4. 元语言抽象(4)

本节讨论

- 逻辑程序设计
- 逻辑程序设计语言的实现方法
- 关键操作：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
- 查询语言的实现（下次课）

逻辑程序设计

- 前面说：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的过程
  - 实际上，程序语言通常也提供一些说明性描述方式

    使用户可以免去很多计算过程的细节描述

    例如，输出函数的格式描述
- 大多数程序语言要求用定义数学函数的方式组织程序
  - 程序要描述“怎么做”的过程
  - 所描述的计算有明确方向，从输入到输出
  - 描述经计算的表达式，给出了从一些值算出结果的方法（和过程）
  - 定义过程描述如何从参数计算出结果

逻辑程序设计

- 也有些例外。前面例子：
  - 约束传递系统中的计算对象是约束关系，没有明确计算方向和顺序，它的基础系统要做很多工作以支持相应的计算
  - 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
- 逻辑程序设计可看作上面想法的推广
  - 基于一种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
- 逻辑程序设计已被用于许多实际领域

  如数据库查询语言 **Datalog**，支持查询基于已有事实的隐含事实

逻辑程序设计

- 一个“是什么”的描述可能蕴涵许多“怎样做”的过程。考虑 **append**：

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

- 可认为，这个程序表达了两条规则：

  1，对任何一个表 **y**，空表与其拼接得到的表是 **y** 本身

  2，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** 的拼接得到 **z**
- **append** 的过程定义和上述两条规则都可以回答下面问题：

  找出 **(a b)** 和 **(c d)** 的 **append**
- 这两条规则还可以回答（但 **append** 过程不行）：

  找出一个表 **y** 使 **(a b)** 与它的拼接得到 **(a b c d)**

  找出所有拼接起来将得到 **(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面各问题的解
- 但各种逻辑语言（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，或产生了不是用户希望的行为
- 这个领域还需要深入，有关新方向是 **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** 很不一样，但前面框架仍然适用：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：其“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
  - 下面介绍的解释器实现里，采用了一种框架结构作为求值的核心数据结构，还用到流的概念

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

  下面可以看到，逻辑式语言不仅能描述基于数据的信息访问，而且支持基于已有数据的推理
- 设 **Microshaft** 公司（波士顿一家高科技公司）需要一个人事数据库，内容是有关公司人事的断言，描述各种与人事有关的事实
- 断言和“解释”

  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)

  一个断言形式上是一个表，其元素还可以是表

  一个断言描述一个事实

演绎信息检索

- Ben 管理公司的计算机分部，管理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 管理一个实习程序员：

```
(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的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员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，只需要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

  系统响应查询时，给出数据库里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)

  处理这个查询，系统将列出所有雇员的地址条目

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）

- 如果查询中没有变量，就相当于问相应事实是否存在
- 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))

  它不匹配下面事实（由于 **?type** 只能匹配一项）：

  (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，可以写：

  (job ?x (computer . ?type))

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

- 系统对简单查询的处理总结：
  - 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
  - 如果查询模式里没有变量，查询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**（注意，不是 **Scheme** 的内部操作）

  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*$_n$>)**

  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，充分利用查询的语义

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，如果被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）的一集断言，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))))
```

  这是一条递归定义的规则：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：若对所有模式变量的赋值能满足一条规则的体，则它就满足其结论。可认为查询语言就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)**，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，这里用到表的点号形式

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

虽然对于 **append** 这个例子，系统的行为令人满意，一般情况下未必如此（下面有讨论）

查询系统

- 先讨论查询系统求值器的原理，以及与实现细节无关的一般性结构
  - 还要提出该求值器的局限性
  - 说明查询系统的逻辑运算与数理逻辑的运算之间的差异
- 显然，查询求值器要做搜索，将查询与数据库里的事实和规则匹配
  - 可以参考 **amb** 解释器的实现技术完成一个解释器（练习**4.78**）
  - 也可以借用流的概念控制搜索
  - 下面介绍的设计采用基于流的技术
- 查询系统的组织围绕着两个核心操作：模式匹配和合一
  - 模式匹配（**pattern match**）操作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（**unification**）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，构造整个查询解释器

查询系统基本操作：模式匹配

- 查询系统的基本部件包括一个模式匹配器
- 模式匹配器检查给定的一个数据项（形式是表）是否与一个给定的模式（形式也是表）匹配，并确定模式中各模式变量的约束
- 例如：考虑数据项 **((a b) c (a b))**
  - 与模式 **(?x c ?x)** 匹配

    确定其中的模式变量 **?x** 约束于 **(a b)**
  - 与模式 **(?x ?y ?z)** 匹配

    其中的 **?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

    这个模式要求表中第二个元素必须是 **a**（模式中的“符号”常量只能与其自身匹配）

模式匹配

- 匹配过程中用框架记录模式变量与其当时确定的约束值的关系
  - 这里的框架与元循环求值器的框架类似
  - 同样需要加入和查找，但已建立的约束不会修改
- 模式匹配器以一个模式、一个数据和一个框架作为输入。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，加入在新的匹配中确定的新约束；找不到匹配时返回一个失败信息
- 假设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**
  - 用同样模式和数据，以及包含 **?y** 约束到 **a** 的框架，匹配将失败
  - 用同一模式和数据，以及包含 **?y** 约束到 **b** 的框架，匹配器返回的框架加入扩充的 **?x** 到 **a** 的约束

查询系统的模式匹配

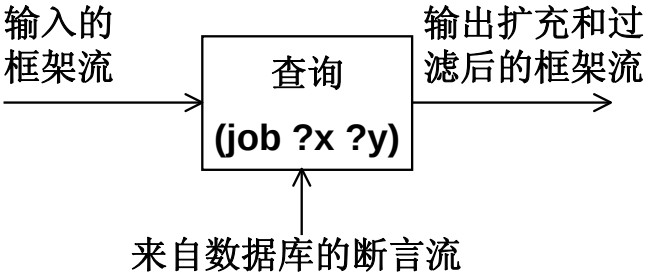
- 查询处理采用统一工作方式。对一个模式（或部分模式）的查询
  - 总以一个框架流作为一项输入
  - 基于流中每个框架去匹配一组数据（来自数据库，为另一输入）
  - 合并产生的所有框架流，得到查询的输出流
- 匹配器基于给定的输入框架流中的框架做匹配：
  - 扫描数据库断言。对每个断言，或产生表示匹配失败的特殊符号，或给出原框架一个扩充，匹配结果形成一个流
  - 用一个过滤器删除匹配失败信息，结果流里包含的框架都是原框架由于断言匹配而得到的扩充框架
  - 查询可能通过许多步骤，总是基于框架流做匹配，过滤掉失败信息，合并得到一个结果框架流
  - 最后，语言求值器用匹配器返回的各个框架实例化查询模式（将其中的变量代换为相应约束值），得到（一组）最终结果

基本查询框架

- 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))

  - 匹配器将从一个空框架出发扫描数据库里的断言
  - 选出其中（基于空框架）与这个模式匹配的断言
  - 得到相应的匹配框架流
- 简单查询的处理：
  - 输入流只包含一个空框架，结果流包含空框架的所有可能扩充
  - 用这个流实例化查询模式，就能得到所有输出



复合查询：and

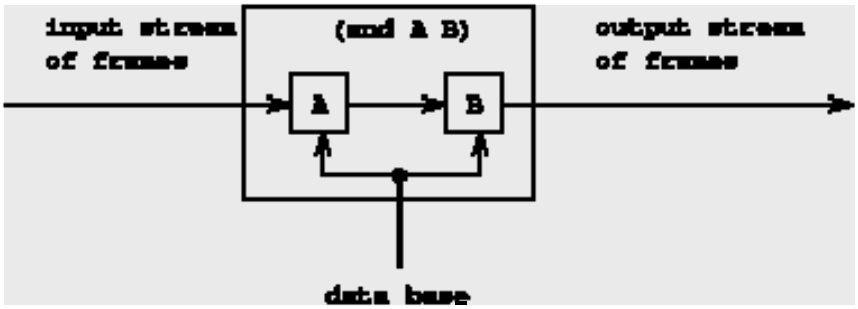
- 处理复合查询的过程中，前面步骤可能已经得到一个框架流。后续的匹配将基于流中的框架去做进一步的匹配
- 例（**and** 的处理过程）：

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

  先得到与 **(can-do-job ?x (computer programmer trainee))** 匹配的框架流（简单匹配），其中每个框架都包含 **?x** 的约束项。再找与模式 **(job ?person ?x)** 匹配的项，要求其匹配与已有 **?x** 的匹配一致

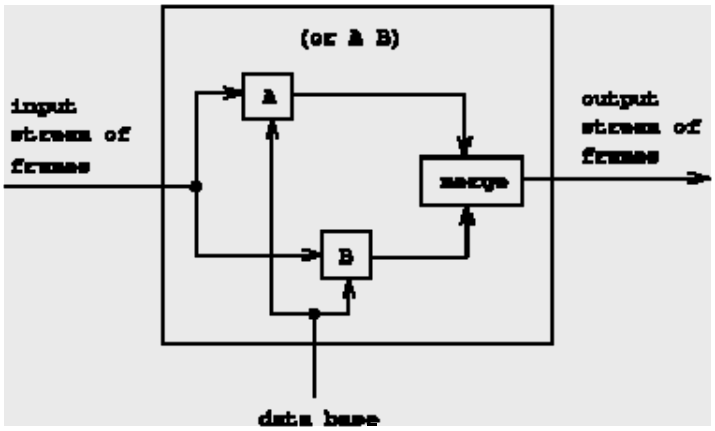
  结果流里的各框架都包含了 **?person** 和 **?x** 的约束

- 右图是 **and** 查询处理过程。框架流顺序通过两个查询。注意：组合查询具有同样的处理形式



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



- 复合查询的处理效率：
  - 对输入流的每个框架，一步查询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
  - and 查询的匹配次数与查询个数可能成指数关系（最坏情况）
- 上面方法适用于较简单的查询，但不适合处理很复杂的查询

  这里有值得研究的实现技术问题

复合查询：not 和 lisp-value

- 查询断言 q 的 not 是一个框架过滤器，删除流中所有满足 q 的框架
- 例如对模式 (not (job ?x (computer programmer)))
  - 设法从框架生成满足 (job ?x (computer programmer) 的扩充
  - 如果一个框架能扩充就丢掉，不能扩充的框架放入输出流
- 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 not 子句删除所有把 ?x 约束到程序员的框架
- 特殊查询形式 lisp-value 也实现为框架过滤器：
  - 用流中框架实例化模式里的变量
  - 对实例化结果应用给定谓词，不满足谓词的框架全部删除

规则的处理和合一

- 处理规则时，要找出结论部分与查询模式匹配的所有规则
  - 结论的形式与断言类似，但是可以包含变量
  - 新问题：匹配的两边（查询模式和规则结论）都可能有变量，模式匹配不能处理这个问题（模式匹配只允许一方有变量）
- 合一是模式匹配的扩充，判断两个模式之间能否匹配。方法：
  - 设法确定是否存在一组变量赋值（变量约束），用这些变量赋值实例化这两个模式后，能得到同一个表达式
  - 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，这个框架里 **?x, ?y** 和 **?z** 都约束到 **a**
  - 对 **(?x ?y a)** 和 **(?x b ?y)** 的合一将失败，因为对 **?y** 的任何赋值都不能使两个模式相同（根据模式里的第二个元素，**?y** 应约束到 **b**；而根据其第三个元素 **?y** 必须约束到 **a**）

合一

- 合一算法是整个查询系统实现的难点

  完成两个复杂模式的合一，看起来好像需要做推理
- 例，合一 **(?x ?x)** 和 **((a ?y c) (a b ?z))**，可以得到一个联立方程：

  ?x = (a ?y c)

  ?x = (a b ?z)

  它等价于解方程 **(a ?y c) = (a b ?z)**

  它蕴涵着 **a = a, ?y = b, c = ?z**

  这样做下去，可以得到：**?x = (a b c)**
- 最一般的合一确实需要解方程。现在情况比较简单，可以直接处理
  - 模式匹配成功将给所有的变量赋值，前例中赋值都为常量
  - 但是，合一成功也可能产生变量值不能完全确定的情况

    可能出现未约束变量，变量的约束值里也可能还包含变量

规则的应用

- 例：考虑 (?x a) 和 ((b ?y) ?z)

  合一得到 ?x = (b ?y)，a = ?z，但?x 和 ?y 的值都不确定。这时也认为合一成功，因为已确定了 ?x 和 ?y 的赋值

  这里 ?y 的取值无限制，但 ?x 必须是 (b ?y)。应该把 ?x 到 (b ?y) 的约束放入框架，如果后来确定了 ?y 的值，?x 就引用这个值

- 例：假设查询是 (lives-near ?x (Hacker Alyssa P))

  先用模式匹配在数据库里找匹配断言（找不到）。再做与各条规则的结论做合一，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

  基于这个框架继续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

规则应用：一般工作过程

- 求值器基于一个框架对一个查询模式尝试应用规则的过程：
  - 将查询模式与规则的结论合一，成功时得到原框架的一个扩充
  - 基于得到的扩充框架求值该规则的体（这又是一个查询）
- 做法类似于 Scheme 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，用得到的框架扩充原环境
  - 基于扩充后的环境去求值过程体
- 这种情况很自然：
  - 过程定义是 Scheme 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，都需要打开相关抽象，就是建立相应约束，而后基于它们求值过程或者规则的体
- 下面讨论更具体的实现问题

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，得到扩充框架流
    - 合一器处理所有可用的规则，得到另一个扩充框架流
  - 归并处理输入框架流里各个框架得到所有扩充框架的流，把这些流组合为一个流，其中的框架都满足给定模式且与原框架相容，是由输入流中各框架扩充而得到的
- 这个系统很像一般语言的求值器，只是其中的匹配操作比较复杂
- 过程 **qeval** 协调各种匹配操作，其作用类似于 **eval**
  - 参数是一个查询和一个框架流，结果是一个框架流，其中包含了通过成功匹配而得到的所有扩充框架
  - **qeval** 根据查询的类型分情况处理，将请求分派到对应的过程（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（查询）
  - 用输入查询和只包含一个空框架的流调用 **qeval**
  - 用 **qeval** 返回的流里的每个框架实例化原查询，得到一个结果
  - 最后打印出实例化的结果
- 驱动循环还检查特殊命令 **assert!**
  - 这个命令说明输入是一条断言或规则，不是查询
  - 处理时把相应断言或规则加入数据库
- 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

- 查询语言的组合符对应于常用逻辑连接词，查询操作看起来也具有逻辑可靠性（例如，**and** 查询要经过两个子成分处理等）

  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和控制流程，是采用过程的方式解释逻辑语句

- 这种隐含的控制结构我们有可能利用

  例如，要找程序员的上司，下面两种写法都行：

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

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

  如果公司里的有关上司关系的事实比有关程序员的事实更多，第一种写法的查询效率更高

逻辑程序设计和数理逻辑

- 逻辑程序设计的目标是开发一种技术，把计算问题分为“要计算什么”和“怎样计算”两个相互独立的子问题，方法是：
  - 找出逻辑语言的一个子集，其
    - 功能足够强，足以描述人们想考虑的某类计算
    - 又不过分的强，有可能为它定义一种过程式的解释
  - 实现一个求值器（解释器），执行对用这种逻辑子集写出的规则和断言的解释（实现其语义，形式上是做推理）
- 前面的查询语言是这种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
  - 一个断言描述一个简单事实
  - 一条规则表示一个蕴涵，能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

逻辑程序设计和数理逻辑

- 上面提出的两方面性质保证了逻辑程序设计语言程序的有效性
  - 用这种语言写出的一组规则，实际上描述了一个计算过程
  - 写出的断言可以交给计算机执行（说明式的描述）
  - 具体的控制流程交给求值器处理（过程式的执行）
- 由于规则是逻辑语句，有逻辑解释。因此可以做一些逻辑工作：
  - 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”）
- 逻辑程序的执行有一些过程式特征
  - 程序员可以安排子句的顺序和各子句中子目标的顺序，用这些顺序控制计算过程。安排得当可能得到较高效的计算
  - 由于采用过程性的解释，因此也有可能写出很低效的逻辑式程序
  - 极端情况是写出的程序导致解释过程（解释器）无穷循环

无穷循环

- 例，假定要做一个有关著名婚姻的数据库，其中有断言

  (assert! (married Minnie Mickey))
- 查询 **(married Mickey ?who)** 得不到结果

  原因：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)))**，再次查询，系统就会陷入无穷循环：
  - 这个规则产生的框架里 **?x** 约束到 **Mickey**，**?y** 约束到 **?who**
  - 规则体要求基于已有框架匹配 **(married ?who Mickey)**。该查询与事实匹配，也与上面规则匹配。由规则体得到的查询还是 **(married Mickey ?who)**，使系统进入无穷循环
  - 能否在进入无穷循环前找到匹配断言依赖于实现细节。对这个例子，系统可以找到 **(married Minnie Mickey)**
- 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

not 问题

- 另一个问题与 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))
```

- 这两个查询会得到不同结果（与逻辑里的情况不同）
  - 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（只包含一个空框架）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
- 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
  - 如果用 not 时模式包含未约束变量，就会产生不希望的结果

not 的问题

- lisp-value 也有问题。如果使用 lisp-value 的谓词时有些参数没有约束值，显然系统无法正常工作
- 此外，查询语言的 not 与逻辑的 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”
  - 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
  - 这显然不符合形式化的数理逻辑，也不符合人们的直观推理

实现，驱动循环和实例化

- 现在考虑系统的具体实现
- 上层：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
    - 遇到断言或规则，把相关信息加入数据库
    - 否则，都认为是查询，把输入（查询）送给 **qeval**，同时送去只包含一个空框架的流
  - 对查询的求值得到一个框架流
    - 流中各框架里的项给出了模式中各个变量的约束值
    - 用流中框架对模式做实例化，得到实例化结果的流
  - 最后输出结果流中的各项

    这些项都是简单的或者复合的断言

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)))))
```

对模式做变形，方便后面的处理

输入是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

迭代，准备处理下一查询

包含一个空框架的流，生成匹配框架流的基础

驱动循环和实例化

- 表达式是一个数据抽象，后面考虑其语法和语法过程
- 处理输入表达式前将其变换为易处理形式，修改变量的表示。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，用框架里的约束代换变量，无约束变量用参数（过程）**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过程

  - 参数是一个查询模式和一个框架流
  - 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，根据类型完成分派（采用数据导向技术）

  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（后面的情况都类似）

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

  参数是合取项的表和一个框架流。允许任意多合取项
- **conjoin** 递归地使用各个合取项

  基于处理第一个合取项得到的框架流去处理其他合取项：

```
(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

  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，最后归并得到的流

  归并采用交错的方式做（用 **interleave-delayed** ）

  两个分支分别处理第一个析取项和其余析取项

- 实现：

```
(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)
```


过滤器

- **not** 和 **lisp-value** 用过滤器的方式实现
- **not** 采用前面讨论的方式

  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

  只把不能扩充的框架留在输出流里

- 实现：

```
(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。遇到未约束的变量时报错误

```
(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：谓词作用的对象已是值，不需要求值。execute 通过基础的 eval 和 apply 实现，把 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

  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**，返回参数 **frame** 与数据匹配得到的框架形成的流。其中 **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，成功时返回的流里值包含一个经过扩充的框架，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）

例：设当前框架里 **?x** 约束到 **(f ?y)** 而 **?y** 无约束，想加入 **?x** 与 **(f b)** 的约束扩大框架。上面过程在框架里查找 **?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，已有约束绝不改变，也不会出现同一变量存在多个约束的情况

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）
  - 虽然模式匹配器没专门处理圆点，但能正确工作
  - 模式和数据都用 **Scheme** 的表表示，圆点自然有正确的意义
- **read** 读查询时遇到圆点就把下一个项作为所构造表达式的 **cdr**。例：
  - 读入模式 **(computer ?type)**，**read** 产生的表结构相当于对表达式 **(list 'computer '?type)** 求值产生的结构
  - 读入模式 **(computer . ?type)** 时，产生的结构相当于对表达式 **(cons 'computer '?type)** 求值产生的结构
- 如果匹配器用模式 **(computer . ?type)** 匹配
  - 它将用 **?type** 与数据的 **cdr** 部分匹配
  - 例如与 **(computer programmer trainee)** 匹配时，**?type** 将约束到 **(programmer trainee)**

规则和合一

- apply-rules 由 simple-query 调用，以一个模式和一个框架为输入(其中 apply-a-rule 应用一条规则)，生成一个框架流：

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

- 一个问题：两条规则里的变量可能同名，它们实际上相互无关

  如两条规则都有 ?x，直接匹配就都可能把 ?x 的约束加入框架

  实际上这两个 ?x 的约束相互无关，而一条规则给框架加入 ?x 的约束，就会导致另一条规则无法加入 ?x 的约束

- 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唯一标识号，应用时给所有变量名加这个编号

  例：如果应用的编号是 5，就把 ?x 改名为 ?x-5，?y 改名为 ?y-5

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”规则的过程：

  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，后面会看到具体方式）

```
(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)))
```

- 每次应用规则之前重新构造一个“干净”规则

  这种方法上述方法简单易行，也很好用，但比较耗时

  可以考虑其他方法

规则和合一

- 合一的不同点就在于匹配的两边都可能有变量，因此都可能建立约束。与简单匹配的仅有差异在对变量的处理：


```
(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)))
```

- 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)。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）

这一检查基本上是按结构递归

```
(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

数据库组织

- 数据库组织的关键是减少检索时需要考察的断言

  把所有断言存入一个流，再把 car 是相同常量的断言存入同一流，以该 car 为索引把流存入一个表格（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，以它为索引到特定流检索

get-stream到表格里按 key1 和 key2 找相应流，没有就返回空流

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


数据库组织

- 规则管理的方法类似，以规则中结论部分的 car 为索引，把 car 相同的规则的流存入表格（另一关键码用 'rule-stream）
- car 是常量的模式可与结论的 car 相同的规则匹配，也与结论的 car 是变量的规则匹配。为方便，所有结论的 car 是变量的规则存入 ? 索引的流。与 car 部分是常量的模式匹配的流可能由两个流组成：

```
(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

- 分情况处理加入断言或规则的请求，不但将它们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

加入各分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

- 可以加入某分支流的条件是模式的 car 是常量符号；对于规则还要考虑其结论的 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。对于规则的结论模式，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：

```
(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，再把得到的流平坦化：

```
(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

- 有类型的表达式应该是表，它的类型就是它的第一个元素，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)，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

这是系统过程

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，模式变量的实际形式是 (? 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，需要特别处理

- 换名变量的特点是表的第二个元素是数

  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。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和作为数学的逻辑之间的关系
  - 两者有相似之处，但并不等价