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：

$$-2^{N-1} \cdots 2^{N-1}-1$$

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。

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
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。**

理想的单身汉

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 STRING1 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)
```