中只出现一次。例如：

```
retract(person("Hans", _Age)),
```

它会删除第一个匹配的事实，这个事实的第一个参数是“**Hans**”而第二个参数可以是任意的。

当删除的事实声明是determ的时，对**retract/1**的调用将是确定性的。

参看retractall/1和retractFactDb。

retract/1谓词不能应用于single事实或事实变量。

如果在工程当前范围中声明了single事实，以自由的**FactTemplate**变量调用**retract/1**时要特别小心。要是对single事实做retract，会产生运行时错误。

再没有匹配的事实时，**retract/1**谓词就会失败。

retractall

```
retractall：(<fact-term> FactTerm) .
```

从事实数据库中删除所有匹配的事实。

这个谓词的调用模板是：

```
retractall(FactTemplate)
```

FactTemplate应是一个事实项。

retractall/1删除所有与给定的*FactTemplate*匹配的事实。它总是成功，即使没有删除什么事实时也会成功。

如果想要删除的是**single**事实，就会产生编译时的错误。

由**retractall/1**不可能得到任何输出值，因此，调用时所有变量必须都是绑定的或是单个下划线（匿名的）。注意，**FactTemplate**可以是实例的任意层级，但自由变量必须是单个下划线（无条件匿名），与retract/1不同，以下划线开头的条件匿名变量（如**_AnyValue**）不能在**retractall/1**中使用。

参看retract/1和retractFactDb/1。

retractFactDb

```
retractFactDb：(factDB FactDB).
```

从命名的内部事实数据库**FactDB**中删除所有的事实。

这个谓词的调用模板是：

```
retractFactDb(FactDB)
```

retractFactDb/1从命名的内部事实数据库**FactDB**中删除所有的事实。注意，对**single**的事实或

事实变量仍会原样保留下来。参看retractall/1和retract/1。

retractall/2

废弃的谓词！用retractFactDb/1替代。

sizeBitsOf

```
sizeBitsOf：(<domain> DomainName) -> unsigned BitSize.
```

获取指定域**DomainName**的实体在内存中占用的比特位数。
这个谓词的调用模板是：

```
BitSize = sizeBitsOf(DomainName)
```

这个编译时谓词以域**DomainName**为输入参数，返回该域的实体占用内存的大小，结果以比特位计。
对整数域**sizeBitsOf/1->**返回的值是域声明中的size字段的定义值。对整数域来说，下面的关系总是成立的：

```
sizeOfDomain(domain)*8 - 7 <= sizeBitsOf(domain) <= sizeOfDomain(domain)*8
```

参看sizeOfDomain/1->。

sizeOf

```
sizeOf：(<term> Term) -> integer ByteSize.
```

获取指定项**Term**占用的内存字节数。
这个谓词的调用模板是：

```
ByteSize = sizeOf(Term)
```

sizeOf/1->函数以一个项为输入参数，返回**Term**项占用内存的字节数**ByteSize**。

sizeOfDomain

```
sizeOfDomain：(<domain> DomainName) -> integer ByteSize.
```

获取指定域**DomainName**的实体在内存中占用的字节数。
这个谓词的调用模板是：

```
ByteSize = sizeOfDomain(DomainName)
```

这个编译时谓词以域**DomainName**为输入参数，返回该域的实体占用内存的大小，结果以字节计，**ByteSize**的值是整数域的。可以对比一下sizeBitsOf/1->，在它那里以比特计算大小。

sourcefile_lineno

```
sourcefile_lineno：() -> unsigned LineNumber.
```

返回编译器处理的源文件的当前行号。

sourcefile_name

```
sourcefile_name : () -> string FileName.
```

返回编译器处理的源文件名。

sourcefile_timestamp

```
sourcefile_timestamp : () -> string TimeStamp..
```

返回表示编译器正在处理的源文件日期时间的串，格式为**YYYY-MM-DD HH:MM:SS**，其中：

YYYY – 年
MM – 月
DD – 日
HH – 小时
MM – 分钟
SS – 秒

succeed

```
succeed : ().
```

谓词**succeed/0**总是成功。

toAny

```
toAny : (Term) -> any UniversalTypeValue.
```

将指定的**Term**转换为**any**通用项类型的值。
这个谓词的调用模板是：

```
UniversalTypeValue = toAny(Term)
```

toBinary

```
toBinary : (Term) -> binary Serialized.
```

将项**Term**转换成binary表示。
这个谓词的调用模板是：

```
Serialized = toBinary(Term)
```

当（某个域*domainName*）的项**Term**转换成二进制项时，可以安全地存储在文件中或是在网络上传给另一个程序。以后这个二进制项值**Serialized**还可以用函数toTerm/1->转换回Visual Prolog项（反转项的域要适合于*domainName*）。

toBoolean

```
toBoolean : (<term> SubGoal) -> boolean Succeed.
```

这个元谓词的目标是要把（对一个谓词或事实的）确定性的调用转换成procedure的，返回boolean域的值。

这个谓词的调用模板是：

```
True_or_False = toBoolean(deterministic_call)
```

元谓词**toBoolean/1->**返回boolean值。如果确定性调用成功则返回结果是**true**，如果确定性调用失败返回结果是**false**。

toEllipsis

```
toEllipsis : (any* AnyTermList) -> ....
```

这个谓词由通用类型**any**的项表中创建省略块 ... （也就是特定的可变量参数块）。这样的省略块可以传递给需要可变数量参数（也就是所声明的ellipsis (...)）的谓词如**write/...** 。

这个谓词的调用模板是：

```
EllipsisBlock = toEllipsis(<any_term_list>), write(EllipsisBlock )
```

参看fromEllipsis/1->。

toString

```
toString : (Term) -> string Serialized.
```

将指定的项**Term**转换为串表示。

这个谓词的调用模板是：

```
Serialized = toString(Term)
```

当（某个域的）**Term**项转换成一个串时，它可以安全地保存在文件中或是经网络传送给另外的程序。以后串值还可以用函数toTerm/1->转换回Visual Prolog项（反转项的域要适合于*domainName*）。

toTerm

```
toTerm : (string Serialized) -> Term.
toTerm : (binary Serialized) -> Term.
toTerm : (any Serialized) -> Term.
toTerm : (<domain> Type, string Serialized) -> Term.
toTerm : (<domain> Type, binary Serialized) -> Term.
toTerm : (<domain> Type, any Serialized) -> Term.
```

将指定的项**Serialized**的串/二进制/**any**表示转换为相应域的返回值变量**Term**。域可以明确指定，也可以留给编译器做合适的选择。

这个谓词的调用模板是：

```
Term = toTerm(Serialized) %隐含域
```

```
Term = toTerm(domainName, Serialized) %以domainName明确指定域
```

如果没有指定域，编译时编译器必须能够确定返回值**Term**的域。要注意，二进制版的**toTerm**谓词几乎是逐字节地做转换，并且只检查返回值**Term**的域所要求的与**Serialized**数据的兼容性。保证提供的**Serialized**二进制数可以正确地转换到期望的域是程序员的事情。

toTerm谓词与toBinary/1->和toString/1->谓词是配套的，当（某个域的）**Term**项由toBinary/1->或toString/1->转换成一个二进制或串的**Serialized**时，它可以安全地保存在文件中或是经网络传送给另外的程序。以后用对应的函数**toTerm/1->**还可以把这个串/二进制值**Serialized**转换回Visual Prolog的项**Term**。为正确地反转项，子句变量**Term**的域要适合于原来的域*domainName*。

参看tryToTerm。

如果编译时编译器不能确定返回值的域，则出现编译时错误。

异常

当**toTerm**谓词不能转换串或二进制项为指定域的项时，会产生运行时错误。

tryToTerm

```
tryToTerm : (string Serialized)-> Term.
tryToTerm : (binary Serialized)-> Term.
tryToTerm : (any Serialized) -> Term.
tryToTerm : (<domain> Type, string Serialized) -> Term.
tryToTerm : (<domain> Type, binary Serialized) -> Term.
tryToTerm : (<domain> Type, any Serialized) -> Term.
```

如同toTerm一样，转换相应的串/二进制/**any**表示的**Serialized**为项**Term**，它们之间的差别是：谓词**tryToTerm**如果不能把相应的串或二进制项转换为指定域的项时就失败，而toTerm谓词则会产生异常。

参看toTerm。

tryConvert

```
tryConvert : (<type> Type, Value) -> <type> Converted determ.
```

核查输入项**Value**能否被严格地转换为指定域**Type**的项，返回转换后的项**Converted**。

这个谓词的调用模板是：

```
ReturnTerm = tryConvert(returnDomain, InputTerm)
```

参数：

- **returnDomain**：规定**tryConvert/2->**谓词尝试将指定项**InputTerm**转换到该域，它可以是当前范围可访问的任何域。域名*returnDomain*在编译时必须是确定的，即不能来自于变量。
- ***InputTerm***：要转换的项，它可以是任意Prolog项或表达式，如果是表达式，转换前要求值。
- ***ReturnTerm***：返回的项，它是*returnDomai*域的。

tryConvert/2->的转换规则与内含谓词convert/2->一样，但convert/2->产生转换错误时，同样

条件下的**tryConvert/2->**会是失败。

如果相应的转换成功则这个谓词成功，反之它就失败。该谓词尝试把**InputTerm**彻底转换到指定的*returnDomain*域，若它不能完成要求的转换，就会失败。当**tryConvert/2->**谓词成功时，它会返回转换到指定域*returnDomain*的项**ReturnTerm**。

允许的转换及规则请参看带核查的显式转换谓词convert/2->。还可参看uncheckedConvert/2->。

uncheckedConvert

```
uncheckedConvert : (<type> Type, Value) -> <type> Converted.
```

将值转换为另一个类型，不带核查。

这个谓词的调用模板是：

```
ReturnTerm = uncheckedConvert(returnDomain, InputTerm)
```

参数：

- **returnDomain**：指定**uncheckedConvert**要将**InputTerm**非安全性地转换到该域，它可以是当前范围能够访问的任何域。**ReturnTerm**和**InputTerm**的比特位的长短应该是一样的。域名称*returnDomain*在编译时必须是确定的，也就是说，不能用变量来表示。
- ***InputTerm***：指定要转换的值，它可以是任意Prolog项或表达式。如果是表达式，在转换前先要求值。
- ***ReturnTerm***：返回参数**ReturnTerm**是*returnDomain*类型的。

uncheckedConvert谓词对**InputTerm**进行求值，将其类型不做任何存储模式修改就转换到*returnDomain*的类型并与**ReturnTerm**合一。它在运行时不做核查，只在编译时检查被转换域的比特尺度同一性。所以差不多任何项都可以很莽撞地转换为别的类型。因而，不正确地使用它可能会导致很糟糕的结果，一定要小心！强烈建议，只要可能就避免使用它而代之以convert/2->和tryConvert/2->。不过，当对象是由COM系统返回的时候，需要用**uncheckedConvert**来进行转换，因为Prolog程序没有关于它的实际类型的信息。

upperBound

```
upperBound : (<numeric-domain> NumericDomain) -> <number-domain> UpperBound.
```

返回指定数字域的上限值。

这个谓词的调用模板是：

```
UpperBound = upperBound(domainName)
```

upperBound是一个编译时谓词，它返回指定数字域*domainName*的上限值。**upperBound**谓词返回的值**UpperBound**也是*domainName*域的。*domainName*参数必须是任意数字域的名称，编译时必须是确定的（即不能来自于变量）。

参看lowerBound/1->。

如果指定的域不是数字域，则会出现编译时错误。

21. 编译指令

编译器的每个指令都是以字符#开头的，所支持的指令如下：

- **#include**, **#bininclude** – 文件包含；
- **#if**, **#then**, **#else**, **#elseif**, **#endif** – 条件语句；
- **#export**, **#externally** –输出及引入类；
- **#message**, **#error**, **#requires**, **#orrequires** – 编译时间信息；
- **#options** – 编译器选项。

源文件的包含

编译指令**#include**用于将其它文件的内容在编译时包含进用户开发的程序代码中。其语法如下：

```
Pp_dir_include :
  #include String_literal
```

文字串（string literal）应该指向一个已有文件名。文件名可以包含路径名，但要记住：用于子目录的反斜杠字符\是一个换码序列符，因此在要使用它的地方要写两次：

```
#include "pfc\\exception\\exception.ph"
% 包含pfc\exception\exception.ph文件
```

或者用@字符在文件名前作前缀，像这样：

```
#include @"pfc\vpi\vpimessage\vpimessage.ph"
%包含pfc\vpi\vpimessage\vpimessage.ph文件
```

从语义上来讲，这个指令使用“仅包含第一次出现的文件”策略，也就是说，如果一个编译单元中对同一个文件有多个包括指令，则只采用第一个出现的包括指令。

各包含文件必须含有分开的已经完成的范围。包含文件不能含有未完成的范围，也就是说，它含有的是分开的已经完成的接口声明、类声明、类实现或/和分开的编译指令。

编译器按下述方法来查找指定包括的源文件：

1. 如果文件名中包括了绝对路径，就直接包括该文件；
2. 否则，编译器就按命令行选项**/Include**定义的路径搜索指定的包含文件名，有多个路径时按顺序使用。可以在VDE中**Project Settings**的**Directories**标签里的**Include Directories**中设置这些路径。

如果编译器找不到指定文件，就会出现编译时错误。

二进制文件的包含

编译指定**#bininclude**用于（将由string literal串指定的）文件内容作为::binary类型的常数包含进用户开发的程序源代码中。其语法如下：

```
Pp_dir_bininclude :
  #bininclude ( String_literal )
```

这个指令可以用在任何::binary常数可以使用的地方。string literal应该指向一个已有文件名。这个指令的语法与前面“源文件的包含”中的**#include**指令是一样的，搜索指定文件的方法也一样。

典型的使用方法如下：

```
constants
  myBin : binary = #bininclude ("Bin.bin").
  % 由“Bin.bin”文件创建一个二进制常数值
```

创建二进制常数时，编译器会在创建的常数之后加上EOS符号，它可以保证编译指令在如下使用方式中的安全性：

```
constants
  text : string = uncheckedConvert(string, #bininclude("text.txt")).
    % 这里的 text.txt 是个文本文件，它通常不是以0结尾的。
```

输出和引入类

编译指令**#export**和**#externally**分别用于确定要输出的类及要引入的类。其语法如下：

```
Pp_dir_export :
  #export ClassNames-comma-sep-list
Pp_dir_export :
  #externally ClassNames-comma-sep-list
```

这两个编译指令只能用于不构造对象的类classNames。而且，它们也只能用于外部范围，这就是说，它们不能用在接口和类的声明里，也不能用在类的实现里。

缺省时，在一个执行单元内部的谓词对其它执行单元来说是完全隐藏的。而编译指令**#export**使它所指定的那些类名对外部开放了，因而运行时其它执行单元就可访问这些类中声明的模块里的所有谓词。

一般情况下，**#export**编译指令用在目标模块是DLL的工程中，它列出一个DLL中所声明的类，这些列出的类就可以被使用这个DLL的其它模块所访问。

如果某个编译单元输出一个类，该编译单元就应该含有这个类的实现。

#export编译指令还可以用在**#if**编译指令中规定condition表达式。例如，假设在一个编译单元开始的地方编译器碰到了这样的**#export**编译指令：

```
#export className
```

而在后面编译器又遇到一个**#if**编译指令，它以**#export**编译指令为条件表达式，比如说是这样的：

```
#if #export className #then ... #endif
```

那么，编译器就会对**#export**条件表达式求值，在我们这里其结果是**true**，接着就会执行条件编译指令的**#then**分支。

比如，下面这个例子中，就能保证把**some.pack**包包含到所编译的单元中：

```
#export className
...
#if #export className #then #requires "some.pack" #endif
```

而另一方面，如果没有前一个**#export**编译指令，编译器对**#export**条件表达式的求值就会是**false**，也就不会执行**#if**的分支**#then**，比如例子是这样：

```
#if #export className #then #requires "some.pack" #endif
```

这时，就不会请求包含**some.pack**包。

#externally编译指令是与**#export**编译指令配对的，它可以替代（也可以同时使用）**IMPORTS**指令用于定义文件中。这个指令列出一些类，它们是在一个模块中声明的但却是在其它模块中实现的。这样，当编译器遇到这样的类时就不会发生错误。引述的类可以是在DLL中实现（并输出）的，运行时可以将这些DLL链接到这个模块上来。

#export和**#externally**编译指令可以在Conditional Compilation中用作条件布尔表达式。例如：

```
#if #export className #then #include "Some package.pack" #endif
```

编译时消息

在编译工程模块时，可以用编译指令**#message**、**#requires**、**#orrequires**和**#error**来发布用户定义的消息到一个列表文件中或中断编译过程。

这些指令既可以用在范围（接口声明、类声明或类实现）之外，也可以用在范围内但在段之外。其语法如下：

```
Pp_dir_message : one of
  #message String_literal
  #error String_literal
  #requires String_literal Pp_dir_orrequires-list-opt
  #orrequires String_literal
```

编译器遇到上述任一指令时，都会产生相应的警告信息并把指令的文本内容写到一个列表文件中去。

编译指令中可以指定一个列表文件名：

```
/listingfile:"FileName"
```

注意，在冒号前后都不要有空格。

缺省时，编译器**不会**为**#message**、**#requires**和**#orrequires**指令产生消息，程序员可以用编译器选项打开这些消息：

```
/listing:message
/listing:requires
/listing:ALL
```

在这种情况下，当编译器遇到这些指令，比如：

```
#message "Some message"
```

就会把下面的文本写到列表文件中去：

```
C:\Tests\test\test.pro(14,10) : information c062: #message "Some message"
```

#requires (**#orrequires**)指令会把用户定义的所需要的源（目标）文件的消息放到列表文件中去。**#orrequires**指令不能单独使用，紧靠它之前（用空格或注释隔开）应有**#requires**指令。

#error指令总是终止编译并发布用户定义的错误消息到列表文件中，如：

```
C:\Tests\test\test.pro(14,10) : error c080: #error "Compilation is interrupted"
```

可以对这些消息进行分析并采纳所需要的措施。例如，VDE会对**#requires**和**#orrequires**指令给出的消息进行分析，自动地添加所有需要的PFC包及标准库到被编译的工程（参看**Handling Project Modules**主题）。

#requires和**#orrequires**指令的例子：

```
#requires @"\Common\Sources\CommonTypes.pack"
#orrequires @"\Common\Lib\CommonTypes.lib"
#orrequires @"\Common\Obj\Foreign.obj"
#if someClass::debugLevel > 0 #then
  #requires @"\Sources\Debug\Tools.pack"
  #orrequires @"\Lib\Debug\Tools.lib"
#else
  #requires @"\Sources\Release\Tools.pack"
  #orrequires @"\Lib\Release\Tools.lib"
#endif
#orrequires "SomeLibrary.lib"
#requires "SomePackage.pack"
#if someClass::debugLevel > 0 #then
  #orrequires @"\Debug\SomePackage.lib"
#else
  #orrequires @"\Release\SomePackage.lib"
#endif
```

#message指令的例子：

```
#message "Some text"
#if someClass::someConstant > 0 #then
  #message "someClass::someConstant > 0"
#else
  #message "someClass::someConstant <= 0"
#endif
class someClass
  #if ::compiler_version > 600 #then
    #message "New compiler"
    constants
      someConstant = 1.
  #else
    #message "Old compiler"
    constants
      someConstant = 0.
  #endif
end class someClass
```

#error指令的例子：

```
#if someClass::debugLevel > 0 #then
  #error "Debug version is not yet implemented"
#endif
```

编译器选项指令

编译指令**#options <string_literal>**影响整个编译单元。这个指令只能用在外部范围及主源文件

的条件编译语句中（编译单元通过这个源文件传递给编译器），否则就会出现编译时的警告消息并忽略这个指令。

<string_literal>只能包含下面的编译选项：

```
"/Warning"
"/Check"
"/NOCheck"
"/Optimize"
"/DEBug"
"/GOAL"
"/MAXErrors"
"/MAXWarnings"
```

对其它的输入编译器会产生非法选项的错误消息。如果有若干个**#options**指令，则会按文本顺序处理它们。

条件编译

条件编程结构是Visual Prolog的部件。只有编译指令、编译单元和程序段（包括空的）可以条件化。其语法如下：

```
ConditionalItem :
  #if Condition #then CompilationItem-list-opt ElseIfItem-list-opt ElseItem-opt #endif

ElseIfItem :
  #elseif Condition #then CompilationItem

ElseItem :
  #else CompilationItem
```

这里的***Condition***可以是任意表达式，它在编译时应能得到失败或成功的求值结果。

各个条件编译语句必须在一个文件中，也就是说，嵌套的相同层次的编译指令**#if**、**#then**、**#elseif**和**#else**（如果有的话）及**#endif**必须在一个文件中。

编译时，编译器会对条件求值，以便确定哪些部分要包括在最终文件中。排除在最终程序外的那些部分称为死分支。

所有条件编译分支项都要进行语法检查，语法上必须正确。即便是对死分支，语法上也要正确。但编译器只对所需要的条件做求值运算，不会对死分支上的条件求值。

条件不能依赖于任何在条件句中的代码。

下面的例子是非法的，因为条件所依赖的范围与常数是在条件分支中声明的。

```
#if aaa::x > 7 #then % 错误!
class aaa
  constants
      x = 3
end class aaa
#else
class aaa
  constants
      x = 23
end class aaa
#endif
```

22. Attributes（特性）

各种定义和声明可以附加其特性。本节描述特性的一般语法及特性安放的位置，还要介绍一些特性的具体意义。

语法

```
Attributes :
   [ Attribute-comma-sep-list ]

Attribute : one of
   LowerCaseIdentifier
   LowerCaseIdentifier ( Literal-comma-sep-list )
```

其中的literal必须是数字或文字串。

插入位置

接口、类和实现的特性放在紧随范围限定符之后。

```
InterfaceDeclaration :
   interface IinterfaceName
      ScopeQualifications
      Attributes-opt
   Sections
   end interface IinterfaceName-opt

ClassDeclaration :
   class ClassName ConstructionType-opt
      ScopeQualifications
      Attributes-opt
   Sections
   end class ClassName-opt

ClassImplementation :
   implement ClassName
      ScopeQualifications
      Attributes-opt
   Sections
   end implement ClassName-opt
```

常数、域、谓词、属性及事实的特性放在其结尾（也就是在结尾的句点之前）。

```
ConstantDefinition: one of
   ConstantName = ConstantValue Attributes-opt
   ConstantName : TypeName = ConstantValue Attributes-opt

DomainDefinition:
   DomainName FormalTypeParameterList-opt = TypeExpression Attributes-opt

PredicateDeclaration :
   PredicateName : PredicateDomain LinkName-opt Attributes-opt
```

PredicateName : *PredicateDomainName LinkName*-**opt** *Attributes*-**opt**

PropertyDeclaration :
PropertyName : *PropertyType FlowPattern*-**list-opt** *Attributes*-**opt**
FactDeclaration :
FactVariableDeclaration Attributes-**opt**
FactFunctorDeclaration Attributes-**opt**

形式参数的特性放在其结尾处。

FormalArgument :
TypeExpression ArgumentName-**opt** *Attributes*-**opt**

特性的具体说明

byVal

参数直接由堆栈传递而不用指针。适用于**language**指定为**stdcall**、**apicall**或**c**的谓词形式参数。

例：

```
predicates
  externalP : (point Point [byVal]) language apicall.
```

deprecated

该实体已经弃用。文字中描述如何迁移。该实体实际上还是存在的，但使用它会引起一个警告。在以后的Visual Prolog版本中这个实体将不再出现。适用于成员声明和范围。

例：

```
predicates
  oldFasioned : (string Arg) [deprecated("Use newFasion instead")].
```

formatString

该参数是后面省略号参数的格式串。适用于谓词带省略号参数的串参数。使用时编译器在可能的条件下会检核实际参数与相关格式串的对应关系。

例：

```
predicates
  writef : (string Format [formatString], ...).
```

in

参数是输入参数。适用于谓词的形式参数。

例：

```
predicates
  pred : (string InputArg [in]).
```

inline

inline更改结构（带有限定符align的单选函子域）的内存配置方式。这个特性主要是为了适应其它的语言，对纯Visual Prolog语言来说用不着。它用于以下三种情况中：

- 对一个结构使用内联而不是用指针
- 内联固定大小的串
- 内联固定数量的字节

在结构字段(struct field)应用inline时，结构的数据是用内联的方式而不是指针与结构发生关系的。

例：

```
domains
  point = p(integer X, integer Y).

domains
  rectangle =
    r(
      point UpperLeft [inline],
      point LowerRight [inline]
    ).
```

由于和是内联结构，它与下面这种内在布局是一样的：

```
domains
  rectangle2 =
    r2(
      integer UpperLeft_X,
      integer UpperLeft_Y,
      integer LowerRight_X,
      integer LowerRight_Y
    ).
```

如果在一个结构中对string或string8应用inline(<size>)，则该结构就含有固定大小的<size>个字符（char或char8）的串。若这个串没有达到<size>指定的长度就会用零做结尾，如果有<size>个字符就不会这样做。

例：

```
domains
  device =
    device(
      integer Id,
      string DeviceName [inline(12)]
    ).
```

DeviceName是一个内联的长度为12的Unicode串。该结构形式与下面的一样：

```
domains
  device =
    device(
      integer Id,
      char DeviceName_01,
      char DeviceName_02,
      char DeviceName_03,
      char DeviceName_04,
      char DeviceName_05,
      char DeviceName_06,
      char DeviceName_07,
      char DeviceName_08,
      char DeviceName_09,
      char DeviceName_10,
      char DeviceName_11,
      char DeviceName_12
    ).
```

如果对pointer应用inline(<size>)，则该结构就含有固定大小的<size>个字节，pointer就会成为那个域（field）的指针：

例：

```
domains
  mark =
    mark(
      integer Position,
      pointer Data [8]
    ).
```

是内联的8字节的指针，其结构形式与下面的一样：

```
domains
  mark2 =
    mark2(
      integer Position,
      byte Data_1,
      byte Data_2,
      byte Data_3,
      byte Data_4,
      byte Data_5,
      byte Data_6,
      byte Data_7,
      byte Data_8
    ).
```

Data指向**Data_1**。

noDefaultConstructor

用于指定类不要隐含的缺省构造器，使这个类根本就没有任何公共的构造器。适用于对象创建类声明。

例：

```
class classWithoutPublicConstructors : myInterface
    [noDefaultConstructor]
  ...
end class classWithoutPublicConstructors
```

out

参数是输出参数。适用于谓词的形式参数。

例：

```
predicates
  pred : (string OutputArg [out]).
```

programPoint

用于谓词或构造器声明，表示其有一个额外的输入参数，这个参数描述在程序中该谓词被调用的位置（程序点）。这个附加的参数具有programPoint类型，这个类型是在PFC的core类中这样声明的：

```
domains
  programPoint = programPoint(hScopeDescriptor ClassDescriptor, string PredicateName, sourceCursor
```

这类谓词或构造器的子句名应该带有后缀“_explicit”。适用于谓词或构造器的声明。

例：

```
predicates
  pred : (string String) [programPoint].
clauses
  pred_explicit(ProgramPoint, String) :-
```

```
...
```

这个特性在对谓词的叙述中有较细致的说明，它是异常机制的关键内容，可以参看异常处理章节。

retired

表明该实体已经不再使用。文字中描述如何迁移。实体已经不存在了。

例：

```
predicates
    veryOldFasioned : (string Arg) [retired("Use newFasion instead")].
```

sealed

指示一个接口不被其它任何接口支持。这可以创建更有效的代码，因为编译器可以对这一类对象做某种优化。适用于对象创建类声明为一个构造器接口。

例：

```
interface myInterface
    [sealed]
...
end interface myInterface

class myClass : myInterface
...
end class myClass
```

union

用于创建有若干选项但并没有真实函子的函子域。它应该只使用在低层接口中模拟C/C++ union结构。适用于带有若干选项及对齐的函子域。

例：

```
domains
    u64var = align 4
        u64(unsigned64 Value64);
        u64_struct(unsigned Low32, unsigned High32)
        [union].
```

used

未使用的局部成员可以标记为 used 以防止编译器发出警告和删除它的代码。适用于局部成员。

例：

```
predicates
    seeminglyUnused : () [used].
```