Chapter 8

STRONG METHOD PROBLEM SOLVING

(问题求解的强方法）

8.0 Introduction

- **We continue studying issues of representation and intelligence by considering an important component of AI:**
  - ***knowledge-intensive*（知识密集型）or *strong method*（强方法）problem solving.**

- Human experts are able to perform at a successful level because they know a lot about their areas of expertise（专业技能）.
- An *expert system* uses knowledge specific to a problem domain to provide "expert quality" performance.
- Expert system designers acquire this knowledge, and the system emulates（仿真）the human expert methodology（系统化方法）.

- As with skilled humans, expert systems tend to be specialists, focusing on a narrow set of problems.
- Like humans, their knowledge is both theoretical and practical（实用的）.

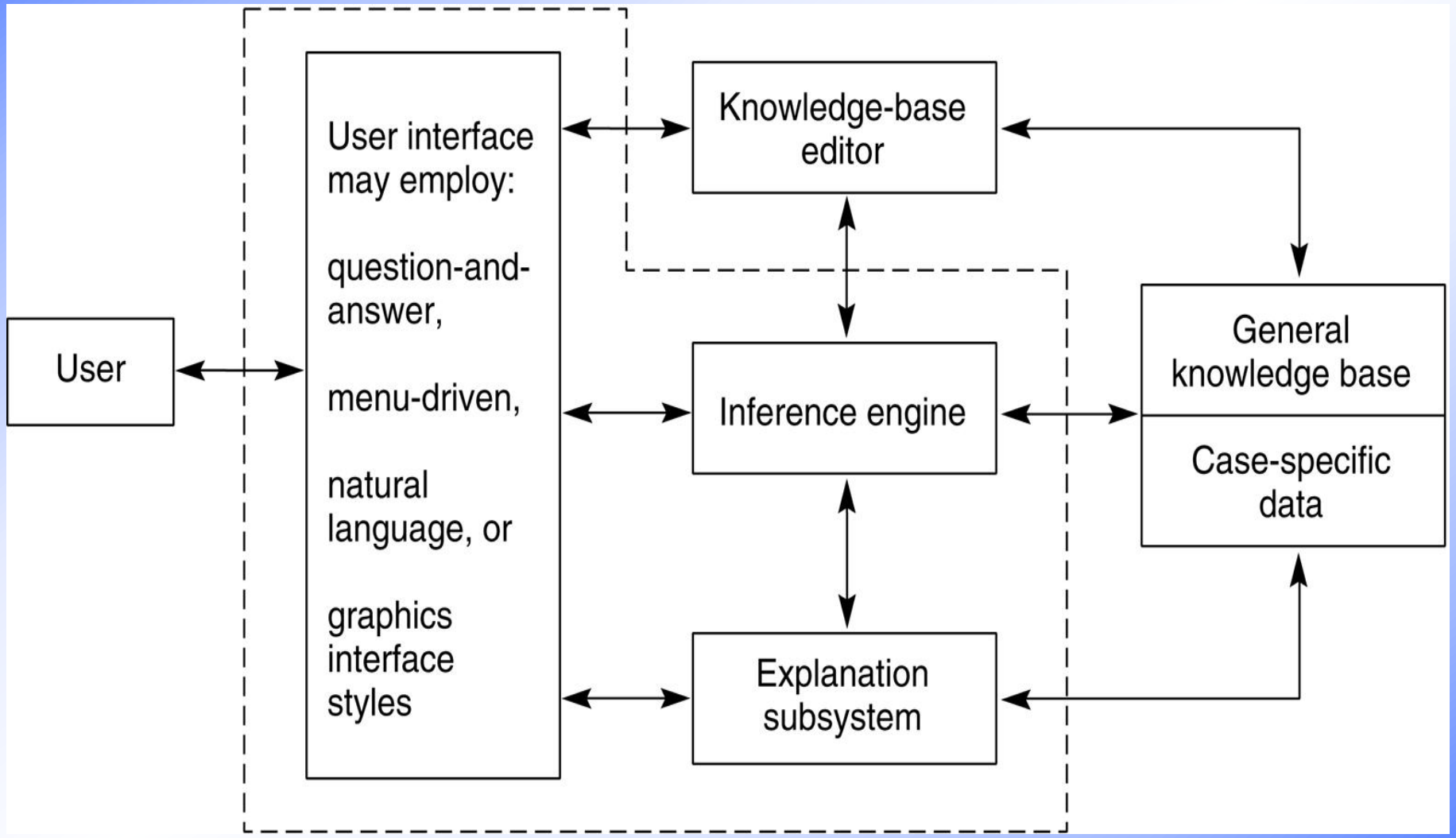
- ***Interpretation*** **— forming high-level conclusion(结论) from collections of raw data（原始数据）.**
- ***Prediction（预测）*** **— projecting（揭示） probable consequences of given situations.**
- ***Diagnosis*** **— determining the cause of malfunctions（故障）in complex situations based on observable symptoms（症状）.**

8.1 Overview of Expert System Technology

8.1.1 The Design of Rule-Based Expert Systems

- Figure 8.1 Shows the modules that make up a typical expert system.
- The user interacts with the system through a user interface that simplifies communication and hides much of the complexity, such as the internal structure（内部结构） of the rule base.
- The heart of the expert system is the knowledge base（知识库）.

architecture of a typical expert system



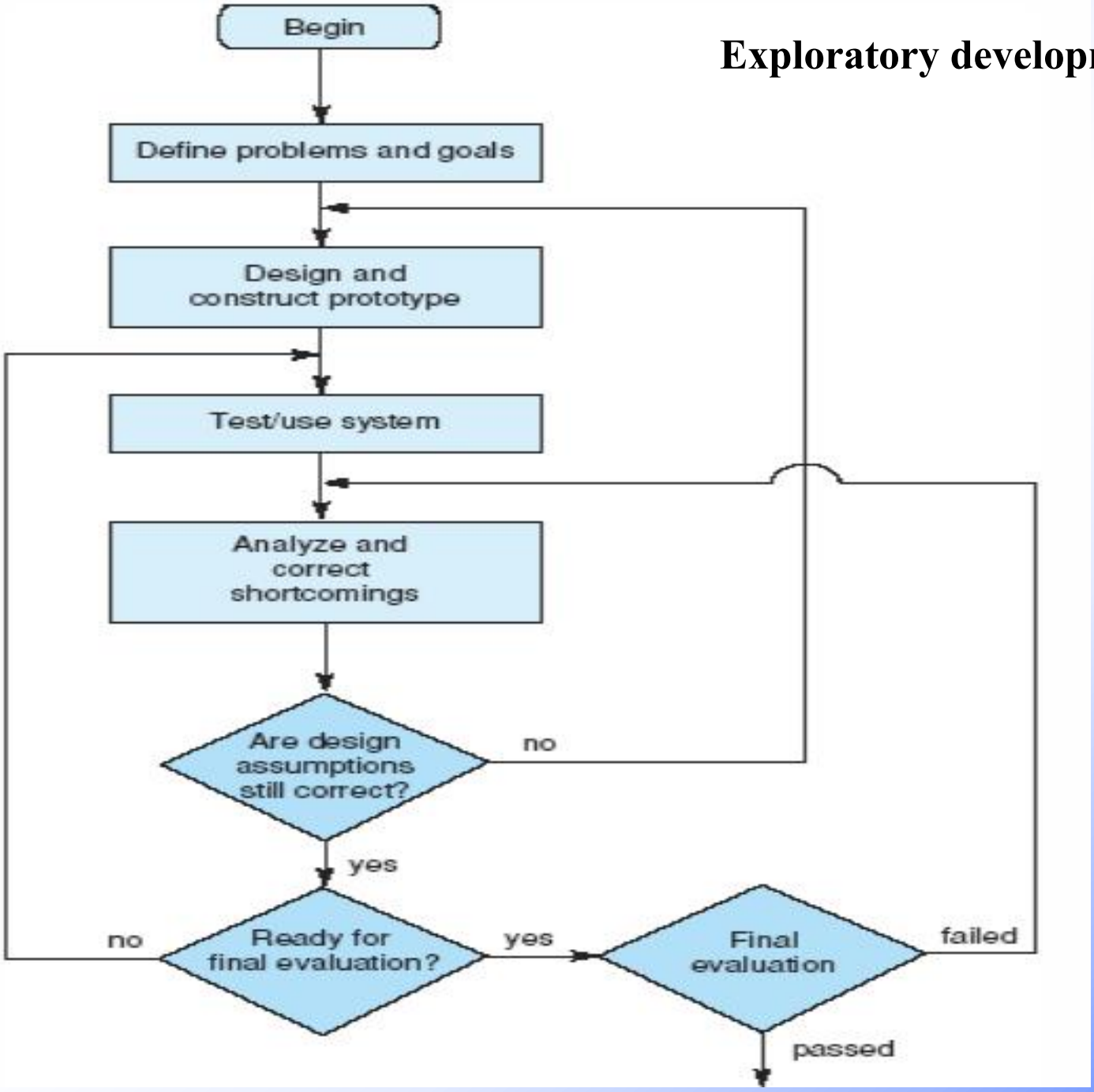
8.1.2 Selecting a Problem and the Knowledge Engineering Process（步骤）

- Guidelines（指导原则）to determine whether a problem is appropriate for expert system :

1. The need（需求） for the solution justifies（证明…合理）the cost and effort of building an expert system.
2. Human expertise is not available in all situations where it is needed.
3. The problem may be solved using symbolic reasoning.

4. The problem domain is well structured and dose not require common sense reasoning（常识推理）.
5. The problem may not be solved using traditional computing methods.
6. Cooperative and articulate（善于表达的） experts exist.（willing and able to share）
7. The problem is of proper size and scope.

Exploratory development cycle



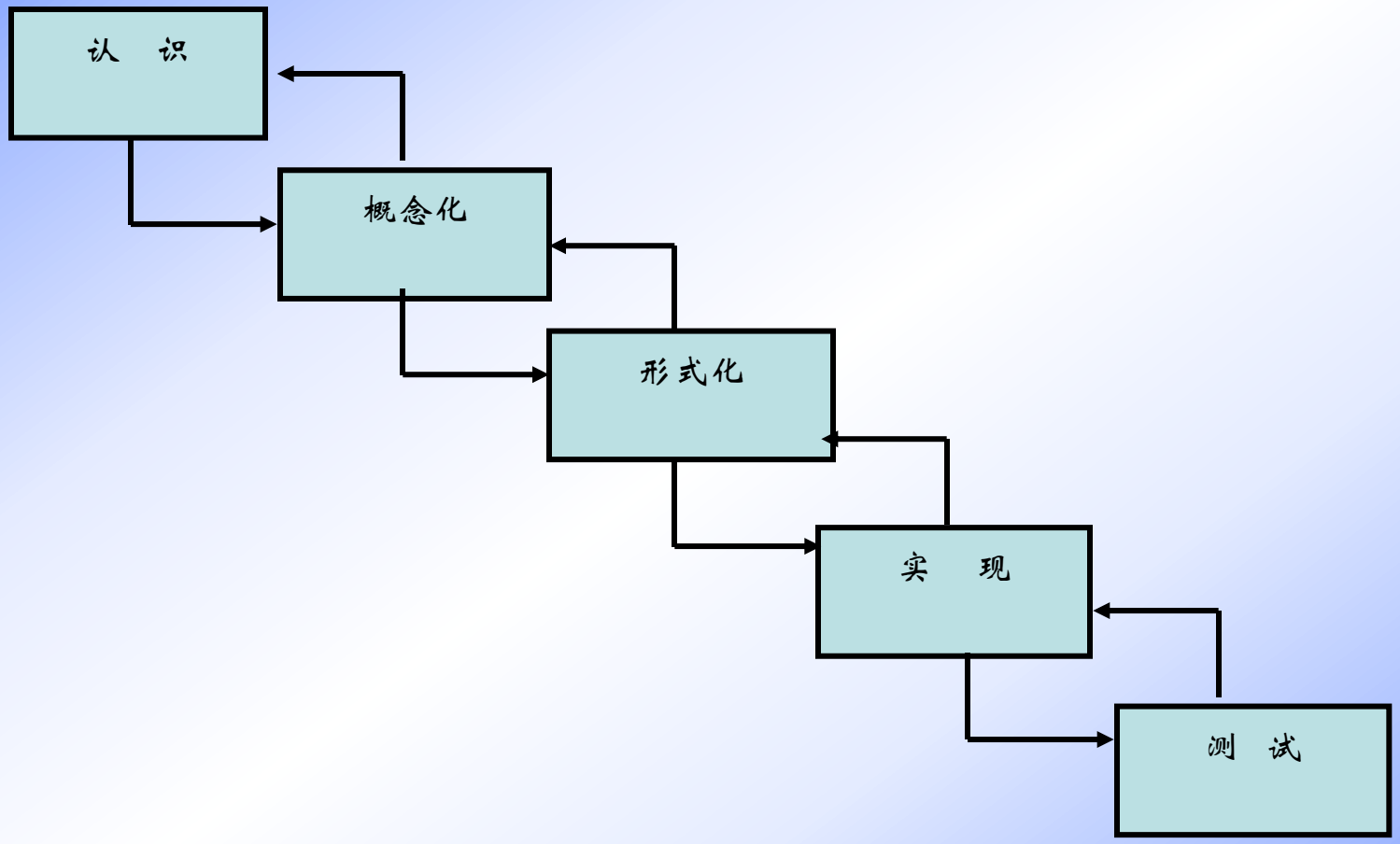
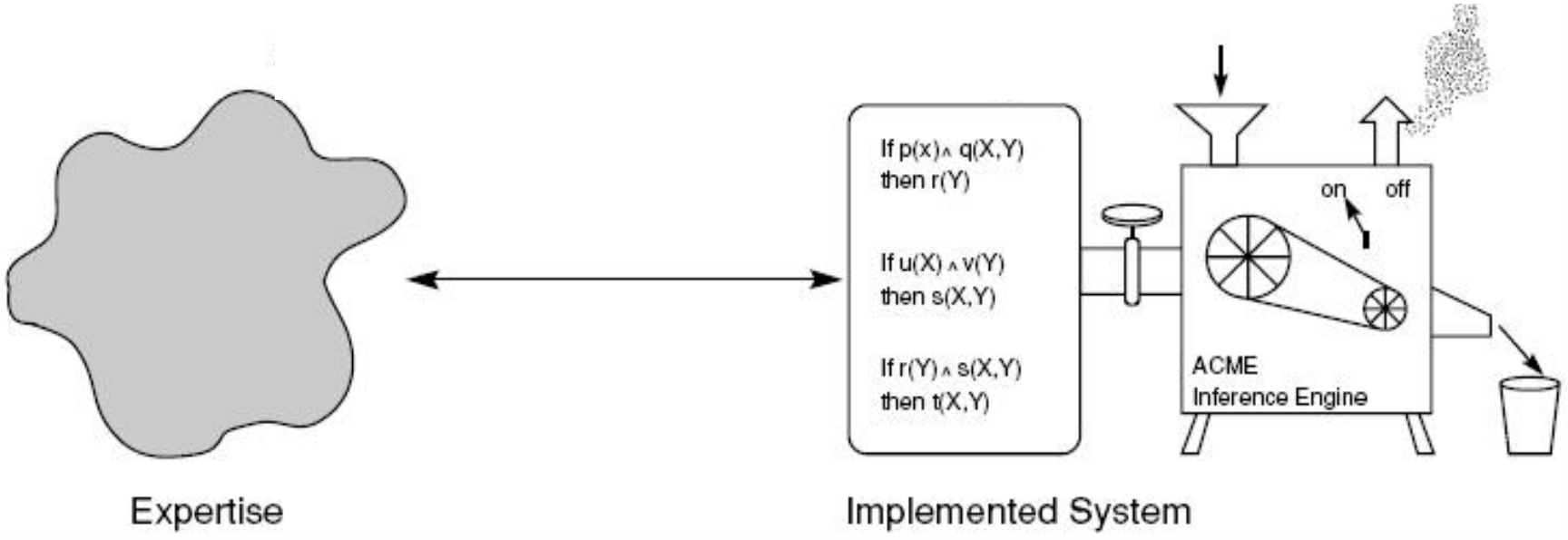


图　专家系统开发过程的瀑布模型

8.1.3 Conceptual Models and Their Role in Knowledge Acquisition

- Figure 8.3 presents a simplified model of the knowledge acquisition process（知识获取过程） that will serve as a useful "first approximation(初步近似)" for understanding the problems involved in acquiring and formalizing（形式化，正式化）human expert performance.

Fig 8.3 The standard view of building an Expert Systems



- A number of important issues arises in the process of formalizing human skilled performance:

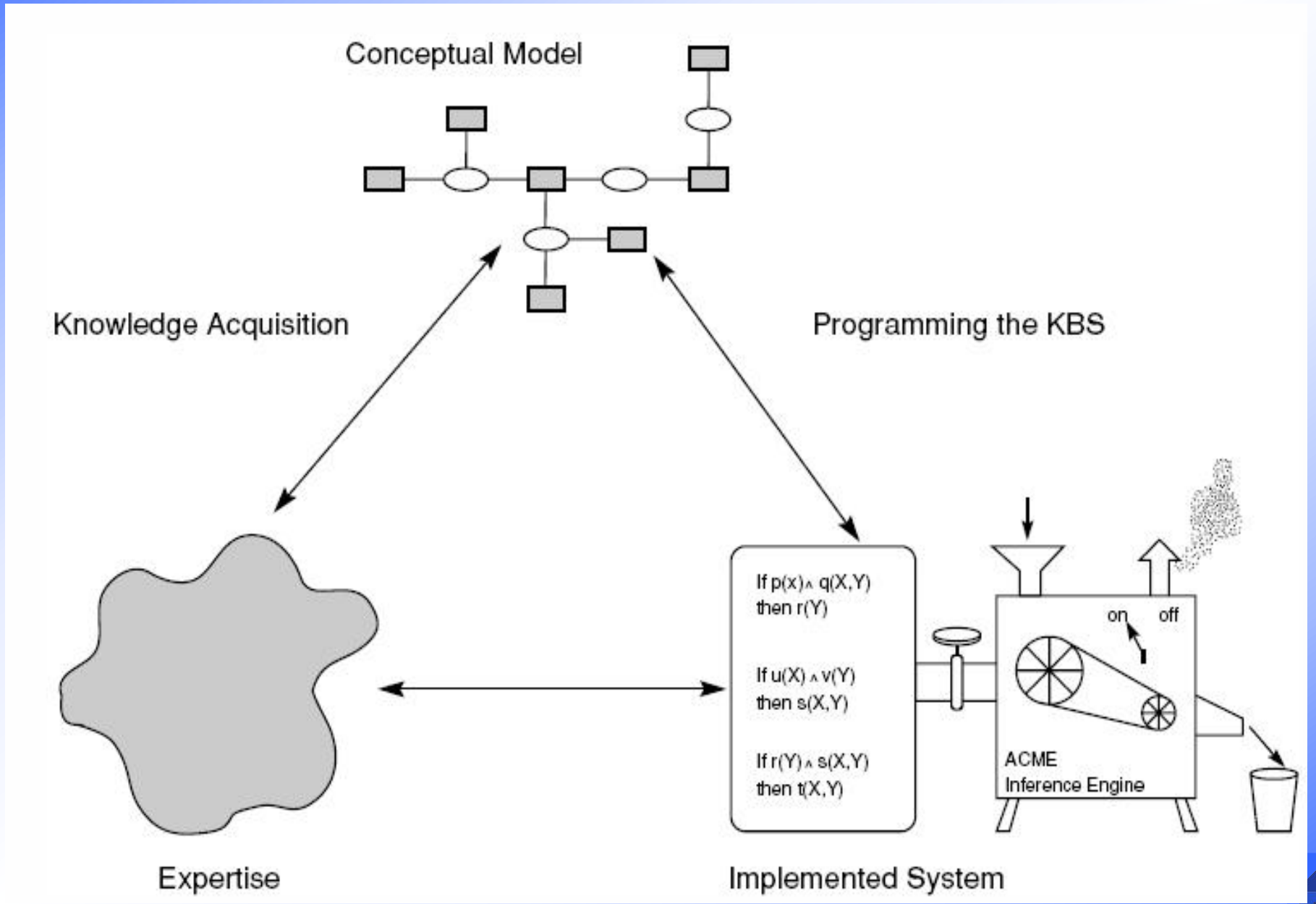
1. Human skill is often inaccessible（难以表述的） to the conscious mind.——下意识的
2. Human expertise often takes the form of knowing how to cope（应对） in a situation rather than knowing what a rational characterization（理性的特征描述）of the situation might be.

Human expertise often takes the form of developing skilled performance mechanisms rather than a fundamental understanding of what these mechanisms are.

3. We often think of knowledge acquisition as gaining factual knowledge of an objective reality, the so-called “real world”.

4. Expertise(专业技能) changes.

Fig 8.4 The role of mental or conceptual models in problem solving



8.2 Rule-Based Expert Systems

- **Rule-based（基于规则的，规则基） expert systems represent problem-solving knowledge as :**

  “ *if* ... *Then* … ” rules.

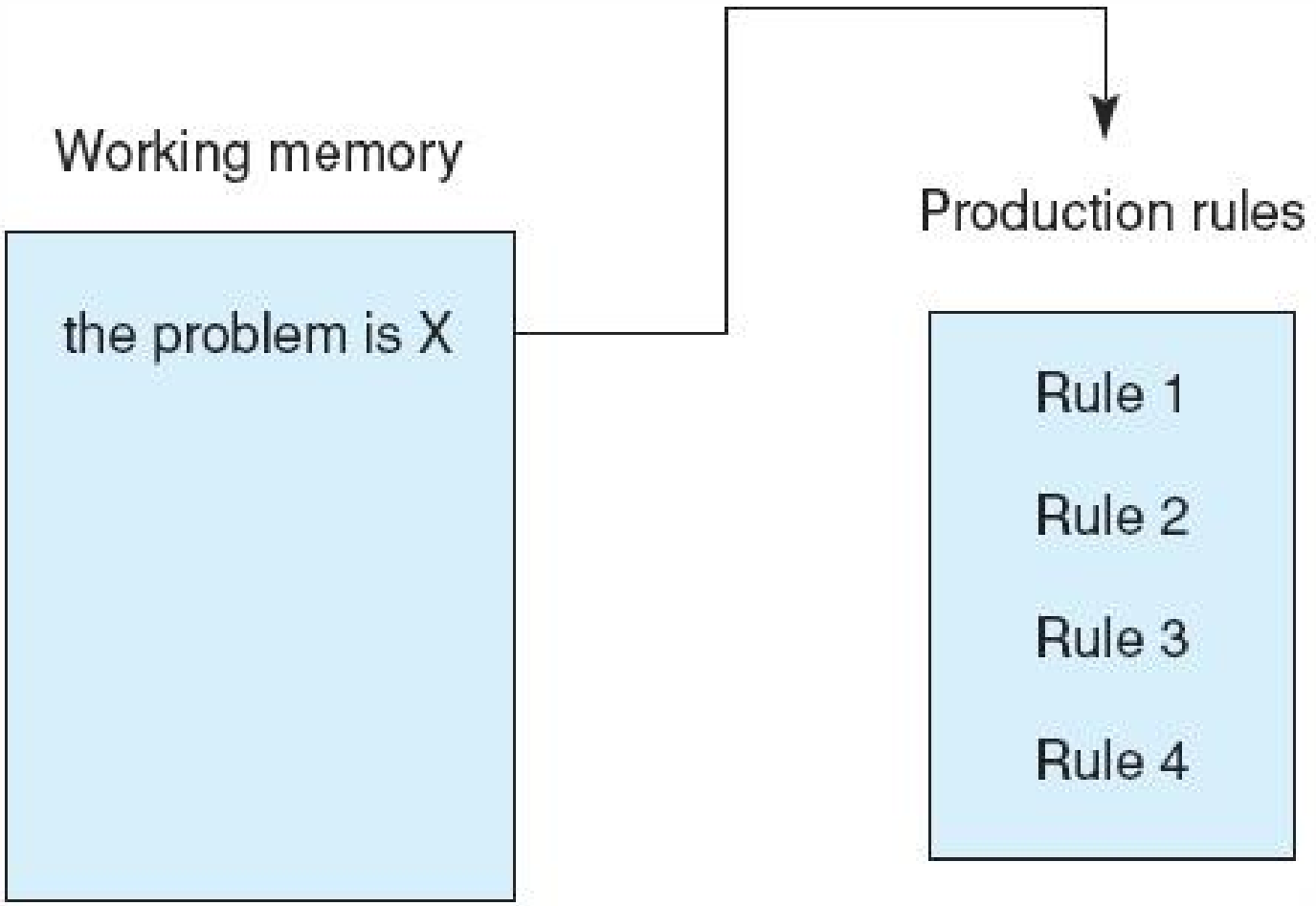
8.2.1 The Production System and Goal-Driven Problem Solving

- **As an example of goal-driven problem solving with user queries（询问）, we next offer a small expert system for analysis of automotive problems.**

- **Rule 1:**
  if
    the engine is getting gas(抽油), and
    the engine will turn over（旋转）,
  then
    the problem is spark plugs（火花塞）.
- **Rule 2:**
  if
    the engine dose not turn over, and
    the lights do not come on
  then
    the problem is battery or cables.

- **Rule 3:**
  if
  the engine does not turn over, and
  the lights do come on
  then
  the problem is the starter motor（启动马达）
- **Rule 4:**
  if
  there is gas in the fuel tank, and
  there is gas in the carburetor（化油器）
  then
  the engine is getting gas.

The production system at the start of a consultation for goal-driven reasoning.



Given the following assertions :

(1) $\forall X(easy(X) \rightarrow like(wang, X))$

(2) $\forall Y(c(Y) \rightarrow easy(Y))$

(3) $c(d)$

Prove : $like(wang, d)$

(a) Solve the problem with data-driven reasoning

(b) Solve the problem with goal-driven reasoning

Fig 8.6 The production system after Rule 1 has fired.

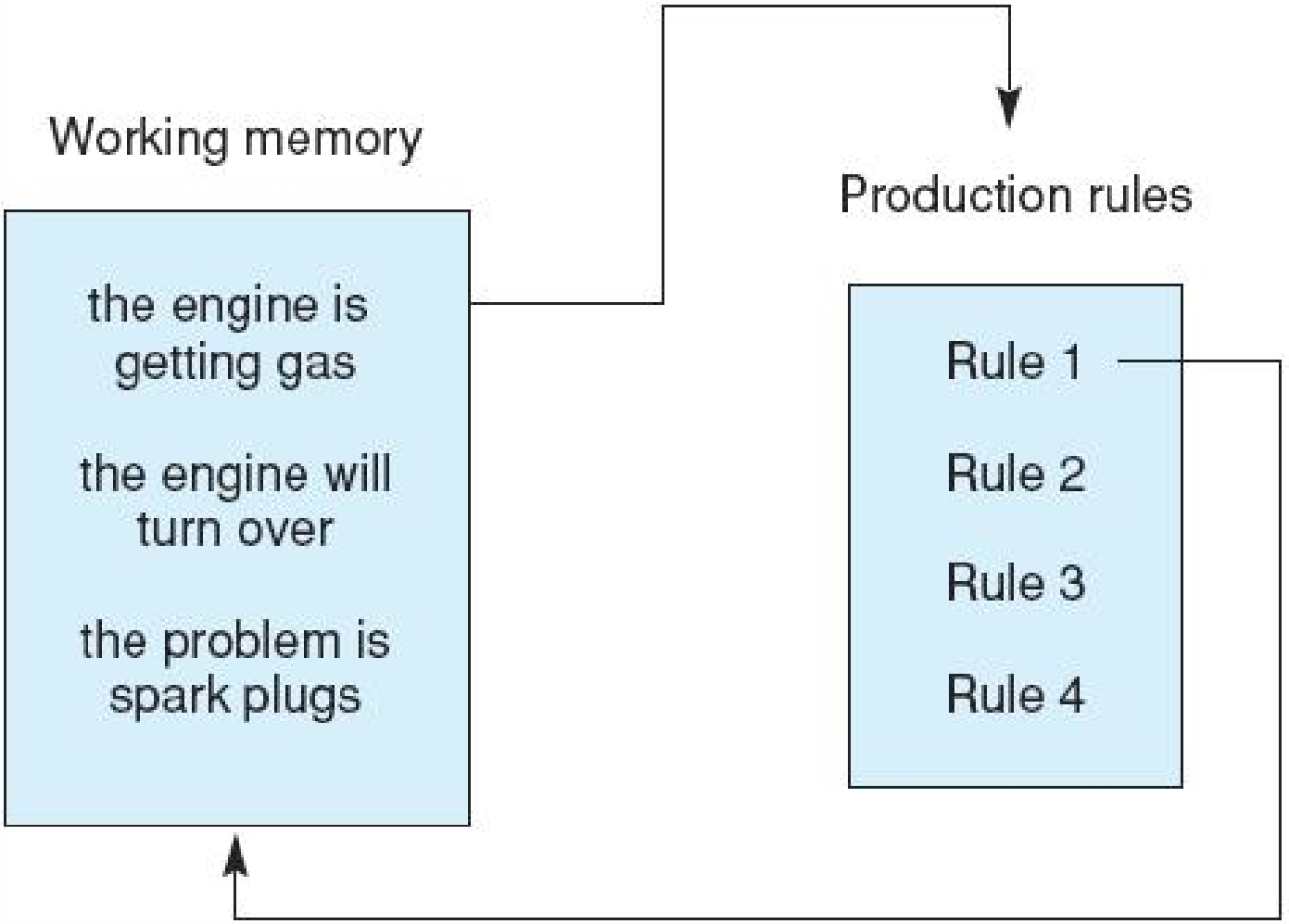
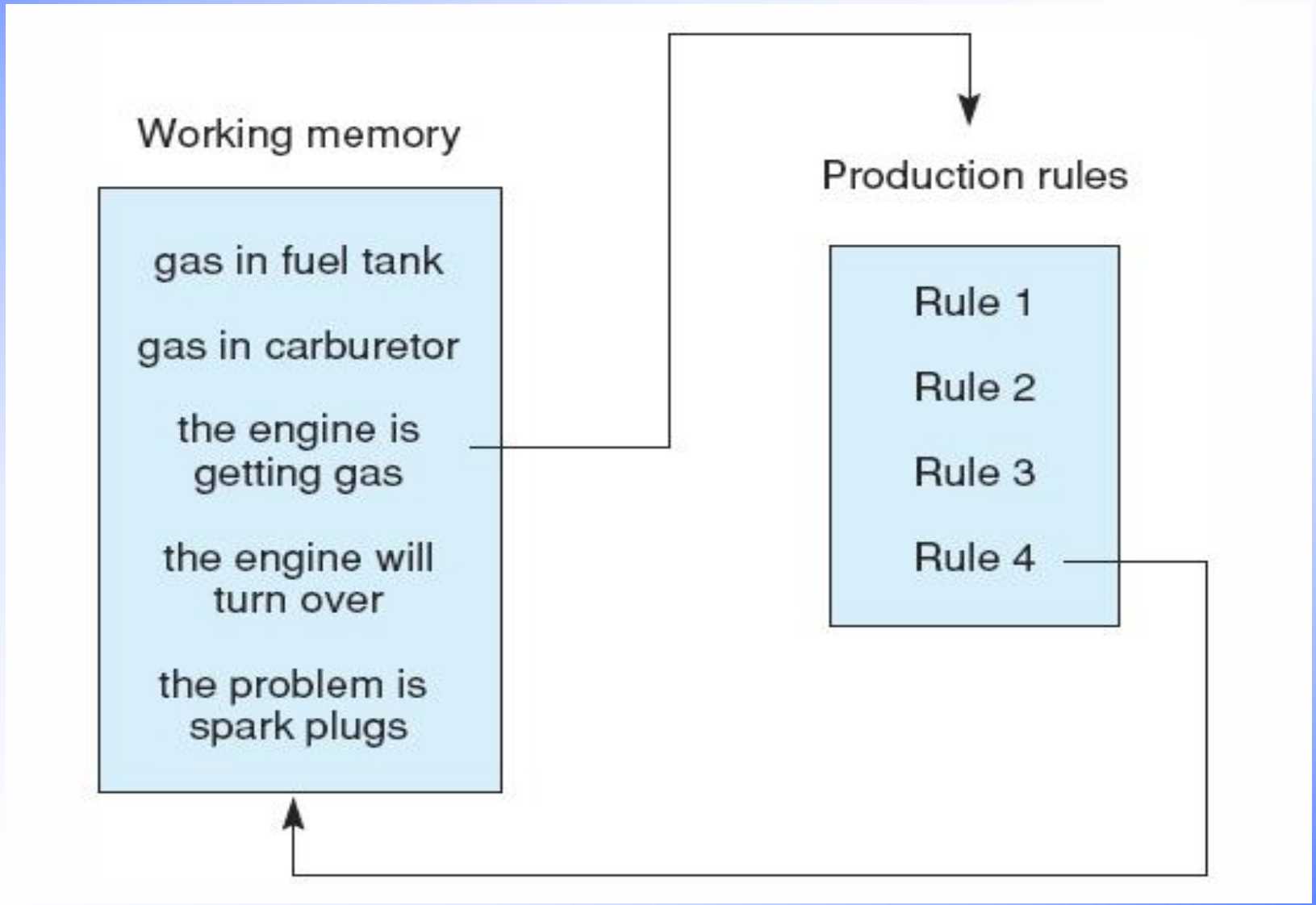
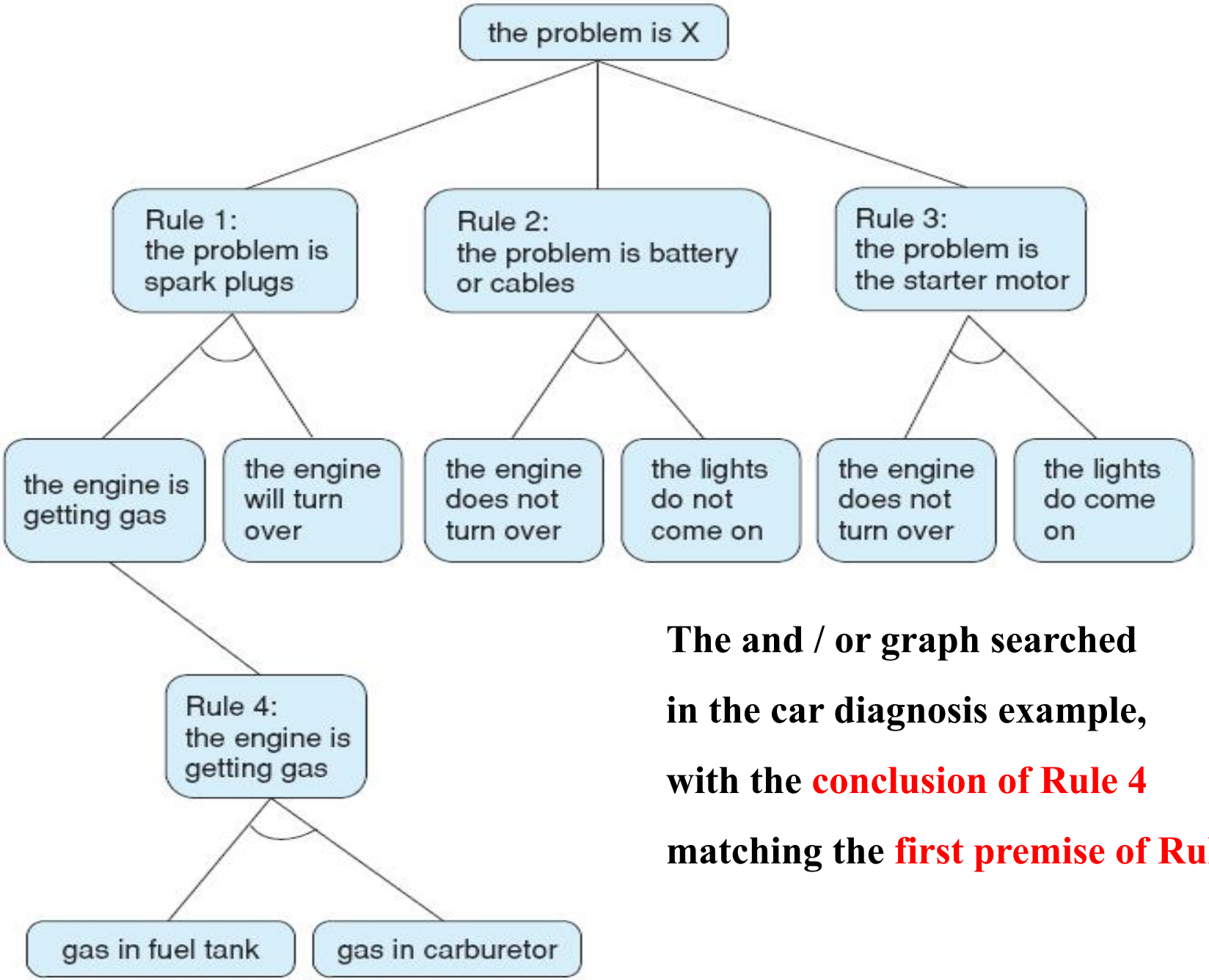


Fig 8.7 The system after Rule 4 has fired.

Note the stack-based approach to goal reduction.





The and / or graph searched in the car diagnosis example, with the conclusion of Rule 4 matching the first premise of Rule 1.

8.2.2 Explanation and Transparency in Goal-Driven Reasoning

- Assume that the consultation has reached the stage as Figure 8.7.
- The following dialogue begins with the computer asking the user about the goals present in working memory.
- User responses are in blue .

gas in fuel tank?
yes
gas in carburetor?
yes
engine will turn over?
why

- **System response :**

It has been established that :

1. **the engine is getting gas,**

therefore if

2. **the engine will turn over,**

then the problem is spark plugs (火花塞).

- **Suppose the user then asks:**

  how the engine is getting gas?

- **System will response :**

  This follows from rule 4:

  if gas in fuel tank, and

  gas in carburetor（化油器）

  then

  engine is getting gas.

  gas in fuel tank was given by the user

  gas in carburetor was given by the user

8.2.3 Using the Production System for Data-Driven Reasoning

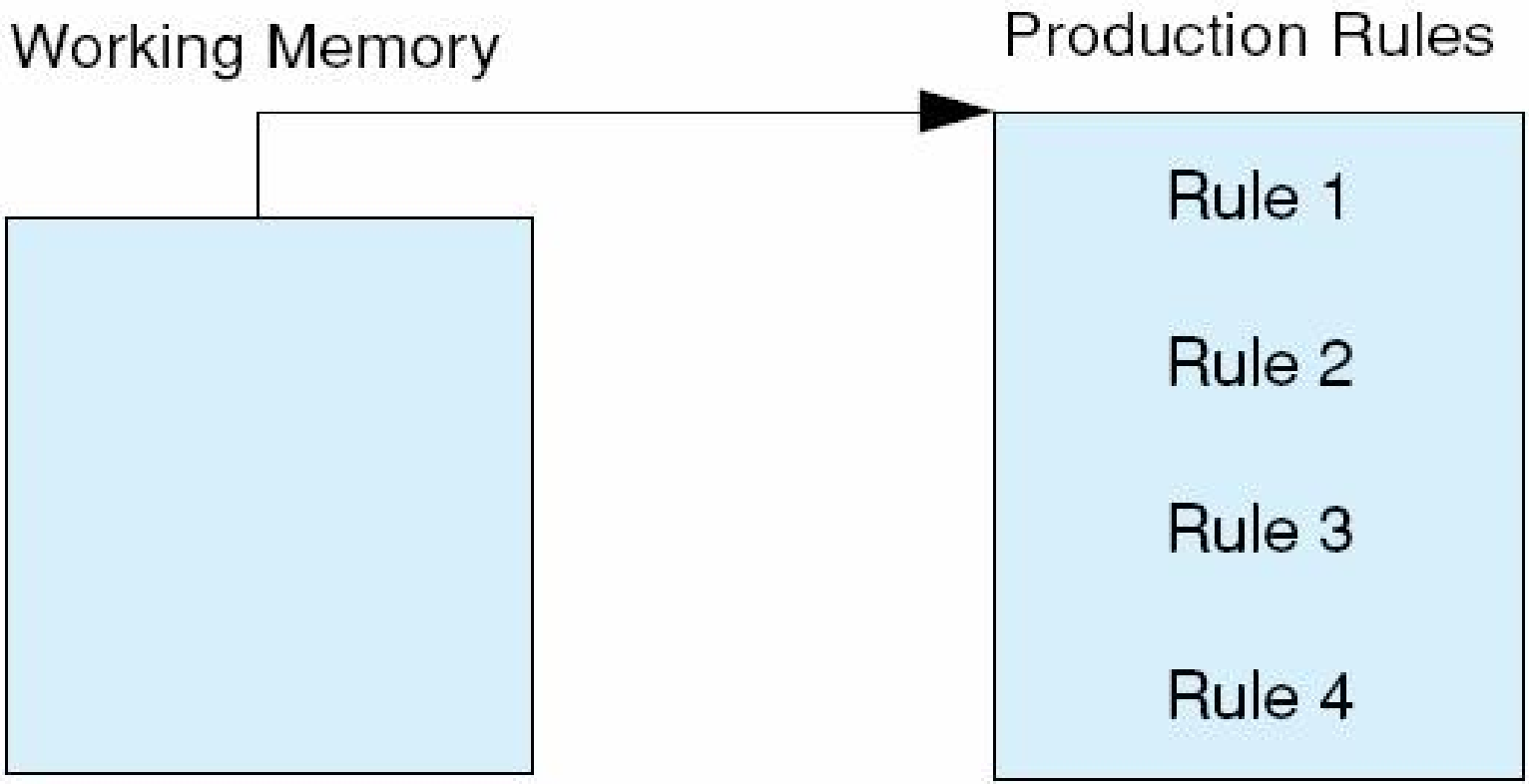
- **Breadth-first** search is even more common in **data-driven** reasoning.
- The algorithm for this is simple:
  1. **compare** the **contents** of working memory with the **conditions** of each rule in the rule base according to the **order** of the rules in the base.
  2. If the **data** in working memory **supports** a rule's **firing**（激发）, the **result** is placed in working memory and then control moves on to the **next rule**.
  3. Once **all rules** have been considered, search **starts again at the beginning** of the rule set.

Note：

- Ask the user when necessary
- If a piece of information that makes up (part of) the premise of a rule is not the conclusion of some other rule, then that fact will be deemed（视为） askable.

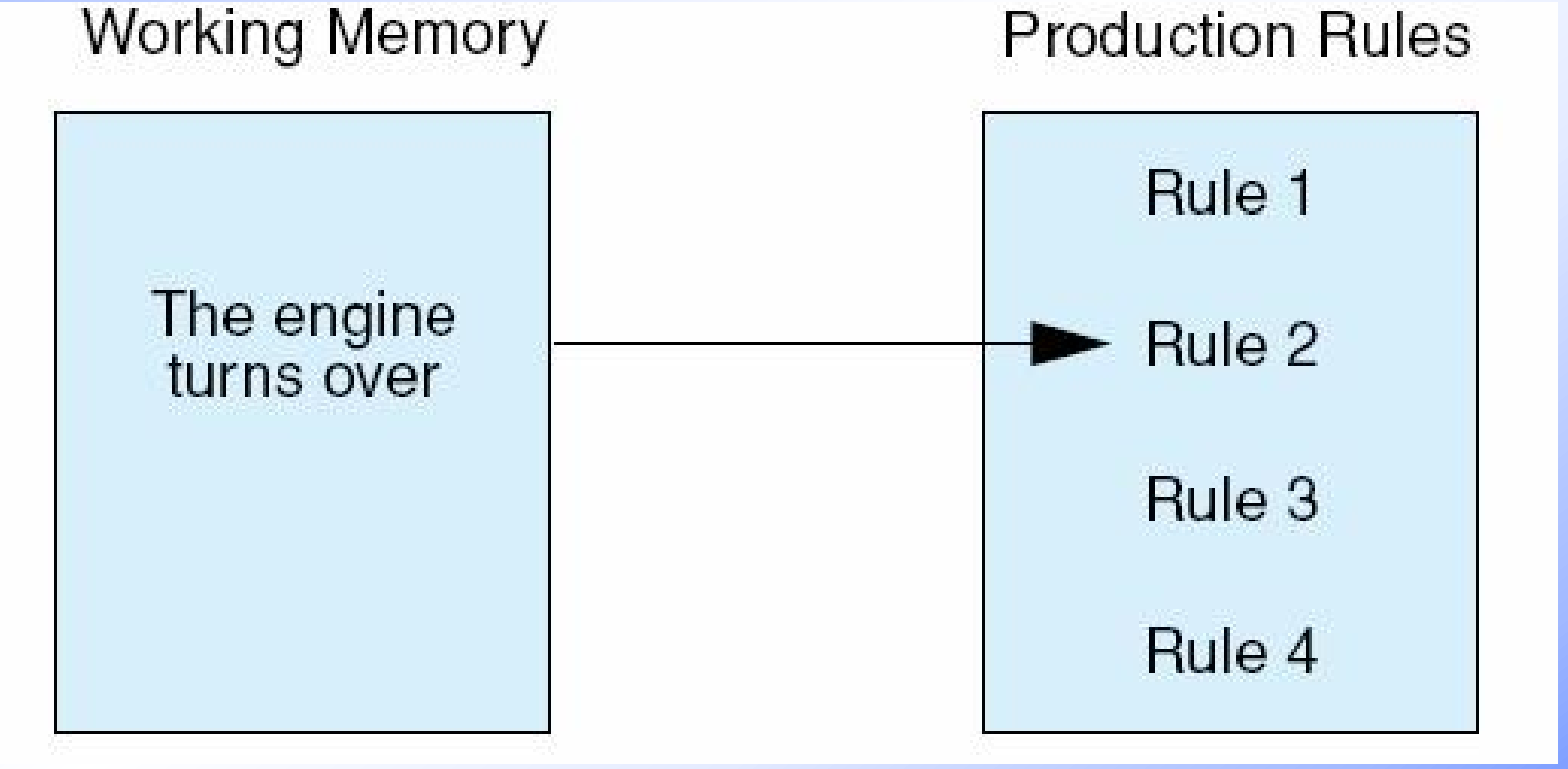
- **Rule 1: if**
  the engine is getting gas, and
  the engine will turn over（旋转）,
  then
  the problem is spark plugs（火花塞）.
- **Rule 2: if**
  the engine dose not turn over, and
  the lights do not come on
  then
  the problem is battery or cables.
- **Rule 3: if**
  the engine does not turn over, and
  the lights do come on
  then
  the problem is the starter motor（启动马达）.
- **Rule 4: if**
  there is gas in the fuel tank, and
  there is gas in the carburetor（化油器）
  then
  the engine is getting gas.

The production system at the start of a consultation for data-driven reasoning.



The production system

after evaluating the first premise of Rule 2, which fails.



The data-driven production system

after considering Rule 4, which succeeds,

beginning its second pass through the rules.

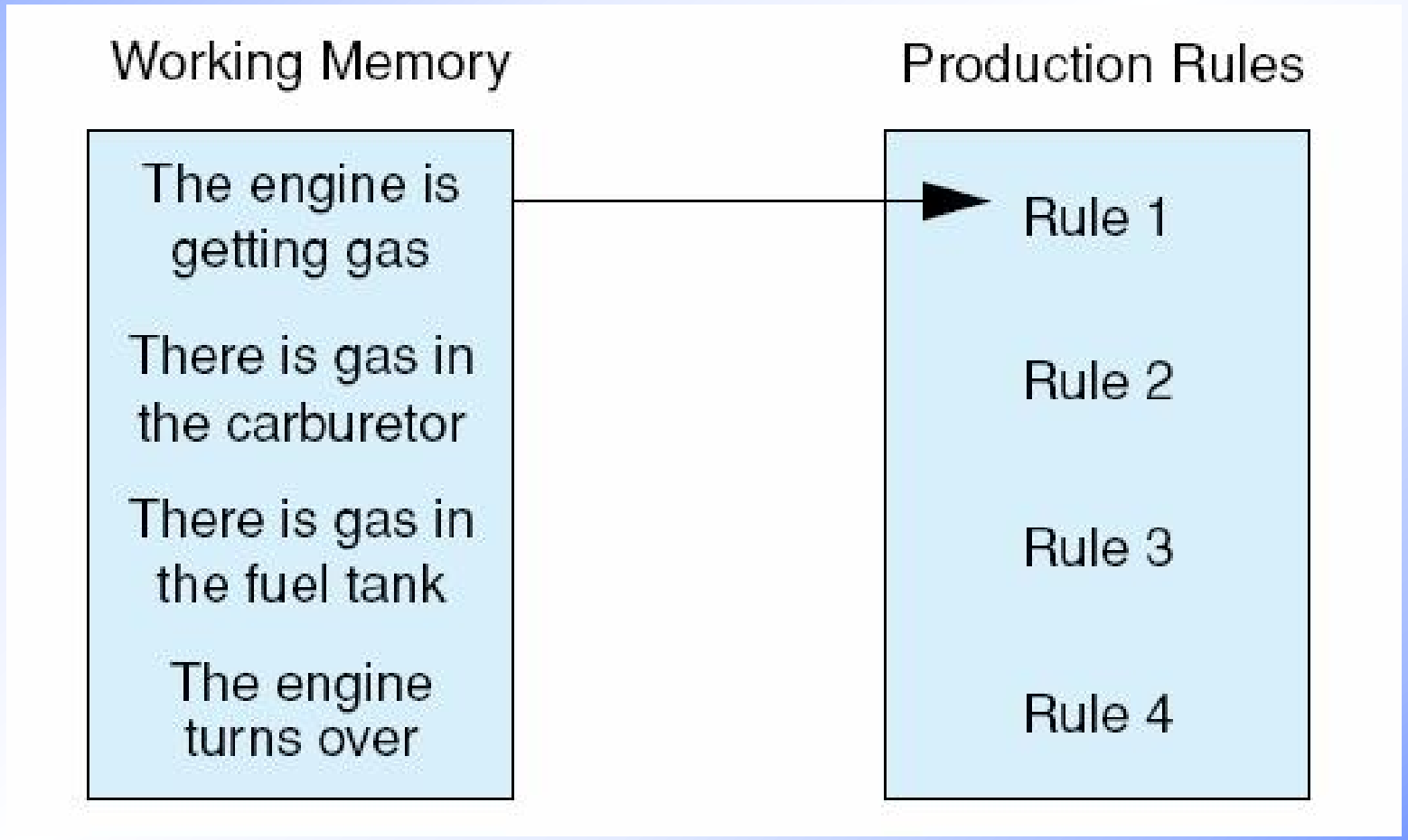
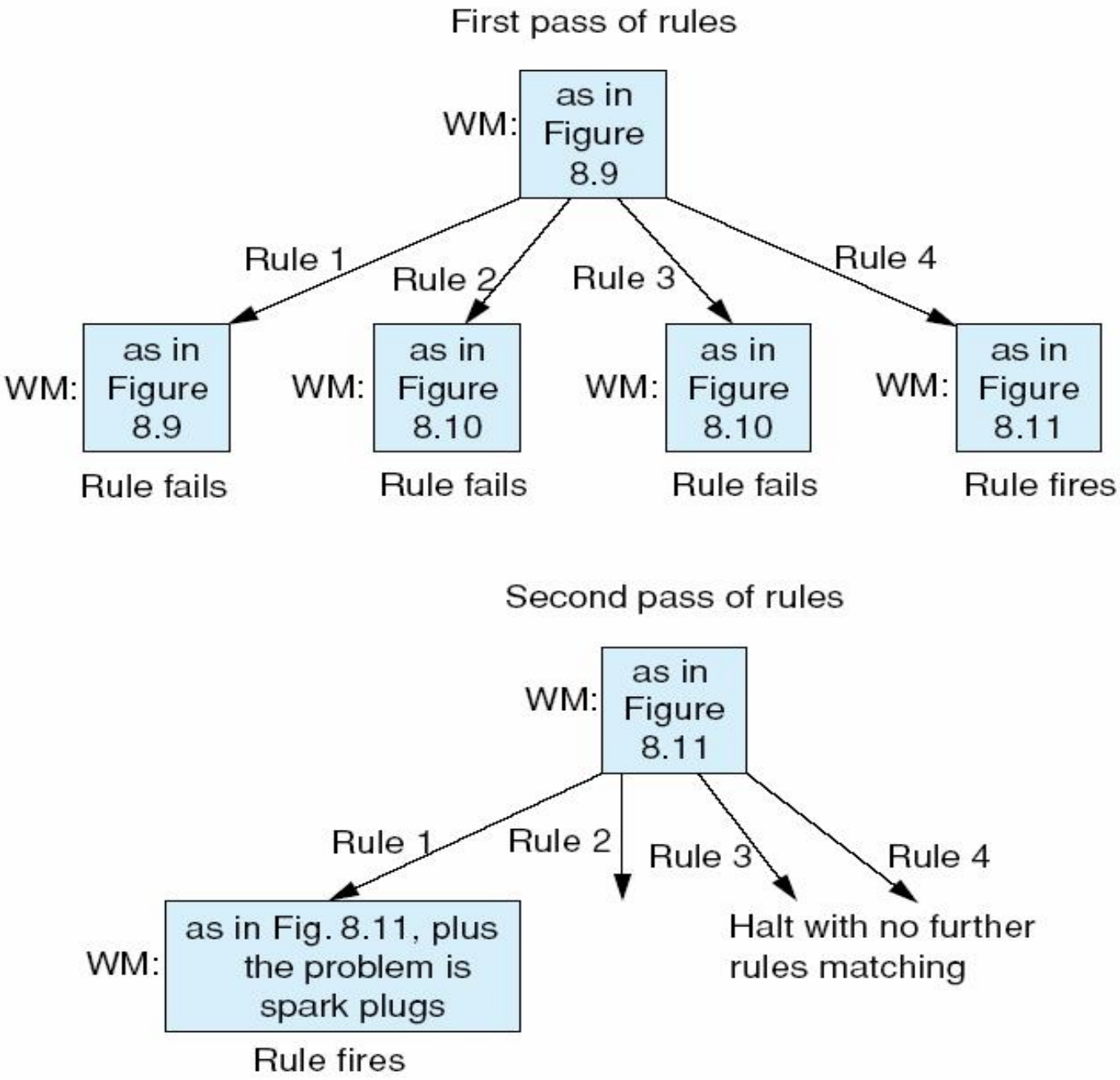


Fig 8.12 (on next page)

The search graph as described by the contents of working memory (WM) for the data-driven breadth-first search of the rule set of Section 8.2.1


First pass of rules
WM:
as in Figure 8.9
Rule 1
Rule 2
Rule 3
Rule 4
WM:
as in Figure 8.9
Rule fails
WM:
as in Figure 8.10
Rule fails
WM:
as in Figure 8.10
Rule fails
WM:
as in Figure 8.11
Rule fires
Second pass of rules
WM:
as in Figure 8.11
Rule 1
Rule 2
Rule 3
Rule 4
WM:
as in Fig. 8.11, plus the problem is spark plugs
Rule fires
Halt with no further rules matching

8.2.4 Heuristics and Control in Expert Systems

- Because of the separation of the knowledge base and the inference engine, and the fixed control regimes（机制）provided by the inference engine, an important method for the programmer to control search is through the structuring and ordering of the rules in the knowledge base.
- Thus the planning of rule order, the organization of a rule premises, and the costs of different tests（检测）are all fundamentally heuristic in nature（本质上）.

Control of Search through Rule Structure

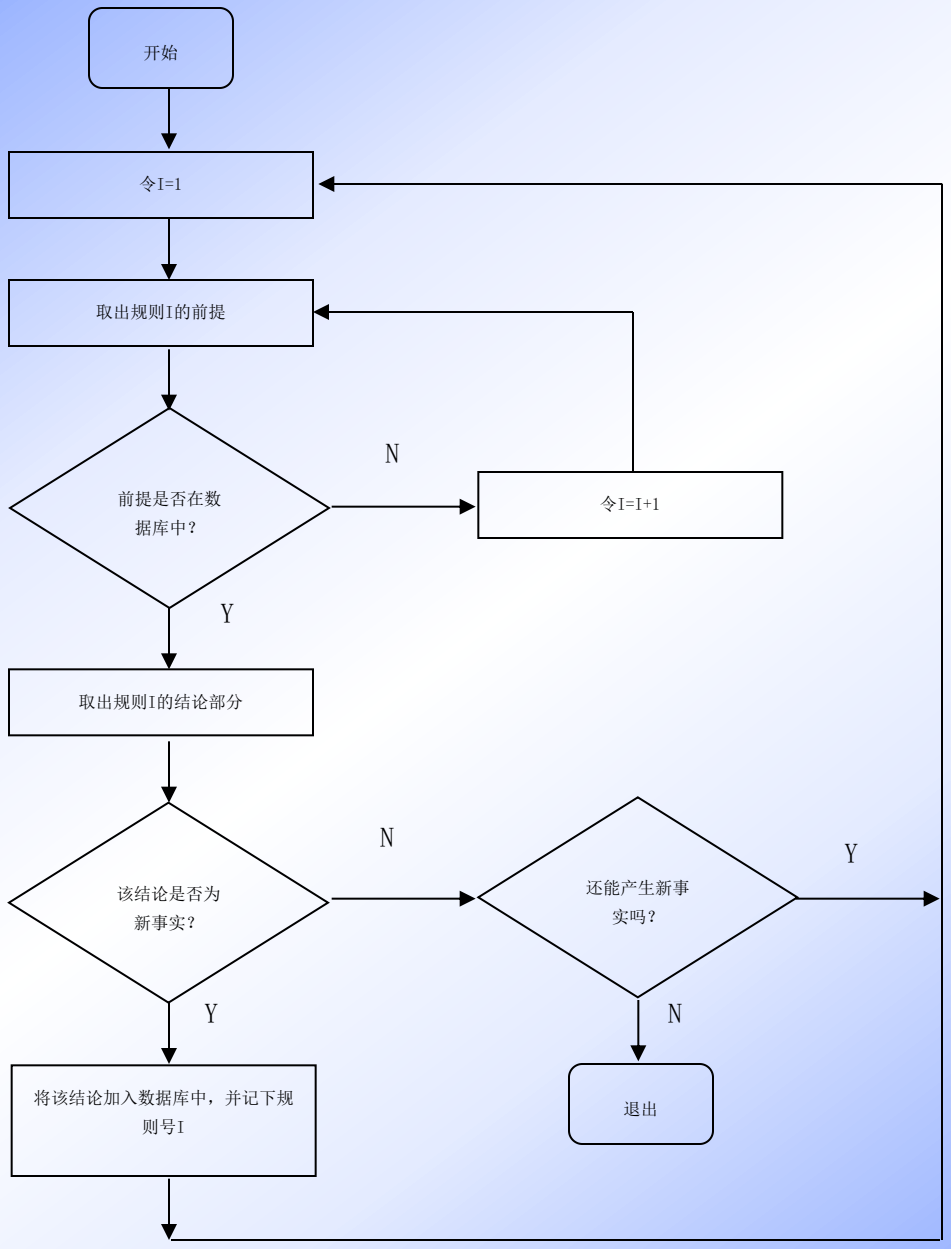
- The structure of rules in a production system, including the distinction between the condition and the action, determines the fashion in which the space is searched.
- EXAMPLE two logically equivalent rules :

  $\forall X (foo(X) \wedge goo(X) \rightarrow moo(X))$

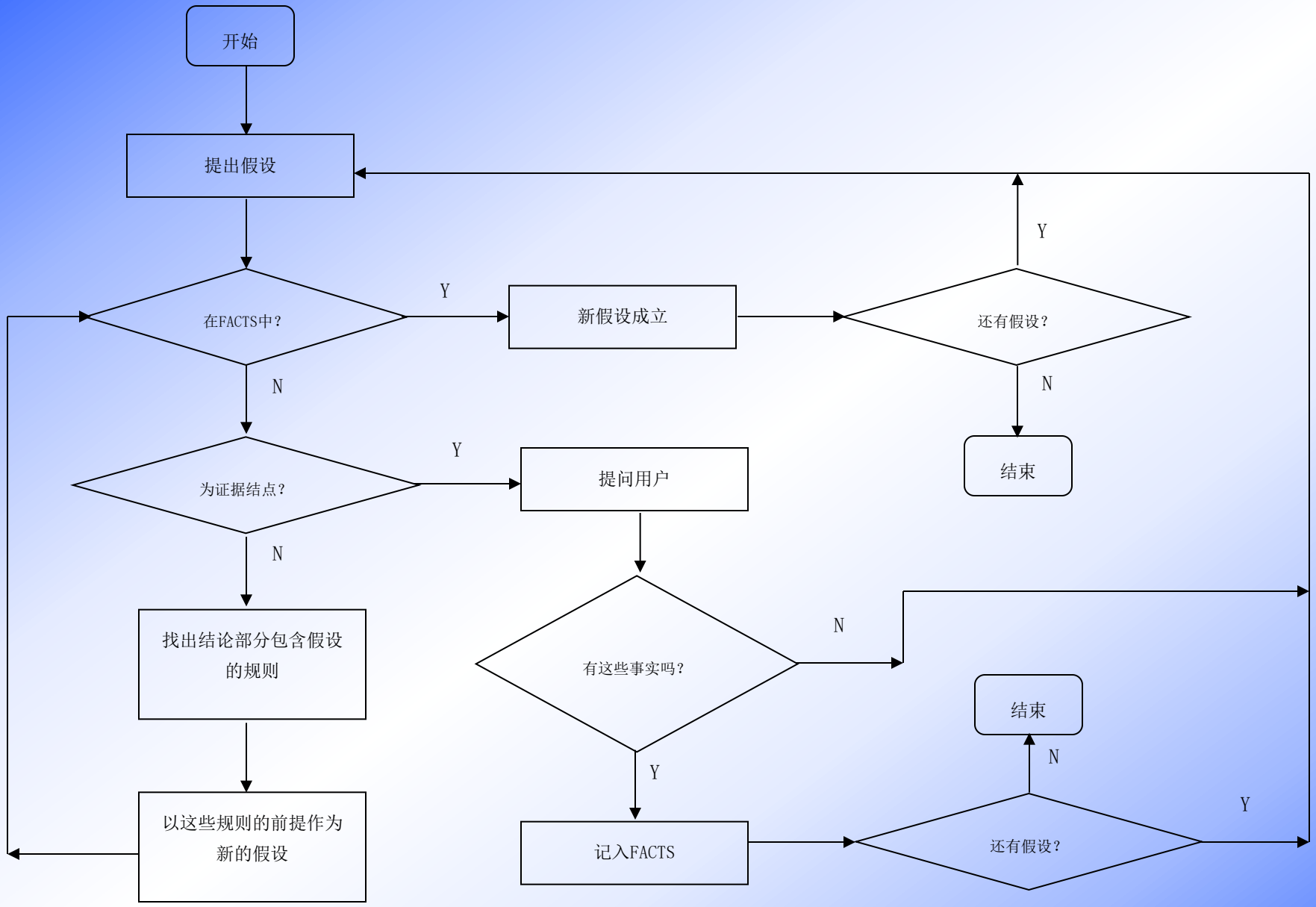
  $\forall X (foo(X) \rightarrow moo(X) \vee \neg goo(X))$

- They don't have the same behavior in the search implementation.
- The order in which conditions are tried also determines the search fashion.

例如：当一个机修工说“如果引擎不转，并且灯不亮，那么检查电池”时，他指出了动作的特定顺序。一条逻辑上等价的语句“引擎转动或者车灯亮或者检查电池”并没有捕捉到这种信息。



正向(Data-Driven Reasoning)推理示意图


开始
提出假设
在FACTS中？
Y
新假设成立
还有假设？
Y
N
结束
N
为证据结点？
Y
提问用户
N
找出结论部分包含假设的规则
有这些事实吗？
N
结束
Y
N
以这些规则的前提作为新的假设
记入FACTS
还有假设？
Y

反向推理示意图

基于规则专家系统应用举例

——动物识别专家系统

建立动物识别专家系统的规则库，并用与/或图描述这个规则库。规则库由15条规则组成，规则名分别是rule1，rule2，…，rule15，规则库的符号名为rules。

- 把若干条规则写入一个规则库的最简单方法是使用setq函数，直接把若干条规则组成一个表赋值给一个规则库符号名。

1.若动物有毛发（F_1），则动物是哺乳动物（M_1）。

2.若动物有奶（F_2），则动物是哺乳动物（M_1）。

3.若动物有羽毛（F9），则动物是鸟（M4）。

4.若动物会飞（F10），且生蛋（F11），则动物是鸟（M4）。

5.若动物吃肉（F3），则动物是食肉动物(M2)。

6.若动物有犀利牙齿(F4)，且有爪(F5)，且眼向前方(F6)，则动物是食肉动物(M2)。

7.若动物是哺乳动物（M1），且有蹄（F7，则动物是蹄类动物（M3）。

8.若动物是哺乳动物（M1），且反刍（F8，则动物是蹄类动物（M3）。

9.若动物是哺乳动物（M1），且是食肉动物(M2），且有黄褐色（F12），且有暗斑点（F13），则动物是豹（H1）。

10.若动物是哺乳动物（M1），且是食肉动物(M2)，且有黄褐色（F12），且有黑色条纹（F15），则动物是老虎（H2）。

11.若动物是蹄类动物（M3），且有长脖子（F16），且有长腿（F14），且有黑斑点（F13），则动物是长颈鹿（H3）。

12.若动物是蹄类动物（M3），且有黑色条纹（F15），则动物是斑马（H4）。

13.若动物是鸟（M4），且不能飞（F17），且有长脖子（F16），且有长腿（F14），且有黑白二色(F18)，则动物是鸵鸟（H5）。

14.若动物是鸟（M4），且不能飞（F17），且会游泳（F19），且有黑白二色(F18)，则动物是企鹅（H6）。

15.若动物是鸟（M4），且善飞（F20），则动物是信天翁(H7)。

R_1: $F_1 \rightarrow M_1$

R_2: $F_2 \rightarrow M_1$

R_3: $F_9 \rightarrow M_4$

R_4: $F_{10} \wedge F_{11} \rightarrow M_4$

R_5: $F_3 \rightarrow M_2$

R_6: $F_4 \wedge F_5 \wedge F_6 \rightarrow M_2$

R_7: $F_7 \wedge M_1 \rightarrow M_3$

R_8: $F_8 \wedge M_1 \rightarrow M_3$

R_9: $F_{12} \wedge F_{13} \wedge M_1 \wedge M_2 \rightarrow H_1$

R_{10}: $F_{12} \wedge F_{15} \wedge M_1 \wedge M_2 \rightarrow H_2$

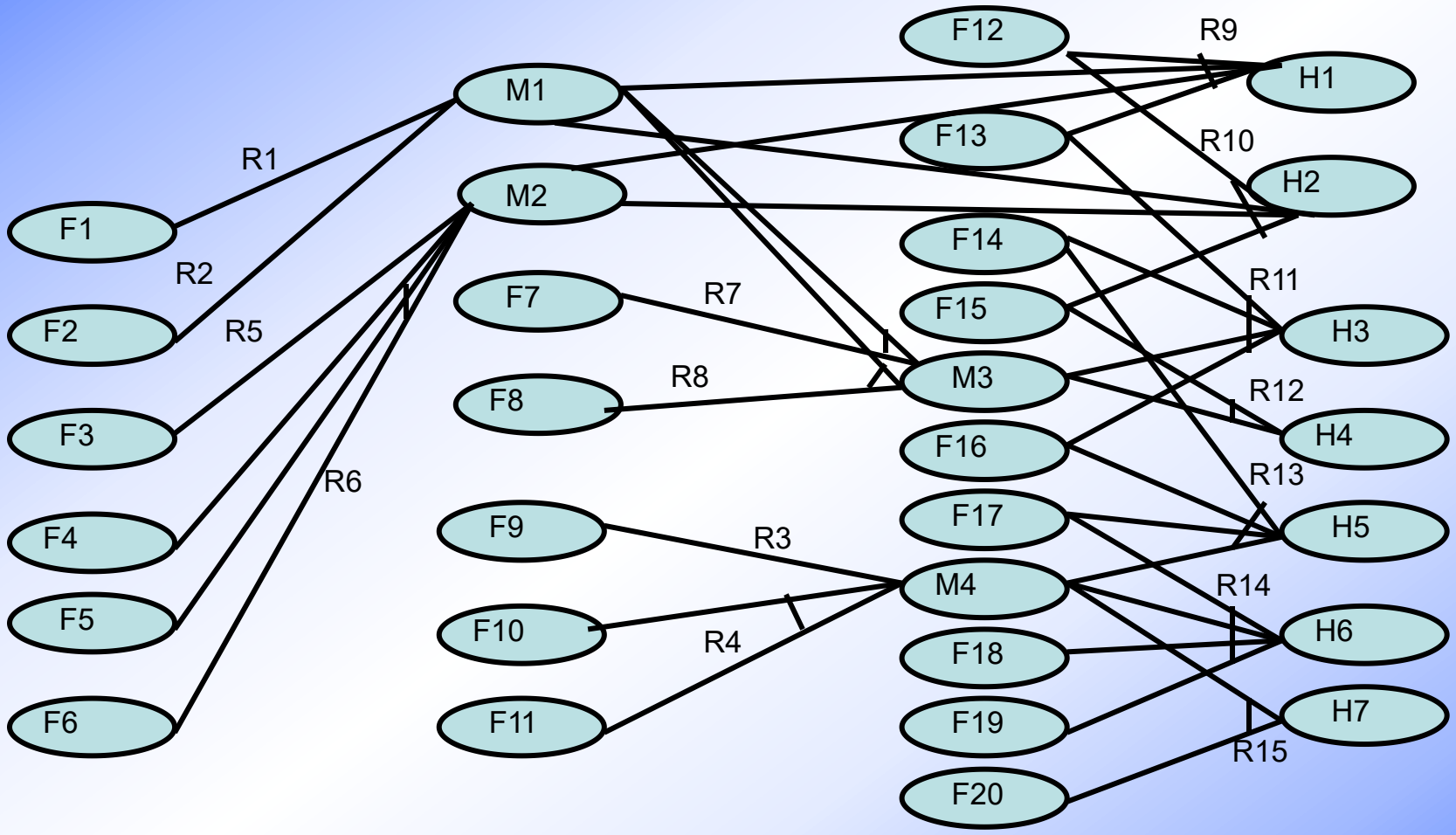
R_{11}: $F_{13} \wedge F_{14} \wedge F_{16} \wedge M_3 \rightarrow H_3$

R_{12}: $F_{15} \wedge M_3 \rightarrow H_4$

R_{13}: $F_{14} \wedge F_{16} \wedge F_{17} \wedge F_{18} \wedge M_4 \rightarrow H_6$

R_{14}: $F_{17} \wedge F_{18} \wedge F_{19} \wedge M_4 \rightarrow H_6$

R_{15}: $F_{20} \wedge M_4 \rightarrow H_7$



动物识别专家系统规则库与/或图

仿真实验

对于动物识别专家系统，若已知初始事实是F_{13}，F_{12}，F_3和 F_1，应用正向推理机（Data-Driven Reasoning），说明推理过程和得出推理结论。

解： 使用setq函数把已知的初始事实赋值给事实表facts：

```
(setq facts
   ((animal has dark spots)
    (animal has tawny color)
    (animal eats meat)
    (animal has hair)))
```

即有：facts=（F_{13} F_{12} F_3 F_1）

1）在rules中查找规则前件的全部条件在当前facts=（F_{13} F_{12} F_3 F_1）中的可用规则，首先找到规则R_1，则把R_1后件中不在facts中的结论M_1添加到facts中，扩充facts为facts=（F_{13} F_{12} F_3 F_1 M_1）。

2）对当前facts在rules中查找可用规则，仍然找到规则R_1，但R_1的后件结论M_1已在facts中，因此不会执行规则R_1。继续查找可用规则，找到规则R_5，因为R_5的后件结论M_1不在当前facts中，故执行R_5，把R_1 R_5不在facts中结论后件M_2添加到facts中，扩充facts为facts=（F_{13} F_{12} F_3 F_1 M_1 M_2）。

3）对当前facts在rules中继续查找可用规则，只有规则R_9的前件包含的全部条件在facts，因此，R_9是可用规则，且R_9的结论H_1不在facts中，故执行R_9，把R_9的结论H_1扩充到facts中，得facts=（F_{13} F_{12} F_3 F_1 M_1 M_2 H_1）。

4）对当前facts，在rules中找不到规则的前件包含的全部条件在facts中且后件有不在facts中的结论的任何规则，至此，正向推理终止，推理机函数deduce返回的变量progrss的值为t。

- 实际上，这个推理过程是按规则库与/或图的一个子图的正向进行推理的。
- 推理过程终止时，由当前facts中的最后一个元素可得出推理的结论是H_1，即（animal is cheetah）。

实例分析——

汽车故障诊断（用产生式系统实现）

- 图中给出了汽车故障诊断领域所发现的一些直觉形式知识片段。

汽车故障诊断领域的知识片段

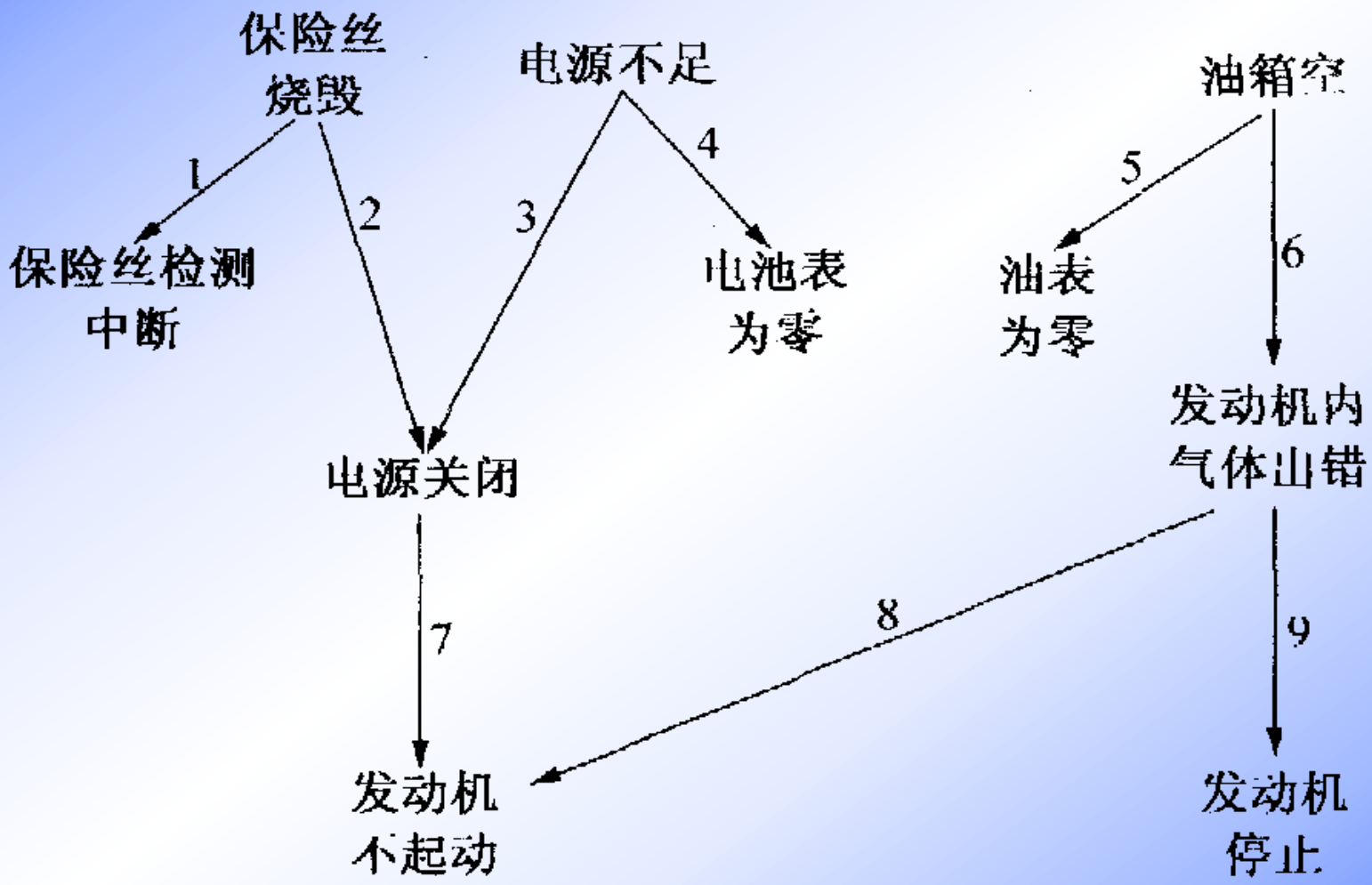


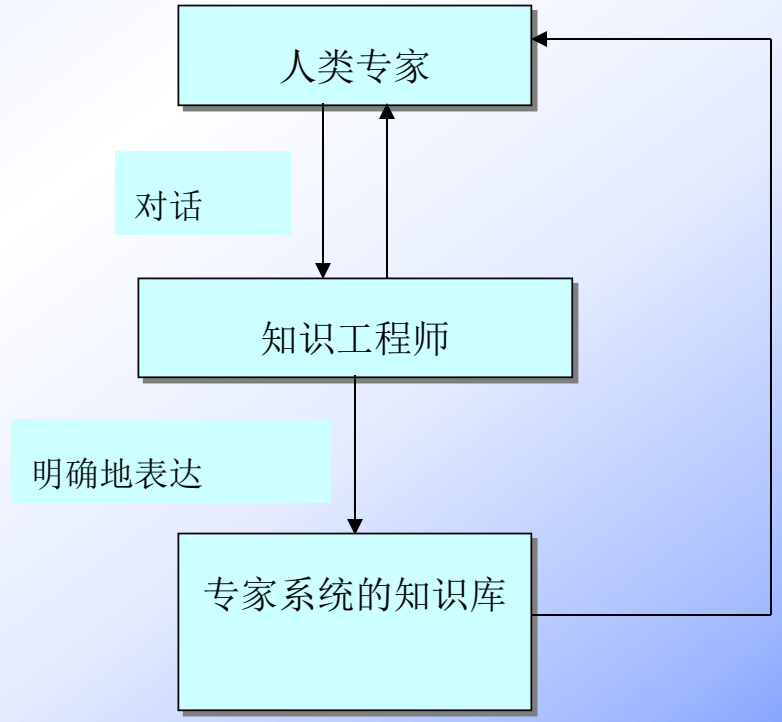
图2. 汽车故障诊断领域的知识片段

一般应用程序与专家系统的区别

- 前者把问题求解的知识隐含地编入程序，而后者则把其应用领域的问题求解知识单独组成一个实体, 即为知识库。
- 更明确地说，一般应用程序把知识组织为两级：数据级和程序级；
- 专家系统则将知识组织成三级；数据、知识库和控制。

知识获取

- 静态知识模型的建立
- 知识库的建立



术语标注和概念抽象

一、 可观察的方面:

保险丝检测、电池表、油表

二、汽车状态:

- 不可见状态：保险丝、电池、油箱、电源、发动机汽体
- 可见状态：发动机行为。

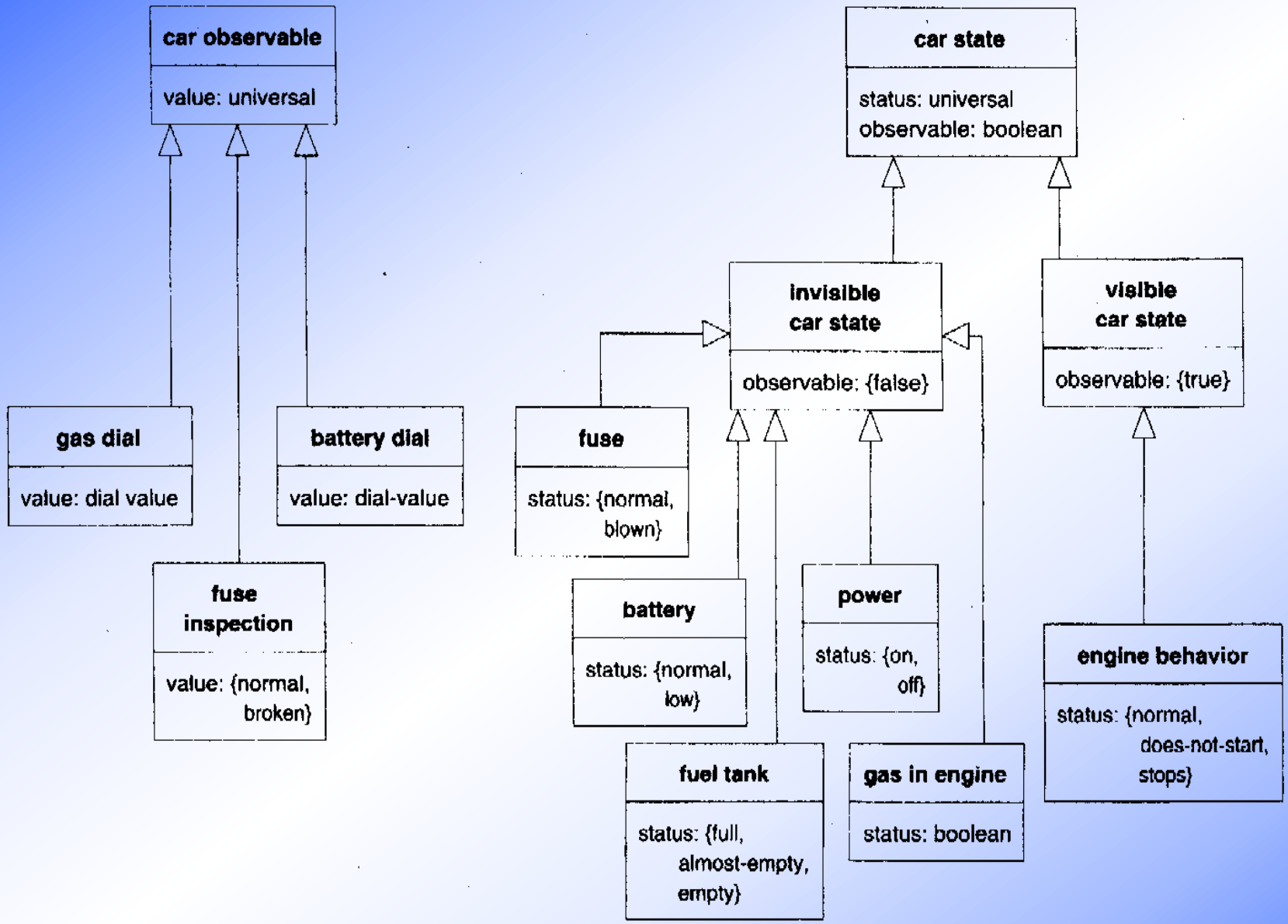


图2. 汽车诊断领域中概念之间的子类型关系

规则的形式

从图中得出汽车状态间依赖性规则：

规则6：**FUEL-TANK. status=empty==>**

GAS-IN-ENGINE. status = false

规则3：**BATTERY. status=low==>**

POWER. status=off

规则类型1:对应图1中的6个箭头（2-3和6-9）。

RULE-TYPE state-dependency;

ANTECEDENT: invisible-car-state;

CARDINALITY: 1;

CONSEQUENT: car-state;

CARDINALITY: 1;

CONNECTION—SYMBOL:

causes;

END RULE-TYPE state-dependency;

规则类型2：对应图1中的3个箭头（1、4、5）

RULE-TYPE manifestation-rule;

DESCRIPTION: “Rule stating the relation between an internal state and its external behavior in terms of observable value”;

ANTECEDENT:

invisible-car-state;

CONSEQUENT:

car-observable;

CONNECTION-SYMBOL:

has-manifestation;

END RULE-TYPE manifestation-rule;

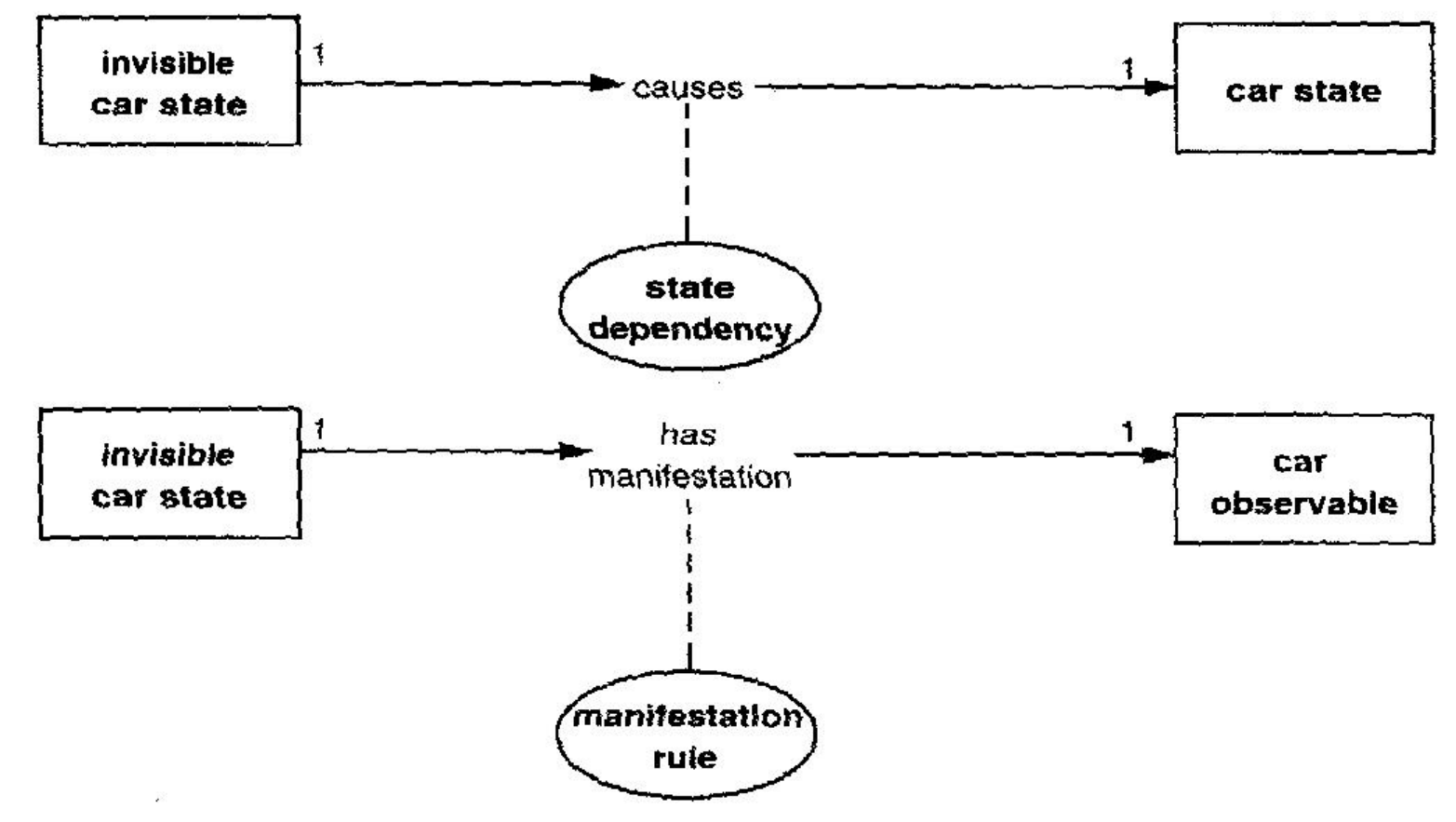


图3 规则类型的图形表示

注：有向线段从前件画到连接符号，再从连接符号到后件。规则类型的名称放在一个椭圆框中并用虚线与连接符号连接，数字表示基数〔前件或后件中表达式的最大值和最小值)。

规则-类型结构能够将知识库建立为只有相似结构的更小的划分(如规则集)。

fule-supply. status=blocked

CAUSES

gas-in-engine. status=false;

battery. status = low

CAUSES

power. value = off;

图2中的子类型层次为通常规则中的前件和后件提供了类型。图3给出了一个规则类型的图形表示，用汽车领域中的两个规则类型作为例子。有向线段从前件画到连接符号，再从连接符号到后件。规则类型的名称放在一个椭圆框中并用虚线与连接符号连接。由于规则类型和“作为类的关系”具有相似性，所以使用虚线：两个例子都是复杂实体。与线相连的数字表示基数。基数用于给出前件或后件中表达式的最大值和最小值的限制。在这个例子中规则必须只有一个条件和结论。

知识库

具有state—dependency(状态—依赖性)和manifestation(表象—规则)规则类型的实例的知识库规范说明包括两部分。

1)USE槽定义哪种类型的领域知识实例存储在知识库中。格式如下:

＜type＞FROM＜domain schema＞

其中，后半部分定义在哪个领域模式中定义类型。在汽车例子中只有一个模式，在更加复杂的应用中经常需要引入多重领域模式。

2)EXPRESSION槽包含实际实例。可以用半正式方式描述规则实例，其中的连接符号用于从后件中分离前件表达式。在这里也可以使用正式语言。但应注意，分析过程中知识库的形式和范围极易变化。因此在知识类型尚未被确认和生效时，避免规则实例过于形式化十分重要。

图4给出了一个知识库的例子，其中包含汽车应用的因果模型。它使用图2中定义的两个规则类型。图4中的实例对应图1中列出的知识片段。

```
KNOWLEDGE-BASE car-network;
    USES:
      state-dependency FROM car-diagnosis-schema,
      manifestation-rule FROM car-diagnosis-schema;
    EXPRESSIONS:
      / *state dependencies* /

      fuse.  status=blown GAUSES power.  status = off;
      battery.  status = low CAUSES power.  status=off;
      power.  status = off CAUSES
          engine-behavior.  status=does-not-start;
      fule-tank.  status=empty CAUSES gas-in—engine.  status = false;
      gas-1n-engine.  status = false CAUSES
          engine-behavior.  status = does-not-start;
      gas-in-engine.  status-false CAUSES
          engine-behavior.  status=slops;

      / *manifestation rules* /

      fuse.  status = blown HAS-MANIFESTATION
          fuse-inspection.  value = broken;
      battery.  status = low HAS-MANIFESTATION battery-dial.  value=zero;
      fuel-tank.  status=empty HAS-MANIFESTATION gas-dial.  value=zero;
  END KNOWLEDGE-BASE car-network;
```

图4. 包含两种规则类型实例的“汽车网络”知识库

正向推理机的实现

——LISP程序

产生式规则与规则库的存储结构

- 一般产生式规则的前件或后件可能是有限个事实或结论的合取式的析取。

- 例如：

  规则R为：

$$(F_1 \wedge F_2 \wedge F_3) \vee (F_4 \wedge F_5) \rightarrow H_1 \vee H_2$$

- 可等价变换为下述4条规则：

R_{11}： $F_1 \wedge F_2 \wedge F_3 \rightarrow H_1$

R_{12}： $F_4 \wedge F_5 \rightarrow H_1$

R_{21}： $F_1 \wedge F_2 \wedge F_3 \rightarrow H_2$

R_{22}： $F_4 \wedge F_5 \rightarrow H_2$

- 在规则库中，允许有前件不同但后件相同的规则。但是产生式规则的**一致性要求**要求规则库中的规则之间满足：凡是后件相同的规则，它们的前件没有包含关系，即一条规则的前件不是另一条规则的前件的子集。
- 可以用一个与/或图表示产生式的规则的事实和结论之间的与或关系。
- **例如**，规则R的与/或图如图所示。

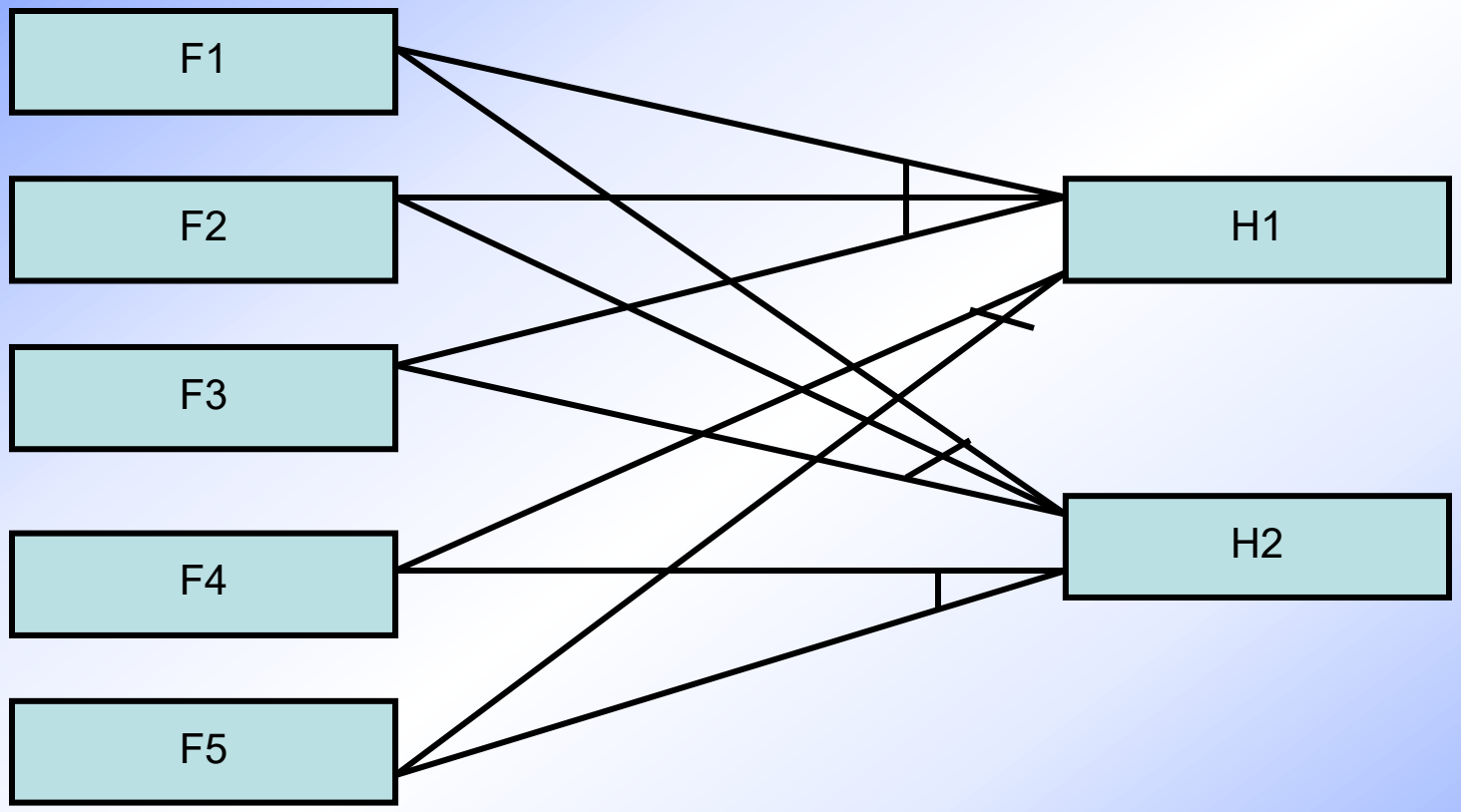


图.产生式规则与/或图

- 在LISP中，一条产生式规则的存储结构是一个表，通常一条规则存储形式是：

（规则名

（if（条件1）（条件2）…（条件n））

（then（结论1）（结论2）…（结论m））））

一条规则的表有3个顶层元素，第一个元素是规则名，第二个元素是包括if在内的规则前件，第三个元素是包括then在内的规则后件。第二个元素和第三个元素是表，元素个数分别为n+1和m+1。

用setq函数把15条规则组成一个表直接赋给rules:

```
(setq rules
      ((rule1
        (if (animal has hair))          若动物有毛发（F1）
        (then (animal is mammal)))      则动物是哺乳动物（M1）
       (rule2
        (if (animal gives milk))        若动物有奶（ F2）
        (then (animal is mammal)))      则动物是哺乳动物（M1）
```

(rule3	
(if (animal has feathers))	若动物有羽毛（F9）
(then (animal is bird)))	则动物是鸟（M4）
(rule4	
(if (animal gives flies)	若动物会飞（F10）
(animal lays eggs))	且生蛋（F11）
(then (animal is bird)))	则动物是鸟（M4）
(rule5	
(if (animal eats meat))	若动物吃肉（F3）
(then (animal is carnivore)))	则动物是食肉动物(M2)

(rule6

(if (animal has pointed teeth)　若动物有犀利牙齿(F4)

(animal has claws)　且有爪(F5)

(animal has forword eyes))　且眼向前方(F6)

(then (animal is carnivore)))　则动物是食肉动物(M2)

(rule7

(if (animal is mammal)　若动物是哺乳动物（M1）

(animal has hoofs))　且有蹄（F7）

(then (animal is ungulate)))　则动物是蹄类动物（M3）

(rule8	
(if (animal is mammal)	若动物是哺乳动物（M1）
(animal chews cud))	且反刍（F8）
(then (animal is ungulate)))	则动物是蹄类动物（M3）
(rule9	
(if (animal is mammal)	若动物是哺乳动物（M1）
(animal is carnivore)	且是食肉动物(M2)
(animal has tawny color)	且有黄褐色（F12）
(animal has dark sports))	且有暗斑点（F13）
(then (animal is cheetah)))	则动物是豹（H1）

(rule10

(if (animal is mammal) 若动物是哺乳动物（M1）

(animal is carnivore) 且是食肉动物(M2)

(animal has tawny color) 且有黄褐色（F12）

(animal has black stripes)) 且有黑色条纹（F15）

(then (animal is tiger))) 则动物是老虎（H2）

(rule11

(if (animal is ungulate) 若动物是蹄类动物（M3）

(animal has long neck) 且有长脖子（ F16）

(animal has long legs) 且有长腿（ F14 ）

(animal has dark spots)) 且有黑斑点（F13）

(then (animal is giraffe))) 则动物是长颈鹿（H3）

(rule12

(if (animal is ungulate) 若动物是蹄类动物（M3）

(animal has black stripes)) 且有黑色条纹（F15）

(then (animal is zebra))) 则动物是斑马（H4）

(rule13

(if (animal is bird) 若动物是鸟（M4）

(animal dose not fly) 且不能飞（F17）

(animal has long neck) 且有长脖子（ F16）

(animal has long legs) 且有长腿（ F14 ）

(animal is black and white)) 且有黑白二色(F18)

(then (animal is ostrich))) 则动物是鸵鸟（H5）

(rule14

(if (animal is bird) 若动物是鸟（M4）

(animal dose not fly) 且不能飞（F17）

(animal swims) 且会游泳（F19）

(animal is black and white)) 且有黑白二色(F18)

(then (animal is penguin))) 则动物是企鹅 （H6）

(rule15

(if (animal is bird) 若动物是鸟（M4）

(animal flies well)) 且善飞（F20）

(then (animal is albatross))) 则动物是信天翁(H7)

- procedure respond

将知识库中规则的前件同当前工作寄存器内容匹配，若匹配成功，则找到一条可用规则送入可用规则集S；否则，用下一条规则进行匹配。

while S非空且问题未求解 do

begin

调用select-rule(S)，从S中选择一条规则，

将该规则的结论添加到数据库中。

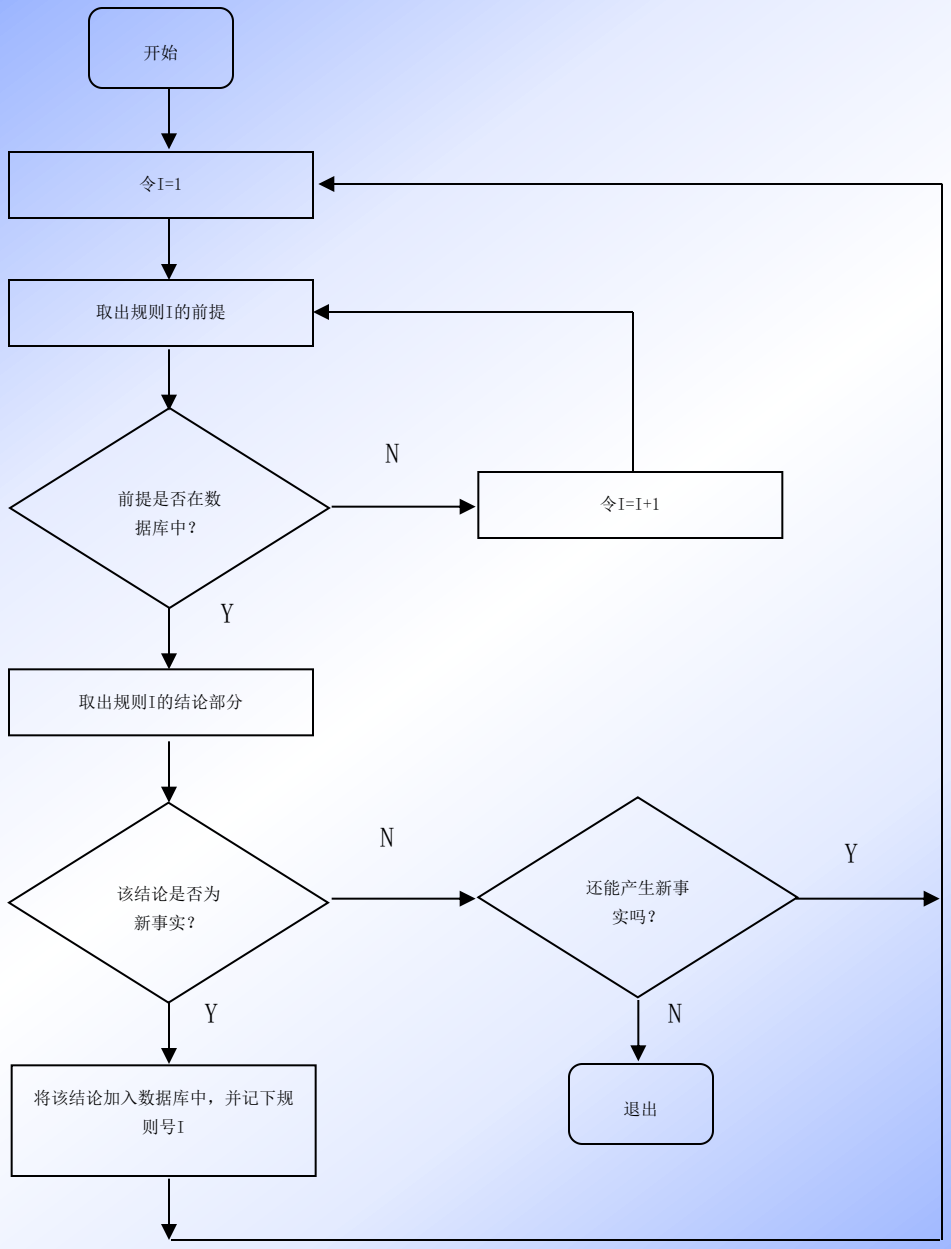
调用respond。

end

- 可见：

  1）正向过程respond是递归的。

  2）若可用规则集S中有两条或两条以上的规则，则由select- rule(S)从S中选择一条规则， select-rule(S)按设计的冲突消解策略来选择规则。


开始
令I=1
取出规则I的前提
前提是否在数据库中？
N
令I=I+1
Y
取出规则I的结论部分
该结论是否为新事实？
N
还能产生新事实吗？
Y
Y
N
将该结论加入数据库中，并记下规则号I
退出

正向推理示意图

（1）函数recall

- 函数表达式：

  （recall fact）

- 函数功能：

  判断变量fact中的一个事实是否在表facts中，若是，recall返回值是fact中的事实；否则，返回nil。

- 函数定义：

```
(defun (recall fact)
        (cond ((member fact facts)  fact)
              (t nil)))
```

(2) 函数test-if

- 函数表达式：

  （test-if rule）

- 函数功能：

  判断变量rule中的一条规则的前件包含的全部事实是否在表facts中，若是，test-if返回t；否则，返回nil。

- 函数定义：

```
(defun (test-if rule)
    (prog (ifs)
      (setq ifs (cdadr rule))
       loop
      (cond ((null ifs) (return t))
            (recall (car ifs))
            (t (return nil)))
     (setq ifs (cdr ifs))
     (go loop)))
```

(3)函数remember

- 函数表达式：

(remember new)

- 函数功能：

判断变量new中的一个事实是否在表facts中，若是，remember返回nil，否则，将new中的事实添加到表facts的表头，且remember返回new中的事实。

- 函数定义：

```
(defun (remember new)
    (cond ((member new facts) nil)
        (t (setq facts (cons new facts)) new)))
```

(4)函数use-then

- 函数表达式：

  (use-then rule)

- 函数功能：

  判断变量rule中的一条规则的后件包含的全部结论是否在表facts中，若全部的结论都在facts中，则use-then返回nil；否则，将不在facts中的结论逐一添加到表facts中，则use-then返回t 。

- 函数定义：

```
(defun (use-then rule)
    (prog (thens success)
        (setq thens (cdddr rule))
        loop
        (cond ((null thens) (return success))
                ((remember (car thens))
                (print (car rule))
                (print |DEDUCES| (car thens))
                (setq success t)))
        (setq thens (cdr thens))
        (go loop)))
```

（5）函数 try-rule

- 函数表达式：

  （try-rule rule）

- 函数功能：

  判断规则变量rule中的一条规则的前件包含的全部事实是否在表facts中，若全部事实都在facts中，且规则后件有不在facts中的结论，则把不在facts中的结论逐一添加到表facts中，try-rule返回t；否则，try-rule返回nil。

- 函数定义：

```
(defun  (try-rule rule)
        (and (test-if rule) (use-then rule)))
```

(6)函数step-forward

- 函数表达式：

  （step-forward rules）

- 函数功能：

  逐次扫描规则库rules中的规则，若发现rules中有一条可用规则，即该规则的前件包含的全部事实在表facts中，则把规则后件中不在facts中的所有结论添加到表facts中，且step-forward返回t；若rules中没有一条可用规则，则step-forward返回nil。

- 函数定义：

```
(defun (step-forward rules)
    (prog (rule-list)
      (setq rule-list rules)
      loop
      (cond ((null rule-list) (return nil))
            ((try-rule (car rule-list)) (return t)))
      (setq rule-list (cdr rule-list))
      (go loop)))
```

(7)正向推理函数deduce

- 函数表达式：

  (deduce facts)

- 函数功能：

  连续不断的从规则库rules中选择可用规则，每选择到一条可用规则，就把该规则的后件中不在facts中的所有结论添加到facts中，对facts扩充，用更新后的facts来选择下一条可用规则对facts再次扩充，直到没有可用规则为止。若曾找到一条规则对facts进行一次扩充，则deduce返回t；否则，deduce返回nil。

- 函数定义：

```
(defun (deduce facts)
 (prog (progress)
  loop
  (cond ((step-forward rules) (setq progrss t))
        (t (return progress)))
  (go loop)))
```

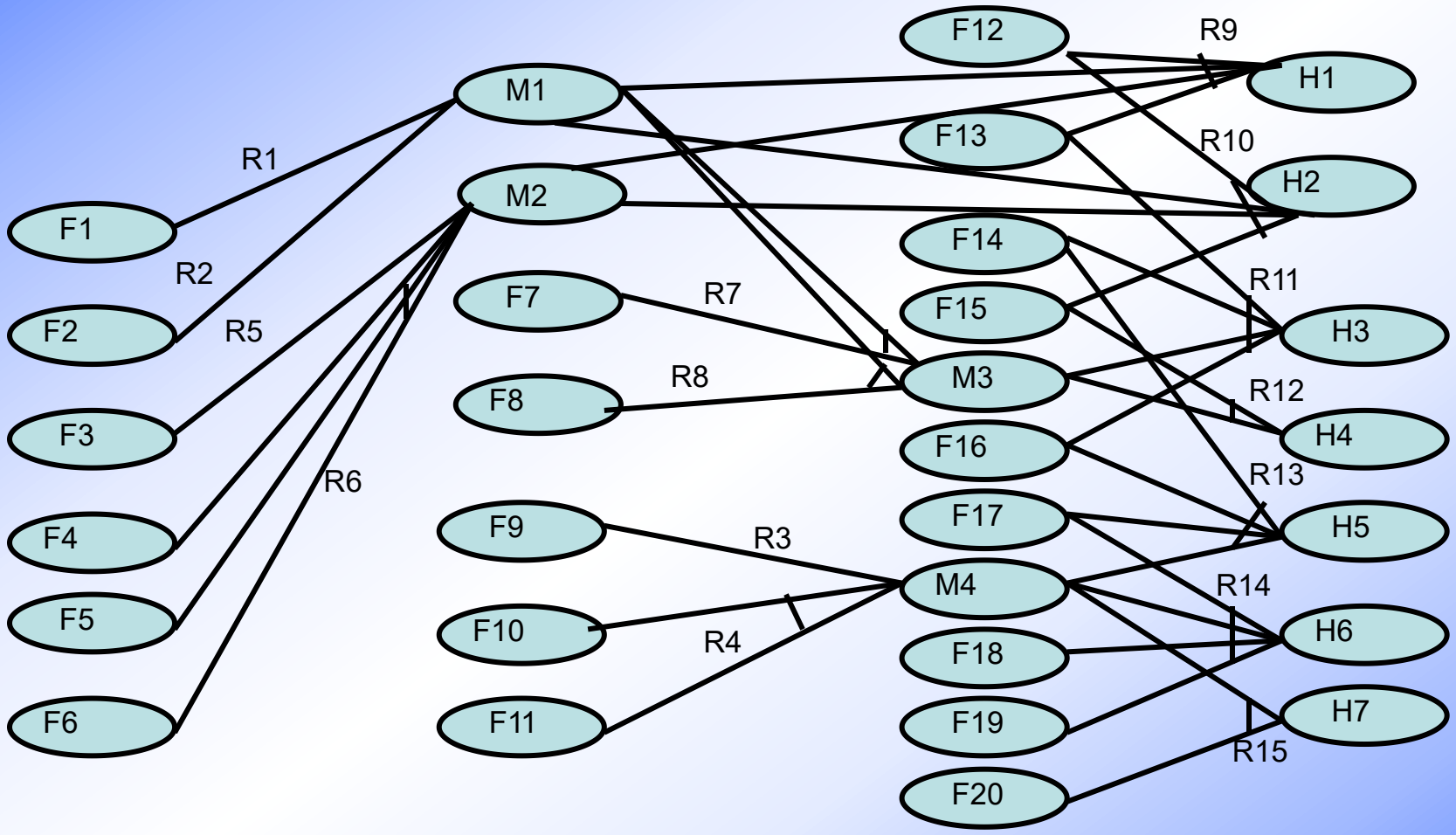
仿真实验

对于动物识别专家系统，若已知初始事实是F_{13}，F_{12}，F_3和 F_1，应用反向推理机（Goal-Driven Reasoning），说明推理过程和得出推理结论。

解： 使用setq函数把已知的初始事实赋值给事实表facts：

```
(setq facts
   ((animal has dark spots)
    (animal has tawny color)
    (animal eats meat)
    (animal has hair)))
```

即有：facts=（F_{13} F_{12} F_3 F_1）



动物识别专家系统规则库与/或图

对于动物识别专家系统，若用户要求确定的假设是H1，应用反向推理机(Goal-Driven Reasoning)，确定假设H1是否成立，说明推理过程。

1）调用反向推理机，WM=（H_1）。

2）调用反向推理机，WM =（$F_{12}H_1$）。

3）调用反向推理机，WM =（$F_{13}F_{12}H_1$）。

4）调用反向推理机，WM =（$M_2\ M_1\ F_{13}F_{12}H_1$）。

最后，WM =（$F_6\ F_5F_4F_3F_2\ F_1\ F_{13}F_{12}H_1$）

- 实际上，这个推理过程是规则库与/或图的一个子图的反向进行推理的。

反向推理机

- 反向推理是对给定的一个假设fact，判断其是否为真。确定一个假设fact是否为真的函数为verify。

Verify定义

1）若fact已在facts中，则fact为真，函数verify终止。

2）若fact不在facts中，则从rules中选出所有的后件含有fact的可用规则组成可用规则表relevant1。若relevant1为空，则直接向用户询问fact的真假，若用户确认fact为真，则把fact放到facts中，函数verify终止。

3）若relevant1不为空，则调用函数try-rule逐条作用于relevant1中的可用规则，试看是否直接最终推出fact，若推出fact，则fact为真，把fact添加到facts中，函数verify终止。

4）若不能直接推出fact，则调用函数try-rule+逐条作用于relevant1中的可用规则，试看能否间接最终推出fact，若推出fact，则fact为真，把fact添加到facts中，函数verify终止。

函数try-rule+的间接推出与函数try-rule的直接推出两者不同之处

- try-rule调用函数test-if来判断一条规则的前件包含的全部事实是否在facts中，test-if函数调用函数recall来判断前件中一个事实是否为真。
- try-rule+调用函数test-if+来判断一条规则的前件包含的全部事实是否在facts中，而test-if+调用函数verify来判断前件中的一个事实是否为真。

反向推理用到的几个函数

（1）函数thenp

- 函数表达式：

  (thenp fact rule)

- 函数功能：

  判断fact是否存在于规则变量rule中的一条规则的后件中，若是，则函数thenp返回规则后件从fact开始的余下表；否则，thenp返回nil。

- 函数定义：

```
(defun (thenp fact rule)
    (member fact  (cdddr rule)))
```

（2）函数inthen

- 函数表达式：

  (inthen fact)

- 函数功能：

  从规则库表rules中选出所有可用规则组成一个可用规则表，表中每一个元素是一条可用规则。若规则后件含有fact，则这条规则是一条可用规则，且函数inthen返回可用规则表。若没有一条可用规则，则返回空表。

- 函数定义：

```
(defun (inthen fact)
    (apply 'append
     (mapcar '(lambda (rule)
                (cond ((thenp fact rule) (list rule))
                      (t nil))
              (rules)))))
```

（3）函数test-if+

- 函数表达式：

  (test-if+ rule)

- 函数功能：

  直接或间接地反向推理确定规则变量rule中的一条规则的前件包含的所有的条件是否都为真，若是，则函数test-if+返回t；否则，test-if+返回nil。

（3）函数test-if+

- 函数定义：

```
(defun (test-if+ rule)
    (prog (ifs)
    (setq ifs (cdadr rule))
    loop
    (cond ((null ifs) (return t))
          (verity (car ifs))
          (t (return nil)))
    (setq ifs (cdr ifs))
    (go loop)))
```

(4) 函数try-rule+

- 函数表达式：

  (try-rule+ rule)

- 函数功能：

  若规则变量rule中的一条规则的前件包含的所有条件都能直接或间接地反向推理确认为真，且规则后件中有不在 facts中的结论，则把规则中不在facts中的结论添加到facts中，函数try-rule+返回t；否则， try-rule+返回nil。

- 函数定义：

```
(defun (try-rule+ rule)
    (and (test-if+ rule) (use-then rule)) )
```

（5）函数verify

- 函数表达式:

  (verify fact)

- 函数功能：

  直接或间接地反向推理确定变量fact中的一个假设是否为真，若为真，函数verify返回t；否则verify返回nil。

（5）函数verify

- 函数定义：

```
(defun (verify fact)
       (prog (relevant1 relevant2)
         (cond ((recall fact) (return t)))
         (setq relevant1 (inthen fact))
         (setq relevant2 relevant1)
         (cond ((null relevant1)
                  (cond ((member fact asked) (return nil))
                    ((and (print |IS THIS TRUE:|fact) (read))
                      (remember fact)/第二个分句的判断条件
                       (return t))/第二个分句
                    (t (setq asked (cons fact asked))/回答为假
                         (return nil)))
```

（5）函数verify

```
loop1
        (cond ((null relevant1) (go loop2))
              ((try-rule (car relevant1)) (return t)))
        (setq relevant1 (cdr relevant1))
        (go loop1)
loop2
        (cond ((null relevant2) (go exit))
              ((try-rule+ (car relevant2)) (return t)))
        (setq relevant2 (cdr relevant2))
        (go loop2)
      exit
        (return nil))))
```

（6）反向推理机函数diagnose

- 函数表达式：

（diagnose hypo）

- 函数功能：

逐一确定假设表hypo中多个假设的真或假，对确定为真的假设给出屏幕说明。

（6）反向推理机函数diagnose

- 函数定义：

```
(defun (diagnose hypo)
 (prog (poss)
  (steq poss hypo)
   loop
   (cond ((null poss)
            (print |NO HYPOTHESES CAN BE   CONFIRMED|)
            (return))
          ((verify (car poss))
           (print |HYPOTHESES| (car poss))
           (print |IS TRUE|)))
          (setq poss (cdr poss))
          (go loop)))
```

例:对于动物识别专家系统，若用户要求确定的假设是H1，应用反向推理机diagnose，确定假设H1是否成立，说明推理过程。

解：使用setq函数把假设H_1赋给假设表hypo：

```
(setq hypo
    ((animal is cheetah)))
```

即有hypo=（H_1）。

调用反向推理机（diagnose hypo），使用规则库rules是例5.1给出的规则库。

1）由（diagnose hypo）先复制poss=（H1），然后调用（verify H1），此时，facts=（）。

- 首先调用（recall H_1），由于fact= H_1，不在facts中，调用（inthen H_1），从rules中选出后件含有H_1的可用规则只有R_9，故relevant1=relevant2=（R_9）。
- 由于relevant1=（R_9）≠（），调用（try-rule R_9）。因为R_9的前件的全部条件不在facts中，故对H_1进行间接反向推理。

- 由于relevant2＝（R_9）≠（），调用（try-rule+ R_9）。它首先调用（test-if+ R_9），把R_9的前件包含的全部条件赋给ifs，即有ifs＝（F_{12} F_{13} M_1 M_2）。在（try-rule+ R_9）中，将顺序调用（verify F_{12}）、（verify F_{13}）、（verify M_1）和（verify M_2）。

2）调用（verify F_{12}），此时facts=（）。

3）调用（verify F_{13}），此时facts=（F_{12}）。

4）调用（verify M_1），此时facts=（F_{13} F_{12}）。

5）调用（verify M_2），此时facts=（F_2 F_1 F_{13} F_{12}）。

最后，facts=（F_6 F_5 F_4 F_3 F_2 F_1 F_{13} F_{12}）

- 调用（verify H_1），当（verify H_1）执行完毕，屏幕输出：“HYPOTHESE H_1 IS TRUE”。由(setq poss (cdr poss))使poss为空表，反向推理机(diagnose hypo)终止。

- 实际上，这个推理过程是按图5.4所示的规则库与/或图的一个子图的反向进行推理的。

- 实际上，推理过程的说明就是对函数的执行情况的说明。

解释机制与解释器

- **专家系统的解释**

  是对系统设计者或用户提出的问题给出解释或说明。

- **专家系统的解释器**

  是专家系统中为完成解释而设置的程序模块。

解释的方法

1.预置文本与路径跟踪法

- 最简单的解释方法是预置文本。
- 即把问题的解释预先用自然语言或其他易于理解的形式写好，插入程序段或相应的数据库中。在推理过程中或推理之后，一旦用户询问到已有预置解释文本的问题，只要把相应的解释文本添入解释框架，组织成对这个问题的解释提交给用户。

- **预置文本方法的缺点：**

  对每一个可能的问题都要编制解释预置文本，甚至对一个问题要编制几个解释预置文本，大大增加了系统开发的工作量。

- **路径跟踪法**

  是对推理过程进行跟踪，将问题求解所使用的知识自动记录下来。当用户提出需要解释时，解释器向用户显示问题求解路径。

- 路径跟踪法向用户提供why和How解释。

（1）正向推理机中函数try-rule的重新定义

- 函数定义：

```
(defun  (try-rule rule)
        (cond ((and (test-if rule) (use-then rule))
               (setq rules-used (cons rule rules-used))
               t)))
```

（2）反向推理机中函数try-rule+的重新定义

- 函数表达式：

  (try-rule+ rule)

- 函数功能：

  若规则变量rule中的一条规则的前件包含的所有条件都能直接或间接地反向推理确认为真，且规则后件中有不在 facts中的结论，则把规则中不在facts中的结论添加到facts中，并把rule中的这条规则添加到表rules-used的表头，函数try-rule+返回t；否则， try-rule+返回nil。

（2）反向推理机中函数try-rule+的重新定义

- 函数定义：

```
(defun (try-rule+ rule)
    (cond ((and (test-if+ rule) (use-then rule))
           (setq rules-used (cons rule rules-used))
            t)))
```

（3）函数usedp

- 函数表达式：

(usedp rulen)

- 函数功能：

若推理过程中使用过由规则名变量rulen记载的一个规则名指定的规则，则usedp返回t；否则，usedp返回nil。

（3）函数usedp

- 函数定义：

```
(defun (usedp rulen)
    (prog (poss)
        (setq poss rules-used)
         loop
         (cond ((null poss) (return nil))
                    ((equal rulen (car poss)) (return t)))
         (setq poss (cdr poss))
         (go loop)))
```

2. How解释

- How解释用于回答用户关于“系统是怎样得出这一结论”的询问。
- 函数表达式：

  (how fact)

- 函数功能：

  告诉用户变量fact中结论是怎么得出的，若fact中的结论或假设是经推理确定为真，则找到支持这一结论或假设成立的规则，显示该规则前件的所有事实。若fact中的结论或假设是已在facts中给出的事实，则给出相应的说明。否则，fact中的结论或假设没有被确认。

```
(defun (how fact)
        (prog (poss success)
           (setq poss rules-used)
           loop
          (cond ((null poss)
            (cond (success (return t))
                  ((recall fact)  (print fact |WAS GIVEN|) (return t))
                  (t (print fact |IS NOT ESTABLISHED|) (return nil))))

           ((thenp fact (car poss))
           (setq success t)
           (print fact |IS DEMONSTRATED BY|)
           (mapcar '(lambda (a) (print a)
                                (cdadr (car poss))))
          (setq poss (cdr poss))
           (go   loop)))
```

2. How解释

- 若使用过规则表rules-used不为空，则调用函数thenp对规则表rule-used逐个规则判断fact是否是规则后件中的一个结论，若有这样的规则，则逐一把这些规则的前件包含的所有事实作为“是怎样得出fact结论”的解释。

- 屏幕上给出说明：

  fact中的结论“IS DEMONSTRATED BY”规则前件的所有事实

- 若使用过规则表rules-used为空或rules-used中没有一条规则的后件中有fact的结论，则调用函数recall判断fact是否是facts中的一个给出的事实，若是，则

  屏幕上给出说明：

  fact中的事实“WAS GIVEN”

- 若不是上述两种情况，则

  屏幕上给出说明：

  fact中的假设“IS NOT ESTABLISHED”

3. Why解释

- Why解释:

  用于回答用户关于“为什么需要这一个事实”的询问。

- 函数表达式:

  (Why fact)

3. Why解释

- 函数功能：

  告诉用户为什么需要变量fact中的事实，若fact中的事实是在推理过程中作为一条规则的前件事实使用过，则找到这条规则，向用户说明fact中的事实用于支持这条规则的后件结论。若fact中的事实是以在facts中的给定事实，则给出相应的说明。否则，fact中的事实没有被确认。

```
(defun (why fact)
        (prog (poss success)
           (setq poss rules-used)
           loop
          (cond ((null poss)
              (cond (success (return t))
                    ((recall fact) (print fact |WAS HYPOTHESIS|) (return t))
                    (t    (print fact |IS NOT ESTABLISHED|) (return nil))))

        ((ifp fact (car poss))
           (setq success t)
           (print fact |NEEDED TO SHOW|)
           (mapcar '(lambda (a) (print a)
                          (cdaddr (car poss))))
         (setq poss (cdr poss))
         (go loop)))))
```

3. Why解释

- 其中函数ifp用于判定fact是否在规则的if部分中。
- Ifp定义：

```
(defun (ifp fact rule) (member fact (cdadr rule)))
```

- 若fact中的事实是在推理过程中作为使用过规则表rules-used中某些规则的前件事实使用过，则逐一把这些规则的后件结论作为“为什么需要fact事实”的解释，在屏幕上说明：

fact中的事实“NEEDED TO SHOW”规则后件的结论

3. Why解释

- 若fact中的事实没有被规则使用过，则判断fact是否是facts中已被确认的事实，若是，则屏幕上给出说明：

  fact中事实“WAS HYPOTHESIS”

- 若不是上述两种情况，则屏幕上直接给出说明：

  fact中的事实“IS NOT ESTABLISHED”

作业：1、2、3