专家系统讲义补充材料

艾海舟 编
E-mail: ahz@mail.tsinghua.edu.cn

清华大学计算机系
1999年10月
第 1 节 专家系统课程的介绍及要求

1. 课程的性质及其与其它课程的联系与区别

专家系统是人工智能领域的一个专题，许多内容是描述性的，是一个面向应用的技术性很强的学科。它的核心是知识及依赖知识的推理过程，它一般体现为咨询程序而为用户所使用。

学习专家系统的最好方法是在学习的过程中亲自编一个专家系统程序（专家系统实例系统）。实际上，对于计算机制技术性课程如程序设计、编译原理、网络、多媒体等等，都需要通过上机实践才能真正学会。

正如许多其它学科一样，人工智能中的各种专题都是互相关联的，有许多共性的方面。因此，资料的阅读面应当比较广，要确保一定量的精读，使知识比较深入而具有系统性。

其它相关领域：

2. 授课大纲

参见相应材料

3. 讲义、教材及参考资料

- 讲义纲要
- 提供一些原版的复印资料
- 提供电子版资料（包括讲义等其它资料）
  教材可从图书馆借阅，数量和种类都很多，挑选篇幅比较多的为佳。

4. 考核方式

参见相应材料

建议：从一开始，就考虑课程设计的选题，并相应确定课程论文的题目，边学边做。这样能够有切实的收获，学习的效率也高。
第2节 人工智能原理简述

1. 问题求解的基本原理（参见[4]pp.18-）

（1）生产式认知模型 IF <前提> THEN <结论> 通用问题求解器（GPS）

根源与数理逻辑中的一个重要结论：任何数学的或逻辑的系统都可以写成某种类型的生产式系统，或者说，生产式系统具有和图灵机相类似的计算能力。

解释：图灵机——计算的数学模型。什么是可计算性？等价的概念：λ演算，递归函数论，Post重写（改写）系统（任何数学的或逻辑的系统纯粹是一个规则集，其中的规则指出一串符号如何改变成另一种符号），它是“产生式系统”的渊源。它也是函数式语言（如LISP）数学基础。

（2）搜索与思维过程 思维或信息加工过程的具体化就是搜索

（3）状态空间（解空间）、搜索树

例子：8数码问题

\[
\begin{array}{ccc}
3 & 8 & 3 \\
1 & 6 & 4 \\
7 & 5 & \\
\end{array}
\quad \rightarrow \quad
\begin{array}{ccc}
2 & 8 & 3 \\
1 & 4 & \\
7 & 5 & \\
\end{array}
\]

（4）启发式搜索

A算法、A*算法

代价函数 \( f(n) = g(n) + h(n) \)  
设最佳路径对应函数 \( f^*(n) = g^*(n) + h^*(n) \)

\( g(n) \)：起点到当前点的实际耗费；\( h(n) \)：当前点距目标点的预测耗费（heuristics）。

当 \( h \) 是 \( h^* \)的下界（即 \( h(n) \leq h^*(n) \)）时，A算法就成为A*算法，既一定能找到一条最佳路径。

特别地当 \( h(n)=0 \) 时，对应于宽度优先算法。（参见[4]pp.54-）

博弈树的搜索

极大极小搜索

\( \alpha-\beta \)剪枝（提高搜索效率）

定义局的势态函数 \( f(p) \)：正值有利于A方，负值有利于B方，0值势均力敌；值越大对A方越有利，值越小对B方越有利。因此下棋时，A方选择目前下一步值最大的一步走棋，B方选择目前下一步值最小的一步走棋。

2. 符号逻辑基础

命题逻辑 一阶谓词逻辑 逻辑推理
参见讲义P7。

解释：数理逻辑 公理化系统 演绎系统 形式系统 完全性（完备性） 一致性（无矛盾性）

归结原理（参见[4]pp.73） Prolog 语言的逻辑基础。

两个子句，\( C_1 = \text{L} \lor C_1' \); \( C_2 = (-\text{L}) \lor C_2' \); 其归结式 \( C = C_1' \lor C_2' \) 是它们的逻辑推导，即 \( C_1' \land C_2 \Rightarrow C \)。当 \( C_1' = C_2' = \) 时，\( C = \)。

知识的逻辑表示参见讲义P7。
知识表示--状态空间

例题：农夫、狐狸、白菜、山羊过河，船只能一次载两件过河。如果山羊和狐狸单独在一起，狐狸就会吃掉山羊；而如果山羊和白菜单独在一起，山羊就会吃掉白菜。如何才能安全地过河？

(shore  shore-1, shore-2)

(shore-farmer, shore-fox, shore-cabbage, shore-goat)

Totally $2^4=16$ arrangements, 10 of which are safe in the sense that nothing is eaten. There is a connecting link if and only if the two arrangements meet two conditions: first, the farmer changes sides; and second, at most one of the farmer's possessions changes sides. Although there are $10 \times 9 = 90$ ordered pairs, but only 20 of these pairs satisfy the conditions required for the links.
例题：野人与传教士问题，三个野人和三名传教士过河，船每次只能载两项，如果野人超过传教士则野人会吃掉传教士，如何安全过河？

（shore-1-missionaries, shore-1-cannibals, shore-2-missionaries, shore-2-cannibals, boat-location）

3 missionaries (传教士), 3 cannibals (食人者)

在河的每一岸，传教士和食人者的数目可能为（0, 1, 2, 3）四种情况，共有 4×4=16 种，而其中在不发生被吃掉的意义下只有 10 种是安全的：（x, y, 3-x, 3-y）约束；

（1）x=0 时，有 y=0,1,2,3 四种情况，即 （0,0,3,3）(0,1,3,2)(0,2,3,1)(0,3,3,0);
（2）x=3 时，有 y=0,1,2,3 四种情况，即 （3,0,0,3）(3,1,0,2)(3,2,0,1)(3,3,0,0);
（3）x=1,2 时，需满足 x≤y 且 3-x≥3-y 即 y≤x，所以 x=y。有两种情况，即 （1,1,2,2)(2,2,1,1)
再考虑船的位置的差别，共有 10×2=20 种状态，其中有四种死锁状态，即：

# (0,0)  # (3,0)  # (0,0)  # (3,0)
(3,3)  (0,3)  (3,3)  (0,3)
第3节 专家系统的基本原理

内容参见讲义，通过一个小专家系统的例子来说明编程特点。

补充说明：几个经常问到的问题，也是大家普遍关心的问题，特别是非计算机专业的学生尤为关心的问题：

1. 专家系统课程的先修课程，有什么样的基础才能学这门课程？

   - 程序设计语言及程序设计基础知识，是必要的条件；专家系统实际上是关于解决与数值计算问题所不同的符号处理类问题的计算机编程方法。
   
   此外，其它课程都不是必须的，如
   
   - 数据结构是程序设计的有关算法方面的知识，有些这方面的初步知识对于理解专家系统程序比较有帮助。诸如面向对象的程序设计概念，对于大多数人而言，只需从它的实用角度来理解，它是一种比较简单的程序设计方法。避免了许多代码的重复，它是针对现代诸如图形化界面设计等复杂的软件开发问题的需要而产生的技术，是程序设计技术发展到一定阶段的必然产物。为了开发应用系统，你不必具有深入的面向对象的概念，利用当前的各种开发工具，完全可以构造出相当复杂的程序，重要的是参照例子多实践。实践是你增长程序设计知识的最好途径。
   
   - 离散数学中的经典数理逻辑即命题逻辑和一阶谓词逻辑对于理解基于逻辑的知识表示和PROLOG语言是有帮助的。
   
   - 人工智能导论及人工智能原理是与专家系统课程关系比较紧密的课程，前者是关于人工智能的基本理论和技术的讲解，而后者是关于人工智能应用的专项技术之一专家系统的讲解。
   
   - 知识工程简单地说是关于如何建立专家系统的方法论研究，讲解专家系统所设计的普遍性问题的基本理论与技术，侧重于一般性问题。

2. 专家系统程序是否可以用传统的程序设计方法来实现？

   当然！只是用传统的程序设计方法来实现效率比较低，也不好维护和更新。

   专家系统方法的核心内容是将知识库和推理机分开，知识库是有关问题的知识，如一系列的产生式规则。存储的方法可以利用数据库来实现；从这一点来看，它与数据库的特点类似，只是内容不一样。当然，数据库存储的是大量的有相同结构的数据，它的处理和检索是基于这种内在的结构，如关系数据库。数据库技术是成熟的，大量应用于方方面面。而知识库技术，还处于发展的初期，由于所涉及的面太宽，知识很难用统一而有效的结构方式来表示，当前在专家系统程序中常用产生式来表示知识是权宜之计。显然，数据库技术会向知识库方向发展，这是数据库的智能化。而知识库技术的发展要利用数据库技术作支持。

   旁注：在计算机技术范围内，效率的概念是非常重要的，如编译的效率，计算的复杂度（计算的效率，包括空间复杂度和时间复杂度）；各种不同的工具适合于做不同的事情，这些都是从工程的角度得出的结论。而从计算机科学的角度，或从纯理论的角度，任何计算机语言都是等价的，即它们的计算能力等价的，通俗地说就是同一问题可以用任何一种语言来编程，也可以用任何另一种语言来实现。另外，计算机语言是一种形式语言，形式语言也有其局限性（图尔克定理：形式系统的不完全性定理），不是完备的，如停机问题；是否存在一个程序能
能够判断任意一程序是否会停机？这样的程序是不存在的。

3. 专家系统的课程设计的标准是什么？

最低的要求是利用专家系统的编程思想，编制一个小程序，称之为原型，体会一下非数值计算类程序的设计特点。
由于课程实验条件的限制，各自只能利用自己的上机条件来做课程设计。评价主要是定性的，只要是切实努力做了，程序的性能好坏只作参考。
只要切实学习和体会了专家系统技术的特点，就达到了本门课程的基本要求。

4. 专家系统所涉及的学术研究资料

如期刊：IEEE Trans. Knowledge and data engineering,
Uncertainty, fuzziness and knowledge-based systems,
Artificial intelligence 等

当前的研究热点：数据挖掘(data mining), 数据库知识发现(knowledge discovery from database),
Internet 上信息的智能检索，即所谓的基于内容的检索等。

有关专家系统的研究书籍，在外文教材中心有各种专业背景的专家系统书籍可以查阅。如果你的论文是做有关专家系统方面的工作，建议去找一本与自己的问题最接近的资料仔细研读。
第 4 节 黑板结构和编程技术

1. 黑板结构和编程技术

1.1 黑板概念

人工智能技术中的“黑板”概念，由 Newell 于 1962 年提出，

“我们可以考虑一组注视着同一块黑板的人：每个人都能阅读黑板上的各种信息，并且都能判断何时他们各自可往黑板上增加一些他们所具有的、有价值的东西...”

考虑一类按“产生-测试”模型来组织的程序结构，主要缺点是死板、不灵活。“通过保持程序的分离，而且允许所有子程序使用一个公共的数据结构便可缓和这种缺点”

1966 年 Simon 指出；“...我们称这些关于在整体问题求解过程中一直被注意和被永久（或相当长期的）存储器里的任务环境信息为‘黑板’”

1.2 黑板模型

（计算机模型）由知识源和黑板组成，

- 知识源：问题求解所需的知识分成若干个独立的独立知识源。
- 黑板数据库：黑板是存放问题状态数据的全局数据库。

知识源改变黑板且导致获得问题的解。知识源之间的通讯和交互仅仅通过黑板才能进行。

知识工程

“知识”本身作为一个研究对象，它太宽泛，试图用统一的方法解决基于知识的问题还为时过早。

当前的研究主要是从工程的角度，讨论用计算机如何处理这类涉及知识的问题。实际上就是如何把要解决的问题表示清楚，以及把如何解决问题的方法表达出来，形成问题的计算模型（“形式描述”），以便能够在计算机上编程解决。

解释：对于不成熟的学科，与其称之为“科学”，不如称之为“艺术”，而工程是在现有的技术条件下如何解决问题的方法。

计算机本质上是遵循二值逻辑的，计算机的本质是命题逻辑演算。而用计算机实现的推理，本质上是通过计算来完成的。

智能三要素：推理能力、学习能力、联想能力
### 第5节 知识表示

**例子1**：计算机问答系统

1. In memory: some facts: 记忆中的一些事实
   - Profession: P(wang, teacher), P(zhang, student)
   - Sex: S(wang, male), S(zhang, female)
   - Age: A(zhao, 40), A(zhang, 20)
   - Work: W(z, v): z 工作了 v 年，如 W(wang, 20)
   - Education: E(x, y): x 的教育程度 y

2. Knowledge: 知识
   - (∀x) (P(x, teacher) → E(x, high))
   - (∀y) (E(y, high) → (∃x) (A(y, x) ∧ GR(x, 23)))
   - (∀z) (∀v) (P(z, teacher) ∧ W(z, v) ∧ (∃w) EQ(w, ADD(v, 23)) → (∃x) (A(z, x) ∧ G(x, w)))

3. Problem 问题
   - A(zhao, 40)
   - A(wang, t): no
     - 聪明？不，这只是数据检索！
   - A(wang, t): no
     - 如果能回答：＞23 或 ＞43 则它有智能（因为有推理能力）
   - 告诉什么，回答什么，只能说明记忆。
   - 从隐含的知识中获得回答，就是智能，却聪明。（有推理能力）。

   - A(wang, t): no
     - ＞23
   - - Why
     - Education level of Mr. Wang is high.
   - - How
     - rule (2)
   - - Why
     - Mr. Wang is a teacher
   - - How
     - rule (1)
   - - Why
     - You told me!

**PROLOG** PROgramming in LOGic 1970 年 developed by Alain Colmerauer.

特点：search: depth-first search inference: backward-chaining

问题描述清楚了，就可以了；但是缺点是效率低。

Facts: profession(Wang, teacher) sex(Wang, male) worktime(Wang, 20)

Rules: education-level(x, high): - profession(x, teacher)
       greater(x, 23): - education-level(y, high), age(y, x)
       age(z, y): - profession(z, teacher), worktime(z, v), y is v+23
retire(v):-Sex(v, male), age(v, x), x>=60

结论可条件：其中 is 的功能是使得其左边的变量得到其右边算术表达式的值。

Ask question:
?retire(Wang)

先查事实，有则结束；没有再查规则。

retire(wang):-sex(wang, male), age(wang, x), x>=60

Age(wang, y):-profession(wang, teacher), worktime(wang, v), y is v+23

Worktime(wang, 20)

Age(wang, 43)

No (结论)

LISP LISt Processing languange invented by MIT J. McCathy 1958

特点：数据与程序一致即 S 表达式 运行就是求值过程。

Programming = (define) function

Running = evaluation(function)

Control = recursive 递归

Facts: (Wang, profession,teacher) (Wang, sex, male) (Wang, worktime, 20)
(Zhao, age, 40)

Rules: define functions

(defun age(x) (cond ((equal (get x 'profession) 'teacher)

          (plus (get x 'worktime) 23) )

)

(defun retire(x) (cond ((and ((equal (get x 'sex) 'male)

       (> age(x) 60)) T)

     ((and ((equal (get x 'sex) 'female)

       (> (age x) 55)) T)

     (T F)

)

)

推荐参考书：（有关规则及基于规则的推理链）


P119-161 Rules and Rule Chaining

P47-62 Generate and Test, Means-Ends Analysis, and Problem Reduction

Good Representations Are the Key to Good Problem Solving

The representation principle:
Once a problem is described using an appropriate representation, the problem is almost solved. A Representation Has Four Fundamental Parts:

- A *lexical* part that determines which symbols are allowed in the representation’s *vocabulary*
- A *structural* part that describes *constraints* on how the symbols can be arranged
- A *procedural* part that specifies *access procedures* that enable you to create descriptions, to modify them, and to answer questions using them
- A *semantic* part that establishes a way of associating *meaning* with the descriptions.

There is the emphasis on *scaling up*. These days, it is hard to attract attention with an idea that appears suited to toy problems only.

**Three powerful problem-solving methods:** p.47-62

- **Generate-and-test method** 产生与测试

  To perform generate and test,
  - Until a satisfactory solution is found or no more candidate solutions can be generated,
    - Generate a candidate solution.
    - Test the candidate solution.
  - If an acceptable solution is found, announce it; otherwise, announce failure.

  The generate-and-test paradigm is used most frequently to solve identification problems. In identification problems, the genreator is said to produce *hypotheses*.

  Good generators are complete, nonredundant, and informed.

- **Means-ends analysis method** 方法（方法，策略） ends(目的、目标)

  To perform means-ends analysis,
  - Until the goal is reached or no more procedures are available,
    - Describe the current state, the goal state, and the difference between the two.
    - Use the difference between the current state and goal state, possibly with the description of the current state or goal state, to select a promising procedure.
    - Use the promising procedure and update the current state.
  - If the goal is reached, announce success; otherwise, announce failure.

  The purpose of means-ends analysis is to identify a procedure that causes a transition from the current state to the goal state, or at least to an intermediate state that is closer to the goal state in a state space. Thus, the identified procedure reduces the observed difference between the current state and the goal state.

  The state of a system is a description that is sufficient to determine the future. In a state space, each node denotes a state, and each link denotes a possible one-step transition from one state to another state.

  The key idea in means-ends analysis is to reduce differences. Means-ends analysis is often mediated via difference-procedure tables.

- **Problem-reduction method** 问题约简（降解）

  When using the problem-reduction method, you generally recognize goals and convert them into appropriate subgoal. When so used, problem reduction is often called , equivalently, goal reduction.

  To determine whether a goal has been achieved, you need a testing procedure. The key
procedure, REDUCE, channels action into the REDUCE-AND and the REDUCE-OR procedures:

To determine, using REDUCE, whether a goal is achieved,
▷ Determine whether the goal is satisfied without recourse to subgoals:
  ▷ If it is, announce that the goal is satisfied.
  ▷ Otherwise, determine whether the goal corresponds to an And goal:
    ▷ If it does, use the REDUCE-AND procedure to determine whether the goal is satisfied.
    ▷ Otherwise, use the REDUCE-OR procedure to determine whether the goal is satisfied.

To determine, using REDUCE-AND, whether a goal has been satisfied,
▷ Use REDUCE on each immediate subgoal until there are no more subgoals, or until REDUCE finds a subgoal that is not satisfied.
▷ If REDUCE has found a subgoal that is not satisfied, announce that the goal is not satisfied; otherwise, announce that the goal is satisfied.

To determine, using REDUCE-OR, whether a goal has been satisfied,
▷ Use REDUCE on each subgoal until REDUCE finds a subgoal that is satisfied.
▷ If REDUCE has found a subgoal that is satisfied, announce that the goal is satisfied; otherwise, announce that the goal is not satisfied.

The key idea in problem reduction is to explore a goal tree. A goal tree consists of And goals, all of which must be satisfied, and Or goals, one of which must be satisfied.

Problem reduction is ubiquitous in programming because subprocedure call is a form of problem reduction.

第 6 节  知识表示__语义网络


Semantic Nets 语义网络
Good Representations Are the Key to Good Problem Solving
P.16-17 例子: A children's puzzle: The Farmer, Fox, Goose, and Grain
Rule-Based Deduction Systems
P.121-129 例子: A toy deduction system identifies animals
P. 130-131 应用: Mycin diagnoses bacterial infections of the blood

p.20-21 Schools of thought about the meaning of semantics
• Equivalent semantics 等价语义
• Procedural semantics 过程语义
• Descriptive semantics 描述语义
第 7 节 专家系统工具 CLIPS

下载的软件说明:
(Ftp:166.111.4.80 Work.For.China/Documents/算法研究于程序代码/ExpertSystem,
http://www.ghg.net/clips/download )
1. CLIPS-6.1 (C language integrated production system)
clicks61.zip  957KB
2. Common LISP
clisp-win32.zip  1,255KB
3. LISP programs for Winston's AI book
aibook_winston  94KB
作业：通过运行其中的例子，体会专家系统的编程特点。

参考书：专家系统工具 CLIPS 及其应用 吴鹤龄，北京理工大学出版社，1991。

CLIPS 本身是前向链结构（规则部分）
软件：clips 6.1 for windows (C language integrated production system)

例：从一批数中找最大数
(deffacts max-num (loop-max 5))
(defrule loop-assert
  (loop-max ?n)
  =>
  (bind ?I 1)
  (while (<= ?I ?n) do
    (printout t "Please input " (+ 1 (- ?n ?I)) "th number: ")
    (bind ?m (read))
    (assert (number ?m))
    (bind ?I (+ ?I 1)))
(defrule largest-number
  (number ?number1
  (not (number ?number2&: (> ?number2 ?number1)))
  =>
  (printout t "largest number is " ?number1 crlf))

例：动物分类
annal Animal Identification Expert System
annal A simple expert system which attempts to identify
annal an animal based on its characteristics.
annal The knowledge base in this example is a
annal collection of facts which represent backward
annal chaining rules. CLIPS forward chaining rules are
annal then used to simulate a backward chaining inference
annal engine.
annal
annal CLIPS Version 6.0 Example
annal
annal To execute, merely load, reset, and run.
annal Answer questions yes or no.
annal

---

(deftemplate rule
  (multislot if)  ;  事实模板
(multislot then))

::* * INFERENC ENGINE RULES *

(defrule propagate-goal "
(goal is ?goal)
(rule (if ?variable $?)
 (then ?goal ?value))
 =>
 (assert (goal is ?variable)))

(defrule goal-satisfied "
(declare (salience 30))
?f <- (goal is ?goal)
(variable ?goal ?value)
(answer ? text ?goal)
 =>
 (retract ?f)
 (format t "%s\n" ?text ?value))

(defrule remove-rule-no-match "
(declare (salience 20))
(variable ?variable ?value)
?f <- (rule (if ?variable ?value))
 =>
 (retract ?f))

(defrule modify-rule-match "
(declare (salience 20))
(variable ?variable ?value)
?f <- (rule (if ?variable ?value and $?rest))
 =>
 (modify ?f (if ?rest)))

(defrule rule-satisfied "
(declare (salience 20))
(variable ?variable ?value)
?f <- (rule (if ?variable ?value)
 (then ?goal ?goal-value))
 =>
 (retract ?f)
 (assert (variable ?goal ?goal-value)))

(defrule ask-question-no-legalvalues "
(declare (salience 10))
(not (legalanswers $?)))
?f1 <- (goal is ?variable)
?f2 <- (question ?variable ?text)
 =>
 (retract ?f1 ?f2)
 (format t ": %s\n" ?text)
 (assert (variable ?variable (read))))

(defrule ask-question-legalvalues "
(declare (salience 10))
(legalanswers ? answers)
?f1 <- (goal is ?variable)
?f2 <- (question ?variable ?text)
 =>
 (retract ?f1)
 (format t ": %s\n" ?text)
 (printout t ?answers " ")
 (bind ?reply (read))
 (if (member (lowcase ?reply) ?answers)
 then (assert (variable ?variable ?reply))
 (retract ?f2)
 else (assert (goal is ?variable)))

(if <condition> then <goal-attribute> is <value>)
<condition>是<attribute>是<value>
<attribute>是<value>和<condition>

; 前提与事实不符，撤消。

; 去掉已满足的部分。

; 插入新的断言。

; 回答不符合要求时，重新输入（回答）。
(deffacts knowledge-base
  (goal is type:animal)
  (legalanswers are yes no)
  (rule (if backbone is yes)
    (then superphylum is backbone))
  (rule (if backbone is no)
    (then superphylum is jellyback))
  (question backbone is "Does your animal have a backbone?")
  (rule (if superphylum is backbone and
    warm.blooded is yes)
    (then phylum is warm))
  (rule (if superphylum is backbone and
    warm.blooded is no)
    (then phylum is cold))
  (question warm.blooded is "Is the animal warm blooded?")
  (rule (if superphylum is jellyback and
    live.prime.in.soil is yes)
    (then phylum is soil))
  (rule (if superphylum is jellyback and
    live.prime.in.soil is no)
    (then phylum is elsewhere))
  (question live.prime.in.soil is "Does your animal live primarily in soil?")
  (rule (if phylum is warm and
    has.breasts is yes)
    (then class is breasts))
  (rule (if phylum is warm and
    has.breasts is no)
    (then type:animal is bird/penguin))
  (question has.breasts is "Normally, does the female of your animal nurse its young with milk?")
  (rule (if phylum is cold and
    always.in.water is yes)
    (then class is water))
  (rule (if phylum is cold and
    always.in.water is no)
    (then class is dry))
  (question always.in.water is "Is your animal always in water?")
  (rule (if phylum is soil and
    flat.bodied is yes)
    (then type:animal is flatworm))
  (rule (if phylum is soil and
    flat.bodied is no)
    (then type:animal is worm/leech))
  (question flat.bodied is "Does your animal have a flat body?")
  (rule (if phylum is elsewhere and
    body.in.segments is yes)
    (then class is segments))
  (rule (if phylum is elsewhere and
    body.in.segments is no)
    (then class is unified))
  (question body.in.segments is "Is the animals body in segments?")
  (rule (if class is breasts and
    can.eat.meat is yes)
    (then order is meat))
  (rule (if class is breasts and
    can.eat.meat is no)
    (then order is vegy))
  (question can.eat.meat is "Does your animal eat red meat?")
  (rule (if class is water and
    boney is yes)
    (then type:animal is fish))
  (rule (if class is water and
    boney is no)
    (then type:animal is shark/ray))
  (question boney is "Does your animal have a boney skeleton?")
(rule (if class is dry and
    scally is yes)
    (then order is scales))
(rule (if class is dry and
    scally is no)
    (then order is soft))
(question scally is "Is your animal covered with scaled skin?"

(rule (if class is segments and
    shell is yes)
    (then order is shell))
(rule (if class is segments and
    shell is no)
    (then type.animal is centipede/millipede/insect))
(question shell is "Does your animal have a shell?"

(rule (if class is unified and
    digest.cells is yes)
    (then order is cells))
(rule (if class is unified and
    digest.cells is no)
    (then order is stomach))
(question digest.cells is "Does your animal use many cells to digest it's food instead of a stomach?"

(rule (if order is meat and
    fly is yes)
    (then type.animal is bat))
(rule (if order is meat and
    fly is no)
    (then family is nowings))
(question fly is "Can your animal fly?"

(rule (if order is vegy and
    hooves is yes)
    (then family is hooves))
(rule (if order is vegy and
    hooves is no)
    (then family is feet))
(question hooves is "Does your animal have hooves?"

(rule (if order is scales and
    rounded.shell is yes)
    (then type.animal is turtle))
(rule (if order is scales and
    rounded.shell is no)
    (then family is noshell))
(question rounded.shell is "Does the animal have a rounded shell?"

(rule (if order is soft and
    jump is yes)
    (then type.animal is frog))
(rule (if order is soft and
    jump is no)
    (then type.animal is salamander))
(question jump is "Does your animal jump?"

(rule (if order is shell and
    tail is yes)
    (then type.animal is lobster))
(rule (if order is shell and
    tail is no)
    (then type.animal is crab))
(question tail is "Does your animal have a tail?"

(rule (if order is cells and
    stationary is yes)
    (then family is stationary))
(rule (if order is cells and
    stationary is no)
    (then type.animal is jellyfish))
(question stationary is "Is your animal attached permanently to an object?"

(rule (if order is stomach and
    multicelled is yes)
    (then family is multicelled))
(rule (if order is stomach and
    multicelled is no)
(then type.animal is protozoa)
(question multicelled is "Is your animal made up of more than one cell?"
(rule (if family is newings and
    opposing.thumb is yes)
    (then genus is thumb))
(rule (if family is newings and
    opposing.thumb is no)
    (then genus is nothumb))
(question opposing.thumb is "Does your animal have an opposing thumb?")
(rule (if family is hooves and
two.toes is yes)
    (then genus is twotoes))
(rule (if family is hooves and
two.toes is no)
    (then genus is onetoe))
(question two.toes is "Does your animal stand on two toes/hooves per foot?")
(rule (if family is feet and
    live.in.water is yes)
    (then genus is water))
(rule (if family is feet and
    live.in.water is no)
    (then genus is dry))
(question live.in.water is "Does your animal live in water?")
(rule (if family is noshell and
    limbs is yes)
    (then type.animal is crocodile/alligator))
(rule (if family is noshell and
    limbs is no)
    (then type.animal is snake))
(question limbs is "Does your animal have limbs?")
(rule (if family is stationary and
    spikes is yes)
    (then type.animal is sea.anemone))
(rule (if family is stationary and
    spikes is no)
    (then type.animal is coral/sponge))
(question spikes is "Does your animal normally have spikes radiating from it’s body?")
(rule (if family is multicelled and
    spiral.shell is yes)
    (then type.animal is snail))
(rule (if family is multicelled and
    spiral.shell is no)
    (then genus is noshell))
(question spiral.shell is "Does your animal have a spiral-shaped shell?")
(rule (if genus is thumb and
    prehensile.tail is yes)
    (then type.animal is monkey))
(rule (if genus is thumb and
    prehensile.tail is no)
    (then species is notail))
(question prehensile.tail is "Does your animal have a prehensile tail?")
(rule (if genus is nothumb and
    over.400 is yes)
    (then species is 400))
(rule (if genus is nothumb and
    over.400 is no)
    (then species is under400))
(question over.400 is "Does an adult normally weigh over 400 pounds?")
(rule (if genus is twotoes and
    horns is yes)
    (then species is hornse))
(rule (if genus is twotoes and
    horns is no)
    (then species is nohorns))
(question horns is "Does your animal have horns?")
(rule (if genus is onetoe and
    plating is yes)
    (then type.animal is rhinoceros))
(rule (if genus is onetoe and
...
plating is no)
(then type.animal is horse/zebra)

(question plating is "Is your animal covered with a protective plating?")

(rule (if genus is water and
        hunted is yes)
        (then type.animal is whale))

(rule (if genus is water and
        hunted is no)
        (then type.animal is dolphin/poison))

(question hunted is "Is your animal, unfortunately, commercially hunted?")

(rule (if genus is dry and
        front.teeth is yes)
        (then species is teeth))

(rule (if genus is dry and
        front.teeth is no)
        (then species is teeth))

(question front.teeth is "Does your animal have large front teeth?")

(rule (if genus is noshell and
        bivalve is yes)
        (then type.animal is clam/oyster))

(rule (if genus is noshell and
        bivalve is no)
        (then type.animal is squid/octopus))

(question bivalve is "Is your animal protected by two half-shells?")

(rule (if species is notail and
        nearly.hairless is yes)
        (then type.animal is man))

(rule (if species is notail and
        nearly.hairless is no)
        (then subspecies is hair))

(question nearly.hairless is "Is your animal nearly hairless?")

(rule (if species is 400 and
        land.based is yes)
        (then type.animal is bear/tiger/lion))

(rule (if species is 400 and
        land.based is no)
        (then type.animal is walrus))

(question land.based is "Is your animal land based?")

(rule (if species is under400 and
        thintail is yes)
        (then type.animal is cat))

(rule (if species is under400 and
        thintail is no)
        (then type.animal is coyote/wolf/fox/dog))

(question thintail is "Does your animal have a thin tail?")

(rule (if species is horns and
        one.horn is yes)
        (then type.animal is hippopotamus))

(rule (if species is horns and
        one.horn is no)
        (then subspecies is nohorn))

(question one.horn is "Does your animal have one horn?")

(rule (if species is nohorns and
        lives.in.desert is yes)
        (then type.animal is camel))

(rule (if species is nohorns and
        lives.in.desert is no)
        (then type.animal is giraffe))

(question lives.in.desert is "Does your animal normally live in the desert?")

(rule (if species is teeth and
        large.ears is yes)
        (then type.animal is rabbit))

(rule (if species is teeth and
        large.ears is no the type.animal is rat/mouse/squirrel/beaver/poconcupine))

(question large.ears is "Does your animal have large ears?")

(rule (if species is noteeth and
        pouch is yes)
        (then type.animal is "kangaroo/koala bear"))

(rule (if species is noteeth and
(question pouch is "Does your animal have a pouch?"
(rule (if subspecies is hair and
        long.powerful.arms is yes)
        (then type.animal is orangutan/gorilla/chimpanzie))
(rule (if subspecies is hair and
        long.powerful.arms is no)
        (then type.animal is baboon))
(question long.powerful.arms is "Does your animal have long, powerful arms?"
(rule (if subspecies is nohorn and
        fleece is yes)
        (then type.animal is sheep/goat))
(rule (if subspecies is nohorn and
        fleece is no)
        (then subspecies is nofleece))
(question fleece is "Does your animal have fleece?"
(rule (if subspecies is nofleece and
        domesticated is yes)
        (then type.animal is cow))
(rule (if subspecies is nofleece and
        domesticated is no)
        (then type.animal is deer/moose/antelope))
(question domesticated is "Is your animal domesticated?"
(answer is "I think your animal is a " type.animal))
第 8 节 CLIPS6.1 程序示例

dilemma1.lsp  例题：农夫、狐狸、白菜、山羊过河，船只能一次载两项过河。如果山羊和狐狸单独在一起，狐狸就会吃掉山羊；而如果山羊和白菜单独在一起，山羊就会吃掉白菜。如何才能安全地过河？

;;;---------------------------------------------------------------------
;;; Farmer's Dilemma Problem
;;;
;;; Another classic AI problem (cannibals and the missionary) in agricultural terms.
The point is to get the farmer, the fox the cabbage and the goat across a stream. But the boat only holds 2 items. If left alone with the goat, the fox will eat it. If left alone with the cabbage, the goat will eat it.
;;; This example uses rules and fact pattern matching to solve the problem.
;;; CLIPS Version 6.0 Example
;;; To execute, merely load, reset and run.
;;;---------------------------------------------------------------------

(defmodule MAIN
  (export deftemplate status)

;;;**************
;;;* TEMPLATES *
;;;**************

;;; The status facts hold the state information of the search tree.

(deftemplate MAIN::status
  (slot search-depth (type INTEGER) (range 1 ?VARIABLE))
  (slot parent (type FACT-ADDRESS SYMBOL) (allowed-symbols no-parent))
  (slot farmer-location
     (type SYMBOL) (allowed-symbols shore-1 shore-2))
  (slot fox-location
     (type SYMBOL) (allowed-symbols shore-1 shore-2))
  (slot goat-location
     (type SYMBOL) (allowed-symbols shore-1 shore-2))
  (slot cabbage-location
     (type SYMBOL) (allowed-symbols shore-1 shore-2))
  (slot last-move
     (type SYMBOL) (allowed-symbols no-move alone fox goat cabbage)))

;;;**************
;;;* INITIAL STATE *
;;;**************

(deffacts MAIN::initial-positions
  (status (search-depth 1)
    (parent no-parent)
    ...
(farmer-location shore-1)
(fox-location shore-1)
(goat-location shore-1)
(cabbage-location shore-1)
(last-move no-move))

(deffacts MAIN::opposites
  (opposite-of shore-1 shore-2)
  (opposite-of shore-2 shore-1))

:::**********************
::* GENERATE PATH RULES *
:::**********************

(defrule MAIN::move-alone
  ?node <- (status (search-depth ?num)
    (farmer-location ?fs))
  (opposite-of ?fs ?ns)
  =>
  (duplicate ?node (search-depth =(+ 1 ?num))
    (parent ?node)
    (farmer-location ?ns)
    (last-move alone)))

(defrule MAIN::move-with-fox
  ?node <- (status (search-depth ?num)
    (farmer-location ?fs)
    (fox-location ?fs))
  (opposite-of ?fs ?ns)
  =>
  (duplicate ?node (search-depth =(+ 1 ?num))
    (parent ?node)
    (farmer-location ?ns)
    (fox-location ?ns)
    (last-move fox)))

(defrule MAIN::move-with-goat
  ?node <- (status (search-depth ?num)
    (farmer-location ?fs)
    (goat-location ?fs))
  (opposite-of ?fs ?ns)
  =>
  (duplicate ?node (search-depth =(+ 1 ?num))
    (parent ?node)
    (farmer-location ?ns)
    (goat-location ?ns)
    (last-move goat)))

(defrule MAIN::move-with-cabbage
  ?node <- (status (search-depth ?num)
    (farmer-location ?fs)
    (cabbage-location ?fs))
  (opposite-of ?fs ?ns)
(defmodule CONSTRAINTS
  (import MAIN deftemplate status)) 输入 status 模板

(defrule CONSTRAINTS::fox-eats-goat
  (declare (auto-focus TRUE))
  ?node <- (status (farmer-location ?s1)
               (fox-location ?s2~?s1)
               (goat-location ?s2))
  =>
  (retract ?node))

(defrule CONSTRAINTS::goat-eats-cabbage
  (declare (auto-focus TRUE))
  ?node <- (status (farmer-location ?s1)
               (goat-location ?s2~?s1)
               (cabbage-location ?s2))
  =>
  (retract ?node))

(defrule CONSTRAINTS::circular-path
  (declare (auto-focus TRUE))
  (status (search-depth ?sd1)
           (farmer-location ?fs)
           (fox-location ?xs)
           (goat-location ?gs)
           (cabbage-location ?cs))
  ?node <- (status (search-depth ?sd2&(< ?sd1 ?sd2))
                 (farmer-location ?fs)
                 (fox-location ?xs)
                 (goat-location ?gs)
                 (cabbage-location ?cs))
  =>
  (retract ?node))
(slot id (type FACT-ADDRESS SYMBOL) (allowed-symbols no-parent))
(multislot moves-list
  (type SYMBOL) (allowed-symbols no-move alone fox goat cabbage)))

(defrule SOLUTION::recognize-solution
  (declare (auto-focus TRUE))
  ?node <- (status (parent ?parent)
    (farmer-location shore-2)
    (fox-location shore-2)
    (goat-location shore-2)
    (cabbage-location shore-2)
    (last-move ?move))

  =>
    (retract ?node)
  (assert (moves (id ?parent) (moves-list ?move))))

(defrule SOLUTION::further-solution
  ?node <- (status (parent ?parent)
    (last-move ?move))
  ?mv <- (moves (id ?node) (moves-list $?rest))
  =>
    (modify ?mv (id ?parent) (moves-list ?move $rest)))

(defrule SOLUTION::print-solution
  ?mv <- (moves (id no-parent) (moves-list no-move $?m))
  =>
    (retract ?mv)
    (printout t t "Solution found: " t t) ; t 为换行符
    (bind ?length (length $?m))
    (bind ?i 1)
    (bind ?shore shore-2)
    (while (< ?i ?length)
      (bind ?thing (nth ?i $?m))
      (if (eq ?thing alone)
        then (printout t "Farmer moves alone to " ?shore "." t)
        else (printout t "Farmer moves with " ?thing " to " ?shore "." t))
      (if (eq ?shore shore-1)
        then (bind ?shore shore-2)
        else (bind ?shore shore-1))
      (bind ?i (+ 1 ?i))))

:==---------------------------------------------------------------------
dilemma2.lsp  ;:: Farmer’s Dilemma Problem
::::  This example uses COOL classes and message-handlers to solve the problem.
::::---------------------------------------------------------------------
:*** DEFCLASSES ***
:***---------------------------------------------------------------------
(defclass status
  (is-a USER)
  (role concrete)
  (slot farmer
    (create-accessor write)
    (create-accessor read)
    (create-accessor read-write))

<role>::=(role concrete | abstract)
(create-accessor) NONE | read | write | read-write)
(default shore-1))
(slot fox
  (create-accessor write)
  (default shore-1))
(slot goat
  (create-accessor write)
  (default shore-1))
(slot cabbage
  (create-accessor write)
  (default shore-1))
(slot parent
  (create-accessor write)
  (default no-parent))
(slot last-move
  (create-accessor write)
  (default no-move)))
:::*********************
:::DEFFUNCTIONS *
:::*********************
(deffunction contradiction
  (?f ?x ?g ?c ?d)
  (if (or (and (eq ?x ?g) (eq ?f ?x)) (and (eq ?g ?c) (eq ?f ?g)))
     then
     TRUE
   Else
     (any-instancep ((?s status))
       (and (eq ?s:farmer ?f)
         (eq ?s:fox ?x)
         (eq ?s:goat ?g)
         (eq ?s:cabbage ?c)
         (< ?s:search-depth ?d)))
     ))) :已有深度比 d 小的状态时为 TRUE

(deffunction opposite-shore
  (?value)
  (if (eq ?value shore-1)
    then
    shore-2
  Else
    shore-1))

(deffunction solve-dilemma ()
  (do-for-all-instances ((?a status))
    TRUE
    (send ?a delete)) ; 消息响应
    (make-instance start of status)
    (send [start] generate-moves))
:::*********************
:::DEFRULES *
:::*********************
(defrule start-it
=>
  (solve-dilemma))

:::DEFMESSAGE-HANDLERS
:::
(defmessage-handler status move-farmer
  ()
  (if (not (contradiction (opposite-shore ?self:farmer) ?self:fox
    then
      (bind ?x (make-instance (gensym) of status
                    (farmer (opposite-shore ?self:farmer))
                    (fox ?self:fox)
                    (goat ?self:goat)
                    (cabbage ?self:cabbage)
                    (last-move farmer)
                    (parent ?self)
                    (search-depth (+ ?self:search-depth 1))))
      (if (not (send ?x solution?))
        then
          (send ?x generate-moves)))
  )))

(defmessage-handler status move-goat
  ()
  (if (and (eq ?self:farmer ?self:goat) (not (contradiction
    then
      (bind ?x (make-instance (gensym) of status
                    (farmer (opposite-shore ?self:farmer))
                    (fox ?self:fox)
                    (goat (opposite-shore ?self:farmer))
                    (cabbage ?self:cabbage)
                    (last-move goat)
                    (parent ?self)
                    (search-depth (+ ?self:search-depth 1))))
      (if (not (send ?x solution?))
        then
          (send ?x generate-moves)))
  )))

(defmessage-handler status move-fox
  ()
  (if (and (eq ?self:farmer ?self:fox)
            (not (contradiction (opposite-shore ?self:farmer)
                                (opposite-shore ?self:fox)
    then
      (bind ?x (make-instance (gensym) of status
                    (farmer (opposite-shore ?self:farmer))
                    (fox (opposite-shore ?self:farmer))
                    (goat ?self:goat)
                    (cabbage ?self:cabbage)
                    (last-move fox)
                    (parent ?self)
                    (search-depth (+ ?self:search-depth 1))))
      (if (not (send ?x solution?))
        then
          (send ?x generate-moves)))
  )))
(last-move fox)
(parent ?self)
(search-depth (+ ?self:search-depth 1)))
(if (not (send ?x solution?))
    then
    (send ?x generate-moves))))

(defmessage-handler status move-cabbage
  ()
  (if (and (eq ?self:farmer ?self:cabbage)
            (not (contradiction (opposite-shore ?self:farmer)
                              ?self:fox ?self:goat
                              (opposite-shore ?self:farmer)
                              ?self:search-depth)))
      then
      (bind ?x (make-instance (gensym) of status
                               (farmer (opposite-shore ?self:farmer))
                               (fox ?self:fox)
                               (goat ?self:goat)
                               (cabbage (opposite-shore ?self:farmer))
                               (last-move cabbage)
                               (parent ?self)
                               (search-depth (+ ?self:search-depth 1)))
      (if (not (send ?x solution?))
          then
          (send ?x generate-moves))))

(defmessage-handler status generate-moves
  ()
  (send ?self move-farmer)
  (send ?self move-fox)
  (send ?self move-goat)
  (send ?self move-cabbage))

(defmessage-handler status print-solution
  ()
  (if (neq ?self:parent no-parent)
      then
      (send ?self:parent print-solution)
      (bind ?move-dest (dynamic-get ?self:last-move))
      (if (eq ?self:last-move farmer)
          then
          (printout t "Farmer moves alone to " ?move-dest "." crlf)
          else
          (printout t "Farmer moves with " ?self:last-move " to " ?move-dest "." crlf))))

(defmessage-handler status solution?
  ()
  (if (and (eq ?self:farmer shore-2) (eq ?self:fox shore-2)
      then
      (printout t crlf "Solution found:" crlf)
      (send ?self print-solution)
TRUE
else
FALSE))

:=====-----------------------------------------------
dilemma3.lsp :;; Farmer's Dilemma Problem
:;;  This example uses rules and object pattern matching to solve the problem.
:;; :===============================================
:;; ************
:;; * CLASSES *
:;; ************
:;; The status instances hold the state
:;; information of the search tree.

(defun status (is-a USER) 状态类
  (role concrete)
  (pattern-match reactive)
  (slot search-depth
    (create-accessor write)
    (type INTEGER) (range 1 ?VARIABLE) (default 1))
  (slot parent
    (create-accessor write)
    (type INSTANCE-ADDRESS) (default ?DERIVE))
  (slot farmer-location
    (create-accessor write)
    (type SYMBOL) (allowed-symbols shore-1 shore-2) (default shore-1))
  (slot fox-location
    (create-accessor write)
    (type SYMBOL) (allowed-symbols shore-1 shore-2) (default shore-1))
  (slot goat-location
    (create-accessor write)
    (type SYMBOL) (allowed-symbols shore-1 shore-2) (default shore-1))
  (slot cabbage-location
    (create-accessor write)
    (type SYMBOL) (allowed-symbols shore-1 shore-2) (default shore-1))
  (slot last-move
    (create-accessor write)
    (type SYMBOL) (allowed-symbols no-move alone fox goat cabbage)
    (default no-move))) 回溯的终点

:;; The moves instances hold the information of all the moves
:;; made to reach a given state.

(defun moves (is-a USER) 移动类
  (role concrete)
  (pattern-match reactive)
  (slot id
    (create-accessor write)
    (type INSTANCE))
  (multislot moves-list
    (create-accessor write)
    (type SYMBOL)
    (allowed-symbols no-move alone fox goat cabbage)))
(defclass opposite-of 对岸类
    (is-a USER)
    (role concrete)
    (pattern-match reactive)
    (slot value (create-accessor write))
    (slot opposite-value (create-accessor write)))

:::**********************
::* INITIAL STATE *
:::**********************

(definstances startups 初始化示例
    (of status)
    (of opposite-of (value shore-1) (opposite-value shore-2))
    (of opposite-of (value shore-2) (opposite-value shore-1)))

:::**********************
::* GENERATE PATH RULES *
:::**********************

(defrule move-alone
    ?node <- (object (is-a status)
        (search-depth ?num)
        (farmer-location ?fs))
        (object (is-a opposite-of) (value ?fs) (opposite-value ?ns))
        确定 farmer 的对岸
    =>
        (duplicate-instance ?node
            (search-depth (+ 1 ?num))
            (parent ?node)
            (farmer-location ?ns)
            (last-move alone)))

(defrule move-with-fox
    ?node <- (object (is-a status)
        (search-depth ?num)
        (farmer-location ?fs)
        (fox-location ?fs))
        (object (is-a opposite-of) (value ?fs) (opposite-value ?ns))
    =>
        (duplicate-instance ?node
            (search-depth (+ 1 ?num))
            (parent ?node)
            (farmer-location ?ns)
            (last-move fox)
            (fox-location ?ns)))

(defrule move-with-goat
    ?node <- (object (is-a status)
        (search-depth ?num)
        (farmer-location ?fs)
        (goat-location ?fs))
        (object (is-a opposite-of) (value ?fs) (opposite-value ?ns))
    =>
        (duplicate-instance ?node
            (search-depth (+ 1 ?num))
            (parent ?node)
            (farme...
(duplicate-instance ?node
  (search-depth (+ 1 ?num))
  (parent ?node)
  (farmer-location ?ns)
  (last-move goat)
  (goat-location ?ns)))

(defrule move-with-cabbage
  ?node <- (object (is-a status)
    (search-depth ?num)
    (farmer-location ?fs)
    (cabbage-location ?fs))
  (object (is-a opposite-of) (value ?fs) (opposite-value ?ns))
  =>
  (duplicate-instance ?node
    (search-depth (+ 1 ?num))
    (parent ?node)
    (farmer-location ?ns)
    (last-move cabbage)
    (cabbage-location ?ns)))

:::***************
::* CONSTRAINT VIOLATION RULES *
:::***************

(defrule fox-eats-goat
  (declare (salience 200))
  ?node <- (object (is-a status)
    (farmer-location ?s1)
    (fox-location ?s2 & ?s1)
    (goat-location ?s2))
  =>
  (unmake-instance ?node)) 撤消示例

(defrule goat-eats-cabbage
  (declare (salience 200))
  ?node <- (object (is-a status)
    (farmer-location ?s1)
    (goat-location ?s2 & ?s1)
    (cabbage-location ?s2))
  =>
  (unmake-instance ?node))

(defrule circular-path
  (declare (salience 200))
  (object (is-a status)
    (search-depth ?sd1)
    (farmer-location ?fs)
    (fox-location ?xs)
    (goat-location ?gs)
    (cabbage-location ?cs))
  ?node <- (object (is-a status)
    (search-depth ?sd2 & (? ?sd1 ?sd2)))
(farmer-location ?fs)
(fire-location ?xf)
(fire-location ?yf)
(fire-location ?zf)
(fire-location ?sf)

=>

(unmake-instance ?node)

::* FIND AND PRINT SOLUTION RULES *

(defun recognize-solution (node)
    (declare (salience 100))

    ?node <- (object (is-a status)
                  (parent ?parent)
                  (farmer-location shore-2)
                  (fox-location shore-2)
                  (goat-location shore-2)
                  (cabbage-location shore-2)
                  (last-move ?move))

    =>

    (unmake-instance ?node)
    (make-instance of moves (id ?parent) (moves-list ?move)))

(defun further-solution (state)
    (declare (salience 100))

    ?state <- (object (is-a status)
                    (parent ?parent)
                    (last-move ?move))

    ?mv <- (object (is-a moves)
                  (id ?state)
                  (moves-list $?rest))

    =>

    (modify-instance ?mv (id ?parent) (moves-list ?move ?rest)))

(defun print-solution (mv)
    (declare (salience 100))

    ?mv <- (object (is-a moves)
                  :id [no-parent]
                  (moves-list no-move $?m))

    =>

    (unmake-instance ?mv)
    (printout t "Solution found: " t t)
    (bind ?length (length ?m))
    (bind ?i 1)
    (bind ?shore shore-2)
    (while (<= ?i ?length)
                (bind ?thing (nth$ ?i ?m))
                (if (eq ?thing alone)
                                then (printout t "Farmer moves alone to " ?shore " t"
                                else (printout t "Farmer moves with " ?thing to " ?shore ". t"))
                (if (eq ?shore shore-1)
                                then (bind ?shore shore-2)
else (bind ?shore shore-1)
   (bind ?i (+ 1 ?i)))

cam.lsp  例题：野人与传教士问题，三个野人和三名传教士过河，船每次只能载两项，如果野人超过传教士则野人会吃掉传教士，如何安全过河？

;::: Cannibals and Missionaries Problem
;:::
;:::  Another classic AI problem. The point is
;:::  to get three cannibals and three missionaries
;:::  across a stream with a boat that can only
;:::  hold two people. If the cannibals outnumber
;:::  the missionaries on either side of the stream,
;:::  then the cannibals will eat the missionaries.
;:::
;:::  CLIPS Version 6.01 Example
;:::
;:::  To execute, merely load, reset and run.
;:::---------------------------------------------------------------------------------

(defmodule MAIN
  (export deftemplate status)
  (export defglobal initial-missionaries initial-cannibals))

;::* ****************
;::*   TEMPLATES *
;::* ****************

;::: The status facts hold the state
;::: information of the search tree.

(deftemplate MAIN::status
  (slot search-depth (type INTEGER) (range 1 ?VARIABLE))
  (slot parent (type FACT-ADDRESS SYMBOL) (allowed-symbols no-parent))
  (slot shore-1-missionaries (type INTEGER) (range 0 ?VARIABLE))
  (slot shore-1-cannibals (type INTEGER) (range 0 ?VARIABLE))
  (slot shore-2-missionaries (type INTEGER) (range 0 ?VARIABLE))
  (slot shore-2-cannibals (type INTEGER) (range 0 ?VARIABLE))
  (slot boat-location (type SYMBOL) (allowed-values shore-1 shore-2))
  (slot last-move (type STRING)))

;::* ****************
;::*   INITIAL STATE *
;::* ****************

(defglobal MAIN ?*initial-missionaries* = 3
   ?*initial-cannibals* = 3)

(deffacts MAIN::initial-positions
  (status (search-depth 1)
   (parent no-parent)
   (shore-1-missionaries ?*initial-missionaries*))
(shore-2-missionaries 0)
(shore-1-cannibals ?*initial-cannibals*)
(shore-2-cannibals 0)
(boat-location shore-1)
(last-move "No move.

(deffacts MAIN::boat-information
  (boat-can-hold 2))

::*FUNCTION FOR MOVE DESCRIPTION STRING *
::*FUNCTION FOR MOVE DESCRIPTION STRING *

船上情况→输出字符串
(deffunction MAIN::move-string (?missionaries ?cannibals ?shore)
  (switch ?missionaries
    (case 0 then ; 无传教士时
      (if (eq ?cannibals 1)
        (then (format nil "Move 1 cannibal to %s,%n" ?shore)
           (else (format nil "Move %d cannibals to %s,%n" ?cannibals ?shore)))
      (case 1 then
        (switch ?cannibals
          (case 0 then
            (format nil "Move 1 missionary to %s,%n" ?shore))
          (case 1 then
            (format nil "Move 1 missionary and 1 cannibal to %s,%n" ?shore))
          (default then
            (format nil "Move 1 missionary and %d cannibals to %s,%n" ?cannibals ?shore))))
    (default
      (switch ?cannibals
        (case 0 then
          (format nil "Move %d missionaries to %s,%n" ?missionaries ?shore))
        (case 1 then
          (format nil "Move %d missionaries and 1 cannibal to %s,%n" ?missionaries ?shore))
        (default then
          (format nil "Move %d missionary and %d cannibals to %s,%n" ?missionaries ?cannibals ?shore)))))

::*GENERATE PATH RULES *
::*GENERATE PATH RULES *

(defrule MAIN::shore-1-move
  ?node <- (status (search-depth ?num)
    (boat-location shore-1)
    (shore-1-missionaries ?s1m)
    (shore-1-cannibals ?s1c)
    (shore-2-missionaries ?s2m)
    (shore-2-cannibals ?s2c))
  (boat-can-hold ?limit)
  =>
  (bind ?max-missionaries (min ?s1m ?limit)) ; 当前可移动的传教士数
移动 missionaries 个传教士和 cannibals 个野人过河

(loop-for-count (missionaries 0 ?max-missionaries) ; 循环生成所有后续态 m=0 to max
  (bind ?min-cannibals (max 0 (- 1 ?missionaries))) ; m=0 时 min=1，其它 min=0
  (bind ?max-cannibals (min ?s1c (- ?limit ?missionaries))) ; 船上空位数(=limit-m)
  (loop-for-count (cannibals ?min-cannibals ?max-cannibals)
    (duplicate ?node (search-depth =(+ 1 ?num))
      (parent ?node)
      (shore-1-missionaries (- ?s1m ?missionaries))
      (shore-1-cannibals (- ?s1c ?cannibals))
      (shore-2-missionaries (+ ?s2m ?missionaries))
      (shore-2-cannibals (+ ?s2c ?cannibals))
      (boat-location shore-2)
      (last-move (move-string ?missionaries ?cannibals shore-2)))))))

(defrule MAIN::shore-2-move
  ?node <- (status (search-depth ?num)
    (boat-location shore-2)
    (shore-1-missionaries ?s1m)
    (shore-1-cannibals ?s1c)
    (shore-2-missionaries ?s2m)
    (shore-2-cannibals ?s2c)
    (boat-can-hold ?limit)
  )
  =>
  (bind ?max-missionaries (min ?s2m ?limit))
  (loop-for-count (missionaries 0 ?max-missionaries)
    (bind ?min-cannibals (max 0 (- 1 ?missionaries)))
    (bind ?max-cannibals (min ?s2c (- ?limit ?missionaries)))
    (loop-for-count (cannibals ?min-cannibals ?max-cannibals)
      (duplicate ?node (search-depth =(+ 1 ?num))
        (parent ?node)
        (shore-1-missionaries (+ ?s1m ?missionaries))
        (shore-1-cannibals (+ ?s1c ?cannibals))
        (shore-2-missionaries (- ?s2m ?missionaries))
        (shore-2-cannibals (- ?s2c ?cannibals))
        (boat-location shore-1)
        (last-move (move-string ?missionaries ?cannibals shore-1)))))))

:::；***************
::* CONSTRAINT VIOLATION RULES *
:::；***************

(defunmodule CONSTRAINTS
  (import MAIN deftemplate status))

(defunmodule CONSTRAINTS::cannibals-eat-missionaries
  (declare (auto-focus TRUE))
  ?node <- (status (shore-1-missionaries ?s1m)
    (shore-1-cannibals ?s1c)
    (shore-2-missionaries ?s2m)
    (shore-2-cannibals ?s2c)
    (test (or (and (> ?s2c > ?s2m) (< ?s2m 0)) ; s2c<s2m 且 s2m<0 时，野人数超过传教士数
      (and (> ?s1c > ?s1m) (< ?s1m 0)))) ; s1c>s1m 且 s1m<0 时，野人数超过传教士数
  )
  =>

32
(retract ?node)

(defrule CONSTRAINTS::circular-path 删除重复的路
  (declare (auto-focus TRUE))
  (status (search-depth ?sd1)
    (boat-location ?bl)
    (shore-1-missionaries ?s1m)
    (shore-1-cannibals ?s1c)
    (shore-2-missionaries ?s2m)
    (shore-2-cannibals ?s2c))
  ?node <- (status (search-depth ?sd2&(() ?sd1 ?sd2))
    (boat-location ?bl)
    (shore-1-missionaries ?s1m)
    (shore-1-cannibals ?s1c)
    (shore-2-missionaries ?s2m)
    (shore-2-cannibals ?s2c))
  =>
  (retract ?node)

;;:**************************
;;:* FIND AND PRINT SOLUTION RULES *
;;:**************************

(deffunction SOLVED
  (assert (solve))
  =>
  (display (solve)))

(deffunction SOLVED::solution
  (assert (solve))
  =>
  (display (solve)))

(defmodule SOLUTION
  (import MAIN deftemplate status)
  (import MAIN defglobal initial-missionaries initial-cannibals))

(deftemplate SOLUTION::moves
  (slot id (type FACT-ADDRESS SYMBOL) (allowed-symbols no-parent))
  (multislot moves-list
    (type STRING)))

(defrule SOLUTION::recognize-solution
  (declare (auto-focus TRUE))
  ?node <- (status (parent ?parent)
    (shore-2-missionaries ?m&: (= ?m #initial-missionaries*))
    (shore-2-cannibals ?c&: (= ?c #initial-cannibals*))
    (last-move ?move))
  =>
  (retract ?node)
  (assert (moves (id ?parent) (moves-list ?move))))

(defrule SOLUTION::further-solution
  ?node <- (status (parent ?parent)
    (last-move ?move))
  ?mv <- (moves (id ?node) (moves-list $?rest))
  =>
  (modify ?mv (id ?parent) (moves-list ?move ?rest)))

(defrule SOLUTION::print-solution
  ?mv <- (moves (id no-parent) (moves-list “No move.” $?m))
  =>
  (retract ?mv)
(printout t t "Solution found: " t t)
(progn$ (?move ?m) (printout t ?move))

(progn$ <list-spec> <expression>*
<list-spec>::=<multifield-expression.> | (<list-variable> <multifield-expression>)

Perform a set of actions for each field of a multifield value.
摘录


P22. Procedural semantics: the idea is that a program can be said to understand a query if it can answer the question correctly, and it can be said to understand a command if it can carry the command out. This is a rather operational view of understanding, that is, a view which explains understanding in terms of behaviour, rather than in terms of mental operations.

P98-117 Representing Uncertainty
Conditional probability    Bayes’ rule
Certainty factors
Fuzzy sets
Possibility theory

P131 Why LISP is almost wonderful
LISP has two features: simplicity and meta-programming, which make it almost infinitely extensible and customizable. Add to this the fact that LISP is first and foremost an interpreted language, and one has a language designed for exploratory programming.

Certainly the medium of symbolic computation is well suited to the implementation of structures for representing knowledge, but the symbolic level of analysis does not tell you what those structures should be. This is rather like the problem we encountered with the predicate calculus. Logic lets you represent and reason about the world, but it does not tell you how to organize your representation or go about your reasoning – that’s your problem.

There would be a level of analysis above the symbol level which, properly understood, would constrain the set of possible representations for solving some problem. Newell (1982) calls this knowledge level, and suggests that knowledge should be characterized functionally, in terms of what it does, rather than structurally.

It is perhaps not too surprising that neither lambda calculus (LISP 的计算模型) nor predicate calculus (PROLOG 的计算模型) is sufficient to unlock the mysteries of intelligent behaviour; they were not designed for that purpose.

P135-151 Production Systems
Productions are really grammar rules for manipulating strings of symbols, sometimes called rewrite rules. Post (1943) studied the properties of rules systems based on productions.

P384 Truth Maintenance Systems
Mechanisms for keeping track of dependencies and detecting inconsistency are often referred to as truth
maintenance systems (TMSs), sometimes called reason maintenance systems.

P401 Diagnosis from First Principle
Many extant (现存) expert systems perform diagnosis by the method of heuristic classification. In this method, diagnostic knowledge is represented mainly in terms of heuristic rules, which perform a mapping between data abstractions (typical symptoms) and solution abstractions (typical disorders). Such a representation of knowledge is sometimes called ‘shallow’ because it does not contain much information about the causal mechanisms underlying the relationship between symptoms and disorders. The rules typically reflect empirical associations derived from experience, rather than a theory of how the device (or organism) under diagnosis actually works. The latter is sometimes called ‘deep’ knowledge, because it involves understanding the structure of the device and the way its components function.

P456 Summary and Conclusions
The use of AI architectures, search methods or programming languages does not prevent knowledge engineers from availing themselves of suitable methods from Applied Mathematics, Computer Science, or any other relevant discipline. Knowledge engineers building an application should always examine existing methods for solving the problem and take the trouble to analyse their strengths and weaknesses. Some parts of the problem may yield to purely algorithmic or mathematical solutions, in which case it is foolish not to use such methods as long as they are computationally efficient.

What Artificial Intelligence offers is a collection of insights, techniques and architectures for complex problem solving in cases where purely algorithmic or mathematical solutions are either unknown or demonstrably inefficient.

The knowledge representation is an empirical and experimental research area, and not merely an arena for philosophical debate. The old issues – such as ‘Can machines think?’ and ‘How is knowledge really represented in the human brain?’ – seems increasingly irrelevant to the Modern Period of AI. The key question is rather ‘What can we do with this technology?’, and it can only be answered by a concerted programme of theoretical and practical work.

Production systems have turned out to be a surprisingly effective tool for encoding and applying human expertise, particularly where the knowledge is plentiful but relatively unstructured. This style of programming is not particularly easy, and the creation and maintenance of large rule-based programs poses its own problems.

Systems based on frames and objects have proved particularly useful in applications where a detailed knowledge of domain objects and procedures is essential to performing the task.

Research on procedural deduction has made substantial advances over the last 10 years, particularly in the field of logic programming. How far this work will impact on practical expert systems applications remains to be seen.

Although AI has produced a variety of representation languages, they all have certain things in common.
Firstly, they are declarative, in the sense that they describe knowledge relevant to solving some task rather than prescribing how the task should be done. Secondly, they are highly modular. As well as concealing implementation details from the user of the language, modules of knowledge conceal implementation detail from each other, communicating through global data structures (as in production systems and blackboard systems) or via strict protocols (as in object-oriented systems). Thirdly, the analogue of procedure invocation in such languages tends to be pattern-directed. Thus the firing of rules in production systems, the triggering of knowledge sources in blackboard systems, and the resolution of clauses in deduction systems all rely on a form of pattern matching.

Knowledge representation languages of whatever kind are normally implemented as pattern-directed inference systems. The resultant programs consist of a number of relatively independent modules (rules, structures or clauses) which are matched against incoming data and which manipulate data structures.


Artificial intelligence is that branch of computer science dealing with symbolic, nonalgorithmic methods of problem solving. The definition focuses on two characteristics of computer programs:
1. Numeric versus symbolic: Computers were originally designed specifically to process numbers (numeric processing). People, however, tend to think symbolically; our intelligence seems to be based, in part on our mental ability to manipulate symbols rather than just numbers. Although symbolic processing is at the core of AI, this does not mean that AI does not involve math; rather, the emphasis in AI is on manipulation of symbols.
2. Algorithmic versus nonalgorithmic: An algorithm is a step-by-step procedure that has well-defined starting and ending points and that is guaranteed to reach a solution to a specific problem. Computer architecture readily lends itself to this step-by-step approach. Many human reasoning processes, however, tend to be nonalgorithmic; in other words, our mental activities consist of more than just following logical, step-by-step procedures.


Artificial intelligence: a branch of computer science concerned with the design and implementation of programs which are capable of emulating human cognitive skills such as problem solving, visual perception and language understanding.


Patrick Henry Winston is Director of the Artificial Intelligence Laboratory at MIT. He is well known throughout the world both for his contributions to artificial intelligence and for his clear and insightful writing about the field.

Current estimates suggest there may be on the order of $10^{11}$ neurons per person. In the cerebellum (小
脑）---that part of the brain that is crucial to motor coordination---a single neuron may receive inputs from as many as $10^5$ synapses(连接). Each brain has on the order of $10^{10}$ synapses. You can get a better feel for numbers like that if you know that there are as many synapses in one brain as there would be characters in about 300 such libraries (the United States Library of Congress).

P119-161  Rules and Rule Chaining
P14-19.

- Artificial intelligence is the study of the computations that make it possible to perceive, reason, and act.
- The engineering goal of artificial intelligence is to solve real-world problems; the scientific goal of Artificial Intelligence is to explain various sorts of intelligence.
- Applications of artificial intelligence should be judged according whether there is a well-defined task, an implemented program, and a set of identifiable principles.
- Artificial intelligence can help us to solve difficult, real-world problems, creating new opportunities in business, engineering, and many other application areas.
- Artificial intelligence sheds new light on questions traditionally asked by psychologists, linguists, and philosophers. A few rays of this new light can help us to be more intelligent.

P400.

Most of the problems in practice can be modeled in one or more of the six strategic forms: competitive, cooperative, weighted, prioritized, hierarchical, and adaptive rule formation. (竞争、合作、加权、优先级、分层的、自适应)
第 1 节 专家系统课程的介绍及要求

1. 课程的性质及其与其它课程的联系与区别
2. 授课大纲
3. 讲义、教材及参考资料
4. 考核方式

第 2 节 人工智能原理简述

1. 问题求解的基本原理
2. 符号逻辑基础

第 3 节 专家系统的基本原理

第 4 节 基板结构和编程技术

1. 基板结构和编程技术
2. 知识工程

第 5 节 知识表示

第 6 节 知识表示——语义网络

第 7 节 专家系统工具 CLIPS

第 8 节 CLIPS6.1 程序示例

摘录