# · CLIPS 用户向导

水银测试版

2007.12.31 Joseph C. Giarratano, Ph.D.

## 自述文件

通往智慧的第一步是你得承认你的无知，其次是你不必让全世界都知道你的无知。

这部分被称作序言，但是在还没有人读过它之前，我将它重新命名成一个惯用的标题，以便让计算机用户选择性的遵从。另一个建议是将之命名为：“别读我”章节，但如今的人们相信所有他们阅读过的一切，我恐怕他们真的就不读它了。

序言的目的，噢，抱歉，我应该称之为自述文件，它提供了书本中所包含知识的元知识。之所以称谓为元知识，是指它是关于知识的知识。所以关于自述文件的描述事实上我们得称之为“元元知识”。如果你被搞糊涂了或者你对这些不感冒，那么你可以跳开去选择从书中任何地方看起，因为我得照顾到所有我的读者。

## CLIPS 是什么？

CLIPS 是一种专家系统工具，最初由 NASA/Lyndon B. Johnson 太空中心软件技术研究室开发出来。自 1986 年首次发布以来，CLIPS 经历了不断的改进和完善。现在它已经被广泛的应用在数以万计的全球用户中。

CLIPS 被开发出来以促进集成人类知识和经验的软件发展。

在 CLIPS 中，知识的表示有三种方式：

- 规则，规则表示法是基于启发式经验知识库的首要选择。
- 自定义函数和通用函数，这种方式是程序式知识表示的首选。

- 面向对象设计，也是程序式知识表示的首选。面向对象的程序设计被支持的 5 个普遍接受的特征是：类，消息处理函数，抽象，封装，继承和多态性。模式匹配可以是对象和事实。

你可以仅用规则，或者仅用对象或者两者混合使用来开发软件。

CLIPS 同时支持与其他语言的集成，如 C 和 Java。事实上，CLIPS 是 C Language Integrated Production 的缩写。规则能基于事实与对象的匹配，规则和对象同时组成了一个集成系统。除了被当作一个独立的工具之外，CLIPS 还能被程序语言调用，运行其函数，然后返回给调用函数控制权。同样的，程序代码也能作为一个外部函数在 CLIPS 中被定义和调用。当外部代码执行完毕后，控制权返回到 CLIPS。

如果你已经对面向对象的程序设计语言如 C++，Smalltalk，Objective C 或者 Java 很熟悉了，想必你已经了解面向对象在软件开发中的诸多优点了。如果你并不熟悉，你将会发现 CLIPS 是一款将面向对象概念贯彻于软件开发非常优秀的工具。

## 这本书关于什么？

CLIPS 用户向导是一个介绍 CLIPS 的基本特征的指南，而不是打算对该工具做一个广泛的探讨。本书姐妹篇为 CLIPS 参考手册，它提供关于该话题的所有广泛的探讨和更多其它内容。

## 这本书的适用读者群？

CLIPS 用户向导的目的是对专家系统提供简单易懂的介绍，适用读者可能对专家系统一无所知或者知之甚少。

CLIPS 用户向导可以被用作教材或者自学材料。仅有的前提是你必须具备高级语言如 Java，Ada，FORTRAN 或者 C 的基本知识。（好的，基本不是指其他的，但是如果被问起，我们不会在公开场和下承认和取消其声

明。）

## 怎样使用这本书？

CLIPS 用户向导为那些想亲身尝试专家系统编程的人们提供了快速入门。例子均具有普遍性。同时，我们知道学习一种新的计算机语言是一个令人沮丧的体验，因此，本书的写作语言将采用轻快和幽默的风格（我希望如此），以代替中规中矩的教科书模式。但愿，这种幽默不会冒犯有幽默感的任何人。

为了最大受益，你最好是在看书的过程中，将书中的实例亲自在文本中打印出来。在你打印实例的过程中，你将会逐渐明白程序的工作原理和当你打错时出现的错误提示。案例的结果输出在案例之后给出。最后，当你看完 CLIPS 用户向导各章节后，你还应该看看在 CLIPS 参考手册中的相关材料。

像其他程序语言一样，你只有亲自写程序才能够真正学好 CLIPS 编程。为了真正学会专家系统编程，你应该多在 CLIPS 中写程序，并对每个问题抱有兴趣。

## 感谢

我十分感谢那些对本书提出意见和评述的人。谢谢 Gary Riley，Chris Culbert，Brian Dantes，Bryan Dulock，Steven Lewis，Ann Baker...（一堆外国人名）。特别感谢 Bob Savely 对 CLIPS 改进的支持。

# · 第一章 事实

如果你忽视事实，你将永远不会担心自己的过错。

本章将对专家系统的基本概念做简单的介绍。你将会知道在 CLIPS 中怎样插入和移出事实。如果你正在使用的机器是苹果机或者是 IBM（或可兼容）的 CLIPS 视窗版本，那么你可以通过鼠标来选择相关的命令来代替输入命令行。键盘上的箭头键也可以移动光标对菜单选项进行选择。

## 序言

CLIPS 是一种被用来编写专家系统应用程序的计算机语言。专家系统是一组计算机程序，专门用来模仿人类专家的技能和知识。相比之下，一些普通的程序如报表程序，文本处理器，电子表格，电脑游戏等等，并没有包含人类的技能和知识。（专家的定义之一：就是某人带着他的公文包在离家 50 公里之外。）

CLIPS 之所以被称之为专家系统工具，是因为它是一个开发专家系统的完整环境，包括一个整合版本和一个调试工具。壳这一词被保留在 CLIPS 负责推理的部分中。CLIPS 的壳提供了专家系统的基本元素：

1. 事实表和实例表：数据的全局存储。
2. 数据库：包括所有的规则和规则表。
3. 推理机：控制所有规则的执行。

CLIPS 的程序一般包含有规则，事实和对象。推理机决定了哪条规则应该被执行和在什么时候被执行。一个用 CLIPS 写成的基于规则库的专家系统程序是一个数据-驱动型程序，程序里的事实，对象数据通过推理机的激活执行。

这里有一个例子可以帮助你知道 CLIPS 是如何与其他程序语言如 Java，Ada，BASIC，FORTRAN 和 C 区别开来的。在程序语言中，执行并不一定需要数据，那是因为在那些语言中的声明已经足够引起执行了。举例说明，在 BASIC 语言中，PRINT 2+2 的声明会被立即执行，该声明是一个完整的声明，并不需要额外的数据去驱动执行。然而，在 CLIPS 中，规则的执行必需数据来驱动。

最初，CLIPS 仅有表示规则和事实的能力，然而，在 6.0 版本中已经允许规则与对象的匹配，与规则与事实匹配一样。同时，通过发送消息来应用对象不必需要规则了，如果你仅仅只是用对象，那么推理机都可以不需要。在第一章到第七章中，我们将讨论 CLIPS 的事实和规则，八到十二章中包含了 CLIPS 的对象特点。

## 开始和结束

你可以在你的系统中输入相应的运行代码来启动 CLIPS，你将看到 CLIPS 的提示如下所示：

```
CLIPS>
```

此时，你可以开始在 CLIPS 中直接输入命令，这种直接输入命令的方式被称之为最高阶层。如果你拥有 CLIPS 的图形界面版本（GUI），你也可以用鼠标选择相应的菜单来代替输入命令行。请参考 CLIPS GUI 版本的 CLIPS 界面向导，探讨一下其里面的命令支持。在本书中，为了简约和一致性，我们假设所有的命令均为输入方式。

离开 CLIPS 的一般方式是输入 exit 命令，如下：

```
（exit）
```

按照 CLIPS 提示点击返回键。

## 建表

与其他编程语言一样，CLIPS 也有关键字。举个例子，如果你想在事实表中输入数据，你可以使用 assert 命令。

作为一个 assert 实例，在 CLIPS 提示后面正确输入下面的命令：

```
CLIPS>(assert (duck))
```

这里，assert 命令以(duck)作为其参数。记住点击回车键将命令行发送到 CLIPS。

你将看到如下响应：

```
<Fact-1>
```

这表示 CLIPS 已经存储了 duck 的事实，并将其标识为 1。在 CLIPS 中，尖括弧被用来作为条目的分隔符。CLIPS 会自动的增加事实的编号，随着一个或更多的事实被添加，从最高事实-索引进行列表。

注意(assert)和它的参数(duck)均用圆括弧括住，像其他一些专家系统语言一样，CLIPS 采用 LISP 式样语法，用圆括弧作为分隔符。虽然 CLIPS 并不是采用 LISP 语言编写，但是 LISP 影响了 CLIPS 的发展。

## 检查两遍

假设你想查看一下事实表中的内容，如果你的 CLIPS 支持 GUI，你便可以在菜单中选择相应的命令，或者，你还可以通过键盘键入相应的命令行。接下来，我们将来描述一下键盘命令。

查看事实库的键盘命令是 facts 命令。在 CLIPS 提示后输入(facts)，CLIPS 响应后会将事实表列出。一定记得将命令用圆括弧括住，否则 CLIPS 会不识别。在该实例中，(facts)命令的句法如下：

```
CLIPS>(facts)
f-0 (initial-fact)
f-1 (duck)
```

```
For a total of 2 facts.
CLIPS>
```

f-0 和 f-1 为 CLIPS 对事实分配的事实标识。每个事实被添加进 CLIPS，被分配唯一的事实标识，以“f”开头，后面的数字为事实索引。当启动 CLIPS，输入如 clear 或 reset（随后有详细的探讨）后，事实索引将会被归零，然后随着每个事实的添加（assert）逐步加一。（clear）和(reset)命令同时增加一个(initial-fact)事实，编号为 f-0。在 CLIPS 的早期版本中，该事实被 CLIPS 隐式用来初始化一些规则和被用户显式调用来使事实库初始化，但是现在，该事实仅被用来提供向后兼容性。

如果你将 duck 在事实表中输入两次，将会出现什么结果呢？让我们试试看，增加一个新事实（duck），然后调用（facts）命令如下所示：

```
CLIPS>(assert (duck))
FALSE
CLIPS>(facts)
f-0 (initial-fact)
f-1 (duck)
For a total of 2 facts.
CLIPS>
```

CLIPS 返回 FALSE 消息，表示不可能执行该条命令，且你将只能见到原始的事实：“f-1 (duck)”。这说明 CLIPS 不能接受事实的复制输入。然而，CLIPS 中还有一个超越命令：set-fact-duplication，该命令允许事实的重复输入。

当然，你可以输入其他不同的事实。举个例子，增加一个(quack)事实，然后运行(facts)命令，如下：

```
CLIPS>(assert (quack))
<fact-2>
CLIPS>(facts)
f-0 (initial-fact)
f-1 (duck)
f-2 (quack)
For a total of 3 facts.
CLIPS>
```

注意，（quack）事实已经被添加到事实表中了。

事实也会被移出和撤销。当一个事实被撤销，其他的事实索引不会改变，因此会出现事实索引的“丢失”。类似于一个足球运动员离开球队如果没有被补充，其他队员的号码不会因为缺失号码而发生调整（除非他们非常讨厌这个离队的家伙，想要忘掉他曾在队中效力过）。

## · 清除所有事实

Clear 命令将所有的事实从内存中移出，代码如下所示：

```
CLIPS>(facts)
f-0 (initial-fact)
f-1 (duck)
f-2 (quack)
For a total of 3 facts.
CLIPS>(clear)
CLIPS>
```

事实表中的所有事实被清除。

（clear）命令实质上将 CLIPS 恢复到起始启动状态，它清除了 CLIPS 的内存空间，重置事实标识为 0 和增加了一个(initial-fact)事实。增加(animal-is duck)事实，然后查看事实表，会发现(animal-is duck)的事实标识为 f-1，这是因为(clear)命令重置了事实表的标识。该命令事实上并不只是起清除所有事实的作用，除此之外，它还清除所有的规则，在下一章中你就会看到。

下面的实例显示了怎样将三个事实加入到事实表，并用(facts)命令查看，然后(clear)命令将这三个事实从内存中清除并重置事实标识为 f-0。

```
CLIPS>(clear)
CLIPS>(assert (a) (b) (c))
<Facts-3>
CLIPS>(facts)
f-0 (initial-fact)
f-1 (a)
f-2 (b)
f-3 (c)
For a total of 4 facts.
CLIPS>(facts 0)
f-0 (initial-fact)
f-1 (a)
f-2 (b)
f-3 (c)
For a total of 4 facts.
CLIPS>(facts 1)
f-1 (a)
```

f-2 (b)

f-3 (c)

For a total of 3 facts.

CLIPS>(facts 2)

f-2 (b)

f-3 (c)

For a total of 2 facts.

CLIPS>(facts 1 2)

f-0 (initial-fact)

f-1 (a)

f-2 (b)

For a total of 2 facts.

CLIPS>(facts 1 3 2)

f-0 (initial-fact)

f-1 (a)

f-2 (b)

For a total of 2 facts.

CLIPS>

注意，仅用一个(assert)便可以增加三个事实：(a),(b)和(c)。最高索引为 3，通过 CLIPS 的信息消息<Fact-3>返回。也可以用每个命令增加一个事实的方式（那些这样做的人也许是为了炫耀他们的打字速度）。

注：(facts)命令的完整语法为：(facts [<start> [<end> [<maximum>]]])，<start>表示显示索引号大于等于<start>的事实，<end>表示小于等于<end>的事实，<maximum>表示显示在<start>和<end>之间最多<maximum>个事实。

## 敏感字段和详解

事实(duck)和(quack)被称之为单字段。一个字段就是一个占位符（命名或未命名），通常拥有一个值。一个简单的类比，你可以将字段想像成一幅画框，这个画框能够装载一幅画，也许画中是你的宠物鸭（也许你会好奇怎样用一幅画表现“quack”,有两个法子：（1）是弄一个示波器来显示一只鸭子说“quack”的波形图，信号的输入来源于一个麦克风；（2）对于那些有科学主义倾向的人，也许还得对“quack”信号做一个傅立叶变换；（3）电视里那些叫卖神奇的祛皱，减肥广告。等等）。只有用 deftemplates 才叫做占位符，将在第五章中进行详细的介绍。

注：这里的（3）提到的电视广告，意思是电视广告里的广告者会大呼小叫的对他们的产品爆发欢呼，声音像鸭子叫一样，讽刺幽默。

(duck)事实是一个单独，未命名占位符的事实，值为 duck。下面有一个关于单字段事实的例子，一个字段即是一个值的占位符。类比想像一下字段，就像碟子(字段)盛食物(值)一样的道理。

未命名字段的顺序非常重要。举例，如果一个事实被定义为：

(Brian duck)

表示一个叫 Brian 的猎人射杀了一只鸭子，那么事实:

(duck Brian)

则表示鸭子猎手射杀了一个叫 Brian 的猎人。与之相比，命名字段的顺序是不重要的，稍后你将在 deftemplate 中看到。

事实上，一个好的软件工程应该采用关系型表示法来表述字段，一个好的事实表示如下：

(hunter-game duck Brian)

表示第一个字段代表猎人，第二个字段代表游戏名称。

现在，一些定义是必需的了。一个表是一组无内在序列的项目集合。之所以称一个表为有序的，意味着表中的位置是非常重要的。一个多字段是有序字段，每个字段都有一个值，特殊符号 nil 意思是无，通常作为一个占位符用在空字段中。举例如下：

(duck nil)

可以表示猎人的捕鸭袋中今天一无所获。

注意，nil 表示了一个占位符，虽然它没有值。举例，试想一个字段就是一个邮箱，没有邮箱和邮箱中没有信件是完全两码事。如果没有 nil，这个事实就是一个单字段事实(duck),如果一个规则依赖于两字段激活，则该单字段事实不会被激活，稍后你会看到的。

这里有许多不同有效的字段类型：float，integer，symbol，string，external-address，fact-address，instance-name 和 instance-address。这些字段类型用来存储字段值的类型。未命名的字段中，值的类型由你的输入决定。在 deftemplates 中，你可以显式的声明字段所包含值的类型。显式的声明加强了软件工程的概念，是产生一个高效软件的编程训练。

Symbol 是一类字段类型，该类型起始于一个可印刷的 ASCII 码并被选择性的加一个 0 或更多的可印刷字符。字段由空格或占位符被普通的分隔。举例：

(duck-shot Brian Gary Rey)

有四个字段，指示了所有的杀鸭猎人。在这个事实中，字段被空格分隔，并由圆括弧括起来。

事实中不能嵌入其他的事实。举例，下面即是一个非法的事实：

(duck (shot Brian Gary Rey))

然而，如果“shot”被当作一个字段名，上面的事实可能是一个合法的 deftemplate 事实。后面的三个人名为该字段下的值。

CLIPS 区分大小写。同样，CLIPS 中特定的符号有特殊的意义。

‘’ ( ) & |< ~ ;? $

“&”，“|”和“～”不会独立的使用或作为符号的任何部分。

一些字符的作用等同于分隔符以结束一个符号。下面的字符的作用等同于分隔符号。

- 所有的不可印刷的 ASCII 码，包括空格，回车键，制表键和换行键。
- 双引号，“”
- 起始和结束圆括号，（）
- &号
- 竖线，|
- 小于，<.这也是尖括号的一部分。
- 波浪字符，~
- 分号，；指示一个注释的开始，回车键结束
- ？和$?也许不能作为一个符号的开始，但是可以插入其中

分号在 CLIPS 的作用是指示一个注释的开始，如果你试图增加一个分号，CLIPS 便会认为你在输入一段注释并等待你的完成。如果你在最高阶

层(top-level)中不经意的输入了一个分号，那么输入一个圆括号的结束部分：）并回车。CLIPS 会以一个错误消息响应并提示给你（就像生活中的某些时候，你得做些错误的事情以使得某些事情正确）。

随着你通读这本手册，你将会逐渐明白上面那些符号的意义。除了“&”，“|”和“~”之外，你将使用其他的表示符号，然而，也许对于有些人，在读程序和试图理解程序运行机理时有些困惑。通常情况下，最好是避免使用这些符号来表示字符，除非你有更好的理由需要用到它们。

下面是这些符号的一些例子：

duck

duck1

duck_soup

duck-soup

duck1-1_soup-soup

d!?#%^

第二类类型的字段是 string。一个字符串必须用双引号引起来，双引号是字段的一部分。引号中可以有 0 个或多个字符。一些例子如下：

“duck”

“duck1”

“duck/soup”

“duck soup”

“duck soup is good!!!”

- 第三和第四种字段类型为数字型字段。该字段用来表示整型或浮点型字段。浮点型通常被简化为 float。（float-point->float）

CLIPS 中的数字均为“long long”整型或双精度浮点型。没有小数点的

数字即是整型，除非它们不属于整型范围。整型的范围由数字的位数决定，N，用来表示整型如下所示：

$-2^{N-1}$ … $2^{N-1}-1$

对于 64 位机器“long long”整型，符合该范围的数字为：

-9，223，372，036，854，775，808 … 9，223，372，036，854，775，807

下面给出一些数字的例子，增加下面的数据到事实中，最后一个数字为指数表示法，用“e”或“E”代替乘以 10。

```
CLIPS>(clear)
CLIPS>(facts)
f-0 (initial-fact)
For a total of 1 fact.
CLIPS>(assert (number 1))
<Fact-1>
CLIPS>(assert (x 1.5))
<Fact-2>
CLIPS>(assert (y -1))
<Fact-3>
CLIPS>(assert (z 65))
<Fact-4>
CLIPS>(assert (distance 3.5e5))
<Fact-5>
```

```
CLIPS>(assert (coordinates 1 2 3))
<Fact-6>
CLIPS>(assert (coordinates 1 3 2))
<Fact-7>
CLIPS>(facts)
f-0 (initial-fact)
f-1 (number 1)
f-2 (x 1.5)
f-3 (y -1)
f-4 (z 65)
f-5 (distance 350000.0)
f-6 (coordinates 1 2 3)
f-7 (coordinates 1 3 2)
For a total of 8 facts.
CLIPS>
```

如你所见，CLIPS 将输入的指数表示法转换成数字 350000.0，这是因为当数字足够小，就会被从指数表示转换到浮点型格式。

注意上面的每个数字前面都有一个符号开头，如“number”，“x”，“y”等。在 CLIPS6.0 版本以前，允许仅一个数字的事实，然而，现在必需一个符号作为第一字段，同时，CLIPS 的一些专用字段不能用来作为第一字段，但是可以用来作为其他字段。举个例子，专用关键字 not 用来指代否定模式，但是不能作为一个事实的第一字段。

一个事实由一个或多个被圆括弧括住的字段组成。为了简单化，我们在前面七章中将仅仅讨论事实，但也有许多对模式匹配应用于对象做了讨论。例外的是，一些函数如 assert 和 retract 仅仅只能用于事实，而不能用于对象。对对象相应的处理方法将会在第八到第十二章中讨论。

一个事实可以是有序的，也可能是无序的。所有前面你已经看到的事实都是有序事实，因为字段的顺序决定了它们的不同。举个例子，注意，CLIPS 会自动将包含相同数字“1”,“2”和“3”的事实区分开。

f-6 (coordinates 1 2 3)

f-7 (coordinates 1 3 2)

有序事实必须用字段对位于其定义的数据。举例说明，有序事实(duck Brian)有两个字段，同样(Brian duck)也有两个字段，然而，CLIPS 将其看作两个不同的事实，因为有序事实字段的值是不同的。相反，事实(duck-Brian)仅有一个字段，因为有一个“-”符号将两个值连结。

定义模板事实(Deftemplate facts)，稍后会做详细的表述，它是无序的，因为它用命名字段来定义数据。这与在 C 和其他语言中应用结构体一样。

多字段通常被由一个或多个的空格，制表，回车或表格组成的空白隔离开来。举例说明，输入下面的例子，你将发现每个被存储的事实都是一样的。

CLIPS>(clear)

CLIPS>(assert (The duck says “Quack”))

<Fact-1>

CLIPS>(facts)

f-0 (initial-fact)

f-1 (The duck says “Quack”)

For a total of 2 facts.

CLIPS>(clear)

CLIPS>(assert (The duck says “Quack” ))

<Fact-1>

CLIPS>(facts)

f-0 (initial-fact)

f-1 (The duck says “Quack”)

For a total of 2 facts.

CLIPS>

回车的使用是为增加可读性。在下面的例子中，每个字段后加一个回车，增加的事实与将字段都写在一行的效果是一样的。

CLIPS>(clear)

CLIPS>(assert (The

duck

says

“Quack”))

<Fact-1>

CLIPS>(facts)

f-0 (initial-fact)

f-1 (The duck says “Quack”)

For a total of 2 facts.

CLIPS>

然而，当你在输入一个字符串的时候，要注意插入回车后的效果，例子如下：

CLIPS>(assert (The

duck

says

“Quack

”))

<Fact-2>

CLIPS>(facts)

f-0 (initial-fact)

f-1 (The duck says “Quack”)

f-2 (The duck says “Quack

”)

For a total of 3 facts.

CLIPS>

如你所见，在双引号中插入的回车在字符串输出中会将双引号的后半部分移到下一行。CLIPS 会认为 f-1 与 f-2 是两个不同的事实，这一点很重要。

同样，我们也注意到 CLIPS 会保存事实中字段里的大写和小写字母。也就是“The”中的“T”和“Quack”中的“Q”。CLIPS 被认为是区分大小写的，因为它将大写和小写字母区别对待。举例说明，增加事实(duck)和(Duck),然后调用(facts)命令，你会发现 CLIPS 增加了两个不同的事实(duck)和(Duck),这正是因为 CLIPS 是区分大小写的缘故。

下面的例子将更清楚的表现了回车应用于表中，增加可读性的作用。增加下面的事实，使用空格和回车将字段合适的安排在行中。破折号和减号被使用来创建单字段，这样，CLIPS 就会将“fudge sauce”作为一个单字段了。

CLIPS>(clear)

CLIPS>(assert (grocery-list

ice-cream

```
                    cookies
                    candy
                    fudge-sauce))
<Fact-1>
CLIPS>(facts)
f-0 (initial-fact)
f-1 (grocery-list ice-cream cookie candy fudge-sauce)
For a total of 2 facts.
CLIPS>
```

如你所见，CLIPS 将回车和制表置换为单空格。当人们在读一段程序时，使用合适的空格会带来许多方便，CLIPS 会将其自动置换为单空格。

- **风格问题**

用事实的第一个字段来描述后续字段的关系是很好的基于规则编程风格。在此风格中，第一个字段被称为关系，事实的剩余字段被用来指定值。例子：(grocery-list ice-cream coolies candy fudge-sauce)中破折号用来将多词组合成一个单字段。

良好的文档处理在专家系统中比其他语言如 Jave，C，Ada 等更显重要，这是因为专家系统中的规则并不是普通的按顺序执行。CLIPS 采用模板(deftemplate)的意义来描述事实以帮助程序员编写程序。

另一个关联的事实是(duck)，(horse)和(cow)。一个好的提交它们的格式如下所示：

```
(animal-is duck)
(animal-is horse)
(animal-is cow)
```

或采用单事实：

(animals duck horse cow)

通过关系 animal-is 和 animals 来表述它们之间的关系，使得人们在阅读代码时能够一目了然。

一个明确的关联，animal-is 和 animals，比隐式的定义(duck)，(horse)和(cow)能使人们得到更多的信息。这个足够简单的例子让任何人都能断定字段间的隐含关系，但当人们在写一个并没有明确关系的事实时，同时也是一个简单的圈套（事实上，使事情复杂化要比使事情简单化简单很多，这是因为人们通常对于复杂的印象比简单要深刻许多。）

## 消除空格

之前我们介绍了空格用来分隔多字段，下面我们将看到在事实中，空格的作用不仅仅如此。举个例子：

```
CLIPS>(clear)
CLIPS>(assert (animal-is walrus))
<Fact-1>
CLIPS>(assert ( animal-is walrus ))
FALSE
CLIPS>(assert (    animal-is    walrus    ))
FALSE
CLIPS>(facts)
f-0 (initial-fact)
f-1 (animal-is walrus)
```

For a total of 2 facts.

CLIPS>

仅有一个事实(animal-is walrus)被添加了，CLIPS 忽视了空格，并认为三个被添加的事实是相同的。因此，当输入相同的事实时，CLIPS 返回 FALSE。CLIPS 不允许输入相同的事实，除非你改变 set-fact-duplicate 设置。

如果你想在事实中包含空格，那么你必须使用双引号，举例如下：

CLIPS>(clear)

CLIPS>(assert (animal-is “duck”))

<Fact-1>

CLIPS>(assert (animal-is “duck ”))

<Fact-2>

CLIPS>(assert (animal-is “ duck”))

<Fact-3>

CLIPS>(assert (animal-is “ duck ”))

<Fact-4>

CLIPS>(facts)

f-0 (initial-fact)

f-1 (animal-is “duck”)

f-2 (animal-is “duck ”)

f-3 (animal-is “ duck”)

f-4 (animal-is “ duck ”)

For a total of 5 facts.

CLIPS>

注意上面，在 CLIPS 中，空格的使用使得每个事实都不同，虽然在我们看来是

同一个事实。

如果你想在字段中包含双引号，该怎么办？正确的方法是使用反斜线符号“\”将双引号插入到事实中，如下面的例子所示：

```
CLIPS>(clear)
CLIPS>(assert (single-quote “duck”))
<Fact-1>
CLIPS(assert (single-quote “\”duck“\”))
<Fact-2>
CLIPS>(facts)
f-0 (initial-fact)
f-1 (single-quote “duck”)
f-2 (single-quote “ “duck” ”))
For a total of 3 facts.
CLIPS>
```

## 撤销事实

现在你已经知道怎么添加一个事实到事实表中，现在是时候学习怎样撤销它们了。将事实表中的事实移除称之为撤销，使用 retract 命令。撤销一个事实，你必须指定所撤销事实的索引作为撤销命令的参数，建立你的事实表如下所示：

```
CLIPS>(clear)
CLIPS>(assert (animal-is duck))
<Fact-1>
CLIPS>(assert (animal-sound quack))
```

<Fact-2>

CLIPS>(assert (The duck says “Quack.”))

<Fact-3>

CLIPS>(facts)

f-0 (initial-fact)

f-1 (animal-is duck)

f-2 (animal-sound quack)

f-3 (The duck says “Quack.”)

For a total of 4 facts.

CLIPS>

如果要移除索引为 f-3 的最后一个事实，键入撤销命令并选择你所要撤销的事实，如下所示：

CLIPS>(retract 3)

CLIPS>(facts)

f-0 (initial-fact)

f-1 (animal-is duck)

f-2 (animal-sound quack)

For a total of 3 facts.

CLIPS>

如果你试图移除一个已经被移除的或者根本不存在的事实，将会出现什么结果？让我们来试试：

CLIPS>(retract 3)

[PRNTUTIL1] Unable to find fact f-3.

```
CLIPS>
```

可以看到，当你试图移除一个不存在的事实时，CLIPS 会发布一个错误提示。

如果你没有给予，你当然也没有道理拿回什么。

现在，让我们撤销其他的事实，如下所示：

```
CLIPS>(retract 2)
CLIPS>(facts)
f-0 (initial-fact)
f-1 (animal-is duck)
For a total of 2 facts.
CLIPS>(retract 1)
CLIPS>(facts)
f-0 (initial-fact)
For a total of 1 fact.
CLIPS>
```

撤销一个事实，你必须指定该事实的索引。

你可以一次撤销多条事实，如下所示：

```
CLIPS>(clear)
CLIPS>(assert (animal-is duck))
<Fact-1>
CLIPS>(assert (animal-sound quack))
<Fact-2>
CLIPS>(assert (The duck says “Quack.”))
<Fact-3>
```

CLIPS>(retract 1 3)

CLIPS>(facts)

f-0 (initial-fact)

f-2 (animal-sound quack)

For a total of 2 facts.

CLIPS>

撤销多条事实，只要在 retract 命令后跟上相应的事实索引号即可。

你也可以用(retract *)撤销所有的事实，这里的*指代所有的事实。

CLIPS>(clear)

CLIPS>(assert (animal-is duck))

<Fact-1>

CLIPS>(assert (animal-sound quack))

<Fact-2>

CLIPS>(assert (The duck says “Quack.”))

<Fact-3>

CLIPS>(facts)

f-0 (initial-fact)

f-1 (animal-is duck)

f-2 (animal-sound quack)

f-3 (The duck says “Quack.”)

For a total of 4 facts.

CLIPS>(retract *)

CLIPS>(facts)

CLIPS>

## · 监视事实

CLIPS 提供了一些帮助你调试程序的命令。其中一个命令可以帮助你连续监视事实(watch facts)的增加和撤销，这比你总是不断输入(facts)命令来查看事实表中的变化要方便得多。

监视事实是通过输入(watch facts)命令来实现的，如下例子所示：

```
CLIPS>(clear)
CLIPS>(watch facts)
CLIPS>(assert (animal-is duck))
==>f-1     (animal-is duck)
<Fact-1>
CLIPS>
```

右双箭头符号==>表示事实正在被添加到内存中，左双箭头<==表示事实正在从内存中移除，如下所示：

```
CLIPS>(reset)
<==f-0     (initial-fact)
<==f-1     (animal-is duck)
==>f-0     (initial-fact)
CLIPS>(assert (animal-is duck))
==>f-1     (animal-is duck)
<Fact-1>
CLIPS>(retract 1)
<==f-1     (animal-is duck)
CLIPS>(facts)
```

f-0 (initial-fact)

For a total of 1 fact.

CLIPS>

(watch facts)命令提供对事实表状态的动态显示，(facts)命令显示的是静态的当前事实表中所包含的事实。关闭监视事实的命令为：(unwatch facts)。

你可以监视的项目有很多，下面列举出来，在《CLIPS 参考指南》中有详细的表述。CLIPS 中的注释以分号开始，分号后面的内容将会被 CLIPS 忽略。

(watch facts)

(watch instances)　　；应用于对象

(watch slots)　　；应用于对象

(watch rules)

(watch activations)

(watch messages)　　；应用于对象

(watch message-handlers)　　；应用于对象

(watch generic-functions)

(watch methods)　　；应用于对象

(watch deffunctions)

(watch compilations)；默认的

(watch statistics)

(watch globals)

(watch focus)

(watch all)　　；监视所有项目

随着你使用到 CLIPS 的更多功能，你将发现(watch)命令在调试过程中非常的有用。通过输入 unwatch 命令可以关闭监视(watch)命令。举例说明，如果要关闭监视编译，则输入(unwatch compilations)即可。

## 一点帮助

CLIPS 提供有效的在线帮助。获得帮助只需输入(help)命令然后回车即可。不久，你将会看到一个细目菜单。更多的关于(help)命令的信息，请参考 HELP_USAGE 帮助章节。退出帮助的方法是一直按回车键，直到 CLIPS 提示出现。如果出现错误消息提示，则表明 CLIPS 没有找到帮助文件：clips.hlp，你可以用(help-path)命令来找出 CLIPS 该文件的路径。

# 第二章 规则

**如果你想你的生活硕果累累，那么别打破规则---而是去制定规则！**

在前面一章中的学习中，你已经对事实有所了解了。现在你将马上看到专家系统的规则将怎样利用事实驱动程序执行。

## 构造良好的规则

完成一项有价值的工作，专家系统必须得有事实和规则。前面你已经知道了事实的添加和撤销，现在你将了解规则是怎样工作的。一条规则与程序语言如 Java，C 或 Ada 中的 IF THEN 表述非常相似。IF THEN 规则可以用自然语言与计算机语言来混合表示，如下所示：

IF certain conditions are true

THEN execute the following actions

上述表述又被称为伪代码，伪代码字面的意思是错误的代码。伪代码

不能被计算机识别和执行，但是它对书写可执行代码提供了有用的指南。伪代码在文档规则中也非常有用。如果你记住 IF THEN 的类比特性，那么将规则从自然语言转化到 CLIPS 语言将很简单。随着你 CLIPS 实践的增加，你将发现在 CLIPS 中写规则非常的简单。你可以在 CLIPS 中直接输入规则，也可以新建一个文本文件，将规则写在里面，然后加载到 CLIPS 中来。

关于鸭子叫声规则的伪代码可以写成如下形式：

```
IF the animal is a duck
THEN the sound made is quack
```

下面是采用 CLIPS 语法将上面的伪代码写成一个事实和一个命名为 duck 的事实。规则名紧跟在关键字 defrule 后面。虽然你可以将规则都写在一行里面，但是我们通常将规则分成几段放在几行里书写，便于程序的阅读和编辑。

```
CLIPS>(unwatch facts)
CLIPS>(clear)
CLIPS>(assert (animal-is duck))
<Fact-1>
CLIPS>(defrule duck
        (animal-is duck)
=>
        (assert (sound-is quack)))
CLIPS>
```

如果你按照上面正确的输入，你便会看到 CLIPS 的提示符出现，否则，你将会看到一个错误消息提示。如果你得到一个错误消息，也许是你拼错

了关键字或你遗漏了圆括号。记住，在一个声明中，圆括弧的左边和右边部分的数目是配套的。

下面将给出一个相同的规则，该规则中增加了对规则每部分的注释。同时也增加了可选的规则头(rule-header)注释：“Here comes the quack”。规则中只能包含一个规则头注释，且必须写在规则名之后和第一个模式(pattern)之前。虽然现在我们只是讨论基于事实的模式匹配，一般来说，模式的匹配时基于模式实体上(pattern entity)的。模式实体是一个事实，也可以是一个用户定义类的实例。基于对象的模式匹配将稍后讨论。

CLIPS 基于模式实体来进行模式匹配。当然，由空格，制表和回车组成的空格将规则的几个部分分隔开来，以增强可读性。其他的注释由分号引导，直到按下回车键结束一行。CLIPS 忽略注释里的内容。

```
(defrule duck  “Here comes the quack”      ; 规则头
      (animal-is duck)                     ; 模式
=>                                         ; THEN 箭头
      (assert (sound-is quack)))           ; 执行
```

- CLIPS 中，同时刻只能仅有一个规则名存在。
  - 输入同一个规则名，如本例中的“duck”，将会更替前面规则名为“duck”里已经存在的一切。也就是说，CLIPS 中可能有许多条规则，但是只能有一条被命名为“duck”的规则。这与其他程序语言中一个程序名只能标识唯一程序段是一样的道理。

规则的常规语法如下所示：

```
(defrule rule_name  “optinal_comment”
      (pattern_1)          ; 由一些在“=>”之前的元素组成的规则左部分
      (pattern_2)
```

```
        .
        .
        .
    (pattern_N)
=>
    (action_1)                  ; 由一些在“=>”之后的元素组成的规则右部分
    (action_2)
        .
        .
        .
(action_M))                     ; 最后一个“)”是与“defrule”前面的“)”配
                                ; 套的。保证你的圆括弧完整，否则你将得到错误
                                ; 消息提示
```

整个规则必须用圆括弧括住，每个模式(pattern)和每个行为(action)都必须用圆括弧括住。行为通常是一类没有返回值(return value)的函数，但是它可以完成一些有用的执行，如(assert)和(retract)。举个例子，一个行为可以是(assert (duck))。这里的函数名是“assert”，它的参数是“duck”。注意，我们并没有希望得到一个如数字型的返回值，而是使得事实(duck)被增加到 CLIPS 中去。CLIPS 中的函数(function)是一段可执行代码，该段代码被特定的函数名标识，返回有用的值或产生有用的副作用，如(printout)。

一个规则通常包含有多个模式和行为。模式和行为的数量并不一定得相等，这就是上面例子中用 N 和 M 来代指的意义。

零个或多个模式写在规则名之后。每个模式包含一个或多个字段。在上面的 duck 规则中，模式为(animal-is duck)，字段为“animal-is”和“duck”。CLIPS 试图将模式与事实表中的事实进行匹配，如果规则的模式与事实匹

配成功，规则将会被激活(activated)而放入到议程(agenda)中。议程中存放的是所有被激活的规则集合。议程中通常包含零个或多个激活的规则。

规则中，模式后面的符号“=>”被称之为箭号(arrow)，箭号是 IF-THEN 规则的 THEN 部分开始的标记（也许可以被读作“意味着”）。

规则的最后部分为零个或多个行为，当规则被触发(fire)时，这些行为将会被执行。在我们的实例中，行为是增加一个事实(sound-is quack)。Fire 一词意味着 CLIPS 已经选定了议程中某条规则并执行。

- 当议程中没有激活的规则时，程序停止执行。

当议程中有多条激活规则时，CLIPS 自动决定哪条规则将被合理的触发。CLIPS 依照增加优先权和特权(salience)来对议程的激活排序。

规则中箭号之前的被称之为左部(LHS)，箭号之后的部分被称之为右部(RHS)。如果没有指定模式，则 CLIPS 会在输入(reset)命令后自动的激活该条规则。

## 让鸭子叫吧

CLIPS 通常会执行议程中最高优先权规则右部的行为部分。随后该条规则将会被移出议程，接下来最高特权规则的行为将会被执行。这样持续执行下去，直到议程中没有激活的规则或输入了停止激活的命令为止。

你可以通过议程(agenda)命令来查看议程中的内容，举例说明：

```
CLIPS>(agenda)
   0      duck: f-1
For a total of 1 activation.
CLIPS>
```

第一个数字“0”表示规则“duck”的激活特权值，“f-1”为事实的标识，(animal-is duck)为匹配激活。如果没有显式的声明特权值，则 CLIPS

默认为 0。特权值的范围为-10000 到 10000。本书中，我们将用 default 的定义来作为标准方式。

如果议程中仅有一个规则，该规则将被触发。前面知道了 duck-sound 规则的模式左部为：

(animal-is duck)

该模式刚好与(animal-is duck)事实符合，因此 duck-sound 规则将会被触发。

模式的字段被称之为字面约束(literal constraint)。之所以称之为字面意味着有一个常数值，与之对立的是值可以改变的变量。在此例中，字面为“animal-is”和“duck”。

输入 run 命令即可使程序运行。敲入(run)并回车，然后输入(facts)命令查看通过该规则有哪些事实被添加。

```
CLIPS>(run)
CLIPS>(facts)
f-0 (initial-fact)
f-1 (animal-is duck)
f-2 (sound-is quack)
For a total of 3 facts.
CLIPS>
```

在操作之前，让我们使用 save 命令来保存 duck 规则，这样你就可以避免重复输入了（如果你还没有将这些保存到编辑器中）。输入命令如下：

(save “duck.clp”)

将 CLIPS 内存中的规则保存到命名为“duck.clp”的文件中，“.clp”是一个简单方便的扩展名，让我们方便知道这是一个 CLIPS 的源文件。注意，从 CLIPS 内存中保存下的代码只保留了双引号内可选规则头的注释，而分号后的注释就没有了。

## 踢你的鸭子

也许此时你会有一个有趣的问题，如果重复执行(run)，结果会这样？当一个规则被事实满足时，该规则会被触发，然而，如果你重复执行(run)，你会发现该条规则不将被触发了。这也许让人有一点沮丧，然而，在你做出一些极端的减轻沮丧的事情之前---如狠踢你的宠物鸭---你得多了解一些专家系统的基本原理。

当规则的模式与下面的几点匹配时，规则被激活：

1. 之前不存在的不同的新的模式实体或
2. 该模式实体存在，但是被撤销或者被重新添加了。举个例子，旧模式实体的副本便是一个新的模式实体。

规则和匹配的模式目录，都是被激活的。如果是规则或模式实体，或者同时被改变了，激活将会被移除。一个激活的也可以通过命令或另一规则的行为被移除，该规则在移除激活的先决条件前被触发。

推理机通过特权值将激活进行分类。这种分类过程被称之为冲突消解(conflict resolution)，因为它消解了决定下一个触发规则的冲突。CLIPS 依照议程中最高的特权值进行规则的激活，并移除激活。这种执行被称之为触发，就像神经细胞的激活。当有适当的刺激时，神经细胞会激发出一定的电压脉冲，神经细胞激活后，将遭受折射(refraction)并在一定时期内不能被再次触发。如果没有折射，神经细胞将会在刺激作用下无休止的被激活下去。

如果没有折射效应，专家系统将会经常陷入到无关重要的循环当中去。因为，一旦规则被触发，那么它将在相同的事实作用下无休止的被触发下

去。在现实世界中，引起触发的刺激最终都会消失。举个例子，一只真的鸭子也许会游走或在电影里充当一个角色，然而，在计算机世界里，一旦数据被存储，它将一直保存在那儿，除非有外部声明移除或电脑断电。

下面的例子展示了一个规则的激活和触发。注意(watch)命令被用来更好的显示每个事实和激活。右箭号表明激活和事实正在被添加，左箭号表明已存在的事实和激活。

```
CLIPS>(clear)
CLIPS>(defrule duck
        (animal-is duck)
=>
        (assert (sound-is quack)))
CLIPS>(watch facts)
CLIPS>(watch activations)
CLIPS>(assert (animal-is duck))
==>f-1     (animal-is duck)
==>Activation 0     duck:f-1     ; 激活的默认权值为 0，其后是规则名：模式
<Fact-1>                          ; 实体索引
CLIPS>(assert (animal-is duck))   ; 注意复制的事实不会被输入
FALSE
CLIPS>(agenda)
    0       duck: f-1
For a total of 1 activation.
CLIPS>(run)
==>f-2     (sound-is quack)
CLIPS>(agenda)                    ; 当规则被触发后，议程为空
CLIPS>(facts)
```

f-0 (initial-fact)　　；即使事实已与规则匹配，折射也不会允许该激

f-1 (animal-is duck)　　；活，因为该规则等待事实的激活

f-2 (sound-is quack)

For a total of 3 facts.

CLIPS>(run)

CLIPS>

你也可以撤销事实然后又重新添加作为新的事实来让规则重复触发。

## • 查看规则

在你运行 CLIPS 时，也许你想查看某一条规则，这里有一个命令：ppdefrule---恰当的打印规则---打印一条规则。查看某条规则，则指定其规则名为 ppdefrule 的参数即可，举例如下：

```
CLIPS>(ppdefrule duck)
(defrule MAIN::duck
        (animal-is duck)
=>
        (assert (sound-is quack)))
CLIPS>
```

为了增加可读性，CLIPS 将规则的不同部分分布在不同的行中。规则箭号之前的模式部分仍然被称之为 LHS，箭号之后的行为部分仍然被称之为 RHS。术语 MAIN 引用 MAIN 模块表明该条规则是自定义的。你可以定义模块，将规则与那些可以被其他编程语言不同包装，模块，过程或函数纳入的声明类比。模块的使用使得编写那些有许多条规则的专家系统变得简单，这样，对于每个模块，它们大多在自己的议程中整合在一起了。如

果你想了解更多，请参考 CLIPS 参考指南。

如果你想打印一条规则，而你又忘掉了该规则的规则名，该怎么办？不用慌，你可以在 CLIPS 提示符后面使用 rules 命令来打印出所有的规则名，举例如下：

```
CLIPS>(rules)
Duck
For a total of 1 defrule.
CLIPS>
```

## 给我写信

规则的 RHS 部分除了添加一条新规则，你还可以使用 printout 函数打印出相应的信息。同样，CLIPS 有回车换行关键字：crlf，该关键字以换行格式来改进输出效果。有一点小改变就是，crlf 不被圆括弧包含。举例如下：

```
CLIPS>(defrule duck
        (animal-is duck)
=>
        (printout t “quack” crlf)) ; 一定要打出“t”
==>Activation 0      duck:f-1
CLIPS>(run)
quack
CLIPS>
```

双引号内的文本即为输出。一定记得在 printout 命令后输入“t”，这将告知 CLIPS 将结果输出到电脑的标准输出设备(standard output device)中。

通常，标准输出设备是你电脑的终端(terminal)(因此在 printout 后面接字母“t”)。然而，这可能会被重新定义，这样标准输出设备也可能是其他的设备，如调制解调器或磁盘。

## 其他特性

declare salience 命令提供对增添到议程中的规则的外部控制。在使用该特性的时候要注意不要太过于自由以免你的程序被人为控制太多。set-incremental-reset 命令禁止在规则被输入之前查看该规则的事实。获取增加的重置值命令为：get-incremental-reset。让一条规则重复触发的一个办法是使用 refresh 规则命令来强制使其重新激活。

load 命令载入前面你已经保存在磁盘中命名为“duck.clp”文件或者相应文件夹下的任何文件名里的规则。你还可以使用 load 命令载入一个包含规则的文本文件。

最快的载入文件的方法是，首先用 bsave 二进制存储命令将规则存储为机器可读二进制格式。载入的二进制命令为 bload。这样，CLIPS 内存会不加解释的快速读取这些二进制规则。

另外两个有用的命令可以帮助你通过一个文件来保存和载入事实。它们是 save-facts 和 load-facts。（save-facts）命令将会保存所有事实表中的事实，(load-facts)命令将会导入文件事实表中的事实。

batch 命令允许你像在顶层输入一样执行一个文件命令。另外一个有用的命令为你的操作系统提供一个界面。system 命令允许操作系统的执行和在 CLIPS 内的可执行。如果你想了解更多此类信息，请查阅 CLIPS 参考指南。

# 第三章 详细资料

问题不是大局，而是细节。

在前面的两章中，你已经学习了 CLIPS 的基础。现在，你将会学到怎样结合这些基础构建一个强大的程序。

## 红绿灯

到目前为止，你还只是看到一些仅包含一条规则的简单程序。然而，只包含一条规则的专家系统无疑作用有限。实际的专家系统通常包含上百，上千条规则。让我们来看看一个需要多条规则的应用软件程序吧。

假设你想写一个专家系统来决定一个移动式遥控装置如何对交通灯进行响应，最好是用多条规则去编写这个问题的类型。举个例子，红灯和绿灯情况下的规则按如下书写：

```
(defrule red-light
        (light red)
=>
        (printout t “Stop” crlf))

(defrule green-light
        (light green)
=>
        (printout t “Go” crlf))
```

当上述规则被输入到 CLIPS 后，增加一个 (light red) 事实并运行，你将会看到“Stop”被打印出来。再增加一个(light green)事实并运行，你会看到“Go”被打印出来。

## 行路

如果你考虑上面所述，交通灯不光只简单的包含有红灯，绿灯，应该还是黄灯存在，同时还有绿色的箭头标识来保护左转等。手型的交通灯亮与灭指示了行人的行与止。行与止的信号取决于我们装置显示是行人还是行车，这可能要关注一些不同的信号。

行人或行车的信息必须被添加，此外交通灯的状态信息也得添加。规则必须覆盖所有的情况，但是它们必须有多个模式。举个例子，假设我们希望当装置状态为行人和行人信号亮时，一个规则被触发，该规则应写成如下所示：

```
(defrule take-a-walk
        (status walking)
        (walk-sign walk)
=>
        (printout t “Go” crlf))
```

上面的规则中包含有两个模式，规则的每个模式必须在事实表中有相对应的事实满足才能触发。看看这些怎样工作，输入上面的规则并添加事实(status walking)和(walk-sign walk)，当执行(run)，规则的模式均被满足，程序输出“Go”。

你可以在一条规则中加入多条模式或行为。重要的一点是，只有当规则中所有的模式都被事实表中的事实满足时，规则才能被触发。这种约束类型被称为逻辑与条件元素（logical AND conditional element(CE)），是关于布尔型的“与”关系。AND 关系只有当所有的条件都为真时才为真。

因为模式为 AND 类型，如果只有一个模式被满足，规则不会被触发。只有给出规则 LHS 中所有的模式满足，规则才能被放入到议程中。

- **策略的问题**

策略(strategy)一词最初是一个军事术语，用在战争的谋划和行动中。现在，该术语普遍被用在商海（商海即是战场）中，适用于一个组织为了达到他们的目的所做的高级计划等。“比世界上其他的人卖出更多的多脂汉堡，赚更多的钱!”

在专家系统中，strategy 术语的一个用法是激活的冲突消解。那么你也许会说，“那好，我现在就将设计好我的专家系统，以便同一时刻仅有一条规则可能被激活，那么就用不上冲突消解了。”好消息是：如果你成功了，那么冲突消解确实无关紧要，坏消息是：你的成功证明了你的应用软件能被一个连续的程序很好的表达出来，那么你还不如首选在 C，Java 或者 Ada 中编写代码，犯不着去编写一个专家系统。

CLIPS 提供了七种不同的冲突消解策略：深度优先(depth)，广度优先(breadth)，LEX，MEA，complexity，simplicity 和随机(random)。在不考虑具体的应用软件程序时，很难说清哪一种策略更好。即使那样，判断上面的七种策略哪一个是“最好”的，也相当困难。如果你想了解更多关于这些策略的详细信息，请参考 CLIPS 参考指南。

深度优先策略(depth strategy)是 CLIPS 标准默认策略(default strategy)。当 CLIPS 第一次启动时，该默认设置便会被自动设置，后面，你可以更改默认设置。在深度优先策略中，在高权值的激活后，同权值或低权值之前，新的激活将会被放到议程中。这就是说议程中是从高权值到低权值进行排序的。

在本书中，所有的讨论和例子均是在假设为深度优先策略前提下的。

现在，你知道了所有的这些可选设置是多么的有用，一定得记住：当你运行一个由你和其他人共同编写的专家系统时，要保证你们的设置是一致的。否则，你会发现操作无效或者甚至是错误的。事实上，最好的办法是在你开发的过程中，对任何系统进行显式的设置编码，以保证正确配置。

## 自定义事实

当你使用 CLIPS 的时候，你也许会对在顶层中输入相同的声明事实而感到厌烦。如果你准备在程序运行的时候用到相同的声明，首先你可以用批处理文件加载磁盘里的声明，其次，你还可以使用自定义事实关键字：deffacts。举例如下：

```
CLIPS>(unwatch facts)
CLIPS>(unwatch activations)
CLIPS>(clear)
CLIPS>(deffacts walk  “Some facts about walking”
        (status walking)     ; 被声明的事实
        (walk-sign walk))   ; 被声明的事实
CLIPS>(reset)                ; 引入被自定义声明的事实
CLIPS>(facts)
f-0 (initial-fact)
f-1 (status walking)
f-2 (walk-sign walk)
For a total of 3 facts.
CLIPS>
```

自定义事实声明，必需指定一个事实名，如上面的 walk，跟在关键字 deffacts 的后面，事实名后面可以跟由双引号包含的注释。同规则中的注释一样，当 CLIPS 载入(deffacts)事实时，(deffacts)的注释将会被保留。事实名或注释后面便是将要被声明到事实表中的事实，自定义的事实由 CLIPS 的(reset)命令声明添加。

事实(initial-fact)由(reset)命令自动添加进来，并且它的事实标识符一直是 f-0。即使没有任何自定义的声明，(reset)命令也会自动声明事实(initial-fact)。在 CLIPS 的早期版本中，该事实被用来激活一些类型的规则，

但是现在它早已不作此目的使用了。它被用来对那些显式匹配于该事实的程序向后兼容。

(reset)命令较之(clear)命令的一个好处是，它不会丢弃所有的规则。(reset)命令使规则完整无缺，而(clear)命令将会移除所有议程中的规则，并移除所有事实表中的旧的事实。用(reset)命令是开始一个程序执行的首选方法，特别是之前程序已经在运行并且事实表已经被旧的事实打乱时。

总而言之，(reset)命令作用于事实有三点：

(1)将存在的事实从事实表中移除，同时也会移除议程中的激活规则。

(2)声明事实(initial-fact)

(3)声明已自定义(deffacts)声明的事实。

事实上，(reset)命令对于对象也有相似的作用。它可以删除实例，创建initial-object，声明添加自定义实例(definstances)。

## 选择性消除

undeffacts 命令的作用是通过消除内存中的自定义事实来撤销(deffacts)声明事实。举个例子：

```
CLIPS>(undeffacts walk)
CLIPS>(reset)
CLIPS>(facts)
f-0 (initial-fact)
For a total of 1 fact.
CLIPS>
```

这个例子演示了怎样将自定义的事实 walk 消除。如果执行了(undeffacts)后，想保存一个自定义事实声明，则必须重新定义。你甚至还可以使用(undeffacts)清除 initial-fact 事实。除了事实之外，CLIPS 还允许使用 undefrule

命令消除选定的规则。

## 注意

你可以对议程监视规则(watch rules)触发和监视激活(watch activations)。监视统计(watch statistics)给出已经触发规则数，执行时间，每秒规则数，事实的平均数，事实的最大数，激活的平均数和激活的最大数等信息。这些统计信息对于调整专家系统、优化运行速度非常有用。另一个命令叫:“watch compilations”,用来显示当规则被加载时的信息。watch all命令监视所有的项目。

使用 dribble 命令打印和查看信息到屏幕或磁盘，将会使你的程序稍微变慢，这是因为 CLIPS 需要花较多的时间去打印或保存信息到磁盘中去。dribble-on 命令会将所有的信息存储到被选入对话框的磁盘文件中，直到dribble-off 命令的输入才终止。这在提供任何事情发生时的参数记录是非常方便的。这两个命令如下：

```
(dribble-on <filename>)
(dribble-off <filename>)
```

另外一个有用的调试命令是(run)，该命令提供了一个触发规则数目的可选参数。举个例子，(run 21)命令将会告知 CLIPS 运行，并当 21 个规则触发后停止。(run 1)命令允许你每次只能执行一步程序。(step)命令等同于(run 1)。

像其它的编程语言一样，CLIPS 也提供断点(breakpoints)支持，断点作为 CLIPS 的一个简单指示符，停止顺序执行而优先执行指定规则。断点由set-break 命令设置。remove-break 命令将移除已经设置的断点。show-breaks命令显示所有设置断点的规则。带参数(rulename)的规则句法如下所示：

(set-break <fulename>)

(remove-break <rulename>)

(show-breaks)

## 合适的匹配

你可能会遭遇到这种情况：当你确定某条规则应该被激活却没有被激活。这也许是你的 CLIPS 中存在有漏洞，因为对于一个技术非常好的 CLIPS 的程序员来说，应该不可能是他们的问题（注意：为开发者做些商业宣传）（这里是反语，幽默）。

多数情况下，出现错误的原因是你书写规则的方式不对。为了给调试提供帮助，CLIPS 有一个被称为 matches 的命令，这个命令可以告诉你那些规则中的模式与事实可以匹配，哪些模式不能匹配而使规则不被激活。出现错误的一个普遍原因是，模式中的元素拼写错误导致与事实不匹配或增加的事实有拼写错误。

(matches)的参数为需要被检查匹配规则的规则名。让我们来看看(matches)起着怎样的作用，首先输入(clear)命令，然后输入下面的规则：

```
(defrule take-a-vacation
        (work done)                    ; 条件因素 1
        (money plenty)                 ; 条件因素 2
        (reservations made)        ; 条件因素 3
=>
    (printout t “Let’s go!!!” crlf))
```

下面将显示(matches)命令的用法，输入所示的命令，注意(watch facts)命令被开启，当你手动声明事实的时候，这是一个不错的方法，它可以提

供给你一次检查事实拼写的机会。

```
CLIPS>(watch facts)
CLIPS>(assert (work done))
==>f-1     (work done)
<Fact-1>
CLIPS>(matches take-a-vacation)
Matches for Pattern 1
f-1
Matches for Pattern 2
None
Matches for Pattern 3
None
Partial matches for CEs 1 - 2    ; CE 即条件元素
None
Partial matches for CEs 1 - 3
None
Activations
None
CLIPS>
```

通过(matches)命令，可以看到事实标识为 f-1 的事实与规则中的第一个模式或称之为条件因素可匹配。规则可能有 N 条模式，术语部分匹配(partial matches)是关于第 N 个模式与第一个事实匹配的所有设置，也就是说，部分匹配开始于规则的第一个模式，终止于任何一个模式，但不包含最后一个模式。当一个部分匹配不能成立时，CLIPS 将不会继续检查后面的匹配。举个例子，一个规则有四个模式，有可能第一个和第二个模式或第三个模

式都可能匹配成功，但，只有当所有的模式都匹配，这条规则才能被激活。

## 其他特性

这里有一些其他有用的关于自定义事的命令。举个例子，list-deffacts 命令将会列出当前 CLIPS 载入的所有自定义事实的事实名。另一个有用的命令是 ppdeffacts，它将所有存储的自定义事实信息打印出来。

| 函数 | 作用 |
| --- | --- |
| assert-string | 以字符串作为参数执行一个字符声明和作为一个无字符串事实的声明 |
| str-cat | 通过字符串(string concatenation)从单项目中构建一个单引号字符串 |
| str-index | 返回第一次出现子串的字符串索引(string index) |
| sub-string | 返回一个字符串的子字符串 |
| str-compare | 执行字符串比较(string compare) |
| str-length | 返回字符串的长度(string compare) |
| sym-cat | 返回连结符号 |

如果你想不用圆括号来输出多变量，最简单的方法就是用 string implode function，implode$。

# · 第四章 变量

**没改变更甚于改变。**

迄今为止，你已经了解了一些规则的类型，简单的阐述了规则的模式与事实匹配的一些内容。在本章中，你将会学到一些更有用的匹配和处理事实的方法。

## 认识变量

同其他编程语言一样，CLIPS 也通过变量(variables)来存储值。与事实不同的是，事实是静态的且不会改变，而变量的内容是随着其分配的值的改变而动态(dynamic)变化的。相比之下，一旦一个事实被声明，它的字段仅仅只能被撤销和重新声明一个该字段的事实而修改，甚至，这些事实的撤销和声明修改(将在本章后面的 deftemplate 中详细描述)是通过你所知道的修改事实索引执行的。

变量名，或者称之为变量标识符(variable identifier)，通常被写在一个问号的后面，即变量名。通用格式如下：

?<variable-name>

全局变量将在后面详细讲到，与上面的句法比较有些许不同。

如同其他的编程语言一样，变量名应该有一种好的命名方式，具有明确的含义。一些有效的变量名实例如下：

?x ?noun ?color

?sensor ?valve ?ducks-eaten

在一个变量能够被使用之前，它必须被分配一个值。下面是一个没有分配值的例子，尝试输入下面的代码，你将会看到 CLIPS 会响应一个错误消息：

```
CLIPS>(unwatch all)
CLIPS>(clear)
CLIPS>(defrule test
```

```
=>
        (printout t ?x crlf))
[PRCCPDE3] Undefined variable x referenced in RHS of defrule.

ERROR:
(defrule MAIN::test
        =>
        (printout t ?x crlf))
CLIPS>
```

当 CLIPS 不能找到?x 变量的约束值(value bound)时，便会抛出一个错误的提示。术语 bound 意味着对变量所分配的值。只有全局变量约束于所有的规则。其他所有的变量均约束于一条规则。在一条规则被触发前后，如果非全局变量没有被约束，那么当你尝试调用该变量时，CLIPS 就会给出一个错误提示。

## 果断点

一个变量的惯用方式是：在 LHS 中匹配一个值，随后在 RHS 中对该变量进行约束。举例如下：

```
(defrule make-quack
        (duck-sound ?sound)
=>
        (assert (sound-is ?sound)))
```

声明事实(duck-sound quack)，然后用(run)命令运行程序，检查规则，你将会发现这条规则产生了新的事实(sound-is quack)，这是因为，变量?sound

已经被约束到 quack 了。

当然，你可以多次使用一个变量。举例说明，输入下面的代码，别忘了输入(reset)命令和重新声明(duck-sound quack)。

```
(defrule make-quack
        (duck-sound ?sound)
=>
        (assert (sound-is ?sound ?sound)))
```

当该条规则被触发，将会产生事实(sound-is quack quack)，这样变量?variable 就被用到两次了。

## 鸭子说了什么

变量也通常被用在打印输出中，如下：

```
(derule make-quack
        (duck-sound ?sound)
=>
        (printout t “The duck said” ?sound crlf))
```

执行(reset)命令后，输入上面的规则，声明事实并运行(run)看看鸭子到底说了些啥？如果你修改这条规则，在输出中用双引号括住 quack，会有怎样的结果呢？

一个模式中可能有一个或多个变量，如下例所示：

```
CLIPS>(clear)
CLIPS>(defrule whodunit
```

```
        (duckshoot ?hunter ?who)
=>
        (printout t ?hunter  “shot”  ?who crlf))
CLIPS>(assert (duckshoot Brian duck))
<Fact-1>
CLIPS>(run)
Brian shot duck                    ; 今晚有鸭子吃了!
CLIPS>(assert (duckshoot duck Brian))
<Fact-2>
CLIPS>(run)
duck shot Brian                    ; 今晚吃 Brian!
CLIPS>(assert (duckshoot duck))    ; 丢失第三个字段
<Fact-3>
CLIPS>(run)
CLIPS>                             ; 规则不被触发，无输出
```

注意上面字段顺序的不同将会决定谁射杀谁。同时你也可以看到当事实(duckshoot duck)被声明后，规则并没有被触发。当事实中的字段不能与规则的第二个模式约束?who 匹配时，规则不能被激活。

## 快乐的单身汉

撤销在专家系统中非常有用，通常被用在 RHS 中要多于顶层中。在一条事实能被撤销之前，它必须被指定给 CLIPS。撤销一条规则中的事实，LHS 中事实地址(fact-address)首先必须被约束到一个变量。

绑定一个变量到事实的内容与绑定一个变量到事实地址有很大的不同。在上面的例子中，你已经看到了事实(duck-sound ?sound)，字段的值被约束了一个变量。因此，?sound 被约束到 quack。然而，当你想移除包含

(duck-sound quack)的事实时，你必须首先告知 CLIPS 被撤销事实的地址。

事实地址指定使用左箭号(left arrow)：“<-”。输入该符号，只要键入一个“<”符号，然后紧跟一个“-”即可。从一个规则中撤销事实的例子如下：

```
CLIPS>(clear)
CLIPS>(assert (bachelor Dopey))
<Fact-1>
CLIPS>(facts)
f-0 (initial-fact)
f-1 (bachelor Dopey)
For a total of 2 facts.
CLIPS>(defrule get-married
        ?duck <- (bachelor Dopey)
=>
        (printout t “Dopey is now happily married” ?duck crlf)
        (retract ?duck))
CLIPS>(run)
Dopey is now happily married <Fact-1>
CLIPS>(facts)
f-0 (initial-fact)
For a total of 1 fact.
CLIPS>
```

注意到，左箭号将事实的地址约束到?duck，因此，(printout)打印出?duck 的事实索引。同样的，事实(bachelor Dopey)也已被撤销。

变量也能被用来拾取事实的值同时作为地址。如下例所示，为了简便，用到自定义事实(deffact)。

```
CLIPS>(clear)
CLIPS>(defrule marriage
        ?duck <- (bachelor ?name)
=>
        (printout t ?name  “is now happily married”  crlf)
        (retract ?duck))
CLIPS>(deffacts good-prospects
        (bachelor Dopey)
        (bachelor Dorky)
        (bachelor Dicky))
CLIPS>(reset)
CLIPS>(run)
Dicky is now happily married
Dorky is now happily married
Dopey is now happily married
CLIPS>
```

注意上面的所有的事实均与模式(bachelor ?name)匹配，规则被触发。CLIPS 还有一个名为事实索引(fact-index)的函数，该函数用来返回事实地址的事实索引。

## · 通配符

代替绑定一个变量到一个字段值，一个非空字段的存在能被检测到单独使用通配符(wildcard)。举个例子，假设你正在经营一个鸭子约会服务部，一只母鸭声明它只与名字为 Richard 的公鸭约会。事实上，关于这个声明有两个标准，因为它的隐含意义是鸭子必须有不止一个的名字，因此这样一

个简单的事实声明：(bachelor Richard)是不充足的，因为在该事实中仅有一个名字。

部分事实被指定的情形，是非常普遍和重要的。为了解决这个问题，可以利用通配符来触发 Richard 们。

最简单的通配符格式被称之为单字段通配符(single-field wildcard)，以一个问号“?”来表示。问号也被称为单字段约束(single-field constraint)。一个单字段通配符仅代表一个字段，如下所示：

```
CLIPS>(clear)
CLIPS>(defrule dating-ducks
        (bachelor Dopey ?)
=>
        (printout t “Date Dopey” crlf))
CLIPS>(deffacts duck
        (bachelor Dicky)
        (bachelor Dopey)
        (bachelor Dopey Mallard)
        (bachelor Dinky Dopey)
        (bachelor Dopey Dinky Mallard))
CLIPS>(reset)
CLIPS>(run)
Date Dopey
CLIPS>
```

模式中包含有一个通配符，指明 Dopey 的姓氏并不重要，只要名字是 Dopey，规则就会被触发。因为模式包含三个字段，其中之一是一个单字段通配符，所以只能是有且仅有三个字段的事实才能满足，只有 Dopey 的有

且仅有两个字的鸭子才能符合这只母鸭的要求。

假设你想指定名字有且仅有三个字的 Dopey，那么你应该按照如下格式书写模式：

```
(bachelor Dopey ? ?)
```

或者，只要是中间名为 Dopey 有三个字的都可满足：

```
(bachelor ? Dopey ?)
```

或者，只要是姓 Dopey 有三个字的都可满足：

```
(bachelor ? ? Dopey)
```

另一个可能出现有趣的事情是，如果 Dopey 必须是名字的第一个字，但那些 Dopey 们仅只能接受两个或三个字的名字。一个解决此问题的方法是写两个规则，如下所示：

```
(defrule eligible
        (bachelor Dopey ?)
=>
        (printout t “Date Dopey” crlf))

(defrule eligible-three-names
        (bachelor Dopey ? ?)
=>
        (printout t “Date Dopey” crlf))
```

输入并运行，你将看到那些包含 Dopey 有两个或三个字的名字被打印出来。当然，如果你想匿名约会日期，那么你需要将 Dopey 名绑定一个变量并打印出来。

## 继续讲通配

将规则分开书写而不处理每个字段，用多字段通配符(multifield wildcard)将会更容易。多字段通配符的符号是在问号前面加上一个美元符号，为“$?”，该符号指代零个或多个字段。注意与指代一个且仅为一个的单字段通配符的区别。

上面分开而写的两个规则，此时便可以写成一个了，如下所示：

```
CLIPS>(clear)
CLIPS>(defrule dating-ducks
        (bachelor Dopey $?)
=>
        (printout t “Date Dopey” crlf))
CLIPS>(deffacts duck
        (bachelor Dicky)
        (bachelor Dopey)
        (bachelor Dopey Mallard)
        (bachelor Dinky Dopey)
        (bachelor Dopey Dinky Mallard))
CLIPS>(reset)
CLIPS>(run)
Date Dopey
Date Dopey
```

```
Date Dopey
CLIPS>
```

通配符的另外一个作用是，它可附属于一个符号字段来创建一个变量，如?x，$?x，?name 或者$?name。依照 LHS 中“?”或 “$?”的使用，变量可以是单字段变量或多字段变量。注意在 RHS 中，只能用?x，这里的 x 可以是任意名。你可以将“$”理解成一个函数，函数的参数是一个单字段通配符或者一个单字段变量，分别返回多字段通配符或多字段变量。

作为一个多字段变量的例子，因为一个变量与匹配的字段名等同，下面的规则同样打印出匹配的事实字段名：

```
CLIPS>(defrule dating-ducks
        (bachelor Dopey $?name)
=>
        (printout t “Date Dopey” ?name crlf))
CLIPS>(reset)
CLIPS>(run)
Date Dopey (Dinky Mallard)
Date Dopey (Mallard)
Date Dopey ()
CLIPS>
```

如你所见，在 LHS 中，多模式是$?name，而在 RHS 中，只能用?name。输入并运行，你将看到所有有资格入选的 Dopey 们的名字。多字段通配符照顾到所有字段的个数，同样，注意多字段变量返回时包含在圆括号里面。

假设你想匹配所有的只要是名字中包含 Dopey 的鸭子，比一定非得是它们的姓氏。下面的例子将会匹配所有包含 Dopey 的事实并打印出它们的

名字：

```
CLIPS>(defrule dating-ducks
        (bachelor $?first Dopey $?last)
=>
        (printout t “Date” ?first “Dopey” ?last crlf))
CLIPS>(reset)
CLIPS(run)
Date () Dopey (Dinky Mallard)
Date (Dinky) Dopey ()
Date () Dopey (Mallard)
Date () Dopey ()
CLIPS>
```

这里的模式将与所有的只要是包含 Dopey 的事实匹配。

单或多字段通配符也可以联合使用，举个例子，模式：

```
(bachelor ? $? Dopey ?)
```

意味着姓氏和名字的最后一个字可以是任意的，但是最后一个字之前一定是 Dopey。同时，这个模式也要求与之匹配的事实至少包含有四个字段，因为 $?可以匹配零个或多个字段，其他的必须都匹配一个字段。

尽管多字段在许多情况下的模式匹配中必不可少，但是它们的滥用会带来许多的副作用，因为它们增加了系统的内存消耗，使执行速度变慢。

- 作为一种普通的规则类型，你可以仅在你不知道字段长度的情况下使用$?，不要将$?简单的用来方便输入。

- 模式中变量的使用有一个非常重要和有用的属性，表述如下：
  - 变量在被首次绑定时，仅在规则内保留其值，包含 LHS 和 RHS，除非在 RHS 中被改变了。

举个例子，在下面的规则中：

```
(defrule bound
        (number-1 ?num)
        (number-2 ?num)
=>)
```

如果有下面事实：

```
f-1 (number-1 0)
f-2 (number-2 0)
f-3 (number-1 1)
f-4 (number-2 1)
```

那么，这条规则这能被 f-1，f-2 的事实对和 f-3，f-4 的事实对激活。f-1 不能与 f-4 同时进行匹配，这是因为在模式中，?num 已经绑定为一个值，那么事实中的代替字段只能是相同的值。同理，当第一个模式中的?num 绑定值到 1 的时候，第二个模式中?num 的值也必须为 1。注意这里的规则将会被激活两次。

作为一个实际的例子，输入下面的规则。注意相同的变量?name 被同时用在模式中。在(reset)和(run)程序之前，输入(watch all)命令，这样你便可以清楚看到执行了什么。

```
CLIPS>(clear)
CLIPS>(defrule ideal-duck-bachelor
        (bill big ?name)
        (feet wide ?name)
=>
        (printout t “The ideal duck is ”?name crlf))
CLIPS>(deffacts duck-assets
        (bill big Dopey)
        (bill big Dorky)
        (bill litter Dicky)
        (feet wide Dopey)
        (feet narrow Dorky)
        (feet narrow Dicky))
CLIPS>(watch facts)
CLIPS>(watch activations)
CLIPS>(reset)
<==f-0 (initial-fact)
==>f-0 (initial-fact)
==>f-1 (bill big Dopey)
==>f-2 (bill big Dorky)
==>f-3 (bill litter Dicky)
==>f-4 (feet wide Dopey)
==>Activation 0      ideal-duck-bachelor: f-1,f-4
==>f-5 (feet narrow Dorky)
==>f-6 (feet narrow Dicky)
CLIPS>(run)
The ideal duck is Dopey
```

```
CLIPS>
```

当程序运行时，第一个模式与 Dopey 和 Dorky 匹配因为他们都非常富有(big bills)，变量?name 被分别绑定到他们的名字，当 CLIPS 尝试匹配第二个模式的时候，只有?name 绑定到 Dopey 的能满足模式(feet wide)。

## 幸运的鸭子

在生活中有许多情况出现，以系统化的方式行事是非常明智的。如果你的期望方式不能解决问题的时候不妨试试系统化（如去一次次的结婚来找到最完美的配偶这样的普通算法）。

一个被总结起来的办法是保存名单（注意：如果你非常想给人留下深刻的印象，那么给他们一份你的清单）。在我们的例子中，我们将保存一份单身鸭子的清单，将最想找到婚姻的放在最前面。一旦一个理想的单身汉被鉴定符合，我们便将他作为一只幸运鸭升至表的前列。

下面的程序显示了通过增加两个规则到 ideal-duck-bachelor 规则，执行该过程：

```
(defrule ideal-duck-bachelor
        (bill big ?name)
        (feet wide ?name)
=>
        (printout t “The ideal duck is ”?name crlf)
        (assert (move-to-front ?name)))

(defrule move-to-front
        ?move-to-front <- (move-to-front ?who)
        ?old-list <- (list $?front ?who $?rear)
```

```
=>
        (retract ?move-to-front ?old-list)
        (assert (list ?who ?front ?rear))
        (assert (change-list yes)))

        (defrule print-list
            ?change-list <- (change-list yes)
            (list $?list)
=>
        (retract ?change-list)
        (printout t “List is: ”?list crlf))

(deffacts duck-bachelor-list
        (list Dorky Dinky Dicky))

(deffacts duck-assets
        (bill big Dicky)
        (bill big Dorky)
        (bill litter Dinky)
        (feet wide Dicky)
        (feet narrow Dorky)
        (feet narrow Dinky))
```

最初的表在 duck-bachelor-list 自定义事实中给出，当程序运行后，将会提供一个新的最有可能候选的表。

```
CLIPS>(unwatch all)
```

```
CLIPS>(reset)
CLIPS>(run)
The ideal duck is Dicky
List is :(Dicky Dorky Dinky)
CLIPS>
```

注意 move-to-front 规则中的声明(change-list yes)，没有这条声明，print-list 规则将会总是触发在初始事实。该声明是一个关于用控制事实(control fact)来控制另一个规则激活的例子，你应该仔细的学习这个例子并弄清为什么使用它。另一个控制方法是模块，将会在 CLIPS 参考指南中详细讨论。

move-to-front 规则移除旧的名单，增加新的名单。如果旧的名单没有被移除的话，那么 print-list 规则能激活两个规则到议程中，但是只有一个被触发。只有一个被触发的原因是 print-list 规则移除了控制事实调用同一规则的其他激活。你不会提前预知哪一个将会被触发，所以在打印的时候新的表单也许会被旧的替代了。

# · 第五章 格式

Style today ,gone tomorrow

本章中，你将学习关键词 deftemplate 的用法，deftemplate 代表定义模板(define template)的意思。这个关键词能帮助你写出具有明确定义模式的规则。

## “精彩”先生

自定义模板(deftemplate)类似于 C 语言中的结构定义。deftemplate 定义模式中一组相关的字段，这类似于在 C 语言中用结构来定义一组相关数据。

自定义模板是由一些被命名为 slot 的字段构成的表。自定义模板允许通过字段名而不一定由指定的字段顺序来进行存取。在专家系统程序中，自定义模板有助于编写好的格式，同时它在软件工程中也是非常有用的。

slot 被称之为单槽(single-slot)或多槽(multislot)。单槽包含且仅包含一个字段，而多槽则可以包含零个或多个字段。自定义模板中可以使用任意数目的单槽和多槽。写一个槽的时候，先写字段名(attribute)后面紧跟着字段值。注意，一个只有一个槽值的多槽并没有单槽那样的严格。类比想像一下，将多槽看成是一个碗橱，里面也许有许多的碗碟。碗橱里只有一个碟子却并不等同于就一个单独的碟子(单槽)。然而，单槽的槽值（或变量）也可以与只有一个字段的多槽（或多槽变量）相匹配。

下面是一个有关自定义模板的例子，通过考察每个鸭子的属性来判定哪个最有可能是母鸭的最佳婚姻对象。

| **Attributes** | **Value** |
|---|---|
| name | “Dopey Wonderful” |
| assets | rich |
| age | 99 |

prospect 关系的自定义模板可以被写成如下形式，空格和注释用来增加程序的可读性。

```
(deftemplate prospect        ; 自定义模板关系名
      “vital information”    ; 可选注释
     (slot name              ; 字段名
      (type STRING)          ; 字段类型
      (default ?DERIVE))     ; 字段“名字”的默认值
    (slot assets
      (type SYMBOL)
      (default rich))
```

```
    (slot age
     (type NUMBER)              ; NUMBER 类型可以是整型 INTEGER 或浮
点型 FLOAT
     (default 80)))
```

在本例中，自定义的构成如下：

- 一个自定义模板关系名
- 字段属性
- 字段类型，可以是一下任意一种允许的类型：SYMBOL，STRING，NUMBER 或其他。
- 字段的默认值

该部分自定义模板包含有三个单槽，分别为 name, asset 和 age。

如果没有外部声明值被定义，则当(reset)执行后，CLIPS 会插入自定义模板的默认值(deftemplate default values)。举个例子，在(clear)命令后输入 prospect 自定义模板结构，声明如下：

```
CLIPS>(assert (prospect))
<Fact-1>
CLIPS>(facts)
f-0 (initial-fact)
f-1 (prospect (name  “ ”)(assets rich)(age 80))
For a total of 2 facts.
CLIPS>
```

如你所见，CLIPS 已经为 name 字段插入了 string 类型默认的值——“ ”。同样的，CLIPS 也插入另两个字段的默认值。不同的类型有不同的默认符号，如对字符串 STRINGl 来说，空字符串为“ ”，对 INTEGER 为整数 0，对 FLOAT

为浮点值 0.0 等等。关键字?DERIVE 自动为槽值的类型选择合适的默认值，如对于字符串 STRING 类型为空字符串“”。

你可以显式声明字段的值，如下例所示：

```
CLIPS>(assert (prospect (age 99)(name “Dopey”)))
<Fact-2>
CLIPS>(facts)
f-0 (initial-fact)
f-1 (prospect (name “”)(assets rich)(age 80))
f-2 (prospect (name “Dopey”)(assets rich)(age 99))
For a total of 3 facts.
CLIPS>
```

注意上面的字段输入顺序是无关紧要的，虽然它们都是命名字段。

在自定义模板中，最重要的是要认识到 NUMBER 不是像字符，字符串，整型和浮点型那些原始的字段类型。NUMBER 是即可整型又可浮点的混合类型，这对于那些在编程中并不需要在乎哪种数字类型存储是非常有用的。NUMBER 的两种类型之一指定类型如下所示：

```
(slot age
    (type INTEGER FLOAT)
    (default 80)))
```

## 再见

通常，一个有 N 个槽的自定义模板的一般结构如下所示：

```
(deftemplate <name>
```

(slot-1)

(slot-2)

...

(slot-N))

在一个自定义模板中，属性值一般被指定精确的值，而不是简单的如 80 或 rich。举个例子，在这个自定义模板中，一个值类型被指定。

字段值可以被显式声明，也可以给出一个值的范围。allowed-values 可以是任意的原始类型，如字符(SYMBOL)，字符串(STRING)，整型(INTEGER)，浮点型(FLOAT)等。举例如下：

| Deftemplate Enumerated Values | Example |
|---|---|
| allowed-symbols | rich filthy-rich loaded |
| allowed-strings | “Dopey”“Dorky”“Dicky” |
| allowed-numbers | 1 2 3 4.5 -2.001 1.3e-4 |
| allowed-integers | -100 53 |
| allowed-floats | -2.3 1.0 300.00056 |
| allowed-values | “Dopey” rich 99 1.e9 |

对于同一个自定义模板字段，同时指定其数字范围和允许值是行不通的。举个例子，如果你指定(allowed-integer 1 4 8)，这与数值范围 1 到 10 的(range 1 10)是矛盾的。如果一些数字碰巧是连续的，如 1，2，3，那么你可以指定一个范围(range 1 3)可以精确匹配。然而，这个范围对于 allowed-integers 来说是多余的了。因此，范围和允许值是互斥的，当你指定了一个范围时就不能指定允许值，反之亦然。通常，范围属性不能被用来与 allowed-values，allowed-numbers，allowed-integer 或 allowed-floats 连接。

去掉可选信息，一个规则使用到自定义模板如下所示：

```
CLIPS>(clear)
CLIPS>
(deftemplate prospect          ; 自定义模板名
        (slot name                    ; 字段名
            (default ?DERIVE))     ; 字段 name 的默认值
    (slot assets
        (default rich))
    (slot age
        (default 80)))
CLIPS>
(defrule matrimonial_candidate
    (prospect (name ?name)(asset ?net_worth)(age ?months))
=>
    (printout t “Prospect: ”?name crlf
                                    ?net_worth crlf)
                                    ?months “months old” crlf))
CLIPS>(assert (prospect (name “Dopey Wonderful”)(age 99)))
<Fact-1>
CLIPS>(run)
Prospect: Dopey Wonderful
rich
99 months old
CLIPS>
```

- 注意在声明事实的命令中，并没有指定 rich 的值，但是，rich 的默认值还是被用在 Dopey 上了。

如果 assets 字段被指定值为 poor，那么，指定值 poor 会重载 assets 的默认值 rich。如下是一个关于 Dopey 那吝啬的侄子的例子：

```
CLIPS>(reset)
CLIPS>(assert (prospect (name “Dopey Notwonderful”)
(assets poor)(age 95)))
<Fact-1>
CLIPS>(run)
Prospect: “Dopey Notwonderful”
Poor
95 months old
CLIPS>
```

自定义模板的模式可以像任意普通模式一样使用。举个例子，下面的规则用来消除不受欢迎的对象。

```
CLIPS>(undefrule matrimonial_candidate)
CLIPS>(defrule bye-bye
        ?bad-prospect <- (prospect (assets poor)(name ?name))
=>
        (retract ?bad-prospect)
        (printout t “bye-bye” ?name crlf))
CLIPS>(reset)
CLIPS>(assert (prospect (name “Dopey Wonderful”)(assets rich)))
<Fact-1>
CLIPS>(assert (prospect (name “Dopey Notwonderful”)(assets poor)))
<Fact-2>
```

```
CLIPS>(run)
bye-bye Dopey Notwonderful
CLIPS>
```

## 多槽的使用

迄今为止我们还只是在模式中应用过单字段，上面的 name，assets 和 age 字段值均是单值。在许多类型的规则中，你也许会要用到多字段。自定义模板允许在一个多槽中使用到多槽值。

作为一个多槽的例子，假设你想将关系 prospect 的 name 写成多字段，这将在处理 prospects 的 name 部分匹配上有很大的灵活性。下面是采用了多槽和改进多字段部分匹配的自定义模板的定义。注意其中的多槽模式 $?name，现在被用来与所有组成 name 的字段匹配。为了方便起见，同时给出一个自定义事实(deffacts)。

```
CLIPS>(clear)
CLIPS>(deftemplate prospect)
    (multislot name
          (type SYMBOL)
          (default ?DERIVE))
    (slot assets
          (type    SYMBOL)
          (allowed-symbols poor rich wealthy loaded)
          (default rich))
    (slot age
          (type INTEGER)
          (range 80 ?VARIABLE)      ; 越老越好
          (default 80)))
```

```
CLIPS>(defrule happy_relationship
        (prospect (name $?name)(assets ?net_worth)(age ?months))
=>
        (printout t “Prospect: ”?name crlf
                                    ?net_worth crlf
                                    ?months” moths old” crlf))
CLIPS>(deffacts duck-bachelor
(prospect (name Dopey Wonderful)(assets rich)(age 99)))
CLIPS>(reset)
CLIPS>(run)
Prospect: (Dopey Wonderful)
rich
99 months old
CLIPS>
```

在输出中，Dopey 的名字被包含在圆括号里，这表明了这是一个多槽值。如果你将多槽与单槽做比较，你会发现双引号不见了。在多槽中，name 槽并不是一个字符串，CLIPS 将 name 做为两个单独的字段 Dopey 和 Wonderful 来看。

## 修改字段槽值

自定义模板大大的方便了对模式中指定字段的访问，因为期望字段能被它的槽名所标定。可以利用修改(modify)行为修改指定的自定义模板中的槽来撤销并增加一个新事实。

作为一个例子，看看当单身鸭子 Dopey Wonderful 失去了它所有的鱼，从 Donald 鸭那里为它的母鸭买些香蕉什么的，下面的规则能执行什么。

```
CLIPS>(undefrule *)
CLIPS>
(defrule make-bad-buys
        ?prospect <- (prospect (name $?name)
                                (assets rich)
                                (age ?months))
=>
        (printout t “Prospect: ”?name crlf
                                “rich” crlf)
                                ?months “ months old” crlf crlf)
    (modify ?prospect (assets poor)))
CLIPS>
(defrule poor-prospect
        ?prospect <- (prospect (name $?name)
                                (assets poor)
                                (age ?months))
=>
        (printout t “Ex-prospect: ”?name crlf
                                Poor crlf
                                ?months “ months old” crlf crlf))
CLIPS>(deffacts duck-bachelor
(prospect (name Dopey Wonderful)(asset rich)(age 99)))
CLIPS>(reset)
CLIPS>(run)
Prospect: (Dopey Wonderful)
rich
99 months old
```

```
Ex-prospect: (Dopey Wonderful)
poor
99 months old

CLIPS>
```

如果你输入(facts)命令，你会发现先前的事实 f-1 被(modify)撤销了，取而代之的是一个新的事实 f-2。

```
CLIPS>(facts)
f-0 (initial-fact)
f-2 (prospect (name Dopey Wonderful)(assets poor)(age 99))
For a total of 2 facts.
CLIPS>
```

make-bad-buys 事实被 prospect 中指定的 assets 槽的槽值 rich 激活，该条规则在 modify 作用下将 assets 的槽值改变为 poor。注意 assets 槽可以通过槽名访问到。如果没有自定义模板，那将必须列出所有的单变量的字段或使用到通配符，那样的话，效率将会降低。poor-prospect 规则的目的是简化打印出 poor prospect，且证明了 make-bad-investments 规则确实改变了 assets 槽。

# · 第六章 功能

**功能性是格式的对立面。**

本章中，你将会学到一些关于模式匹配非常有用的函数和一些非常有

用的多字段变量。同时，你也将学到怎样进行数字计算。

## ～约束

让我们重新考虑设计一个帮助机器人穿越大街的专家系统的问题，你可能已写好的一个规则如下：

```
(defrule green-light
        (light green)
=>
        (printout t “Walk” crlf))
```

另一个可能涉及红灯情形的规则为：

```
(defrule red-light
        (light red)
=>
        (printout t “Don’t walk” crlf))
```

第三个规则可能涉及当一个 walk-sign 显示不能行走的规则，这比绿灯要优先。

```
(defrule walk-sign
        (walk-sign-says dont-walk)
=>
        (printout t “Don’t walk” crlf))
```

前面所给的规则都很简单，且没有涵括到所有的情形，比如交通灯故障。

举个例子，当红灯或黄灯和行走信号显示为 walk 的时候，机器人该怎么做？

一个处理该情形的办法是通过使用字段约束(field constraint)来限制 LHS 模式中的值。字段约束的作用就像是对模式的约束。

一种字段约束被称之为连接约束(connective constraint)。连接约束有三种类型，第一种被称之为～约束(～ constraint)，带有一个波浪字符“～”。波浪字符后面跟一值，表示不允许为该值。

以一个简单的例子来介绍～约束，假设你想写一个规则，当交通灯不为绿灯的时候便打印出“Don’t walk”。一种方式是对每种交通灯情况写不同的规则，包括所有可能的情况：黄灯，红灯，闪烁的黄灯，闪烁的红灯，闪烁的绿灯，瞬间黄灯，闪烁黄灯和瞬间红灯等等。然而，一个很简单的方法是通过使用～约束写如下规则：

```
(defrule walk
        (light ～green)
=>
        (printout t “Don’t walk” crlf))
```

通过使用～约束，该条规则可以适用于多条要求制定不同交通灯情况。

## |约束

第二种连接约束是竖线约束(bar constraint)，“|”约束用来表示允许一组可以匹配的变量。

举个例子，假设你想写一条规则，当是黄灯或黄灯闪烁的时候，打印出“Be cautious”。下面的例子显示了竖线约束所起的作用。

```
CLIPS>(clear)
    CLIPS>(defrule cautious
```

```
        (light yellow|blinking-yellow)
=>
        (printout t “Be cautious” crlf))
CLIPS>
(assert (light yellow))
<Fact-1>
CLIPS>(assert (light blinking-yellow))
<Fact-2>
CLIPS>(agenda)
    0       cautious: f-2
    0       cautious: f-1
For a total of 2 activations.
CLIPS>
```

## &约束

第三种连接约束是与约束(& connective constraint)，带有一个&符号。与约束使连接约束同时匹配，你将在下面的例子中看到。与约束通常与其他的连接约束一同使用，除此之外并不常用。举个例子，假设你想写一个规则，该规则会被黄灯或闪烁黄灯的事实触发，这非常简单，用前面所学到的竖线约束就可以解决。但是这里假设你还需要分清灯的颜色。

解决的方法是对颜色 color 绑定一个变量，通过使用&匹配打印出变量，这里“&”是非常有用的，如下所示：

```
(defrule cautious
        (light ?color&yellow|blinking-yellow)
=>
        (printout t “Be cautious because light is ”?color crlf))
```

?color 变量将会被限制在可与匹配的字段 yellow 或 blinking-yellow。

“&”与“～”搭配使用也非常有用。举个例子，假设你想写一个规则，当交通灯不是黄色和红色时被触发，如下所示：

```
(defrule not-yellow-red
        (light ?color&～red&～yellow)
=>
        (printout t “Go, since light is ”?color crlf))
```

## · 初级数字运算

除了处理符号事实，CLIPS 还可以执行数字计算。然而，你始终要明白专家系统语言如 CLIPS 并不是设计用来做些数字运算的。虽然 CLIPS 的数学函数功能非常强大，它们也只是用来对应用程序中进行一些数字修改而已。其他的语言如 FORTRAN 有更好的数字运算能力，因为它很少或没有符号推理。在一些应用程序中，你将发现 CLIPS 的计算能力非常有用。

CLIPS 提供最基本的算术和数学函数，+，-，*，/，div，max，min，abs，float 和 integer。了解更多的详细情况，可以参考 CLIPS 参考指南。

CLIPS 数学表达式的表示是从 LISP 的风格而来的。一个通常情况下的数学表达式写成 2+3，在 LISP 和 CLIPS 中则必须被写成前缀形式(prefix form)：（+ 2 3）。在 CLIPS 的前缀形式中，运算符号在参数之前，数学表达式被用圆括号括住。通常的数学表达式被称为插入形式(infix form)，因为数学运算符号在参数(arguments)之间。

数学运算符号能被用在 LHS 和 RHS 中。举个例子，下面演示的是一个加法算术运算的例子，在规则的 RHS 中，通过加法运算题声明一个事实，该事实中包含数字?x 和?y 的和。注意例子中的注释是用插入式表示的。

```
CLIPS>(clear)
CLIPS>(defrule addition
        (numbers ?x ?y)
=>
        (assert (answer-plus (+ ?x ?y))))      ; ?x+?y 加法
CLIPS>(assert (number 2 3))
<Fact-1>
CLIPS>(run)
CLIPS>(facts)
f-0 (initial-fact)
f-1 (number 2 3)
f-2 (answer-plus 5)
Fpra a total of 3 facts.
CLIPS>
```

算术运算符可以用在 LHS 中，等于号(equal sign)---“=”用来告诉 CLIPS 计算后面的数学表达式，而不是用它来进行字面上的模式匹配。下面的例子用来显示 LHS 中三角形斜边的计算，并以一些直角边数字对进行模式匹配。求幂(exponentiatin)符号“**”用来对 x 和 y 的值进行乘方运算。乘方符号的第一个参数是底数，第二个参数为指数。

```
CLIPS>(clear)
CLIPS>(deffacts database
        (stock A 2.0)
        (stock B 5.0)
        (stock C 7.0))
CLIPS>(defrule addition
```

```
        (numbers ?x ?y)
        (stock ?ID =(sqrt (+ (** ?x 2)(** ?y 2))))
=>
        (printout t “Stock ID=“ ?ID” crlf))
CLIPS>(reset)
CLIPS>(assert (number 3 4))
<Fact-4>
CLIPS>(run)
Stock ID=B          ; Stock ID 匹配乘方计算
CLIPS>
```

## 扩展的参数

在许多数学运算符上，数学表达式中的参数可以扩展到两个以上。相同顺序的算术运算对多个参数被执行。下面的例子阐述了三个参数的使用方法，计算过程从左至右。在输入这些之前，你得使用(clear)命令清楚所有旧的事实和规则。

```
(defrule addition
        (numbers ?x ?y ?z)
=>
        (assert (answer-plus (+ ?x ?y ?z))))       ; ?x+?y+?z
```

输入上面的程序并声明一个事实(numbers 2 3 4)，运行后，你将看到如下的事实。注意，事实索引在载入程序前，会因你事先输入的(clear)或(reset)命令的不同而有所不同。

```
CLIPS>(facts)
```

f-0 (initial-fact)

f-1 (numbers 2 3 4)

f-2 (answer-plus 9)

For a total of 3 facts.

CLIPS>

多参数 CLIPS 表达式的插入等量可以被表示为：

arg [function arg]

这里的中括号意味可能是多条。

除了基本的数学运算外，CLIPS 还有扩展的数学函数(Extended Math function)功能，包括 trig，hyperbolic 等等。完整的列表请参看 CLIPS 参考指南。这些被称之为扩展的数学函数，因为通常认为基本的数学运算为“+”，“-”等。

## 混合结果的形式

在表达式的处理中，CLIPS 试图保持相同模式的参数。举个例子：

CLIPS>(+ 2 2)　；两个整型参数的运算结果还是整型

4

CLIPS>(+ 2.0 2.0)　；两个浮点参数的运算结果还是浮点型

4.0

CLIPS>(+ 2 2.0)　　；混合型参数输出浮点型结果

4.0

注意最后一个情形是混合的参数，CLIPS 自动将结果转变为双精度浮点型。

你可以显式的转变结果的类型，通过使用 float 和 integer 运算符，如下所示：

```
CLIPS>(float (+ 2 2))   ; 显式转换整型到浮点型
4.0

CLIPS>(integer (+ 2.0 2.0))   ; 显式转换浮点到整型
4
```

圆括号用来指定表达式的运算顺序。在?x+?y*?z 的例子中，通常的计算顺序是先计算?y*?z，然后再与?x 相加。然而，在 CLIPS 中，如果你想按照此顺序计算的话，那么必须显式的使用圆括号，如下：

```
(defrule mixed-cals
        (numbers ?x ?y ?z)
=>
        (assert (answer (+ ?x (* ?y ?z)))))
```

在这条规则中，最内一层的圆括号里的运算最先执行，所以先执行?y*?z，然后再与?x 相加。

- **约束变量**

由模式匹配在 LHS 中分配一个值给变量类似于通过绑定函数(bind function)在 RHS 中绑定(binding)一个值到变量。如果同一个变量被重复的使用到，那么在 RHS 中绑定其变量值将非常方便。

以一个简单的数学计算为例，让我们首先将答案绑定到一个变量，并随后打印约束变量(bound variable)。

```
CLIPS>(clear)
CLIPS>(defrule addition
        (numbers ?x ?y)
=>
        (assert (answer (+ ?x ?y)))
        (bind ?answer (+ ?x ?y))
        (printout t “answer is ”?answer crlf))
CLIPS>
(assert (numbers 2 2))
<Fact-1>
CLIPS>(run)
answer is 4
CLIPS>(facts)
f-0 (initial-fact)
f-1 (numbers 2 2)
f-2 (answer 4)
For a total of 3 facts.
CLIPS>
```

(bind)同样可以被用在 RHS 中，用来绑定单或多字段值到一个变量。(bind)被用来绑定零个，一个或多个值到一个变量，而不带“$”运算符。调用 LHS 中的变量，你可以在一个字段中，使用“$”运算符创建多字段模式，如“$?x”。然而，在 RHS 中，“$”运算符是不需要的，因为(bind)的参数显式的告知了 CLIPS 它绑定值的个数。事实上，“$”运算符在 RHS 中只

是一个无用的附属物。

下面的例子给出的是在 RHS 中绑定多个变量。多字段值函数(multifield value function)，create$被用来创建一个多字段值。它的基本语法如下所示：

(create$ <arg1> <arg2>…<argN>)

这里，任意个数的参数都可以被作为创建多字段值附属在一起。这些多字段值，或单字段值，可以被约束到 RHS 行为中的一个变量，如下所示：

```
CLIPS>(clear)
CLIPS>(defrule bind-values-demo
=>
        (bind ?duck-bachelors (create$ Dopey Dorky Dinky))
        (bind ?happy-bachelor-mv (create$ Dopey))
        (bind ?none (create$))
        (printout t
                “duck-bachelors” ?duck-bachelors crlf
                “duck-bachelors-no-()” (implode$ ?duck-bachelor) crlf
                “happy-bachelor-mv ”?happy-bachelor-mv crlf
                “none” ?none crlf))
CLIPS>(reset)
CLIPS>(run)
duck-bachelors (Dopey Dorky Dinky)
duck-bachelors-no-() Dopey Dorky Dinky
happy-bachelor-mv (Dopey)
none ()
CLIPS>
```

# 自定义函数

像其他语言一样，CLIPS 允许程序员通过 deffunction 来定义自己的函数。众所周知，deffuntion 可以帮助你节省重复输入相同的行为(actions)。

自定义函数(deffunction)在提高程序的可读性上也是非常有用的，你可以像调用其他函数一样调用自定义函数，自定义函数也可以被用来当作其他函数的参数使用。在自定义函数中，(printout)可以在任何位置使用，甚至不是作为最后一个行为，因为打印的一个副作用是调用了(printout)函数。

自定义函数的通用语法如下所示：

```
(deffunction <function-name> [optional comment]
    (?arg1 ?arg2 … ?argM [$?argN])   ; 参数表，最后一个为可选多字段参数
    (<action1>          ; actionK 之前的行为不会返回值，仅最后一个行为返回值
    <action2>
    …
    <action(K-1)>
    <actionK>))
```

其中，?arg 为虚拟参数(dummy arguments)，代表参数的名字，如果在一条规则中参数名相同，变量不会发生冲突。虚拟变量在其他书本中，通常被称之为参量(parameter)。

尽管每个行为都可以从函数中返回值，这些都被自定义函数返回给用户。自定义函数仅仅返回最后一个行为，该行为可能是个函数，一个变量或一个常量。

下面的例子是一个用来计算三角形斜边的自定义函数，然后被用在规则

中。即使规则中作为虚拟参数的变量名都是一样的，也不会有冲突，这是因为它们都是虚拟的，并不指代任何参数。

```
CLIPS>(clear)
CLIPS>(deffunction hypotenuse       ; 函数名
        (?a ?b)                     ; 虚拟参数
        (sqrt(+ (* ?a ?a)(* ?b ?b)))) ; 行为
CLIPS>(defrule calculate-hypotenuse
        (dimensions ?base ?height)
=>
        (printout t “Hypotenuse=” (hypotenuse ?base ?height) crlf))
CLIPS>(assert (dimensions 3 4))
<Fact-1>
CLIPS>(run)
Hypotenuse=5.0
CLIPS>
```

自定义函数也通常被用在多字段值中，如下例子所示：

```
CLIPS>(clear)
CLIPS>(deffunction count ($?arg)
        (length $?arg))
CLIPS>(count 1 2 3 a duck “quacks”)
6
CLIPS>
```

## 其他特性

其他一些有用的函数如下所示。更多的信息，请参看 CLIPS 参考指南。

| 函数 | 含义 |
| --- | --- |
| round | 四舍五入 |
| integer | 取整 |
| format | 格式输出 |
| list-deffunctions | 函数列表 |
| ppdeffuntion | 打印出函数 |
| undeffunction | 删除函数 |
| length | 字段的长度，或字符串中字母的个数 |
| nth$ | 指定存在的字段，否则为 nil |
| member$ | 如果变量存在，则返回字段的成员，否则为 FALSE |
| subsetp | 如果一个多字段值是另一多字段值的一部分则返回 TRUE，否则返回 FALSE |
| delete$ | 删除给出数字字段内的值 |
| explode$ | 将多字段值以每个字符串元素返回 |
| subseq$ | 返回字段的指定范围 |
| replace$ | 替代指定的值 |

# · 第七章 程序的控制

**当你年轻的时候，你被世界所控制，当你老的时候，你将控制世界。**

到目前为止，你已经学历了 CLIPS 的基本句法。现在，你将学习怎样将所学的句法应用到实际有用的程序当中去。同时，你还将会学到一些关于输入的新的句法，怎样比较变量和产生循环。

## 读入函数

除了模式匹配外，规则还可以通过其他方式获取信息。CLIPS 可以通过使用读入函数(read function)来读入用户输入的键盘信息。

下面是使用(read)命令来输入数据的例子。注意(read)命令在新的一行中插入光标后并不需要多余的(crlf)。(read)自动将光标重置到新的一行中。

```
CLIPS>(clear)
CLIPS>(defrule read-input
=>
        (printout t “Name a primary color” crlf)
        (assert (color (read))))
CLIPS>
(defrule check-input
        ?color <- (color ?color-read&red|yellow|blue)
=>
    (retract ?color)
    (printout t “Correct” crlf))
CLIPS>(reset)
CLIPS>(agenda)
    0       read-input:*
For a total of 1 activation.
CLIPS>(run)
Name a primary color
green
CLIPS>                    ; 没有打印出“correct”
```

上面的规则中，在 RHS 中使用键盘输入，不用指定任意 LHS 中的模式就可以非常方便的被触发，且当(reset)出现后自动的被激活。当输入(agenda)命令后，读入规则的激活将被显示，打印一个*号，而不是像事实标识符，如 f-1。*号的使用用来表示该模式不用指定事实，均可满足。

(read)函数并不是可以读入所有键盘输入的通用函数，它仅能读入一个字段。所以，当你想读入下面的：

primary color is red

那么，只有第一个字段“primary”能被读入。如果你想读入所有的字段，那么必须使用双引号将其包含起来。当然，一旦使用了双引号，那么就会被当作一个单字符串。然后可以通过 str-explode 或 sub-string function 访问子字符串，如“primary”，“color”，“is”和“and”。

(read)的第二个限制是不能输入圆括号，除非使用双引号。就像不能声明一个包含圆括号的事实一样，也不能使用(read)直接读入圆括号。

readline 函数被用来读入多值，直到输入回车键为止。该函数将读入的数据作为一个字符串。为了声明(readline)数据，(assert-string)函数用来声明一个非字符串事实，就像用(readline)输入。(assert-string)例子如下：

CLIPS>(clear)

CLIPS>(assert-string “(primary color is red)”)

<Fact-1>

CLIPS>(facts)

f-0 (initial-fact)

f-1 (primary color is red)

For a total of 2 facts.

```
CLIPS>
```

(assert-string)的参数必须是一个字符串，下面的例子是使用(readline)来声明一个多字段事实。

```
CLIPS>(clear)
CLIPS>(defrule test-readline
=>
        (printout t “Enter input” crlf)
        (bind ?string (readline))
        (assert-string (str-cat “(” ?string “)”)))
CLIPS>(reset)
CLIPS>(run)
Enter input
Primary color is red
CLIPS>(facts)
f-0 (initial-fact)
f-1 (primary color is red)
For a total of 2 facts.
CLIPS>
```

因为(assert-string)声明要求字符串被圆括号括住，(str-cat)用来将括号括住字符串?string。

(read)和(readline)还可以被用来读入文件信息，通过指定文件的逻辑名作为其参数。了解更多的信息，请参看 CLIPS 参考指南。

- **效率问题**

CLIPS 是基于规则的语言，使用了高效的模式匹配算法，即 Rete 算法，该算法是由卡内基-梅隆大学的 Charles Forgy 和他的 OPS 团队设计出来的。Rete 在拉丁语中是网(net)的意思，用来描述模式匹配过程中的软件结构。

很难给出一个精确的规则能够总是促进在 Rete 算法下程序运行的效率。然而，下面的几点可以作为有所帮助的一般指南：

1.将最明确的模式放在规则首位。无界变量和通配符的模式应该放在规则模式的后面。一个控制事实应该放在模式的最前面。

2.较少匹配事实的模式应该先行以减小部分匹配。

3.经常被撤销和声明的模式，不稳定模式(volatile patterns)，应该放在模式列表的最后。

如你所见，这个指南有潜在的对立。一个非明确的模式可能只有少数的匹配（参见 1 和 2），那么它应该怎么归类？以上的指南都是以减少一个推理引擎循环内部分匹配的变化为目的的。这也许要求程序员花更多的心思在监视部分匹配上。一个简单易行的办法是买一台更快的电脑，或更换一个加速主板。这在电脑硬件的价格一路下滑，人力价值一路飙升的今天，听起来很具吸引力。因为 CLIPS 具有可移植性，那么在一台机器上能运行的代码应该同样能在另一台机器上运行无恙。

## 其他特性

条件检验元素(test conditional element)通过比较 LHS 中的数字，变量和字符串提供了一个非常有用的方法。(test)被作为一个模式用在 LHS 中。只有当(test)与其他模式一起满足时，规则才能被触发。

CLIPS 提供了许多预定义函数，如下表所示：

Predefined Functions

| 逻辑 | | 算术 | |
|---|---|---|---|
| not | Boolean not | / | 除法 |

| | | | |
|---|---|---|---|
| and | Boolean and | * | 乘法 |
| or | Boolean or | + | 加法 |

比较

| | |
|---|---|
| eq | 等于(任何类型)。比较类型和大小 |
| neq | 不等于(任何类型) |
| = | 等于(数字类型)。比较大小 |
| <> | 不等于(数字类型) |
| >= | 大于等于 |
| > | 大于 |
| <= | 小于等于 |
| < | 小于 |

以上所有的比较函数除了“eq”和“neq”，在被用于比较一个数字与一个非数字的时候都会提示错误信息。如果类型不能预知，那么就可以用“eq”和“neq”了。eq 函数检查比较参数的大小和类型，而“=”函数仅检查比较数值型参数的大小，而与数字是否为整型还是浮点型无关。

CLIPS 的逻辑函数(logical functions)为 and,or 和 not。他们可以被当作布尔型函数使用在表达式中。CLIPS 中，真和假由符号 TRUE 和 FALSE 表示。注意在 CLIPS 中，必须用大写来表示逻辑值。

除了以上预定义的函数外，你还可以在 C，Ada 或其他程序语言中写外部函数(external functions)或用户自定义函数(userdefine functions)，然后链接到 CLIPS 中来。这些外部函数被用来作为任意的预定义函数使用。

CLIPS 同时还有在 LHS 中指定确定的与条件元素(and conditional element),或条件元素(or conditional element)和非条件元素(not conditional element)。使用“not”条件元素指定 LHS 中缺损的事实。

改变信息以符合实际，被称之为正确性保持(truth maintenance)。也就

是说，我们试图保持我们的心情以包容真实的信息，为的是减少与真实世界的冲突。

当人类能相当简单的做这些的时候(熟能生巧)，对电脑来说却很难，这是因为计算机不能正常的得知哪个模式实体逻辑依赖(logically dependent)另一个模式实体。CLIPS 还支持正确性保持特性，该特性将在内部标记出哪些模式实体是依赖与另一些模式实体的。如果那些模式实体被撤销了，CLIPS 会自动撤销这种逻辑依存。逻辑条件元素(logical conditional element)使用关键字 logical 包含模式，表明模式匹配实体提供 RHS 中声明以逻辑支持(logical support)。

虽然逻辑支持为声明服务，但是它不能重新声明撤销事实。按道理讲，如果由于错误的信息而使你丢失了某些东西，你将不能再找回来了(就像是在你的股票经纪人劝说下损失钱财一样)。

CLIPS 有两个函数用来协助逻辑支持。一是相关函数(dependencies function)，该函数列出了所有的部分匹配，这些匹配来自模式实体接收到逻辑支持，或根本没有支持。第二个逻辑函数是从属函数(dependents)，该函数列出了所有的模式实体，这些模式实体从另一模式实体中接收逻辑支持。

连接约束为“&”，“|”和“～”。另一种字段约束被称之为谓词约束(predicate constraint)，它通常被用在对复杂字段的模式匹配当中。谓词约束的目的是依据布尔表达式的结果来约束字段。如果布尔返回值为 FALSE，那么约束不被满足且模式匹配失败。你将会发现谓词约束在数字模式中非常有用。

谓词函数(predicate function)用来返回 FALSE 或一个非假值(non-FALSE)。谓词函数后面跟着冒号“：”被称之为谓词约束。“：”的优先级高于“&”，“|”或“～”，如写在一起的模式(fact （:＞2 1）))。与“与约束&”的典型使用为“&:”。

谓词函数　　核查参数

(evenp <arg>)　　正好是数字

(float <arg>)　　浮点型数字

(integerp <arg>)　　整型

(lexemep <arg>)　　字符或字符串

(numberp <arg>)　　浮点型或整型

(oddp <arg>)　　奇数

(pointerp <arg>)　　外部地址

(sequencep <arg>)　　多字段值

(stringp <arg>)　　字符串

(symbolp <arg>)　　字符

专家系统中非常方便的获取全局值，有多种情形。举例而言，重新定义通用常量如π是不必要的。CLIPS 提供自定义全局常量(defglobal construct)，这样这些值就会被所有规则广泛知道。

另一个非常有用的函数是随机数字。CLIPS 有随机函数(random function)，用来返回随机整型值。CLIPS 的随机数字函数实际上返回的是伪随机(pseudorandom)数字，这意味着实际上并不是真正的随机，而是由一个数学公式产生出来的。伪随机数字能满足多数目的，CLIPS 的随机函数使用的是 ANSI C 的 rand 库函数，该函数可能在不支持它的电脑系统上无效。关于更多的信息，可参看 CLIPS 参考指南。

除了控制事实以控制程序的执行外，CLIPS 提供了一个更直接的控制事实的方式，那就是显式的声明规则的权值。使用权值进行显式声明的一个问题是当你从一开始用 CLIPS 时就在连续程序中过度使用权值，过度的使用将会达不到使用基于规则语言的目的，基于规则的语言提供的是通过规则应用最好表达的自然手段。同样的道理，对于强导向应用，程序语言是最好的，对于描述对象来说，对象导向语言是最好的。CLIPS 有一个关键词为声明权值(declare salience)，被用在声明规则的优先级别上。

权值被设置在一个数字范围，最低为-10000，最高为 10000。如果一个规则没有被程序员显式的声明权值，CLIPS 自动设置其值为 0。权值 0 介于最大与最小值之间。权值为 0 并不意味着规则没有权值，相反，它有一个中间的优先级。

CLIPS 提供一些过程程序设计结构，这些结构可以用在 RHS 中。这些结构是 while 和 if then els，这些同样可以在其他语言如 Ada，C 和 Java 中找到。

另一个与(while)有关的有用函数是 break，它结束(while)循环的当前执行。return 函数立即结束当前执行的自定义函数，类属函数，方法或消息句柄。

任何函数都可以从 RHS 中调用，这大大增强了 CLIPS 的功能。另一些 CLIPS 函数在返回数字，字符或字符串中很有效。这些函数可以用在它们的返回值或他们的副作用(side-effects)中。一个仅使用副作用的例子是(printout)，由它返回的值都是没有意义的。(printout)的重要性在它的输出的副作用上。通常情况下，如果要到达你期望的效果，函数都会有嵌套的参数的。

文件的读写访问之前，必须使用 open 函数来打开它。一次打开文件的个数取决于你的操作系统和硬件情况。当你不再访问一个文件的时候，你应该使用 close 函数去关闭它。除非文件是关闭的，不能保证写入它的文件将会被保存。

CLIPS 通过逻辑名(logical name)来识别一个文件。逻辑名是一个全局名，可以被 CLIPS 在所有的规则中访问。尽管逻辑名可以与文件名是相同的，但你最好使用不同的。逻辑名的另一个好处是你将很方便的替代一个不同的文件名，而不需要做主程序的修改。

从文件中读取数据的函数是前面介绍的(read)和(readline)。唯一你需要做的是必须制定文件名，作为这两个函数的参数。

使用(read)读取多个字段，必须使用循环(loop)。即使是使用(readline)，

在读取多行字段的时候也是必需的。使用 while-loop 循环可以通过一个规则而连续触发。循环不能用来读文件或操作系统的末端，会出现错误信息。为了防止这个问题，在如果你尝试读入文件的末端时，CLIPS 会返回一个 EOF 字符字段。

评价函数(evaluation)---eval，被用来评估任意的字符串或字符，除了自定义类型结构，如 defrule，deffacts 等，像在顶端输入一样。build 函数用于自定义类型结构，是 eval 函数的补充。build 函数评估字符串和字符就像被在顶端输入一样，如果参数符合自定义类型结构如 deffrule，deffacts 等，则返回 TRUE。

## · 第八章 继承事项

**获取财富最简单的办法莫过于继承遗产，其次就是剥削他人的劳动，与财富联姻太像一项工作了。**

本章为 CLIPS 面向对象编程的概述章节。与基于规则的编程不同，基于规则的编程中，你可以不用考虑系统中其它的东西，而及时投入进去，编写规则，而面向对象的编程需要许多重要的背景资料。

## 如何客观

好的程序设计的一个重要特性是应具有灵活性(flexibility)。不幸的是，结构化编程技术刻板的方法论不能提供快速，可靠和高效变化的灵活性。面向对象编程范式(object-oriented programming (OOP) paradigm)提供了这种灵活性。

术语——范式来源于希腊语 paradeigma，意思是一个模型，例子或模式。在计算机科学中，范式为一致的，有组织的尝试解决问题的方法论。当前，有很多的设计范式，如 OOP，过程式的(procedural)，基于规则的(rule-based)和联系式的(connectionist)。术语——“人工神经网络”(artificial

neural systems)，是旧术语“联接”(connectionist)的现代同义词。

传统的程序设计是过程化的，因为它强调在解决问题中的算法和程序。有相当多的语言被开发以支持这样的过程化范式，如 Pascal，C，Ada，FORTRAN 和 BASIC。这些语言同样也适用于面向对象的设计(object-oriented design (OOD))，它们通过增加延伸或利用程序员的设计方法学。相比而言，新的语言已经被设计出来，用以提供 OOP，与 OOD 并不一样。你可以在任意语言中使用 OOD，甚至汇编语言。

CLIPS 提供三个范式：规则，对象和过程。你将在 CLIPS 面向对象语言(CLIPS Object-Oriented Language (COOL))中了解到更多关于对象的信息，COOL 整合了规则和过程两个 CLIPS 基本范式。CLIPS 通过类属函数，自定义函数和用户自定义外部函数来支持过程化范式。视应用程序的不同，你可以选择使用规则，对象，过程或是它们的组合。

与其利用单个的范式，我们的哲学是：多种专门工具，多范式途径要比尝试去强制所有人使用单一万能的工具。类似的，你可以用锤子和钉子来固定一切，但同时也有很多其他首选的方法来固定某物。举个例子，想象一下你用锤子和钉子来代替拉链来扣紧你的裤子吧。(注意：如果有人使用锤子和钉子来扣紧裤子，那么请联系吉尼斯世界纪录吧!)

## 类

OOP 中，类是描述具有相同特性和属性(attributes)的对象的模板。注意这里使用的术语——“模板”与前面所讲的自定义模板(deftemolate)是两码事。这里，模板一词用的是工具的表意，用来构建具有相同属性的对象。类似的，直尺是画直线的模板，饼干模子是做出曲奇的模板。

对象的类在层次和图线上被安排来描述系统中对象之间的关系。每个类都是实际系统或其他一些我们正尝试塑造的逻辑系统的抽象。举个例子，一个现实系统的抽象有可能是一辆汽车，逻辑系统的另一个抽象模型可能是金融证券，如股票，契约或复数。术语——“抽象”一词引用于两点，(1)

对现实对象或其他我们正尝试塑造系统的抽象描述。（2）用术语类表示一个系统的过程。抽象是一个真正 OOP 语言所广泛接受的五个特征之一。其他分别是继承(inheritance)，封装(encapsulation)，多样性(polymorphism)和动态绑定(dynamic binding)。当你通读本书后将会对以上术语有所详细了解。CLIPS 支持以上五种特性。

抽象一词意味着我们不用关注一些非实质性的细节。复杂系统的抽象描述是一种集中于指定目标重大信息的精简描述。那样，系统就被更简单，易懂的模型表示了。作为一个熟悉的例子，当某人开车的时候，他利用的是包含两个部分——转向轮和油门的抽象驾驶模型。这样，人们不会去关心组成摩托车的上百个零部件和内部燃烧发动机的理论知识，交通法规等等了。知道怎样使用转向轮和油门就是他们驾驶的抽象模型。

继承是 OOP 的五个基础特性之一。类在继承的设计上，遵循将最一般的类放在顶层，最特殊化的类放在底层。这允许新的类被作为特殊定制或对已存在类修改而重新定义。

继承的使用可以大大加速软件的开发和增加可靠性，因为建立一个新的软件，不必要每次从编写新的程序白手起家。OOP 利用可再用代码(reusable code)使得以上变得简单。OOP 程序员常常利用包含成百上千个对象的对象库，这些对象能被使用或作为设计新程序而修改。除了公共领域的对象库，还有许多公司市场商业化对象库。尽管可再用软件组件的概念早已在 1960 的 FORTRAN 子程序库中被贯彻，但是它还从未被如此成功的应用于通用软件开发中。

定义一个类，你必须指定被定义类的一个或多个父类(parent classes)或超类(superclasses)。关于超类的比如，每个人都有父母，没有哪个人是天然就存在的（尽管有些时候你也许想知道某些人是否真的有父母）。超类相对的是子类(child class)和亚类(subclass)。

这些决定新类的继承。子类继承属性来源于一个或多个超类。COOL 中的属性一词引用于对象的道具(properties)，被命名为槽(slot)来描述它。

举个例子，一个表示人的对象可能包含有姓名，年龄，地址等等的槽。

实例(instance)是拥有了槽值的对象，如约翰.史密斯，28 岁，清湖市主街道 1000 号。底层类自动继承高层类的槽，除非某个槽被显式的关闭了。除了继承的槽值被设置所有的属性，新的槽也被定义来描述类。

对象的行为(behavior)由它的消息句柄(message-handlers)或槽的操作(handlers)。对象的消息句柄对消息(messages)响应和执行要求的行为。举个例子，发送消息：

```
(send [John_Smith] print)
```

将引起相应的消息句柄以打印实例 John_Smith 的槽值。实例通常被指定在中括号[](brackets)内。消息由 send 函数开始，后面跟实例名，消息名和所有要求的参数。举个例子，在打印消息的情况下，没有参数。CLIPS 的对象就是一个类的实例。

对象内槽和操作的封装是 OOP 广泛接受的五个特性之一。封装一词的意思是类按照它的槽和操作被定义。尽管一个类的对象可以继承它超类的槽和操作，（一些例外稍后再讲），如果没有发送消息到对象，对象的槽值不会被改变或消除。

CLIPS 的根类(root class)或简单根(root)是被称之为 OBJECT 的预定义系统类(predefined system class)。预定义系统类 USER 是 OBJECT 的子类。

- **继承**

一个例子，假设我们想定义一个类，名为 UPPIE（优皮），是城市专业人员(urban professional)的口语化称呼。注意在本书中，我们都采用以大写来书写类的惯例。

图 1.1 说明了 UPPIE 们怎样从根类 OBJECT 中得到继承的。注意，UPPIE 被作为 USER 的子类来定义。盒子或节点用来表示类，连接箭头被称为链

接(links)。直线常被用来代替箭头以便简化画图。同理，因 CLIPS 仅支持 is-a 链接，从现在起，“is-a”关系将不被靠近每个链接显式书写。

我们将遵循类之间关系(relationship)的惯例为：箭头的末端为子类，箭头所指的为超类。图 1.1 中的关系遵循该惯例。另一个可能的惯例是用箭头指向子类。

is-link 链接指示从一个类到它的子类槽的继承。一个类可能有零个或多个子类。除了 OBJECT，所有的类必须有个超类。UPPIE 继承了 USER 所有的槽，USER 又继承了 OBJECT 所有的槽，也就是 UPPIE 继承了 OBJECT 所有的槽。继承的理论同样适用于每个类的消息处理。举个例子，UPPIE 继承了 USER 和 OBJECT 所有的操作。

槽与操作的继承在 OOP 中尤其重要，它意味着你不用对每个已经定义过的类的对象属性和行为重新定义。相反，每个新的类继承了它的高层类所有的属性和行为。因为新的行为是继承的，它可能本质上减少了操作的验证与测试(verification and validation (V&V))。V&V 实质上意味着合理的打造产品并满足所有的要求。验证和测试软件的任务可能比软件开发本身要花更多的时间和金钱，特别是如果该软件能影响到人类生活和财产。操作的继承允许高效再利用

已存在的代码和加快开发速度。

CLIPS 中使用自定义类(defclass)结构来定义一个类。类 UPPIE 被定义声明如下：

```
(defclass UPPIE (is-a USER))
```

注意与图 1.1 中 UPPIE-USER 关系中的相似点和(defclass)结构。

你不必输入 USER 或 OBJECT 类，因为它们都是预定义的类，CLIPS

已经知道它们之间的关系。事实上，如果你尝试定义 USER 和 OBJECT，将会出现错误消息，这是因为你不能改变预定义的类，除非你改变 CLIPS 的源代码。

因为 CLIPS 是区分大小写(case-sensitive)的，命令和函数必须以小写输入。预定义系统类，如 USER 和 OBJECT 则必须以大写输入。尽管你可以以小写或大写输入用户定义类，我们将遵循以大写来输入类，以增加可读性。

定义类（非槽）的自定义类命令基本的格式，如下：

(defclass <class> (is-a <direct-superclasses>))

类表(direct-superclasses)被称为直接超类优先表(direct superclass precedence list)，因为它定义了类直接链接的超类。类的直接超类为一个或多个类，命名在 is-a 关键字后面。在我们的例子中，类 DUCKLING 为 DUCK 的直接超类。注意，直接超类优先表中至少有一个直接超类。

如果直接超类表如下所示：

(defclass DUCK (is-a DUCKLING USER OBJECT))

那么，USER 和 OBJECT 同样也是 DUCK 的直接超类。在这个例子中，除了 DUCKLING，无论 USER 与 OBJECT 哪一个被指定都没有分别。事实上，因为 USER 和 OBJECT 是预定义类，它们都被链接，以至于 USER is-a OBJECT，OBJECT 为根类，除了当定义一个 USER 的子类时，你不必分清它们。因为 USER 仅继承于 OBJECT，如果 USER 已经被指定，那么 OBJECT 就不必再被指定了。

类的非直接超类(indirect superclasses)是所有的没有命名在 is-a 后面，继承属性槽和消息处理的类。在我们的例子中，非直接超类为 USER 和

OBJECT。一个类从所有它的直接和非直接超类中继承槽和消息处理。因此，DUCK 继承于 DUCKLING，USER 和 OBJECT。

一个直接子类(direct subclass)由一个单链接与它上面的类连接。一个非直接子类(indirect subclass)有多于一个的链接。图 1.2 概述了类术语。

根类 OBJECT 是唯一没有超类的类。

使用这些新奇属于允许我们声明：

**Principle of OOP Inheritance**

A class may inherit from all its superclasses.

这是一个简单，但在 OOP 中被全面利用的很有用的概念。这个理论意味着槽和消息处理可以被继承，以节省为新子类重新定义它们的麻烦。此外，新子类的槽可以简单的用户化，以修改和作为子类槽的合成。由允许的简单灵活、已经存在的可再用代码，程序开发的时间和花销大大被缩减了。另外，有效、存在的可再用代码，可以减少验证和确认的任务量。一旦这些代码被释放，上述所有的优点可以推进程序维持调试，修改和增强等任务。

理论上使用“可能”的原因是强调从类中继承的槽，可能被包含一个非继承面的类槽定义封锁。

类的直接和非直接类都是那些依赖于 OBJECT 的继承路径(inheritance path)。继承路径是一组类与 OBJECT 间的关联节点。在我们的例子中，DUCK 的单继承路径为 DUCK，DUCKLING,USER 和 OBJECT。后面的例子，如图 1.6 中，你将发现一个类到 OBJECT 有多继承路径(multiple inheritance)。

图 1.3 阐明的是一个简单的有机体分类学(taxonomy)，图解了自然界中的继承。术语分类学的意思为分类。生物分类学的目的是显示有机体之间

的血缘关系。也就是说，有机体分类学强调的是个体与聚群之间的相似性。

在如图 1.3 的分类中，所有的连接线都是 is-a 链接。举个例子，DUCK is-a BIRD,BIRD is-a ANIMAL,ANIMAL is-a ORGANISM 等等。尽管每个个体的起源继承都不尽相同，但是 MAN 和 DUCK 物种的特性是相同的。

图 1.3 中，最通用的类为 ORGANISM 处在最顶端，其他物种类均在分类的下面。在 CLIPS 术语中，我们可以说每个子类继承了父类的槽。举个例子，哺乳动物是胎生的暖血动物，除鸭嘴兽外，MAN 类继承了父类 MAMMAL 类的属性。MAMMAL 的直接超类为 ANIMAL，且直接子类为 MAN，MAMMAL 的非直接超类为 ORGANISM。

其他的类，如 BIRD，DUCK 等都与 MAMMAL 无关，因为它们不是在从原始类 ORGANISM 下来的同一条继承路径(inheritance path)中。继承路径为任意一条从一个类到另一个类，不包含回溯和重新的路径。类 PLANT 不在 MAMMAL 的继承路径当中，因为在达到 MAMMAL 之前，我们必须回溯到 ORAGNISM。这样，MAMMAL 就不能自动获取 PLANT 和其他不在其继承路径上类的任何槽。继承模型反映了真实的世界，否则我们头上的头发也许会被杂草代替了。

## · 非法继承类

现在你已经对类有了初步的认识，让我们在图表 1.1 的 UPPIE 中增加一些其他的类，以使的例子更具有实际意义。通过增加低层类的开发类型正是 OOP 的方式，由最一般到最特殊的方式增加类。

图 1.5 所示的是非合法 YUKKIE 的继承图表。为了简便，这里没有标出 OBJECT 和 USER 类。图 1.5 是一个树(tree)形分级系统，每个节点有唯

一的父节点。

一个常见的树形组织结构例子是经常用在公司中，由总裁，副总裁，部门领导，经理等组成的雇员等级分层(hierarchy)。在这种情况下的等级分层反映的是组织中每个人权利的大小。树通常被用来表示人的组织关系，因为每个人都有唯一的老板，当然 CEO 除外。组织图表中的节点代表着人在组织中的职位，如总裁，副总裁等。连接各职位的线指代各职能部门分支。在树中的链接通常被称之为分支。

图 1.5 中，除了 YUKKIE 意外的类都是合法或不合法的。举个例子，SUPPIE is-a UPPIE，MUPPIE is-a UPPIE,YUPPIE is-a UPPIE，PUPPIE is-a YUPPIE(没有中年 YUPPIE 妈妈允许的)。我们也同样可以说 YUKKIE is-a YUPPIE 和通过继承，YUKKIE is-a UPPIE。然而，我们不想因 YUKKIE 与 PIPPIE 之间的 is-a 链接，说 YUKKIE is-a PUPPIE。

YUKKIE 与 PUPPIE 之间的 is-a 链接是一个自然错误，让人误以为 YUKKIE 是 PIPPIE 的子辈（事实上，当她生孩子后是前 PUPPIE)。尽管在 YUKKIE 与 PUPPIE 之间的 is-a 链接允许 YUKKIE 可以继承 YUPPIE 和 UPPIE，但同时也产生了一个说 YUKKIE is-a PUPPIE 的不合法的关系。这意味着 YUKKIE 将继承 PUPPIE 所有的槽。假设 PUPPIE 的一个槽用来指定 PUPPIE 怀孕的月份数，这意味着每个 Yuppie 小孩将拥有相同的槽来表示他或她怀孕的月份数了。

更正图表是有可能的。然而，我们需要使用图表来代替树，与树对比，树中除根节点，每个节点都有唯一的父节点，图表中的每个节点可以有零个或多个节点与之连接。一个类似的例子是地图，每个城市都是一个节点，道路将他们互相链接。树与图表的另一个不同是，多数类型的树都有分层结构，而普通的图表没有。

图 1.6 显示的是非法 Yuppie 类 YUKKIE。一个新类 CHILD 被创建，YUKKIE 与他的两个超类 YUPPIE 和 CHILD 之间用 is-a 链接。注意，YUKKIE

与 PUPPIE 之间的非法链接不再存在了。

这里采用图表，因为 YUKKIE 类有两个直接的超类，所以代替了只能有一个的树。图表中同样有分层结构，因为所有的类通过 is-a 链接自最一般的 USER 到最特殊的 SUPPIE,MUPPIE,PUPPIE 和 YUKKIE 安排。通过图 1.6，我们可以说 YUKKIE is-a YUPPIE，同时也可以说 YUKKIE is-a CHILD。

下面显示的是增加如图 1.6 子类的命令。

```
CLIPS>(clear)
CLIPS>(defclass UPPIE (is-a USER))
CLIPS>(defclass CHILD (is-a USER))
CLIPS>(defclass SUPPIE (is-a UPPIE))
CLIPS>(defclass MUPPIE (is-a UPPIE))
CLIPS>(defclass YUPPIE (is-a UPPIE))
CLIPS>(defclass PUPPIE (is-a YUPPIE))
CLIPS>(defclass YUKKIE (is-a YUPPIE CHILD))
```

定义类的顺序必须如上，一个类必须在他的子类之前被定义，如：

```
(defclass CHILD (is-a USER))
```

必须在下面之间输入：

```
(defclass YUKKIE (is-a YUPPIE CHILD))
```

当你试图在 CHILD 之前输入 YUKKIE 类时，CLIPS 会产生一个错误的消息。

注意图 1.6 中 SUPPIE，MUPPIE 和 YUPPIE 自左到右的出现顺序。这

对应了我们在 CLIPS 中输入顺序的惯例。你同样可以看到为什么 CLILD 被画在 UPPIE 的右边，因为它是在 UPPIE 类的后面输入。

在图 1.6 中，注意到 YUKKIE-YUPPIE 的链接出现在 YUKKIE-CHILD 的左边。另一个惯例是，我们通常由图表中自左到右出现的顺序，来自左到右书写优先表中的直接超类。YUKKIE 优先表中的 YUPPIE，CHILD 顺序符合这种惯例。

- **查看**

CLIPS 提供了许多查看类信息的函数，如谓语函数，用来检测一个类是否为另一个类的超类或子类。

如果<class1>为<class2>超类，superclassp 函数返回 TRUE，否则返回 FALSE。如果<class1>为<class2>的子类，subclassp 函数返回 TRUR，否则返回 FALSE。两函数的通用格式为：

(function <class1><class2>)

举个例子：

```
CLIPS>(superclassp UPPIE YUPPIE)
TRUE
CLIPS>(superclassp YUPPIE UPPIE)
FALSE
CLIPS>(subclassp YUPPIE UPPIE)
TRUE
CLIPS>(subclassp UPPIE YUPPIE)
FALSE
CLIPS>
```

现在，让我们检查一下 CLIPS 是否接受了所有的新类。一种方式是通过执行 list-defclasses 命令。下面是该命令的输出：

```
CLIPS>(list-defclasses)
FLOAT
INTEGER
SYMBOL
STRING
MULTIFIELD
EXTERNAL-ADDRESS
FACT-ADDRESS
INSTANCE-ADDRESS
INSTANCE-NAME
OBJECT
PRIMITIVE
NUMBER
LEXEME
ADDRESS
INSTANCE
USER
INITIAL-OBJECT
UPPIE
CHILD
SUPPIE
MUPPIE
YUPPIE
```

PUPPIE

YUKKIE

For a total of 24 defcalsses.

CLIPS>

注意(list-defclasses)命令不会指示出分层类的结构。也就是说，该命令不会指示出某个类是另外一个类的超类或子类。

如果你往该表的下面看，你会看到所有你输入的用户自定义类：UPPIE,CHILD,YUPPIE,MUPPIE,SUPPIE,PUPPIE 和 YUKKIE。除了已经讨论的预定义系统类 OBJECT 和 USER 外，还有许多其他的预定义类。你应该像熟悉前几章所学的 CLIPS 类一样知道它们的名字。CLIPS 的预定义类型同样也被作为类来定义，这样它们就可以使用 COOL 了。

图 1.7 为 CLIPS 参考指南中的一般继承图表，里面的箭头指向子类。

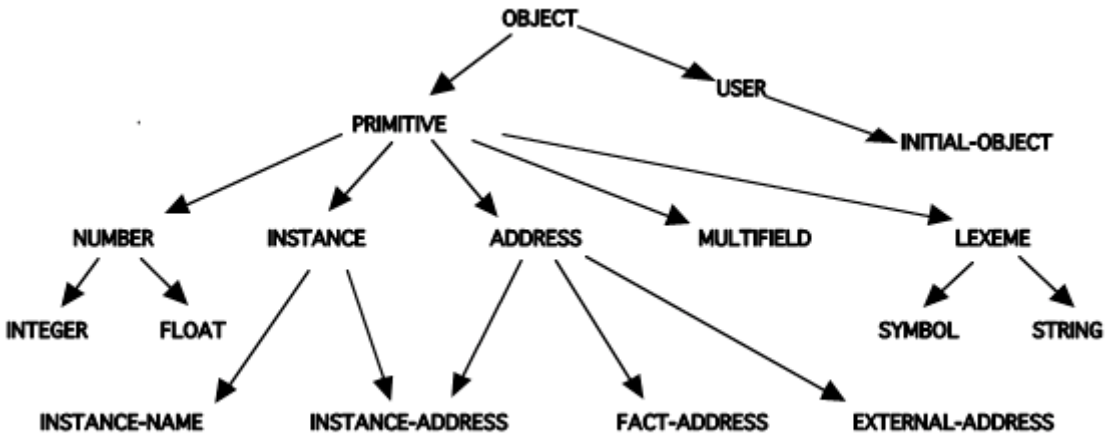


Fig.1.7　The Predefined CLIPS Classes

OBJECT 类为树的根，由分支(branches)与子类相连。术语分支，边缘(edge)，链接和弧(arc)是基本同类名词，它们均指代节点之间的连接。每个子类比它的超类要低一级。类由层级(level)编号。根类 OBJECT 为 0 级。

大级别数意味着高度定制(specificity)。术语 specificity 意味着类限制

性越强。举个例子，LEXEME 为 SYMBOL 和 STRING 的超类。如果你知道一个对象是 LEXEME 类，那么你就知道它只可能是 SYMBOL 或 STRING。然而，如果一个对象是 SYMBOL，那么它不可能是 STRING，反之亦然。因此，SYMBOL 和 STRING 类要比 LEXEME 更特殊。

browse-classes 命令以首行缩进格式显示了类的层级。

```
CLIPS>(browse-classes)
OBJECT
    PRIMITIVE
        NUMBER
            INTEGER
            FLOAT
        LEXEME
            SYMBOL
            STRING
        MULTIFIELD
        ADDRESS
            EXTERNAL-ADDRESS
            FACT-ADDRESS
            INSTANCE-ADDRESS *
        INSTANCE
            INSTANCE-ADDRESS *
            INSTANCE-NAME
    USER
        INITIAL-OBJECT
        UPPIE
            SUPPIE
```

```
            MUPPIE
            YUPPIE
               PUPPIE
               YUKKIE *
         CHILD
            YUKKIE *
CLIPS>
```

类名后面的星号指示它有多个超类。

(browse-classes)命令有一个可选参数，用来指定你想从哪个类开始查看。这对于当你对表中一些其他类不感冒时，非常方便。下面的例子阐述了怎样显示图 1.6 中部分 YUPPIE 图表，子树(subtrees)或子图表(subgraphs)依据来自树或图表中的节点和链接。

```
CLIPS>(browse-classes UPPIE)
UPPIE
   SUPPIE
   MUPPIE
   YUPPIE
         PUPPIE
         YUKKIE *
CLIPS>(browse-classes YUPPIE)
YUPPIE
   PUPPIE
   YUKKIE *
CLIPS>(browse-classes YUKKIE)
YUKKIE *
```

CLIPS>

• 有一个抽象类(abstract class)专门为继承设计。抽象类 USER 不能有直属实例(direct instance)为之定义，而是作为实例直接定义给类。除了类信息，类的优先继承也被表示。这是一个顺序表(ordered list)，从顺序表左到右表示了实例的类的优先等级。继承优先列出了所有类到根类 OBJECT 的超类。同时，你也可以看到直接超类信息指示了这些超类为一个类的上部链接，同时继承优先表显示了所有的超类。

即使一个类没有直接实例，如果它的子类有实例，则它也有直接实例。一个类的非直接实例是所有它子类的实例。

一个具体的类(concrete class)被允许拥有直接实例。举个例子，一个具体的类 COW，它的直接实例“基站”可能是著名的电视“salescow”。从抽象类继承而来的普通类同样是抽象类。然而，从系统类继承而来的类，如 USER，被认为是具体的，除非有别的指定。所以，UPPIE 和 YUPPIE 同样是具体的类。

强烈推荐你在 CLIPS 中定义的所有类都为 USER 的子类。这样，CLIPS 将自动提供句柄用以打印，初始化和删除。

因为有两个直接超类 CHILD 和 YUPPIE，YUKKIE 类有多种继承性。如果你回想起组织中老板下的继承相似性，那么多重继承性将会产生一个有趣的问题——谁才是老板？在树结构的例子中，每个类仅有一个直接超类(boss)，这样就可以简单的统计出谁从谁哪里接受命令这样的关系。然而，在多重继承的例子中，出现了多个具有同等职权的“老板”，这里的“老板”为直接超类。

对于排列在一棵树结构中的类来说，也就是单继承，继承关系简单，所有类按照继承路径回溯到 OBJECT。树中的继承路径即从一个类到 OBJECT 之间的最短路径。继承的概念同样适用于图表与子图表中。举个例子，子图 UPPIE，SUPPIE，MUPPIE，YUPPIE 和 PUPPIE 的优先继承是棵树结构，因为每个子图仅有一个父图。这样，每个子图的优先继承关系即为回溯到

OBJECT 的最短顺序。举个例子，PUPPIE 的优先继承为 PUPPIE，YUPPIE，UPPIE，USER 和 OBJECT。

图 1.8 为另外一个图例。注意有些节点如 rhombus 有多个父节点。

为了简化，我们将在本书中仅讨论单层继承。多重继承的细节，可以参考 CLIPS 参考手册。

## 其他特性

类的一些其他有用函数如下所列：

| 函数 | 含义 |
| --- | --- |
| ppdefclass | 格式打印自定义类的内部结构 |
| undefclass | 清除类 |
| describe-class | 关于类的附加信息 |
| class-abstractp | 预定义函数，如果类为抽象类返回 TRUE |

如果你需要了解更多以上函数或本章中提到的函数的信息，请参看 CLIPS 参考手册。

# · 第九章 富有意义的消息

**取悦你上司的好处总是比取悦你的下属要多。---鲍文**

在本章中，你将学到许多关于类和对象调用实例的方法。同时，你也将学到怎样利用槽值指定类的属性，怎样发送消息到对象。

## OOP 特性

在第一章中，我们已经学习了继承的基本观点。OOP 中的继承之所以如此重要，缘自于：继承允许了定制软件(customized software)的最简结构。

对于“定制软件”，我们指的并不是胡乱拼凑起来的软件，而是，它接近于带有批量项目并为特殊应用而修改的那种。批量项目可以被认为是一种软件工厂(software factory)式的产品，它可以快速、经济并可靠的生成一般形式的项目，这意味着方便定制。

OOP 范式的核心是建立起具有级联的类，以更简便快速，可靠的开发出新的软件。通常情况下，新的软件都是在现有软件基础上做些修改，这样，程序员就不必总是重复循环已有的工作了。

尽管在过去，人们便已经开始尝试提供可再利用代码，通过将这些代码作为子程序库的原理，计算机语言中的纯粹 OOP(pure OOP)，如 smalltalk 利用系统中所有的软件都作为再生代码，将这个概念应用到它的逻辑结论中。在 smalltalk 中，所有的都是对象，甚至包括类。在 CLIPS 中，原始类型的实例，如 NUMBER,SYMBOL,STRING 等等，还有像用户自定义的实例都是对象。在 CLIPS 中，类不是对象。举个例子，NUMBER 类型的 1，SYMBO 类型的 Duck，STRING 类型的“Duck”，和用户自定义实例 Duck 都是对象。

OOP 范式与子程序库非常之不同，在于子程序代码中的片段语句可以被使用，也可以不被使用，这取决于程序员一时的突发奇想。OOP 范式鼓励和支持模块代码——即消息句柄——这样方便于修改和维护。随着系统规模和花费的增加，这种可维护性代码的特点发挥着越来越重要的作用。

OOP 中的类就像是一个软件工厂(software factory)，它包含了关于对象的设计信息。换句话说，类就是一个模板，它可以被用来产生相同的对象，这些对象是类的实例。最经典的比如是，类好比一头奶牛的蓝图，对象可以产生牛奶，如 Elsie，便是实例。

实例名的通用语法为一个被方括号——[]括住的简单字符。如下所示：

[<name>]

上面的方括号事实上并不是实例名的一部分，它只是一个字符，如 Elsie。

方括号被用来括住实例名，以防止在使用实例名时出现的模棱两可。这也可以出现在(send)函数中。在有模糊不清的情况下，使用方括号，有百益而无一害。

CLIPS 中对象的一些不同类型指示如下：

| 对象 | 类 |
| --- | --- |
| Dorky_Duck | SYMBOL |
| “Dorky_Duck” | STRING |
| 1.0 | FLOAT |
| 1 | INTEGER |
| (1 1.0 Dorky_Duck “Dorky_Duck”) | MULTIFIELD |
| <Pointer:00AB12CD> | EXTERNAL-ADDRESS |
| [Dorky_Duck] | DUCK |

类 SYMBOL,STRING,FLOAT,INTEGER 和 MULTIFIELD 拥有相同的名字，这些你已经在 CLIPS 的基于规则编程中熟悉了。这些被称为原始对象类型(primitive object types)，因为它们由 CLIPS 提供并且按照需求自动维护。这些原始类型主要被提供用在类函数(generic functions)中。NUMBER 为复合类，它可以是 FLOAT，也可以是 INTEGER，LEXEME 可以是 SYMBOL 或 STRING。复合类的提供是为了方便数字和字符类型的匹配。

相比之下，用户自定义对象类型是那些通过用户自定义类产生的。如果你回溯到第一章 1.7 图中的预定义 CLIPS 类，你会发现，原始类和自定义类在 CLIPS 中都是类的最高阶层分类。

两个函数转换一个字符到一个实例名，反之亦然。Symbol-to-instance-name 转换一个字符到实例名，如下所示：

```
CLIPS>(clear)
CLIPS>(symbol-to-instance-name Dorky_Duck)
[Dorky_Duck]
CLIPS>(symbol-to-instance-name (sym-cat Dorky “_” Duck))
[Dorky_Duck]
CLIPS>
```

注意 CLIPS 标准函数如(sym-cat)怎样将两个部分连结起来，可以同 CLIPS 的对象系统一起使用。

反函数——instance-name-symbol，将一个实例名转换为一个字符，如下所示：

```
CLIPS>(instance-name-to-symbol [Dorky_Duck])
Dorky_Duck
CLIPS>(str-cat (instance-name-to-symbo [Dorky_Duck]) “ is a DUCK”)
“Dorky_Duck is a Duck”
CLIPS>
```

## 其他特性

OOP 与 Nature 有所不同。在 Nature 中，对象仅仅只能类似对象复制，就像鸟和蜜蜂(鸡和蛋不属于此类规则中)。然而，在 OOP 中，实例只能由类模板创建。在纯 OOP 系统中，如 Smalltalk，指定类的对象由发送相关消息到类而创建。事实上，OOP 的核心包含可以发送许多不同类型的消息从一个对象到另一个的特性，甚至可以从一个对象到其本身。

为了查看消息的工作机制，让我们输入以下命令创建一个自定义 DUCK 类，并检查是否已经被输入。注意：DUCK 类并没有指定角色(role)

描述符。如果一个类有具体化角色(role concrete)，类的直接实例便能被创建。如果角色没有被指定，CLIPS 依据继承来确定角色。在继承决定角色中，系统类起着具体类的作用。通过缺省值，任何从 USER 继承的类都是具体类，并且不必被声明作为允许直接实例的创建。

如果一个类拥有抽象角色(role abstract)，那么它的非直接实例能被创建。抽象类仅仅只能作为继承目的来被定义。举个例子，一个命名为 PERSON 的抽象类能被定义如名字，地址，年龄，身高，体重等属性，这些均继承自具体类 MAN 和 WOMAN。MAN 的一个直接实例可以是一个叫 Harold 的男人，WOMAN 的一个直接实例可以是一个叫 Henrietta 的女人。

```
CLIPS>(defclass DUCK (is-a USER))
CLIPS>(describe-class DUCK)
----------------------------------------------------------------------------------------
    ----------------------------------------
**************************************************************************
    **
Concrete: direct instances of this class can be created.
Reactive: direct instances of this class can match defrule
patterns.

Direct Superclasses: USER
Inheritance Precedence: DUCK USER OBJECT
Direct Subclasses:
---------------------------------------------------------------------
Recognized message-handlers:
init primary in class USER
delete primary in class USER
```

```
create primary in class USER
print primary in class USER
direct-modify primary in class USER
message-modify primary in class USER
direct-duplicate primary in class USER
message-duplicate primary in class USER
*****************************************************************
    **
-------------------------------------------------------------------------------------------
    ----------------------------------------
CLIPS>
```

因为在 CLIPS 中，类不是对象，所以我们不能发送消息来创建一个对象。取而代之的是，make-instance 函数可以被用来创建一个实例对象。基本语法如下所示：

```
(make-instance [<instance-name>] of <class><slot-override>)
```

通常情况下，可以指定实例名。然而，如果你没有指定实例名，CLIPS 会利用 gensym*函数产生，同时槽值也会被指定。

现在，我们拥有一间鸭子工厂，让我们按如下步骤创建一些实例，实例名由大括号括住，注意使用“of”关键字区分实例名和类名。你必须包含“of”或者将会得到一个语法错误的结果。同样，注意代码中的大括号意味着实例名，当元句法中的大括号，如(make-instance)，意思是可选。

```
CLIPS>(make-instance [Dorky] of DUCK)
[Dorky]
```

CLIPS>(make-instance [Elsie] of COW)

[PRNTUTTL1] Unable to find class COW.

CLIPS>(make-instance Dorky_Duck of DUCK)

[Dorky_Duck]

CLIPS>(instances)

[initial-object] Of INITIAL-OBJECT

[Dorky] of DUCK

[Dorky_Duck] of DUCK

For a total of 3 instances.

CLIPS>

## • 删除实例

尽管一个(reset)命令可以删除除了[initial-instance]外所有的实例，同时也可以用(definstances)命令来创建新的实例。如果你想永久的删除一个实例，那么你可以试试(unmake-instance)函数，它可以删除一个或所有的实例，删除时需指定参数以确定哪条该被删除，如果删除所有的，则用“*”。

下面即为(unmake-instance)命令的例子：

CLIPS>(unmake-instance *)　;删除所有的实例

TRUE

CLIPS>(instance)　　;验证下是否全部删除

CLIPS>(reset)　　;重新创建新的实例

CLIPS>(instances)　　;验证新实例的创建

[initial-object] of INITIAL-OBJECT

[Dorky] of DUCK

[Dorky_Duck] of DUCK

For a total of 3 instances .

```
CLIPS>(unmake-instance [Dorky])   ;删除一个指定的实例
TRUE
CLIPS>(instances)
[initial-object] of INITIAL-OBJECT
[Dorky_Duck] of Duck
For a total of 2 instances .
CLIPS>
```

另一个删除指定实例的方法是发送(send)一个删除(delete)消息。(send)函数的基本语法如下所示：

```
(send [<instance-name>] <message>)
```

- 一条命令中只能有一个实例名被指定，如果是一个自定义实例名，则必须由中括号包含。

举个例子，下面的例子将使实例 Dorky_Duck 消失。

```
CLIPS>(reset)              ;重新创建实例
CLIPS>(instances)      ;验证新创建的实例
[initial-object] of INITIAL-OBJECT
[Dorky] of DUCK
[Dorky_Duck] of DUCK
For a total of 3 instances .
CLIPS>(send [Dorky_Duck] delete)
TRUE
CLIPS>(instances)
[initial-object] of INITIAL-OBJECT
```

[Dorky] of DUCK

For a total of 2 instances .

CLIPS>

在(send)函数使用“*”是无效的，“*”只有在(unmake)函数中才起作用。另一个办法是你可以定义自己的句柄用以删除，这样系统可以接受“*”允许你这样删除实例：(send [instance-name] my_delete *)。

(send)消息仅仅只有依靠有合适句柄的目标对象(target object)才能执行行为。CLIPS 自动为每个自定义类提供如 print,init,delete 等等句柄。认识到消息(send [Dorky_Duck] delete)只在该实例是自定义类下才起作用非常重要。如果你定义一个不是继承于 USER 的类，如 INTEGER 的子类，那么你必须也要创建合适的句柄来实现你的设计要求，如 printing,creating,deleteing 实例。定义 USER 的子类，并利用系统提供句柄的优点则更容易些。

## · 消息函数的作用

(send)函数是 OOP 的核心，因为它是对象间关联的唯一合适途径。通过对象封装原理，一个对象可以通过发送消息来获取另一对象的数据。

举个例子，如果某人想知道你早餐吃了些什么，他们通常会问你，比如发一个信息给你。一个不太礼貌的回应或许是突然张开你的嘴巴，耸动一下你的喉结。如果对象的封装原理不起作用，任何对象对于其他对象的私有部分将毫无意义，这是个潜在的灾难性后果。

(send)函数的一个非常有用的应用是用来打印对象的信息。到目前为止，你看到所有对象的例子都没有结构。然而，就像自定义模板赋予一个规则模式以结构，槽给出了对象的结构。对于自定义模板和对象，槽作为一个被命名的位置用来存储数据。然而，不同于自定义模板的槽，对象通过类来获取它们的槽，类应用了继承。这样，对象槽的信息就可以通过对象的

子类来高效的被继承了。无界(unbound)槽没有分配槽值。所有的槽都是有界的。

作为一个简单的例子，让我们来定义一个对象，通过槽来存储人的信息，并向它发送消息。下面的命令将首先设置 CLIPS 环境到恰当的结构。Sound 槽和 age 槽起初并没有包含任何数据，如 nil 值。

```
CLIPS>(clear)
CLIPS>(defclass DUCK (is-a USER)
        (slot sound)
        (slot age))
CLIPS>(definstances DORK_OBJECTS
        (Dorky_Duck of DUCK))
CLIPS>(reset)
CLIPS>(send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound nil)
(age nil)
CLIPS>(send [Dorky_Duck] put-sound quack)
quack
CLIPS>(send [Dorky_Duck] print)
[Dorky_Duck] of Duck
(sound quack)
(age nil)
CLIPS>
```

上面的槽以类中定义的顺序被打印出来。然而，如果实例从多个类中继承槽，来自于通用类的槽则将会优先打印。

可以通过 put-message 来改变槽值。默认状态下，CLIPS 为每个类的槽创建了一个 put-句柄来获取 put-message。注意 put 后面的破折号，它是该消息句法的重要组成部分，因为它将 put 和槽名分开。(send)函数里只能包含一个“put-”。因此，如果要改变多槽或一些实例的相同槽时，你必须发送多个消息。你可以通过写一个多发送函数，或使用 modify-instance 函数来代替上面的手动操作。

在 make-instance 中，可以通过 slot-override 来设置槽值。示例如下：

```
CLIPS>(make-instance Dixie_Duck of DUCK (sound quack) (age 2))
[Dixie_Duck]
CLIPS>(send [Dixie_Duck] print)
[Dixie_Duck] of DUCK
(sound quack)
(age 2)
CLIPS>
```

“Put-”的补充消息是“get-”，用来获取槽中的数据，将会在下一个例子中体现出来。当 put-成功执行后将会返回新的值，当 get-被成功执行时，将返回恰当的值。如果 put-和 get-没有成功执行时，将会返回一个出错消息。下面的例子将演示它们的用法。

```
CLIPS>(send [Dorky_Duck] put-color white)        ;没有 color 槽
[MSGFUN1] No applicable primary message-handlers found for put-color.
FALSE
CLIPS>(send [Dorky_Duck] get-age)
nil
CLIPS>(send [Dorky_Duck] put-age 1)              ;age 获取值
```

```
1
CLIPS>(send [Dorky_Duck] get-age)                ;验证值为正确
1
CLIPS>
```

对比与 put-消息，get-消息返回一个槽的槽值。get-的值被返回后，可以通过使用其他的函数，分配其一个变量等等。相比之下，一个被打印出来的值不能被用于其他函数，分配变量等等，因为该值已发送到标准输出设备了。一个规避该问题的方法是将值打印到一个文件，然后读出该文件内的数据。虽然这不是一个很好的解决方案，当它确实能解决问题。另外一个方法就是重新写一个打印消息句柄(print message-handler)，同样返回值。

关于槽的一个重要特点是，不能通过增加，删除或改变槽的特性来修改一个类的槽。唯一改变类的方法是(1)删除类的所有实例,(2)使用(defclass)来定义一个相同名字的类并所想要的槽。这个方法类似于通过载入一个相同名字的新的规则来修改规则。

## · 类程式

现在，你已经学习了槽和实例，下面让我们来熟悉类程式(class etiquette)。这里的术语——规则，指的是一套做事情的指导方针。

与标准的程序设计相比，OOP 范式是类导向(class oriented)的。每个对象都与类有内在的联系，这些类是类等级的一部分。OOP 程序员关注的是所有的类或类的结构层次(class architecture)和对象之间消息的传递，而不是首先考虑执行。因此，在普通的程序设计语言中，执行是显式表示的，而在 OOP 中为隐式。两种方式的结果却是相同的。不过，OOP 系统能更简单的被验证和维护。

正确使用类的方法总结在下面三条规则中了：

*类程式用法*

1. 类的级别分层必须使用 is-a 链接而被指定逻辑的增加。
2. 如果一个类只包含一个实例，则这个类就是多余的。
3. 类的命名不能是实例名，反之亦然。

以上第一条规则限制了你的程序中单个类的创建。如果没有该限制，那么你也许根本就不需要 OOP。随着类被创建的递增，你可以更简单的验证和维护你的代码。另外，增加的类链能更容易的被添加到类库中，以更大程度的方便促进新代码的生成。类库的概念与执行子程序库非常类似。只有 is-a 链接能被使用，因为它是 CLIPS6.3 版本提供的唯一联系了。

第二条规则促进了以类作为一个模板去产生多个同类对象方式的实现，当然，你也可以从零或一个实例开始去创建。然而，如果你只会用到一个类中的一个实例，那么你可以考虑修改它的超类以适应该实例，而不要定义一个新的子类。如果你所有的类都仅有一个实例，那么你的程序也许过于简单，用 OOP 不能很好体现它的长处，你的代码用程序化语言写或许更好。

第三条规则的意思是，类不能命名为一个实例的名，反之亦然，以消除歧义。

# 其他特性

这里还有许多关于槽的有用函数。如果你使用这些函数去测试函数值的合法性，那么你的程序将会更严密。通常一个程序如果不是返回 TRUE 就是返回 FALSE。

| 函数 | 作用 |
| --- | --- |
| class-slot-existp | 如果类槽存在，则返回 TRUE |
| slot-existp | 如果实例槽存在，则返回 |

TRUE

slot-boundp 如果指定的槽包含一个值，则返回 TRUE

instance-address 返回指定存储实例的物理地址

instance-name 返回给定地址的名，反之亦然

instancep 如果它的参数是实例，则返回 TRUE

instance-addressp 如果参数是实例的地址，则返回 TRUE

instance-namep 如果参数是实例名，则返回 TRUE

list-definstances 输出所有自定义实例

ppdefinstances 格式打印自定义实例

watch instances 允许查看和删除被创建的实例

unwatch instances 关闭查看实例

save-instances 保存实例到文件

load-instances 加载实例到文件

undefinstances 删除命名的自定义实例

# · 第十章 奇妙的槽面

**如果你想拥有类，那么就像是和老朋友一样，去互动，去尝试对话。**

在本章中，你将学到许多关于槽和利用一些槽面 (facets)来指定它们属性的方法。就像槽用来描述实例一样，槽面用来描述槽。槽面的使用是非常好的软件工程技术，它对于 CLIPS 而言避免了非法值的插入，从而降低

了运行时错误或崩溃的可能性。有许多种槽面用来指定槽，总结在下面的表格中。

| 槽面名 | 含义 |
| --- | --- |
| **default and default-dynamic** | 为槽设定初始值 |
| **cardinality** | 多槽的个数 |
| **access type** | 读-写，只读，只初始化访问 |
| **storage** | 实例中的本地槽或类中的共享槽 |
| **propagation** | 继承或非继承槽 |
| **source** | 复合的或独占式继承 |
| **documentation** | 文件或槽 |
| **override-message** | 消除槽的显示信息 |
| **create-accessor** | 创建 put-和 get-句柄 |
| **visibility** | 仅公有或私有的定义类 |
| **reactive** | 槽触发器模式匹配的改变 |

由于版面的关系，本章中，我们将仅详细讲解上面诸多槽面中的一小部分。了解更多的细节，可以参看 CLIPS 参考指南(CLIPS Reference Manual).

# Default 槽

当一个实例被创建或初始化后，default facet 用来设置槽的默认值，如下例所示：

```
CLIPS>(clear)
CLIPS>(defclass DUCK (is-a USER)
      (slot sound (default quack))
      (slot ID)
      (slot sex (default male)))
```

```
CLIPS>(make-instance Dorky_Duck of DUCK)
[Dorky_Duck]
CLIPS>(send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack)
(ID nil)
(sex male)
CLIPS>
```

正如你所见，sex 槽和 sound 槽通过 default facet 设定了默认值。通过 default 关键字，任意有效的 CLIPS 表达式都可以不用包含一个变量值。举个例子，sound 槽的默认表达式为字符 quack。槽面表达(facet expression)也可以用到函数，这将在下一个例子中呈现。

Default 槽面是静态的默认(static default)，因为槽面表达式的值在类被定义且无任何改变，除非被重新定义时已经被限定了。举个例子，我们来设定 ID 槽的默认值为 gensym*函数，该函数用来返回一个新值，该值不为系统每次调用。

```
CLIPS>(clear)
CLIPS>(defclass DUCK (is-a USER)
        (slot sound (default quack))
        (slot ID (default (gensym*)))
        (slot sex (default male)))
CLIPS>(make-instance [Dorky_Duck] of DUCK)      ;Dorky_Duck #1
[Dorky_Duck]
CLIPS>(send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
```

```
(sound quack)
(ID gen1)
(sex male)
CLIPS>(make-instance [Dorky_Duck] of DUCK)      ; Dorky_Duck #2
[Dorky_Duck]
CLIPS>(send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack)
(ID gen1)
(sex male)
CLIPS>
```

如你所见，ID 一直保持为 gen1，这是因为(gensym*)仅被定值一次，即使在第二个实例被创建时，也不会重新定值。注意，如果你已经调用过(gensym*), 那么(gensym*)值在你的电脑里显示可能不是如此，因为它每次被调用后会自动递增，并不会被(clear)命令消解。只有当你重启 CLIPS 时，(gensym*)函数才会被重新设置到初始值。

现在，假设我们想留意一下已经创建的 Dorky_Duck 实例的改变。与其使用静态默认(static default)，我们可以用使用默认动态(default dynamic)槽面，它可以在每一个新实例被创建时，为槽面表达式定值。注意下面一个例子，看看与前面的例子有什么不同。

```
CLIPS>(clear)
CLIPS>(defclass DUCK (is-a USER)
        (slot sound (default quack))
        (slot ID (default-dynamic (gensym*)))
        (slot sex (default male)))
```

```
CLIPS>(make-instance [Dorky_Duck] of DUCK)      ; Dorky_Duck #1
[Dorky_Duck]
CLIPS>(send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack)
(ID gen2)
(sex male)
CLIPS>(make-instance [Dorky_Duck] of DUCK)       ;Dorky_Duck #2
[Dorky_Duck]
CLIPS>(send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack)
(ID gen3)                              ;注意这里的 ID 不同于 Dorky_Duck #1
(sex male)
CLIPS>
```

上面的例子中使用了动态默认(dynamic default)，第二个实例的 ID 是 gen3，与第一个实例的 gen2 不同。比较而言，前一个使用静态默认(static default)的例子中，ID 的值都是相同的 gen1，因为在类被定义，(gensym*)只会被定值一次，而在后者动态默认(dynamic default)中每个新实例被创建时，(gensym*)定值自动增加。

## · 重要特性

槽的集合引用两种类型字段之一，这两种类型是：（1）单字段，（2）多字段。术语 cardinality 指代一个集合。有界单字段槽仅包含一个字段，有界多字段槽可包含零或多个字段。每个有界单字段槽和多字段槽包含一个值，然而，一个多字段值也许包含多个字段。举个例子，(a b c)是一个单一的多字段值，该字段值包含三个字段。空字符串“”是一个单字段值，就像“a

b c”一样。如此相反，一个无界槽没有值。

与单、多字段变量相似，请将槽想像成你的信箱，你有时会收到单独的一封没有信封的垃圾邮件，地址标签却被粘贴在一叠纸上，上面写好了寄往 “住址”。其他时候，你也许会在信箱里找到一封上面写多个地址的信封。没有信封的单张垃圾邮件就像是单字段值，多个地址的信封就像是多字段值。如果垃圾邮件发送者发送给你一封信，里面什么都没有，那么这可对应一个空多槽字段变量。（试想，如果垃圾邮件是空的，那么你是否算得上收到了垃圾邮件？）

多槽面(multiple facet)关键字为 mulitslot，被用来存储多字段变量，如下例所示：

```
CLIPS>(clear)
CLIPS>(defclass DUCK (is-a USER)
      (multislot sound (default quack quack)))
CLIPS>(make-instance [Dorky_Duck] of DUCK)
[Dorky_Duck]
CLIPS>(send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack quack)
CLIPS>
```

多字段值可以被利用来使用 get-和 put-，如下例所示，下面例子显示了怎样联系 quacks。

```
CLIPS>(send [Dorky_Duck] put-sound quack1 quack2 quack3)
(quack1 quack2 quack3)
CLIPS>(send [Dorky_Duck] get-sound)
```

```
(quack1 quack2 quack3)
CLIPS>
```

标准 CLIPS 函数，如 nth$，被用来获取多字段值的第 n 个字段。下面的例子显示了怎样选取一个确定的 quack。

```
CLIPS>(nth$ 1 (send [Dorky_Duck] get-sound))
quack1
CLIPS>(nth$ 2 (send [Dorky_Duck] get-sound))
quack2
CLIPS>(nth$ 3 (send [Dorky_Duck] get-sound))
quack3
CLIPS>
```