4. 元语言抽象(3)

本节讨论

- 逻辑程序设计
- 逻辑程序设计语言的实现
- 模式匹配和合一（unification）
- 否定问题
- 查询和数据库
- 逻辑编程语言和逻辑的关系

逻辑程序设计

- 前面说过：数学处理说明式知识，计算机科学处理命令式知识
  - 程序语言要求用算法的方式描述解决问题过程
  - 实际上，它们也提供了一些说明性描述方式，使用户可以忽略计算过程的很多细节。例如输出函数的格式描述
- 多数程序语言要求用定义数学函数的方式组织程序
  - 被描述的计算有明确方向，从输入到输出
  - 描述函数关系的表达式也给出了一种计算结果的方法
  - 写出的程序描述“怎么做”
- 也有例外。前面的例子：
  - 约束传递系统中的计算对象是约束关系，没有明确的计算方向和顺序，其基础系统要做很多工作来支持计算
  - 非确定性程序求值器里的表达式可有多个值，求值器设法根据表达式描述的关系找出满足要求的值

逻辑程序设计

- 下面讨论的逻辑程序设计可看作上述想法的推广
  - 它基于关系模型和一种称为合一的重要操作
  - 在这里编程就是用逻辑公式描述事物之间的约束关系（属于“是什么”的范畴），支持多重结果和无确定方向的计算
- 逻辑程序设计特别适合一些应用领域的需要
  - 前面的自然语言语法分析是一个例子
  - 实际中逻辑程序设计已被用于许多领域。典型的例子是做数据库查询的 Datalog 语言，可以方便地描述复杂的查询，以及基于已有事实的隐含事实等
  - 下面会看到更多例子

逻辑程序设计

- 有关“是什么”的一个描述可能蕴涵许多“怎样做”的过程。考虑过程 append，它描述的是如何从表 x 和 y 算出它们的拼接：

```
(define (append x y)
  (if (null? x)
      y
      (cons (car x) (append (cdr x) y))))
```

- 可认为这个程序写的是两条计算规则：

  对任何一个表 y，空表与其拼接得到的表就是 y 本身

  对任何表 u, v, y, z，(cons u v) 与 y 拼接得到 (cons u z) 的条件是 v 与 y 的 append 是 z
- 上面 append 过程定义和两条规则都可以回答下面问题：

  找出 (a b) 和 (c d) 的 append
- 然而上面两条规则还可以回答下面问题（append 过程不行）：

  找出一个表 y 使 (a b) 与它的拼接能得到 (a b c d)

  找出所有拼接起来能得到 (a b c d) 的表 x 和 y

逻辑程序设计

- 在逻辑式程序语言里可以写出与上面两条规则直接对应的表达式，求值器可以基于它得到上述几个问题的解
- 各种逻辑语言（包括下面介绍的）都有缺陷，简单提供“做什么”知识有时会使求值器陷入无穷循环或产生非用户希望的行为
- 本领域还需要深入研究，新方向是 constraint programming（CP）
- 下面研究一个逻辑语言的解释器
  - 该语言称为查询语言，在描述数据库中信息查询方面很有用（存在实际的演绎数据库，Datalog 是一种影响很广的逻辑查询语言）
  - 该语言与 Scheme 大相径庭，但前面的框架仍适用：基本元素、组合手段和抽象手段（这里的抽象机制是规则）
  - 解释器比 Scheme 解释器复杂，但仍有许多类似元素：“求值”部分分类处理各类表达式，“应用”部分实现语言中的抽象机制
  - 这里用了一种框架结构作为核心数据结构，还用到流技术

演绎信息检索

- 逻辑式编程语言特别适合用作数据库接口，完成复杂的信息检索
  - 查询语言就是为此设计的，先用一个实例展示逻辑式编程的使用
- 设 Microshaft 公司（在波士顿的高科技公司）需要人事数据库，逻辑式语言不仅能做数据导向的信息访问，还能基于已有数据做推理
- 数据库内容是有关公司人事的断言，有许多描述各种事实的断言

  Ben 是公司的计算机专家，关于他的断言如下

  (address (Bitdiddle Ben) (Slumerville (Ridge Road) 10))

  (job (Bitdiddle Ben) (computer wizard))

  (salary (Bitdiddle Ben) 60000)

  每个断言在形式上是一个表，其元素还可以是表

  一个断言描述一个客观事实

演绎信息检索

- Ben 管理公司的计算机分部，下属两个程序员和一个计算机技师：

  (address (Hacker Alyssa P) (Cambridge (Mass Ave) 78))
  (job (Hacker Alyssa P) (computer programmer))
  (salary (Hacker Alyssa P) 40000)
  (supervisor (Hacker Alyssa P) (Bitdiddle Ben))
  (address (Fect Cy D) (Cambridge (Ames Street) 3))
  (job (Fect Cy D) (computer programmer))
  (salary (Fect Cy D) 35000)
  (supervisor (Fect Cy D) (Bitdiddle Ben))
  (address (Tweakit Lem E) (Boston (Bay State Road) 22))
  (job (Tweakit Lem E) (computer technician))
  (salary (Tweakit Lem E) 25000)
  (supervisor (Tweakit Lem E) (Bitdiddle Ben))

  Hacker Alyssa 管着一个实习程序员：

  (address (Reasoner Louis) (Slumerville (Pine Tree Road) 80))
  (job (Reasoner Louis) (computer programmer trainee))
  (salary (Reasoner Louis) 30000)
  (supervisor (Reasoner Louis) (Hacker Alyssa P))

  计算机分部所有人员职务的第一个符号都是 computer

演绎信息检索

- Ben 是公司的高级雇员，其上司是公司大老板 Oliver

  (supervisor (Bitdiddle Ben) (Warbucks Oliver))
  (address (Warbucks Oliver) (Swellesley (Top Heap Road)))
  (job (Warbucks Oliver) (administration big wheel))
  (salary (Warbucks Oliver) 150000)

- 公司有一个财务分部，人员包括一个主管会计和一个助手

  (address (Scrooge Eben) (Weston (Shady Lane) 10))
  (job (Scrooge Eben) (accounting chief accountant))
  (salary (Scrooge Eben) 75000)
  (supervisor (Scrooge Eben) (Warbucks Oliver))
  (address (Cratchet Robert) (Allston (N Harvard Street) 16))
  (job (Cratchet Robert) (accounting scrivener))
  (salary (Cratchet Robert) 18000)
  (supervisor (Cratchet Robert) (Scrooge Eben))

演绎信息检索

- 老板有一个秘书

  (address (Aull DeWitt) (Slumerville (Onion Square) 5))
  (job (Aull DeWitt) (administration secretary))
  (salary (Aull DeWitt) 25000)
  (supervisor (Aull DeWitt) (Warbucks Oliver))

- 数据库里还有一些断言说明各种人能从事的工作之间的关系

  计算机专家可以做程序员和技师的工作：

  (can-do-job (computer wizard) (computer programmer))
  (can-do-job (computer wizard) (computer technician))

  程序员可以做实习程序员的工作：

  (can-do-job (computer programmer)
  (computer programmer trainee))

  秘书可以做老板的工作

  (can-do-job (administration secretary)
  (administration big wheel))

简单查询

- 要查询数据库里的信息，只需在提示符下输入查询。如：

  ;;; Query input:
  (job ?x (computer programmer))

  系统的响应：

  ;;; Query results:
  (job (Hacker Alyssa P) (computer programmer))
  (job (Fect Cy D) (computer programmer))

- 查询语句描述要查询信息的模式，其中有些项是具体信息；问号开头的模式变量项（上面 ?x）可与任何东西匹配。系统响应查询，给出数据库里能与查询模式匹配的所有条目
- 需要区分多个匹配和同一匹配的多次出现，因此模式变量需要名字：

  (address ?x ?y)

  这导致系统列出所有雇员的地址条目

- 如果查询中没有变量，就相当于问相应事实是否存在

简单查询

- 同一模式变量可在一个查询里出现多次，表示需要同一匹配。如：

  (supervisor ?x ?x)

  要求给出所有自己管自己的雇员的条目（这里没有）

- 例，列出所有从事计算机工作的雇员：

  (job ?x (computer ?type))

  系统响应是：

  (job (Bitdiddle Ben) (computer wizard))
  (job (Hacker Alyssa P) (computer programmer))
  (job (Fect Cy D) (computer programmer))
  (job (Tweakit Lem E) (computer technician))

  它不匹配（由于 **?type** 只能匹配一个项）：

  (job (Reasoner Louis) (computer programmer trainee))

简单查询

- 如果希望匹配第一个元素是 **computer** 的所有条目，应写：

  (job ?x (computer . ?type))

  (computer . ?type) 能匹配 **(computer programmer trainee)**，也能匹配 **(computer technician)** 和 **(computer)**

- 总结一下系统对简单查询的处理：
  - 设法找出使查询语句中的模式变量满足查询模式的所有赋值。即，找出这些变量的所有可能指派（具体表达式），使得把模式中的变量代换为具体表达式后得到的条目在数据库里
  - 对查询的响应是列出数据库里所有满足模式的条目，用找到的所有可能赋值对查询模式实例化，显示得到的结果
  - 如果查询模式里无变量，就简化为对该查询是否出现在数据库里的检验。相应的赋值是空赋值

复合查询

- 简单查询是基本操作，可以在其基础上构造复合查询。查询语言的组合手段是连接词 and, or 和 not（不是 Scheme 内部操作）
- 对复合查询，系统也是设法找出所有能满足它的赋值，并显示用这些赋值实例化查询模式得到的结果
- and 复合的一般形式 (and <*query*$_1$> <*query*$_2$> ... <*query*$_n$>)，要求找到的变量赋值满足 <*query*$_1$> <*query*$_2$> ... <*query*$_n$> 中的每个查询
- 例：找出所有程序员的住址

```
(and (job ?person (computer programmer))
     (address ?person ?where))
```

系统的响应：

```
(and (job (Hacker Alyssa P) (computer programmer))
     (address (Hacker Alyssa P) (Cambridge (Mass Ave) 78)))
(and (job (Fect Cy D) (computer programmer))
     (address (Fect Cy D) (Cambridge (Ames Street) 3)))
```


复合查询

- or 查询的一般形式为 (or <*query*$_1$> <*query*$_2$> ... <*query*$_n$>)，它要求找出所有能满足 <*query*$_1$> <*query*$_2$> ... <*query*$_n$> 之一的赋值，给出用这些赋值实例化的结果
- 例如：

```
(or (supervisor ?x (Bitdiddle Ben))
    (supervisor ?x (Hacker Alyssa P)))
```

得到由 Ben Bitdiddle 或 Alyssa P. Hacker 管理的雇员名单：

```
(or (supervisor (Hacker Alyssa P) (Bitdiddle Ben))
    (supervisor (Hacker Alyssa P) (Hacker Alyssa P)))
(or (supervisor (Fect Cy D) (Bitdiddle Ben))
    (supervisor (Fect Cy D) (Hacker Alyssa P)))
(or (supervisor (Tweakit Lem E) (Bitdiddle Ben))
    (supervisor (Tweakit Lem E) (Hacker Alyssa P)))
(or (supervisor (Reasoner Louis) (Bitdiddle Ben))
    (supervisor (Reasoner Louis) (Hacker Alyssa P)))
```


复合查询

- (not <*query*>) 要求得到所有使 <*query*> 不成立的赋值
- 例：

```
(and (supervisor ?x (Bitdiddle Ben))
     (not (job ?x (computer programmer))))
```

要求找出 Ben 管的所有人中的非程序员

- 组合形式 (lisp-value <*predicate*> <*arg*$_1$> ... <*arg*$_n$>)，第一参数是一个 Lisp 谓词。要求将谓词作用于后面的参数（赋值后得到的值），选出使谓词为真的所有赋值。例如：

```
(and (salary ?person ?amount)
     (lisp-value > ?amount 30000))
```

选出所有工资高于 30000 的人

利用 lisp-value 可以很灵活地描述各种查询，利用查询的语义

规则

- 查询语言的抽象手段是建立规则
- 例：

```
(rule (lives-near ?person-1 ?person-2)
      (and (address ?person-1 (?town . ?rest-1))
           (address ?person-2 (?town . ?rest-2))
           (not (same ?person-1 ?person-2))))
```

语义：两个（不同的）人住得近，如果他们住在同一个 town

- “同一个”关系可以用规则定义：

```
(rule (same ?x ?x))
```

- 例：一个人是组织里的大人物，如果其管理的人还管别人：

```
(rule (wheel ?person)
      (and (supervisor ?middle-manager ?person)
           (supervisor ?x ?middle-manager)))
```

规则

- 规则的一般形式：

  (rule <*conclusion*> <*body*>)

  其中 **<*conclusion*>** 是模式，**<*body*>** 是任何形式的查询

  可以认为一条规则表示了很大（甚至无穷大）的一个断言集，其元素是由 **<*conclusion*>** 求出的所有满足 **<*body*>** 的赋值

- 对简单查询，如果其中变量的某个赋值满足某查询模式，那么用这个赋值实例化模式得到的断言一定在数据库里

  但满足规则的断言不一定实际存在在数据库里（推导出的事实）

- 查询实例：找出所有住在 **Bitdiddle Ben** 附近的雇员

  (lives-near ?x (Bitdiddle Ben))

  得到

  (lives-near (Reasoner Louis) (Bitdiddle Ben))
  (lives-near (Aull DeWitt) (Bitdiddle Ben))

规则

- 查询实例：找出所有住在 **Bitdiddle Ben** 附近的程序员

  (and (job ?x (computer programmer))
  (lives-near ?x (Bitdiddle Ben)))

- 与复合过程类似，已定义的规则可以用于定义新规则。例如：

  (rule (outranked-by ?staff-person ?boss)
  (or (supervisor ?staff-person ?boss)
  (and (supervisor ?staff-person ?middle-manager)
  (outranked-by ?middle-manager ?boss))))

  这是个递归定义的规则：一个职员是某老板的下级，如果该老板是其主管，或者（递归的）其主管是该老板的下级

把逻辑看作程序

- 规则可看作逻辑蕴涵式：如果对所有模式变量的赋值能满足一条规则的体，那么它满足相应结论。可以认为查询语言是基于规则做逻辑推理
- 考虑 append 的例子，描述它的规则说：

  对任何表 y，空表与它 append 得到的就是 y 本身

  对任何表 u, v, y, z，(cons u v) 与 y 的 append 是 (cons u z) 的条件是 v 与 y 的 append 是 z

- 用查询语言描述，需要描述关系 (append-to-form x y z)，直观解释是“x 和 y 的拼接得到 z”。用规则定义是：

  (rule (append-to-form () ?y ?y))
  (rule (append-to-form (?u . ?v) ?y (?u . ?z))
  (append-to-form ?v ?y ?z))

  第一条规则没有体，说明它对任何 y 成立

  第二条规是递归定义的。注意，这里用了表的点号形式

把逻辑看作程序

- 有了上面有关 append 的规则，可以做许多查询。实例：

  ;;; Query input:
  (append-to-form (a b) (c d) ?z)
  ;;; Query results:
  (append-to-form (a b) (c d) (a b c d))

  ;;; Query input:
  (append-to-form (a b) ?y (a b c d))
  ;;; Query results:
  (append-to-form (a b) (c d) (a b c d))

  ;;; Query input:
  (append-to-form ?x ?y (a b c d))
  ;;; Query results:
  (append-to-form () (a b c d) (a b c d))
  (append-to-form (a) (b c d) (a b c d))
  (append-to-form (a b) (c d) (a b c d))
  (append-to-form (a b c) (d) (a b c d))
  (append-to-form (a b c d) () (a b c d))

  这些例子展示了不同方向的计算，正是前面提出希望解决的问题

  虽然具体到 append，系统行为很令人满意，一般情况下未必如此（下面有讨论）

查询系统

- 现在讨论查询系统求值器的原理，以及与实现细节无关的一般结构
- 还要提出该求值器的局限性，说明这一查询系统里的逻辑运算与数理逻辑里的运算之间存在微妙差异
- 显然查询求值器要搜索，设法将查询与数据库里的事实和规则匹配
  - 可参考 amb，将系统实现为一个非确定性程序（练习4.78）
  - 也可以借用流的概念控制搜索。下面采用后一技术
- 查询系统的组织围绕两个核心操作：模式匹配和合一
  - 模式匹配实现简单查询和复合查询，下面要考虑它与基于框架流组织的信息的集成
  - 合一是模式匹配的推广，用于实现规则

  最后讨论如何通过表达式的分情况处理，构造整个的查询解释器

查询系统：模式匹配

- 模式匹配器检查一个数据项是否与某个给定模式匹配
- 例如：数据表 ((a b) c (a b))

  与模式 (?x c ?x) 匹配，其中模式变量 ?x 约束于 (a b)

  与模式 (?x ?y ?z) 匹配，其中 ?x 和 ?z 都约束到 (a b)，?y 约束到 c

  与模式 ((?x ?y) c (?x ?y)) 匹配，其中的 ?x 约束到 a，?y 约束到 b

  与模式 (?x a ?y) 不匹配，因这个模式要求表中第二个元素必须是 a
- 模式匹配器以一个模式、一个数据和一个框架为输入（框架里包含一些模式变量约束）。它
  - 检查数据是否以某种方式与模式匹配，而且该匹配与框架里（已有的）约束相容（不矛盾）
  - 匹配成功时返回原框架的扩充，加入新确定的所有约束
  - 找不到匹配时失败

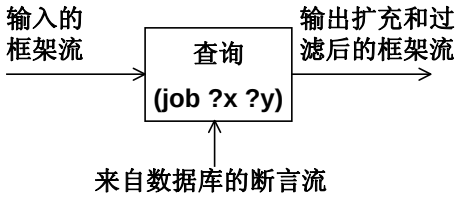
查询系统：模式匹配

- 例，要基于空框架用模式 (?x ?y ?x) 匹配 (a b a)
  - 匹配器返回的框架里 ?x 约束到 a，?y 约束到 b
  - 如果用同一模式、同一数据和包含 ?y 约束到的 a 框架去匹配，这个匹配将失败
  - 如果用同一模式、同一数据和包含 ?y 约束到的 b 框架去匹配，匹配器返回的框架扩充了?x 到 a 的约束
- 模式匹配器处理所有不涉及规则的查询。如，输入查询

  (job ?x (computer programmer))
  - 匹配器将基于空框架扫描数据库里的所有断言，选出其中与这个模式匹配的断言
  - 对每个成功的匹配，求值器都用匹配器返回的框架里 ?x的值实例化上述模式，得到最终结果

查询的框架流和简单查询

- 匹配器采用流的方式，基于给定框架做模式匹配：
  - 基于给定框架扫描数据库条目。对每个条目，或产生表示匹配失败的特殊符号，或给出原框架的一个扩充，匹配结果形成一个流
  - 用一个过滤器删除匹配失败信息，结果流里包含的框架都是原框架由于断言匹配而得到的扩充
- 一个查询以一个框架流作为输入，基于流中每个框架做上述匹配，合并产生的所有的流，得到作为查询结果的输出流
- 回答简单查询时，初始输入流里只有一个空框架，得到的流包含这一空框架的所有扩充。用这个流实例化查询模式，就能得到所有输出



复合查询：and

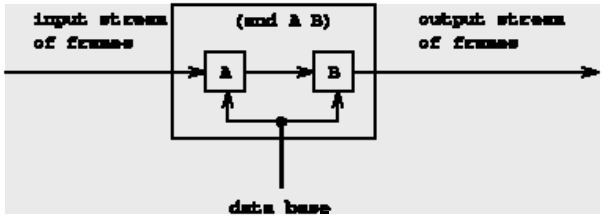
- 处理复合查询时利用了匹配器带着框架去检查匹配的功能。例：

  (and (can-do-job ?x (computer programmer trainee))
  (job ?person ?x))

  先找出与模式 (can-do-job ?x (computer programmer trainee)) 匹配的框架流，其中每个框架都包含对 ?x 的约束项。再找所有与模式 (job ?person ?x) 匹配的项，其匹配与给定的 ?x 匹配一致

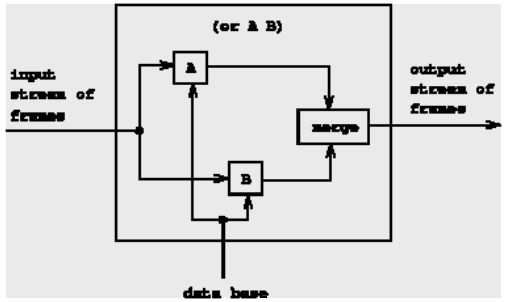
  作为结果的流中各框架都包含了 ?person 和 ?x 的约束

- 右图显示了 and 查询的处理过程。框架流顺序地通过两个查询，最终得到结果流



复合查询：or

- 两个查询的 or 是两个查询分别得到的框架流的归并
- 归并可以采用交错方式



- 注意这样处理复合查询的效率问题：
  - 一步查询中，对输入流里的每个框架都可能产生多个框架，一系列 and 查询里的每个查询都是从前一个查询得到框架流
  - 这使 and 查询中可能的匹配次数是查询个数的指数函数

  处理较简单的查询时这种方法很实用，但不适合处理很复杂的查询

复合查询：not 和 lisp-value

- 查询 q 的 not 就像框架流的过滤器，它删除流中所有满足 q 的框架
- 例如对模式 (not (job ?x (computer programmer)))
  - 设法对框架生成满足 (job ?x (computer programmer) 的扩充
  - 如果一个框架能扩充就丢掉它。不能产生扩充的留在输出流里
- 例如，查询

  (and (supervisor ?x ?y)
  (not (job ?x (computer programmer))))

  and 的第一个子句生成一批带有 ?x 和 ?y 的约束的框架

  后面的 not 子句删除所有使 ?x 的工作是程序员的框架
- 特殊查询形式 lisp-value 也实现为框架流的过滤器：
  - 用流中框架实例化模式的变量
  - 对实例化结果应用给定谓词，删去不满足谓词的框架

合一

- 在处理规则时，要找出其结论与被处理查询模式匹配的所有规则
  - 结论的形式像断言，但它可以包含变量
  - 现在匹配的两边（查询模式和规则的结论）都可以有变量，模式匹配不能处理这个问题（它只允许一方有变量）
- 合一是模式匹配的扩充，它能判断两个模式之间能否匹配。它
  - 设法确定是否存在一组变量赋值，使得这两个模式经过赋值的实例化后得到的表达式相同
  - 合一成功时返回得到的赋值（框架），否则返回失败信息
- 例：对 (?x a ?y) 和 (?y ?z a) 的合一操作将产生一个框架，在框架里 ?x, ?y 和 ?z 都约束到 a
  - 对 (?x ?y a) 和 (?x b ?y) 的合一将会失败，因为对 ?y 的任何赋值都不能使两个模式相同（根据模式的第二个元素 ?y 应约束到 b；然而根据它们的第三个元素 ?y 又应约束到 a）

合一

- 合一算法是查询系统的难点。复杂模式的合一好像需要做推理
- 例，合一 (?x ?x) 和 ((a ?y c) (a b ?z))，可以得到联立方程：

  ?x = (a ?y c)

  ?x = (a b ?z)

  它等价于 (a ?y c) = (a b ?z)

  它蕴涵 a = a, ?y = b, c = ?z

  由此又可得：?x = (a b c)
- 最一般的合一确实需要解方程。但这里情况简单，可以直接处理
- 模式匹配成功将给所有变量赋值，前面例子都是赋值为常量的情况
  - 但成功的合一可能产生变量值不能完全确定的情况
  - 可能有未约束的变量，或有变量约束的值还含有变量

规则的应用

- 例：考虑 (?x a) 和 ((b ?y) ?z)
  - 合一得赋值 ?x = (b ?y)，a = ?z，但无法确定 ?x 和 ?y 的值。这时也认为成功，因为 ?x 和 ?y 的赋值可以使两个模式一样
  - 这个匹配没限制 ?y 取值，但 ?x 必须是 (b ?y)。应把 ?x 到 (b ?y) 的约束放入框架。如果后来 ?y 值确定，?x 也要引用相应的值
- 例：假设要处理 (lives-near ?x (Hacker Alyssa P))

  先用模式匹配到数据库里找匹配断言（找不到）。再做与规则结论的合一，发现它与下面规则合一成功

```
(rule (lives-near ?person-1 ?person-2)
      (and (address ?person-1 (?town . ?rest-1))
           (address ?person-2 (?town . ?rest-2))
           (not (same ?person-1 ?person-2))))
```

  得到 ?person-2 约束到 (Hacker Alyssa P)，?x 约束到 ?person-1

  基于此框架对规则体的复合查询求值。匹配成功时 ?person-1 将建立约束，从而也给 ?x 建立了约束

规则的应用：一般工作过程

- 当求值器试图基于一个框架完成对某查询模式的匹配时，设法应用一条规则的过程是：
  - 将查询模式与规则结论合一，成功时形成原框架的一个扩充
  - 基于这样扩充的框架去求值该规则的体（是一个查询）
- 这一做法很像 Lisp 的 eval/apply 求值器中的过程应用：
  - 将过程的形式参数约束于实际参数值，用得到的框架扩充原环境
  - 基于这样扩充的环境去求值过程体
- 这种相似性也很自然：
  - 过程定义是 Lisp 里的抽象手段
  - 规则定义是查询语言里的抽象手段
  - 无论是应用过程还是应用规则，都需要打开相关的抽象，方法就是建立相应约束，而后基于它们去求值过程或规则的体

简单查询

- 使用规则和断言求值简单查询的完整过程：
  - 给定一个查询模式和一个框架流，对流中每个框架产生两个流：
    - 模式匹配器用给定模式与数据库断言匹配，得到扩充框架的流
    - 合一器应用所有可用规则，得到另一个扩充框架的流
    - 归并得到与原框架相容的满足给定模式的所有扩充框架的流
  - 把处理给定框架流里各个框架得到的流组合为一个流，其中包含了由输入流中各框架扩充而得到的与给定模式匹配的所有结果
- 系统很像一般语言的求值器，只是匹配操作比较复杂
- 协调各种匹配操作的过程是 qeval，它扮演着类似 eval 的角色
  - qeval 的参数是一个查询和一个框架流，结果是一个框架流，其中包含所有成功匹配得到的扩充框架
  - qeval 也根据查询的类型分情况处理，将请求分派到对应过程（简单查询，and，or，not，lisp-value）

查询求值器和驱动循环

- 驱动循环由终端取得输入
  - 用得到的查询和只包含一个空框架的流调用 qeval
  - 用 qeval 返回的流中的每个框架去实例化原查询
  - 最后打印出实例化的结果
- 驱动循环还检查特殊命令 assert!。该命令说明输入不是查询，而是一条应加入数据库的断言或规则。
- 例子：

```
(assert! (job (Bitdiddle Ben) (computer wizard)))

(assert! (rule (wheel ?person)
               (and (supervisor ?middle-manager ?person)
                    (supervisor ?x ?middle-manager))))
```


逻辑程序设计和数理逻辑

- 查询语言里的组合符对应逻辑连接词，查询的做法看起来具有逻辑可靠性（and 查询要经过两个子成分处理，等等）
- 但这种对应关系并不严格，因为查询语言的基础是求值器，其中隐含着控制结构，用过程性的方式解释逻辑语句
- 我们可以利用这种控制结构。如要找程序员的上司，下面写法都行：

```
(and (job ?x (computer programmer))
     (supervisor ?x ?y))

(and (supervisor ?x ?y)
     (job ?x (computer programmer)))
```

  如果公司里的上司比程序员多，第一种写法的查询效率更高
- 逻辑程序设计希望开发一种技术，把计算问题划分为两个相互独立的子问题：需要计算“什么”，“怎样”计算。途径是：

  找出逻辑语言的一个子集，其功能足够强，足以描述要考虑的计算；又足够弱，有可能为它定义一种过程式的解释

逻辑程序设计和数理逻辑

- 上述两方面性质保证逻辑程序设计语言程序的有效性，因为它们可以由计算机执行。具体控制交给了语言背后的求值器
- 这样，如果很好安排子句的顺序和各子句中子目标的顺序，有可能得到更高效的计算，计算结果就是一些逻辑断言和规则的推论
- 前面描述的查询语言是上面想法的一个具体实施：
  - 查询语言是数理逻辑的一个可以过程式解释的子集
  - 一个断言描述了一个简单事实；一条规则表示一个蕴涵，所有使规则体成立的情况都使结论成立
  - 规则有自然的过程式解释：要得到其结论，只需确定其体成立

  这样，一组规则实际上描述了一个计算过程
- 由于规则是逻辑语句，有逻辑解释。因此可以做些逻辑工作：
  - 检查逻辑推理是否总能得到同样的结果
  - 如果确实如此，就确认了求值器的“可靠性”（或说“正当性”）

无穷循环

- 由于逻辑式程序采用过程性解释，因此可能写出极度低效的程序，极端情况是写出的程序使推导陷入无穷循环
- 例，假定现在要设计一个有关著名婚姻的数据库，加入断言

```
(assert! (married Minnie Mickey))
```

- 查询

```
(married Mickey ?who)
```

- 得不到结果，因系统不知道婚姻是相互的（对称的）。如果加入规则

```
(assert! (rule (married ?x ?y)
               (married ?y ?x)))
```

- 再查询时系统无穷循环：在这条规则产生的框架里 ?x 约束到 Mickey，?y 约束到 ?who；规则体要求基于框架匹配 (married ?who Mickey)
- 显然这一查询与事实匹配，也与上面规则匹配。由规则体得到的查询还是 (married Mickey ?who)，系统进入无穷循环

无穷循环，not问题

- 系统能不能在进入无穷循环前找到匹配的断言，依赖于查询过程的实现细节。上面例子里系统找到了 (married Minnie Mickey)
- 另外，一组相关规则也可能导致无穷循环，练习 4.64 说明 and 中各子句的顺序可能导致无穷循环。具体情况也依赖于实现细节
- 查询系统的另一问题与 not 有关。对前面数据库做下面两个查询：

```
(and (supervisor ?x ?y)
     (not (job ?x (computer programmer))))
(and (not (job ?x (computer programmer)))
     (supervisor ?x ?y))
```

- 与逻辑里的情况不同，这两个查询会得到不同结果
  - 第一个查询找出所有与 (supervisor ?x ?y) 匹配的条目，从得到的框架中删去 ?x 满足 (job ?x (computer programmer)) 的框架
  - 第二个查询从初始框架流（只含一个空框架）开始检查能否扩展出与 (job ?x (computer programmer)) 匹配的框架。显然空框架可扩展，not 删除流中的空框架得到空流，查询最后返回空流

not 的问题

- 出问题的原因是对 not 的解释。这里把 not 模式看作一种过滤器。如果 not 子句作用到框架时在模式里存在未约束的变量，系统就会产生我们不希望的结果（如前面例子）
- lisp-value 也有类似问题。如果使用 lisp-value 的谓词时有些参数没有约束，系统显然无法正常工作
- 此外，查询语言里的 not 与逻辑里的 not 还有一个本质差异：
  - 逻辑里 not P 的意思是 P 不真
  - 查询系统里 not P 则是说 P 不能由数据库里的知识推导出来
- 从前面的人事数据库可以推导出许多 not 断言，例如：
  - Ben Bitdiddle 不喜欢打篮球
  - 外面没有下雨
  - 2 + 2 不等于 4，等等
- 逻辑程序语言里的 not 反映的是一种“封闭世界假说”，它认为所有知识都包含在数据库里，凡是没有的东西，其 not 都成立

实现，驱动循环和实例化

- 分析了与查询语言相关的各种问题之后，现在考虑系统的实现
- 先考虑驱动循环和实例化
  - 查询系统的驱动循环反复读输入表达式
    - 如果是断言或规则，就把相关信息加入数据库
    - 否则认为是查询，送给 qeval，并送去只包含一个空框架的流
  - 求值查询得到一个框架流
    - 各框架里的项说明模式中变量的约束值
    - 用框架流中的框架对模式做实例化，得到实例化结果的流
  - 最后输出流中的各项，这是一些简单或复合的断言

驱动循环和实例化

```
(define input-prompt ";;; Query input:")
(define output-prompt ";;; Query results:")
(define (query-driver-loop)
  (prompt-for-input input-prompt)
  (let ((q (query-syntax-process (read))))
    (cond ((assertion-to-be-added? q)
           (add-rule-or-assertion! (add-assertion-body q))
           (newline) (display "Assertion added to data base.")
           (query-driver-loop))
          (else
           (newline) (display output-prompt)
           (display-stream
            (stream-map
             (lambda (frame)
               (instantiate q
                            frame
                            (lambda (v f) (contract-question-mark v))))
             (qeval q (singleton-stream '()))))
           (query-driver-loop)))))
```

要求加入断言或规则

迭代

用结果流中的框架做查询模式 q 的实例化

处理未约束变量，产生适当的输出形式

从包含一个空框架的流出发生成匹配的框架流

迭代

驱动循环和实例化

- 表达式是数据抽象，后面再考虑其语法和语法过程
- 处理输入表达式前将其变换为一种易处理形式，修改其中变量的表示。查询后打印前把未约束变量变回原形式（contract-question-mark）
- 实例化表达式时需要复制，用给定框架里的约束代换其中的变量，无约束变量用 instantiate 的参数 unbound-var-handler 处理：

```
(define (instantiate exp frame unbound-var-handler)
  (define (copy exp)
    (cond ((var? exp)
           (let ((binding (binding-in-frame exp frame)))
             (if binding
                 (copy (binding-value binding))
                 (unbound-var-handler exp frame))))
          ((pair? exp)
           (cons (copy (car exp)) (copy (cdr exp))))
          (else exp)))
  (copy exp))
```

使用 frame 里的约束构造 exp 的实例化副本

处理未约束变量，用由参数得到的过程处理

求值器

- qeval-driver-loop 调用基本求值过程 qeval
- qeval 是查询求值器的核心，其参数是一个查询模式和一个框架流。它返回扩充后的框架流
- qeval 用 type 识别各种特殊形式，基于 get 和 put 组织操作，根据类型完成数据导向的分派
- 任何非特殊形式的表达式都当作简单查询：

```
(define (qeval query frame-stream)
  (let ((qproc (get (type query) 'qeval)))
    (if qproc
        (qproc (contents query) frame-stream)
        (simple-query query frame-stream))))
```

如果找到特殊处理过程就用该过程处理

type 和 contents 是语法过程，后面定义

简单查询

- simple-query 处理简单查询，参数是一个模式和一个框架流。它逐个处理流中各框架：
  - find-assertions 找数据库里的匹配断言，生成扩充框架的流
  - apply-rules 应用可应用的规则，生成扩充框架的流
  - stream-append-delayed 组合上面两个流
  - stream-flatmap 把处理各框架得到的流合并为一个流（平坦化）

```
(define (simple-query query-pattern frame-stream)
  (stream-flatmap
   (lambda (frame)
     (stream-append-delayed
       (find-assertions query-pattern frame)
       (delay (apply-rules query-pattern frame))))
   frame-stream))
```

这里用到了流的延时处理（后面情况类似）

复合查询

- 几个特殊查询组合形式（复合查询）由专门过程处理
- 过程 conjoin 处理 and 查询，其参数是合取项的表和一个框架流。允许任意多个合取项
- conjoin 递归地使用各个合取项，处理后续合取项时使用处理第一个合取项得到的框架流：

```
(define (conjoin conjuncts frame-stream)
  (if (empty-conjunction? conjuncts)
      frame-stream
      (conjoin (rest-conjuncts conjuncts)
               (qeval (first-conjunct conjuncts)
                      frame-stream))))
```

- 为使 qeval 能使用 conjoin，需要将它设置好：

```
(put 'and 'qeval conjoin)
```

复合查询

- 过程 disjoin 处理 or 查询，它以一些析取项和一个框架流为参数，用各个析取项去扩充框架流里的框架，并归并得到的流：

```
(define (disjoin disjuncts frame-stream)
  (if (empty-disjunction? disjuncts)
      the-empty-stream
      (interleave-delayed
        (qeval (first-disjunct disjuncts) frame-stream)
        (delay (disjoin (rest-disjuncts disjuncts)
                        frame-stream)))))

(put 'or 'qeval disjoin)
```

归并需要用交错的方式做（用 interleave-delayed ）

两个分支分别处理第一个析取项和其余的析取项

过滤器

- not 和 lisp-value 用过滤器的方式实现
- not 用前面讨论的方式：设法扩充输入流中每个框架，看它能否满足作被 not 否定的模式。只将无法扩充的框架留在输出流里

```
(define (negate operands frame-stream)
  (stream-flatmap
   (lambda (frame)
     (if (stream-null? (qeval (negated-query operands)
                              (singleton-stream frame)))
         (singleton-stream frame)
         the-empty-stream))
   frame-stream))

(put 'not 'qeval negate)
```


过滤器

- lisp-value 的工作方式与 not 类似。先用流中各框架去实例化模式里的变量，而后将谓词应用于这些变量，丢掉使谓词返回假的框架。遇到未约束的变量是错误

```
(define (lisp-value call frame-stream)
  (stream-flatmap
   (lambda (frame)
     (if (execute
          (instantiate
           call
           frame
           (lambda (v f)
             (error "Unknown pat var -- LISP-VALUE" v))))
         (singleton-stream frame)
         the-empty-stream))
   frame-stream))

(put 'lisp-value 'qeval lisp-value)
```

被应用的谓词

处理实例化后的谓词

类似于 eval，但不求值谓词的参数（因它们已经是值）

过滤器

- instantiate 用 frame 实例化 call 里的变量，得到所需的谓词表达式
- execute 将谓词应用于实际参数。与 eval 不同，这里谓词作用的对象已是值，不应再次求值。execute 通过基础系统的 eval 和 apply 实现，它将 exp 里的谓词作用于参数

```
(define (execute exp)
  (apply (eval (predicate exp) user-initial-environment)
         (args exp)))
```

- 特殊形式 always-true 描述总能满足的查询，它忽略查询内容，直接返回作为参数的框架流

```
(define (always-true ignore frame-stream) frame-stream)
(put 'always-true 'qeval always-true)
```

这一特殊形式在一些选择函数里使用

- 与 not 和 lisp-value 有关的语法过程（选择函数）在后面定义

用模式匹配找断言

- 简单查询调用 find-assertion，返回将参数 frame 与数据匹配得到的框架形成的流。其中的 fetch-assertions 返回数据库中断言的流，它先用 pattern 和 frame 做简单检查，丢掉明显不可能匹配的断言

```
(define (find-assertions pattern frame)
  (stream-flatmap (lambda (datum)
                    (check-an-assertion datum pattern frame))
                  (fetch-assertions pattern frame)))
```

- check-an-assertion 对一个断言调用匹配过程，成功时返回包含一个扩充框架的流，不成功时返回空流

```
(define (check-an-assertion assertion pattern frame)
  (let ((match-result
         (pattern-match pattern assertion frame)))
    (if (eq? match-result 'failed)
        the-empty-stream
        (singleton-stream match-result))))
```


用模式匹配找断言

- pattern-match 是基本匹配器，匹配失败时返回符号 failed，或者返回成功扩充的框架。这里按结构递归地匹配

```
(define (pattern-match pat dat frame)
  (cond ((eq? frame 'failed) 'failed)
        ((equal? pat dat) frame)
        ((var? pat) (extend-if-consistent pat dat frame))
        ((and (pair? pat) (pair? dat))
         (pattern-match (cdr pat)
                        (cdr dat)
                        (pattern-match (car pat)
                                       (car dat)
                                       frame)))
        (else 'failed)))
```

相同时匹配成功，直接返回原框架

以匹配 car 部分得到的可能扩充的框架作为框架，递归匹配模式和数据的 cdr 部分

模式是个变量，基于frame 和新约束做扩充，检查是否协调

用模式匹配找断言

- extend-if-consistent：在 var 和 dat 的约束与 frame 里的约束协调时产生扩充的框架（if 的第二个分支）

```
(define (extend-if-consistent var dat frame)
  (let ((binding (binding-in-frame var frame)))
    (if binding
        (pattern-match (binding-value binding) dat frame)
        (extend var dat frame))))
```

找出 var 在 frame 里的约束

var 无约束，把新约束加入 frame

如果 var 在 frame 里已有约束，只有这一约束和现数据 dat 匹配时整个匹配才成功。注意：(binding-value binding) 取出的已有匹配里还可能有变量（由合一得到的约束）

示例：假设目前框架里 ?x 约束到 (f ?y) 而 ?y 无约束，现在想加入 ?x 与 (f b) 的约束来扩大框架。这一过程在框架里查找 ?x 并发现它已约束到 (f ?y)，这导致要在同一框架里做 (f ?y) 与新值 (f b) 的匹配，最终将 ?y 到 b 的约束加入框架，变量 ?x 仍约束到 (f ?y)

匹配中已有的约束绝不改变，也不会出现一个变量有多个约束的情况

带点号尾部的模式

- 如果模式中有圆点，圆点后面应是一个模式变量，该变量将与数据表的剩下部分匹配（而不是与下一元素匹配）
- 虽然在模式匹配器里没有专门处理圆点，但它能正确工作，这得益于模式和数据都用 Scheme 表表示，圆点自然得到上面的意义
- 当 read 读入查询时遇到圆点，它就会把下一个项直接作为正在构造的表达式的 cdr。例如：
  - 读入模式 (computer ?type)，read产生的表结构相当于对表达式 (cons 'computer (cons '?type '())) 求值所产生的结构
  - 读入模式 (computer . ?type) 时，产生的结构相当于对表达式 (cons 'computer '?type) 求值构造出的结构
- 当匹配器使用模式 (computer . ?type) 时，将用 ?type 与数据的 cdr 部分匹配。例如，将它与 (computer programmer trainee) 匹配时，?type 就会约束到 (programmer trainee)

规则和合一

- apply-rules 应用规则，以一个模式和一个框架为输入，生成一个框架流。它由 simple-query 调用，其中的 apply-a-rule 应用一条规则

```
(define (apply-rules pattern frame)
  (stream-flatmap (lambda (rule)
                    (apply-a-rule rule pattern frame))
                  (fetch-rules pattern frame)))
```

- 应用规则时有个问题：两条不同规则里的变量可能同名，实际上它们毫无关系。设两条规则里都有变量 ?x，如果直接用规则去匹配，两条规则都可能向框架里加入 ?x 的约束，从而相互干扰
  - 原本这两个 ?x 的约束相互无关，而一条规则在框架里加入 ?x 的约束却会造成另一条规则无法将相应约束加入
- 为防止这种相互干扰，这里采用重命名技术：为每个规则应用关联一个唯一标识号，应用时给所有变量名加上这个编号。例，如果应用的编号是 5，就把 ?x 改名为 ?x-5，?y 改名为 ?y-5，其余变量也一样改

规则和合一

- apply-a-rule 应用一条规则

```
(define (apply-a-rule rule query-pattern query-frame)
  (let ((clean-rule (rename-variables-in rule)))
    (let ((unify-result
           (unify-match query-pattern
                        (conclusion clean-rule)
                        query-frame)))
      (if (eq? unify-result 'failed)
          the-empty-stream
          (qeval (rule-body clean-rule)
                 (singleton-stream unify-result))))))
```

规则里的变量统一改名，使之不会与其他规则冲突

做实际的合一匹配

基于得到的新框架流做规则体的匹配

规则和合一

- 构造新的“干净”规则很容易：递归遍历该规则，重命名所有变量（加唯一编号后缀，实际形式也是抽象，下面会看到具体方式）

```
(define (rename-variables-in rule)
  (let ((rule-application-id (new-rule-application-id)))
    (define (tree-walk exp)
      (cond ((var? exp)
             (make-new-variable exp rule-application-id))
            ((pair? exp)
             (cons (tree-walk (car exp)) (tree-walk (cdr exp))))
            (else exp)))
    (tree-walk rule)))
```

- 每次应用规则前重新构造一个“干净”规则，这种方法比较耗时。可以考虑其他方法，但这一方法简单易行

规则和合一

- 合一与简单匹配的不同就在于匹配的两边都可能有变量，因此都可能建立约束。与简单匹配过程的仅有不同是对变量的处理：

```
(define (unify-match p1 p2 frame)
  (cond ((eq? frame 'failed) 'failed)
        ((equal? p1 p2) frame)
        ((var? p1) (extend-if-possible p1 p2 frame))
        ((var? p2) (extend-if-possible p2 p1 frame))
        ((and (pair? p1) (pair? p2))
         (unify-match (cdr p1)
                      (cdr p2)
                      (unify-match (car p1) (car p2) frame)))
        (else 'failed)))
```

两边都可能是变量

- 遇变量时要考虑两种情况，由 extend-if-possible 完成
  - 如果另一方也是变量，则需考虑它是否已有约束。如果有，就让被处理变量取相同约束；否则就直接将其约束于另一方变量
  - 如果要将变量约束于一个模式，而模式里有这个变量。那么任何赋值都不可能实现这一匹配，应作为匹配失败

规则和合一

- 如两模式里都有重复变量，可能出现第二种情况。例：匹配 (?x ?x) 和 (?y (a ?y))。先得到了 ?x 约束于 ?y ，下面要用 ?x 匹配 (a ?y)。由于 ?x 约束于 ?y，因此要匹配 ?y 和 (a ?y)。显然这一匹配不可能成功
- 处理这两个问题的过程：

```
(define (extend-if-possible var val frame)
  (let ((binding (binding-in-frame var frame)))
    (cond (binding
           (unify-match (binding-value binding) val frame))
          ((var? val)
           (let ((binding (binding-in-frame val frame)))
             (if binding
                 (unify-match var (binding-value binding) frame)
                 (extend var val frame))))
          ((depends-on? val var frame) 'failed)
          (else (extend var val frame)))))
```

匹配的另一方也是变量。如果该变量有约束，则要求 var 可与该变量的约束值合一

若 var 已有约束，要求其约束值可与 val 合一

val 依赖于 var 时匹配失败

规则和合一

- depends-on? 检查一个表达式是否依赖于一个变量 ?x。这一检查也需要相对于一个 frame 进行，因为可能在模式里出现另一变量 ?y，而 ?y 在 frame 里的约束依赖于 ?x（还可能继续传递）
- 这一检查基本是按结构递归

```
(define (depends-on? exp var frame)
  (define (tree-walk e)
    (cond ((var? e)
           (if (equal? var e)
               true
               (let ((b (binding-in-frame e frame)))
                 (if b
                     (tree-walk (binding-value b))
                     false))))
          ((pair? e) (or (tree-walk (car e)) (tree-walk (cdr e))))
          (else false)))
  (tree-walk exp))
```

e 是变量且不同于 var。此时要检查 e 在 frame 里是否有约束，其约束是否依赖于 var

数据库维护

- 考虑数据库维护。关键是做出安排，使检索时需考察的断言尽可能少。这里先把所有断言存入一个大流，同时把 car 部分是相同常量的断言都存入同一个流，用这一共同的 car 作为索引，将个流存入一个表格（另一关键码用 assertion-stream）

```
(define THE-ASSERTIONS the-empty-stream)

(define (fetch-assertions pattern frame)
  (if (use-index? pattern)
      (get-indexed-assertions pattern)
      (get-all-assertions)))

(define (get-all-assertions) THE-ASSERTIONS)

(define (get-indexed-assertions pattern)
  (get-stream (index-key-of pattern) 'assertion-stream))
```

pattern 的 car 是常量，此时到特定的流里去检索

get-stream 按 key1 和 key2 到表格里找相应流，找不到时返回空流

```
(define (get-stream key1 key2)
  (let ((s (get key1 key2)))
    (if s s the-empty-stream)))
```


数据库维护

- 规则管理的方式类似，以规则中结论部分的 car 为索引，将结论 car 相同的规则的流存入表格（另一关键码是 rule-stream）
- car 部分是常量的模式可以与结论具有相同 car 的规则匹配，但它还可以与结论的 car 是变量的规则匹配。为便于处理，把所有结论的 car 部分是变量的规则存入以 ? 为索引的流。对 car 部分是常量的模式，与之匹配的流可能由这两个流组成：

```
(define THE-RULES the-empty-stream)

(define (fetch-rules pattern frame)
  (if (use-index? pattern)
      (get-indexed-rules pattern)
      (get-all-rules)))

(define (get-all-rules) THE-RULES)

(define (get-indexed-rules pattern)
  (stream-append
   (get-stream (index-key-of pattern) 'rule-stream)
   (get-stream '? 'rule-stream)))
```

数据库维护

- 加入断言或规则的请求分情况处理，不但将它加入包含所有断言或规则的主流，还根据其（或结论）的 car 加入表格里的支流

```
(define (add-rule-or-assertion! assertion)
  (if (rule? assertion)
      (add-rule! assertion)
      (add-assertion! assertion)))

(define (add-assertion! assertion)
  (store-assertion-in-index assertion)
  (let ((old-assertions THE-ASSERTIONS))
    (set! THE-ASSERTIONS
          (cons-stream assertion old-assertions))
    'ok))

(define (add-rule! rule)
  (store-rule-in-index rule)
  (let ((old-rules THE-RULES))
    (set! THE-RULES (cons-stream rule old-rules))
    'ok))
```


数据库维护

加入支流的工作由两个专门过程完成：

```
(define (store-assertion-in-index assertion)
  (if (indexable? assertion)
      (let ((key (index-key-of assertion)))
        (let ((current-assertion-stream
               (get-stream key 'assertion-stream)))
          (put key
               'assertion-stream
               (cons-stream assertion current-assertion-stream))))))

(define (store-rule-in-index rule)
  (let ((pattern (conclusion rule)))
    (if (indexable? pattern)
        (let ((key (index-key-of pattern)))
          (let ((current-rule-stream
                 (get-stream key 'rule-stream)))
            (put key
                 'rule-stream
                 (cons-stream rule current-rule-stream)))))))
```


数据库维护

- 可以加入支流的条件是模式的 car 是常量符号。对规则，还要考虑其结论（模式）的 car 是模式变量的情况：

```
(define (indexable? pat)
  (or (constant-symbol? (car pat))
      (var? (car pat))))
```

模式存入表格用的关键码就是其 car。对规则的结论模式，如果其 car 是模式变量，关键码用 ?:

```
(define (index-key-of pat)
  (let ((key (car pat)))
    (if (var? key) '? key)))
```

如模式的 car 是常量符号，就用它作为索引去提取相应的流用于检索：

```
(define (use-index? pat)
  (constant-symbol? (car pat)))
```


流操作

- 查询系统用了几个前面没定义的流操作。包括流的 append

```
(define (stream-append-delayed s1 delayed-s2)
  (if (stream-null? s1)
      (force delayed-s2)
      (cons-stream
       (stream-car s1)
       (stream-append-delayed (stream-cdr s1) delayed-s2))))
```

流的交错归并：

```
(define (interleave-delayed s1 delayed-s2)
  (if (stream-null? s1)
      (force delayed-s2)
      (cons-stream
       (stream-car s1)
       (interleave-delayed (force delayed-s2)
                           (delay (stream-cdr s1))))))
```

流操作

- 把过程 proc 用于 s 的每个元素后将得到的流平坦化：

```
(define (stream-flatmap proc s)
  (flatten-stream (stream-map proc s)))

(define (flatten-stream stream)
  (if (stream-null? stream)
      the-empty-stream
      (interleave-delayed
       (stream-car stream)
       (delay (flatten-stream (stream-cdr stream))))))
```

- 构造只包含一个元素的流：

```
(define (singleton-stream x)
  (cons-stream x the-empty-stream))
```


查询的语法过程

- 有类型表达式应该是表是其第一个元素，其内容就是去掉第一个元素之后的那个表：

```
(define (type exp)
  (if (pair? exp)
      (car exp)
      (error "Unknown expression TYPE" exp)))

(define (contents exp)
  (if (pair? exp)
      (cdr exp)
      (error "Unknown expression CONTENTS" exp)))
```

- 断言的类型是 assert，其内容就是表的第二个元素（在基本驱动循环里用 add-assertion-body）

```
(define (assertion-to-be-added? exp)
  (eq? (type exp) 'assert!))

(define (add-assertion-body exp)
  (car (contents exp)))
```


查询的语法过程

- 几种组合断言的语法过程（and/or/not/lisp-value）

```
(define (empty-conjunction? exps) (null? exps))
(define (first-conjunct exps) (car exps))
(define (rest-conjuncts exps) (cdr exps))
(define (empty-disjunction? exps) (null? exps))
(define (first-disjunct exps) (car exps))
(define (rest-disjuncts exps) (cdr exps))
(define (negated-query exps) (car exps))
(define (predicate exps) (car exps))
(define (args exps) (cdr exps))
```

- 规则的语法过程：

```
(define (rule? statement) (tagged-list? statement 'rule))
(define (conclusion rule) (cadr rule))
(define (rule-body rule)
  (if (null? (cddr rule))
      '(always-true)
      (caddr rule)))
```


查询的语法过程

- 把模式中的模式变量变形，例如 ?x 变成 (? x)，使处理更方便

```
(define (query-syntax-process exp)
  (map-over-symbols expand-question-mark exp))

(define (map-over-symbols proc exp)
  (cond ((pair? exp)
         (cons (map-over-symbols proc (car exp))
               (map-over-symbols proc (cdr exp))))
        ((symbol? exp) (proc exp))
        (else exp)))

(define (expand-question-mark symbol)
  (let ((chars (symbol->string symbol)))
    (if (string=? (substring chars 0 1) "?")
        (list '?
              (string->symbol
               (substring chars 1 (string-length chars))))
        symbol)))
```

注意：这里用到符号到字符串和字符串到符号的转换

取得 symbol 的名字字符串

名字的第一个字符是否?

取得 symbol 的名字除去? 后的字符串

查询的语法过程

- 经过前面变换，模式变量就是以 ? 为类型（car）的表。常量符号就是 Scheme 里的一般符号

```
(define (var? exp)
  (tagged-list? exp '?))

(define (constant-symbol? exp) (symbol? exp))
```

- 为完成规则中的模式变量换名，需要下面几个过程：

```
(define rule-counter 0)

(define (new-rule-application-id)
  (set! rule-counter (+ 1 rule-counter))
  rule-counter)

(define (make-new-variable var rule-application-id)
  (cons '? (cons rule-application-id (cdr var))))
```

换名后各模式变量用的形式是 (? 3 x)，(? 8 y)

查询的语法过程

- 驱动循环打印结果前要把结果中未约束的变量变换回原来形式。由于可能出现规则换名中生成的模式变量，因此需要分别处理
- 换名变量的特点是表里的第二个元素是数，下面过程生成原变量名，换名后的变量生成的变量名加了后缀

```
(define (contract-question-mark variable)
  (string->symbol
   (string-append "?"
    (if (number? (cadr variable))
        (string-append (symbol->string (caddr variable))
                       "-"
                       (number->string (cadr variable)))
        (symbol->string (cadr variable))))))
```


框架和约束

- 框架就是以一组约束为元素的表，约束用 cons 序对表示

```
(define (make-binding variable value)
  (cons variable value))
(define (binding-variable binding)
  (car binding))
(define (binding-value binding)
  (cdr binding))
(define (binding-in-frame variable frame)
  (assoc variable frame))
(define (extend variable value frame)
  (cons (make-binding variable value) frame))
```

- 至此，整个解释器就完成了
- 本节有许多练习，提出了这个求值器的一些修改和扩充，还提出了许多相关问题。请自己看一看

总结

- 逻辑程序设计语言的基本想法是
  - 在逻辑的层次上描述要求计算什么
  - 由语言解释器实现一个计算过程，把需要的东西算出来
- 一个“做什么”的描述可能蕴涵着许多“怎样做”的过程，它可能描述了多种不同方向的计算，也可能得到许多结果（非确定性）
- 这里研究的逻辑编程语言是一种查询语言，用于建立和查询断言数据库
  - 断言描述基本事实
  - 规则描述事实之间的抽象关系
  - 提供了一些组织查询的机制（and、or 等等）
- 这里用框架流的方式实现查询语言的解释器
- 应特别注意逻辑程序设计的作为数学的逻辑之间的关系，两者有相似之处，但并不等价