SISTEMAS EXPERTOS

Principios y programación

Tercera edición

Joseph Giarratano
University of Houston–Clear Lake

Gary Riley
Calico Technology

International Thomson Editores

Mexico • Albany • Boston • Johannesburgo • Londres • Madrid • Melbourne • Nueva York
San Francisco • San Juan, PR • Santiago • São Paulo • Singapur • Tokio • Toronto • Washington

Traducción del libro *Expert Systems, principles and programming*, publicado en inglés por
PWS Publishing Company © 1998 ISBN 0-534-95053-1

Sistemas expertos, principios y programación
ISBN 970-686-059-2
Derechos reservados respecto a la edición en español.
© 2001 por International Thomson Editores, S. A. de C. V.
International Thomson Editores, S. A. de C. V. es una empresa de
Thomson Learning

México y América Central
Séneca 53, Colonia Polanco
México, D. F. 11560
Tel. (525) 281-2906
Fax (525) 281-2656
editor@thomsonlearning.com.mx
MÉXICO

El Caribe
Tel. (787) 758-7580
Fax (787) 758-7573
thomson@coqui.net
Hato Rey, PUERTO RICO

América del Sur
Tel. (54-11)4325-2236
Fax (54-11)4328-1829
thomson@pop.ba.net
Buenos Aires, ARGENTINA

España
Tel. (3491) 446-3350
Fax (3491) 445-6218
Madrid, ESPAÑA

Traducción
Eloy Pineda
Traductor profesional

Revisión técnica
Manuel López Michelone
Universidad Iberoamericana

Director editorial y de producción: Miguel Ángel Toledo Castellanos
Editor de desarrollo: Pedro de la Garza Rosales
Gerente de producción: René Garay Argueta
Corrección de estilo: Sergio Antonio Durán Reyes
Editora de Producción: Olga Adriana Sanchez Navarrete
Tipografía: MC Graphic
Lecturas: Edmundo Bassail y Aurelio García

987654321 0IV0

Queda prohibida la reproducción o transmisión total o parcial del texto de la presente obra bajo cualesquiera formas, electrónica o mecánica, incluyendo el fotocopiado, el almacenamiento en algún sistema de recuperación de información, o el grabado, sin el consentimiento previo y por escrito del editor.

All rights reserved. No part of this work covered by the copyright hereon may be reproduced or used in any form or by any means—graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems—without the written permission of the publisher.

Impreso en Mexico
Printed in Mexico

Contenido

PREFACIO xi

PRÓLOGO xiii

CAPÍTULO 1: INTRODUCCIÓN A LOS SISTEMAS EXPERTOS 1

1.1 Introducción 1
1.2 ¿Qué es un sistema experto? 1
1.3 Ventajas de los sistemas expertos 4
1.4 Conceptos generales de los sistemas expertos 5
1.5 Características de un sistema experto 8
1.6 El desarrollo de la tecnología de sistemas expertos 10
1.7 Aplicaciones y dominios de los sistemas expertos 15
1.8 Lenguajes, shells y herramientas 21
1.9 Elementos de un sistema experto 23
1.10 Sistemas de producción 28
1.11 Paradigmas de procedimientos 32
1.12 Paradigmas que no son de procedimientos 38
1.13 Sistemas neuronales artificiales 43
1.14 Sistemas expertos interconectados y aprendizaje inductivo 49
1.15 Resumen 50
Problemas 50
Bibliografía 51
Una breve bibliografía histórica 54

CAPÍTULO 2: LA REPRESENTACIÓN DEL CONOCIMIENTO 57

2.1 Introducción 57
2.2 El significado de conocimiento 57
2.3 Producciones 60
2.4 Redes semánticas 63
2.5 Tripleta objeto-atributo-valor 66
2.6 PROLOG y redes semánticas 67
2.7 Dificultades con redes semánticas 72
2.8 Esquemas 73
2.9 Marcos 74

2.10 Dificultades con los marcos 77
2.11 Lógica y conjuntos 78
2.12 Lógica de proposiciones 81
2.13 La lógica de predicado de primer orden 86
2.14 El cuantificador universal 87
2.15 El cuantificador existencial 88
2.16 Cuantificadores y conjuntos 90
2.17 Limitaciones de la lógica de predicado 91
2.18 Resumen 91
Problemas 91
Bibliografía 94

CAPÍTULO 3: MÉTODOS DE INFERENCIA 97

3.1 Introducción 97
3.2 Árboles, rejillas y gráficas 97
3.3 Estado y espacios de problema 101
3.4 Árboles y objetivos y-o 106
3.5 Lógica deductiva y silogismos 109
3.6 Reglas de inferencia 114
3.7 Limitaciones de la lógica de proposición 122
3.8 Lógica de predicado de primero orden 123
3.9 Sistemas lógicos 125
3.10 Resolución 128
3.11 Sistemas de resolución y deducción 131
3.12 Razonamiento superficial y causal 133
3.13 Resolución y lógica de predicado de primer orden 137
3.14 Encadenamiento hacia delante y hacia atrás 143
3.15 Otros métodos de inferencia 148
3.16 Metaconocimiento 156
3.17 Resumen 157
Problemas 158
Bibliografía 161

CAPÍTULO 4: RAZONAMIENTO EN CONDICIONES DE INCERTIDUMBRE 165

4.1 Introducción 165
4.2 Incertidumbre 165
4.3 Tipos de error 166
4.4 Errores e inducción 168
4.5 Probabilidad clásica 169

4.6 Probabilidades experimentales y subjetivas 173
4.7 Probabilidades compuestas 175
4.8 Probabilidades condicionales 178
4.9 Razonamiento hipotético e inducción hacia atrás 183
4.10 Razonamiento temporal y cadenas de Markov 186
4.11 Particularidades de creencia 191
4.12 Suficiencia y necesidad 193
4.13 Incertidumbre en cadenas de inferencia 195
4.14 La combinación de evidencia 200
4.15 Redes de inferencia 207
4.16 La propagación de probabilidades 216
4.17 Resumen 220
Problemas 220
Bibliografía 223

CAPÍTULO 5: RAZONAMIENTO INEXACTO 227

5.1 Introducción 227
5.2 Incertidumbre y reglas 227
5.3 Factores de certidumbre 233
5.4 Teoría de Dempster y Shafer 243
5.5 Razonamiento aproximado 256
5.6 El estado de incertidumbre 300
5.7 Resumen 301
Problemas 301
Bibliografía 305

CAPÍTULO 6: DISEÑO DE SISTEMAS EXPERTOS 309

6.1 Introducción 309
6.2 Selección del problema apropiado 309
6.3 Etapas del desarrollo de un sistema experto 311
6.4 Errores en etapas de desarrollo 313
6.5 Ingeniería de software y sistemas expertos 315
6.6 El ciclo de vida del sistema experto 317
6.7 Un modelo detallado de ciclo de vida 320
6.8 Resumen 325
Problemas 326
Bibliografía 326

CAPÍTULO 7: INTRODUCCIÓN A CLIPS 327

7.1 Introducción 327
7.2 CLIPS 328
7.3 Notación 328
7.4 Campos 330
7.5 Entrada y salida en CLIPS 333
7.6 Hechos 334
7.7 Adición y eliminación de hechos 337
7.8 Modificación y duplicación de hechos 340
7.9 El comando watch 341
7.10 La construcción deffacts 342
7.11 Los componentes de una regla 344
7.12 La agenda y ejecución 346
7.13 Comandos para manipulación de construcciones 350
7.14 El comando printout 353
7.15 Uso de varias reglas 353
7.16 El comando set-break 355
7.17 Carga y guardado de construcciones 357
7.18 Construcciones de comentario 358
7.19 Resumen 359
Problemas 360

CAPÍTULO 8: COMPARACIÓN DE PATRONES 365

8.1 Introducción 365
8.2 Variables 365
8.3 Uso múltiple de variables 366
8.4 Direcciones de hecho 367
8.5 Comodines de un solo campo 370
8.6 Mundo de bloques 371
8.7 Comodines y variables de campo múltiple 376
8.8 Restricciones de campo 382
8.9 Funciones y expresiones 385
8.10 Suma de valores utilizando reglas 389
8.11 Funcion bind 391
8.12 Funciones de Entrada/Salida 392
8.13 Resumen 398

CAPÍTULO 9: COMPARACIÓN AVANZADA DE PATRONES 405

9.1 Introducción 405

9.2 El juego de los palillos 405
9.3 Técnicas de alimentación de datos 405
9.4 Funciones de predicado 407
9.5 El elemento condicional test 407
9.6 La restricción del campo de predicado 410
9.7 La restricción del campo del valor devuelto 411
9.8 El programa palillos 412
9.9 El elemento condicional *OR* 413
9.10 El elemento condicional *AND* 415
9.11 El elemento condicional *NOT* 417
9.12 El elemento condicional *EXITS* 419
9.13 El elemento condicional *FORALL* 422
9.14 El elemento condicional *LOGICAL* 424
9.15 Comandos de uso general 428
9.16 Resumen 430
Problemas 430

CAPÍTULO 10: DISEÑO MODULAR Y CONTROL DE EJECUCIÓN 437

10.1 Introducción 437
10.2 Atributos de deftemplate 437
10.3 Prominencia 445
10.4 Fases y hechos de control 448
10.5 Uso inapropiado de la prominencia 453
10.6 El elemento condicional defmodule 456
10.7 Importación y exportación de hechos 459
10.8 Módulos y control de ejecución 463
10.9 Resumen 471
Problemas 471

CAPÍTULO 11: EFICIENCIA DE LOS LENGUAJES BASADOS EN REGLAS 477

11.1 Introducción 477
11.2 El algoritmo de comparación de patrones de Rete 477
11.3 La red patrón 480
11.4 La red de unión 483
11.5 La importancia del orden de los patrones 486
11.6 Ordenamiento de patrones por eficiencia 492
11.7 Las variables de campo múltiple y la eficiencia 493
11.8 El elemento condicional test y la eficiencia 493
11.9 Restricciones integradas de comparación de patrones 495

ra, predicción del clima, predicciones del mercado de valores, recomendaciones de compra al consumidor y consejos dietéticos.

El plan sugerido para un curso de un semestre es el siguiente:

1. Cubrir el capítulo 1 para ofrecer una introducción rápida a los sistemas expertos. En particular asignar los problemas 1, 2 y 3.

2. Cubrir del capítulo 7 al 10 para introducir la sintaxis de CLIPS. Es de gran ayuda para los estudiantes volver a codificar el problema 2 del capítulo 1 y comparar el método del sistema experto con el que usaron originalmente en el capítulo 1. Este contraste es muy útil para puntualizar las diferencias entre un lenguaje basado en reglas, como CLIPS y LISP, PROLOG, o cualquiera que sea el leguaje original usado para el problema 2.

3. Los capítulos 11 y 12 deben cubrirse para dar al curso un mayor énfasis en programación. El capítulo 11, que trata de la eficiencia de los lenguajes basados en reglas no es esencial para un curso introductorio. Debe analizarse si hay interés en construir sistemas expertos de gran tamaño o si la eficiencia es lo importante. En el capítulo 12 se presenta una serie de temas que ilustran interesantes problemas resueltos utilizando CLIPS.

Como alternativa, después del capítulo 10 el instructor puede regresar a la sección teórica. Si los estudiantes tienen bases fuertes en lógica y PROLOG, puede omitirse la mayor parte de los capítulos 2 y 3. Tanto los estudiantes que tengan un curso de introducción en AI basado en LISP como los que no tengan nada de esto se beneficiarán de los capítulos 2 y 3, sobre todo si se desea enfatizar la lógica y la teoría fundamental de los sistemas expertos. Si los estudiantes tienen bases fuertes en probabilidad y estadística, puede evitarse el material del capítulo 4 hasta la sección 4.11.

4. Desde la sección 4.11 hasta el final del capítulo 4 y en todas las secciones del capítulo 5, se analizan temas avanzados de los sistemas expertos relacionados con el manejo de la incertidumbre. Estos temas incluyen inferencia probabilística, factores de certidumbre, la teoría Dempster-Shafer y la teoría de la lógica borrosa. Los estudiantes llegarán a comprender estos métodos con amplitud suficiente para leer los artículos de actualidad en el medio y comenzar a realizar investigación, si así lo desean.

5. En el capítulo 6 se analiza la ingeniería de software de los sistemas expertos y su significado, para aquellos estudiantes que planean trabajar en sistemas de gran tamaño. No es necesario analizar este capítulo antes de asignar proyectos finales. En realidad, sería mejor cubrir este capítulo más adelante, para que el estudiante pueda apreciar todos los factores que se incluyen en la construcción de un sistema experto de calidad.

Reconocimientos

Durante la escritura de este libro, muchas personas hicieron comentarios de gran ayuda: entre ellos Chris Culbert, Dan Bochsler, Tim Dawn, Ted Leibfried, Jack Aldridge, Bob Lea, Chet Lund, Andre de Korvin, Arlan DeKock, Terry Feagin, y los revisores anónimos. Agradecemos la sobresaliente cooperación y las facilidades otorgadas por el Centro de Ediciones Desktop en el Centro AV, de la Universidad de Houston en Clear Lake. Gracias también a Phillip Johnston, Kwok-Bun Yue, Naveed Quraishi, Jena Giarratano, Antony Giarratano y al doctor Andrew Griffin.

Prólogo a la primera edición

Cuando la AI nació a mediados de los años 50, sus inquietudes principales estaban dirigidas a los juegos, la planeación y la solución de problemas. En aquella época habría sido muy difícil predecir que tres décadas después, las áreas de aplicación más importantes de la AI estarían centradas en el conocimiento de la ingeniería y particularmente en los sistemas expertos.

Los sistemas expertos como los conocemos hoy tienen sus raíces en el trabajo pionero que Feigenbaum, Lederberg, Shortliffe y Buchanan realizaron en la Universidad de Stanford a finales de los años sesenta y principios de los setenta. Lo más notable de este trabajo, y de MYCIN en particular, es que todavía sirve de modelo para gran parte de la actividad actual en sistemas expertos y, de manera más general, para aquellos basados en conocimiento.

En años más recientes, los éxitos demostrados de los sistemas expertos en áreas de aplicación especializadas, como el diagnóstico médico y la configuración de sistemas para computadora, han producido un súbito incremento del interés en una variedad de aplicaciones más amplia, que va desde la administración de planes de batalla y el enrutamiento óptimo, hasta el diagnóstico de fallas, la selección del portafolios recomendable y la evaluación de solicitantes de préstamos.

La multiplicación de aplicaciones ha generado dos necesidades distintas: a) el desarrollo de una mejor comprensión acerca de cómo tratar los problemas clave: la representación del conocimiento, la inferencia y la administración de la incertidumbre; y b) el desarrollo de shells y/o lenguajes de programación para los sistemas expertos, que minimicen el esfuerzo necesario para construir un sistema de representación de conocimiento y un motor de inferencia para una aplicación en particular.

El libro escrito por Joseph Giarratano y Gary Riley es el único en esta coyuntura que atiende estas dos necesidades. Su primera parte trata con autoridad los temas básicos que se encuentran tras la representación del conocimiento, la inferencia lógica y el manejo de la incertidumbre. En esta parte, son particularmente notables los capítulos de razonamiento bajo incertidumbre, que presentan un cúmulo de información acerca de los métodos básicos que se utilizan en la actualidad. En estos capítulos se incluyen agudas descripciones de las técnicas que emplean la teoría de probabilidad clásica, factores de certidumbre *à la* MYCIN y PROSPECTOR, la teoría Dempster-Shafer y la lógica borrosa. Ningún otro texto ofrece una cobertura tan amplia.

La segunda parte presenta un recuento detallado de CLIPS, un lenguaje de programación para los sistemas expertos escrito en C, que tiene el propósito de servir como una herramienta versátil para el desarrollo y ejecución de estos sistemas. CLIPS está relacionado con OPS5 y con ART, y tiene capacidad para inferir bajo incertidumbre.

En combinación, las dos partes proporcionan un recuento moderno y compresible de los principios que rigen a los sistemas expertos y su práctica. Escribir un libro de este tipo es, por supuesto, una tarea difícil y el lector debe estar preparado para invertir una gran cantidad de tiempo y esfuerzo para asimilar el gran volumen de información presentada por los autores.

11.10 Comparación de reglas generales con reglas específicas 496
11.11 Funciones de procedimientos 498
11.12 Reglas simples comparadas con reglas complejas 500
11.13 Carga y guardado de hechos 503
11.14 Resumen 504

CAPÍTULO 12: EJEMPLOS DE DISEÑO DE SISTEMAS EXPERTOS 509

12.1 Introducción 509
12.2 Factores de certidumbre 509
12.3 Árboles de decisión 513
12.4 Encadenamiento hacia atrás 526
12.5 Un problema de monitoreo 538
12.6 Resumen 554
Problemas 554
Bibliografía 556

APÉNDICE A: ALGUNAS EQUIVALENCIAS ÚTILES 557

APÉNDICE B: ALGUNOS CUANTIFICADORES ELEMENTALES Y SUS SIGNIFICADOS 559

APÉNDICE C: ALGUNAS PROPIEDADES DE CONJUNTO 561

APÉNDICE D: INFORMACIÓN DE SOPORTE DE CLIPS 563

APÉNDICE E: RESUMEN DE COMANDOS Y FUNCIONES DE CLIPS 565

APÉNDICE F: CLIPS BNF 579

ÍNDICE 585

Prefacio

Desde su introducción comercial a principios de los años ochenta los sistemas expertos ha experimentado un crecimiento y una popularidad tremendos. Hoy se utilizan en los negocios la ciencia, la ingeniería, la manufactura y muchos otros campos.

El propósito de este libro es enseñar a los estudiantes la teoría de los sistemas exper tos y su programación. El material está escrito al nivel de división superior o posgrado, y e adecuado para especialistas en ciencias de la computación, en sistemas de información po computadora, ingeniería y otros campos que estén interesados en los sistemas expertos. L nueva terminología se muestra en negritas y se explica inmediatamente. Numerosos ejemplo y referencias ayudan a hacer más claro el significado del texto y proporcionan una guía par consultar lecturas complementarias.

Sistemas expertos: principios y programación está dividido en dos partes. del capítulo 1 a 6 se expone la teoría que se encuentra detrás de los sistemas expertos y la manera en que ésto encajan en la esfera de las ciencias de la computación; es decir, lógica, probabilidad, estruc turas de datos, inteligencia artificial (AI, artificial intelligence) y otros temas que conforma la teoría de los sistemas expertos. Del capítulo 7 al 12 se expone la programación de lo sistemas expertos. Aunque sería de ayuda un curso previo en inteligencia artificial, este libr proporciona una introducción autónoma a los temas de AI que son apropiados para los siste mas expertos.

Hemos tratado de explicar la teoría que sustenta a los sistemas expertos, de maner que el estudiante pueda tomar una decisión con conocimiento de causa en cuanto al uso apro piado de esta tecnología. Lo importante y en lo que ponemos especial énfasis es en que, com cualquier otra herramienta, los sistemas expertos tienen ventajas y desventajas. La teorí también explica cómo se relacionan los sistemas expertos con otros métodos de programa ción, como la programación convencional. Otra razón para analizar la teoría es preparar a estudiante para leer artículos de las actuales investigaciones en sistemas expertos. Como és tos tienen elementos de campos tan diversos, es difícil para un principiante leer artículos si conocer alguna base teórica.

La segunda parte de este libro es una introducción a la herramienta CLIPS de lo sistemas expertos. Se trata de una introducción práctica a la programación de sistemas exper tos que sirve para reforzar y aclarar los conceptos teóricos desarrollados en la primera parte Al igual que la parte teórica del libro, los estudiantes con alguna experiencia en programación en un lenguaje de alto nivel podrán entender la parte de programación. Los estudiantes apren den los problemas prácticos asociados con el desarrollo del sistema experto utilizando CLIPS una herramienta moderna y llena de opciones desarrollada por la NASA en el centro espacia Johnson para los sistemas expertos. Actualmente CLIPS se usa en proyectos reales del go bierno, los negocios y la industria. Está disponible para casi todos los tipos de computadora incluyendo IBM-PC y clones, VAX, HP, SUN, Macintosh y Cray. El código desarrollado en CLIPS se ejecuta a gran velocidad y puede llevarse a cualquier plataforma.

Algunos cursos en sistemas expertos incluyen un proyecto de final, que es una exce lente manera de desarrollar habilidades para la programación. Los estudiantes suelen completar pequeños sistemas expertos de 50 a 150 reglas en un proyecto semestral de su elección. Unos cuantos proyectos hechos por alumnos que usaron CLIPS incluyen diagnóstico de automóvi les, programas de salud para taxis, agenda personal, administración de redes por computado

Como una de las áreas más importantes y sofisticadas de la AI, la teoría y la práctica de los sistemas expertos contiene muchos problemas complejos y todavía no entendidos del todo. Joseph Giarratano y Gary Riley merecen el agradecimiento de todos nosotros por emprender una tarea tan compleja y hacer una importante contribución para el mejor entendimiento de los fundamentos de los sistemas expertos y su aplicación a problemas reales.

Lotfi A. Zadeh
Berkeley, CA. (enero, 1989)

CAPÍTULO 1

Introducción a los sistemas expertos

1.1 INTRODUCCIÓN

Este capítulo desarrolla una amplia introducción a los sistemas expertos, aquí se presentan sus principios fundamentales, se analizan sus ventajas y desventajas y se describen las áreas de aplicación apropiadas para ellos. También se analiza su relación con otros métodos de programación.

1.2 ¿QUÉ ES UN SISTEMA EXPERTO?

El primer paso para resolver cualquier problema es definir el área o **dominio** del problema que será resuelto; esta consideración es tan cierta para la inteligencia artificial (AI, artificial intelligence) como en la programación convencional. Sin embargo, a causa de la mística antes asociadas con la AI, hay una prolongada tendencia a seguir creyendo en el viejo adagio: "Si todavía no ha sido resuelto es un problema de AI"; otra definición popular es: "la AI está haciendo que las computadoras actúen como en las películas". Tal vez este tipo de cuadro mental haya sido común en los años setenta, cuando la AI se encontraba en plena fase de investigación, pero, hoy en día la AI resuelve muchos problemas reales y tiene muchas aplicaciones comerciales.

Aunque no se han encontrado soluciones generales a los problemas clásicos de la AI, como la traducción del lenguaje natural, la comprensión del habla y la visión, restringir el dominio del problema ayudaría a producir una solución útil. Por ejemplo, no es difícil construir sistemas sencillos de lenguaje natural si la entrada se restringe a oraciones de la forma sustantivo, verbo y predicado. Actualmente este tipo de sistemas funcionan bien para proporcionar una interfaz amigable con el usuario en muchos productos de software, como sistemas de base de datos y hojas de cálculo. De hecho, los analizadores gramaticales relacionados con los populares juegos de aventuras para computadora, basados en texto, exhiben un sorprendente grado de habilidad en la comprensión de lenguaje natural.

Como se muestra en la figura 1.1, la AI tiene muchas áreas de interés. El área de **sistemas expertos** es una aproximación muy exitosa a la solución de los problemas clásicos de AI en la programación de inteligencia. El profesor Edward Feigenbaum de la Universidad de Stanford, pionero en la tecnología de los sistemas expertos, los ha definido como "un programa de computación inteligente que usa el conocimiento y los procedimientos de inferencia para resolver problemas que son lo suficientemente difíciles como para requerir significativa ex-

periencia humana para su solución" (Feigenbaum 82). Es decir, un sistema experto es un sistema de cómputo que **emula** la habilidad de tomar decisiones de un especialista humano. El término *emular* significa que el sistema experto tiene el objetivo de actuar en todos los aspectos como un especialista humano. Una emulación es mucho más fuerte que una simulación, que en algunos aspectos sólo requiere que se actúe como en la realidad.

A pesar de que una solución a problemas de propósito general aún nos elude, los sistemas expertos funcionan bien en sus dominios restringidos. Como prueba de su éxito, sólo se necesita observar las muchas aplicaciones que tienen hoy en los negocios, la medicina, la ciencia y la ingeniería, además de todos los libros, diarios, conferencias y productos dedicados a ellos.

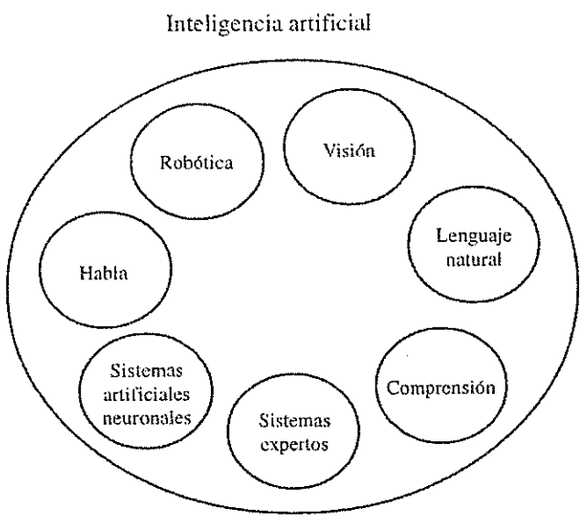


Figura 1.1 Algunas áreas de inteligencia artificial

Los sistemas expertos son una rama de la AI que hace un amplio uso del conocimiento especializado para resolver problemas como un **especialista** humano. Éste es una persona que tiene **experiencia desarrollada** en cierta área. Esto es, el especialista tiene conocimientos o habilidades especiales que la mayoría no conoce o de las que no dispone; puede resolver problemas que la mayoría no podría resolver, o los resuelve con mucha mayor eficiencia (y a otro costo). Cuando los sistemas expertos se desarrollaron por primera vez en los años setenta, contenían exclusivamente conocimiento experto; sin embargo, hoy en día a menudo se aplica el término *sistema experto* a cualquier sistema que utiliza tecnología de sistemas expertos. Esta tecnología puede incluir a los lenguajes y programas especiales de sistemas expertos, además del hardware diseñado para ayudar en su desarrollo y ejecución.

El conocimiento de los sistemas expertos puede obtenerse por experiencia o consulta de los conocimientos que suelen estar disponibles en libros, revistas y con personas capacitadas. Los términos *sistema experto*, **sistema basado en conocimiento,** o **sistema experto basado en conocimiento,** se usan como sinónimos. La mayoría utiliza el término sistema experto tan sólo porque es más corto, a pesar de que quizá no tengan experiencia en su sistema, sólo conocimiento general.

En la figura 1.2 se ilustra el concepto básico de un sistema experto basado en conocimiento. El usuario aporta los hechos u información al sistema experto y recibe consejo o experiencia como respuesta. En su interior, el sistema experto incluye dos componentes principales. La base de conocimiento contiene el conocimiento que le permite al **mecanismo de inferencia** sacar conclusiones; éstas son las respuestas del sistema experto a la consulta especializada del usuario.

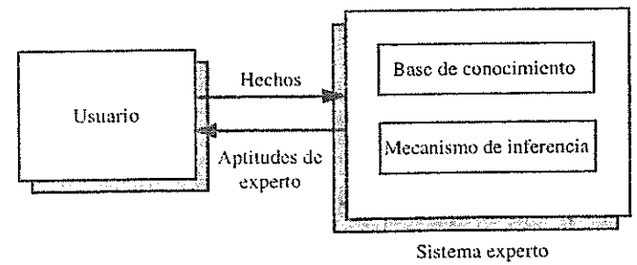


Figura 1.2 Conceptos básicos del funcionamiento de un sistema experto

También se han diseñado excelentes sistemas basados en conocimientos que funcionan como el asistente inteligente de un especialista humano. Estos asistentes inteligentes están diseñados con tecnología de sistemas expertos debido a sus ventajas de desarrollo. Cuanto más conocimiento se añada a un asistente inteligente, actuará más como un especialista. Por tanto, desarrollar un asistente inteligente puede constituir un útil acontecimiento en la producción de un sistema experto completo. Además, es posible que al acelerar la solución de problemas proporcione más tiempo disponible a los especialistas. Los tutores inteligentes son otra nueva aplicación de la inteligencia artificial; a diferencia de los viejos sistemas de instrucción asistida por computadora, el nuevo sistema puede proporcionar instrucciones sensibles al contexto (Giarratano 91a).

En oposición a las técnicas de solución general de problemas el conocimiento de un especialista se centra específicamente en **dominio de problema**. Un dominio de problema es el área específica de problema, como medicina, finanzas, ciencias, ingeniería, etc., en el que un especialista puede resolver problemas con facilidad. Los sistemas expertos, como los especialistas humanos, suelen diseñarse especializados en un dominio de problemas. Por ejemplo, normalmente no se esperaría que un especialista en ajedrez tuviera conocimientos especializados en medicina. Las experiencias en un dominio de problema no llevan automáticamente a otro.

Al conocimiento del especialista para resolver problemas específicos se le llama **dominio de conocimiento** del experto. Por ejemplo, un sistema experto médico, diseñado para diagnosticar enfermedades infecciosas debe tener una gran cantidad de conocimiento acerca de los síntomas causados por este tipo de enfermedades. En este caso, el dominio del conocimiento es la medicina y consta del conocimiento de las enfermedades, sus síntomas y tratamientos. En la figura 1.3 se ilustra la relación entre el problema y el dominio de conocimiento. Observe que éste está completamente incluido dentro del dominio del problema. La porción exterior al dominio del conocimiento simboliza un área en que no existe conocimiento acerca de todos los problemas.

Un sistema experto médico normalmente no tiene conocimientos acerca de otras ramas de la medicina, como cirugía o pediatría. A pesar de que su conocimiento de las enfermedades

infecciosas es equivalente al de un especialista humano, el sistema experto no sabría nada de otros dominios de conocimiento a menos que estuviera programado con ellos.

Figura 1.3 Un probable problema y su relación con el dominio del conocimiento

En su dominio de conocimiento, el sistema experto razona o hace **inferencias** de la misma forma en que un especialista humano inferiría la solución de un problema. Esto es, dados algunos hechos, se infiere una conclusión. Por ejemplo, si su cónyuge no le ha hablado durante un mes, usted puede deducir que no ha tenido nada importante que decir, sin embargo, ésta es sólo una de las muchas inferencias posibles.

Como con cualquier nueva tecnología, existe todavía mucho que aprender acerca de los sistemas expertos. En la tabla 1.1 se resumen las diferentes perspectivas de quienes participan en una tecnología. En esta tabla, el tecnólogo puede ser un ingeniero o un diseñador de software y la tecnología puede ser hardware o software. Al resolver cualquier problema, éstas son preguntas que deben responderse o la tecnología no podrá usarse con éxito. Como cualquier otra herramienta, los sistemas expertos tienen aplicaciones apropiadas e inapropiadas. A medida que crezca nuestra experiencia al respecto, descubriremos cuáles son estas aplicaciones.

Persona	Pregunta
Gerente	¿Para qué puedo usarla?
Tecnólogo	¿Cómo puedo implantarla mejor?
Investigador	¿Cómo puedo ampliarla?
Consumidor	¿Cómo me ayudará?
	¿Merece el trabajo y el costo?
	¿Qué tan confiable es?

Tabla 1.1 Diferentes visiones de la tecnología

1.3 VENTAJAS DE LOS SISTEMAS EXPERTOS

Los sistemas expertos tiene varias características atractivas:

- *Mayor disponibilidad.* La experiencia está disponible para cualquier hardware de cómputo adecuado. En un sentido muy real, un sistema experto es la producción masiva de experiencia.

- *Costo reducido.* El costo de poner la experiencia a disposición del usuario se reduce enormemente.
- *Peligro reducido.* Los sistemas expertos pueden usarse en ambientes que podrían ser peligrosos para un ser humano.
- *Permanencia.* La experiencia es permanente. A diferencia de los especialistas humanos, que pueden retirarse, renunciar o morir, el conocimiento del sistema experto durará indefinidamente.
- *Experiencia múltiple.* El conocimiento de varios especialistas puede estar disponible para trabajar simultánea y continuamente en un problema, a cualquier hora del día o de la noche. El nivel de experiencia combinada de muchos sistemas expertos puede exceder el de un solo especialista humano (Harmon 85).
- *Mayor confiabilidad.* Al proporcionar una segunda opinión, los sistemas expertos incrementan la confianza en que un especialista ha tomado la decisión correcta o al dar un voto de calidad en caso de desacuerdos entre varios especialistas. Por supuesto, este método probablemente no funcionará si uno de ellos fue quien programó al sistema. Ambos deben coincidir siempre, a menos que el especialista haya cometido un error, lo que puede suceder si estaba cansado o bajo presión.
- *Explicación.* El sistema experto puede explicar clara y detalladamente el razonamiento que conduce a una conclusión, lo que aumenta la confianza en que se tomó la decisión correcta. Un ser humano puede estar demasiado cansado, mostrarse renuente o ser incapaz de hacerlo siempre.
- *Respuesta rápida.* Tal vez sea necesaria una respuesta rápida, o en tiempo real, para ciertas aplicaciones. Dependiendo del software y hardware usado, un sistema experto puede responder más rápido y estar más dispuesto que un especialista. Algunas situaciones de emergencia pueden exigir respuestas más rápidas que las de un humano, de modo que un sistema experto en tiempo real resulta una buena elección (Hugh 88; Ennis 86).
- *Respuestas sólidas, completas y sin emociones, en todo momento.* Esto puede ser muy importante en tiempo real y en situaciones de emergencia, cuando un especialista quizá no funcionaría a toda su capacidad a causa de la presión y la fatiga.
- *Tutoría inteligente.* El sistema experto puede actuar como un tutor inteligente, dejando que el estudiante ejecute programas de ejemplo y explicando el razonamiento del sistema.
- *Base de datos inteligente.* Los sistemas expertos pueden usarse para tener acceso a una base de datos en forma inteligente (Kerschberg 86; Schur 88).

El proceso de desarrollo de un sistema experto también tiene un beneficio indirecto, dado que el conocimiento de los especialistas humanos debe disponerse clara y formalmente para introducirlo en la computadora. Como se dispone explícitamente del conocimiento, en vez de tenerlo implícito en la mente del especialista, puede examinarse para corregirlo, darle más consistencia y completarlo. El conocimiento puede entonces ajustarse o reexaminarse, lo que aumenta su calidad.

1.4 CONCEPTOS GENERALES DE LOS SISTEMAS EXPERTOS

El conocimiento de un sistema experto puede representarse de varias maneras (puede estar encapsulado en reglas y objetos). Un método común de representar el conocimiento es en forma de reglas tipo SI...ENTONCES, como:

```
SI la luz es roja ENTONCES deténgase
```

Si existe el hecho de que la luz sea roja, esto concuerda con el patrón "la luz es roja", la regla se satisfizo y se ejecuta la acción "deténgase". Aunque se trata de un ejemplo muy simple, se han construido muchos sistemas expertos significativos expresando en **reglas** el conocimiento de los especialistas. En realidad, el método basado en el conocimiento para desarrollar sistemas expertos ha reemplazado completamente la antigua propuesta de AI de los años cincuenta y sesenta, que trataba de utilizar sofisticadas técnicas de razonamiento que no dependían del conocimiento. Algunas herramientas de los sistemas expertos, como CLIPS, permiten **objetos** además de reglas. Las reglas pueden corresponder a los objetos y también a los hechos mientras que, los objetos pueden operar independientemente de las reglas.

EJEMPLOS

Hoy en día se ha construido una gran variedad de sistemas expertos basados en el conocimiento. Grandes sistemas, que contienen miles de reglas, tales como el sistema XCON/R1 de Digital Equipment Corporation, saben mucho más que cualquier especialista acerca de cómo configurar sistemas de computadora (McDermontt 84). También se han construido muchos sistemas pequeños para tareas especializadas, con varios cientos de reglas. Quizá estos pequeños sistemas no operan al nivel de un especialista, pero están diseñados para aprovechar la tecnología de los sistemas expertos y ejecutar tareas intensivas de conocimiento. El conocimiento para estos pequeños sistemas puede estar en libros, revistas u otros documentos del dominio público.

CÓMO OBTIENE SU CONOCIMIENTO

Por el contrario un sistema experto clásico abarca conocimiento no escrito que debe obtenerse de un especialista a través de extensas entrevistas con un **ingeniero del conocimiento**, durante un largo periodo. Al proceso de construir un sistema experto se le llama ***ingeniería del conocimiento*** (Michie 73), y consiste en la adquisición de conocimiento a partir de un especialista humano o de otra fuente y su codificación en el sistema experto.

Las etapas generales en el desarrollo de un sistema experto se ilustran en la figura 1.4. El ingeniero del conocimiento establece primero un diálogo con el especialista, con el fin de obtener su conocimiento. Esta etapa es análoga a la de un diseñador de sistemas en programación convencionales, cuando analiza los requisitos del sistema con el cliente para el que está construyendo el programa. El ingeniero en conocimiento codifica explícitamente el conocimiento en la base de conocimiento. El especialista evalúa entonces al sistema experto y ofrece una crítica al ingeniero del conocimiento. Este proceso se repite hasta que el experto juzga la actuación del sistema como satisfactoria.

La expresión ***sistema basado en el conocimiento*** es un mejor término para la aplicación de tecnología basada en el conocimiento, que puede usarse para la creación de sistemas expertos o basados en el conocimiento. Sin embargo, tal y como ocurre con el término inteligencia artificial, es común usar el término sistemas expertos cuando se refieren a ambos tipos de sistemas, aunque el conocimiento no esté al nivel de un especialista humano.

Los sistemas expertos suelen estar diseñados de manera distinta a los programas convencionales porque los problemas no tienen generalmente una solución algorítmica y dependen de inferencias para obtener una solución razonable, considerando ésta como la mejor que se puede esperar si no hay un algoritmo disponible que ayude a obtener la solución óptima. Como el sistema experto depende de la inferencia, debe ser capaz de explicar su razonamiento para que éste pueda verificarse. La **facilidad de explicación** es parte integral de los sistemas expertos sofisticados. De hecho deben diseñarse elementos de explicación elaborada que permitan al usuario la exploración de múltiples líneas de conjeturas "¿Qué pasaría si...?", o preguntas de **razonamiento hipotético**, e incluso la traducción de lenguaje natural a reglas.

Algunos sistemas expertos permiten incluso que el sistema aprenda reglas mediante el ejemplo, a través de la **inducción de reglas**, en la que el sistema crea reglas a partir de tablas de datos. No es fácil dar forma de reglas al conocimiento de los especialistas sobre todo cuando nunca se le ha explorado sistemáticamente. Puede haber inconsistencias, ambigüedades, duplicaciones u otros problemas que no se notan hasta que se trata de representar formalmente el conocimiento del experto en un sistema experto.

TAREA:

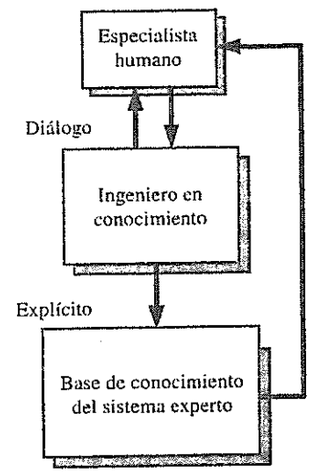


Figura 1.4 Desarrollo de un sistema experto

Los especialistas humanos también conocen el alcance de sus conocimientos y evalúan su consejo cuando el problema alcanza sus **límites de ignorancia**. Un especialista humano también sabe cuándo "romper las reglas". A menos que los sistemas expertos se diseñen explícitamente para lidiar con la incertidumbre, harán recomendaciones con la misma confianza, aunque los datos con los que están tratando sean imprecisos o estén incompletos. El consejo de un sistema experto, como el de un especialista humano, se degrada suavemente en los límites de la ignorancia.

Hoy en día, una limitante práctica de muchos sistemas expertos es la carencia de **conocimiento causal**. Esto es, el sistema experto no comprende realmente las causas y los efectos fundamentales de un sistema. Es mucho más fácil programar sistemas expertos con **conocimiento superficial**, basado en el conocimiento empírico y heurístico, que con **conocimiento profundo**, basado en las estructuras básicas, funciones y comportamientos de los objetos. Por ejemplo, es mucho más fácil programar un sistema experto para que prescriba una aspirina para el dolor de cabeza de una persona, que programar todo el conocimiento bioquímico, fisiológico, anatómico y neurológico fundamentales acerca del cuerpo humano. La programación de un modelo causal del cuerpo humano sería una tarea enorme y, aunque tuviera éxito, probablemente la velocidad de respuesta sería demasiado lenta a causa de toda la información que tendría que procesar el sistema.

Una clase de conocimiento superficial es el **conocimiento heurístico** (*heurístico* viene del griego y significa "descubrir"). La heurística no es una garantía de éxito, como un algoritmo es una solución garantizada para un problema. En cambio, la heurística es un método práctico

o conocimiento empírico obtenido de la experiencia y que puede ayudar en la solución, pero que no garantiza su funcionamiento siempre. Sin embargo, en muchos campos como la medicina y la ingeniería, la heurística juega un papel esencial en algunos tipos de solución de problemas. Aunque se conozca una solución exacta, puede resultar poco práctica a causa de los costos o de las restricciones de tiempo. La heurística puede proporcionar atajos valiosos que reduzcan tiempo y dinero.

Otro problema con los sistemas expertos de hoy, es que su experiencia está limitada al dominio del conocimiento que maneja el sistema. Los sistemas expertos típicos no pueden generalizar su conocimiento usando la **analogía** para razonar acerca de nuevas situaciones, como lo hace la gente. A pesar de que la inducción de reglas ayuda, sólo algunos tipos limitados de conocimiento pueden agregarse al sistema de esta manera. La forma acostumbrada de construir un sistema experto, haciendo que el ingeniero en conocimiento repita el ciclo de entrevistar al especialista, construir un prototipo, probar, entrevistar, etcétera, es una tarea que requiere de mucho tiempo y trabajo. En realidad, este problema de transferir el conocimiento humano a un sistema experto es tan grande que se le llama el **cuello de botella de la adquisición del conocimiento.** Se trata de un término descriptivo porque el cuello de botella de la adquisición del conocimiento restringe la construcción de un sistema experto del mismo modo en que un cuello de botella restringe el flujo de líquido.

A pesar de sus limitaciones actuales, los sistemas expertos han tenido éxito al tratar con problemas del mundo real que los métodos de programación convencional habían sido incapaces de resolver, sobre todo aquellos que trataban con incertidumbre o con información incompleta. Lo más importante es estar informado de las ventajas y limitantes de esta nueva tecnología, para utilizarla apropiadamente.

1.5 CARACTERÍSTICAS DE UN SISTEMA EXPERTO

Un sistema experto suele diseñarse para que tenga las siguientes características generales:

- *Alto desempeño.* El sistema debe tener la capacidad de responder a un nivel de competencia igual o superior al de un especialista en el campo. Esto significa que la calidad del consejo dado por el sistema debe ser muy alta.
- *Tiempo de respuesta adecuado.* El sistema debe actuar en un tiempo razonable, comparable o mejor al tiempo requerido por un especialista, para alcanzar una decisión. Si un sistema experto necesita un año para tomar una decisión que un especialista tomaría en una hora, no sería muy útil. Las **restricciones de tiempo** en el desempeño de un sistema experto pueden ser especialmente severas en el caso de los sistemas en tiempo real, cuando una respuesta debe darse dentro de un intervalo.
- *Confiabilidad.* El sistema experto debe ser confiable y no propenso a "caídas", o no será usado.
- *Comprensible.* El sistema debe ser capaz de explicar los pasos de su razonamiento mientras se ejecutan, de tal modo que sea comprensible. En lugar de ser sólo una "caja negra" que produce una respuesta milagrosa, el sistema debe tener capacidad de explicación, de la misma forma en que los especialistas pueden explicar su razonamiento. Este rasgo es muy importante por varias razones.

Una razón es que la vida humana puede depender de las respuestas del sistema experto. A causa de su gran potencial para perjudicar, todo sistema experto debe ser capaz de justificar sus conclusiones tal como un especialista humano puede explicar por qué se llegó a cierta

conclusión. Por tanto, una característica de explicación proporciona una revisión del razonamiento comprensible para los seres humanos.

Una segunda razón para tener un medio de explicación en un sistema experto se da durante la fase de desarrollo, para confirmar que el conocimiento ha sido adquirido y está siendo utilizado correctamente por el sistema. Esto es importante al depurar porque es posible que el conocimiento se introduzca con errores o que haya malentendidos entre el ingeniero del conocimiento y el especialista; un buen medio de explicación les permite verificar que el conocimiento sea correcto. Además, a causa de la forma en que están construidos los sistemas expertos comunes, es muy difícil leer un listado de un programa y entender su funcionamiento.

Otra fuente de errores pueden ser las interacciones imprevistas en el sistema experto, las que pueden detectarse ejecutando casos de prueba con razonamientos conocidos que el sistema debe seguir. Como se analizará posteriormente con más detalle, pueden aplicarse varias reglas a una situación dada sobre la cual está razonando el sistema. En un sistema experto el curso de la ejecución no es secuencial, de manera que no es posible simplemente leer su código línea tras línea y entender cómo opera. Es decir, el orden en que se introducen las reglas al sistema no es necesariamente el mismo en que se ejecutarán. El sistema experto actúa como un programa paralelo en que las reglas son procesadores de conocimiento independientes.

- *Flexibilidad.* Debido a la gran cantidad de conocimiento que un sistema experto puede tener, es importante contar con un mecanismo eficiente para añadir, modificar y eliminar conocimiento. Una razón de la popularidad de los sistemas basados en reglas es la capacidad de almacenaje eficiente y modular de las reglas.

Dependiendo del sistema, un mecanismo de explicación puede ser simple o elaborado. Uno simple, en un sistema basado en reglas, puede presentar una lista de todos los hechos que hicieron que la última regla se ejecutara. Los sistemas más elaborados pueden hacer lo siguiente:

- *Enumerar todas las razones a favor y en contra de una hipótesis en particular.* Una hipótesis es una proposición que debe probarse, tal como "el paciente tiene una infección por tétanos" en un sistema experto de diagnóstico médico. En un problema real puede haber varias hipótesis, de la misma manera en que el paciente puede tener varias enfermedades al mismo tiempo. Una hipótesis también puede verse como un hecho cuya verdad está en duda y debe probarse.
- *Enumerar todas las hipótesis que puedan explicar la evidencia observada.*
- *Explicar todas las consecuencias de una hipótesis.* Por ejemplo, suponiendo que el paciente tiene tétanos, debe haber también evidencia de fiebre a medida que la infección sigue su curso. Si este síntoma se observa, añade credibilidad a la hipótesis; si no se observa, reduce la credibilidad de la hipótesis.
- *Dar un **pronóstico** o predicción de lo que ocurrirá si la hipótesis es verdadera.*
- *Justificar las preguntas que el programa hace al usuario para obtener más información.* Estas preguntas pueden usarse para dirigir la línea de razonamiento hacia las rutas de diagnóstico probables. En la mayor parte de los problemas reales, es muy caro o toma mucho tiempo explorar todas la posibilidades y debe proporcionarse una guía para buscar la solución correcta. Por ejemplo, considerar el costo, el tiempo y el efecto de administrar todas las posibles pruebas médicas a un paciente que padece dolor de garganta.
- *Justificar el conocimiento del programa.* Por ejemplo, si el programa asegura que la hipótesis "el paciente tiene una infección por tétanos" es verdadera, el usua-

rio puede pedir una explicación. El programa puede justificar esta conclusión basándose en una regla que dice que si el paciente tiene una prueba de sangre positiva para tétanos, entonces el paciente tiene tétanos. Ahora, el usuario puede pedir al programa que justifique esta regla. El programa puede responder estableciendo que un examen de sangre positivo para una enfermedad es prueba de la enfermedad.

En este caso el programa está citando una **metarregla**, un conocimiento acerca de reglas (el prefijo *meta* significa "por encima" o "más allá"). Algunos programas, como Meta-DENDRAL, han sido creados específicamente para inferir nuevas reglas (Buchanan 78). El conocimiento justifica una hipótesis y éste, a su vez, se justifica con **una garantía** de que es correcto. Una garantía es, en esencia, una meta-explicación que esclarece la explicación dada por el sistema experto acerca de su razonamiento.

El conocimiento puede crecer fácilmente de manera acrecentada en un sistema basado en reglas. Esto quiere decir que la base de conocimiento puede crecer poco a poco a medida que se agregan reglas, de modo que pueden revisarse continuamente la ejecución y corrección del sistema. Si las reglas están diseñadas adecuadamente, las interacciones entre ellas se minimizarán o eliminarán como protección contra efectos inesperados. El crecimiento de conocimiento facilita la **construcción rápida de prototipos**, de modo que el ingeniero del conocimiento pueda mostrar pronto al especialista un prototipo de trabajo del sistema experto. Ésta es una característica importante porque mantiene el interés del especialista y de la administración en el proyecto. La construcción rápida de prototipos también muestra rápidamente las lagunas, inconsistencias o errores en el conocimiento del especialista o del sistema, de manera que puedan corregirse inmediatamente.

EQUIPOS 3

Pedir documento donde expliquen 6 líneas de razonamiento

1.6 EL DESARROLLO DE LA TECNOLOGÍA DE SISTEMAS EXPERTOS

La AI tiene muchas ramas relacionadas con el habla, la visión, la robótica, la comprensión y aprendizaje del lenguaje natural y los sistemas expertos. Las raíces de los sistemas expertos abarcan muchas disciplinas; en particular, una de las raíces principales de los sistemas expertos es el área del procesamiento de la información humana, llamada **ciencia cognitiva**. La *cognición* es el estudio de la manera en que los humanos procesan la información. En otras palabras la cognición es el estudio de la manera en que piensa la gente, especialmente cuando resuelve problemas.

El estudio de la cognición es muy importante si se pretende lograr que las computadoras emulen a los especialistas humanos. A menudo, ellos no pueden explicar cómo resuelven problemas (simplemente las soluciones les llegan). Si el especialista no puede explicar cómo se soluciona un problema, no es posible codificar el conocimiento en un sistema experto basado en el conocimiento explícito. En este caso, la única posibilidad son los programas que aprenden por sí mismos a emular al especialista. Estos son programas basados en la inducción y en sistemas neuronales artificiales, que se analizarán más adelante.

Solución humana de problemas y producciones

El desarrollo de la tecnología de sistemas expertos tiene amplios antecedentes. La tabla 1.2 es un breve resumen de algunos desarrollos importantes que han convergido en los modernos sistemas expertos. Cuando es posible, se dan las fechas de inicio de los proyectos, aun cuando muchos de ellos abarcaron varios años. Estos desarrollos se tratan con más detalle en este y

otros capítulos. La mejor referencia para todos los primeros sistemas la componen los tres volúmenes del *Manual de Inteligencia Artificial* (Feigenbaum 81).

Año	Evento
1943	Reglas de postproducción; Modelo neuronal de McCulloch y Pitts
1954	Algoritmo de Markov para controlar la ejecución de reglas
1956	Conferencia de Darmouth; Teórico lógico; Búsqueda Heurística; acuñación del término "AI"
1957	Perceptrón inventado por Rosenblatt; Inicia el GPS (General Problem Solver; Solucionador general de problemas) (Newell, Shaw y Simon)
1958	Lenguaje LISP de AI (McCarthy)
1962	Principios de neurodinámica (*Principles of Neurodynamics*) en la percepción, de Rosenblatt
1965	Método de solución para la comprobación automática de teoremas (Robinson)
	Lógica confusa para el razonamiento acerca de objetos confusos (Zadeh)
	Inicia el trabajo en DENDRAL, el primer sistema experto (Feigenbaum, Buchanan, *et al.*)
1968	Redes semánticas, modelo de memoria asociativa (Quillian)
1969	MACSYMA, sistema experto en matemáticas (Martin y Moses)
1970	Inicia el trabajo en PROLOG (Colmerauer, Rousell, *et al.*)
1971	HEARSAY I para reconocimiento del habla
	Human Problem Solving populariza las reglas (Newell y Simon)
1973	MYCIN, sistema experto para diagnóstico médico (Shortliffe, *et. al.*) que conduce a
	GUIDON, tutoría inteligente (Clancey)
	TEIRESIAS, concepto de mecanismo de explicación (Davis), y
	EMYCIN, primer shell (Van Melle, Shortliffe y Buchanan)
	HEARSAY II, modelo de pizarrón de cooperación múltiple entre especialistas.
1975	Marcos, representación del conocimiento (Minsky)
1976	AM (Artificial Mathematician, matemático artificial), descubrimiento creativo de conceptos matemáticos (Lenat)
	Teoría de la evidencia de Dempster-Shafer para el razonamiento bajo incertidumbre
	Inicia el trabajo en el sistema experto PROSPECTOR para la exploración mineral (Duda, Hart, *et. al.*)
1977	OPS shell (o armazón) de sistema experto (Forgy), usado en XCON/R1
1978	Inicia el trabajo en XCON/R1 (McDermott, DEC) para configurar los sistemas de computadora DEC
	Meta-DENDRAL, metarreglas e inducción de reglas (Buchanan)
1979	Algoritmo Rete para correspondencia rápida de patrones (Forgy).
	Inicia la comercialización de la AI
	Se forma Inference Corp. (libera ART, herramienta del sistema experto en 1985)
1980	Se funda Symbolics, LMI para fabricar máquinas LISP
1982	Sistemas expertos en matemáticas SMP; Hopfield Neural Net;
	Proyecto japonés de quinta generación para desarrollar computadoras inteligentes
1983	KEE, herramienta del sistema experto (IntelliCorp)
1985	CLIPS herramienta del sistema experto (NASA)

Tabla 1.2 Algunos eventos importantes en la historia de los sistemas expertos

A finales de los años cincuenta y principios de los sesenta, se escribieron varios programas orientados a la solución de problemas en general. El más famoso de éstos fue el **Solucionador general de problemas**, creado por Newell y Simon y descrito en una serie de artículos que culminaron en el monumental trabajo de 920 páginas sobre la cognición, *Human Problem Solving* (Newell 72).

Uno de los resultados más significativos demostrados por Newell y Simon fue que gran parte de la solución humana de problemas o cognición puede expresarse con reglas de pro-

ducción del tipo SI...ENTONCES. Por ejemplo SI parece que va a llover ENTONCES lleve un paraguas, o SI su cónyuge está de mal humor ENTONCES no se muestre contento. Una regla corresponde a una pequeña colección modular de conocimiento llamada **fragmento.** Los fragmentos se organizan en una disposición suelta, con nexos hacia fragmentos de conocimiento relacionados. Una teoría sostiene que toda la memoria humana está organizada en fragmentos. Un ejemplo de una regla que representa un fragmento de conocimiento es:

```
SI el automóvil no avanza y
   el indicador de combustible está en vacío
ENTONCES cargue gasolina
```

Newell y Simon popularizaron el uso de las reglas para representar al conocimiento humano y mostraron cómo puede razonarse con ellas. Los psicólogos cognitivos han usado reglas como modelos para explicar el procesamiento humano de la información. La idea básica es que la entrada sensorial proporciona estímulos al cerebro, el estímulo dispara las reglas apropiadas de la **memoria a largo plazo**, y ésta produce la respuesta oportuna. La memoria a largo plazo es dónde nuestra memoria está almacenada. Por ejemplo, todos tenemos reglas como:

```
SI hay una flama ENTONCES hay fuego
SI hay humo ENTONCES puede haber fuego
SI hay una sirena ENTONCES puede haber fuego
```

Observe que las últimas dos reglas no se expresan con total certeza, el fuego puede haberse apagado y aún haber humo en el aire. Del mismo modo, una sirena no prueba que haya fuego porque puede estar respondiendo a una falsa alarma. Los estímulos de ver las flamas, oler humo y escuchar una sirena dispararán éstos y otros tipos de reglas similares.

La memoria a largo plazo incluye muchas reglas que tienen la estructura simple SI...ENTONCES. De hecho, un gran maestro especialista en ajedrez puede tener 50,000 o más fragmentos de conocimiento acerca de los patrones del ajedrez. En contraste con la memoria a largo plazo, la **memoria a corto plazo** se utiliza para el almacenamiento temporal de conocimiento durante la solución del problema. A pesar de que la memoria a largo plazo puede guardar cientos de miles o más fragmentos, la capacidad de la memoria activa es sorprendentemente pequeña, de cuatro a siete fragmentos. Como un ejemplo sencillo de esto, trate de visualizar algunos números en su mente; la mayoría de las personas sólo pueden ver de cuatro a siete números al mismo tiempo, y también pueden memorizar mucho más de cuatro a siete números, sin embargo estos números se mantienen en la memoria a largo plazo.

Una teoría propone que la memoria a corto plazo representa el número de fragmentos que pueden estar activos de manera simultánea, y considera la solución humana de problemas como la diseminación de los fragmentos activados en la mente. Eventualmente, un fragmento puede activarse con tal intensidad que genera un pensamiento consciente y alguien se dice a sí mismo: "Hmm... algo se está quemando."

El otro elemento necesario para la solución humana de problemas es un **procesador cognitivo**, éste trata de encontrar las reglas que se activarán con el estímulo apropiado, pero no lo hará cualquier regla. Por ejemplo, usted no llenaría su tanque de gasolina cada vez que oye una sirena; solamente se activará una regla que corresponda al estímulo. Si se activan al mismo tiempo varias reglas, el procesador cognitivo debe ejecutar una solución de conflicto para decidir cuál tiene la prioridad más alta. Por ejemplo, esa regla se ejecutará si se activan estas dos:

```
SI hay fuego ENTONCES debo alejarme
SI mis ropas se están quemando ENTONCES debo apagarlas
```


Entonces, las acciones de la regla con mayor prioridad, se ejecutarán antes que las otras. El **mecanismo de inferencia** de los sistemas expertos modernos corresponde al procesador cognitivo.

El modelo de Newell y Simon para la solución humana de problemas desde la perspectiva de la memoria a largo plazo (reglas), la memoria a corto plazo (memoria activa) y un procesador cognitivo (mecanismo de inferencia) es la base de los modernos sistemas expertos basados en reglas.

Reglas como éstas son una especie de **sistema de producción**. Hoy en día, los sistemas de producción basados en reglas son un método popular para implantar sistemas expertos. Las reglas individuales que comprenden un sistema de producción son las **reglas de producción**. Al diseñar un sistema experto, un factor importante es la cantidad de conocimiento o **granularidad** de las reglas. Muy poca granularidad dificulta la comprensión de una regla sin relación con otras. Demasiada granularidad hace que el sistema experto resulte muy difícil de modificar a causa de la gran cantidad de fragmentos entremezclados en una regla.

Hasta mediados de los sesenta, una de las principales búsquedas de la AI era producir sistemas inteligentes que dependieran poco del dominio de conocimiento y más de métodos de razonamiento poderosos. Incluso el nombre del Solucionador General de Problemas ilustra la concentración en máquinas que no estaban diseñadas para un dominio específico, sino que tenían el propósito de resolver muchas clases de problemas. A pesar de que los métodos de razonamiento usados por los solucionadores de problemas eran poderosos, las máquinas resultaban eternos principiantes. Cuando se presentaba un nuevo dominio, tenían que descubrir todo a partir de sus principios básicos y no eran tan buenos como los especialistas humanos, que dependían del dominio de conocimiento para acciones de alto nivel.

Un ejemplo del poder de conocimiento es el juego de ajedrez. A pesar de que las computadoras compiten ahora con los humanos, la gente juega bien, sin importar el hecho de que la computadora puede hacer cálculos millones de veces más rápido. Los estudios han demostrado (Chase 73) que los ajedrecistas expertos no tienen superpoderes de razonamiento, sino que dependen de su conocimiento de los patrones de las piezas de ajedrez adquirido con los años de juego. Como se mencionó anteriormente, se estima el conocimiento de un jugador experto de ajedrez en 50,000 patrones y los seres humanos son muy buenos para reconocer patrones como los de las piezas en un tablero de ajedrez. En lugar de tratar de razonar diez o veinte movimientos posibles para cada pieza por anticipado, como lo hace una computadora, el hombre analiza el juego a partir de patrones que revelan las amenazas a largo plazo mientras se mantiene alerta contra movimientos sorpresivos a corto plazo.

A pesar de que el conocimiento del dominio es poderoso, generalmente se limita al dominio. Por ejemplo, una persona que se convierte en ajedrecista experto, no se vuelve al mismo tiempo un experto para resolver problemas matemáticos, ni un experto en damas. Aunque parte del conocimiento se puede llevar a otro dominio, como la planeación cuidadosa de movimientos, ésta es una habilidad más que una pericia genuina.

A principios de los setenta se había hecho evidente que el conocimiento del dominio era la clave para construir máquinas que resolvieran problemas y que pudieran funcionar al nivel de los especialistas humanos. Aunque los métodos de razonamiento son importantes, los estudios han mostrado que los especialistas no dependen primordialmente del razonamiento para resolver un problema; en realidad, el razonamiento puede jugar un papel menor en la solución. En vez de esto, los expertos dependen de un vasto conocimiento de la heurística y de la experiencia adquirida con los años. Si un especialista no puede resolver un problema basándose en su pericia, entonces debe razonar partiendo de los principios básicos y la teoría (o, con mayor probabilidad, preguntar a otro perito). Generalmente la habilidad de razonamiento de un especialista no es mejor que la de la persona promedio al tratar con una situación que no le es familiar. Los primeros intentos por construir poderosos solucionadores de

problemas basados únicamente en el razonamiento, han demostrado que éstos están incompletos si dependen sólo del razonamiento.

El discernimiento de que el dominio del conocimiento era la clave para construir solucionadores de problemas reales, condujo al éxito de los sistemas expertos. Por tanto, los sistemas expertos que han tenido éxito hoy en día son sistemas basados en conocimiento, más que solucionadores de problemas generales. Además, la misma tecnología que condujo al desarrollo de los sistemas expertos, ha llevado también al desarrollo de sistemas basados en conocimiento que no necesariamente contienen experiencia humana.

Aunque a la experiencia se le considera conocimiento especializado y conocido sólo por unos pocos, el conocimiento suele encontrarse en libros, periódicos y otras recursos ampliamente disponibles. Por ejemplo, el conocimiento de cómo resolver una ecuación cuadrática o integrales y derivadas está ampliamente disponible. Se cuenta con programas de computadora basados en el conocimiento, como MACSYMA y SMP, para ejecutar de manera automática éstas y muchas otras operaciones matemáticas con operandos numéricos o simbólicos. Otros programas basados en conocimiento pueden ejecutar el proceso de control de plantas de manufactura. En la actualidad, los términos **programación basada en conocimiento** y *sistemas expertos* se utilizan como sinónimos. En realidad, los sistemas expertos se consideran ahora como un modelo de programación alternativo o **paradigma** para la programación algorítmica convencional.

El ascenso de los sistemas basados en conocimiento

Con la aceptación del paradigma basado en conocimiento durante los setenta, se crearon con éxito varios prototipos de sistemas expertos. Estos sistemas podían interpretar espectrogramas de masas para identificar los componentes químicos (DENDRAL), diagnosticar enfermedades (MYCIN), analizar datos geológicos para la búsqueda de petróleo (DIPMETER) y minerales (PROSPECTOR), y configurar sistemas de cómputo (XCON/R1). La noticia de que PROSPECTOR descubrió un depósito de minerales con valor superior a los cien millones de dólares y de que XCON/R1 ahorraba a la Digital Equipment Corporation (DEC) millones de dólares al año, desató un sensacional interés en la nueva tecnología de los sistemas expertos en 1980. La rama de la AI que inició en los cincuenta como un estudio del procesamiento humano de la información, había crecido hasta alcanzar el éxito comercial por el desarrollo de programas prácticos para usarse en la vida real.

El sistema experto MYCIN fue importante por muchas razones. En primer lugar, demostró que la AI podía usarse en problemas prácticos reales. En segundo lugar, MYCIN fue el campo de pruebas para nuevos conceptos como el mecanismo de explicación, la adquisición automática de conocimientos y la tutoría inteligente, que se encuentran en muchos de los sistemas expertos actuales. La tercera razón por la que MYCIN fue importante es que demostró la viabilidad de la **shell** del sistema experto.

Los sistemas expertos previos, como el DENDRAL, eran sistemas únicos en los que la base de conocimiento estaba entremezclada con el software que aplicaba el conocimiento: el mecanismo de inferencia. MYCIN separó explícitamente la base de conocimiento del mecanismo de inferencia. Esto fue demasiado importante para el desarrollo de la tecnología de sistemas expertos porque significó que podía reutilizarse el núcleo esencial del sistema experto. O sea que un nuevo sistema experto podía construirse con mucha mayor rapidez que un sistema del tipo DENDRAL, vaciando el viejo conocimiento e introduciendo conocimiento acerca del nuevo dominio. La parte de MYCIN que trataba con la inferencia y la explicación. El shell, podía llenarse entonces con conocimiento acerca del nuevo sistema. A el shell obtenido al eliminar el conocimiento médico de MYCIN se le llamaba EMYCIN (MYCIN esencial o vacío).

Para finales de los setenta, habían convergido los tres conceptos básicos para la mayor parte de los sistemas expertos actuales, como se muestra en la figura 1.5. Estos conceptos son: las reglas, el shell y el conocimiento.

Para 1980, compañías nuevas comenzaron a sacar los sistemas expertos del laboratorio universitario y a fabricar productos comerciales; se introdujo nuevo software con múltiples opciones para el desarrollo de sistemas expertos, incluyendo la Herramienta Automática de Razonamiento (Automated Reasoning Tool, ART) de la Inference Corp., la Knowledge Engineering Tool (KEE) de IntelliCorp, y Rulemaster de Radian Corp. Además, se desarrolló nuevo hardware especializado para ejecutar el software con mayor velocidad que antes. Compañías como Symbolics y LMI introdujeron computadoras consideradas como máquinas LISP, porque estaban diseñadas para LISP, el lenguaje fundamental de base del software desarrollado para los sistemas expertos. En estas máquinas, el lenguaje ensamblador nativo, el sistema operativo y todos los demás códigos fundamentales se crearon en LISP.

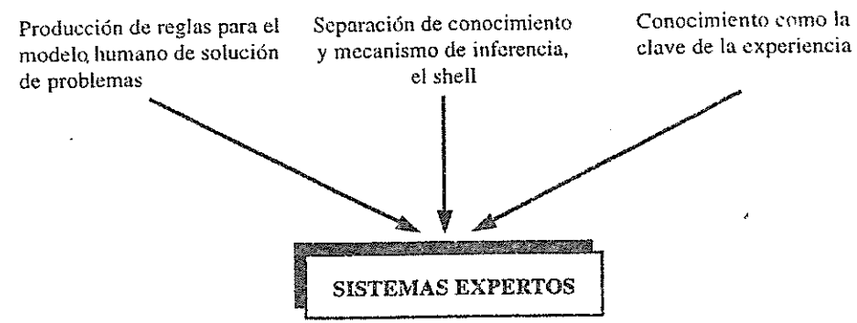


Figura 1.5
Convergencia de tres factores importantes en la creación del moderno sistema experto basado en reglas

Desafortunadamente, esta alta tecnología también tiene un precio alto. Aunque una máquina LISP para un solo usuario era mucho más potente y productiva que una para propósitos generales, la máquina y la licencia de software para un solo usuario cuestan 100,000 dólares. Caro Establecer un laboratorio de AI con media docena de programadores podría costar fácilmente 500,000 dólares.

Estas objeciones fueron sobrepasadas a mediados de los ochenta por la introducción del nuevo y potente software desarrollado, como CLIPS de la NASA. CLIPS está escrito en C para tener rapidez y movilidad, usa un poderoso acoplador de patrones llamado Algoritmo Rete, es gratuito para usuarios y contratistas gubernamentales; está disponible a un costo nominal para otros usuarios de COSMIC, que distribuye el software desarrollado por la NASA; las universidades pueden obtenerlo a mitad de precio. CLIPS está libre de derechos y cualquiera que obtenga una copia legítima de COSMIC puede distribuirlo gratuitamente.

CLIPS puede ser instalado en cualquier compilador C que soporte el lenguaje C de Kernigan y Richie. Ha sido instalado en PC de IBM y compatibles, VAX, Hewlett Packard, Sun, Cray y muchos otros fabricantes. COSMIC dispone de versiones para Windows de Macintosh y otras.

1.7 APLICACIONES Y DOMINIOS DE LOS SISTEMAS EXPERTOS

Los programas de cómputo convencionales se utilizan para resolver muchos tipos de problemas. Por lo general, estos problemas tienen soluciones algorítmicas que se prestan bien a los programas y a los lenguajes de programación convencionales como FORTRAN, Pascal, Ada, etcetera. En muchas áreas de aplicación como los negocios y la ingeniería, los cálculos nume-

ricos son de importancia primordial. En contraste, los sistemas expertos están diseñados principalmente para el razonamiento simbólico.

Aunque lenguajes como LISP y PROLOG también se utilizan para el manejo simbólico, sirven a propósitos más generales que los shells de sistemas expertos. Esto no significa que no sea posible construir sistemas expertos en LISP y PROLOG, en realidad, muchos sistemas expertos han sido construidos con ellos. Sobre todo PROLOG tiene varias ventajas especiales para los sistemas de diagnóstico debido a su encadenamiento hacia atrás integrado. Sin embargo, es más conveniente y eficiente construir los grandes sistemas expertos con shells y programas con shell y equipo diseñados específicamente para ellos. En lugar de "reinventar la rueda" cada vez que se construye un nuevo sistema experto, es más eficiente utilizar las herramientas especializadas.

Aplicaciones de los sistemas expertos

Los sistemas expertos se han aplicado casi a todos los campos del conocimiento. Algunos se han diseñado como herramientas de investigación, mientras que otros satisfacen importantes funciones de negocios e industriales. Un ejemplo de sistema experto usado en negocios de rutina es el sistema XCON de Digital Equipment Corporation (DEC). Este sistema llamado originalmente R1 fue desarrollado junto con John McDermott, de la Universidad Carnegie-Mellon. Y es un sistema experto para la configuración de sistemas de cómputo (McDermott 84).

La **configuración** de un sistema de cómputo significa que, cuando un consumidor hace un pedido, se suministran todas las partes correctas (software, hardware y documentación). Para los sistemas grandes, el sistema del consumidor se configura en la fábrica y se prueba para asegurar que cumple con los requisitos del cliente. A diferencia de la compra de un televisor o de una computadora para el hogar, un gran sistema de computo permite una gran cantidad de opciones e interconexiones. Al ensamblar un sistema grande, no basta con enviar el número solicitado de CPU, unidades de disco, terminales y demás; también deben proporcionarse las interconexiones y el cableado apropiados, además de revisar el sistema para verificar que funciona adecuadamente.

El sistema XCON es probablemente uno de los sistemas expertos de uso rutinario más exitoso y ahorra a la DEC millones de dólares al año, reduce el tiempo para configurar los pedidos y mejora su exactitud. XCON puede configurar una orden promedio en cerca de dos minutos, quince veces más rápido que un ser humano. Además, mientras que los humanos configuran las órdenes correctamente 70% de las veces, XCON tiene una eficiencia del 98%. Por esto, configurar un sistema de cómputo complejo en la fábrica constituye un gran esfuerzo. Es costoso configurar parcialmente un sistema para luego descubrir que no cumple con los requisitos del cliente, o que se necesitan otros componentes y que el envío debe retrasarse hasta que lleguen las piezas.

Cientos de sistemas expertos han sido construidos y comentados en publicaciones de cómputo, libros y conferencias. Tal vez esto sólo representa la punta del *iceberg*, porque muchas compañías y organizaciones militares no informarán de sus sistemas debido a las propiedades o secretos contenidos en el sistema. Basados en los sistemas descritos de la literatura pública, puede discernirse una amplia gama de aplicaciones para los sistemas expertos, como se muestra en la tabla 1.3.

De la tabla 1.4 a la 1.9 (Waterman 86), y en la bibliografía histórica al final de este capítulo, se muestran ejemplos de algunos sistemas expertos.

Clase	Área general
Configuración	Ensamblar correctamente los componentes apropiados de un sistema.
Diagnóstico	Inferir los problemas subyacentes basándose en la evidencia observada.
Instrucción	Enseñanza inteligente para que un estudiante pueda preguntar *Por qué, Cómo y Qué pasaría si*, como lo haría con un maestro humano.
Interpretación	Explicar los datos observados.
Supervisión	Comparar los datos observados con los esperados para juzgar el desempeño.
Planeación	Idear acciones para obtener el resultado deseado.
Pronóstico	Predecir el resultado de una situación dada.
Remedio	Prescribir tratamiento para un problema.
Control	Regular un proceso. Tal vez requiera de interpretación, diagnóstico, supervisión, planeación, pronóstico y remedios.

Tabla 1.3 Gran cantidad de sistemas expertos

Nombre	Química
CRYSALIS	Interpretar la estructura tridimensional de una proteína.
DENDRAL	Interpretar la estructura molecular.
TQMSTUNE	Remediar el espectrómetro masivo triple y cuádruple Toms (mantenerlo afinado).
CLONER	Diseñar nuevas moléculas biológicas.
MOLGEN	Diseñar experimentos para clonación de genes.
SECS	Diseñar moléculas orgánicas complejas.
SPEX	Planear experimentos de biología molecular.

Tabla 1.4 Sistemas expertos para la química

Nombre	Electrónica
ACE	Diagnosticar fallas en las redes telefónicas.
IN-ATE	Diagnosticar fallas en el osciloscopio.
NDS	Diagnosticar la red de comunicaciones nacional.
EURISKO	Diseñar microelectrónica para tercera dimensión.
PALLADIO	Diseñar y probar nuevos circuitos VLSI.
REDESIGN	Rediseñar circuitos digitales.
CADHELP	Instruir para diseño apoyado por computadora.
SOPHIE	Instruir en diagnóstico de falla de circuitos.

Tabla 1.5 Sistemas expertos en electrónica

Nombre	Medicina
PUFF	Diagnosticar enfermedades de los pulmones.
VM	Supervisar pacientes en terapia intensiva.
ABEL	Diagnosticar electrolitos/ácido-base.
AI/COAG	Diagnosticar enfermedades de la sangre.
AI/REUHM	Diagnosticar enfermedades reumáticas.
CADUCEUS	Diagnosticar enfermedades de medicina interna
ANNA	Supervisar terapia para dedos.
BLUE BOX	Diagnosticar/remediar depresión.
MYCIN	Diagnosticar/remediar infecciones bacterianas.
ONCOCIN	Remediar/administrar pacientes de quimioterapia.
ATTENDING	Capacitar en administración anestésica.
GUIDON	Capacitar en infecciones bacterianas.

Tabla 1.6 Sistemas expertos médicos

Nombre	Ingeniería
REACTOR	Diagnosticar/remediar accidentes de reactor.
DELTA	Diagnosticar/remediar locomotoras GE.
STEAMER	Instruir en operación, planta de energía a vapor.

Tabla 1.7 Sistemas expertos en ingeniería

Nombre	Geología
DIPMETER	Interpretar los registros del medidor de profundidad.
LITHO	Interpretar los datos de registro de pozos petroleros.
MUD	Diagnosticar/remediar problemas de perforación.
PROSPECTOR	Interpretar datos geológicos para buscar minerales.

Tabla 1.8 Sistemas expertos en geología

Nombre	Sistemas de computadora
PTRANS	Dar pronóstico para la administración de computadoras. DEC
BDS	Diagnosticar partes deficientes en la red de conmutación.
XCON	Configurar sistemas de cómputo DEC.
XSEL	Configurar las órdenes de venta de las computadoras DEC.
XSITE	Configurar el sitio del cliente para las computadoras DEC
YES/MVS	Supervisar/controlar el sistema operativo MVS de IBM.
TIMM	Diagnosticar computadoras DEC.

Tabla 1.9 Sistemas expertos en computadoras

¿Cómo saber si construir un sist. experto?

Dominios apropiados para los sistemas expertos

Antes de empezar a construir un sistema experto, es esencial decidir si es el paradigma apropiado. Por ejemplo, una preocupación es si debe usarse un sistema experto en lugar de un paradigma alternativo, como la programación convencional. El dominio apropiado de un sistema experto depende de varios factores:

- *¿Puede solucionarse eficazmente el problema con programación convencional?* Si la respuesta es sí, entonces un sistema experto no es la mejor elección. Por ejemplo, consideremos el problema de diagnosticar cierto equipo: si todos los síntomas de mal funcionamiento se conocen de antemano, entonces lo adecuado es una simple búsqueda de la falla en una tabla o árbol de decisiones. Los sistemas expertos son más adecuados para situaciones en las que no hay una solución algorítmica eficiente. A esos casos se les llama **problemas de estructura nociva** y sólo el razonamiento puede ofrecer esperanzas de una solución adecuada.

Poner

Tomemos como ejemplo de un problema de estructura nociva, el caso de una persona que está pensando en viajar y visita a un agente de viajes. Aunque la mayoría de las personas tiene un destino y planes elaborados, hay excepciones. En la tabla 1.10 se presenta una lista de algunas de esta características de un problema de estructura nociva, como lo indican las respuestas de esta persona a las preguntas del agente de viajes.

A pesar de que probablemente se trata de un caso exagerado, ilustra el concepto básico de un problema de estructura nociva. Como puede verse, este tipo de problema no se presta a una

ción predeterminada cuando todo lo demás falle. Por ejemplo, el agente de viajes diría: "¡Ajá! Tengo el viaje perfecto para usted, un crucero alrededor del mundo. Por favor llene este formulario de tarjeta de crédito y nos ocuparemos de todo".

Preguntas del agente de viajes	Respuestas
¿Puedo ayudarle?	No estoy seguro.
¿A dónde desea ir?	A algún lado.
¿Algún destino en particular?	Aquí y allá.
¿Qué tanto puede pagar?	No lo sé.
¿Puede conseguir algo de dinero?	No lo sé.
¿Cuándo quiere salir?	Tarde o temprano.

Tabla 1.10 Ejemplo de un problema de estructura nociva

¿Cuándo se cae en una estructura de control?

Al tratar con problemas de estructura nociva, existe el peligro de que el sistema experto pueda accidentalmente dar una solución idéntica a una algorítmica; es decir, el sistema experto puede descubrir en forma inadvertida una solución algorítmica. Un indicador de que se ha dado esta situación es que se encuentre una solución que requiera una rígida **estructura de control.** O sea, el ingeniero del conocimiento obliga a que las reglas se ejecuten en una determinada secuencia, ajustando explícitamente las prioridades de muchas reglas. Al forzar una estructura de control rígida en el sistema experto, se cancela una de las principales ventajas de su tecnología, que es la de tratar con entradas inesperadas que no siguen un patrón predeterminado (Parrello 88). Es decir, los sistemas expertos reaccionan de manera oportuna ante su entrada, cualquiera que ésta sea. Los programas convencionales suelen esperar que la entrada siga cierta secuencia. Un sistema experto con demasiado control a menudo indica un algoritmo encubierto y puede ser un buen candidato para volver a codificarse como un programa convencional.

Para un sist. Ex. para diagnosticar dolores de cabeza ¿Cuántos dominios debe tener?

- *¿Está bien delimitado el dominio?* Es importante tener límites bien definidos en lo que se espera que sepa el sistema experto y en las aptitudes que debe tener. Por ejemplo, supongamos que se desea crear un sistema experto para diagnosticar dolores de cabeza. Es cierto que el conocimiento médico de un facultativo debería ponerse en la base de conocimientos. Sin embargo, para una comprensión más profunda de los dolores de cabeza, quizá también se deban insertar conocimiento de neuroquímica, después su área padre, bioquímica, después química, biofísica molecular y así sucesivamente hasta llegar a la física subnuclear. Otros dominios como retroalimentación biológica, psicología, psiquiatría, fisiología, ejercicio, yoga y manejo del estrés pueden contener también conocimiento pertinente acerca de los dolores de cabeza. El punto es: ¿cuándo dejar de agregar dominios? Cuantos más dominios haya, más complejo se volverá el sistema experto.

En particular, la tarea de coordinar toda la experiencia se vuelve sumamente importante. Sabemos por experiencia qué tan difícil es coordinar equipos conformados por especialistas sobre todo cuando ofrecen recomendaciones conflictivas. Si sabemos cómo programar adecuadamente la coordinación de la experiencia, entonces podemos tratar de programar un sistema experto para que tenga el conocimiento de varios especialistas. Se han hecho intentos de coordinar varios sistemas expertos en los sistemas HEARSAY II y HEARSAY III. Sin embargo, esta coordinación es una tarea compleja que debería ser vista más como una investigación que como la elaboración de un producto confiable para los sistemas expertos.

- *¿Hay la necesidad y el deseo de tener un sistema experto?* Aunque es una gran experiencia construir un sistema experto, no tiene caso si nadie está dispuesto a usarlo. Si hay muchos especialistas humanos, es difícil justificar un sistema experto con base en la escasez de experiencia humana. Además, si los expertos o los usuarios no quieren el sistema, éste no será aceptado aunque sea necesario.

En especial, la administración debe estar dispuesta a apoyar el sistema. Esto es aún más crítico para los sistemas expertos que para los programas convencionales, porque los sistemas expertos son una nueva tecnología, lo que significa que hay poca gente con experiencia en ella y mucha incertidumbre acerca de lo que puede lograrse. Sin embargo, el área de sistemas expertos requiere más apoyo, porque trata de resolver problemas que los programas convencionales no pueden. Los riesgos son mayores, pero también las recompensas.

- *¿Hay al menos un especialista que esté dispuesto a cooperar?* Debe haber un perito que esté dispuesto a aplicar y de preferencia entusiasmado, con el proyecto; pues, no todos los especialistas están dispuestos a que se examine su conocimiento en busca de fallas y luego sea colocado en una computadora. Aunque haya muchos dispuestos a cooperar en el desarrollo, quizá sea sabio limitar el número de especialistas que participan en él, ya que pueden tener distintas formas de resolver un problema, como la aplicación de diferentes pruebas diagnósticas, e incluso llegar a diferentes conclusiones. Tratar de codificar varios métodos para resolver problemas en una sola base de conocimiento puede crear conflictos internos e incompatibilidades.
- *¿Puede el experto explicar el conocimiento en forma comprensible al ingeniero del conocimiento?* Aunque el especialista esté dispuesto a cooperar, puede tener dificultades al expresar el conocimiento en términos explícitos. Como un ejemplo simple de esta dificultad, ¿podría usted explicar con palabras cómo mueve un dedo? Aun si pudiera decir que se hace contrayendo un músculo, la siguiente pregunta sería, ¿cómo contrae un músculo del dedo? La otra dificultad de comunicación entre el especialista y el ingeniero en conocimiento es que éste último no conoce la terminología técnica del especialista, problema que se acentúa con la terminología médica. Puede tomar un año o más para que el ingeniero del conocimiento entienda de qué está hablando el perito, sin contar con la traducción del conocimiento a código explícito de computadora.
- *¿El conocimiento para resolver problemas es principalmente heurístico e incierto?* Los sistemas expertos *son* apropiados cuando el conocimiento del especialista es muy heurístico e incierto. Es decir, el conocimiento puede estar basado en la experiencia, lo que se le llama **conocimiento empírico**, y el especialista puede probar varios métodos en caso de que uno no funcione. En otras palabras, su conocimiento puede ser un método de ensayo y error, en lugar de uno basado en lógica y algoritmos. Sin embargo, el especialista aún puede resolver el problema más rápido que alguien que no lo es. Esta es una buena aplicación para los sistemas expertos. Si el problema puede resolverse únicamente con la lógica y los algoritmos, es mejor manejarlo con un programa convencional.

1.8 LENGUAJES, SHELLS Y HERRAMIENTAS

Una decisión fundamental al definir un problema es decidir qué tan bien modelarlo. Algunas veces se dispone de la experiencia que ayuda a escoger el mejor paradigma. Por ejemplo, la experiencia sugiere que una nómina se elabora mejor con un procedimiento de programación convencional; también sugiere que es preferible usar un paquete comercial, si está disponible, en lugar de escribir uno desde cero. Una guía general para seleccionar un paradigma es considerar primero el más tradicional: la programación convencional. La razón para esto es la vasta experiencia que tenemos con la programación convencional y la amplia variedad de paquetes comerciales disponibles. Si un problema no puede solucionarse eficazmente con la programación convencional, entonces hay que mirar hacia los paradigmas no convencionales, como la AI.

A pesar de que los sistemas expertos son una rama de la AI, existen lenguajes especializados para sistemas expertos que son completamente diferentes a los lenguajes comunes de la AI como LISP y PROLOG. Aunque se han desarrollado muchos otros, como IPL-II, SAIL, CONNIVER, KRL y Smalltalk, pocos se utilizan ampliamente, excepto con fines de investigación (Scown 85).

Un lenguaje para sistemas expertos es un lenguaje de orden más alto que los lenguajes como LISP o C porque permite hacer ciertas cosas con mayor facilidad, pero también permite atacar un rango más pequeño de problemas. Esto significa que la naturaleza especializada de estos lenguajes los hace adaptables para elaborar sistemas expertos, pero no para la programación en general. En muchos casos incluso es necesario salir temporalmente del lenguaje para sistemas expertos y ejecutar una función en el lenguaje de procesamiento.

La principal diferencia funcional entre estos lenguajes, es el enfoque de la representación. Los lenguajes de procedimiento se enfocan en proporcionar técnicas flexibles y robustas para representar datos. Por ejemplo, las estructuras de datos como matrices, registros, listas ligadas, pilas, colas y árboles son creadas y manipuladas fácilmente. Los lenguajes modernos como Modula-2 y Ada están diseñados para ayudar en la **abstracción de datos**, proporcionando estructuras para su encapsulamiento como módulos y paquetes. Esto proporciona un nivel de abstracción que después se implanta con métodos como operadores e instrucciones de control para producir un programa. Los datos están estrechamente entrelazados con los métodos para manipularlos. Por el contrario los lenguajes para los sistemas expertos se concentran en proporcionar formas robustas y flexibles de representar el conocimiento; el paradigma del sistema experto permite dos niveles de abstracción: abstracción de datos y **abstracción de conocimiento.** Los lenguajes para sistemas expertos separan específicamente los datos de los métodos para manipularlos. Un ejemplo de esta separación es la de los hechos (abstracción de datos) y de las reglas (**abstracción de conocimiento**) que emplea un lenguaje para el sistema experto basado en reglas.

Esta diferencia de enfoque también produce diferencias en la metodología de diseño del programa. A causa del estrecho entrelazamiento de los datos y el conocimiento en los programas orientados al procesamiento, los programadores deben describir con cuidado la secuencia de ejecución. Sin embargo, la separación explícita de datos y conocimiento en los lenguajes para sistemas expertos requiere un control mucho menos rígido de la secuencia de ejecución. Por lo general, se usa una pieza de código completamente separada, el mecanismo de inferencia, para aplicar el conocimiento a los datos; esta separación entre conocimiento y datos permite un mayor grado de paralelismo y modularidad.

Al escoger un lenguaje, una pregunta básica debe ser si el problema exige más conocimiento o inteligencia. Los sistemas expertos dependen de una gran cantidad de conocimiento especializado o experiencia para resolver un problema, mientras que la AI enfatiza un método para la solución de problema. Es común que los sistemas expertos dependan de la correspondencia de patrones en un dominio de conocimiento restringido para guiar su ejecución, mientras que la AI suele concentrarse en la búsqueda de paradigmas en dominios menos restringidos.

La manera acostumbrada para definir si hay necesidad de un sistema experto es decidir si se quiere programar la experiencia de un especialista humano, si éste existe y si va a cooperar, es entonces cuando una propuesta de sistema experto puede tener éxito.

El camino para seleccionar el lenguaje para el sistema experto está lleno de confusiones. Hace algunos años, la elección de un lenguaje de sistema experto era muy directa, solo había cerca de media docena de lenguajes disponibles y por lo general eran gratuitos o costaban una cantidad nominal que se pagaba a la universidad en la que se desarrollaban.

Sin embargo, con el explosivo crecimiento comercial en el campo de los sistemas expertos, desde los setenta, la selección de un lenguaje ya no es tan simple. En la actualidad hay docenas de lenguajes disponibles, con precios que llegan hasta los $75,000 dólares. A pesar de que todavía es posible obtener gratis algunos de los viejos lenguajes como el OPS5, o a costo nominal en las universidades donde se desarrollaron, se debe pagar un precio en eficiencia, falta de funciones modernas y soporte.

Además de la confusión al elegir uno entre los muchos lenguajes disponibles hoy, la terminología usada para describirlos es confusa. Algunos vendedores aluden a sus productos como "herramientas", mientras que otros hablan de shells y algunos otros de "ambientes integrados". Para mantener la claridad en este libro los términos se definirán como sigue:

- **Lenguaje:** es un traductor de comandos escrito con una sintaxis específica. Un lenguaje para sistemas expertos también proporcionará un mecanismo de inferencia que ejecute las instrucciones del lenguaje. Dependiendo de la forma en que esté implantado, el mecanismo de inferencia puede proporcionar encadenamiento hacia atrás, hacia delante o ambos. Bajo esta definición de lenguaje, LISP no es un lenguaje para los sistemas expertos, pero PROLOG sí. Sin embargo, es posible escribir un lenguaje de éstos usando LISP e AI en PROLOG. Para el caso, incluso puede escribirse un lenguaje para sistemas expertos o AI en lenguaje ensamblador. Las preguntas acerca del tiempo de desarrollo, la conveniencia, la conservación, la eficiencia y la velocidad, determinan en qué lenguaje debe estar escrito el software.
- **Herramienta:** es un lenguaje adicionalmente asociado con programas de utilerías para facilitar el desarrollo, la depuración y el uso de los programas de aplicación. Los programas de utilerías pueden incluir editores de texto e imágenes, depuradores, administradores de archivos e incluso generadores de código. También pueden proporcionarse ensambladores de plataforma cruzada para portar el código de desarrollo a un hardware diferente. Por ejemplo, un sistema experto puede desarrollarse en un VAX de DEC y después ensamblarse para ejecutarse en un Motorola 68000. Algunas herramientas pueden incluso admitir el uso de paradigmas diferentes, como el encadenamiento hacia delante y hacia atrás en una aplicación.

En algunos casos, una herramienta puede integrarse con todos sus programas utilitarios en un solo ambiente, para presentar al usuario una interfaz común. Este método minimiza la necesidad de que el usuario abandone el ambiente para ejecutar una tarea. Por ejemplo, una herramienta simple tal vez no proporcione medios para la administración de archivos y, por tanto, el usuario tendrá que salir de la herramienta para aplicar comandos del sistema operativo. Un ambiente integrado permite un fácil intercambio de datos entre varios programas utilitarios dentro del mismo ambiente. Algunas herramientas ni siquiera requieren que el usuario escriba algún código; en cambio, la herramienta permite al usuario introducir conocimiento mediante el ejemplo, a partir de tablas u hojas de cálculo, y genera el código apropiado por sí misma.

- **Shell:** herramienta con propósitos especiales, diseñada para cierto tipo de aplicaciones en las que el usuario sólo debe proporcionar la base de conocimiento. El ejemplo clásico de esto es el shell de EMYCIN (MYCIN vacío), éste se hizo al eliminar la base del conocimiento médico del sistema experto MYCIN.

MYCIN estaba diseñado como un sistema de encadenamiento hacia atrás para diagnosticar enfermedades. EMYCIN se creó por la simple eliminación del conocimiento médico y pudo usarse como una shell que contenía conocimiento acerca de otros tipos de sistemas consultivos que utilizan el encadenamiento hacia atrás. El shell EMYCIN demostró que podía volver a utilizarse el software esencial de MICYN como mecanismo de inferencia y la interfaz del usuario. Este fue un paso muy importante en el desarrollo de la tecnología para los modernos sistemas expertos, porque significó que no era necesario construir desde cero cada nueva aplicación.

Hay muchas formas de caracterizar a los sistemas expertos: como representación del conocimiento, encadenamiento hacia atrás o hacia delante, manejo de la incertidumbre, razonamiento hipotético, medios de explicación, etcétera. A menos que una persona haya construido cierto número de sistemas expertos, es difícil que aprecie todas estas funciones, sobre todo aquellas que se encuentran en las herramientas más caras. La mejor manera de aprender la tecnología de los sistemas expertos es desarrollar varios sistemas con un lenguaje fácil de aprender y después, si usted necesita sus funciones, invertir en una herramienta más sofisticada.

1.9 ELEMENTOS DE UN SISTEMA EXPERTO

En la figura 1.6 se muestran los elementos de un sistema experto típico. En un sistema basado en reglas, la base de conocimientos contiene el conocimiento de dominio necesario para resolver los problemas codificados en forma de reglas; mientras que las reglas son un paradigma popular para representar conocimiento, otros tipos de sistemas expertos usan diferentes representaciones, como se analiza en el capítulo 2.

Un sistema experto consta de los siguientes componentes:

- **interfaz de usuario:** el mecanismo que permite la comunicación entre el usuario y el sistema experto.
- **medio de explicación:** explica al usuario el razonamiento del sistema.
- **memoria activa: una base de datos global** de los hechos usados por las reglas.
- **mecanismo de inferencia:** hace inferencias al decidir cuáles reglas satisfacen los hechos u objetos, da prioridad a las reglas satisfechas y ejecuta la regla con la prioridad más elevada.
- **agenda:** una lista con prioridades asignadas a las reglas, creada por el mecanismo de inferencia, cuyos patrones satisfacen los hechos u objetos de la memoria activa.
- **medio para la adquisición de conocimiento.** vía automática para que el usuario introduzca conocimientos en el sistema, sin tener al ingeniero del conocimiento para que codifique éste en forma explícita.

El medio para la adquisición del conocimiento es una función opcional en muchos sistemas. En algunas herramientas para los sistemas expertos como KEE y First Class, ésta puede aprender por inducción de reglas mediante ejemplos, y también generar reglas automáticamente. Sin embargo, los ejemplos provienen generalmente de datos de tipo tabular u hojas de cálculo, que se aplican mejor a los árboles de decisión. Las reglas generales que construye un

ingeniero en conocimiento pueden ser mucho más complejas que las reglas simples creadas mediante inducción.

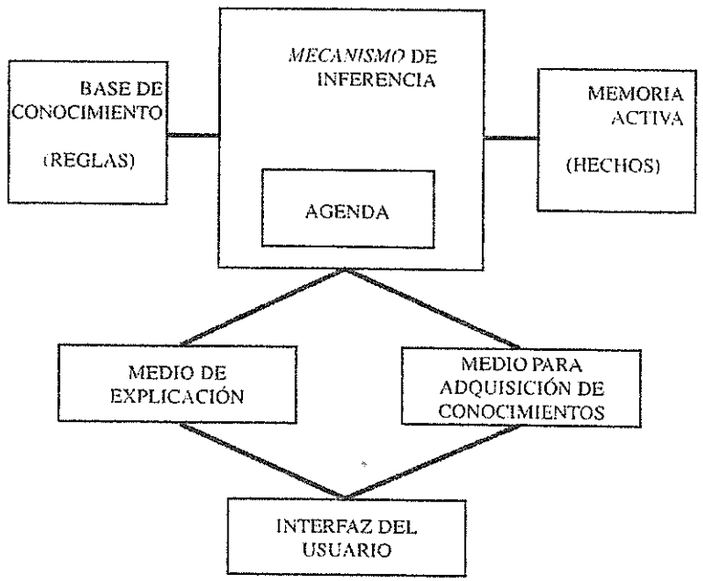


Figura 1.6 Estructura de un sistema experto basado en reglas

Dependiendo de la implantación del sistema, la interfaz de usuario puede ser una simple pantalla de texto o una pantalla más sofisticada, de alta resolución, con mapas de bits, que por lo común, se usa para simular un panel de control con botones y ventanas.

En un sistema experto basado en reglas, la base de conocimiento también recibe el nombre de **memoria de producción**. Tomemos como ejemplo el problema de decidir el cruce de una calle. La producción para las dos reglas es como sigue, donde las flechas significan que el sistema ejecutará las acciones que se encuentran a la derecha de la flecha si las condiciones de la izquierda son verdaderas:

```
La luz es roja → alto
La luz es verde → adelante
```

La producción de reglas puede expresarse en un pseudocódigo equivalente al formato SI...ENTONCES, como:


```
Regla: Luz_roja
SI
        la luz es roja
ENTONCES
        alto

Regla: luz_verde
SI
        la luz es verde
ENTONCES
        adelante
```

Cada regla se identifica con un nombre, seguido por la parte SI de la regla. A la sección entre las partes SI y ENTONCES de la regla se le da varios nombres como son **antecedente, parte condicional, patrón** o **lado izquierdo.** A la condición individual

```
la luz es verde
```

se le llama **elemento condicional** o **patrón.**

Algunos ejemplos de reglas de sistemas expertos reales son:

```
Sistema MYCIN para diagnóstico de meningitis y bacteremia
(infecciones bacterianas)
SI
   El medio de cultivo es la sangre, y
   La identidad del organismo no es conocida con
      certeza, y
   La coloración del organismo es gramnegativa, y
   La morfología del organismo es de bastoncillo, y
   El paciente presenta elevación de la temperatura
ENTONCES
   Hay una débil evidencia que sugiere(.4) que la
      identidad del organismo es pseudomona.

XCON/R1 para configurar los sistemas de cómputo VAX de DEC
SI
   El contexto actual es la asignación de dispositivos a
      los módulos Unibus y
   Hay una unidad de disco con puerto dual que no ha sido
      asignada y
   El tipo de controlador que se requiere es conocido y
   Hay dos controladores y ninguno de ellos tiene un
      dispositivo asignado, y
   Se conoce el número de dispositivos que estos
      controladores pueden soportar
```

```
ENTONCES
   Asignar la unidad de disco a cada uno de los
        controladores, y
   Revisar que los dos controladores han sido asociados y
        que cada uno soporta un drive
```

En un sistema basado en reglas, el mecanismo de inferencia determina cuáles antecedentes de regla, si hay alguno, queda satisfecho por los hechos. Dos métodos generales de inferencia que se usan con frecuencia como estrategias para la solución de problemas con los sistemas expertos son: **encadenamiento hacia delante** y **encadenamiento hacia atrás**. Otros métodos usados para necesidades más específicas pueden incluir análisis de medios y fines, reducción del problema, localización en reversa, prueba de generación de planes, planeación jerárquica y principio mínimo de cometido, y manejo restringido.

El encadenamiento hacia adelante es el razonamiento desde los hechos hacia las conclusiones que resultan de ellos. Por ejemplo, si usted ve que está lloviendo antes de salir de casa (el hecho), entonces debe llevar un paraguas (la conclusión).

El encadenamiento hacia atrás implica el razonamiento en reversa desde una hipótesis, habrá de comprobarse una posible conclusión, a los hechos que la sustentan. Por ejemplo, si usted no ha mirado hacia fuera y alguien entra con los zapatos mojados y un paraguas, su hipótesis será que está lloviendo; para apoyarla podría preguntar a la persona si en verdad está lloviendo, si la respuesta es sí, entonces la hipótesis es verdadera y se convierte en un hecho. Como se dijo antes, una hipótesis puede verse como un hecho cuya veracidad está en duda y necesita establecerse. La hipótesis puede interpretarse entonces como un objetivo a probar.

Dependiendo del diseño, un mecanismo de inferencia llevará a cabo el encadenamiento hacia atrás o hacia delante. Por ejemplo, OPS5 y CLIPS están diseñados para el encadenamiento hacia delante, mientras que EMYCIN ejecuta el encadenamiento hacia atrás, y otros tipos de mecanismo de inferencia, como ART y KEE, ofrecen ambos. La elección de este mecanismo depende del tipo de problema. El diagnóstico de problemas se resuelve mejor con el encadenamiento hacia atrás, mientras que la prognosis, la supervisión y el control se realizan mejor mediante el encadenamiento hacia delante.

La memoria activa puede contener hechos que contemplan el estado actual de la luz del semáforo, como "la luz es verde" o "la luz es roja". Uno de estos hechos o ambos pueden estar en la memoria activa al mismo tiempo. Si el semáforo está trabajando con normalidad, sólo uno de estos hechos estará en la memoria. Sin embargo, es posible que ambos hechos puedan estar en la memoria si hay un mal funcionamiento en el semáforo. Nótese la diferencia entre la base de conocimientos y la memoria activa. Los hechos no pueden interactuar entre sí, el hecho "la luz es verde" no tiene efecto sobre el hecho "la luz es roja"; en cambio, nuestro conocimiento del semáforo dice que si ambos hechos están presentes de manera simultánea, entonces hay una falla en el semáforo.

Si hay un hecho "la luz es verde" en la memoria activa, el mecanismo de inferencia se dará cuenta de que este hecho satisface la parte condicional de la regla de luz verde y pondrá esta regla en su agenda. Si una regla tiene varios patrones, entonces todos ellos deben satisfacerse de manera simultánea para que la regla pase a la agenda. Algunos patrones pueden satisfacerse especificando la ausencia de ciertos hechos en la memoria activa.

Cuando todos los patrones de una regla están satisfechos, se dice que está **activada** o **iniciada**. Muchas reglas activadas pueden estar en la agenda al mismo tiempo, en cuyo caso, el mecanismo de inferencia debe seleccionar una regla de **disparo**. El término *disparar* viene de la neurofisiología, el estudio del sistema nervioso. Una célula nerviosa individual o neurona emite una señal eléctrica cuando se estimula; la falta de más estímulos puede provocar que la neurona se dispare otra vez durante un breve periodo, a este fenómeno se le llama *refracción*. Los sistemas expertos basados en reglas están construidos usando la refracción con el propósito de prevenir enredos triviales. Es decir, que si la regla de la luz verde se mantiene disparando una y otra vez sobre el mismo hecho, el sistema experto no realizaría un trabajo útil.

Se han inventado muchos métodos para conseguir la refracción. En un lenguaje para sistemas expertos llamado OPS5, cada hecho recibe un identificador único, que se conoce como **etiqueta de tiempo**, cuando se introduce en la memoria activa. Después de que una regla disparó un hecho, el motor de inferencia no disparará otra vez sobre el mismo porque su etiqueta de tiempo ya se utilizó.

Después de la parte ENTONCES de una regla, hay una serie de **acciones** que se ejecutarán cuando la regla se dispare. A esta parte de la regla se le conoce como **la consecuencia** o **lado derecho (RHS)**. Cuando la regla de la luz roja se dispara, se ejecuta su acción, que es detenerse. Del mismo modo, cuando la regla de la luz verde se dispara, su acción es avanzar. Por lo general, las acciones específicas incluyen la adición o remoción de los hechos en la memoria activa o de los resultados de impresos. El formato de estas acciones depende de la sintaxis del lenguaje utilizado; por ejemplo, en OPS5, ART y CLIPS, la adición de un nuevo hecho llamado "ALTO" a la memoria activa sería (afirmar detener). Debido a su antecesor LISP, estos lenguajes se diseñaron para requerir paréntesis alrededor de los patrones y las acciones.

El mecanismo de inferencia opera en **ciclos**. Se han dado varios nombres para describir el ciclo, como: **ciclo acto-reconocimiento, ciclo selección-ejecución, ciclo situación-respuesta,** y **ciclo situación-acción.** Cualquiera que sea el nombre, el mecanismo de inferencia ejecutará un grupo de ciertas tareas repetidas hasta que ciertos criterios causen el cese de la ejecución. Las tareas de un ciclo para OPS5, un shell de sistema experto típico, se muestran en el siguiente seudocódigo como **resolución de conflicto, acto, correspondencia** y **verificación de interrupciones.**

MIENTRAS no se haga
  Resolución de conflicto: si hay activaciones, entonces seleccione aquella con la prioridad más alta.
  Acto: ejecutar en forma secuencial las acciones en el lado derecho de la activación seleccionada. Aquellas que cambian la memoria activa tienen un efecto inmediato en este ciclo. Eliminar de la agenda la activación que se acaba de disparar.
  Correspondencia: actualizar la agenda revisando si el lado izquierdo de cualquier regla están satisfechos. De ser así, activarlos. Eliminar acciones si el lado izquierdo de las reglas ya no se satisfacen.
  Verificación de interrupciones: si se ejecuta una acción de interrupción o se da un comando de ruptura, entonces hacerlo.
  FIN-MIENTRAS
  Aceptar las órdenes de un nuevo usuario

Varias reglas pueden activarse y pasar a la agenda durante un ciclo, las activaciones de ciclos previos también se dejarán en la agenda a menos que hayan sido desactivadas porque su lado izquierdo ya no está satisfecho. Así, el número de activaciones en la agenda variará mientras procede la ejecución. Dependiendo del programa, una activación puede estar siempre en la agenda pero nunca seleccionada para dispararse; del mismo modo, algunas reglas quizá nunca se activen. En estos casos, los propósitos de estas reglas deberían volver a examinarse porque o son innecesarias, o sus patrones no se diseñaron bien.

El mecanismo de inferencia ejecuta las acciones con mayor prioridad de activación en la agenda, luego, la siguiente en grado de prioridad y así sucesivamente hasta que no quede ninguna activación. Se han diseñado varios esquemas de prioridad en los shells de sistemas

expertos. En general, todos los shells permiten que el ingeniero del conocimiento defina la prioridad de las reglas.

Los conflictos en la agenda suceden cuando diferentes activaciones tienen la misma prioridad y el mecanismo de inferencia debe decidir cuál regla disparar. Los distintos shells tienen formas diferentes de tratar este problema: en el paradigma original de Newell y Simon, aquellas reglas que se introdujeron primero en el sistema tienen la prioridad predeterminada más alta (Newell 72, p. 33); en OPS5, las reglas con patrones más complejos tienen la más alta prioridad; en ART y CLIPS, las reglas tienen la misma prioridad predeterminada a menos que el ingeniero en conocimiento les asigne prioridades distintas.

Para este momento, el control ha regresado al **nivel máximo** de intérprete de comandos para que el usuario dé más instrucciones al shell del sistema experto. El nivel máximo es el modo predeterminado en que el usuario se comunica con el sistema experto, y está indicado por la tarea "Aceptar nuevas órdenes del usuario".

El nivel máximo es la interfaz del usuario con el shell mientras se está desarrollando una aplicación del sistema experto. Con regularidad, se diseñan interfaces de usuario más sofisticadas para facilitar la operación de los sistemas expertos. Por ejemplo, un sistema experto para controlar una planta de fabricación, puede tener una interfaz de usuario que muestra el diagrama de bloques de la planta, con despliegue de alta resolución de mapa de bits a colores. Las advertencias y los mensajes de estado pueden aparecer en colores brillantes, con botones y escalas simulados. En realidad, el mayor esfuerzo puede darse al diseñar e implementar la interfaz del usuario, y no en la base de conocimiento del sistema experto, sobre todo en un prototipo. Dependiendo de las capacidades del shell, la interfaz del usuario puede implementarse a partir de reglas o en otro lenguaje llamado por el sistema experto.

Un medio de explicación debe admitir que el usuario pregunte cómo llegó el sistema a cierta conclusión y por qué es necesaria cierta información. Para un sistema basado en reglas, la pregunta de cómo llegó el sistema a cierta conclusión es fácil de responder porque es posible mantener en un panel una historia de las reglas activadas y del contenido de la memoria activa. Los medios de explicación sofisticados permiten que el usuario haga preguntas del tipo "¿Qué pasa si?" para explorar las rutas de razonamiento que se alternan a través del razonamiento hipotético.

1.10 SISTEMAS DE PRODUCCIÓN

Uno de los tipos más populares de los sistemas expertos en la actualidad, es aquel basado en reglas. Las reglas son populares por varias razones.

- *Naturaleza modular.* Esto hace más fácil encapsular el conocimiento y expandir el sistema experto a partir de un desarrollo creciente.
- *Medios de explicación.* Es sencillo construir medios de explicación con reglas porque los antecedentes de una regla especifican con exactitud lo que es necesario para activarla. Al mantener el registro de las reglas que se dispararon, un medio de explicación puede presentar la cadena de razonamiento que condujo a cierta conclusión.
- *Semejanza con el proceso cognitivo humano.* Con base en el trabajo de Newell y Simon, las reglas aparecen como un modelo natural de la manera en que los humanos resuelven problemas. La representación SI...ENTONCES de las reglas facilita la explicación de los especialistas sobre la estructura del conocimiento que se trata de obtener de ellos. Otras ventajas de las reglas se describen en Hayes-Roth (85).

Las reglas son un tipo de producción cuyos orígenes se remontan a los cuarenta. Debido a la importancia de los sistemas basados en reglas, vale la pena examinar el desarrollo del concepto de regla. Esto dará una mejor idea de por qué los sistemas basados en reglas son tan útiles para los sistemas expertos.

Sistemas de producción de Post

Los sistemas de producción fueron usados primero en lógica simbólica por Post, lo que dio origen a su nombre (Post 43). Él comprobó el importante y sorprendente resultado de que cualquier sistema de matemáticas o lógica podría escribirse como cierto tipo de sistema de reglas de producción. Este resultado establecía la gran capacidad de las reglas de producción para representar clases de conocimiento más grandes en lugar de estar limitadas a unos cuantos tipos. Bajo el término **reescribir reglas**, éstas también se usan en lingüística como una manera de definir la gramática de un lenguaje. Los lenguajes de computadora suelen definirse utilizando la Forma Backus-Naur (BNF) de reglas de producción.

La idea básica de Post era que cualquier sistema matemático o lógico representaba sólo un conjunto de reglas que especificaban cómo cambiar una hilera de símbolos por otra. Es decir que, dada una hilera de entrada, el antecedente, una regla de producción podría producir una hilera nueva, la consecuencia. Esta idea también es válida con programas y sistemas expertos en los que la hilera inicial de símbolos la constituyen los datos de entrada y la hilera de salida es una transformación de lo introducido.

Como un caso muy simple, supóngamos que la cadena de entrada es "el paciente tiene fiebre", entonces la cadena de salida podría ser "tome una aspirina". Nótese que no hay ningún significado que enlace a estas cadenas. Es decir, la manipulación de las cadenas se basa en la sintaxis y no en la semántica o en la comprensión de lo que significan *fiebre, paciente* o *aspirina.* Un humano sabe lo que estas cadenas significan en términos del mundo real, pero un sistema de producción Post es sólo una forma de transformar una cadena en otra. Una regla de producción para este ejemplo podría ser

```
Antecedente → Consecuencia
La persona tiene fiebre → que tome una aspirina
```

donde la flecha indica la transformación de una cadena en otra. Esta regla se puede interpretar en términos de la notación más familiar SI...ENTONCES de esta manera:

```
SI la persona tiene fiebre ENTONCES que tome un aspirina
```

Las reglas de producción también pueden tener múltiples antecedentes. Por ejemplo:

```
La persona tiene fiebre Y
   La fiebre es más alta que 39 → vea al doctor
```

Nótese que el conectivo especial Y no es parte de la hilera, sencillamente indica que la regla tiene varios antecedentes.

Un sistema de producción de Post consta de un grupo de reglas de producción, como las siguientes (los números entre paréntesis son para propósitos de este análisis):

```
(1)   el carro no arranca → revisar batería
(2)   el carro no arranca → revisar gasolina
(3)   revisar batería Y si la batería está mal →
      reemplazarla
(4)   revisar gasolina Y si no hay gasolina → llenar el
      tanque de gasolina
```

Si hay una cadena "el carro no arranca", es posible usar las reglas (1) y (2) para generar las cadenas "revisar la batería" y "revisar gasolina", sin embargo, no hay un mecanismo de control que aplique ambas reglas a la cadena. Sólo puede aplicarse una regla, o ambas o ninguna. Si hay otra cadena "la batería está mal" y una cadena "revisar batería", entonces la regla (3) puede aplicarse para generar la cadena "remplazar batería".

El orden con el que se escriban las reglas escritas no tiene un significado especial. Las de nuestro ejemplo también podrían escribirse en el siguiente orden y el sistema seguiría siendo el mismo:

```
(4)   revisar gasolina Y si no hay → llenar el
      tanque
(2)   el carro no arranca → revisar gasolina
(1)   el carro no arranca → revisar batería
(3)   revisar batería Y si la batería está mal →
      reemplazar la batería
```

A pesar de que las reglas de producción de Post eran útiles en la colocación de la parte fundamental de los sistemas expertos, no eran adecuadas para escribir los programas prácticos. La limitante básica de las reglas de producción de Post para la programación es la falta de una **estrategia de control** para guiar la aplicación de las reglas. Un sistema Post permite que las reglas se apliquen de cualquier manera en las hileras porque no hay una especificación dada sobre cómo deberían aplicarse.

Como una analogía, supóngamos que alguien va a la biblioteca para buscar un libro acerca de sistemas expertos. Una vez en ella, comienza a mirar aleatoriamente los libros; si la biblioteca es muy grande, encontrar el libro que necesita puede tomar mucho tiempo. Aun si encuentra la sección de libros en sistemas expertos, su siguiente decisión puede llevarlo por accidente a una sección completamente distinta, como la de cocina francesa. La situación empeora aún más si necesita material del primer libro para ayudarle a determinar el segundo libro que requiere. Una búsqueda al azar del segundo libro también tomará mucho tiempo.

Algoritmos de Markov

El siguiente adelanto en la aplicación de las reglas de producción fue hecho por Markov, quien especificó una estructura de control para los sistemas de producción (Markov 54). Un **algoritmo de Markov** es un grupo ordenado de producciones que se aplican en orden de prioridad a una hilera de entrada. Si la regla de prioridad más alta no es aplicable, entonces se aplica la siguiente y así sucesivamente. El algoritmo de Markov termina si (1) la última producción no es aplicable a una hilera o (2) se aplica una producción que termina con un punto.

Los algoritmos de Markov también pueden aplicarse a las subhileras de una hilera, comenzando de la izquierda. Por ejemplo, el sistema de producción que sólo incluye la regla

```
AB → HIJ
```

cuando se aplica a la hilera de entrada GABKAB produce la nueva hilera GHIJKAB. Puesto que la producción ahora se aplica a la nueva hilera, el resultado final es GHIJKHIJ.

El símbolo especial ^ representa la **hilera nula** sin caracteres. Por ejemplo, la producción

```
A → ^
```

borra las apariciones del carácter A en una hilera.

Otros símbolos especiales representan cualquier símbolo individual y están indicados por las letras minúsculas a, b, c, etcétera. Estos símbolos representan variables de caracteres individuales y son parte importante de los lenguajes para los sistemas expertos modernos. Por ejemplo, la regla

```
AxB → BxA
```

invertirá los caracteres A y B.

Las letras griegas α, β, y las siguientes se usan para la puntuación especial de las cadenas. Se usan las letras griegas porque son distintas del alfabeto ordinario.

Un ejemplo de un algoritmo de Markov que mueve la primera letra de una hilera de entrada al último sitio es como a continuación lo muestra Elson (73). Las reglas están ordenadas en términos de la prioridad más alta (1), la siguiente (2) y así en sucesión. A las reglas se les da prioridad en el orden en el que se introdujeron.

```
(1)   αxy → yαx
(2)   α → ^
(3)   ^ → α
```

El trazo de la ejecución se muestra en la tabla 1.11 para la cadena de entrada ABC.

Regla	Éxito o fracaso	Hilera
1	F	ABC
2	F	ABC
3	E	αABC
1	E	BαAC
1	E	BCαA
1	F	BCαA
2	E	BCA

Tabla 1.11 Trazo de la ejecución de un algoritmo de Markov

Nótese que el símbolo α actúa de manera análoga a la de una variable temporal en un lenguaje de programación convencional. Sin embargo, en lugar de mantener un valor, el símbolo α se usa para apartar lugares y marcar la progresión de los cambios en la cadena de entrada. Una vez que el trabajo está hecho, la regla 2 elimina el símbolo α. El programa termina entonces cuando la regla 2 se aplica, porque hay un periodo después de la regla 2.

Algoritmo Rete

Puede notarse que hay una estrategia de control definitiva para los algoritmos de Markov, con las reglas de más alta prioridad ordenadas en primer lugar. Mientras procede la regla de mayor prioridad, ésta se utiliza. Si no, el algoritmo de Markov prueba reglas de prioridad más baja. A pesar de que el algoritmo de Markov puede utilizarse como la base de un sistema experto, es muy poco eficaz en sistemas con muchas reglas. El problema de la eficiencia es de gran importancia si queremos crear sistemas expertos para problemas reales que contengan cientos o miles de reglas. Sin importar qué tan bueno sea todo lo demás en un sistema, si el usuario tiene que esperar mucho tiempo para una respuesta, el sistema no se usará. Lo que en verdad es necesario es un algoritmo que conozca todas las reglas y pueda aplicar cualquier regla sin tener que probar cada una en forma secuencial.

Una solución para este problema es el **algoritmo Rete** desarrollado en 1979 por Charles L. Forgy en la Universidad Carnegie-Mellon para su tesis de doctorado en filosofía, en el shell del sistema experto OPS (sistema de producción oficial) de esa institución. El algoritmo Rete es un rápido igualador de patrones que obtiene su velocidad del almacenamiento de información sobre las reglas de una red. En lugar de tener que igualar los hechos con todas las reglas en cualquier ciclo-acto reconocimiento, el algoritmo Rete sólo busca los cambios en las correspondencias de cada ciclo. Esto acelera en gran medida la correspondencia de los hechos con los antecedentes, porque los datos estáticos que no cambiaron de un ciclo a otro pueden pasarse por alto. (Este tema se analizará con mayor amplitud en los capítulos sobre CLIPS.) Los algoritmos de igualado rápido como Rete completaron las bases para la aplicación práctica de los sistemas expertos. En la figura 1.7 se resumen los fundamentos de la tecnología de los sistemas expertos modernos basados en reglas.

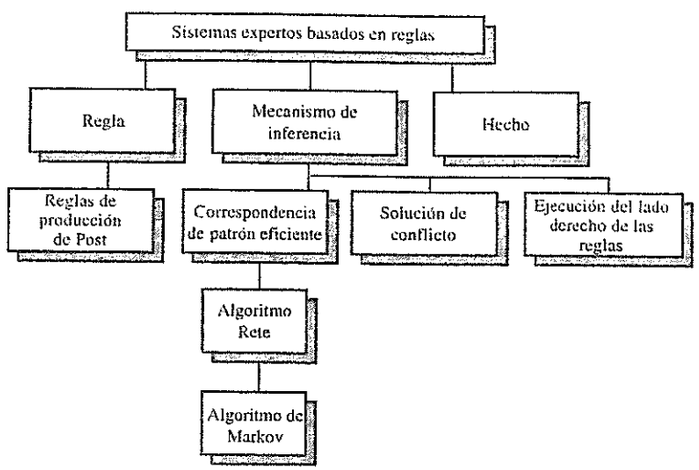


Figura 1.7 Fundamentos de los modernos sistemas expertos basados en reglas

Se han desarrollado diferentes versiones del lenguaje OPS y de su shell, incluyendo OPS2, OPS4 y OPS5. Una versión comercial más reciente desarrollada por Forgy es OPS83, un shell muy veloz. Sin embargo, ésta tiene diferencias significativas de sintaxis con el estilo OPS5 y depende, en parte, de los procedimientos para una ejecución más rápida.

1.11 PARADIGMAS DE PROCEDIMIENTOS

Los paradigmas de programación pueden clasificarse como procedurales y no procedurales. En la figura 1.8 se muestra una **taxonomía** o clasificación de los paradigmas de procedimientos en términos de lenguajes. En la figura 1.9 se muestra una taxonomía de paradigmas que no son de procedimientos. En estas figuras se ilustra la relación de los sistemas expertos con otros paradigmas y sólo deben considerarse como guías generales, no como definiciones estrictas. Algunos de los paradigmas y lenguajes tienen características que los sitúan en más de

una clase. Por ejemplo, algunos consideran que la programación funcional es de procedimientos, mientras que otros la consideran declarativa (Ghezzi 87).

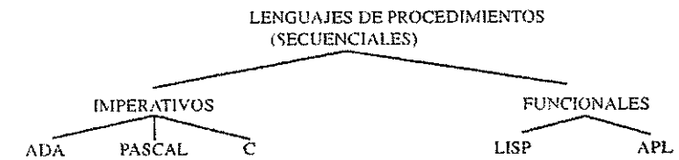


Figura 1.8 Lenguajes de procedimientos

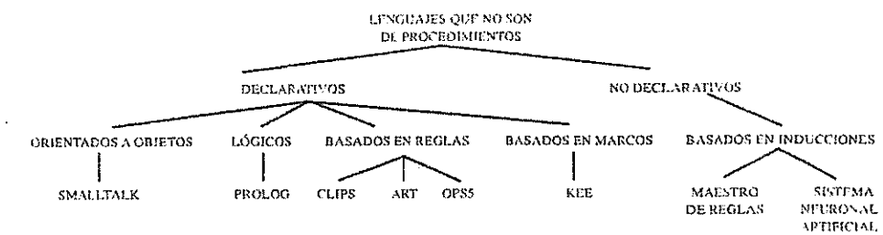


Figura 1.9 Lenguajes que no son de procedimientos

Un **algoritmo** es un método de solución de problemas en un número finito de pasos. Implementar un algoritmo en un programa es un **programa de procedimientos**. Los términos ***programación algorítmica, programación de procedimientos y programación convencional*** suelen usarse como conceptos intercambiables que se refieren a programas que no son del tipo AI. Una concepción frecuente del programa de procedimientos es que procede en forma secuencial, instrucción por instrucción, hasta que se encuentra una instrucción de ramificación. Otro sinónimo común para estos programas es el de programas **secuenciales**. Sin embargo, el término *programación secuencial* implica demasiada restricción si consideramos que todos los lenguajes de programación modernos apoyan la repetición y, por tanto, no son secuenciales en el sentido estricto.

El rasgo que distingue a un paradigma de procedimientos es que el programador debe especificar con exactitud *cómo* debe codificarse la solución de un problema. Incluso los generadores de código deben producir código de procedimientos. En cierto sentido, el uso de generadores de código pertenece a la **programación que no es de procedimientos**, porque remueve todos o la mayor parte del código de procedimientos que escribe el programador. La meta de esta clase de programación es hacer que el programador especifique cuál es el objetivo y luego deje que el sistema determine la manera en que lo alcanzará.

Programación imperativa

Los términos **imperativa** y **orientada a las afirmaciones** se usan como sinónimos. Los lenguajes como FORTRAN, Ada, Pascal, Modula 2, COBOL y BASIC tienen la característica predominante de que las afirmaciones son imperativos u órdenes, que indican a la computadora lo que tiene que hacer. Los lenguajes imperativos se desarrollaron como una forma de liberar al programador de la necesidad de codificar el lenguaje ensamblador en la arquitectura de Von Neumann. En consecuencia, los lenguajes imperativos ofrecen gran soporte a varia-

bles, asignación de operaciones y repetición. Todas éstas son operaciones de bajo nivel que los lenguajes modernos tratan de sustituir proporcionando funciones como reiteración, procedimientos, módulos, paquetes, etcétera. Los lenguajes imperativos también se caracterizan por su énfasis en las estructuras de control rígidas y sus diseños de programa **inferiores.**

Un problema serio en todos los lenguajes es la dificultad de probar si los programas son correctos. Desde el punto de vista de la AI, otro problema serio es que los lenguajes imperativos no son muy eficientes para manipular símbolos. Como la arquitectura del lenguaje imperativo se moldeó para ajustarse a la arquitectura de la computadora de von Neumann, hay lenguajes que pueden soportar sin problemas el procesamiento de números, pero no las manipulaciones simbólicas. Sin embargo, los lenguajes imperativos como C y Ada se han usado como el lenguaje básico subyacente para escribir shells de los sistemas expertos. Estos lenguajes y las shells construidas a partir de ellos se ejecutan con mayor eficiencia y rapidez en las computadoras comunes para propósitos generales que los shells antiguos que se construyeron usando LISP.

Debido a su naturaleza secuencial, los lenguajes imperativos no son muy eficientes para implementar directamente sistemas expertos, sobre todo los basados en reglas. Como una ilustración de este problema, consideremos el conflicto de codificar la información de un problema del mundo real con cientos o miles de reglas, por ejemplo, el sistema XCON, utilizado por DEC para configurar sistemas de computadora, tiene en la actualidad cerca de 7000 reglas en su base de conocimiento. Previamente, se hicieron intentos infructuosos para codificar este programa en FORTRAN y BASIC, antes de tomar con éxito el método del sistema experto. El modo directo para codificar este conocimiento en un lenguaje imperativo hubiera requerido 7000 afirmaciones SI...ENTONCES o un CASE muy largo; pues este estilo de codificar presentaría grandes problemas de eficiencia si consideramos que las 7000 reglas necesitan revisarse en busca de los patrones de correspondencia en cada ciclo de acto de reconocimiento. Nótese que el mecanismo de inferencia y su ciclo acto reconocimiento tendrían que estar codificados también en lenguaje imperativo.

La eficiencia del programa podría aumentar si las reglas se ordenaran de manera que se pusiera al principio aquella con mayor probabilidad de ejecutarse. Sin embargo, esto requeriría una sintonización considerable del sistema y cambiaría cuando se añadieran nuevas reglas o cuando se borraran o modificaran las anteriores. Un mejor método para aumentar la eficiencia podría ser la construcción de un árbol de los patrones de reglas, para reducir el tiempo de búsqueda al determinar cuáles reglas habrán de activarse. En lugar de hacer que el programador construya en forma manual el árbol, sería preferible hacer que la computadora lo construyera automáticamente, basándose en el patrón y en la sintaxis de acción de las reglas SI...ENTONCES. También sería útil tener una sintaxis SI...ENTONCES que fuera más conducente a representar el conocimiento y tuviera pruebas versátiles de correspondencia de patrones, pero esto requiere el desarrollo de un analizador que estudie la estructura de entrada y un intérprete o compilador para ejecutar la nueva sintaxis SI...ENTONCES.

Cuando todas estas técnicas se implementan para aumentar la eficiencia, el resultado es un sistema experto calificado. Si el mecanismo de inferencia, el analizador y el intérprete se remueven para facilitar el desarrollo de otros sistemas expertos, éstos constan de una shell de sistema experto. Por supuesto, en lugar de hacer todo este desarrollo desde el principio, en la actualidad es mucho más fácil usar un shell existente que esté documentado y que haya sido probado varias veces.

Programación funcional

La naturaleza de la **programación funcional**, como se ejemplificó mediante lenguajes como LISP y APL, es diferente a los lenguajes orientados a las afirmaciones, con su gran dependencia en elaboradas estructuras de control y en los diseños inferiores. La idea fundamental de la

programación funcional es combinar las funciones simples para producir funciones más potentes. En esencia, se trata de un diseño **superior**, comparado con los diseños comunes de los lenguajes imperativos.

La programación funcional se centra en las **funciones**. En matemáticas, una función es una **asociación** o regla que define la posición de los miembros de un conjunto, el **dominio**, en otro conjunto, el **codominio**. Un ejemplo de la definición de una función es:

```
cubo (x) ≡ x*x*x, donde x es un número real y cubo es una
función con valores reales
```

Las tres partes de la definición de la función son:

(1) la asociación, $x*x*x$
(2) el dominio, números reales
(3) el codominio, números reales

de la función cubo. El símbolo ≡ significa "es equivalente a" o "se define como" La siguiente notación es una manera más corta de escribir que la definición de posiciones . cubo es del dominio de números reales, simbolizado como $\Re$, al codominio de números reales.

$$\text{cubo}: \Re \rightarrow \Re$$

Una notación general para una función f que asigna las posiciones de un dominio S a un dominio T es $f: S \rightarrow T$ (Gersting 82). El **rango** de la función f es el grupo de todas las imágenes $f(s)$, donde s es un elemento de S. En el caso de la función cúbica, las imágenes de s son $s*s*s$ y el rango es el conjunto de todos los números reales. El rango y el codominio son el mismo para la función cúbica. Sin embargo, esto puede no ser cierto para otras funciones como la función al cuadrado, $x*x$, con dominio y codominio de números reales. Como el rango de la función al cuadrado es sólo el de los números reales no negativos, el rango y el codominio no son el mismo.

Usando notación de conjuntos, el rango de una función pude escribirse como:

$$R \equiv \{f(s) \mid s \in S\}$$

Las **llaves** ({ }) denotan un **conjunto**. La barra (|) se lee como "tal que". La afirmación anterior puede leerse: el rango R es equivalente al conjunto de valores $f(s)$, tal que todos los elementos s se encuentran en el conjunto S. La asociación es un conjunto de pares ordenados (s,t), donde $s \in S$, $t \in T$, y $t=f(s)$. Cada miembro de S debe tener uno y sólo un elemento de T asociado con él. Sin embargo, varios valores de t pueden estar asociados con un solo s. Como ejemplo, cada número positivo n tiene dos raíces cuadradas, $\pm\sqrt{n}$

Las funciones también pueden definirse en forma recursiva como en:

```
factorial (n) ≡ n*factorial (n-1)
   Donde n es un número entero y
   factorial es una función entera
```

Las funciones recursivas se utilizan con frecuencia en lenguajes funcionales como LISP.

Los conceptos y las expresiones matemáticas son **referencias transparentes**, porque el significado del todo está determinado en su totalidad por sus partes. No hay sinergismo implicado entre las partes Por ejemplo, consideremos la expresión funcional $x+(2*x)$. Ob-

viamente, el resultado es $3*x$. Ambos, $x+(2*x)$ y $3*x$, dan el mismo resultado sin importar los valores que se sustituyan por x. Incluso cuando otras funciones se sustituyen por x, el resultado es el mismo. Por ejemplo, considerando que $h(y)$ es una función arbitraria. Entonces $h(y)+(2*h(y))$ sería equivalente a $3*h(y)$.

Ahora, consideremos la siguiente afirmación de tarea en un lenguaje de computadora imperativo como Pascal:

```
sum := f(x) + x
```

Si el parámetro x pasa por referencia y su valor se cambia en la llamada de función, $f(x)$. ¿Cuál valor se usará para x? Dependiendo de la manera en que esté escrito el compilador, el valor de x puede ser el valor original si se guardó en una pila, o el nuevo valor si x no se guardó. Puede surgir más confusión si un compilador evalúa las expresiones de derecha a izquierda mientras que otro lo hace de izquierda a derecha. En este caso, $f(x)+x$ no se evaluaría a igual que $x+f(x)$ en diferentes compiladores; incluso cuando se usa el mismo lenguaje pueden ocurrir otros efectos secundarios debido a variables globales. Por tanto, a diferencia de las funciones matemáticas, las funciones de los programas no son referencias transparentes.

Los lenguajes de programación funcionales se crearon para ser referencias transparentes. Un lenguaje funcional se compone de cinco partes:

- **objetos de datos** sobre los que operan las funciones del lenguaje
- **funciones primitivas** para operar sobre los objetos de datos
- **formas funcionales** para sintetizar nuevas funciones a partir de otras funciones
- **operaciones de aplicación** sobre funciones que devuelven un valor
- **procedimientos de nominación** para identificar nuevas funciones.

Por lo general, los lenguajes funcionales se implementan para facilitar la construcción y la respuesta inmediata del usuario.

En LISP (procesamiento de listas), los objetos de datos son **expresiones simbólicas** (expresiones-S) que pueden ser **listas** o **átomos**, mientras que en **APL** (lenguaje de programación A) los objetos son matrices. Algunos ejemplos de listas son:

```
(leche huevos queso)
(compras (abarrotes (leche huevos queso) ropa
(pantalones)))
()
```

Las listas siempre están encerradas entre paréntesis de correspondencia, con espacios que separan los elementos. Los elementos de la lista pueden ser átomos, como leche, huevos y queso, o listas insertadas como (leche huevos queso) y (pantalones). Las listas pueden separarse pero los átomos no. La **lista vacía**, ( ), no contiene ningún elemento y se le llama **nula**.

Debe haber algunas funciones primitivas proporcionadas por el lenguaje como una base para la construcción de funciones más complejas. En la versión original de LISP, creada por John McCarthy en 1960, había pocas funciones primitivas, como se muestra en la tabla 1.12. Las funciones primitivas CAR, CDR, CPR y CTR son acrónimos que deben su nombre a los registros específicos del hardware en la primera máquina en que se corrió LISP; CPR y CTR ahora son obsoletos, pero los otros se han mantenido. También se muestran los **predicados** de LISP, que son funciones especiales que devuelven valores que representan verdadero o falso.

La versión original de LISP se llamó LISP "puro" porque era funcional en su totalidad. Sin embargo, no era muy eficiente para escribir programas. Las adiciones no funcionales han

hecho que LISP incremente su eficiencia en la escritura de programas. Por ejemplo, SET actúa como el operador de asignación, mientras que LET y PROG pueden usarse para crear variables locales y para ejecutar una secuencia de expresiones S. Aunque éstos actúan como funciones, no son funcionales en el sentido matemático original.

Función	Predicados
QUOTE	ATOM
CAR	EQ
CDR	NULL
CPR	
CTR	
CONS	
EVAL	
COND	
LAMBDA	
DEFINE	
LABEL	

Tabla 1. 12 Primitivos y funciones del LISP original

Desde su creación, LISP ha sido el lenguaje para AI líder en los Estados Unidos. Muchos de los shells originales de los sistemas expertos se escribieron en LISP porque es muy fácil experimentar con él. No obstante, las computadoras convencionales no ejecutan LISP con mucha eficiencia e incluso son peores al ejecutar los shells construidos usando LISP. Para evitar este problema, varias compañías ofrecen máquinas diseñadas de manera específica para ejecutar código LISP, éstas lo usan a plenitud, incluso como lenguaje ensamblador, sin embargo, cuestan mucho más que las máquinas convencionales y son para usuarios individuales.

Este problema de alto costo tiene un impacto tanto en el desarrollo como en el **problema de distribución**. No es suficiente desarrollar un programa estupendo si éste no puede distribuirse para su uso debido a su alto costo. Una buena estación de trabajo de desarrollo no es necesariamente un buen vehículo de distribución debido a su velocidad, poder, tamaño, peso, ambiente o restricciones de costo. Algunas aplicaciones incluso pueden requerir que el código final se sitúe en ROM por razones de costo y duración. Poner el código en ROM puede ser un problema con algunas herramientas de la AI y los sistemas expertos que requieren hardware especial para ejecutarse. Es mejor considerar esta posibilidad por adelantado, en vez de tener que volver a codificar un programa.

Un problema adicional es el de insertar la AI con los lenguajes de programación convencional como C, Ada, Pascal y FORTRAN. Por ejemplo, algunas aplicaciones que requieren procesamiento extensivo de números se hacen mejor en los lenguajes convencionales, o en los lenguajes para sistemas expertos escritos en LISP o PROLOG.

A menos que se tomen previsiones especiales, los sistemas expertos escritos en LISP a menudo son difíciles de insertar en cualquier programa diferente. Una consideración importante al seleccionar un lenguaje de AI debe ser el lenguaje en el que la herramienta está escrita. Por razones de transporte, eficiencia y velocidad, muchas herramientas de los sistemas expertos en la actualidad se escriben en C o se convierten a él. Esto también elimina el problema de necesitar costoso hardware especial para las aplicaciones basadas en LISP.

1.12 PARADIGMAS QUE NO SON DE PROCEDIMIENTOS

Los paradigmas que no son de procedimientos no dependen de que el programador proporcione los detalles exactos sobre cómo solucionar un problema. Esto es lo opuesto a los paradigmas de procedimientos, que especifican la manera en que se computa una función o una secuencia de afirmaciones. Hay que recordar que en los paradigmas que no son de procedimientos, el énfasis está en especificar lo que se va a realizar y en permitir que el sistema determine cómo va a realizarlo.

Programación declarativa

El **paradigma declarativo** separa la **meta** de los métodos utilizados para alcanzarla. El usuario especifica la meta mientras que el mecanismo subyacente de implementación trata de satisfacerla. Se han creado varios paradigmas y lenguajes de programación asociados para implementar el modelo declarativo.

Programación orientada a objetos

El paradigma **orientado a objetos** es otro caso de paradigma que puede considerarse en parte imperativo y en parte declarativo. Hoy en día, el término *orientado a objetos* se usa en dos formas diferentes. La expresión ***diseño orientado a objetos*** está creciendo en popularidad como una metodología de programación para los lenguajes de programación imperativa como Ada y Modula-2. La idea básica es diseñar un programa considerando los datos usados en él como objetos y después implementando operaciones en esos objetos. Esto es lo opuesto a los diseños inferiores, que proceden a partir del refinamiento gradual de una estructura de control del programa. En realidad, el diseño orientado a objetos es, en esencia, lo que solía llamarse diseño superior; por desgracia, el término *superior* nunca sonó tan impresionante como *orientado a objetos*.

Como un ejemplo del diseño orientado a objetos, consideremos la tarea de escribir un programa para administrar una cuenta de cobros con menú interactivo (Riley 87). Los objetos de datos importantes son el saldo actual, el monto del cobro y el monto del pago. Es posible definir varias operaciones para actuar en los objetos de datos; estas operaciones serían añadir cobro, hacer pago y añadir interés mensual. Una vez que todos los objetos de datos, la operaciones y la interfaz del menú están definidos, puede empezar la codificación. Esta metodología de diseño orientado a objetos se adapta de manera adecuada a un programa con una estructura de control débil. No sería tan adaptable en un programa que requiere una estructura de control fuerte, como una aplicación de nómina de pago. Para el caso de la nómina de pago hay una secuencia definida de pasos a seguir.

1. Obtener los datos de la tarjeta checadora
2. Contabilizar las incapacidades y las vacaciones
3. Multiplicar las horas trabajadas por la tasa de pago
4. Multiplicar las horas de tiempo extra por la tasa de tiempo extra
5. Añadir bonos o comisiones de venta, si son aplicables
6. Sustraer las deducciones por impuestos, seguro, cuotas y retiro
7. Emitir cheque de pago

El diseño orientado a objetos no requiere funciones especiales de lenguaje, puede hacerse en FORTRAN, BÁSICO, C, etcétera. Los lenguajes como Ada y Modula-2 apoyan el diseño orientado a objetos porque estos lenguajes cuentan con funciones para encapsular los datos en módulos y paquetes.

El término ***programación orientada a objetos*** se usó por primera vez para referirse a lenguajes como Smalltalk, que estaba diseñado de manera específica para objetos. En la actualidad, el término se usa con frecuencia para referirse a la programación de un diseño orientado a objetos, incluso en un lenguaje que no tiene soporte de objetos verdadero.

Smalltalk tiene funciones para soportar objetos construidos dentro del lenguaje, de hecho, tiene un ambiente de programación construido en su totalidad mediante objetos. Smalltalk desciende de SIMULA 67, un lenguaje desarrollado para simulación que introdujo el concepto de **clase**, que lleva al concepto de información oculta en módulos. Una clase es en esencia un tipo, que es una plantilla que define la estructura de los datos. En realidad, los conceptos de tipo en Pascal y Modula-2 viene del de clase. Un **caso** de una clase es un objeto de datos que puede manipularse; este término ha pasado a los sistemas expertos, donde denota un hecho que corresponde a un patrón. Del mismo modo, se dice que una regla ha sido requerida si su lado izquierdo está satisfecho. En los sistemas basados en reglas, los términos *activada* y *requerida* son sinónimos.

Otro concepto significativo que proviene de SIMULA 67 es el de **herencia**, en este sistema podía definirse una **subclase** para heredar las propiedades de las clases. Por ejemplo, una clase puede estar definida para componerse de objetos que pueden usarse en una pila y otra clase definida para integrarse de números complejos. Una subclase puede definirse con facilidad como objetos que son números complejos usados en una pila. Es decir, estos objetos han heredado propiedades de las dos clases superiores, llamadas **superclases**. El concepto de herencia puede extenderse para organizar los objetos en una jerarquía en la que los objetos puedan heredarse de sus clases, que a su vez puedan heredarse de sus clases y así sucesivamente. La herencia es útil porque los objetos pueden heredar propiedades de sus clases sin que el programador tenga que especificar cada propiedad. Muchas de las herramientas de los sistemas expertos que se usan hoy en día, como ART y KEE, permiten la herencia porque es una herramienta versátil en la construcción de grandes grupos de hechos.

Programación lógica

Una de las primeras aplicaciones de la inteligencia artificial en las computadoras, fue la de probar teoremas lógicos con el programa Logic Theorist de Newell y Simon. Se informó por primera vez sobre este programa en la conferencia de Dartmouth, sobre AI en 1956 y causó sensación porque las computadoras electrónicas sólo se habían usado antes para cálculos numéricos. Ahora una computadora estaba razonando de verdad las pruebas de los teoremas matemáticos, una tarea que se pensaba que sólo los matemáticos eran capaces de realizar.

En el Logic Theorist y su sucesor, el General Problem Solver (GPS), Newell y Simon se concentraron en tratar de implementar algoritmos avanzados que pudieran resolver cualquier problema. Mientras que el Logic Theorist estaba pensado sólo para probar teoremas matemáticos, el GPS estaba diseñado para resolver cualquier clase de problema lógico, incluyendo juegos y acertijos como los del ajedrez, la Torre de Hanoi, Misioneros y caníbales y criptaritmético. Un ejemplo de sus famosos acertijos criptaritméticos (aritmética secreta) es

```
    DONALD
+   GERALD
    ROBERT
```

donde se sabe que D = 5. El objetivo es descubrir los valores aritméticos de las otras letras en el rango de 0 a 9.

El GPS fue el primer programa para solución de problemas que separó con claridad el conocimiento para resolver problemas del dominio de conocimiento. Este paradigma de separar en forma explícita el conocimiento para resolver problemas del dominio de conocimiento se usa

hoy en día como la base de los sistemas expertos. En los sistemas expertos actuales, el mecanismo de inferencia decide cuál conocimiento debe usarse y cómo debe aplicarse.

Los esfuerzos para mejorar la comprobación de los teoremas mecánicos continuaron. A principios de los setenta, se descubrió que la computación es un caso especial de deducción mecánica lógica (Kowalski 1985). Cuando el encadenamiento hacia atrás se aplicó a enunciados con la forma "conclusión si condiciones", éste fue bastante eficaz para la comprobación significativa de teoremas. Las condiciones pueden considerarse como patrones representativos que serán igualados, como en las reglas de producción que ya se analizaron. A los enunciados expresados en esta forma, se les llama cláusulas de Horn en honor a Alfred Horn, quien fue el primero en investigarlas. En 1972, Kowalski, Colmerauer y Roussel crearon el lenguaje PROLOG, para implementar la programación lógica por encadenamiento hacia atrás, usando las cláusulas de Horn.

El encadenamiento hacia atrás puede usarse para representar el conocimiento en una representación declarativa y para controlar el proceso de razonamiento; por lo general, procede definiendo **submetas** más pequeñas que deben satisfacerse si la meta inicial va a satisfacerse. Estas submetas a su vez se dividen en submetas más pequeñas y así sucesivamente.

Un ejemplo de conocimiento declarativo es el siguiente ejemplo clásico:

```
Todos los hombres son mortales
Sócrates es un hombre
```

lo que podría expresarse en cláusulas de Horn:

```
alguien es mortal
   si alguien es un hombre
Sócrates es un hombre
   si (en todos los casos)
```

En el enunciado sobre Sócrates, la condición *si* es verdadera en todos los casos. En otras palabras, el conocimiento sobre Sócrates no requiere correspondencia con ningún patrón. Comparando esto con el caso mortal en el que alguien debe ser un hombre para satisfacer el patrón de la condición *si*, nótese que la cláusula de Horn puede interpretarse como un procedimiento que indica cómo satisfacer un objetivo. Esto quiere decir que para determinar si alguien es mortal es necesario determinar si alguien es un hombre. A continuación se muestra un ejemplo algo más complejo:

```
Un automóvil necesita gasolina, aceite y  las llantas
infladas para correr
```

que puede expresarse en una cláusula de Horn como:

```
x es un automóvil y corre
   si x tiene gasolina y
   si x tiene aceite y
   si x tiene llantas infladas
```

Nótese que el problema de determinar si un automóvil correrá ha sido reducido a tres subproblemas o submetas más simples. Ahora supongamos que hay algún conocimiento declarativo adicional como el siguiente:

```
El medidor de combustible muestra cargado si un automóvil
tiene gasolina
La varilla del aceite muestra cargado si un automóvil
tiene aceite
```



```
El medidor de presión de aire señala al menos 20 si un
   automóvil tiene las llantas infladas
El medidor de combustible muestra cargado
La varilla del aceite indica vacío
El medidor de presión de aire muestra cuando menos 15
```

Esto puede traducirse en las siguientes cláusulas de Horn:

```
x  tiene gasolina
   si el medidor de combustible muestra cargado
x  tiene aceite
   si la varilla del aceite muestra cargado
x  tiene las llantas infladas
   si el medidor de presión de aire muestra cuando menos 20
El medidor de combustible no está vacío
   si (en todos los casos)
La varilla del aceite indica vacío
La presión de aire es al menos de 15
   si (en todos los casos)
```

A partir de estas cláusulas, un comprobador de teoremas mecánicos puede probar que el automóvil no correrá porque no hay aceite y la presión de aire es insuficiente.

Una de las ventajas de los sistemas de encadenamiento hacia atrás es que su ejecución puede proceder en paralelo. Esto quiere decir que si están disponibles varios procesadores, podrían trabajar de manera simultánen en la satisfacción de submetas. Esta operación paralela acelera en gran medida la ejecución de los programas y es una de las razones por las que los japoneses escogieron PROLOG como el lenguaje de programación para su proyecto de Computadoras de quinta generación (Moto-Oka 1982). Éste fue un proyecto ambicioso para desarrollar la siguiente generación de computadoras para los noventa, con funciones que sobrepasaran con mucho las de los modelos actuales. Estas máquinas iban a programarse con técnicas declarativas, pero este método ha tenido dificultades. Los japoneses anunciaron el proyecto de Quinta generación en 1981, con un presupuesto de 850 millones de dólares; las 26 metas iniciales del proyecto incluían la traducción de lenguaje natural del japonés a inglés y viceversa, la solución de problemas y la inferencia mil veces más rápida que las máquinas actuales; además de las funciones para leer texto manuscrito, entender imágenes, introducir enormes bases de datos y conversar en lenguaje natural.

PROLOG es más que sólo un lenguaje. A un nivel primario, PROLOG es un shell porque requiere

- un intérprete o mecanismo de inferencia
- una base de datos (hechos y reglas)
- un formulario de correspondencia de patrones llamado **unificación**
- un mecanismo de **rastreo hacia atrás** para perseguir submetas alternas, si fracasa la búsqueda para satisfacer una meta.

Versiones más elaboradas de PROLOG, como TurboPROLOG de Borland, son herramientas debido a todas las mejoras que tienen.

Como ejemplo del encadenamiento hacia atrás, supongamos que si tiene cambio o una tarjeta de crédito, puede pagar por el aceite para hacer que el automóvil corra, una submeta es revisar si tiene cambio. Si no existe el hecho de traer cambio, el mecanismo de encadenamiento hacia atrás explorará la otra submeta para ver si trae una tarjeta de crédito; si es así, la

meta de pagar por el aceite puede satisfacerse. Nótese que la ausencia de un hecho para comprobar una meta es tan efectivo, aunque quizá menos eficiente, como un hecho negativo como "la varilla del aceite indica vacío". Tanto los hechos negativos como los faltantes pueden provocar que no se satisfaga una meta.

Si el encadenamiento hacia atrás y los mecanismos de correspondencia de patrones no son necesarios para el problema, entonces el programador puede trabajar en torno a ellos o codificar en un lenguaje diferente. Una de las ventajas de la programación lógica son las especificaciones ejecutables. Es decir, especificar los requerimientos de un problema con cláusulas Horn produce un programa ejecutable, diferente a la programación convencional, donde el documento de necesidades no tiene un aspecto del todo parecido al código ejecutable final. Los lenguajes imperativos más recientes, como Ada, intentan evitar este problema ofreciendo la capacidad de usar Ada mismo como el lenguaje de descripción del programa (PDL). El PDL puede usarse como un esqueleto sobre el cual construir el resto del programa, pero si el PDL no trabaja correctamente, tampoco lo hará el programa.

A diferencia de los sistemas de reglas de producción, el orden en que se introducen las submetas, los hechos y las reglas en un programa PROLOG tiene efectos significativos. La eficiencia (y por consiguiente la velocidad) se ven afectadas por la manera en que PROLOG busca en su base de datos, sin embargo, hay algunos programas que se ejecutan correctamente si las submetas, los hechos y las reglas se introducen de una manera, pero van en un ciclo infinito o tienen un mensaje de error de tiempo de ejecución si la orden cambia (Ghezzi 87, pp. 304-306).

Sistemas expertos

Los sistemas expertos pueden considerarse programación declarativa porque el programador no especifica cómo es que el programa va a alcanzar su objetivo al nivel de un algoritmo. Por ejemplo, en un sistema experto basado en reglas, cualquier regla puede activarse y pasar a la agenda si su lado izquierdo corresponde a los hechos. El orden en que las reglas se introdujeron no afecta cuáles reglas se activan, de este modo, la orden de enunciado del programa no especifica un flujo de control rígido. Otros tipos de sistemas expertos se basan en **marcos**, que se analizarán en el capítulo 2 , y **redes de inferencia**, que se comentan en el capítulo 4.

Hay muchas diferencias entre los sistemas expertos y los programas convencionales. En la tabla 1.3 se muestran algunas de las diferencias típicas.

Característica	Programa convencional	Sistema experto
Controlado por...	Orden de enunciado	Mecanismo de inferencia
Control y datos	Integración implícita	Separación explícita
Fuerza de control	Fuerte	Débil
Solución por...	Algoritmo	Reglas e inferencia
Búsqueda de solución	Pequeña o nula	Grande
Solución del problema	El algoritmo es correcto	Reglas
Entrada	Se asume como correcta	Incompleta, incorrecta
Entrada inesperada	Difícil de tratar	Gran capacidad de respuesta
Salida	Siempre correcta	Varía con el problema
Explicación	Ninguna	De manera habitual
Aplicaciones	Numérica, archivo y texto	Razonamiento simbólico
Ejecución	Por lo general, secuencial	Reglas oportunistas
Diseño del programa	Diseño estructurado	Poca o ninguna estructura
Modificable	Con dificultad	Razonable
Expansión	Hecha a grandes saltos	Creciente

Tabla 1.13
Algunas diferencias comunes entre los programas convencionales y los sistemas expertos

Los sistemas expertos a menudo están diseñados para tratar con la incertidumbre debido a que el razonamiento es una de las mejores herramientas que hemos descubierto para hacerlo. La incertidumbre puede surgir en los datos que se introducen al sistema experto e incluso en la base de conocimiento misma; al principio esto puede parecer sorprendente a la gente acostumbrada a la programación convencional. Sin embargo, gran parte del conocimiento humano es heurístico, lo que significa que sólo puede trabajar de manera adecuada parte del tiempo. Además, los datos de entrada pueden ser incorrectos, estar incompletos, resultar inconsistentes, o tener otros errores. Las soluciones algorítmicas no son capaces de tratar con estas situaciones porque un algoritmo garantiza la solución de un problema en una serie finita de pasos.

Dependiendo de los datos de entrada y de la base de conocimiento, el sistema experto puede llegar a la respuesta correcta, a una buena respuesta, a una mala respuesta o a una falta de respuesta. Aunque esto pudiera parecer perturbador al principio, la alternativa sería no tener respuesta en ningún caso. De nuevo, lo importante, y que debe tenerse en mente es que un sistema experto no actuará peor que el mejor para resolver este problema (un especialista humano), y quizá lo hará mejor. Si conociéramos un método algorítmico que fuera superior a un sistema experto, lo usaríamos. Lo importante es usar la mejor, y quizá única, herramienta para el trabajo.

Programación no declarativa

Los paradigmas no declarativos se están volviendo populares. Estos nuevos paradigmas se utilizan cada vez más para una amplia variedad de aplicaciones. Pueden usarse en aplicaciones individuales o junto con otros paradigmas.

Programación basada en la inducción

Una aplicación de la AI que está despertando interés es la programación **basada en inducción**, en este paradigma el programa aprende a partir de ejemplos. Una aplicación a la que se ha aplicado este paradigma es el acceso a bases de datos; en lugar de que el usuario tenga que escribir los valores específicos de uno o más campos para una búsqueda, basta con seleccionar de los ejemplos apropiados uno o más campos con esas características. El programa infiere las características de los datos y busca su correspondencia en la base de datos.

Las herramientas para los sistemas expertos, como 1st-Class y KDS, ofrecen aprendizaje por inducción, por lo que aceptan ejemplos y estudios de caso, y generan reglas automáticamente.

1.13 SISTEMAS NEURONALES ARTIFICIALES

Durante los ochenta surgió un nuevo desarrollo en la programación de paradigmas, los **sistemas neuronales artificiales (ANS, artificial neural systems)**, basados en la forma en que el cerebro procesa la información. A este paradigma a veces se le llama conexionismo, porque modela las soluciones a los problemas entrenando neuronas simuladas, conectadas en una red de trabajo. Muchos investigadores están estudiando las redes neuronales, pues las redes mantienen un gran potencial como frente de los sistemas expertos que requieren cantidades masivas de entrada mediante sensores, además de respuestas en tiempo real.

El problema del agente viajero

ANS ha tenido un éxito notable al proporcionar respuestas en tiempo real a complejos problemas de reconocimiento de patrones. En una ocasión, una red neuronal ejecutándose en una microcomputadora ordinaria obtuvo una muy buena solución al problema del agente viajero

en 0.1 segundos, comparada con la solución óptima que requiere una hora de tiempo de CPU en un mainframe (Port 86). El problema del agente viajero es importante porque se trata del problema clásico que se enfrenta con optimizar la ruta de señales en un sistema de telecomunicaciones. Optimizar la ruta es importante para minimizar el tiempo de viaje y, por tanto, la eficiencia y velocidad.

El problema básico del agente viajero es el de calcular las rutas más cortas a través de una lista de ciudades dada. En la tabla 1.14 se muestran las rutas posibles de una a cuatro ciudades. Observe que el número de rutas es proporcional al factorial del número de ciudades menos uno, (N – 1)!

Número de ciudades	Rutas
1	1
2	1-2-1
3	1-2-3-1
	1-3-2-1
4	1-2-3-4-1
	1-2-4-3-1
	1-3-2-4-1
	1-3-4-2-1
	1-4-2-3-1
	1-4-3-2-1

Tabla 1.14 Rutas del problema del agente viajero

Mientras que hay 9! = 362880 rutas para diez ciudades, hay 29! = 8.8E30 rutas posibles para treinta ciudades. Este problema es un ejemplo clásico de la **explosión combinatoria,** porque el número de rutas posibles se incrementa tan rápido que no hay soluciones prácticas para una cantidad realista de ciudades. Si toma una hora del CPU de un mainframe resolver este problema para 30 ciudades, tomaría 30 horas para 31 ciudades y 330 horas para 32 ciudades. Estos son números muy pequeños en realidad cuando se comparan con los miles de interruptores de telecomunicaciones y las ciudades que se utilizan en la asignación de rutas para paquetes de datos y artículos reales.

Una red neuronal puede resolver el caso de 10 ciudades con la misma rapidez que para 30, mientras que a una computadora convencional le toma mucho más. Para el caso de las 10 ciudades, la red neuronal dio con una de las dos mejores rutas y para el caso de las 30 ciudades, dio con una de las 100,000,000 mejores rutas. Esto es más impresionante si uno se da cuenta de que esta solución se encuentra entre las IE-22 mejores soluciones. A pesar de que las redes neuronales no siempre obtienen respuesta óptima, pueden proporcionar la mejor conjetura en tiempo real. En muchos casos, es mejor tener un 99.999999999999999999 por ciento de respuestas correctas en un milisegundo, que una respuesta cien por ciento correcta en 30 horas.

Elementos de un sistema neuronal artificial

Un ANS es básicamente una computadora análoga que usa elementos de procesamiento simples, conectados de una manera altamente paralela. Los elementos de procesamiento ejecutan en sus entradas funciones boleanas o aritméticas simples. La llave para el funcionamiento de un ANS son los pesos asociados con cada elemento, son éstos los que representan la información almacenada en el sistema.

En la figura 1.10 se muestra una neurona artificial típica, puede tener múltiples entradas pero solo una salida. El cerebro humano contiene cerca de 10^{10} neuronas y una neurona puede tener miles de conexiones con otra. Las señales de entrada a la neurona se multiplican por los

pesos y se suman para obtener la entrada total de la neurona, I. Los pesos pueden representarse como una matriz e identificarse con subíndices.

$$I \equiv \text{Entrada de la neurona}_i = \sum_j W_{ij} I_j$$

$$\text{Salida de la neurona} = \frac{1}{1+e^{-(I-\theta)}}$$

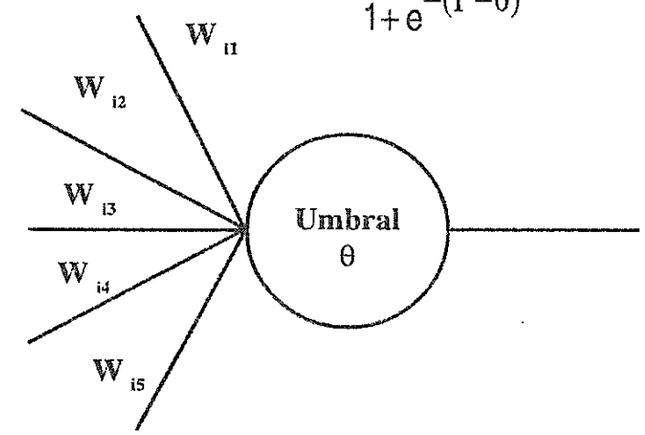


Figura 1.10 Elemento de proceso de una neurona

A menudo, la salida de la neurona se toma como una **función sigmoidea** de la entrada. La sigmoidea es representativa de las neuronas reales, que se acercan a los límites para entradas muy pequeñas o muy grandes, y se le llama ***función de activación***, y una función de uso muy común es $(1 + e^{-x})^{-1}$. Cada neurona tiene también un **valor de entrada** asociado, θ, que se resta a la entrada total, I. En la figura 1.11 se muestra un ANS que puede calcular la **O exclusiva (XOR)** de sus entradas, usando una técnica llamada propagación hacia atrás. La XOR sólo da una salida verdadera cuando sus entradas no son todas verdaderas ni todas falsas. El número de nodos en el estrato escondido variará dependiendo de la aplicación y el diseño.

Las redes neuronales no están programadas en el sentido convencional: Se conocen cerca de 13 algoritmos de aprendizaje para las redes neuronales, como ***propagación de contador*** y propagación hacia atrás para entrenar redes. El programador "programa" la red con sólo sustituir la entrada y los datos de salida correspondientes. La red aprende ajustando automáticamente los pesos en la red de trabajo que conecta las neuronas. Los pesos y los valores de umbral de las neuronas determinan la propagación de los datos a través de la red y, por tanto, su respuesta correcta a los datos de entrenamiento. Entrenar la red para que obtenga las respuestas correctas puede tomar horas o días, dependiendo del número de patrones que la red deba aprender, el hardware y el software. Sin embargo, una vez que se realiza el aprendizaje, la red responde con rapidez.

Si la simulación del software no es lo bastante rápida, ANS puede fabricarse en chips para que responda en tiempo real; una vez que se ha entrenado a la red de trabajo y que se han determinado los pesos, se puede construir un chip. AT&T y otras compañías están fabricando chips experimentales para redes neuronales que contienen cientos de neuronas, es muy pro-

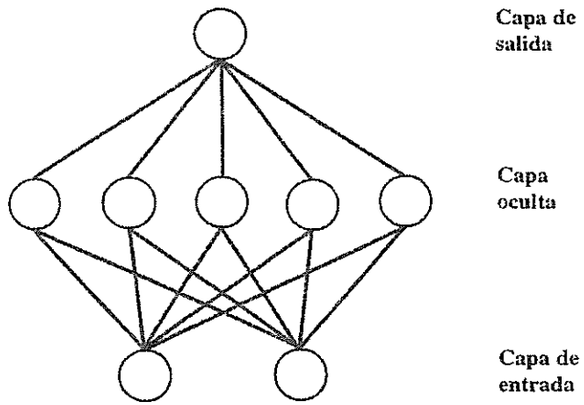


Figura 1.11 Red de propagación hacia atrás

Características de los Sistemas Neuronales Artificiales

La arquitectura de los ANS es muy diferente de la arquitectura de las computadoras convencionales. En una computadora convencional es posible correlacionar información discontinua con las celdas de memoria. Por ejemplo, un número de seguro social puede almacenarse como código ASCII en un grupo contiguo de celdas de memoria, al examinar el contenido de este grupo contiguo, puede reconstruirse directamente el número de seguro social; esta reconstrucción es posible porque hay una relación de uno a uno entre cada carácter del número de seguro social y la celda de memoria que contiene el código ASCII de ese carácter.

Los ANS están modelados a partir de las teorías actuales del cerebro, donde la información se representa por los pesos. Sin embargo, no existe correlación directa entre un peso y un artículo específicos de la información almacenada. Esta representación distributiva de la información es similar a la de un holograma, en el que las líneas actúan como una rejilla de difracción para reconstruir la imagen almacenada cuando la luz del láser la atraviesa.

Una red neuronal es una buena opción cuando hay una cantidad considerable de datos empíricos y no hay un algoritmo que proporcione velocidad y precisión suficientes. Los ANS tienen muchas ventajas en comparación con el almacenamiento de las computadoras convencionales.

- *El almacenamiento es tolerante a fallas.* Pueden eliminarse partes de la red y sólo hay una degradación en la calidad de los datos almacenados. Esto ocurre porque la información está almacenada de manera distribuida.
- *La calidad de la imagen almacenada se degrada poco a poco en proporción a la cantidad de red que se elimina.* No hay pérdida catastrófica de la información. El almacenamiento y la calidad de las funciones también son características de los hologramas.
- *Los datos se almacenan con naturalidad en forma de memoria asociativa.* Una memoria asociativa es aquella en la que datos parciales bastan para recordar la información completa almacenada. Esto difiere de la memoria convencional, en la que los datos se recuerdan especificando su dirección. Una entrada parcial o con ruido puede seguir produciendo la información original completa.
- *Las redes pueden extrapolarse e interpolarse a partir de su información almacenada.* El entrenamiento enseña a una red a buscar rasgos significativos o relaciones entre los datos; después la red puede extrapolar para sugerir relaciones con nuevos datos. En un experimento (Hinton 86), se entrenó a una red neuronal sobre las relaciones familiares de 24 personas hipotéticas; luego la red también pudo responder correctamente acerca de relaciones sobre las que no había sido entrenada.
- *Las redes tienen **plasticidad**.* Aunque se eliminen varias neuronas, la red puede volver a entrenarse a su nivel original de habilidad, si quedan las neuronas suficientes. Ésta es también una característica del cerebro, en el que, si se destruyen algunas partes, pueden reaprenderse los niveles originales de habilidad.

Estas características hacen muy atractivos a los ANS para las naves espaciales robotizadas, equipos de campos petroleros, dispositivos submarinos, controles de procesamiento y otras aplicaciones que necesitan funcionar largo tiempo en ambientes hostiles sin reparaciones. Además de la confiabilidad, los ANS ofrecen la posibilidad de tener bajos costos de mantenimiento debido a su plasticidad. Aunque se pueden hacer reparaciones de hardware, probablemente sea más rentable reprogramar la red neuronal en lugar de reemplazarla.

Los ANS no suelen ser adecuados para aplicaciones que requieren procesamiento de números o una solución óptima. Además, si existe una solución algorítmica práctica, un ANS no es una buena elección.

Desarrollo de la tecnología de los ANS

Los orígenes de ANS se encuentran en el modelo matemático de las neuronas hecho por McCulloch y Pitts en 1943 (McCulloch 43). Hebb explicó en 1949 cómo aprenden las neuronas (Herb 49). En el aprendizaje hebbiano, la eficiencia de una neurona para estimular a otra se incrementa con el **disparo**. El término *disparo* significa que una neurona emite un impulso electroquímico que puede estimular a otras neuronas conectadas a ella. Hay evidencia de que la conductividad de las conexiones entre neuronas, llamada **sinapsis**, se incrementa con los disparos. En los ANS, el peso de las conexiones entre neuronas cambia para simular la conductividad cambiante de las neuronas naturales.

En 1961, Rosenblatt publicó un libro de gran influencia que trataba con el nuevo tipo de sistema de neuronas artificiales que investigaba, llamado **perceptrón** (Rosenblatt 61). El perceptrón era un dispositivo notable que mostraba aptitudes para el aprendizaje y reconocimiento de patrones; consistía básicamente de dos capas de neuronas y un algoritmo de aprendizaje simple. Los pesos tenían que asignarse manualmente, en contraste con los modernos ANS, que asignan los pesos por sí solos, basados en su entrenamiento. Muchos investigadores entraron al campo de los ANS y comenzaron a estudiar los perceptrones durante los sesenta.

La era inicial del perceptrón terminó en 1960, cuando Minsky y Papert publicaron un libro, *Perceptrons*, que mostraba las limitaciones teóricas de los perceptrones como máquinas de cálculo general (Minsky 69). Señalaron una deficiencia del perceptrón: sólo era capaz de calcular 14 de las 16 funciones lógicas básicas, lo que significa que un perceptrón no es un dispositivo de cálculo para propósitos generales; en particular, probaron que un perceptrón no puede reconocer el O exclusivo. Aunque no habían investigado seriamente los ANS de múltiples capas, expresaron la visión pesimista de que las múltiples capas probablemente no lograrían resolver el problema XOR. Los fondos gubernamentales para la investigación de

los ANS pasaron a favorecer al método simbólico de AI utilizando lenguajes como LISP y algoritmos. Durante los años setenta, se popularizaron nuevos métodos para representar información simbólica en la AI a través de marcos, inventados por Minsky. Se ha seguido trabajando en los perceptrones, y nuevos tipos de éstos han logrado superar las objeciones de Minsky (Reece 87). Debido a su simplicidad, es fácil construir perceptrones y otros sistemas ANS con la tecnología moderna de circuitos integrados.

La investigación de los ANS continuó en pequeña escala durante los setenta. Sin embargo, el campo entró finalmente en un renacimiento que se inició con el trabajo de Hopfield en 1982 (Hopfield 82). Él dio a los ANS una firme base teórica con la red Hopfield de dos capas, y demostró cómo los ANS pueden resolver una amplia variedad de problemas. La estructura general de una red Hopfield se muestra en la figura 1.12. En particular, Hopfield mostró cómo un ANS podía resolver el problema del agente viajero en un tiempo constante, comparado con la explosión combinatoria encontrada por las soluciones algorítmicas convencionales. Una forma de circuito electrónico de un ANS pudo resolver el problema en un microsegundo. Los ANS pueden resolver fácilmente otros problemas de optimización combinatoria, como el mapa de los cuatro colores, la correspondencia euclidiana (Hopfield 86b) y el código de transposición (Tank 85).

Un ANS que puede resolver fácilmente el problema del XOR es la red de **propagación hacia atrás**, también conocida como la **regla delta generalizada** (Rumelhart 86). La red de propagación hacia atrás suele implantarse como una red de tres capas, aunque pueden especificarse capas adicionales. A las capas que se encuentran entre las capas de entrada y salida se les llama **capas ocultas**, porque sólo las capas de entrada y de salida están visibles para el mundo exterior. Otro tipo popular de ANS es el de propagación de conteo, inventado por Hetch-Nielsen en 1986 (Hetch-Nielsen 90). Un importante resultado teórico de las matemáticas, el teorema de Kolmogorov, puede interpretarse como prueba de que una red de trabajo con tres capas, n entradas y $2n + 1$ neuronas en la capa oculta, puede crear un mapa de cualquier función continua (Hetch-Nielsen 87).

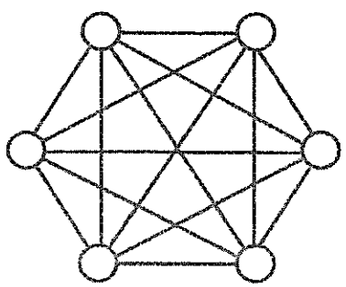


Figura 1.12 Red artificial neuronal de Hopfield

Aplicaciones de la tecnología de ANS

Un ejemplo significativo del aprendizaje por propagación hacia atrás se demostró con una red neuronal que aprendió la pronunciación correcta de palabras tomadas de un texto (Sejnowski 86). El ANS fue entrenado corrigiendo su salida con el uso de un dispositivo de texto a pronunciación de DEC llamado DECTalk. Se requirieron 20 años de investigación lingüística para determinar las reglas para pronunciación correcta que utilizó DECTalk. El ANS aprendió las habilidades de pronunciación equivalentes en una sola noche, escuchando únicamente la pronunciación correcta del habla a partir del texto; no se programó ninguna habilidad lingüística en el ANS.

Existen investigaciones de los ANS encaminadas al reconocimiento de blancos de radar, a través de computadoras electrónicas y ópticas (Farhat 86). Las nuevas implantaciones de redes neuronales, utilizando componentes ópticos, prometen computadoras ópticas con velocidades millones de veces superiores a las electrónicas. La implementación óptica de los ANS es atractiva a causa del paralelismo inherente de la luz. Es decir, los rayos de luz no interfieren entre sí mientras viajan; es posible generar y manipular fácilmente grandes cantidades de fotones con componentes ópticos como espejos, lentes, moduladores de luz espacial de alta velocidad programables, series de dispositivos ópticos biestables que pueden funcionar como neuronas ópticas, y rejillas de difracción. Las computadoras ópticas diseñadas como ANS parecen complementarse; los avances futuros de los ANS parecen promisorios (Giarratano 90 a). En particular, los ANS son útiles para controlar sistemas en los que las propuestas convencionales no resultan satisfactorias (Giarratano 91 b).

Desarrollos comerciales de los ANS

Varias nuevas compañías y otros ya existentes se han organizado para desarrollar tecnología y productos para los ANS. Nestor comercializa un producto de ANS llamado NestorWriter, que puede reconocer entradas manuscritas y convertirlas en texto usando una PC. Otras compañías como TRW, SAIC, HNC, Synaptics, Neural Tech, Revelations Research y Texas Instruments, comercializan diversos simuladores ANS y tarjetas aceleradoras de hardware para acelerar el aprendizaje. Una de las mejores gangas para entrar en la computación neuronal es el volumen tres de las obras de Rumelhart sobre procesamiento distribuido en paralelo (editorial MIT Press). El libro describe media docena de simuladores ANS y también incluye un disquete con software para una máquina compatible con IBM-PC por $27.50. Adquiriendo una tarjeta aceleradora, el usuario puede tener un potente sistema para ANS a bajo costo.

1.14 SISTEMAS EXPERTOS INTERCONECTADOS Y APRENDIZAJE INDUCTIVO

Es posible construir sistemas expertos usando los ANS. En un sistema, el ANS es la base de conocimiento, construida mediante ejemplos de entrenamiento, tomados de la medicina (Gallant 88). En este caso, los sistemas expertos tratan de clasificar una enfermedad, por medio de sus síntomas, dentro de una de las enfermedades conocidas en que el sistema ha sido entrenado. Un mecanismo de inferencia conocido como MACIE (Matrix Controlled Inference Engine: mecanismo de inferencia controlado por matriz) fue diseñado para usar la base de conocimiento ANS. El sistema usa el encadenamiento hacia delante para hacer inferencias y el encadenamiento hacia atrás para solicitar al usuario cualquier dato adicional necesario para llegar a una respuesta. A pesar de que los ANS no pueden explicar por sí solos la razón por la que se le asignaron ciertos valores a sus pesos, MACIE puede interpretar los ANS y generar reglas SI...ENTONCES para explicar su conocimiento.

Un sistema experto de ANS como éste utiliza el **aprendizaje inductivo**; es decir, el sistema **induce** la información en su base de conocimiento a través de ejemplos. La inducción es el proceso de inferir casos generales a partir de casos específicos. Además de los ANS, hay en el mercado varias estructuras de los sistemas expertos que, de manera explícita, generan reglas a partir de ejemplos. La meta del aprendizaje inductivo es reducir o eliminar el cuello de botella que representa la adquisición de conocimiento. Al colocar el esfuerzo de adquisición de conocimiento en el sistema experto, es posible reducir el tiempo de desarrollo y aumentar la confiabilidad si el sistema induce reglas desconocidas para un ser humano. Los sistemas expertos se han combinado con los ANS (Giarratano 90b).

1.15 RESUMEN

En este capítulo se han revisado los problemas y desarrollos que han conducido hacia los sistemas expertos. Por lo general, con la programación convencional no es posible resolver los problemas para los que se usan los sistemas expertos, porque ésta carece de conocimiento o de un algoritmo eficiente. Como los sistemas expertos están basados en el conocimiento, pueden usarse con efectividad en problemas del mundo real, de estructura nociva y difíciles de resolver por otros medios.

También se analizaron las ventajas y desventajas de los sistemas expertos, en el contexto de la selección de un dominio de problema apropiado para su aplicación y se describieron los criterios para seleccionar las aplicaciones apropiadas.

Se analizaron las bases de una shell para los sistemas expertos con referencia a los basados en reglas. Se describió el ciclo básico de mecanismo de inferencia acto reconocimiento y se ilustró con un ejemplo de regla simple. Finalmente, se describió la relación entre los sistemas expertos y otros paradigmas de programación a partir del dominio apropiado de cada paradigma. Lo importante es el concepto de que los sistemas expertos deben ser vistos como otra herramienta de programación que resulta adecuada para algunas aplicaciones y no los es para otras. En capítulos posteriores se describirán con más detalle las características y la adaptabilidad de los sistemas expertos.

La mejor fuente de información actual acerca de la inteligencia artificial es el grupo de noticias comp.ai de Internet y Usenet. Allí está disponible gran cantidad de información y software. Otros grupos de noticias están interesados en lógica confusa, redes neuronales, estructuras de sistema experto y muchos otros temas. Además de la información y el software, estos grupos de noticias permiten a los usuarios enviar preguntas y recibir respuestas de otros.

PROBLEMAS

1.1 Identifique a una persona, aparte de usted, que esté considerada como un especialista o como alguien con mucho conocimiento. Entrevístela y analice qué tan bien podría modelar un sistema experto la experiencia de esta persona, a partir de cada criterio de la sección "Ventajas de los sistemas expertos", de las páginas 4 y 5.

1.2 a) Escriba diez reglas no triviales que expresen el conocimiento del especialista del problema 1.
b) Escriba un programa que le dará consejo experto. Incluya prueba de resultados para demostrar que cada una de las diez reglas ofrece el consejo correcto. Para facilitar la programación usted puede permitir que el usuario proporcione las entradas en un menu.

1.3 a) En la obra de Newell y Simon, *Human Problem Solving*, mencionan el problema de los nueve puntos. Si se dan nueve puntos organizados como se muestra, ¿Cómo dibujaría usted cuatro líneas que unan todos los puntos sin a) separar su lápiz del papel y b) sin pasar dos veces por algún punto. (Sugerencia: puede extender la línea más allá de los puntos.)
...
...
...

b) Explique su razonamiento (si lo hay) para encontrar la solución y analice si un sistema experto o algún otro tipo de programa podría ser un buen paradigma para resolver este problema.

1.4 Escriba un programa que pueda resolver problemas criptoaritméticos. Muestre el resultado para el siguiente problema, en el que D = 5.

```
  DONALD
+ GERALD
  ROBERT
```

1.5 Escriba un conjunto de reglas de producción para extinguir cinco tipos diferentes de fuego, como el provocado por petróleo, productos químicos, y otros, dando el tipo de fuego.

1.6 a) Escriba un conjunto de reglas de producción para diagnosticar tres tipos de veneno, basándose en los síntomas.
b) Modifique el programa para que éste recomiende un tratamiento, una vez que se ha identificado el veneno.

1.7 Escriba un informe sobre tres diferentes aplicaciones de los sistemas expertos que no se hayan analizado en este capítulo. Incluya sus fuentes de información.

1.8 Elabore diez reglas heurísticas del tipo SI...ENTONCES que le ahorren tiempo.

1.9 Elabore diez reglas heurísticas del tipo SI...ENTONCES para comprar un automóvil usado.

1.10 Elabore diez reglas heurísticas del tipo SI...ENTONCES para planear su horario de clases.

1.11 Elabore diez reglas heurísticas del tipo SI...ENTONCES para comprar una motocicleta.

1.12 Elabore diez reglas heurísticas del tipo SI...ENTONCES para comprar un vehículo de tres ruedas.

1.13 Elabore diez reglas heurísticas del tipo SI...ENTONCES para dar excusas por un trabajo no entregado a tiempo.

1.14 Escriba un informe en XCON, XSEL y todos los otros sistemas expertos para ventas y manufactura usados por Digital Equipment Corp. En particular, analice cómo es que los sistemas expertos cooperan para trabajar juntos. Use los dos artículos más recientes como referencia e incluya una copia de los artículos con su informe.

1.15 Responda las siguientes preguntas acerca de uno de los sistemas expertos clásicos a los que se hizo referencia en la bibliografía histórica, buscando por lo menos tres textos apropiados u otras referencias. Incluya una copia de todas sus referencias con sus respuestas.

a) ¿Cuáles fueron los propósitos de su desarrollo?
b) ¿Qué tanto éxito tuvo para propósitos de investigación?
c) ¿Tuvo alguna vez éxito comercial o se usó rutinariamente?
d) ¿Qué otros sistemas expertos, si los hubo, se debieron a éste?
e) ¿Cuáles nuevos conceptos desarrolló este sistema?

BIBLIOGRAFÍA

(Buchanan 78). Buchanan, B.G. y Mitchell, T., "Model-directed Learning of Production Rules." Waterman, D. A., and Hayes-Roth, F., eds., *Pattern Directed Inference Systems*, Academic Press, pp 297-312, 1978.

(Buchanan 78). Bruce G. Buchanan y Edward A. Feigenbaum, "Dendral and Meta-Dendral: Their Applications Dimension." *Artificial Intelligence*, 11, (1), pp. 5-24, 1978.

(Chase 73). W. G. Chase y Herbert A. Simon, "The Mind's Eye in Chess", *Visual Information Processing*. W.G. Chase, de., Academic Press, 1973.

(Elson 73). Mark Elson, *Concepts of Programming Languages*, Science Research Associates, p. 290, 1973.

(Ennis 86). R.L. Ennis, *et al.*, "A Continuos Real-time Expert System for Computer Operations", *IBM J. Res. Develop.*, **30**, (1), pp. 14-28, 1986.

(Farhat 86). N. H. Farhat, *et al.*, "Optical Analog of Two-Dimensional Neutral Networks and Their Application in Recognition of Radar Targets", *Procceedings of the Neutral Networks for Computing Meeting*, Snowbird, UT, pp. 146-152, 1986.

(Feigenbaum 81). Edward A. Feigenbaum, *Handbook of Artificial Intelligence*, Heuris Tech Press/ William Kaufman, Inc., 1981-2.

(Feigenbaum 82). Edward A. Feigenbaum, "Knowledge Engineering in the 1980s", Dept. of Computer Science, Stanford University, Stanford, CA, 1982.

(Forgy 82). Charles Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Macht Problem", *Artificial Intelligence*, **19**, pp. 17-37, 1982.

(Gallant 88). Stephen L. Gallant, "Connectionist Expert Systems", *Comm. of the ACM*, **31**, (2), pp. 152-169, Feb. 1988.

(Gersting 82). Judith I. Gerstting, *Mathematical Structures for Computer Science*, W. L. Freeman and Co., pp. 64-67, 1982.

(Ghezzi 87). Carlo Ghezzi y Medih Jazayeri, *Programming Language Concepts*, John Wiley and Sons, Inc., p. 293. 1987.

(Giarratano 90a). Joseph Giarratano, *et al.*, "Future Impacts of Artificial Neutral System on Industry", *ISA Transsactions*, pp. 9-14, enero de 1990.

(Giarratano 90b). Joseph Giarratano, *et al.*, "The State of the Art for Current and Future Expert System Tools", *ISA Transactions*, pp. 17-25, enero de 1990.

(Giarratano 91a). Joseph Giarratano, *et al.*, "An Intelligent SQL Tutor", 1991 Conference on Intelligent Computer-Aided Training (ICAT'91), pp 309-316, 1991.

(Giarratano 91b). Joseph Giarratano, *et al.*, "Neutral Networks Techniques in Manufacturing and Automation Systems", en *Control and Dynamic Systems*, Vol. 49, ed. by C.T. Leondes, Academic Press, pp. 37-98, 1991.

(Harmon 85). Paul Harmon y David King, *Expert Systems*, John Wiley and Sons, Inc., p. 21, 1985.

(Hayes-Roth 85). Federick Hayes-Roth, "Rule-based Systems", *Communications of the ACM*, **28**, (9), pp. 921-932, septiembre de 1985.

(Hebb 49). D.O. Hebb, *Organization of Behavior*, John Wiley and Sons, Inc., 1949.

(Hecht-Nielsen 87). Robet Hecht-Nielsen, "Kolmogorov's Mapping Neutral Networks Existence Theorem", *Proceedings of the First International Conference on Networks*, **III**, pp. 11-14, junio de 1987.

(Hecht-Nielsen 90). Robert Hecht-Nielsen, *Neurocomputing*, Addison-Wesley Pub. Co., p. 147, 1990.

(Hinton 86). Geoffrey E. Hinton, "Learning Distributed Representations of Concepts", *Proceedings 8th Annual Conference of the Cognitive Science Society*. Lawrence Erlbaum Assoc., 1986.

(Hopfield 82). J. J. Hopfield, "Neural Networks and Physical Systems with Emergent Collective Computational Abilities", *Proceedings of the National Academy of Sciences*, USA, **79**, (8), pp. 2554-2558, 1982.

(Hopfield 86a). John J. Hopfield y David W. Tank, "Computing with Neural Circuits: A Model", *Science*, **233**, pp. 625-633, 1986.

(Hofield 86b). John J. Hopfield y David W. Tank, *Disordered Systems and Biological Organization*, Springer-Verlag, 1986.

(Hugh 88). Dafydd Ab Hugh, "The Future of Flying", *AI Expert*, pp. 66-69, Jan. 1988.

(Kerschberg 86). Larry Kerschberg, ed., *Expert Database System: Proceedings from the First International Workshop*, Benjamín/Cummings Publishing Co., 1986.

(Kowalski 1985). Robert Kowalski, "Logic Programming", *Byte*, p. 161, August 1985.

(Markov 54). A. A. Markov, *A Theory of Algorithms*, National Academy of Sciences, USSR (en inglés), 1954.

(McCulloch 43). Waren S. McCulloch y Walter Pitts, "A Logical Calculus of the Ideas Immanent in Nervous Activity", *Bulletin of Mathematical Biophysics*, **5**, pp. 115-137, 1943.

(McDermott 84). John S. McDermott and Judith Bachant, "R1 Revisited: Four Years in the Trenches", *AI Magazine*, **V**, (3), pp. 21-32, otoño de 1984.

(Michie 73). D. Michie, "Knowledge Engineering", *Cybernetics*, **2**, pp. 197-200, 1973.

(Minsky 69). *Perceptrons-An Introduction to Computational Geometry*, MIT Press, 1969. *Nota*: la segunda edición fue publicada con algunas correcciones hechas a mano y con capítulos adicionales.

(Moto-Oka 1982). Moto-Oka, T., ed., *Fifth Generation Computer Sytems*, Elsevier, 1982.

(Newell 72). Allen Newell y Herbert A. Simon, *Human Problem Solving*, Prentice-Hall, 1972.

(Parrello 88). Bruce Parrello, "Car Wars: The (Almost) Birth of an Expert System", *AI Expert*, pp. 60-64, Jan. 1988.

(Post 43). Emil L. Post, "Formal Reductions of the General Combinatorial Decision Problem", *American Journal Of Mathematics*, **65**, pp. 197-215, 1943.

(Reece 87). Peter Reece, "Perceptrons & Neural Nets", *AI Expert*, pp. 50-57, enero de 1987.

(Riley 87). David D. Riley, *Data Abstraction and Structures*, Boyd & Fraser, pp. 305-349, 1987.

(Rosenblatt 61). F. Rosenblatt, *Principles of Neurodynamics*, Sparten Books, 1961.

(Rumelhart 86). David E. Rumelhart, James L. McClelland, *et al.*, "Learning Internal Representations by Error Propagation", *Parrallel Distributed Processing*, **1**, MIT Press, 1986.

(Schur 88). Stephen Schur, "The Intelligent Data Base", *AI Expert*, pp. 26-34, Jan. 1988.

(Scown 85). Susan B. Scown. *The Artificial Intelligence Experience: An Introduction*, Digital Equipment Corp., p. 78, 1985.

(Sejnowski 86). Terence J. Sejnowski y Charles R. Rosenberg, "NETtalk: A Parallel Network that Learns to Read Aloud", *Technical Report JHU/EECS-86/01*, Johns Hopkins University, 1986.

(Tank 85). David W. Tank y John J. Hopfield, *IEEE Circuits Systems*, **CAS-33**, p. 533, 1985.

(Waldrop 87). M. Mitchell Waldrop, *Man-Made Minds*, Walker and Co., p. 35, 1987.

(Waterman 86). Donald A. Waterman, *A Guide to Expert Systems*, Addison-Wesley Pub. Co., pp. 41-48, 1986.

UNA BREVE BIBLIOGRAFÍA HISTÓRICA DE LOS SISTEMAS EXPERTOS

AGE — Ayuda en ingeniería de conocimiento

Nii, H. P., and Aiello, N., "AGE (Attempt to Generalize): A Knowledge-based Program for Building Knowledge-based Programs", IJCAI, pp. 645-655, 1979.

AL/X — Shell de sistema experto basada en MYCIN y PROSPECTOR

Reiter, J., "AL/X: An Expert System Using Plausible Inference", Intelligent Terminals, Ltd., Oxford, 1980.

CASNET — Administración de tratamiento para glaucoma

Weiss, S *et al.*, "A Model-based Method for Computer-aided Medical Decision Making", *Artificial Intelligence*, **11**, pp. 145-172, 1978.

CENTAUR — Análisis de pruebas pulmonares (pulmón)

Aikins, J. S., "Prototypes and Production Rules: An Approach to Knowledge Representation for Hypothesis Formation", IJCAI, pp. 1-3, 1979.

CRYSALIS — Usa datos de los rayos X para determinar la estructura de proteínas

Englemore, R., y Terry, A., "Structure and Funcion of the CRYSALIS System", IJCAI, pp. 250-256, 1979.

DENDRAL — Identifica componentes orgánicos a partir de datos del espectrómetro de masas

Buchanan, B. G. y Feigenbaum, E. A. "DENDRAL and Meta-DENDRAL: Their Applications Dimension", *Artificial Intelligence*, **11**, pp. 5-24, 1978.

EMYCIN — Shell de sistema experto desarrollada a partir de MYCIN

Van Melle, W., "A Domain-independent Production Rule System for Consultation Programs", IJCAI, pp. 923-925, 1979.

GUIDON — Tutor inteligente para sistemas basados en EMYCIN

Clancey, W. J., "Dialogue Management for Rule-based Tutorials", IJCAI, pp. 155-161, 1979.

INTERNIST — Diagnóstico de problemas de medicina interna

Pople, H. E., Myers, J. D. y Miller, R. A., "DIALOG: A Model of Diagnostic for Internal Medicine", IJCAI, pp. 848-855, 1975.

MACSYMA — Resuelve problemas matemáticos de álgebra, trigonometría y cálculo

Genesereth, M. R., "The Role of Plans in Automated Consultation", IJCAI, pp. 311-319, 1979.

META-DENDRAL — Induce reglas de estructura molecular a partir de datos del espectrómetro de masas

Buchanan, B. G., y Mitchell, T., "Model-directed Learning of Production Rules", Waterman, D.A., and Hayes-Roth, F., eds., Pattern Directed Inference Systems, Academic Press, pp. 297-312, 1978.

MOLGEN — Planea experimentos de genética molecular

Marin, N., *et al.*, "Knowledge-based Management for Experiment Planning in Molecular Genetics", IJCAI, pp. 882-887, 1977.

MYCIN — Diagnóstico y tratamiento de meningitis y otras enfermedades de la sangre

Davis, R., Buchanan, B. G., and Shortliffe, E. H., "Production Systems as a Representation for a Knowledge-based Consultation Program", *Artificial Intelligence*, **8**, (1), pp. 15-45, 1977.

PROSPECTOR — Identificación de minerales y selección de sitios de perforación

Duda, R., Gasching, J., y Hart, P., "Model Design in the PROSPECTOR Consultant System for Mineral Exploration", D. Michie, ed., *Expert Systems in the Micro-Electronic Age*, Edinburgh University Press, pp. 153-167, 1979.

PUFF — Analiza pruebas pulmonares (pulmón)

Kunz, J. C. *et al.*, "A Physiological Rule-based System for Interpreting Pulmonary Function Test Rules", Heuristic Programming Memo HPP-78-19, Stanford University, Stanford CA.

XCON/RI — Configura sistemas de computadoras VAX

McDermott, J. Bachant J., "RI Revisited: Four Years in the Trenches", *AI Magazine* V, (3), pp. 21-32, Fall 1984.

SACON — Ingeniería estructural

Bennet, J. S. y Engelmore, R. S., "SACON: A Knowledge-based Consultant for Structural Analysis", IJCAI, pp. 47-49, 1979.

SU/X — Interpretación de señales

Nii, H. P., y Feigenbaum, E. A., "Rule-based Understanding of Signals", Waterman, D. A., and Hayes-Roth, F., eds., *Pattern Directed Inference Systems*, Academic Press, pp. 483-501, 1978.

TEIRESIAS — Programa para la adquisición de conocimientos desarrollado por MYCIN

Davis, R., "Interactive Transfer of Expertise: Acquisition of New Inference Rules", *Artificial Intelligence*, **12**, (2), pp. 121-158, 1979.

CAPÍTULO 2

La representación del conocimiento

2.1 INTRODUCCIÓN

En este capítulo se analizan algunas de las representaciones más comunes del conocimiento para los sistemas expertos. La representación del conocimiento es de gran importancia en estos sistemas por dos razones. En primer lugar, los shells de los sistemas expertos están diseñados para cierto tipo de representación del conocimiento, como las reglas o la lógica. En segundo lugar, la forma en que un sistema experto representa al conocimiento afecta su desarrollo, eficiencia, velocidad y mantenimiento. En el capítulo 3 analizaremos cómo es que se hacen las inferencias.

2.2 EL SIGNIFICADO DEL CONOCIMIENTO

Conocimiento, como *amor*, es una de esas palabras de las que todos sabemos el significado, aunque resulta difícil definirlo: ambas tienen también muchos significados. Otras palabras como *datos*, *hechos* e *información* se utilizan para sustituir *conocimiento*.

La **epistemología** es el estudio del conocimiento (Angeles 81); se ocupa de su naturaleza, su estructura y sus orígenes. En la figura 2.1 se ilustran algunas de las categorías de la epistemología. Además de los tipos de conocimiento filosófico que expresaron Aristóteles, Platón, Descartes, Hume, Kant y otros, hay dos tipos especiales llamados **a priori** y **a posteriori**. El primer término viene del latín y significa "lo que precede". Un conocimiento *a priori* precede y es independiente del conocimiento obtenido por medio de los sentidos; por ejemplo, las frases "todo tiene una causa" y "la suma de los ángulos de todos los triángulos en un plano es de 180 grados". El conocimiento *a priori* se considera verdad universal y no puede negarse sin contradicción. Las frases lógicas, las leyes matemáticas y el conocimiento de los adolescentes son otros ejemplos de conocimiento *a priori*.

Lo opuesto al conocimiento *a priori* es el que se deriva de los sentidos, o conocimiento *a posteriori*. La verdad o falsedad de este conocimiento puede verificarse mediante la experiencia de los sentidos, como en la frase "la luz es verde". Sin embargo, como la experiencia sensorial no siempre es confiable, un conocimiento *a posteriori*, puede rechazarse con base en nuevo conocimiento, sin necesidad de caer en contradicciones. Por ejemplo, si vemos a alguien de ojos café, creeríamos que de ese color son sus ojos. Sin embargo, si más tarde

observamos que esa persona se quita los lentes de contacto de color café y tiene los ojos azules, tendríamos que revisar nuestro conocimiento.

El conocimiento puede clasificarse aún más como **conocimiento por procedimientos, conocimiento declarativo y conocimiento tácito.** Los primeros dos tipos corresponden a los paradigmas de procedimientos y declarativo abordados en el capítulo 1.

A menudo el conocimiento por procedimientos se refiere a la forma en que sabemos hacer algo, un buen ejemplo sería saber cómo hervir agua en un pote. El conocimiento declarativo alude a la capacidad de saber que algo es verdadero o falso, que puede expresarse en forma de frases declarativas como "no toques con los dedos un pote con agua hirviendo".

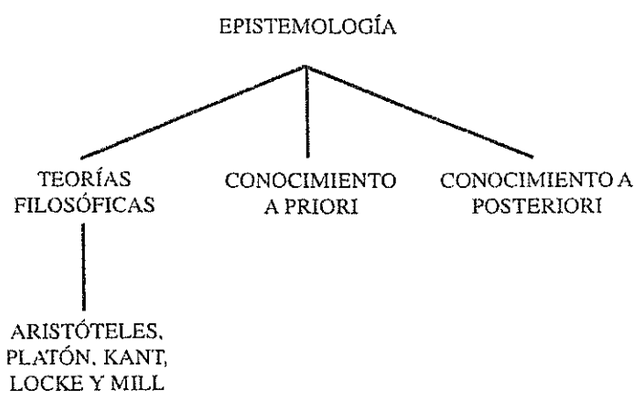


Figura 2.1 Algunas categorías de la epistemología

Al conocimiento tácito en ocasiones se le llama ***conocimiento inconsciente***, porque no puede expresarse mediante el lenguaje. Un ejemplo es saber cómo mover la mano, en gran medida, se podría decir que el movimiento proviene de tensar o relajar ciertos músculos y tendones. Pero en el siguiente nivel inferior, ¿cómo *saber* tensar o relajar los músculos y tendones? Otros ejemplos son caminar o andar en bicicleta. Los sistemas de computación ANS se relacionan con este tipo de conocimiento porque normalmente la red neuronal no puede explicar su conocimiento de manera directa, pero podría hacerlo si se tiene un programa apropiado (consulte la sección 1.14).

El conocimiento es de vital importancia en los sistemas expertos. En efecto, una analogía con la expresión clásica de Wirth

```
Algoritmos + Estructura de datos = Programas
```

para sistemas expertos es

```
Conocimiento + Inferencia = Sistemas expertos
```

En este libro se utiliza el conocimiento como parte de una jerarquía, lo que se ilustra en la figura 2.2. En la base se encuentra el ruido, formado por elementos que son de poco interés y que ocultan datos. El siguiente nivel de la jerarquía son los datos, elementos de interés potencial. En el tercer nivel hay información, o datos procesados que resultan de interés. El siguiente nivel es el conocimiento, que representa información muy especializada. En el capítulo 1, el conocimiento de los sistemas expertos basados en reglas se definió como las reglas activadas por hechos para producir nuevos hechos o conclusiones. Este proceso de *inferencia* es la segunda parte esencial de un sistema experto. El término inferencia se usa generalmente para sistemas mecánicos como los sistemas expertos. Para el pensamiento humano, generalmente se utiliza *razonamiento*.

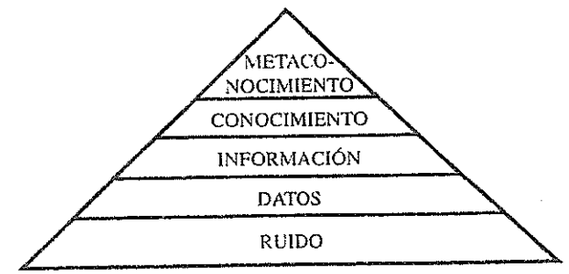


Figura 2.2 La jerarquía del conocimiento

El término *hechos* puede significar datos o información. Dependiendo de la forma en que estén escritos, los sistemas expertos pueden formular inferencias utilizando estos hechos. También pueden (1) separar los datos del ruido, (2) transformar datos en información o (3) transformar la información en conocimiento.

Como ejemplo de estos conceptos, observe la siguiente secuencia de 24 números:

```
137178766832525156430015
```

Sin conocimiento, toda esta secuencia aparecería como ruido. Sin embargo, si se sabe que esta secuencia tiene un significado, entonces se convierte en datos. Determinar lo que son datos y lo que es ruido es parecido al viejo dicho relacionado con la jardinería, "hierba es cualquier cosa que crece y que no es lo que usted quiere".

Quizá exista cierto conocimiento que permita convertir estos datos en información. Por ejemplo, el siguiente algoritmo procesa los datos para producir información.

```
Agrupe los números en pares.
Ignore cualquier número de dos dígitos menor a 32.
Sustituya los caracteres ASCII por los números de dos
digitos.
```

La aplicación de este algoritmo a los anteriores 24 dígitos produce la siguiente información

```
ORO 436+
```

Ahora el conocimiento puede aplicarse a esta información. Por ejemplo, tal vez haya una regla

```
SI oro es menor a 500
   y el precio está subiendo (+)
ENTONCES
   compra oro
```

Aunque no se muestra explícitamente en la figura 2.2, *la experiencia* es un tipo de conocimiento que poseen los especialistas. Ésta no se encuentra comúnmente en fuentes públicas de información, como libros y artículos; es el conocimiento implícito de los especialistas que debe extraerse y hacerse explícito para que pueda codificarse en un sistema experto. Por encima del conocimiento está el **metaconocimiento**, el significado del prefijo **meta** es "más allá de". El metaconocimiento es conocimiento acerca del conocimiento y la experiencia. Como es posible diseñar un sistema experto con conocimiento de varios dominios diferentes, el metaconocimiento especificaría cuál base de conocimiento es la aplicable. Por ejemplo, un sistema experto puede tener bases de conocimiento acerca de reparación de autos Chevrolet 1988, Ford 1985 y Cadillac 1989; se emplearía la base apropiada dependiendo del automóvil que se necesite reparar. Esto no es eficiente para la memoria y rapidez, porque todas la bases del conocimiento estarían trabajando al mismo tiempo. Además, pueden surgir conflictos cuando el sistema experto trate de decidir las reglas aplicables de todas las bases del conocimiento a la vez. El metaconocimiento podría utilizarse también dentro de un dominio para decidir cuál grupo de reglas es el más aplicable.

En un sentido filosófico, la **sabiduría** es la cumbre del conocimiento. La sabiduría es el metaconocimiento de determinar los mejores objetivos en la vida y cómo lograrlos. Una regla de sabiduría sería

```
SI tengo suficiente dinero para mantener feliz a mi esposa
ENTONCES me retiraré y disfrutaré de la vida
```

Sin embargo, debido a la extrema escasez de sabiduría en el mundo, debemos limitarnos a sistemas basados en el conocimiento y dejarle los basados en la sabiduría a los políticos.

2.3 PRODUCCIONES

Se han concebido varias técnicas diferentes de representación del conocimiento. Entre éstas se incluyen reglas, redes semánticas, marcos, guiones, lenguajes de representación del conocimiento como KL-1 (Woods 83) y KRYPTON (Brachman 83), graficas conceptuales (Sowa 84) y otros (Brachman 80). Como se describió en el capítulo 1, las reglas de producción suelen utilizarse como base de conocimiento para los sistemas expertos, porque sus ventajas sobrepasan en mucho a sus desventajas.

Una notación formal para definir producciones es la forma Backus-Naur (BNF; McGettrick 80). Esta notación es un **metalenguaje** para definir la **sintaxis** de un lenguaje. La sintaxis define la forma, mientras que la **semántica** se refiere al significado. Un metalenguaje es un lenguaje para describir otros. El prefijo *meta* significa "más allá de", por lo tanto un metalenguaje va "más allá de" un lenguaje normal.

Hay muchos tipos de lenguajes: naturales, lógicos, matemáticos y de cómputo. La notación de BNF para la simple regla de una oración formada por un sujeto y un verbo seguidos por un signo de puntuación, es la siguiente regla de producción:

```
<oración> ::= <sujeto> <verbo> <signo final>
```


Los **paréntesis en ángulo, <>** y **::=** son símbolos del metalenguaje y no del lenguaje que se especifica. El símbolo "::=" significa "se define como" y es el equivalente en BNF de la flecha, →, utilizada en el capítulo 1 con las reglas de producción. Para evitar confusión con el operador de asignación de Pascal, :=, utilizaremos la flecha.

A los términos dentro de los paréntesis en ángulo se les llama ***símbolos no terminales*** o simplemente no terminales. Un símbolo no terminal es una variable en que se representa otro término. Éste puede ser no terminal o **terminal**. Un terminal no se puede reemplazar con algo más y, por tanto, es una constante.

La <frase> no terminal resulta especial porque se trata del único **símbolo de inicio** a partir del cual se definen los otros. En la definición de lenguajes de programación, al símbolo de inicio a menudo se le llama <programa>. La siguiente regla de producción

```
<oración> → <sujeto> <verbo> <signo final>
```

establece que una oración se integra con un sujeto seguido por un verbo y luego un signo final. Las siguientes reglas completan las no terminales especificando sus terminales posibles. La **barra** significa "o" en el metalenguaje.

```
<sujeto> → Yo | Tú | Nosotros
<verbo> → fui | vine | fuiste | vinistes | fuimos |
vinimos
<signo final> → . | ? | !
```

Todas las oraciones posibles en el lenguaje, las producciones, pueden producirse reemplazando sucesivamente cada uno de los no terminales con sus no terminales o terminales del lado derecho, hasta eliminar todos los no terminales. Las siguientes son algunas producciones:

```
Yo fui.
¿Yo fui?
¡Yo fui!
Tú fuistes.
¿Tú fuistes?
¡Tú fuistes!
Nosotros fuimos.
¿Nosotros fuimos?
¡Nosotros fuimos!
```

A un conjunto de terminales se le llama ***cadena*** del lenguaje. Si la cadena puede derivarse del símbolo de inicio, mediante el reemplazo de no terminales, con sus reglas de definición, entonces se le llama ***oración válida***. Por ejemplo, "Nosotros", "NosotrosNosotros", y "fufuistefuimos" son cadenas válidas del lenguaje, pero no son oraciones válidas.

Una **gramática** es un conjunto completo de reglas de producción que definen un lenguaje sin ambigüedades. Aunque las reglas anteriores definen una gramática, es un conjunto muy limitado porque hay pocas producciones posibles. Por ejemplo, una gramática más elaborada también podría incluir objetos directos, como en las producciones

```
<oración> → <sujeto> <verbo> <objeto> <signo final>
<objeto> → casa | trabajo | escuela
```

Aunque ésta es una gramática válida, resulta demasiado simple para el uso práctico. Una gramática más práctica es la siguiente, en la que se ha omitido el signo final para simplificar:

```
<oración> → <frase de sujeto> <verbo> <frase de objeto>
<frase de sujeto> → <determinador> <sustantivo>
<frase de objeto> → <determinador> <adjetivo> sustantivo>
<determinador> → un | uno | unos | unas | el | la | los |
las | este | esta | estos | estas | esos | esas
<sustantivo> → hombre | devorador
<verbo> → es | está | fue, estuvo
<adjetivo> → postres | gran
```

El <determinador> se utiliza para indicar un elemento específico. Utilizando esta gramática, pueden generarse oraciones como:

```
el hombre fue un devorador de postres
un devorador fue el gran hombre
```

Un **árbol de análisis gramatical** o **árbol de derivación** es una representación gráfica de una oración, descompuesta en todos los terminales y no terminales utilizados para derivarla. En la figura 2.3 se muestra el árbol de análisis gramatical para la frase "el hombre fue un gran devorador". Sin embargo la cadena "hombre fue un gran devorador" no es una oración válida porque carece del determinador en la frase de sujeto. Un compilador crea un árbol de análisis gramatical cuando trata de determinar si las oraciones de un programa se adecuan a la sintaxis válida de un lenguaje.

El árbol de la figura 2.3 muestra que la oración "el hombre fue un gran devorador" puede

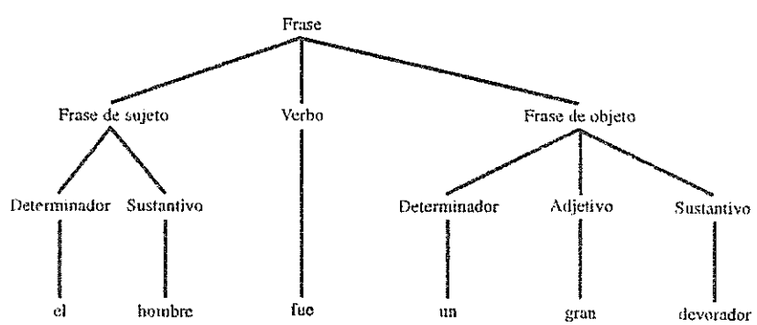


Figura 2.3 Árbol de análisis gramatical de una oración

derivarse del símbolo de inicio mediante la aplicación de las producciones apropiadas. A continuación se explican los pasos de este proceso; la **flecha doble**, =>, significa aplicar las producciones mostradas.


```
<frase> => <frase de sujeto> <verbo> <frase de objeto
<frase de sujeto> => <determinador> <sustantivo>
<determinador> => el
<sustantivo> => hombre
<verbo> => fue
<frase de objeto> => <determinador> <adjetivo> susta.
<determinador> => un
<adjetivo> => gran
<sustantivo> => devorador
```

Una forma alterna de utilizar las producciones consiste en generar frases válidas sustituyendo todas las terminales apropiadas por no terminales. Por supuesto, no todas las producciones, como "el hombre fue el postre", tendrán sentido.

2.4 REDES SEMÁNTICAS

Una **red semántica**, o red, es una técnica clásica de representación de la AI utilizada para información relativa a las proposiciones (Stillings 87). A una red semántica también se le llama ***red de proposiciones***. Como se analizó antes, una proposición es una frase verdadera o falsa, como "todos los perros son mamíferos" y "un triángulo tiene tres lados". Son formas de conocimiento declarativo porque establecen hechos. En términos matemáticos, una red semántica es una gráfica rotulada y con dirección. Una proposición siempre es verdadera o falsa y se le llama **atómica** porque su veracidad ya no puede dividirse más, el término *atómica* se emplea en el sentido clásico de objeto indivisible. En contraste, las proposiciones que trataremos en el capítulo 5 normalmente sólo necesitan ser verdaderas o falsas.

Las redes semánticas se desarrollaron para la AI como una forma de representar la memoria y la comprensión del lenguaje del ser humano (Quillian 68). Quillian las utilizó para analizar el significado de palabras en frases. Desde entonces se han aplicado a muchos problemas relacionados con la representación del conocimiento.

La estructura de una red semántica se muestra gráficamente en términos de **nodos** y los **arcos** que los conectan. A los nodos suele denominárseles como objetos y a los arcos como ***vínculos*** o ***bordes***.

Los vínculos de una red semántica se utilizan para expresar relaciones. Por lo general, los nodos se utilizan para representar objetos físicos, conceptos o situaciones. En la figura 2.4 (a) se muestra una red ordinaria, en realidad una gráfica con dirección, donde los vínculos indican las rutas de una línea aérea entre ciudades. Los nodos son los círculos, los vínculos, las líneas que los conectan y las flechas muestran la dirección en que pueden volar los aviones, de aquí el término gráfica con dirección. En la figura 2.4 (b) los vínculos muestran las relaciones entre los miembros de una familia. Éstas son de gran importancia en las redes semánticas porque proporcionan la estructura básica para la organización del conocimiento. Sin relaciones, el conocimiento es una simple colección de hechos inconexos. Con ellas, el conocimiento es una estructura cohesiva a partir de la cual puede inferirse conocimiento adicional. Por ejemplo, en la figura 2.4 (b) se puede inferir que Ann y Bill son los abuelos de John, aunque no haya un vínculo explícito rotulado "abuelo de".

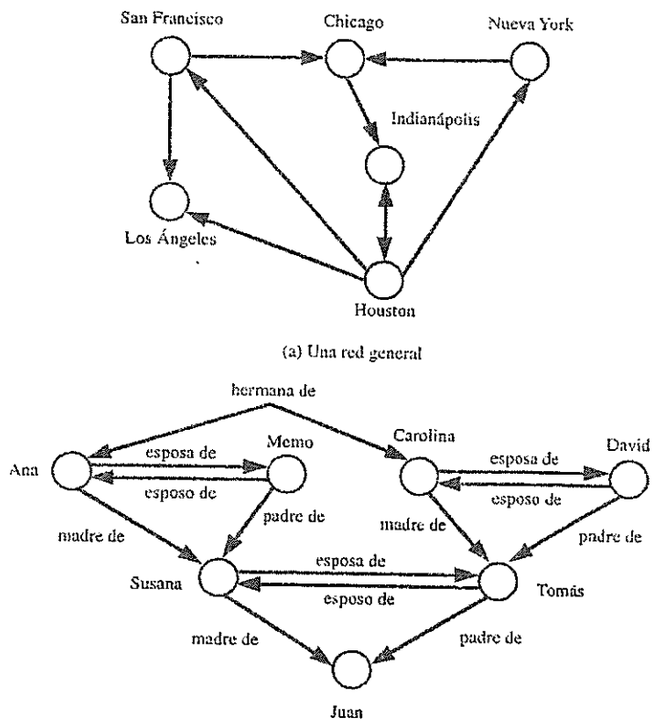


Figura 2.4 Dos tipos de red

A las redes semánticas a veces se les denomina **redes asociativas** porque los nodos están asociados o relacionados con otros nodos. De hecho, el trabajo original de Quillian modeló la memoria humana como una red asociativa en lo cual los conceptos eran los nodos y los vínculos formaban las conexiones entre ellos. De acuerdo con este modelo, cuando un nodo de concepto se estimula al leer las palabras de una frase, sus vínculos con otros nodos se activan en un patrón de dispersión. Si otro nodo recibe activación suficiente, el concepto sería forzado dentro de la mente consciente. Por ejemplo, aunque se conozcan miles de palabras, sólo se está pensando en las palabras específicas de la frase que se lee.

Ciertos tipos de relaciones han probado ser muy útiles en una amplia variedad de representaciones del conocimiento. En lugar de definir nuevas relaciones para problemas diferentes, es habitual usar estos tipos estándar. El uso de tipos comunes facilita la comprensión de una red no muy conocida.

Dos tipos de vínculos utilizados con frecuencia son **ES-UN** y **UN TIPO DE** (Winston 84). La figura 2.5 es un ejemplo de una red semántica que utiliza estos vínculos; en esta figura, ES-UN significa "es un caso de" y se refiere a un miembro específico de una clase. Una **clase** está relacionada con el concepto matemático de conjunto, que alude a un grupo de objetos.

Aunque un conjunto puede tener elementos de cualquier tipo, los objetos de una clase tienen alguna relación entre sí. Por ejemplo, es posible definir un conjunto formado por

```
{ 3, huevos, azul, llantas, arte }
```

Sin embargo, los miembros de este conjunto no tienen relación entre sí. Por el contrario, aviones, trenes y automóviles se relacionan en una clase porque todos son tipos de transporte.

El vínculo UN-TIPO-DE se utiliza aquí para relacionar una clase con otra. No se usa para relacionar un individuo específico porque esa es la función de ES-UN. UN-TIPO-DE relaciona una clase individual con una clase padre a la que pertenece una clase hijo.

Desde otro punto de vista, el UN-TIPO-DE relaciona nodos **genéricos** con nodos genéricos, mientras que ES-UN relaciona un caso o un **individuo** con una clase genérica. Observe en la figura 2.5 que las clases más generales se encuentran en la parte superior y las más específicas en la base. La clase más general, a la que señala una flecha UN-TIPO-DE, se le denomina **superclase**; si ésta tiene un vínculo UN-TIPO-DE apuntando a otro nodo, entonces también es una clase de la superclase a la que señala el vínculo UN-TIPO-DE. Otra forma de expresarlo es que un vínculo UN-TIPO-DE señale de una **subclase** a una clase. El vínculo **SON** se usa algunas veces para UN-TIPO-DE y se lee como el verbo "son".

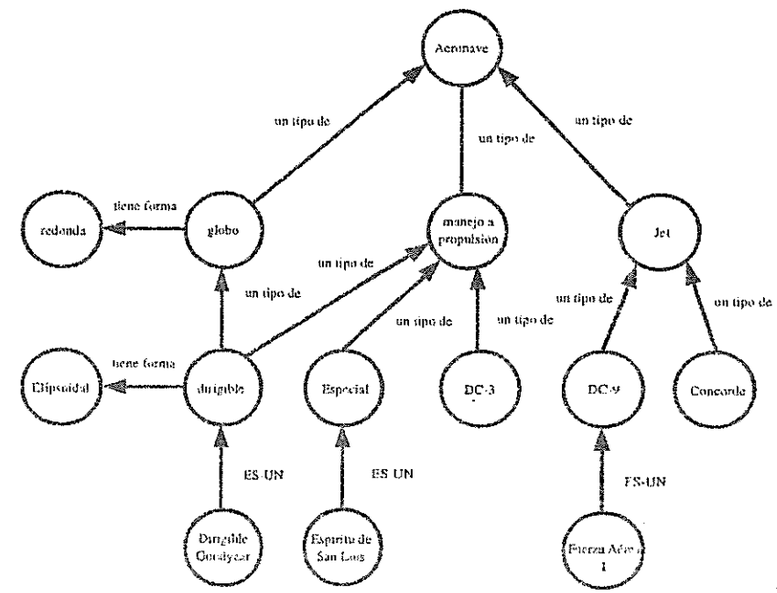


Figura 2.5 Red semántica con vínculos ES-UN y UN-TIPO-DE

Los objetos de una clase tienen uno o más **atributos** en común y cada atributo tiene un **valor**. La combinación de atributo y valor es una **propiedad**. Por ejemplo, un dirigible tiene

atributos de tamaño, peso, forma, color, etcétera. El valor del atributo de forma es elipsoidal. En otras palabras, un dirigible tiene la forma elipsoidal como una de sus propiedades. Como el Goodyear es un dirigible y los dirigibles tienen forma elipsoidal, entonces el Goodyear es elipsoidal. También se pueden encontrar otros tipos de vínculos en las redes semánticas. El vínculo **ES** define un valor. Por ejemplo, cualquier aeronave presidencial de Estados Unidos es el Fuerza Aérea 1. Si el presidente está en un DC-9, entonces el Fuerza Aérea 1 es un DC-9. El vínculo **CAUSA** expresa conocimiento causal. Por ejemplo, el aire caliente CAUSA que un globo se eleve.

A la duplicación de las características de un nodo por parte de un descendiente se le llama **herencia**. A menos que haya más evidencias específicas de lo contrario, se supone que todos los miembros de una clase heredarán todas las propiedades de sus superclases. Por ejemplo, los globos tienen una forma redonda. Sin embargo, como la clase dirigible tiene un vínculo que señala a la forma elipsoidal, ésta tiene precedencia. La herencia es una herramienta muy útil en la representación del conocimiento porque elimina la necesidad de repetir características comunes. Los vínculos y la herencia proporcionan medios muy eficientes de representación del conocimiento porque muchas relaciones complejas pueden mostrarse con pocos nodos y vínculos.

2.5 TRIPLETA OBJETO-ATRIBUTO-VALOR

Un problema con el uso de las redes semánticas es que no hay definiciones estándar para nombrar lo vínculos. Por ejemplo, algunos libros utilizan ES-UN para relaciones genéricas e individuales (Barr 81; Staugaard 87). ES-UN se utiliza entonces en el sentido ordinario de las palabras "es un" y también en el sentido de UN-TIPO-DE. En la herramienta para sistemas expertos ART, IS-A (ES-UN) relaciona dos clases de objetos e INSTANCE-OF (CASO-DE) relaciona a un individuo con una clase.

Otro vínculo muy utilizado es TIENE-UN, que relaciona una clase con una subclase. TIENE-UN señala a la inversa de UN-TIPO-DE y suele utilizarse para relacionar un objeto con una parte del objeto. Por ejemplo,

```
el automóvil TIENE-UN motor
el automóvil TIENE-UNAS llantas
el automóvil ES-UN ford
```

Más específicamente, ES-UN relaciona un valor con un atributo mientras que TIENE-UN relaciona un objeto con un atributo.

Los elementos objeto, atributo y valor ocurren tan frecuentemente que es posible construir una red semántica simplificada utilizando únicamente estos elementos. Una **tripleta objeto-atributo-valor** (OAV), puede utilizarse para caracterizar todo el conocimiento de una red semántica y se usó en el sistema experto MYCIN para diagnosticar enfermedades infecciosas. La representación del trío OAV es conveniente para ordenar el conocimiento en forma de tabla y luego traducirla al código de la computadora mediante inducción de reglas. En la tabla 2.1 se muestra el ejemplo de una tabla con el trío OAV.

Los tríos OAV son especialmente útiles para representar los hechos y sus patrones para llevarlos al antecedente de una regla. La red semántica para este tipo de sistema consta de nodos para objetos, atributos y valores, conectados por vínculos TIENE-UN y ES-UN. Si sólo habrá de representarse un objeto y no se requiere de herencia, tal vez bastaría una simple representación llamada ***pares atributo-valor***, o simplemente *AV*.

Objeto	Atributo	Valor
manzana	color	roja
manzana	tipo	mcintosh
manzana	cantidad	100
uvas	color	rojas
uvas	tipo	sin semilla
uvas	cantidad	500

Tabla 2.1 Tabla OAV

2.6 PROLOG Y REDES SEMÁNTICAS

Las redes semánticas son fáciles de traducir a PROLOG. Por ejemplo,

```
es un (dirigible_goodyear,dirigible).
es un (espíritu_de_san_luis,especial).
tiene forma (dirigible,elipsoidal).
tiene forma (globo,redondo).
```

son instrucciones de PROLOG que expresan algunas de las relaciones de la red semántica de la figura 2.5. Un punto marca el final de una frase.

Elementos esenciales de PROLOG

Cada una de las frases anteriores es una **expresión de un predicado** de PROLOG, o simplemente un predicado, porque está basado en lógica de predicados. Sin embargo, PROLOG no es un verdadero lenguaje de lógica de predicados porque es un lenguaje computacional con instrucciones ejecutables. En PROLOG, un predicado está formado por el nombre de predicado, como es un, seguido de cero o más argumentos encerrados entre paréntesis y separados por comas. A continuación se presentan algunos ejemplos de predicados y comentarios elaborados en PROLOG precedidos por punto y coma:

```
color(rojo).              ; rojo es un color
padre_de(tom,john).       ; tom es el padre de john
madre_de(susan,john).     ; susan es la madre de john
padres(tom,susan,john).   ; tom y susan son
                          ;  padres de john
```

Los predicados con dos argumentos son más fáciles de entender si coloca el nombre de predicado siguiendo al primer argumento. Los significados de estos predicados con más de dos argumentos deben establecerse explícitamente, como lo ilustra el predicado padres. Otra dificultad es que las redes semánticas se utilizan primordialmente para representar relaciones binarias, porque una línea sólo tiene dos extremos, por supuesto. No es posible poner la afirmación padres como un borde con una sola dirección porque hay tres argumentos. La idea de colocar a Tom y Susan juntos en un nodo de padres y a John en las otras direcciones lleva a una nueva complicación. Entonces es imposible utilizar el nodo de los padres para otra relación binaria, como madre-de, porque también se incluiría a Tom.

Los predicados pueden expresarse también con relaciones IS-A (ES-UN) y HAS-A (TIENE-UN).

```
is_a(rojo,color).
has_a(john,padre).
has_a(john,madre).
has_a(john,padres).
```

Observe que el predicado has_a no expresa el mismo significado que antes, porque el padre, la madre y los padres de John no son nombrados explícitamente. Para ello, se deben agregar algunos predicados adicionales.

```
is_a(tom,padre).
is_a(susan,madre).
is_a(tom,padre).
is_a(susan,padre).
```

Incluso estos predicados adicionales no expresan el mismo sentido de los originales. Por ejemplo, sabemos que John tiene un padre y que Tom es un padre, pero esto no informa que Tom sea el padre de John.

Todas las frases anteriores describen realmente hechos de PROLOG. Los programas de PROLOG están formados por hechos y reglas en la forma general de **metas**:

$$p \text{ :- } p_1, p_2, \ldots p_N.$$

en los que p es la cabeza de la regla y p_i son las **submetas.** Normalmente esta expresión es una cláusula de Horn, que establece que la meta principal, p, se satisface si y sólo si son satisfechas todas las submetas. La excepción se presenta cuando se utiliza el predicado especial para fallas, que resulta sumamente conveniente porque no es fácil probar una negación en lógica clásica y PROLOG está basado en la lógica clásica. Una negación es vista como la falla al tratar de encontrar una prueba, esto puede ser un proceso muy largo y complicado si hay muchas coincidencias probables. El predicado de **corte** y de falla especial hace que el proceso de negación sea más eficiente reduciendo la búsqueda de una prueba.

Las comas que separan las submetas representan una disyunción Y lógica. El símbolo, :-, se interpreta como un SI. Si sólo existe la cabeza y no hay lado derecho, como en

```
p.
```

entonces la cabeza se considera verdadera. Esto se debe a que predicados como los siguientes se consideran hechos y, por tanto, deben ser verdaderos.

```
color(rojo).
has_a(john,padre).
```

Otra forma de pensar en los hechos es considerandolos una conclusión incondicional que no depende de nada más y, por lo tanto, el SI, :-, no es necesario. Por el contrario, las reglas de PROLOG necesitan el SI porque son conclusiones condicionales que en verdad dependen de una o más condiciones. Como ejemplo, las reglas de los padres quedan así:

```
padre(X,Y) :- papá (X,Y).
padre(X,Y) :- mamá (X,Y).
```


lo que significa que X es un padre de Y, si X es el papá de Y o si X es la mamá de Y. Del mismo modo, un abuelo puede definirse como:

```
abuelo(X,Y) :- padre(X,Z),padre(Z,Y).
```

y un antepasado puede definirse como:

```
(1)  ancestro(X,Y) :- padre(X,Y).
(2)  ancestro(X,Y) :- ancestro (X,Z),ancestro(Z,Y).
```

en donde (1) y (2) sólo se utilizan con propósitos de identificación.

Búsqueda en PROLOG

Un sistema para ejecución de instrucciones de PROLOG es generalmente un intérprete, aunque algunos sistemas pueden generar código compilado. En la figura 2.6 se muestra la forma general de un sistema PROLOG. El usuario interactúa con PROLOG mediante la introducción de consultas de predicados y la recepción de respuestas. La **base de datos de los predicados** contiene a los predicados de regla y de hecho que se han introducido, y con ellos forma la base del conocimiento. El intérprete trata de determinar si un predicado de consulta que el usuario introdujo se encuentra en la base de datos. La respuesta devuelta es afirmativa en caso de que se encuentre en la base de datos; de lo contrario, es negativa. Si la consulta es una regla, el interprete tratará de satisfacer los subobjetivos conduciendo primero una búsqueda a fondo, como se muestra en la figura 2.7. Por el contrario, también se muestra una búsqueda primero a lo ancho, aunque éste no es el modo normal de PROLOG. En una **búsqueda primero a fondo**, la búsqueda desciende lo más posible y luego regresa. En PROLOG también va de izquierda a derecha. Una **búsqueda primero a lo ancho** pasa de nivel en nivel antes de descender al siguiente nivel inferior.

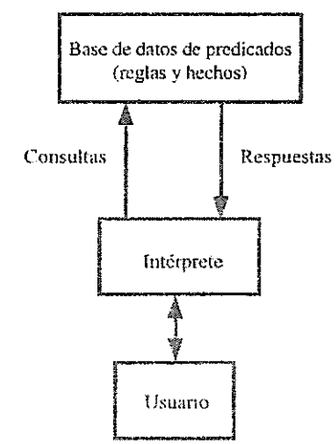


Figura 2.6 Organización general de un sistema PROLOG

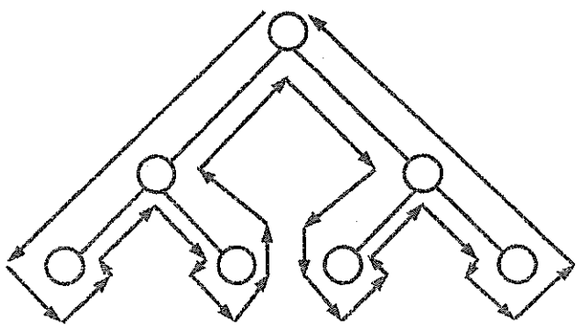
Consideremos las reglas de los antepasados, definidas como (1) y (2), como ejemplo de búsqueda de objetivos en PROLOG. Tomemos los siguientes hechos como entrada.

```
(3)  padre(ann,mary).
(4)  padre(ann,susan).
(5)  padre(mary,bob).
(6)  padre(susan,john).
```

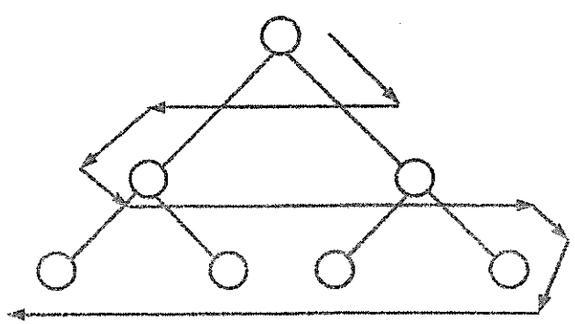
Ahora supongamos que se le consulta a PROLOG si Ann es antepasado de Susan

```
:-ancestro(ann,susan).
```

donde la ausencia de una cabeza indica que se trata de una **consulta**, que es una condición que habrá de probar PROLOG. Hechos, reglas y consultas son los tres tipos de cláusula de Horn aplicables a PROLOG. Se puede probar una condición, si es la conclusión del caso de una cláusula. Por supuesto, la propia cláusula debe poderse probar por sí misma, lo que se hace mediante la demostración de las condiciones de la cláusula.



(a) Búsqueda primero a fondo



(b) Búsqueda primero a lo ancho

Figura 2.7 Búsquedas primero a fondo y primero a lo ancho para un árbol arbitrario

Al recibir esta entrada, PROLOG empezaría a buscar una instrucción cuya cabeza coincida con el patrón de entrada ancestro(ann,susan). A esto se le llama ***coincidencia de patrones***, de manera parecida a la coincidencia de patrones de hechos en el antecedente de una regla de producción. La búsqueda empieza desde la primera instrucción que se introdujo (1), la **superior**, y sigue hasta la última instrucción (6), la **inferior**. Hay una posible coincidencia con la primera regla de ancestro (1). La variable X coincide con ann y la variable Y con susan. Como la cabeza coincide, ahora PROLOG tratará de equiparar el cuerpo de (1), produciendo la submeta padre(ann,susan). PROLOG trata de que esta submeta coincida con las cláusulas y en algún momento coincidirá con el hecho (4) padre(ann,susan). No hay más objetivos que coincidan y PROLOG responde "sí" porque la consulta original es verdadera.

Otro ejemplo, supongamos que la consulta es

```
:-ancestro(ann,john)
```

La primera regla de ancestros (1) coincide y X toma el valor de ann y Y el de john. PROLOG ahora trata de hacer que coincida el cuerpo de (1), padre(ann,john), con todas las instrucciones de padre. Ninguna coincide y, por tanto, el cuerpo de (1) no puede ser cierto. Como el cuerpo de (1) no es cierto, la cabeza no puede ser cierta.

Como (1) no puede ser verdadero, PROLOG prueba la segunda instrucción de ancestros (2). X toma el valor de ann y Y el de john. PROLOG trata de probar que la cabeza de (2) es verdadera probando que ambas submetas de (2) son verdaderas. Es decir, debe probarse que ancestro(ann,Z) y ancestro(Z,john) son verdaderos. PROLOG trata de equiparar las submetas de izquierda a derecha a partir de ancestro(ann,Z). Empezando desde la parte superior, coincide primero con la instrucción (1), por lo tanto PROLOG trata de probar el cuerpo de (1), padre(ann,Z). Empezando a buscar nuevamente desde la parte superior, este cuerpo coincide primero con la instrucción (3) y por ello PROLOG asigna a Z el valor de mary. Ahora PROLOG trata de equiparar la segunda submeta de (2) como ancestro (mary,john). Esto coincide con (1) y, por tanto, PROLOG trata de satisfacer su cuerpo, padre(mary,john). Sin embargo, no hay padre(mary,john) en la instrucción (3) a la (6) y, por tanto, esta búsqueda falla.

Entonces PROLOG examina de nuevo su suposición de que Z debe ser mary. Como esta opción no funcionó, trata de encontrar otro valor que funcione. Otra posibilidad surge de (4) al asignar a Z el valor de susan. A esta técnica de retroceso para probar una ruta de búsqueda diferente cuando otra falla, se le denomina rastreo hacia atrás o *backtrack* y se utiliza a menudo para resolver problemas.

Con la elección de Z como susan, PROLOG intenta probar que ancestro(susan,john) es verdadero. Mediante (1), debe probarse que el cuerpo padre(susan,john) es verdadero. Efectivamente, el hecho (3) coincide y entonces se prueba que la consulta

```
:-ancestro(ann,john).
```

es verdadera porque se ha probado que sus condiciones son verdaderas.

Observe que la estructura de control de PROLOG tiene el tipo del algoritmo de Markov, en el que la búsqueda de patrones coincidentes se determina normalmente por el orden en que se introducen las cláusulas de Horn. Esto difiere con los sistemas expertos basados en reglas, en que suelen seguir el paradigma de Post, donde el orden en que se introducen las reglas no afecta la búsqueda.

PROLOG tiene muchas otras funciones y capacidades que no se mencionan en esta breve introducción. Desde el punto de vista de un sistema experto, el backtrack y la equiparación de patrones son muy útiles, como también lo es la naturaleza declarativa de PROLOG, porque la especificación de un programa es un programa ejecutable.

2.7 DIFICULTADES CON REDES SEMÁNTICAS

Aunque las redes semánticas pueden ser muy útiles en la representación del conocimiento, tienen limitantes, como la falta de estándares para asignación de nombres a los vínculos, que ya se explicó. Esto hace difícil entender para qué se diseñó realmente la red y si fue diseñada de una forma congruente. Un problema complementario al asignar nombre a los vínculos es el nombre de los nodos. Si un nodo se rotula "silla", ¿Representa

una silla específica
la clase de todas las sillas
el concepto de una silla
el nombre del cerro de la Silla

o algo más? Para que una red semántica represente **conocimiento definitivo** (es decir, conocimiento que puede definirse), deben definirse con rigor los nombres de vínculos y nodos. Claro que los mismos problemas pueden ocurrir en los lenguajes de programación.

Otro problema es la explosión combinatoria en la búsqueda de nodos, sobre todo si la respuesta a una consulta es negativa. Para que una consulta produzca un resultado negativo, tendrían que buscarse muchos o todos los vínculos de la red. Como se mostró en el problema del agente viajero del capítulo 1, el número de vínculos es el factorial del número de nodos menos uno, si están todos conectados. Aunque no todas las representaciones necesitarán este grado de conectividad, existe la posibilidad de que surja una explosión combinatoria.

Las redes semánticas fueron propuestas originalmente como modelos de la memoria asociativa humana, en la que un nodo tiene vínculos con otros y la recuperación de la información ocurre debido a una dispersión de la activación de los nodos. No obstante, el cerebro humano también debe disponer de otros mecanismos, porque no le toma mucho tiempo responder la consulta: ¿hay un equipo de fútbol en Plutón? Hay alrededor de 10^{10} neuronas en el cerebro humano y alrededor de 10^{15} vínculos. Si todo el conocimiento estuviera representado por una red semántica, tomaría demasiado tiempo responder consultas negativas como en el caso de la consulta del fútbol, porque todas las búsquedas incluyen 10^{15} vínculos.

Las redes semánticas son lógicamente inadecuadas porque no pueden definir el conocimiento de la forma en que lo hace la lógica. Una representación lógica puede especificar cierta silla, algunas sillas, todas las sillas, ninguna silla, etcétera, como se analizará más adelante en este capítulo. Otro problema es que las redes semánticas también son heurísticamente inadecuadas porque no hay forma de insertar en la red información heurística relacionada con la manera de explorarla de forma eficiente. Una regla **heurística** es una regla práctica que puede ayudar a encontrar una solución, pero que no está garantizada de la misma forma que un algoritmo. Las reglas heurísticas son muy importantes en la AI porque los problemas típicos en ello son tan difíciles que no existe una solución de algoritmo o, si la hay, no resulta eficiente para uso práctico. La herencia es la única estrategia estándar de control integrada en una red que puede ayudar, pero no todos los problemas pueden tener esta estructura.

Se han probado varios métodos para corregir los problemas inherentes a las redes semánticas: se han hecho mejoras logicas y se han probado mejoras heurísticas al adjuntar procedimientos a los nodos, que serán ejecutados cuando el nodo se active. Sin embargo, los sistemas resultantes mejoraron poco su capacidad a costa de la fuerza de expresión natural de las redes semánticas. La conclusión de toda este esfuerzo es que, como cualquier herramienta, las redes semánticas deben utilizarse para aquello que hacen mejor, mostrar relaciones binarias, y no considerarse una herramienta universal.

2.8 ESQUEMAS

Una red semántica es un ejemplo de una **estructura de conocimiento superficial.** La superficialidad se encuentra presente porque todo el conocimiento de la red semántica se encuentra en los vínculos y nodos. El término *estructura del conocimiento* es análogo a una estructura de datos en la que más que sólo datos se representa una colección ordenada de conocimiento. Una estructura de conocimiento **profundo** tiene conocimiento causal que explica por qué ocurre algo. Por ejemplo, es posible construir un sistema experto médico con conocimiento superficial como sigue:

```
SI una persona tiene fiebre
ENTONCES toma una aspirina
```

Pero estos sistemas no conocen la bioquímica fundamental del cuerpo ni la razón por la que la fiebre disminuye con la aspirina. La regla podría haberse definido:

```
SI una persona tiene un mono rosa
ENTONCES toma un refrigerador
```

En otras palabras, el conocimiento del sistema experto es superficial porque está basado en la sintaxis más que en la semántica, donde dos palabras cualquiera podrían ser sustituidas por X y Y en la siguiente regla

```
SI una persona tiene un (X)
ENTONCES toma un (Y)
```

Observe que (X) y (Y) no son variables en esta regla, sino que representan dos palabras cualquiera. Los médicos tienen conocimiento causal porque han tomado muchos cursos y han adquirido experiencia en el tratamiento de personas enfermas. Si un tratamiento no está funcionando correctamente, los médicos pueden razonar acerca de él para encontrar una alternativa. En otras palabras, un especialista sabe cuándo romper las reglas.

Muchos tipos de conocimientos del mundo real no pueden representarse con la simple estructura de una red semántica. Se necesitan estructuras más complejas para representar mejor complejas estructuras del conocimiento. En la AI, el término *esquema* se utiliza para describir una estructura de conocimiento más compleja que la red semántica. Este término viene de la psicología, donde alude a la organización continua del conocimiento o la respuesta de una criatura debida a un estímulo. Esto es, a medida que las criaturas aprenden la relación entre una causa y su efecto, tratarán de repetir la causa si el efecto es agradable, o evitarla, si no lo es.

Por ejemplo, los actos de comer y beber son esquemas **sensomotores** agradables que involucran información de coordinación de los sentidos con los movimientos motores (músculos) necesarios para realizarlos. Una persona no tiene que pensar en este conocimiento tácito para saber cómo realizar estos actos y es muy difícil explicar exactamente cómo desciende al nivel de controlar los músculos. Un esquema aún más difícil de explicar es cómo andar en bicicleta. ¡Cómo tratar de explicar el sentido del equilibrio!

Otro tipo de esquema es el **esquema de concepto.** Por ejemplo, todos tienen un concepto de *animal.* La mayoría de las personas a las que se les pida que expliquen cómo es un animal probablemente lo describirían como algo que tiene cuatro patas y pelo. Por supuesto, el concepto de *animal* será diferente si la persona creció en una granja, en una ciudad, cerca de un río, etcetera. Sin embargo, todos tenemos **estereotipos** de conceptos en nuestras mentes. Aun-

que el término *estereotipo* quizá posea un significado negativo en el lenguaje coloquial, para la AI significa un ejemplo típico. Para mucha gente un estereotipo de un animal podría ser algo como un perro.

Un esquema conceptual es una abstracción en la cual se clasifican objetos específicos, de acuerdo con sus propiedades generales. Por ejemplo, si ve un pequeño objeto rojo, redondo con tallo verde, sobre un letrero que dice "Fruta artificial", probablemente lo reconocerá como una manzana artificial. El objeto tiene las propiedades que usted asocia con el esquema conceptual de manzana, salvo que sea una manzana real, por supuesto.

El esquema conceptual de una manzana real incluirá propiedades generales de las manzanas como tamaños, colores, sabores, usos, etcétera; pero no incluirá detalles acerca de dónde se cosechó exactamente, el medio por el que se transportó hasta el supermercado, el nombre de la persona que la colocó en el anaquel, etcétera. Estos detalles no son importantes para las propiedades que conforman su concepto abstracto de una manzana. Además, tome en consideración que un ciego quizá tenga un esquema de concepto muy diferente, en el que la textura sea más importante.

Al concentrarse en las propiedades generales de un objeto, resulta más fácil razonar acerca de ellos sin distraerse con detalles poco importantes. Habitualmente los esquemas tienen una estructura interna en sus nodos, mientras que las redes semánticas no, en ellas el rótulo es todo el conocimiento acerca del nodo. Una red semántica es como un estructura de datos de la ciencia de la computación, en que la clave de búsqueda corresponde también a los datos almacenados en el nodo. Un esquema es como una estructura de datos donde los nodos contienen registros. Cada registro puede incluir datos, registros o apuntadores a otros nodos.

2.9 MARCOS

Un tipo de esquema que se ha utilizado en muchas aplicaciones de la AI es el **marco** (Minsky 75). Otro es el **guión**, que es en esencia una secuencia de marcos ordenada temporalmente (Schank 77). Propuestos como un método para comprender la visión, el lenguaje natural y otras áreas, los marcos proporcionan una estructura conveniente para representar objetos que son comunes a una situación dada, como los estereotipos. En particular, los marcos son útiles para simular conocimiento de sentido común, que es un área que le resulta muy difícil dominar a las computadoras. Las redes semánticas son básicamente representaciones bidimensionales del conocimiento: los marcos agregan una tercera dimensión para permitir que los nodos tengan estructuras, que pueden ser valores simples u otros marcos.

La característica básica de un marco, es que representa conocimiento relacionado con un tema concreto que cuenta con mucho conocimiento predeterminado. Un sistema de marcos sería una buena elección para describir un dispositivo mecánico como un automóvil: porque sus componentes, como el motor, la carrocería, los frenos, etcétera, se relacionarían entre sí para dar una visión general de sus vínculos. Sería posible obtener aún más detalles de los componentes si se examina la estructura de los marcos. Aunque las marcas individuales de automóviles variarán, muchos autos tienen características comunes como llantas, motor, carrocería, transmisión, etcétera. El marco difiere de la red semántica, que suele utilizarse para una amplia representación del conocimiento. Del mismo modo que con las redes semánticas, no hay estándares para definir los sistemas basados en marcos. Se han diseñado varios lenguajes específicos para los marcos, como FRL, SRL, KRL, KEE, HP-RL y funciones para acrecentarlos como LISP tales como LOOPS y FLAVORS (Finin 86).

Un marco es análogo a la estructura de registro en un lenguaje de alto nivel como Pascal o un átomo con su lista de propiedades en LISP. Correspondiendo a los campos y los valores de un registro son las **ranuras** y sus **rellenos** de un marco. Un marco es básicamente un grupo de ranuras y rellenos que definen un objeto estereotípico. En la figura 2.8 se muestra un ejemplo

de un marco para un automóvil. Desde el punto de vista de OAV, el automóvil es un objeto, el nombre de la ranura es el atributo y el relleno es el valor.

Ranuras	Rellenos
fabricante	General Motors
modelo	Chevrolet Caprice
año	1979
transmisión	automática
motor	a gasolina
ruedas	4
color	azul

Figura 2.8 Marco de automóvil

Casi ningún marco es tan simple como el que se muestra en la figura 2.8. La utilidad de los marcos recae en los sistemas jerárquicos de marcos y en la herencia. Al utilizar marcos en las ranuras de relleno y en la herencia, pueden construirse sistemas de representación del conocimiento muy versátiles. En particular, los sistemas expertos basados en marcos son muy útiles para representar al conocimiento causal, porque su información está organizada en causas y efectos. Por el contrario, los sistemas expertos basados en reglas dependen generalmente del conocimiento no organizado, que no es causal.

Algunas herramientas basadas en marcos como KEE permiten un amplio rango de elementos para almacenar en las ranuras. Éstas pueden contener reglas, imágenes, comentarios, depuración de información, preguntas a los usuarios, hipótesis relacionadas con una situación u otros marcos.

Los marcos se diseñan generalmente para representar conocimiento genérico o específico. En la figura 2.9 se ilustra un marco genérico para el concepto de propiedad.

Espacios	Rellenos
nombre	propiedad
especialización_de	un_tipo_de objeto
tipos	(automóvil, bote, casa)
	si-se-agrega: Procedimiento AGREGAR_PROPIEDAD
propietario	predeterminado: gobierno
	si-se-necesita: Procedimiento BUSCAR_PROPIETARIO
ubicación	(casa, trabajo, móvil)
estado	(falta, malo, bueno)
bajo_garantía	(si, no)

Figura 2.9 Marco genérico para propiedad

Los rellenos pueden ser valores, como una propiedad en la ranura del nombre, o un rango de valores, como en la ranura de tipos. Las ranuras también pueden contener procedimientos adjuntos, llamados ***anexos de procedimiento***. Por lo general son de tres tipos: los de tipo **si-es-necesario** son procedimientos que se ejecutarán cuando se necesita un valor de relleno pero no hay ninguno presente o el valor **predeterminado** no es adecuado. Los valores predeterminados son de importancia fundamental en los marcos, porque son modelos de algunos aspectos del cerebro. Corresponden a las expectativas de una situación que se construye a partir de la experiencia. Cuando encontramos una nueva situación, se modifica el marco más cercano para ayudarnos a ajustar la situación, así las personas no empiezan desde cero en cada situación nueva: solamente, se modifican las opciones predeterminadas o los otros rellenos. A menudo se utilizan las opciones predeterminadas para representar el conocimiento de

sentido común. Éste puede considerarse como el conocimiento que se conoce de manera general. Utilizamos el sentido común cuando no se dispone de más conocimiento específico para la situación.

El tipo **si-se-agrega** se ejecuta cuando se agrega un valor de un ranura. En la ranura de tipos, el procedimiento si-se-agrega ejecuta un procedimiento llamado AGREGAR-PROPIEDAD para añadir un nuevo tipo de propiedad, si es necesario. Por ejemplo, este procedimiento se ejecutaría para accesorios, TV, estéreo, etcétera, porque la ranura de tipos no contiene estos valores.

Un tipo **si-se-elimina** se ejecuta cada vez que se elimina un valor de una ranura. Este tipo de procedimiento se ejecuta cuando un valor es obsoleto.

Los rellenos de ranura también pueden contener relaciones, como en las ranuras especialización_de. Las relaciones un-tipo-de y es-un se utilizaron en las figura 2.9, 2.10 y 2.11 para mostrar la manera en que estos marcos están relacionados jerárquicamente. Los marcos de las figuras 2.9 y 2.10 son genéricos mientras que el de la figura 2.11 es un marco específico porque se trata de un caso del marco del automóvil. Supongamos aquí que las relaciones un-tipo-de son genéricas y las es-un son específicas.

Ranuras	Rellenos
nombre	automóvil
especialización_de	propiedad un-tipo-de
tipos	(sedan, deportivo, convertible)
fabricante	(GM, Ford, Chrysler)
ubicación	móvil
ruedas	4
transmisión	(manual, automática)
motor	(gasolina, diesel)

Figura 2.10 Marco de automóvil: un submarco genérico de propiedad

Los sistemas de marcos están diseñados para que los marcos más genéricos se encuentren en la parte superior de la jerarquía. Se supone que los marcos pueden personalizarse al modificar los casos predeterminados y crear marcos más específicos. Los marcos tratan de modelar objetos del mundo real utilizando el conocimiento genérico para la mayor parte de los atributos de un objeto, y el conocimiento específico para casos especiales. Por ejemplo, generalmente consideramos a las aves como criaturas que pueden volar, pero ciertos tipos de aves, como los pingüinos y los avestruces, no pueden hacerlo. Estos tipos representan clases más específicas de aves y, en la jerarquía de un marco, sus características se encontrarían en niveles inferiores a los de aves como los canarios o petirrojos. En otras palabras, la parte superior de la jerarquía del marco para aves especifica cosas que son más ciertas para todas las aves y los niveles inferiores reflejarían los límites confusos entre los objetos reales. Al objeto que tiene todas las características comunes se le llama *prototipo*, que significa literalmente el primer tipo.

Ranuras	Rellenos
nombre	automóvil de John
especialización_de	es_un automóvil
fabricante	GM
propietario	Juan Pérez
transmisión	automática
motor	a gasolina
estado	bueno
bajo_garantía	sí

Figura 2.11 Ejemplo de un marco de automóvil

Los marcos también pueden clasificarse por sus aplicaciones. Un **marco situacional** contiene conocimiento sobre lo que se espera en una situación determinada, como una fiesta de cumpleaños. Un **marco de acción** contiene ranuras que especifican las acciones que se realizarán en una situación determinada. Es decir, los rellenos son procedimientos para realizar algunas acciones, como eliminar una parte defectuosa de un cinturón de seguridad. Un marco de acción representa conocimiento del procedimiento. La combinación de marcos de situación y de acción puede utilizarse para describir relaciones de causa y efecto, en la forma de **marcos de conocimiento causal**.

Se han construido sistemas de marcos muy sofisticados para diversas tareas. Dos de los sistemas más impresionantes, han demostrado la capacidad de los marcos para descubrir creativamente conceptos matemáticos (Lenat 77) y describir comprensión matemática en dominios como el álgebra lineal (Rissland 78). En estos sistemas, las ranuras se usaron como generadores de tareas, tales como respuestas a preguntas. El sistema AM de Lenat haría conjeturas de interesantes conceptos novedosos y los exploraría.

2.10 DIFICULTADES CON LOS MARCOS

Los marcos se concibieron originalmente como un paradigma para representar conocimiento estereotipado. La característica más importante de un estereotipo es que cuenta con funciones bien definidas, de modo que muchas de sus ranuras tienen valores predeterminados. Los conceptos matemáticos son buenos ejemplos de estereotipos bien adecuados a los marcos. El paradigma de los marcos tiene un aspecto intuitivo, porque la representación organizada de marcos de conocimiento suele ser más fácil de comprender que cualquier sistema lógico o de producción con muchas reglas (Jackson 86). Sin embargo, han aparecido problemas importantes en los sistemas de marcos que parecen una modificación o cancelación de las ranuras sin restricciones. El ejemplo clásico de este problema se ilustra en la figura 2.12 con un marco que describe elefantes.

nombre	elefante
especialización_de	un-tipo-de mamífero
color	gris
patas	4
tronco	cilíndrico

Figura 2.12 Marco de elefante

A primera vista, el marco para el elefante parece una descripción genérica razonable de los elefantes. Sin embargo, si suponemos que existe un elefante de tres patas llamado Clyde, que ha perdido una pata debido a un accidente de cacería o que simplemente ha mentido sobre el hecho de perder una pata para que su nombre apareciera en este libro. Los importante es que el marco de elefante asegura que un elefante tiene cuatro patas, no tres. Por tanto, no podemos considerar que el marco para el elefante sea una definición de elefante.

Por supuesto, el marco puede modificarse para permitir elefantes con tres patas o menos. Sin embargo, esto no proporciona una buena definición, por que pueden surgir problemas adicionales con otras ranuras. Suponiendo que Clyde adquiere ictericia y su piel se vuelve amarilla, ¿deja de ser un elefante?

Una alternativa para observar un marco como una definición, es considerarlo como descripción de un elefante típico. Sin embargo, esto conduce a otros problemas debido a la herencia. Observe que el marco indica que un elefante es un-tipo-de mamífero. Como estamos

interpretando los marcos como típicos, nuestro sistema de marco indica que un elefante típico es un mamífero típico; aunque un elefante es un mamífero, probablemente no sea un mamífero típico. Sobre la base de la cantidad, los seres humanos, las vacas, las ovejas o las ratas probablemente sean más representativos de los mamíferos típicos.

Casi ninguno de los sistemas de marcos proporciona una manera de definir ranuras inalterables. Cualquier ranura puede cambiarse, de modo que pueden modificarse o cancelarse las propiedades que un marco hereda hacia cualquier lugar de la jerarquía. Esto significa que cada marco es, en realidad, un marco primitivo y como no hay garantía de que las propiedades sean comunes, cada marco crea sus propias reglas. Nada es realmente cierto en un sistema sin restricciones como éste. Por lo tanto, es imposible hacer frases universales como las creadas en la definición de un elefante. Además, a partir de definiciones más simples como las del elefante, es imposibles construir de manera confiable objetos compuestos, como en el caso del elefante con tres patas. Los mismos tipos de problema se aplican a las redes semánticas con la herencia. Si las propiedades de cualquier nodo pueden cambiarse, entonces nada es cierto.

2.11 LÓGICA Y CONJUNTOS

Además de reglas, marcos y redes semánticas, el conocimiento puede representarse mediante los símbolos de la **lógica**, que es el estudio de las reglas del razonamiento exacto. Una parte importante del razonamiento es la inferencia de conclusiones a partir de premisas. La aplicación de computadoras para realizar razonamientos ha tenido como resultado la **programación lógica** y el desarrollo de lenguajes basados en la lógica como PROLOG. La lógica también tiene importancia primordial en los sistemas expertos, donde el mecanismo de inferencia razona a partir de los hechos para extraer conclusiones. En realidad, un término descriptivo para la programación lógica y los sistemas expertos es **sistemas de razonamiento automatizados**.

En el siglo IV antes de nuestra era, el filósofo griego Aristóteles desarrolló la lógica formal. Esta lógica aristotélica se basaba en los **silogismos**, de los que inventó 14 tipos, cinco más se inventaron en la Edad Media. Los silogismos tienen dos **premisas** y una **conclusión**, que se infiere a partir de aquéllas. El siguiente es el ejemplo clásico de un silogismo:

```
Premisa:      Todos los hombres son mortales
Premisa:      Sócrates es un hombre
Conclusión:   Sócrates es mortal
```

En un silogismo, la **premisa** proporciona la evidencia a partir de la cual surgirá la conclusión. Un silogismo es una manera de representar el conocimiento. Otra manera es un **diagrama de Venn**, como el que se muestra en la figura 2.13.

El círculo exterior representa a todas las criaturas mortales. El círculo interno representa a un hombre dibujado por completo en el interior del círculo mortal, para indicar que todos los hombres son mortales; como Sócrates es un hombre, el círculo que lo representa se dibuja por completo dentro del segundo círculo. Hablando estrictamente, el círculo que representa a Sócrates debe ser un punto, porque un círculo implica una clase, pero por razones de legibilidad utilizaremos círculos para todo. La conclusión de que Sócrates es mortal es una consecuencia del hecho de que su círculo se encuentra dentro del círculo de las criaturas mortales y, por tanto, debe ser mortal.

En términos matemáticos, un círculo del diagrama de Venn representa un conjunto, que es una colección de objetos. A los objetos de un conjunto se les llama **elementos**. Algunos ejemplos de conjuntos son


```
A = (1,3,5)
B = (1,2,3,4,5)
C = (0,2,4,...)
D = (...,-4,-2,0,2,4,...)
E = (aviones,globos,dirigibles,jets)
F = (aviones,globos)
```

donde los puntos suspensivos, ..., llamados elipses, indican que los términos continúan indefinidamente.

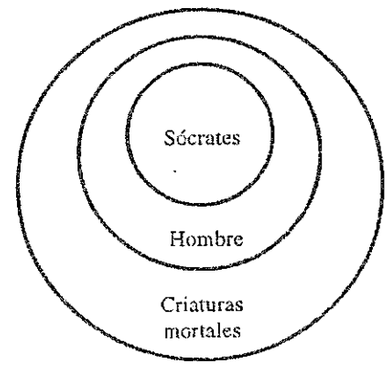


Figura 2.13 Diagrama de Venn

La letra griega épsilon, $\in$, indica que un elemento es un miembro de un conjunto. Por ejemplo, $1 \in A$ significa que el número 1 es un elemento del conjunto A previamente definido. Si un elemento no es miembro de un conjunto, se utiliza el símbolo $\notin$, como en $2 \notin A$.

Si dos conjuntos arbitrarios, como X y Y se definen para que cada elemento de X sea un elemento de Y, entonces X es un **subconjunto** de Y y se escribe en la forma matemática X Y o $Y \supset X$. A partir de la definición de un subconjunto, se desprende que cada conjunto es un subconjunto de sí mismo. A un subconjunto que no es el propio conjunto se le llama **subconjunto propio**. Por ejemplo, el conjunto X definido previamente es un subconjunto propio de Y. Al analizar los conjuntos, es útil considerarlos como subconjuntos de un **conjunto universal**. Éste cambia a medida que cambia el tema de análisis. En la figura 2.14 se ilustra un subconjunto formado como la **intersección** de dos conjuntos en el universo de todos los automóviles. Si se trata de números, el universo sería todos los números. El universo se dibuja como un rectángulo que rodea a sus subconjuntos. Los conjuntos universales no se usan sólo por conveniencia. El uso indiscriminado de condiciones para definir conjuntos puede dar como resultado paradojas lógicas.

Si dejamos que A sea el conjunto de todos los automóviles azules, B el conjunto de todos los automóviles con transmisión manual y C el conjunto de todos los automóviles azules con transmisión manual, escribiríamos:

$$C = A \cap B$$

En que el símbolo $\cap$ representa la intersección de los conjuntos. Otra manera de escribir esto es a partir de elementos x, como se muestra a continuación

$$C = \{x \in U \mid (x \in A) \wedge (x \in B)\}$$

donde

U representa el conjunto universal
| se lee como "tal que" (a veces se utilizan dos puntos, :, en lugar de |)
$\wedge$ es el **Y lógico**

La expresión para C se lee como C contiene aquellos elementos x que se encuentran en el universo tal que x es un elemento de A y x es un elemento de B. El Y lógico proviene del álgebra booleana. Una expresión integrada por dos operandos conectados por el operador lógico Y, es verdadero si y sólo si ambos operandos son verdaderos. Si A y B no tienen elementos en común, entonces $A \cap B = \varnothing$, donde la letra griega fi, $\varnothing$, representa el **conjunto vacío** o **conjunto nulo**, { }, que es el que no tiene elementos, aunque veces se utiliza la letra griega lambda, Λ, para representarlo. A veces se utilizan otras notaciones para el conjunto universal: el número 1 (y el número 0 para el conjunto vacío). Aunque el conjunto vacío no tiene elementos, aún es un conjunto. Una analogía es un restaurante con un conjunto de clientes. Si nadie viene, el conjunto de clientes está vacío.

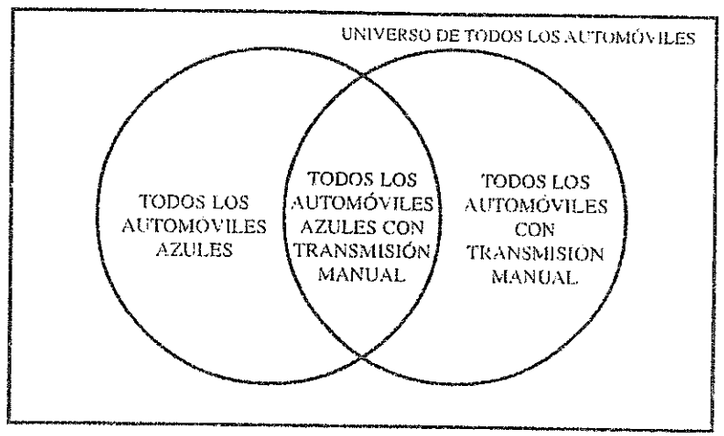


Figura 2.14 Intersección de conjuntos

Otra operación de conjunto es la **unión**, que es el conjunto de todos los elementos en A o B. Se representa de la siguiente forma

$$A \cup B = \{x \in U \mid (x \in A) \vee (x \in B)\}$$

donde

$\cup$ es el operador de conjunto para **unión**
$\vee$ es el operador **logico O.**

El **complemento** del conjunto A es el conjunto de todos los elementos que no están en A.

$$A' = \{x \in U \mid \sim (x \in A)\}$$

donde

el apóstrofe, ', significa complemento de un conjunto
la tilde, ~, es el operador **NO lógico** es decir negación

Los diagramas de Venn para estas operaciones básicas se muestran en la figura 2.15.

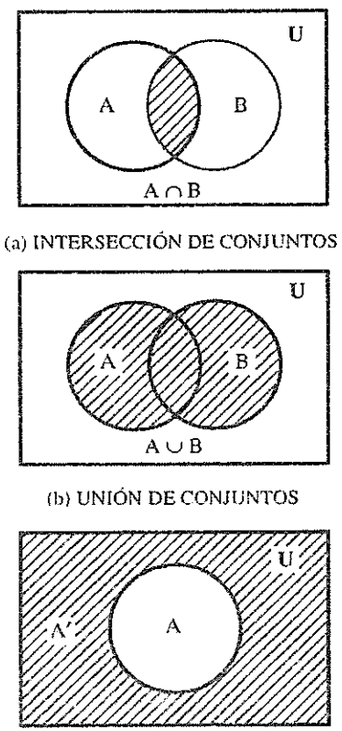


Figura 2.15 Operaciones básicas con conjuntos

2.12 LÓGICA DE PROPOSICIONES

La más antigua y uno de los tipos más simples de **lógica formal** es el silogismo. El término *formal* significa que la lógica se relaciona con la forma de las frases lógicas más que con su

significado. En otras palabras, tiene que ver con la sintaxis más que con la semántica de las frases. Aunque el término *lógica formal* puede sonar intimidatorio, no es más difícil que el álgebra. En realidad, el álgebra es realmente la lógica formal para números. Por ejemplo, supongamos le pidieron resolver el siguiente problema: una escuela tiene 25 computadoras con un total de 60 tarjetas de memoria. Algunas de ellas tienen dos y algunas cuatro tarjetas. ¿Cuántas computadoras hay de cada tipo? La solución puede escribirse algebraicamente como sigue:

```
25 = X + Y
60 = 2X + 4Y
```

que puede resolverse fácilmente: X = 20 y Y = 5.

Ahora considere este problema. Hay 25 animales con un total de 60 patas en un corral. Algunos tienen dos patas y otros cuatro. (Nota: Clyde, nuestro elefante de tres patas está en África, no en el corral). ¿Cuántos animales hay de cada tipo? Como puede ver, las mismas ecuaciones algebraicas se aplican si hablamos de computadoras, animales o cualquier otra cosa. De la misma forma que las ecuaciones algebraicas nos permiten concentrarnos en la manipulación matemática de símbolos sin considerar lo que representan, la lógica formal nos permite concentrarnos en el razonamiento sin confundirnos con los objetos sobre los que razonamos.

Como un ejemplo de lógica formal, tomaremos en cuenta el siguiente silogismo con las palabras sin sentido *squeeg* y *moof*.

```
Premisa:      Todos los squeegs son moofs
Premisa:      John es un squeeg
Conclusión:   John es un moof
```

Aunque las palabras *squeeg* y *moof* no tienen sentido ni significado, la *forma* de este argumento sigue siendo correcta. Es decir, el argumento es **válido** sin importar cuáles palabras se usen, porque el silogismo tiene una forma válida. En realidad, cualquier silogismo con la forma

```
Premisa:      Todas las X son Y
Premisa:      Z es una X
Conclusión:   Z es una Y
```

es válido sin importar qué se sustituye con X, Y y Z. Este ejemplo ilustra esos significados, sin importar la lógica formal. Sólo es importante la forma o apariencia. Este concepto de separar la forma del significado o semántica, es lo que convierte a la lógica en una herramienta poderosa. También hace que la validez de un argumento pueda considerarse objetivamente, sin el prejuicio causado por la semántica. Esto se parece al álgebra en que la corrección de expresiones como X + X = 2X se mantiene sin importar si X representa un entero, manzanas, aviones o cualquier cosa.

Los silogismos aristotélicos fueron el fundamento de la lógica hasta 1847, cuando el matemático inglés George Boole publicó la primer obra que describía la **lógica simbólica**. Aunque Leibnitz había desarrollado su propia versión en el siglo XVII, nunca llegó a ser de uso general. Uno de los nuevos conceptos que Boole propuso, fue una modificación de la idea aristotélica de que el sujeto tiene existencia, llamada ***importancia existencial***. De acuerdo con la concepción clásica de Aristóteles, una proposición como "todas las sirenas nadan bien" no se podría utilizar como premisa o conclusión porque las sirenas no existen. La idea de Boole, ahora llamada *concepción moderna*, hace más relajada esta restricción: su importancia se debe a

que ahora se puede razonar acerca de clases vacías. Por ejemplo, una proposición como "todos los discos que fallan son baratos" no podría utilizarse en un silogismo aristotélico, a menos que hubiera por lo menos un disco que falló.

Otra contribución de Boole fue la definición de un conjunto de **axiomas** integrado por símbolos que representan tanto objetos como clases, y operaciones algebraicas para manipular los símbolos. Son las definiciones fundamentales a partir de las cuales se construyen sistemas lógicos como las matemáticas y la propia lógica. Utilizando únicamente axiomas se pueden elaborar teoremas. Un teorema es una afirmación que se puede probar mostrando cómo se deriva de los axiomas. Entre 1910 y 1913, Whitehead y Russell publicaron su monumental obra de tres volúmenes y 2000 páginas, *Principios matemáticos*, que mostró una lógica formal como base de las matemáticas. Fue recibido como un gran hito porque, aparentemente, pone a las matemáticas sobre bases firmes en lugar de apelar a la intuición, eliminando totalmente el significado de la aritmética y concentrándose, en cambio, en las formas y su manipulación. Debido a esto, las matemáticas se han descrito como una colección de marcas sin sentido en el papel. Sin embargo, en 1931 Gödel probó que los sistemas formales basados en axiomas no siempre resultaban internamente congruentes y, por tanto, libres de contradicciones.

La lógica de proposiciones, también llamada ***cálculo de proposiciones***, es una lógica simbólica para la manipulación de proposiciones. En particular, se ocupa de la manipulación de las **variables lógicas** que representan proposiciones. Aunque muchas personas consideran al cálculo dentro de la perspectiva que inventaron Newton y Leibnitz, la palabra tiene un significado más general; procede de la palabra latina *calculus*, pequeña piedra utilizada para realizar cálculos. Su significado general es un sistema especial para la manipulación de símbolos. Otros términos utilizados para la lógica de proposiciones son **cálculo de afirmaciones** y **cálculo de frases**. Las afirmaciones se clasifican generalmente en cuatro tipos, como se ilustra en la tabla 2.2.

Tipo	Ejemplo
Imperativo	¡Qué te digo!
Interrogativo	¿Qué es eso?
Admirativo	¡Eso es grandioso!
Declarativo	Un cuadrado tiene cuatro lados iguales.

Tabla 2.2 Tipos de afirmaciones

La lógica de proposiciones tiene que ver con el subconjunto de oraciones declarativas que pueden clasificarse como verdaderas o falsas. Una afirmación como "Un cuadrado tiene cuatro lados iguales", tiene un valor de verdad definido como verdadero, mientras que la afirmación "George Washington fue el segundo presidente de Estados Unidos" tiene un valor de verdad falso. A una afirmación en la que es posible determinar su valor de verdad, se le llama ***frase*** o ***proposición***. A una frase también se le llama ***afirmación cerrada***, porque su valor de verdad no está abierto a cuestionamientos.

Si en el prefacio de este libro se afirma "todo en estas páginas es una mentira", esa frase no puede clasificarse como verdadera o falsa. Si es verdadera, entonces dije la verdad en el prefacio, cosa que no puedo hacer; si no es verdad, entonces cada palabra escrita debe ser verdadera, por lo tanto yo miento, lo que tampoco puede ser verdad. Frases de este tipo son conocidas como la Paradoja de Liar. A las afirmaciones que no se puede responder de manera absoluta se les llama **afirmaciones abiertas**. La afirmación "Las espinacas son deliciosas" es también una frase abierta, porque es cierto para unas personas y no lo es para otras. "Él es alto" es una afirmación abierta porque tiene una variable "Él". No es posible asignar valores

verdaderos para abrir afirmaciones hasta que conozcamos a la persona o el caso específico al que se refiere la variable. Otra dificultad con esta afirmación es el significado de la palabra "alto", pues lo que parece alto para algunos no lo parece para otros. Aunque la ambigüedad de "alto" no se puede manejar en el cálculo de proposiciones o de afirmaciones, puede resolverse fácilmente con la lógica de confusión, descrita en el capítulo 5.

Una **afirmación compuesta** está formada por el uso de conectores lógicos o frases individuales. Los conectores comunes de la lógica se muestran en la tabla 2.3.

Conector	Significado
$\wedge$	Y; **conjunción**
$\vee$	O; **disyunción**
~	NO; **negación**
$\rightarrow$	si...entonces; **condicional**
$\leftrightarrow$	sí y sólo si; **doble condicional**

Tabla 2.3 Algunos conectores lógicos

Hablando estrictamente, la negación no es un conector porque es una operación unitaria que aplica al siguiente operando, de modo que en realidad no conecta nada. Tiene mayor prioridad que los otros operadores, así que no es necesario colocarla entre paréntesis. Una frase como $\sim p \wedge q$ significa lo mismo que $(\sim p) \wedge q$.

El condicional es análogo a la flecha de las reglas de producción en las que se expresa como una forma si...entonces. Por ejemplo

```
si está lloviendo entonces lleva una sombrilla
```

se puede poner en la forma

$$p \rightarrow q$$

donde

```
p = está lloviendo
q = lleva una sombrilla
```

Algunas veces se utiliza $\supset$ por $\rightarrow$. Otro término para el condicional es **implicación material.**

El condicional doble, $p \leftrightarrow q$, es equivalente a

$$(p \rightarrow q) \wedge (q \rightarrow p)$$

y es verdadero sólo cuando p y q tienen los mismos valores de verdad. Esto es, $p \leftrightarrow q$ sólo es verdadero cuando ambos son verdaderos o cuando ambos son falsos. El condicional doble tiene los siguientes significados:

```
p si y sólo si q
q si y sólo si p
si p entonces q, si q entonces p
```


Una **tautología** es una frase compuesta que siempre es verdadera, sin importar si sus frases individuales son verdaderas o falsas. Una **contradicción** es una frase compuesta que siempre es falsa. Una frase **condicional** no es una tautología ni una contradicción. A las tautologías y a las contradicciones se les denomina analíticamente verdaderas o analíticamente falsas, respectivamente, porque sus valores de verdad pueden determinarse a partir de sus simples formas. Por ejemplo, la tabla de verdad de $p \vee \sim p$ muestra que es una tautología, mientras que $p \wedge \sim p$ es una contradicción.

Si un condicional es también una tautología entonces se le llama ***implicación*** y tiene el símbolo $\Rightarrow$ en lugar de $\rightarrow$. A un condicional doble que también es una tautología se le llama ***equivalencia lógica*** o ***equivalencia de material*** y se simboliza con $\Leftrightarrow$ o $\equiv$. Dos frases lógicamente equivalentes siempre tienen los mismos valores de verdad. Por ejemplo, $p \equiv \sim\sim p$.

Desafortunadamente, ésta no es la única definición posible para una implicación, ya que hay 16 posibles tablas de verdad para dos variables que pueden tomar valores de verdadero o falso. En efecto, en un artículo se han revisado 11 definiciones diferentes del operador de implicación utilizado en los sistemas expertos (Whalen 85).

El condicional no significa exactamente lo mismo que SI ... ENTONCES en un lenguaje de procedimientos o un sistema experto basado en reglas. En los sistemas de procedimientos y experto, SI ... ENTONCES significa ejecutar las acciones siguiendo el ENTONCES *si* las condiciones del SI son verdaderas. En la lógica, el condicional se define por su tabla de verdad y su significado puede traducirse al lenguaje natural de varias maneras. Por ejemplo, si

$$p \rightarrow q$$

donde p y q son cualquier afirmación, esto puede traducirse como

```
p implica q
si p entonces q
p, sólo si q
p es suficiente para q
q si p
q es necesario para p
```

Como ejemplo, hagamos que p represente "usted tiene 18 años o más" y que q represente "usted puede votar". La $p \rightarrow q$ condicional puede significar

```
usted tiene 18 años o más implica que puede votar
si tiene 18 años o más entonces puede votar
usted tiene 18 años o más, sólo si puede votar
usted tiene 18 años o más, es suficiente para votar
usted puede votar si tiene 18 años o más
usted puede votar si es necesario porque tiene 18 años o más
```

En algunos casos, es necesario un cambio en el orden de las palabras para que sean frases gramaticalmente correctas. El último ejemplo dice que si no ocurre q entonces p tampoco. Se expresa de manera apropiada como "La posibilidad de votar requiere que usted tenga 18 años o más".

En la tabla 2.4 se muestran los valores para los conectores binarios lógicos. Son binarios porque requieren dos operandos. El conector de negación, ~, es un operador unitario en el operando que le sigue, como se muestra en la tabla 2.5.

Un conjunto de conectores lógicos es **adecuado** si cada función de verdad puede representarse utilizando sólo la conectividad del conjunto adecuado. Ejemplos de conjuntos adecuados son {~, ∧, ∨}, {~, ∧}, {~, ∨} y {~, →}.

A un conjunto de un solo elemento se le llama **unitario**. Hay dos conjuntos de unitarios adecuados. Son NO-O y NO-Y. El conjunto NO-O es { ↓ } y el conjunto NO-Y es { | }. Al operador " | " se le llama **negación alternativa**. Se utiliza para negar que p y q son verdaderos. Es decir p | q afirma que por lo menos una de las afirmaciones, p o q, es verdadera. El **operador de negación de unión**, "↓" niega que p o q sean verdaderos. Es decir, p ↓ q afirma que tanto p como q son falsos.

p	q	p ∧ q	p ∨ q	p → q	p ↔ q
V	V	V	V	V	V
V	F	F	V	F	F
F	V	F	V	V	F
F	F	F	F	V	V

Tabla 2.4 Tabla de verdad de los conectores lógicos binarios

p	~p
V	F
F	V

Tabla 2.5 Tabla de verdad de la negación

2.13 LA LÓGICA DE PREDICADO DE PRIMER ORDEN

Aunque la lógica de proposiciones es útil, tiene limitaciones. El principal problema es que la lógica de proposiciones sólo puede tratar con afirmaciones completas. Es decir, no puede examinar la estructura interna de una afirmación. La lógica de proposiciones ni siquiera puede probar la validez de un silogismo como

```
Todos los seres humanos son mortales
Todas las mujeres son seres humanos
Por tanto, todas las mujeres son mortales
```

La **lógica de predicado** se desarrolló con el fin de analizar casos más generales. Su forma más simple es la lógica de predicados de **primer orden**, la base de lenguajes de programación lógicos como PROLOG. En esta sección utilizaremos el término *lógica de predicado* para aludir a la lógica de predicado de primer orden. La lógica de proposiciones es un subconjunto de la lógica de predicado.

La lógica de predicado se relaciona con la estructura interna de las afirmaciones, sobre todo, se relaciona con el uso de palabras especiales llamadas ***cuantificadores***, como "todo", "algo" y "no". Estas palabras son muy importantes porque cuantifican explícitamente otras palabras y hacen más exactas las afirmaciones. Todos los cuantificadores se relacionan con "cuánto" y, por tanto, permiten un alcance más amplio de la expresión que la lógica de proposiciones.

2.14 EL CUANTIFICADOR UNIVERSAL

Una afirmación cuantificada universalmente tiene el mismo valor de verdad para todos los reemplazos en el mismo dominio. La **variable de dominio** el **cuantificador universal** se representa con el símbolo ∀ seguido por uno o más argumentos. El símbolo ∀ se interpreta como "para cada uno" o "para todos". Por ejemplo, en el dominio de los números

```
(∀ x) (x + x = 2x)
```

establece que para cada x (donde x es un número), la afirmación x + x = 2x es verdadera. Si representamos esta frase con el símbolo p, entonces puede expresarse aún más brevemente como:

```
(∀ x) (p)
```

Como otro ejemplo, dejemos que p represente la afirmación "todos los perros son animales", como en

```
(∀ x) (p) ≡ (∀ x) (si x es un perro → x es un animal)
```

La afirmación opuesta es "los perros no son animales" y se escribe como

```
(∀ x) (si x es un perro → ~x es un animal)
```

También puede leerse como

```
Cada perro no es un animal
Todos los perros no son animales
```

Otro ejemplo sería "todos los triángulos son polígonos", y se escribe como sigue:

```
(∀ x) (x es un triángulo → x es un polígono)
```

y se lee "para todo x, si x es un triángulo, entonces x es un polígono". Una manera más corta de escribir afirmaciones lógicas que incluyen predicados es utilizando las **funciones de predicado** para describir las propiedades del sujeto. La afirmación lógica anterior también puede escribirse

```
(∀ x) (triángulo(x) → polígono(x))
```

Las funciones de predicado suelen escribirse en notación más breve utilizando mayúsculas para representar los predicados. Por ejemplo, hagamos que T = triángulo y P = polígono. Entonces la afirmación de triángulo puede escribirse de manera aún más breve como

```
(∀ x) (T(x) → P(x)) o (∀ y) (T(y) → P(y))
```

Observe que cualquier variable puede utilizarse en lugar de las variables de modelo x y y. Un ejemplo más, hagamos que H sea la función de predicado de *ser humano* y M sea la función de *mortal*. Entonces la afirmación de que todos los seres humanos son mortales puede escribirse como:

```
(∀ x) (H(x) → M(x))
```

y se lee que para toda x, si x es ser humano, entonces x es mortal. Esta afirmación de lógica de predicado también puede representarse como una red semántica, tal como se muestra en la figura 2.16. También puede expresarse con reglas como

```
SI x es un ser humano
ENTONCES x es mortal
```

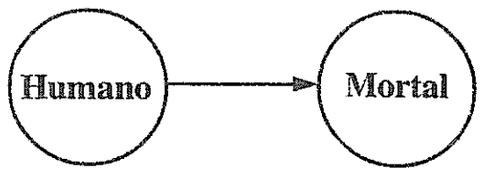


Figura 2.16 Representación de la red semántica de una afirmación de lógica de predicado

El cuantificador universal también puede interpretarse como un conjunto de predicados acerca de casos. Por ejemplo, un perro llamado Sparkler es un caso particular de la clase perros y puede escribirse como

```
Perro(Sparkler)
```

Donde Perro es la función de predicado y Sparkler es un caso.

Una frase de lógica de predicado como

$$(\forall\ x)\ P(x)$$

puede interpretarse casos de a_i como

$$P(a_1) \wedge P(a_2) \wedge P(a_3) \wedge \ldots\ P(a_N)$$

donde los puntos suspensivos indican que el predicado se extiende a todos los miembros de la clase. Esta afirmación indica que el predicado se aplica a todos los casos de la clase.

Es posible utilizar cuantificadores múltiples. Por ejemplo, la ley conmutativa de la adición requiere dos cuantificadores, como en

$$(\forall\ x)\ (\forall\ y)\ (x + y = y + x)$$

que establece que "para todo x y para todo y, la suma de x y y es igual a la suma de y y x".

2.15 EL CUANTIFICADOR EXISTENCIAL

Otro tipo de cuantificador es el **cuantificador existencial**. Éste describe una afirmación como verdadera por lo menos para un miembro del dominio. Esto es una forma restringida del cuantificador universal que indica que una instrucción es verdadera para todos los miembros del dominio. El cuantificador existencial se escribe como ∃ seguido por uno o más archivos. Por ejemplo

$$(\exists\ x)\ (x \cdot x = 1)$$
$$(\exists\ x)\ (elefante(x) \wedge nombre(Clyde))$$

La primera afirmación establece que hay algún x cuyo producto al multiplicarse por sí mismo es igual a 1. La segunda instrucción indica que hay algún elefante con el nombre de Clyde.

El cuantificador existencial puede leerse de varias maneras, como

```
existe
por lo menos un
para algún
hay un
algún
```

Por ejemplo:

$$(\forall\ x)\ (elefante(x) \rightarrow cuatro\text{-}patas(x))$$

dice que todos los elefantes tienen cuatro patas. Sin embargo, la afirmación de que algunos elementos tienen tres patas se escribe con un Y lógico y con cuantificadores universales como sigue

$$(\exists\ x)\ (elefante(x) \wedge tres\text{-}patas(x))$$

Así como el cuantificador universal puede expresarse como conjunción, el cuantificador existencial puede expresarse como una disyunción de casos, a_i.

$$P(a_1) \vee P(a_2) \vee P(a_3) \vee \ldots\ P(a_N)$$

Las afirmaciones cuantificadas y sus negaciones para el ejemplo en que P representa "elefantes y mamíferos" se muestra en la tabla 2.6. Los números entre paréntesis sólo sirven para identificar los ejemplos para el análisis siguiente.

Ejemplo	Significado
(1a) (∀ x)(P)	Todos los elefantes son mamíferos
(1b) (∃ x)(-P)	Algunos elefantes no son mamíferos
(2a) (∃ x)(P)	Algunos elefantes son mamíferos
(2b) (∀ x)(-P)	Ningún elefante es mamífero

Tabla 2.6 Ejemplos de cuantificadores negados

Los ejemplos (1a) y (1b) son negaciones entre sí, como (2a) y (2b). Observe que la negación de una afirmación cuantificada universalmente de (1a) es una afirmación cuantificada existencialmente de la negación de P, como se muestra en (1b). De igual manera, la negación de una afirmación cuantificada existencialmente de (2a) es la afirmación cuantificada universalmente de la negación de P, como se muestra en (2 b).

2.16 CUANTIFICADORES Y CONJUNTOS

Los cuantificadores pueden utilizarse para definir conjuntos en un universo, U, como se muestra en la tabla 2.7.

La relación de que A es un subconjunto propio de B, escrito como $A \subset B$, significa que a pesar de que todos los elementos de A están en B, hay por lo menos un elemento de B que no se encuentra en A. Si E representa elefantes y M representa mamíferos, la relación de conjunto

$$E \subset M$$

establece que todos los elefantes son mamíferos, pero no todos los mamíferos son elefantes. Si G = gris y C = cuatro-patas, la afirmación de que todos los elefantes grises con cuatro patas son mamíferos se escribe

$$(E \cap G \cap C) \subset M$$

Expresión de conjunto	Equivalente lógico
$A = B$	$\forall x\,(x \in A \leftrightarrow x \in B)$
$A \subseteq B$	$\forall x\,(x \in A \rightarrow x \in B)$
$A \cap B$	$\forall x\,(x \in A \wedge x \in B)$
$A \cup B$	$\forall x\,(x \in A \vee x \in B)$
A'	$\forall x\,(x \in U \mid \sim(x \in A))$
U (Universo)	V (Verdadero)
$\varnothing$ (conjunto vacío	F (Falso)

Tabla 2.7 Algunas expresiones de conjuntos y sus equivalentes lógicos

Utilizando las siguientes definiciones, se muestran algunos ejemplos de afirmaciones cuantificadas

```
E = elefantes
R = reptiles
G = gris
C = cuatro-patas
P = perros
M = mamíferos
```

Ningún elefante es reptil
$E \cap R = \varnothing$
Algunos elefantes son grises
$E \cap G \neq \varnothing$
Ningún elefante es gris
$E \cap G = \varnothing$
Algunos elefantes no son grises
$E \cap G' \neq \varnothing$
Todos los elefantes son grises y de cuatro patas
$E \subset (G \cap C)$
Todos los elefantes y perros son mamíferos
$(E \cup P) \subset M$
Algunos elefantes tienen cuatro patas y son grises
$(E \cap F \cap G) \neq \varnothing$

Como otra analogía de los conjuntos y formas lógicas, las leyes de De Morgan se muestran en la tabla 2.8, donde el símbolo de equivalencia, ≡, el (bicondicional) significa que la afirmación de la izquierda tiene el mismo valor de verdad que la de la derecha. Es decir, las afirmaciones son equivalentes.

Conjunto	Lógica
$(A \cup B)' \equiv A' \cup B'$	$\sim(p \wedge q) \equiv \sim p \vee \sim q$
$(A \cap B)' \equiv A' \cap B'$	$\sim(p \vee q) \equiv \sim p \wedge \sim q$

Tabla 2.8 Conjunto y formas lógicas de las leyes de De Morgan

2.17 LIMITACIONES DE LA LÓGICA DE PREDICADO

Aunque esta lógica es muy útil en muchas situaciones, hay algunos tipos de afirmaciones que no pueden expresarse en ella utilizando los cuantificadores universal y existencial (Rescher 64). Por ejemplo, la siguiente afirmación no puede expresarse con lógica de predicado:

```
La mayoría de la clase recibió dieces
```

En esta afirmación, el cuantificador *mayoría* significa más de la mitad, pero no puede expresarse con los cuantificadores universal y existencial. Para implantar *mayoría* una lógica debe proporcionar algunas predicados para conteo, como la lógica borrosa, descrita en el capítulo 5. Otra limitante de esta lógica es en la expresión de cosas que a veces son verdaderas, pero no siempre. Este problema también puede resolverse con lógica borrosa. Sin embargo, la introducción de un conteo también produce más complicaciones en el sistema lógico y la hace de manera más parecida a las matemáticas.

2.18 RESUMEN

En este capítulo revisamos los elementos de la teoría del conocimiento y las técnicas para representarlo. La representación del conocimiento es de la mayor importancia para los sistemas expertos. El conocimiento puede clasificarse de varias maneras, como *a priori*, *a posteriori*, de procedimiento, declarativo y tácito. Las reglas de producción, las redes semánticas, los esquemas, los marcos y la lógica son métodos comunes para la representación del conocimiento en los sistemas expertos. Cada uno de estos paradigmas tiene ventajas y desventajas. Antes de diseñar un sistema experto, debe decidir cuál es el mejor paradigma para el problema que habrá de resolverse. En lugar de tratar de usar una herramienta para todos los problemas, elija la mejor herramienta para el problema particular.

PROBLEMAS

2.1 Dibuje una red semántica para computadoras utilizando vínculos UN-TIPO-DE y ES-UN. Considere las clases microcomputadora, minicomputadora, supercomputadora, sistema de cómputo, dedicado, de propósito general, a nivel de tarjeta, computadora-en-un-chip, de un solo procesador y de varios procesadores. Incluya casos específicos.

2.2 Dibuje una red semántica para comunicación con computadora utilizando vínculos UN-TIPO-DE y ES-UN. Considere las clases de red de área local, red de área amplia, token ring, estrella, centralizada, descentralizada, distribuida, modems, telecomunicaciones, tableros de noticias y correo electrónico. Incluya casos específicos.

2.3 Dibuje un sistema de marco para el edificio donde asiste a clases. Considere oficinas, salones, laboratorios, etcétera. Incluya casos con ranuras rellenas para cada tipo de marco, como oficina y salón de clases.

2.4 Dibuje un sistema de marco de acción que explique qué hacer en caso de falla del hardware de su sistema de cómputo. Considere la descompostura del disco, la falla en el suministro de electricidad, problemas del CPU y la memoria.

2.5 Dibuje el diagrama de Venn y escriba la expresión de conjuntos para lo siguiente:

(a) el O exclusivo de dos conjuntos, A y B, contiene todos los elementos que hay en uno, pero no en ambos conjuntos. Al O exclusivo también se le llama ***diferencia de conjunto simétrico*** y se simboliza con "/". Por ejemplo,

{1,2}/{2/3}={1,3}

(b) la **resta de conjuntos** de dos conjuntos, simbolizado por "–" contiene todos los elementos del primer conjunto que no están también en el segundo. Por ejemplo,

{1,2}-{2/3}={1}

donde {1,2} es el primer conjunto y {2,3} es el segundo.

2.6 Escriba las tablas de verdad y determine cuál de las siguientes son tautologías, contradicciones o **afirmaciones contingentes**, y cuales no lo son. Para (a) y (b), primero exprese las afirmaciones con símbolos lógicos y conectores.

(a) Si I pasa este curso y saca '10' entonces
I pasa este curso o I saca '10'.
(b) Si I pasa este curso luego I saca '10'
e
I pasa este curso e I no saca un '10'
(c) $(A \wedge \sim B \rightarrow (C \wedge \sim C)) \rightarrow (A \rightarrow B)$
(d) $(A \rightarrow B) \wedge (\sim B \vee C) \wedge (A \wedge \sim C)$
(e) $A \rightarrow \sim B$

2.7 Dos afirmaciones son equivalentes lógicamente si y sólo si tienen el mismo valor de verdad. Por tanto, si A y B son afirmaciones, la afirmación bicondicional $A \leftrightarrow B$ o la equivalencia $A \equiv B$ será verdadera en todos los casos, dando lugar a una tautología. Determine (a) si las dos afirmaciones de abajo son equivalentes lógicamente al escribirlas utilizando símbolos lógicos y (b) si la tabla de verdad de su bicondicional es una tautología.

Si come banana split entonces no puede comer pay
Si come pay, entonces no puede comer banana split

2.8 Escriba el equivalente lógico correspondiente a la resta de conjuntos y la resta de conjuntos simétricos.

2.9 Demuestre que las siguientes son identidades para cualquier conjunto A, B y C y para $\varnothing$ como conjunto nulo.

(a) $(A \cup B) \equiv (B \cup A)$
(b) $(A \cup B) \cup C \equiv A \cup (B \cup C)$
(c) $A \cup \varnothing \equiv A$
(d) $A \cap B \equiv B \cap A$
(e) $A \cap A' \equiv \varnothing$

2.10 Escriba lo siguiente en forma cuantificada:

(a) Todos los perros son mamíferos
(b) Ningún perro es un elefante
(c) Algunos programas tienen errores
(d) Ninguno de mis programas tiene errores
(e) Todos sus programas tienen errores

2.11 El **conjunto de poder**, P(S), de un conjunto S es el conjunto de todos los elementos que son subconjuntos de S. P(S) siempre tendrá por lo menos el conjunto nulo, $\varnothing$, y S como miembros.

(a) Encuentre el conjunto de poder de A = {2, 4, 6}.
(b) Para un conjunto con N elementos, ¿cuántos elementos tiene el conjunto de poder?

2.12 (a) Elabore la tabla de verdad para lo siguiente:

Significado	Definición
o p o q	$(p \vee q) \wedge \sim(p \wedge q)$
ni p ni q	$\sim(p \vee q)$
p a menos que q	$\sim q \rightarrow p$
p porque q	$(p \wedge q) \wedge (q \rightarrow p)$
no p es q	$p \rightarrow \sim p$

(b) Demuestre que $(p \vee q) \wedge \sim(p \wedge q) \equiv p/q$, donde "/" es el O exclusivo.

2.13 (a) Escriba las tablas de verdad de NO O y de NO Y.
(b) Pruebe que { ↓ } y { | } son conjuntos adecuados para expresar ~, ∧ y ∨ en términos de ↓ y luego de | al construir tablas de verdad para mostrar las equivalencias lógicas como sigue.

$\sim\sim p \equiv p$
$(p \wedge q) \equiv (p \downarrow p) \downarrow (q \downarrow q)$
$\sim p \equiv p \mid p$
$(p \vee q) \equiv (p \mid p) \mid (q \mid q)$

(c) Como $p \rightarrow q \equiv \sim(p \wedge \sim q)$, exprese $p \rightarrow q$ a partir de ↓.
(d) ¿Cuáles son las ventajas y cuáles las desventajas de utilizar conjuntos unitarios adecuados desde el punto de vista de (i) la notación y (ii) la construcción de chips para circuitos electrónicos?

2.14 ¿Cuáles son las ventajas y las desventajas de diseñar un sistema experto con conocimiento sobre varios dominios?

BIBLIOGRAFÍA

(Angeles 81). Peter A. Angeles, *Dictionary of Philosophy*, Barnes & Noble Books, 1981.

(Barr 81). Arron Barr y Edward A. Feigenbaum, eds., *The Handbook of Artificial Intelligence*, Vol. 1, William Kaufmann, Inc., 1981.

(Brachman 80). R. J. Brachman and B. C. Smith, eds., *SIGART Newsletter, Special Issue on Knowledge Representation*, **70**, febrero de 1980.

(Brachman 83). R. J. Brachman, R. E. Fikes, y H. L. Levesque, "KRYPTON: A Functional Approach to Knowledge Representation", *IEEE Computer*, **16**, p. 67, 1983.

(Brachman 85). Ronald J. Brachman, "I Lied about the Trees", *AI Magazine*, 6, (3), pp. 80-93, otoño de 1985.

(Finin 86). Tim Finin, "Understanding Frame Languages, Parte I", *AI Expert*, pp. 44-50, noviembre de 1986.

(Jacson 86). Peter Jackson, *Introduction to Expert Ssytems*, Addison-Wesley Pub., 1986.

(Lenat 77). Doug B. Lenat, "Automated Theory Formation in Mathematics", *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 1977.

(McGettrick 80). Andrew D. McGettrick, *The Definitions of Programming Languages*, Cambridge University Press, 1980.

(Minsky 75). Marvin Minsky, "A Frameworks for Representing Knowledge", *in The Psychology of Computer Vision*, Patrick Winston, de., McGraw-Hill, pp. 211-217, 1975.

(Quillian 68). M. R. Quillian, "Semantic Memory", *Semantic Information Processing*, Marvin Minsky, ed., MIT Press, pp. 227-270, 1968.

(Rescher 64). Nicolas Rescher, *Introduction to Logic*, St. Marrtin's Press, pp. 258-259, 1964.

(Rissland 78). E. L. Rissland, "Understanding Understanding Mathematics", *Cognitive Science*, **2**, pp. 361-383, 1978.

(Schank 77). Roger C. Shank y R. P. Abelson, *Scripts, Plans, Goals, and Understanding*, Lawrence Erlbaum, 1977.

(Sowa 84). J. F. Sowa, *Conceptual Structures: Information Processing in Mind and Machine*, Addison-Wesley Pub., 1984.

(Staugaard 87). Andrew Staugaard Jr., *Robotics and AI: An Introduction to Applied Machine Intelligence*, Prentice-Hall, 1987.

(Stillings 87). Neil A. Stillings, *et al.*, *Cognitive Science: An Introduction*, MIT Press, 1987.

(Whalen 85). Thomas Whalen and Brian Schott, "Alternative Logics for Approximate Reasoning in Expert Systems: A Comparative Study", *Int. J. Man-Machine Studies*, **22**, pp. 327-346, 1985.

(Winston 84). Patrick Henry Winston, *Artificial Intelligence*, 2nd ed., Addison-Wesley Pub., 1984.

(Woods 83). W. A. Woods, "What's Important about Kwnowledge Representation?", *IEEE Computer*, **16**, p. 22, 1983.

CAPÍTULO 3

Métodos de inferencia

3.1 INTRODUCCIÓN

En este capítulo analizaremos varios métodos de razonamiento o inferencia, tema que reviste particular importancia en los sistemas expertos porque el razonamiento es la técnica normal que utilizan los sistemas expertos para resolver problemas. Es decir, se usan regularmente cuando no existe una solución algorítmica o cuando ésta es inadecuada y el razonamiento ofrece la única posibilidad de obtener una solución.

3.2 ÁRBOLES, REJILLAS Y GRÁFICAS

Un **árbol** es un estructura jerárquica de datos conformada por **nodos** que almacenan información o conocimiento, y por **ramas**, que conectan a los nodos. A veces a las ramas se les denomina ***vínculos*** o ***bordes***, y a los nodos ***vértices***. En la figura 3.1 se muestra un árbol binario general que tiene cero, una o dos ramas por nodo. En un **árbol orientado**, el **nodo raíz** ocupa el lugar más alto de la **jerarquía**, y las **hojas**, el más bajo. Un árbol puede considerarse como una clase especial de red semántica en la que cualquier nodo, excepto el raíz, tiene exactamente un nodo **padre** y cero o más nodos **hijo**. Para el tipo usual de árbol binario hay un máximo de dos hijos por nodo y los nodos hijo de la derecha son diferentes a los de la izquierda.

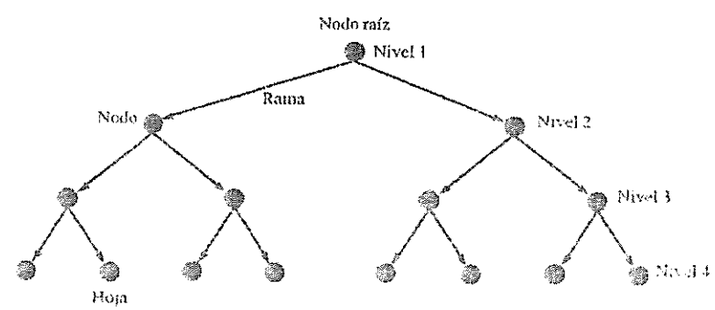


Figura 3.1 Árbol binario

Si un nodo tiene más de un padre, se encuentra en una red. En la figura 3.1, se observa que sólo hay una secuencia de bordes, o **ruta**, de la raíz a cualquier nodo; porque no es posible moverse contra una flecha. En los árboles orientados todas las flechas apuntan hacia abajo.

Los árboles son un caso especial de una estructura general matemática llamada **gráfica**. El término *red* suele utilizarse como sinónimo de *gráfica* cuando se describe un ejemplo particular de gráfica como una red telefónica. Una gráfica puede tener cero o más vínculos entre los nodos y ninguna distinción entre padres e hijos. Un ejemplo sencillo de gráfica es un mapa: las ciudades son los nodos y los vínculos son los caminos. Los vínculos pueden tener flechas o direcciones asociadas con ellos y un **peso** para diferenciarse; se puede hacer una analogía con las calles de un solo sentido que tienen límites de peso para los camiones que circulan por ellas. Los pesos de una gráfica pueden representar cualquier tipo de información. Si la gráfica es la ruta de una línea aérea, los pesos pueden ser los kilómetros entre las ciudades, el costo del vuelo, el consumo de combustible, etcétera.

Un sistema neuronal artificial es otro ejemplo de una gráfica con ciclos, porque durante el entrenamiento hay retroalimentación de la información de una capa a la otra de la red, lo que modifica los pesos. Una gráfica simple no tiene vínculos que se encuentren inmediatamente detrás del propio nodo, como se muestra en la figura 3.2(a). Un **circuito** o **ciclo** es una ruta que recorre una gráfica y que inicia y termina en el mismo nodo, como la ruta ABCA de la figura 3.2(a). Como su nombre lo indica, una gráfica **acíclica** no tiene ciclos, mientras que una **gráfica conectada** tiene vínculos con todos sus nodos. En la figura 3.2(c) se muestra una gráfica con vínculos dirigidos, llamada **digráfica,** y un **autolazo**. Una gráfica acíclica con dirección es una **rejilla**, y se ejemplifica en la figura 3(d); y un árbol con una sola ruta desde la raíz hasta su única hoja es un **árbol degenerado**. Los árboles degenerados binarios de tres nodos se muestran en la figura 3.2(e). Por lo general, las flechas *no* se muestran explícitamente en un árbol, porque se supone que éstas apuntan hacia abajo.

Árboles y rejillas son útiles para clasificar objetos a causa de su naturaleza jerárquica. Un ejemplo es un árbol familiar que muestra las relaciones y la ascendencia de las personas relacionadas. Otra aplicación de los árboles y las rejillas es la toma de decisiones, en estos casos se les llama ***árboles*** o ***rejillas de decisión;*** en lo sucesivo se utilizará el término ***estructura*** para denominarlos a ambos. Una estructura de decisión es tanto un esquema de representación del conocimiento como un método de razonamiento sobre su conocimiento. En la figura 3.3, se muestra un ejemplo de árbol de decisión para clasificar animales. Este ejemplo sirve para el juego clásico de las 20 preguntas: los nodos contienen preguntas, las ramas "sí" o "no" las respuestas y las hojas las suposiciones de cuál animal se trata.

A diferencia de los árboles de ciencias de la computación, los árboles de clasificación pueden dibujarse con la raíz hacia abajo. En la figura 3.4 se muestra una pequeña parte de un árbol de decisión para clasificar frambuesas, en él no se muestra la raíz, que tiene una rama hacia el nodo "Hojas simples" y otra hacia el nodo "Hojas compuestas". El proceso de decisión inicia en la parte inferior, al identificar los rasgos más notorios, como si las hojas son simples o compuestas, los detalles más específicos, que requieren una observación más cercana, se usan a medida que subimos por el árbol. Es decir, los conjuntos más grandes de alternativas se examinan primero y después el proceso de decisión comienza a reducir las posibilidades a grupos más pequeños. Esta es una buena forma de organizar las decisiones tomando en consideración el tiempo y el esfuerzo necesarios para llevar a cabo observaciones más detalladas.

Si las decisiones son binarias, un árbol de decisión binario es a la vez fácil de construir y muy eficiente. Cualquier pregunta baja un nivel en el árbol: una pregunta puede decidir una de dos posibles respuestas, dos preguntas pueden decidir una de cuatro posibles respuestas, tres preguntas pueden decidir una de ocho posibles respuestas, etc. Si se construye un árbol binario de modo tal que todas las hojas sean respuestas y todos los nodos dirigidos hacia

abajo sean preguntas, puede haber un máximo de 2^N respuestas para N preguntas. Por ejemplo, diez preguntas pueden clasificar uno de 1,024 animales, mientras que 20 preguntas pueden clasificar una de 1,048,576 posibles respuestas.

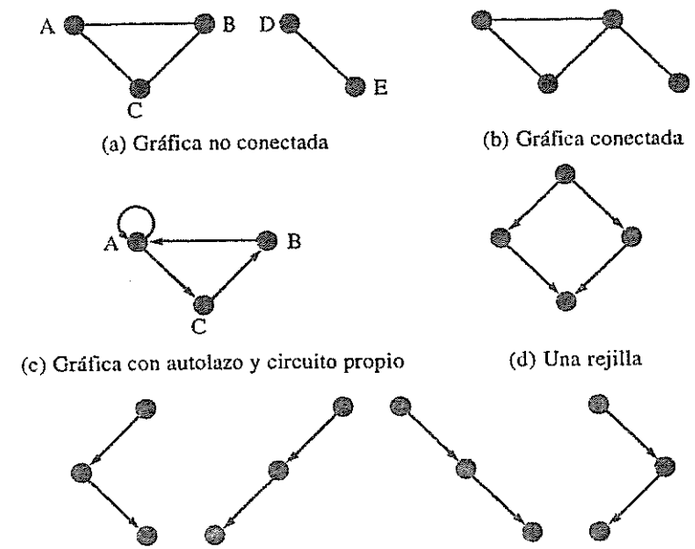


Figura 3.2 Gráficas simples

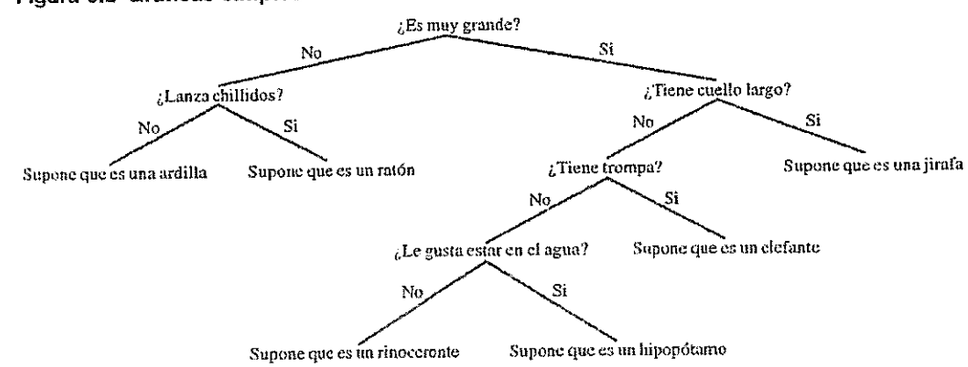


Figura 3.3 Árbol de decisión mostrando conocimiento acerca de animales

Otra característica útil de los árboles de decisión es que pueden realizar un **autoaprendizaje**. Si la suposición es errónea, puede llamarse a un procedimiento para pedir al usuario una nueva pregunta de clasificación correcta y las respuestas a las opciones "sí" y "no". Deben

crearse dinámicamente nuevos nodos, ramas y hojas y añadirse al árbol. En el programa original Animales, escrito en BASIC, el conocimiento estaba almacenado en instrucciones DATA; cuando el usuario enseñaba al programa un nuevo animal, tenía lugar el aprendizaje automático mientras el programa generaba nuevas instrucciones DATA que contenían información acerca del nuevo animal. En Pascal y otros lenguajes con soporte de apuntadores, el conocimiento sobre el animal puede almacenarse en árboles. Con la herramienta CLIPS para sistemas expertos, pueden construirse automáticamente nuevas reglas mientras el programa aprende nuevo conocimiento (Giarratano 92). Esta **adquisición automatizada de conocimiento** es muy útil porque puede evitar el cuello de botella de la adquisición de conocimiento descrito en el capítulo 1.

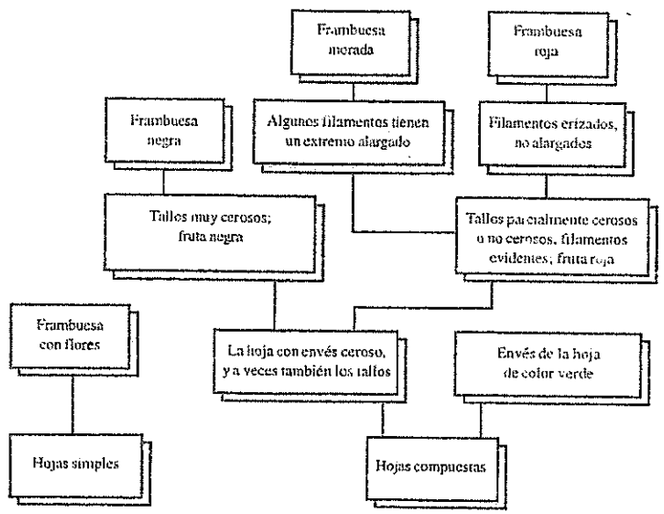


Figura 3.4 Fragmento de un árbol de decisión para clasificar varias especies de frambuesas

Las estructuras de decisión pueden traducirse a reglas de producción en forma mecánica. Esto puede lograrse fácilmente a partir de una búsqueda primero a lo ancho de la estructura y de la generación de reglas SI... ENTONCES en cada nodo. Por ejemplo, el árbol de decisión de la figura 3.3 podría traducirse a reglas de la siguiente manera:

```
SI PREGUNTA = "¿ES MUY GRANDE?" Y RESPUESTA = "NO"
   ENTONCES PREGUNTA := "¿LANZA CHILLIDOS?"

SI PREGUNTA = "¿ES MUY GRANDE?" Y RESPUESTA = "SÍ"
   ENTONCES PREGUNTA := "¿TIENE UN CUELLO LARGO?"
```

y así para los otros nodos. Un nodo de hoja podría generar una RESPUESTA en lugar de una pregunta. Los procedimientos apropiados también consultarían al usuario para obtener una entrada y podrían construir nuevos nodos si ésta es errónea.

A pesar de que las estructuras de decisión son importantes herramientas de clasificación, están limitadas porque no pueden tratar con variables, mientras que un sistema experto sí lo puede hacer. Los sistemas expertos, más que simples clasificadores, son herramientas de propósito general.

3.3 ESTADO Y ESPACIOS DE PROBLEMA

Los diagramas pueden aplicarse a muchos problemas prácticos. Un método útil para describir el comportamiento de un objeto, es definir un gráfica llamada **espacio de estado.** Un *estado* es una colección de características que pueden usarse para definir el status o **estado** de un objeto. El espacio de estado es el conjunto de estados que muestran las **transiciones** entre los estados que puede experimentar el objeto. Una transición lleva un objeto de un estado a otro.

Ejemplos de espacios de estado

Como un simple ejemplo de espacios de estado, consideremos la compra de una bebida gaseosa en una máquina expendedora: cuando se depositan monedas en la máquina, ésta hace una transición de un estado a otro. En la figura 3.5 se ilustra el espacio de estado, suponiendo que sólo están disponibles monedas de 25 y de 5 centavos, y que se requieren 55 centavos para obtener una bebida. Si consideramos monedas de otras denominaciones, como de 10 y 50 centavos, se complica más la gráfica y no se muestra aquí.

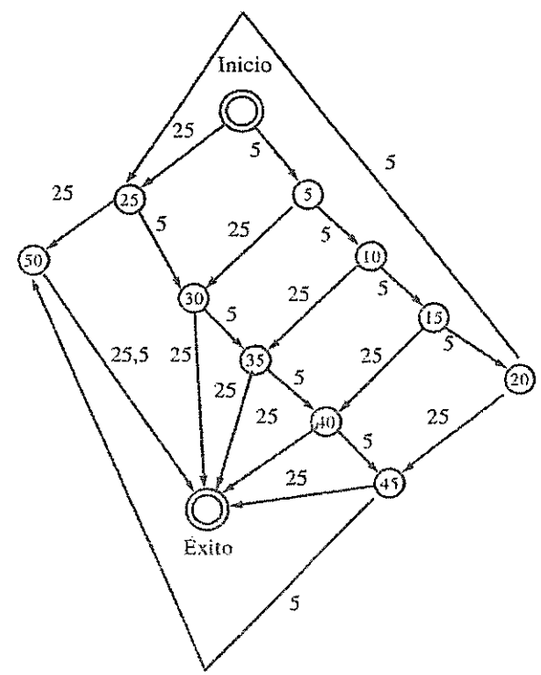


Figura 3.5
Gráfica de estado para una máquina expendedora de bebidas gaseosas que acepta monedas de 25 y 5 centavos

Los estados de inicio y éxito se dibujan como círculos dobles para facilitar su identificación. Los estados se muestran como círculos y las transiciones posibles a otros estados se muestran como flechas. Observe que esta gráfica es una gráfica con pesos, donde éstos son las monedas que pueden entrar a la máquina en cada estado.

A esta gráfica también se le llama gráfica de **máquina de estado finito**, porque describe el número finito de *estados* de una máquina. El término *máquina* se usa en un sentido muy general, puede ser un objeto real, un algoritmo, un concepto, etc. Asociadas con cada *estado* hay acciones que la conducen a otro. En cualquier momento, la máquina puede estar en un solo estado. A medida que acepta la entrada a un *estado*, pasa de ese estado a otro. Si se dan las entradas correctas, pasará del inicio al éxito o estado final. Si un estado no está diseñado para aceptar cierta entrada, la máquina se detendrá en él. Por ejemplo, la máquina de bebidas gaseosas no tiene la capacidad de aceptar monedas de diez centavos, si alguien pone una de estas monedas en la máquina, la respuesta no está definida. Un buen diseño incluirá la posibilidad de entradas no válidas para cada estado y proporcionará transiciones a un estado de error. Éste es diseñado para proporcionar mensajes de error apropiados y llevar a cabo cualquier acción necesaria.

Las máquinas de estado finito se usan con frecuencia en compiladores y otros programas para determinar la validez de una entrada. Por ejemplo, la figura 3.6 muestra parte de una máquina de estado finito para probar la validez de cadenas de entrada. Los caracteres de la entrada son examinados uno por uno y sólo serán aceptadas las cadenas de caracteres MIENTRAS, ESCRIBIR y EMPEZAR. Las flechas se muestran desde el estado EMPEZAR para una entrada exitosa y también para una entrada errónea; en este caso, avanza hacia el estado de error. Por eficiencia, algunos estados como el señalado por "L" y "T" se usan para probar a MIENTRAS y a ESCRIBIR.

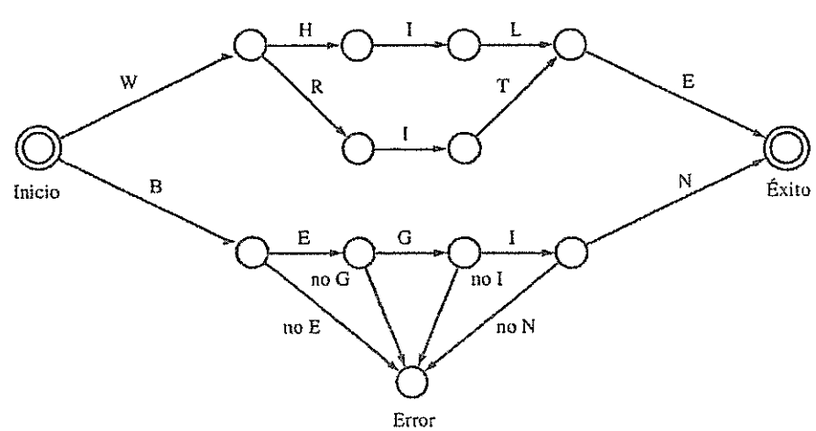


Figura 3.6 Parte de un máquina de estado finito para determinar cadenas válidas MIENTRAS, ESCRIBIR y EMPEZAR.
Nota: ***Sólo se muestran algunas transiciones al estado de error.***

Los diagramas de estado también son útiles en la descripción de soluciones a problemas. En este tipo de aplicaciones se puede considerar al espacio de estado como un **espacio de problema**, en el que algunos estados corresponden a etapas intermedias de la solución de

problemas y algunos estados corresponden a las respuestas. En un espacio de problema puede haber varios estados de éxito que correspondan a soluciones posibles. Para encontrar la solución a un problema dentro de su espacio de problema, es necesario encontrar una ruta válida desde el inicio (afirmación del problema) hasta el éxito (respuesta). El árbol de decisión de animales puede verse como un espacio de problema donde las respuestas sí/no a las preguntas determinan el estado de transición.

Otro ejemplo de espacio de problema sucede en el problema clásico del mono y los plátanos que se muestra en la figura 3.7. El problema consiste en dar instrucciones a un mono para decirle cómo alcanzar algunos plátanos que cuelgan del techo. Como los frutos están fuera de su alcance, pero dentro del cuarto hay un sofá y una escalera. La configuración inicial típica es la del mono en el sofá. Las instrucciones pueden ser:

```
salta del sofá
ve a la escalera
pon la escalera bajo los plátanos
trepa la escalera
agarra los plátanos
```

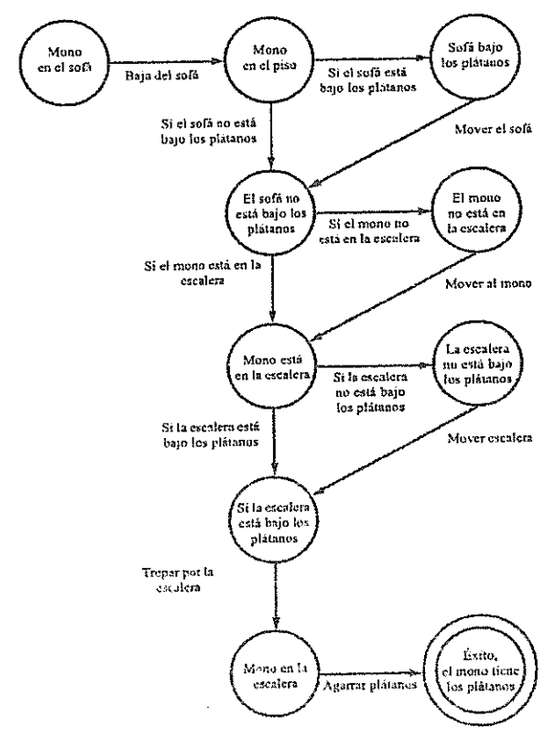
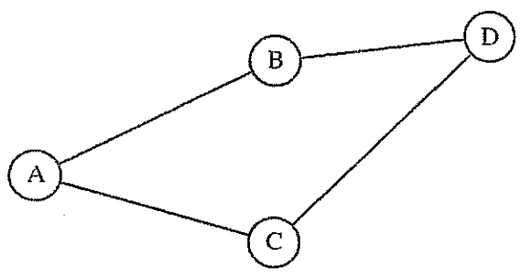


Figura 3.7 El espacio de estado para el problema del mono y los plátanos

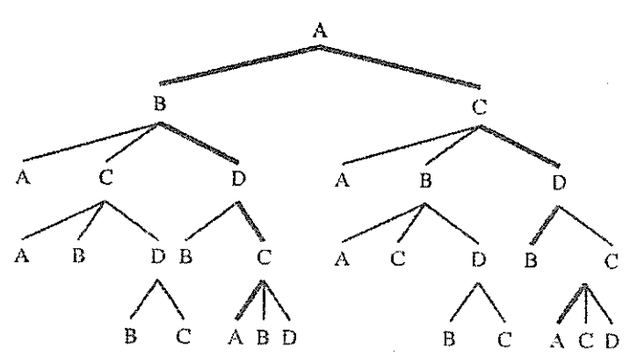
Las instrucciones variarán de acuerdo con las posiciones iniciales del mono, el sofá y la escalera. Como hay varios estados iniciales, no se muestra aquí el doble círculo especial de inicio. Por ejemplo, otro estado de inicio posible es con el mono en el sofá debajo de los plátanos, de esta manera, el mono tendrá entonces que hacer el sillón a un lado antes de poner la escalera debajo de los plátanos. En el estado inicial más simple, el mono ya se encuentra en la escalera debajo de las bananas.

Aunque este problema parece obvio para un ser humano, requiere una considerable cantidad de razonamiento. Una aplicación práctica de un sistema de razonamiento como éste, es la de dar a un robot instrucciones relacionadas con la realización de una tarea. En lugar de suponer que todos los objetos del ambiente se mantienen fijos en un lugar, una solución general es un sistema de razonamiento que puede tratar con diversas situaciones.

Otra aplicación útil de los diagramas es la exploración de rutas para encontrar la solución a un problema. En la figura 3.8(a) se muestra una red simple para el problema del agente viajero; en este ejemplo, se supone que el problema está en encontrar una ruta completa para visitar todos los nodos desde el nodo A. Como es común en estos casos, ningún nodo puede visitarse dos veces. En la figura 3.8(b) se muestran todas las rutas posibles que empiezan desde el nodo A, en forma de un árbol. Las rutas correctas, ABDCA y ACDBA, se muestran con bordes gruesos en esta gráfica.



(a) Gráfica de un problema del agente viajero



(b) Rutas de búsqueda (las rutas óptimas se muestran con bordes gruesos)

Figura 3.8 Problema del vendedor viajero

Dependiendo del algoritmo de búsqueda, es probable que la exploración de rutas para encontrar la correcta requiera una cantidad considerable de rastreo hacia atrás. Por ejemplo, puede investigarse primero la ruta ABA sin éxito y después rastrear hacia atrás hasta B. Desde B se buscará sin éxito en las rutas CA, CB, CDB y CDC. A continuación se buscará sin éxito en la ruta BDB hasta que se encuentre la primera ruta correcta, ABDCA.

Espacios de problemas con estructura nociva

Una aplicación útil de los espacios de estado se encuentra al distinguir los problemas con estructura nociva (Pople 82). En el capítulo 1 se definió que un problema con estructura nociva es aquel asociado con la incertidumbre. Ésta puede especificarse con mayor precisión usando un espacio de problema.

Como ejemplo de un problema con estructura nociva, consideremos otra vez el caso de una persona que piensa viajar y visita a un agente de viajes, que se analizó en el capítulo 1. En la tabla 3.1 se enumeran algunas características de este problema con estructura nociva como un espacio de problema, indicado por las respuestas de la persona a las preguntas del agente de viajes.

Comparando la tabla 3.1 con la 1.10 (página 19), se verá que el concepto de un espacio de problema permite especificar con mayor precisión las características de un problema con estructura nociva. Es esencial diferenciar estos parámetros con precisión para determinar si una solución es factible y, si lo es, qué es lo que requiere. Un problema no tiene necesariamente una estructura nociva sólo por presentar una, varias o incluso todas estas características, porque depende mucho del rigor. Por ejemplo, todos los problemas de comprobación de teoremas tienen un número infinito de posibles soluciones, pero esto no hace que la comprobación de teoremas sea un problema de estructura nociva.

Como puede verse en la tabla 3.1, existen muchas incertidumbres, y aún así los agentes de viajes tratan con ellas todos los días. Aunque no todos los casos pueden ser tan malos como éste, indica por qué sería muy difícil una solución algorítmica.

Característica	Respuesta
Objetivo no explícito	Estoy pensando en ir a alguna parte
Espacio de problema no delimitado	No estoy seguro de adónde ir
Estados de problema no discretos	Solo quiero viajar; el destino no es importante
Estados intermedios difíciles de alcanzar	No tengo suficiente dinero para ir
Operadores de estado desconocidos	No sé cómo conseguir el dinero
Restricciones de tiempo	Debo ir pronto

Tabla 3.1 Ejemplo de un problema con estructura nociva para un viaje

Un problema bien formado es en el que conocemos el problema explícito, el objetivo y los operadores que pueden aplicarse para ir de un estado al otro; y se le considera **determinístico** porque al aplicar un operador a un estado, estamos seguros del estado siguiente, el espacio de problema está delimitado y los estados son discretos. Esto significa que hay un número finito de estados y que cada uno está bien definido.

En el problema del viaje los estados no están delimitados porque los destinos posibles son infinitos. Una situación análoga puede suceder con un medidor analógico, que indica un número infinito de lecturas posibles. Si consideramos que cada lectura del medidor es un estado, entonces hay un número infinito de estados y éstos no están bien definidos porque corresponden a los números reales. Como hay un número infinito de números reales entre dos números reales cualesquiera, los estados no son discretos porque el siguiente estado sólo difiere infinitesimalmente. Por el contrario, las lecturas de un medidor digital están delimitadas y son discretas.

3.4 ÁRBOLES Y OBJETIVOS Y-O

Muchos tipos de sistemas expertos usan el encadenamiento hacia atrás para encontrar soluciones. PROLOG es un buen ejemplo de un sistema de encadenamiento hacia atrás que trata de resolver un problema dividiéndolo en subproblemas más pequeños y resolviéndolos individualmente. Por su parte, los optimistas consideran la solución de un problema como una meta que debe alcanzarse. Con el propósito de alcanzar una meta, deben ejecutarse cero o más submetas.

Un tipo de árbol o rejilla que es útil en la representación de problemas de encadenamiento hacia atrás, es el árbol Y-O. En la figura 3.9 se muestra un ejemplo sencillo de una rejilla Y-O para alcanzar la meta de obtener un título universitario. Para cumplir este objetivo, puede asistir en persona a una universidad o tomar cursos por correspondencia; con éstos el trabajo puede realizarse ya sea con tareas por correo o vía electrónica, usando una computadora casera y un módem.

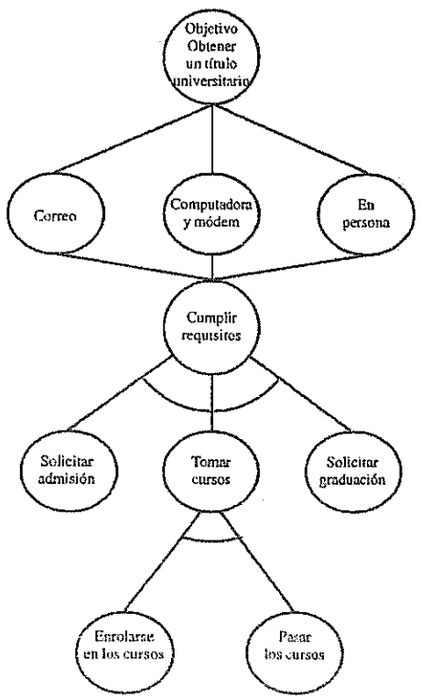


Figura 3.9 Rejilla Y-O que muestra cómo obtener un título universitario

Con el fin de satisfacer los requisitos para el título, deben completarse tres subobjetivos: 1) solicitar la admisión, 2) tomar los cursos y 3) solicitar la graduación. Observe que bajo el círculo de satisfacer los requisitos, hay un arco que une los bordes para unir estos tres subobjetivos. El arco indica que satisfacer los requisitos es un nodo Y que sólo puede satisfacerse si se satisfacen sus tres subobjetivos. Los objetivos sin el arco, como computadora y módem, correo, y en persona, son nodos O en los que la satisfacción de cualquiera de estos subobjetivos satisface el objetivo padre de obtener un título universitario.

Esta gráfica es una rejilla porque el subobjetivo de satisfacer los requisitos tiene tres nodos padre: 1) correo, 2) computadora y módem y 3) en persona. Observe que sería posible dibujar esta gráfica como un árbol con la simple duplicación del subobjetivo de satisfacer requisitos y de su subárbol de objetivos para los objetivos correo, computadora y módem, y en persona. Sin embargo, como el subobjetivo Satisfacer requisitos es el mismo para cada padre, no hay una ventaja real y sólo se usa más papel para dibujar el árbol.

Como otro ejemplo sencillo, en la figura 3.10 se muestra un árbol Y-O para el problema de llegar al trabajo por diferentes medios. Para hacerlo más completo, podría convertirse en una rejilla; por ejemplo, podría añadirse un borde desde el nodo manejar a la estación del tren hasta el nodo automóvil y otro desde caminar a la estación del tren hasta el nodo caminar. En la figura 3.11 se muestra una rejilla exclusiva del tipo Y-O.

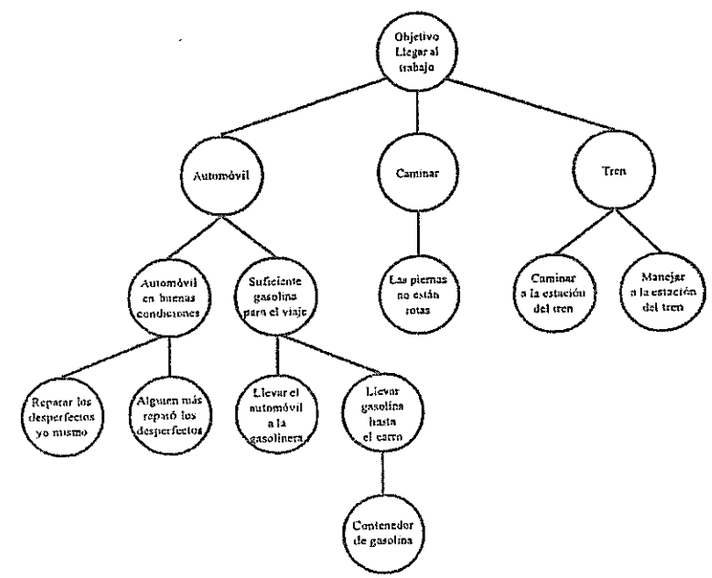


Figura 3.10 Árbol Y-O simple que muestra métodos para llegar al trabajo.

Otra forma de describir las soluciones a un problema es la rejilla Y-O-NO, que usa símbolos de compuerta lógica en lugar de la notación de los árboles Y-O. En la figura 3.12 se muestran los símbolos de entradas lógicas para Y, O y NO. Estas compuertas implantan las tablas de verdad para Y, O y NO que se analizaron en el capítulo 2. En la figura 3.13 se muestra la figura 3.9 implantada con entradas Y y O.

Los árboles Y-O y los árboles de decisión tienen las mismas ventajas y desventajas básicas. La ventaja principal de las rejillas Y-O-NO es su posible implantación en hardware para altas velocidades de procesamiento, ya que pueden diseñarse a la medida para su fabricación como circuitos integrados. En la práctica, un tipo de entrada lógica como la NO-Y se usa por

razones de economía en la fabricación más que para separar compuertas Y, O y NO. A partir de la lógica, puede probarse que es posible implantar cualquier función lógica mediante una compuerta NO-Y. Un circuito integrado con un sólo tipo de dispositivo es más barato de fabricar que uno con varios tipos de compuerta lógica.

Un chip que utiliza el encadenamiento hacia delante puede calcular rápidamente la respuesta como una función de sus entradas porque el procesamiento se desarrolla en paralelo. Chips como éstos pueden utilizarse para la supervisión de los datos en tiempo real de sensores y genera una respuesta apropiada, dependiendo de las entradas. La principal desventaja es que, como otras estructuras de decisión, un chip diseñado para lógica no puede manejar situaciones para las que no fue diseñado. Sin embargo, un ANS implantado en un chip puede manejar entradas inesperadas.

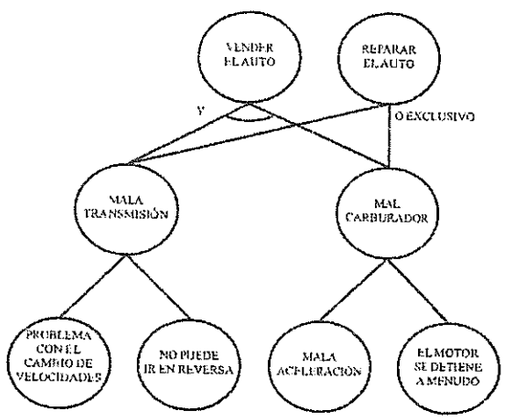


Figura 3.11 Rejilla Y-O para decidir si vender o reparar un automóvil

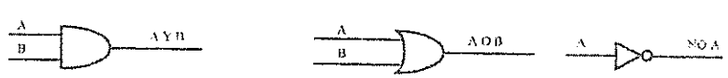


Figura 3.12 Símbolos de compuerta lógica Y, O y NO

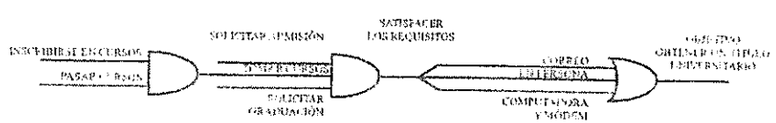


Figura 3.13 Representación de la compuerta lógica Y-O para la figura 3.9

3.5 LÓGICA DEDUCTIVA Y SILOGISMOS

En el capítulo 2 analizamos la representación del conocimiento por medio de la lógica. Ahora se verá cómo hacer inferencias para generar nuevos conocimientos o información. En el resto de este capítulo se explican diferentes métodos de inferencia. La figura 3.14 muestra un panorama general de los métodos de inferencia, que a continuación se resumen:

- *Deducción.* Razonamiento lógico en el que las conclusiones deben desprenderse de sus premisas.
- *Inducción.* Inferencia pasando del caso específico al general.
- *Intuición.* Teoría no comprobada. La respuesta simplemente aparece, tal vez por reconocimiento inconsciente de un patrón subyacente. Los sistemas expertos no realizan todavía este tipo de inferencia. Los ANS permanecerían como una promesa para este tipo de inferencia porque pueden extrapolar su enseñanza en lugar de simplemente proporcionar una respuesta condicionada o una interpolación. Es decir, una red neuronal siempre ofrecerá su mejor suposición como respuesta.
- *Heurística.* Reglas empíricas basadas en la experiencia.
- *Ensayo y error.* A menudo se utiliza con la planeación por razones de eficiencia.
- *Abducción.* Razonamiento hacia atrás desde una conclusión verdadera hacia las premisas que habrían causado la conclusión.
- *Predeterminación.* En la ausencia de conocimiento específico, adopta conocimiento general o común como opción predeterminada.
- *Autoepistemología.* Conocimiento de sí mismo.
- *No monótono.* El conocimiento previo podría ser incorrecto cuando se obtiene nueva evidencia.
- *Analogía.* Inferencia de una conclusión con base en las similitudes con otra situación.

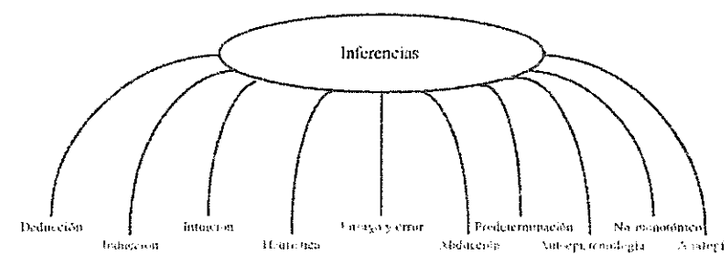


Figura 3.14 Tipos de inferencia

Aunque no se muestra explícitamente en la figura 3.14, el **conocimiento por sentido común** puede ser una combinación de cualquiera de estos tipos. El razonamiento del sentido común es el que las personas utilizan en situaciones ordinarias y es muy difícil que las computadoras lo dominen. La aplicación de la lógica de confusión para razonamientos de este tipo se analiza en el capítulo 5.

Uno de los métodos más comunes para formular inferencias es la **lógica deductiva**, que se ha utilizado desde tiempos remotos para determinar la validez de un **argumento**. Un argu-

mento lógico es un grupo de afirmaciones en el que la última de ellas está justificada con base en las anteriores de la **cadena de razonamiento.** Un tipo de argumento lógico es el silogismo que abordamos brevemente en el capítulo 2. Como ejemplo de un silogismo:

```
Premisa:     Cualquiera que pueda programar es inteligente
Premisa:     John puede programar
Conclusión:  Por lo tanto, John es inteligente
```

En un argumento, las premisas se utilizan como evidencia para respaldar las conclusiones. También se les llama **antecedentes**, y la conclusión es la **consecuencia.** La característica esencial de la lógica deductiva es que la conclusión verdadera *debe* obtenerse de premisas verdaderas. Se traza una línea para separar las premisas de la conclusión, como se muestra en el ejemplo precedente, de modo que no es necesario rotular explícitamente las premisas y la conclusión.

El argumento podría haberse escrito de forma más breve como:

```
Cualquiera que pueda programar es inteligente
John puede programar
∴ John es inteligente
```

donde los tres puntos, ∴, significan "por tanto".

Ahora veamos más de cerca a la lógica silogística. La principal ventaja del estudio de silogismos es que se trata de una simple y bien entendida rama de la lógica que puede probarse por completo. Además, a menudo los silogismos son útiles porque se pueden expresar a partir de reglas SI...ENTONCES. Por ejemplo, el silogismo anterior se puede reordenar como:

```
SI        cualquiera que puede programar es inteligente y
          John puede programar
ENTONCES John es inteligente
```

En general, un silogismo es cualquier argumento deductivo válido que cuenta con dos premisas y una conclusión. El silogismo clásico es un tipo especial llamado **silogismo categórico;** las premisas y conclusiones se definen como afirmaciones categóricas de las siguientes cuatro formas, como se muestra en la tabla 3.2.

Forma	Esquema	Significado
A	Toda S es P	afirmativo universal
E	No S es P	negativo universal
I	Alguna S es P	afirmativo particular
O	Alguna S no es P	negativo particular

Tabla 3.2 **Afirmaciones categóricas**

Observe que, en la lógica, el término *esquema* especifica la forma lógica de la afirmación. Esto ilustra también otro uso de la palabra, que es diferente de su utilización en AI, analizado en el capítulo 2. En la lógica, la palabra esquema se usa para mostrar la forma esencial de un argumento. También podría especificar la forma lógica de un silogismo entero como en:


```
Toda M es P
Toda S es M
∴ Toda S es P
```

Al sujeto de la conclusión, S, se le llama ***término menor***, mientras que al predicado de la conclusión, P, se le llama ***término mayor***. A la premisa que tiene el término mayor se le llama ***premisa mayor*** y a la que tiene el término menor se le llama ***premisa menor***. Por ejemplo

```
Premisa mayor: Toda M es P
Premisa menor: Toda S es M
Conclusión:    Toda S es P
```

es un silogismo que se dice que se encuentra en **forma normal** con sus premisas mayor y menor identificadas. El **sujeto** es el objeto que se describe y el **predicado** describe algunas propiedades del sujeto. Por ejemplo, en la afirmación

```
Todas las microcomputadoras son computadoras
```

el sujeto es "microcomputadoras" y el predicado es "computadoras". En la afirmación:

```
Todas las microcomputadoras con 8 megabytes
son computadoras con gran memoria
```

el sujeto es "microcomputadoras con 8 megabytes" y el predicado es "computadoras con gran memoria".

Las formas de las afirmaciones categóricas se han identificado desde la antigüedad con las letras A, E, I y O. La A y la I indican afirmación y vienen de las dos primeras vocales de la palabra latina "affirmo"; mientras que E y O vienen de "nego". Se dice que las formas A e I son **afirmativas en calidad** al afirmar que los sujetos se incluyen en la clase del predicado. La E y O son **negativas en calidad** porque excluyen los sujetos de la clase del predicado.

A los verbos *ser o estar* se les llama ***cópula***, que significa "conectar" en latín. La cópula conecta las dos partes de la afirmación. En el silogismo categórico normal, es la forma del tiempo presente del verbo *ser o estar.* Así que otra versión es:

```
Todas las S son P
```

Al tercer término del silogismo, M, se le llama **término medio** y es común para ambas premisas. Éste es esencial porque un silogismo se define de modo tal que la conclusión no puede deducirse únicamente de una u otra de las premisas. De modo que el argumento:

```
Toda A es B
Toda B es C
∴ Toda A es B
```

no es un silogismo válido porque se desprende únicamente de la primera premisa.

La **cantidad** o **cuantificador** describe la porción de la clase incluida. Los cuantificadores *Todo* y *No* son **universales** porque se refieren a clases enteras; al cuantificador *Algunos* se le llama **particular** porque sólo alude a una parte de la clase.

El **modo** de un silogismo está definido por las tres letras que dan forma a la premisa mayor, la premisa menor y la conclusión, respectivamente. Por ejemplo, el silogismo

```
Toda M es P
Toda S es M
∴ Toda S es P
```

es un modo AAA.

Hay cuatro patrones posibles para ordenar los términos S, P y M, como se muestra en la tabla 3.3. A cada patrón se le llama **figura**, y el número de la figura especifica su tipo

	Figura 1	Figura 2	Figura 3	Figura 4
Premisa mayor	M P	P M	M P	P M
Premisa menor	S M	S M	M S	M S

Tabla 3.3 Patrones de afirmaciones categóricas

De modo que el ejemplo anterior se describe totalmente como un tipo de silogismo AAA-1. El solo hecho de que un argumento tenga forma silogística no significa que sea válido. Considerando la forma del silogismo AEE-1:

```
Toda M es P
No S es M
∴ No S es P
```

no es un silogismo válido, como puede verse en el ejemplo:

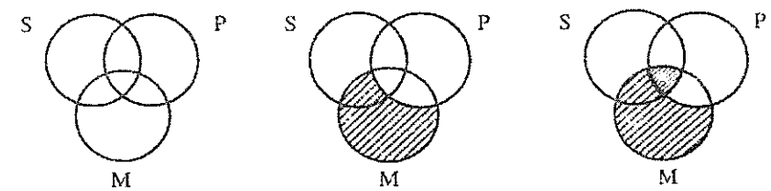
```
Todas las microcomputadoras son computadoras
Ningún mainframe es una microcomputadora
∴ Ningún mainframe es una computadora
```

En lugar de inventar ejemplos para probar la validez de argumentos silogísticos, puede utilizarse un **procedimiento de decisión**. Se trata de un método para probar la validez y es un método mecánico o algoritmo un poco general, por medio del cual puede automatizarse el proceso de determinación de la validez. Aunque hay procedimientos de decisión para la lógica silogística y la lógica de proposición, Church mostró en 1936 que no hay ninguno para la lógica de predicado. En cambio, las personas deben aplicar la creatividad para generar pruebas.

El procedimiento de decisión para las proposiciones consiste simplemente en construir una tabla de verdad y revisarla en busca de tautologías. El procedimiento de decisión para los silogismos puede darse utilizando diagramas de Venn con tres círculos superpuestos que representen S, P y M, como se muestra en la figura 3.15(a). Para la forma del silogismo AEE-1:

```
Toda M es P
Ninguna S es M
∴ Ninguna S es P
```


la premisa mayor se ilustra en la figura 3.15(b). La sección de M sombreada con líneas indica que no hay elementos en esa porción. En (c) se incluye la premisa menor, trazando líneas en su porción sin elementos. En (c) puede verse que la conclusión de AEE-1 es falsa porque hay algunas S en P.

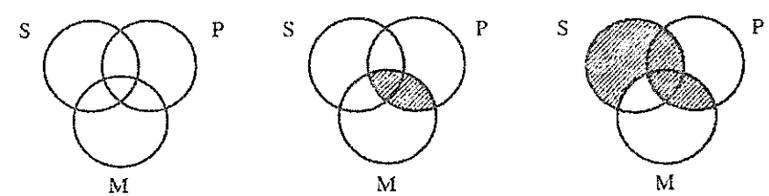


(a) Diagrama de Venn (b) Después de la premisa mayor (c) Después de la premisa menor

Figura 3.15 Procedimiento de decisión para el silogismo AEE-1

En otro ejemplo, el siguiente EAE-1 es válido, como puede observarse en la figura 3.16(c):

```
Ninguna M es P
Toda S es M
∴ Ninguna S es P
```



(a) Diagrama de Venn (b) Después del término mayor (c) Después del término menor

Figura 3.16 Procedimiento de decisión para el silogismo EAE-1

Los diagramas de Venn que incluyen "algunos" cuantificadores son un poco más difíciles de trazar. Las reglas generales para dibujar silogismos categóricos bajo la concepción de Boole de que tal vez no haya miembros en las afirmaciones A y E son:

1. Si una clase está vacía, se sombrea.
2. Las afirmaciones universales, A y E, siempre se dibujan antes que las particulares.
3. Si una clase tiene por lo menos un miembro, márquela con un asterisco.
4. Si una afirmación no especifica en cuál de las dos clases adyacentes existe un objeto, coloque un asterisco en la línea entre las clases.
5. Si un área ha sido sombreada, no puede introducirse un asterisco en ella.

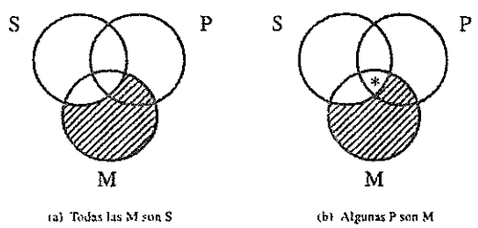
Por ejemplo:

```
Algunas computadoras son laptops
Todas las laptops son portátiles
∴ Algunas portatiles son computadoras
```

que se puede poner en un tipo IAI-4 como

```
Algunas P son M
Todas las M son S
∴ Algunas S son P
```

Siguiendo las reglas 2 y 1 de los diagramas de Venn, iniciamos con la afirmación universal para la premisa menor y la sombreamos, como lo muestra la figura 3.17(a). A continuación, se aplica la regla 3 a la premisa mayor en particular y se dibuja un *, como se muestra en la figura 3.17(b). Como en el diagrama se muestra la conclusión "Algunas portátiles son computadoras", se deriva que el argumento IAI-4 es un silogismo válido.



(a) Todas las M son S (b) Algunas P son M

Figura 3.17 Silogismo del tipo IAI-4

3.6 REGLAS DE INFERENCIA

Aunque los diagramas de Venn son un procedimiento de decisión para silogismos, son inconvenientes para argumentos más complejos, porque se vuelven más difíciles de leer. Sin embargo, hay un problema más importante con los silogismos, debido a que sólo atienden una pequeña parte de las afirmaciones lógicas posibles. En particular, los silogismos categóricos sólo atienden afirmaciones categóricas de la forma A, E, I y O.

La lógica proposicional ofrece otros mecanismos para describir argumentos; en efecto, a menudo la utilizamos sin darnos cuenta. Por ejemplo, considerando el siguiente argumento de proposición:

```
Si hay energía eléctrica, la computadora trabajará
Hay energía eléctrica
∴ La computadora trabajará
```

Este argumento puede expresarse de manera formal utilizando letras para representar las proposiciones, como sigue:


```
A = Hay energía eléctrica
B = La computadora trabajará
```

de modo que el argumento puede escribirse como

```
A → B
A
∴ B
```

Argumentos como éste se dan a menudo; un esquema general para la representación de argumentos de este tipo es:

```
p → q
p
∴ q
```

donde p y q son variables lógicas que pueden representar cualquier afirmación. El uso de variables lógicas en la lógica proposicional permite afirmaciones más complejas que las de las cuatro formas silogísticas A, E, I y O. Al esquema de inferencia de esta forma proposicional se le llama con diversos nombres: **razonamiento directo, *modus ponens*, ley de desprendimiento y suposición del antecedente.**

Observe que este ejemplo también puede expresarse en la forma silogística:

```
Todas las computadoras con energía eléctrica trabajarán
Esta computadora tiene energía eléctrica
Esta computadora trabajará
```

que demuestra que el *modus ponens* es realmente un caso especial de lógica silogística. El *modus ponens* es importante porque forma la base de los sistemas expertos basados en reglas. La proposición compuesta p → q corresponde a la regla y la p corresponde al patrón que debe marcar el antecedente de la regla que habrá de satisfacerse. Sin embargo, como se analiza en el capítulo 2, el condicional p → q no es equivalente exacto de una regla, porque el condicional es una definición lógica definida por una tabla de verdad y hay muchas definiciones posibles para él.

En este texto se utilizará la convención de la teoría lógica de utilizar mayúsculas como A, B, C ... para representar proposiciones constantes como "Hay energía eléctrica". Las minúsculas como p, q, r ..., representarán variables lógicas que pueden corresponder a diferentes proposiciones constantes. Observe que esta convención es opuesta a la de PROLOG, que utiliza mayúsculas para las variables.

Este esquema de *modus ponens* pudo haberse escrito con variables lógicas denominadas de otra manera como:

```
r → s
r
∴ s
```

y el esquema aún significaría lo mismo.

Otra notación para el esquema es:

```
r, r → s; ∴ q
```

donde la coma se utiliza para separar una premisa de otra y el punto y coma indica el final de las premisas. Aunque hasta ahora sólo hemos visto argumentos con dos premisas, una forma más general de un argumento es:

$$P_1,\ P_2,\ \ldots\ P_N;\ \therefore\ C$$

donde las mayúsculas P_i representan premisas como r, $r \rightarrow s$, y C es la conclusión. Observe cómo se parece a la instrucción de cumplimiento de objetivos de PROLOG, analizado en el capítulo 2.

$$p\ :-\ p_1,\ p_2,\ \ldots\ p_N.$$

El objetivo, p, se satisface si se satisfacen todos los subobjetivos $p_1, p_2, \ldots p_N$. Un argumento análogo para reglas de producción puede escribirse en la forma general:

$$C_1 \wedge C_2 \wedge \ldots\ C_N \rightarrow A$$

lo que significa que si se satisface cada condición, C_i, de una regla, entonces se realiza la acción, A, de la regla. Como se analizó previamente, una afirmación lógica de la forma anterior no es estrictamente equivalente a una regla, porque la definición lógica del condicional no es la misma que una regla de producción. Sin embargo, esta forma lógica es una ayuda intuitiva útil para razonar acerca de las reglas.

La notación de los operadores lógicos Y y O toma diferentes formas en PROLOG, en comparación con los usuales $\wedge$ y $\vee$. La coma entre subobjetivos en PROLOG significa una conjunción, $\wedge$, y la disyunción, $\vee$, se indica con un punto y coma. Por ejemplo:

$$p\ :-\ p_1;\ p_2.$$

significa que p se satisface si p_1 o p_2 se satisfacen. Las conjunciones y las disyunciones pueden combinarse. Por ejemplo:

$$p\ :-\ p_1,\ p_2;\ p_3,\ p_4.$$

es lo mismo que las dos afirmaciones de PROLOG:

$$p\ :-\ p_1,\ p_2.$$
$$p\ :-\ p_3,\ p_4.$$

En general, si las premisas y la conclusión son esquemas, el argumento:

$$P_1,\ P_2,\ \ldots\ P_N;\ \therefore\ C$$

es un argumento deductivo formalmente válido si y sólo si:

$$P_1 \wedge P_2 \wedge \ldots\ P_N \rightarrow C$$

es una tautología. Como ejemplo:

$$(p \wedge q) \rightarrow p$$

es una tautología porque es verdadero para cualquier valor, V o F, de p y q. Esto se puede verificar construyendo la tabla de verdad.

El argumento de *modus ponens:*

$$\begin{array}{l} p \rightarrow q \\ \underline{p\qquad} \\ \therefore\ q \end{array}$$

es válido porque puede expresarse como tautología.

$$(p \rightarrow q) \wedge p \rightarrow q$$

Observe que se está suponiendo que la flecha tiene precedencia inferior que la conjunción y la disyunción. Esto ahorra la escritura de paréntesis adicionales como:

$$((p \rightarrow q) \wedge p) \rightarrow q$$

En la tabla 3.4 se muestra la tabla de verdad para el *modus ponens*. Es una tautología porque todos los valores del argumento, mostrados en la columna de la derecha, son verdaderos sin importar los valores de sus premisas. Observe que en la tercera, cuarta y quinta columnas, los valores de verdad se escriben bajo ciertos operadores, como $\rightarrow$ y $\wedge$. Se les llama **conectores principales** porque conectan las dos partes principales de una proposición compuesta.

p	q	$p \rightarrow q$	$(p \rightarrow q) \wedge p$	$(p \rightarrow q) \wedge p \rightarrow q$
V	V	V	V	V
V	F	F	F	V
F	V	V	F	V
F	F	V	F	V

Tabla 3.4 Tabla de verdad para *modus ponens*

Aunque este método funciona para determinar argumentos válidos, requiere que se verifique cada fila de la tabla de verdad. El número de filas es 2^N, donde N es el número de premisas, de modo que las filas se incrementarán rápidamente. Por ejemplo, para cinco premisas se requerirá verificar 32 filas, mientras que para 10 premisas se verificarían 1024 filas. Un método más corto para determinar un argumento válido, es considerar únicamente aquellas filas de la tabla de verdad en las que todas las premisas son verdaderas; la definición equivalente de un argumento válido establece que es válido si y sólo si la conclusión es verdadera para cada una de estas filas. Es decir, la conclusión está tautológicamente implícita en las premisas. Para el *modus ponens*, la premisa $p \rightarrow q$ y la premisa p sólo son verdaderas en la primera fila, y también en la conclusión, por tanto, el *modus ponens* es un argumento válido. Si hubiera alguna otra fila en la que todas las premisas fueran verdaderas y la conclusión falsa, entonces el argumento no sería válido.

En la tabla 3.5 se muestra la manera más corta de expresar la tabla de verdad para el *modus ponens*, donde todas las filas se muestran explícitamente. En la práctica, sólo necesitan considerarse aquellas filas que tienen premisas verdaderas, como la primera.

La tabla de verdad para el *modus ponens* muestra que es válida porque la primera fila tiene premisas verdaderas y una conclusión verdadera, y no hay otras filas que tengan premisas verdaderas y una conclusión falsa.

		Premisas		Conclusión
p	q	p → q	p	q
V	V	V	V	V
V	F	F	V	F
F	V	V	F	V
F	F	V	F	F

Tabla 3.5 Forma corta de una tabla de verdad para el *modus ponens*.

Los argumentos pueden ser engañosos. Para demostrar esto, consideremos primero el siguiente ejemplo válido de *modus ponens*:

```
Si no hay errores, entonces el programa se cumple
No hay errores
∴ El programa se cumple
```

Comparando esto con el siguiente argumento que, de alguna manera, semeja al del *modus ponens*

```
Si no hay errores, entonces el programa se cumple
El programa se cumple
∴ No hay errores
```

¿Se trata de un argumento válido? El esquema para argumentos de este tipo es:

```
p → q
q
∴ p
```

y su tabla de verdad en forma corta se muestra en la tabla 3.6.

		Premisas		Conclusión
p	*q*	*p* → *q*	*q*	*p*
V	V	V	V	V
V	F	F	F	V
F	V	V	V	F
F	F	V	F	F

Tabla 3.6 Forma corta de la tabla de verdad para p → q, q; ∴p

Observe que este argumento no es válido. Aunque la primera fila muestra que la conclusión es verdadera si todas las premisas lo son, la tercera fila muestra que si la premisa es verdadera, la conclusión es falsa. Por tanto, este argumento falla para el criterio si y sólo si de un argumento válido. Aunque muchos programadores desearían que argumentos como éstos fueran verdaderos, la lógica (y la experiencia) prueban que es una **falacia**, o argumento no válido. A este argumento en particular se le llama *falacia del recíproco*.

Tomando otro ejemplo, el esquema de argumento:

```
p → q
~q
∴ ~p
```


es la razón por la que la tabla 3.7 muestra que la conclusión es verdadera sólo cuando las premisas son verdaderas.

		Premisas		Conclusión
p	*q*	*p* → *q*	*q*	~*p*
V	V	V	F	F
V	F	F	V	F
F	V	V	F	V
F	F	V	V	V

Tabla 3.7 Forma corta de la tabla de verdad para p → q, ~q; ∴ ~p

A este esquema en particular se le llama de diversas maneras: ***razonamiento indirecto***, ***modus tollens*** y ***ley de la contraposición***.

Modus ponens y *modus tollens* son **reglas de inferencia**, a veces llamadas ***leyes de inferencia***. En la tabla 3.8 se muestran algunas de ellas.

La palabra en *modus* significan "modo" en latín, mientras que *ponere* significa "asertar" y *tollere* significa "negar". Los nombres reales de las reglas del *modus* y sus significados literales se muestran en la tabla 3.9. *Modus ponens* y *modus tollens* son abreviaturas de los dos primeros tipos (Stebbing 50). Los números de la regla de inferencia corresponden a los de la tabla 3.8.

Las reglas de inferencia pueden aplicarse a argumentos con más de dos premisas. Por ejemplo, considerando el siguiente argumento:

```
Los precios bajos suben sólo si el yen sube
El yen sube sólo si el dólar baja y
   si el dólar baja entonces el yen sube.
Como los precios bajos han subido,
   el dólar debe haber bajado.
```

Definamos las proposiciones como sigue:

```
B = precios bajos suben
Y = yen sube
D = dólar baja
```

Si se recuerda de la sección 2.12 que uno de los significados del condicional es "p, sólo si q". Una proposición como "El yen sube sólo si el dólar baja" tiene este significado y se representa como B → Y. Entonces todo el argumento tiene la siguiente forma.

```
B → Y
(Y → D) ∧ (D → Y)
B
∴ D
```

La segunda premisa tiene una forma interesante, que puede reducirse aún más si se utiliza una variante del condicional. El condicional p → q tiene diversas variantes, que son la **recíproca**, la **inversa** y la **contrapositiva**. Se presentan en la lista con el condicional, para completar este aspecto, en la tabla 3.10.

Ley de inferencia	Esquema	
1. Ley del desprendimiento	$p \to q$ p $\therefore q$	
2. Ley de la contrapositiva	$p \to q$ $\therefore \sim q \to \sim p$	
3. Ley de *modus tollens*	$p \to q$ $\sim q$ $\therefore \sim p$	
4. Regla de cadena (Ley del silogismo)	$p \to q$ $q \to r$ $\therefore p \to r$	
5. Ley de la inferencia disyuntiva	$p \vee q$ $\sim p$ $\therefore q$	$p \vee q$ $\sim q$ $\therefore p$
6. Ley de la doble negación	$\sim(\sim p)$ $\therefore p$	
7. Ley de De Morgan	$\sim(p \wedge q)$ $\therefore \sim p \vee \sim q$	$\sim(p \vee q)$ $\therefore \sim p \wedge \sim q$
8. Ley de simplificación	$p \wedge q$ $\therefore p$	$\sim(p \vee q)$ $\therefore q$
9. Ley de conjunción	p q $\therefore p \wedge q$	
10. Ley de la adición disyuntiva	p $\therefore p \vee q$	
11. Ley del argumento conjuntivo	$\sim(p \wedge q)$ p $\therefore \sim q$	$\sim(p \wedge q)$ q $\therefore \sim p$

Tabla 3.8 Algunas reglas de inferencia para la lógica proposicional

Número de regla de inferencia	Nombre	Significado "modo por el que la..."
1	*modus ponendo ponens*	afirmación, afirma
3	*modus tollendo tollens*	negación, niega
5	*modus tollendo ponens*	negación, afirma
11	*modus ponendo tollens*	afirmación, niega

Tabla 3.9 Los significados de modus

Condicional	$p \to q$
recíproco	$q \to p$
inverso	$\sim p \to \sim q$
contrapositivo	$\sim q \to \sim p$

Tabla 3.10 El condicional y sus variantes

Como siempre, se supone que el operador de negación tiene una mayor prioridad que los otros operadores lógicos, de modo que no se usan paréntesis con $\sim p$ y $\sim q$.

Si el condicional $p \to q$ y su recíproco $q \to p$ son verdaderos, entonces p y q son equivalentes; es decir, $p \to q \wedge q \to p$ es equivalente al bicondicional $p \leftrightarrow q$ o a la equivalencia $p \equiv q$. En otras palabras, p y q siempre toman los mismos valores de verdad, si p es verdadero entonces q es verdadero y si p es falso entonces q es falso. El argumento se vuelve

$$\begin{array}{ll} (1) & B \to Y \\ (2) & Y \equiv D \\ (3) & B \\ \hline \therefore & D \end{array}$$

donde los números se utilizan ahora para identificar las premisas. Como Y y D son equivalentes a partir de (2), podemos sustituir D por Y en (1) para obtener:

$$(4) \quad B \to D$$

donde (4) es una inferencia hecha con base en (1) y (2). Las premisas (3) y (4) y la conclusión son:

$$\begin{array}{ll} (4) & B \to D \\ (3) & B \\ \hline \therefore & D \end{array}$$

que puede reconocerse como un esquema de *modus ponens*. Por tanto, el argumento es válido.

La sustitución de una variable que es equivalente a otra, es una regla de inferencia llamada ***regla de sustitución***. Las reglas de *modus ponens* y de sustitución son básicas en la lógica deductiva.

Una comprobación de lógica formal al enumerar las premisas, la conclusión y las inferencias, suele escribirse como se muestra a continuación:

```
1. B → Y
2. (Y → D) ∧ (D → Y)
3. B                        / ∴ D
4. Y ≡ D                    2 Equivalencia
5. C → D                    1 Sustitución
6. D                        3,5 Modus ponens
```

Las líneas 1, 2 y 3 son las premisas y la conclusión; 4, 5 y 6 son las inferencias obtenidas. La columna de la derecha presenta la regla de inferencia y los números de línea se utilizan para justificar la inferencia.

3.7 LIMITACIONES DE LA LÓGICA DE PROPOSICIÓN

Considerando nuestro familiar y clásico argumento:

```
Todos los hombres son mortales
Sócrates es un hombre
Por tanto, Sócrates es mortal
```

Sabemos que es un argumento válido porque es un silogismo válido. ¿Se puede probar su validez utilizando la lógica proposicional? Para responder a esta pregunta, primero se escribe el argumento como un esquema:

```
p = Todos los hombres son mortales
q = Sócrates es un hombre
r = Sócrates es mortal
```

de este modo, el esquema de argumento es:

```
p
q
∴ r
```

Observe que no hay conectores lógicos ni conclusiones en las premisas y, por tanto, cada premisa y cada conclusión tiene una variable lógica diferente. Además, la lógica proposicional no ha previsto cuantificadores, así, no hay manera de representar al cuantificador "todos" en la primera premisa. Por lo tanto, la única representación de este argumento en la lógica proposicional es el esquema anterior con tres variables independientes.

Para determinar si es un argumento válido, se considera la tabla de verdad con tres variables independientes para todas las combinaciones posibles de V y F que se muestran en la tabla 3.11. La segunda fila de esta tabla muestra el argumento que habrá de invalidarse porque las premisas son verdaderas pero la conclusión es falsa.

La invalidez de este argumento *no* debe interpretarse como si significara que la conclusión es incorrecta. Cualquier persona reconocería que se trata de un argumento correcto, la invalidez simplemente significa que *el argumento no puede probarse con base en la lógica proposicional.* Puede probarse que el argumento es válido si examinamos la estructura interna de las premisas; por ejemplo, tendríamos que atribuir algún significado a "todos" y reconocer "hombres" como plural de "hombre". Sin embargo, los silogismos y el cálculo proposicional no permiten examinar la estructura interna de las proposiciones. Esta limitación se supera con la lógica de predicados; y el argumento es válido bajo esta lógica. En realidad, toda la lógica silogística es un subconjunto válido de la lógica de predicados de primer orden, y con ella puede probarse su validez.

p	q	∴ r
V	V	V
V	V	F
V	F	V
V	F	F
F	V	V
F	V	F
F	F	V
F	F	F

Tabla 3.11 Tabla de verdad para el esquema p, q; ∴ r

La única forma silogística válida de la proposición es:

```
Si Sócrates es un hombre, entonces Sócrates es mortal.
Sócrates es un hombre.
Por tanto, Sócrates es mortal.
```

Digamos que:

```
p = Sócrates es un hombre
q = Sócrates es mortal
```

El argumento se convierte en:

```
p → q
p
∴ q
```

que es una forma silogística válida de *modus ponens*.

Consideremos el siguiente argumento clásico como otro ejemplo.

```
Todos los caballos son animales
Por tanto, la cabeza de un caballo es la cabeza de un
animal
```

Se sabe que este argumento es correcto, y aún así no podemos probarlo bajo una lógica de proposición; aunque puede probarse bajo la lógica de predicado (véase el problema 3.12).

3.8 LÓGICA DE PREDICADO DE PRIMER ORDEN

La lógica silogística puede describirse por completo con la lógica de predicado. En la tabla 3.12 se muestran las cuatro afirmaciones categóricas y sus representaciones en esta clase de lógica de predicado.

Tipo	Esquema	Representación de predicado
A	Todo S es P	$(\forall x)(S(x) \rightarrow P(x))$
E	Ningún S es P	$(\forall x)(S(x) \rightarrow \sim P(x))$
I	Algún S es P	$(\exists x)(S(x) \wedge P(x))$
O	Algún S no es P	$(\exists x)(S(x) \wedge \sim P(x))$

Tabla 3.12 Representación de los cuatro silogismos categóricos utilizando lógica de predicados

Además de las reglas de inferencia previamente analizadas, la lógica de predicados tiene reglas que tratan con cuantificadores.

La Regla de Asignación Universal de Casos establece, en esencia, que un individuo puede sustituirse por uno universal. Por ejemplo, si ϕ es cualquier proposición o **función de proposición**:

$$\frac{(\forall x)\ \phi(x)}{\therefore\ \phi(a)}$$

es una inferencia válida, donde a es un caso; es decir, alude a un individuo específico, mientras que x es una variable que abarca todos los individuos. Por ejemplo, esto puede utilizarse para probar que Sócrates es un ser humano:

$$\frac{(\forall x)\ H(x)}{\therefore\ H(\text{Sócrates})}$$

donde H(x) es la función proposicional que indica que x es un ser humano. Lo anterior establece que para toda x, esa x es un ser humano y por inferencia Sócrates es un ser humano.

Otros ejemplos de la Regla de Asignación Universal de Casos son:

$$\frac{(\forall x)\ A(x)}{\therefore\ A(c)}$$

$$\frac{(\forall y)\ (B(y) \vee C(b))}{\therefore\ B(a) \vee C(b)}$$

$$\frac{(\forall x)\ [A(x) \wedge (\exists x)\ (B(x) \vee C(y))]}{\therefore\ A(b) \wedge (\exists x)\ (B(x) \vee C(y))}$$

En el primer ejemplo, el caso c se sustituye con x, pero en el segundo ejemplo, nótese que el caso a se ha sustituido por y, pero no por b, porque b no está incluida en el **ámbito** del cuantificador; es decir, un cuantificador como $\forall$ x sólo se aplica a las variables x. A las variables como x y y utilizadas con cuantificadores se les llama ***delimitadas*** y a los otros se les llama ***libres***. En el tercer ejemplo, el cuantificador x sólo tiene como ámbito A(x). Es decir, $\forall$ x no se aplica al cuantificador existencial $\exists$ x y a su ámbito sobre $B(x) \vee C(y)$. La convención de cuantificadores anidados como éste, es que el ámbito termina cuando se utiliza un nuevo cuantificador, aunque utilice la misma variable. La prueba formal del silogismo:


```
Todos los hombres son mortales
Sócrates es un hombre
∴ Sócrates es mortal
```

se muestra a continuación, donde H = hombre, M = mortal y s = Sócrates:

```
1. (∀ x) (H(x) → M(x))
2. H(s)                    /∴ M(s)
3. H(s) → M(s)             1 Asignación universal de casos
4. M(s)                    2,3 Modus ponens
```

3.9 SISTEMAS LÓGICOS

Un sistema lógico es una recopilación de objetos como reglas, axiomas, instrucciones, etcétera, organizada de manera congruente. El sistema lógico tiene varios objetivos, el primero consiste en especificar las formas de los argumentos. Como los argumentos lógicos no tienen significado en un sentido semántico, una forma válida es esencial si habrá de determinarse la validez del argumento. Por tanto, una función importante de un sistema lógico es determinar las **fórmulas bien formadas (fbf)** que se utilizan en los argumentos; sólo éstas pueden utilizarse en argumentos lógicos. Por ejemplo, en la lógica silogística:

```
Todo S es P
```

podría ser una fbf, pero:

```
Todo
Todo es S P
Es S todo
```

no son fbfs. Aunque los símbolos del alfabeto carecen de significado, la secuencia de símbolos que conforman la fbf sí lo tiene.

El segundo objetivo de un sistema lógico es indicar las reglas de inferencia que son válidas. El tercer objetivo de un sistema lógico es extenderse al descubrir nuevas reglas de inferencia y, por tanto, extender el rango de argumentos que pueden probarse. Al extender el rango de argumentos, pueden probarse nuevas fbf, llamadas ***teoremas***, con un argumento lógico.

Cuando un sistema lógico está bien desarrollado, puede utilizarse para determinar la validez de los argumentos de manera análoga a los cálculos en sistemas como la aritmética, la geometría, el cálculo, la física y la ingeniería. Se han desarrollado sistemas lógicos como el Cálculo de frases o proposiciones, el Cálculo de afirmaciones, etc. Cada sistema depende de definiciones formales de sus **axiomas** o **postulados**, que son sus definiciones fundamentales. A partir de estos axiomas, la gente (y a veces los programas de computadora como el AM), tratan de determinar lo que puede probarse. Cualquier persona que haya estudiado geometría euclidiana en la preparatoria está familiarizado con los axiomas y la derivación de teoremas geométricos. Al igual que los teoremas geométricos pueden derivarse de los axiomas geométricos, los teoremas lógicos pueden derivarse de axiomas lógicos.

Un axioma es sólo un hecho o **aseveración** (o aserto) que no puede probarse desde el interior del sistema. A veces aceptamos ciertos axiomas porque tienen "sentido" al apelar al sentido común o la observación. Otros axiomas, como "las líneas paralelas se unen en el infinito", no tienen un sentido intuitivo porque parecen contradecir el axioma de Euclides de que las líneas

paralelas nunca se unen. Sin embargo, este axioma acerca de las líneas paralelas que se unen en el infinito es tan razonable desde un punto de vista puramente lógico como el de Euclides y es la base de un tipo de geometría no euclidiana.

Un sistema formal requiere lo siguiente:

1. Un alfabeto de símbolos.
2. Un conjunto de cadenas finitas de estos símbolos, las fbf.
3. Axiomas, las definiciones del sistema.
4. Reglas de inferencia, que permiten una fbf, A, que habrá de deducirse como la conclusión de un conjunto finito, G, de otras fbf donde $G = \{A_1, A_2 \ldots A_n\}$. Estas fbf deben ser axiomas u otros teoremas del sistema lógico. Por ejemplo, un sistema de lógica proposicional puede definirse utilizando sólo *modus ponens* para derivar nuevos teoremas.

Si el argumento:

$$A_1,\ A_2,\ \ldots\ A_N;\ \therefore\ A$$

es válido, entonces se dice que A es un **teorema** del sistema de lógica formal y se escribe con el símbolo $\vdash$. Por ejemplo, $\Gamma \vdash A$ significa que A es un teorema del conjunto de fbf, Γ. Un esquema más explícito de una prueba de que A es un teorema se muestra a continuación:

$$A_1,\ A_2,\ \ldots\ A_N \vdash A$$

El símbolo $\vdash$, que indica que el siguiente fbf es un teorema, no es un símbolo del sistema. En cambio, $\vdash$ es un **metasímbolo**, porque se utiliza para describir al propio sistema. Una analogía es un lenguaje de computadora como Pascal, pues aunque pueden especificarse programas utilizando la sintaxis de Pascal, no hay sintaxis en Pascal para indicar un programa válido.

Una regla de inferencia en un sistema formal especifica la manera exacta en que pueden obtenerse nuevas asertos, los teoremas, a partir de axiomas y teoremas previamente derivados. Un ejemplo de un teorema es nuestro silogismo sobre Sócrates, escrito en forma de lógica de predicados:

$$(\forall\ x)\ (H(x) \rightarrow M(x)),\ H(s) \vdash M(s)$$

donde H es la función de predicado para hombre y M es la función de predicado para mortal. Como M(s) puede probarse a partir de sus axiomas de la izquierda, es un teorema de estos axiomas. Sin embargo, nótese que M(Zeus) no sería un teorema porque Zeus, el dios griego, no es un hombre, y no hay una manera alterna de demostrar M(Zeus).

Si un teorema es una tautología, se deduce que Γ es un conjunto vacío porque la fbf siempre es verdadera y, por tanto, no depende de ningún otro axioma o teorema. Un teorema que es una tautología se escribe con el símbolo $\models$, como en $\models A$. Por ejemplo, si $A ip \vee \sim p$, entonces $\models p \vee \sim p$ establece que $p \vee \sim p$ es un teorema, que es una tautología. Observe que, independientemente de cualquier valor asignado a p, V o F, el teorema $p \vee \sim p$ siempre es verdadero. Una asignación de valores verdaderos es una **interpretación** de una fbf. Un **modelo** es una interpretación en la cual la fbf es verdadera. Por ejemplo, un modelo de $p \rightarrow q$ es $p = V$ y $q = V$. A una fbf se le llama ***consistente*** si hay una interpretación que la hace verdadera, e ***inconsistente*** si la fbf es falsa en *todas* las interpretaciones. Un ejemplo de una fbf inconsistente es $p \wedge \sim p$.

Una fbf es **válida** si es verdadera en todas las interpretaciones; de otra manera es **inválida**. Por ejemplo, la fbf $p \vee \sim p$ es válida, mientras que $p \rightarrow q$ es una fbf inválida porque no es verdadera para $p = V$ y $q = F$. Una fbf está **probada** si puede demostrarse que es válida. Todas las fbf de proposiciones pueden comprobarse con el método de la tabla de verdad, porque sólo hay un número finito de interpretaciones para fbf y, así, se puede determinar el cálculo proposicional **decidible**. Sin embargo, el cálculo de predicados no es decidible porque no hay un método general de comprobación como las tablas de verdad para todas las fbf del cálculo de afirmaciones.

Un ejemplo de una fbf de cálculo de predicados válida que puede probarse es la de cualquier predicado B;

$$(\exists\ x)\ B(x) \rightarrow \sim\ [(\forall\ x)\ \sim B(x)]$$

que muestra cómo el cuantificador existencial puede reemplazarse con el cuantificador universal. Por tanto, esta fbf de cálculo de predicados es un teorema.

Hay una gran diferencia entre una expresión como $\vdash A$ y $\models B$. La A es un teorema y puede probarse a partir de los axiomas, mediante las reglas de inferencia. La B es una fbf y tal vez no haya prueba conocida para demostrar su derivación. Mientras que la lógica proposicional es decidible, la lógica de predicado no lo es. Es decir, no hay un procedimiento ni algoritmo mecánico para comprobar un teorema de lógica proposicional con un número finito de pasos. En realidad, existe una prueba teórica de que no hay un procedimiento de decisión para la lógica de predicados. Sin embargo, no hay procedimientos de decisión para sus subconjuntos como los silogismos y la lógica de proposición. Debido a esto, a veces se alude a la lógica de predicado como **semidecidible**.

Un ejemplo muy simple de un sistema formal completo, define lo siguiente:

Alfabeto: el símbolo "1"
Axioma: La cadena "1" (que resulta igual que el símbolo 1)
Regla de inferencia: si cualquier cadena $ es un teorema, entonces la cadena $11 lo es. Esta regla puede escribirse como una regla de producción de Post:

$\$ \rightarrow \$11.$
Si $ = 1 entonces la regla da $11 = 111
Si $ = 111 entonces la regla da $11 = 11111 y en general
1, 111, 11111, 1111111,...

Estas cadenas son los teoremas de este sistema formal (Minsky 67).

Aunque algo como 11111 no se parece a los teoremas que acostumbramos ver, son teoremas lógicos perfectamente válidos. Estos teoremas particulares también tienen un significado semántico, porque son los números nones expresados en un **sistema de números unitario** del símbolo 1. Al igual que el sistema binario sólo tiene los símbolos del alfabeto 0 y 1, el sistema de números unitario sólo tiene el símbolo 1. Los números de los sistemas unitario y decimal se expresan como sigue:

Unitario	Decimal
1	1
11	2
111	3
1111	4
11111	5

y así sucesivamente.

Observemos que debido a la regla de inferencia y al axioma, las cadenas 11, 1111 y demás, no pueden expresarse en nuestro sistema formal. Es decir, 11 y 1111 son cadenas de nuestro alfabeto formal, pero no son teoremas ni fbf porque no pueden probarse utilizando solamente la regla de inferencia y el axioma. El sistema formal sólo permite la derivación de los números nones, no la de los pares. Debe agregarse el axioma "11" para poder derivar los números pares.

Otra propiedad de un sistema formal es su **capacidad de estar completo**. Un conjunto de axiomas está **completo** si cada fbf puede probarse o **refutarse**, este término significa probar que alguna aseveración es falsa. En un sistema completo, cada fbf lógicamente válida es un teorema. Sin embargo, como no es posible determinar la lógica de predicados, obtener una prueba depende de nuestra suerte e inteligencia. Por supuesto, otra posibilidad es escribir un programa de computadora que tratará de derivar pruebas y dejar que actúe.

Una propiedad aún más deseable de un sistema lógico es que sea **sólido**. Por sistema sólido se entiende que cada teorema es una fbf lógicamente válida. En otras palabras, un sistema sólido no permitirá que se infiera una conclusión que no sea una consecuencia lógica de sus premisas. No se inferirán argumentos inválidos como válidos.

Hay diferentes **órdenes** de lógica. Un lenguaje de **primer orden** se define para que los cuantificadores operen sobre objetos que son variables como $\forall$ x. Un lenguaje de **segundo orden** tendría características adicionales como dos tipos de variables y cuantificadores. Además de variables y cuantificadores de orden, la lógica de segundo orden puede tener cuantificadores que abarcan símbolos de función y de predicado. Un ejemplo de lógica de segundo orden es el **axioma de igualdad**, que establece que dos objetos son iguales si todos sus predicados son iguales. Si P es cualquier predicado de un argumento, entonces:

$$x = y \equiv (\forall P)\ [P(x) \leftrightarrow P(y)]$$

es una afirmación del axioma de igualdad que usa un cuantificador de segundo orden, $\forall$ " P, que abarca todos los predicados.

3.10 RESOLUCIÓN

La muy eficaz regla de **resolución** de inferencia, introducida por Robinson en 1965, con frecuencia se implanta en programas de AI para prueba de teoremas (Robinson 65). En realidad, la resolución es la principal regla de inferencia de PROLOG. En lugar de muchas reglas de inferencia de aplicación limitada, como *modus ponens*, *modus tollens*, combinación, encadenamiento, etcétera, PROLOG utiliza una regla de inferencia general de resolución. Esta aplicación de la resolución hace que los probadores automáticos de teoremas como PROLOG resulten herramientas prácticas para resolver problemas, por que en lugar de tener que probar diferentes reglas de inferencia esperando que una tenga éxito, puede aplicarse la regla de resolución simple, método que puede reducir mucho el espacio de búsqueda.

Como una manera de introducir la resolución, consideremos primero el silogismo sobre Sócrates expresado en PROLOG como se muestra a continuación, donde los comentarios se muestran con un signo de porcentaje:

```
mortal(X) :- hombre(X).     % Todos lo hombres son mortales
hombre(sócrates).           % Sócrates es un hombre
:- mortal(sócrates).        % Consulta: ¿Sócrates es mortal?
sí                          % PROLOG responde sí
```


PROLOG utiliza una notación **libre de cuantificador**. Obsérvese que el cuantificador universal, $\forall$, está implícito en la afirmación de que todos los hombres son mortales.

PROLOG se basa en lógica de predicados de primer orden. Sin embargo, también tiene varias extensiones para facilitar las aplicaciones de programación. Estas funciones de programación especiales violan la lógica de predicados pura y se les denomina ***funciones extralógicas***: entrada/salida, corte (que altera el espacio de búsqueda) y asegurar/retractar (para modificar valores de verdad sin justificación lógica).

Antes de que pueda aplicarse la resolución, la fbf debe estar en una forma **normal** o estándar. Los tres principales tipos de formas normales son la **forma normal conjuntiva**, la forma de cláusula y su subconjunto cláusula de Horn. La idea básica de la forma normal es expresar la fbf de una forma estándar que sólo utilice $\wedge$, $\vee$ y tal vez $\sim$. El método de resolución se aplica entonces a las fbf de forma normal en la que todos los conectores y cuantificadores se han eliminado. Esta conversión es necesaria porque la resolución es una operación sobre pares de disyunciones, que produce nuevas disyunciones, lo que simplifica la fbf.

A continuación se ilustra una fbf de forma normal conjuntiva, que se define como la conjunción de disyunciones, que son **literales**.

$$(P_1 \vee P_2 \vee \ldots) \wedge (Q_1 \vee Q_2 \vee \ldots) \wedge \ldots (Z_1 \vee Z_2 \vee \ldots)$$

Términos como P_1 deben ser literales, lo que significa que no contienen conectores lógicos como el condicional o bicondicional, o cuantificadores. Una literal es una fórmula atómica o una fórmula atómica negada. Por ejemplo, la siguiente fbf:

$$(A \vee B) \wedge (\sim B \vee C)$$

está en forma normal conjuntiva. Los términos entre paréntesis son cláusulas:

$$A \vee B \text{ y } \sim B \vee C$$

Como se mostrará más adelante, cualquier fbf de lógica de predicados, que incluye a la lógica proposicional como un caso especial, puede escribirse en forma de cláusulas. La **forma de cláusula** completa puede expresar cualquier fórmula de la lógica de afirmaciones, pero tal vez no sea tan natural o legible para una persona (Kowalski 79). La sintaxis de PROLOG es el conjunto de la cláusula de Horn, que facilita en gran medida la comprobación de los teoremas mecánicos con PROLOG y hace su implantación más eficiente que la notación de lógica de predicados normal o la forma de cláusula completa. Como se mencionó en el capítulo 1, PROLOG sólo permite un encabezado. Por lo general, una expresión en forma de cláusula completa se escribe de una forma especial llamada forma de cláusula de Kowalski:

$$A_1, A_2, \ldots A_N \rightarrow B_1, B_2, \ldots B_M$$

la cual se interpreta como si se dijera que si todos los subobjetivos $A_1, A_2, \ldots A_N$ son verdaderos, entonces uno o más de $B_1, B_2, \ldots$ o B_N también son verdaderos. Obsérvese que a veces la dirección de la flecha está invertida en esta notación. Esta cláusula, escrita en notación de predicados normal, es:

$$A_1 \wedge A_2 \ldots A_N \rightarrow B_1 \vee B_2 \ldots B_N$$

Esto puede expresarse en forma disyuntiva como la disyunción de literales utilizando la equivalencia:

$$p \rightarrow q \equiv \sim p \vee q$$

así:

$$A_1 \wedge A_2 \ldots A_N \rightarrow B_1 \vee B_2 \ldots B_M$$
$$\equiv \sim(A_1 \wedge A_2 \ldots A_N) \vee (B_1 \vee B_2 \ldots B_M)$$
$$\equiv \sim A_1 \vee \sim A_2 \ldots \sim A_N \vee B_1 \vee B_2 \ldots B_M$$

donde la ley de De Morgan:

$$\sim(p \wedge q) \equiv \sim p \vee \sim q$$

se usa para simplificar la última expresión.

Como se comentó en el capítulo 1, PROLOG utiliza un tipo restringido de forma de cláusula, la cláusula de Horn, en la que sólo se permite un encabezado.

$$A_1, A_2, \ldots A_N \rightarrow B$$

que se escribe en sintaxis de PROLOG como:

$$B :- A_1, A_2, \ldots A_N$$

El problema que surge al tratar de probar directamente un teorema, es la dificultad de deducirlo utilizando sólo las reglas de inferencia y los axiomas del sistema. Esto puede requerir mucho tiempo para deducir un teorema o tal vez no seríamos lo suficientemente listos para hacerlo. Para probar que un teorema es verdadero, se utiliza el método clásico de ***reducción al absurdo***, o método de contradicción; en éste, se trata de probar que la fbf negada es un teorema, si resulta una contradicción, entonces la fbf original no negada es un teorema.

El objetivo básico de la resolución es inferir una nueva cláusula, el **resolutivo**, de otras dos cláusulas, llamadas ***cláusulas padre***; el resolutivo tendrá menos términos que los padres. Al seguir el proceso de resolución, terminará por obtenerse una contradicción o el proceso se terminará porque no se ha hecho ningún progreso. En el siguiente argumento se muestra un ejemplo simple de resolución.

$$\begin{array}{l} A \vee B \\ \underline{A \vee \sim B} \\ \therefore A \end{array}$$

Una manera de ver cómo se llega a la conclusión, es escribiendo las premisas como:

$$(A \vee B) \wedge (A \vee \sim B)$$

Uno de los axiomas de distribución es:

$$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$$

Aplicando esto a las premisas se tiene:

$$(A \vee B) \wedge (A \vee \sim B) \equiv A \vee (B \wedge \sim B) \equiv A$$

donde el último paso seguido desde $(B \wedge \sim B)$ siempre es falso. Esto se desprende de la Ley del medio excluido, que establece que algo no puede ser verdadero y falso a la vez. En la lógica de confusión, analizada en el capítulo 5, veremos que esta ley no se sostiene. Otra

manera de escribir esto utiliza el término **nulo**, que significa vacío, nada o falso. Por ejemplo, un apuntador nulo en Pascal apunta a nada, y la Ley del medio excluido establece que $(B \wedge \sim B) \equiv$ nulo.

El ejemplo de resolución muestra la manera en que las cláusulas padre $(A \vee B)$ y $(A \vee \sim B)$ pueden simplificarse en el resolutivo A. En la tabla 3.13 se resumen algunas cláusulas padre básicas y sus resolutivos en notación de cláusula, donde las comas que separan a las cláusulas significa $\wedge$.

Cláusulas padre	Resolutivo	Significado
$p \rightarrow q, p$ o $\sim p \vee q, p$	q	*Modus ponens*
$p \rightarrow q, q \rightarrow r$ o $\sim p \vee q, \sim q \vee r$	$p \rightarrow r$ o $\sim p \vee r$	Encadenamiento o silogismo hipotético
$\sim p \vee q, p \vee q$	q	Combinación
$\sim p \vee \sim q, p \vee q$	$\sim p \vee p$ o $\sim q \vee q$	VERDADERO (una tautología)
$\sim p, p$	nulo	FALSO (una contradicción)

Tabla 3.13 Cláusulas y resolutivos

3.11 SISTEMAS DE RESOLUCIÓN Y DEDUCCIÓN

Dadas las fbf $A_1, A_2, \ldots A_N$ y una conclusión lógica o teorema C, sabemos que

$$A_1 \wedge A_2 \ldots A_N \vdash C$$

es equivalente a establecer que

$$(1)\ A_1 \wedge A_2 \ldots A_N \rightarrow C \quad \equiv \sim(A_1 \wedge A_2 \ldots A_N) \vee C$$
$$\equiv \sim A_1 \vee \sim A_2 \ldots \sim A_N \vee C$$

es válida. Suponga que tomamos la negación como se muestra en seguida:

$$\sim[A_1 \wedge A_2 \ldots A_N \rightarrow C]$$

Ahora

$$p \rightarrow q \equiv \sim p \vee q$$

y así lo anterior se vuelve:

$$\sim[A_1 \wedge A_2 \ldots A_N \rightarrow C] \equiv \sim[\sim(A_1 \wedge A_2 \ldots A_N) \vee C]$$

A partir de la ley de De Morgan,

$$\sim(p \vee q) \equiv \sim p \wedge \sim q$$

y así lo anterior se vuelve:

$$(2) \quad \sim[A_1 \wedge A_2 \ldots A_N \rightarrow C] \equiv [\sim\sim(A_1 \wedge A_2 \ldots A_N) \wedge \sim C] \equiv A_1 \wedge A_2 \ldots A_N \wedge \sim C$$

Ahora si (1) es válida, entonces su negación (2) debe ser inválida. En otras palabras, si (1) es una tautología, entonces (2) debe ser una contradicción. Las fórmulas (1) y (2) representan dos maneras equivalentes de comprobar que una fórmula C es un teorema. La fórmula (1) puede utilizarse para probar un teorema mediante verificación para ver si es verdadera en todos los casos; al mismo tiempo, la fórmula (2) puede utilizarse para probar un teorema al mostrar que (2) conduce a una contradicción.

Como se mencionó en la sección anterior, la comprobación de un teorema mostrando que su negación lleva a una contradicción, se prueba mediante la *reducción al absurdo*. La parte principal de este tipo de prueba es la **refutación**; refutar algo significa probar que es falso. La resolución es una regla de inferencia sólida que es también una **refutación completa**, porque la cláusula vacía siempre será el resultado eventual si hay una contradicción en el conjunto de cláusulas. En esencia, esto significa que si hay una contradicción la **refutación de resolución** terminará en un número finito de pasos. Aunque la refutación de resolución no puede indicarnos cómo *producir* un teorema, sin duda nos indicará si una fbf *es* un teorema.

Como un ejemplo simple de prueba por refutación de resolución, tomemos el argumento:

$$\begin{array}{l} A \rightarrow B \\ B \rightarrow C \\ \underline{C \rightarrow D} \\ \therefore A \rightarrow D \end{array}$$

Para probar que la conclusión $A \rightarrow D$ es un teorema por refutación de resolución, primero se convierte a forma disyuntiva utilizando la equivalencia:

$$p \rightarrow q \equiv \sim p \vee q$$

Así;

$$A \rightarrow D \equiv \sim A \vee D$$

y su negación es:

$$\sim(\sim A \vee D) \equiv A \wedge \sim D$$

La conjunción de las formas disyuntivas de las premisas y la conclusión negada da la forma conjuntiva normal adecuada para refutar la resolución:

$$(\sim A \vee B) \wedge (\sim B \vee C) \wedge (\sim C \vee D) \wedge A \wedge \sim D$$

El método de resolución puede aplicarse ahora a las premisas y la conclusión. En la figura 3.18 se muestra un método para representar la refutación de una resolución con la forma de un diagrama de **árbol de refutación de una resolución**, donde se resuelven las cláusulas del mismo nivel. La raíz, que es el resolutivo final, es nula, como puede verificarse en la última fila de la tabla 3.13 para ~p, p, y así la conclusión original $A \rightarrow D$ es un teorema.

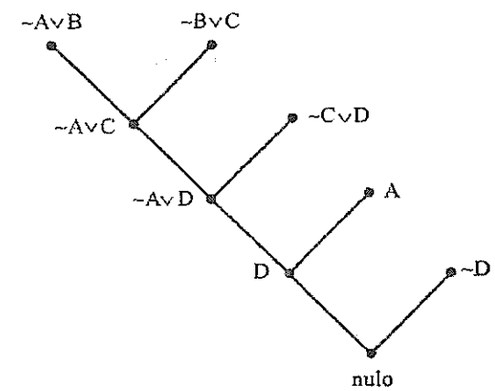


Figura 3.18 Árbol de refutación de resolución

3.12 RAZONAMIENTO SUPERFICIAL Y CAUSAL

Los sistema de resolución y de reglas de producción son dos paradigmas populares para la comprobación de teoremas. Aunque la mayoría considera a los teoremas en el sentido matemático, se ha visto que un teorema en realidad es la conclusión de un argumento lógico válido. Ahora, consideremos un sistema experto que utiliza una cadena de inferencia; en general, una cadena más larga representa conocimiento más causal o profundo, mientras que el razonamiento superficial suele utilizar una sola regla o unas cuantas inferencias. Además de la longitud de la cadena de inferencia, la calidad del conocimiento en las reglas también es un factor importante para distinguir el razonamiento profundo y superficial. A veces se usa otra definición de conocimiento superficial, llamada **conocimiento de experiencia**, que es el conocimiento basado en la experiencia.

La conclusión de una cadena de inferencia es un teorema porque lo prueba la cadena de inferencia, como se demostró en el ejemplo anterior:

$$A \rightarrow B,\ B \rightarrow C,\ C \rightarrow D \vdash A \rightarrow D$$

De hecho, los sistemas expertos que usan una cadena de inferencia para establecer una conclusión están utilizando teoremas; este resultado es muy importante porque de otra manera no sería posible utilizarlos para la inferencia causal. Por supuesto, los sistemas expertos estarían restringidos a inferencias superficiales de reglas simples sin encadenamiento.

A continuación se verán algunas reglas para comparar mejor el razonamiento superficial y profundo. Para empezar, considere la siguiente regla, en la que el número entre paréntesis es sólo para propósitos de identificación:

```
(1)  SI un automóvil tiene
        una buena batería
        buenas bujías
        gasolina
        buenas llantas
     ENTONCES el automóvil puede avanzar
```

Ésta es una regla perfectamente correcta que podría utilizarse en un sistema experto.

Una de las funciones importantes de un sistema experto es su capacidad de explicación, como se analiza en el capítulo 1. Los sistemas expertos basados en reglas facilitan que el sistema explique su razonamiento. En este caso, si el usuario pregunta cómo puede avanzar el automóvil, el sistema experto podría responder con una lista de sus elementos condicionales:

```
una buena batería
buenas bujías
gasolina
buenas llantas
```

Se trata de un tipo de mecanismo de explicación elemental, porque el sistema presenta una lista sólo de los elementos condicionales de la regla. Es posible diseñar medios de explicación más sofisticados para presentar una lista de reglas previas que se han disparado y que dieron como resultado el disparo de la regla actual. Otros medios de explicación pueden permitir que el usuario haga preguntas del tipo "qué pasaría si" para explorar rutas alternativas de razonamiento.

Esta regla también es un ejemplo de **razonamiento superficial**, es decir, en este hay poca o nula comprensión de la causa y el efecto, porque existe poca o nula cadena de inferencia. En esencia, la regla anterior es heurística, porque todo el conocimiento está contenido en una regla. Ésta se activa cuando sus elementos condicionales se satisfacen y no porque el sistema experto comprenda la función de los elementos condicionales. En el razonamiento superficial, hay poca o nula **cadena causal** de la causa y el efecto de una regla a otra. En el caso más simple, la causa y el efecto están contenidos en una regla que no guarda relación con ninguna otra. Si se considera a las reglas desde el punto de vista de los fragmentos de conocimiento analizados en el capítulo 1, el razonamiento superficial no realiza las conexiones entre los fragmentos, de modo que es como una reacción refleja simple.

La ventaja del razonamiento superficial en comparación con el razonamiento causal, es la facilidad de programación; esto significa que el tiempo de desarrollo es más corto y el programa es más pequeño y rápido, y su desarrollo es más barato.

Los marcos y las redes semánticas son dos modelos útiles para el **razonamiento profundo** o causal. El término *profundo* se usa a menudo como sinónimo de razonamiento causal para indicar un conocimiento profundo del tema. Sin embargo, una comprensión profunda requiere que además de comprender la cadena causal por la que ocurre el proceso, también se comprenda el proceso en un sentido abstracto.

Un sistema experto clásico que utiliza marcos y que tiene una comprensión causal es el sistema Steamer, construido por la Marina de Estados Unidos para enseñar la operación de naves con plantas de propulsión (Hollan 84). Un sistema experto profundo es el Tutor LISP, que está diseñado para actuar como un tutor inteligente para enseñar LIPS a los estudiantes (Reiser 85).

Podemos agregar razonamiento causal simple a nuestra regla por medio de la definición de reglas adicionales como:


```
(2)  SI la batería es buena
     ENTONCES hay electricidad

(3)  SI hay electricidad
        y las bujías son buenas
     ENTONCES las bujías producirán una chispa

(4)  SI las bujías producen una chispa
        y hay gasolina
     ENTONCES el motor encenderá

(5)  SI el motor enciende
        y hay buenas llantas
     ENTONCES el automóvil avanzará
```

Observe que con el razonamiento causal, el mecanismo de explicación puede indicar lo que hace cada componente del automóvil, porque cada elemento está especificado en una regla. Este tipo de sistema causal también facilita la escritura de un sistema de diagnóstico para determinar cuál efecto tendrá un mal componente. El razonamiento causal puede utilizarse para un refinamiento arbitrario de la operación de un sistema limitado por la velocidad de ejecución, el tamaño de la memoria y el aumento en el costo de desarrollo.

El razonamiento causal puede utilizarse para construir un **modelo** del sistema real que se comporte en todos los aspectos como el elemento real. Este tipo de modelo puede utilizarse para simulación, con el fin de explorar el razonamiento hipotético del tipo de consultas "qué pasaría si". Sin embargo, los modelos causales no siempre son necesarios ni deseables. Por ejemplo, el sistema MUD sirve como consultor para ingenieros de líquidos de perforación (Kahn 85). El líquido de perforación, llamado *mud* (fango) debido a su parecido con el fango, es una ayuda importante en la perforación por muchas razones, como el enfriamiento y la lubricación del taladro. MUD diagnostica problemas con el fango y sugiere tratamientos.

Un sistema causal no sería muy útil porque, en condiciones normales, el ingeniero de perforación no podría observar la cadena causal de los acontecimientos que ocurren más allá de la superficie. En cambio, el ingeniero sólo puede observar los síntomas en la superficie y no los eventos intermedios no observables de importancia potencial para el diagnóstico.

La situación es muy diferente en medicina, donde los médicos tienen un amplio rango de pruebas de diagnóstico que pueden utilizarse para verificar los eventos intermedios. Por ejemplo, si una persona se queja de sentirse enferma, el médico puede comprobar si hay fiebre; de ser así, puede tratarse de una infección y realizará un análisis de sangre. si éste arroja una infección por tétanos, el médico puede verificar si la persona ha sufrido cortadas con un objeto oxidado en fecha reciente. Por el contrario, si el líquido de perforación se vuelve salado, el ingeniero sospecharía que el taladro ha perforado un manto de sal. Sin embargo, no hay una manera simple de verificar esto, porque no es posible entrar en la perforación, y una prueba sísmica es cara y no siempre resulta confiable. En virtud de que los ingenieros de perforación por lo general no pueden probar las hipótesis intermedias como los médicos, no resuelven los problemas de diagnóstico de la misma manera que éstos, y MUD refleja este método.

Otra razón para no utilizar el razonamiento causal en MUD es que existe un número muy limitado de diagnósticos y síntomas. La mayor parte de las pruebas relevantes que utiliza MUD son rutinarias y se introducen de antemano, por lo que hay pocas ventajas en una sesión de consulta interactiva con un ingeniero para explorar rutas de diagnóstico alternas, si el sistema ya conoce todas las pruebas y rutas de diagnóstico relevantes. Si hubiera muchas

rutas de diagnóstico posibles a seguir con una hipótesis verificable intermedia, habría ventajas para el conocimiento causal porque el ingeniero podría trabajar con el sistema para reducir la búsqueda a rutas factibles.

Debido al mayor número de requisitos para el razonamiento causal, sería necesario combinar ciertas reglas en un razonamiento superficial. El método de resolución con refutación puede utilizarse para probar que una sola regla es una conclusión verdadera de varias reglas. La regla simple es el teorema que se probará mediante resolución.

Por ejemplo, suponiendo que queremos probar que la regla (1) es una conclusión lógica de las reglas (2) a (5). Utilizando las siguientes definiciones proposicionales, las reglas pueden expresarse como sigue:

```
B = la batería es buena      A = el automóvil avanzará
E = hay electricidad         C = las bujías producen una chispa
G = hay gasolina             P = el motor prenderá
S = las bujías son buenas    L = hay buenas llantas
```

(1) $B \wedge S \wedge G \wedge L \rightarrow A$
(2) $B \rightarrow E$
(3) $E \wedge S \rightarrow C$
(4) $C \wedge G \rightarrow P$
(5) $P \wedge L \rightarrow A$

El primer paso para aplicar la refutación de resolución es negar la conclusión o regla de objetivo:

(1') $\sim(B \wedge S \wedge G \wedge L \rightarrow A) = \sim[\sim(B \wedge S \wedge G \wedge L) \vee A]$
$= \sim[-B \vee -S \vee -G \vee -L \vee A]$

Ahora, cada una de las otras reglas se expresa en forma disyuntiva utilizando equivalencias como:

$p \rightarrow q \equiv -p \vee q$ y $\sim(p \wedge q) \equiv -p \vee -q$

para obtener las siguientes nuevas versiones de (2) a (5):

(2') $\sim B \vee E$
(3') $\sim(E \wedge S)$ $C = \sim E \vee \sim S \vee C$
(4') $\sim(C \wedge G)$ $P = \sim C \vee \sim G \vee P$
(5') $\sim(P \wedge L)$ $A = \sim P \vee \sim L \vee A$

Como se describe en la sección anterior, una manera conveniente de representar los resolutivos sucesivos de (1') a (5') es con un árbol de refutación de resolución, como se muestra en la figura 3.19. Empezando en la parte superior del árbol, las cláusulas se representan como nodos, que se resuelven para producir el siguiente resolutivo. Por ejemplo:

$\sim B \vee E$ y $\sim E \vee \sim S \vee C$

se resuelven para inferir:

$\sim B \vee \sim S \vee C$

que se resuelve luego con:

$\sim C \vee \sim G \vee P$

para inferir:

$\sim B \vee \sim S \vee \sim G \vee P$

y así sucesivamente. Para simplificar el dibujo, los últimos resolutivos estan implícitos, en vez de dibujar cada uno de ellos.

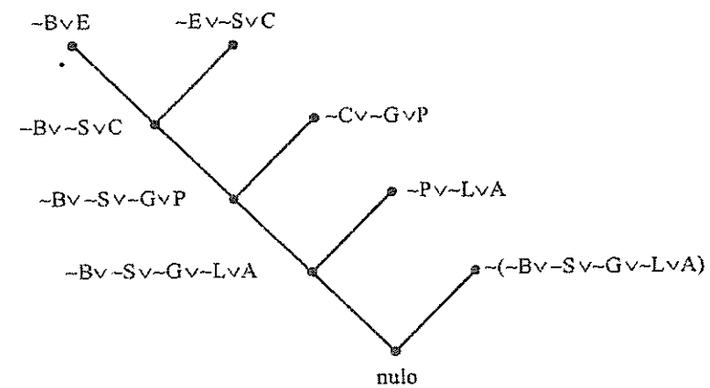


Figura 3.19 Árbol de refutación de la resolución para el ejemplo del automóvil

Como la raíz del árbol es nula, esta resolución es una contradicción. Por refutación, la conclusión original:

$B \wedge S \wedge G \wedge L \rightarrow A$

es un teorema porque su negación lleva a una contradicción. Por tanto, la regla (1) se desprende lógicamente de la regla (2) a la (5).

3.13 RESOLUCIÓN Y LÓGICA DE PREDICADO DE PRIMER ORDEN

El método de resolución también se utiliza con la lógica de predicados de primer orden. En realidad, es el principal mecanismo de inferencia de PROLOG. Sin embargo, antes de que pueda aplicarse la resolución, debe ponerse una fbf en forma de cláusula. Por ejemplo:

```
Algunos programadores odian las fallas
Ningún programador odia el éxito
∴ Ninguna falla es un éxito
```

Definiendo los siguientes predicados:

```
P(x) = x es un programador
F(x) = x es una falla
E(x) = x es un éxito
O(x,y) = x odia a y
```

Las premisas y la conclusión negada se escriben como:

```
(1) (∃ x) [P(x) ∧ (∀ y) (F(y) → O(x,y))]
(2) (∀ x) [P(x) → (∀ y) (E(y) → ~O(x,y))]
(3) ~(∀ y) (F(y) → ~E(y))
```

donde la conclusión se ha negado como preparación para la resolución.

Conversión a la forma de cláusula

Los siguientes nueve pasos son un algoritmo para convertir las fbf de predicado de primer orden a la forma de cláusula. Este procedimiento se ilustra utilizando la fbf (1) de la página anterior.

1. Eliminar condicionales, →, utilizando la equivalencia

```
p → q ≡ ~p ∨ q
```

de modo que la fbf(1) se convierte en:

```
(∃ x) [P(x) ∧ (∀ y) (~F(y) → O(x,y))]
```

2. Cuando sea posible, eliminar las negaciones o reducir el ámbito de la negación a un átomo. Utilizar las equivalencias mostradas en el apéndice A como:

```
~~p ≡ q
~(p ∧ q) ≡ ~p ∨ ~q
~(∃ x) P(x) ≡ (∀ x) ~P(x)
~(∀ x) P(x) ≡ (∃ x) ~P(x)
```

3. Estandarizar las variables dentro de una fbf, para que las variables de límite o modelos de cada cuantificador tengan nombres únicos. Nótese que los nombres de variable de un cuantificador son modelos. Es decir:

```
(∀ x) P(x) ≡ (∀ y) P(y) ≡ (∀ z) P(z)
```

y así, la forma estandarizada de:

```
(∃ x) ~P(x) ∨ (∀ x) P(x)
```

es:

```
(∃ x) ~P(x) ∨ (∀ y) P(y)
```

4. Eliminar cuantificadores existenciales, ∃, utilizando **funciones de Skolem**, denominadas en honor del lógico noruego Thoralf Skolem. Considerado la fbf:

```
(∃ x) L(x)
```

donde L(x) se define como el predicado, que es verdadera si x es < 0. Esta fbf puede reemplazarse con:

L(a)

donde a es una constante como −1, que hace que L(a) sea verdadera. A la a se le llama constante de Skolem, que es un caso especial de la función de Skolem. Para el caso en el que haya un cuantificador universal antes de uno existencial;

```
(∀ x) (∃ y) L(x,y)
```

donde L(x,y) es verdadera si el entero x es menor que el entero y. Esta fbf significa que para cada entero x, hay un entero y mayor que x. Observe que la fórmula no indica cómo calcular *y*, dado un valor para x. Suponiendo que existe una función f(x) que produce una y mayor que x la fbf anterior se le aplica la función de Skolen como:

```
(∀ x) L(x,f(x))
```

La función de Skolem de una variable existencial dentro del ámbito de un cuantificador universal, es una función de todos los cuantificadores de la izquierda. Por ejemplo:

```
(∃ u) (∀ v) (∀ w) (∃ x) (∀ y) (∃ z) P(u,v,w,x,y,z)
```

se le aplica la función de Skolen como:

```
(∀ v) (∀ w) (∀ y) P(a,v,w,f(v,w),y,g(v,w,y))
```

donde a es alguna constante y la segunda función de Skolem, g, debe ser diferente a la primera función, f. Nuestra fbf del ejemplo se convierte en:

```
P(a) ∧ (∀ y) (~F(y) ∨ H(a,y))
```

5. Convertir la fbf a una **forma prenex**, que es una secuencia de cuantificadores. Q, seguida de una **matriz**, M. En general, los cuantificadores pueden ser ∀ o ∃. Sin embargo, en este caso, en el paso 4 ya se han eliminado todos los cuantificadores existenciales y la Q sólo puede ser ∀. Además, como cada ∀ tiene su propia variable modelo, todas las ∀ pueden moverse a la izquierda de la fbf y el ámbito de cada ∀ puede ser toda la fbf.

Nuestro ejemplo se convierte en:

```
(∀ y) [P(a) ∧ (~F(y) ∨ H(a,y)]
```

donde la matriz es el término entre corchetes.

6. Convertir la matriz a la forma conjuntiva normal, que es una conjunción de cláusulas. Cada cláusula es una disyunción. El ejemplo ya es una forma normal conjuntiva donde una cláusula es P(a) y la otra es (~F(y) ∨ O(a,y). Si es necesario, puede utilizarse la siguiente regla distributiva para poner la matriz en la forma normal conjuntiva:

```
p ∨ (q ∧ r) ≡ (p ∨ q) ∧ (p ∨ r)
```

7. Desechar los cuantificadores universales innecesarios porque en esta etapa todas las variables de una fbf deben estar delimitadas. La fbf ahora es la matriz. Nuestra fbf del ejemplo se convierte en:

```
P(a) ∧ (~F(y) ∨ O(a,y))
```

8. Eliminar los signos ∧, escribiendo la fbf como un conjunto de cláusulas. El ejemplo ahora es:

```
{ P(a), ~F(y) ∨ O(a,y) }
```

que por lo general se escribe sin las llaves, como las cláusulas:

```
P(a)
~F(y) ∨ O(a,y)
```

9. Cambiar el nombre de las variables en las cláusulas, si es necesario, para que el mismo nombre de la variable sea el único utilizado en una cláusula. Por ejemplo, si tuviéramos las cláusulas:

```
P(x) ∧ Q(x) ∨ L(x,y)
~P(x) ∨ Q(y)
~Q(z) ∨ L(z,y)
```

pueden renombrarse como:

```
P(x1) ∧ Q(x1) ∨ L(x1,y1)
~P(x2) ∨ Q(y2)
~Q(z) ∨ L(z,y3)
```

Si se realiza el procedimiento para convertir la segunda premisa a la forma de cláusula y la conclusión negada de nuestro ejemplo, al final se obtendrán las cláusulas:

```
(1a) P(a)
(1b) ~F(y) ∨ O(a,y)
(2a) ~P(x) ∨ ~E(y) ∨ ~O(x,y)
(3a) F(b)
(3b) E(b)
```

donde los números aluden a las premisas originales y a la conclusión negada. Por tanto, las premisas (1) y (3) se convierten en dos cláusulas con sufijos (a) y (b), mientras que la premisa (2) se convierte en la cláusula (2a).

Unificación y reglas

Una vez que se han convertido las fbf a la forma de cláusula, suele ser necesario encontrar **casos de sustitución** apropiados para las variables. Es decir, cláusulas como:

```
~F(y) ∨ O(a,y)
F(b)
```

no pueden resolverse con base en el predicado F, hasta que coincidan los argumentos de F. Al proceso de encontrar sustituciones para las variables, de modo que los argumentos coincidan, se le llama **unificación**.

La unificación es una función que distingue a los sistemas expertos de los simples árboles de decisión. Sin unificación, los elementos condicionales de las reglas sólo se podrían igualar

con las constantes, lo que significa que tendría que escribirse una regla específica para cada hecho posible. Por ejemplo, suponiendo que alguien quiere hacer sonar una alarma si un sensor indica humo. Si hay N sensores, se necesitaría una regla para cada sensor, como se muestra a continuación:

```
SI sensor 1 indica humo ENTONCES suena alarma 1
SI sensor 2 indica humo ENTONCES suena alarma 2
...
SI sensor N indica humo ENTONCES suena alarma N
```

Sin embargo, con la unificación, puede utilizarse una variable llamada ?N para el identificador del sensor, de modo que puede escribirse la siguiente regla:

```
SI sensor ?N indica humo
ENTONCES suena alarma ?N
```

Cuando dos literales complementarias están unificadas, pueden eliminarse por resolución. En las dos cláusulas anteriores, la sustitución de y por b resultaría:

```
~F(b) ∨ O(a,b)
F(b)
```

El predicado F ahora se ha unificado y puede resolverse en:

```
O(a,b)
```

Una sustitución se define como el reemplazo simultáneo de variables por términos que los diferencian de las variables, estos pueden ser constantes, variables o funciones. La sustitución de términos por variables se indica con el conjunto:

$$\{ t_1/v_1,\ t_2/v_2\ \ldots\ t_N/v_N \}$$

Si θ es un conjunto como éste y A es un argumento, entonces Aθ se define como el caso de sustitución de A. Por ejemplo, si:

```
θ = { a/x, f(y)/y, x/z }
A = P(x) ∨ Q(y) ∨ R(z)
```

entonces:

```
Aθ = P(a) ∨ Q(f(y)) ∨ R(x)
```

Obsérvese que la sustitución es simultánea, de modo que obtenemos R(x) y no R(a).

Suponiendo que hay dos cláusulas, C_1 y C_2, definidas como:

$$C_1 = \sim P(x)$$
$$C_2 = P(f(x))$$

Una posible unificación de P es:

$$C_1' = C_1\ \{ f(x)/x \} = \sim P(f(x))$$

Otra posible unificación para P son las dos sustituciones utilizando la constante a:

```
C1'' = C1 { f(a)/x } = ~P(f(a))
C2' = C2 { a/x } = P(f(a))
```

Observe que P(f(x)) es más general que P(f(a)), porque hay un número infinito de casos de P(f(x)) que pueden producirse al sustituir una constante por x. A una cláusula como C_1 se le llama la **cláusula más general**.

En general, a una sustitución θ se le llama ***unificador*** para un conjunto de cláusulas { A_1, A_2, ... A_n } si y sólo si $A_1\theta = A_2\theta = \ldots A_n\theta$. A un conjunto que tiene un identificador se le llama ***unificable***. El **unificador más general (umg)** es aquel del que todos los demás unificadores son casos. Esto puede expresarse de manera más formal estableciendo que θ es el unificador más general si y sólo si para cada unificador, α, de un conjunto, hay una sustitución β tal que:

$$\alpha = \theta\beta$$

donde θβ es la sustitución compuesta creada al aplicar primero θ y luego β. El **algoritmo de unificación** es un método para encontrar el unificador más general, para un conjunto unificable de argumentos finito (Chang 73; Nilsson 80).

Para el ejemplo, con cláusulas (1a) a (3b), los resultados de la sustitución y la unificación se muestran en el árbol de refutación de la resolución de la figura 3.20. Como la raíz es nula, la conclusión negada es falsa y, por tanto, la conclusión "ninguna falla es un éxito" es válida.

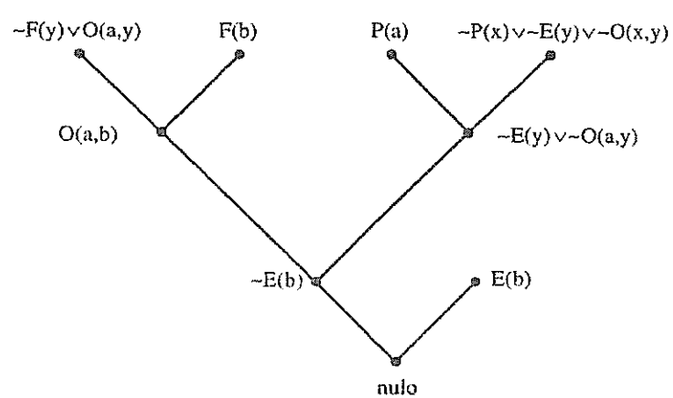


Figura 3.20 Árbol de refutación de la resolución para probar que ninguna falla es un éxito

Los ejemplos utilizados hasta ahora son simples y su resolución es directa, aunque tediosa para los seres humanos. Sin embargo, en muchas otras situaciones el proceso de resolución puede llegar a un punto muerto, de modo que es necesario el rastreo hacia atrás para probar cláusulas alternas para resolución. Aunque la resolución es muy eficaz y es la base de PROLOG, puede resultar ineficiente para algunos problemas. Uno de los problemas de la resolución es que

no tiene una estrategia eficiente de búsqueda integrada y, por tanto, el programador debe proporcionar heurística, como el corte de PROLOG, para una búsqueda eficiente.

Se han investigado varias versiones modificadas de resolución, como la **preferencia de unidad**, la **resolución de entrada**, la **resolución lineal** y el **conjunto de soporte** (Amble 87). La principal ventaja de la resolución reside en que es una técnica eficaz y sencilla que resulta adecuada para muchos casos. Esto hace más fácil la construcción de sistemas mecánicos como PROLOG que un sistema que pretende implantar muchas reglas de inferencia diferentes. Otra ventaja de la resolución es que cuando tiene éxito, proporciona automáticamente una prueba al mostrar la secuencia de pasos que lleva a nulo.

3.14 ENCADENAMIENTO HACIA ADELANTE Y HACIA ATRÁS

A un grupo de varias inferencias que conectan un problema con su solución se le llama **cadena**. A una cadena que se explora o recorre desde un problema hasta su solución se le llama cadena hacia adelante. Otra manera de describir el encadenamiento hacia adelante es el razonamiento que va de los hechos a las conclusiones que se desprenden de ellos. Una cadena que se recorre de la hipótesis a los hechos que la sustentan, es una cadena hacia atrás. Otra manera de describir una cadena hacia atrás es desde el punto de vista de un objetivo que puede lograrse al satisfacer los subobjetivos. Como se muestra con esto, la terminología utilizada para describir el encadenamiento hacia adelante y hacia atrás depende del problema que se está analizando.

El encadenamiento puede expresarse con facilidad a partir de la inferencia. Por ejemplo, suponiendo que se tienen reglas del tipo *modus ponens*:

```
p → q
p
---------
∴ q
```

que forma una cadena de inferencia, como la siguiente:

```
elefante(x) → mamífero(x)
mamífero(x) → animal(x)
```

Estas reglas pueden usarse en una cadena causal de inferencia hacia adelante que deduzca que Clyde es un animal, dado que Clyde es un elefante. La cadena de inferencia se ilustra en la figura 3.21. Observe que el mismo diagrama también ilustra el encadenamiento hacia atrás.

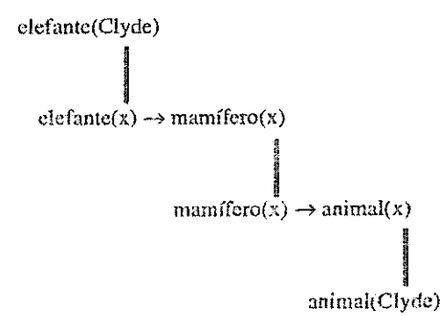


Figura 3.21 Cadena causal hacia adelante

En la figura 3.21, la cadena causal se representa con la secuencia de barras, que conectan el consecuente de una regla con el antecedente de la siguiente. Una barra también indica la unificación de las variables con los hechos. Por ejemplo, la variable x en el predicado elefante(x) debe unificarse primero con el hecho elefante(Clyde), antes de que pueda aplicarse la regla del elefante. La cadena causal es en realidad una secuencia de implicaciones y unificaciones, como se muestra en la figura 3.22.

```
                elefante(Clyde)
Unificación             |
Implicación     elefante(Clyde) → mamífero(Clyde)
Unificación                             |
Implicación                     mamífero(Clyde) → animal(Clyde)
```

Figura 3.22 Cadena causal explícita

El encadenamiento hacia atrás es el proceso inverso. Suponiendo que queremos probar la hipótesis animal(Clyde), el problema central del encadenamiento hacia atrás es encontrar una cadena que vincule la evidencia con la hipótesis. Al hecho elefante(Clyde), se le llama **evidencia** en el encadenamiento hacia atrás para indicar que se usa para sustentar la hipótesis, de la misma manera en que la evidencia en un juzgado se utiliza para probar la culpabilidad del acusado.

Como un ejemplo simple de encadenamiento hacia adelante y hacia atrás, suponga que está conduciendo un automóvil y de pronto ve una patrulla con las luces encendidas. Por encadenamiento hacia adelante puede inferir que la policía quiere detenerlo a usted o a alguien más; es decir, los hechos iniciales sustentan dos conclusiones posibles. Si el carro de policía se detiene justo atrás del suyo o si le hace señas, una inferencia adicional es que el oficial quiere detenerlo a usted y no a alguien más. Adoptando esto como hipótesis de trabajo, usted puede aplicar el encadenamiento hacia atrás para razonar el por qué.

Algunas posibles hipótesis intermedias son que conduce ebrio, con exceso de velocidad, con algún malfuncionamiento en el equipo y que conduce un vehículo robado. Ahora debe examinarse la evidencia para sustentar estas hipótesis intermedias. ¿Fue la botella de cerveza que arrojó por la ventana, ir a 160 en una zona con velocidad máxima de 50 kilómetros por hora, las luces traseras rotas o que las placas identifican el automóvil robado que está conduciendo? En este caso, cada pieza de evidencia sustenta una hipótesis intermedia y así todas son verdaderas. Cualquiera de estas hipótesis intermedias, o todas ellas, son razones posibles para probar la hipótesis de trabajo de que la policía quiere detenerlo.

Es útil visualizar el encadenamiento hacia adelante y hacia atrás como una ruta a través del espacio de problema en la que los estados intermedios corresponden a hipótesis intermedias bajo encadenamiento hacia atrás o conclusiones intermedias bajo encadenamiento hacia adelante. En la tabla 3.14 se resumen algunas de las características comunes del encadenamiento hacia adelante y hacia atrás. Observe que las características de esta tabla sólo pretenden ser una guía. Sin duda es posible diagnosticar en un sistema de encadenamiento hacia adelante y la planeación en un encadenamiento hacia atrás. En particular, la explicación se facilita en el encadenamiento hacia atrás debido a que el sistema puede explicar con facilidad y exactitud qué objetivo trata de cumplirse. En el encadenamiento hacia adelante, la explicación no se facilita tanto, porque los subobjetivos no se conocen de manera explícita sino hasta que se descubren.

En la figura 3.23 se ilustra el concepto básico del encadenamiento hacia adelante en un sistema basado en reglas. Las reglas se disparan por los hechos que satisfacen su antecedente o lados izquierdos (LI). Por ejemplo, para activar la regla R_1, deben satisfacerse los hechos B y C; sin embargo, sólo el hecho C está presente, de modo que R_1 no se activa. La regla R_2 se

activa con los hechos C y D, que están presentes, de modo que R_2 produce el hecho intermedio H. Otras reglas que se satisfacen son R_4, R_6, R_7, R_8 y R_9. La ejecución de las reglas R_6 y R_9 produce las conclusiones del proceso de encadenamiento hacia delante. Estas conclusiones pueden ser otros factores, salida, etcétera.

Encadenamiento hacia adelante	Encadenamiento hacia atrás
Planeación, supervisión, control	Diagnóstico
Presente a futuro	Presente a pasado
Antecedente a consecuencia	Consecuencia a antecedente
Controlado por datos, razonamiento de abajo hacia arriba	Controlado por objetivos, razonamiento de arriba hacia abajo
Trabaja hacia adelante para encontrar cuáles soluciones se desprenden de los hechos	Trabaja hacia atrás para encontrar los hechos que sustenten la hipótesis
Búsqueda primero a lo ancho facilitada	Búsqueda primero a fondo facilitada
Los antecedentes determinan la búsqueda	Las consecuencias determinan la búsqueda
Explicación no facilitada	Explicación facilitada

Tabla 3.14 Algunas características del encadenamiento hacia adelante y hacia atrás

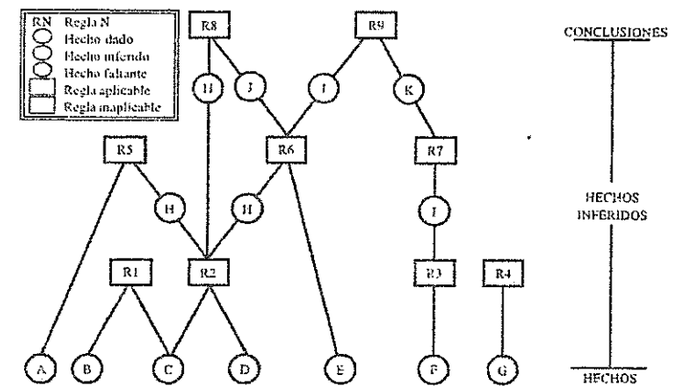


Figura 3.23 Encadenamiento hacia adelante

Al encadenamiento hacia adelante se le llama **razonamiento de abajo hacia arriba**, porque se razona a partir de la evidencia o los hechos de nivel más bajo a las conclusiones de nivel superior que se basan en los hechos. El razonamiento de abajo hacia arriba de un sistema experto es análogo a la programación convencional de abajo hacia arriba que se analiza en el capítulo 1. Los hechos son las unidades elementales del paradigma basado en el conocimiento, porque no pueden descomponerse en ninguna unidad más pequeña con significado. Por ejemplo, el hecho "pato" tiene significados definidos como un sustantivo o como un verbo. Sin embargo, si se divide aún más, el resultado son las letras p, a, t y o, que no tienen significado especial. En programas convencionales, las unidades básicas de significado son los datos.

Por costumbre, los constructores de nivel superior que están compuestos por los de nivel inferior se ponen en la parte superior. De modo que al razonamiento que parte de constructores de nivel superior, como las hipótesis, hacia abajo, hasta los hechos de nivel inferior que pueden sustentarla, se le llama **razonamiento de arriba hacia abajo** o encadenamiento hacia atrás. En la figura 3.24 se ilustra el concepto de encadenamiento hacia atrás. Para probar o refutar la hipótesis H, debe probarse por lo menos una de las hipótesis intermedias, H_1, H_2 o

H_3. Nótese que este diagrama se dibuja como un árbol Y-O para indicar que en algunos casos, como H_2, deben estar presentes todas las hipótesis de nivel inferior para sustentarla. En otros casos, como las hipótesis de nivel superior, H, sólo es necesaria una hipótesis de nivel inferior. En el encadenamiento hacia atrás, por lo general el sistema producirá evidencia del usuario para ayudar a probar o refutar la hipótesis. Esto contrasta con un sistema de encadenamiento hacia delante, en el que todos los hechos relevantes suelen conocerse por anticipado.

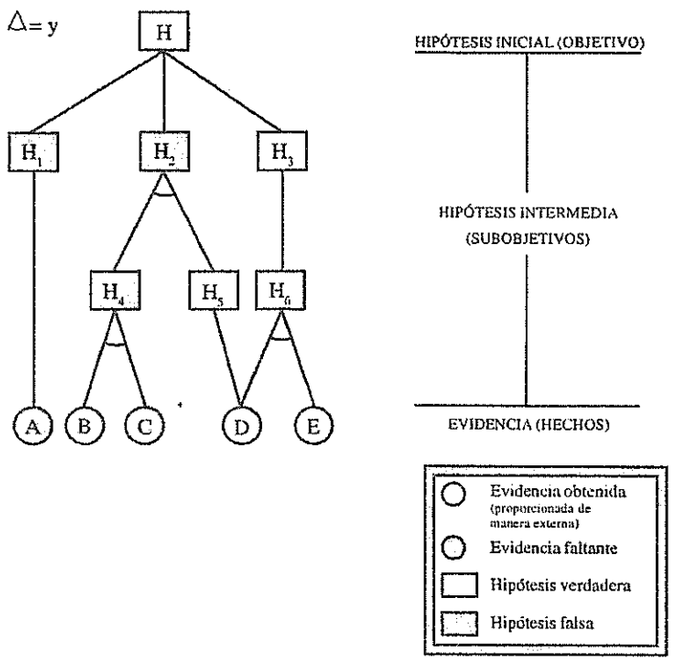


Figura 3.24 Encadenamiento hacia atrás

Si se vuelve a observar la figura 3.4 (página 100), se verá que la red de decisión está bien organizada para el encadenamiento hacia adelante. Las hipótesis de nivel superior son los tipos diferentes de frambuesas como la frambuesa con flores, la negra, la morada y la roja. La evidencia para sustentar estas hipótesis desciende al nivel inferior. Es posible escribir con facilidad las reglas para identificar una frambuesa. Por ejemplo:

```
SI las hojas son simples ENTONCES es frambuesa con flores
```

Para obtener evidencia es importante hacer las preguntas correctas; aquellas que mejoran la eficiencia para determinar la respuesta correcta. Un requisito obvio para esto es que el sistema experto sólo debe hacer preguntas que se relacionen con la hipótesis que se está tratando de probar; aunque el sistema puede hacer cientos o miles de preguntas, debe pagarse un costo en tiempo y dinero para obtener la evidencia que permita responder las preguntas. Además, la acumulación de ciertos tipos de evidencia, como los resultados de pruebas médicas, puede ser incómoda y tal vez peligrosa para el paciente (en realidad, es difícil pensar en un examen médico placentero).

En condiciones ideales, el sistema experto también debe permitir al usuario presentar voluntariamente la evidencia, aunque el sistema no la haya pedido. Al permitir que el usuario presente evidencia se acelera el proceso de encadenamiento hacia atrás y hace que el sistema sea más conveniente. La evidencia presentada de manera voluntaria puede llevar al sistema a omitir algunos vínculos en la cadena causal o seguir un método totalmente nuevo. La desventaja es que se requiere una programación más compleja del sistema experto, porque éste tal vez no siga una cadena vínculo por vínculo.

En la figura 3.25 se muestran buenas aplicaciones para el encadenamiento hacia adelante y hacia atrás. Por simplicidad, estos diagramas se dibujan como árboles en lugar de hacerlo como una red general. Una buena aplicación para el encadenamiento hacia adelante ocurre si el árbol es ancho y no muy profundo; esto se debe a que el encadenamiento hacia adelante facilita una búsqueda primero a lo ancho. Es decir, este encadenamiento es bueno si la búsqueda de conclusiones avanza nivel por nivel. Por el contrario, el encadenamiento hacia atrás facilita una búsqueda primero a fondo. Un buen árbol para este tipo de búsqueda es estrecho y profundo.

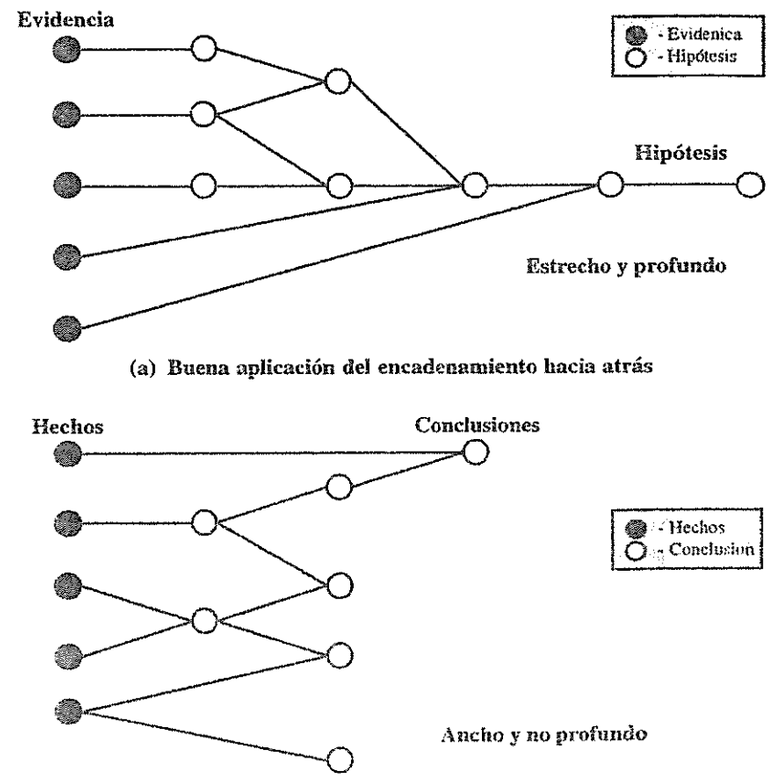


Figura 3.25 Encadenamiento hacia atrás y hacia adelante

Obsérvese que la estructura de las reglas determina la búsqueda de una solución. Es decir, la activación de una regla depende de los patrones que según su diseño, debe satisfacer. Los patrones del lado izquierdo determinan si una regla puede activarse mediante hechos. Las acciones del lado derecho determinan los hechos que se afirman y eliminan, de modo que afectan otras reglas. Existe una situación análoga en el encadenamiento hacia atrás, pero se utilizan hipótesis en lugar de reglas. Desde luego, una hipótesis intermedia puede ser tan sólo una regla que se equipara con su consecuencia en lugar de su antecedente.

Consideremos las siguientes reglas del tipo SI...ENTONCES:

```
SI A ENTONCES B
SI B ENTONCES C
SI C ENTONCES D
```

Si se da el hecho A y el mecanismo de inferencia está diseñado para comparar los hechos con los antecedentes, entonces los hechos intermedios B y C serán acertados y se alcanzará la conclusión D. Este proceso corresponde al encadenamiento hacia adelante.

En contraste, si el hecho (en realidad, la hipótesis) D es acertado y el mecanismo de inferencia compara los hechos con las consecuencias, el resultado corresponde al encadenamiento hacia atrás. En los sistemas diseñados para este tipo de encadenamiento, como PROLOG, el mecanismo incluye varias funciones, como el rastreo automático hacia atrás, para facilitar el encadenamiento.

El encadenamiento hacia atrás puede lograrse en un sistema de encadenamiento hacia adelante, y viceversa, si se rediseñan las reglas. Por ejemplo, las reglas anteriores para el encadenamiento hacia delante pueden rescribirse como:

```
SI D ENTONCES C
SI C ENTONCES B
SI B ENTONCES A
```

Ahora C y B se consideran subobjetivos o hipótesis intermedias que deben cumplirse para satisfacer la hipótesis D. La evidencia A es el hecho que indica el final de la generación de subobjetivos. Si hay un hecho A, entonces se sustenta a D y se considera verdadera bajo esta cadena de inferencia hacia atrás. Si no hay A, entonces no se sustenta a la hipótesis D y se considera falsa.

Una dificultad con este método es la eficiencia. Un sistema de encadenamiento hacia atrás facilita la búsqueda primero a fondo, mientras que un sistema de encadenamiento hacia delante facilita la búsqueda primero a lo ancho. Aunque es posible escribir una aplicación de encadenamiento hacia atrás en un sistema de encadenamiento hacia delante, y viceversa, el sistema no será tan eficiente en su búsqueda de una solución. La segunda dificultad es conceptual, el conocimiento obtenido del especialista tendrá que modificarse para cumplir las demandas del mecanismo de inferencia. Por ejemplo, un mecanismo de inferencia de encadenamiento hacia adelante compara el antecedente de las reglas y uno de encadenamiento hacia atrás compara la consecuencia. Es decir, si el conocimiento del especialista es, de manera natural, un encadenamiento hacia atrás, tendrá que reestructurarse por completo para aplicarlo en un modo de encadenamiento hacia delante, y viceversa.

3.15 OTROS MÉTODOS DE INFERENCIA

A veces, se usan otros tipos de inferencia con los sistemas expertos. Aunque estos métodos no son de propósito tan general, como la deducción, son muy útiles.

Analogía

Además de la deducción y la inducción, otro método de inferencia eficaz es la **analogía**. La idea básica del razonamiento por analogía es tratar de relacionar situaciones previas como guías para las nuevas. La mayor parte de las criaturas vivas son muy buenas para aplicar el razonamiento por analogía en sus vidas, lo que es esencial debido al enorme número de situaciones nuevas que se encuentran en el mundo real. En lugar de tratar cada situación nueva como única, a menudo es útil tratar de relacionarlas con aquellas que ya son familiares. El razonamiento analógico se relaciona con la inducción, pero mientras que la inducción hace inferencias de lo específico a lo general de la misma situación, la analogía trata de hacer inferencias a partir de situaciones que no son iguales. La analogía no puede hacer pruebas formales como la deducción. En cambio, la analogía es una herramienta de razonamiento heurístico que a veces funciona.

Un ejemplo de razonamiento por analogía es el diagnóstico médico. Cuando usted acude a un doctor debido a un problema de salud, el doctor obtendrá información de usted y anotará los síntomas; si sus síntomas son idénticos o muy similares a los de otras personas con el problema X, el doctor puede inferir por analogía que usted tiene el problema X. Por supuesto, también puede usarse el encadenamiento hacia atrás.

Obsérvese que este diagnóstico no es una deducción, porque usted es único. El simple hecho de que alguien con el mismo problema muestre ciertos síntomas, no significa que usted mostrará esos síntomas; en cambio, el doctor supone que sus síntomas lo hacen análogo a una persona con los mismos síntomas y problema conocido. Este diagnóstico inicial es una hipótesis que los exámenes médicos pueden probar o refutar. Es importante tener una hipótesis de trabajo inicial, porque ésta reduce los miles de problemas potenciales. Sería muy caro y consumiría tiempo empezar a realizar cada examen posible sin una hipótesis inicial.

Como ejemplo de la utilidad del razonamiento por analogía, supóngase que dos personas participan en un juego llamado el Juego del 15 (Fischler 87). Eligen números entre el 1 y el 9 por turnos, con la restricción de que no puede usarse el mismo número dos veces, y gana la primera persona cuyos números sumen 15. Aunque a primera vista parece un juego que requiere algún razonamiento, una analogía puede facilitar el juego.

Veamos el tablero de gato con valores asignados a cada celda, como se muestra abajo:

6	1	8
7	5	3
2	9	4

Se trata de un **cuadro mágico** porque la suma de los valores en las filas, columnas y diagonales es una constante, 15. Este tablero con los valores del cuadro mágico puede considerarse una analogía del Juego del 15. Ahora resulta muy fácil jugarlo si lo considera un gato y luego traduce la estrategia ganadora al Juego del 15.

A este cuadro se le llama el **cuadro estándar** de **orden** 3. El término orden alude al número de filas o columnas de un cuadro, por tanto, sólo hay un cuadro de orden 3. Es posible crear otros cuadros mágicos si se gira o invierte el cuadro estándar. Otra manera de construir cuadros mágicos es agregando la misma constante a cada celda. Conocer esta información nos permite deducir la estrategia ganadora para el Juego del 18, donde los números deben seleccionarse del conjunto:

```
{2, 3, 4, 5, 6, 7, 8, 9, 10}
```

o el juego del 21, utilizando el conjunto:

{3, 4, 5, 6, 7, 8, 9, 10, 11}

Ahora podemos utilizar la inducción para inferir la estrategia ganadora para el juego del 15 + 3N, donde N es cualquier número natural 1, 2, 3,... al considerar al juego como análogo al tablero de gato, que es análogo a un cuadro mágico de valores:

{1+N, 2+N, 3+N, 4+N, 5+N, 6+N, 7+N, 8+N, 9+N}

Utilizando la analogía del cuadro de orden 3 para juegos de tres movimientos, por inducción podemos inferir que los cuadros de orden superior pueden usarse para juegos que implican más de tres movimientos. Por ejemplo, el cuadro de orden 4:

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

le permite jugar la estrategia ganadora del Juego del 34 de cuatro movimientos al considerarlo desde el punto de vista del gato. En contraste con el cuadro estándar de orden 3, hay 880 cuadros estándar de orden 4, lo que le permite muchos juegos más (Smith 84).

El razonamiento por analogía es una parte importante del razonamiento de sentido común, que es muy difícil para las computadoras (y los niños). Otras aplicaciones de la analogía han servido para el aprendizaje (Carbonell 82).

Ensayo y error

Otro método de inferencia es la clásica estrategia de la AI de **ensayo y error**, a veces llamada generación y prueba, que requiere la generación de una solución probable y luego probarla para ver si la solución propuesta cumple todos los requisitos. Si la solución es satisfactoria, entonces se abandona, de otro modo genera una nueva solución, prueba de nuevo y así sucesivamente. Este método se utilizó en el primer sistema experto, DENDRAL, concebido en 1965, para ayudar a identificar estructuras moleculares orgánicas (Buchanan 78). Un espectrómetro de masas proporciona datos de una muestra desconocida y los introduce en DENDRAL, que genera todas las estructuras moleculares potenciales que podría producir el espectrograma desconocido. Luego DENDRAL prueba las moléculas más probables estimulando sus espectrogramas de masa y comparándolos con el original desconocido. Otro programa que utiliza generar y probar es el programa AM (Artificial Mathematician: matemático artificial), que infiere nuevos conceptos matemáticos (Lenat 82).

Con el fin de reducir el enorme número de soluciones posibles, ensayo y error suele utilizarse con un programa de planeación que limita las soluciones probables para generación. A esta variación se le llama ***plan-ensayo y error*** y se usa por eficiencia en muchos sistemas. Por ejemplo, el sistema experto de diagnóstico médico MYCIN también tiene la capacidad de planear un tratamiento terapéutico con medicamentos, después de que se ha diagnosticado la enfermedad de un paciente (Chancey 85). En esencia, un **plan** consiste en encontrar cadenas de reglas o inferencias que conectan un problema con una solución u objetivo, con la evidencia para sustentarla. La planeación se hace con mayor eficiencia al buscar de manera simultánea hacia delante a partir de los hechos y hacia atrás a partir del objetivo.

El planeador MYCIN crea primero una lista de los medicamentos terapéuticos a los que el paciente es sensible. Para reducir interacciones indeseables entre los medicamentos, es mejor limitar el número de éstos que recibe un paciente, aunque se piense que sufre de varias infecciones diferentes. El generador toma la lista con prioridades del planeador y genera sublistas de uno o dos medicamentos, si es posible. Luego se prueban la eficacia de estas sublistas contra las infecciones, las alergias del paciente y otras consideraciones, antes de tomar la decisión de administrarlas.

Generar y probar también puede considerarse el paradigma de inferencia básico de reglas. Si se satisfacen los elementos condicionales de una regla, genera algunas acciones como nuevos hechos. El mecanismo de inferencia prueba estos hechos contra los elementos condicionales de las reglas en la base de conocimiento. Las reglas que se satisfacen se ponen en la agenda y la regla de mayor prioridad genera sus acciones, que entonces se prueban, etcétera. Por tanto, generar y probar produce una cadena de inferencia que puede llevar a la solución correcta.

Abducción

La inferencia por **abducción** es otro método que se utiliza con frecuencia en la solución de un problema de diagnóstico. El esquema de abducción se parece al de *modus ponens*, pero en realidad es muy diferente, como se muestra en la tabla 3.15.

Abducción	***Modus ponens***
$p \rightarrow q$	$p \rightarrow q$
q	p
$\therefore p$	$\therefore q$

Tabla 3.15 Comparación de abducción y *modus ponens*

Abducción es otro nombre para un argumento falaz que analizamos en la sección 3.6, como la Falacia del recíproco. Aunque la abducción no es un argumento deductivo válido, es un método útil de inferencia y se ha usado en los sistemas expertos. Al igual que la analogía, que tampoco es un argumento deductivo válido, la abducción puede ser útil como una regla heurística de inferencia. Es decir, cuando no tenemos un método deductivo de inferencia, la abducción puede resultar útil, pero no está garantizado que funcione. Ni la analogía, ni generar y probar, ni la abducción son deductivos ni está garantizado que funcionen todo el tiempo. A partir de premisas verdaderas, estos métodos no pueden probar conclusiones verdaderas. Sin embargo, estas técnicas son útiles en la reducción del espacio de búsqueda al generar hipótesis razonables que luego pueden utilizarse con la deducción.

A veces se hace referencia a la abducción como un razonamiento de hechos observados para una mejor explicación (Reggia 85). Consideremos lo siguiente como ejemplo de abducción:

```
SI x es un elefante ENTONCES x es un animal
SI x es un animal ENTONCES x es un mamífero
```

Si sabemos que Clyde es un mamífero, ¿Podemos concluir que Clyde es un elefante?

La respuesta a esta pregunta depende de si estamos hablando de la realidad o de nuestro sistema experto. En la realidad no podríamos obtener esta conclusión con algún grado de certidumbre. Clyde podría ser un perro, un gato, una res o cualquier otro tipo de animal que sea un mamífero pero no un elefante. En realidad, al considerar cuántas especies de animales existen, sin conocer ninguna información adicional sobre Clyde, la probabilidad de que sea un elefante es muy baja.

No obstante, en un sistema experto que sólo incluye las reglas anteriores, diríamos por abducción con un 100 por ciento de certidumbre que si Clyde es un mamífero, entonces es un elefante. Esta inferencia se desprende de una **suposición de mundo cerrado** en la que suponemos que no existe nada más fuera del mundo cerrado de nuestro sistema experto, cualquier cosa que no pueda probarse en él se considera como falsa. Bajo la suposición del mundo cerrado, se conocen todas las posibilidades. Como el sistema experto sólo contiene estas dos reglas y sólo un elefante puede ser un mamífero, entonces si Clyde es un mamífero, debe ser un elefante.

Suponiendo que añadimos una tercera regla, como se muestra en seguida:

```
SI x es un perro ENTONCES x es un animal
```

Aun podemos operar nuestro sistema experto bajo una suposición de mundo cerrado; sin embargo, ya no podemos concluir con un 100 por ciento de certeza que Clyde es un elefante, todo lo que podemos asegurar es que Clyde es un elefante o un perro. Para decidir entre las dos, se necesita más información. Por ejemplo, si hay otra regla;

```
SI x es un perro ENTONCES x ladra
```

y hay evidencias de que Clyde ladra, las reglas pueden revisarse como se muestra a continuación:

```
(1) SI x es un animal ENTONCES x es un mamífero
(2) SI x ladra ENTONCES x es un animal
(3) SI x es un perro ENTONCES x ladra
(4) SI x es un elefante ENTONCES x es un animal
```

Ahora puede hacerse una cadena de inferencia abductiva hacia atrás, empleando las reglas (1), (2) y (3) para demostrar que Clyde debe ser un perro.

Una cadena de abducción hacia atrás no es lo mismo que el significado acostumbrado de encadenamiento hacia atrás. El término *encadenamiento hacia atrás* significa que estamos tratando de probar una hipótesis al buscar evidencia para sustentarla; y se usaría para tratar de probar que Clyde es un mamífero. Desde luego, para nuestro pequeño sistema no hay otras posibilidades. Sin embargo, podrían añadirse otras clasificaciones para reptiles, aves, etcétera.

Si se sabe que Clyde es un mamífero, la abducción podría utilizarse para determinar si Clyde es un elefante o un perro. El encadenamiento hacia delante se usaría si se sabe que Clyde es un elefante y queremos saber si es un mamífero. Como puede verse, la elección del método de inferencia depende de lo que habrá de determinarse. Como el encadenamiento hacia delante es deductivo, sólo se garantiza que sus conclusiones son válidas. En la tabla 3.16 se resume el propósito de cada una de las tres técnicas de inferencia.

Varios sistemas de AI y expertos han usado la abducción basada en marcos para solución de problemas de diagnóstico (Basili 85; Miller 82; Pople 82; Reggia 83). En estos sistemas, la base de conocimiento contiene **asociaciones causales** entre trastornos y síntomas. Se hacen inferencias utilizando ensayo y error de hipótesis para los trastornos.

Inferencia	Inicio	Propósito
Encadenamiento hacia delante	Hechos	Conclusiones que deben seguir
Encadenamiento hacia atrás	Conclusión incierta	Hechos para sustentar la conclusión
Abducción	Conclusión verdadera	Hechos que deben seguir

Tabla 3.16 Resumen del propósito de encadenamiento hacia delante, encadenamiento hacia atrás y abducción

Razonamiento no monotónico

En condiciones normales, la adición de nuevos axiomas a un sistema lógico significa que pueden probarse más teoremas porque hay más axiomas a partir de los cuales pueden derivarse nuevos teoremas. A esta propiedad de aumentar los teoremas con el aumento de axiomas se le conoce como ***monotonicidad*** y a los sistemas como la lógica deductiva se les llama ***sistemas monotónicos***.

Sin embargo, puede surgir un problema si un axioma recién introducido contradice parcial o completamente un axioma previo. En este caso, los teoremas que se han probado tal vez ya no sean válidos. Por tanto, en un **sistema no monotónico**, los teoremas no necesariamente aumentan a medida que lo hace el número de axiomas.

El concepto de no monotonicidad tiene una aplicación importante en los sistemas expertos. A medida que se generan nuevos hechos (lo que es análogo a la comprobación de los teoremas), un sistema experto monotónico seguiría elaborando hechos. Puede ocurrir un problema importante si uno o más hechos se vuelven falsos porque un sistema monotónico no pudo tratar con cambios en la veracidad de axiomas y teoremas. Por ejemplo, suponiendo que hay un hecho que afirma la hora, en cuanto pase un segundo, el hecho anterior ya no es válido. Un sistema monotónico no sería capaz de tratar con la nueva situación. Otro ejemplo, sería que el sistema de identificación de una aeronave afirma el hecho de que un blanco era hostil, y más adelante, nueva evidencia prueba que el blanco es amigable. En un sistema monotónico, la identificación original no podría cambiar. Un sistema no monotónico permite la retracción de los hechos.

Como otra aplicación, supongamos que se quiere escribir un medio de explicación para un sistema experto que permitiría al usuario regresar a inferencias previas y explorar rutas de inferencia alternas del tipo "*¿Qué pasaría si?*". Todas las inferencias hechas después de la inferencia deseada previa deben retractarse del sistema. Además de los hechos, las reglas también deben haberse extirpado del sistema, reintegrándose a la base del conocimiento para no monotonicidad. Otras complicaciones surgen en sistemas como OPS5, en los que las reglas pueden crearse automáticamente en el lado derecho de reglas durante la ejecución. Para la no monotonicidad, cualquier regla inferida hecha después de la inferencia deseada anterior tendría que haberse eliminado del sistema. El registro de todas las inferencias hechas puede consumir una gran cantidad de la memoria y reducir en gran medida la velocidad del sistema.

Para proporcionar una no monotonicidad, es necesario adjuntar una justificación o dependencia a cada hecho y regla, que explique la razón para creer en ella. Si se toma una decisión no monotónica, entonces el mecanismo de inferencia puede examinar la justificación de cada hecho y regla, para ver si todavía es creíble, y también para una posible restauración de las reglas extirpadas y los hechos retractados en los que se cree otra vez.

El problema de justificar hechos se señaló por primera vez en el **problema de cuadro**, (McCarthy 69). Éste es un término descriptivo que recibe su nombre del problema de identificar lo que ha cambiado o no ha cambiado en el cuadro de una película. Las películas se fotografían como una sucesión de fotos fijas, llamadas cuadros. Cuando se reproducen a 24 cuadros por segundo o más rápido, el ojo humano no puede distinguir los cuadros individuales y así se da la ilusión de movimiento. Un problema de cuadro para la AI es reconocer los

cambios en un ambiente a través del tiempo. Por ejemplo, considerando el problema del mono y los plátanos (figura 3.7), supongamos que el mono debe pararse en la caja roja para alcanzar los plátanos, de modo que la acción es "empuja la caja roja hasta abajo de los plátanos". Ahora el problema de cuadro es ¿Cómo reconocemos que la caja todavía es roja después de la acción? Al empujar la caja no debe cambiar el ambiente. No obstante, otras acciones como "pinta la caja de azul" *cambiarán* el ambiente. En algunas herramientas de los sistemas expertos, un ambiente se conoce como ***mundo*** y consta de un conjunto de hechos relacionados (Filman 88). Un sistema experto puede llevar el registro de varios mundos para hacer razonamiento hipotético de manera simultánea. Al problema de mantener la corrección o el valor de verdad de un sistema se le llama ***conservación de verdad*** (Doyle 79). Ésta, o una variación llamada ***conservación de la verdad basada en suposiciones***, es esencial para mantener la pureza de cada mundo al retractarse de hechos injustificados (de Kleer 86).

Consideremos el clásico ejemplo del pájaro Piolín como un ejemplo simple de razonamiento no monotónico. En ausencia de cualquier otra información, supondríamos que como Piolín es un pájaro, puede volar. Éste es un ejemplo de **razonamiento predeterminado**, algo muy parecido a las opciones predeterminadas usadas en las ranuras de los marcos. El razonamiento predeterminado puede considerarse como una regla que hace inferencias acerca de reglas, una **metarregla** que establece

```
SI se sabe que X no es cierto, y
   No hay evidencia de contradicción en X
ENTONCES tentativamente concluye Y
```

Las metarreglas se analizan de manera más extensa en la siguiente sección.

En nuestro caso la metarregla tiene la forma específica:

```
X es la regla "Todos los pájaros pueden volar" y
   El hecho es que "Piolín es un pájaro"
Y es la inferencia "Piolín puede volar"
```

Desde la perspectiva de las reglas de producción, esto puede expresarse como una regla en nuestra base de conocimiento, que indica:

```
SI X es un pájaro ENTONCES X puede volar
```

y el hecho que existe en la memoria de trabajo es:

```
Piolín es un pájaro
```

La unificación del hecho con el antecedente de la regla, produce la inferencia de que Piolín puede volar.

Ahora viene el problema. Supongamos que se añade un hecho adicional a la memoria de trabajo que dice que Piolín es un pingüino. Sabemos que los pingüinos no pueden volar, de modo que la inferencia de que Piolín puede volar es incorrecta. Por supuesto, debe haber una regla en el sistema que también establezca este conocimiento o el hecho se ignorará.

Para que un sistema siga siendo correcto, debe eliminarse la inferencia incorrecta. Sin embargo, tal vez no baste con esto si otras inferencias se basaron en la inferencia incorrecta. Es decir, tal vez otras reglas usaron la inferencia incorrecta como evidencia para dibujar inferencias adicionales, etcétera. Este es un problema de conservación de la verdad. La inferencia de que Piolín puede volar fue una **inferencia plausible**, basada en el razonamiento predeterminado. (El término *plausible* significa "no imposible" y se analizará más a fondo en el capítulo 4.)

Una manera de permitir la inferencia no monotónica es al definir un operador de frase M, que puede definirse informalmente como "es consistente" (McDermont 82). Por ejemplo,

```
(∀ x) [Pájaro(x) ∧ M(Puede_volar(x)) → Puede_volar(x)]
```

puede definirse como "Para toda x, si x es un pájaro y es consistente que un pájaro puede volar, entonces x puede volar". Una manera más informal de establecer esto es "Casi todos los pájaros pueden volar". El término "es consistente" significa que no hay contradicción con otro conocimiento. Sin embargo, se ha criticado esta interpretación como si en realidad estableciera que sólo los pájaros que no pueden volar son aquellos que se ha inferido que no pueden volar (Moore 85). Se trata de un ejemplo de **razonamiento autoepistemológico**, que significa literalmente razonamiento sobre el conocimiento propio. Ambos tipos de razonamiento, predeterminado y autoepistemológico se usan en el **razonamiento de sentido común**, que los seres humanos suelen aplicar muy bien pero que resulta muy difícil para las computadoras.

El razonamiento autoepistemológico es un razonamiento acerca del propio conocimiento que es distinto del conocimiento en general. Por lo general, una persona puede hacer esto muy bien porque conoce los límites de su conocimiento. Por ejemplo, supongamos que un extraño llega frente a usted y le dice que fue su cónyuge, de inmediato sabría (a menos que padeciera amnesia) que esto no es cierto porque no lo conoce. La metarregla general del razonamiento autoepistemológico es:

```
SI no tengo conocimiento de X
ENTONCES X es falso
```

Observe cómo el razonamiento autoepistemológico depende de la suposición de mundo cerrado. Se supone que cualquier hecho que no se conoce es falso. En este razonamiento, el mundo cerrado es su conocimiento propio.

Tanto el razonamiento autoepistemológico como el predeterminado son no monotónicos. Sin embargo, las razones son diferentes; el razonamiento predeterminado es no monotónico porque es **derrotable**. Este término significa que cualquier inferencia es tentativa y tal vez tenga que retirarse cuando se dispone de nueva información. No obstante, el razonamiento autoepistemológico puro no es derrotable debido a la suposición del mundo cerrado que declara que usted ya conoce todo el conocimiento verdadero. Por ejemplo, como la gente casada sabe quién es su cónyuge (a menos que quieran olvidarlo), una persona casada no aceptará que un extraño fue su cónyuge, aunque se lo diga. Como la mayoría reconoce que no tiene recuerdos perfectos, no se adhieren al razonamiento autoepistemológico puro. Por supuesto, las computadoras no tienen este problema.

El razonamiento autoepistemológico es no monotónico debido al significado de la afirmación autoepistemológica **sensible al contexto**. El término *sensible al contexto* significa que el significado cambia con el contexto. Como ejemplo simple de sensibilidad al contexto, puede considerarse la manera en que se pronuncia la palabra "sí" en las dos frases:

```
Claro que sí quiero ese aumento
Creo que sí te voy a pagar
```

La pronunciación de "sí" es sensible al contexto.

Ahora consideremos un sistema que consta de los dos siguientes axiomas:

```
(∀ x) [Pájaro(x) ∧ M(Puede_volar(x)) → Puede_volar(x)]
Pájaro(Piolín)
```

En este sistema lógico, Puede_volar(Piolín) es un teorema derivado de la unificación de Piolín con la variable x y la implicación.

Ahora supongamos que se agrega un nuevo axioma que establece que Piolín no puede volar y, por tanto, contradice el teorema previamente derivado.

```
~Puede_volar(Piolín)
```

El operador M debe cambiar su operación sobre su argumento sobre todo porque ahora M(Puede_volar(Piolín)) no es consistente con el nuevo axioma. En este nuevo contexto de tres axiomas, el operador M no daría un resultado VERDADERO para Puede_volar(Piolín) porque entra en conflicto con el nuevo axioma. El valor devuelto por el operador M debe ser FALSO en el nuevo contexto, y así la conjunción es FALSA. Por ello, no hay implicación para producir el teorema Puede_volar(Piolín) y no hay conflicto.

Una manera de implantar esto por medio de reglas es como sigue:

```
SI x es un pájaro Y x es típico
ENTONCES x puede volar
SI x es un pájaro Y x es atípico
ENTONCES x no puede volar

Piolín es un pájaro
Piolín es atípico
```

Observe que este sistema no invalida la conclusión Puede_volar(Piolín), sino que más bien evita que se dispare la regla incorrecta. Éste es un método mucho más eficiente de conservación de la verdad que si tenemos una regla y el axioma especial ~Puede_volar(Piolín). Ahora tenemos un sistema más general que puede manejar fácilmente a otros pájaros no voladores sin que se tengan que agregar continuamente nuevas inferencias, lo que se supone que hace el sistema.

3.16 METACONOCIMIENTO

El programa Meta-DENDRAL usa la inducción para inferir nuevas reglas de estructura química. Meta-DENDRAL es un intento por sobreponerse al cuello de botella del conocimiento de tratar de extraer reglas estructurales moleculares de los especialistas humanos. Meta-DENDRAL ha tenido mucho éxito en redescubrir reglas conocidas e inferir nuevas.

Por ejemplo, la siguiente metarregla se toma del programa de adquisición de conocimiento TEIRESIAS para MYCIN, el sistema experto para diagnosticar infecciones sanguíneas y meningitis (Feigenbaum 79).

```
METARREGLA 2
SI
El paciente es un anfitrión comprometido, y
Hay reglas que mencionan pseudomonas en su premisa, y
Hay reglas que mencionan klebsiellas en su premisa
```



```
ENTONCES
Hay evidencia que sugiere (.4) que lo primero
   debe hacerse antes de lo último
```

(El número .4 en la acción de la regla es un grado de certidumbre y se analizará en un capítulo posterior.)

TEIRESIAS adquiere interactivamente el conocimiento de un perito. Si MYCIN hace un diagnóstico incorrecto, entonces TEIRESIAS llevará al especialista de regreso por la cadena del razonamiento incorrecto hasta que establezca el punto donde inicia. Mientras recorre hacia atrás la cadena de razonamiento, TEIRESIAS también interactúa con el perito para modificar reglas incorrectas o adquirir nuevas reglas.

El conocimiento sobre nuevas reglas no se introduce inmediatamente en MYCIN. En cambio TEIRESIAS verifica si la nueva regla es compatible con reglas similares. Por ejemplo, si la nueva regla describe la forma en que una infección entra en el cuerpo y otras reglas aceptadas tienen un elemento condicional que establece la puerta de entrada entonces la nueva regla deberá tener el mismo elemento. Si la nueva regla no establece la puerta de entrada, entonces TEIRESIAS consultará al usuario acerca de esta discrepancia. TEIRESIAS tiene un patrón de **modelo de regla,** de reglas similares, conocidas y trata de adecuar la nueva regla a él. En otras palabras, el modelo de regla es el conocimiento que TEIRESIAS tiene acerca de su conocimiento. Una situación análoga para una persona ocurriría si va con un distribuidor de automóviles para comprar un automóvil nuevo y el vendedor trata de venderle un automóvil con tres ruedas.

El metaconocimiento de TEIRESIAS es de dos tipos: la METARREGLA 2, descrita anteriormente, es una estrategia de control que indica la manera en que se aplican las reglas. Por otra parte, el tipo de modelo de regla de metaconocimiento determina si la nueva regla tiene la forma apropiada para introducirse en la base del conocimiento. En un sistema experto basado en reglas, al hecho de determinar si la nueva regla está en la forma correcta se le llama ***verificación*** de la regla. Al hecho de determinar si una cadena de inferencias correctas lleva a la respuesta correcta se le llama ***validación***. Ambas son tan interdependientes que suele usarse el acrónimo **VyV** para referirse a ellas. Una definición más coloquial de los términos, tomada de la ingeniería de software es (Boehm 84):

```
Verificación:  "¿Estoy elaborando el producto correctamente?"
Validación:    "¿Estoy elaborando el producto correcto?"
```

V y V se analizará con más detalle en un capítulo posterior.

3.17 RESUMEN

En este capítulo se expusieron los métodos de inferencia para los sistemas expertos de uso frecuente. La inferencia es particularmente importante en ellos porque es la técnica con que resuelven los problemas. Se analizó la aplicación de árboles, gráficas y la representación de conocimiento. Se ilustraron también las ventajas de estas estructuras para desarrollar inferencias.

Se analizó la lógica proposicional y de predicados de primer orden. Las tablas de verdad y las reglas de inferencia se describieron como medios para probar teoremas y afirmaciones. Se mencionaron las características de los sistemas lógicos: estar completos, ser sólidos y permitir decisiones.

El método de resolución para comprobar teoremas se analizó para la lógica proposicional y la de predicados de primer orden. Los nueve pasos relacionados con la conversión de una

fórmula bien adecuada a la forma de cláusula se ilustró con un ejemplo. La aplicación de la función de Skolem, la forma normal prenex y la unificación se analizaron en el contexto de la conversión de las fbf a forma de cláusula.

Se analizó además otro importante método de inferencia, la analogía. Aunque no se usa ampliamente en sistemas expertos debido a la dificultad para implantarla, la gente la utiliza con frecuencia y debe considerarse en el diseño de sistemas expertos. También se expuso la de ensayo y error con un ejemplo de su uso en MYCIN. Se describió la aplicación de metaconocimiento en TEIRESIAS y su relación con la verificación y validación de sistemas expertos.

PROBLEMAS

3.1 Escriba un programa de árbol de decisión que sea de autoaprendizaje. Enséñele el conocimiento de los animales de la figura 3.3.

3.2 Escriba un programa que traducirá automáticamente el conocimiento almacenado en un árbol de decisión binario en reglas del tipo SI ... ENTONCES. Pruébelo con el árbol de decisión animal del problema 3.1.

3.3 Dibuje una red semántica que contenga el conocimiento sobre frambuesas de la figura 3.4.

3.4 Dibuje el diagrama de estado que muestre una solución al problema clásico del granjero, el zorro, la cabra y la calabaza. En este problema, un granjero, un zorro, una cabra y una calabaza se encuentran en el banco de un río. Debe utilizarse un bote para transportarlos a la otra orilla. Sin embargo, sólo caben dos en el bote, al mismo tiempo (y sólo el granjero puede remar). Si se deja al zorro con la cabra y el granjero no está presente, el zorro se comerá a la cabra. Si se deja a la cabra sola con la calabaza, la cabra se la comerá.

3.5 Dibuje un diagrama de estado para un problema de viaje bien estructurado que tenga

(a) tres métodos de pago: efectivo, cheque y tarjeta
(b) los intereses del viajero: sol, nieve
(c) cuatro destinos posibles, dependiendo de los intereses y el presupuesto del viajero
(d) tres tipos de transporte.

Escriba reglas SI ... ENTONCES para aconsejar al viajero a dónde ir, dependiendo del presupuesto y los intereses. Elija destinos reales y encuentre los costos para ir allí desde el lugar en que se encuentra.

3.6 Determine si los siguientes son argumentos válidos o no:

(a) $p \rightarrow q$, $\sim p \rightarrow r$, r; $\therefore p$
(b) $\sim p \vee q$, $p \rightarrow (r \wedge s)$, $s \rightarrow q$; $\therefore q \vee r$
(c) $p \rightarrow (q \rightarrow r)$, q; $\therefore p \rightarrow r$

3.7 Utilizando el procedimiento de decisión con diagrama de Venn, determine si los siguientes son silogismos válidos o no:

(a) AEE-4
(b) AOO-1
(c) OAO-3
(d) AAI-1

3.8 Pruebe si las siguientes son falacias o reglas de inferencia. Dé un ejemplo de cada una:

(a) Dilema constructivo complejo

$$\begin{array}{l} p \rightarrow q \\ r \rightarrow s \\ \underline{p \vee r} \\ \therefore q \vee s \end{array}$$

(b) Dilema destructivo complejo

$$\begin{array}{l} p \rightarrow q \\ r \rightarrow s \\ \underline{\sim q \vee \sim s} \\ \therefore \sim p \vee \sim r \end{array}$$

(c) Dilema destructivo simple

$$\begin{array}{l} p \rightarrow q \\ p \rightarrow r \\ \underline{\sim q \vee \sim r} \\ \therefore \sim p \end{array}$$

(d) Inverso

$$\begin{array}{l} p \rightarrow q \\ \underline{\sim p} \\ \therefore \sim q \end{array}$$

3.9 Dibuje la conclusión de las siguientes premisas, tomadas de Lewis Carroll, autor de *Alicia en el país de las maravillas*.

(a) Todas las cartas fechadas en este cuarto están escritas en papel azul.
(b) Ninguna de las cartas están escritas con tinta negra, excepto aquellas que están redactadas en tercera persona.
(c) No he archivado ninguna de las cartas que puedo leer.
(d) Ninguna de ellas, que están escritas en una hoja, carece de fecha.
(e) Todas ellas, que no están tachadas, están en tinta negra.
(f) Todas ellas, escritas por Brown, empiezan con "Estimado señor".
(g) Todas ellas, escritas en papel azul, están archivadas.
(h) Ninguna de ellas, escritas en más de una hoja, está tachada.
(i) Ninguna de ellas, que empieza con "Estimado señor", está escrita en tercera persona.

Sugerencia: utilice la ley de los contrapositivos y defina

A = empezando con "Estimado señor"
B = tachada
C = fechada
D = archivada
E = en tinta negra
F = en tercera persona
G = cartas que puedo leer
H = en papel azul
I = en una hoja
J = escritas por Brown

3.10 Dé una prueba formal utilizando la lógica de predicados para el siguiente silogismo:

Ningún software está garantizado
Todos los programas son software
∴ Ningún programa está garantizado

3.11 Utilice lo siguiente para escribir fórmulas de lógica de afirmaciones de primer orden:

P(x) = x es un programador
I(x) = x es inteligente
A(x,y) = x ama a y

(a) Todos los programadores son inteligentes
(b) Algunos programadores son inteligentes
(c) Ningún programador es inteligente
(d) Alguien no es un programador
(e) No todos son programadores
(f) Todos no son programadores
(g) Todos son programadores
(h) Algunos programadores no son inteligentes
(i) Hay programadores
(j) Todos aman a alguien

Sugerencia: use el apéndice B

3.12 Considere el siguiente argumento de lógica de predicados:

Un caballo es un animal

Por tanto, la cabeza de un caballo es la cabeza de un animal. Defina lo siguiente:

C(x,y) = x es la cabeza de y
A(x) = x es un animal
S(x) = x es un caballo

La premisa y la conclusión son:

$(\forall x)(S(x) \rightarrow A(x))$
$(\forall x)\{[(\exists y)(S(y) \wedge C(x,y))] \rightarrow [(\exists z)(A(z) \wedge C(x,z))]\}$

Compruebe la conclusión utilizando la refutación de resolución. Muestre los nueve pasos en una conversión de cláusula para la conclusión.

3.13 (a) Obtenga información de su banco sobre sus ahorros y préstamos bajo el criterio de obtener un préstamo para automóvil, escriba un sistema de reglas con encadenamiento hacia atrás para determinar si un solicitante puede obtener un préstamo para automóvil. Sea lo mas específico posible.
(b) Obtenga información sobre la calificación para la hipoteca de una casa. Proporcione las modificaciones de su programa de préstamo para automóvil para manejar los préstamos hipotecarios igual que los préstamos para automóvil.
(c) Obtenga información sobre la calificación para un préstamo empresarial. Realice las modificaciones al programa de automóvil-hipoteca para manejar los préstamos empresariales.

3.14 Escriba un sistema de producción de reglas causales para simular la operación del carburador de un automóvil. Puede obtener esta información de la guía de reparación de su automóvil. Sea lo más detallado posible.

3.15 Utilizando la guía de reparación de su automóvil, escriba un sistema de producción que pueda diagnosticar y remediar los problemas del carburador de su automóvil, dados los síntomas.

3.16 Si una conclusión es falsa y quiere determinar los hechos que se desprenden de ella, ¿Utilizaría abducción u otro método de inferencia? Explíquelo.

3.17 Escriba reglas SI ... ENTONCES para identificar frambuesas, utilizando la parte del árbol de decisión mostrada en la figura 3.4.

3.18 Un padre y dos hijos están en el banco izquierdo de un río y desean cruzar a la orilla derecha. El padre pesa 80 kilogramos y cada hijo pesa 40. Hay un bote disponible con capacidad de carga de 870 kilogramos. Por lo menos se requiere una persona para remar en el bote. Dibuje el árbol de decisión que muestre todas las posibilidades para cruzar el río e indique en él la ruta exitosa para que puedan cruzar. Tome los siguientes operadores para rotular su arbol:

HI-Un solo hijo cruza de izquierda a derecha
HD-Un solo hijo cruza de derecha a izquierda
AI-Ambos hijos cruzan de izquierda a derecha
AD-Ambos hijos cruzan de derecha a izquierda
PI-El padre cruza de izquierda a derecha
PD-El padre cruza de derecha a izquierda

3.19 (a) Dibuje una versión del diagrama lógico para la decisión sobre el automóvil del Y-O-exclusivo de la figura 3.11. Observe que tendrá que implantar el O-exclusivo utilizando elementos normales Y, O y NO.
(b) Escriba reglas de encadenamiento hacia delante SI ENTONCES para determinar cuál hipótesis seguir (por ejemplo, vender o reparar).

BIBLIOGRAFÍA

(Amble 87). Tore Amble, *Logic Programming and Knowledge Engineering*, Addison-Wesley Pub., 1987, pp. 41-42, 1987.

(Basili 85). V. Basili y C. Ramsey, "ARROWSMITH-P". A. Prototype Expert System for Software Engineering Management", en K. Karna, ed. *Proc. Expert System in Government Symposium*, 1985.

(Boehm 84). Barry W. Boehm, "Verifying and Validating Software Requirements and Design Specifications", *IEEE Software J.*, enero de 1984.

(Buchanan 78). Bruce G. Buchanan y Edward A. Feigenbaum, "Dendral and Meta-Dentral: Their Applications Dimension", en *Artificial Intelligence*, **11**, (1), pp. 5-24, 1978.

(Carbonell 82). James G. Carbonell, "Learning by Analogy: Formulating and Generalizing Plans from Past Experience", en *Machine Learning*, R. S. Michalski, J. G. Carbonelll, y T. M. Mitchell, eds., Tioga Pub. Co. 1982.

(Chancey 85). William J, Chancey, "Details of the Revised Therapy Algorithm", en *Rulebased Expert Systems*, Bruce G. Buchanan y Edward H. Shorliffe, eds., pp. 133-146, 1985.

(Chang 73). Chin-Ling Chang y Richard Char Tung Lee, *Symbolic Logic and Mathematical Theorem Proving*, Academic Press, p. 77, 1973.

(de Kleer 86). J. de Kleer, "An Assumption-based Truth Maintenance System", en *Artif. Intell.*, **28**, (2), pp. 127-162, enero de 1986.

(Doyle 79), J. Doyle, "A Truth Maintenence System", *Artificial Intelligence*, **12** (3), pp. 231-272, 1979.

(Feigenbaum 79), Edward A. Feigenbaum, "Themes and Case Studies of Knowledge Engineering", en *Expert System in the Micro-Electronic Age*, Donald Michie, ed., pp. 3-25, 1979.

(Filman 88). Robert E. Filman, "Reasoning with Worlds and Truth Maintenance in a Knowledge based Programming Enviroment", *C. ACM*, **31**, (4), pp. 382-401, abril de 1988.

(Fischler 87). Martin A., Fischler y Oscar Firscheim, *The Eye, the Brain, and the Computer*, Addison-Wesley Pub., p. 65, 1987.

(Giarratano 92). Joseph Giatarrano, *CLIPS User's Guide*, Vol. I, Rules, NASA, Version 5.1 of CLIPS, 1992.

(Hollan 84). James D. Hollan, *et al*, "Steamer: An Interactive Inspectable Simulation-based Training System", en *AI Magazine*, **5**, (2), pp. 15-27, verano de 1984.

(Kahn 85). Gary Kahn, "On When Diagnostic Systems Want to Do Without Causal Knowledge", en *Advances in Artificial Intelligence*, T. O'Shea, ed., Elsevier Science Publishers, pp. 41-50, 1985.

(Kowalski 79), Robert Kowalski, "Algorithm = Logic + Control", *Comm. ACM*, **22**, (7), pp. 424-436, 1979.

(Lenat 82). Douglas B. Lenat, "AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search", en *Knowledge based System in Artificial Intelligence*, R. Davis y D. B. Lenat, eds., McGraw-Hill, 1982.

(McCarthy 69). J. McCarthy y P. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence", *en Machine Intelligence*, 4, B. Meltzer y D. Michie, eds., Edinburg Univ. Press, pp. 463-502, 1969.

(McDermott 82). D. McDermontt, "Nonmonotonic Logic II: Nonmonotonic Modal Theories", *J. ACM*, 29 (1), pp. 33-57, 1982.

(Miller 82). R. Miller, H. Pople, y J. Myers, "INTERNIST-1, An Experimental Computer-based Diagnostic Consultant for General Internal Medicine", en *New England J. of Medicine*, **307**, pp. 468-476, 1982.

(Minsky 67). Marvin Minsky, *Computation: Finite and Infinite Machines*, Prentice-Hall, p. 228, 1967.

(Moore 85). Robert C. Moore, "Semantical Considerations on Nonmonotonic Logic", en *Artificial Intelligence*, **25**, pp. 75-95, 1985.

(Nilsson 80). Nils J. Nilsson, *Principles of Artificial Intelligence*, Tioga Publishing Co., pp. 142-143, 1980.

(Pople 82). H. Pople, "Heuristic Methods for Imposing Structure on Ill-structured Problems: The Structure of Medical Diagnostics", en P. Szolovits, ed., *Artificial Intelligence in Medicine*, pp. 119-190, 1982.

(Reggia 83). J. Reggia, D. Nau, y P. Wang, "Diagnostic Expert Systems Based on a Set Covering Model", en *Int. J. of Man-Machine Studies*, pp. 437-460, 1983.

(Reggia 85). B. J. Reggia, "Abductive Inference", en Proc. of Expert Systems in Goverment Symposium, K. Karna, ed., *IEEE Press*, pp. 484-489, 1985.

(Reiser 85). B. J. Reiser, *et al.*, "Dynamic Student Modelling in an Intelligent Tutor for LIPS Programming", *IJCAI*, pp. 8-14, 1985.

(Robinson 65). J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle", *J. ACM*, 12, (1), pp. 23-41, enero de 1986.

(Smith 84), Karl J. Smith, *The Nature of Mathematics*, 4th. ed. Brooks/Cole, p. 12, 1984.

(Stebbing 50). L. S. Stebbing, A *Modern Introduction to Logic*, Methuen & Co. Ltd., 7th ed. pp. 104-105, 1950.

CAPÍTULO 4

Razonamiento en condiciones de incertidumbre

4.1 INTRODUCCIÓN

En este capítulo se analizan varios métodos de razonamiento en situaciones de **incertidumbre**. Este tema es muy importante en los sistemas expertos porque en muchas de sus aplicaciones se incluye información dudosa.

4.2 INCERTIDUMBRE

La incertidumbre puede considerarse como la falta de información adecuada para tomar una decisión; esto es un problema porque puede evitar que se tome la mejor e incluso puede provocar que se tome una equivocada. En términos médicos, la incertidumbre puede pasar por alto el mejor tratamiento para un paciente o contribuir a la aplicación una terapia incorrecta. En los negocios, la incertidumbre puede causar pérdidas financieras, en vez de utilidades.

Se han desarrollado varias teorías para manejar la incertidumbre, como la probabilidad clásica, la probabilidad bayesiana, la teoría de Hartley basada en los conjuntos clásicos (Hartley 1928), la teoría de Shannon basada en la probabilidad (Shannon 1948), la teoría de Dempster-Shafer (Shafer 1976) y la teoría de la confusión de Zadeh (Zadeh 1965). En particular la teoría de Shannon ha resultado muy popular en las telecomunicaciones y también se ha aplicado a muchas áreas diferentes como la biología, psicología, música y física.

Todas las criaturas vivas son expertas en el manejo de la incertidumbre, de otra manera no sobrevivirían en el mundo real. Sobre todo, los seres humanos estamos acostumbrados a la incertidumbre acerca del tráfico, el clima, el trabajo, la escuela, etc. Después de un tiempo, todas las personas se conviertan en expertos para manejar bajo diferentes condiciones de tráfico, tratar un resfriado, elegir clases sencillas, etc. El manejo de la incertidumbre exige del razonamiento en condiciones inciertas.

Al método deductivo de razonamiento descrito en el capítulo 3, se le llama ***razonamiento exacto*** porque trata con hechos exactos y con las conclusiones también exactas, que se desprenden de ellos. Recordamos que dentro de un argumento deductivo, la conclusión debe seguir a las premisas. Cuando se enfrentan hechos inciertos, el número de posibles conclusiones puede aumentar enormemente la dificultad para encontrar las mejores.

Desafortunadamente, puede ser muy complicado determinar cuál es la "mejor" conclusión; sin embargo, se han propuesto varios métodos diferentes para manejar la incertidumbre y ayudar en esta especificación. Es responsabilidad del diseñador elegir el método más apropia-

do para un sistema experto, dependiendo de su aplicación. Aunque existen herramientas para los sistemas expertos que incorporan mecanismos de razonamiento en condiciones de incertidumbre, por lo general no son tan flexibles para permitir que se utilicen otros métodos.

Aun cuando es posible construir muchas aplicaciones de los sistemas expertos utilizando el razonamiento exacto, muchas otras requieren un **razonamiento inexacto** que se relacione con hechos o reglas inciertos, o ambos. Ejemplos clásicos de sistemas expertos que manejan con éxito la incertidumbre, son MYCIN para diagnóstico médico y PROSPECTOR para la exploración mineral.

Los sistemas MYCIN y PROSPECTOR, obtienen conclusiones incluso cuando no se conoce toda la evidencia necesaria para comprobarlos de manera absoluta; pero es posible llegar a un conclusión más confiable si se realizan más pruebas aunque existen problemas con el tiempo y el costo crecientes para desarrollarlas. Estas restricciones de tiempo y dinero son particularmente importantes en el caso de los tratamientos médicos pues si se retrasa un tratamiento para aplicar más exámenes, se agregan costos considerables y, entre tanto, el paciente puede morir. En el caso de la exploración mineral, el costo de pruebas adicionales también es un factor importante; sería más costeable que se empezara la perforación cuando hay un 95 por ciento de certeza de éxito que gastar cientos de miles de dólares más para obtener una certeza de 98 por ciento.

4.3 TIPOS DE ERROR

Muchos tipos de error diferentes **pueden** aportar incertidumbre. Las diferentes teorías de incertidumbre tratan de resolver parte o todo esto, para proporcionar la inferencia más confiable. En la figura 4.1 se ilustra un esquema simplificado de la clasificación de errores.

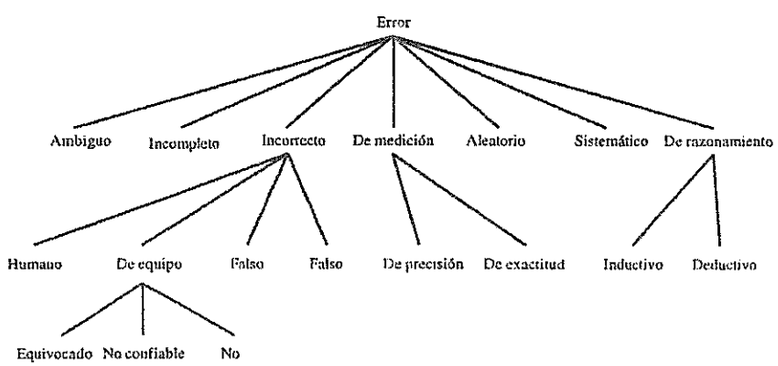


Figura 4.1 Tipos de error

Esta figura debe dibujarse estrictamente como una rejilla porque pueden darse interconexiones entre los diferentes tipos de error. Por ejemplo, el error humano podría vincularse con la ambigüedad, la medición, el razonamiento, etc. En la tabla 4.1 se representan ejemplos de estos errores.

El primer tipo de error que se muestra es la **ambigüedad**, en el que algo puede interpretarse de más de una manera; el segundo tipo es la **falta de información completa**; un tercero es la **incorrección**, donde la información es incorrecta. Las posibles causas de la incorrección son

el error humano, como la falla accidental en la lectura de un cuadrante o de los datos, el engaño o la mala información y el mal funcionamiento del equipo.

Ejemplo	Error	Razón
Apagar la válvula	Ambiguo	¿Cuál válvula?
Activar la válvula 1	Incompleto	¿De qué manera?
Apagar la válvula 1	Incorrecto	Lo correcto es encender
La válvula está pegada	Falso positivo	Las válvulas no se pegan
La válvula no está pegada	Falso negativo	La válvula está pegada
Llevar la válvula 1 a 5	Impreciso	Lo correcto es 5.4
Llevar la válvula 1 a 5.4	No confiable	Lo correcto es 9.2
Llevar la válvula 1 a 5.4, 6 o 0	No confiable	Error del equipo
El valor de la válvula 1 es 5.4, 5.5 o 5.1	Error aleatorio	Fluctuación estadística
El valor de la válvula 1 es 7.5	Error sistemático	Mala calibración
La válvula 1 no está pegada porque nunca se ha pegado antes	Inducción no válida	La válvula está pegada
El resultado es normal y, por tanto, la válvula 1 está en buenas condiciones	Deducción inválida	La válvula se quedó pegada en posición abierta

Tabla 4.1 Ejemplos de errores más comunes

Una **hipótesis** es una suposición que debe probarse; una **hipótesis nula** es la suposición hecha inicialmente, como "la válvula está pegada". A un tipo de información incorrecta se le llama ***falso positivo***, que significa la aceptación de una hipótesis cuando no es verdadera. De igual manera, un ***falso negativo*** significa el rechazo de una hipótesis cuando es verdadera. Por tanto, si la válvula en realidad no está pegada, entonces la aceptación de la hipótesis de que está pegada es un falso positivo, lo que en estadística se denomina ***error de tipo I***. De igual manera, si la válvula realmente está pegada y la hipótesis "la válvula está pegada" se rechaza, se trata de un falso negativo o ***error de tipo II***.

Los siguientes dos tipos de error mostrados en la tabla 4.1 son los **errores de medición**, éstos son fallas de **precisión** y **exactitud**; aunque estos términos se utilizan a menudo como sinónimos, son muy diferentes. Si se examinan dos reglas, una graduada en milímetros y la otra en centímetros, obviamente la primera es más precisa; sin embargo, si su graduación fuera incorrecta, la regla sería inexacta y sus medidas no resultarían confiables, a menos que se conociera un factor de corrección. Esta exactitud corresponde a la verdad, mientras que la precisión corresponde a la corrección con que se conoce la verdad. Para el caso de las reglas, la graduada en milímetros mide con una precisión diez veces mayor que la graduada en centímetros.

Otro tipo de error es la **falta de confiabilidad**. Si el equipo de medición que proporciona los hechos no es confiable, los datos resultan **erráticos**. Una lectura errática es aquella que no es constante sino que fluctúa, por lo que unas veces sería correcta y otras no.

Una lectura que fluctúa puede ser causada por la naturaleza aleatoria básica del sistema que se está estudiando, como la decadencia de los átomos radioactivos. Como se trata de un fenómeno de mecánica cuántica, la tasa de decadencia fluctúa al azar alrededor de un valor promedio, este tipo de fluctuación constituye un **error aleatorio** y conduce a la incertidumbre de la media. Otros tipos de error aleatorio pueden deberse al movimiento browniano, el ruido electrónico causado por efectos térmicos, etc. Sin embargo, las lecturas erráticas pueden deberse a una conexión suelta.

Un **error sistemático** es aquel que no es aleatorio sino que se introduce debido a cierta desviación. Por ejemplo, la mala calibración de una regla, cuya graduación es más pequeña de lo normal, produce un error sistemático que indica que las lecturas son mayores a lo normal.

4.4 ERRORES E INDUCCIÓN

El siguiente tipo de error es una **inducción inválida**: "la válvula no está pegada porque nunca antes se ha pegado". El proceso de **inducción** es lo opuesto de la deducción, pues mientras ésta parte de lo general a lo particular, como en

```
Todos los hombres son mortales
Sócrates es un hombre
```

para deducir la conclusión específica

```
Sócrates es mortal
```

la inducción trata de generalizar de lo particular a lo general, como en

```
Mi disco nunca se ha descompuesto
∴ Mi disco nunca se descompondrá
```

Con excepción de la inducción matemática, nunca puede probarse que los argumentos inductivos son correctos, sólo pueden proporcionar cierto grado de confianza de que la conclusión es correcta. En realidad, no es posible confiar en el argumento inductivo previo. El siguiente es un argumento más sólido:

```
La alarma de fuego sonó
∴ Hay fuego
```

Un argumento aún más fuerte es

```
La alarma de fuego sonó
Huelo a humo
∴ Hay fuego
```

Aunque se trata de un argumento fuerte, no comprueba que haya fuego. El humo podría deberse a que se están asando hamburguesas en una parrilla y la alarma de fuego pudo accionarse por accidente. Lo siguiente si es una prueba de que hay fuego:

```
La alarma de fuego sonó
Huelo a humo
Mis ropas están quemándose
∴ Hay fuego
```

Observe que este es un argumento deductivo, porque es posible ver las flamas.

Los sistemas expertos pueden estar formados por reglas deductivas e inductivas, donde las reglas inductivas son de naturaleza heurística. La inducción también puede aplicarse a la generación automática de reglas, como se estudia en el capítulo referente a la adquisición de conocimiento.

Además de la incertidumbre en relación con los hechos, las reglas de un sistema experto pueden tener incertidumbre si se basan en la heurística; a esto suele llamársele *regla elemental* porque se basa en la experiencia. A veces la experiencia es una buena guía, sin embargo, no puede aplicarse al 100 por ciento de los casos porque la experiencia es heurística. Por ejemplo, se esperaría que todos aquellos que saltaran de un edificio de 50 pisos morirían, no obstante, alguien puede saltar de un edificio de 50 pisos y no morir, si lleva un paracaídas.

Una de las características interesantes de los especialistas humanos es que razonan bien bajo condiciones de incertidumbre, aunque la haya en gran cantidad; ellos suelen hacer buenos juicios, o ya no se les consideraría como especialistas.

Otra característica es que los peritos pueden corregir fácilmente sus opiniones, si algunos de los hechos originales eran incorrectos. A esto se le denomina *razonamiento no monótono*, como se analiza en el capítulo 3. Es más difícil programar sistemas expertos que rastrear hacia atrás sus procesos de razonamiento, porque todos los hechos intermedios deben estar guardados y debe mantenerse una historia del disparo de reglas. Al parecer, los especialistas humanos revisan su razonamiento con poco esfuerzo.

Además de los errores inductivos, también puede haber errores deductivos o falacias, también descritos en el capítulo 3. Un ejemplo de un esquema falaz que se analizó en el mismo capítulo 3, es:

$$\begin{array}{l} p \rightarrow q \\ \underline{q \qquad} \\ \therefore\ p \end{array}$$

Por ejemplo:

```
Si la válvula está en buenas condiciones,
   Entonces la salida es normal
La salida es normal
∴ La válvula está en buenas condiciones
```

es un argumento falaz. La válvula puede estar pegada en la posición abierta, de manera que la salida es normal, pero si es necesario cerrarla el problema aparecerá. Esto puede ser muy serio si la válvula necesita cerrarse rápidamente, como en el caso de un apagón de emergencia en un reactor nuclear.

En contraste con los errores que hemos analizado, los inductivos y deductivos son errores de razonamiento que llevan a la formulación incorrecta de reglas.

Al parecer, los seres humanos no suelen procesar la información incierta de la mejor manera posible (Hink 87). Ni siquiera los peritos son inmunes a los errores, sobre todo en condiciones de incertidumbre. Esto puede ser un importante problema durante la adquisición de conocimiento, cuando el conocimiento del especialista debe cuantificarse en reglas; es posible que se muestren incongruencias, inexactitudes y demás errores posibles de la incertidumbre. Los expertos tendrán entonces que corregir su conocimiento, lo que puede retardar que se complete el sistema experto.

4.5 PROBABILIDAD CLÁSICA

Una herramienta antigua, pero todavía muy importante para la solución de problemas de AI es la **probabilidad** (Farley 1983). Se trata de una manera cuantitativa de manejar la incertidumbre que se originó en el siglo XVII, cuando algunos apostadores franceses pidieron ayuda a importantes matemáticos como Pascal, Fermat y otros. Los juegos de azar se habían vuelto

muy populares y como había grandes cantidades de dinero en disputa, los apostadores buscaban métodos que les ayudaran a calcular sus probabilidades de ganar.

Pascal y Fermat fueron quienes primero propusieron la teoría de la **probabilidad clásica**, en 1654 (Parratt 61). Desde entonces se ha trabajado mucho con la probabilidad y se han desarrollado varias ramas nuevas. Muchas aplicaciones de la probabilidad han servido a la ciencia, la ingeniería, los negocios, la economía y casi a todos los demás campos.

La definición de la probabilidad clásica

A la probabilidad clásica también se le llamó ***probabilidad* a priori** debido a que trata con juegos o sistemas de azar. Como se analiza en el capítulo 2, el término *a priori* significa "antes"; es decir, sin tomar en cuenta a la realidad. Una probabilidad *a priori* considera juegos como los dados, las cartas, las monedas o cualquier cosa similar, como sistemas ideales que no se desgastan.

Los sistemas ideales no muestran las características de desgaste reales porque entonces no tendrían la característica de poder reproducirse con precisión. Es decir, un dado real puede exhibir una desviación hacia algunos números en particular, cuando después de muchos tiros, un lado se gasta. Además, dependiendo de la fabricación, un dado real puede mostrar preferencia hacia números mayores debido a que tienen más puntos (pequeños huecos coloreados) perforados. Esta desviación se determinó al analizar los datos de más de un millón de tiros de un dado real, se utilizó un dado nuevo cada 20 mil tiros para evitar la desviación causada por el desgaste disparejo de las caras. En la tabla 4.2 se muestra la fracción de veces que caen diferentes números en cada tiro (Shafer 76).

Número	1	2	3	4	5	6
Fracción	.155	.159	.164	.169	.174	.179

Tabla 4.2 Resultados de un millón de tiros de un dado

En un sistema ideal, todos los números salen equitativamente, lo que facilita mucho el análisis.

La fórmula fundamental de la probabilidad clásica se define como la probabilidad

$$P = \frac{W}{N}$$

donde W es el número de triunfos y N es el número de **eventos** igualmente posibles, que son los posibles resultados de un experimento o **prueba**. Por ejemplo, si lanza una vez un dado, hay seis posibles eventos de esta prueba única, pues el dado puede caer con la parte superior mostrando 1, 2, 3, 4, 5 o 6. Conforme a la probabilidad clásica suponemos que cualquiera de estos seis eventos es igualmente posible y así la probabilidad de tirar 1, $P(1)$, es

$$P(1) = \frac{1}{6}$$

De igual manera,

$$P(2) = \frac{1}{6}$$

etcétera.

La probabilidad de perder, Q, es

$$Q = \frac{N - W}{N} = 1 - P$$

La fórmula fundamental para P es una definición *a priori* porque la probabilidad se calcula *antes* de que se juegue. El término *a priori* significa "lo que antecede" o "antes del evento" cuando se analizan probabilidades. En una probabilidad *a priori* se supone que se conocen todos los eventos posibles y que hay una probabilidad igual de que suceda cada evento.

Por ejemplo, al tirar un dado, los tiros posibles son 1, 2, 3, 4, 5 y 6; si además, si el dado es **limpio** (no está cargado), entonces cada cara tiene igual probabilidad de salir. De igual manera, en un mazo limpio de 52 cartas, cada una tiene la misma probabilidad de salir si se barajan limpiamente. La probabilidad de que salga cualquiera de las caras del dado es de 1/6 y para cada carta es de 1/52.

Cuando una prueba se repite varias veces y arroja exactamente el mismo resultado, el sistema es **determinístico**; en caso contrario, es **no determinístico** o aleatorio. En algunas clases de sistemas expertos, se utilizan las shells no determinísticas para evitar desviación en el disparo de las reglas. Cuando se satisfacen los patrones de varias reglas y no hay preferencias explícitas sobre cuál de ellas debe ejecutarse, el mecanismo de inferencia elige arbitrariamente la que habrá de ejecutarse; esto evita una desviación en la cual se dispara una regla (por ejemplo, la primera que se introduce al sistema).

Espacios de la muestra

El resultado de una prueba es un **punto de muestra** y el conjunto de todos los puntos de muestra posibles define el **espacio de la muestra**. Por ejemplo, si se arroja un solo dado, puede ocurrir cualquiera de los puntos de muestra 1, 2, 3, 4, 5 o 6. En la figura 4.2(a) se ilustra el espacio de la muestra para el tiro, que es el conjunto { 1, 2, 3, 4, 5, 6 }.

Un **evento** es un subconjunto del espacio de la muestra. Por ejemplo, el evento { 1 } ocurre si el dado cae con la cara 1 en la parte superior. Un **evento simple** sólo tiene un elemento y uno **compuesto** tiene más de uno, como se muestra en la figura 4.2(b).

(a) Espacio de la muestra para un solo dado (b) Un evento simple (5) y uno compuesto (3,6)

Figura 4.2 Espacio de la muestra y eventos

Una manera gráfica de determinar el espacio de la muestra consiste en construir un **árbol de eventos**. Por ejemplo, supongamos que hay dos computadoras que funcionan, F, o que no funcionan, N. En la figura 4.3 vemos el árbol de eventos y en la tabla 4.3 el espacio de la muestra en forma de tabla. Nótese que los eventos compuestos se presentan en el orden { computadora 1, computadora 2 }. El espacio de la muestra es { *FF, FN, NF, NN* }.

Este tipo de árbol de eventos es binario porque sólo incluye probabilidades binarias, es decir, la computadora funciona o no. Muchos otros casos tienen este tipo de árbol; por ejemplo, lanzar una moneda lleva a un evento binario, porque sólo hay dos resultados posibles para cada lanzamiento.

La probabilidad aprovecha de manera extensa las teorías de la estadística, y viceversa. La estadística se relaciona con la recolección y el análisis de datos acerca de las **poblaciones**, conjunto del que se extraen las muestras; uno de sus objetivos es hacer inferencias acerca de la población padre de la que se extrajo la población muestra. Un ejemplo típico de la inferencia estadística consiste en determinar qué proporción de los votantes registrados favorece a cierto candidato, con base en una encuesta telefónica.

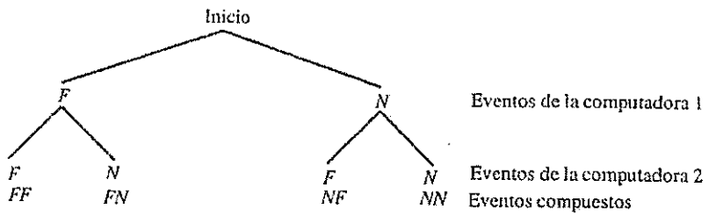


Figura 4.3 Árbol de eventos para un evento compuesto

		Computadora 2	
		F	*N*
Computadora 1	*F*	*FF*	*FN*
	N	*NF*	*NN*

Tabla 4.3 Espacio de la muestra para los binarios

Una aplicación de la teoría de la probabilidad consiste en deducir si la muestra es realmente representativa de todos los votantes, o si puede desviarse a favor de algún partido. Aunque este ejemplo se relaciona con objetos reales como las personas, otras poblaciones son hipotéticas, como el número posible de lanzamientos de monedas, porque cada lanzamientos es una muestra de la población hipotética de todos los lanzamientos de monedas.

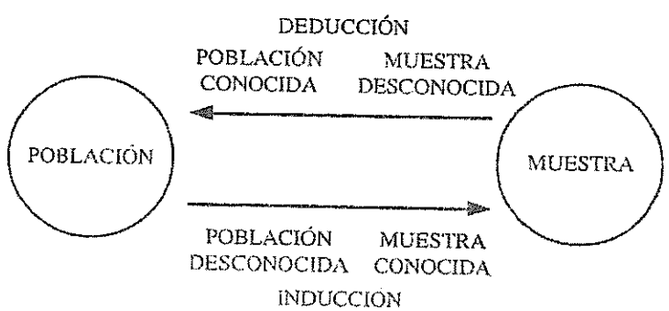


Figura 4.4 Razonamiento deductivo e inductivo acerca de poblaciones y muestras

Como se muestra en la figura 4.4., la deducción y la inducción son la base del razonamiento acerca de las poblaciones. Dada una población conocida, la deducción nos permite hacer inferencias sobre la muestra desconocida; de manera correspondiente, dada una muestra conocida, la inducción nos permite hacer inferencias sobre la población desconocida.

Teoría de la probabilidad

Es posible crear una teoría formal de la probabilidad mediante tres axiomas:

$$\textit{axioma 1:}\ 0 \le P(E) \le 1$$

Este axioma define el rango de probabilidad utilizando los números reales 0 y 1, no se permiten probabilidades negativas. A un **evento seguro** se le asigna la probabilidad 1 y a un **evento imposible** se le asigna la probabilidad 0.

$$\textit{axioma 2:}\ \sum_i P(E_i) = 1$$

Este axioma establece que la suma de todos los eventos que no se afectan entre sí, denominados ***eventos mutuamente excluyentes***, es 1. Estos eventos no tienen puntos de muestra en común. Por ejemplo, una computadora no puede funcionar correctamente y, al mismo tiempo, no hacerlo.

Como corolario de este axioma

$$P(E) + P(E') = 1$$

donde E' es el complemento del evento E. Este corolario significa que la probabilidad de que un evento ocurra más la probabilidad de que no ocurra es igual a 1. Es decir, la ocurrencia y la no ocurrencia de un evento es mutuamente excluyente y completan el espacio de la muestra.

$$\textit{axioma 3:}\ P(E_1 \cup E_2) = P(E_1) + P(E_2)$$

donde E_1 y E_2 son eventos mutuamente excluyentes. Este axioma significa que si E_1 y E_2 no pueden ocurrir simultáneamente (eventos mutuamente excluyentes), entonces la probabilidad de que ocurra uno o el otro es la suma de sus probabilidades.

A partir de estos axiomas, pueden deducirse teoremas relacionados con el cálculo de probabilidades en otras circunstancias, como los eventos mutuamente excluyentes, que se analizan más adelante. Aunque estos axiomas integran la base de la teoría de probabilidad, observe que *no especifican las probabilidades básicas de* P(E), porque éstas se determinan con otros métodos, como *a priori*.

Estos axiomas colocan a la probabilidad sobre una base teórica sólida, de hecho, a la teoría axiomática también se le llama ***teoría objetiva de la probabilidad***. Estos axiomas en particular fueron ideados por Kolmogorov. Renyi creó una teoría equivalente que usa axiomas de probabilidad condicional (Lindley 65).

4.6 PROBABILIDADES EXPERIMENTALES Y SUBJETIVAS

Aparte de los juegos ideales con iguales posibilidades, la probabilidad clásica no puede responder a preguntas como cuál es la probabilidad de que su unidad de disco se descomponga mañana o cuál es la expectativas de vida para un hombre de 25 años.

A diferencia del método *a priori*, la **probabilidad experimental** define la probabilidad de un evento, P(E), como el límite de una distribución de frecuencia:

$$P(E) = \lim_{N \to \infty} \frac{f(E)}{N}$$

donde f(E) es la frecuencia de los resultados de un evento para N resultados totales observados. A este tipo de probabilidad también se le llama ***probabilidad a posteriori***, que significa "después del evento"; también se conoce como ***probabilidad posterior***. La idea de ésta es medir la frecuencia con que ocurre un evento, durante un gran número de pruebas y, a partir de ésta, inducir la probabilidad experimental.

Por ejemplo, para determinar la probabilidad experimental de que una unidad de disco se descomponga, se podría aplicar una encuesta entre otras personas con el mismo tipo de unidad. En la tabla 4.4 se muestran los resultados de esta hipotética encuesta.

Porcentaje total de descomposturas	Horas de uso
10	100
25	250
50	500
75	750
99	1000

Tabla 4.4 Tiempo hipotético para que un unidad de disco se descomponga

Si la unidad se ha utilizado durante 750 horas, entonces puede inducirse que hay un 75 por ciento de probabilidades de que se descomponga mañana, (Observe que esta cifra de 75 por ciento es una inducción, más que una deducción). A diferencia de un juego ideal, esa unidad de disco no es exactamente igual a las demás, puede haber diferencias en los materiales usados, el control de calidad, las condiciones ambientales y el uso; todas ellas afectan al disco y su longevidad. Es mucho más sencillo exponer estrategias simples como un dado o unas cartas, con tolerancias mucho mayores, que una pieza compleja de equipo como una unidad de disco.

Otro tipo de probabilidad experimental puede inducirse a partir de las tablas de mortalidad utilizadas por las compañías de seguros, que muestran la probabilidad de que una persona muera en función de su edad y sexo. El cálculo de las probabilidades de que alguien muera con base en estas tablas es una inducción, porque cada persona es única. Para los seguros habitacionales se presenta una situación análoga: A menos que una casa se incendie para que pueda calcularse la probabilidad experimental individual, las probabilidades experimentales utilizadas se basarán en casas similares.

También existe una clase de probabilidad llamada **probabilidad subjetiva**; suponga que se pretende calcular la probabilidad de que los automóviles con motor eléctrico superconductor cuesten 10 mil dólares en el año 2020. Como no hay datos del precio de estos automóviles, no habrá manera de extrapolarlos para ver si 10 mil dólares es algo razonable.

La probabilidad subjetiva trata con eventos que no son reproducibles y no tienen base histórica que sirva para la extrapolación, tales como perforar un pozo petrolero en un nuevo sitio. Sin embargo, una probabilidad subjetiva determinada por un perito es mejor que no estimar nada y suele ser muy exacta (o el especialista dejaría de ser serlo) (Sioshansi 1983).

Una probabilidad subjetiva en realidad es una creencia u opinión expresada como una probabilidad, más que una probabilidad basada en axiomas o mediciones empíricas. Las creencias y opiniones de un especialista desempeñan importantes papeles en los sistemas expertos, como se estudia en este capítulo. En la tabla 4.5 se resumen los diferentes tipos de probabilidades.

Nombres	Fórmula	Características
a priori (clásica, teórica, matemática, simétrica, equiprobable, de igualdad)	$P(E) = \frac{W}{N}$ donde W es el número de salidas del evento E para un total de N posibles	Eventos repetibles, resultados con probable igualdad, forma matemática exacta conocida, no basada en experimentos, conocidos todos los eventos y salidas posibles
a posteriori (experimental, empírica, científica, de frecuencia relativa, estadística) $P(E) \approx \frac{f(E)}{N}$	$P(E) = \lim_{N \to \infty} \frac{f(E)}{N}$ donde f(E) es la frecuencia, f, de que el evento, E, se observe para N salidas totales	Eventos reproducibles basados en experimentos, aproximado mediante un número finito de experimentos, forma matemática exacta desconocida
subjetiva (personal)	Vea la sección 4.12	Eventos no repetibles, forma matemática exacta desconocida, método de frecuencia relativa imposible, basado en la opinión, experiencia, juicio y creencias del especialista o perito

Tabla 4.5 Tipos de probabilidades

4.7 PROBABILIDADES COMPUESTAS

Las probabilidades de los eventos compuestos pueden calcularse a partir de sus espacios de la muestra. En un ejemplo simple, consideramos la probabilidad de lanzar un dado de manera tal que el resultado sea un número par y divisible entre tres. Esto puede expresarse con un diagrama de Venn para los conjuntos

$$A = \{ 2, 4, 6 \} \qquad B = \{ 3, 6 \}$$

en el espacio de la muestra del dado, como se aprecia en la figura 4.5.

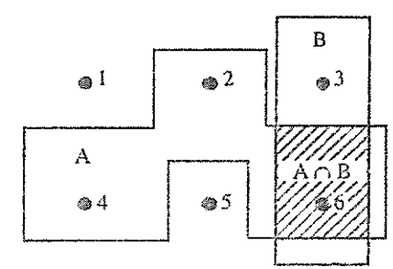


Figura 4.5 Probabilidad compuesta de tirar un solo dado para obtener un número par y divisible entre tres

Observe que la intersección de los conjuntos A y B es

$$A \cap B = \{ 6 \}$$

La probabilidad compuesta de tirar un número par y divisible entre tres es de:

$$P(A \cap B) = \frac{n(A \cap B)}{n(S)} = \frac{1}{6}$$

donde n es el número de elementos en los conjuntos y S es el espacio de la muestra.

A los eventos que no se afectan entre sí de ninguna manera se les denomina ***eventos independientes***. Para dos eventos independientes A y B, la probabilidad es simplemente el producto de las probabilidades individuales. Se dice que los eventos A y B son ***pares independientes***.

$$P(A \cap B) = P(A)\ P(B)$$

A dos eventos se les llama ***eventos estocásticamente independientes*** si (y sólo si) la fórmula previa es verdadera. El término ***estocástico*** viene del griego y significa "suposición"; suele utilizarse como sinónimo de *probabilidad, azar* u *oportunidad*. De este modo, un experimento fortuito tiene un resultado aleatorio, en contraste con el resultado no aleatorio de un experimento determinístico.

Para tres eventos, se podría suponer que la independencia es

$$P(A \cap B \cap C) = P(A)\ P(B)\ P(C)$$

Desafortunadamente, la vida y la probabilidad no son tan simples. La fórmula para la **independencia mutua** de N eventos requiere que se satisfagan 2^N ecuaciones, requisito que se resume en la siguiente ecuación, donde los asteriscos significan que deben satisfacerse todas las combinaciones de un evento y su complemento.

$$P(A^*_1 \cap A^*_2 \ldots \cap A^*_n) = P(A^*_1)\ P(A^*_2)\ P(A^*_n)$$

Para tres eventos, la anterior ecuación para la independencia mutua requiere que se satisfagan *todas* las siguientes ecuaciones:

$$\begin{aligned}
P(A \cap B \cap C) &= P(A)\ P(B)\ P(C) \\
P(A \cap B \cap C') &= P(A)\ P(B)\ P(C') \\
P(A \cap B' \cap C) &= P(A)\ P(B')\ P(C) \\
P(A \cap B' \cap C') &= P(A)\ P(B')\ P(C') \\
P(A' \cap B \cap C) &= P(A')\ P(B)\ P(C) \\
P(A' \cap B \cap C') &= P(A')\ P(B)\ P(C') \\
P(A' \cap B' \cap C) &= P(A')\ P(B')\ P(C) \\
P(A' \cap B' \cap C') &= P(A')\ P(B')\ P(C')
\end{aligned}$$

La independencia de un solo par de cada dos eventos no basta para garantizar la independencia mutua, como se muestra en el problema 4.6.

Para nuestro dado, los eventos de un número par y divisible entre tres se afectan definitivamente entre sí y, de este modo, *no* es un experimento estocástico. La probabilidad de que caiga un número par en un dado y un número divisible entre tres en el otro *es* estocástica.

$$P(A \cap B) = P(A)\ P(B) = \frac{3}{6} \cdot \frac{2}{6} = \frac{1}{6}$$

Ahora consideremos el caso de la unión de eventos, $P(A \cup U)$. Definida n como una función que designa el número de elementos de un conjunto. Si se agrega el número de elementos de A a los de B y los dividimos entre todos los elementos del espacio de la muestra n(S);

$$(1)\quad P(A \cup B) = \frac{n(A) + n(B)}{n(S)} = P(A) + P(B)$$

entonces el resultado será demasiado grande si los conjuntos se superponen. Si observa la figura 4.5, al aplicar esta fórmula se obtiene:

$$P(A \cup B) = \frac{3 + 2}{6} = \frac{5}{6}$$

pero puede verse fácilmente que sólo hay cuatro elementos en la unión porque:

$$A \cup B = \{ 2, 3, 4, 6 \}$$

El problema es que al sólo agregar n(A) a n(B), se está contando dos veces la intersección { 6 } porque ésta pertenece tanto a A como a B.

La fórmula correcta simplemente requiere la resta de la probabilidad extra del conjunto intersección:

$$(2)\quad P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

que para nuestro dado se reduce a:

$$P(A \cup B) = \frac{3}{6} + \frac{2}{6} - \frac{1}{6} = \frac{4}{6} = \frac{2}{3}$$

La fórmula (1) permanece igual cuando los conjuntos se separan y, por tanto, no tienen elementos en común. Es un caso especial de la fórmula (2), a la que se le llamó ***Ley aditiva***; ésta puede derivarse también como un teorema utilizando los tres axiomas de probabilidad antes analizados. La Ley aditiva para los tres eventos es:

$$\begin{aligned}
P(A \cup B \cup C) = {} & P(A) + P(B) + P(C) \\
& - P(A \cap B) - P(A \cap C) - P(B \cap C) \\
& + P(A \cap B \cap C)
\end{aligned}$$

También pueden derivarse de los axiomas otras leyes para ni A ni B y el O-excluyente de A y B. El valor de estas leyes para probabilidades compuestas es que (1) no tienen que hacerse experimentos para cada combinación de probabilidades y (2) no tienen que contarse los elementos individuales de los espacios de muestra grandes.

4.8 PROBABILIDADES CONDICIONALES

Los eventos que no son mutuamente excluyentes se influyen entre sí. Saber que un evento ha ocurrido puede llevarnos a revisar la probabilidad de que otro evento ocurrirá.

Ley multiplicativa

A la probabilidad de un evento A, dado que ocurrió el evento B, se le llama **probabilidad condicional,** se indica con P(A | B) y se define como

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)} \quad \text{para} \quad P(B) \neq 0$$

La probabilidad P(B) es la probabilidad *a priori* o previa antes de que se conozca cualquier información, cuando se usa en asociación con una probabilidad condicional, se le llama ***probabilidad no condicional*** o ***probabilidad absoluta.***

Esta definición de la probabilidad condicional puede explicarse intuitivamente si observa el ejemplo de la figura 4.6, que es un espacio de la muestra para ocho eventos. A partir esta figura, pueden calcularse las probabilidades como la relación entre el número de eventos, n(A) o n(B), y el número total en el espacio de la muestra n(M). Es decir:

$$P(A) = \frac{n(A)}{n(M)} = \frac{4}{8}$$

$$P(B) = \frac{n(B)}{n(M)} = \frac{6}{8}$$

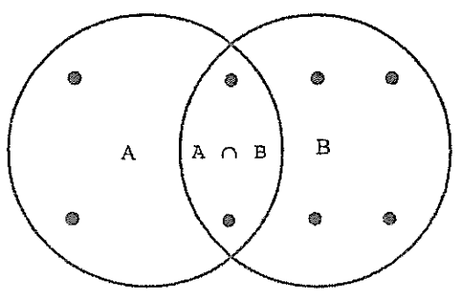


Figura 4.6 Espacio de la muestra para dos eventos que se intersectan

Si sabemos que ha ocurrido el evento B, el espacio de la muestra es sólo el de B:

$$N(M) = 6$$

Como B ocurrió, sólo se toman en consideración los eventos de A que están asociados con B:

$$P(A \mid B) = \frac{n(A \cap B)}{n(B)} = \frac{2}{6}$$

Para expresar el resultado en términos de probabilidades, sólo se dividen el numerador y el denominador de lo anterior entre n(M):

$$P(A \mid B) = \frac{\dfrac{n(A \cap B)}{n(M)}}{\dfrac{n(B)}{n(M)}}$$

$$= \frac{P(A \cap B)}{P(B)} \quad \text{para } P(B) \neq 0$$

Entonces, la **Ley multiplicativa** de la probabilidad para dos eventos se define como:

$$P(A \cap B) = P(A \mid B)\ P(B)$$

que es equivalente a:

$$P(A \cap B) = P(B \mid A)\ P(A)$$

La Ley multiplicativa para tres eventos es la siguiente:

$$P(A \cap B \cap C) = P(A \mid B \cap C)\ P(B \mid C)\ P(C)$$

y la **Ley multiplicativa generalizada** es:

$$\begin{aligned} P(A_1 \cap A_2 \cap \ldots \cap A_N) = {} & P(A_1 \mid A_2 \cap \ldots \cap A_N) \\ & P(A_2 \mid A_3 \cap \ldots \cap A_N) \\ & \ldots (A_{N-1} \mid A_N)\ P(A_N) \end{aligned}$$

A manera de ejemplo, en la tabla 4.6 se muestran las probabilidades hipotéticas de que un disco se descomponga en el transcurso de un año, utilizando una unidad de marca X.

	Marca X	Marca diferente de X	Total de filas
Se descompone C	0.6	0.1	0.7
No se descompone C´	0.2	0.1	0.3
Total de columnas	0.8	0.2	1.0

Tabla 4.6 Probabilidades hipotéticas de que un disco se descomponga en el transcurso de un año

A las probabilidades de 0.6, 0.1, 0.2 y 0.1 que se encuentran en medio de la tabla se les llama ***probabilidades interiores*** y representan la intersección de eventos. Las sumas de las filas y columnas se despliegan como *totales* y se les llama ***probabilidades marginales,*** porque quedan en los márgenes de la tabla.

En las tablas 4.7 y 4.8 vemos el conjunto y el significado de las probabilidades con más detalle. En la figura 4.7 observamos el diagrama de Venn de la intersección del espacio de la muestra. En la tabla 4.8 puede verse que la suma de los totales de las probabilidades de filas y columnas es 1.

	X	X′	Total de filas
C	$C \cap X$	$C \cap X'$	$C = (C \cap X) \cup (C \cap X')$
C′	$C' \cap X$	$C' \cap X'$	$C' = (C' \cap X) \cup (C' \cap X')$
Total de columnas	$X = (C' \cap X) \cup (C \cap X)$	$X' = (C' \cap X') \cup (C \cap X')$	M (Espacio de la muestra)

Tabla 4.7 Interpretación del conjunto

	X	X′	Total de filas
C	$P(C \cap X)$	$P(C \cap X')$	$P(C)$
C′	$P(C' \cap X)$	$P(C' \cap X')$	$P(C')$
Total de columnas	$P(X)$	$P(X')$	1.0

Tabla 4.8 Interpretación de la probabilidad de dos conjuntos

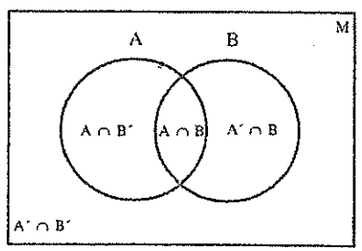


Figura 4.7 Interpretación del espacio de la muestra de dos conjuntos

Utilizando la tabla 4.8, pueden calcularse las probabilidades de todos los conjuntos. Algunas de ellas son:

(1) La probabilidad de descompostura para unidades de disco marca X y marca diferente de X (el espacio de la muestra) es:

$$P(C) = 0.7$$

(2) La probabilidad de no descompostura para el espacio de la muestra es

$$P(C') = 0.3$$

(3) La probabilidad de usar la marca X es

$$P(X) = 0.8$$

(4) La probabilidad de no usar la marca X es

$$P(X') = 0.2$$

(5) La probabilidad de una descompostura mientras se usa la marca X es

$$P(C \cap X) = 0.6$$

(6) La probabilidad de una descompostura, dado que se utilizó la marca X, es

$$P(C \mid X) = \frac{P(C \cap X)}{P(X)} = \frac{0.6}{0.8} = 0.75$$

(7) La probabilidad de una descompostura, dado que no se usó la marca X, es

$$P(C \mid X') = \frac{P(C \cap X')}{P(X')} = \frac{0.1}{0.2} = 0.50$$

Tal vez parezca que las probabilidades (5) y (6) tienen significados similares cuando se leen sus descripciones. Sin embargo, (5) es simplemente la intersección de dos eventos, mientras que (6) es una probabilidad condicional. El significado de la intersección (5) es el siguiente:

> Si un unidad de disco se elige al azar, entonces 0.6 de las veces será de la marca X y se descompondrá.

En otras palabras, simplemente se están eligiendo muestras de la población de discos, algunos de ellos son marca X y se han descompuesto (0.6), otros no lo son y se han descompuesto (0.1), algunos son de la marca X y no se han descompuesto (0.2) y algunos no son de la marca X y no se han descompuesto (0.1).

Por el contrario, el significado de la probabilidad condicional (6) es muy diferente:

> Si se elige un disco de la marca X, entonces 0.75 de las veces se habrá descompuesto.

Observe que en la probabilidad condicional se están eligiendo aquellos elementos (marca X) que resultan de interés y se les considera como el nuevo espacio de la muestra.

Si cualquiera de las siguientes ecuaciones es verdad, entonces los eventos A y B son **independientes**.

$$P(A \mid B) = P(A) \text{ o}$$
$$P(B \mid A) = P(B) \text{ o}$$
$$P(A \cap B) = P(A)\ P(B)$$

Si cualquiera de estas ecuaciones es verdadera, entonces las demás también lo son.

Teorema de Bayes

La probabilidad condicional, $P(A \mid B)$, establece la probabilidad de un evento A dado que el evento B ocurrió. El problema contrario consiste en encontrar la **probabilidad inversa**, que establece la probabilidad de un evento anterior dado que ha ocurrido uno posterior. Este tipo de probabilidad sucede a menudo, como en el caso del diagnóstico médico o de equipo, donde los síntomas aparecen y el problema consiste en encontrar la causa más probable. La solución a este problema es el **teorema de Bayes**, también llamado fórmula de Bayes, regla de Bayes o ley de Bayes, en honor del clérigo y matemático del siglo XVIII, Thomas Bayes.

Como ejemplo del teorema de Bayes, veamos cómo se aplica a la descompostura de la unidad de disco. A partir de la probabilidad condicional (6), hay 75 por ciento de probabilidades de que una unidad de la marca X se descomponga en el transcurso de un año; mientras que, con base en (7), la probabilidad de que lo haga un disco con marca diferente es 50 por

ciento. La pregunta inversa es: suponiendo que se tiene una unidad y no conoce su marca, ¿cuál es la probabilidad de que, si se descompone, sea de la marca X? ¿Y que no lo sea?

Situaciones como ésta, en las que no se conoce la unidad, ocurren todo el tiempo debido a que en raras ocasiones los fabricantes de computadoras hacen sus propios discos; muchos compran unidades de un fabricante de equipo original y las reempaquetan para venderlas bajo su propia marca. Dependiendo del fabricante que ofrezca el menor costo, la unidad variará año tras año, mientras que sólo permanece igual la marca del fabricante de computadoras.

Dado que una unidad se descompuso, la probabilidad de que sea de la marca X puede determinarse utilizando la probabilidad condicional y los resultados (1), (5).

$$P(X \mid C) = \frac{P(C \cap X)}{P(C)} = \frac{0.6}{0.7} = \frac{6}{7}$$

Como alternativa, utilizando la Ley multiplicativa en el numerador y (1), (3), (6)

$$P(X \mid C) = \frac{P(C \mid X)\ P(X)}{P(C)} = \frac{(0.75)\ (0.8)}{0.7} = \frac{0.6}{0.7} = \frac{6}{7}$$

La probabilidad de P(X | C) es la probabilidad **inversa** o *a posteriori*, la cual establece que si la unidad se descompuso, era de la marca X. En la figura 4.8 se muestra un árbol de decisión para las descomposturas de disco. El rectángulo indica un **acto** o decisión, y el círculo indica un **evento** u ocurrencia. Las probabilidades previas son las determinadas antes de que se realicen los experimentos para medir las descomposturas, mientras que las probabilidades posteriores o inversas son aquellas determinadas después de que se completan los experimentos y nos permiten revisar las probabilidades previas para obtener resultados más exactos.

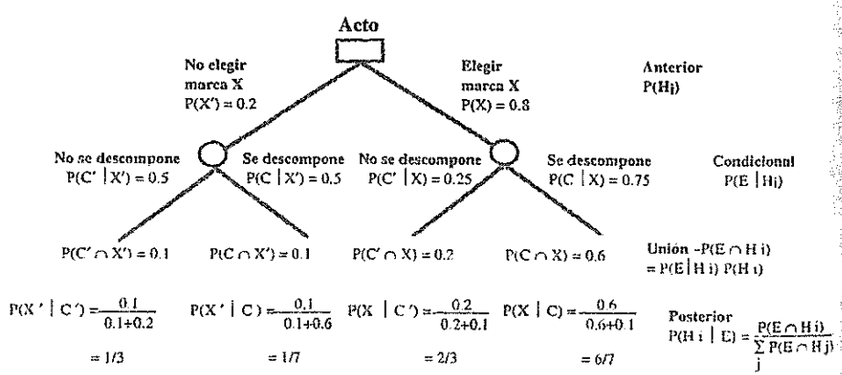


Figura 4.8 Árbol de decisión para las descomposturas de disco

La forma general del teorema de Bayes puede escribirse mediante eventos, E, e hipótesis (suposiciones), H, en las siguientes formas alternas:

$$P(Hi \mid E) = \frac{P(E \cap Hi)}{\sum_j P(E \cap Hj)}$$

$$= \frac{P(E \mid Hi)\ P(Hi)}{\sum_j P(E \mid Hj)\ P(Hj)}$$

$$= \frac{P(E \mid Hi)\ P(Hi)}{P(E)}$$

4.9 RAZONAMIENTO HIPOTÉTICO E INDUCCIÓN HACIA ATRÁS

El teorema de Bayes suele usarse para los análisis de árbol de decisión en los negocios y las ciencias sociales. El sistema PROSPECTOR también utiliza el método para **toma de decisiones bayesiana** al determinar sitios favorables para la exploración mineral. PROSPECTOR ha obtenido gran fama como el primer sistema experto en descubrir un valioso depósito de molibdeno con valor de 100 millones de dólares.

Para ilustrar de la toma de decisiones bayesiana en condiciones de incertidumbre, consideremos el problema de la exploración petrolera. Para empezar, el explorador debe decidir cuáles son las oportunidades de encontrar petróleo; si no hay evidencias a favor ni en contra, puede asignar probabilidades previas subjetivas de encontrar petróleo, O, de:

$$P(O) = P(O') = 0.5$$

Sin evidencia, se dice que la asignación de probabilidades con iguales valores relativos entre los resultados posibles se hizo **a la desesperada**. Este término no significa que el explorador esté (necesariamente) desesperado, solamente es un vocable técnico para la asignación previa de probabilidades sin desviaciones. El explorador cree que hay una oportunidad mayor que 50-50 de encontrar petróleo, y supone lo siguiente:

$$P(O) = 0.6 \quad P(O') = 0.4$$

Una herramienta muy importante para la exploración petrolera y mineral es la **exploración sísmica**. En esta técnica, se crean pulsos de sonido que viajan a través de la Tierra. Las ondas sonoras se detectan por medio de micrófonos colocados en varios sitios. Observando el tiempo de llegada de los pulsos y la distorsión de la onda de sonido, es posible determinar las estructuras geológicas y la posibilidad de que haya petróleo o minerales. Desgraciadamente las pruebas sísmicas no son 100 por ciento exactas; algunos tipos de estructuras geológicas pueden afectar las ondas de sonido, causando que la prueba reporte la presencia de petróleo cuando en realidad no lo hay (falso positivo). De igual manera, la prueba puede reportar la ausencia de petróleo cuando en realidad sí lo hay (falso negativo). Supongamos que el historial de la prueba sísmica ha dado las siguientes probabilidades condicionales, en las que + es un resultado positivo y – es un resultado negativo. Conviene recordar que se trata de probabilidades condicionales, porque la causa (petróleo o no) debe haber ocurrido antes del efecto (resultado de la prueba). Una probabilidad posterior regresaría desde el efecto (resultado de la prueba) hacia la causa (petróleo o no). Por lo general, una probabilidad condicional va hacia adelante en el tiempo mientras que una probabilidad posterior va hacia atrás.

```
P(+ | O)  = 0.8            P(- | O)  = 0.2 (falso -)
P(+ | O') = 0.1 (falso +)  P(- | O') = 0.9
```

Utilizando las probabilidades previa y condicional, puede construirse el árbol de probabilidad inicial, como se muestra en la figura 4.9. También muestra las probabilidades de unión, que se calculan a partir de las probabilidades previa y posterior.

A continuación se utiliza la Ley de adición para calcular la probabilidad total de una prueba de a+ y a–.

```
P(+) = P(+ ∩ O) + P(+ ∩ O') = 0.48 + 0.04 = 0.52
P(-) = P(- ∩ O) + P(- ∩ O') = 0.12 + 0.36 = 0.48
```

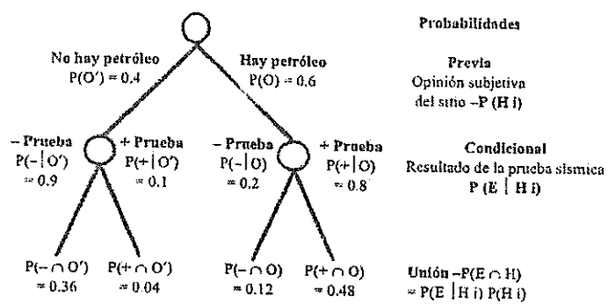


Figura 4.9 Árbol de probabilidad inicial para la exploración petrolera

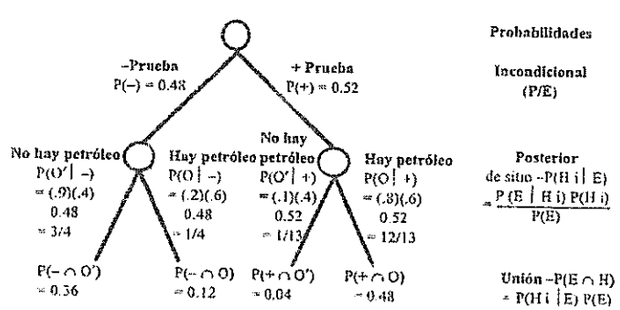


Figura 4.10 Árbol de probabilidad para la exploración petrolera revisado

P(+) y P(–) son probabilidades incondicionales que ahora pueden utilizarse para calcular las probabilidades posteriores del sitio, como se muestra en la figura 4.10. Por ejemplo, P(O' | –) es la probabilidad posterior de no encontrar petróleo en el sitio con base en la prueba negativa, luego se calculan las probabilidades de unión. Observe que las probabilidades de unión de la figura 4.10 son las mismas que en la figura 4.9. Es necesario revisar las probabilidades para obtener buenos resultados cuando se obtiene información experimental, como los resultados de la prueba sísmica, después de que se ha estimado (o supuesto) la probabilidad inicial.

En la figura 4.11 se muestra el árbol de decisión bayesiano inicial, utilizando los datos de la figura 4.10. Las **utilidades** que se muestran en la parte inferior son positivas si se gana dinero y negativas si se pierde. Las cantidades supuestas se muestran en la tabla 4.9.

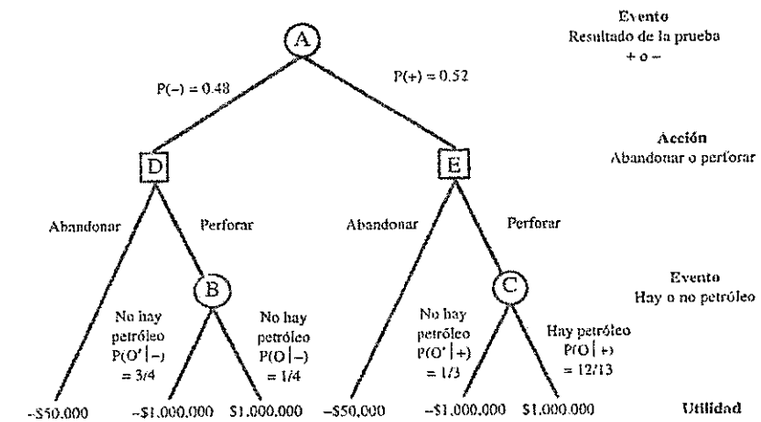


Figura 4.11 Árbol de decisión bayesiano inicial para la exploración petrolera

Contrato petrolero, si se tiene éxito	$1,250,000
Gasto de perforación	–$200,000
Exploración sísmica	–$50,000

Tabla 4.9 Tabla de rentabilidad para el problema de exploración petrolera

De este modo, si se encuentra petróleo las utilidades son de $1,250,000 – $200,000 – $50,000 = $1,000,000; mientras que la decisión de abandonar después del resultado de la prueba sísmica produce una utilidad de –$50,000.

Para que el explorador tome la mejor decisión, las **utilidades esperadas** deben calcularse en el nodo de evento A. Y son la cantidad que el explorador puede obtener si sigue el mejor curso de acción. Para calcularlas al principio, A, debe trabajarse hacia atrás a partir de las hojas. En la teoría de probabilidad, a este proceso se le llama **inducción hacia atrás**. Es decir, para lograr las utilidades esperadas o el objetivo que deseamos, debemos razonar hacia atrás para encontrar las causas que nos llevarán a la meta.

Las utilidades esperadas a partir del nodo de evento son la suma de las utilidades multiplicada por las probabilidades de obtenerlas.

```
Utilidad esperada en el nodo C
$846,153 = ($1,000,000) (12/13) - ($1,000,000) (1/13)

Utilidad esperada en el nodo B
-$500,000 = ($1,000,000) (1/4) - ($1,000,000) (3/4)
```

En el nodo de acción E debemos decidir entre las utilidades de –$50,000 si abandonamos o de $846,153 si perforamos. Como $846,153 es mayor que –$50,000, induciremos que ésta es la mejor acción y la escribimos en el nodo E. La ruta para salir se refinará o enverdecerá al dividirla con un símbolo "–" para indicar que esta ruta no se seguirá, como en la figura 4.12.

De igual manera, en la acción del nodo D, la mejor opción entre una utilidad de –$50,000 y una de –$500,000 es –$50,000 y así se escribe en D.

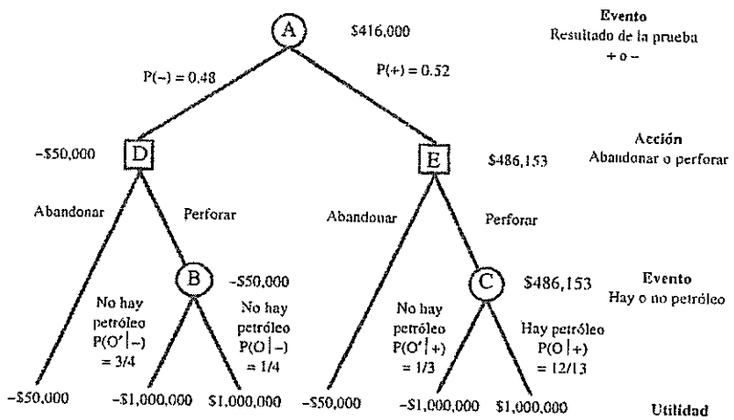


Figura 4.12
Árbol de decisión bayesiano completo para exploración petrolera, utilizando la inducción hacia atrás

Finalmente, la utilidad esperada al principio se calcula como sigue:

```
Utilidad esperada en el nodo A
$416,000 = ($846,153) (0.52) - ($50,000) (0.48)
```

y este valor se escribe en el nodo A. Ninguna ruta puede refinarse en el nodo A, porque ya se ha decidido hacer la prueba sísmica. Para decidir una prueba sísmica, el árbol de decisión tendría que extenderse hacia atrás, hasta antes del nodo A. Observe que en la inducción hacia atrás se está razonando hacia atrás en el tiempo para que las acciones en el futuro sean óptimas.

El árbol de decisión es un ejemplo de razonamiento hipotético o situaciones del tipo "¿Qué pasaría si?" Al explorar rutas de acción alternativas, se refinan las rutas que no conducen a las utilidades óptimas. Algunos tipos de herramientas de los sistemas expertos, como ART, de la Inference Corp., han elaborado mecanismos para el razonamiento hipotético y el envenenamiento.

El árbol de decisión de la figura 4.12 muestra la estrategia optima para el explorador. Si la prueba sísmica es positiva, el sitio debe perforarse; si es negativa, el sitio debe abandonarse. Aunque éste es un ejemplo muy simple de una toma de decisiones bayesiana, ilustra el tipo de razonamiento utilizado para manejar la incertidumbre; en casos más complejos, como decidir la aplicación de una prueba sísmica, el árbol de decisión puede volverse mucho más grande (Lapin 78).

4.10 RAZONAMIENTO TEMPORAL Y CADENAS DE MARKOV

Al razonamiento sobre eventos que dependen del tiempo se le llama ***razonamiento temporal*** y es algo que los seres humanos hacemos muy fácilmente. Sin embargo, resulta difícil formalizar **eventos temporales** para que una computadora pueda realizar las inferencias correspondien-

trol de tráfico aéreo, pueden ser muy útiles. En medicina se han desarrollado varios de estos sistemas, entre los que se incluyen VM (Fagan 1979) para administración de la ventilación a pacientes, con el fin de ayudarlos a respirar en pulmones artificiales; CASNET para tratamiento de glaucoma (Weiss 78) y el consejero de terapia digital para pacientes cardiacos (Gorry 78).

Con excepción del VM, los otros sistemas médicos mencionados tienen un problema menos difícil con el razonamiento temporal, que un sistema de control de una aeronave, pues éste debe operar en **tiempo real**. Casi ningún sistema experto opera en tiempo real debido al diseño del mecanismo de inferencia y a la gran cantidad de procesamiento necesario. Es muy difícil construir un sistema que realice una gran cantidad de razonamiento temporal para explorar múltiples hipótesis en tiempo real. Se han desarrollado diferentes lógicas temporales sobre diferentes axiomas (Turner 84). Las distintas teorías se basan en la manera de responder ciertas preguntas: ¿El tiempo tiene un primero y un último momento? ¿El tiempo es continuo o discontinuo? ¿Hay sólo un pasado pero muchos futuros posibles?

Dependiendo de las respuestas, pueden desarrollarse muchas lógicas diferentes (McDermott 82; Allen 81). La lógica temporal también es útil en los programas convencionales, como la síntesis y sincronización de procesos de los programas concurrentes (Manna y Wolper 84).

Otro enfoque para el razonamiento temporal es el de las probabilidades. Puede considerarse que un sistema pasa de una etapa a otra a medida que transcurre el tiempo, Y puede seleccionar cualquier cosa que se relacione con las probabilidades, como inventarios, votantes, clima, negocios, enfermedad, equipo, genética, etc. Al paso del sistema por una secuencia de estados se le llama ***proceso estocástico***, si es probabilístico.

Es conveniente representar un proceso estocástico con la forma de una matriz de transición. Para el simple caso de dos estados, S_1 y S_2, la matriz de transición es

$$\begin{array}{cc} & \text{Futuro} \\ & \begin{array}{cc} S_1 & S_2 \end{array} \\ \text{Presente} \begin{array}{c} S_1 \\ S_2 \end{array} & \begin{bmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{bmatrix} \end{array}$$

donde P_{mn} es la probabilidad de una transición del estado m al n.

Por ejemplo, supongamos que 10 por ciento de todas las personas que utilizan discos de la marca X comprarán otra unidad de esa misma marca cuando la necesiten. Además, 60 por ciento de las personas que no usan la marca X, ahora la comprarán cuando necesiten una nueva unidad (lo único bueno de la marca X es su publicidad). En un periodo, ¿cuántas personas utilizarán la marca X?

La matriz de transición, T, es

$$T = \begin{array}{c} \\ X \\ X' \end{array} \begin{array}{c} \begin{array}{cc} X & X' \end{array} \\ \begin{bmatrix} 0.1 & 0.9 \\ 0.6 & 0.4 \end{bmatrix} \end{array}$$

donde la suma de cada fila debe ser 1. A un vector cuyos componentes no son negativos y

probabilidad. Una manera de interpretar la matriz de transición es a partir de un diagrama de estado, como se muestra en la figura 4.13. Observe que hay una probabilidad de 0.1 de permanecer en el estado X, una probabilidad de 0.4 de permanecer en el estado X´, una probabilidad de 0.9 de pasar de X a X´ y una de 0.6 de pasar de X´ a X.

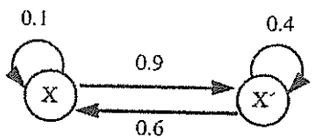


Figura 4.13 Interpretación del diagrama de estado de una matriz de transición

Supongamos que inicialmente 80 por ciento de las personas usa la marca X. En la figura 4.14 (a) se muestra un árbol de probabilidad para unas cuantas transiciones de estado, donde los estados se rotulan por número de estado y disco poseído. Observe cómo el árbol empieza a crecer. Si hubiera diez transiciones, el árbol tendría $2^{10} = 1024$ ramas. En la figura 4.14 (b) se muestra una manera alternativa de dibujar el árbol, como una retícula, La ventaja de esta representación es que no necesita demasiados vínculos para conectar los estados.

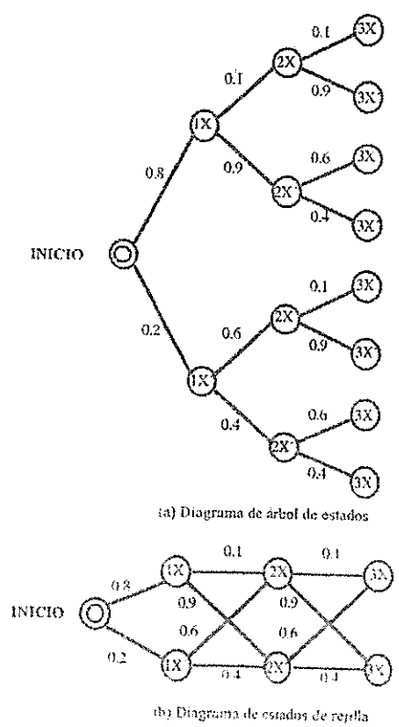


Figura 4.14 Diagramas de árbol y de rejilla para los estados que

La probabilidad de que el sistema se encuentre en cierto estado puede representarse como una matriz de fila llamada la ***matriz de estado***.

$$S = [\ P_1\ P_2\ \ldots\ P_N\ ]$$

donde $P_1 + P_2 + \ldots P_N = 1$

Inicialmente, 80 por ciento de las personas poseen la marca X, por lo que la matriz de estado es

$$S_1 = [\ 0.8\quad 0.2\ ]$$

A medida que pasa el tiempo, estos números cambiarán, dependiendo de cuál disco compren.

Con el fin de calcular el número de personas en el estado 2 que tienen la marca X y una la marca diferente, sólo se multiplica la matriz de estado por la matriz de transición, utilizando las leyes ordinarias para la multiplicación de matrices:

$$S_2 = S_1\ T$$

que produce:

$$S_2 = [\ 0.8\quad 0.2\ ] \begin{bmatrix} 0.1 & 0.9 \\ 0.6 & 0.4 \end{bmatrix}$$

$$= [\ (0.8)\ (0.1) + (0.2)\ (0.6)\quad (0.8)\ (0.9) + (0.2)\ (0.4)\ ]$$

$$= [\ 0.2\quad 0.8\ ]$$

La multiplicación de este segundo estado por la matriz de transición arroja:

$$S_3 = S_2\ T$$

$$= [\ 0.2\quad 0.8\ ] \begin{bmatrix} 0.1 & 0.9 \\ 0.6 & 0.4 \end{bmatrix}$$

$$S_3 = [\ 0.5\quad 0.5\ ]$$

La multiplicación de este tercer estado por la matriz de transición proporciona:

$$S_4 = S_3\ T$$

$$= [\ 0.5\quad 0.5\ ] \begin{bmatrix} 0.1 & 0.9 \\ 0.6 & 0.4 \end{bmatrix}$$

$$S_4 = [\ 0.35\quad 0.65\ ]$$

Los siguientes estados son

$$S = [\ 0.425\quad 0.575\ ]$$

S_6 = [0.3875 0.6125]

S_7 = [0.40625 0.59375]

S_8 = [0.396875 0.602125]

Observe que los estados están convergiendo en

[0.4 0.6]

a lo que se denomina ***matriz de estado constante***. Se dice que el sistema está en equilibrio cuando se encuentra en estado constante porque no cambia. Es interesante que los valores de estado constante no dependan del estado inicial. Si se ha usado cualquier vector de probabilidad inicial, los valores de estado constante serían los mismos.

Un vector de probabilidad S es una matriz de estado constante para la matriz de transición T si

(1) S = S T

Si T es una **matriz de transición regular**, que tiene alguna potencia con elementos positivos únicamente, entonces existe una sola S de estado constante (Kemeny 59). El hecho de que los elementos de la matriz de transición sean positivos, significa que en algún momento es posible estar en cualquier estado, sin importar cuál ha sido el estado inicial. Es decir, cada estado es potencialmente accesible.

Un **proceso de cadena de Markov** se define como si tuviera las siguientes características:

(1) Un número finito de estados posibles.
(2) El proceso puede estar en uno y sólo un estado a la vez.
(3) El proceso se mueve o **pasa** sucesivamente de un estado a otro con el tiempo.
(4) La probabilidad de un movimiento sólo depende del estado inmediatamente anterior.

Por ejemplo, dado un conjunto finito de estados { A, B, C ,D, E, F, G, H }, entonces si el siguiente estado al que pasa el proceso, después de H, es I, la probabilidad condicional es la siguiente.

$$P(I \mid H) = P(I \mid H \cap G \cap F \cap E \cap D \cap C \cap B \cap A)$$

Observe cómo el diagrama de rejilla de la figura 4.14 (b) parece una cadena.

El caso de la unidad de disco es un proceso de cadena de Markov y es posible encontrar la matriz de estado constante aplicando la ecuación (1). Suponiendo algún vector arbitrario S con componentes X y Y, y aplicando la ecuación (1) como sigue:

$$[\ X \quad Y\ ] \begin{bmatrix} 0.1 & 0.9 \\ 0.6 & 0.4 \end{bmatrix} = [\ X \quad Y\ ]$$

Multiplicando el lado izquierdo e igualando sus elementos con los elementos correspondientes

$$0.1\ X + 0.6\ Y = X$$
$$0.9\ X + 0.4\ Y = Y$$

que es un sistema de ecuaciones dependientes. La resolución de X en términos de Y resulta:

$$X = \frac{0.6}{0.9}\ Y = \frac{2}{3}\ Y$$

Para resolver completamente para X y Y, se aprovecha que la suma de las probabilidades es igual a 1. Es decir:

$$X + Y = 1$$

y así

$$X = 1 - y = \frac{2}{3}\ Y$$

y de esto se desprende

$$X = \frac{2}{5} \qquad y = \frac{3}{5}$$

así que la matriz de estado constante es

[0.4 0.6]

que es, por supuesto, con lo que los valores de prueba convergían.

4.11 PARTICULARIDADES DE CREENCIA

Hasta ahora nos hemos ocupado de las probabilidades como la medición de eventos repetibles en sistemas ideales. Sin embargo, los seres humanos somos expertos en calcular las probabilidades de muchos eventos no repetibles, como los diagnósticos médicos y la exploración mineral, porque cada paciente y sitio es único. Para crear sistemas expertos en áreas como éstas, debemos ampliar el alcance de los eventos para tratar con proposiciones, que son afirmaciones verdaderas o falsas. Por ejemplo, un evento podría ser:

"El paciente está cubierto con manchas rojas"

y la proposición es:

"El paciente tiene sarampión"

Dado que A es una proposición, la probabilidad condicional:

P(A | B)

no es necesariamente una probabilidad en el sentido clásico, si los eventos y proposiciones no pueden repetirse o tener una base matemática. En cambio P(A | B) puede interpretarse como el

Si $P(A \mid B) = 1$, entonces se considera que A es verdadero, pero si $P(A \mid B) = 0$, entonces se cree que A es falso, mientras que los otros valores, $0 < P(A \mid B) < 1$, significan que no estamos completamente seguros de que A sea verdadero o falso. En estadística, el término ***hipótesis*** se usa para algunas proposiciones cuya verdad o falsedad no se conoce con certeza sobre la base de alguna **evidencia**. En este caso a la probabilidad condicional se le llama entonces **posibilidad**, como en $P(H \mid E)$, que expresa la posibilidad de una hipótesis, H, con base en alguna evidencia, E.

Aunque $P(H \mid E)$ tiene la forma de una probabilidad condicional, en realidad significa algo diferente: la posibilidad o grado de certeza. *Probabilidad* se refiere a eventos repetibles y *posibilidad* al grado de factibilidad en eventos no repetibles. Como los sistemas expertos son modelos de especialistas humanos, $P(H \mid E)$ es generalmente el grado de factibilidad del especialistas de que alguna hipótesis sea verdadera, dada alguna evidencia, E. Por supuesto, si los eventos son repetibles, entonces $P(H \mid E)$ es simplemente la probabilidad.

Si estamos de acuerdo en que $P(H \mid E)$ significa la posibilidad o el grado de certeza, entonces ¿qué significa un valor como 50 o 95 por ciento? Por ejemplo, supongamos que usted está seguro en 95 por ciento de que su automóvil arrancará la próxima vez que lo encienda. Una manera de interpretar esa posibilidad es como los **momios** de una apuesta. Los momios de A contra B dado algún evento C son:

$$\text{momios} = \frac{P(A \mid C)}{P(B \mid C)}$$

Si $B = A'$:

$$\text{momios} = \frac{P(A \mid C)}{P(A' \mid C)} = \frac{P(A \mid C)}{1 - (P(A \mid C)}$$

definiendo:

$$P = P(A \mid C)$$

da

$$\text{momios} = \frac{P}{1 - P} \quad \text{y} \quad P = \frac{\text{momios}}{1 + \text{momios}}$$

Desde el punto de vista de los momios para juegos de azar, puede interpretarse P como ganados y $1 - P$ como perdidos, de modo que:

$$\text{momios} = \frac{\text{ganados}}{\text{perdidos}}$$

Sabiendo que los momios permiten que se calcule la probabilidad o posibilidad, y viceversa

La posibilidad de $P = .95$ por ciento es equivalente entonces a

$$\text{momios} = \frac{.95}{1 - .95} = 19 \text{ a } 1$$

de que usted crea que el automóvil arrancará. Es decir, usted debe ser **indiferente** a una apuesta donde los momios sean de 19 a 1 a que el automóvil arrancará. Si alguien apuesta $1 [illegible] el automóvil no arranca, usted pagaría $19 si no arranca. Cada vez que descubrimos [illegible]

momios equivalentes. En otras palabras, debe ser indiferente a la situación real con su grado de certeza, o sus equivalentes los momios de una apuesta.

Las probabilidades suelen usarse con problemas deductivos en los que pueden ocurrir varios eventos diferentes, E_i, dada la misma hipótesis. Por ejemplo, puesto que un dado arrojó un número par, hay tres posibles eventos:

```
P(2 | par)
P(4 | par)
P(6 | par)
```

En la probabilidad, generalmente nos interesamos en $P(E_i \mid H)$ donde E_i son los eventos posibles a partir de una hipótesis común. En estadística y razonamiento inductivo, se sabe que el evento ha ocurrido y pretende encontrarse las posibilidades de la hipótesis que pueda causar E, que es $P(E \mid H_i)$. La probabilidad es encadenamiento hacia delante o deductivo, mientras que la posibilidad es encadenamiento hacia atrás e inductivo. Aunque podemos usar el mismo simbolismo, $P(X \mid Y)$, para probabilidad y posibilidad, sus aplicaciones son diferentes. Normalmente nos referimos a la posibilidad de una hipótesis o la probabilidad de una evento.

Una teoría que se ha estado desarrollando mediante la utilización de grados de certeza es la **probabilidad personal** (Savage 54). En ésta, los **estados** son las posibles hipótesis y las **consecuencias** los resultados de la acción basados en las certezas.

4.12 SUFICIENCIA Y NECESIDAD

El teorema de Bayes es

$$(1) \quad P(H \mid E) = \frac{P(E \mid H)\ P(H)}{P(E)}$$

y con las negación de H se vuelve

$$(2) \quad P(H' \mid E) = \frac{P(E \mid H')\ P(H')}{P(E)}$$

Al dividir (1) entre (2) se tiene

$$(3) \quad \frac{P(H \mid E)}{P(H' \mid E)} = \frac{P(E \mid H)\ P(H)}{P(E \mid H')\ P(H')}$$

La definición de la factibilidad previa en H es:

$$M(H) = \frac{P(H)}{P(H')}$$

y las factibilidades posteriores como

$$M(H \mid E) = \frac{P(H \mid E)}{P(H' \mid E)}$$

y finalmente la definición de la **razón de posibilidad**

$$(4)\quad PS = \frac{P(E \mid H)}{P(E \mid H')}$$

entonces (3) se vuelve

$$(5)\quad M(H \mid E) = PS\ M(H)$$

A la ecuación (5) se le conoce como **forma de factibilidad-posibilidad** del teorema de Bayes, y resulta más conveniente para trabajar que la ecuación (1).

Al factor PS también se le llama **posibilidad de suficiencia** porque si PS = ∞, entonces la evidencia E es lógicamente suficiente para concluir que H es verdadero. Si E es lógicamente suficiente para concluir H, entonces P(H | E) = 1 y P(H | E′) = 0. La ecuación (5) puede utilizarse para resolver la PS como sigue:

$$(6)\quad PS = \frac{M(H \mid E)}{M(H)} = \frac{\dfrac{P(H \mid E)}{P(H' \mid E)}}{\dfrac{P(H)}{P(H')}}$$

Ahora P(H)/P(H') es alguna constante, C, y así la ecuación (6) se convierte en

$$PS = \frac{\frac{1}{0}}{C} = \infty$$

La ecuación (4) también muestra en este caso que H es suficiente para E. En la tabla 4.10 se resumen los significados de otros valores de la PS.

La probabilidad de necesidad, PN, se define de manera similar a PS como:

$$(7)\quad PN = \frac{M(H \mid E')}{M(H)} = \frac{P(E' \mid H)}{P(E' \mid H')} = \frac{\dfrac{P(H \mid E')}{P(H' \mid E')}}{\dfrac{P(H)}{P(H')}}$$

$$(8)\quad M(H \mid E') = PN\ M(H)$$

Si PN = 0, entonces P(H | E′) = 0. Esto significa que H debe ser falsa cuando E′ es verdadero. Por tanto, si E no está presente, entonces H es falso, lo que significa que E es necesario para H.

En la tabla 4.11 se muestra la relación entre PS, E y H. Observe que es igual a la tabla 4.10, con la excepción de que "evidencia" se reemplaza con "ausencia de evidencia".

Un perito humano debe proporcionar los valores de posibilidad, PS y PN, para calcular las factibilidades posteriores. El factor PS muestra cómo cambian la factibilidad previa cuando la evidencia está presente y el factor PN muestra cómo cambian cuando está ausente. Estas formas facilitan la especificación de los factores PS y PN por parte del especialista.

Por ejemplo, en el sistema experto PROSPECTOR hay una regla que especifica cómo la evidencia de un cierto mineral sustenta una hipótesis.

```
SI hay vetas de sulfuro de cuarzo
ENTONCES hay una alteración favorable
   para la zona de potasio
```


Esta hipótesis intermedia en particular, proporciona apoyo para otras hipótesis que llevan a la hipótesis de nivel superior que sostiene que hay cobre. Los valores de PS y PN para esta regla son

```
PS = 300
PN = 0.2
```

lo que significa que la observación de vetas de sulfuro de cuarzo es muy favorable mientras que no observarlas es un poco desfavorable. Si PN fuera <<1, entonces la ausencia de vetas de sulfuro de cuarzo sugerirá fuertemente que la hipótesis es falsa. Un ejemplo es la regla:

```
SI hay limonita cristalina
ENTONCES se favorece mejor mineralización
```

con

```
PS = 1,000,000
PN = 0.01
```

PS	Efecto en la hipótesis
0	H es falsa cuando E es verdadera o E′ es necesario para concluir H
pequeña(0<PS<<1)	E es desfavorable para concluir H
1	E no tiene efecto en la factibilidad de H
grande (1<<PS)	E es favorable para concluir H
∞	E es lógicamente suficiente para H o La observación de E significa que H debe ser verdadera

Tabla 4.10 Relación entre de posibilidad de similitud, hipótesis y evidencia

PN	Efecto en la hipótesis
0	H es falsa cuando E está ausente o E es necesario para H
pequeña (0<PN<<1)	La ausencia de E es desfavorable para concluir H
1	La ausencia de E no tiene efecto en H
grande (1<<PN)	La ausencia de E es favorable para H
∞	La ausencia de E es lógicamente suficiente para H

Tabla 4.11 Relación entre posibilidad de necesidad, hipótesis y evidencia

4.13 INCERTIDUMBRE EN LAS CADENAS DE INFERENCIA

La incertidumbre puede estar presente en las reglas, en la evidencia utilizada por ellas o en ambas. En esta sección se contemplan algunos problemas reales de incertidumbre y la manera en que la probabilidad proporciona una respuesta.

Inconsistencia del experto

A partir de la ecuación (4) de la sección precedente previa,

si PS > 1 entonces P(E | H') < P(E | H)

Restando 1 a cada lado, se invierte la desigualdad

1 - P(E | H') > 1 - P(E | H)

Como P(E´ | H) = 1 – P(E | H) y P(E´ | H´) = 1 – P(E | H´), entonces (7) se convierte en:

$$PS = \frac{1 - P(E \mid H)}{1 - P(E \mid H')} < 1$$

Las restricciones de los valores de PS y PN se resumen como sigue:

Caso 1: PS > 1 y PN < 1

De las ecuaciones (4) y (7), los otros casos son

Caso 2: PS < 1 y PN > 1

Caso 3: PS = PN = 1

Aunque estos tres casos son restricciones matemáticamente rigurosas de los valores PS y PN, no siempre funcionan en la realidad. Para el sistema experto PROSPECTOR, usado para la exploración de minerales, no es poco común que los peritos especifiquen PS > 1 y PN = 1, que no es uno de los tres casos. Es decir, el especialista está diciendo que la observación de evidencia es importante, pero que la ausencia de evidencia no es importante.

Este último caso muestra que la teoría de posibilidad basada en la teoría de la probabilidad bayesiana, está incompleta para el problema de la exploración mineral. Es decir, esta teoría sólo es una aproximación a la teoría, que también puede tratar con el caso de que PS > 1 y PN = 1. Para dominios en los que la opinión de los especialistas satisface uno de los primeros tres casos, la teoría de la posibilidad bayesiana es satisfactoria.

Evidencia incierta

En la realidad es raro que se tenga la seguridad absoluta de algo, con excepción de la muerte y los impuestos. Aunque hasta ahora nos hemos concentrado en hipótesis inciertas, la situación más general y realista es con hipótesis y evidencias inciertas.

Para el caso general, supongamos que el grado de factibilidad en la **evidencia completa**, E, es dependiente de la **evidencia parcial**, e, mediante:

P(E | e)

La evidencia completa es la evidencia total, que representa toda la evidencia posible, y la hipótesis, que incluye E. La evidencia parcial, e, es la parte de E que conocemos. Si conocemos toda la evidencia, entonces E = e y:

P(E | e) = P(E)

donde P(E) es la posibilidad previa de la evidencia E. La posibilidad P(E | e) es nuestra creencia de E dado nuestro conocimiento imperfecto, e, de la evidencia completa E.

Por ejemplo, suponiendo que todas las personas del vecindario que empezaron a perforar en sus cocinas en busca de petróleo se han vuelto ricas. Puede considerarse una hipótesis y cierta evidencia:

H = voy a volverme rico
E = hay petróleo bajo mi cocina

como se expresa en la regla:

SI hay petróleo bajo mi cocina
ENTONCES voy a volverme rico P(H | E) = 1

En esta etapa inicial, no es necesario saber con seguridad que hay petróleo bajo la cocina. La evidencia concluyente sería perforar un pozo de prueba, pero esto es más bien caro. Así que se considera la evidencia parcial, e, en apoyo de E, como sigue:

- Otras personas del vecindario se han vuelto ricas de súbito
- Siempre hay cierta sustancia negra alrededor de la estufa de la cocina (y hasta ahora su cónyuge le ha echado la culpa de que cocina y no limpia).
- Un extraño tocó a su puerta y le ofreció $20,000,000 por su casa porque dijo le gusta el paisaje.

Con base en esta evidencia parcial, se puede decidir que la posibilidad de que haya petróleo bajo su cocina es más bien alta, P(E | e) = 0.98.

En una cadena de información de tipo probabilístico o de posibilidad, si H depende de E y E está basado en alguna evidencia parcial, e, entonces P(H | e) es la posibilidad de que H dependa de e. Por medio de la regla de la probabilidad condicional se tiene:

$$P(H \mid e) = \frac{P(H \cap e)}{P(e)}$$

En la figura 4.15 se ilustra la manera en que puede considerarse que H está conformada por E y e. Así, lo anterior se convierte en

$$P(H \mid e) = \frac{P(H \cap E \cap e) + P(H \cap E' \cap e)}{P(e)}$$

Utilizando la regla de la probabilidad condicional se obtiene:

$$P(H \mid e) = \frac{P(H \cap E \mid e) + P(e) + P(H \cap E' \mid e)\ P(e)}{P(e)}$$

$$(1)\quad P(H \mid e) = P(H \cap E \mid e) + P(H \cap E' \mid e)$$

Otra manera de expresar (1) se desprende porque

$$P(H \cap E \mid e) = \frac{P(H \cap E \cap e)}{P(e)}$$

$$(2)\quad P(H \cap E \mid e) = \frac{P(H \mid E \cap e)\ P(E \cap e)}{P(e)}$$

y como

$$P(E \mid e) = \frac{P(E \cap e)}{P(e)}$$

entonces (2) se vuelve:

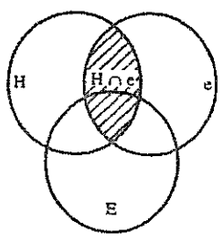
$$P(H \cap E \mid e) = P(H \mid E \cap e)\ P(E \mid e)$$

De igual manera:

$$P(H \cap E' \mid e) = P(H \mid E' \cap e)\ P(E' \mid e)$$

y (1) se vuelve:

$$(3)\quad P(H \mid e) = P(H \mid E \cap e)\ P(E \mid e) + P(H \mid E' \cap e)\ P(E' \mid e)$$



$$H \cap e = H \cap E \cap e + H \cap E' \cap e$$

Figura 4.15 La intersección de H y e

Normalmente no se conocen las probabilidades $P(H \mid E \cap e)$ y $P(H \mid E' \cap e)$. Sin embargo, si suponemos que esto puede aproximarse mediante $P(H \mid E)$ y $P(H \mid E')$, entonces (3) se simplifica:

$$(4)\quad P(H \mid e) = P(H \mid E)\ P(E \mid e) + P(H \mid E')\ P(E' \mid e)$$

La ecuación (4) es esencialmente una interpolación lineal de $P(H \mid e)$ contra $P(E \mid e)$. Los puntos finales son:

(i) E es verdadero y así $P(H \mid e) = P(H \mid E)$.
(ii) E es falsa y así $P(H \mid e) = P(H \mid E')$.

El problema con la ecuación (4) destaca cuando $P(E \mid e)$ se iguala con la probabilidad previa $P(E)$. Si el sistema obedece la pura probabilidad bayesiana, entonces

$$(5)\quad P(H \mid e) = P(H \mid E)\ P(E) + P(H \mid E')\ P(E')$$
$$(6)\quad P(H \mid e) = P(H)$$

que es correcta.

Sin embargo, en la realidad, la experiencia ha mostrado que los especialistas humanos confieren probabilidades subjetivas que son casi ciertas pero frágiles. Por ejemplo, si el perito usa el caso sin sustento de PS > 1 y PN = 1, entonces:

$$M(H \mid E') = PN\ M(H) = M(H)$$

y como:

$$M = \frac{P}{1 - P}$$

entonces

$$(7)\quad P(H \mid E') = P(H)$$

Utilizando (7) en (5) se tiene:

$$P(H \mid e) = P(H \mid E)\ P(E) + P(H)\ P(E')$$
$$= P(H \mid E)\ P(E) + P(H)\ (1 - P(E)$$
$$(8)\quad P(H \mid e) = P(H) + P(H \mid E)\ P(E) - P(H)\ P(E)$$

Ahora:

$$M(H \mid E) = PS\ M(H)$$

Si PS = 1 entonces

$$M(H \mid E) = M(H)$$
$$P(H \mid E) = P(H)$$

Como el especialista ha especificado PS > 1, entonces $P(H \mid E) > P(H)$ y así el término

$$P(H \mid E)\ P(E) - P(H)\ P(E)$$

en la ecuación (8) será > 0, donde 0 es el límite inferior cuando PS = 1. Así, de (8)

$$P(H \mid e) > P(H)$$

cuando PS> 1 y PN = 1. Esto contradice el hecho de que $P(H \mid e)$ debe ser igual a $P(H)$ cuando $P(E \mid e) = P(E)$. Como $P(H \mid e) > P(H)$, la probabilidad es mayor de lo que debería y puede ampliarse aun más a medida que la inferencia a partir de una regla es usada por otra regla en

Corrección de la incertidumbre

Una manera de corregir este problema es suponer que P(H | e) es una función lineal de una sola pieza (Duda 76). Ésta es una suposición *ad hoc* que funciona bien con PROSPECTOR, pero que no está basada en la teoría de la probabilidad tradicional. La función lineal se elige por su simplicidad en los cálculos.

La función iguala las restricciones en los tres puntos de la figura 4.16. La fórmula para P(H | e) se calcula utilizando la interpolación lineal, como sigue:

$$P(H \mid e) = \begin{cases} P(H \mid E') + \dfrac{P(H) - P(H \mid E')}{P(E)} P(E \mid e) \\ \qquad \text{para } 0 \le P(E \mid e) < P(E) \\ P(H) + \dfrac{P(H \mid E) - P(H)}{1 - P(E)} [P(E \mid e) - P(E)] \\ \qquad \text{para } P(E) \le P(E \mid e) \le 1 \end{cases}$$

Utilizando esta fórmula, puede satisfacerse el caso analizado antes, donde de PS > 1 y PN = 1. Ahora, el valor de P(H | e) permanece igual si P(E | e) < P(E) y aumenta si P(E | e) ≥ P(E). La razón para elegir esta única pieza más que en una sola línea recta, es para asegurarse de que cuando P(E | e) = P(E), entonces P(H | e) = P(H).

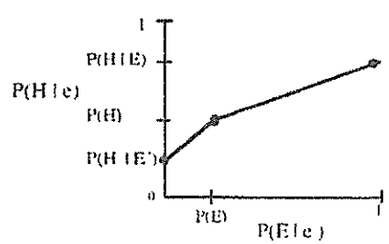


Figura 4.16
Función de interpolación lineal de una sola pieza para la evidencia parcial en PROSPECTOR

4.14 COMBINACIÓN DE EVIDENCIA

El tipo de regla más simple es con la forma

```
SI E ENTONCES H
```

donde E es una sola pieza de la evidencia conocida, a partir de la cual puede concluirse que H es verdadera. Por desgracia, no todas las reglas pueden ser así de simples, de modo que es necesario compensar la incertidumbre.

Clasificación de evidencia incierta

Si hay incertidumbre en relación con la regla surgen situaciones más complejas. Esta incertidumbre puede expresarse como una posibilidad, dependiendo de si se está tratando con eventos reproducibles o probabilidades subjetivas. Para simplificar el análisis, se utiliza el término *probabilidad* para las incertidumbres.

Las diferentes situaciones pueden clasificarse conforme a si la evidencia es cierta o incierta y a si hay **evidencia simple** o **evidencia compuesta**. La evidencia simple incluye una sola pieza de evidencia, como en:

```
SI la transmisión está descompuesta
ENTONCES repare el automóvil
```

mientras que la evidencia compuesta consta de varias piezas de evidencia normalmente vinculada mediante conexiones Y, como en:

```
SI la transmisión está descompuesta Y
   el carburador está descompuesto
ENTONCES venda el automóvil
```

o expresada formalmente:

SI E_1 Y E_2 entonces H

Es posible asignar una probabilidad $P(H \mid E_1 \cap E_2) = 0.80$, a esta regla lo que significa que, con base en esta evidencia hay 80 por ciento de seguridad de que el automóvil debe venderse.

Para refinar aún más la evidencia, podemos determinar su probabilidad. Por ejemplo, la probabilidad de que la transmisión esté descompuesta puede depender de dos síntomas:

(a) El fluido de la transmisión está goteando
(b) El automóvil no avanza en reversa

Esta observación conduce a una probabilidad de la evidencia de la transmisión. Por ejemplo, con base en los síntomas (a) y (b), puede decidirse:

```
P(E | e) = 0.95

donde E = transmisión está descompuesta
      e = síntomas (a) y (b)
```

Ahora examinemos con más detalle las probabilidades asociadas con las diferentes situaciones.

Situación 1: una pieza de evidencia conocida concluye H

Éste es el caso más simple, con las reglas de la forma

```
SI E ENTONCES H
```

Antes de cualquier evidencia, se sabe que la probabilidad previa de H es P(H); después de la

$$P(H \mid E) = \frac{P(H \cap E)}{P(E)} = \frac{P(H \cap E)}{P(E \cap H) + P(E \cap H')}$$

$$P(H \mid E) = \frac{P(H \mid E)\ P(H)}{P(E \mid H)\ P(H) + P(E \mid H')\ P(H')}$$

donde P(E) es la probabilidad de observar E.

Situación 2: dos piezas de evidencia conocidas concluyen H

Es un caso más complejo que la situación 1 y corresponde a las reglas con la forma:

SI E_1 y E_2 entonces H

Después de que se observan E_1 y E_2, la probabilidad de H cambia de P(H) a:

$$P(H \mid E_1 \cap E_2) = \frac{P(H \cap E_1 \cap E_2)}{P(E_1 \cap E_2)}$$

$$= \frac{P(H \cap E_1 \cap E_2)}{P(E_1 \cap E_2 \cap H) + P(E_1 \cap E_2 \cap H')}$$

$$(1)\quad P(H \mid E_1 \cap E_2)$$

$$= \frac{P(E_1 \cap E_2 \mid H)\ P(H)}{P(E_1 \cap E_2 \mid H)\ P(H) + P(E_1 \cap E_2 \mid H')\ P(H')}$$

Esta fórmula no puede reducirse más a menos que se suponga una simplificación. Si se supone que las evidencias E_1 y E_2 son condicionalmente independientes entre sí, entonces

$$P(E_1 \cap E_2 \mid H) = P(E_1 \mid H)\ P(E_2 \mid H)$$
$$P(E_1 \cap E_2 \mid H') = P(E_1 \mid H')\ P(E_2 \mid H')$$

y así (vea el problema 4.12 para la independencia condicional)

$$(2)\quad P(H \mid E_1 \cap E_2)$$

$$= \frac{P(E_1 \mid H)\ P(E_2 \mid H)}{P(E_1 \mid H)\ P(E_2 \mid H) + O(H')\ P(E_1 \mid H')\ P(E_2 \mid H')}$$

La ecuación (2) es una simplificación importante de la ecuación (1); todas las probabilidades se expresan desde el punto de vista de las probabilidades individuales más que de las uniones, como $P(E_1 \cap E_2 \mid H')$. Más adelante se analizan los problemas con la suposición de independencia condicional.

Aunque (2) es una simplificación, todavía requiere que se conozcan las probabilidades previas $P(E_1)$ y $P(E_2)$. Por lo general, es difícil para los especialistas establecer probabilidades previas porque no razonan de esa manera. Por ejemplo, si una persona acude al doctor, éste no suponе *a priori* que la persona tiene un resfrío [illegible]

El principal problema de asignar probabilidades de antemano es la dificultad de determinarlas a partir de sus posibilidades. Para las probabilidades relacionadas con eventos reproducibles, como lanzar un dado, es fácil adquirir las probabilidades por estudios empíricos o teóricos; pero cuando se trata con eventos como encontrar un depósito importante de minerales, es imposible determinar con exactitud las posibilidades previas. Por ejemplo, ¿cuál es la posibilidad previa de que su casa se asiente en un terreno que cubre un importante yacimiento de oro? ¿Es de 0.0000001,0.00000001 o algún otro número? Éstas son situaciones únicas en las que no pueden determinarse las probabilidades previas.

Situación 3: N piezas de evidencia incierta concluyen H

Éste es un caso general de incertidumbre en la evidencia y en la regla que depende de la primera. A medida que aumenta el número de piezas de evidencia, es imposible determinar todas las probabilidades o posibilidades de unión y previas. Se han usado diferentes aproximaciones para tratar el caso general de *N* piezas de evidencia.

Combinación de evidencia mediante la lógica borrosa

Conjunción de evidencia

Supuesta la regla:

SI E ENTONCES H

en que E es la conjunción de evidencia, como en

SI E_1 Y E_2 Y ... E_N ENTONCES H

Toda E_i debe ser verdadera con alguna probabilidad para el antecedente de que ha sido verdadera. En el caso general, cada pieza de evidencia se basa en evidencia parcial, e. La probabilidad de la evidencia es:

$$P(E \mid e) = (P(E_1 \cap E_2 \cap \ldots E_N \mid e)$$

$$= \frac{P(E_1 \cap E_2 \cap \ldots E_N \cap e)}{P(e)}$$

Si toda la evidencia, E_i, es condicionalmente independiente, entonces la probabilidad de unión es igual al producto de las probabilidades individuales. Esto se desprende de la Ley multiplicativa generalizada. Un ejemplo para dos piezas de evidencia:

$$P(E_1 \cap E \mid e) = \frac{P(E_1 \cap E_2 \cap e)}{P(e)}$$

$$= \frac{P(E_1 \mid E_1 \cap e)\ P(E_2 \mid e)\ P(e)}{P(e)}$$

Utilizando la suposición de independencia:

porque la evidencia E_1 no aporta ningún conocimiento a E_2. Así:

$$P(E_2 \cap E_1 \mid e) = P(E_2 \mid e)\, P(E_1 \mid e)$$

y en general

$$P(E_1 \cap E_2 \cap \ldots E_N \mid e) = \prod_{i=1}^{N} P(E_i \mid e)$$

Aunque esta fórmula es correcta en el sentido teórico, hay dos dificultades en aplicarla a problemas reales. En primer lugar, la $P(E_i \mid e)$ individual no suele ser independiente en la realidad; en segundo, la multiplicación de muchos factores conduce a un producto que generalmente es demasiado pequeño para $P(E \mid e)$.

Una solución aproximada a este problema es el uso de la **lógica de confusión** para calcular $P(E \mid e)$, como sigue:

$$P(E \mid e) = \text{mín}\ [P(E_i \mid e)]$$

donde la **función min** devuelve el valor mínimo de toda la $P(E_i \mid e)$. En el sistema PROSPECTOR, esta fórmula fue satisfactoria. Una vez determinado $P(E \mid e)$, puede usarse en la fórmula lineal de una sola pieza.

El principal problema con la fórmula de la lógica de confusión es que hace a $P(E \mid e)$ insensible a cualquier $P(E_i \mid e)$, excepto el mínimo. Esto significa que si todas las demás probabilidades aumentan y el mínimo sigue igual, no se efectuará ningún cambio en $P(E \mid e)$. Es decir, el mínimo $P(E_i \mid e)$ bloquea otros cambios de probabilidad para la propagación a través de la cadena de inferencia. Una ventaja de la fórmula de la lógica de confusión es que resulta simple hacer cálculos con ella.

Disyunción de evidencia

Si la regla es una disyunción de evidencia

SI E_1 O E_2 O ... E_N ENTONCES H

entonces puede demostrarse que (vea el problema 4.13) la suposición de la independencia de la evidencia da

$$P(E \mid e) = 1 - \prod_{i=1}^{n} [1 - P(E_i \mid e)]$$

El problema con esta fórmula es que la probabilidad calculada es muy elevada. La fórmula utilizada por PROSPECTOR como alternativa para esto se basa en la lógica de confusión:

$$P(E \mid e) = \text{máx}\ [P(E_i \mid e)]$$

Combinación lógica de evidencia

Si el antecedente es una combinación lógica de evidencia, entonces pueden usarse la lógica de confusión y las reglas de negación para combinar la evidencia. Por ejemplo:

SI E_1 Y (E_2 O E_3') ENTONCES H

entonces

$$E = E_1 \text{ Y } (E_2 \text{ O } E_3')$$
$$E = \text{mín}\ \{P(E_1 \mid e),\ \text{máx}\ [P(E_2 \mid e),\ 1 - P(E_3 \mid e)]\}$$

Aunque estas fórmulas de la lógica de confusión se han usado con éxito en varios sistemas, pueden definirse otras funciones para la combinación de evidencia. Por ejemplo, la siguiente es una alternativa a la función máx de disyunción (Quinlan 86):

$$P(E_1 \cup E_2 \mid H) = \text{mín}\ [1,\ P(E_1 \mid H) + P(E_2 \mid H)]$$

Posibilidades efectivas

En general, varias reglas con evidencia incierta y probabilidades previas inconsistentes pueden concluir una hipótesis específica. Suponiendo que la independencia condicional de la evidencia y todo E_i contribuye a que H sea verdadera, entonces:

$$M(H \mid E_1 \cap E_2 \cap \ldots E_N) = \left[\prod_{i=1}^{N} PS_i\right] M(H)$$

donde PS_i se define como

$$PS_i = \frac{P(E_i \mid H)}{P(E_i \mid H')}$$

Y se sostiene una fórmula similar si toda la evidencia que contribuye a H es falsa:

$$M(H \mid E_1' \cap E_2' \cap \ldots E_N') = \left[\prod_{i=1}^{N} PN_i\right] M(H)$$

donde

$$PN_i = \frac{P(E_i' \mid H)}{P(E_i' \mid H')}$$

En el caso general de las probabilidades previas inconsistentes y de la evidencia incierta, la **relación de posibilidad efectiva**, PE, se define como

$$O(H \mid e)$$

donde e_i es la i-ésima evidencia parcial que contribuye a H. La actualización de H basada en evidencia incierta e inconsistente es análoga al caso previo:

$$M(H \mid e_1' \cap e_2' \cap \ldots e_N') = \left[\prod_{i=1}^{N} PE_i\right] M(H)$$

Para un sistema experto que utiliza evidencia incierta y probabilidades previas inconsistentes, esta fórmula puede utilizarse de la siguiente manera:

(a) Almacena las creencias previas de cada regla y la PE de cada contribución a la regla
(b) Cada vez que se actualiza $P(E_i \mid e_i)$, calcula la nueva PE_i y las creencias posteriores.

Dificultades con la independencia condicional

Aunque la suposición de la independencia condicional es útil para simplificar el teorema de Bayes, existen otros problemas. Esta suposición es útil sobre todo en las fases iniciales de un sistema experto, cuando el comportamiento general del sistema es más importante que los resultados numéricos correctos. Al principio, un ingeniero del conocimiento puede estar más interesado en el desarrollo de las cadenas de inferencia correctas que en los números correctos. Es decir, en algún sistema, la hipótesis intermedia 10 debe activarse con las evidencias 23 y 34; luego la hipótesis 10 y la evidencia 8 deben activar las hipótesis 52 y 96; la hipótesis 15 *no* debe ser activada por las evidencias 23 y 24, etcétera.

Como se muestra en la ecuación (2) de esta sección, en términos de la independencia condicional sólo se requieren cinco probabilidades para calcular $P(H \mid E_1 \cap E_2)$. Estas son:

$$P(E_1 \mid H),\ P(E_2 \mid H),\ P(E_1 \mid H'),\ P(E_2 \mid H'),\ P(H)$$

La otra probabilidad $P(H')$ necesaria para calcular $M(H')$ no es independiente de las otras porque

$$P(H') = 1 - P(H)$$

Otra fórmula para $P(H \mid E_1 \cap E_2)$ se muestra en el problema 4.12 (b). Esta fórmula incluye otras cuatro probabilidades:

$$P(H \mid E_1),\ P(H \mid E_2),\ P(H' \mid E_1),\ P(H' \mid E_2)$$

Por tanto, puede utilizarse nueve probabilidades en total para calcular $P(H \mid E_1 \cap E_2)$, pero en realidad sólo es necesario especificar cinco. Dadas cinco, pueden calcularse las demás.

Esta restricción en el número de probabilidades independientes se vuelve un problema real cuando el sistema experto madura. Una vez que las cadenas de inferencia funcionan correctamente, el ingeniero en conocimiento debe asegurarse que da salida a los números correctos. Debido a la suposición de la independencia condicional, el ingeniero en conocimiento no tiene libertad completa para afinar las nueve probabilidades mediante el ajuste de valores, y obtener los resultados deseados.

La suposición de independencia condicional fija la probabilidad de unión $P(E_1 \cap E_2)$ porque

$$P(E_1 \cap E_2) = P(E_1)P(E_2)\left[\frac{P(H \mid E_1)P(H \mid E_2)}{P(H)} + \frac{P(H' \mid E_1)P(H' \mid E_2)}{P(H')}\right]$$

Esto significa que ahora las probabilidades previas $P(E_1)$, $P(E_2)$ y $P(H)$ restringen $P(E_1 \cap E_2)$. Esto es contrario al conocimiento de los peritos que pueden conocer las probabilidades $P(E_1)$, $P(E_2)$ y $P(H)$ por separado. De este modo, la independencia condicional restringe la aplicación de todo el conocimiento del especialista humano.

Aunque la suposición de la independencia condicional parece razonable, depende de la situación real que el sistema experto está modelando y no tiene que ser así en todos los casos. Un artículo de investigación fue más allá en esto y aseguró que la independencia condicional suele ser falsa (Szolovits 78). ¡Otro artículo aseguró que requiere independencia estricta y que esto prueba que la evidencia es irrelevante para actualizar las hipótesis (Pednault 81)! Sin embargo, esta aseveración ha sido desaprobada en otro artículo (Glymour 85).

Otro método subjetivo de aplicación de probabilidades bayesianas muestra cómo puede debilitarse la presunción de independencia condicional, de modo que ya no se requiera la independencia estricta (Pearl 82).

4.15 REDES DE INFERENCIA

Hasta este momento, los ejemplos estudiados de encadenamiento hacia delante y hacia atrás son muy pequeños y contienen pocas reglas. Para los problemas reales, el número de inferencias necesario para sustentar una hipótesis o alcanzar una conclusión es mucho mayor; además, muchas o todas estas inferencias se hacen en condiciones de incertidumbre para la evidencia y las propias reglas. El razonamiento probabilístico y el teorema de Bayes se han usado con éxito en sistemas reales como éstos.

Una red de inferencia es una buena configuración para los sistemas expertos que dependen de la taxonomía del conocimiento. Una taxonomía es una clasificación y se usa con frecuencia en ciencias naturales como la geología y la biología. Ya ha visto una taxonomía simple en la figura que clasifica los diferentes tipos de frambuesas en los problemas del capítulo 2.

Una taxonomía es útil para dos propósitos: ayuda a organizar el conocimiento acerca de un tema al clasificar objetos y al mostrar sus relaciones con otros objetos. Los taxónomos aclaran funciones importantes como la herencia de propiedades.

Una segunda razón por la que los taxónomos son útiles es que pueden guiar la investigación para probar una hipótesis, como "hay mineral de cobre en este sitio". Este apoyo para probar o desaprobar hipótesis es muy importante en áreas como la exploración mineral, porque en raras ocasiones existen evidencias directas de mineral valioso en la superficie. Antes de decidir que se invierta tiempo y dinero para perforar, un geólogo tratará de reunir la evidencia que sustente su hipótesis.

PROSPECTOR

PROSPECTOR es el sistema experto clásico que usa el razonamiento probabilístico (Duda 79); está diseñado para ayudar en la exploración geológica destinada a determinar si un sitio es favorable para ciertos tipos de depósitos minerales. La idea básica de PROSPECTOR es codificar en el sistema el conocimiento económico que tiene el perito geólogo de los diferentes

identificación de minerales, PROSPECTOR también puede recomendar el mejor lugar para perforar en el sitio. A medida que se crean más modelos, las aptitudes de PROSPECTOR aumentan.

Los datos de cada modelo se organizan como una **red de inferencia**. En la figura 4.17 se muestra un resumen de los tipos de redes analizadas hasta ahora, además de una red de inferencia muy simple en (d). Los nodos de la red de inferencia pueden representar evidencia para respaldar otros nodos que representan hipótesis, como el tipo de mineral presente. En 1983 había 22 modelos de minerales en PROSPECTOR, de los que se resumen unos cuantos en la tabla 4.12.

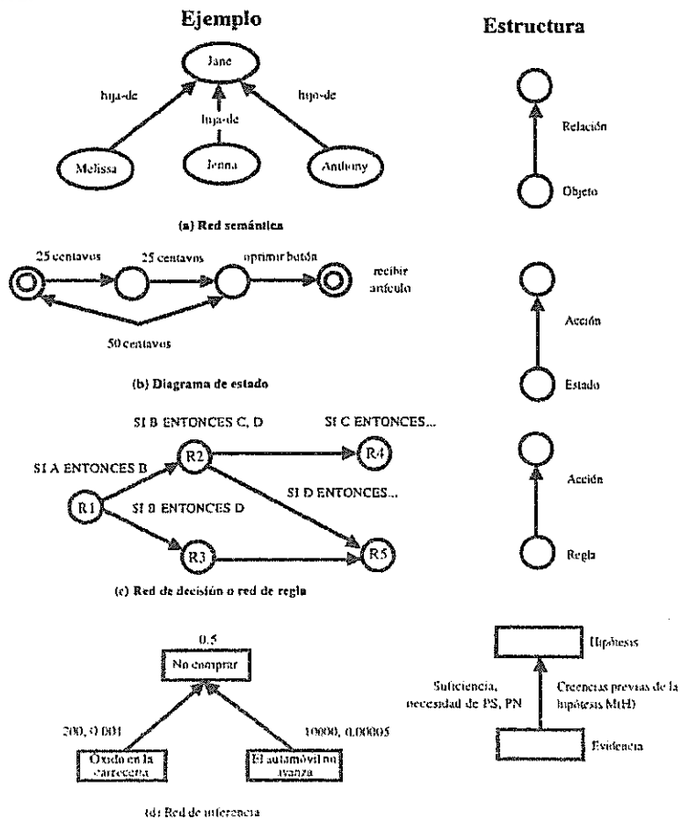


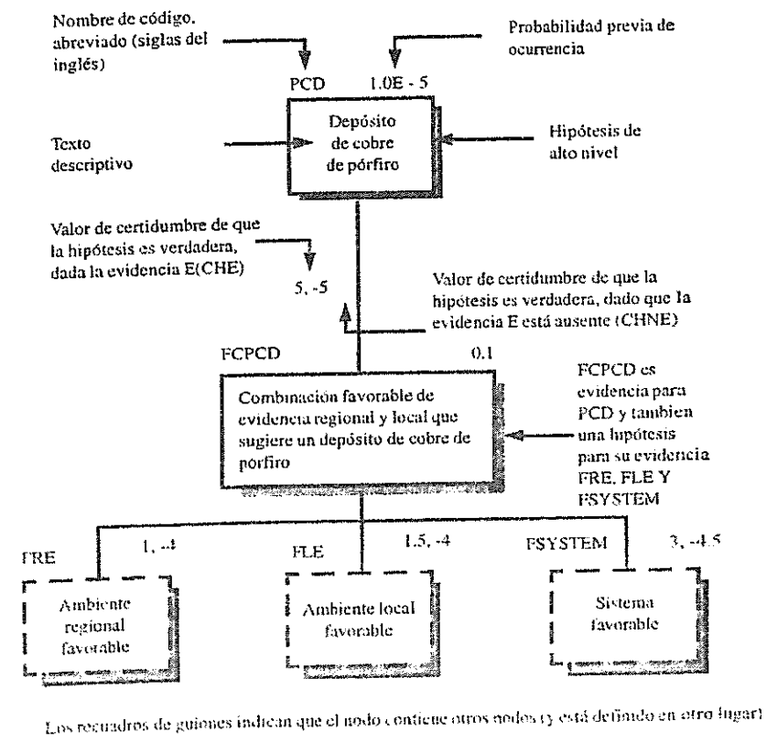
Figura 4.17 Algunos tipos de redes

Nombre del modelo	Descripción del modelo	Número de nodos	Número cuestionable	Número de reglas
PCD	Cobre de pórfiro	200	97	135
RFU	Uranio	185	147	169
SPB	Plomo en banco de arena	35	21	32

En la tabla 4.12, el número de nodos es el número total que hay en el modelo. Los **nodos cuestionables** son aquellos que hacen preguntas al usuario para obtener evidencia observable. El número de reglas en la tabla se refiere a reglas de inferencia probabilística. Como puede verse, hay mucha incertidumbre en los modelos, y la probabilidad es importante para respaldar las hipótesis.

Redes de inferencia

Cada modelo de PROSPECTOR se codifica como una red de conexiones o relaciones entre evidencias e hipótesis; de modo que una red de inferencia es un tipo de red semántica. Los hechos observables, como el tipo de formaciones rocosas obtenidas de la exploración del campo, conforman la evidencia para respaldar las hipótesis intermedias. Se utilizan grupos de hipótesis intermedias para sostener las **hipótesis de nivel superior**, que es lo que se pretende probar. Si no es importante distinguir entre evidencia e hipótesis, se usa el término **aserción** para referirse a ellas. Una pequeña parte de la red de inferencia para la hipótesis de alto nivel del modelo para cobre de pórfiro se muestra en la figura 4.18.



Los recuadros de guiones indican que el nodo contiene otros nodos (y está definido en otro lugar)

Figura 4.18
Hipótesis de alto nivel para el cobre de pórfiro, establecida por el sistema PROSPECTOR. Expresada utilizando factores de certidumbre

PROSPECTOR utiliza los **factores de certidumbre** CHE Y CHNE porque la experiencia ha mostrado que a los peritos les resulta difícil especificar probabilidades posteriores o relaciones de posibilidad. Experiencias similares se encontraron en el sistema MYCIN para diagnóstico de enfermedades sanguíneas, en él los médicos no quisieron especificar probabilidades y se utilizaron factores de certidumbre. Como en MYCIN, los factores de certidumbre se clasifican, en una escala de 11 puntos, de –5 a +5, en la que –5 significa "definitivamente no" y +5 significa "definitivamente sí".

PROSPECTOR no es un sistema probabilístico puro, porque utiliza lógica de confusión y factores de certidumbre (que se analizan con más detalle en el capítulo 5) para combinar la evidencia.

En la figura 4.19 se muestra el nodo FRE expandido con más detalle.

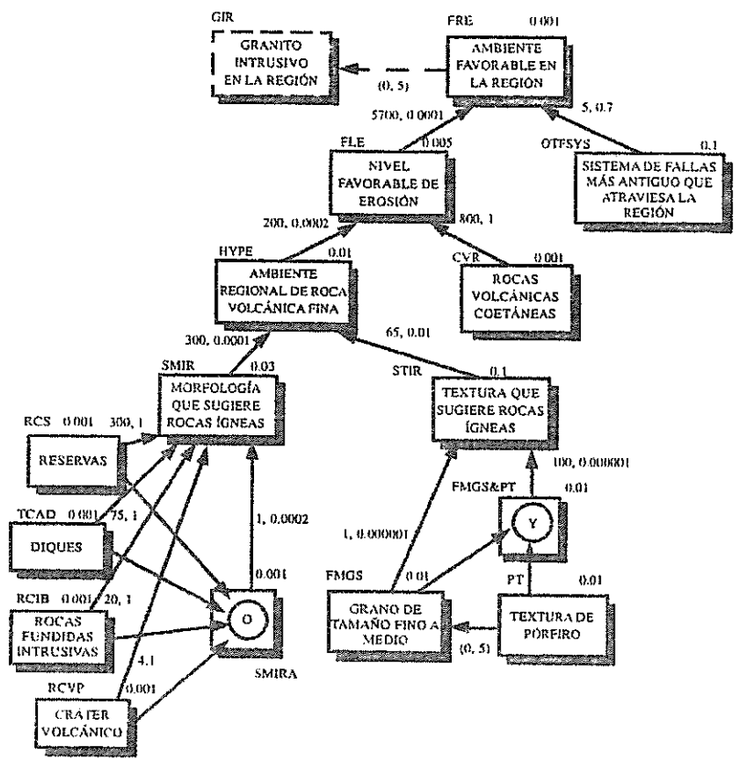


Figura 4.19 Una pequeña parte de la red de inferencia PROSPECTOR para el cobre de pórfiro

Este diagrama en particular tiene dos números arriba de los nodos, que son la relación de posibilidad, PS, y la medida necesaria, PN, separados por comas. Por ejemplo, los valores PS, PN para el nodo rotulado RCIB de la parte inferior izquierda son 20,1. El acrónimo de cada nodo representa las iniciales en inglés de su descripción, como RCIB, para "Región que contiene rocas fundidas intrusivas (Region Contains Intrusive Breccia). En la parte superior de cada

Relaciones de inferencia

La evidencia que sustenta o descarta una hipótesis es enviada hacia arriba de la red de inferencia. Por ejemplo, RCS, RCAD, RCIB, RCVP y SMIRA sustentan o descartan la hipótesis intermedia SMIR. Ésta también es evidencia de su hipótesis HYPE, que es evidencia de FLE, y así sucesivamente.

La evidencia puede combinarse de dos maneras generales, para obtener las relaciones deseadas por el geólogo en la definición del modelo:

- **Combinaciones lógicas** como las de nodos Y y O. Como se menciona en la sección anterior, puede utilizarse la lógica de confusión para calcular el resultado de estos nodos lógicos.
- **Combinaciones ponderadas** utilizando las relaciones de posibilidad, PS, y las relaciones de necesidad, PN. Las creencias posteriores se calculan de:

  $M(H \mid E) = PS\ M(H)$

  cuando se sabe que la evidencia E es verdadera y:

  $M(H \mid E') = PN\ M(H)$

  cuando se sabe que E es falsa. Si no se sabe que E es cierta, entonces $P(H \mid E)$ puede calcularse mediante la interpolación lineal, como se analiza en la sección anterior.

El término *combinación ponderada* surge debido al caso general en el que varias piezas de evidencia contribuyen a una hipótesis. Como ya se mostró:

$$M(H \mid E_1 \cap E_2 \cap \ldots E_N) = \left[\prod_{i=1}^{N} PS_i\right] M(H)$$

El logaritmo de esto es

$$\log M(H \mid E_1 \cap E_2 \cap \ldots E_N) = \log M(H) + \sum_{i=1}^{N} \log PS_i$$

que puede interpretarse como cada L_{si} que arroja el logaritmo de LS_i "vota" por la hipótesis. Cada logaritmo de LS_i es un valor relativo que afecta a la hipótesis.

Aunque a PROSPECTOR a veces se le llama un *sistema basado en reglas* porque las combinaciones ponderadas corresponden a reglas como

SI E_1 Y E_2 Y ... E_N ENTONCES H

no es tan flexible como un verdadero sistema de producción basado en reglas. Una de sus limitantes es la falta de un mecanismo completo para unir variables. PROSPECTOR es un sistema con diseño personalizado que enfatiza la eficiencia y el control de aplicaciones geológicas, más que sobre la generalidad de un sistema de producción.

Las combinaciones ponderadas también son un ejemplo de **relaciones plausibles**; el término *plausible* significa que hay cierta evidencia de credibilidad. PROSPECTOR es un ejemplo

Las inferencias plausibles de PROSPECTOR están basadas en las probabilidades bayesianas, donde los especialistas humanos proporcionan los valores de PS y PN.

La creencia plausible y de otros grados se ilustra en la gráfica de confusión de la figura 4.20. Los términos tienen los significados generales que se muestra en la tabla 4.13.

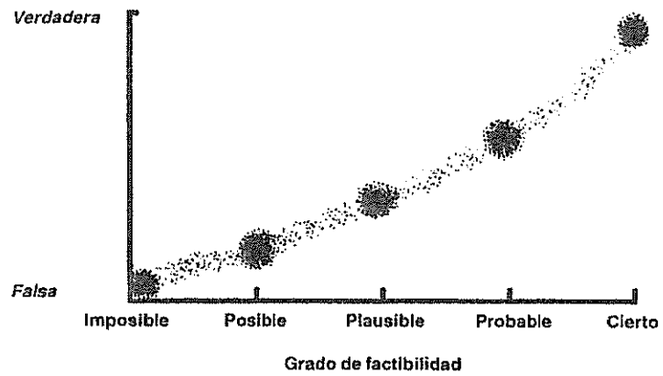


Figura 4.20 Significados relativos de algunos términos utilizados para describir evidencia

Término	Evidencia relativa a la hipótesis
imposible	definitivamente se tiene conocimiento contrario
posible	no se desaprueba definitivamente
plausible	existe alguna evidencia
probable	alguna evidencia a favor
cierto	definitivamente se sabe que se sustenta

Tabla 4.13 Algunos términos utilizados con la evidencia

La gráfica de la figura 4.20 se trazó borrosa deliberadamente, para ilustrar la naturaleza vaga de estos términos y la vaguedad en la progresión de uno a otro. En la figura 4.20, observe cómo nuestra creencia en una hipótesis aumenta desde "imposible" hasta "cierto". Una **creencia cierta** significa que la creencia es verdadera y una **creencia imposible** significa que es falsa. No hay incertidumbre relacionada con las creencias ciertas e imposibles, estas creencias son equivalentes a la verdad lógica y la falsedad lógica. El término ***evidencia cierta*** en ocasiones se utiliza de manera ambigua, la evidencia cierta es lógicamente verdadera o falsa. Es decir, no hay incertidumbre asociada con la evidencia cierta, lo que significa que ésta corresponde a una creencia cierta (lógicamente verdadera) o a una creencia imposible (lógicamente falsa).

Una **creencia posible** significa que no importa cuán remota sea, la hipótesis no puede descartarse. Por ejemplo, antes de los análisis científicos de la superficie de la luna, era posible, pero demasiado remoto que estuviera hecha de queso verde. Esta posibilidad existió debido a que no se disponía de una prueba definitiva.

Una **creencia plausible** significa que existe más de una posibilidad. El término *plausible* se utiliza con frecuencia en asuntos legales para indicar algo razonable, pero sin evidencia contundente para respaldar la creencia. Así, incluso sin estudios científicos, no es plausible que la luna esté formada por queso verde.

Una **creencia probable** significa que hay alguna evidencia a favor de la hipótesis, pero no la suficiente para probarla de manera definitiva.

Sin evidencia de campo, como la observación de RCIB, las relaciones de la figura 4.19 para un sitio en particular son meramente plausibles. A medida que se acumula evidencia, las relaciones plausibles pueden volverse *probables* y luego *definitivas*.

Un tercer tipo de relación en las redes de inferencia es el **contexto**, que bloquea la propagación de información hasta que es apropiada. El uso de contextos activa o desactiva fracciones de la red de inferencia, hasta conocer que determinadas porciones están presentes, ausentes o son desconocidas. Un propósito del contexto es evitar que el sistema tenga que preguntar al usuario sobre alguna evidencia, hasta que se establece que ésta es necesaria.

La idea básica del contexto es controlar el orden en el que se siguen las aserciones en el sistema. El contexto establece una condición requerida que debe probarse antes de poder usar una aserción. El contexto se representa con una flecha de guiones y un rango de certidumbre debajo, como en el caso de FMGS y PT. El nodo PT se bloquea, a menos que haya evidencia con valores de certidumbre en el rango de 0 a +5 de que hay granos que van de tamaño fino a medio de rocas **porfiríticas**. Éstas son rocas ígneas cuya **textura** o aspecto es de pequeños cristales incrustados en roca, que es la **matriz**. Las rocas ígneas son aquellas formadas por la solidificación de la roca fundida, llamada magma, en la profundidad de la tierra. Ciertos tipos de roca ígnea que se extiende en el terreno, llamada *rocas ígneas intrusivas*, son evidencia de formaciones de mineral como el cobre de pórfiro.

Así, este mineral consta de pequeños cristales de cobre incrustados dentro de una matriz de roca. El cobre de pórfiro es la forma más común de los yacimientos de cobre. A menos que estén presentes cristales cuyo tamaño va de pequeño a mediano, no vale la pena preguntar sobre la textura del pórfiro; el sistema experto debe ser, por tanto, lo suficientemente inteligente para no hacer esta pregunta. Un sistema experto que hace preguntas innecesarias es ineficiente y molestará rápidamente al usuario.

El rango de certidumbre de 0 a +5 para el contexto significa que el nodo PT no se seguirá a menos que el usuario indique la ausencia de evidencia que corresponde a un factor de certidumbre de 0 o alguna cantidad positiva hasta +5, para "definitivamente presente".

Los tres métodos para combinar o permitir evidencia son realmente relaciones entre nodos. Todos ellos especifican la manera en que un cambio en la probabilidad de una aserción afecta a los demás.

Arquitectura de una red de inferencia

Formalmente, una **red de inferencia** puede definirse como una gráfica acíclica con dirección, en la que los nodos son aserciones y los arcos son medidas de incertidumbre como PS y PN. En la figura 4.21 (a), (b) y (c) podrían usarse como arquitecturas de redes de inferencia válidas, porque no tienen ciclos. Observemos que en el árbol de inferencia las flechas apuntan hacia la hipótesis, a diferencia de las flechas del árbol para estructura de datos, que señalan para alejarse de la raíz. Como se describe en el capítulo 3, en una gráfica acíclica no hay manera de volver al punto de partida siguiendo las flechas. En (d) hay un ciclo que relaciona cuatro nodos, si la razón para prohibir ciclos es evitar el razonamiento circular al establecer una hipótesis, una excepción válida para los sistemas basados en reglas consiste en establecer un lazo al tener dos reglas que se disparan entre sí, hasta que se satisfaga una condición de terminación.

En la figura 4.21 (e) se encuentra un tipo indeseable de gráfica para una red de inferencia. El problema es que cuando muchas piezas de evidencia contribuyen a una sola hipótesis, hay grandes posibilidades de que se establezcan interacciones indeseables entre la evidencia. Por lo

general, es mejor reestructurar este tipo en una estructura más parecida a un árbol, con hipótesis intermedias.

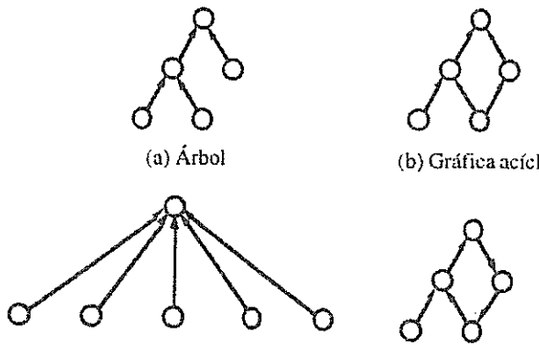


Figura 4.21 Algunos tipos de gráficas

La red de inferencia PROSPECTOR también es una **red semántica particionada** para agrupar porciones de la red en unidades significativas. Hendrix desarrolló este tipo de red para permitir la capacidad del cálculo de predicados como las cuantificaciones, la implicación, la negación, la disyunción y la conjunción (Hendrix 79). Del capítulo 2, recordemos que en realidad las redes semánticas comunes están diseñadas para expresar el conocimiento descriptivo a través de relaciones. Aunque los marcos son mejores para el conocimiento causal, son difíciles de usar en la expresión de relaciones lógicas.

La idea básica de una red semántica particionada es agrupar conjuntos de nodos y arcos en **espacios** abstractos que definen el alcance de sus relaciones. Una analogía familiar para un espacio es el ámbito de un módulo o paquete en un lenguaje estructurado. Por ejemplo, el nodo FRE de la figura 4.18 puede considerarse un espacio con la estructura mostrada en la figura 4.19.

La capacidad de las redes semánticas surge en afirmaciones de modelado; de hecho, Hendrix las desarrolló primero para expresar lenguaje natural en su disertación doctoral. Las afirmaciones son como las proposiciones de evidencia y las hipótesis que conforman los nodos de las redes de inferencia. Como ejemplo de una red semántica particionada para una afirmación, consideremos "hay una computadora con pantalla a color"; observe que se trata de una afirmación existencial, debido al cuantificador "hay".

En la figura 4.22 se muestra la expresión de la red semántica particionada de esta afirmación utilizando tres espacios: el ESPACIO-1 incluye algunos conceptos generales relacionados con las computadoras; el ESPACIO-2 contiene la computadora de la que estamos hablando, a la que se le ha dado el identificador COMPUTADORA-1; y el ESPACIO-3, que consta de la relación específica COMPONENTES-DE-1, que tiene como objeto la PANTALLA-A-COLOR-1 que nos ocupa. Cada arco rotulado como *elemento* muestra que un estado es un elemento de otro y el arco rotulado *entidad* muestra cuál computadora específica es la entidad con la pantalla a color.

La ventaja de una red semántica particionada se demuestra con una porción de la regla de PROSPECTOR para STIR, utilizando la evidencia FMGS, como se muestra en la figura 4.23. La regla puede establecerse como "si hay una entidad llamada E-1 cuya cardinalidad es ABUNDANTE y está compuesta por ROCAS INTRUSIVAS, entonces la consecuencia es la regla de Textura que sugiere rocas ígneas". Observe que el diagrama se ha simplificado al no incluir el requisito de que el antecedente también tiene arcos [illegible]

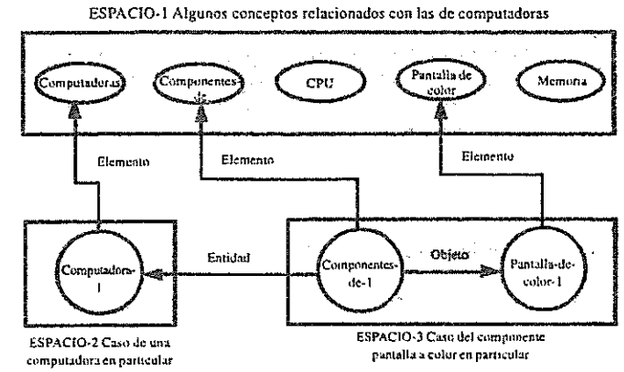


Figura 4.22 Red semántica particionada simple acerca de una computadora

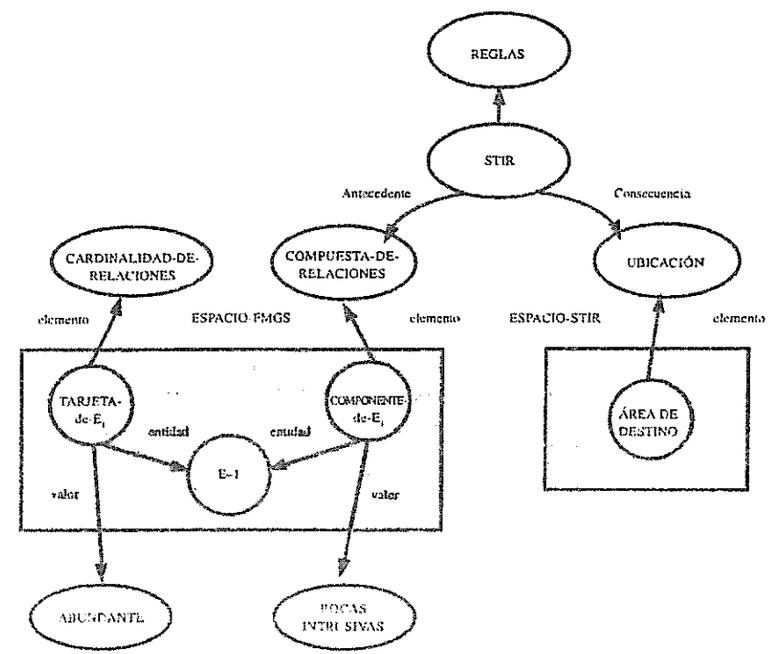


Figura 4.23
Parte simplificada de la red semántica particionada de PROSPECTOR, para la regla STIR, [illegible]

Observe en la figura 4.23 que una red semántica particionada conforma el antecedente de la regla y que una regla conforma la consecuencia. Aunque la consecuencia de esta regla es trivial, otras reglas tendrán estructuras más complejas (Duda 78). La ventaja de las redes semánticas particionadas, es que el sistema puede inferir relaciones entre nodos que están vinculados indirectamente. De allí que el conocimiento del sistema es más profundo que el conocimiento superficial.

4.16 PROPAGACIÓN DE PROBABILIDADES

Las redes de inferencia como PROSPECTOR tienen una **estructura de conocimiento estática**. Es decir, los nodos y las conexiones entre ellos están fijos, con el fin de retener las relaciones entre nodos en la estructura de conocimiento. Esto se diferencia con un sistema basado en reglas, en el cual las reglas cuyos patrones coinciden con los hechos se ponen en la agenda, se resuelven los conflictos y luego se ejecuta la regla de mayor prioridad. De allí que un sistema basado en reglas sea una **estructura de conocimiento dinámica**, porque no hay conexiones fijas entre las reglas como las hay entre los nodos de una red de inferencia.

Aunque la estructura de una red de inferencia es estática, las probabilidades asociadas con cada nodo de hipótesis cambian a medida que se obtienen evidencias. En realidad, este cambio de probabilidades es lo *único* que no cambia en la red de inferencia. La característica básica de una red de inferencia es el cambio de las probabilidades previas a posteriores, conforme se acumula la evidencia. Este cambio de probabilidades se mueve hacia arriba para finalmente sustentar o descartar la hipótesis de alto nivel, como la existencia de cobre porfirítico, PCDA.

Examinemos algunas de las etapas en la propagación de probabilidades para PCDA con base en alguna evidencia. Esto servirá como ejemplo explícito de muchas de las fórmulas que se han visto. Tomando como referencia la figura 4.19, seguiremos la propagación de probabilidades desde el nodo de evidencia la "Región contiene rocas ígneas intrusivas", con el acrónimo RCIB.

Si el usuario dice que hay rocas ígneas intrusivas presentes, entonces

$$P(E \mid e) = P(RCIB \mid e) = 1$$

De la figura 4.19, observe que todos los valores de PN para las otras evidencias de SMIR son de 1. Es decir, si el usuario no hace comentarios acerca de la evidencia para reservas, diques y cráteres volcánicos, esto no contribuye al SMIR y, de este modo, puede ignorarse en la propagación de probabilidades.

La razón para que toda esta evidencia pueda ignorarse, proviene de la fórmula para la evidencia compuesta analizada en la sección 4.14:

$$M(H \mid e) = \left[\prod_{i=1}^{N} P_i\right] M(H)$$

donde $P_i = PS_i$ para toda evidencia que se sabe que es verdadera y $P_i = PN_i$ se usa para la evidencia que se sabe es falsa. Si no hay evidencia conocida, entonces $P(E_i \mid e)$ reduce a los momios previos de la evidencia, $P(E_i)$, y así $P(H \mid E_i)$ se reduce a los momios previos $P(H)$. De allí que

$$P_i = \frac{M(H \mid e_i)}{M(H)} = \frac{M(H)}{M(H)} = 1$$

y esta P_i no contribuye a $M(H \mid E)$. Por ejemplo, cuando sólo RCIB se conoce como verdadera, la factibilidad de SMIR son:

$$M(H \mid E) = PS_{RCVP}\ PS_{RCAD}\ PS_{SMIRA}\ PS_{RCIB}$$

$$= 1 \cdot 1 \cdot 1 \cdot 1 \cdot 20$$

$$M(SMIR \mid RCIB) = 20$$

Otra característica interesante de la evidencia SMIR es el nodo O llamado SMIRA. Observe que toda la evidencia para SMIR también pasa a SMIRA. Si hay alguna evidencia que contribuye a SMIR, como RCS, RCAD, RCIB O RCVP, entonces el nodo O de SMIRA no contribuye en nada a SMIR, porque la PS para SMIRA es 1. Sin embargo, si no hay evidencia de RCS, RCAD, RCIB y RCVP, entonces el valor PN de 0.0002 de SMIRA hace que la probabilidad de SMIR sea cero. Básicamente, esto significa que aunque la ausencia de una o más evidencias de SMIR es poco importante (PS = 1), la ausencia de toda evidencia es muy importante para descartar SMIR.

La adición de nodos O, nodos Y, y valores de PS y PN permite al diseñador del modelo controlar la manera en que la evidencia afecta a una hipótesis. No obstante, en algunos casos complejos, la adición de muchos nodos O e Y para adecuarse a los requisitos de evidencia confundirá a la red de inferencia, de modo que resultará difícil que una persona la entienda. La adición de estos nodos extra también requiere que se pruebe más el modelo.

De la figura 4.20, PS = 20 para las rocas ígneas intrusivas, RCIB y P(SMIR) = 0.03. Como

$$\text{Creencia de factibilidad} = M = \frac{P}{1 - P}$$

entonces los momios previos de SMIR son:

$$M(SMIR) = \frac{0.03}{1 - 0.03} = 0.0309$$

Si la evidencia E es cierta, entonces las creencias posteriores son

$$M(H \mid E) = PS\ M(H)$$

$$M(SMIR \mid RCIB) = 20 \cdot 0.0309 = 0.618$$

y la probabilidad posterior se calcula a partir de la factibilidad fundamental de la fórmula.

$$P = \frac{M}{1 + M}$$

$$P(H \mid E) = \frac{M(H \mid E)}{1 + M(H \mid E)}$$

$$P(H \mid E) = \frac{0.618}{1 + 0.618} = 0.382$$

y así, la probabilidad posterior de SMIR, dada cierta evidencia de RCIB, es

$$P(SMIR \mid RCIB) = 0.382$$

La factibilidad previa de HYPE son:

$$M(HYPE) = \frac{0.01}{1 - 0.01} = 0.0101$$

En este punto sería tentador usar

$$M(H \mid E) = PS\ M(H)$$

$$M(HYPE \mid SMIR) = PS_{SMIR}\ M(SMIR)$$

$$= 300 \cdot \frac{0.0101}{1 + 0.0101} = 3.00$$

como las creencias de HYPE debido a SMIR, pero sería erróneo. La fórmula

$$M(H \mid E) = PS\ M(H)$$

es verdadera cuando la evidencia E es cierta. Esta fórmula expresa el cambio en la probabilidad de la hipótesis, suponiendo que la evidencia sea cierta. Sin embargo, no se sabe que SMIR sea cierta; en realidad, la probabilidad de SMIR, dadas ciertas rocas ígneas intrusivas, es

$$P(SMIR \mid RCIB) = 0.382$$

lo que significa que la posibilidad de que nuestra creencia es 38.2 por ciento de que SMIR sea verdadera.

Lo que en realidad necesitamos, es la probabilidad de la hipótesis HYPE basada en la evidencia incierta de SMIR, que es P(HYPE | RCIB). Una manera de calcularla utiliza la fórmula para P(H | e), analizada en la sección 4.14, como:

$$P(H \mid e) = \begin{cases} P(H \mid E') + \dfrac{P(H) - P(H \mid E')}{P(E)} P(E \mid e') \\ \qquad \text{para } 0 \le P(E \mid e) < P(E) \\ P(H) + \dfrac{P(H \mid E) - P(H)}{1 - P(E)} [P(E \mid e) - P(E)] \\ \qquad \text{para } P(E) \le P(E \mid e) \le 1 \end{cases}$$

donde la incertidumbre en la evidencia de SMIR es:

$$P(E \mid e) = P(SMIR \mid RCIB) = 0.382$$

$$P(H \mid e) = P(HYPE \mid SMIR)$$

donde todo lo que se conoce es:

$$P(H \mid E) = \frac{M(H \mid E)}{1 + M(H \mid E)} = \frac{3.00}{1 + 3.00} = 0.75$$

y los momios previos de la figura 4.19 son

$$P(H) = P(HYPE) = 0.01$$

$$P(E) = P(SMIR) = 0.03$$

Como P(E | e) > P(E), utilizaremos la fórmula para P(E) ≤ P(E | e) ≤ 1 para calcular P(H | E) como sigue:

$$P(H \mid e) = 0.01 + \frac{(0.75 - 0.01)(0.382 - 0.03)}{1 - 0.03}$$

$$P(HYPE \mid RCIB) = 0.279$$

que es la probabilidad posterior de HYPE.

Esta propagación de probabilidades continúa hacia arriba por la red de inferencia. Ahora la probabilidad P (HYPE | RCIB) es la evidencia incierta de FLE. La probabilidad posterior P(FLE | RCIB) se calcula utilizando la misma fórmula con valores ajustados para FLE.

$$P(E) = P(HYPE) = 0.01$$

$$P(E \mid e) = P(HYPE \mid RCIB) = 0.279$$

$$P(H) = P(FLE) = 0.005$$

$$M(H \mid E) = M(FLE \mid HYPE) = PS_{HYPE}\ M(FLE)$$

$$= 200 \cdot 0.005 = 5$$

$$P(H \mid E) = P(FLE \mid HYPE)$$

$$= M/(1 + M)$$

$$= 1/(1 + 1)$$

$$= 0.5$$

$$P(H \mid e) = P(FLE \mid RCIB)$$

$$= 1.005 + \frac{(0.5 - 0.005)(0.279 - 0.01)}{1 - 0.01}$$

4.17 RESUMEN

En este capítulo primeramente se analizaron los conceptos básicos del razonamiento en condiciones de incertidumbre y los posibles tipos de error causados por ésta; se revisaron los elementos de la teoría de la probabilidad clásica, se expusieron las diferencias entre la teoría clásica y otras teorías de probabilidad como la experimental y la subjetiva. Se cubrieron los métodos para combinar probabilidades y el teorema de Bayes, se describió la relación entre creencia y probabilidad, además del significado de la posibilidad. Se analizó el sistema experto clásico PROSPECTOR para ilustrar la manera en que se usaron estos conceptos de probabilidad en un sistema real. El sistema PROSPECTOR también se empleó para introducir redes de inferencia y redes semánticas particionadas.

PROBLEMAS

4.1 (a) Utilizando sólo los tres axiomas de probabilidad, pruebe la Ley aditiva de la probabilidad

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

(Sugerencia: cualquier evento $X \cup Y$ puede escribirse en la forma $X \cup (X' \cap Y)$ donde X y $X' \cap Y$ son mutuamente excluyentes. Además, escriba B como la unión de dos conjuntos mutuamente excluyentes.)

(b) Dadas dos computadoras que funcionan correctamente o no, ¿Cuál es la probabilidad, utilizando la Ley aditiva, de que por lo menos una funcione correctamente?

4.2 Dados los eventos A y B que pueden superponerse, derive la probabilidad en términos de conjuntos, utilizando los axiomas de probabilidad para

(a) ni A ni B
(b) A o B, pero no ambas (O-excluyente).

4.3 Tres recipientes contienen componentes que funcionan y algunos defectuosos, como se explica a continuación:

Recipiente	Bueno	Malo
1	8	2
2	3	1
3	2	2

En un periodo largo, 20 por ciento de los componentes se extraen del compartimiento 1, 30 por ciento del compartimiento 2 y 50 por ciento del compartimiento 3.

(a) Dibuje un árbol de probabilidades
(b) Si se extrae una parte defectuosa, ¿cuáles son las probabilidades de que provengan de cada recipiente? Escriba la tabla de probabilidad que muestre los resultados y las probabilidades previa, condicional, de unión y posterior.

4.4 Fred está considerando la compra de una unidad de disco y trata de decidir entre tres marcas. En la siguiente tabla se muestran sus preferencias y la probabilidad de que la unidad se descomponga en el transcurso de un año (las preferencias están en función de costo, velocidad, capacidad y confiabilidad).

Marca	Probabilidad de elección	Probabilidad de descompostura
X	0.3	0.1
Y	0.5	0.3
Z	0.2	0.6

(a) Haga una tabla de probabilidad que muestre los resultados y las probabilidades previa, condicional, de unión y posterior.
(b) ¿Cuál es la probabilidad de que todas las unidades se descomponga antes de un año?
(c) ¿Cuál es la probabilidad de que cualquier unidad se descomponga antes de un año?
(d) Para una descompostura antes de un año, ¿cuál es la probabilidad de que sean de la marca X, Y o Z?

4.5 Un chequeo es una manera barata de revisar la salud de grupos grandes de personas: una prueba más costosa pero más exacta, muestra que 1 por ciento de las personas están enfermas. El chequeo indica enfermedad (prueba positiva) en 90 por ciento de los enfermos y en 20 por ciento de quienes no están enfermos (falso positivo).

(a) Elabore una tabla que muestre los resultados y las probabilidades previa, condicional, de unión y posterior.
(b) ¿Qué porcentaje de las personas que dieron positivo no tienen la enfermedad (falso positivo)?
(c) ¿Qué porcentaje de las personas que dieron negativo sí tienen la enfermedad (falso negativo)?

4.6 Para demostrar que la independencia de un par no necesariamente significa independencia mutua, defina los siguientes eventos para el lanzamiento de un dado.

A = primer dado es par
B = segundo dado es par
C = suma par

(a) Dibuje el espacio de la muestra de los dos dados. Dibuje A, B y $A \cap B$.
(b) Escriba los elementos de $A \cap B$, C, $A \cap C$ y $B \cap C$.
(c) ¿Cuál es P(C)?
(d) Muestra la independencia de un par:

$$P(A \cap B) = P(A)\,P(B)$$
$$P(A \cap C) = P(A)\,P(C)$$
$$P(B \cap C) = P(B)\,P(C)$$

(e) Muestre

$$P(A \cap B \cap C) = P(A \cap B) \neq P(A \cap B)\,P(C)$$

lo cual prueba que la independencia de un par no significa independencia mutua.

4.7 Para los conjuntos A y B mutuamente excluyentes:

(a) ¿Qué es $P(A \mid B')$ expresada en términos de A y B (no expresado en términos de A y B')?
(b) ¿Cuál es el valor numérico de $P(A' \mid B)$?

(d) ¿Qué es $P(A' \mid B')$ expresado en términos de A y B (no expresado en términos de A´ y B´)?
(e) ¿Qué es?:

(i) $P(A \mid B) + P(A' \mid B)$
(ii) $P(A \mid B') + P(A' \mid B')$

4.8 (a) Pruebe que:

$$P(A \cap B \cap C) = P(A \mid B \cap C)\, P(B \mid C)\, P(C)$$

(b) Pruebe que:

$$P(A \cap B \mid C) = P(A \mid B \cap C)\, P(B \mid C)$$

4.9 Una unidad de disco puede funcionar mal con la falla F_1 o F_2, pero no ambas. Los síntomas posibles son:

A = { escribe mal | lee mal }

B = { escribe mal }

y F_2 tiene tres veces más probabilidades de ocurrir que F_1. Para este tipo de unidad

$P(A \mid F_1) = 0.4 \quad P(B \mid F_1) = 0.6$

$P(A \mid F_2) = 0.2 \quad P(B \mid F_2) = 0.8$

¿Cuáles son las probabilidades de que una unidad de disco tenga la falla F_1? y ¿F_2?

4.10 Dada la matriz de transición para cambio de marcas de unidades de disco en un año:

Año siguiente

$$\text{Este año} \quad \begin{array}{c} \\ X \\ Y \\ Z \end{array} \begin{array}{c} \begin{array}{ccc} X & Y & Z \end{array} \\ \begin{bmatrix} .5 & .5 & 0 \\ .25 & .5 & .25 \\ 0 & .5 & .5 \end{bmatrix} \end{array}$$

(a) Suponiendo que inicialmente 50 por ciento de las personas utiliza la unidad X, 25 por ciento utiliza la unidad Y y 25 por ciento utiliza la unidad Z, ¿qué porcentaje estará usando cada unidad en 1, 2 y 3 años?
(b) Determine la matriz de estado constante.

4.11 (a) Dadas N personas seleccionadas al azar, ¿cuál es la probabilidad de que dos no tengan la misma fecha de cumpleaños?
(b) ¿Cuál es la probabilidad para 30 personas?

4.12 Bajo la independencia condicional de E_1 y E_2, demuestre que para una regla relacionada con la conjunción de evidencia conocida

que

(a)

$$P(H \mid E_1 \cap E_2) = \frac{P(E_1 \mid H)\, P(E_2 \mid H)}{P(E_1 \mid H)\, P(E_2 \mid H) + M(H')\, P(E_1 \mid H')\, P(E_2 \mid H')}$$

(b)

$$P(H \mid E_1 \cap E_2) = \frac{P(H \mid E_1)\, P(H \mid E_2)}{P(H \mid E_1)\, P(H \mid E_2) + M(H)\, P(H' \mid E_1)\, P(H' \mid E_2)}$$

4.13 Dada una regla con una disyunción de evidencia:

SI E_1 O E_2 O ... E_N ENTONCES H

Demuestre que sobre la base de la teoría de la probabilidad y suponiendo independencia condicional de la evidencia

$$P(E \mid e) = 1 - \prod_{i=1}^{N} (1 - P(E_i \mid e))$$

donde E es la evidencia y e es las observaciones relevantes para E.

4.14 Dada la siguiente evidencia como el antecedente de:

SI E ENTONCES H

escriba las expresiones de lógica de confusión para $P(E \mid e)$.

(a) $E = E_1$ O $(E_2$ Y $E_3)$
(b) $E = (E_1$ Y $E_2)$ O $(E_1$ Y $E_4)$
(c) $E = (E_1'$ Y $E_2')$ O E_3
(d) $E = E_1'$ Y $(E_2'$ O $E_3)$
(e) $E = E_1'$ O $(E_2$ Y $E_3)$

4.15 Para la red de inferencia del cobre porfírico, suponga que se conoce que RCS es verdadera y que RCAD es falsa. ¿Cuál sería la probabilidad de FKE?

BIBLIOGRAFÍA

(Allen 81), J. F. Allen, "An Internal-based Representation of Temporal Knowledge", *Proc. 7th IJCAI*, pp. 221-226, 1981.

(Duda 78). Richard 0. Duda, *et al* Semantic Network Representations in Rule-based Inference Systems", en *Pattern-directed Inference* Systems, D. A. Waterman and Frederick Hayes-Roth. eds., Academic Press, pp. 203-221, 1978.

(Duda 79). Richard Duda, Hohn Gaschnig, y Peter Hart, "Model Design in the PROSPECTOR Consultant System for Mineral Exploration", en *Expert Systems in the Micro-electronic Age,* Donald Michie, ed., Edinburgh University Press, pp. 153-167, 1979.

(Duda 84). Richard 0. Duda y René Rebob, "AI and Decision Making: The PROSPECTOR Experience", en *Artificial Intelligence for Business,* ed., Walter Reitman, Ablex Pub. Corp, p. 136, 1984.

(Fagan 79). L. M. Fagan, J. C. Kunz, E. A. Feigenbaum, y J. J. Osborn, "Representation of Dynamic Clinical Knowledge: Measurement Interpretation in the Intensive Care Unit", en *Proc. 6th IJCAI,* pp. 260-262, 1979.

(Farley 83). Arthur M. Farley, "A Probabilistic Model for Uncertain Problem Solving", en *IEEE Transactions on Systems, Man, and Cybernetics,* **SMC-13,** (4), pp. 568-579, julio/agosto de 1983.

(Glymour *85).* Clark Glymour, "Independence Assumptions and Bayesian Updating", en *Artificial Intelligence,* 25, pp. 95-99, 1985.

(Gorry 78). G. A. Gorry, H. Silverman, y S. G. Pauker, "Capturing Clinical Expertise: A Computer Program That Considers Clinical Responses to Digitalis", en *Amer. J. Med.,* **64,** pp. 452-460, 1978.

(Hartley 28). R. V. L. Hartley, "Transmission of Information", en *The Bell System 'Iechnical Journal,* 7, pp. 535-563, 1928,

(Hendrix 79). Gary Hendrix, "Encoding Knowledge in Partitioned Networks", en *Associative Networks,* N. Findler, ed., Academic Press, pp. 51-92, 1979.

(Hink 87). Robert F. Hink y David L. Woods, "How Humans Process Uncertain Knowledge: An Introduction for Knowledge Engineers", en *AI Magazine,* **8,** (3), pp. 41-53, otoño de 1987,

(Kemeny 59). John G. Kemeny, *et al., Finite Mathematical Structures,* Prentice-Hall, p. 399, 1959.

(Lapin 78). Lawrence Lapin, *Statistics for Modern Business Decisions,* 2a ed., I Harcourt Brace Jovanovich, p. 704, 1978.

(Lindley 65). D. V. Lindley, *Introduction to Probability and Statistics front a Bayesian Viewpoint, Part I, Probability,* Cambridge University Press, p. xi, 1965.

(Manna y Wolper 84). Z. Manna y P. Wolper, "Synthesis of Communicating Processes from Temporal Logic", *ACM Trans. on Programming Languages and Systems, 6,* (1), pp. 68-93, 1984.

(McDermott 82). D. McDermott, "A Temporal Logic for Reasoning about Plans and Actions", en *Cognitive Science,* **6,** pp. 101-155, 1982.

(Parrat 86). Lyman G. Parratt, *Probability and Experimental Errors in Science,* John Wiley and Sons, Inc., p. 6, 1986.

(Pearl 82). Judea Pearl, "Reverend Bayes on Inference Engines: A Distributed Hierarchical Approach", en *Proceedings AAAI,* 2, pp. 133-136, 1982,

(Pednault 81). E. D. P. Pednault, S. W. Zucker, y L. V. Muresan, "On the Independence Assumption Underlying Subjective Bayesian Inference", en *Artificial Intelligence* , **16,** pp. 213-222, 1981.

(Quinlan 86). J. Ross Quinlan, "INFERNO: A Cautious Approach to Uncertain Inference", en *Expert Systems*, Phillip Klahr y Donald Waterman, eds., pp. 350-390, 1986.

(Savage 54). L. J. Savage, *The Foundations of Statistics,* John Wiley and Sons, Inc., 1954.

(Shafer 76). Glenn Shafer, *A Mathematical Theory of Evidence,* Princeton University Press, p. 10, 1976.

(Shannon 48). Claude Shannon, "The Mathematical Theory of Communication", en *The Bell System Technical Journal* **27,** pp. 379-423; 623-656, 1948.

(Sioshansi 83). F. Perry Sioshansi, "Subjective Evaluation Using Expert Judgement: An Application", en *IEEE Transactions on Systems, Man, and Cybernetics,* **SMC-13, (3),** pp. 391-397, mayo/junio 1983.

(Stebbing 50). L.S. Stebbing, *A Modern Introduction to Logic,* Londres, 7a ed., Methuen & Co. Ltd., pp. 104-105, 1950.

(Szolovits 78). P. 5. Szolovits y S. G. Pauker, "Categorical and Probabilistic Reasoning in Medical Diagnosis", en *Artificial Intelligence* **11,** pp. 115-144, 1978.

(Turner 84). Raymond Turner, *Logics for Artificial Intelligence,* chapter 6, Ellis Horwood, Ltd., 1984.

(Weiss 78). S. Weiss, C. Kulikowski, S. Amarel, y A. Safir, "A Model-Based Method for Computer-Aided Medical Decision Making", en *Artificial Intelligence* **11,** pp. 145-172, 1978.

(Zadeh 65). Lotfi A. Zadeh, "Fuzzy Sets", *Information and Control,* 8, pp. 338-353, junio de 1965.

CAPÍTULO 5

Razonamiento inexacto

5.1 INTRODUCCIÓN

En este capítulo se continúa con la exposición del razonamiento en condiciones de incertidumbre iniciada en el capítulo 4, donde los paradigmas principales para el razonamiento en estas condiciones fueron el razonamiento probabilístico y el teorema de Bayes. En este capítulo se examinan otros métodos para tratar con la incertidumbre.

La teoría de la probabilidad puede considerarse como una teoría de **incertidumbre reproducible.** Como se describe en el capítulo 4, la probabilidad se desarrolló originalmente para juegos de azar ideales en donde el mismo experimento se puede reproducir de manera indefinida. La teoría subjetiva de probabilidad descrita en el capítulo anterior se ha utilizado con éxito en el sistema PROSPECTOR, pero hay muchas otras aplicaciones en las que son más útiles otras teorías; estas alternativas se desarrollaron de manera específica para tratar con las creencias humanas más que con la interpretación clásica de la frecuencia de la probabilidad; todas ellas son ejemplos del **razonamiento inexacto,** en cual el antecedente, la conclusión, e incluso el significado de la regla son inciertos en alguna medida.

5.2 INCERTIDUMBRE Y REGLAS

En esta sección se hace un reconocimiento general de las reglas y la incertidumbre. En secciones posteriores se analizan los métodos específicos para tratar con la incertidumbre en las reglas.

Fuentes de la incertidumbre en reglas

En la figura 5.1 se muestra una vista de alto nivel de la incertidumbre para los sistemas basados en reglas, ésta puede surgir de reglas individuales, resolución del conflicto e incompatibilidades entre las consecuencias de las reglas. El objetivo del ingeniero en conocimiento es reducir lo más posible o eliminar estas incertidumbres.

La reducción de la incertidumbre de las reglas individuales es parte de la verificación de la regla. Como se mencionó en la sección 3.15, la verificación tiene que ver con lo correcto que sean los bloques de construcción del sistema y en un sistema basado en reglas, los bloques de construcción son las propias reglas.

El hecho de que las reglas individuales sean correctas no significa que el sistema proporcionará una respuesta correcta, pues a causa de las incompatibilidades entre ellas, las cadenas de inferencia pueden ser incorrectas y, por tanto, es necesaria la validación. En un sistema basado en reglas, parte de la validación consiste en minimizar la incertidumbre de las cadenas

de inferencia; si la verificación puede verse como la disminución de incertidumbres locales, la validación disminuye la incertidumbre global de todo el sistema experto. Como una analogía burda con la ingeniería civil, la verificación preguntaría si un puente se está construyendo de manera apropiada, considerando esto como buenos materiales y montaje; mientras tanto, la validación cuestionaría si el puente puede controlar la carga de tráfico necesaria y, lo más importante, ¿Se está construyendo en el lugar correcto? Tanto la verificación como la validación son necesarias para obtener con seguridad un sistema experto de calidad, como se analiza en el capítulo 6.

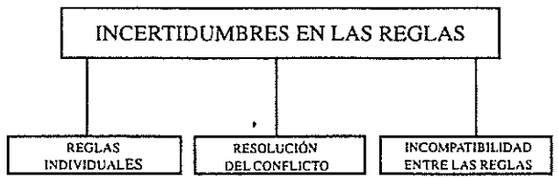


Figura 5.1 Principales incertidumbres de un sistema experto basado en reglas

La vista del nivel superior de las incertidumbres para las reglas individuales de la figura 5.1 se encuentra más detallada en la figura 5.2. Además de los posibles errores derivados de la creación de reglas, como se describe en la sección 4.3, hay incertidumbres asociadas con la asignación de valores de posibilidad. En el razonamiento probabilístico, como se explica en el capítulo anterior, estas incertidumbres están con los valores de suficiencia, PS, y necesidad, PN, y puesto que estos valores se basan en estimaciones de seres humanos, hay incertidumbre en ellos. También existe incertidumbre con la posibilidad de la consecuencia; para efectos de razonamiento probabilístico, esto se describe como P(H | E) para evidencia cierta y P(H | e) para evidencia incierta.

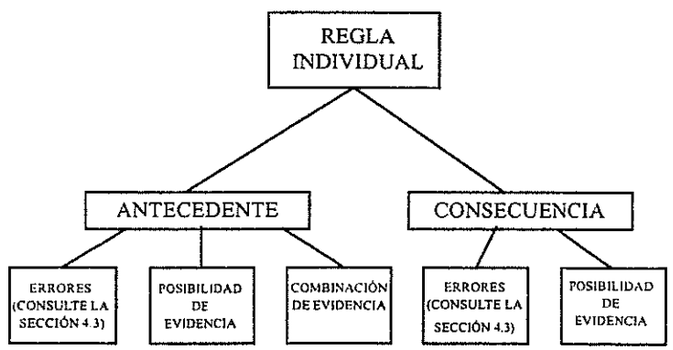


Figura 5.2 Incertidumbres de reglas individuales

Otra fuente de incertidumbre es la combinación de la evidencia. ¿La evidencia se puede combinar

como E_1 Y E_2 Y E_3

o como E_1 Y E_2 O E_3

o como E_1 Y NO E_2 O E_3

o en alguna otra forma posible de combinación lógica utilizando Y, O y NO?

Falta de teoría

Como se menciona en el capítulo 4, la introducción de fórmulas *ad hoc*, como la lógica borrosa en un sistema probabilístico introduce un problema, el sistema experto entonces no tiene una base teórica sólida basada en la probabilidad clásica y, por tanto, representa un método *ad hoc* que funciona para un caso limitado. El peligro de estos métodos es que no existe una teoría completa que dirija la aplicación o advierta situaciones inapropiadas.

Otro ejemplo de un método *ad hoc* que ha visto es el uso de los factores PS y PN con redes de inferencia. En teoría, una red de inferencia con N nodos es un sistema probabilístico con un espacio de evento de N eventos posibles, de esta manera, hay 2^N probabilidades posibles, pero en la práctica se conocen pocas de estas probabilidades. En lugar de tratar de enumerar todas las probabilidades, los factores PS y PN se utilizan para arcos significativos (esperamos) entre los nodos, lo que reduce en gran medida la complejidad de la red.

Sin embargo, el uso de los métodos abreviados de PS y PN también tienen el efecto nocivo de que no hay garantía teórica de que la suma de las probabilidades condicionales para cada hipótesis de la red será la unidad. Tal vez esto no revista gran importancia para el usuario si su objetivo es simplemente obtener una clasificación relativa de todas las hipótesis de la red. Por ejemplo, tal vez el usuario interesado en la hipótesis del cobre porfírico se sentiría satisfecho al saber que el nodo que indica una posibilidad favorable de cobre es la más grande, aunque no sea igual a 1, sin embargo, la probabilidad absoluta también es importante. A pesar de que la hipótesis del cobre está clasificada como número 1, ¿vale la pena perforar si la probabilidad de encontrar cobre es muy baja, digamos 0.00000001?

Interacción de reglas

Otra fuente de incertidumbre es la resolución de conflictos de incertidumbre. Si el ingeniero en conocimiento especifica una **prioridad explícita** de las reglas, hay una fuente potencial de error debida a que la prioridad podría no ser óptima o correcta. Otros problemas con la resolución de conflictos ocurre debido a la **prioridad implícita** de reglas.

La resolución de conflictos es parte de una fuente importante de la incertidumbre causada por las interacciones entre reglas, pues éstas dependen de la resolución de conflictos y de la **compatibilidad de las reglas,** como se muestra en la figura 5.3. Dicha compatibilidad acarrea incertidumbre debido a cinco causas principales.

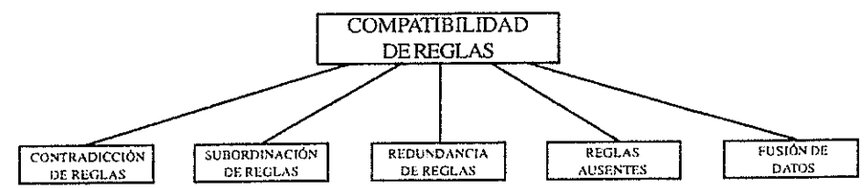


Figura 5.3 Incertidumbre asociada con la compatibilidad de reglas

Una causa de incertidumbre es la posible **contradicción de reglas**, porque varias reglas pueden dispararse con consecuencias contradictorias, lo que puede ocurrir si los antecedentes no se especifican de manera apropiada. Como ejemplo muy sencillo, supongamos dos reglas en la base de conocimiento:

```
(1) SI hay fuego ENTONCES arrojarle agua
(2) SI hay fuego ENTONCES no arrojarle agua
```

La regla (1) aplica para fuegos ordinarios como incendios de madera, en tanto que la regla (2) se aplica a fuego por petróleo o grasas; el problema es que no se especifican los antecedentes con la suficiente precisión para cada tipo de fuego. Si existe un hecho "hay fuego", entonces ambas reglas se ejecutarán con consecuencias contradictorias y con un aumento resultante de la incertidumbre.

Una segunda fuente de incertidumbre es la **subordinación de reglas.** Una regla está subordinada a otra si una parte de su antecedente es un subconjunto de otra regla. Por ejemplo, suponga que hay dos reglas que tienen la misma conclusión:

(3) SI E_1 ENTONCES H

(4) SI E_1 Y E_2 ENTONCES H

Si sólo está presente E_1, entonces no hay problema porque sólo se activará la regla (3), sin embargo, si existe tanto E_1 como E_2, entonces se activarán las reglas (3) y (4) y será necesaria la resolución de conflicto entre ellas.

Resolución de conflictos

Hay incertidumbre en la resolución de conflictos con respecto a la prioridad para disparo, esta incertidumbre depende de varios factores, como se ilustra en la figura 5.4.

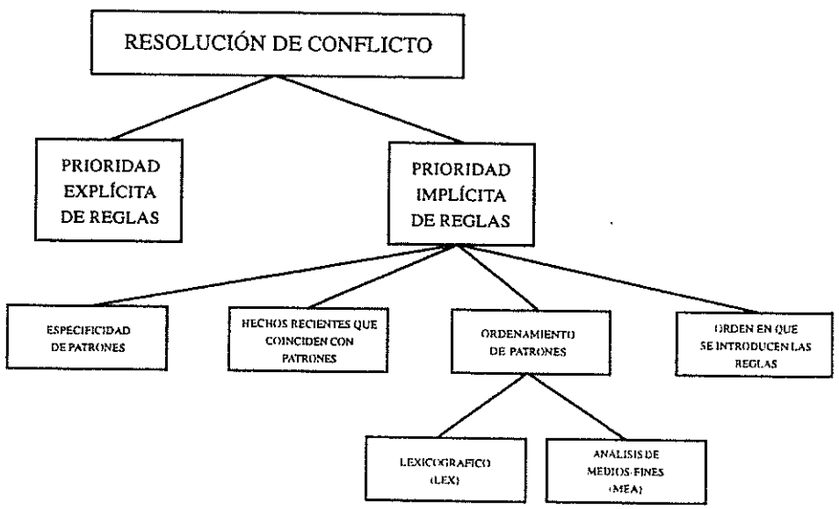


Figura 5.4 Incertidumbre asociada con la resolución de conflictos

El primer factor es el shell o herramienta que se utiliza. En el sistema OPS5, una regla con patrones más específicos tiene mayor prioridad. La **especificidad** de una regla OPS5 depende del número de patrones y la complejidad interna de cada uno. Por ejemplo, un patrón


```
(dirigible elipsoidal)
```

es más específico que un patrón:

```
(dirigible)
```

Para nuestras dos reglas, la (4) es más específica porque tiene dos patrones en su antecedente y, por tanto, tiene una prioridad implícita más alta. En un sistema OPS5, la regla (4) se ejecuta primero.

Sin embargo, hay otras complicaciones en OPS5. Además de la especificidad, también se encuentra la consideración de **qué tan recientes son los hechos.** Cada vez que se introduce un hecho en la memoria de trabajo, ésta recibe una **etiqueta de tiempo** que indica cuándo se introdujo. Así, una regla con el patrón:

(5) SI E_3 ENTONCES H

tendría una prioridad más alta que la regla (3), si se introdujo el hecho E_3 después del E_1.

El ordenamiento de patrones dentro de una regla también produce diferencias. OPS5 permite dos diferentes **estrategias de control** llamadas **lexicográfico (LEX) y análisis de medios-fines (MEA).** Estas estrategias determinan la manera en que el mecanismo de inferencia interpreta los patrones de regla. Una estrategia de control se selecciona mediante la introducción de un comando de estrategia en el intérprete del sistema experto. En LEX, el orden del patrón no produce ninguna diferencia, excepto tal vez en eficiencia. De esta manera, utilizando LEX las siguientes dos reglas son esencialmente lo mismo:

SI E_1 Y E_2 ENTONCES H

SI E_2 Y E_1 ENTONCES H

MEA es la estrategia que utiliza el Solucionador general de problemas de Newell y Simon mencionado en el capítulo 1. La idea básica de MEA es tratar de reducir la diferencia entre el estado de inicio y el estado de éxito. En la implantación de MEA en el sistema OPS5, el primer patrón es muy importante porque actúa para controlar el proceso de comparación de patrones. El propósito de MEA es ayudar al sistema a mantener la ejecución de cierta tarea sin distraerse con hechos muy recientes que han entrado en la memoria de trabajo. MEA da prioridad a las reglas con base en el primer patrón, que sigue al SI. Una regla se selecciona si predomina sobre las otras con base en el primer patrón. Este predominio puede darse por especificidad o por ser más reciente. Si hay más de un patrón dominante de primera regla, entonces MEA or-dena las reglas con base en los patrones restantes. A la postre, si no hay regla dominante, se elige arbitrariamente una de ellas.

El orden en el que se introducen las reglas en el sistema experto también podría ser un factor en la resolución de conflictos. Si el mecanismo de inferencia no puede asignar prioridades a las reglas, debe hacerse una elección arbitraria. No obstante, a menos que los diseñadores del mecanismo de inferencia hayan sido cuidadosos, la elección podría ser determinística en cierto sentido porque depende de la entrada de la regla. Esto puede ocurrir debido a que los diseñadores del mecanismo de inferencia utilizaron una pila o fila para almacenar reglas en la agenda. Para una elección arbitraria de reglas con prioridades iguales, en realidad habría una elección al azar de la pila o fila hecha. En cambio, los diseñadores pueden tomar la salida más fácil y deshacer la pila o tomar la siguiente regla de la fila. Este método de selección es fácil pero introduce un artefacto conocido en el sistema porque la elección de reglas con igual prioridad no es arbitraria.

Subordinación e incertidumbre

El problema de la subordinación se vuelve más incierto si hay posibilidades asociadas con las reglas, como en

(6) SI E_1 ENTONCES H con PS_1

(7) SI E_1 Y E_2 ENTONCES H con PS_2

Por ejemplo, consideremos las siguientes reglas:

```
SI la marcha no arranca
ENTONCES revisar la batería con PS1 = 5
SI la marcha no arranca Y
   las luces no encienden
ENTONCES revisar la batería con PS2 = 10
```

donde PS_2 es más grande que PS_1 porque tiene más evidencia sustentando la conclusión.

En el capítulo 4, la fórmula posterior de factibilidad

$$(8)\quad M(H \mid E) = \left[\prod_{i=1}^{N} LS_i\right] M(H)$$

significa que el producto es $5 \cdot 10 = 50$, si se observan E_1 y E_2.

Sin embargo, (8) en realidad tiene el propósito de describir reglas cuya evidencia respectiva se combina de manera independiente. Los antecedentes de (6) y (7) en realidad no son independientes porque comparten la evidencia E_1. Así, E_1 y E_2 subordina a E_1 porque ésta por sí solo es un caso especial de E_1 Y E_2. El producto de las relaciones de posibilidad PS_1 y PS_2 es, por tanto, erróneamente elevado porque los antecedentes no son independientes. Una solución consiste en remplazar PS_2 por la relación PS_2/PS_1 de modo que cuando existen E_1 y E_2, se calcula el producto correcto PS_2. Puede ser muy difícil detectar manualmente una subordinación en una base de conocimiento grande.

Una tercera causa de incertidumbre son las **reglas redundantes** que tienen igual consecuencia y evidencia. Por lo general, el ingeniero en conocimiento las introduce inadvertidamente u ocurren por accidente cuando se modifica una regla por eliminación de patrón. Por ejemplo:

SI E_1 Y E_2 ENTONCES H

SI E_2 Y E_1 Y E_3 ENTONCES H

son reglas redundantes si se elimina E_3, porque los antecedentes son el mismo patrón. No siempre es sencillo decidir cuál regla redundante eliminar. Una regla redundante puede conducir a una mayor eficiencia del sistema si en la lista aparece primero un patrón poco frecuente. Tal vez el mecanismo de inferencia no necesite revisar el segundo patrón en busca de una coincidencia, si el primero no coincide. Una complicación más se origina en los shells de los sistemas expertos como el OPS5, donde el orden de activación depende de la estrategia de control seleccionada por el usuario. Como en el caso de la subordinación, las reglas redundantes también generan un producto erróneamente alto de la relación de posibilidad.

La cuarta causa de incertidumbre en las consecuencias de las reglas surge de las **reglas ausentes**, que ocurren si el experto humano olvida o ignora una regla como:

SI E_4 ENTONCES H

Si se ignora la evidencia E_4, entonces H no se concluye con la fuerza que merece. Una ventaja de una red de inferencia como PROSPECTOR es su naturaleza explícita, que facilita la identificación de hipótesis difíciles o imposibles de alcanzar, por lo que si hay reglas ausentes, la red de inferencia puede suponer la necesidad de esas reglas.

La quinta causa de incertidumbre ocurre por los problemas con la **fusión de datos.** Este término se refiere a la incertidumbre asociada con la fusión de datos de diferentes tipos de información. Por ejemplo, al hacer un diagnóstico, un médico consideraría evidencia de orígenes muy diferentes como un examen físico, pruebas de laboratorio, historial del paciente, ambiente socioeconómico, estado mental y emocional, problemas familiares y laborales, etc; todos éstos son diferentes tipos de evidencia que deben fusionarse para sustentar una hipótesis final. La fusión de esta evidencia de tantos orígenes diferentes es mucho más difícil que la evidencia en un dominio, como la geología.

Del mismo modo, una decisión de negocios quizá dependa del mercado de bienes, condiciones económicas, comercio internacional, tentativas de absorción, políticas de la empresa, problemas personales, sindicatos y muchos otros factores y como en el problema médico, es muy difícil asignar relaciones de posibilidad para todos estos factores y determinar una función de combinación razonable.

5.3 FACTORES DE CERTIDUMBRE

Otro método para tratar con la incertidumbre utiliza **factores de certidumbre,** originalmente desarrollados para el sistema experto MYCIN.

Dificultades con el método bayesiano

Al igual que el problema geológico tratado con PROSPECTOR, el diagnóstico médico casi siempre está sujeto a incertidumbre. La principal diferencia es que hay un número limitado de hipótesis geológicas acerca de los minerales porque sólo hay 92 elementos naturales en tanto que las hipótesis de una posible muerte son mucho más grandes porque hay muchos microorganismos más.

Aunque el teorema de Bayes es útil en la medicina, su uso preciso depende del conocimiento de muchas probabilidades. Por ejemplo, el teorema puede utilizarse para determinar la probabilidad de una enfermedad específica, D, dados ciertos síntomas como:

$$P(D_i \mid E) = \frac{P(E \mid D_i)\,P(D_i)}{P(E)} = \frac{P(E \mid D_i)\,P(D_i)}{\sum_j P(E \mid D_j)\,P(D_j)}$$

donde la suma de j se extiende para todas las enfermedades, y

D_i es la i-ésima enfermedad,
E es la evidencia,
P(E) es la probabilidad previa de que el paciente tenga la enfermedad antes de que se conozca cualquier evidencia,
$P(E \mid D_i)$ es la probabilidad condicional de que el paciente mostrará la evidencia E, dado que tiene la enfermedad D_i

A menudo es imposible determinar valores compatibles y completos de todas estas probabilidades para la población general.

En la práctica, la evidencia tiende a acumularse pieza por pieza, lo que requiere de mucho tiempo y dinero, en especial cuando requiere pruebas médicas. Estos factores y el riesgo potencial de la prueba para el paciente, a menudo limitan el número de pruebas realizadas a un mínimo requerido para un buen diagnóstico (y protección contra la negligencia).

La siguiente es una forma del teorema de Bayes que expresa la acumulación de evidencia incremental como ésta (consúltese 5.1)

$$P(D_i \mid E) = \frac{P(E_2 \mid D_i \cap E_1)\,P(D_i \mid E_1)}{\sum_j P(E_2 \mid D_j \cap E_1)\,P(D_j \mid E_1)}$$

donde E_2 es la nueva evidencia que se agregó al cuerpo de evidencia existente, E_1, para producir la nueva evidencia.

$$E = E_1 \cap E_2$$

Aunque esta fórmula es exacta, por lo general no se conocen todas estas probabilidades, además la situación empeora conforme se acumulan más piezas de evidencia y, por tanto, se necesitan más probabilidades.

Creencia e incredulidad

Además del problema de acumulación de todas las probabilidades condicionales para el método bayesiano, otro problema importante que apareció con los especialistas médicos fue la relación entre creencia e incredulidad. A primera vista, esto parecería trivial porque obviamente, incredulidad simplemente es lo opuesto de creencia. En realidad, la teoría de la probabilidad establece que:

$$P(H) + P(H') = 1$$

y por tanto:

$$P(H) = 1 - P(H')$$

Para el caso de una hipótesis posterior que depende de la evidencia, E:

$$(1)\quad P(H \mid E) = 1 - P(H' \mid E)$$

Sin embargo, cuando los ingenieros en conocimiento de MYCIN empezaron a entrevistar especialistas médicos, encontraron que los médicos eran muy renuentes a establecer su conocimiento en forma de ecuación (1).

Por ejemplo, imaginemos una regla de MYCIN como la siguiente (Shortliffe 85):

```
SI   1) El teñido gram del microorganismo es positivo, y
     2) La morfología del microorganismo es coco, y
     3) La forma de crecimiento del microorganismo es con
        cadena
ENTONCES Hay evidencia sugestiva (0.7) de que
```


En términos simples, esta regla indica que si un microorganismo bacterial se coloreó cuando se trató con colorante de Gram y tomó un aspecto de cadenas de esferas, entonces hay un 70% de posibilidades de que sea del tipo llamado *estreptococo*. Esto puede escribirse en términos de probabilidad posterior como

$$(2)\quad P(H \mid E_1 \cap E_2 \cap E_3) = 0.7$$

donde E_i corresponde a los tres patrones del antecedente.

Los ingenieros en conocimiento de MYCIN encontraron que aunque los especialistas aprobarían la ecuación (2), se preocuparían y rehusarían estar de acuerdo con el resultado probabilístico

$$(3)\quad P(H' \mid E_1 \cap E_2 \cap E_3) = 1 - 0.7 = 0.3$$

Esta renuencia del especialista a estar de acuerdo con la ecuación (3) demuestra una vez más que números como 0.7 y 0.3 son posibilidades de creencia, no probabilidades.

Con el fin de entender por completo el problema de creencia e incredulidad contradictorias, consideremos el siguiente ejemplo: supongamos que éste es el último curso que necesita para obtener su título, que su promedio de calificaciones no ha sido bueno y que necesita una A en este curso para subirlo. La siguiente fórmula expresaría su creencia en la probabilidad de graduarse:

$$(4)\quad P(\text{graduación} \mid \text{A en este curso}) = 0.70$$

Observe que esta probabilidad no es del 100%. La razón es que la escuela hará la revisión final de sus calificaciones y curso; es decir, podrían surgir problemas debido a varias razones que aún evitarían que obtuviera su título, como las siguientes:

1) El catálogo de la escuela cambia, de modo que no se cuentan todos sus cursos para la titulación.
2) Olvidó tomar un curso obligatorio.
3) Rechazo de transferencia de cursos.
4) Rechazo de algunos de los cursos optativos que eligió.
5) Multas de matrícula y biblioteca que debe y que esperaba se olvidarían.
6) Su promedio era más bajo de lo que pensaba y una A aún no lo aumentaría.
7) "Ellos" no están para ayudarlo.

Suponiendo que está de acuerdo con (4) (o quizá su propio valor para la posibilidad), entonces mediante la ecuación (1):

$$(5)\quad P(\text{no hay graduación} \mid \text{A en este curso}) = 0.30$$

Aunque la ecuación (5) es correcta desde un punto de vista probabilístico, de algún modo parece incorrecta por intuición. No es correcto que si en realidad trabaja duro y obtiene una A en este curso, entonces haya una oportunidad de 30 por ciento de que no se gradúe. La ecuación (5) le parecería tan incómoda como al experto médico que cree que:

$$P(H \mid E_1 \cap E_2 \cap E_3) = 0.70$$

pero que no en la consecuencia probabilística:

$$P(H' \mid E_1 \cap E_2 \cap E_3) = 0.30$$

El problema fundamental es que mientras P(H | E) implica una relación de causa y efecto entre E y H, tal vez no existe esta relación entre E y H . Aún la ecuación:

$$P(H \mid E) = 1 - P(H' \mid E)$$

implica una relación de causa y efecto entre E y H′, si hay una entre E y H.

Estos problemas con la teoría de la probabilidad llevaron a Shortliffe a investigar otras formas de representación de la incertidumbre. El método que utilizó con MYCIN se basó en los **factores de certidumbre** derivados de la teoría de la confirmación de Carnap (Carnap 1950), que distinguió dos tipos de probabilidad.

Un tipo es la probabilidad ordinaria asociada con la frecuencia de eventos reproducibles. Al segundo tipo se le llama ***probabilidad epistémica*** o ***grado de confirmación***, porque confirma una hipótesis que se basa en alguna evidencia. Este segundo es otro ejemplo del grado de posibilidad de una creencia.

Medición de creencia e incredulidad

En MYCIN, el grado de confirmación se definió originalmente como el factor de certidumbre, que es la diferencia entre creencia e incredulidad:

$$FC(H, E) = MC(H, E) - MI(H, E)$$

donde

FC es el factor de certidumbre en la hipótesis H debido a la evidencia E
MC es la **medida de creencia incrementada** en H debido a E
MI es la **medida de escepticismo incrementado** en H debido a E

El factor de certidumbre es una manera de combinar creencia y escepticismo en un número único.

La combinación de medidas de creencia y escepticismo en un número único tiene dos usos: en primer lugar, el factor de certidumbre puede utilizarse para clasificar hipótesis por orden de importancia. Por ejemplo, si un paciente tiene ciertos síntomas que sugieren varias enfermedades posibles, entonces la enfermedad con el FC más alto sería la primera en investigarse mediante el ordenamiento de pruebas.

Las medidas de creencia e incredulidad se definieron en términos de probabilidades a través de:

$$MC(H, E) = \begin{cases} 1 & \text{si } P(H) = 1 \\ \dfrac{m\,x[P(H \mid E),\ P(H)] - P(H)}{\text{máx}[1, 0] - P(H)} & \text{de otro modo} \end{cases}$$

$$MI(H, E) = \begin{cases} 1 & \text{si } P(H) = 0 \\ \dfrac{m\,n[P(H \mid E),\ P(H)] - P(H)}{\text{mín}[1, 0] - P(H)} & \text{de otro modo} \end{cases}$$

Ahora máx[1,0] siempre es 1 y mín[1,0] siempre es 0. El motivo para escribir 1 y 0 en términos de máx y mín es para mostrar la simetría formal entre MC y MI. Las ecuaciones para MC y MI sólo son diferentes en el reemplazo de máx por mín, y viceversa.

De acuerdo con estas definiciones, se muestran algunas características en la tabla 5.1.

Características	Valores
Rangos	$0 \le MC \le 1$ $0 \le MI \le 1$ $-1 \le FC \le 1$
Ciertas Hipótesis verdaderas P(H | E) = 1	MC = 1 MI = 0 FC = 1
Ciertas hipótesis falsas? P(H′ | E) = 1	MC = 0 MI = 1 FC= –1
Falta de evidencia P(H | E) = P(H)	MC = 0 MI= 0 FC= 0

Tabla 5.1 Algunas características de MC, MI y FC

El factor de certidumbre, FC, indica la creencia neta en una hipótesis basada en alguna evidencia. Un FC positivo significa que la evidencia sustenta la hipótesis, porque MC > MI. FC = 1 significa que la evidencia prueba en definitiva a la hipótesis. FC = 0 significa una de dos posibilidades. Primera, un FC = MC – MI = 0 podría significar que tanto MC como MI son 0, es decir, no hay evidencia. La segunda posibilidad es que MC = MI y ambas sean diferentes de 0. El resultado es que la incredulidad cancela a la creencia.

Un FC negativo significa que la evidencia favorece la negación de la hipótesis debido a que MC < MI. Otra manera de establecer esto es que hay más razones para no creer en una hipótesis que para creerla. Por ejemplo, un FC = –70 por ciento significa que la escepticismo es un 70% mayor que la creencia. Un FC = 70 por ciento significa que la certidumbre es un 70% más grande que el escepticismo. Obsérvese que con factores de certidumbre no hay restricciones en los valores individuales de MC y MI, sólo es importante la diferencia. Por ejemplo:

$$\begin{aligned} FC &= 0.70 = 0.70 - 0 \\ &= 0.80 - 0.10 \end{aligned}$$

y así sucesivamente.

Los factores de certidumbre le permiten a un experto expresar una creencia sin otorgar un valor a la incredulidad. Como se muestra en el problema 5.2:

$$FC(H, E) + FC(H', E) = 0$$

lo que significa que si la evidencia confirma una hipótesis mediante algún valor FC(H|E), la confirmación de la negación de la hipótesis no es 1 – FC(H|E), que es lo que podría esperarse utilizando la teoría de la probabilidad. Es decir:

$$FC(H, E) + FC(H', E) \neq 1$$

El hecho de que FC(H | E) + FC(H'| E) = 0 significa que la evidencia que sustenta una hipótesis reduce el respaldo a la negación mediante una cantidad igual, de modo que la suma siempre es 0.

Para el ejemplo de la graduación del estudiante, si obtiene una A en el curso:

$$FC(H, E) = 0.70 \qquad FC(H', E) = -0.70$$

lo que significa que

(6) Estoy un 70 por ciento seguro de que me graduaré si obtengo una A en este curso.

(7) Estoy un -70 por ciento seguro de que no me graduaré si obtengo una A en este curso.

Observe que el –70 por ciento surge debido a que los factores de certidumbre se definen en el intervalo:

$$-1 \leq FC(H, E) \leq +1$$

donde 0 significa falta de evidencia. Así, los valores de certidumbre mayores a 0 favorecen la hipótesis, mientras que los factores de certidumbre menores a 0 favorecen la negación de la hipótesis. Las afirmaciones (6) y (7) son equivalentes utilizando factores de certidumbre análogos para el hecho de que "sí = no no".

Los anteriores valores de FC pueden obtenerse mediante la pregunta:

¿Cuánto cree que obtener una A le ayudará a graduarse?

si la evidencia es para confirmar la hipótesis, o

¿Cuánto no cree que obtener una A le ayudará a graduarse?

Una respuesta de 70 por ciento para cada pregunta establecerá FC(H | E) = 0.70 y FC(H' | E) = –0.70. En el sistema MYCIN, en lugar de preguntar certidumbre en una escala de porcentajes, se pidió a los especialistas que proporcionaran la certidumbre en una escala del 1 al 10, donde 10 era definido. El usuario podría contestar DES para la evidencia desconocida correspondiente a FC = 0.

Cálculo con factores de certidumbre

Aunque la definición original de FC era

$$FC = MC - MI$$

hubo dificultades con ella porque una pieza de evidencia que no sirviera de confirmación podía controlar la confirmación de muchas otras piezas de evidencia. Por ejemplo, diez piezas de evidencia producirían un MC = 0.999 y una pieza de no confirmación con MI = 0.799 podría dar entonces

$$FC = 0.999 - 0.799 = 0.200$$

En el sistema MYCIN, un FC antecedente de regla debe ser > 0.2 para que el antecedente se considere verdadero y active la regla. Este **valor de umbral** de 0.2 no se hizo como un axioma fundamental de la teoría de FC, sino más bien como una manera *ad hoc* de disminuir al mínimo la activación de reglas que sólo sugieren débilmente una hipótesis. Sin un umbral, podrían activarse muchas reglas con escaso o nulo valor y reducir en gran medida la eficiencia del sistema.

En 1977, se cambió la definición de FC en MYCIN a:

$$FC = \frac{MC - MI}{1 - \text{mín}(MC, MI)}$$

para atenuar los efectos de una sola pieza de evidencia que no confirma la hipótesis sobre muchas otras piezas que si la confirman. Usando esta definición con MC = 0.999, MI = 0.799:

$$FC = \frac{0.999 - 0.799}{1 - \text{mín}(0.999, 0.799)} = \frac{0.200}{1 - 0.799} = 0.995$$

que es muy diferente a la definición anterior de 0.999 – 0.799 = 0.200 y, por tanto, no activaba una regla porque no era mayor que 0.2; ahora el 0.995 hará que la regla se active.

Los métodos de MYCIN para combinar la evidencia en el antecedente de una regla se muestran en la tabla 5.2. Obsérvese que son los mismos que las reglas de PROSPECTOR basadas en la lógica confusa.

Evidencia, E	Certidumbre de antecedente
E_1 Y E_2	mín[FC(H,E_1),FC(H,E_2)]
E_1 O E_2	máx[FC(H,E_1),FC(H,E_2)]
NO E alinear	–FC(H,E)

Tabla 5.2 Reglas de MYCIN para combinación de evidencia de antecedente de expresiones elementales

Por ejemplo, dada una expresión lógica para combinar evidencia como:

$$E = (E_1 \text{ Y } E_2 \text{ Y } E_3) \text{ O } (E_4 \text{ Y NO } E_5)$$

la evidencia E se calcularía como:

$$E = \text{máx}[\text{mín}(E_1, E_2, E_3), \text{mín}(E_4, -E_5)]$$

Para valores:

$$E_1 = 0.9 \qquad E_2 = 0.8 \qquad E_3 = 0.3$$
$$E_4 = -0.5 \qquad E_5 = -0.4$$

el resultado es:

$$\begin{aligned} E &= \text{máx}[\text{mín}(0.9, 0.8, 0.3), \text{mín}(-0.5, -(-0.4))] \\ &= \text{máx}[0.3, -0.5] \\ &= 0.3 \end{aligned}$$

La fórmula fundamental para el FC de una regla

```
SI E ENTONCES H
```

La proporciona la fórmula

$$(8)\quad FC(H,e) = FC(E,e)\ FC(H,E)$$

donde:

FC(E,e) es el factor de certidumbre de la evidencia E que integra el antecedente de la regla basada en la evidencia incierta e.
FC(H,E) es el factor de certidumbre de la hipótesis que supone que la evidencia se conoce con certidumbre, cuando FC(E,e) = 1.
FC(H,e) es el factor de certidumbre de la hipótesis basada en evidencia incierta *e*.

Por tanto, si se conoce con certidumbre toda la evidencia del antecedente, la fórmula para el factor de certidumbre de la hipótesis es:

$$FC(H,e) = FC(H,E)$$

ya que FC(E,e) = 1.

Como ejemplo de estos factores de certidumbre, consideremos el FC para la regla del estreptococo explicada antes:

```
SI  1) El teñido gram del microorganismo es positivo, y
    2) La morfología del microorganismo es coco, y
    3) La forma de crecimiento del organismo es en cadenas
ENTONCES Hay evidencia que sugiere (0.7) que la identidad
         del microorganismo es estreptococo
```

donde el factor de certidumbre de la hipótesis bajo evidencia cierta es:

$$FC(H,E) = FC(H,E_1 \cap E_2 \cap E_3) = 0.7$$

y también se le llama ***factor de atenuación.***

El factor de atenuación se basa en la suposición de que toda la evidencia (E_1, E_2 y E_3) se conoce con certidumbre, es decir:

$$FC(E_1,e) = FC(E_2,e) = FC(E_3,e) = 1$$

donde e es la evidencia observada que conduce a la conclusión de que E_1 se conoce con certidumbre. Estos valores de FC son análogos a las probabilidades condicionales de la evidencia del sistema PROSPECTOR, P(E | e). El factor de atenuación expresa el grado de certidumbre de la hipótesis, dada la evidencia cierta.

Al igual que con PROSPECTOR, si no se conocen con certidumbre todas las evidencias se presentan complicaciones. Con PROSPECTOR, la fórmula de interpolación P(H | e) se utilizó para evidencia de incertidumbre; en el caso de MYCIN, la fórmula fundamental (8) se debe utilizar para determinar el valor de FC, dado que $FC(H,E_1 \cap E_2 \cap E_3) = 0.7$ ya no es válido para evidencia incierta.

Por ejemplo, supongamos que:

$$FC(E_1,e) = 0.5$$
$$FC(E_2,e) = 0.6$$
$$FC(E_3,e) = 0.3$$

entonces

$$\begin{aligned} FC(E,e) &= FC(E_1 \cap E_2 \cap E_3,e) \\ &= \text{mín}[FC(E_1,e),FC(E_2,e),FC(E_3,e)] \\ &= \text{mín}[0.5,\ 0.6,\ 0.3] \\ &= 0.3 \end{aligned}$$

En virtud del FC del antecedente, FC(E,e) > 0.2, el antecedente se considera verdadero y, por tanto, se activa la regla. El factor de certidumbre de la conclusión es:

$$\begin{aligned} FC(H,e) &= FC(E,e)\ FC(H,E) \\ &= 0.3 \cdot 0.7 \\ &= 0.21 \end{aligned}$$

Suponiendo que otra regla también concluye la misma hipótesis, pero con un factor de certidumbre diferente. Los factores de certidumbre de reglas que concluyen la misma hipotesis se calculan a partir de la **función de combinación** para factores de certidumbre definidos como:

$$(9)\qquad FC_{COMBINADO}(FC_1,FC_2) = \begin{cases} FC_1 + FC_2(1 - FC_1) & \text{ambos } > 0 \\ \dfrac{FC_1 + FC_2}{1 - \text{mín}(|FC_1|,|FC_2|)} & \text{uno} < 0 \\ FC_1 + FC_2(1 + FC_1) & \text{ambos} < 0 \end{cases}$$

donde la fórmula para el $FC_{COMBINADO}$ utilizada depende de si los factores de certidumbre individuales son positivos o negativos. La función de combinación para más de dos factores de certidumbre se aplica de manera gradual, es decir, el $FC_{COMBINADO}$ se calcula primero para dos valores y luego se combina con el tercer valor utilizando la fórmula (9), y así sucesivamente. En la figura 5.5 se resumen los cálculos con factores de certidumbre para dos reglas basadas en evidencia incierta y que concluyen la misma hipótesis. Obsérvese que éste *no* es un árbol Y/O, porque el $FC_{COMBINADO}$ no tiene nada que ver con los objetivos Y y O.

En el ejemplo, si otra regla concluye que se trata de estreptococo con un $FC_2 = 0.5$, entonces la certidumbre combinada, utilizando la primera fórmula de (9), es:

$$FC_{COMBINADO}(0.21,\ 0.5) = 0.21 + 0.5\ (1 - 0.21) = 0.605$$

Supongamos que una tercera regla tiene la misma conclusión, pero con un $FC_3 = -0.4$, entonces se utiliza la segunda fórmula de (9):

$$FC_{COMBINADO}(0.605,\ -0.4) = \frac{0.605 - 0.4}{1 - \text{mín}(|0.605|,|-0.4|)}$$

$$= \frac{0.205}{1 - 0.4} = 0.34$$

La fórmula $FC_{COMBINADO}$ conserva la conmutatividad de la evidencia, es decir:

$$FC_{COMBINADO}(X,Y) = FC_{COMBINADO}(Y,X)$$

y, por consiguiente, el orden en el que se recibe la evidencia no afecta el resultado.

En lugar de almacenar valores separados de MC y MI para cada hipótesis, MYCIN almacenó el $FC_{COMBINADO}$ actual con cada hipótesis y lo combinó con evidencia nueva a medida que quedaba disponible

$$FC_{COMBINADO}(FC_1, FC_2) = \begin{cases} FC_1 + FC_2(1 - FC_1) & \text{ambos } >0 \\ \dfrac{FC_1 + FC_2}{1 - \text{mín}(|FC_1|, |FC_2|)} & \text{uno } < 0 \\ FC_1 + FC_2(1 + FC_1) & \text{ambos } < 0 \end{cases}$$

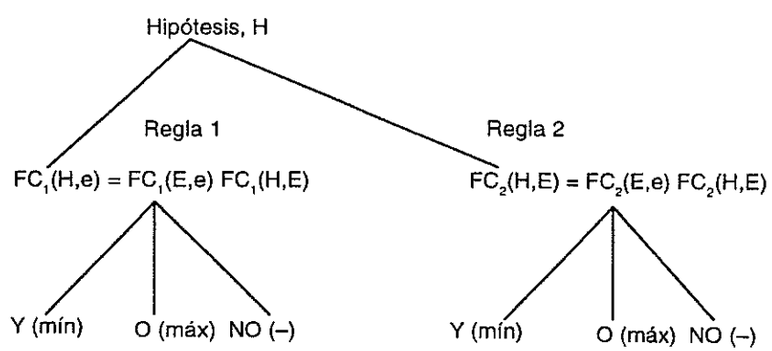


Figura 5.5 Factores de certidumbre para dos reglas con la misma hipótesis basada en evidencia incierta

Dificultades con los factores de certidumbre

Aunque MYCIN tuvo éxito en el diagnóstico, hubo dificultades con los fundamentos teóricos de los factores de certidumbre, pues aunque éstos tenían la misma base en la teoría de la probabilidad y en la de la confirmación, también fueron en parte *ad hoc*. La principal ventaja del factor de certidumbre fueron los cálculos simples que permitieron la propagación de la incertidumbre en el sistema. Los factores de certidumbre también fueron fáciles de comprender y separaron claramente creencia e incredulidad.

Sin embargo, hubo problemas con los factores de certidumbre; uno de ellos fue que sus valores podrían ser lo opuesto de las probabilidades condicionales. Por ejemplo, si:

$$P(H_1) = 0.8 \qquad P(H_2) = 0.2$$
$$P(H_1 \mid E) = 0.9 \qquad P(H_2 \mid E) = 0.8$$

entonces

$$FC(H_1, E) = 0.5 \text{ y } FC(H_2, E) = 0.75$$

Como uno de los propósitos de los factores de certidumbre es organizar hipótesis a partir de diagnósticos posibles, es una contradicción que una enfermedad tenga una probabilidad condicional más alta P(H | E) y que aún así tenga un factor de certidumbre más bajo, FC(H,E).

El segundo problema mayor con los factores de certidumbre es que, en general:

$$P(H \mid e) \neq P(H \mid i)\ P(i \mid e)$$

donde i es una hipótesis intermedia basada en la evidencia e, y todavía el factor de certidumbre de dos reglas en una cadena de inferencia se calcula como probabilidades independientes mediante

$$FC(H,e) = FC(H,i)\ FC(i,e)$$

La fórmula anterior sólo es verdadera en el caso especial de que la población estadística con la propiedad H esté contenida en la población con la propiedad i, que a su vez esté contenida en la población con la propiedad e (Adams 85). El éxito de MYCIN, a pesar de estos problemas, se debe probablemente a sus cortas cadenas de inferencia e hipótesis simples. Podría haber problemas reales en la aplicación de factores de certidumbre a otros dominios que no tuvieran ese tipo de cadenas e hipótesis. En realidad, Adams demostró que la teoría de factores de certidumbre es en realidad una aproximación a la teoría de la probabilidad normal.

5.4 TEORÍA DE DEMPSTER Y SHAFER

En esta sección se analiza un método de razonamiento inexacto denominado **teoría de Dempster y Shafer** (o teoría de Shafer y Dempster). Se basa en el trabajo hecho originalmente por Dempster, quien intentó modelar la incertidumbre mediante un rango de probabilidades más que como un simple número probabilístico (Dempster 67). Shafer extendió y refinó ese trabajo en un libro publicado en 1976 y llamado *Una teoría matemática de la evidencia* (Shafer 76). Una extensión mayor llamada **razonamiento evidencial** trata con la información que se espera incierta, imprecisa y en ocasiones inexacta (Garvey 81; Lowrance 82). Los factores de certidumbre pueden mostrarse como un caso especial de la teoría Demster-Shafer, que los coloca como una base teórica más que como una base *ad hoc* (Gordon 85).

Marcos de discernimiento

La teoría de Dempster y Shafer supone que hay un conjunto fijo de elementos mutuamente excluyentes y absolutos a los que llama el **ambiente** y simboliza con la letra Griega Θ.

$$\Theta = \{\theta_1, \theta_2, \ldots \theta_n\}$$

El ambiente es otro término para el universo del discurso en la teoría de conjuntos. Es decir, el ambiente es el conjunto de objetos que nos interesan. Algunos ejemplos podrían ser:

Θ = {carguero, bombardero, caza}
Θ = {rojo, verde, azul, naranja, amarillo}
Θ = {granero, hierba, persona, vaca, automóvil}

Obsérvese que los elementos son mutuamente excluyentes. Por ejemplo un carguero no es un bombardero ni un caza, el rojo no es verde, la hierba no es una vaca, etc. Supongamos que todos los elementos posibles del universo están en el conjunto y por lo tanto es absoluto, y para simplificar el análisis, supongamos también que Θ es un conjunto finito. Sin embargo, se han hecho trabajos en ambientes de Dempster y Shafer cuyos elementos son variables continuas como tiempo, distancia, velocidad, etcétera (Strat 84).

Una forma de considerar a Θ es a manera de preguntas y respuestas. Supongamos:

Θ = {comercial, bombardero, caza}

y la pregunta es "¿cuáles son las aeronaves militares?" La respuesta es el subconjunto de Θ:

$\{\theta_2, \theta_3\}$ = {bombardero, caza}

Del mismo modo, la pregunta "¿cuáles son las aeronaves civiles?" es el conjunto

$\{\theta_1\}$ = {comercial}

al que se le llama **conjunto de un solo elemento.**

Cada subconjunto de Θ puede interpretarse como una posible respuesta a una pregunta. Debido a que los elementos son mutuamente excluyentes y a que el ambiente es absoluto, puede haber un subconjunto de una sola respuesta correcta para una pregunta. Por supuesto, no todas las preguntas posibles son significativas o interesantes, pero lo importante es que los subconjuntos del ambiente sean todas las respuestas válidas posibles en este universo de discurso. Cada subconjunto puede considerarse una proposición implícita, como:

La respuesta correcta es $\{\theta_1, \theta_2, \theta_3\}$
La respuesta correcta es $\{\theta_1, \theta_2\}$

y así para todos los subconjuntos, en los que está implícito "La respuesta correcta es" con el subconjunto.

En la figura 5.6 se muestran todos los subconjuntos posibles del ambiente de aeronaves, con líneas que indican la relación entre los conjuntos; las abreviaturas A, B y F se utilizan para *comercial, bombardero* y *caza.* Este diagrama está dibujado como una rejilla jerárquica, con Θ en la parte superior y el conjunto nulo $\varnothing$ = { } en la parte inferior; aunque por lo general, no se muestra explícitamente el conjunto nulo porque siempre corresponde a la respuesta falsa. Como $\varnothing$ no tiene elementos, su elección como respuesta contradice la suposición del ambiente como exhaustivo.

La respuesta correcta es ningún elemento

Observe que la figura es una rejilla y no un árbol porque los nodos del subconjunto podrían tener más de un padre. La rejilla es jerárquica porque se dibuja de los conjuntos más grandes a los más pequeños. Por ejemplo, una ruta de Θ a $\varnothing$ expresa la relación jerárquica de subconjuntos que conectan a padre a hijo, como:

$\varnothing \subset \{A\} \subset \{A,B\} \subset \{A,B,F\}$

Como se analiza en la sección 2.10, una relación entre dos conjuntos X y Y, como en:

$X \subseteq Y$

significa que todos los elementos de X son elementos de Y y se escribe más formalmente:

$X \subseteq Y = \{x \mid x \in X \rightarrow x \in Y\}$

Esto establece que si x es un elemento del conjunto de X entonces esto implica que también lo es del conjunto Y. Si $X \subseteq Y$, pero $X \neq Y$, hay por lo menos un elemento de Y que no es elemento de X entonces éste es un *subconjunto propio* de Y y se escribe:

$X \subset Y$

A un ambiente se le llama un ***marco de discernimiento*** cuando sus elementos pueden interpretarse como respuestas posibles y sólo una respuesta es correcta. El término *discernir* significa que es posible distinguir una respuesta correcta de todas las otras respuestas posibles a una pregunta, si la respuesta no está en el marco, entonces éste debe ampliarse para acomodar el conocimiento adicional de elementos θ_{N+1}, θ_{N+2}, etc. Una respuesta correcta requiere que el conjunto sea exhaustivo y que los subconjuntos estén desunidos.

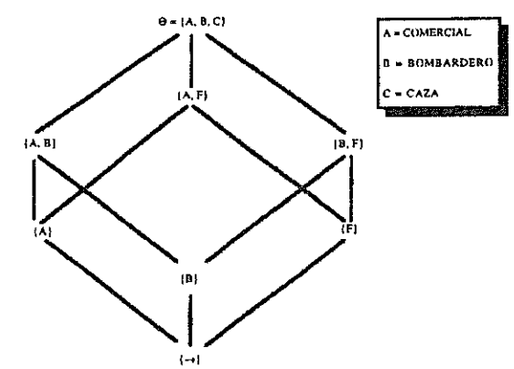


Figura 5.6 Todos los subconjuntos del ambiente de aeronaves

Un conjunto de tamaño N tiene exactamente 2^N subconjuntos, incluyéndose a sí mismo, y estos subconjuntos definen el conjunto de capacidad (consúltese el problema 2-11), escrito como $P(\Theta)$. De esta manera, para el ambiente de aeronaves:

$P(\Theta) = \{\varnothing, \{A\}, \{B\}, \{F\}, \{A,B\}, \{A,F\}, \{B,F\}, \{A,B,F\}\}$

El conjunto de capacidad del ambiente tiene como elementos todas las respuestas para las preguntas posibles del marco de discernimiento. Esto significa que hay una correspondencia uno a uno entre los elementos de $P(\Theta)$ y los subconjuntos de Θ.

Funciones de masa e ignorancia

En la teoría bayesiana, la probabilidad posterior cambia a medida que se adquiere evidencia. Del mismo modo, en la teoría de Dempster y Shafer la creencia en la evidencia puede variar; es costumbre en esta teoría pensar acerca del *grado de creencia en la evidencia* como analogía a la **masa** de un objeto físico, es decir, la masa de evidencia sustenta una creencia. La **medida de evidencia**, simbolizada por la letra m, es análoga a la cantidad de masa; otro término para designarla es **asignación básica de probabilidad (abp),** o en ocasiones sólo **asignación básica,** debido a la analogía en la forma de las ecuaciones que describen densidades y masa de probabilidades. Sin embargo, debido a su posible confusión con la teoría de la probabilidad, no utilizaremos estos términos y lo llamaremos simplemente *masa*. La razón para tomar esta analogía con una masa es considerar a la creencia como una cantidad que puede desplazarse, dividirse y combinarse, como objetos hechos de barro, de modo que sus partes puedan desprenderse y unirse nuevamente.

Una diferencia fundamental entre la teoría de Dempster y Shafer y la teoría de la probabilidad, es el tratamiento de la **ignorancia.** Como se analiza en el capítulo 4, la teoría de la probabilidad debe distribuir una cantidad igual de probabilidad incluso en la ignorancia, por ejemplo, si no se tiene conocimiento previo, entonces debe suponerse que la probabilidad, P, de cada posibilidad es:

$$P = \frac{1}{N}$$

donde N es el número de posibilidades. Como se menciona en el mismo capítulo, esta asignación de P está hecha *a la desesperada* o, utilizando un término que suena más impresionante, mediante el uso del **principio de indiferencia.**

El caso extremo de aplicación del principio de indiferencia ocurre cuando sólo hay dos posibilidades, como hay o no hay petróleo, simbolizado por H y H′. En casos como éste, P = 50 por ciento aún cuando no hay conocimiento en absoluto, puesto que la teoría de la probabilidad dice que:

$$P(H) + P(H') = 1$$

Es decir, cualquier cosa que no apoye *debe* refutar, porque no se permite la ignorancia.

Esto puede acarrear algunas consecuencias ridículas, si se aplica sin pensar. Por ejemplo, haya o no haya petróleo debajo de su casa, conforme al principio de indiferencia, si no tiene *absolutamente ningún otro conocimiento,* hay un 50 por ciento de probabilidades de que lo haya. Si lo piensa, una posibilidad de 50 por ciento es muy impresionante y ofrece una mejor oportunidad de obtener riquezas de forma más rápida que cualquier otra inversión legal. Como hay un 50 por ciento de oportunidades de que haya petróleo, ¿debería sacar inmediatamente todos sus ahorros, contratar un equipo de perforación y empezar a perforar la cocina?

Siguiendo esta misma línea de razonamiento, la aplicación del principio de indiferencia y la teoría de la probabilidad dice que hay un 50 por ciento de oportunidades de:

```
diamantes
el tesoro de un pirata
abrigos de piel
queso verde
asignación de su próxima tarea
```


y cualquier otra cosa que pueda pensar que hay debajo de su casa. (En realidad, puede volverse rico con sólo salir en televisión y decir a todos cómo *los demás* pueden volverse ricos si envían $9.95 para comprar su libro sobre teoría de la probabilidad. Ésta es, también, la mejor razón para no tirar la basura nunca, pues si ignora lo que hay ahí, podría haber cualquier cosa, así que existe un 50 por ciento de oportunidades de volverse rico si descubre algo valioso.)

Aunque no se utilice el principio de indiferencia, la restricción

$$P(H) + P(H') = 1$$

obliga a la asignación de probabilidad a la negación de la hipótesis, aunque no haya evidencia de ésta. Como se explica en la sección 5.3, esto no es una buena suposición con muchos tipos de creencias, como el conocimiento médico. No obstante, la teoría de la probabilidad requiere que la evidencia que no sustente una hipótesis, la refute.

La teoría de Dempster y Shafer no obliga a que se asigne una creencia a la ignorancia o refutación de una hipótesis. En cambio, sólo se asigna la masa a los subconjuntos del ambiente a los que se desee asignar una creencia. Cualquier creencia que no se asigne a un subconjunto específico se considera **no creencia** o **falta de creencia** y sólo se asocia con el ambiente Θ. La creencia que refuta una hipótesis es incredulidad, lo que significa *no* creencia.

Por ejemplo, supongamos que un sensor Identificador de amigo o enemigo (IFF, Identification Friend or Foe) no obtiene respuesta por radio de una aeronave (Bogler 87). Un IFF es un radio transmisor/receptor que transmite un mensaje de radio a una aeronave; si ésta es amigable, su radio respondería enviando su código de identificación. Como opción predeterminada, la aeronave que no conteste será considerada hostil. Habría varias razones por las que una aeronave no respondería a un IFF, por ejemplo

- Mal funcionamiento del IFF
- Mal funcionamiento del radio del avión
- El avión no tiene radio IFF
- Interferencia de la señal IFF
- Tiene órdenes de mantener el radio en silencio

Suponiendo que la falla del IFF para emitir una respuesta indica una creencia en la evidencia de 0.7 de que el avión es hostil, donde las aeronaves hostiles sólo se consideran bombarderos y de caza. De esta manera, la masa se asigna al subconjunto {B,F}, y:

$$m_1(\{B,F\}) = 0.7$$

donde m_1 se refiere a esta primera evidencia del sensor IFF.

El resto de la creencia se deja con el ambiente, Θ, como no creencia.

$$m_1(\Theta) = 1 - 0.7 = 0.3$$

Cada conjunto del conjunto de capacidad del ambiente que tiene una masa mayor que 0 es un **elemento focal.** El término *elemento focal* se utiliza porque un conjunto X tal como m(X) > 0 es un elemento de conjunto de capacidad en el que la evidencia disponible está enfocada o concentrada.

La teoría de Dempster y Shafer tiene una diferencia primordial con la teoría de la probabilidad, en la que se supondría que

P(hostil) = 0.7
P(no hostil) = 1- 0.7 = 0.3

En la teoría de la probabilidad, si la creencia en la hostilidad es de 0.7, entonces el escepticismo de que sea hostil debe ser 0.3. En cambio, el 0.3 de la teoría de Dempster y Shafer permanece como no creencia en el ambiente por m(Θ). Esto significa *ni creencia ni incredulidad* en la evidencia en un grado de 0.3. Se presume que el objetivo es hostil en un grado de 0.7 y hay un juicio reservado de 0.3 de escepticismo y la creencia adicional de hostilidad. Es muy importante percatarse que la asignación de 0.3 al ambiente Θ no asigne *ningún* valor a los subconjuntos de Θ, aunque los subconjuntos incluyan los subconjuntos hostiles {B,F}, {B} y {F}.

Regresando al ejemplo del estudiante de la sección anterior:

m(obtener una A y graduarse) = 0.7

no significaría inmediatamente:

m(obtener una A y no graduarse) = 0.3

a no ser que se les asignen valores a ambas.
Una masa tiene considerablemente más libertad que probabilidades, como se muestra en la tabla 5.3.

Teoría de Dempster y Shafer	Teoría de la probabilidad
m(Θ)no tiene que ser 1	$\sum P_1 = 1$
Si $X \subseteq Y$, no es necesario que $m(X) \leq m(Y)$	$P(X) \leq P(Y)$
No es necesaria una relación entre m(X) y m(X')	$P(X) + P(X') = 1$

Tabla 5.3 Comparación de la masa de Dempster y Shafer con la teoría de la probabilidad

Cada masa puede expresarse formalmente como una función que relaciona cada elemento del conjunto de capacidad con un número real dentro del intervalo de 0 a 1. Esto significa simplemente que la creencia en un subconjunto podría tomar cualquier valor desde 0 hasta 1. Esta relación se establece de manera formal como:

$$m: P(\Theta) \rightarrow [0,1]$$

Se acostumbra definir a la masa del conjunto vacío como cero:

$$m(\varnothing) = 0$$

y la suma de todas las masas para cada subconjunto, X, del conjunto de capacidad es 1:

$$\sum_{X \in P(\Theta)} m(X) = 1$$

Por ejemplo, en el ambiente de aeronaves:

$$\sum_{X \in P(\Theta)} m(X) = m(\{B, F\} + m(\Theta) = 0.7 + 0.3 = 1$$

Combinación de la evidencia

Ahora consideremos el caso en donde aparece evidencia adicional. Nos gustaría combinar toda la evidencia para producir una mejor estimación de la creencia en ella. Para mostrar como se hace, se muestra primero un ejemplo que es un caso especial de la fórmula general para combinación de la evidencia.

Supongamos que un segundo tipo de sensor identifica el objetivo como un bombardero con una creencia en la evidencia de 0.9. Ahora las masas de evidencia de los sensores son las siguientes:

$$m_1(\{B,F\}) = 0.7 \qquad m_1(\Theta) = 0.3$$

$$m_2(\{B\}) = 0.9 \qquad m_2(\Theta) = 0.1$$

donde m_1 y m_2 se refieren al primer y segundo tipo de sensores.

Esta evidencia puede combinarse utilizando la siguiente forma especial de **Regla de combinación de Dempster** para producir la **masa combinada:**

$$m_1 \oplus m_2(Z) = \sum_{X \cap Y = Z} m_1(X)\, m_2(Y)$$

donde la suma se extiende a todos los elementos cuya intersección $X \cap Y = Z$. El **operador** ⊕ denota la **suma ortogonal** o **suma directa**, que se define mediante la sumatoria de las intersecciones de los productos de masa en el lado derecho de la regla. La regla de Dempster combina masas para producir una nueva masa que representa un **consenso** en la evidencia original y que posiblemente está en conflicto. Esta nueva masa es un *consenso* porque tiende a favorecer acuerdos más que desacuerdos al incluir solamente masas en las intersecciones de conjunto, que representan elementos comunes de evidencia. Un punto importante es que la regla debe emplearse para combinar evidencia que tiene *errores independientes,* que *no* es lo mismo que evidencia reunida independientemente.

En la tabla 5.4 se muestran las intersecciones de masas y productos para el ambiente de las aeronaves organizadas en una tabla. A cada intersección de conjunto le sigue el producto numérico de su masa.

	m2({B}) = 0.9	**m2(Θ) = 0.1**
m1({B,F}) = 0.7	{B} 0.63	{B,F} 0.07
m1(Θ) = 0.3	{B} 0.27	Θ 0.03

Tabla 5.4 Confirmación de la evidencia

Las entradas de la tabla se calculan al multiplicar de manera cruzada la masa de filas y columnas, como se muestra a continuación, donde T_{ij} denotan la fila i y la columna j de la tabla:

$$T_{11}(\{B\}) = m_1(\{B,F\})\, m_2(\{B\}) = (0.7)\,(0.9) = 0.63$$

$$T_{21}(\{B\}) = m_1(\Theta)\, m_2(\{B\}) = (0.3)\,(0.9) = 0.27$$

$$T_{12}(\{B,F\}) = m_1(\{B,F\})\, m_2(\Theta) = (0.7)\,(0.1) = 0.07$$

$$T_{22}(\Theta) = m_1(\Theta)\, m_2(\Theta) = (0.3)\,(0.1) = 0.03$$

Una vez que se calculan los productos de masa individuales, como se muestra antes de este párrafo, entonces de acuerdo a la regla de Dempster, se agregan los productos sobre las intersecciones de conjunto:

$$m_3(\{B\}) = m_1 \oplus m_2(\{B\}) = 0.63 + 0.27$$
$$= 0.90 \qquad \text{bombardero}$$
$$m_3(\{B,F\}) = m_1 \oplus m_2(\{B,F\})$$
$$= 0.07 \qquad \text{bombardero o caza}$$
$$m_3(\Theta) = m_1 \oplus m_2(\Theta)$$
$$= 0.03 \qquad \text{no creencia}$$

El $m_3(\{B\})$ representa la creencia de que el objetivo es un bombardero y *sólo* un bombardero. Sin embargo, $m_3(\{B,F\})$ y $m_3(\Theta)$ implican información adicional; debido a que sus conjuntos incluyen un bombardero, es plausible que sus sumas ortogonales puedan contribuir a la creencia en el bombardero. De modo que su $0.07 + 0.03 = 0.1$ se agregaría a la creencia de 0.90 en el conjunto del bombardero, para producir la creencia máxima de que podría ser un bombardero: la creencia plausible. En lugar de restringir la creencia a un solo valor, hay un **rango de creencia** en la evidencia. La creencia va del mínimo conocido de 0.9 para el bombardero a un máximo plausible de $0.90 + 0.1 = 1$ de que *podría* ser un bombardero. La creencia verdadera se supone en algún punto dentro del rango 0.9 a 1.

En razonamiento evidencial, se dice que la evidencia induce un **intervalo evidencial**. Al **límite inferior** se le llama **soporte (Spt)** en razonamiento evidencial y **Bel** en la teoría de Dempster y Shafer mientras que al **límite superior** se le llama **plausibilidad (Pls)**. Para este ejemplo, el intervalo evidencial es [0.90,1], el límite inferior es 0.90 y el superior es 1. El soporte es la *creeencia mínima* basada en la evidencia, mientras que la plausibilidad es la *creencia máxima* que se desea otorgar. En general, los rangos de Bel y Pls son $0 \leq Bel \leq Pls \leq 1$. En la teoría de Dempster y Shafer, a los límites inferior y superior se les llama en ocasiones probabilidades inferior y superior, conforme al escrito original de Dempster (Dempster 67). En la tabla 5.5 se muestran algunos intervalos evidenciales comunes (Wesley 86).

Intervalo evidencial		Significado
[1,1]		Completamente verdadero
[0,0]		Completamente falso
[0,1]		Completamente ignorante
[Bel,1]	donde $0 < Bel < 1$ aquí	Tiende a sustentar
[0,Pls]	donde $0 < Pls < 1$ aquí	Tiende a refutar
[Bel,Pls]	donde $0 < Bel \leq Pls < 1$ aquí	Tiende a sustentar y refutar

Tabla 5.5 Algunos intervalos evidenciales comunes

El sustento o **función de creencia, Bel,** es la *creencia total* de un conjunto y *todos* sus subconjuntos. Bel es toda la masa que sustenta un conjunto y se define a partir de la masa.

$$Bel(X) = \sum_{Y \subseteq X} m(Y)$$

Como ejemplo, el ambiente de las aeronaves para el primer sensor:

$$Bel_1(\{B,F\}) = m_1(\{B,F\}) + m_1(\{B\}) + m_1(\{F\})$$
$$= 0.7 + 0 + 0 = 0.7$$

A la función Bel se le llama a veces la *medida de creencia,* o simplemente *creencia.* Sin embargo, conviene advertir que la función de la creencia es muy diferente de la masa, que es la creencia en la evidencia asignada a *un solo* conjunto. Por ejemplo, suponga que posee un Ford y escucha que la policía busca un Ford utilizado para huir después de un asalto bancario; hay una gran diferencia entre escuchar que la policía está buscando *un* Ford y escuchar que la policía está buscando *su* Ford. La masa es la creencia en un conjunto y *no* en alguno de sus subconjuntos, mientras que una función de creencia se aplica a un conjunto y a *todos* sus subconjuntos. Bel es la *creencia total* y, por lo tanto, es más global que la *creencia local* de masa. Debido a la relación entre masa y Bel, a la teoría de Dempster y Shafer también se le llama *teoría de funciones de creencia.* En un sentido general, la regla de Demspter puede entonces interpretarse como una forma de combinar funciones de creencia. Las funciones de masa y creencia se relacionan con:

$$m(X) = \sum_{Y \subseteq X} (-1)^{|X-Y|} Bel(Y)$$

Donde $|X-Y|$ es la **cardinalidad** del conjunto:

$$X - Y = \{x \mid \in \mid X \text{ y } x \notin Y\}$$

Es decir, $|X-Y|$ es el número de elementos del conjunto $X - Y$.

Como las funciones de creencia se definen en términos de masas, la combinación de dos de ellas también pueden expresarse a partir de las sumas ortogonales de las masas de un conjunto y todos sus subconjuntos. Por ejemplo:

$$Bel_1 \oplus Bel_2(\{B\}) = m_1 \oplus m_2(\{B\}) + m_1 \oplus m_2(\varnothing)$$
$$= 0.90 + 0 = 0.90$$

Normalmente, la masa del conjunto vacío no se escribe porque usualmente se define como cero. La creencia total para el subconjunto bombardero-caza {B,F}, tiene más subconjuntos que el anterior.

$$Bel_1 \oplus Bel_2(\{B, F\})$$
$$= m_1 \oplus m_2(\{B,F\}) + m_1 \oplus m_2(\{B\}) + m_1 \oplus m_2(\{F\})$$
$$= 0.07 + 0.90 + 0 = 0.97$$

Los términos para {B} y {F} están incluidos porque son subconjuntos de {B,F}. En la figura 5.6 puede verse que {B,F} tiene subconjuntos {B} y {F} y como no se dio masa a {F}, entonces $m(\{F\}) = 0$ y, por lo tanto, no contribuye en nada a la suma. En realidad, $m(\{F\})$ y las otras masas con valor cero nunca entraron en la tabla 5.4 porque el resultado de cualquier producto cruz con ellas sería cero. Si se han asignado masas a cada subconjunto de {A,B,F}, con excepción del conjunto nulo, que es cero, entonces la tabla 5.4 sería una tabla de célula $(2^3 - 1)(2^3 - 1) = 7 \cdot 7 = 49$.

La función de creencia combinada para Θ con base en toda la evidencia es la siguiente:

$$\begin{aligned} Bel_1 \oplus Bel_2(\Theta) &= m_1 \oplus m_2(\Theta) + m_1 \oplus m_2(\{B,F\}) + \\ &\quad m_1 \oplus m_2(\{B\}) \\ &= 0.03 + 0.07 + 0.90 = 1 \end{aligned}$$

En realidad, Bel(Θ) = 1 en todos los casos, porque la suma de masas debe ser siempre igual a 1. La combinación de evidencia sólo *redistribuye* las masas en diferentes subconjuntos.

El intervalo evidencial de un conjunto S, IE(S), podría definirse en términos de la creencia:

$$EI(S) = [Bel(S), 1 - Bel(S')]$$

Si S = {B}, entonces S′ = {A,F} y

$$\begin{aligned} Bel(\{A,F\}) &= m_1 \oplus m_2(\{A,F\}) + m_1 \oplus m_2(\{A\}) + \\ &\quad m_1 \oplus m_2(\{F\}) \\ &= 0 + 0 + 0 = 0 \end{aligned}$$

debido a que éstos no son elementos focales y la masa es cero para elementos de este tipo. De esta manera, el intervalo evidencial para {B} es el siguiente:

$$\begin{aligned} EI(\{B\}) &= [0.90, 1 - 0] \\ &= [0.90, 1] \end{aligned}$$

Del mismo modo, si S = {B,F}, entonces S′ = {A} y

$$Bel(\{A\}) = 0$$

ya que {A} no es un elemento focal. También

$$Bel(\{B,F\}) = Bel_1 \oplus Bel_2(\{B,F\}) = 0.97$$

$$IE(\{B,F\}) = [0.97, 1 - 0] = [0.97, 1]$$

$$IE(\{A\}) = [0, 1]$$

donde el intervalo evidencial [0,1] refleja el *desconocimiento total* de {A}.

El intervalo evidencial de [creencia total, plausibilidad] puede expresarse como

[evidencia para sustento, evidencia para sustento + ignorancia]

Siguiendo la teoría de la probabilidad, este intervalo es el punto único:

[evidencia para sustento, evidencia para sustento]

porque no se permite la ignorancia, es decir, la evidencia que no sustenta debe refutar.

La plausibilidad se define como el grado en que la evidencia falla para refutar X:

$$Pls(X) = 1 - Bel(X') = 1 - \sum_{Y \subseteq X'} m(X')$$

La creencia plausible, Pls, amplía la creencia al máximo absoluto en el cual la creencia no asignada m(Θ) posiblemente contribuiría a la creencia. Mientras que m(Θ) podría ser un bombardero, un caza o un avión comercial, conforme a la suposición de plausibilidad se considera que contribuye a la creencia en uno de sus subconjuntos. Como {B} es un subconjunto de Θ, es plausible que pueda asignarse el 0.3 de creencia de $m_1(\Theta)$ a un bombardero. No hay que olvidar, como se menciona en la sección 4.15 que una creencia plausible es un poco más fuerte que una creencia posible, pero no necesariamente una creencia sustentada por evidencia fuerte. Otro punto es que Θ no es el único tipo de conjunto que amplía la creencia de cualquier conjunto X, cualquier conjunto que interseque con X y su complemento también lo hace.

La **duda (Dud)** representa el grado al que no se cree o se refuta X. La **ignorancia (Igr)** es el grado al que la masa soporta X y X′. Éstos se definen como sigue:

$$Dud(X) = Bel(X') = 1 - Pls(X)$$

$$Igr(X) = Pls(X) - Bel(X)$$

La normalización de creencia

Supongamos que ahora un tercer sensor reporta evidencia conflictiva de un avión comercial:

$$m_3(\{A\}) = 0.95 \qquad m(\Theta) = 0.05$$

En la tabla 5.6 se muestra cómo se calculan los productos cruzados.

	$m_1 \oplus m_2(\{B\})$ 0.90		$m_1 \oplus m_2(\{B,F\})$ 0.07		$m_1 \oplus m_2(\Theta)$ 0.03	
$m_3(\{A\}) = 0.95$	∅	0.855	∅	0.0665	∅	0.0285
$m_3(\Theta) = 0.05$	{B}	0.045	{B,F}	0.0035	Θ	0.0015

Tabla 5.6 Combinación de evidencia adicional, m_3

Se presenta conjunto nulo, ∅, porque {A} y {B} no tienen elementos en común y tampoco {A} y {B,F}. De esta manera:

$$\begin{aligned} m_1 \oplus m_2 \oplus m_3(\{A\}) &= 0.0285 \\ m_1 \oplus m_2 \oplus m_3(\{B\}) &= 0.045 \\ m_1 \oplus m_2 \oplus m_3(\{B,F\}) &= 0.0035 \\ m_1 \oplus m_2 \oplus m_3(\Theta) &= 0.0015 \\ m_1 \oplus m_2 \oplus m_3(\varnothing) &= 0 \quad \text{(por definición de conjuntos nulos)} \end{aligned}$$

Obsérvese que para este ejemplo, la suma de todas las masas es menor que 1:

$$\begin{aligned} \sum m_1 \oplus m_2 \oplus m_3(X) &= .0285 + .045 + .0035 + .0015 \\ &= .0785 \end{aligned}$$

donde la suma abarca todos los elementos focales. Sin embargo se requiere una suma igual a 1 porque la evidencia combinada, $m_1 \oplus m_2 \oplus m_3$, es una masa válida y la suma de todos los elementos focales debe ser 1.

La solución a este problema es una **normalización** de los elementos por medio de la división de cada uno de ellos entre:

$$1 - \kappa$$

donde k se define por cualquiera de los conjuntos X y Y como

$$\kappa = \sum_{X \cap Y = \varnothing} m_1(X)\ m_2(Y)$$

En nuestro ejemplo:

$$\kappa = 0.855 + 0.0665 = 0.9215$$

y así

$$1 - \kappa = 1 - 0.9215 = 0.0785$$

Al dividir cada elemento focal $m_1 \oplus m_2 \oplus m_3$ entre $1 - \kappa$ se obtienen los valores normalizados:

$$\begin{aligned}
m_1 \oplus m_2 \oplus m_3\ \{A\} &= 0.363 \\
m_1 \oplus m_2 \oplus m_3\ \{B\} &= 0.573 \\
m_1 \oplus m_2 \oplus m_3\ \{B,F\} &= 0.045 \\
m_1 \oplus m_2 \oplus m_3\ (\Theta) &= 0.019
\end{aligned}$$

Observe que la única evidencia de {A} ha erosionado considerablemente la creencia en {B}, como era de esperar.

La creencia total normalizada en {B} es ahora:

$$Bel(\{B\}) = m_1 \oplus m_2(\{B\} = 0.573$$

$$\begin{aligned}
Bel(\{B\}') &= Bel(\{A,F\}) \\
&= m_1 \oplus m_2 \oplus m_3(\{A,F\}) + \\
&\quad m_1 \oplus m_2 \oplus m_3(\{A\}) + \\
&\quad m_1 \oplus m_2 \oplus m_3(\{F\}) \\
&= 0 + 0.363 + 0 = 0.363
\end{aligned}$$

y así el intervalo de evidencia es ahora:

$$\begin{aligned}
EI\ (\{B\}) &= [Bel(\{B\}), 1 - Bel(\{B\}')] \\
&= [0.573, 1 - 0.363] \\
&= [0.573, 0 - 0.637]
\end{aligned}$$

Nótese que el sustento y la plausibilidad de {B} se redujeron en gran medida por la evidencia conflictiva de {A}.

La forma general de la regla de combinación de Dempster es:

$$m_1 \oplus m_2(Z) = \frac{\sum_{X \cap Y = Z} m_1(X)\ m_2(Y)}{1 - \kappa}$$

donde κ se define otra vez por conveniencia. La suma no ortogonal se define si $\kappa = 1$.

$$\kappa = \sum_{X \cap Y = \varnothing} m_1(X)\ m_2(Y)$$

κ indica la cantidad de **conflicto evidencial**. Entonces $\kappa = 0$ para la compatibilidad completa y 1 para contradicción completa. Valores $0 < \kappa < 1$ muestran una compatibilidad parcial.

Movimiento de masas y conjuntos

La analogía del movimiento de masas es útil para entender el sustento y la plausibilidad. Los conceptos principales son los siguientes:

- El sustento es la masa asignada a un conjunto y todos sus subconjuntos.
- La masa de un conjunto puede moverse libremente a sus subconjuntos.
- La masa de un conjunto no puede moverse a sus superconjuntos.
- El movimiento de masa de un conjunto a su subconjunto sólo puede contribuir a la plausibilidad del subconjunto, no a su sustento.
- La masa del ambiente, Θ, puede moverse en *cualquier* subconjunto, porque Θ está fuera de todo.

En la figura 5.7 (a) se supone que toda la masa se encuentra dentro del conjunto X y así

$$m(X) = 1$$

lo que significa que el sustento de X es 1. La plausibilidad de X también es 1 debido a que toda la masa está en X y no hay un superconjunto que pueda contribuir con masa. De este modo

$$IE(X) = [1,1]$$

Si m(X) = 0.5 entonces IE(X) = [0.5,0.5] y, en general, si m(X) = a, donde *a* es cualquier constante, entonces IE(X) = [a,a].

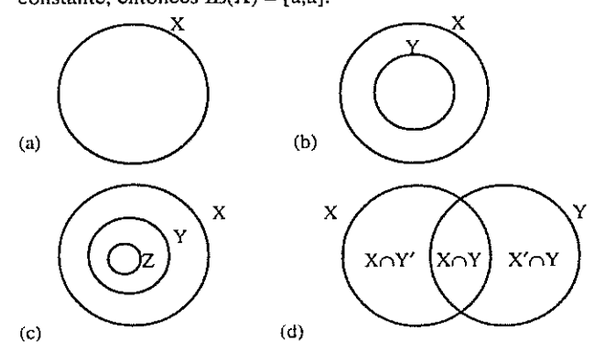


Figura 5.7 Conjuntos que ilustran el sustento y la plausibilidad

En la figura 5.7 (b), se supone que m(X) = 0.6 y m(Y) = 0.4, que también son sus sustentos. La plausibilidad de X es 0.6, porque la masa de Y no puede moverse a X. Sin embargo, la masa de X puede moverse a Y porque este es un subconjunto de X y, de este modo, su plausibilidad de Y es 0.4 + 0.6 = 1. Por tanto, los intervalos de evidencia para X y Y son los siguientes:

```
IE({X}) = [0.6,0.6]
IE({Y}) = [0.4,1]
```

Las figuras 5.7 (c) y (d) se utilizaron en el problema 5.4.

Incovenientes con la teoría de Dempster y Shafer

Una dificultad que se presenta con la teoría de Dempster y Shafer al utilizar la normalización y que podría conducir a resultados apuestos a las expectativas (Zadeh 84a), es que ignora la creencia de que el objeto considerado no existe.

Un ejemplo citado por Zadeh es la creencia de dos doctores, A y B, en la enfermedad de un paciente. En este problema las creencias son como sigue:

m_A (meningitis) = 0.99
m_A (tumor cerebral) = 0.01
m_B (conmoción) = 0.99
m_B (tumor cerebral) = 0.01

Observese que ambos doctores piensan que hay una oportunidad muy baja, 0.01, de un tumor cerebral pero están muy en desacuerdo acerca del problema principal. La regla de combinación de Dempster proporciona una creencia combinada de 1 para el del tumor cerebral; este resultado es inesperado y contra nuestra intuición, porque ambos doctores estuvieron de acuerdo en que el tumor cerebral era muy improbable. El mismo resultado de 1 para el tumor cerebral ocurriría sin importar cuáles eran las otras probabilidades.

5.5 RAZONAMIENTO APROXIMADO

En esta sección se analiza una teoría de incertidumbre basada en **lógica borrosa.** Esta teoría se relaciona principalmente con la cuantificación y el razonamiento utilizando el lenguaje natural en el que muchas palabras tienen significados ambiguos tales como *alto, caliente, peligroso, un poco, demasiado,* etc. La lógica borrosa se desarrolló a partir de la teoría básica de conjuntos borrosos, establecidos por primera vez en un artículo de Lotfi Zadeh en 1965 (Zadeh 65). Desde entonces la teoría se ha extendido y aplicado a muchos campos, como el seguimiento de un objeto con una cámara automática (Giarratano 91c). Muchas cámaras de video y cámaras de una sola lente usan la lógica borrosa. En la tabla 5.7 (Maiers 85) se presenta una lista de algunas de las aplicaciones principales de este tipo de lógica. El artículo de Maiers también incluyen 450 referencias y otro libro contiene más de 3000 (Kandel 82).

Conjuntos borrosos y lenguaje natural

La manera tradicional de representar cuáles objetos son miembros de un conjunto, es desde el punto de vista de una **función característica,** algunas veces llamada ***función de discriminación***. Si un objeto es elemento de un conjunto, entonces su función característica es 1, si no lo es entonces su función característica es 0. Esta definición se resume en la siguiente función característica:

$$\mu_A(X) = \begin{cases} 1 \text{ si } x \text{ es un elemento del conjunto } A \\ 0 \text{ si } x \text{ no es un elemento del conjunto } A \end{cases}$$

donde los objetos x son elementos de algún universo X.

La función característica también puede definirse a partir de un mapeo funcional (consulte la sección 1.10 acerca de programación funcional):

$$\mu_A(X) : x \rightarrow \{0,1\}$$

que establece que la función de característica mapea un conjunto universal X para el conjunto compuesto de 0 y 1, definición que simplemente expresa el concepto clásico de que un objeto está o no está en un conjunto. A los conjuntos en los que se aplica esto se les llama ***conjuntos definidos***, en contraste con los conjuntos borrosos. Este tipo de pensamiento data de la concepción aristotélica de **bivalente** o **lógica de dos valores,** en la que verdadero y falso son las únicas posibilidades.

Algoritmos de control
Diagnóstico médico
Toma de decisiones
Economía
Ingeniería
Ambiental
Literatura
Investigación de operaciones
Reconocimiento de patrones
Psicología
Confiabilidad
Seguridad
Ciencias

Tabla 5.7 Algunas aplicaciones de la teoría de lo borroso

El problema con esta lógica bivalente es que vivimos en un mundo análogo, no en uno digital, es decir, en la realidad las cosas no están generalmente en un estado u otro. Sólo en la arquitectura convencional de computadoras, que utiliza lógica digital, se sostiene la lógica bivalente. El desarrollo de teorías análogas de computación, como los sistemas neuronales artificiales y la teoría de lo borroso, representan con más exactitud el mundo real.

En los conjuntos difusos, un objeto podría pertenecer parcialmente a un conjunto. El grado de membresía en un conjunto difuso se mide mediante una generalización de la función de característica llamada **membresía** o **función de compatibilidad** y se define como:

$$\mu_A(x) : X \rightarrow [0,1]$$

Aunque esta definición se parece superficialmente a la definición de la función característica, en realidad es muy diferente. La función característica marca a todos los elementos de X como uno de dos elementos: 0 o 1. Por el contrario la función de membresía marca X en el codominio de los números reales definidos en el **intervalo** de 0 a 1 inclusive y simbolizado por [0,1]. Es decir, la función de membresía es un número real:

$$0 \leq \mu_A \leq 1$$

donde 0 significa que no existe y 1 significa membresía completa del conjunto, en tanto que a un valor intermedio de la función de membresía, se le llama ***grado de membresía.***

Aunque en principio parezca extraño hablar de que un elemento sólo se encuentra parcialmente en un conjunto, es en realidad más natural que los conjuntos clásicos con dos valores, pues la realidad no es sólo si o no, negro o blanco, correcto o equivocado, encender o apagar. Así como hay muchos matices de gris, no sólo negro o blanco, también hay diferentes grados de significado en la realidad. Sólo las deudas y el código fuente de una computadora deben ser exactos.

Utilizando la función de membresía, pueden describirse situaciones reales, como los días nublados. Una descripción de conjunto definido necesita una decisión arbitraria como qué es lo que constituye un día nublado. ¿Un día tiene muchas o pocas nubes, está totalmente nublado, parcialmente nublado o requiere alguna otra definición? ¿Un día lluvioso tiene 3 centímetros de precipitación, 6, 9 o algún otro índice de precipitación?

Los conjuntos y los conceptos borrosos son de uso común en lenguaje natural, como:

"Juan es *alto*"
"El clima es *cálido*"
"Gira el botón un *poco más arriba*"
"*Casi todas* las pruebas son *difíciles*"
"Si la masa está *demasiado* espesa, agregue *mucha* agua"

donde las palabras en cursivas se refieren a conjuntos y cuantificadores borrosos. Todos estos conjuntos y cuantificadores pueden representarse en la teoría de lo borroso y realizarse operaciones con ellos. En particular, el cuantificador "casi todos", que se señaló como una de las principales limitaciones de la lógica de predicado en la sección 2.16, puede manejarse en lógica borrosa, como podrá verse en breve.

En lenguaje natural los términos *vago* y *confuso* se utilizan algunas veces como sinónimos, sin embargo, hay una diferencia importante entre ellos en el contexto de la teoría de lo borroso. Una **proposición borrosa** contiene palabras como *alto,* que es el identificador de un conjunto borroso ALTO, (aquí seguiremos la costumbre de utilizar sólo mayúsculas para rotular conjuntos borrosos como éste), contrario a una proposición clásica como "Juan mide exactamente 1.50 metros de alto", que representa una proposición que es verdadera o falsa, una proposición borrosa podría tener grados de verdad. Por ejemplo, la proposición borrosa "Juan es alto", sería verdadera en algún grado: *Un poco verdadera, Algo verdadera, Bastante verdadera, Muy verdadera,* etc. A un valor de verdad borroso se le llama ***calificador borroso,*** y puede utilizarse como conjunto borroso o para modificar un conjunto borroso. A diferencia de las proposiciones definidas, a las que no se les permite tener calificadores, las proposiciones borrosas pueden tener **cuantificadores borrosos** como *Casi todo, Mucho, A menudo,* etc., sin distinción entre afirmaciones y proposiciones como en el caso clásico.

El término *vago* se emplea en el sentido de información incompleta. "Juan está en alguna parte" es vago si no se proporciona información suficiente para tomar una decisión; una proposición borrosa como "Él es alto" también puede ser vaga si no sabemos a quién se refiere el pronombre. Además, hay grados de vaguedad, una proposición como "Juan es alto" es menos vaga que "Él es alto" pero aún es vaga si no sabemos quién es Juan.

Muchas palabras borrosas, como las que se muestran en la tabla 5.8, se utilizan en el lenguaje natural y sus significados pueden definirse a partir de conjuntos borrosos, como verá en breve. Los términos compuestos, como los que se muestran en la tabla 5.9, también pueden definirse y manejarse en la teoría de lo borroso.

Aunque es difícil pensar que un objeto sólo se encuentra parcialmente en un conjunto, otra forma es considerar las funciones de membresía como representación del grado en que el objeto tiene algunos atributos. Este concepto de grado de atributo se expresa mediante el significado alterno de la función de membresía como una función de compatibilidad, donde el término *compatibilidad* indica qué tan bien se adecúa un objeto a algún atributo y es realmente mejor para describir conjuntos borrosos, sin embargo, el término *función de membresía* se utiliza con mayor frecuencia en la literatura y así se le nombra en este texto. Al pensar acerca de los conjuntos borrosos, tal vez resulte útil considerar a los elementos borrosos como una terna objeto-atributo-valor, como se describe en el capítulo 2. En el caso de los conjuntos definidos, sólo existe el objeto-atributo porque se supone que el valor es 0 o 1, esto es, en los conjuntos definidos, un elemento se encuentra o no en un conjunto. En el caso de conjuntos borrosos, el valor podría estar en cualquier sitio entre el rango de 0 a 1.

bajo
medio
alto
muy
no
más o menos
pequeño
varios
pocos
muchos
más
el más
acerca
aproximadamente
una especie de
una gran cantidad

Tabla 5.8 Algunos términos borrosos del lenguaje natural

muy bajo
más o menos bajo
aproximadamente bajo
no bajo
no muy bajo
no más o menos bajo
de medio a una especie de alto
más alto que ligeramente bajo
de bajo a una especie de medio
el más alto

Tabla 5.9 Términos borrosos compuestos del lenguaje natural

Para ilustrar el concepto de conjuntos borrosos, consideremos de nuevo el ejemplo anterior

"Juan es alto".

Si la persona es un adulto entonces una posible función de membresía se muestra en la figura 5.8. Se considera que cualquiera que mida alrededor de 2.10 metros o más, tiene una función de membresía de 1.0; y cualquiera que mida menos de 1.50 metros no está considerado en el conjunto borroso ALTO y, por lo tanto, la función de membresía es 0. Entre 1.50 y 2.10 metros, la función de membresía aumenta monótonamente con la estatura.

Esta función de membresía en particular es sólo una de las muchas funciones posibles, que serán muy diferentes para la persona promedio, los jugadores de básquetbol, los jockeys, etc. Por ejemplo, un jockey de 1.50 metros de alto se considera alto en algún grado para esa actividad, a pesar de que la función de membresía para una persona de 1.50 de altura es 0 en la figura 5.8.

Dependiendo en la aplicación, una función de membresía podría construirse a partir de las opiniones de una persona o de un grupo. En un sistema experto, esta función se construirá a partir de la opinión del especialista con el que se modela el sistema. Aunque no es probable que se modele la opinión acerca de la estatura en un sistema experto, pueden modelarse muchas otras opiniones. Algunos ejemplos podrían ser el riesgo de crédito para un préstamo, un intento hostil de una aeronave desconocida, la calidad de un producto, la aptitud de un candidato a un trabajo, etc. Observe que estas opiniones no son sólo casos de sí o no. Aunque es posible establecer valores de umbral para una decisión de sí o no, existe un cuestionamiento real como la validez de un umbral definido. Por ejemplo, ¿debe rechazarse a una persona para un préstamo de hipoteca porque sus ingresos son de $29,999.99 y el umbral es de $30,000.00?

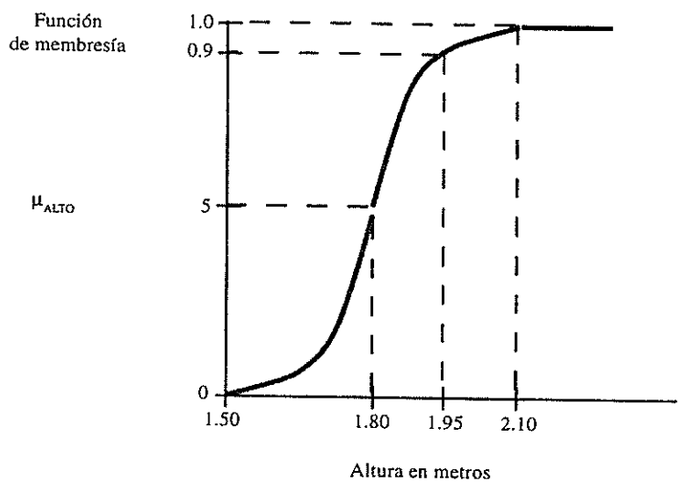


Figura 5.8 Función de membresía para el conjunto difuso ALTO

Intuitivamente, también puede considerarse la función de membresía para un grupo de personas a partir de un sondeo de opinión. Suponiendo que se le pide a un grupo de personas en la calle que especifique un valor mínimo para la palabra *alto*, es probable que nadie diga que alguien que mida menos de 1.50 es alto. Del mismo modo, es probable que todos digan que alguien que mide 2.10 o más es alto. Entre 1.50 y 2.10 metros, el porcentaje de personas que está de acuerdo en lo que constituye *alto*, es análogo a la curva de la función de membresía que se muestra en la figura 5.8. A medida que el valor de alto aumenta de 1.50 a 1.80 metros, más y más gente estaría de acuerdo en que alguien es alto. Para esta función de membresía en particular, el **punto de cruce** para alto es 1.80; un punto de cruce es donde $\mu = 0.5$. Tomando como punto de vista nuestra analogía de la encuesta de opinión, el 50 por ciento de las personas estaría de acuerdo en que alguien que mide 1.80 o más es alto; a 1.95, el porcentaje de personas que

estaría de acuerdo es de 90 por ciento y a más de 2.10 metros, todos estarían de acuerdo en la altura, por lo tanto, la función de membresía es 1.

Es importante darse cuenta de que, a pesar de que este ejemplo está dado en términos de una encuesta de opinión entre un grupo de personas, la función de membresía en realidad *no* es una distribución de frecuencia. Como se analiza en el capítulo 4, las probabilidades se utilizan para observaciones repetibles de los mismos objetos idénticos. Aunque cada persona del grupo podría dar una opinión repetible cuando se le hace otra vez la pregunta, las opiniones son posibilidades porque expresan una creencia personal.

La **función S** es una función matemática que se utiliza a menudo en los conjuntos borrosos, como una función de membresía. Se define como sigue:

$$S(x;\alpha,\beta,\gamma) = \begin{cases} 0 & \text{para } x \le \alpha \\ 2\left(\dfrac{x-\alpha}{\gamma-\alpha}\right)^2 & \text{para } \alpha \le x \le \beta \\ 1 - 2\left(\dfrac{x-\gamma}{\gamma-\alpha}\right)^2 & \text{para } \beta \le x \le \gamma \\ 1 & \text{para } x \ge \gamma \end{cases}$$

En la figura 5.9 se muestra una gráfica de la función

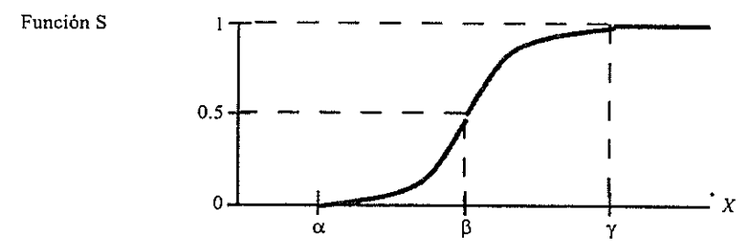


Figura 5.9 La función S

La función S puede ser una herramienta valiosa en la definición de las funciones borrosas como la palabra *alto*. En lugar de mantener una tabla de datos que defina la función de membresía, todos los datos pueden representarse de manera compacta con una fórmula. En esta definición, α, β y γ son parámetros que pueden ajustarse hasta adecuarse a los datos de membresía deseados. Dependiendo de los datos de membresía dados, sería posible dar un ajuste exacto o aproximado para algunos valores de α, β y γ. Por supuesto, la función de membresía puede definirse simplemente como una función S sin referencia a ningún dato de alguna tabla. Otras funciones, como las triangulares, también pueden definirse dependiendo de la aplicación.

La función S es plana a un valor de 0 para $x \le \alpha$ y de 1 para $x \ge \gamma$, y cuadrática con respecto a x entre α y γ. Como puede verse en la figura 5.9, el parámetro β corresponde al punto de cruce de 0.5 y es $(\alpha + \gamma)/2$. Para la función de membresía ALTO de la figura 5.8, la función S es la siguiente:

$$(1)\quad S(x;\ 5,\ 6,\ 7) = \begin{cases} 0 & \text{para } x \le 5 \\ 2\left(\dfrac{x-5}{7-5}\right)^2 = \dfrac{(x-5)^2}{2} & \\ & \text{para } 5 \le x \le 6 \\ 1 - 2\left(\dfrac{x-7}{7-5}\right)^2 = 1 - \dfrac{(x-7)^2}{2} & \\ & \text{para } 6 \le x \le 7 \\ 1 & \text{para } x \ge 7 \end{cases}$$

En la figura 5.10 se muestra una función de membresía para la proposición difusa "X está cerca de γ". Por ejemplo, esta función de membresía podría representar todos los números cercanos a un valor específico γ, como en "X está cerca de 6", donde X podría definirse como {5.9,6,6.1}. Esta función puede expresarse como:

$$\mu_{\text{CERCA}}(x) = \frac{1}{1 + \left(\dfrac{x-\gamma}{\beta}\right)^2}$$

con puntos de cruce

$$x = \gamma \pm \beta$$

El parámetro β está a la **mitad de la anchura** de la curva en el punto de cruce, como se muestra en la figura 5.10. Los valores más grandes de β corresponden a una curva más ancha y valores más pequeños corresponden a una más angosta. Un β más grande significa que los números deben estar más cerca de γ para tener un valor de membresía significativamente más grande. Observe que en esta definición la función de membresía sólo va de cero a infinito.

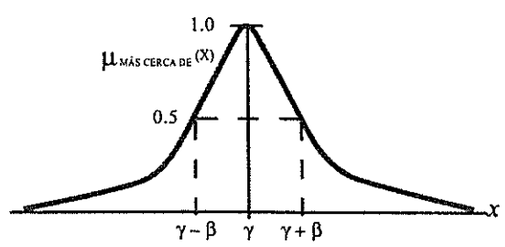


Figura 5.10 Función de membresía para la proposición difusa "X está cerca de γ"

La siguiente es una función que también da una curva similar pero que llega a cero en los puntos específicos:

$$\Pi(x;\ \beta,\ \gamma) = \begin{cases} S(x;\ \gamma-\beta,\ \gamma-\beta/2,\ \gamma) & \text{Para } x \le \\ 1 - S(x;\ \gamma,\ \gamma+\beta/2,\ \gamma+\beta) & \text{Para } x \ge \end{cases}$$

Esta **función** Π aparece graficada en la figura 5.11. Nótese que el parámetro β ahora es el **ancho de banda** o **ancho total** en el punto de cruce. La función Π tiende a cero en los puntos:

$$x = \gamma \pm \beta$$

mientras que los puntos de cruce están en:

$$x = \lambda \pm \frac{\beta}{2}$$

Π-Función

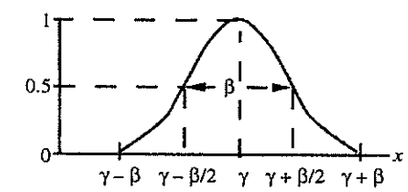


Figura 5.11 La función Π

En lugar de una función continua, la función de membresía puede ser un conjunto finito de elementos. Por ejemplo, en el universo de estaturas definido como:

U = {1.50,1.65,1.80,1.95,2.10,2.25,2.40}

Un subconjunto borroso podría estar definido por un conjunto finito de elementos para ALTO como sigue:

ALTO = {0/1.50,.125/1.65,.5/1.80,.875/1.95,1/2.10,1/2.25,1/2.40}

En este conjunto difuso el símbolo "/" separa los grados de membresía de los números que corresponden a las estaturas. Tome nota de que la diagonal no significa división en la notación personalizada de conjuntos borrosos. Los elementos del conjunto borroso para el que $\mu(x) > 0$ integran el **sustento** del conjunto; para ALTO, el sustento son todos los elementos, excepto 0/1.50.

Un subconjunto borroso finito con N elementos, se representa en notación borrosa estándar como la unión de elementos únicos borrosos μ_i/x_i, donde los signos "+" son una unión booleana.

$$(2)\quad D = \mu_1/x_1 + \mu_2/x_2 + \ldots \mu_N/x_N$$

$$D = \sum_{i=1}^{N} \mu_i/x_i$$

$$D = \bigcup_{i=1}^{N} \mu_i/x_i$$

En algunos artículos de investigación, el símbolo "/" no se escribe y, así, D puede escribirse en la forma

$$(3)\quad D = \mu_1 x_1 + \mu_2 x_2 + \ldots \mu_n x_N$$

$$D = \sum_{i=1}^{N} \mu_i x_i$$

$$D = \bigcup_{i=1}^{N} \mu_i x_i$$

Ambas ecuaciones (2) y (3) representan subconjuntos difusos finitos de N elementos. La ecuación (3) resulta útil para escribir conjuntos borrosos en una forma simbólica compacta, sin embargo, su forma es difícil de leer cuando incluye números, como en:

$$D = .127 + .385$$

Sin el separador "/", es imposible saber si los grados de membresía son .1 y .3 o .12 y .38, o algo más, por esto la notación:

$$D = .1/27 + .38/5$$

Es mejor cuando se trata con números.

El **sustento** de un conjunto borroso, D, es un subconjunto del conjunto universo, X, definido como:

$$\text{sustento}(D) = \{x \mid x \in X \text{ y } \mu_D(x) > 0\}$$

El conjunto de sustento por lo general se escribe con la abreviatura **sust**, como en:

$$\text{sust}(D)$$

para que sea más compacto. La ventaja del sustento es que un conjunto borroso D puede escribirse como:

$$D = \{\mu_D(x)/x \mid x \in \text{sust}(D)\}$$

lo que significa que sólo aquellos elementos borrosos cuya función de membresía es mayor que cero contribuyen a D. Por tanto, el conjunto ALTO puede escribirse sin el elemento 0/1.50, porque éste no se encuentra en el conjunto de sustento.

$$\text{ALTO} = \{.125/1.65, .5/1.80, .875/1.95, 1/2.10, 1/2.25, 1/2.40\}$$

Aunque sólo hay un pequeño ahorro de un elemento, la reducción puede ser significativa en los conjuntos borrosos con muchos elementos de membresía cero.

Un concepto relacionado con el sustento son los **cortes** α, el corte α de un conjunto es un conjunto no borroso del universo cuyos elementos tienen una función de membresía mayor o igual a algún valor α.

$$D_\alpha = \{x \mid \mu_D(x) \geq \alpha\} \qquad \text{para } 0 < \alpha \leq 1$$

Algunos cortes α del conjunto ALTO son

$$\text{ALTO}_{0.1} = \{1.65, 180, 1.95, 2.10, 2.25, 240\}$$

$$\text{ALTO}_{0.5} = \{180, 1.95, 2.10, 2.25, 240\}$$

$$\text{ALTO}_{0.8} = \{1.95, 2.10, 2.25, 240\}$$

$$\text{ALTO}_{1} = \{2.10, 2.25, 240\}$$

Observe que los cortes α de un conjunto son subconjuntos del sustento. Los valores de α pueden elegirse arbitrariamente pero suelen designarse para seleccionar los subconjuntos deseados del universo.

Otro término que se utiliza con frecuencia en los conjuntos borrosos es el de **altura**, que es el grado de membresía máximo de un elemento. Para nuestro conjunto ALTO, el grado de membresía máximo es 1. Si un elemento de un conjunto difuso contiene el grado máximo posible, entonces al conjunto se le llama ***normalizado***. Por lo general, los grados de membresía se definen en el intervalo cerrado [0,1] y así el grado de membresía máximo posible es 1, sin embargo, los grados pueden definirse en otros intervalos y así el grado de membresía no necesariamente sería 1.

Un subconjunto borroso arbitrario del universo sobre el **continuum** se escribe en forma de integral. El término *continuum* se refiere al conjunto de los números reales y la integral representa la unión de los elementos borrosos únicos, $\mu(x)/x$. Por ejemplo, podemos definir:

$$\text{ALTO} = \int_X \mu_{\text{ALTO}}(x)/x$$

$$= \int_5^8 \mu_{\text{ALTO}}(x)/x$$

$$\int_5^6 \frac{(x-5)^2}{2}/x + \int_6^7 \left[1 - \frac{(x-7)^2}{2}\right]/x + \int_7^8 1/x$$

utilizando una función S para ALTO. En esta fórmula, el signo "+" que separa las integrales representa la unión como en la notación de la lógica booleana, más que la suma aritmética.

Hay diferentes tipos de conjuntos borrosos, como el **subconjunto borroso de tipo 1**, D, de un universo, X, que se define como sigue:

$$\mu_D: X \rightarrow [0,1]$$

Es decir, un subconjunto borroso de tipo 1 se define simplemente dando los valores numéricos para su función de membresía en el intervalo cerrado de los números reales entre 0 y 1. Por ejemplo:

$$\text{ALTO} = .125/1.65 + .5/1.80 + .875/1.95 + 1/2.10 + 1/2.25 + 1/2.40$$

es un conjunto de tipo 1 porque sus grados de membresía son números reales en [0,1]. De igual manera, de la ecuación (1):

$$\mu_{ALTO}(x) = S(x;\ 1.50,\ 1.80,\ 2.10)$$

es un subconjunto borroso de tipo 1.

En general, un **subconjunto borroso de tipo N** se define al mapear el μ_D del universo al conjunto de subconjuntos borrosos de tipo N – 1. Por ejemplo, un **subconjunto borrosos de tipo 2** se define a partir de un subconjunto de tipo 1. Para las estaturas, un conjunto difuso de tipo 2 puede ser:

$$\mu_{ALTO}(5) = \text{MENOR AL PROMEDIO}$$

$$\mu_{ALTO}(6) = \text{PROMEDIO}$$

$$\mu_{ALTO}(7) = \text{MAYOR AL PROMEDIO}$$

donde MENOR AL PROMEDIO, PROMEDIO y MAYOR AL PROMEDIO son subconjuntos difusos de tipo 1. Esto podría definirse como los siguientes subconjuntos difusos:

$$\mu_{\text{MENOR AL PROMEDIO}}(x) = 1 - S(x;\ 1.35,\ 1.50,\ 1.65)$$

$$\mu_{\text{PROMEDIO}}(x) = \Pi\ (x;\ 1,\ 1.65)$$

$$\mu_{\text{MAYOR AL PROMEDIO}}(x) = S(x;\ 1.65,\ 1.80,\ 1.95)$$

Operaciones con conjuntos borrosos

Un conjunto definido común es un caso especial de un conjunto borroso con función de membresía {0,1}. Todas las definiciones, pruebas y teoremas de los conjuntos borrosos deben ser compatibles hasta el límite en el que lo borroso tiende a 0 y los conjuntos borrosos se vuelven conjuntos definidos. Por consiguiente, la teoría de los conjuntos borrosos tiene un rango más amplio de aplicaciones que la de los conjuntos definidos y, por tanto, puede tratar con situaciones en las que intervienen opiniones subjetivas. La idea básica de los conjuntos borrosos consiste en especificar conceptos borrosos reales como ALTO para un conjunto de elementos borrosos en lugar de exigir un umbral binario tajante. A continuación se presenta un resumen de algunos de los operadores de los conjuntos borrosos en un universo X.

- **igualdad de conjunto**

  $A = B$

  $\mu_A(x) = \mu_B(x)$ para toda $x \in X$

- **complemento de conjunto**

  A'

  $\mu_A'(x) = 1 - \mu_A(x)$ para toda $x \in X$

En la figura 5.12 se ilustra el complemento borroso de un conjunto. Esta definición de complementación está justificada por Bellman y Giertz (Bellman 1973).

- **contención del conjunto**

  $A \subseteq B$

El conjunto borroso A está contenido en B o es un **subconjunto** de B, si y sólo si:

$$\mu_A(x) \le \mu_B(x) \quad \text{para toda } x \in X$$

El conjunto borroso A es un **subconjunto propio** de B si A es un subconjunto de B y los dos son diferentes.

$$\mu_A(x) \le \mu_B(x) \text{ y } \mu_A(x) < \mu_B(x)$$
$$\text{para por lo menos una } x \in X$$

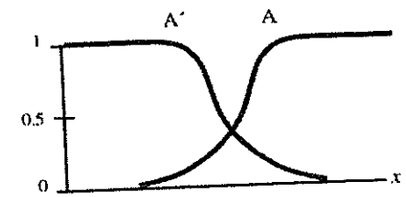


Figura 5.12 Complemento borroso

- **unión de conjuntos**

  $A \cup B$

  $\mu_{A \cup B}(x) = \vee(\mu_A(x), \mu_B(x))$ para toda $x \in X$

donde el **operador de unión**, $\vee$, significa el máximo de los argumentos.

- **intersección de conjunto**

  $A \cap B$

  $\mu_{A \cap B}(x) = \wedge(\mu_A(x), \mu_B(x))$ para toda $x \in X$

donde el **operador de cumplimiento**, $\wedge$, significa el mínimo de los argumentos.

Ya se han encontrado los operadores de unión y cumplimiento cuando se analizó la aplicación de la lógica borrosa a la combinación de antecedentes en PROSPECTOR, en la subsección "Combinación de evidencia con lógica borrosa" de la sección 4.13. La justificación para el uso de funciones máx y mín para la unión y el cumplimiento está en (Bellman 1973).

Con estas definiciones, las propiedades normales de los conjuntos definidos (conmutatividad, asociatividad, etc.), también se aplican a los conjuntos borrosos, con la excepción de las leyes del medio excluido y la contradicción. Por tanto, es posible que para un conjunto borroso A:

$$A \cup A' \ne U \text{ y } A \cap A' \ne \varnothing$$

donde $\varnothing$ es el conjunto vacío y U es el universo. Como los conjuntos borrosos pueden tener límites no definidos, la única restricción a la intersección es que:

$$A \cap A' = \text{mín}(\mu_A(x), \mu_{A'}(x)) \le 0.5$$

si $\mu(x)$ está definido en el rango cerrado [0,1] y $\mu A'(x) = 1 - \mu_A(x)$.

De igual manera, la única restricción en la unión es

$$A \cup A' = \text{máx}(\mu_A(x), \mu_{A'}(x)) \geq 0.5$$

Como los conjuntos borrosos A y A′ pueden superponerse, tal vez no cubran por completo el universo.

Si las leyes del medio excluido y la contradicción se definen para contener conjuntos borrosos, entonces tal vez no se satisfagan la inmutabilidad y la distribución. Para la inmutabilidad, se tiene:

$$A \cup A \neq A \qquad A \cap A' \neq A$$

Debido a la naturaleza borrosa de los conjuntos, parece más razonable aceptar que las leyes del medio excluido y la contradicción no se sostengan para que se satisfaga la inmutabilidad.

- **producto de conjuntos**

$$A\ B$$

$$\mu_{AB}(x) = \mu_A(x)\mu_B(x)$$

- **potencia de un conjunto**

$$A^N$$

$$\mu_{A^N}(x) = (\mu_A(x))^N$$

- **suma probabilística**

$$A \hat{+} B$$

$$\mu_{A \hat{+} B}(x) = \mu_A(x) + \mu_B(x) - \mu_A(x)\mu_B(x)$$

$$= 1 - (1 - \mu_A(x))(1 - \mu_B(x)$$

donde los signos "+" y "–" son operadores aritméticos comunes.

- **suma delimitada** o **unión fuerte**

$$A \oplus B$$

$$\mu_{A \oplus B}(x) = \wedge\ (1, (\mu_A(x) + \mu_B(x)))$$

donde el $\wedge$ es la función mín y el "+" es el operador aritmético.

- **producto delimitado** o **intersección fuerte**

$$A\ ß\ B$$

$$\mu_{A.B}(x) = \vee\ (0, (\mu_A(x) + \mu_B(x) - 1))$$

donde $\vee$ es la función máxima y "+" es el operador aritmético. Los operadores de suma y producto delimitados no satisfacen la inmutabilidad, la distributividad y la absorción, pero sí satisfacen la conmutatividad, la asociatividad, las leyes de Morgan, $A \oplus U = U$, $A\ ß \varnothing = \varnothing$ y las leyes del medio excluido y la contradicción (consulte el apéndice C para conocer un resumen).

- **diferencia delimitada**

$$A\ |-|\ B$$

$$\mu_{A|-|B}(x) = \vee(0, \mu_A(x) - \mu_B(x)))$$

donde el signo "–" que separa a μ_A y μ_B es el operador de resta aritmética. A | – | B representa aquellos elementos que están más en A que en B. El complemento puede escribirse en términos del conjunto universal U, y la diferencia delimitada como:

$$A' = U\ |-|\ A$$

- **concentración**

$$\text{CON}\ (A)$$

$$\mu_{\text{CON}(A)}(x) = (\mu_A(x))^2$$

La operación CON concentra elementos borrosos al reducir más los grados de membresía de los elementos que tienen grados más pequeños de membresía, en la figura 5.13 se ilustra esta operación. Las operaciones DIL, NORM e INT no tienen contraparte en las operaciones ordinarias con conjuntos. El operador CON puede usarse para aproximarse de manera burda al efecto del modificador lingüístico *Muy*. Es decir, para algún conjunto borroso D:

$$\text{Muy } D = D^2$$

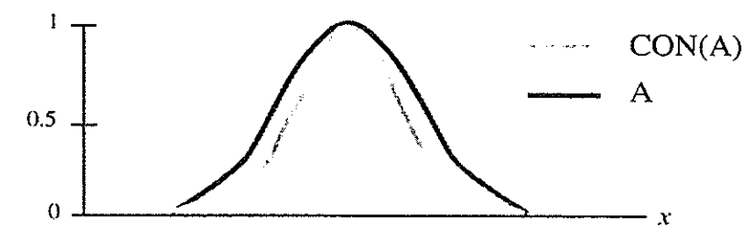


Figura 5.13 Concentración de un conjunto borroso

Usando el ejemplo del conjunto ALTO:

```
ALTO = .125/1.50 + .5/1.80 + .875/1.95 + 1/2.10 + 1/2.25 +
       1/2.40
```

Entonces:

```
Muy ALTO = .016/1.50 + .25/1.80 + .76/1.95 +
           1/2.10 + 1/2.25 + 1/2.40
```

Observe como los grados de membresía han disminuido para todos, excepto para el grado 1. El efecto neto es hacer que el conjunto borroso Muy ALTO incluya menos grados pequeños.

- **dilatación**

DIL(A)

$$\mu_{DIL(A)}(x) = (\mu_A(x))^{0.5}$$

La operación DIL dilata los elementos borrosos al aumentar más el grado de membresía de elementos con menores grados de membresía, como se ilustra en la figura 5.14. Observe que realiza la operación inversa a la concentración para aquellas opciones de potencias 2 y 0.5.

A = DIL(CON(A)) = CON(DIL(A))

El operador de dilatación se aproxima de manera burda al modificador *Más o menos*. Por tanto, para cualquier conjunto borroso D:

Más o menos D = $D^{0.5}$ = DIL(D)

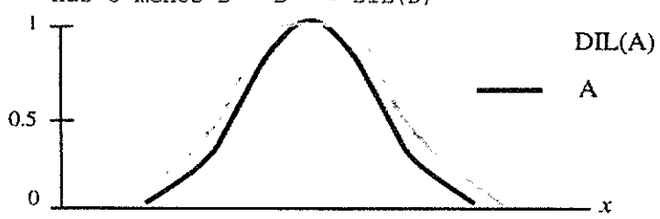


Figura 5.14 Dilatación de un conjunto borroso

- **intensificación**

INT(A)

$$\mu_{INT(A)}(x)\begin{cases} 2(\mu_A(x))^2 & \text{para } 0 \le \mu_A(x) \le 0.5 \\ 1 - 2(1 - \mu_A(x))^2 & \text{para } 0.5 < \mu_A(x) \le 1 \end{cases}$$

La operación INT es parecida a contrastar la intensificación de una fotografía. Como se ilustra en la figura 5.15, la intensificación eleva el grado de membresía de aquellos elementos

que se encuentran dentro de los puntos de cruce y reduce el grado de membresía de aquellos que se en-cuentran fuera. Haciendo una analogía electrónica, puede considerarse a los puntos de cruce como la definición del ancho de banda de una señal. La intensificación amplifica la señal dentro del ancho de banda mientras que reduce el "ruido" fuera de él. De este modo, la intensificación incrementa el contraste de aquellos elementos que se encuentran dentro de los puntos de cruce.

Figura 5.15 Intensificación de un conjunto borroso

- **normalización**

NORM(A)

$$\mu_{NORM(A)}(x) = \mu_A(x)/\text{máx}\{\mu_A(x)\}$$

donde la función máx devuelve el grado de membresía máximo para todos los elementos x. Si el grado máximo es < 1, entonces todos los grados de membresía permanecen sin cambio.

Relaciones borrosas

Un concepto importante que también puede modelarse con conjuntos borrosos es el de una **relación**. La idea intuitiva de una relación es la de alguna asociación entre elementos; algunos ejemplos de relaciones son:

Bob y Ellis son *amigos*
Los Ángeles y Nueva York están muy distantes
1, 2, 3 y 4 son *mucho menos* que 100
1, 2 y 3 son números *pequeños*
las manzanas y las naranjas son *una especie de frutas redondas*

donde las palabras en cursivas son difusas.

El **producto cartesiano** de N conjuntos definidos es el conjunto definido cuyos elementos están ordenados en N-tuplos $(x_1, x_2, x_3, \ldots x_N)$, donde cada x_i es un elemento de este conjunto X_i. Para dos conjuntos A y B:

$$A \times B = \{(a,b) \mid a \in A \text{ y } b \in B\}$$

Definiendo

A = {chocolate, fresa}
B = {pay, leche, dulce}

entonces el producto cartesiano es:

```
A × B = {(chocolate, pay), (chocolate, leche)
         (chocolate, dulce), (fresa, pay)
         (fresa, leche), (fresa, dulce)}
```

Observe que B × A no suele ser igual a A × B si A y B tienen elementos diferentes. Es decir, generalmente (a,b) ≠ (b,a). Se dice que A × B define una **variable binaria**, (a,b).

Una relación, R, es un subconjunto del producto cartesiano. Por ejemplo, la relación de que a R = LE GUSTA EL PAY podría definirse como un subconjunto de A × B mediante:

```
R = {(chocolate, pay), (fresa, pay)}
```

mientras que la relación R = LE GUSTAN LOS DULCES podría definirse como

```
R = {(chocolate, pay), (chocolate, dulce)
     (fresa, pay), (fresa, dulce)}
```

A una relación a veces también se le llama ***mapeado*** porque asocia elementos de un dominio con otro. En A × B, la relación es un mapeo de A → B, donde → significa un mapeo en este contexto. El chocolate en A está mapeado o asociado con el pay y el dulce en B, igual que con la fresa.

Si X y Y son conjuntos universales, entonces:

$$R = \{ \mu_R(x,y)/(x,y \mid (x,y) \subseteq X \times Y\}$$

es una relación borrosa en X × Y.

Una relación borrosa es fundamentalmente un subconjunto difuso en el universo del producto cartesiano. Otra definición de los conjuntos borrosos es útil cuando se trata con gráficas borrosas (Zimmerman 85).

La **relación borrosa** para N conjuntos se define como una extensión de la relación definida para incluir el grado de membresía. Es decir:

$$R = \{\mu_R(x_1, x_2, \ldots x_N)/(x_1, x_2 \ldots x_N) \mid x_i \in X_i, i = 1, .. N\}$$

que asocia el grado de membresía de cada N-tuplo. Para la relación binaria del ejemplo del pay, otra definición podría ser:

```
RE = {.9/(chocolate, pay), .2/(fresa, pay)}
```

como la relación borrosa para indicar que a una persona le gusta más el chocolate que el pay de fresa.

Una manera conveniente de representar una relación es mediante una matriz. Para la relación LE GUSTAN LOS DULCES:

$$M_R = \begin{array}{c} \\ \text{chocolate} \\ \text{fresa} \end{array} \begin{array}{c} \begin{array}{cc} \text{pay} & \text{dulce} \end{array} \\ \begin{bmatrix} 0.9 & 0.7 \\ 0.2 & 0.1 \end{bmatrix} \end{array}$$

donde M_R es la representación de la matriz de la relación difusa. Observe que para los conjuntos definidos, M_R sólo incluiría ceros y unos para expresar si le gustan o le disgustan totalmente los pares ordenados (sabor, dulce). (Ésta es probablemente la mejor prueba de que el mundo es realmente más borroso que definido.)

La **composición** de las relaciones es el efecto neto de aplicar una relación después de otra. Para el caso de las relaciones binarias P y Q, la composición de sus relaciones es la relación binaria R.

$$R(A,C) = Q(A,B) \circ P(B,C)$$

donde:

R(A,C) es una relación entre A y C
Q(A,B) es una relación entre A y B
P(B,C) es una relación entre B y C
A, B y C son conjuntos

y o es el **operador de composición.** La relación de R es la misma que aplicar P primero seguido de Q. En términos de grados de membresía:

$$R = \{\mu_R(a,c)/(a,c) \mid a \in A, c \in C\}$$

donde μ_R se define como sigue:

$$\mu_R(a.c) = \bigvee_{b \in B} [\mu_Q(a,b) \wedge \mu_P(b,c)]$$

$$= \text{máx}_{b \in B} [\text{mín} (\mu_Q(a,b), \mu_P(b,c))]$$

Esta composición suele definirse mediante el **producto de la matriz máx-mín,** o simplemente **máx-mín**. Las funciones máx y mín pueden utilizarse en lugar de la suma y la multiplicación para operaciones de matriz. Por ejemplo, definidas:

$$Q = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{bmatrix} \qquad P = \begin{bmatrix} 0.1 & 0.3 & 0.5 \\ 0.2 & 0.0 & 0.4 \end{bmatrix}$$

luego la composición, R, es:

$$R = Q \circ P = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{bmatrix} \circ \begin{bmatrix} 0.1 & 0.3 & 0.5 \\ 0.2 & 0.0 & 0.4 \end{bmatrix}$$

$$R = Q \circ P = \begin{bmatrix} \text{máx}(0.1, 0.2) & \text{máx}(0.1, 0.0) & \text{máx}(0.1, 0.2) \\ \text{máx}(0.1, 0.2) & \text{máx}(0.3, 0.0) & \text{máx}(0.3, 0.4) \end{bmatrix}$$

$$= \begin{bmatrix} 0.2 & 0.1 & 0.2 \\ 0.2 & 0.3 & 0.4 \end{bmatrix}$$

Otras definiciones del operador relacional común son el **producto máx** y la **unión relacional** (Klir 88).

Las relaciones borrosas tienen aplicaciones importantes para el razonamiento de aproximación, como se muestra más adelante. La **proyección** de una relación es otro concepto útil en los conjuntos borrosos; básicamente la proyección elimina elementos específicos. Antes de dar la definición formal, veamos un ejemplo simple en la tabla 5.10.

	y1	y2	y3	1ª proyección	
x_1	0.2	0.1	0.2	0.2	
x_2	0.2	0.3	0.4	0.4	
2ª proyección	0.2	0.3	0.4	0.4	Total proyección

Tabla 5.10 Una relación y sus proyecciones

En la tabla 5.10 se muestran las proyecciones de la relación anterior para R, donde, por conveniencia, se les han dado identificadores x_i y y_j a las filas y columnas. Observe que la columna rotulada *1ª proyección* contiene el grado máximo de membresía para la fila; de igual manera, la fila rotulada *2ª proyección* contiene el grado máximo de membresía para cada columna. La celda con valor 0.4 en la esquina inferior derecha rotulada *Total proyección*, es el grado máximo de membresía de la relación total.

La relación para R puede escribirse como sigue, en términos de x_i y y_j:

$$R = .2/x_1,y_1 + .1/x_1,y_2 + .2/x_1,y_3 + .2/x_2,y_1 + .3/x_2,y_2 + .4/x_2,y_3$$

La primera proyección está denotada por R_1 y se obtiene al colocar todo, excepto el primer punto, x_i, del par cartesiano x_i,y_j, donde el elemento del extremo izquierdo se define primero.

$$R_1 = .2/x_1 + .1/x_1 + .2/x_1 + .2/x_2 + .3/x_2 + .4/x_2$$

La ecuación para R_1 se reduce aún más después de la proyección, porque el "+" representa el operador de unión y así sólo se conserva el máximo elemento borroso.

$$R_1 = .2/x_1 + .4/x_2$$

De igual manera, la 2ª proyección sólo retiene el segundo punto, y_j, de cada par cartesiano:

$$R_2 = .2/y_1 + .1/y_2 + .2/y_3 + .2/y_1 + .3/y_2 + .4/y_3$$

que se reduce a lo siguiente después de la aplicación del operador de unión:

$$R_2 = .2/y_1 + .3/y_2 + .4/y_3$$

En el caso general de una relación que relaciona N puntos cartesianos, coloque todos los componentes del N-tuplo, excepto aquellos puntos en los que habrá de realizarse la proyección. Por ejemplo, si la relación es sobre N puntos, entonces R_{136} colocará todos los puntos, con excepción del primero, el tercero y el sexto.

Para una relación que abarca el universo X × Y:

$$R = \{\mu_R(x,y)/(x,y)\} \text{ para todo } (x,y) \in X \times Y$$

La 1ª proyección se define como:

$$\text{proy}(R;X) = R_1$$

donde:

$$R_1 = \{\text{máx}_y\ \mu_R(x,y)/x \mid (x,y) \in X \times Y\}$$

y se toma el máximo sobre todo *y*. De igual manera:

$$\text{proy}(R;Y) = R_2$$

donde:

$$R_2 = \{\text{máx}_x\ \mu_R(x,y)/y \mid (x,y) \in X \times Y\}$$

donde el máximo se toma sobre todos los valores *x* de R_2.

La **extensión cilíndrica** de la relación de proyección se define como la relación borrosa más grande que es compatible con la proyección. La proyección cilíndrica es de alguna manera análoga a la proyección porque expande el valor proyectado (que es máx) para todos los demás elementos, por ejemplo:

$$\text{proy}(R;X) = R_1 = .2/x_1 + .4/x_2$$

de modo que

$$R_1 = .2/x_1,y_1 + .2/x_1,y_2 + .2/x_1,y_3 + .4/x_2,y_1 + .4/x_2,y_2 + .4/x_2,y_3$$

$$R_2 = .2/y_1,x_1 + .2/y_1,x_2 + .3/y_2,x_1 + .3/y_2,x_1 + .3/y_2x_2 + .4/y_2,x_1 + .4/y_3,x_2$$

es decir, reemplaza la segunda variable, $.2/x_1$,{toda var} + $.4/x_2$,{toda var}. Como la proyección proporciona el máx_μ, entonces la extensión cilíndrica es la relación más grande compatible con la proyección.

Para el ejemplo anterior,

$$\overline{R_1} = \begin{bmatrix} 0.2 & 0.2 & 0.2 \\ 0.4 & 0.4 & 0.4 \end{bmatrix}$$

$$\overline{R_2} = \begin{bmatrix} 0.2 & 0.3 & 0.4 \\ 0.2 & 0.3 & 0.4 \end{bmatrix}$$

donde la barra sobre una proyección simboliza la extensión cilíndrica de la misma.

La composición puede definirse a partir de proyecciones y extensiones cilíndricas (Zadeh 75). Para la relación binaria R, definida en el conjunto universal $U_1 \times U_2$ y S, definida en $U_2 \times U_3$, la composición es:

$$ROS = proy\ (\overline{R} \cap \overline{S}\ ;\ U_1 \times U_3)$$

Variables lingüísticas

Una aplicación muy importante de los conjuntos borrosos se hace en la **lingüística computacional**. El objetivo es calcular estados financieros de consolidación con afirmaciones del lenguaje natural, de manera análoga a como se realizan los cálculos lógicos con instrucciones lógicas. Los conjuntos borrosos y las **variables lingüísticas** pueden utilizarse para cuantificar el significado del lenguaje natural, con lo que puede manipularse. A una variable lingüística se le asignan valores que son expresiones, como palabras, frases u oraciones en un lenguaje natural o artificial. En la tabla 5.11 se muestran algunas variables lingüísticas y algunos valores típicos que podrían asignárseles.

Variable lingüística	Valores típicos
estatura	enano, bajo, promedio, alto, gigante
número	casi ninguno, varios, pocos, muchos
etapa de la vida	bebé, niño, adolescente, adulto
color	rojo, azul, verde, amarillo, anaranjado
luz	clara, tenue, normal, brillante, intensa
postre	pay, pastel, helado, pastel helado

Tabla 5.11 Valores típicos

Aunque es posible definir valores, como el color rojo, que corresponden a esos valores lingüísticos, son de naturaleza muy subjetiva; por ejemplo, el color rojo corresponde a un rango de frecuencias que el ojo percibe como rojo, no sólo a una frecuencia. Otro problema radica en colores como el aguamarina. ¿Es azul o verde?

Las variables lingüísticas se usan de manera regular en las reglas heurísticas, sin embargo, pueden estar implícitas, como se ilustra en las dos primeras reglas de la tabla 5.12.

SI la TV está demasiado clara ENTONCES aumente el brillo
SI está demasiado caliente ENTONCES agregue algo frío
SI la presión es muy alta ENTONCES abra la válvula
SI las tasas de interés suben ENTONCES compre bonos
SI las tasas de interés bajan ENTONCES compre acciones

Tabla 5.12 Algunas reglas heurísticas que se relacionan con variables lingüísticas implícitas

Las variables lingüísticas implícitas son Calidad de imagen y Temperatura del agua.

Algunas variables lingüísticas como Calidad de imagen pueden ser rótulos o conjuntos borrosos de 2° orden. Por ejemplo, los valores de Calidad de imagen pueden ser

color, tinte, brillo, ruido, consistencia

donde cada uno de estos valores pueden ser variables lingüísticas que toman valores que son conjuntos borrosos. De esta manera, las variables lingüísticas pueden organizarse en una jerarquía que corresponde a su orden de conjunto borroso. A la larga, se alcanza un conjunto borroso de 1er orden , como ALTO o BRILLO, que se define como un mapeado en el intervalo cerrado [0,1], y así el valor lingüístico se convierte en un rango numérico.

El **conjunto de términos**, T(L), de una variable lingüística, L, es el conjunto de valores que podría tomar. Por ejemplo:

T(PAY) = CHOCOLATE + MANZANA + FRESA + NUECES

donde cada término de T(PAY) es el rótulo de un conjunto borroso. Estos conjuntos pueden ser la unión de otros conjuntos que contienen subconjuntos, Por ejemplo:

CHOCOLATE = CHOCOLATE SEMI-DULCE + CHOCOLATE CON LECHE +
CHOCOLATE ALEMÁN + CHOCOLATE OSCURO + ...

Otra definición para el conjunto borroso CHOCOLATE puede relacionarse con **modificadores** para transformar el significado de un conjunto. Por ejemplo, un conjunto borroso CHOCOLATE de un tipo de chocolate podría definirse como sigue:

CHOCOLATE = Muy CHOCOLATE + Muy Muy CHOCOLATE +
Más o menos CHOCOLATE
Ligeramente CHOCOLATE
Más CHOCOLATE + No Muy CHOCOLATE + ...

Los modificadores normales pueden definirse a partir de algunas operadores para conjuntos borroso y de un conjunto borroso, D, como se muestra en la tabla 5.13.

Modificador	Definición de operador
Muy D	$CON(D) = D^2$
Más o menos D	$DIL(D) = D^{0.5}$
Más D	$D^{1.25}$
No D	$1 - D$
No muy D	$1 - CON(D)$
Ligeramente D	INT[NORM(MÁS D Y NO(MUY D))]

Tabla 5.13 Algunos modificadores lingüísticos como operadores

Como se mostró con el modificador "No muy D", pueden hacerse otros modificadores compuestos al combinar operadores. Note que el "Y" en el modificador "Ligeramente" es el operador de conjunto borroso para la intersección, $\cap$, actuando en los conjuntos borrosos "Más D" y "No(Muy D)".

En la figura 5.16 se ilustra la jerarquía de la variable lingüística "Apetito". Los conjuntos borrosos LIGERO y PESADO se suponen como funciones S, mientras que el conjunto MODERADO se toma como una función Π.

Observemos que hay superposición de los conjuntos borrosos LIGERO y MODERADO, e incluso entre LIGERO y PESADO. En los conjuntos definidos clásicos no habría superposición porque todos estos conjuntos estarían desunidos. Es decir, un apetito LIGERO no podría ser un apetito MODERADO ni PESADO. Sin embargo, en los conjuntos borrosos por lo general no hay límites claros (a menos que se definan) entre los conjuntos.

Los conjuntos con modificadores se muestran con líneas atenuadas dentro de los bordes de los conjuntos borrosos. Los modificadores como *Muy*, actúan sobre los valores lingüísticos para obtener los conjuntos borrosos Muy LIGERO, Muy MODERADO y Muy PESADO.

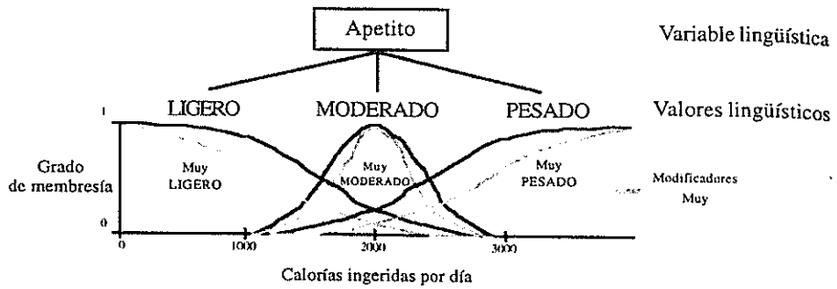


Figura 5.16 La variable lingüística "Apetito" y sus valores

Una variable lingüística debe tener sintaxis y semántica válidas, que puedan especificarse por conjuntos borrosos o reglas. Una **regla sintáctica** define las expresiones bien formadas en T(L). Por ejemplo, el conjunto de términos:

T(Edad) = {VIEJO, Muy VIEJO, Muy Muy VIEJO, ...}

Puede generarse recursivamente utilizando la siguiente regla sintáctica (Zadeh 73):

T^{i+1} = {VIEJO} ∪ {Muy T^i}

Por ejemplo:

T^0 = ∅ (el conjunto nulo)

T^1 = {VIEJO}

T^2 = {VIEJO, Muy VIEJO}

T^3 = {VIEJO, Muy VIEJO, Muy Muy VIEJO}

La **regla semántica** asociada con T(L) define el **significado** de un término, L_i, en L, para un conjunto borroso. Por ejemplo, la regla semántica para Muy VIEJO podría definirse como:

Muy VIEJO = μ^2_{VIEJO}

donde la función de membresía podría definirse como la siguiente función S:

$\mu_{VIEJO}(X)$ = S(x; 60, 70, 80)

Un **término primario** es un término como JOVEN, VIEJO, CHOCOLATE, FRESA, etc., cuyo significado puede definirse antes de un modificador. Éste cambia el significado de los términos primarios para producir otros en un conjunto de términos, como Muy JOVEN, Muy VIEJO, Muy CHOCOLATE, Ligeramente CHOCOLATE, etc. El significado de los conjuntos borrosos con modificador se determina al aplicar los operadores para conjuntos borrosos apropiados. Por ejemplo:

$\mu_{\text{Muy CHOCOLATE}} = \mu^2_{\text{CHOCOLATE}}$

$\mu_{\text{No CHOCOLATE}} = 1 - \mu_{\text{CHOCOLATE}}$

$\mu_{\text{Más o menos CHOCOLATE}} = \mu^{0.5}_{\text{CHOCOLATE}}$

$\mu_{\text{No Muy CHOCOLATE}} = 1 - \mu_{\text{Muy CHOCOLATE}}$

Al igual que la gramática convencional puede definirse con notación BNF (consulte la sección 2.2), así también puede hacerse con una **gramática borrosa**. De hecho, la gramática descrita en la sección 2.2 usa un modificador borroso, "gran" como en

```
<adjetivo> → gran
```

para generar la **producción borrosa**

```
un devorador fue el gran hombre
```

En esta producción, *gran* es un modificador del conjunto borroso *hombre*. Al principio tal vez no se piense en *hombre* como en un conjunto borroso, sin embargo, ¿Cuándo se convierte un niño en hombre? En algunas culturas un niño se vuelve hombre a los 12 años o después de una ceremonia religiosa, algunos gobiernos definen a un hombre a los 18 o 21 años, en los periódicos parece haber una política de referirse a los varones de 17 a 19 años como hombres si se les acusa de haber cometido un crimen, pero se refieren a alguien de 17 o 19 años como un joven si no ha hecho nada reprimible. En las fuerzas armadas, a los hombres a veces se les considera jóvenes y a veces hombres, sobre todo en los discursos políticos.

Puede especificarse una gramática borrosa con la notación BNF, mediante la adición de no terminales como:

```
<Frase de rango> ::= <Modificador principal> A
                     <Modificador principal>
<Modificador principal> ::= <Modificador> <Principal> |
<Principal>
<Modificador> ::= Muy | Más o menos | Un poco
<Principal> ::= CORTO | MEDIO | ALTO
```

para generar producciones como:

```
Muy CORTO A Muy ALTO
Un poco ALTO A Muy CORTO
Más o menos MEDIO A ALTO
```

Una novedosa aplicación del concepto de variable lingüística es la construcción del automóvil borroso de Sugeno, en el Instituto de Tecnología de Tokio (Zadeh 88). El automóvil de Sugeno utiliza un sistema de control, basado en la lógica borrosa, que permite su operación autónoma en una pista rectangular, puede estacionarse por sí solo en un espacio específico y también aprende con el ejemplo. En las en reglas que controlan el movimiento del auto, se usan variables lingüísticas. Se han construido muchos otros tipos de sistemas de control de

lógica borrosa para controlar dispositivos y procesos industriales como un horno para fabricar cemento (Larsen 81).

Principio de extensión

El **principio de extensión** es un concepto muy importante en la teoría de lo borroso. Este principio define la manera de extender el dominio de una función definida determinada para incluir los conjuntos borrosos. Al utilizar el principio de extensión, cualquier función ordinaria o definida de matemáticas, ciencias, ingeniería, negocios, etc., puede extenderse para trabajar en un dominio borroso con conjuntos borrosos. Este principio hace que estos últimos resulten aplicables a todos los campos.

Supongamos que f puede ser una función común que se mapea del universo X a Y. Si D es un conjunto borroso de X de modo que

$$D = \int_X \mu_D(x) / x$$

entonces el principio de extensión define la imagen del conjunto borroso D bajo un mapeo de la función f(x) como:

$$f(D) = \int_X \mu_D(x) / f(x)$$

Por ejemplo, si se define a f(x) como una función definida que eleva al cuadrado sus argumentos:

$$f(x) = x^2$$

el principio de extensión establece cómo implantar una función al cuadrado de los conjuntos borrosos:

$$f(D) = \int_X \mu_D(x) / f(x) = \int_X \mu_D(x) / x^2$$

Por ejemplo, definidos los universos X y Y como el intervalo cerrado [0,1000] y el conjunto borroso:

$$D = .3/15 + .8/20 + 1/30$$

entonces el principio de extensión define el mapeo f(D) como

$$f(D) = \int_X \mu_D(x) / f(x)$$

$$= .3/f(15) + .8/f(20) + 1/f(30)$$

$$f(D) = .3/225 + .8/400 + 1/900$$

Lógica borrosa

Así como la lógica clásica integra la base de los sistemas expertos convencionales, la lógica borrosa da forma a las bases de los **sistemas expertos borrosos**. Además de tratar con la

incertidumbre, los sistemas expertos pueden modelar **razonamiento de sentido común**, que es muy difícil para los sistemas convencionales.

Una limitante básica de la lógica clásica es su restricción a los dos valores de verdadero y falso. Como se analiza en los capítulos 2 y 3, esta restricción tienen ventajas y desventajas. La principal ventaja es que los sistemas basados en la lógica de dos valores son fáciles de modelar deductivamente y, de ese modo, las inferencias pueden ser exactas. La principal desventaja es que muy poco en la realidad tiene realmente dos valores, porque el mundo real es un mundo análogo, no digital.

Las limitantes de la lógica de dos valores se han conocido desde los tiempos de Aristóteles, quien aunque formuló primero las reglas silogísticas de la inferencia y la ley del medio excluido, reconoció que las proposiciones sobre eventos futuros no eran ni realmente verdaderas ni realmente falsas hasta que ocurrían.

Se han formulado varias teorías diferentes de la lógica basadas en valores múltiples de verdad, incluyendo las de Lukasiewicz, Bochvar, Kleene, Heyting y Reichenbach. Algunos tipos comunes son aquellos basados en los tres valores de verdad que representan VERDADERO, FALSO y DESCONOCIDO. Estas **lógicas trivalentes** suelen representar los tres valores de verdad con 1, 0 y 1/2 para VERDADERO, FALSO y DESCONOCIDO respectivamente.

Se han desarrollado varias lógicas generalizadas de N valores verdaderos, donde *N* es un número entero arbitrario mayor o igual a dos. Lukasiewicz desarrolló la lógica del primer valor N en los años 30. En la lógica del valor N, el conjunto de valores de verdad T_N se suponen divididos equitativamente en el intervalo cerrado [0,1].

$$T_N = \left\{\frac{i}{N-1}\right\} \text{ Para } 0 \le i < N$$

Por ejemplo:

$$T_2 = \{0, 1\} \quad T_3 = \{0, 1/2, 1\}$$

En la tabla 5.14 aparecen definidos algunos **operadores lógicos de Lukasiewicz** para la lógica de valor N, donde $N \ge 2$. Como se muestra en el problema 5.13, éstos se reducen a los valores lógicos normales para N = 2. Observe que los operadores menos, mín y máx son los mismos en la lógica de lo difuso.

x'	$= 1 - x$
$x \wedge y$	$=$ mín(x, y)
$x \vee y$	$=$ máx(x, y)
$x \rightarrow y$	$=$ mín$(1, 1 + y - x)$

Tabla 5.14 Operadores primitivos para la lógica de valor N de Lukasiewicz

Cada valor N de la lógica de Lukasiewicz, o **lógica L**, se escribe como L_N, donde N es el número de valores de verdad. L_2 es la lógica clásica de dos valores, mientras que en el otro extremo de $N = \infty$, la $L\infty$ define una **lógica de valor infinito** con valores de verdad en el conjunto $T\infty$. Aunque $T\infty$ está definido en los números racionales, una lógica de valor infinito alternativa puede definirse en el *continuum*, que es el conjunto de todos los números reales. El término *lógica de valor infinito* suele tomarse para esta lógica alternativa donde los valores de verdad son los números reales en [0,1] y a esta lógica se le llama L_1.

Sin embargo, L_1 *no* es lo mismo que la **lógica unitaria** con N = 1. La lógica unitaria no es una lógica de Lukasiewicz porque la **lógica L** sólo se define para $N \ge 2$. El 1 en L_1 es en realidad

una abreviatura de $\aleph_1$ (que se lee "aleph 1"), la cardinalidad de los números reales, donde $\aleph_1$ no es un número finito, sino un **número transfinito**. Cantor fue quien desarrolló primero esta teoría como una manera de realizar cálculos con números infinitos. En lugar de un número infinito, Cantor definió diferentes órdenes de lo infinito, por ejemplo, el número transfinito más pequeño es $\aleph_0$ que es la cardinalidad de los números naturales; $\aleph_1$ es un infinito de orden superior a $\aleph_0$, porque hay una cantidad infinita de números reales para cada número natural.

La lógica de lo borroso puede considerarse una extensión de la lógica de múltiples valores. Sin embargo, los objetivos y aplicaciones de la lógica borrosa son diferentes porque ésta es la lógica del **razonamiento aproximado** más que el razonamiento exacto de múltiples valores. En esencia, el **razonamiento borroso** o aproximado es la inferencia de una conclusión posiblemente imprecisa de un conjunto de premisas posiblemente imprecisas. La gente está muy familiarizada con el razonamiento aproximado porque es el tipo más común de razonamiento en la realidad y es la base de muchas reglas heurísticas. Algunos ejemplos de reglas heurísticas de razonamiento aproximado son las siguientes:

```
SI la imagen de la TV se desplaza verticalmente
ENTONCES ajuste el control vertical

SI la imagen de la TV es demasiado tenue
ENTONCES suba al control de brillo

SI el color rojo en la imagen de la TV se ve demasiado
verde
ENTONCES baje al control de tinte

SI está subiendo mucho de peso por comer banana splits y
pays y helados y pastel
ENTONCES reduzca el número de bananas
```

El razonamiento aproximado se relaciona con el razonamiento que no es exacto ni totalmente inexacto, como una suposición pura. El razonamiento aproximado se relaciona en particular con el razonamiento sobre afirmaciones en lenguaje natural y las inferencias que le siguen. La lógica de lo borroso se relaciona con el razonamiento aproximado de la misma manera que la lógica de dos valores se relaciona con el razonamiento preciso. Como ejemplo de razonamiento preciso o exacto se encuentra la deducción y la comprobación de un teorema, como se analiza en el capítulo 3.

Existen muchos tipos diferentes de teoría de conjuntos borrosos, lógica borrosa y razonamiento aproximado. El tipo de lógica borrosa analizado a partir de aquí es la teoría de Zadeh del razonamiento aproximado, que usa valores de verdad de la lógica borrosa cuya base es la lógica L_1 de Lukasiewicz. En esta lógica borrosa, los valores de verdad son variables lingüísticas que están representadas al final de cuentas por conjuntos borrosos.

En la tabla 5.15 se definen los operadores para la de lógica borrosa basados en los operadores de Lukasiewicz de la tabla 5.14. x(A) es el valor de verdad numérico en el rango [0,1] que representa la verdad de la proposición "x es A", que puede interpretarse como el grado de membresía $\mu_A(x)$.

Como ejemplo de operadores de lógica borrosa, supongamos un conjunto borroso llamado VERDADERO, definido como:

```
VERDADERO = .1/.1 + .3/.5 + 1/.8
```


El uso de los operadores de la tabla 5.15 da:

```
FALSO = 1 - VERDADERO

      = ( 1 -.1)/.1 + (1 -.3)/.5 + (1 - 1)/.8

      = .9/.1 + .7/.5
```

El uso del operador CON para el modificador Muy da

```
Muy VERDADERO = .01/.1 + .09/.5 + 1/.8

Muy FALSO = .81/.1 + .49/.5
```

$x(A')$	$= x(\text{NO } A)$	$= 1 - \mu_A(x)$
$x(A) \wedge x(B)$	$= x(A \text{ Y } B)$	$= \text{mín}(\mu_A(x), \mu_B(x))$
$x(A) \vee x(B)$	$= x(A \text{ O } B)$	$= \text{máx}(\mu_A(x), \mu_B(x))$
$x(A) \rightarrow x(B)$	$= x(A \rightarrow B)$	$= x((\sim A) \vee B) = \text{máx}[(1 - \mu_A(x)), \mu_B(x)]$

Tabla 5.15 Algunos operadores de lógica de lo borroso

Reglas borrosas

Tomando como ejemplo el de los operadores para conjuntos borrosos, consideremos el problema del reconocimiento de patrones; éstos podrían representar objetos examinados en el control de calidad, como partes manufacturadas o frutas recién cortadas. Otros tipos importantes de problemas de reconocimiento de patrones son las imágenes médicas, los datos sísmicos de exploración minera y de petróleo, y otros.

En la tabla 5.16 se muestran algunos datos hipotéticos que representan el grado de membresía en conjuntos borrosos de misiles, cazas y aviones comerciales correspondientes a algunas imágenes. Estas imágenes podrían producirse con un sistema de TV con láser de rango largo y contienen incertidumbre debido al movimiento y la orientación del blanco, el ruido, etcétera.

	Grado de membresía		
Imagen	**Misil**	**Caza**	**Comercial**
1	1.0	0.0	0.0
2	0.9	0.0	0.1
3	0.4	0.3	0.2
4	0.2	0.3	0.5
5	0.1	0.2	0.7
6	0.1	0.6	0.4
7	0.0	0.7	0.2
8	0.0	0.0	1.0
9	0.0	0.8	0.2
10	0.0	1.0	0.0

Tabla 5.16 Grados de membresía para imágenes

La unión de los conjuntos borrosos para cada imagen representa la incertidumbre total en cuanto a la identificación del objetivo. En la figura 5.17 se ilustra una unión de conjuntos

borrosos para las diez imágenes del objetivo de la tabla 5.16. Por supuesto, en una situación real habría muchas más imágenes dependiendo del sistema de resolución y la distancia del objetivo. Además de la incertidumbre en el hardware del sistema, también hay incertidumbre en los conjuntos borrosos para misiles, cazas y aviones comerciales. Los grados de membresía se asignan de manera subjetiva, con base en el conocimiento de sus configuraciones típicas. En una situación real hay muchos tipos de cada uno de estos conjuntos primitivos.

Las uniones de conjuntos borrosos que se muestran en la figura 5.17 pueden considerarse representaciones de reglas como

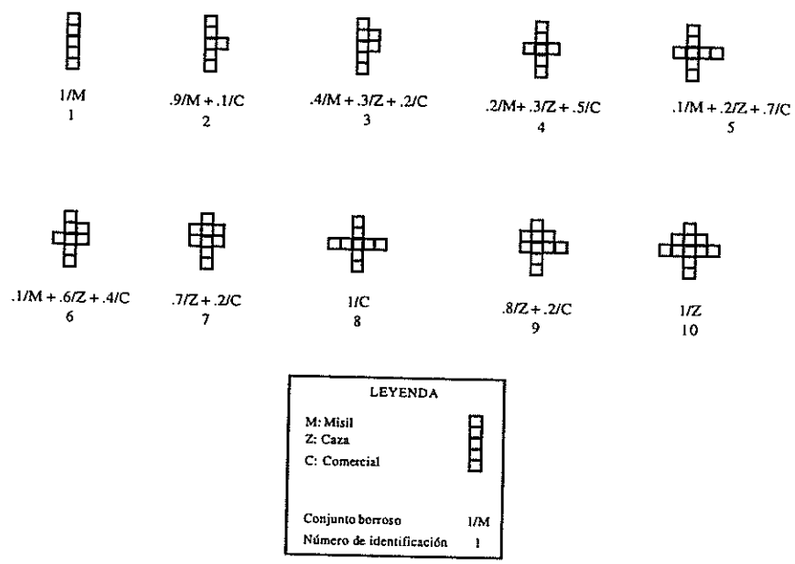


Figura 5.17 Conjuntos borrosos para identificación de aeronaves

```
SI E ENTONCES H
```

donde E es la imagen observada y H es la unión del conjunto borroso. Por ejemplo:

```
SI IMAGEN4 ENTONCES OBJETIVO (.2/M + .3/Z + .5/C)
```

donde la expresión entre paréntesis es la unión del conjunto borroso del objetivo. Como alternativa, la regla podría expresarse:

```
SI IMAGEN4 ENTONCES OBJETIVO4
```

Donde:

```
OBJETIVO4 = .2/M + .3/Z + .5/C
```


Suponiendo que hay tiempo para hacer otra observación del objetivo y que se observa IMAGEN6. Esto corresponde a una regla:

```
SI IMAGEN6 ENTONCES OBJETIVO6
```

donde:

```
OBJETIVO6 = .1/M + .6/Z + .4/C
```

Los elementos totales que se han medido para el objetivo son:

```
OBJETIVO = OBJETIVO4 + OBJETIVO6
```

donde el "+" denota la unión de conjuntos. De allí que:

```
OBJETIVO = .2/M + .3/Z + .5/C + .1/M + .6/Z + .4/C

OBJETIVO = .2/M + .6/Z + .5/C
```

donde sólo los grados de membresía máximos para cada elemento se retienen en el conjunto borroso OBJETIVO.

Si el elemento con el grado máximo se interpreta como el objetivo más probable, entonces el objetivo es más probable que sea un caza, porque tiene un grado de membresía 0.6 más alto. Sin embargo, si el grado de membresía del comercial fuera también de 0.6, entonces podríamos decir que es igualmente posible que resulte un caza o un comercial.

En general, dadas N observaciones y reglas:

SI E_1 ENTONCES H_1
SI E_2 ENTONCES H_2
.
.
.
SI E_N ENTONCES H_N

donde todas las H_i portan alguna hipótesis común H, entonces la unión de las hipótesis determina el grado de membresía de H. Es decir:

$$\mu_H = \text{máx}(\mu_{H_1}, \mu_{H_2}, \ldots \mu_{H_N})$$

Observe que este resultado difiere de las teorías de factores de certidumbre y de Dempster y Shafer. Al μ_H de la hipótesis H se le llama ***valor de verdad*** de H.

También es una suposición razonable que la verdad de la hipótesis no puede ser mayor que la verdad de su evidencia. En términos de reglas, la verdad de la consecuencia no puede ser mayor que la verdad de su antecedente. De allí que:

$$\mu_H = \text{máx}(\mu_{H_1}, \mu_{H_2} \ldots \mu_{H_N})$$

$$= \text{máx}[\text{mín}(\mu_{E_1}), \text{mín}(\mu_{E_2}), \ldots \text{mín}(\mu_{E_N})]$$

donde cada E_i puede ser alguna expresión difusa. Por ejemplo, E_i podría definirse como:

$$E_1 = E_A \text{ Y } (E_B \text{ O NO } E_C)$$

y los operadores para la lógica borrosa se usan para evaluar la expresión. Es decir.

$$\mu_{E_1} = \text{mín}(\mu_{E_A}, \text{máx}(\mu_{E_B}, 1 - \mu_{E_C}))$$

Al grado de membresía combinado del antecedente se le llama ***valor de verdad del antecedente***. Esto es análogo a la evidencia parcial del antecedente en las reglas de PROSPECTOR. De hecho, la evidencia de antecedentes de PROSPECTOR se combinó utilizando la lógica de lo borroso sobre una base *ad hoc*. Ahora puede verse que esta combinación está justificada por la regla de inferencia composicional de la teoría de lo borroso.

Composición máx-mín

La ecuación anterior para H es la **regla de la inferencia composicional máx-mín** de la lógica borrosa. En un simple caso de dos elementos de evidencia por regla:

$$\begin{array}{l} \text{SI } E_{11} \text{ y } E_{12} \text{ ENTONCES } H_1 \\ \text{SI } E_{21} \text{ y } E_{22} \text{ ENTONCES } H_2 \\ \quad\quad\quad\quad \vdots \\ \text{SI } E_{N1} \text{ y } E_{N2} \text{ ENTONCES } H_N \end{array}$$

y así, la regla de la inferencia composicional máx-mín es:

$$\mu_H = \text{máx}[\text{mín}(\mu_{E_{11}}, \mu_{E_{12}}), \text{ mín}(\mu_{E_{21}}, \mu_{E_{22}}), \ldots \text{ mín}(\mu_{E_{N1}}, \mu_{E_{N2}})]$$

con extensiones similares para evidencia adicional E_{i3}, E_{i4}, etcétera.

En otro ejemplo de la regla composicional de inferencia, veamos cómo se usa con una relación. Definida la relación borrosa R(x,y) = APROXIMADAMENTE IGUAL sobre la relación binaria de los pesos de la gente en el rango de 60 a 80 kilos de la tabla 5.17.

			y		
x	**60**	**65**	**70**	**75**	**80**
60	1.0	0.7	0.4	0.2	0.0
65	0.7	1.0	0.6	0.5	0.2
70	0.4	0.6	1.0	0.8	0.5
75	0.2	0.5	0.8	1.0	0.8
80	0.0	0.2	0.5	0.8	1.0

Tabla 5.17 La relación APROXIMADAMENTE IGUAL definida en pesos

Al construir esta tabla, los grados de membresía se definieron para que la diferencia de la media de los dos valores fuera una disminución de 0.075 / por ciento. Por ejemplo, si los valores x y y son 75 y 65, el promedio es 70; la diferencia absoluta entre 75 y 65 y el promedio es 5, que, dividido entre el promedio, 70, arroja como resultado 7.1 por ciento. Este valor se multiplica por el factor constante de –.075/ por ciento para obtener –0.5, de allí que el grado de membresía final para 65 y 55 sea 1 – 0.5 = 0.5 y ésta es la entrada de la tabla. Una definición alterna y simple, desde el punto de vista computacional, sería definir cualquier cambio de 5 como una disminución fija en el grado de membresía, como .3. Sin embargo, esta definición alterna no produce resultados razonables para pesos pequeños como 5 y 10, que sería APROXIMADAMENTE IGUAL al grado 0.7.

Observe cómo la relación R(x,y) actúa como una **restricción borrosa** en cualquier par de valores x y y que tengan una membresía diferente a cero R(x,y). Una relación borrosa actúa como una **restricción elástica** al permitir un rango de grados de membresía más que exigir la restricción rígida de relaciones definidas. En realidad, el término APROXIMADAMENTE IGUAL ni siquiera se definiría en la lógica no borrosa. La restricción rígida de relaciones definidas exigiría que los valores sean exactamente iguales o no exactamente iguales, es decir, x es exactamente igual a y, o no lo es.

Como ejemplo de una restricción borrosa, consideremos la proposición, p, como sigue:

p = X es D

en la que D es algún conjunto borroso que actúa como una restricción borrosa en la variable lingüística llamada X. Los siguientes son algunos ejemplos de proposiciones borrosas de la forma anterior.

Juan es alto
Susana tiene más de 21 años
La mezcla de concreto es demasiado espesa
Herman es amigable
x es un número aproximadamente igual a 100
El pay es fruta

En el último ejemplo podría definirse un conjunto borroso como:

FRUTA = 1/manzanas + 1/naranjas

para indicar los tipos de elementos que se consideran frutas. Observe que en la teoría borrosa usted *puede* sumar manzanas y naranjas de una manera significativa.

Definamos ahora una restricción borrosa $R_1(x)$. Por ejemplo, el conjunto borroso PESADO podría definirse como:

$$R_1(x) = \text{PESADO} = .6/70 + .8/75 + 1/80$$

La regla composicional de la inferencia define la restricción borrosa en valores de y como:

$$R_3(y) = R_1(x) \text{ o } R_2(x,y)$$

donde el operador composicional, o, es la operación máx-mín:

$$\text{máx}_x \text{ mín}(\mu_1(x), \mu_2(x,y))$$

Otra manera de ver $R_3(y)$ es interpretándola como la solución de las ecuaciones relacionales:

$R_1(x)$
$R_2(x,y)$

para $R_3(y)$. Es decir, dadas las restricciones borrosas de x, y sobre x y y, puede deducirse una restricción borrosa en y. Deducciones como ésta incluyen el **cálculo de restricciones borrosas**, que es la base del razonamiento aproximado.

Utilizando estas definiciones, $R_3(y)$ puede calcularse como sigue:

$$R_3(y) = R_1(x) \circ R_2(x,y)$$

$$R_3(y) = [\ 0.0 \quad 0.0 \quad 0.6 \quad 0.8 \quad 1.0\ ] \circ \begin{bmatrix} 1.0 & 0.7 & 0.4 & 0.2 & 0.0 \\ 0.7 & 1.0 & 0.6 & 0.5 & 0.2 \\ 0.4 & 0.6 & 1.0 & 0.8 & 0.5 \\ 0.2 & 0.5 & 0.8 & 1.0 & 0.8 \\ 0.0 & 0.2 & 0.5 & 0.8 & 1.0 \end{bmatrix}$$

donde $R_1(x)$ se representa como un vector de fila. Los elementos diferentes a cero de $R_3(y)$ se calculan como sigue:

$$R_3(120 = \text{máx mín}[(.6,.4),(.8,.2)]$$
$$= \text{máx}(.4,.2) = .4$$
$$R_3(130 = \text{máx mín}[(.6,.6),(.8,.5),(1,.2)]$$
$$= \text{máx}(.6,.5,.2) = .6$$
$$R_3(140) = \text{máx mín}[(.6,1),(.8,.8),(1,.5)]$$
$$= \text{máx}(.6,.8,.5) = .8$$
$$R_3(150) = \text{máx mín}[(.6,.8),(.8,1),(1,.8)]$$
$$= \text{máx}(.6,.8,.8) = .8$$
$$R_3(160) = \text{máx mín}[(.6,.5),(.8,.8),(1,1)]$$
$$= \text{máx}(.5,.8,1) = 1$$

y de este modo, si $R_1(x)$ es PESADO, entonces:

$$R_3(y) = .4/60 + .6/65 + .8/70 + .8/75 + 1/80$$

que tiene la aproximación lingüística burda: $R_3(y)$ es MÁS O MENOS PESADO.

La relación $R_3(y)$ es una aproximación lingüística burda porque la operación DIL de $\mu^{0.5}$ que actúa sobre PESADO en realidad produce:

$$\text{DIL(PESADO)} = .8/70 + .9/75 + 1/80$$

y faltan los términos para 60 y 65. Sin embargo, los elementos con grados de membresía grandes de $\mu \geq .8$ se representan bien y esto justifica la afirmación de que MÁS O MENOS PESADO es una aproximación burda. De este modo, la inferencia composicional mediante máx-mín ha demostrado la relación lingüística borrosa:

MÁS O MENOS PESADO = PESADO o APROXIMADAMENTE IGUAL

Observe que estas relaciones dependen de las definiciones de conjunto borroso y los rótulos lingüísticos de los conjuntos. De este modo, la relación anterior no es verdadera en un sentido absoluto porque depende de las definiciones de los conjuntos borrosos, las relaciones y los rótulos. Sin embargo, una vez que se hacen estas definiciones básicas, la teoría borrosa proporciona un mecanismo para manipular estas expresiones de una manera formal y congruente. Esto es muy importante porque la manipulación lingüística y los significados de los términos son una sólida base teórica y no dependen de una comprensión *ad hoc* o intuitiva de una persona.

Métodos de máximo y de momentos

A la elección de un elemento con el máximo grado de membresía se le llama ***método de máximo*** para decidir la verdad de las consecuencias de la regla. Un método alternativo, ll*amado el* ***método de momentos***, asigna la verdad de las consecuencias de la regla de una manera análoga al cálculo del primer momento por la inercia de un objeto. La idea básica del método de momentos es considerar las consecuencias de todas las reglas en una decisión en lugar de considerar sólo el máximo. Como ya se mencionó, incluso el método de máximo puede llevar a la ambigüedad si todas las consecuencias de varias reglas tienen el mismo valor máximo.

Para ilustrar el método de momentos, consideremos el siguiente conjunto de reglas de producción borrosas para elaborar concreto:

```
R1:  SI LA MEZCLA ES MUY FLOJA
     ENTONCES AGREGAR ARENA Y GRAVA

R2:  SI LA MEZCLA ES ADECUADA
     ENTONCES DEJARLA COMO ESTÁ

R3:  SI LA MEZCLA ESTÁ MUY ESPESA
     ENTONCES DISMINUIR LA ARENA Y LA GRAVA
```

El concreto está formado por cemento, agua, arena y grava, combinados en las proporciones adecuadas, y de la **mezcla** se hace una prueba para determinar las proporciones óptimas para la aplicación deseada. Las directrices generales para las proporciones están disponibles dependiendo de la fuerza que se desea que tenga el concreto. Sin embargo, hay una variación debida a los materiales usados, el tamaño de la grava, las condiciones ambientales y otros factores, por lo que es una buena idea hacer una mezcla de prueba antes de empezar la construcción de un edificio de diez millones de dólares.

Un método común para determinar si la mezcla es correcta es la **prueba de caída**: se pone en un cono una cantidad determinada de concreto y luego se quita el cono, la distancia a la que cae la mezcla indica la condición del material. El concreto diseñado para losas y vigas debe tener caídas mínimas y máximas de 10 y 20 centímetros, respectivamente (Adams 79).

Los conjuntos borrosos de la figura 5.18 ilustran definiciones posibles para los antecedentes de las reglas borrosas en la elaboración de mezcla de concreto. Estos conjuntos borrosos también pueden definirse en una tabla o como funciones S y Π.

Los conjuntos borrosos utilizados para las consecuencias de la regla borrosa se definen en la figura 5.19. Al igual que con los antecedentes, existe una superposición entre estas acciones de consecuencia borrosas para proporcionar una mezcla con la que sea posible trabajar. Observe que los límites de –20 y 20 por ciento para el cambio de arena y grava se definen arbitrariamente. Como los antecedentes del conjunto borroso, las consecuencias también pueden definirse con las funciones S y Π.

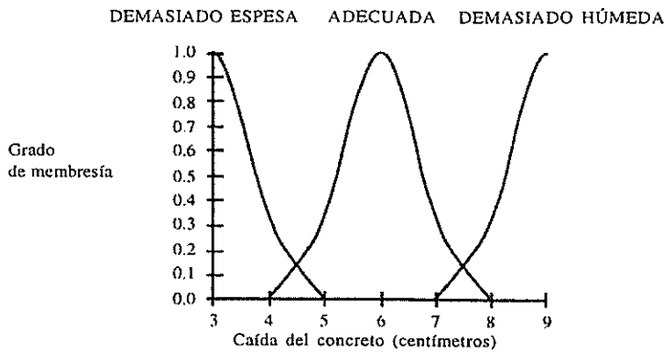


Figura 5.18 Antecedentes de la regla de producción borrosa para controlar la elaboración de la mezcla de concreto.

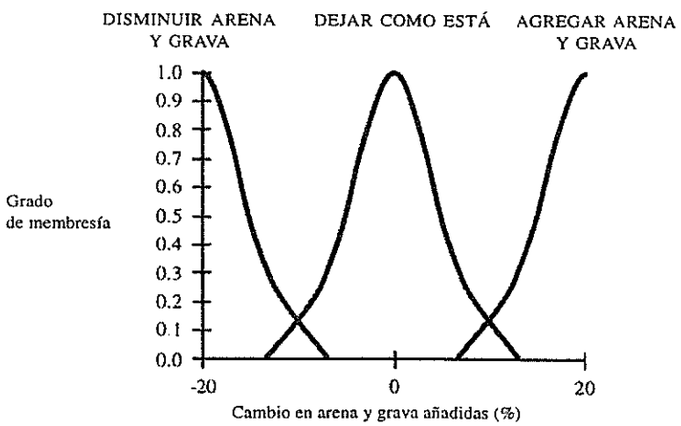


Figura 5.19 Consecuencias de la regla de producción borrosa para controlar la elaboración de la mezcla de concreto.

Para ejemplificar como funciona este conjunto de regla borrosa de producción, supongamos que la caída del concreto es de 15 centímetros. De la figura 5.18, el grado de membresía o valor de verdad del antecedente de cada regla es el siguiente:

$\mu_{\text{DEMASIADO ESPESA}}(15)=0$
$\mu_{\text{ADECUADA}}(15)=1$
$\mu_{\text{DEMASIADO BLANDA}}(15)=0$

de modo que R_2 es la única regla cuyo antecedente se satisface, por lo que se activa y se dispara, con la consecuencia borrosa resultante DEJAR COMO ESTÁ.

Con la aplicación de la regla composicional de la inferencia se tiene

$$\mu_{\text{DEJAR COMO ESTÁ}} = \text{máx}[\text{mín}(\mu_{\text{ADECUADA}}(15)] = \text{máx}[\text{mín}(1)] = 1$$

y de la figura 5.19, esto se traduce en un cambio de 0 por ciento en la cantidad de arena y grava.

Ahora, supongamos que la caída es de 12 centímetros. De la figura 5.18:

$\mu_{\text{DEMASIADO ESPESA}}(12)=0.5$
$\mu_{\text{ADECUADA}}(12)=.2$
$\mu_{\text{DEMASIADO BLANDA}}(12)=0$

Las reglas R_1 y R_2 tienen antecedentes que se satisfacen parcialmente, de manera análoga a la satisfacción de evidencia parcial de los antecedentes de la regla probabilística de PROSPECTOR; como hay cierta verdad, diferente a cero, de que la mezcla esté muy espesa o sea adecuada, entonces ambas reglas, R_1 y R_2, se activarán y dispararán. Observe que en un sistema de producción borrosa, a menos que haya un umbral en la verdad del antecedente, cada regla se disparará con una verdad de antecedente diferente a cero. Puede ser conveniente establecer un umbral para evitar ineficiencias debidas a la activación y el disparo de muchas reglas con valores de verdad bajos. Recuerde que en el sistema MYCIN el nivel de certidumbre mínimo para el disparo de una regla debe ser mayor a 0.2 para mejorar su eficiencia del sistema experto. Es posible definir un umbral similar en el valor de verdad del antecedente de conjuntos borrosos.

En el caso de una caída de 12 centímetros, se activan dos reglas. Con la aplicación de la regla de composición máx-mín se tiene:

$$\mu_{\text{DISMINUIR ARENA Y GRAVA}} = \text{máx}\ [\text{mín}(\mu_{\text{DEMASIADO ESPESA}}(12)]$$
$$= .05$$

$$\mu_{\text{DEJAR COMO ESTA}} = \text{máx}\ [\text{mín}(\mu_{\text{ADECUADA}}(12) = .2$$

Como se puede ver, a partir de un término con un sólo antecedente las funciones máx y mín son innecesarias.

Como hay dos reglas con consecuencias diferentes a cero, debe decidirse una acción de control. El método del máximo simplemente elegirá la regla con el grado de membresía más alto. En este caso, la acción DEJAR COMO ESTÁ se elige porque su grado es .2, contra el .05 de la otra regla.

El método de momentos calcula básicamente el **centro de gravedad** de las reglas de consecuencias borrosas. El término *centro de gravedad* proviene de la física, donde representa el punto en que, si todas las masas de un objeto estuvieran concentradas, la masa del punto reaccionaría igual ante la influencia de una fuerza externa. La definición del centro de gravedad, también llamado ***primer momento de inercia***, I, es:

$$I = \frac{\int m(x)x \; d(x)}{\int m(x) \; d(x)}$$

donde el signo de integral denota a la integral común.

En la figura 5.20 se muestran los conjuntos borrosos para las reglas R_2 y R_3; observe cómo se truncan en los valores de verdad de sus antecedentes, esto refleja la regla composicional para la inferencia. El truncamiento se hace porque, intuitivamente, tiene sentido que el valor de verdad de una consecuencia no exceda al de su antecedente.

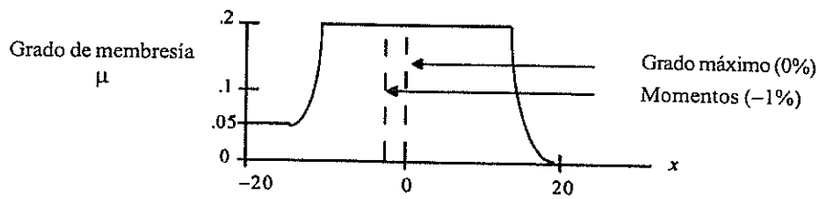


Figura 5.20 Métodos de máximo y momentos para las reglas borrosas de control para el proceso del concreto

El momento de las consecuencias se calcula como:

$$I = \frac{\int \mu(x) \; x \; d(x)}{\int \mu(x) \; d(x)} \qquad \text{para elementos continuos}$$

o

$$I = \frac{\sum_i \mu_i x}{\sum_i \mu_i} \qquad \text{para elementos discretos}$$

y que es del alrededor de −1 por ciento para la figura 5.20. Aunque esto es muy cercano a 0 por ciento obtenido con el método de máximo, la diferencia puede volverse importante para los conjuntos borrosos que se definen con una cantidad considerable de superposición. Ambos métodos se han usado en un controlador borroso para control de tráfico aéreo (Larkin 85). En este controlador, con el método de máximo se calcula la media aritmética de todos los máximos de las consecuencias borrosas. Por tanto, si hubiera varios elementos con el mismo valor máximo, aún se calcularía un valor de control definido.

Además de los de máximo y de momentos, se han usado otros métodos para resolver el **problema de especificación** de traducir un grado de membresía a un valor de control definido, o una aproximación lingüística que describa la variable de control. Sin embargo,

es difícil describir un conjunto de valores (el conjunto borroso) mediante un solo número o una frase lingüística.

Posibilidad y probabilidad

El término ***posibilidad*** tiene un significado específico en la teoría borrosa; en esencia, se refiere a valores permitidos. Por ejemplo, supongamos que una proposición se define en relación con el lanzamiento de dos dados en un universo U de su suma, como sigue:

```
p = X es un entero en U

U = {2,3,4,5,6,7,8,9,10,11,12}
```

En terminología borrosa, para cualquier entero *i*:

```
Pos (X = i) = 1  para 2 ≤ i ≤ 12
Pos (X = i) = 0  de otra manera
```

donde **Pos** (X = i) es una abreviatura de "La posibilidad de que X pueda tomar el valor i" (Zadeh 81). La posibilidad de que los dados muestren un valor entre 2 y 12 es muy diferente de la probabilidad de un valor i. Es decir, la **distribución de posibilidad** no es la misma que la **distribución de probabilidad**. La distribución de probabilidad de un dado es la frecuencia de ocurrencia esperada de la **variable aleatoria** X, la suma. Por ejemplo, puede obtenerse debido a 1 + 6, 2 + 5 y 3 + 4. De este modo, la probabilidad de que salga 7 es:

$$\frac{2 \cdot 3}{36} = \frac{1}{6}$$

mientras que la probabilidad de tirar 2 es:

$$\frac{1}{36}$$

Por el contrario la distribución de la posibilidad es un valor constante de 1 para todos los enteros, desde 2 hasta 12. Se dice que una proposición *p induce una distribución de posibilidades*, P_x; es decir, dada una proposición borrosa, *p* basada en un conjunto borroso D y una variable lingüística X:

```
p = X es D
```

Se dice que la proposición se encuentra en **forma canónica** cuando se expresa de esta manera, en la que el término *canónico* significa de manera normal. El conjunto difuso, D, es un **predicado borroso**, en contraste con los predicados de la lógica común. D también puede ser una relación borrosa. La distribución de posibilidad inducida por p es igual a D y se define con la siguiente **ecuación de asignación de posibilidad**:

$$\Pi_x = D$$

lo que significa que para todos los valores x en el universo, U:

$$\text{Pos } \{X = x\} = \mu_D(x) \quad x \in U$$

Por ejemplo, dada la proposición

```
p = Juan es alto
```

es posible definir una variable lingüística, Altura, con un valor Juan. La forma canónica

```
X es D
```

se representa en términos de la variable Altura con

```
Altura(Juan) es ALTO
```

y así:

$$\text{Pos } \{\text{Altura(Juan)} = x\} = \mu_{ALTO}(x)$$

La proposición, p, puede escribirse como una distribución de posibilidad como sigue:

$$\text{Juan es alto} \rightarrow \Pi_{\text{Altura(Juan)}} = \text{ALTO}$$

Donde la flecha simboliza "se traduce a". Altura es una variable lingüística y ALTO es un conjunto borroso, pero Juan no es una variable lingüística.

Aunque un conjunto borroso puede asignarse a una distribución de posibilidad, como en $\Pi_X = D$, en realidad no son lo mismo. Para explicar la diferencia, consideremos el conjunto borroso definido por un tiro de dados, como sigue:

$$\text{TIRO}(1) = 1/3 + 1/4$$

Este conjunto está definido para significar que un tiro específico de dados, Tiro 1, arroja un 3 en un dado y un 4 en el otro. Por su parte la distribución de posibilidad:

$$\Pi_{\text{TIRO}(1)} = 1/3 + 1/4$$

significa que el tiro arrojó 3 *o* 4, donde "o" es el o excluyente que representa la incertidumbre en nuestro conocimiento del tiro, existe una posibilidad igual a la unidad de que sea un 3 y otra de que sea un 4. En el conjunto borroso, es cierto que los valores de los dados fueron 3 *y* 4. La distribución de posibilidad significa que el dado está limpio en relación con el hecho de que pueda ocurrir un 3 o un 4. El conjunto borroso indica cuáles valores se mostraron después del tiro.

Veamos otro ejemplo, considerando la proposición de que "Javier odia X huevos para desayunar", donde X es cualquier valor del universo, X = {1, 2, ... 8} (Zadeh 78).

X	1	2	3	4	5	6	7	8
$\Pi_{\text{ODIO(Javier)}}(X)$	1.0	1.0	1.0	1.0	0.8	0.6	0.4	0.2
P(X)	0.1	0.8	0.1	0.0	0.0	0.0	0.0	0.0

La distribución de posibilidad $\Pi_{\text{ODIO(Javier)}}(X)$ se interpreta como la facilidad con que Javier come X huevos. La distribución de probabilidad, P(X), se determina empíricamente al preguntar a Javier si se le puede vigilar durante el desayuno durante un año para conducir un estudio científico.

Un tema importante del que hay que estar consciente es que la posibilidad es *no estadística* mientras que la probabilidad es *estadística*. Por ejemplo, aunque Javier pueda comer ocho huevos, el estudio empírico muestra que durante el periodo de observación nunca había comido más de tres (y casi siempre comía menos). En este sentido, la posibilidad es la capacidad. Un alto grado de posibilidad no necesariamente significa un alto grado de probabilidad, es decir, tal vez no haya correlación entre la posibilidad y la probabilidad.

La extensión de lo borroso a la probabilidad común es la **probabilidad borrosa**, que describe las probabilidades que sólo se conocen de manera imprecisa (Zadeh 84b). Algunos ejemplos de probabilidades borrosas son los **calificadores borrosos** *Muy probablemente*, *Improbablemente*, *No muy probablemente*, etcétera. Un ejemplo de proposición borrosa con una probabilidad borrosa, es el siguiente:

```
la batería está MAL es Muy probablemente
```

Reglas de traducción

La probabilidad borrosa se incorpora en la lógica borrosa, llamada **LD,** basada en la lógica L_1 de Lukasiewicz (Zadeh). Un componente principal de la LD es un grupo de **Reglas de traducción** que especifican la manera en que las proposiciones modificadas o compuestas se generan a partir de sus proposiciones elementales.

Las reglas de traducción se dividen en cuatro categorías:

- *Tipo I*: **reglas de modificación,** como

```
X es muy grande

Juan es más alto que Miguel
```

- *Tipo II*: **reglas de composición**, como:

composición condicional

```
Si X es ALTO entonces Y es BAJO
```

composición conjuntiva

```
X es ALTO y Y es BAJO
```

composición disyuntiva

```
X es ALTO o Y es BAJO
```

composición condicional y conjuntiva

```
Si X es ALTO entonces Y es BAJO
   Por tanto Y Más bien es BAJO
```

- *Tipo III*: **reglas de cuantificación**, como:

  Casi todos los postres son MARAVILLOSOS

  Demasiada comida nutritiva causa GORDURA

- *Tipo IV*: **reglas de calificación**, como:

  calificación verdadera

  el pay de chocolate es DELICIOSO es Muy cierto

  calificación de probabilidad

  el pay de chocolate se sirve PRONTO es Muy posiblemente

  calificación de posibilidad

  el pay de chocolate es MALO para usted es Imposible

Las reglas de calificación son aquellas que pertenecen a las probabilidades borrosas. Las reglas de cuantificación tratan con cuantificadores borrosos como *la mayoría*, que no pueden definirse con los cuantificadores universales y existenciales clásicos.

La traducción de una regla de tipo I se representa con:

$$X \text{ es } D \rightarrow \Pi_x = D$$

con la proposición traducida:

$$X \text{ es } mD \rightarrow \Pi_x = D^*$$

donde *m* es un modificador como *No*, *Muy*, *Más o menos*, etc. D^* representa la modificación de D por *m*. En la tabla 5.18 se muestra algunas definiciones predeterminadas para *m* y D^*. Éstas se definieron de la misma manera que los modificadores lingüísticos antes analizados. También es posible utilizar otras definiciones para *m* y D^*.

m	D^*
No	$D' = \int [1-\mu_D(x)]/x$
Muy	$D^2 = \int \mu_D^2(x)/x$
Más o menos	$\sqrt{D} = \int \sqrt{\mu_D(x)/x}$

Tabla 5.18
Valores de los parámetros de traducción para algunas reglas de tipo 1 (observe que todas las integrales están dentro del universo)

Por ejemplo, definiendo ALTO

$$\text{ALTO} = .2/5 + .6/6 + 1/7$$

entonces las traducciones son como sigue:

$$\text{Juan no es alto} \rightarrow .8/1.50 + .4/1.80 + 0/2.10$$
$$\text{Juan es muy alto} \rightarrow .04/1.50 + .36/1.80 + 1/2.10$$
$$\text{Juan es más o menos alto} \rightarrow .45/1.50 + .77/1.80 + 1/2.10$$

La variable X no tiene que ser unitaria, en cambio, puede ser una relación binaria o N-aria en general. Por ejemplo, una proposición como "Y y Z son D" se considera como parte de "X es D". Definiendo CERCA como una relación binaria borrosa en U × U:

$$\text{CERCA} = 1/(1,1) + .5/1,2) + .5/(2,1)$$

Entonces

$$X \text{ y } Y \text{ están cerca} \rightarrow \Pi_{(X,Y)} = \text{CERCANO}$$

$$X \text{ y } Y \text{ están muy cerca} \rightarrow \Pi_{(X,Y)} = \text{CERCANO}^2 = 1/(1,1) + .25/(1,2) + .25/(2,1)$$

Un ejemplo de una regla de tipo II es:

$$\text{SI } X \text{ es } D \text{ entonces } Y \text{ es } E \rightarrow \Pi_{(X,Y)} = \bar{D}' \oplus \bar{E}$$

donde $\bar{D}$ y $\bar{E}$ son las extensiones cilíndricas de D y E en sus universos:

$$\bar{D} = D \times V \qquad \bar{E} = U \times E$$

y ⊕ es la suma delimitada. Para esta regla

$$\mu_{\bar{D}', \oplus \bar{G}}(x,y) = 1 \wedge [1 - \mu_D(x) + \mu_E(y)]$$

y la función mín es ∧. Esta definición compagina con la implicación en la lógica L_1, mientras que otras definiciones tal vez no (Zadeh 79). Un ejemplo de una regla de tipo II, definidas:

$$U = V = \{4, 5, 6, 7\}$$

$$D = \text{ALTO} = .2/1.50 + .6/1.80 + 1/2.10$$

$$E = \text{BAJO} = 1/1.20 + .2/1.50$$

$$\text{SI } X \text{ es alto entonces } Y \text{ es bajo} \rightarrow \Pi_{(X,Y)}$$

$$= 1/(5,4) + 1/(5,5) + 1/(6,4) + .6/(6,5) + 1/(7,4) + .2/(7,5)$$

donde el grado de membresía de un elemento como (5,4) se calcula como sigue:

$$\mu_{\bar{D}\,,\,\oplus\bar{G}}(5,4) = 1 \wedge [1 - .2 + 1] = 1 \wedge [1.8] = 1$$

Incertidumbre en los sistemas expertos borrosos

Cuando las probabilidades borrosas se usan en los sistemas expertos, hay una diferencia en comparación con la inferencia probabilística ordinaria (Zadeh 83). Considerando la regla borrosa canónica:

```
Si X es D entonces Y es E (con una probabilidad ß)
```

Esta regla puede escribirse como una probabilidad condicional:

```
P(Y es E | X es D) = ß
```

En un sistema experto convencional que utiliza la teoría de la probabilidad clásica se supondría entonces

```
P(Y no es E | X es D) = 1 - ß
```

Sin embargo, esto no es correcto en la teoría de lo borrosa, si D es un conjunto borroso. El resultado borroso válido es más débil:

```
P(Y no es E | X es D) + P(Y es E | X es D) ≥ 1
```

porque sólo establece un límite inferior en las probabilidades, que pueden ser números borrosos. En general, para sistemas borrosos

```
P(H|E) no es necesariamente igual a 1 - P(H'|E)
```

En los sistemas expertos borrosos puede haber difusión en tres áreas:
(1) antecedentes y/o consecuencias de reglas, como:

```
Si X es D entonces Y es E
Si X es D entonces Y es E con FC = α
```

donde FC es el factor de certidumbre y α es un valor numérico como 0.5.

(2) coincidencia parcial entre el antecedente y los hechos que coinciden con los patrones del antecedente. En los sistemas expertos no borrosos, una regla no se activa a menos que los patrones coincidan exactamente con los hechos, mientras que en los borrosos todo es cosa del grado de coincidencia y las reglas pueden activarse en alguna medida, a menos que se establezca un umbral.

(3) los cuantificadores borrosos, como *Casi todos* y los calificadores borrosos, como *Muy probablemente*, *Muy cierto*, *Definitivamente posible*, etcétera.

Las proposiciones a menudo contienen cuantificadores borrosos implícitos y/o explícitos. Por ejemplo, considerando la **disposición:**

```
m = los postres son MARAVILLOSOS
```


El término *disposición* significa una proposición que suele ser verdad, con la forma canónica:

```
Normalmente(X es R)
```

donde *Normalmente* es un cuantificador borroso implícito y R es una **relación de restricción** que actúa sobre la variable X limitando los valores que podría tomar. Muchas reglas heurísticas que la gente conoce son disposiciones. En realidad, el **conocimiento de sentido empírico común** es básicamente una colección de disposiciones sobre el mundo real.

Esta disposición puede traducirse a las formas propositivas explícitas:

```
p = Normalmente los postres son maravillosos
p = La mayoría de los postres son maravillosos
```

que pueden expresarse como una regla heurística:

```
r = Si x es un postre
    entonces es probable que x sea maravilloso
```

A continuación se enumeran algunas reglas de inferencia para sistemas borrosos:

- **principio de vinculación:**

```
X es D
D ⊂ E
------
X es E
```

- **vinculación disposicional:** limitación de casos donde *Normalmente* se vuelve *Siempre*

```
Normalmente (X es D)
D ⊂ E
------
Normalmente (X es E)
```

- **regla composicional**

```
X es D
(X,Y) es R
------
Y es D o R
```

donde R es una relación binaria sobre la variable binaria (X,Y) y

$$\mu_{D \circ R}(y) = \sup_{x}[\mu_D(x) \wedge \mu_R(x,y)]$$

y el **supremum**, simbolizado por **sup**, es el **límite superior**. Por lo general, el supremum es lo mismo que la función máx. La diferencia aparece cuando no hay un valor máximo, como en el caso del intervalo de número real de los números menores a 0. Como no hay un número máximo real menor a 0, el supremum se utiliza para tomar 0 como el límite superior.

- ***modus ponens* generalizado**

```
X es D
Y es E si X es F
-----------------
Y es D o (H' ⊕ E)
```

donde H´ es la negación difusa de H y la suma delimitada se define

$$\mu_{H' \oplus G}(x, y) = 1 \wedge [1 - \mu_H(x) + \mu_E(y)]$$

El *modus ponens* generalizado no requiere que el antecedente "X es H" sea idéntico a la premisa "X es D". Observe que es muy diferente de la lógica clásica, que requiere que coincidan con exactitud. El *modus ponens* generalizado es en realidad un caso especial de la regla composicional de inferencia. A diferencia de los sistemas expertos convencionales, en los que el *modus ponens* es la regla básica de inferencia, la regla composicional de inferencia es la regla básica de los sistemas expertos borrosos.

Los sistemas expertos que usan razonamiento aproximado eligen uno de dos métodos diferentes, uno es la restricción del valor de verdad y otro es la inferencia composicional (Whalen 85). En la encuesta de 11 sistemas expertos borrosos, casi todos usaban inferencia composicional.

5.6 EL ESTADO DE INCERTIDUMBRE

No hay un consenso claro sobre el mejor método para tratar con la incertidumbre, aunque se han estudiado varias técnicas como la bayesiana clásica, el **conjunto bayesiano convexo**, la de Dempster y Shaffer, de **Kyburg** y la teoría de la posibilidad de la lógica borrosa (Thompson 85). En el método de conjunto convexo de la teoría de Bayes, el estado de creencia no es una sola función del método clásico de Bayes, sino que está caracterizado por un conjunto de **funciones convexas**. Esto significa que cualquier función puede representarse como una combinación lineal de otras dos funciones.

Dado que la teoría de Dempster y Shafer se presentó primero como una generalización de la probabilidad clásica, se han dado impugnaciones. Por ejemplo, en un artículo Kyburg asegura que la teoría de Dempster y Shafer no es una generalización de la probabilidad clásica, sino todo lo contrario. Kyburg cree que esta teoría está incluida en la probabilidad clásica y que los intervalos de Dempster y Shafer están incluidos en el método del conjunto convexo de Bayes (Kyburg 87). La teoría de Dempster y Shafer también parece tener dificultades al tratar con creencias que están cercanas a cero, pues se obtienen resultados muy diferentes cuando las creencias son cero en comparación con los resultados cuando las creencias son muy pequeñas (Dubois 85). Otro problema con ella es la explosión exponencial en el número de cálculos, a medida que aumenta el número de respuestas posibles a un problema de diagnóstico.

Aunque la aproximación de Gordon y Shortliffe evita la explosión exponencial, puede producir resultados deficientes cuando la evidencia es muy conflictiva. Se ha dado un método alternativo que no es una aproximación y produce buenos resultados para la evidencia jerárquica sin la explosión combinatoria (Shafer 87). En varios otros artículos se ha tratado de sobreponerse al problema de la explosión exponencial mediante generalizaciones diferentes a la teoría de Dempster y Shafer (Liu 86; Yen 86; Lee 87).

El principal beneficio de todo este trabajo ha sido una reexaminación de los fundamentos de la teoría de la probabilidad y la ampliación del interés en los métodos para tratar con la incertidumbre.

5.7 RESUMEN

En este capítulo, se analizaron las teorías no probabilísticas de la incertidumbre. La teoría de Dempster y Shafer y la teoría de lo borroso son maneras de tratar con la incertidumbre en los sistemas expertos. Los factores de certidumbre son sencillos de implantar y se han usado con éxito en sistemas expertos como MYCIN, en el que las cadenas de inferencia son cortas. Sin embargo, la teoría de los factores de certidumbre es una teoría *ad hoc* que no parece tener validez general para las cadenas de inferencia.

La teoría de Dempster y Shafer tiene una base rigurosa y contiene premisas para sistemas expertos. Sin embargo, en la actualidad no parece haber ningún consenso claro sobre la manera de aplicarla para uso general en sistemas expertos.

La teoría de lo borroso es la teoría más general de la incertidumbre que se ha formulado. Tiene amplia aplicabilidad debido al principio de extensión, y desde el primer artículo clásico de Zadeh en 1965, se ha aplicado a muchos campos.

PROBLEMAS

5.1 Dado que la evidencia E_2 se suma a la evidencia original E_1, demuestre que

$$P(D_i \mid E_1 \cap E_2) = \frac{P(E_2 \mid D_i \cap E_1)P(D_i \mid E_1)}{\sum_j P(E_2 \mid D_j \cap E_1)P(D_j \mid E_1)}$$

(Consejo: utilice los resultados del problema 4.8 (b)).

5.2 Demuestre que

$FC(H,E) + FC(H',E) = 0$

5.3 Dadas las reglas:

SI E_1 Y E_2 Y E_3
ENTONCES H (FC_1)

SI E_4 O E_5
ENTONCES H (FC_2)

donde $FC_1(E_1$ e) = 1 $FC_1(E_2,e$ = 0.5 $FC_1(E_3,e$
= 0.3 $FC_2(E_4,e$ 0.7 $FC_2(E_5,e$
= 0.2 $FC_1(H,E$ 0.5 $FC_2(H,E$

=0 9 (a) Dibuje un árbol que ilustre la manera en que estas reglas sustent
n a H.(b) Calcule los factores de certidumbre $FC_1(H,e)$ y FC
(H,e).
(c) Calcule $FC_{COMBINADO}$ $[FC_1(H,e),FC_2(H,e)]$

5.4 (a) En la figura 5.7 (c) suponga:

m(X) = 0.2
m(Y) = 0.3
m(Z) = 0.5

encuentre los intervalos evidenciales de X,Y y Z utilizando la teoría de Dempster y Shafer.

(b) En la figura 5.7 (d), suponiendo que:

$m(X) = 0.4$
$m(Y) = 0.6$

encuentre los intervalos evidenciales para:

X
$X \cap Y$
Y
$X \cap Y'$
$X' \cap Y$

5.5 Dadas las reglas:

Regla 1: SI E ENTONCES H
Regla 2: SI E ENTONCES H'

y suponiendo que:

$\Theta = \{H, H'\}$
$m_1(H) = 0.5$ $m_1(\Theta) = 0.5$ para Regla 1
$m_2(H') = 0.3$ $m_2(\Theta) = 0.7$ para Regla 2

(a) Elabore la tabla de Dempster y Shafer que muestra la combinación de evidencia y calcula las funciones de creencia combinadas.
(b) Calcule las plausibilidades.
(c) Calcule los intervalos evidenciales.
(d) Calcule las dudas.
(e) Calcule las ignorancias.

5.6 Con base en informes de diferentes tipos de sensores, en la siguiente tabla se proporcionan los grados de creencia del ambiente de avión comercial (C), bombardero (B) y caza (Z).

Elementos focales	Sensor 1 (m1)	Sensor 2 (m2)
Θ		0.15
A,B	0.3	0.1
A,Z	0.1	0.05
B,Z	0.1	0.1
A 0.05	0.3	
B	0.2	0.05
Z	0.1	0.2

(a) Calcule las funciones de creencia, plausibilidad, intervalos evidenciales, dudas e ignorancias iniciales.
(b) Calcule estos mismos parámetros después de combinar la evidencia.

5.7 Un policía detiene a un automovilista por exceso de velocidad. Debido a las fallas en el radar del policía y en el velocímetro del automovilista, las funciones de creencia son como sigue:

Policía	Automovilista
m1(57) = 0.3	m2(56) = 0.2
m1(56) = 0.5	m2(55) = 0.6
m1(55) = 0.2	m2(54) = 0.2

(a) Calcule las funciones de creencia, plausibilidad, intervalos evidenciales, dudas e ignorancia de cada persona.
(b) Calcule estos parámetros después de combinar las evidencias.
(c) Con base en estos parámetros explique por qué cree que el automovilista iba o no a alta velocidad.

5.8 Dados los conjuntos difusos,

$A = .1/1 + .2/2 + .3/3$ $\quad B = .2/1 + .3/2 + .4/3$

calcule y explique lo siguiente:

(a) ¿Son iguales los conjuntos? Explique.
(b) Complemento de conjuntos.
(c) Unión de conjuntos.
(d) Intersección de conjuntos.
(e) ¿Permanece la ley del medio excluido para cada conjunto? Explique.
(f) Producto de conjuntos.
(g) Segunda potencia de cada conjunto.
(h) Suma probabilística.
(i) Suma delimitada.
(j) Producto delimitado.
(k) Diferencia delimitada.
(l) Concentración.
(m) Dilatación.
(n) Intensificación.
(o) Normalización.

5.9 Dados los conjuntos borrosos

$$Q = \begin{bmatrix} 0.2 & 0.3 \\ 0.4 & 1.0 \end{bmatrix} \quad \text{definido en } U_1 \times U_2$$

$$P = \begin{bmatrix} 0.1 & 0.5 & 0.3 \\ 0.2 & 0.0 & 0.4 \end{bmatrix} \quad \text{definido en } U_2 \times U_3$$

(a) Calcule la primera y la segunda proyecciones, además de las proyecciones totales para cada conjunto.
(b) Calcule las extensiones cilíndricas de cada conjunto.
(c) Demuestre que

$$Q \circ P = \text{proj}\,(\overline{Q} \cap \overline{P}\,;\, U_1 \times U_3)$$

5.10 Considere la variable lógica Persona como un conjunto borroso de tercer orden.

(a) Defina Persona como tres conjuntos borrosos de orden 2.
(b) Defina cada conjunto de orden 2 en términos de tres conjuntos borrosos de 1er orden.
(c) Defina tres de los conjuntos borrosos de orden 1 en términos de funciones S y/o Π.

5.11 (a) Defina cinco valores lingüísticos para la variable lingüística Incertidumbre.
(b) Trace las funciones apropiadas para estos valores y explique sus opciones.
(c) Dibuje los conjuntos difusos para

No VERDADERO
Más o menos VERDADERO
VERDADERO de alguna manera
Muy VERDADERO
Bastante VERDADERO
VERDADERO

suponiendo que VERDADERO es una función S, ¿cuáles son los límites de VERDADERO? Explique.

5.12 (a) Defina por lo menos seis valores para la variable lingüística Temperatura del agua.
(b) Trace las funciones apropiadas para los valores del conjunto difuso en una gráfica.
(c) Proporcione tres funciones de conjuntos borrosos con modificador basadas en CONGELACIÓN.

5.13 (a) Demuestre con la tabla de verdad para N = 2 y N = 3 los valores primitivos de L = operadores lógicos de la tabla 5.14.
(b) Derive x ↔ y en términos de los valores absolutos de x y y.

5.14 Dados los valores de verdad numéricos:

x(A) = .2/.1 + .6/.5 +1/.9
x(B) = .1/.1 + .3/.5 + 1/.9

calcule la verdad de la lógica de lo difuso de lo siguiente:

(a) NO A
(b) A Y B
(c) A O B
(d) A → B
(e) B → A

5.15 Defina una gramática borrosa utilizando frases primarias y de rango, con modificador, para generar producciones como:

SI LA PRESIÓN ES ALTA
ENTONCES GIRE LA VÁLVULA MÁS ABAJO

SI LA PRESIÓN ES MUY ALTA
ENTONCES GIRE LA VÁLVULA MUCHO MÁS ABAJO

SI LA PRESIÓN ES MUY MUY ALTA
ENTONCES GIRE LA VÁLVULA MUCHO MUCHO MÁS ABAJO
SI LA PRESIÓN ES DE BAJA A MEDIA .
ENTONCES GIRE LA VÁLVULA MÁS ARRIBA

Suponiendo que (1) las primarias para PRESIÓN son sólo BAJO y ALTO, (2) las frases del rango que incluyen A sólo ocurren en el antecedente, y (3) las primarias para VÁLVULA en la conclusión sólo son MÁS ABAJO y MÁS ARRIBA.

BIBLIOGRAFÍA

(Adams 79). J. T. Adams, *The Complete Concrete, Masonry and Brick Handbook Arco*, Publishing, p. 1059, 1979.

(Adams 85). J. Barclay Adams, "Probabilistic Reasoning and Certainty Factors", en *Rule-based Expert Systems*, Addison Wesley Pub., pp. 263-271, 1985.

(Bellman 73). R. E. Bellman y M. Giertz, "On the Analytic Formalism of the Theory of Fuzzy Sets", en *Information Science*, **5**, pp. 149-156, 1973.

(Bogler 87). Philip L. Bogler, Shafer-Dempster Reasoning with Applications to Multisensor Target Identification Systems", en *IEEE Tran on. System, Man, and Cybernetics*, **SMC-17**, (6), pp. 968-977. Nov/Dic. de 1987.

(Carnap 50). R. Carnap, "The Two Concepts of Probability", en *Logical Foundations of Probability*, University of Chicago Press, pp. 19-5I, 1950.

(Dempster 67). A. P. Dempster, "Upper and Lower Probabilities Induced by Multivalued Mappings", *Annals* of *Math Stat.*, 38, pp. 325-329, 1967.

(Dubois 85). Didier Dubois y Henri Prade, "Combination and Propagation of Uncertainty with Belief Functions", *IJCAI '85*, pp. 111-113, 1985.

(Garvey 81). T. D. Garvey, J. D. Lowrance, y M. A. Fischler, "An Inference Technique for Integrating Knowledge from Disparate Sources", en *Proc. 7th IJCAI*, pp. 319-325, 1981.

(Giarratano 91c). Joseph Giarratano, *et al.*, "Fuzzy Logic Control for Camera Tracking System", en Fith Annual Workshop on Space Operations Applications and Research (SOAR '91), pp. 94-99, 1991.

(Gordon 85). Jean Gordon y Edward H. Shortliffe. "The Dempster-Shafer Theory of Evidence", en *Rule-based Expert Systems*, Bruce Buchanan and Edward Shortliffe, eds., pp. 272-292, 1985.

(Kandel 82). Abraham Kandel, *Fuzzy Techniques in Pattern Recognition*, John Wiley and Sons, Inc., 1982.

(Klir 88). George J. Klir y Tina A. Folger, *Fuzzy Sets, Uncertainty, and Information*, Prentice-Hall, pp. 75-77, 1988.

(Kyburg 87). Henry E. Kyburg, Jr., "Bayesian and Non-Bayesian Evidential Updating", en *Artificial Intelligence*, 31, (3), pp. 271-293, 1987.

(Larkin 85). L. I. Larkin, "A Fuzzy Logic Controller for Aircraft Flight Control", en *Industrial Applications of Fuzzy Control*, M. Sugeno, ed., North-Holland, pp. 87-103, 1985.

(Larsen 81). P. Martin Larsen, "Industrial Applications of Fuzzy Logic Control", en *Fuzzy Reasoning and Its Applications*, E. H. Mamdani y B. R. Gaines, eds., Academic Press, pp. 335-342, 1981.

(Lee 87). Sunggu Lee y Kang G. Shin, "Uncertain Iference Using Belief Functions", en *Proceedings of the Third Conference on Artificial Intelligence Applications*, pp. 238-243, 1987.

(Liu 86). Gearald Shao-Hung Liu, "Causal and Plausible Reasoning in Expert Systems", en *AAAI-86*, pp. 220-225, 1986.

(Lowrance 82). J. D. Lowrance y T. D. Garvey, "Evidential Reasoning: A Developing Concept", en *Proc. Int. Conf. on Cybernetics and Society*, pp. 6-9, 1982.

(Maiers 85). J. Maiers y Y. S., Sherif, "Applications of Fuzzy Set Theory", en *IEEE Transactions on Systems, Man, and Cybernetics*. **SMC-15**, (1), pp. 175-189, enero/febrero de 1985.

(Shafer 76). Gleen Shafer, *A Mathematical Theory of Evidence*, Princeton University Press, 1976.

(Shafer 87). Gleen Shafer y Roger Logan, "Implementing Dempster's Rule for Hierarchical Evidence", *Artificial Intelligence*, **33**. (3), pp. 217-298, Nov. 1987.

(Shortliffe 85). Edward H. Shortliffe y Bruce G. Buchanan, "A Model of Inexact Reasoning in Medicine", en *Rule-based Expert Systems*, Addison-Wesley Pub., pp. 233-262, 1985.

(Strat 84). Thomas M. Strat, "Continuos Belief Functions for Evidential Reasoning", en *Procedings of the National Conference on AI*, pp. 308-313, 1984.

(Wesley 86). Leonard P. Wesley, "Evidential Knowledge-based Computer Vision", en *Optical Eng.*, **25**, (3) pp. 363-379, 1986.

(Whalen 85). Thomas Whalen y Brian Schott, "Alternative Logics for Approximate Reasoning in Expert Systems: A Comparative Study", en *Int. J. of Man-Machine Studies*, **22**, pp. 327-346, 1985.

(Yen 86). John Yen, "A Reasoning Model Based on an Extended Dempster-Shafer Theory", en *AAAI-86*, pp. 125-131, 1986.

(Zadeh 65). Lotfi A. Zadeh, "Fuzzy Sets", en *Information and Control*, pp. 338-353, 1965.

(Zadeh 73). Lotfi A. Zadeh, "The Concept of a Linguistic Variable and Its Application to Approximate Reasoning", en *Memorandum ERL-M 411*, University of California at Berkeley p. 83, octubre de 1973.

(Zadeh 75). Lofti A. Zadeh, "Calculus of Fuzzy Restrictions", en *Fuzzy Sets and Their Applications to Cognitive and Decision Processes*, Academic Press, p. 36, 1975.

(Zadeh 78). Lotfi A. Zadeh, "Fuzzy Sets as a Basis for a Theory of Possibility", en *Fuzzy Sets and Systems*, 1, pp. 3-28, 1978.

(Zadeh 79). Lotfi A. Zadeh, "A Theory of Approximate Reasoning", en *Machine Intelligence*, 9, J. E. Hayes, Donald Michie, y L. I. Mikulich, eds. pp. 149-194, 1979.

(Zadeh 81). Lotfi A. Zadeh, "PRUF –A Meaning Representation Language for Natural Languages", en *Fuzzy Reasoning and Its Applications*, E. H. Mamdani y B. R. Gaines, eds., Academic Press, 1-66, 1981.

(Zadeh 83). Lotfi A. Zadeh, "The Role of Fuzzy Logic in the Management of Uncertainty in Expert Systems", en *Fuzzy Sets and Systems*, **11**, pp. 199-227, 1983.

(Zadeh 84a). Lotfi A. Zadeh, "Review of Books: A Mathematical Theory of Evidence", en *The AI Magazine*, pp. 81-83, otoño de 1984.

(Zadeh 84b). Lotfi A. Zadeh, "Fuzzy Probabilities", en *Information Processing & Management*, **20**, (3), pp. 363-372, 1984.

(Zadeh 88). Lofti A. Zadeh, "Fuzzy Logic", en *IEEE Computer*, pp. 83-93, abril de 1988.

(Zimmerman 85). H. J. Zimmerman, *Fuzzy Set Theory and Its Applications*, Kluwer-Nijhoff Publishing, pp. 62-63, 1985.

CAPÍTULO 6

Diseño de sistemas expertos

6.1 INTRODUCCIÓN

En los capítulos anteriores, se analizaron los conceptos generales y la teoría de los sistemas expertos. En este capítulo se presentan las directrices generales para construir sistemas expertos prácticos diseñados para aplicaciones reales, no como prototipos de investigación, y se describe una **metodología para la ingeniería de software**, de modo que el sistema experto construido pueda ser un producto de calidad desarrollado de manera oportuna, rentable y efectiva.

6.2 SELECCIÓN DEL PROBLEMA APROPIADO

Como se explica en la sección 1.6, antes de construir un sistema experto debe seleccionarse un problema apropiado. Al igual que con cualquier proyecto de software, pueden hacerse varias consideraciones generales antes que se comprometa una gran cantidad de gente, recursos y tiempo en el sistema experto propuesto. Estas reflexiones son típicas en la administración del proyecto para los programas convencionales, pero debe hacerse a la medida para cubrir las necesidades especiales de los sistemas expertos. En la figura 6.1 se muestra una concepción de la administración al más alto nivel del desarrollo de estos sistemas, incluyendo las tres etapas generales que contienen más consideraciones específicas; mismas que se explicarán a manera de preguntas y respuestas, para que sirvan como directrices en la planeación de los sistemas expertos.

Selección del paradigma apropiado

¿Por qué se está construyendo un sistema experto?

Tal vez ésta sea la pregunta más importante que se hará sobre *cualquier* proyecto. Aunque en la sección 2 del capítulo 1 se explican las ventajas generales de los sistemas expertos, sólo la administración autoriza el sistema y personal técnico que se necesita para implantar lo necesario. En particular, la respuesta a esta pregunta debe darse eventualmente a los propietarios o accionistas que aportan los fondos para el desarrollo. Antes de iniciar, debe haber una clara identificación del problema, el especialista y los usuarios.

Ganancia

¿Cuál es la ganancia?

Esta pregunta se relaciona con la primera, sin embargo, es más pragmática, porque pregunta acerca del rendimiento específico de una inversión en personal, recursos, tiempo y dinero. La ganancia puede ser económica, un aumento de la eficiencia o cualquiera de las otras ventajas de los sistemas expertos descritas en el capítulo 1. También es importante recordar que si nadie usa el sistema, no habrá ganancia. Como los sistemas expertos son una tecnología nueva, es más difícil y arriesgado responder a esta pregunta en relación con la programación convencional.

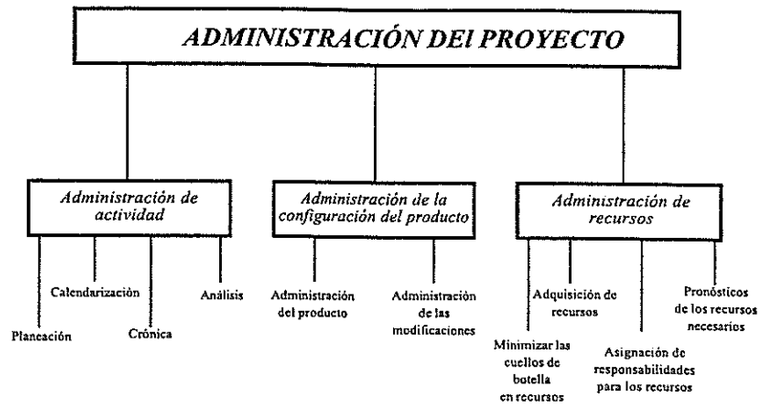


Figura 6.1 Tareas de administración del proyecto

Herramientas

¿Cuáles herramientas están disponibles para construir el sistema?

Hoy en día se encuentran disponibles muchas herramientas para sistemas expertos que presentan ventajas y desventajas. En el apéndice D se resumen las características de algunas de ellas, sin embargo, sólo debe tomarse como una guía debido a que las herramientas de software evolucionan con gran rapidez. Por lo general, puede contarse con una mejora sustancial de cada herramienta cada año y una revisión importante cada dos o tres años.

Estas mejoras no se limitan a las herramientas de software, muchas de las herramientas más novedosas que sólo corrían en máquinas LISP de 50 mil dólares a mediados de los años 80, se reescribieron más tarde para ejecutarse en microcomputadoras o en microprocesadores personalizados. Esto redujo en gran medida el costo del hardware para ejecutar las herramientas. La mejor guía es consultar literatura actual y hablar con gente que ha construido sistemas expertos.

Costo

¿Cuánto costará?

El costo de *construir* un sistema experto depende del personal, los recursos y el tiempo dedicado a su construcción. Además del hardware y el software necesario para ejecutar una

herramienta de sistema experto, también puede haber un costo considerable en la capacitación. Si el personal tiene poca o ninguna experiencia con una herramienta, puede resultar costoso instruirlo, por ejemplo, la capacitación de una sola persona en el manejo de una herramienta moderna para sistemas expertos le costaría 2,500 dólares semanales.

6.3 ETAPAS EN EL DESARROLLO DE UN SISTEMA EXPERTO

¿Cómo se desarrollará el sistema?

En gran medida, el desarrollo de un sistema experto dependerá de los recursos proporcionados. Sin embargo, como cualquier otro proyecto, el desarrollo también dependerá de la forma en que se organice y administre el proceso.

Administración del proyecto

Se espera que la administración de proyecto proporcione lo siguiente; en realidad, la administración de proyecto ha sido el tema de un método de sistema experto (Sathi 86):

Administración de actividad

- planeación
  - definir actividades
  - especificar prioridad de actividades
  - recursos necesarios
  - pruebas comparativas
  - duración
  - responsabilidades
- calendarización
  - asignar tiempos de inicio y final
  - resolver conflictos en las tareas de calendarización con igual prioridad
- crónica
  - vigilar el desempeño del proyecto
- análisis
  - analizar planes, programas y actividades, de las que se hace una crónica

Administración de la configuración del producto

- administración de producto
  - administrar las diferentes versiones del producto
- administración de las modificaciones
  - administrar propuestas de cambio y evaluaciones de impacto
  - asignar personal para hacer los cambios
  - instalar nuevas versiones del producto

Administración de recursos

- pronosticar los recursos necesarios
- adquirir los recursos
- asignar responsabilidades para el uso óptimo de los recursos
- proporcionar los recursos críticos para minimizar cuellos de botella

Para el desarrollo de un sistema experto, las actividades son aquellas tareas que se necesitan para construir el sistema. En la figura 6.2 se muestra una perspectiva de alto nivel de las actividades necesarias para producir un sistema, describiendo las etapas que debe recorrer.

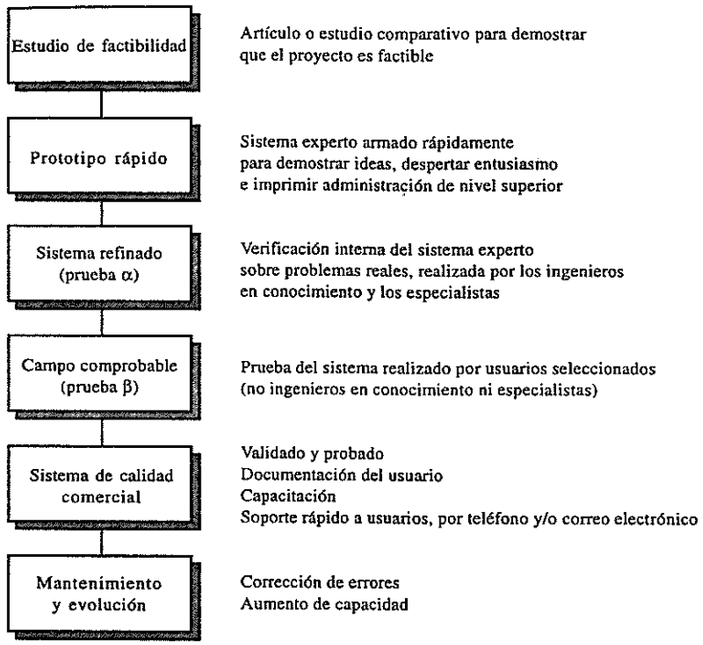


Figura 6.2 Etapas generales del desarrollo de un sistema experto

El problema de distribución

¿Cómo se distribuirá el sistema?

Aunque es conveniente y posible disminuir el tiempo de desarrollo para utilizar una estación de trabajo moderna, a menudo resulta demasiado costoso distribuir el sistema en estas estaciones de trabajo. Por ejemplo, es caro que una estación de trabajo con un valor de 100 mil dólares en hardware y software se use en 100 o más oficinas de campo de una compañía. Además, las cuotas de mantenimiento anual pueden aumentar sustancialmente el costo.

Dependiendo del número de sistemas expertos desplegados, el **problema de distribución** de los sistemas desarrollados puede ser un factor importante del desarrollo. El problema de distribución debe considerarse durante las primeras etapas de desarrollo (Freedman 86).

Idealmente, los sistemas expertos distribuidos deben ejecutarse en hardware estándar. Sin embargo, algunas herramientas para sistemas expertos necesitan una tarjeta LISP de microprocesador especial, que aumentará sustancialmente el costo.

En muchos casos, el sistema experto debe estar integrado con otros programas existentes, por lo que debe tomarse en cuenta la comunicación y coordinación de la entrada/salida entre

ellos. También sería deseable llamar al sistema experto con un procedimiento elaborado con un lenguaje de programación convencional, y el sistema debe soportarlo.

Mantenimiento y evolución

¿Cómo se mantendrá y evolucionará el sistema?

A diferencia de los programas convencionales, el mantenimiento y la evolución de un sistema experto es una actividad con final indeterminado. Como los sistemas expertos no se basan en algoritmos, su desempeño depende del conocimiento y, a medida que se adquiere nuevo conocimiento y se modifica el previo, se perfecciona el desempeño del sistema.

En un producto con calidad comercial, debe haber una forma sistemática y eficiente de recolectar los errores reportados por los usuarios. Aunque la recolección y reparación de los errores informados no es una prioridad alta en los sistemas con nivel de investigación, sí lo es en los de calidad comercial. Sólo puede darse un buen mantenimiento si se obtienen informes de errores.

El mejoramiento de un sistema experto después de la distribución es también de importancia primordial en un sistema con calidad comercial. Los encargados de desarrollar un sistema comercial quieren que sea un éxito financiero, lo que significa escuchar qué quieren los usuarios y cuánto están dispuestos a pagar por las mejoras. En un sentido muy real, un sistema experto comercial nunca se terminará, solamente mejorará.

6.4 ERRORES DURANTE LAS ETAPAS DE DESARROLLO

Los principales errores posibles en el desarrollo de sistemas expertos pueden clasificarse a partir de las etapas en las que probablemente ocurren, como ilustra la figura 6.3. Entre estos errores se incluyen los siguientes:

- **Errores en el conocimiento del especialista.** Como el especialista es la **fuente del conocimiento** del sistema experto, si su conocimiento es erróneo, los resultados pueden propagarse por todo el proceso de desarrollo. Como se menciona en el capítulo 1, un beneficio colateral de construir un sistema experto es la posibilidad de detectar conocimiento erróneo cuando el conocimiento del especialista se hace explícito.

  En los proyectos con una **misión crítica,** en los que la vida y la propiedad están en riesgo, tal vez sea necesario configurar procedimientos para certificar el conocimiento del especialista. Un método que la NASA ha utilizado con éxito para vuelos espaciales es el de **paneles de técnica de vuelo,** los cuales revisan regularmente soluciones de problemas y las técnicas de análisis usadas para desarrollarlas (Culbert 87). Estos paneles están integrados por los usuarios del sistema, especialistas independientes en el tema, encargados de desarrollar sistemas y administradores, para asegurar la cobertura de todas las áreas que afectan el desarrollo.

  La ventaja de un método de panel es que el conocimiento del especialista se inspecciona minuciosamente al inicio del proceso de desarrollo, cuando es más fácil corregir el conocimiento equivocado. Cuanto más conocimiento erróneo pase inadvertido, más costoso será corregirlo, si el conocimiento del especialista no se verifica al principio, la prueba definitiva es la validación del sistema experto. La validación final del sistema experto demuestra si el sistema cumple con los requisitos, en especial que las soluciones sean correctas y estén completas.

  La desventaja del este método es, principalmente, su costo. Sin embargo, es posible compensarlo mediante que una mayor eficiencia en el proceso de desarrollo.
- **Errores semánticos.** Un error de semántica se presenta cuando no se comunica de forma apropiada el significado del conocimiento. Un ejemplo muy sencillo sería si un especialista

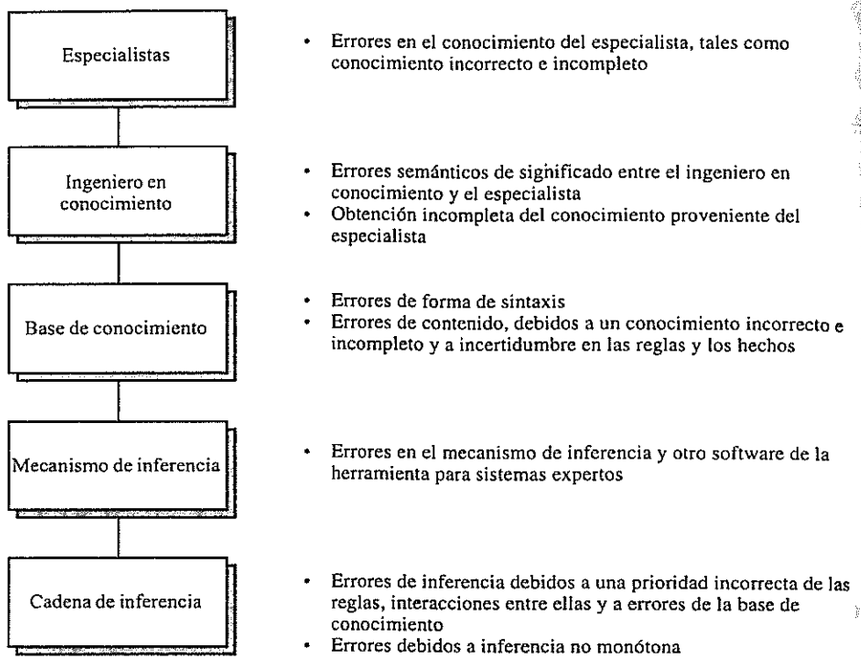


Figura 6.3 Principales errores en los sistemas expertos y algunas de sus causas

dice "Usted puede extinguir un fuego con agua" y el ingeniero en conocimiento lo interpreta como "Todos los fuegos se pueden extinguir con agua". Los errores semánticos ocurren si el ingeniero en conocimiento malinterpreta las respuestas del especialista, si éste malinterpreta las preguntas del ingeniero, o cuando se dan ambos casos.

• **Errores de sintaxis.** Son errores simples que ocurren cuando se introduce la forma incorrecta de una regla o un hecho. Las herramientas del sistema experto pueden marcar estos errores y emitir un mensaje apropiado. Otros errores que suceden en la fase de construcción de la base de conocimiento se deben a errores de la fuente de conocimiento que no se detectaron en las etapas previas.

• **Errores del mecanismo de inferencia.** Igual que con cualquier otra pieza de software, el mecanismo de inferencia puede tener errores. Cuando una herramienta de sistema experto se libera para uso general, todos los errores comunes deben estar corregidos. Sin embargo, tal vez haya errores que sólo se revelan en circunstancias poco frecuentes, como cuando se tienen 159 reglas en la agenda. Algunos errores pueden ser muy sutiles y sólo se descubren en ciertas operaciones de correspondencia de patrón.

En general, los errores del mecanismo de inferencia pueden descubrirse durante la correspondencia de patrones, la resolución de conflictos y la ejecución de acciones. Esos errores pueden ser difíciles de detectar si no son continuos. Si la herramienta del sistema experto se utiliza para aplicaciones con una misión crítica, se podría determinar cómo se validó la herramienta.

El método más simple para buscar errores en las herramientas es el clásico de preguntar a otros usuarios y al vendedor, este último estaría dispuesto a proporcionar una lista de clientes, informes de errores y arreglos, además del tiempo que ha estado en uso la herramienta; un grupo de usuarios es una excelente fuente de información.

• **Errores en la cadena de inferencia.** Estos errores pueden deberse a conocimiento erróneo, errores de semántica o en el mecanismo de inferencia, especificación incorrecta de las prioridades de una regla e interacciones imprevistas entre reglas. Los errores en la cadena de inferencia más complejos se deben a la incertidumbre en las reglas y en la evidencia, la propagación de incertidumbre en la cadena de inferencia y no monotonía.

La elección de un método para tratar con la incertidumbre no resuelve automáticamente todos los temas relacionados con ella. Por ejemplo, antes de elegir la inferencia bayesiana simple, debe revisarse si está garantizada la suposición de independencia condicional.

• **Límites de los errores de ignorancia.** Un problema común para todas las etapas de desarrollo es la especificación de los límites de ignorancia del sistema. Como se explica en el capítulo 1, los peritos humanos conocen la extensión de su conocimiento, su desempeño se degrada suavemente (esperamos) en los límites de su ignorancia, y deben ser lo bastante honestos para admitir más incertidumbre en sus conclusiones cerca de esos límites. No obstante, a menos que un sistema experto esté programado específicamente para admitir incertidumbre, continuaría proporcionando respuestas aunque las cadenas de inferencia y evidencia sean débiles.

6.5 INGENIERÍA DE SOFTWARE Y SISTEMAS EXPERTOS

En las secciones anteriores se han explicado las consideraciones generales para el uso del paradigma del sistema experto. Ahora examinaremos las etapas de desarrollo del sistema experto desde el punto de vista más técnico, del ingeniero en conocimiento, que es en realidad quien construye el sistema.

Como los sistemas expertos se han alejado de la etapa de investigación existe la necesidad real de entregar software con la calidad necesaria para cumplir las normas del software convencional. La metodología aceptada para el desarrollo de software de calidad de acuerdo con las normas comercial, industrial y de gobierno es la ingeniería de software.

Es importante seguir buenos estándares en el desarrollo de un producto, de otra manera no será de buena calidad. Ahora los sistemas expertos deben considerarse como cualquier otro producto de software: un procesador de palabras, un programa de nómina, un juego de computadora, etc. Sin embargo, hay una diferencia trascendente entre la **misión** de los sistemas expertos y los productos de consumo típico como procesadores de palabras y juegos de video, entendiendo como *misión* al al propósito global de un proyecto o una tecnología. Por lo general, la tecnología de los sistemas expertos tiene la seria misión de proporcionar experiencia en situaciones de alto desempeño y posiblemente arriesgadas, donde la vida humana y la propiedad están en juego, que son las aplicaciones de misión crítica mencionadas en la sección anterior.

Las aplicaciones para una misión crítica son diferentes de la misión más relajada de procesadores de palabras y juegos de video, que es aumentar la eficiencia y proporcionar recreación, ninguna vida humana depende de un error en alguno de ellos (o por lo menos no debería).

Los sistemas expertos son sistemas de alto desempeño que deben tener alta calidad o estarán propensos a errores. La ingeniería de software proporciona metodologías para construir software de calidad, como se ilustra en la figura 6.4.

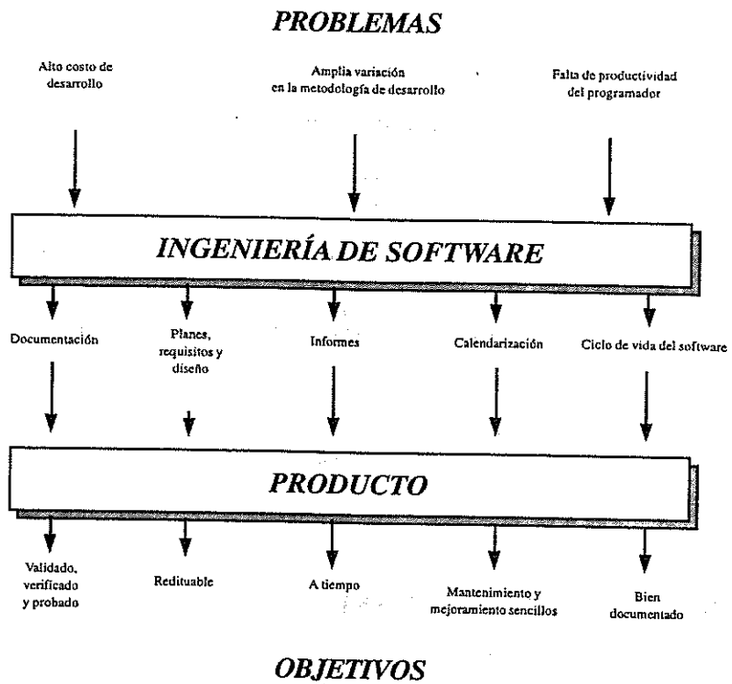


Figura 6.4 Metodología de la ingeniería de software

Calidad es un término difícil de describir en un sentido general porque significa cosas diferentes para personas diferentes; una forma de definirla es como los atributos necesarios y deseables de un objeto determinado en cierta escala. El término *objeto* se utiliza aquí para denominar a cualquier software o hardware; a los atributos y sus valores se les llama ***mediciones*** porque se usan como medidas de un objeto. Por ejemplo, la confiabilidad medida de una unidad de disco es una medición de su calidad. Una medida de este atributo es el **tiempo medio entre fallas** (MTBF, mean time between failures) de la unidad. Una unidad confiable tendría un MTBF de 1000 horas de uso entre descomposturas, mientras que una unidad no confiable tendría un MTBF de 100 horas.

En la tabla 6.1 se presenta la lista de verificación para algunas mediciones que se usarían para evaluar la calidad de un sistema experto. Estas mediciones deben tomarse sólo como guía, porque un sistema experto específico puede tener más o menos de ellas. Sin embargo, el concepto importante es tener la lista de mediciones necesarias que pueda usarse para describir la calidad.

Una lista de mediciones permite asignarles prioridades más fácilmente, porque muchas pueden estar en conflicto con otras, puesto que si se incrementa el proceso de prueba de un sistema experto para cerciorarse de que sea correcto, aumentará el costo. Decidir cuándo termina la prueba suele ser una decisión compleja que involucra factores como la calendarización, costo y requisitos, necesidades que es deseable satisfacer. En la práctica, puede juzgarse que algunas son más importantes que otras y las restricciones para satisfacer todos los factores se debilitarán.

Salidas correctas dada una entrada correcta
Salida completa dada una entrada correcta
Salida congruente dada una misma entrada otra vez
Confiable de modo que no se bloquee (a menudo) debido a errores
Para que lo use la gente y preferiblemente amigable con el usuario
Con posibilidades de darle mantenimiento
Mejorable
Validado para probar que satisface las necesidades y los requisitos del usuario
Probado para comprobar que es correcto y está completo
Redituable
Código reutilizable para otras aplicaciones
Mudable a otros ambientes de software/hardware
Sirve como interfaz con otro software
Código comprensible
Exacto
Preciso
Degradación suave de sus límites de conocimiento
Capacidad de insertarse en otros lenguajes
Base de conocimiento verificada
Medio de explicación

Tabla 6.1 Algunas mediciones de la calidad del software para los sistemas expertos

6.6 EL CICLO DE VIDA DEL SISTEMA EXPERTO

Una de los métodos clave de la ingeniería de software es el **ciclo de vida,** que es el periodo que empieza con el concepto inicial del software y termina con su retiro del uso. Más que pensar en desarrollo y mantenimiento por separado, el concepto del ciclo de vida proporciona una continuidad que conecta todas las etapas. La planeación para el mantenimiento y la evolución en las primeras etapas del ciclo de vida reduce el costo de estas etapas más adelante.

Costos de mantenimiento

Para software convencional, el mantenimiento suele representa 60 a 80 por ciento de todos los costos del software y con frecuencia es dos o cuatro veces el costo original de desarrollo. Aunque hay poca información acerca del mantenimiento de sistemas expertos debido a que son muy nuevos, las cifras probablemente serán peores. Si los programas convencionales con algoritmos conocidos requieren demasiado mantenimiento, entonces los sistemas expertos probablemente necesitarán más mantenimiento, porque se basan en mucho conocimiento heurístico y experimental. Los sistemas expertos que hacen mucha inferencia en condiciones de baja incertidumbre deben ser aún más susceptibles a elevados costos de mantenimiento y de evolución.

Modelo de cascada

Se han desarrollado varios modelos diferentes de ciclo de vida para el software convencional. El modelo clásico de cascada, familiar para los programadores de software convencional, fue el modelo original de ciclo de vida y se ilustra en la figura 6.5 (Boehm 84).

En este modelo *de cascada,* cada etapa termina con una actividad de verificación y validación (V & V) para minimizar algunos problemas en esa etapa. También, observe que las flechas

van hacia atrás y hacia adelante únicamente de etapa en etapa, esto representa el desarrollo reiterativo entre dos etapas adyacentes para reducir todo lo posible el costo, en comparación con el mayor costo de iteración del desarrollo sobre varias etapas.

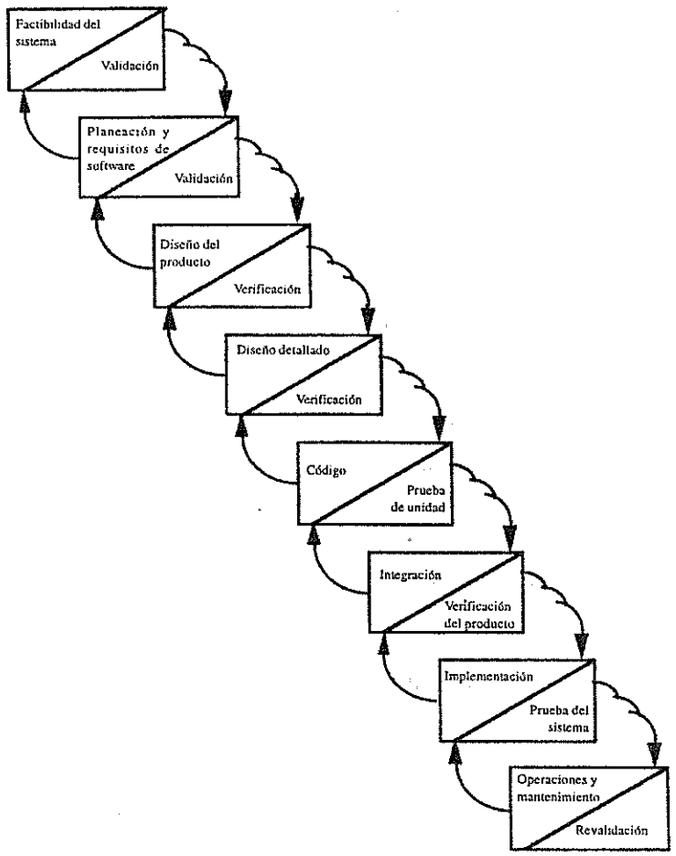


Figura 6.5 Modelo de cascada para el ciclo de vida del software

Otro término para el ciclo de vida es ***modelo de proceso,*** porque se relaciona con los dos temas fundamentales del desarrollo de software:

(1) ¿Qué debo hacer a continuación?
(2) ¿En cuánto tiempo debe realizarse la siguiente etapa?

El modelo de proceso es en realidad una **metametodología** porque determina el orden en el que se aplican los métodos para el software común, y su duración. Los métodos (o metodologías) de desarrollo del software común muestran métodos específicos para cumplir una etapa, como:

- planeación
- requisitos
- adquisición de conocimiento
- pruebas
- representación de productos de etapa
- documentación
- código
- diagramas

Modelo de código y reparación

Se han utilizado varios modelos de proceso para el desarrollo de software. El "modelo" más primitivo es el famoso **modelo de código y reparación**, con él se escribe algo del código y luego se repara cuando no trabaja de forma correcta (Boehm 88). Éste suele ser el método que eligen los nuevos estudiantes de programación tanto en programación convencional como en sistemas expertos.

Por el año 1970, las deficiencias en el método de código y reparación eran tan obvias, que se desarrolló el modelo de cascada para proporcionar una metodología sistemática que fue especialmente útil para proyectos grandes. Sin embargo, hubo dificultades con este modelo porque suponía que ya era conocida toda la información necesaria para una etapa. En la práctica, a menudo no era posible escribir los requisitos completos hasta que se había construido un prototipo, lo que condujo al concepto **"hágalo dos veces"**, en el que se construía un prototipo, se determinaban los requisitos y después se construía el sistema final.

Modelo progresivo

El **modelo de cascada progresivo** es un refinamiento del modelo de cascada y progresivo. La idea básica del desarrollo progresivo es producir software en incrementos de su capacidad funcional. El modelo progresivo se ha utilizado con éxito en grandes proyectos de software convencional y también es útil para el desarrollo de sistemas expertos, donde la adición de reglas aumenta la capacidad del sistema para pasar de nivel de asistente a colega y finalmente a especialista. De esta manera, en un sistema experto, los **incrementos principales** son de asistente a colega y de colega a especialista, los **incrementos menores** corresponden al aumento de experiencia dentro de cada nivel que ofrece algún incremento significativo y un **micro incremento** es el cambio en la experiencia al agregar o refinar una regla individual.

La principal ventaja de este modelo progresivo es que los incrementos en la capacidad funcional resultan más fáciles de probar, verificar y validar que los productos de etapas individuales del modelo de cascada. Cada incremento funcional puede probarse, verificarse y validarse de forma inmediata con el especialista en lugar de tratar de hacer la validación completa al final, lo que disminuye el costo de incorporar correcciones al sistema. En esencia, el modelo progresivo es similar a un prototipo rápido continuo que se extiende sobre todo el desarrollo, pero en lugar del prototipo rápido de las etapas iniciales para determinar los requisitos usados con el método "hágalo dos veces", la evolución del prototipo *es* el sistema.

Modelo en espiral

Una forma de visualizar el modelo progresivo es una adaptación del **modelo en espiral** convencional, como se muestra en la figura 6.6. Cada circuito de la espiral agrega alguna capacidad funcional al sistema. El rótulo de punto final, "Sistema distribuido", en realidad no es el fin

de la espiral, sino que iniciará una nueva espirial con el mantenimiento y la evolución del sistema. La espiral puede refinarse aún más para especificar con mayor precisión las etapas generales de Adquisición de conocimiento, Codificación, Evaluación y Planeación.

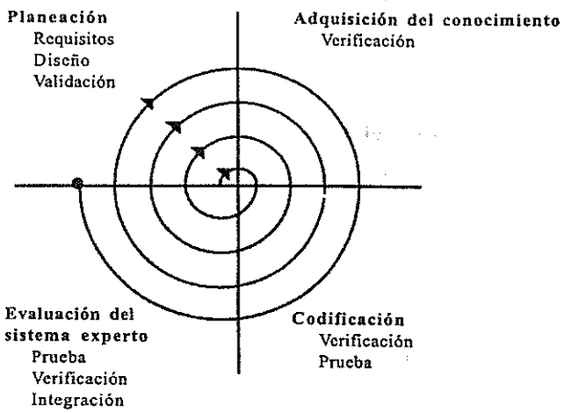


Figura 6.6 Modelo en espiral del desarrollo de sistemas expertos

6.7 MODELO DETALLADO DEL CICLO DE VIDA

El modelo del ciclo de vida utilizado con éxito en varios proyectos de sistemas expertos es el **modelo lineal**, ilustrado en la figura 6.7, adaptado de Bochsler (Bochsler 88). Este ciclo de vida está formado por varias etapas, que van de la planeación a la evaluación de sistemas, y describe el desarrollo del sistema hasta el punto en que se evaluarán sus capacidades funcionales. Después de esto, el ciclo de vida repite la misma secuencia de planeación a evaluación hasta que el sistema se distribuye para uso rutinario; en consecuencia, se usa para el mantenimiento y la evolución posteriores del sistema. Aunque no se muestra de manera explícita, la verificación y validación proceden en paralelo con las etapas. Más que sólo corregir algunos errores, es importante seguir la misma secuencia de etapas para mantener la calidad del sistema experto, porque saltar etapas o incluso reparar un pequeño error, daña la calidad del sistema completo.

El ciclo de vida mostrado podría considerarse un circuito del modelo en espiral, donde cada etapa está formada por **tareas**. Quizá no sean necesarias todas las tareas para una etapa, en especial una vez que el sistema se encuentra en mantenimiento y evolución; en cambio, las tareas tienen la intención de servir como una composición de todas las tareas para el ciclo de vida entero, desde el concepto inicial hasta el retiro del software. Las tareas también dependerán del tipo exacto de aplicación que se está construyendo, sólo deben considerarse como una guía y no como requisitos absolutos que deben realizarse para completar cada etapa.

Este modelo se explicará con todo detalle para ilustrar los muchos factores que deben considerarse para un sistema experto de gran calidad. Para pequeños prototipos de investigación que no están orientados al uso general, no son necesarias todas las tareas e incluso las etapas. Sin embargo, es asombrosa la cantidad de software que se desarrolla para uso personal o investigación que se libera a los socios y luego se lanza para uso general.

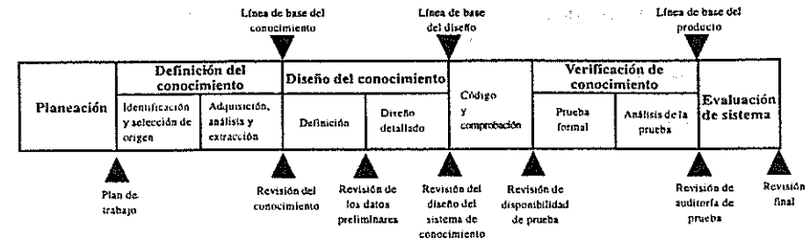


Figura 6.7 Modelo lineal del ciclo de vida para desarrollar sistemas expertos

Planeación

El propósito de la etapa de planeación es producir un **plan de trabajo** formal para desarrollar el sistema experto. El plan de trabajo es un conjunto de documentos que se utilizará para guiar y evaluar el desarrollo. En la tabla 6.2 se ilustran las tareas de esta etapa.

Tarea	Objetivo
Valoración de factibilidad	Determinar si vale la pena construir el sistema y, de ser así, si debe utilizarse la tecnología de sistema experto.
Administración de recursos	Evaluar los recursos necesarios de personal, tiempo, dinero, software y hardware. Cómo adquirir y administrar esos recursos.
Determinación de las fases de las tareas	Especificar las tareas y su orden en las etapas.
Calendarización	Especificar las fechas de inicio y distribución de tareas en las etapas.
Disposición funcional preliminar	Definir *qué* debe lograr el sistema mediante la especificación de las funciones de alto nivel del sistema. Esta tarea especifica el propósito del sistema.
Requisitos de alto nivel	Describir en términos de alto nivel *cómo* se llevarán a cabo las funciones del sistema.

Tabla 6.2 Planeación de las tareas de la etapa

La valoración de factibilidad es la tarea más importante del ciclo de vida y debe responder si el proyecto vale la pena y si un sistema experto es el paradigma apropiado. La respuesta a estas dos preguntas determina si el proyecto debe seguir usando una aproximación de sistemas expertos. Muchos factores están relacionados en la valoración de factibilidad, como explicamos en la sección 6.1, e incluyen la selección de un dominio apropiado, costo, utilidad y otros factores del sistema experto.

Definición de conocimiento

El objeto de la **etapa de definición de conocimiento** es definir el conocimiento requerido por el sistema experto. Esta etapa está conformada por dos tareas principales, como sigue:

- identificación y selección del origen del conocimiento
- adquisición, análisis y extracción del conocimiento

Cada una de estas importantes tareas está compuesta por otras tareas. En la tabla 6.3 se describen las tareas relacionadas con la identificación y selección del origen.

Tarea	Objetivo
Identificación del origen	Quién y cuáles son los orígenes del conocimiento, sin considerar la disponibilidad.
Importancia del origen	Lista con prioridades de orígenes de conocimiento, en orden de importancia para el desarrollo
Disponibilidad del origen	Lista de orígenes del conocimiento clasificados en orden de disponibilidad. Es más fácil disponer de libros y otros documentos que de especilistas humanos.
Selección del origen	Seleccionar los orígenes del conocimiento con base en la importancia y la disponibilidad.

Tabla 6.3 Tareas de identificación y selección del origen del conocimiento

Las tareas adquisición, análisis y extracción están descritas en la tabla 6.4.

Tarea	Objetivo
Estrategia de adquisición	Especificar cómo se adquirirá el conocimiento mediante métodos para entrevistar expertos, leer documentos, inducción de regla, rejillas de repertorio, etcétera.
Identificación de los elementos del conocimiento	Elegir el conocimiento específico de los orígenes que serán útiles en esta iteración del ciclo de vida.
Sistema de clasificación del conocimiento	Clasificar y organizar el conocimiento como ayuda para la verificación y comprensión del conocimiento por parte de quienes desarrollan el sistema mediante desarrolladores. Utilice grupos jerárquicos cada vez que sea posible.
Disposición funcional detallada	Especificar de manera detallada las capacidades funcionales del sistema. Esto es a un nivel más técnico, mientras que la disposición de funcionalidad preliminar se encontraba en un nivel administrativo.
Flujo de control preliminar	Describir fases generales que ejecutará el sistema experto. Estas fases corresponden a reglas lógicas que se activan/desactivan en grupos para controlar el flujo de la ejecución.
Manual preliminar del usuario	Es una parte a menudo ignorada pero esencial del sistema describirlo desde el punto de vista del usuario. Es absolutamente importante relacionar a los usuarios en cuanto sea posible para retroalimentación. Si ellos no utilizan el sistema, no vale la pena.
Especificaciones de requisitos	Definir exactamente qué se supone que hace el sistema. El sistema experto se validará usando estos requisitos.
Línea referencial de conocimiento	Es la línea referencial de conocimiento para el sistema. Cualquier cambio debe darse ahora a través de un requisito de cambio formal. Ahora el conocimiento de alto nivel es adecuado para la etapa de diseño del conocimiento.

Tabla 6.4 Tareas de adquisición, análisis y extracción del conocimiento

El objetivo principal de la **tarea de adquisición del conocimiento,** la **tarea de análisis del conocimiento** y la **tarea de extracción de conocimiento** es producir y verificar el conocimiento que necesita el sistema. En cuanto se define una línea referencial para el conocimiento, debe

ser correcto y adecuado para la siguiente etapa de diseño del conocimiento. Además del método acostumbrado de entrevistar especialista, podrían utilizarse técnicas como las **rejillas de repertorio** o la **teoría de construcción personal** para la adquisición automática de conocimiento (Boose 84).

Diseño del conocimiento

El objetivo de la **etapa de diseño del conocimiento** es producir el diseño detallado para un sistema experto. Son dos las tareas principales que integran esta etapa:

- definición del conocimiento
- diseño detallado

En la tabla 6.5 se describen las tareas asociadas con la definición del conocimiento.

Tarea	Objetivo
Representación del conocimiento	Especificar cómo se representará el conocimiento: como reglas, marcos o lógica, dependiendo de la herramienta de sistemas expertos que le dará soporte.
Estructura de control detallada	Especificar tres estructuras de control general: (1) Si el sistema se insertó en código de procedimiento, cómo se llamará. (2) Control de los grupos de reglas relacionados dentro de un sistema de ejecución. (3) Estructuras de control de metanivel para reglas.
Estructura interna de hechos	Especificar la estructura interna de hechos de manera congruente para ayudar a la comprensión y el buen estilo.
Interfaz preliminar del usuario	Especificar una interfaz preliminar con el usuario. Obtener retroalimentación de los usuarios acerca de la interfaz.
Plan de prueba inicial	Especificar cómo se probará el código. Definir los datos y controladores de prueba y la forma en que se analizarán los resultados.

Tabla 6.5 Tareas de definición de conocimiento

Las estructuras internas de hechos descritas en la tabla 6.5 se analizan con mucho mayor detalle en los capítulos de CLIPS. La idea básica de la especificación de estructuras de hechos es adaptar buen estilo. Por ejemplo, un hecho como "10" no es muy significativo por sí solo. ¿Qué representa el "10"? Si se incluye información adicional con el hecho como "precio 10" o, mejor aún, "precio del oro 10", entonces el precio del oro es significativo. Observe que esta forma del hecho se encuentra en la forma convencional de objeto-atributo-valor y, de este modo, la gente puede leerlo y entenderlo de manera conveniente. CLIPS soporta esto mediante la construcción de plantillas de definición para reglas y también mediante objetos.

En algunos de los lenguajes para sistema experto los campos puede tener una **escritura fuerte** para que sólo se permitan ciertos valores, así, cuando una regla trata de especificar un valor que no está permitido, el mecanismo de inferencia lo señala como un error. En la tabla 6.6 se muestra la **etapa de diseño detallado** del conocimiento.

El producto de la etapa de diseño detallada es el documento de diseño que se toma como línea referencial del que puede proceder la codificación. Antes de iniciar la codificación, el documento del diseño de la línea referencial experimenta una revisión del diseño de sistema de conocimiento como una verificación final

Tarea	Objetivo
Estructura de diseño	Especificar cómo se organiza lógicamente el conocimiento en la base de conocimiento y qué hay en ella.
Estrategia de implantación	Especificar la forma en que se establece el sistema.
Interfaz detallada del usuario	Especificar la interfaz del usuario detallada, después de recibir retroalimentación por el uso del diseño preliminar.
Especificaciones e informe de diseño	Documentar el diseño.
Plan detallado de prueba	Especificar exactamente cómo se probará y verificará el código.

Tabla 6.6 Diseño detallado de las tareas de conocimiento

Código y verificación

En la tabla 6.7 se describe la **etapa de código y verificación,** que inicia la implantación del código actual.

Tarea	Objetivo
Codificación	Poner en práctica la codificación.
Pruebas	Probar el código utilizando los datos de prueba, probar controladores y probar procedimientos de análisis.
Listados de origen	Producir código fuente comentado y documentado.
Manual del usuario	Elaborar un manual de trabajo para el usuario, de manera que los especialistas y los usuarios puedan retroalimentar al sistema.
Guía de instalación/operaciones	Documentar la instalación/operación del sistema para usuarios.
Documento de descripción de sistema	Documentar funcionalidad, limitantes y problemas globales del sistema.

Tabla 6.7 Tareas de código y verificación

En esta etapa se termina con la **revisión de disponibilidad de prueba,** que determina si el sistema experto está listo para la siguiente etapa: verificación del conocimiento.

Verificación del conocimiento

El objetivo de esta etapa consiste en determinar que el sistema sea correcto, esté completo y sea congruente. Esta etapa se divide en dos tareas principales:

- pruebas formales
- análisis de prueba

En la tabla 6.8 se describe la **tarea de prueba formal** de la etapa de verificación de conocimiento.

Tarea	Objetivo
Procedimientos de prueba	Realizar los procedimientos de prueba formal.
Informes de prueba	Documentar los resultados de la prueba.

Tabla 6.8 Tareas de prueba formal de la etapa de verificación de conocimiento

En la tabla 6.9 se muestran las **tareas de análisis de prueba.** El análisis de prueba busca los siguientes problemas importantes:

- respuestas incorrectas
- respuestas incompletas
- respuestas incongruentes

y determina si el problema se encuentra en reglas, cadenas de inferencia, incertidumbre o alguna combinación de estos tres factores. Si el problema no puede definirse en el sistema experto, entonces es necesario analizar el software de la herramienta para sistemas expertos.

Tarea	Objetivo
Evaluación de los resultados	Analizar los resultados de la prueba.
Recomendaciones	Documentar las recomendaciones y conclusiones de pruebas.

Tabla 6.9 Tareas de análisis de prueba

Evaluación del sistema

Como se describe en la tabla 6.10, la etapa final del ciclo de vida del desarrollo es la **etapa de evaluación del sistema**, el propósito de esta etapa es resumir lo que se ha aprendido con las mejoras y correcciones recomendadas.

Tarea	Objetivo
Evaluación de resultados	Resumir los resultados de la prueba y la verificación.
Recomendaciones	Recomendar cualquier cambio al sistema.
Validación	Validar que el sistema está correcto en relación con las necesidades y requisitos del usuario.
Informe provisional o final	Si el sistema está completo, elaborar informes finales; de lo contrario, elaborar uno provisional.

Tabla 6.10 Tareas de la etapa de evaluación del sistema

Como una sistema experto suele estar construido a partir de iteraciones, usualmente el informe de la etapa de evaluación del sistema será un informe provisional que describa la funcionalidad incrementada del sistema a medida que se añade nuevo conocimiento. Sin embargo, deben verificarse las nuevas posibilidades del sistema de manera independiente y como parte del conocimiento previo del sistema; es decir, el sistema de verificación también debe ponerse en acción junto con todo el conocimiento del sistema, no sólo el nuevo conocimiento. El sistema experto también debe validarse cada vez en esta etapa en lugar de esperar a la iteración final. En la actualidad se están investigando sistemas automatizados para validación de bases de conocimiento (Stachowitz 87).

6.8 RESUMEN

En este capítulo se analizó un método de ingeniería de software para la construcción de sistemas expertos. Ahora que la tecnología de sistemas expertos se está utilizando para resolver problemas reales, los sistemas expertos *deben* ser productos de calidad.

Deben considerarse varios factores en el diseño de un sistema experto, como la selección de problemas, el costo y la rentabilidad. Deben considerarse los aspectos administrativos y técnicos para construir un sistema exitoso.

Un concepto muy útil de la ingeniería de software es el ciclo de vida, éste considera al desarrollo de software como una serie de etapas que van del concepto inicial al retiro del software. Al seguir continuamente un ciclo de vida, es posible construir software de calidad.

Se analizaron varios modelos diferentes de ciclo de vida para sistemas expertos, y uno de ellos se mostró en detalle.

PROBLEMAS

6.1 Considere un sistema simple basado en conocimiento para problemas de diagnóstico en automóviles. Describa este sistema propuesto para cada etapa del modelo lineal (no es necesario hacer esto para cada tarea). Suponga que mucha gente trabaja en su proyecto y considere la coordinación de sus esfuerzos. Explique cualquier suposición que haga.

6.2 Escriba un informe sobre una herramienta automática para la validación de bases de conocimiento como EVA (Stachowitz 87). Elabore una lista de los problemas que puede detectar y sus limitantes.

6.3 Explique cualquier cambio o diferencia en el modelo lineal para el desarrollo de un proyecto más grande en el que hay muchos encargados de su desarrollo, en comparación con un proyecto pequeño en que interviene una sola persona.

BIBILIOGRAFÍA

(Bochsler 88). Daniel C., Bochsler, "A Project Management Approach to Expert System Applications", *ISA* 88. pp. 1458-1466, 1988.

(Boehm 84). Barry W. Boehm, "Software Life Cycle Factors", en *Handbook of Software. Engineering*, C. R. Vick y C. V. Ramamoorthy. Eds. Van Nostrand Reinhold Co., . pp. 494-518, 1984.

(Boehm 88). Barry W. Boehm, "A Spiral Model of Software Development and Enhancement", en *IEEE Computer*, pp. 61-72, mayo de 1988.

(Boose 84). John H. Boose, "Personal Construct Theory, and the Transfer of Human Expertise", en *AAA I84*, pp. 27-33, 1984.

(Culbert 87). Chris Culbert, Gary Riley y Robert T. Savely, "Approaches to the Verification of Rule-based Expert Systems", NASA Conference Publication 2491, *First Annual Workshop on Space Operations Automation and Robotics* (*SOAR 87*), NASA, Houston, Texas, pp. 191-196, 1987.

(Freedman 86). Roy S. Freedman y Robert P. Frail, "OPGEN: The Evolution of an Expert System for Process Planning", *The AI Magazine*, pp. 58-70, invierno de 1986.

(Sathi 86). Arvind Sathi, Thomas E. Morton y Steven F. Roth, "Callisto: An Intelligent Project Management System", en *The AI Magazine*, pp. 34-52, invierno de 1986.

(Stachowitz 87). R. A. Stachowitz, *et al.*, "Building Validation Tools for Knowlegde-based Systems", en *First Annual Workshop on Space Operations Automation and Robotics* (*SOAR 87*), NASA, Houston, Texas, pp. 209-216, 1987.

CAPÍTULO 7

Introducción a CLIPS

7.1 INTRODUCCIÓN

Con este capítulo inicia la introducción de los conceptos prácticos necesarios para construir un sistema experto. Del capítulo 1 al 6 se observaron los antecedentes, la historia, las definiciones, la terminología, los conceptos, las herramientas y las aplicaciones de los sistemas expertos; en resumen, en ellos se proporciona una idea de lo que son y lo que pueden hacer los sistemas expertos. Este marco teórico de conceptos y algoritmos resulta esencial para construirlos, sin embargo muchos aspectos prácticos de su elaboración deben aprenderse con la práctica, ya que construir un sistema experto es muy parecido a escribir un programa en un lenguaje de procedimientos. Conocer la forma en que trabaja un algoritmo no es lo mismo que poder escribir un programa por procedimientos para ejecutarlo, del mismo modo, capturar el conocimiento de un especialista no es lo mismo que construir un sistema. Por esta razón, la experiencia práctica en el uso de la herramienta de un sistema experto es de un valor incalculable para el proceso de aprendizaje acerca de sistemas expertos.

El lenguaje de sistema experto que se utilizará para demostrar varios conceptos en el resto de este libro es CLIPS, que soporta tres tipos de paradigmas de programación: basado en reglas, orientado a objetos y por procedimiento; enfocándose principalmente en los aspectos de la programación basada en reglas. Los manuales que acompañan al programa proporcionan cobertura detallada de todas sus funciones, incluso aquellas que no se explican en este libro. En este capítulo se describen los componentes básicos de un sistema experto basado en reglas (como los analizados en el capítulo 1) que se encuentran dentro de CLIPS como son:

1. **lista de hechos:** contiene los datos de los que se derivan las inferencias
2. **base de conocimiento:** comprende todas las reglas
3. **mecanismo de inferencia:** controla la ejecución global

Estos tres componentes de CLIPS serán el enfoque primordial de este capítulo. El primero, los hechos, se explicará con todo detalle, se abordarán los hechos de adición, eliminación, modificación, duplicación, despliegue y trazado, seguidos de una explicación de la forma en que las reglas de un programa CLIPS interactúan con los hechos, para que un programa se ejecute. Por último, se demostrará la interacción de las reglas múltiples.

7.2 CLIPS

CLIPS es un lenguaje de programación con paradigmas múltiples que proporciona soporte para programación basada en reglas, orientada a objetos y por procedimientos. Las capacidades de inferencia y representación que proporciona el lenguaje de programación basado en reglas de CLIPS son similares a las del OPS5, pero con mayor número de funciones. En cuanto a la sintaxis, las reglas de CLIPS se parecen mucho a las de lenguajes como ART, ART-IM, Eclipse y Cognate, pero sólo soporta reglas de encadenamiento hacia adelante (y no hacia atrás).

Las opciones de programación orientada a objeto de CLIPS, conocidas colectivamente como lenguaje orientado a objeto de CLIPS (COOL, por las siglas de CLIPS Object-Oriented Language), son una combinación híbrida de funciones encontradas en otros lenguajes orientados a objetos, como el Sistema común de objeto de Lisp (CLOS, por las siglas de Common Lisp Object System) y SmallTalk, además de varias ideas nuevas. El lenguaje de programación por procedimientos que proporciona CLIPS tiene características parecidas a lenguajes como C, Ada y Pascal y su sintaxis es también parecida a LISP.

CLIPS, acrónimo de Sistema de producción integrada en lenguaje C por sus siglas, (C Language Integration Production System), se diseñó utilizando el lenguaje de programación C en el Centro espacial Johnson de la NASA, con el propósito específico de proporcionar una gran capacidad de transporte, bajo costo y fácil integración con sistemas externos. Sin embargo, partes del acrónimo de CLIPS pueden ser engañosos, pues originalmente CLIPS sólo proporcionaba soporte para programación basada en reglas (de allí lo de "Sistema de producción"). Posteriormente en la versión 5.0 se introdujo soporte de programación por procedimientos y orientada a objetos. La parte "Lenguaje C" del acrónimo también es engañosa porque hay una versión de CLIPS desarrollada enteramente en Ada.

Debido a su facilidad de transporte, CLIPS se ha instalado en diversas computadoras que van de PC hasta supercomputadoras CRAY. La mayor parte de los ejemplos que se muestran en este capítulo y los siguientes deben funcionar en cualquier computadora que se haya instalado CLIPS, pero se recomienda que los usuarios tengan algún conocimiento del sistema operativo, ya que el método para especificación de archivos tiende a variar de un sistema operativo a otro. Esto se nota cuando los comandos varían dependiendo de la máquina o el sistema operativo.

7.3 NOTACIÓN

En este capítulo y los siguientes se usará la misma notación para describir la sintaxis de los diversos comandos y construcciones que se introducen. Esta notación incluye tres tipos diferentes de texto.

El primer tipo de notación es para símbolos y caracteres que se habrán de introducir exactamente como se muestra; esto se refiere a cualquier cosa que no esté encerrada entre los caracteres < >, [ ] o { }. Por ejemplo, considerando la siguiente descripción de sintaxis:

```
(ejemplo)
```

Esta descripción de sintaxis significa que *(ejemplo)* debe introducirse como se muestra. Para ser exactos, debe introducirse primero el carácter '(', seguido por el carácter 'e', después 'j', 'e', 'm', 'p', 'l', 'o' y finalmente el carácter ')'.

Los corchetes, [ ], indican que el contenido de los paréntesis es opcional. Por ejemplo, la descripción de sintaxis


```
(ejemplo [1])
```

indica que el 1 que se encuentra entre paréntesis es opcional. Así que la siguiente entrada sería tan congruente con la sintaxis anterior:

```
(ejemplo)
```

como lo sería esta entrada:

```
(ejemplo 1)
```

Los caracteres menor que y mayor que, < >, indican que se reemplaza un valor del tipo especificado por el contenido que se encuentra dentro de ellos. Por ejemplo, lo siguiente dentro de una descripción de sintaxis:

```
<entero>
```

indica que debe hacerse una sustitución con un valor entero real. Siguiendo los ejemplos anteriores, la descripción de la sintaxis

```
(ejemplo <entero>)
```

podría reemplazarse con:

```
(ejemplo 1)
```

o

```
(ejemplo 5)
```

o

```
(ejemplo -20)
```

o muchas otras entradas que tengan los caracteres "(ejemplo", seguidos por un entero, seguidos por el carácter ")". Es importante observar que los espacios que se muestran en la descripción de sintaxis también deben incluirse en la entrada.

Otra notación se indica con un asterisco, *, siguiendo una descripción; esto indica que la descripción puede reemplazarse con *cero* o más casos del valor especificado. Por ejemplo, la descripción de sintaxis:

```
<entero>*
```

puede reemplazarse con:

```
1
```

o

```
12
```

o

```
1 2 3
```

o con cualquier número de enteros, o con nada.

Una descripción seguida de un +, indica que *uno* o más de los valores especificados por la descripción deben utilizarse en lugar de la descripción de sintaxis. Observe que para esta notación, la descripción de sintaxis:

```
<entero>+
```

es equivalente a la descripción de sintaxis

```
<entero> <entero>*
```

Una barra vertical, |, indica la elección entre uno o más de los elementos separados por las barras. Por ejemplo, la descripción de sintaxis

```
todos | ninguno | alguno
```

puede reemplazarse con:

```
todos
```

o

```
ninguno
```

o

```
alguno
```

7.4 CAMPOS

A medida que se construye una base de conocimiento para ejecutar los comandos y cargar los programas, CLIPS debe leer la entrada desde el teclado o los archivos; conforme lee los caracteres los agrupa en **tokens** o señales. Los tokens representan grupos de caracteres que tienen un significado especial para CLIPS. Algunos tokens, como los paréntesis izquierdo y derecho, constan de un solo carácter.

El grupo de tokens denominado **campos** es de particular importancia; hay siete tipos de campos, también llamados tipos de datos primitivos de CLIPS: **flotantes, enteros, símbolos, cadenas, dirección externa, nombre del caso y dirección de caso.**

A los dos primeros tipos de campo, flotantes y enteros, se les llama ***campos numéricos*** o simplemente ***números.*** Un campo numérico está formado por tres partes: el signo, el valor y el exponente. El signo y el exponente son opcionales, el signo es + o – y el exponente se forma con la letra e o E seguida del + o – opcional y de 1. El valor, que contiene uno o más dígitos con un solo punto decimal opcional. Cualquier número formado por un signo opcional y seguido únicamente por dígitos se almacena como un **entero,** todos los demás números se almacenan como **flotantes.**

Los siguientes son ejemplos de flotantes válidos para CLIPS:

```
1.5
1.0
0.7
9e+1
3.5e10
```

Los siguientes son ejemplos de enteros válidos para CLIPS:

```
1
+3
-1
65
```

Un **símbolo** es un campo que inicia con un carácter imprimible ASCII y es seguido por cero o más caracteres y termina con un símbolo **delimitador.** Los delimitadores son cualquier carácter ASCII no imprimible (incluso espacios, tabulaciones, retornos de carro y avances de línea), como los caracteres " (comillas dobles), ((abrir paréntesis),) (cerrar paréntesis), ; (punto y coma), & (signo de unión en inglés), | (barra vertical), ~ (tilde) y < (menor que). Los símbolos no pueden incluir delimitadores dentro de sí (con excepción del carácter <, que puede ser el primero de un símbolo). Tampoco el carácter ? y la secuencia $? pueden colocarse al principio de un símbolo, porque se utilizan para denotar variables (véase la sección 8.2). Además, a una secuencia de caracteres que no sigue exactamente el formato de campo numérico se le trata como un símbolo.

Los siguientes son ejemplos de símbolos válidos:

```
Estación_espacial
Febrero
Fuego
Activar_sistema_riego
Avisar_a_bomberos
Cerrar_empalme_eléctrico-387
!?#$^*
345B
346-95-6156
```

Observe cómo se utiliza el guión bajo y el guión normal para unir símbolos y hacer que formen un solo campo.

CLIPS preservará las letras mayúsculas y minúsculas que se encuentran en los tokens, por lo que se dice que es ***sensible a mayúsculas y minúsculas.*** Por ejemplo, CLIPS considera diferentes a los siguientes símbolos:

```
sensible-a-mayúsculas
Sensible-a-mayúsculas
SENSIBLE-A-MAYÚSCULAS
```

El siguiente tipo de campo es una **cadena**, ésta debe comenzar y terminar con comillas dobles, que son parte del campo. Puede haber cero o más caracteres de cualquier tipo entre las comillas, incluyendo caracteres que CLIPS suele usar como delimitadores. Los siguientes son ejemplos de cadenas válidas de CLIPS:

```
"Activar el sistema de riego."
"Cerrar el empalme eléctrico 387."
"!?#$^"
"Francisco I. Madero
```

Los espacios suelen actuar como delimitadores en CLIPS para separar campos (como los símbolos) y otros tokens, pero descartando el uso de espacios adicionales entre ellos. Sin embargo, sí se preservan los espacios incluidos como parte de una cadena, por ejemplo, CLIPS considera que las siguientes cuatro cadenas son distintas:

```
"espacios"
"espacios "
" espacios"
" espacios "
```

Si se eliminaran las comillas dobles, CLIPS consideraría que cada una de las líneas contiene la misma palabra porque se ignorarían los espacios utilizados como delimitadores.

Como las comillas dobles se utilizan para delimitar cadenas, no es posible colocar directamente una comilla doble dentro de una cadena; por ejemplo, CLIPS interpretaría la línea

```
""tres-tokens""
```

como los siguientes tres tokens, con las comillas dobles actuando como delimitadores:

```
""
tres-tokens
""
```

Dentro de una cadena, pueden utilizarse comillas dobles utilizando el operador de diagonal invertida, \; por ejemplo, CLIPS interpretaría la línea:

```
"\"un-token\""
```

como un campo de cadena:

```
""un-token""
```

Sólo se crea un campo porque el carácter de diagonal invertida evita que las dobles comillas siguientes actúen como un delimitador. El propio carácter de diagonal invertida puede colocarse dentro de una cadena al utilizar dos diagonales invertidas en sucesión. Por ejemplo, CLIPS interpretará la línea:

```
"\\un-token\\ "
```


como el campo de cadena:

```
"\un-token\"
```

Los tres siguientes tipos de campos son las **direcciones externas**, las **direcciones de casos** y los **nombres de casos,** y son de interés limitado en la exploración de las capacidades de la programación basada en reglas de CLIPS. Las direcciones externas representan la dirección de una estructura de datos externa devuelta por una **función definida por el usuario** (una función escrita en un lenguaje como C o Ada, vinculada con CLIPS para agregar funciones adicionales). Como no es posible especificar el valor de una dirección externa con una secuencia de caracteres que forman un token, y como la versión básica y sin modificar de CLIPS no contiene funciones que devuelven direcciones externas, no es posible crear este tipo de campo sin modificar la versión básica y de CLIPS. Los nombres de caso y las direcciones de caso son campos usados junto con COOL, los nombres de caso son símbolos incluidos dentro de corchetes como [bomba-1] mientras que las Direcciones de caso, como las externas, sólo pueden obtenerse como el valor devuelto a partir de una función.

A una serie de cero o más campos agrupados se le llama **valor de varios campos**. Por lo general, éstos se crean al llamar a una función (como se muestra en los capítulos siguientes) o cuando se especifican valores iniciales para los campos. Cuando se imprime, un valor de varios campos se incluye entre paréntesis; por ejemplo, el campo múltiple de longitud cero se imprimiría como sigue:

```
()
```

y el campo múltiple que contiene los símbolos *esto* y *eso* se imprimiría como:

```
(esto eso)
```

7.5 ENTRADA Y SALIDA EN CLIPS

Puede entrarse a CLIPS activando el comando de ejecución apropiado para la máquina en que se ha instalado. Debe aparecer el de CLIPS, como se muestra a continuación:

```
CLIPS>
```

En ese momento, pueden introducirse comandos directamente a CLIPS; a este modo se le llama ***nivel superior***.

El modo normal de abandonar CLIPS es con el comando **exit**, cuya sintaxis es:

```
(exit)
```

Observe que la palabra *salir* se incluye dentro del paréntesis, muchos lenguajes basados en reglas tienen sus orígenes en LISP, que utiliza paréntesis como delimitadores, y como CLIPS está basado en un lenguaje que se desarrolló originalmente utilizando máquinas LISP, retiene esos delimitadores. El símbolo *exit* sin paréntesis tiene un significado muy diferente al símbolo *exit* con paréntesis, los paréntesis indican que se trata de un comando ejecutable y no del símbolo *exit*. Más adelante se verá que los paréntesis son importantes delimitadores de comandos.

Por ahora, es importante recordar que cada comando de CLIPS debe tener un número igual de paréntesis izquierdos y derechos y que deben estar adecuadamente balanceados.

El paso final en la ejecución de un comando de CLIPS, luego de que se ha introducido con paréntesis adecuadamente balanceados es oprimir Enter; también es posible oprimir esta tecla antes o después de que se ha introducido cualquier token. Por ejemplo, si oprime Enter después de introducir los caracteres "ex", pero antes de introducir los caracteres "it", se crearían dos tokens: uno para el símbolo *ex* y otro para el símbolo *it*.

En la siguiente secuencia de comandos se muestra una sesión de ejemplo para entrar a CLIPS, la evaluación de un valor de campo constante y de una función, y luego la salida utilizando el comando exit. Este ejemplo corresponde a una PC de IBM con MS-DOS en la que el CLIPS ejecutable se encuentra almacenado en un disco en la unidad A, siendo ésta la unidad en uso, aquí se supone que el nombre del CLIPS ejecutable es CLIPS, la salida desplegada por MS-DOS o CLIPS se muestra en tipo normal, toda la entrada que debe escribir el usuario se muestra en negritas y la tecla Enter se indica con el carácter ↵. Como CLIPS es sensible a ellas, es importante escribir las mayúsculas y las minúsculas tal como aparecen:

```
A>CLIPS↵
          CLIPS (V6. 05 09/01/97)
CLIPS> EXIT↵
EXIT
CLIPS> (+ 3 4)↵
7
CLIPS> (EXIT)↵
A>
```

Mientras que al nivel superior, CLIPS aceptará entradas del usuario y tratará de evaluarlas para determinar la respuesta apropiada, los campos introducidos por sí mismos se consideran congruentes y el resultado de evaluar una constante es la propia constante. De modo que cuando se escribe el símbolo *exit* y va seguido por un Enter, CLIPS evalúa esta entrada y se despliega el símbolo *exit* como resultado. A un símbolo entre paréntesis se le considera un comando o una llamada a una función, de este modo la entrada (+ 3 4) es una llamada de la función (+) que realiza la suma en tanto que sus argumentos son los valores 3 y 4 y el valor de devolución de esta llamada a función es 7 (los valores de devolución se analizan con mucho mayor detalle en el capítulo 8); por último, la entrada (exit) invoca al comando *exit*, que sale de CLIPS. Debe considerarse que los términos *comando* y *función* son intercambiables. En todo este texto *función* se utilizará para indicar que se devuelve un valor y *comando* indicará que no se devolverá ningún valor o que la acción suele realizarse en el indicador de comandos de nivel superior.

7.6 HECHOS

Con el fin de resolver problemas, un programa de CLIPS debe tener datos e información con la que pueda razonar, a cada "fragmento" de información se le llama ***hecho***. Los hechos constan de un **nombre de relación** (un campo simbólico) seguido por cero o más **ranuras** (también campos simbólicos) y sus valores asociados. El siguiente es un ejemplo de un hecho:


```
(persona (nombre "Francisco I. Madero")
         (edad 23)
         (color-ojos azul)
         (color-pelo negro))
```

Todo el hecho, además de cada ranura, está delimitado por un paréntesis izquierdo de apertura y uno derecho de cierre. El símbolo *persona* es el nombre de relación del hecho y éste contiene cuatro ranuras. *nombre*, *edad*, *color-ojos*, *color-pelo*. El valor de la ranura *nombre* es "Francisco I. Madero", el valor de la ranura *edad* es 23, el valor de la ranura *color-ojos* es *azul* y el valor de la ranura *color-pelo* es *negro*. Observe que el orden en que se especifican las ranuras carece de relevancia pues CLIPS trata al hecho:

```
(persona (color-pelo negro)
         (nombre "Juan Q. Ramos)
         (color-ojos azul)
         (edad 23))
```

de manera idéntica al primero.

La construcción plantillas de definición

Antes de que puedan crearse los hechos, CLIPS debe tener información sobre la lista de ranuras válidas para un nombre de relación determinado. Los grupos de hechos que comparten el mismo nombre de relación y contienen información común pueden describirse utilizando la construcción **plantillas de definición** o *deftemplates*. Las **construcciones** integran el núcleo de un programa de CLIPS al agregar el conocimiento del programador al ambiente de CLIPS y son diferentes sus funciones y comandos. La construcción plantillas de definición es análoga a una estructura de registro en un lenguaje como Pascal. El formato general de una plantilla de definición es:

```
(deftemplate<nombre-relación> [<comentario-opcional>]
  <definición-ranura>*)
```

La descripción sintáctica <definición-ranura> se define como:

```
(ranura <nombre-ranura>) | (ranura múltiple <nombre-ranura>)
```

Utilizando esta sintaxis, el hecho *persona* puede describirse con la siguiente deftemplate:

```
(deftemplate persona "Una deftemplate de ejemplo"
   (slot nombre)
   (slot edad)
   (slot color-ojos)
   (slot color-pelo))
```

Ranuras de varios campos

Las ranuras de un hecho que se ha especificado con la palabra clave *slot* en sus deftemplates correspondientes sólo puede contener un valor (se les denomina *ranuras de un solo campo*). A menudo es deseable colocar cero o más campos en una ranura determinada. A las ranuras de un hecho que se ha especificado con la palabra clave **multislot** en sus plantillas de definición correspondientes se les permite contener cero o más valores (a los que se denomina *ranuras*

con varios campos), por ejemplo, la ranura *nombre* de la plantilla de definición *persona* almacena el nombre de la persona como un solo valor de cadena. Si la ranura *nombre* se definiera utilizando la palabra clave *multislot* en lugar de *slot*, entonces podría almacenarse en él cualquier número de campos. Por lo tanto, el hecho

```
(persona (nombre Francisco I. Madero)
         (edad 23)
         (color-ojos azul)
         (color-pelo negro))
```

es ilegal si *nombre* es una ranura de un solo campo, pero legal si *nombre* es con ranura de varios campos. Una plantilla de definición puede tener cualquier combinación de ranuras de un solo campo y de varios.

Hechos ordenados

A los hechos con un nombre de relación que tienen una plantilla de definición correspondiente se les llama ***hechos de plantilla de definición***. A los hechos con un nombre de relación que no la tiene se les llama ***hechos ordenados***; éstos tienen implícita una sola ranura de varios campos que se utiliza para almacenar todos los valores que se encuentran después del nombre de relación. En realidad, cada vez que CLIPS encuentra un hecho ordenado crea automáticamente una **plantilla de definición implícita** para ese hecho (en contraposición con una **plantilla de definición explícita**, creada utilizando la construcción plantilla de definición). Como un hecho ordenado sólo tiene una ranura, no se requiere el nombre de ésta para definirlo, por ejemplo, una lista de números podría estar representada con el siguiente hecho:

```
(número-lista 7 9 3 4 20)
```

En esencia, esto es equivalente a definir la siguiente plantilla de definición:

```
(deftemplate número-lista (valores de varias ranuras))
```

y luego se define el hecho como sigue:

```
(número-lista (valores 7 9 3 4 20)
```

Por lo general, los hechos de plantilla de definición deben usarse cada vez que sea posible porque los nombres de ranura hacen que los hechos sean más legibles y que resulte más fácil trabajar con ellos. Hay dos casos donde los hechos ordenados resultan útiles: primero, los hechos que sólo incluyen un nombre de relación son útiles como marcas y tienen un aspecto idéntico, sin importar si se ha definido una plantilla de definición. Por ejemplo, el hecho ordenado:

```
(todas-las-órdenes-procesadas)
```

puede utilizarse como una marca para indicar cuando se han procesado todas las órdenes.

En segundo lugar, para hechos que contienen una sola ranura, el nombre de ésta suele ser sinónimo del nombre de relación. Por ejemplo, los hechos

```
(hora 8:45)
(grupos-alimenticios carne lácteos pan
                     frutas-y-vegetales)
```

tienen tanto significado como:


```
(hora (valor 8:45))
(grupos-alimenticios (valores carne lacteos pan
                              frutas-y-vegetales))
```

En la figura 7.1 se describe gráficamente la relación entre los términos introducidos en esta sección.

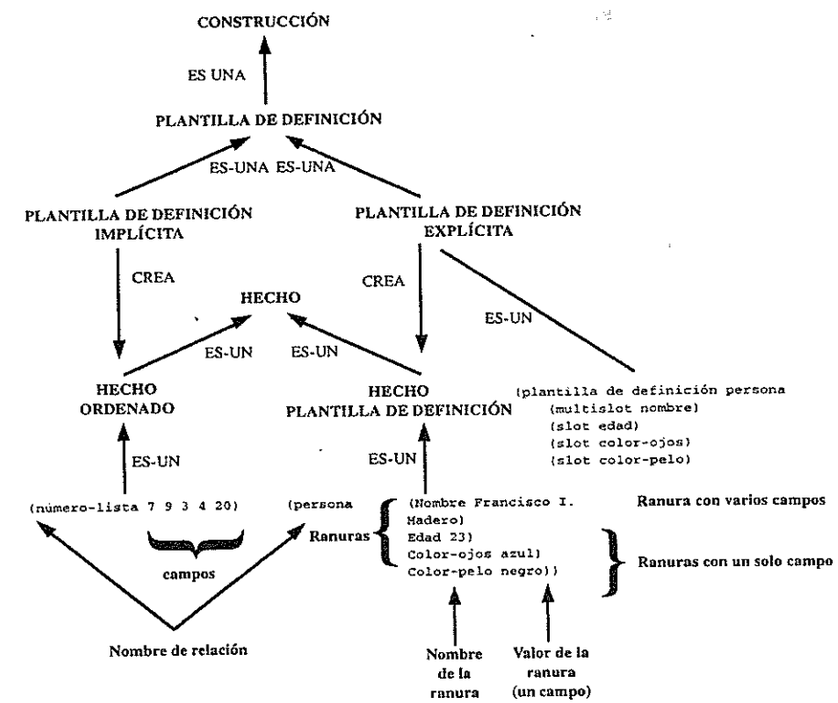


Figura 7.1 Panorama general de una plantilla de definición

7.7 ADICIÓN Y ELIMINACIÓN DE HECHOS

El grupo de todos los hechos conocidos por CLIPS se almacena en la lista de hechos. En ésta es posible agregar y eliminar hechos que representan información. Es posible agregar nuevos hechos a la lista usando el comando **assert**, cuya sintaxis es:

```
(assert <hecho>+)
```

Por ejemplo, utilizando la plantilla de definición *persona* para describir algunas personas como hechos, la información sobre Francisco I. Madero puede agregarse a la lista de hechos utilizando los siguientes comandos:

```
CLIPS>
(deftemplate persona
   (slot nombre)
   (slot edad)
   (slot color-ojos)
   (slot color-pelo))
CLIPS>
(assert (persona (nombre "Francisco I. Madero")
                 (edad 23)
                 (color-ojos azul)
                 (color-pelo negro))
<Fact-0>
CLIPS>
```

Observe que el comando assert devuelve un valor, <Fact-0>. El uso de los valores devueltos se analiza en el capítulo 8.

El comando **facts** puede utilizarse para desplegar los hechos que se encuentran en la lista y su sintaxis básica es:

```
(facts)
```

Por ejemplo:

```
CLIPS> (facts)
f-0    (persona  (nombre "Francisco I. Madero")
                 (edad 23)
                 (color-ojos azul)
                 (color-pelo negro))↵
For a total of 1 fact
CLIPS>
```

El término "f-0" es el **identificador de hechos** que CLIPS le asigna. A cada hecho insertado en la lista de hechos se le asigna un identificador único, empezando con la letra f y seguido por un entero llamado ***índice de hechos***. El índice de hechos 0 también se desplegó en el valor de devolución <Fact-0>, del comando *assert* previo. Observe que, como se muestra en este libro, se han agregado espacios en la salida del comando *facts*, entre las ranuras *(edad 23)* y *(color-ojos azul)*, para mejorar la legibilidad. Normalmente, CLIPS sólo pondría un espacio entre las ranuras y ajustaría las líneas de la salida para que pasaran de la columna del extremo izquierdo a la del extremo derecho. En algunos de los ejemplos de CLIPS que se encontrarán en todo este libro se añadirán espacios en la salida para mejorar la legibilidad, siempre y cuando la adición de espacios no cause confusión.

Por lo general, CLIPS no acepta entradas duplicadas de un hecho (aunque este comportamiento puede cambiarse, como se muestra en la sección 12.2), por lo que no será posible colocar un segundo hecho "Francisco I. Madero" con idénticos valores de ranura en la lista de hechos. Por supuesto, es fácil agregar a la lista otros hechos que no dupliquen los ya existentes, por ejemplo:

63-76.

```
CLIPS>
(assert (persona (nombre "Francisca I. Madero")
                 (edad 36)
                 (color-ojos verde)
                 (color-pelo rojo))↵
<Fact-1>
CLIPS> (facts) ↵
f-0      (persona (nombre "Francisco I. Madero")
                  (edad 23)
                  (color-ojos azul)
                  (color-pelo negro))↵
f-1      (persona (nombre "Francisca I. Madero")
                  (edad 36)
                  (color-ojos verde)
                  (color-pelo rojo))↵
For a total of 2 facts
CLIPS>
```

Como lo indica la sintaxis del comando *assert*, puede afirmarse más de un hecho utilizándolo una sola vez, por ejemplo el comando:

```
(assert (persona (nombre "Francisco I. Madero")
                 (edad 23)
                 (color-ojos azul)
                 (color-pelo negro))↵
        (persona (nombre "Francisca I. Madero")
                 (edad 36)
                 (color-ojos verde)
                 (color-pelo rojo)))
```

afirmará dos hechos en la lista.

Es importante percatarse de que los identificadores de la lista de hechos no necesariamente son secuenciales. Al igual que es posible agregar hechos a la lista, también hay una manera de eliminarlos; cuando se eliminan de la lista de hechos, los identificadores del hecho eliminado desaparecerán de la lista, de modo que los identificadores de hechos tal vez no sean estrictamente secuenciales mientras se ejecuta un programa CLIPS.

Debido a la gran cantidad de hechos que pueden estar incluidos en ella, a menudo es útil tener la posibilidad de ver sólo una parte de la lista de hechos. Esto puede lograrse utilizando argumentos adicionales para el comando *facts*, cuya sintaxis completa es:

```
(facts [<inicio> [<final> [<máximo>]]])
```

donde <inicio>, <final> y <máximo> son enteros positivos. Observe que la sintaxis del comando *facts* permite de cero a tres argumentos, que cuando no se especifican se muestran todos los hechos. Si se especifica el argumento <inicio>, se muestran todos los hechos con índices de hecho mayores o iguales a <inicio>. Si se especifican <inicio> y <final>, se muestran todos los hechos con índices de hecho mayores o iguales a <inicio> y menores o iguales a <final> y por último, si se especifica <máximo> junto con <inicio> y <final>, solamente se muestran los hechos de <máximo>.

Al igual que es posible agregar hechos a la lista, también se pueden eliminar. A la eliminación de hechos de la lista se le llama ***retractación*** y se realiza con el comando **retract**, su sintaxis:

```
(retract <hecho-índice>+1)
```

El índice de uno o más hechos que habrán de retirarse se incluye como los argumentos del comando *retract*. Por ejemplo, Francisco I. Madero puede eliminarse de la lista de hechos utilizando:

```
(retract 0)
```

De manera similar, el comando

```
(retract 1)
```

retirará el hecho Francisca I. Madero.

Tratar de retirar un hecho inexistente produciría el siguiente mensaje de error donde [PRNTUTIL 1] es una clave para encontrar el mensaje de error en el *Manual de referencia de CLIPS*:

```
(PRNTUTIL1) Unable to find fact <identificador-hecho>.
```

por ejemplo:

```
CLIPS> (retrac 1)↵
CLIPS> (retrac 1)↵
[PRNTUTIL1] Unable to find fact f-1.
CLIPS>
```

Puede utilizarse un solo comando *retract* para retirar varios hechos a la vez. Por ejemplo, el comando:

```
(retract 0 1)
```

retraerá los hechos f-0 y f-1.

7.8 MODIFICACIÓN Y DUPLICACIÓN DE HECHOS

Los valores de ranura de hechos de plantilla de definición pueden modificarse utilizando el comando **modify**, cuya sintaxis es:

```
(modify <índice-hecho> <modificador-ranura>+)
```

donde <modificador-ranura> es

```
(<slot-nombre-ranura> <slot-valor-ranura>)
```

Por ejemplo, si Francisco I. Madero sólo tiene un cumpleaños, se podría modificar su edad de 23 a 24 utilizando el comando modify:


```
CLIPS>  (modify 0 (edad 24))↵
<Fact-2>
CLIPS>  (facts)↵
f-2      (persona (nombre "Francisco I. Madero")
                  (edad 24)
                  (color-ojos azul)
                  (color-pelo negro))↵
For a total of 1 fact.
CLIPS>
```

El comando modify funciona retirando el hecho original y luego afirmando uno nuevo con los valores específicos de ranura modificados. Debido a esto, para un hecho modificado se genera un nuevo índice de hecho.

El comando **duplicate** funciona de la misma manera, sólo que no retira el hecho original. Por tanto, si se encuentra al hermano gemelo perdido de Francisco, José, éste puede agregarse a la lista de hechos al duplicar el hecho de Francisco y cambiar la ranura de nombre:

```
CLIPS> (duplicate 2 (nombre "José A. Madero))↵
<Fact-3>
CLIPS> (facts)↵
f-2    (persona   (nombre "Francisco I. Madero")
                  (edad 24)
                  (color-ojos azul)
                  (color-pelo negro))
f-3    (persona   (nombre "José A. Madero")
                  (edad 24)
                  (color-ojos azul)
                  (color-pelo negro)
For a total of 2 facts
CLIPS>
```

Los comando modify y duplicate no pueden utilizarse con hechos ordenados.

7.9 EL COMANDO WATCH

El comando **watch** es útil para depurar programas. En esta sección se analizará el efecto de la observación de hechos; los elementos de observación restantes se comentarán en este capítulo y en posteriores. La sintaxis de este comando es:

```
(watch <elemento-observado>)
```

donde <elemento-observado> es uno de los símbolos *hechos, reglas, actuaciones, estadísticas recopiladas, enfoque* o *todo*.

Estos elementos pueden observarse en cualquier combinación para proporcionar la cantidad apropiada de información para la depuración. El comando *watch* puede utilizarse más de una vez para observar más de una función de la ejecución de CLIPS, con la palabra *all* puede utilizarse para habilitar todas sus funciones. Como opción predeterminada, cuando CLIPS se inicia por primera vez, se observan las recopilaciones.

Si se están observando los hechos, CLIPS imprimirá automáticamente un mensaje que indica que se ha hecho una actualización de la lista de hechos cada vez que alguno se afirma

o retira. El siguiente diálogo de comandos ilustra el uso de este comando de depuración:

```
CLIPS> (facts 3 3)↵
f-3      (persona  (nombre "José A. Madero")
                   (edad 24)
                   (color-ojos azul)
                   (color-pelo negro))
For a total of 1 fact.
CLIPS> (watch facts)↵
CLIPS> (modify 3 (edad 25))↵
<== f-3      (persona  (nombre "José A. Madero")
                       (edad 24)
                       (color-ojos azul)
                       (color-pelo negro)
==> f-4      (persona  (nombre "José A. Madero")
                       (edad 25)
                       (color-ojos azul)
                       (color-pelo negro)
<Fact-4>
CLIPS>
```

La secuencia de caracteres <== indica que se está retirando el hecho previo, y ==> indica que se está afirmando el nuevo.

Los efectos de un comando *watch* pueden desactivarse utilizando el comando correspondiente **unwatch,** cuya sintaxis es:

```
(unwatch <elemento-observado>)
```

7.10 LA CONSTRUCCIÓN DEFFACTS

A menudo es conveniente tener la capacidad de afirmar automáticamente un conjunto de hechos en lugar de escribir las mismas afirmaciones desde el nivel superior, sobre todo en hechos que se saben verdaderos antes de ejecutar un programa (es decir, el conocimiento inicial). La ejecución de casos de prueba para depurar un programa es otro ejemplo en el que es útil afirmar automáticamente un grupo de hechos. Los grupos de hechos que representan el conocimiento inicial pueden definirse utilizando la construcción **deffacts**. Por ejemplo, la siguiente instrucción de deffacts proporciona la información inicial sobre algunas personas ya encontradas:

```
(deffacts persona "Algunas personas que conocemos"
   (persona (nombre "Francisco I. Madero") (edad 24)
            (color-ojos azul) (color-pelo negro))
   (persona (nombre "José A. Madero") (edad 24)
            (color-ojos azul) (color-pelo negro))
   (persona (nombre) "Francisca I. Madero) (edad 36)
            (color-ojos verde) (color-pelo rojo)))
```

El formato general de deffacts es


```
(deffacts  <nombre de deffacts> [<comentario opcional>]
   <hechos>*)
```

Después de la palabra deffacts, se encuentra el nombre necesario para esta construcción y es posible utilizar cualquier símbolo válido como tal. En este caso, el nombre elegido fue *persona;* después del nombre sigue un comentario adicional entre comillas que al igual que el comentario opcional de una regla, se retendrá con la construcción de deffacts después de que CLIPS lo cargue. Después del nombre o comentario siguen los hechos que habrán de afirmarse en la lista de hechos mediante esta instrucción de deffacts.

Los hechos de una instrucción de deffacts se afirman utilizando el comando **reset** de CLIPS, que elimina todos los hechos de la lista de hechos y luego afirma los hechos de la instrucción de deffacts existente. La sintaxis del comando *reset* es:

```
(reset)
```

Suponiendo que se ha introducido el deffacts *persona* (después del deftemplate *persona*), el siguiente diálogo muestra la manera en que el comando *reset* agrega los hechos a la lista correspondiente:

```
CLIPS> (unwatch facts) ↵
CLIPS> (reset) ↵
CLIPS> (facts) ↵
f-0      (hecho-inicial)
f-1      (persona  (nombre "Francisco I. Madero")
                   (edad 24)
                   (color-ojos azul)
                   (color-pelo negro)
f-2      (persona  (nombre "José A. Madero")
                   (edad 24)
                   (color-ojos azul)
                   (color-pelo negro)
f-3      (persona  (nombre "Francisca I. Madero")
                   (edad 36)
                   (color-ojos verde)
                   (color-pelo rojo)
For a total of 4 facts.
CLIPS>
```

La salida muestra los hechos de la instrucción deffacts y un nuevo hecho generado por el comando reset, llamado ***hecho-inicial***. En el arranque, CLIPS define automáticamente las siguientes dos construcciones:

```
(deftemplate hecho-inicial)
(deffacts hecho-inicial
   (hecho-inicial))
```

Por tanto, aunque no se haya definido ninguna instrucción de deffacts, un reset afirmará el hecho (hecho-inicial). El identificador de hecho de initial-fact siempre es f-0. La utilidad de (hecho-inicial) radica en iniciar la ejecución de un programa (como se analizará en la siguiente sección).

7.11 LOS COMPONENTES DE UNA REGLA

Con el fin de realizar trabajo útil, un sistema experto debe tener reglas y hechos. Ahora que se han analizado las afirmaciones y retractación de los hechos, es posible ver cómo funcionan las reglas.

Las reglas pueden escribirse directamente en CLIPS o cargarse desde un archivo de reglas creado por un editor (la carga de construcciones desde un archivo se analiza en la sección 7.17). En todos los programas, excepto los más pequeños, tal vez se querrá utilizar uno de los editores integrados que forman parte de CLIPS que permiten redefinir construcciones de manera selectiva durante el desarrollo del programa, lo que puede ser muy útil. Por ejemplo, si se tiene un error tipográfico en una construcción introducida en el indicador de comandos de nivel superior, se tiene que volver a escribir toda la construcción, pero si se introduce primero en el editor, puede corregirse el error y redefinir la construcción con unos cuantos tecleos en él. En principio, los ejemplos mostrados serán reglas introducidas directamente en CLIPS desde el nivel superior.

Por ejemplo, consideremos los tipos de hechos y reglas que pueden utilizarse para vigilar y responder a un rango de urgencias posibles como un incendio o una inundación. A continuación se muestra el pseudocódigo para una de las posibles reglas del sistema experto de vigilancia de una planta industrial:

```
SI la urgencia es un incendio
ENTONCES la respuesta es activar
     el sistema antiincendios
```

Antes de convertir el pseudocódigo en una regla, deben definirse las plantillas de definición para los tipos de hechos a los que hace referencia la regla. Una urgencia puede representarse mediante la siguiente deftemplate:

```
(deftemplate urgencia (slot tipo))
```

donde el campo *tipo* del hecho *urgencia* contendría símbolos como fuego, inundación e interrupción de energía. De igual manera, la respuesta puede representarse mediante la siguiente plantilla de definición

```
(deftemplate respuesta (slot acción))
```

donde el campo *acción* del hecho *respuesta* indica la respuesta que se emprenderá.

La regla expresada en sintaxis de CLIPS, se muestra al término del párrafo y puede introducirse al escribirla después del indicador de comandos de CLIPS, pero tendrá que introducir las deftemplates *urgencia* y *respuesta* antes de introducirla. Sin embargo, antes de introducir cualquiera de estas construcciones, escriba el comando (clear) seguido por un enter en la línea de comandos de nivel superior; esto eliminará las deftemplates y deffacts creadas en la sección anterior. El comando *clear* se explica ampliamente en la sección 7.13.


```
(defrule urgencia-incendio "Una regla de ejemplo"
   (urgencia (tipo incendio))
    =>
   (assert (respuesta
            (acción activar-sistema-antiincendios))))
```

Si la regla se introduce correctamente como se muestra, reaparecerá el indicador de comandos de CLIPS; de lo contrario, aparecerá un mensaje de error indicando muy probablemente que una palabra clave está mal escrita o que hay paréntesis mal colocados.

A continuación se muestra la misma regla, pero con comentarios añadidos para igualar sus partes. Los comentarios empiezan con punto y coma y continúan hasta un enter; en tanto CLIPS los ignora.

```
; Encabezado de regla
(defrule urgencia-incendio "Una regla de ejemplo"
   ; Patrones
   (urgencia  (tipo incendio))
   ; flecha ENTONCES
    =>
   ; Acciones
   (assert (respuesta
           (acción activar-sistema-antiincendios))))
```

El formato general de una regla es

```
(defrule <nombre de regla> [<comentario>]
   <patrones>*  ; lado izquierdo de la regla
   =>
   <acciones>*) ; lado derecho de la regla
```

La regla completa debe ponerse entre paréntesis, al igual que cada uno de sus patrones y acciones, que puedan ser múltiples. Los paréntesis que rodean a patrones y acciones deben equilibrarse de manera apropiada si están anidados. En la regla *incendio-urgencia* hay un patrón y una acción.

El encabezado de la regla consta de tres partes; la regla debe empezar con la palabra clave **defrule**, seguida por su nombre, que puede ser cualquier palabra de CLIPS válida. Si se introduce una regla con un nombre que ya se ha asignado a otra, entonces la nueva regla reemplazará a la anterior. En esta regla, el nombre es *incendio-urgencia*, a continuación viene una cadena de comentarios opcional; el comentario es "Una regla de ejemplo". Ésta suele utilizarse para describir el propósito de la regla o cualquier otra información que desee el programador. A diferencia de los comentarios que empiezan con punto y coma, el comentario que sigue al nombre de regla no se ignora y puede mostrarse junto con el resto de la regla (utilizando el comando *ppdefrule*, que se expone en la sección 7.13).

Después del encabezado de la regla siguen cero o más elementos condicionales (EC). El tipo más simple de ellos es un **patrón EC** o sólo **patrón**, cada uno consta de una o más restricciones destinadas a igualar los campos de un hecho de deftemplate. En la regla *incendio-urgencia*, el patrón es (urgencia (tipo de incendio)). La restricción para el campo *tipo* indica que esta regla sólo será satisfecha por hechos de *urgencia* que contenga el símbolo *incendio* en su campo *tipo*. CLIPS trata de comparar los patrones de reglas con los hechos de

la lista de hechos, si todos los patrones de una regla coinciden con los hechos, la regla se **activa** y se coloca en la **agenda**, que es la colección de reglas activadas, donde puede haber cero o más reglas.

Al símbolo => que sigue a los patrones de una regla se le llama **flecha**, se forma al escribir el signo de igual y luego el signo de mayor que y representa el principio de la parte ENTONCES de una regla SI...ENTONCES. A la parte de la regla que va antes de la flecha se le llama **lado izquierdo** y a la parte que va después de la flecha se le llama **lado derecho**.

Si una regla no tiene patrones, se agregará el patrón especial (hecho-inicial). Al igual que la deffacts *hecho-inicial* se define automáticamente, cualquier regla sin patrones en su lado izquierdo se activará cuando se realice un comando *reset*, ya que el hecho (hecho-inicial) se afirmará automáticamente. Por tanto, cualquier regla sin patrones al lado izquierdo se colocará en la agenda cuando se realice un reset.

La última parte de una regla es la lista de acciones que se ejecutará cuando se **dispare** la regla, aunque no es obligatorio que una regla tenga acciones. En el ejemplo, la acción es afirmar el hecho (respuesta (acción activar-sistema-antiincendios)). El término *disparar* significa que CLIPS ejecuta las acciones de la regla en la agenda. En condiciones normales, cuando hay reglas en la agenda cesa la ejecución de un programa; cuando hay varias reglas en ella, CLIPS determina automáticamente cuál es la regla que debe dispararse. CLIPS ordena las reglas de la agenda a partir de su prioridad y dispara la regla con mayor prioridad. La prioridad de una regla es un atributo entero llamado **prominencia**, que se analiza con mayor detalle en el capítulo 10.

7.12 LA AGENDA Y EJECUCIÓN

Es posible hacer que un programa de CLIPS se ejecute con el comando **run**, que tiene la siguiente sintaxis:

```
(run [<límite>])
```

donde el argumento opcional <límite> es el número máximo de reglas que habrán de dispararse, pero si no se incluye <límite> o es igual a –1, se dispararán todas las reglas hasta que no quede ninguna en la agenda. De lo contrario, la ejecución de las reglas cesará después de haber disparado un <límite> número de reglas.

Cuando se ejecuta un programa de CLIPS, se dispara la regla en la agenda con mayor prominencia, si sólo hay una regla en la agenda, ésa es la que se disparará. Como el elemento condicional de la regla *incendio-urgencia* se satisface con el hecho (urgencia (tipo incendio)), la regla *incendio-urgencia* debe dispararse cuando se ejecuta el programa.

Las reglas se activan cada vez que se igualan todos los patrones de regla mediante hechos. El proceso de igualación de patrones siempre se mantiene actualizado y se lleva a cabo independientemente de si los hechos se afirman antes o después de que se ha definido una regla.

Como las reglas requieren de los hechos para ejecutarse, el comando *reset* es el método clave para iniciar o reiniciar un sistema experto en CLIPS. Por lo general, los hechos asegurados con reset satisfacen los patrones de una o más reglas y colocan en agenda, las activaciones de estas reglas. Entonces al aplicar el comando *run* se inicia la ejecución del programa.

Contenido de la agenda

El comando **agenda** permite mostrar la lista de reglas en la agenda y su sintaxis es:

```
(agenda)
```

Si no hay activaciones en la agenda, reaparecerá el indicador de comandos de CLIPS después de emitir el comando *agenda*. Si la regla *incendio-urgencia* ha sido activada por el hecho (urgencia (tipo incendio)) con un índice de hecho de 1, un comando *agenda* produciría la siguiente salida:

```
CLIPS> (reset) ↵
CLIPS> (assert (urgencia (tipo incendio))) ↵
<Fact-1>
CLIPS> (agenda)
0       incendio-urgencia: f:1
For a total of 1 activation
CLIPS>
```

El 0 indica la prominencia de la regla en la agenda, le sigue el nombre de la regla y luego los identificadores de hecho que igualan los patrones de la misma. En este caso, sólo hay un identificador de hecho, f-1.

Reglas y refracción

Con la regla *incendio-urgencia* en la agenda, ahora el comando *run* provocará que se dispare la regla. El hecho (respuesta (acción activar-sistema-antiincendios)) se agregará a la lista de hechos como la acción de la regla, como se muestra en la siguiente salida:

```
CLIPS> (run) ↵
CLIPS> (facts) ↵
f-0     (hecho-inicial)
f-1     (urgencia (tipo incendio))
f-2     (respuesta
           (acción activar-sistema-antiincendios))
For a total of 3 facts.
CLIPS>
```

En este punto surge una pregunta interesante: ¿qué pasa si se vuelve a aplicar el comando *run*? Hay una regla y un hecho que la satisface, por lo que debe dispararse de nuevo, sin embargo, si se intenta de nuevo un comando *run* no producirá resultados. Al revisar la agenda se comprobará que no se dispararon reglas porque no había reglas en ella.

La regla no se volvió a disparar debido a la manera en que está diseñado CLIPS, en éste las reglas muestran una propiedad llamada **refracción**, lo que significa que no se disparan más de una vez para un conjunto específico de hechos. Sin la refracción, los sistemas expertos siempre caerían en ciclos triviales, es decir, en cuanto se disparara una regla, lo haría una y otra vez sobre el mismo hecho. En la realidad, el estímulo que provocó el fuego desaparecería tarde o temprano, ya sea porque el sistema antiincendios lo extingue o porque se dispara por sí solo. Sin

embargo, en el mundo de las computadoras, una vez que se introduce un hecho en la lista de hechos, permanece ahí hasta que se elimina de manera explícita.

Si es necesario, puede crearse una regla de modo que se dispare de nuevo al retraer el hecho (urgencia (tipo incendio)) y afirmarlo una vez más. En esencia, CLIPS recuerda los identificadores de hecho que desencadenaron una regla en el disparo y no activará nuevamente la regla con la misma combinación exacta de identificadores. Los conjuntos idénticos de identificadores de hechos deben ser iguales, uno a uno, a los índices de orden y de hechos. En los ejemplos del capítulo 8 se muestra cómo un solo hecho puede igualar un patrón en más de una manera. En este caso, es posible colocar en la agenda varias activaciones con el mismo conjunto de identificadores de hechos para una sola regla, y cada una correspondería a una coincidencia distinta.

Como alternativa, puede utilizarse el comando **refresh** para que la regla se vuelva a disparar, este comando vuelve a colocar en la agenda todas las activaciones que ya se han disparado para una regla (con la limitante de que los hechos que desencadenaron la activación aún están presentes en la lista de hechos). La sintaxis del comando *refresh* es:

```
(refresh <nombre-regla>)
```

y los siguientes comandos muestran la manera en que puede utilizarse el comando *refresh* para reactivar la regla *incendio-urgencia*:

```
CLIPS>  (agenda) ↵
CLIPS>  (refresh incendio-urgencia) ↵
CLIPS>  (agenda) ↵
0       incendio-urgencia: f-1
For a total of 1 activation.
CLIPS>
```

Observación de activaciones, reglas y estadísticas

Si se están observando las activaciones, CLIPS imprimirá automáticamente un mensaje cada vez que se agregue o se elimine una activación a la agenda. Al igual que con los hechos, la secuencia de caracteres <== indica que se está eliminando una activación de la agenda, y ==> que se está agregando. Después de la secuencia de caracteres inicial, la activación se imprime con el mismo formato utilizado por el comando *agenda*. La siguiente secuencia de comandos ilustra activaciones que se están observando:

```
CLIPS>  (reset) ↵
CLIPS>  (watch activaciones) ↵
CLIPS>  (assert (urgencia (tipo incendio))) ↵
==> Activación 0          incendio-urgencia: f-1
<Fact-1>
CLIPS>  (agenda)
0       incendio-urgencia: f-1
For a total of 1 activation.
CLIPS>  (retract 1) ↵
<== Activación 0          incendio-urgencia: f-1
CLIPS>  (agenda) ↵
CLIPS>
```


Si se están observando las reglas, CLIPS imprimirá un mensaje cada vez que se dispare una de ellas. La siguiente secuencia de comandos ilustra activaciones y reglas que se están observando:

```
CLIPS>  (reset) ↵
CLIPS>  (watch rules) ↵
CLIPS>  (assert (urgencia (tipo incendio))) ↵
==> Activación 0          incendio-urgencia: f-1
<Fact-1>
CLIPS>  (run) ↵
FIRE    1 incendio-urgencia: f-1
CLIPS>  (agenda) ↵
CLIPS>
```

El número que sigue al símbolo *FIRE* indica cuántas reglas se han disparado desde que se proporcionó el comando *run* por ejemplo, si se disparó otra regla después de la regla *urgencia-incendio*, sería precedida por "FIRE 2". Después de imprimir el orden de disparo, se imprime el nombre de la regla, seguido por el índice de hecho que igualó los patrones de la regla. Note que la observación de activaciones no hará que se muestre un mensaje cuando se dispara una regla (y, por tanto, se elimina de la agenda).

Si se están observando estadísticas, al terminar una ejecución CLIPS imprimirá mensajes informativos como el siguiente:

```
CLIPS>  (unwatch all) ↵
CLIPS>  (reset) ↵
CLIPS>  (watch statistics) ↵
CLIPS>  (assert (urgencia (tipo incendio))) ↵
<Fact-1>
CLIPS>  (run) ↵
1 rules disparada         Run time is 0-02 seconds
50.0 rules per second
3 mean number of facts (3 maximum)
1 mean number of instances (1 maximum)
1 mean number of activations (1 maximum)
CLIPS>  (unwatch statistics) ↵
CLIPS>
```

Se observa que se imprimen seis estadísticas después de un comando *run*; así mismo se muestra el número total de reglas disparadas, la cantidad de tiempo en segundos que se requirió para dispararlas y el promedio de reglas disparadas por segundo (la primera estadística dividida entre la segunda); además, para cada ciclo de ejecución, CLIPS mantiene estadísticas sobre el número de hechos, activaciones y casos. La media de hechos es la suma del número total de hechos en la lista de hechos después de disparar cada regla, dividida entre el número de reglas disparadas. El número entre paréntesis seguido por la palabra *maximum* indica el número más grande de hechos contenido en la lista de hechos para cualquier disparo de regla. De manera similar, los números medio y máximo de las estadísticas de activaciones indican el promedio de activaciones y el número más grande de activaciones en la agenda para cualquier disparo de regla. Los números medio y máximo de casos muestran la información asociada con COOL.

7.13 COMANDOS PARA EL MANEJO DE CONSTRUCCIONES

Despliegue de la lista de miembros de una construcción específica

El comando **list-defrules** se utiliza para mostrar la lista actual de reglas mantenidas por CLIPS. De manera similar, los comandos **list-deftemplates** y **list-deffacts**, pueden utilizarse para mostrar respectivamente la lista actual de deftemplates o deffacts. La sintaxis de esos comandos es:

```
(list-defrules)

(list-deftempletes)

(list deffacts)
```

Por ejemplo:

```
CLIPS> (list-defrules) ↵
incendio-urgencia
For a total of 1 rule.
CLIPS> (list-deftemplates) ↵
hecho-inicial
urgencia
respuesta
For a total of 3 deftemplates.
CLIPS> (list-deffacts) ↵
hecho-inicial
For a total of 1 deffacts.
CLIPS>
```

Despliegue de la representación de texto de un miembro específico de la construcción

Los comandos **ppdefrule** (pretty print defrule, defrule de impresión), **ppdeftemplate** (pretty print deftemplate, deftemplate de impresión) y **ppdeffacts** (pretty print deffacts, deffacts de impresión) se utilizan para mostrar las representaciones en texto de una defrule, una deftemplate y una deffacts, respectivamente. Las sintaxis de estos comandos son:

```
(ppdefrule <nombre-defrule>)

(ppdeftemplate <nombre-deftemplate>)

(ppdeffacts <nombre-deffacts>)
```

El único argumento de cada comando especifica el nombre de la defrule, deftemplate o deffacts que se mostrará. Cuando se despliega, CLIPS pone diferentes partes de las construcciones en diferentes líneas para mejorar la legibilidad. Por ejemplo:


```
CLIPS> (ppdefrule incendio-urgencia) ↵
(defrule MAIN: :incendio-urgencia "Una regla de ejemplo"
    (urgencia (tipo incendio))
  =>
    (assert  (respuesta
                (acción activar-sistema-antiincendios))))
CLIPS> (ppdeftemplate respuesta) ↵
(deftemplate MAIN:respuesta)
    (slot acción))
CLIPS> (ppdeffacts hecho-inicial) ↵
CLIPS> (deffacts hecho-principio) (hecho-principio)) ↵
CLIPS> (ppdeffacts hecho-principio) ↵
(deffacts MAIN: :hecho-principio
    (hecho-principio))
CLIPS>
```

El símbolo MAIN:: que precede a cada uno de los nombres de construcción indica el módulo en el que se han colocado las construcciones. Los módulos proporcionan un mecanismo para dividir una base de conocimiento y se analizan con mayor detalle en el capítulo 10. Observe que la deffacts *hecho-inicial* no tiene representación en texto (porque CLIPS lo crea automáticamente), pero la deffacts *hecho-principio* que se introdujo si la tiene.

Eliminación de un miembro específico de una construcción

Los comandos **undefrule**, **undeftemplate** y **undeffacts** se utilizan para eliminar una defrule, una deftemplate y una deffacts, respectivamente; sus sintaxis son:

```
(undefrule <nombre-defrule>)

(undeftemplate <nombre-deftemplate>)

(undeffacts <nombre-deffacts>)
```

El único argumento para cada comando especifica el nombre de la defrule, deftemplate o deffacts que habrá de eliminarse. Por ejemplo:

```
CLIPS> (undeffacts hecho-principio) ↵
CLIPS> (list-deffacts) ↵
hecho-inicial
For a total of 1 deffacts.
CLIPS> (undefrule incendio-urgencia) ↵
CLIPS> (list-defrules) ↵
CLIPS>
```

Observe que la deffacts *hecho-inicial* y la deftemplate *hecho-principio* pueden eliminarse de la misma forma que cualquier otra construcción definida por el usuario. Si se aplicara un comando *reset* en este momento, el hecho (hecho-inicial) no se agregaría a la lista de hechos.

Si se proporciona el símbolo * como un argumento para cualquiera de los comandos para eliminación de construcciones, entonces se eliminan todas las construcciones del tipo apro-

piado; por ejemplo, (undefrule *) eliminaría todas las construcciones de defrule. El símbolo * también puede utilizarse con el comando *retract* para eliminar todos los hechos.

Las construcciones que son utilizados como referencia por otras construcciones, sólo pueden eliminarse después de las segundas. Como lo muestra el siguiente diálogo, la deftemplate *hecho-inicial* no puede eliminarse hasta que se elimine la deffacts *hecho-inicial*, el hecho (hecho-inicial) y la defrule *ejemplo* (que utiliza el patrón *hecho-inicial* predeterminado).

```
CLIPS> (defrule ejemplo =>) ↵
CLIPS> (undeftemplate hecho-inicial) ↵
Unable to delete deftemplate initial-fact while
outstanding references to it still exist.
CLIPS> (undeffacts hecho-inicial) ↵
CLIPS> (undeftemplate hecho-inicial) ↵
Unable to delete deftemplate initial-fact while
outstanding references to it still exist.
CLIPS> (undefrule ejemplo) ↵
CLIPS> (undeftemplate hecho-inicial) ↵
Unable to delete deftemplate initial-fact while
outstanding references to it still exist.
CLIPS> (retract *) ↵
CLIPS> (undeftemplate hecho-inicial) ↵
CLIPS>
```

Eliminación de todas las construcciones del ambiente CLIPS

El comando **clear** se utiliza para borrar toda la información contenida en el entorno de CLIPS, elimina todas las construcciones contenidas actualmente y todos los hechos de la lista de hechos. La sintaxis del comando *clear* es:

```
(clear)
```

Después de *borrar* el entorno CLIPS, el comando clear agrega la deffacts *hecho-inicial* al ambiente CLIPS:

```
CLIPS> (list-deffacts) ↵
CLIPS> (list-deftemplates) ↵
urgencia
respuesta
hecho-principio
For a total of 3 deftemplates.
CLIPS> (clear) ↵
CLIPS> (list-deffacts) ↵
Hecho-inicial
For a total of 1 deffacts.
CLIPS> (list-deftemplates) ↵
Hecho-inicial
For a total of 1 deftemplate.
CLIPS>
```


7.14 EL COMANDO PRINTOUT

Además de afirmar hechos, en el lado derecho de las reglas, ese lado también puede utilizarse para imprimir información utilizando el comando **printout**, cuya sintaxis es:

```
(printout <nombre-lógico> <imprimir-elementos>*)
```

donde <nombre-lógico> indica el destino de la salida del comando *printout* e <imprimir-elementos>* son los cero o más elementos que habrán de imprimirse con este comando.

La siguiente regla demuestra el uso del comando *printout:*

```
(defrule incendio-urgencia
   (urgencia   (tipo incendio))
     =>
   (printout t "Activar el sistema antiincendios"
               crlf))
```

Es muy importante incluir la letra t después del comando *printout* porque este argumento indica el destino de la salida, también conocido como ***nombre lógico***. En este caso, el nombre lógico t indica a CLIPS que envíe la salida al **dispositivo de salida estándar** de la computadora, que suele ser la terminal, pero puede redefinirse para que sea cualquier cosa, como un módem o una impresora. El concepto de nombres lógicos se presenta por completo en la sección 8.12.

Los argumentos que van después del nombre lógico son los elementos que habrán de imprimirse mediante el comando *printout.* La cadena

```
"Activar el sistema antiincendios"
```

se imprimirá en la terminal sin las comillas. El comando *printout* da trato especial a la palabra **crlf.** Ésta obliga una reacción de enter, lo que mejora el aspecto de la salida al formarla en diferentes líneas.

7.15 USO DE VARIAS REGLAS

Comentar

Hasta ahora, sólo se ha mostrado el tipo más simple de programa que contiene únicamente una regla. Sin embargo, los sistemas expertos con una sola regla no son muy útiles. En realidad los sistemas prácticos pueden constar de cientos o miles de reglas. Además de la regla *incendio-urgencia*, el sistema experto que vigila la planta industrial puede incluir una regla para urgencias en la que ha ocurrido una inundación. El conjunto de reglas expandido ahora se parece a esto:

```
(defrule incendio-urgencia
   (urgencia (tipo incendio))
     =>
   (printout t "Activar el sistema antiincendios"
               crlf))
```

```
(defrule inundación-urgencia
   (urgencia (tipo inundación))
   =>
   (priuntout t "Apagar el equipo eléctrico"
              crfl))
```

Una vez que estas reglas se han introducido en CLIPS, la afirmación del hecho (urgencia (tipo incendio)) y luego la aplicación de un comando *run*, producirían la salida "Activar sistema contra incendios", en tanto que la afirmación del hecho (urgencia (tipo inundación)) y la emisión de un comando *run*, producirán la salida "Apagar el equipo eléctrico".

Captura de la realidad en una regla

Si los incendios y las inundaciones son las únicas urgencias que deben controlarse, las reglas anteriores serán suficientes. Sin embargo, el mundo real no es tan simple; no todos los incendios pueden extinguirse utilizando agua, tal vez algunos requieran extintores químicos ¿Qué debe hacerse si un incendio produce gases venenosos u ocurre una explosión? ¿Hay que notificar a un departamento de incendios externo además de los bomberos locales? ¿Es importante el piso del edificio donde se encuentra el fuego? La activación del sistemas contra incendios del segundo piso podría deteriorar con agua el primero y segundo pisos. La energía eléctrica para el equipo podría interrumpirse en ambos pisos, pero un incendio en el primero tal vez sólo requeriría que se interrumpiera la energía en él. Si el edificio está inundado, ¿hay puertas herméticas que pueden cerrarse para evitar daño? ¿Qué debe hacerse en caso de detectar un corte eléctrico en la planta? Si todas estas situaciones se incluyen como reglas, ¿ya se han cubierto todas las posibilidades?

Por desgracia, la respuesta es no. En la realidad, las cosas no siempre funcionan a la perfección y capturar todo el conocimiento pertinente en un sistema experto puede ser muy difícil. En el mejor de los casos, tal vez sea posible reconocer la mayor parte de las urgencias y proporcionar reglas que permitan al sistema experto reconocer cuándo no se trata de una urgencia.

Reglas con varios patrones

La mayor parte de la heurística real es demasiado complicada para expresarse con reglas de un solo patrón, por ejemplo, la activación del sistema contra incendios para cualquier tipo de incendio tal vez no sólo sea incorrecto, sino también peligroso. Los incendios que surgen a partir de combustibles ordinarios como el papel, la madera y la tela (incendios clase A), pueden extinguirse utilizando agua o extintores basados en agua. Los incendios que involucran líquidos flamables y combustibles, lubricantes y materiales similares (incendios clase B), deben extinguirse con un método diferente, como un extintor de dióxido de carbono. Es posible utilizar reglas con más de un patrón para expresar estas condiciones, por ejemplo:

```
(deftemplate sistema-extintor
   (slot tipo)
   (slot estado))
```



```
(defrule incendio-clase-A-urgencia
   (urgencia (tipo incendio-clase-A))
   (sistema-extintor (tipo aspersar-agua)
                     (estado apagado))
   =>
   (printout t "Activar contra incendios de agua" crlf))

   (defrule incendio-clase-B-urgencia)
   (urgencia (tipo incendio-clase-B))
   (sistema-extintor (tipo dióxido-carbono)
                     (estado apagado))
   =>
   (printout t "Usar extintor de dióxido de carbono"
               crlf))
```

Ambas reglas tienen dos patrones, el primero determina que existe una urgencia de incendio y si éste es de clase A o B; el segundo patrón determina si se ha habilitado el extintor apropiado. Podrían agregarse más reglas para apagar los extintores (por ejemplo, si ya se activó el sistema de aspersión y ocurre un incendio clase B, entonces se apaga; o si el fuego se ha extinguido, se apagan los extintores). Podrían utilizarse aún más reglas para determinar si un material específico constituye un incendio clase A o clase B.

Es posible colocar cualquier número de patrones en una regla; lo importante es observar que la regla sólo se coloca en la agenda si los hechos satisfacen a todos los patrones; a este tipo de limitación se le llama ***elemento condicional y***. Puesto que los patrones de todas las reglas están contenidos de manera implícita dentro de un elemento condicional *y*, la regla no se disparará si sólo se satisface uno de los patrones; deben estar presentes todos los hechos antes de que el lado izquierdo de una regla se satisfaga y la regla se coloque en la agenda.

7.16 EL COMANDO SET-BREAK

CLIPS tiene un comando de depuración llamado ***set-break*** que permite detener la ejecución antes de que se dispare cualquier regla de un grupo específico. A una regla que detiene la ejecución antes de dispararse se le llama ***punto de interrupción***. La sintaxis del comando set-break es:

```
(set-break <nombre-regla>)
```

donde <nombre-regla> es el nombre de la regla para la que se establece el punto de interrupción. Considerando como ejemplo las siguientes reglas (observe el uso de hechos ordenados, como se describe en la sección 7.6):

```
(defrule primera
     =>
      (assert (disparar segunda)))

(defrule segunda
   (disparar segunda)
     =>
      (assert (disparar tercera)))
```

```
(defrule tercera
   (disparar tercera)
     =>)
```

El siguiente diálogo de comandos muestra la ejecución de la reglas sin ningún punto de interrupción:

```
CLIPS> (watch rules) ↵
CLIPS> (reset) ↵
CLIPS> (run) ↵
FIRE    1 primera: f-0
FIRE    2 segunda: f-1
FIRE    3 tercera: f-2
CLIPS>
```

Las tres reglas se disparan en sucesión cuando se introduce el comando *run*. El siguiente diálogo de comandos demuestra el uso del comando *set-break* para detener la ejecución:

```
CLIPS> (set-break segunda) ↵
CLIPS> (set-break tercera) ↵
CLIPS> (reset) ↵
CLIPS> (run) ↵
FIRE    1 primera: f-0
Breaking on rule second
CLIPS> (run) ↵
FIRE    1 segunda: f-1
Breaking on rule third
CLIPS> (run) ↵
FIRE    1 tercera: f-2
CLIPS>
```

En este caso, la ejecución se detiene antes de permitir que se disparen las reglas *segunda* y *tercera*. Observe que el comando *run* debe disparar cuando menos una regla antes de que un punto de interrupción detenga la ejecución. Por ejemplo, después de que se ha detenido la ejecución de la *segunda* regla, no lo vuelve a hacer cuando se proporciona nuevamente el comando *run*.

El comando **show-breaks** puede utilizarse para enumerar todos los puntos de interrupción, su sintaxis es:

```
(show-breaks)
```

El comando **remove-break** puede usarse para eliminar los puntos de interrupción, su sintaxis es:

```
(remove-break [<nombre-regla>])
```

Si <nombre-regla> se proporciona como argumento, sólo se eliminará el punto de interrupción para esa regla; de lo contrario, se suprimirán todos los puntos de interrupción.

7.17 CARGA Y GUARDADO DE CONSTRUCCIONES

Carga de construcciones desde un archivo

El comando **load** permite cargar en CLIPS un archivo de construcciones hecho con un editor de texto. La sintaxis del comando *load* es:

```
(load <nombre-archivo>)
```

donde <nombre-archivo> es una cadena o símbolo que contiene el nombre del archivo que habrá de cargarse.

Suponiendo que las reglas y deftemplates de urgencia se almacenan en un archivo llamado incendio.clp en la unidad B de una PC, el siguiente comando cargaría las construcciones en CLIPS:

```
(load "B:incendio.clp")
```

Por supuesto, la especificación de un nombre de archivo dependerá de la máquina, de modo que este ejemplo sólo debe tomarse como guía. Uno de los problemas que pueden ocurrir durante la carga se debe al carácter de diagonal invertida utilizado en algunos sistemas operativos como separador de la ruta al directorio pues CLIPS la interpreta como un carácter de escape, es necesario usar dos diagonales invertidas para crear una sola en una cadena. Por ejemplo, normalmente un nombre de ruta puede escribirse como:

```
B:\usr\clips\incendio.clp
```

Para preservar la diagonal invertida, el nombre de ruta se escribiría como se muestra en el siguiente comando:

```
(load "B:\\usr\\clips\\incendio.clp")
```

Las construcciones no tienen que permanecer en un solo archivo. Pueden almacenarse en varios y cargarse empleando diversos comandos de carga. Si no ocurren errores cuando se carga un archivo, el comando de carga devolverá el símbolo *TRUE* (en el capítulo 8 se abordan detalladamente los valores de devolución), de lo contrario, devolverá el símbolo *FALSE*.

Observación de las compilaciones

Cuando se observan compilaciones (como opción predeterminada), se imprime un mensaje informativo que incluye el nombre de la construcción para cada una de las cargadas por medio del comando de carga. Por ejemplo, supongamos que CLIPS acaba de iniciarse y se introdujeron los siguientes comandos:

```
CLIPS> (load "incendio.clp") ↵
Defining deftemplate: urgencia
Defining deftemplate: respuesta
Defining defrule: incendio-urgencia +j
TRUE
CLIPS>
```

El mensaje indica que se cargaron dos deftemplates (*urgencia* y *respuesta*), seguidas por la regla *incendio-urgencia* y la cadena "+j" al final del mensaje "Defining defrule" es información de CLIPS sobre la estructura interna de las reglas compiladas: Esta información será útil para afinar un programa y se analiza en el capítulo 11, que trata sobre la eficiencia.

Si no se están observando las compilaciones, entonces CLIPS imprime un solo carácter para cada construcción cargada: * para defrules, % para deftemplates y $ para deffacts. Por ejemplo:

```
CLIPS> (clear) ↵
CLIPS> (unwatch compilaciones) ↵
CLIPS> (load incendio.clip) ↵
%%*
TRUE
CLIPS>
```

Guardado de construcciones en un archivo

CLIPS también proporciona lo opuesto al comando *load*; el comando **save** permite guardar en un archivo de disco el conjunto de construcciones almacenado en CLIPS y la sintaxis del comando *save* es:

```
(save <nombre-archivo>)
```

Por ejemplo, el siguiente comando guardará las construcciones de disparo en un archivo llamado *incendio.clp* en la unidad B:

```
(save "B:incendio.clp")
```

El comando *save* guardará todas las construcciones de CLIPS en el archivo especificado, pero no es posible guardar construcciones especificadas en un archivo. En condiciones normales, si se utiliza un editor para crear y modificar las construcciones, no hay necesidad de utilizar el comando *save*, ya que las construcciones se guardarán mientras está usando el editor. Sin embargo, algunas veces es conveniente introducir construcciones directamente en el indicador de CLIPS y luego guardarlas en un archivo.

7.18 CONSTRUCCIONES DE COMENTARIO

Es una buena idea incluir comentarios en un programa CLIPS, porque a veces las construcciones pueden resultar difíciles de comprender, y es posible utilizar los comentarios para explicar al lector lo que están haciendo las construcciones. Los comentarios también se emplean para tener una buena documentación de los programas y serán útiles en programas extensos.

Un comentario de CLIPS es cualquier texto que empieza con punto y coma y termina con un enter. El siguiente es un ejemplo de comentarios en un programa de incendios:


```
;*************************************
;*                                   *
;* Programador: G. D. Riley          *
;*                                   *
;* Título: El programa incendio      *
;*                                   *
;* Fecha: 10/03/92                   *
;*                                   *
;*************************************

; Deftemplates

(deftemplate urgencia "plantilla #1"
   (slot tipo))       ; Qué tipo de urgencia

(deftemplate respuesta "plantilla #2"
   (slot acción))     ; Cómo responder

; El propósito de esta regla es activar
; el sistema contra incendios en caso de un incendio

(defrule incendio-urgencia "Una regla de ejemplo" ; SI
   ; Hay una urgencia de incendio
   (urgencia (tipo incendio))
   =>                                                ; ENTONCES
   ; Activar el sistema aspersar
   (assert (respuesta
             (acción activar-sistema-contra incendios))))
```

La carga de estas construcciones en CLIPS y su impresión demostrará que cada comentario que empieza con punto y coma se elimina del programa CLIPS. El único comentario que se conserva es el que se encuentra entre comillas después del nombre de la construcción.

7.19 RESUMEN

En este capítulo se introdujeron los componentes fundamentales de CLIPS. Los hechos son el primer componente de un sistema de CLIPS y se componen de campos, que son símbolos, cadenas, enteros o puntos flotantes. El primer campo de un hecho suele utilizarse para indicar el tipo de información almacenada en el hecho y se le llama *nombre de relación*, la construcción deftemplate se utiliza para asignar nombres de ranura a campos específicos de un hecho que empieza con un nombre de relación especificado y la construcción deffacts se emplea para especificar hechos como conocimiento inicial.

Las reglas son el segundo componente de un sistema de CLIPS, se dividen en un lado izquierdo y un lado derecho, el izquierdo puede considerarse como la parte SI, y el derecho, como la parte ENTONCES. Las reglas pueden tener varios patrones y acciones.

El tercer componente de CLIPS es el mecanismo de inferencia, cuando los hechos satisfacen los patrones de las reglas se produce una activación que se coloca en la agenda, la refracción evita que hechos antiguos activen constantemente las reglas.

PROBLEMAS

Hacer

7.1 Convierta los siguientes enunciados en hechos de una instrucción deffacts. Para cada grupo de hechos relacionados, defina una deftemplate que describa una relación más general.

```
El padre de Juan es Tomás.
La madre de Juan es Susana.
Los padres de Juan son Tomás y Susana.
Tomás es un padre.
Susana es una madre.
Juan es un hijo
Tomás es hombre.
Susana es mujer.
Juan es hombre
```

7.2 Defina una deftemplate para un hecho que contiene información acerca de un conjunto. La plantilla de definición debe incluir información sobre el nombre o descripción del conjunto, la lista de elementos del conjunto y si es un subconjunto de otro. Represente los siguientes conjuntos como hechos, utilizando el formato especificado por su deftemplate.

```
A = { 1, 2, 3 }
B = { 1, 2, 3, rojo, verde }
C = { rojo, verde, amarillo, azul }
```

7.3 Una matriz escasamente poblada contiene relativamente pocos elementos diferentes a cero, y se representa de manera más eficiente como una lista o un árbol vinculado. ¿Cómo podría representarse una matriz escasamente poblada utilizando hechos? Describa la plantilla de definición empleada para los hechos que representará la matriz. ¿Cuáles son las posibles desventajas de representar una matriz con hechos en comparación con una estructura de datos de matriz en un lenguaje por procedimientos?

7.4 Convierta la red general que representa las rutas de línea aérea, que se muestran en la figura 2.4 (a) de la página 64, en una serie de hechos de una instrucción deffacts. Utilice una deftemplate sencilla para describir los hechos.

7.5 Convierta la red semántica que representa una familia, descrita en la figura 2.4 (b) de la página 64, como una serie de hechos en una instrucción de deffacts. Utilice varias deftemplates para describir los hechos producidos.

7.6 Convierta la red semántica que se muestra en la figura 2.5 de la página 65, en una serie de hechos de una instrucción deffacts. Utilice varias deftemplates para describir los hechos. Por ejemplo, los vínculos ES UN y UN TIPO DE deben ser nombres de relación y cada uno debe tener su propia deftemplate.

7.7 Convierta el árbol de decisión binaria que representa la información acerca de la clasificación de animales que se muestra en la figura 3.3 de la página 99, en una serie de hechos de una instrucción deffacts. Muestre la manera en que pueden representarse los vínculos entre los nodos. ¿Las hojas del árbol requieren una representación diferente a la de los otros nodos del árbol?

7.8 Establezca la red semántica del problema 2.1 como una instrucción de deffacts, utilizando deftemplates UN TIPO DE y ES UN.

7.9 Las plantas requieren muchos tipos diferentes de nutrientes para tener un crecimiento apropiado, tres de los más importantes nutrientes que proporcionan los fertilizantes son el nitrógeno, el fósforo y el potasio. Traduzca la siguiente heurística en reglas que determinen deficiencia de los nutrientes. Suponga que el color normal de la planta es verde.

```
Una planta con crecimiento atrofiado puede tener una
    deficiencia de nitrógeno.
Una planta de color amarillo pálido puede tener una
    deficiencia de nitrógeno.
Una planta que tiene los bordes de sus hojas café rojizos
    puede tener una deficiencia de nitrógeno.
Una planta con crecimiento de raíz atrofiado puede tener
    una deficiencia de fósforo.
Una planta con un tallo zanquivano puede tener una
    deficiencia de fósforo.
Una planta de color púrpura puede tener una deficiencia
    de fósforo.
Una planta que tiene maduración retardada puede tener
    una deficiencia de fósforo.
Una planta con los bordes de sus hojas quemados puede
    tener una deficiencia de potasio.
Una planta con tallos debilitados puede tener una
    deficiencia de potasio.
Una planta con semillas o frutos secos puede tener
    una deficiencia de potasio.
```

Utilice deftemplates para describir los hechos utilizados en las reglas. La entrada al programa debe hacerse afirmando los síntomas como hechos, la salida debe imprimirse en la terminal e indicar cuáles deficiencias de nutriente existen. Establezca un método que evite que se hagan varias impresiones para una sola deficiencia a causa de varios síntomas. Pruebe su programa con las siguientes entradas

```
La planta tiene crecimiento de raíz atrofiado.
La planta es de color púrpura
```

7.10 Los incendios se clasifican de acuerdo con el principal material de combustión. Traduzca la siguiente información en reglas para determinar clases de incendios:

```
Los incendios tipo A implican combustibles ordinarios como
    papel, madera y tela.
Los incendios tipo B implican líquidos flamables y combustibles
    (como aceite y gas), lubricantes
    y materiales similares.
```

```
Los incendios tipo C implican equipo eléctrico energizado.
Los incendios tipo D implican metales combustibles,
    como magnesio, sodio y potasio.
```

El tipo de extintor que debe utilizarse depende de la clase de incendio. Traduzca la siguiente información en reglas:

```
Los incendios tipo A deben extinguirse con extintores de
    absorción de calor o de retardo de combustión, como agua
    o líquidos basados en agua y sustancias químicas secas.
Los incendios clase B deben extinguirse extrayendo aire,
    inhibiendo la liberación de vapores combustibles o
    interrumpiendo la reacción en cadena de la combustión.
    Los extintores incluyen sustancias químicas secas, dióxido
    de carbono, espuma y bromo-trifluorometano.
Los incendios clase C deben extinguirse con un agente no
    conductor para evitar cortos circuitos. Si es posible,
    debe interrumpirse la energía eléctrica. Los extintores
    incluyen sustancias químicas secas, dióxido de carbono y
    bromotrifluorometano.
Los incendios clase D deben extinguirse con sustancias
    químicas de sofocación y de absorción de calor que no
    reaccionan con los metales de combustión. Estas sustancias
    incluyen trimetoxiboroxina y coque grafitizado cernido.
```

Describa los hechos utilizados en las reglas. La entrada al programa debe hacerse afirmando el tipo de material de combustión como un hecho; la salida debe indicar cuáles extintores pueden utilizarse y cuáles otras acciones deben realizarse, como la desconexión del suministro de energía eléctrica. Compruebe que su programa funciona para un material de cada una de las clases de incendio.

7.11 Escriba un programa de CLIPS que ayude al proceso de selección del arbusto adecuado para plantarlo. En la siguiente tabla se muestran varios arbustos y se indica si cada uno de ellos posee ciertas características, que incluyen tolerancia al frío, a la sombra, a la sequía, al suelo húmedo, al suelo ácido, al clima de la ciudad (contaminación elevada) y al crecimiento en una maceta, fácil de cuidar y rapidez de crecimiento. Una viñeta indica que el arbusto posee esa característica. La entrada al programa deben ser hechos que indiquen una característica deseada en el arbusto, y la salida debe ser una lista de las plantas que tienen la característica especificada.

Arbusto	Frío	Sombra	Sequía	Suelo húmedo	Suelo ácido	Ciudad	Maceta	Fácil de cuidar	Crecimiento rápido
Hidrangea francés		●				●	●		●
Adelfa						●	●	●	●
Arrayán brabántico	●	●	●	●		●		●	●
Madreselva de boj						●	●	●	●
Gardenia		●			●		●		
Junípero común	●		●		●	●			●
Arbusto pimentero dulce	●	●		●	●			●	
Cerezo tártaro	●	●		●		●		●	
Aucuba japonés		●	●				●	●	
Azalea de pantano		●		●	●		●		

7.12 Entre las nubes de baja altitud, aquellas que se encuentran a 2,000 metros o menos, se incluyen estratos y estratocúmulos. Entre las nubes de mediana altitud, aquellas que se encuentran entre 2,000 y 6,300 metros, se incluyen altoestratos, altocúmulos y nimboestratos. Entre las nubes de elevada altitud, aquellas que se encuentran a más de 6,300 metros, se incluyen cirros, cirroestratos y cirrocúmulos. Los cúmulos y cumulonimbos pueden encontrarse a bajas y altas altitudes. Los estratos, altroestratos, cirroestratos y cumulonimbos tienen el aspecto de grandes pilas redondeadas. Los cirros tienen un aspecto de mechones de cabello. Los nimboestratos y los cumulonimbos son nubes de lluvia y tienen un color gris oscuro. Escriba un programa que identifique tipos de nubes, la entrada del programa deben ser hechos que describan los atributos de la nube. El programa debe imprimir después el tipo de nube identificada.

7.13 Las estrellas pueden clasificarse en grupos de color llamadas *clases espectrales*. El rango de clases va desde las estrellas de clase "O" de color azul hasta las estrellas de clase "G" de color amarillo y las de clase "M" de color rojo. Esta clase espectral de la estrella se correlaciona con su temperatura: las estrellas de clase "O" tienen una temperatura de más de 20,000° C; las temperaturas de clase "B" van de 10,001 a 20,000° C; las temperaturas de clase "A" van de 7,001 a 10,000° C; las de clase "F" van de 5,701 a 7,000° C; las de clase "G" van de 4,401 a 5,700; las de clase "K" van de 3,100 a 4,400° C y las de clase "M" son de 3,100° C o menos. Las estrellas pueden clasificarse por su magnitud, que es una medida de su brillo; a menor magnitud, mayor brillo; suponga que las magnitudes van de –7 a 15. En la siguiente tabla se

presenta una lista de estrellas brillantes comunes con su clase espectral y su magnitud, además de su distancia a la Tierra en años luz. Escriba un programa que tome como entrada dos hechos que representen la clase espectral y la magnitud de una estrella, el programa debe dar salida a la siguiente información en este orden: todas las estrellas con su clase espectral especificada, todas las estrellas con su magnitud espectral y, por último, todas las estrellas que coinciden con la clase y la magnitud espectral junto con sus distancias en años luz de la Tierra.

Estrella	Clase espectral	Magnitud	Distancia
Sirius	A	1	8.8
Canopus	F	–3	98
Arturus	K	0	36
Vega	A	1	26
Capella	G	–1	46
Rigel	B	–7	880
Procyon	F	3	11
Betelgeuse	M	–5	490
Altair	A	2	16
Aldebaran	K	–1	68
Spica	B	–3	300
Antares	M	–4	250
Pollux	K	1	35
Deneb	A	–7	1630

7.14 La siguiente tabla presenta una lista de las características de gemas comunes, incluyendo su dureza (resistencia a fuerzas externas de acuerdo con la escala de Mohs), densidad (peso por unidad de volumen en gramos por centímetro cúbico) y colores. Dados tres hechos que representan la dureza, la densidad y el color de una gema, escriba las reglas necesarias para determinar si la gema es un quirsoberilo.

Gema	Dureza	Densidad	Colores
Diamante	10	3.52	amarillo, café, verde, azul, blanco, sin color
Corindón	9	4	rojo, rosa, amarillo, café, verde, azul, violeta, negro, blanco, sin color
Quirsoberilo	8.5	3.72	amarillo, café, verde
Espinel	8	3.6	rojo, rosa, amarillo, café, verde, azul, violeta, blanco, sin color
Topacio	8	3.52-3.56	rojo, rosa, amarillo, café, azul, violeta, blanco, sin color
Berilo	7.5-8.0	2.7	rojo, rosa, amarillo, café, verde, azul, blanco, sin color
Zirconio	6-7.5	4.7	amarillo, café, verde, violeta, blanco, sin color
Cuarzo	7	2.65	rojo, rosa, verde, azul, violeta, blanco, negro, sin color
Turmalina	7	3.1	rojo, rosa, amarillo, café, verde, azul, blanco, negro, sin color
Peridoto	6.5-7	3.3	amarillo, café, verde
Jadeíta	6.5-7	3.3	rojo, rosa, amarillo, café, verde, azul, violeta, blanco, negro, sin color
Opalina	5.5-6.5	2-2.2	rojo, rosa, amarillo, café, blanco, negro, sin color
Nefrita	5-6	2.9-3.4	verde, blanco, negro, sin color
Turquesa	5-6	2.7	azul

CAPÍTULO 8

Comparación de patrones

8.1 INTRODUCCIÓN

Los tipos de regla mostrados en el capítulo 7 ilustran la comparación simple de patrones con hechos. En este capítulo se introducen varios conceptos que proporcionan conocimientos útiles para igualar y manejar los hechos. El primer concepto será el uso de variables en la comparación de patrones; el segundo, las restricciones de campo; luego se introduce el uso de funciones de CLIPS y más adelante se analiza la aritmética básica. También se introducen varias funciones para realizar operaciones E/S en CLIPS.

8.2 VARIABLES

Al igual que otros lenguajes de programación, CLIPS tiene **variables** disponibles para almacenar valores. Las variables de CLIPS siempre se escriben con un signo de interrogación seguido de un nombre de campo simbólico. Los nombres de las variables siguen la sintaxis de un símbolo, con la variante de iniciar con un carácter. Para un buen estilo de programación, a las variables se les asignan nombres con significado. Algunos ejemplos de variables son

```
?velocidad
?sensor
?valor
?nombre
?color
```

No debe haber espacio entre el signo de interrogación y el nombre de campo simbólico. Como analizaremos más adelante, el signo tiene su propio uso. Las variables se utilizan en el lado izquierdo de una regla para que contengan valores de ranura que puedan compararse más tarde con otros valores del mismo lado o introducirse en el lado derecho. Los términos ***enlazado*** y ***unir*** se utilizan para describir la asignación de un valor a una variable.

Uno de los usos frecuentes de las variables es unir una de ellas al lado izquierdo de una regla y luego utilizar su valor en el lado derecho. Por ejemplo,

```
CLIPS> (clear)↵
CLIPS>
(deftemplate persona
   (slot nombre)
   (slot ojos)
    (slot cabello))↵
CLIPS>
(defrule encontrar-ojos-azules
   (persona (nombre ?nombre) (ojos azules))
   =>
    (printout t ?nombre "tiene ojos azules." crlf))↵
CLIPS>
(deffacts gente
   (persona (nombre Julia)
          (ojos azules) (cabello rojo))
   (persona (nombre Jorge)
          (ojos verdes) (cabello castaño))
   (persona (nombre Jaime)
          (ojos azules) (cabello negro))
   (persona (nombre José)
             (ojos verdes) (cabello castaño)))↵
CLIPS> (reset) ↵
CLIPS> (run) ↵
Jaime tiene ojos azules.
Julia tiene ojos azules.
CLIPS>
```

Julia y Jaime tienen ojos azules, de modo que se activa dos veces la regla *encontrar-ojos-azules,* una por cada uno por los hechos que describen a Julia y a Jaime. Cuando la regla se dispara, examina la ranura *nombre* para el hecho que activó la regla que se está disparando y usa ese valor para la instrucción de impresión.

Si se hace referencia a una variable del lado derecho de una regla, pero no está enlazada con el lado izquierdo, CLIPS imprimirá el siguiente mensaje de error (suponiendo que la variable no enlazada era ?x):

```
[PRCCODE3] Undefined variable x referenced in
           RHS of defrule.
```

8.3 USO MÚLTIPLE DE VARIABLES

Las variables con el mismo nombre que se utilizan en varios lugares del lado izquierdo de una regla tienen una propiedad importante y útil. La primera vez que una variable se enlaza a un valor, lo retiene dentro de la regla, en tanto que otras variables con el mismo nombre que están enlazadas a un valor deben enlazarse al de la primera variable.

En lugar de escribir una sola regla que busque únicamente a las personas con ojos azules, se puede afirmar un hecho que indique el color específico de ojos que habrá de buscarse, por ejemplo:

```
CLIPS> (undefrule *) ↵
CLIPS> (deftemplate encontrar (slot ojos))↵
```



```
CLIPS>
(defrule encontrar-ojos
   (encontrar (ojos ?ojos))
   (persona (nombre ?nombre) (ojos ?ojos))
   =>
    (printout t ?nombre " tiene " ?ojos " ojos." crlf))↵
CLIPS>
```

La deftemplate *encontrar* indica el color de ojos, lo que se especifica con la ranura *ojos.* Entonces la regla *encontrar-ojos* recupera el valor de la ranura *ojos* en la plantilla de definición *encontrar* y busca todos aquellos hechos de persona cuyo valor de la ranura *ojos* es el mismo que se enlazó a la variable ?ojos. El siguiente diálogo muestra la manera como funciona esto:

```
CLIPS> (reset) ↵
CLIPS> (assert (encontrar (ojos azules)))↵
<Fact-5>
CLIPS> (run)↵
Jaime tiene ojos azules.
Julia tiene ojos azules.
CLIPS> (assert (encontrar (ojos verdes)))↵
<Fact-6>
CLIPS> (run)↵
José tiene ojos verdes.
Jorge tiene ojos verdes.
CLIPS> (assert (encontrar (ojos morados))) ↵
<Fact-7>
CLIPS> (run)
CLIPS>
```

Observe que no se disparan reglas cuando se afirma el hecho (encontrar (ojos morados)) porque no hay hechos de *persona* con el valor *morados* en la ranura *ojos.*

8.4 DIRECCIONES DE HECHO

La retracción, modificación y duplicación de hechos son operaciones muy comunes y más que en el nivel superior a menudo se efectúan del lado derecho de una regla aunque ya se ha demostrado la realización de estas operaciones en la línea de comandos de nivel superior utilizando los índices de hechos. Sin embargo, antes de que pueda manipularse un hecho en el lado derecho de una regla, debe haber alguna manera de especificar el hecho que coincide con un patrón particular. Para lograr esto, es posible enlazar una variable a la **dirección de hecho** del que coincide con el patrón del lado izquierdo de una regla, empleando el operador de **unión de patrones,** "<-". Una vez enlazado, puede usarse con los comandos retract, modify o duplicate en vez de un índice de hechos. Por ejemplo, el siguiente diálogo muestra cómo actualizar los valores de ranura de la deftemplate del lado derecho de una regla:

```
CLIPS> (clear)↵
CLIPS>
(deftemplate persona
   (slot nombre)
```

```
   (slot dirección)) ↵
CLIPS>
(deftemplate cambiada
  (slot nombre)
   (slot dirección)) ↵
CLIPS>
(defrule procesar-información-cambiada
 ?f1 <- (cambiada (nombre ?nombre)
                  (dirección ?dirección))
 ?f2 <- (persona (nombre ?nombre))
 =>
 (retract ?f1)
  (modify ?f2 (dirección ?dirección))) ↵
CLIPS>
(deffacts ejemplo
   (persona (nombre "Juan Hernández")
            (dirección "Magnolias 25"))
   (cambiada (nombre "Juan Hernández")
             (dirección "Camelias 37"))) ↵
CLIPS> (reset)↵
CLIPS> (watch rules)↵
CLIPS> (watch facts)↵
CLIPS> (run)↵
FIRE     1 procesar-información-cambiada: f-2,f-1
<== f-2      (cambiada (nombre "Juan Hernández")
                       (dirección "Camelias 37"))
<== f-1      (persona (nombre "Juan Hernández")
                      (dirección "Magnolias 25"))
==> f-3      (persona (nombre "Juan Hernández")
                      (dirección "Camelias 37"))
CLIPS>
```

En este ejemplo se utilizan dos deftemplates: la deftemplate *persona* se usa para almacenar información sobre una persona, en este caso sólo su nombre y dirección, aunque también podría almacenarse otra información, como edad y color de ojos; y la deftemplate *cambiada* se usa para indicar que se cambió la dirección de una persona, que se encuentre en la ranura *dirección*.

El primer patrón de la regla *procesar-información-cambiada* determina si necesita procesarse un cambio de dirección y el segundo patrón encuentra el hecho *persona* en el que se necesitará cambiar la información de la dirección. La dirección del hecho *cambiada* se enlaza con la variable ?f1, de modo que este hecho puede retirarse una vez que se procesa el cambio. La dirección del hecho *persona* se limita a la variable ?f2, que se utiliza posteriormente en el lado derecho de la regla para modificar el valor de la ranura *dirección*.

Cabe notar que pueden emplearse variables enlazadas con valores dentro de un hecho y variables enlazadas con la dirección de hecho; además, el valor de la variable ?dirección aún puede usarse en el lado derecho después que se retira el hecho del que recibió su valor.

La retracción del hecho *cambiada* en el lado derecho de la regla *procesar-información-cambiada* es importante para que la regla funcione correctamente. Observemos lo que sucede cuando se elimina el comando *retract:*

```
CLIPS>
(defrule procesar-información-cambiada
 (cambiada (nombre ?nombre) (dirección ?dirección))
 ?f2 <- (persona (nombre ?nombre))
 =>
  (modify?f2 (dirección?dirección))).↵
CLIPS> (unwatch facts)↵
CLIPS> (unwatch rules)↵
CLIPS> (reset)↵
CLIPS> (watch facts)↵
CLIPS> (watch rules)↵
CLIPS> (watch activations)↵
CLIPS> (run 3)↵
FIRE     1 procesar-información-cambiada: f-2, f-1
<== f-1      (persona (nombre "Juan Hernández")
                      (dirección "Magnolias 25"))
==> f-3      (persona (nombre "Juan Hernández")
                      (dirección "Camelias 37"))
==> Activation 0
                procesar-información-cambiada: f-2, f-3
FIRE     2 procesar-información-cambiada: f-2, f-3
<== f-3      (persona (nombre "Juan Hernández")
                      (dirección "Camelias 37"))
==> f-4      (persona (nombre "Juan Hernández")
                      (dirección "Camelias 37"))
==> Activation 0   procesar-información-cambiada: f-2, f-4
FIRE     3 procesar-información-cambiada: f-2, f-4
<== f-4     (persona (nombre "Juan Hernández")
                     (dirección "Camelias 37"))
==> f-5              (persona (nombre "Juan Hernández")
                     (dirección "Camelias 37"))
==> Activation 0
                procesar-información-cambiada: f-2, f-5
CLIPS>
```

El programa ejecuta un lazo infinito porque afirma un nuevo hecho *persona* después de la modificación del hecho enlazado con ?f2, lo que reactiva la regla *procesar-información-cambiada.* Recuerde que un comando *modify* se trata como un comando *retract* seguido de un comando *assert.* Debido a que se indicó al comando *run* que disparara sólo tres reglas, aquí se detiene el programa; si no se proporcionan argumentos limitantes, el programa seguiría ejecutando un lazo infinito, y la única manera de detenerlo es mediante la interrupción de CLIPS usando Control-C u otro comando interruptor apropiado para la computadora en la que se esté ejecutando.

8.5 COMODINES DE UN SOLO CAMPO

A veces es útil probar la existencia de un campo dentro de una ranura sin asignar en realidad un valor a una variable, especialmente en ranuras de campo múltiple. Por ejemplo, suponiendo que se desea imprimir el número de seguro social de todas las personas con un apellido específico, a continuación se muestran la deftemplate usada para describir a cada persona, una deffacts con algunas personas predefinidas y una regla para imprimir los números del seguro social de todas las personas con ese apellido:

```
(deftemplate persona
   (multislot nombre)
   (slot número-seguro-social))

(deffacts algunas-personas
   (persona (nombre Francisco I. Madero)
            (número-seguro-social 483-98-9083))
   (persona (nombre A. Madero)
            (número-seguro-social 483-98-9084)))

(defrule imprimir-números-seguro-social
   (imprimir números-ss-para ?apellido)
   (persona (nombre ?nombre ?segundo-nombre
                    ?apellido)
            (número-seguro-social ?número-ss))
   ==>
   (printout t ?número-ss crlf))
```

La regla *imprimir-números-seguro-social* une el primer y segundo nombres de la persona con las variables ?nombre y ?segundo-nombre, respectivamente. Sin embargo, las acciones del lado derecho u otro patrón o ranura del lado izquierdo de la regla no hacen referencia a estas variables. Cuando se necesita un campo, pero el valor no es importante, puede utilizarse un comodín de un solo campo que se represente con un signo de interrogación-en lugar de una variable. Usando comodines de un solo campo, la regla *imprimir-números-seguro- social* puede reescribirse como sigue:

```
(defrule imprimir-números-seguro-social
   (imprimir-números-ss-para ?apellido)
   (persona (nombre ? ? ?apellido)
            (número-seguro-social ?número-ss))
   ==>
   (printout t ?número-ss crlf))
```

Observemos que la ranura *nombre* de un hecho *persona* debe contener exactamente tres campos para activar la regla *imprimir-números-seguro social.* Por ejemplo, el hecho

```
(persona (nombre Jorge Pérez)
         (número-seguro-social 345-89-3039))
```

no satisface la regla *imprimir-números-seguro-social.*

Es extremadamente importante no incluir un espacio entre el signo de interrogación y el nombre simbólico de una variable; el patrón


```
(persona (nombre ?nombre ?apellido))
```

espera dos campos para la ranura *nombre,* pero el patrón

```
(persona (nombre ?nombre ? apellido))
```

espera tres campos para la ranura *nombre* y el último campo debe ser el símbolo *apellido.*

Cuando una ranura de un solo campo se deja sin especificar en un patrón, CLIPS le agrega automáticamente un comodín de un solo campo. Por ejemplo, el patrón:

```
(persona (nombre ?nombre ?apellido))
```

se convierte en

```
(persona (nombre ?nombre ?apellido)
         (número-seguro-social ?))
```

8.6 MUNDO DE BLOQUES

Para demostrar la unión de variables, elaboremos un programa para mover los bloques de un modelo simple de grupos; este programa es análogo al clásico programa de mundo de bloques donde el dominio del conocimiento está en bloques (Firebaugh 88). Se trata de un buen ejemplo de planeación y podría aplicarse a la fabricación automatizada, en la que un brazo robot manipula piezas.

La segunda restricción será que aún no debe haberse alcanzado algún objetivo; es decir, el objetivo no puede ser el movimiento del bloque x a la parte superior del bloque y, si ya se encuentra ahí. Se trata de una condición cuya verificación resulta simple; sin embargo, la sintaxis apropiada para probar esta condición sólo se introducirá hasta el capítulo 9.

Para empezar a resolver este problema, será útil establecer una configuración de bloques que pueda usarse para probar el programa. En la figura 8.1 se muestra la configuración que se utilizará; en ella hay dos pilas , la primera tiene al bloque A encima del bloque B que está arriba del bloque C; la segunda tiene el bloque D encima del bloque E que se encuentra arriba del bloque F.

Para determinar el tipo de reglas eficaz para solucionar el problema, lo adecuado es completar paso a paso un objetivo del mundo de bloques. ¿Cuáles pasos deben darse para poner el bloque C arriba del bloque E? La solución más fácil para este problema sería colocar directamente el bloque C encima del bloque E; no obstante, esta regla sólo podría aplicarse si ambos no tuvieran otros bloques encima de ellos. El pseudocódigo para esta regla sería

```
REGLA MOVER-DIRECTAMENTE
SI        El objetivo es poner el bloque ?superior encima
            del bloque ?inferior y
          el bloque ?superior es el bloque superior en su
            pila y
          el bloque ?inferior es el bloque superior en su
            pila,
ENTONCES Mover el bloque ?superior a la parte superior del
            bloque ?inferior.
```

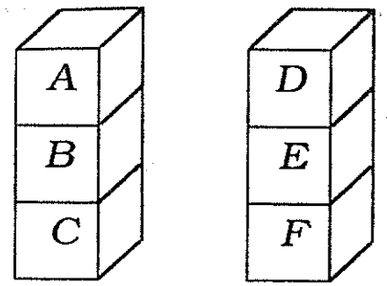


Figura 8.1 Configuración inicial del modelo de bloques

Lo único que interesa en un modelo de bloques son los propios bloques, y sólo puede apilarse uno sobre otro. El objetivo de un programa complejo de mundo de bloques es reorganizar las pilas hasta la configuración deseada con la cantidad mínima de movimientos. En este ejemplo se harán varias restricciones para simplificarlo, la primera de ellas es que sólo se permite un objetivo inicial y éste sólo puede ser colocar un bloque encima de otro; con esta restricción resulta más bien trivial determinar los movimientos óptimos para alcanzar el objetivo, pues si éste consiste en colocar el bloque x encima del bloque y, entonces hay que poner en el piso todos los bloques (si hay alguno) que se encuentran encima de x y de y para luego colocar el bloque x encima del y.

Para la configuración en la figura 8.1, no puede utilizarse la regla *mover-directamente*, porque los bloques A y B se encuentran encima del bloque C, y el bloque D está arriba del bloque E. Para que la regla *mover-directamente* ponga el bloque C encima del bloque E, los bloques A, B y D deben colocarse en el piso y como éste es el paso más sencillo de realizar para quitarlos de la ruta, eso es lo que pasa. El mundo de bloques simple no requiere que los bloques se vuelvan a apilar y sólo permite un objetivo inicial, de modo que no hay necesidad de apilar los bloques que se apartan de la ruta. Esta regla puede expresarse como dos reglas de pseudocódigo: una para quitar los bloques que están encima del que se moverá y otra para quitar los que están encima del que se apilará.

```
REGLA      DESPEJAR-BLOQUE-SUPERIOR
SI         El objetivo es mover el bloque ?x y éste
           no es el bloque superior de su pila y
           el bloque ?arriba está encima del bloque ?x,
ENTONCES   El objetivo es colocar el bloque ?arriba en el
           piso

REGLA      DESPEJAR-BLOQUE-INFERIOR
SI         El objetivo es poner otro bloque encima del
           bloque ?x y éste
           no es el bloque superior de su pila y
           el bloque ?arriba está encima del bloque ?x,
ENTONCES   El objetivo es colocar el bloque ?arriba en el
           piso
```

La regla *despejar-bloque-superior* funcionará para liberar al bloque C, primero determinará que el bloque B debe moverse al piso, y para hacerlo, también determinará que es necesario

mover primero el bloque A. Del mismo modo, la regla *despejar-bloque-inferior* determinará que para poner algo arriba del bloque E necesita moverse al piso el bloque D.

Ahora hay subobjetivos para colocar los bloques A, B y D en el piso, lo que puede hacerse directamente con A y D. Si se escribe en forma apropiada, la regla *mover-directamente* puede manejar los bloques colocados en el piso, además de otros bloques. Como el piso no es en realidad un bloque, puede ser necesario tratarlo de manera diferente. *La siguiente regla en forma de pseudocódigo* manejará el caso especial del movimiento de un bloque al piso:

```
REGLA      MOVER-AL-PISO
SI         El objetivo es colocar el bloque ?superior en el
             piso y
           el bloque ?superior es el bloque superior de su
             pila,
ENTONCES   Mover el bloque ?superior a la superficie del
             piso.
```

La regla *mover-al-piso* ahora puede poner los bloques A y D en el piso; una vez que lo hace con A, se puede activar de nuevo para mover el bloque B. Con los bloques A, B y D en el piso, los bloques C y E ahora ocupan la parte superior de sus pilas y es posible usar la regla *mover-directamente* para poner el bloque C encima del E.

Ahora que se han escrito las reglas usando un pseudocódigo, deben determinarse los hechos que utilizarán las reglas, aunque no siempre es posible determinar sin algún prototipo los tipos de hechos que son necesarios. En este caso, las reglas de pseudocódigo señalan varios tipos de hechos que se necesitarán; por ejemplo, es crucial conocer cuáles bloques están encima de otros. Esta información podría describirse con la siguiente deftemplate:

```
(deftemplate encima-de
   (slot superior)
   (slot inferior))
```

y los hechos descritos para esta plantilla serían

```
(encima-de (superior A) (inferior B))
(encima-de (superior B) (inferior C))
(encima-de (superior D) (inferior E))
(encima-de (superior E) (inferior F))
```

Debido a que también es importante saber cuáles bloques están en la parte superior y en la base de una pila, sería útil incluir los siguientes hechos:

```
(encima-de (superior nada) (inferior A))
(encima-de (superior C) (inferior piso))
(encima-de (superior nada) (inferior D))
(encima-de (superior F) (inferior piso))
```

Las palabras *nada* y *piso* tienen un significado especial en estos hechos. Los hechos (arriba-de (superior nada) (inferior A)) y (arriba-de (superior nada) (inferior D)) indican que A y D son los bloques ubicados en la parte superior de sus pilas; del mismo modo, los hechos (arriba-de (superior C) (inferior piso)) y (arriba-de (superior F) (inferior piso)) indican que los bloques C y F son la base de sus pilas. La inclusión de estos hechos no resuelve necesaria-

mente el problema de determinar los bloques superior e inferior de una pila, porque si las reglas no están escritas de manera correcta, las palabras *piso* y *nada* podrían confundirse con los nombres de los bloques. Los hechos que indican los nombres de los bloques pueden ser útiles, como los siguientes hechos que usan la deftemplate *construir* implícita para diferenciar los bloques de las palabras especiales *nada* y *piso*:

```
(bloque A)
(bloque B)
(bloque C)
(bloque D)
(bloque E)
(bloque F)
```

Por último, se necesita un hecho para describir los objetivos del movimiento de los bloques que se procesan. Estos objetivos podrían describirse con la siguiente deftemplate

```
(deftemplate objetivo (slot mover) (slot encima-de))
```

y el objetivo inicial utilizando esta deftemplate sería

```
(objetivo (mover C) (arriba-de E))
```

Con los hechos y deftemplates definidos ahora, la configuración inicial del mundo de bloques puede describirse mediante los siguientes deffacts:

```
(deffacts estado-inicial
   (bloque A)
   (bloque B)
   (bloque C)
   (bloque D)
   (bloque E)
   (bloque F)
   (encima-de (superior nada) (inferior A))
   (encima-de (superior A) (inferior B))
   (encima-de (superior B) (inferior C))
   (encima-de (superior C) (inferior piso))
   (encima-de (superior nada) (inferior D))
   (encima-de (superior D) (inferior E))
   (encima-de (superior E) (inferior F))
   (encima-de (superior F) (inferior piso))
   (objetivo (poner C) (encima-de E))
```

La regla *mover-directamente* se escribe como sigue:

```
(defrule mover-directamente
   ?objetivo <- (objetivo (poner ?bloque1)
                          (encima-de ?bloque2))
   (bloque ?bloque1)
   (bloque ?bloque2)
   (encima-de (superior nada) (inferior ?bloque1))
```



```
   ?pila-1 <- (encima-de (superior ?bloque1)
                         (inferior ?bloque3))
   ?pila-2 <- (arriba-de (superior nada)
                         (inferior ?bloque2))
   =>
   (retract ?objetivo ?pila-1 ?pila-2)
   (assert (encima-de (superior ?bloque1)
                      (inferior ?bloque2))
           (encima-de (superior nada)
                      (inferior ?bloque3)))
(printout t ?bloque1 " colocado encima de " ?bloque2
          " . " crlf))
```

Los primeros tres patrones determinan que existe el objetivo de colocar un bloque encima de otro. Los patrones dos y tres aseguran que esta regla no procesará un objetivo para mover un bloque al piso, los patrones cuarto y sexto revisan que los bloques sean los de la parte superior de sus pilas. El quinto y sexto patrones se comparan con la información necesaria para actualizar las pilas de las que se están tomando y a las que se están moviendo los bloques. Las acciones de la regla actualizan la información para ambas pilas e imprimen un mensaje, pues ahora el bloque que se encuentra abajo del que se movió es el bloque superior de esa pila, y el que se movió es ahora el bloque superior de la pila en la que se colocó.

La regla *mover-al-piso* se establece como sigue:

```
(defrule mover-al-piso
   ?objetivo <- (objetivo (colocar ?bloque1) (en-el piso))
   (bloque ?bloque1)
   (encima-de (superior nada (inferior ?bloque1))
   ?pila <- (encima-de (superior ?bloque1)
                       (inferior ¿?bloque 2))
   =>
   (retract ?objetivo ?pila)
   (assert (encima-de (superior ?bloque1)
                      (inferior piso))
           (encima-de (superior nada)
                      (inferior ?bloque2)))
   (printout t ?bloque1 " colocado en el piso."
             crlf))
```

Esta regla es similar a *mover-directamente* con la excepción de que no es necesario actualizar la información acerca del piso, porque no es éste un bloque.

La regla *despejar-bloque-superior* se establece como sigue:

```
(defrule despejar-bloque-superior
   (objetivo (mover ?bloque1))
   (bloque ?bloque1)
   (encima-de (superior ?bloque2) (inferior ?bloque1))
   (bloque ?bloque2)
   =>
```

```
    (assert (objetivo (poner ?bloque2)
                      (en-el piso))))
```

La regla *despejar-bloque-inferior* se establece así:

```
(defrule despejar-bloque-inferior
   (objetivo (encima-de ?bloque1))
   (bloque ?bloque1)
   (encima-de (superior ?bloque2) (inferior ?bloque1))
   (bloque ?bloque2)
   =>
   (assert (objetivo (poner ?bloque2)
                     (en-el piso))))
```

Ahora, con las reglas *mover-directamente, mover-al-piso, despejar-bloque-superior* y *despejar-bloque inferior,* las deftemplates *objetivo* y *arriba-de* y la deffacts *estado-inicial*, el programa está completo La siguiente salida muestra una ejecución de este programa de mundo de bloques:

```
CLIPS> (unwatch all) ↵
CLIPS> (reset) ↵
CLIPS> (run) ↵
A colocado en el piso.
B movido al piso.
D movido al piso.
C colocado encima de E.
CLIPS>
```

Primero se colocan en el piso los bloques A y B para despejar el bloque C, luego el bloque D se mueve al piso para despejar el bloque E, y por último, el bloque C puede colocarse encima del E para lograr el objetivo inicial.

Con este ejemplo se ha demostrado cómo construir un programa utilizando un método paso a paso: en primer lugar, se escribieron las pseudorreglas usando un texto común, se usaron las pseudorreglas para determinar los tipos de hechos que se necesitarían, se diseñaron deftemplates para describir los hechos y se codificó el conocimiento inicial para el programa utilizando esas deftemplates; finalmente, se tradujeron las pseudorreglas a reglas de CLIPS usando las deftemplates como guía.

El desarrollo de un sistema experto suele requerir mucho más desarrollo de prototipo e iterativo que el que se muestra en este ejemplo, puesto que no siempre es posible determinar el mejor método para representar los hechos o tipos de reglas que se necesitarán para construirlo. Sin embargo, el uso de una metodología estable puede ayudar al desarrollo del sistema.

8.7 COMODINES Y VARIABLES DE CAMPO MÚLTIPLE

Comodines de campo múltiple

Es posible utilizar comodines y variables de campo múltiple para compararlos con cero o más campos de un patrón. El **comodín de campo múltiple** se indica con un signo de pesos seguido

por un signo de interrogación, "$?", y representa a cero o más casos de un campo; considerando que las variables y los comodines ordinarios coinciden exactamente con un solo campo, ésta es una diferencia pequeña pero significativa. Para ilustrar el uso de los comodines de campo múltiple, tomemos la regla *imprimir-números-seguro-social* descrita en la sección 8.5:

```
(defrule imprimir-números-seguro-social
   (imprimir-números-ss-para ?apellido)
   (persona (nombre ? ? ?apellido)
            (número-seguro-social ?número-ss))
   =>
   (printout t ?número-ss crlf))
```

Esta regla sólo coincidirá con una ranura de nombre que tenga exactamente tres campos. El hecho:

```
(persona (nombre Jorge Pérez)
         (número-seguro-social 345-89-3039))
```

no coincidirá con la regla. Sin embargo, si los dos comodines de un solo campo se reemplazan con un comodín de campo múltiple (como se muestra a continuación), entonces el patrón *persona* coincidirá con cualquier ranura de nombre que contenga cuando menos un campo y tenga como último campo el nombre especificado (como campo apellido):

```
(defrule imprimir-números-seguro-social
   (imprimir-números-ss-para ?apellido)
   (persona (nombre $? ?apellido)
            (número-seguro-social ?número-ss))
   =>
   (printout t ?número-ss crlf))
```

Asimismo, cuando una ranura de campo múltiple se deja sin especificar en un patrón, CLIPS le agrega automáticamente una marca de comodín de campo múltiple. Por ejemplo, el patrón:

```
(persona (número-seguro-social ?número-ss))
```

se convierte en:

```
(persona (nombre $?)
         (número-seguro-social ?número-ss))
```

Variables de campo múltiple

Al igual que las variables de un solo campo van precedidas por "?", las variables de campo múltiple van precedidas por "$?". Las siguientes construcciones muestran cómo imprimir los nombres de todos los hijos de una persona específica:

```
(deftemplate persona
   (multislot nombre)
   (multislot hijos))
```

```
(deffacts algunas-personas
   (persona (nombre Francisco I. Madero)
            (hijos Julia Pablo María))
   (persona (nombre José A. Madero)
            (hijos Ricardo)))

(defrule imprimir-hijos
   (imprimir-hijos $?nombre)
   (persona (nombre $?nombre)
            (hijos $?hijos))
   =>
   (printout t ?nombre " tiene hijos " ?hijos
               crlf))
```

El primer patrón de la regla *imprimir-hijos* une la variable $?nombre con el nombre de la persona cuyos hijos se imprimirán. El segundo patrón compara el hecho *persona* con el nombre específico contenido en $?nombre y luego une la lista de los hijos de esa persona con la variable $?hijos; después imprime este valor del lado derecho de la regla.

Observe que cuando se hace referencia a una variable de campo múltiple en el lado derecho de una regla, no es necesario incluir el signo $ como parte del nombre de la variable; este signo sólo se usa en el lado izquierdo para indicar que pueden enlazarse cero o más campos a la variable.

El siguiente diálogo muestra cómo funciona la regla *imprimir- hijos:*

```
CLIPS> (reset)↵
CLIPS> (assert (imprimir-hijos Francisco I. Madero))↵
<Fact-3>
CLIPS> (run)↵
(Francisco I. Madero) tiene hijos (Julia Pablo María)
CLIPS>
```

Observe que los valores de campo múltiple enlazados a las variables ?nombre e ?hijos están encerrados entre paréntesis al momento de imprimirse.

Se puede usar más de una variable de campo múltiple en una sola ranura; por ejemplo, suponiéndose que se desea encontrar a todas las personas que tiene un hijo con un nombre específico, la siguiente regla realiza esta tarea:

```
(defrule encontrar-hijo
   (encontrar-hijo ?hijo)
   (persona (nombre $?nombre)
            (hijos $?antes ?hijo $?después))
   =>
   (printout t ?nombre " tiene hijo " ?hijo crlf)
   (printout t "Otros hijos son "
               ?antes " " ?después crlf))
```

Por lo general, si sólo estamos interesados en el valor de la variable ?hijo, las variables ?antes y ?después tendrían que reemplazarse con comodines de campo múltiple (y se eliminaría la instrucción de impresión que hace referencia a estas variables). El siguiente diálogo muestra cómo funciona la regla *encontrar-hijo:*


```
CLIPS> (reset)↵
CLIPS> (assert (encontrar-hijo Pablo))↵
<Fact-3>
CLIPS> (run)↵
(Francisco I. Madero) tiene un hijo Pablo
Otros hijos son (Julia) (María)
CLIPS> (assert (encontrar-hijo Ricardo))
<Fact-4>
CLIPS> (run) ↵
(José A. Madero) tiene hijo Ricardo
Otros hijos son () ()
CLIPS> (assert (encontrar-hijo Beto)) ↵
<Fact-5>
CLIPS> (run) ↵
CLIPS>
```

Cuando Pablo es el hijo enlazado con la variable ?hijo, la variable ?antes se enlaza con (Julia) y la variable ?después se enlaza con (María). Del mismo modo, cuando la variable ?hijo se enlaza con Ricardo, entonces las variables ?antes y ?después se enlazan con (), un campo múltiple que tiene cero campos.

Comparación de patrones en más de una forma

Hasta ahora se ha tratado con situaciones en las que un solo hecho puede igualar un patrón de una sola forma, pero utilizando variables o comodines de campo múltiple, es posible que coincida con un patrón en más de una forma. Supongamos que se introduce un hecho para una persona que puso a todos sus hijos su propio nombre:

```
CLIPS> (reset)↵
CLIPS> (assert (persona (nombre Jorge Cincohombres)
                        (hijos Jorge Jorge Jorge)))↵
<Fact-3>
CLIPS> (assert (encontrar-hijo Jorge)) ↵
<Fact-4>
CLIPS> (agenda) ↵
0       encontrar-hijo: f-4, f-3
0       encontrar-hijo: f-4, f-3
0       encontrar-hijo: f-4, f-3
For a total of 3 activations.
CLIPS> (run) ↵
(Jorge Cincohombres) tiene hijo Jorge
Otros hijos son () (Jorge Jorge)
(Jorge Cincohombres) tiene hijo Jorge
Otros hijos son (Jorge) (Jorge)
(Jorge Cincohombres) tiene hijo Jorge
Otros hijos son (Jorge Jorge) ()
CLIPS>
```

Como lo muestran las reglas disparadas, hay tres maneras distintas en que pueden enlazarse las variables ?hijo, ?antes y ?después con el hecho f–3. En el primer caso, ?antes está enla-

zado con (), ?hijo con *Jorge* y ?después con (Jorge Jorge). En el segundo caso, ?antes se enlaza con (Jorge), ?hijo con *Jorge* y ?después con (Jorge). En el tercer caso, ?antes está enlazado con (Jorge Jorge), ?hijo con *Jorge* y ?después con ().

Introducción de una pila

Una **pila** es una estructura de datos ordenada a la que se le pueden agregar y eliminar elementos en uno de sus "extremos". Se puede incluir (agregar) un nuevo elemento o extraer (eliminar) el último elemento agregado de la pila, de esta manera el primer valor agregado a una pila es el último elemento que se eliminará y el último elemento agregado es el primero que se eliminará.

Una analogía útil para una pila es un conjunto de charolas de cafetería, si se agregan (incluyen) nuevas charolas encima de las charolas que ya están en la pila, las últimas charolas que se agregan a la pila serán las primeras que se eliminen (extraigan).

Resulta relativamente fácil introducir una pila que pueda realizar operaciones de inclusión y extracción con el uso de variables de campo múltiple. En primer lugar, se usará un hecho ordenado *pila* que contiene una lista de elementos; la siguiente regla insertará un valor al hecho *pila:*

```
(defrule incluir-valor
   ?incluir-valor <- (incluir-valor ?valor)
   ?pila <- (pila $?resto)
   =>
   (retract ?incluir-valor ?pila)
   (assert (pila ?valor $?resto))
   (printout t "Inclusión de valor " ?valor crlf))
```

Son necesarias dos reglas para establecer la extracción: una para una pila vacía y otra si la pila tiene un valor por extraer.

```
(defrule extraer-valor-válido
   ?extraer-valor <- (extraer-valor)
   ?pila <- (pila ?valor $?resto)
   =>
   (retract ?extraer-valor ?pila)
   (assert (pila $?resto))
   (printout t "Extracción de valor " ?valor crlf))

(defrule extraer-valor-no-válido
   ?extraer-valor <- (extraer-valor)
   (pila)
   =>
   (retract ?extraer-valor)
   (printout t "Extracción desde pila vacía" crlf))
```

Estas reglas pueden cambiarse con facilidad para incluir y extraer valores de pilas con nombre. Por ejemplo, los patrones:

```
?incluir-valor <- (incluir-valor ?valor)
?pila <- (pila $?resto)
```


pueden reemplazarse con:

```
?incluir-valor <- (incluir-valor ?nombre ?valor)
?pila <- (pila ?nombre $?resto)
```

donde ?nombre representa el nombre de la pila.

Utilizando de nuevo el modelo de bloques

El problema del modelo de bloques puede reejecutarse de una manera mucho más sencilla usando comodines y variables de campo múltiple. Como se muestra a continuación cada pila puede representarse por medio de un hecho simple y las operaciones sobre el movimiento de bloques son similares a las que se usan en el ejemplo de incluir/excluir:

```
(deffacts estado-inicial
   (pila A B C)
   (pila D E F)
   (objetivo (poner C) (encima-de E))
   (pila))
```

Se incluye el hecho *pila* vacío para evitar que este hecho se agregue más tarde; por ejemplo, cuando una pila tiene un bloque y ese se mueve para colocarlo encima de otro.

Las reglas para el programa del modelo de bloques que usan variables de campo múltiple son:

```
(defrule mover-directamente
   ?objetivo <- (objetivo (poner ?bloque1)
                  (encima-de ?bloque2))
   ?pila-1 <- (pila ?bloque1 $?resto1)
   ?pila-2 <- (pila ?bloque2 $?resto2)
   =>
   (retract ?objetivo ?pila-1 ?pila-2)
   (assert (pila $?resto1))
   (assert (pila ?bloque1 ?bloque2 $?resto2))
   (printout t ?bloque1 " puesto encima de "
               ?bloque2 "." crlf))

(defrule poner en-piso
   ?objetivo <- (objetivo (poner ?bloque1) (en-el piso))
   ?pila-1 <- (pila ?bloque1 $?resto)
   =>
   (retract ?objetivo ?pila-1)
   (assert (pila ?bloque 1))
   (assert (pila $?resto))
   (printout t ?bloque1 " puesto en piso."
               crlf))

(defrule despejar-bloque-superior
   (objetivo (poner ?bloque1))
   (pila ?encima $? ?bloque1 $?)
```

```
    =>
    (assert (objetivo (poner ?arriba) (en-el piso))))

(defrule despejar-bloque-inferior
    (objetivo (encima-de ?bloque1))
    (pila ?arriba $? ?bloque1 $?)
    =>
    (assert (objetivo (poner ?arriba) (en-el piso))))
```

8.8 RESTRICCIONES DE CAMPO

Restricción de campo *Not*

Además de las funciones elementales de comparación de patrones por uniones de constantes y variables literales, CLIPS tiene operadores para comparación de patrones más eficaces. A fin de explicar estas funciones, se utiliza nuevamente el problema de determinar grupos de personas con cierto color de ojos y cabello. Por ejemplo, suponiendo que sea necesario encontrar a todas las personas que *no* tengan cabello castaño, una manera de hacerlo es escribiendo reglas para cada tipo de color de cabello. La siguiente regla encuentra personas con cabello negro (la deftemplate *persona* de la sección 8.2 se usa para los ejemplos de esta sección):

```
(defrule cabello-negro-no-es-cabello-castaño
    (persona (nombre ?nombre) (cabello negro))
    =>
    (printout t ?nombre " no tiene cabello castaño"
              crlf))
```

Otra regla encuentra personas con cabello rubio.

```
(defrule cabello-rubio-no-es-cabello-castaño
    (persona (nombre ?nombre) (cabello rubio))
    =>
    (printout ?nombre " no tiene cabello castaño"
              crlf))
```

Incluso podría escribirse otra regla para encontrar personas pelirrojas. El problema de escribir las reglas de esta manera es que la condición que se comprueba es que el cabello no sea castaño; las reglas anteriores tratan de probar esta condición de forma indirecta (es decir, determinan todos los colores de cabello diferentes al castaño y escriben una regla para cada uno), y solo si se pueden especificar todos los colores funcionará esta técnica. Para este ejemplo, sería más sencillo suponer que el color del cabello podría ser cualquiera (incluso morado o verde).

Una manera de manejar este problema es usando una **restricción de campo** para restringir los valores que un campo puede tener en el lado izquierdo, una de ellas es la ***restricción conectiva*** (llamada así porque se utiliza para conectar variables y otras restricciones). Hay tres tipos de restricciones conectivas, a la primera se le llama **restricción *not***, su símbolo es la **tilde**, ~, y actúa sobre la restricción o variable que le sigue en forma inmediata; si la siguiente restricción coincide con un campo, entonces la restricción *not* falla; si no lo hace, entonces tiene éxito. En esencia, *not* niega el resultado de la siguiente restricción.

La regla que busca a las personas que no tienen cabello castaño puede escribirse mucho más fácilmente utilizando la restricción *not:*

```
(defrule persona-sin-cabello-castaño
    (persona (nombre ?nombre) (cabello ~castaño))
    =>
    (printout t ?nombre " no tiene cabello castaño "
              crlf))
```

Con el uso de la restricción *not,* esta sola regla hace el trabajo de muchas otras que necesitan especificar cada posible color de cabello.

Restricción de campo *Or*

La segunda restricción conectiva es la **restricción *or***, que se representa con una barra, |, y se usa para permitir que uno o más valores posibles coincidan con el campo de un patrón.

Por ejemplo, la siguiente regla encuentra a todas las personas con cabello negro o castaño usando la restricción *or:*

```
(defrule cabello-negro-o-castaño
    (persona (nombre ?nombre) (cabello castaño | negro))
    =>
    (printout t ?nombre " tiene cabello oscuro " crlf))
```

Al afirmar los hechos (persona (nombre Jorge) (ojos azules) (cabello castaño)) y (persona (nombre Marcos) (ojos café (cabello negro)) se colocaría una activación de esta regla para cada uno de los hechos en agenda.

Restricción de campo *And*

El tercer tipo de restricción conectiva es la **restricción *and***, que es diferente del elemento condicional *and* explicado en la sección 7.15, su símbolo es **&** y por lo general sólo se usa junto con otras restricciones, de otra manera no es de uso muy práctico.

La restricción *and* resulta útil en el caso la colocación de restricciones adicionales en el proceso de enlazado de una variable. Suponiendo, por ejemplo, que un hecho *persona* con el cabello color castaño o negro dispara una regla, el patrón para encontrarlo puede expresarse fácilmente utilizando la restricción *or,* como se mostró en el ejemplo anterior, sin embargo, ¿Cómo puede identificarse el valor del color del cabello? La solución es unir una variable al color que se compara utilizando la restricción *and* y después imprimir la variable:

```
(defrule cabello-negro-o-castaño
    (persona (nombre ?nombre) (cabello ?color&castaño |
negro))
    =>
    (printout t ?nombre " tiene " ?color " cabello " crlf))
```

La variable ?color se enlazará con cualquier color que coincida con la restricción castaño|negro.

La restricción *and* también es útil en conjunto con la restricción *not*. Por ejemplo, la siguiente regla se dispara cuando el color de cabello de una persona no es negro ni castaño:

```
(defrule cabello-negro-o-castaño
   (persona (nombre ?nombre)
            (cabello ?color&~castaño~negro))
   =>
   (printout t ?nombre " tiene " ?color " cabello " crlf))
```

Combinación de las restricciones de campo

Las restricciones de campo pueden usarse junto con variables y otros valores literales para proporcionar eficaces opciones de comparación de patrones. Por ejemplo, suponiendo que se necesita una regla para determinar si existen dos personas con las condiciones descriptivas siguientes: la primera persona tiene ojos azules o verdes y no tiene cabello negro, la segunda persona no tiene el mismo color de ojos que la primera y tiene cabello rojo o del mismo color que la primera; la siguiente regla comparará estas restricciones:

```
(defrule comparación-compleja-ojos-cabello
   (persona (nombre ?nombre1)
            (ojos ?ojos1&azules|verdes)
            (cabello ?cabello1&~negro))
   (persona (nombre ?nombre2&~?nombre1)
            (ojos ?ojos2&~?ojos1)
            (cabello ?cabello2&rojo|?cabello1))
   =>
   (printout t ?nombre1 " tiene " ?ojos1 " ojos y "
               ?cabello1 " cabello " crlf)
   (printout y ?nombre2 " tiene " ?ojos2 " ojos y "
               ?cabello2 " cabello " crlf))
```

Analizando este ejemplo con más detalle, se verá que la restricción ?ojos1&azules|verdes de la primera ranura *ojos* del primer patrón, une el color de ojos de la primera persona con la variable ?ojos1, si el valor del color de ojos del hecho que se compara es azules o verdes; la restricción ?cabello1&~negro de la ranura *cabello* del primer patrón, une la variable ?cabello1 si el valor de color del cabello del hecho que se compara no es negro.

La restricción ?nombre2&~?nombre1 de la ranura *nombre* del segundo patrón realiza una útil operación, unir el valor de la ranura *nombre* del hecho *persona* con la variable ?nombre2, si ésta no tiene el mismo valor de ?nombre1. Si los nombres son identificadores únicos (es decir, dos personas no tienen el mismo nombre en la base de datos), esto asegurará que ambos patrones no coincidan con el mismo hecho, lo que no puede pasar con estos dos patrones porque el color de ojos de la segunda persona debe ser diferente del de la primera. Sin embargo, es una técnica que conviene comprender y que tiene muchas aplicaciones útiles. En la sección 12.2 se muestra cómo puede aplicarse esta misma técnica por medio de direcciones de hecho para garantizar que todos los hechos son diferentes. Esto permitirá que la regla funcione para personas con los mismos nombres pero con diferente color de ojos y cabello.

La restricción ?ojos2&~?ojos1 de la ranura *ojos* del segundo patrón realiza muchas de las mismas pruebas que la restricción previa en tanto que La restricción final, ?cabello2&-rojo|?cabello1, de la ranura *cabello* del segundo patrón une el valor color del cabello de la segunda persona con la variable ?cabello2 si el color del cabello es rojo o tiene el mismo valor de la variable ?cabello1. Observe que si una variable se usa como parte de una restricción de campo *or* debe estar ya enlazada.

Las variables sólo se unirán si son la primera condición de un campo y sólo si ocurren individualmente o si están vinculadas con otras condiciones mediante una restricción conectiva *and*. Por ejemplo, la regla:

```
(defrule mal-uso-de-variable
   (persona (nombre ?nombre) (cabello rojo|?cabello))
   =>
   (printout t ?nombre " tiene " ?cabello " cabello "
crlf))
```

producirá un error porque ?cabello no está enlazado.

Como nota final para la combinación de restricciones, recordemos que hay combinaciones que no tienen propósitos útiles: por ejemplo, usar la restricción *and* para unir constantes literales (por ejemplo, negro&azul) siempre hará que la restricción no se satisfaga, a menos que las literales sean idénticas. De igual manera, unir literales negadas utilizando la restricción *or* (por ejemplo, ~negro|~azul) siempre hará que se satisfaga la restricción.

8.9 FUNCIONES Y EXPRESIONES

Funciones matemáticas elementales

Además de tratar con hechos simbólicos, CLIPS puede realizar cálculos. Sin embargo, hay que tener presente que un lenguaje de sistema experto como CLIPS no está diseñado para un trabajo intensivo con números pues aún cuando las funciones matemáticas de CLIPS son versátiles, sirven principalmente para modificar los números sobre los que está razonando el programa de aplicación. Otros lenguajes como FORTRAN son mejores para el trabajo intensivo con números con poco o ningún razonamiento. CLIPS proporciona los operadores aritméticos elementales, como se muestra en la tabla 8.1. (El apéndice E contiene una lista de otras funciones matemáticas que proporciona CLIPS.)

Operadores aritméticos	Significado
+	Suma
–	Resta
*	Multiplicación
/	División

Tabla 8.1 Operadores aritméticos elementales de CLIPS

En CLIPS, las expresiones numéricas se representan de acuerdo con el estilo de LISP. En LISP y CLIPS, una expresión numérica que normalmente se escribiría como 2 + 3 debe escribirse en **forma de prefijo,** (+ 2 3); en tanto que a la forma acostumbrada de escribir expresiones numéricas se le llama ***forma infija***, porque los operadores matemáticos están entre los **operandos** o **argumentos.** En la forma de prefijo de CLIPS, el operador debe ir antes de los operandos y la expresión numérica debe estar entre paréntesis.

Es relativamente fácil convertir de formato infijo a prefijo; suponiendo por ejemplo que habrán de revisarse dos puntos para ver si tienen una pendiente positiva, en la forma infija acostumbrada esto se escribe:

```
(y2 - y1 ) / (x2 - x1) > 0
```

Observe que el símbolo mayor que, >, es una función de CLIPS que determina si el primer argumento es mayor que el segundo. (La función > se describe en el apéndice E.) Para escribir esta expresión en forma de prefijo será útil simplificarla pensando en el numerador como (Y) y en el denominador como (X); de esta manera, la expresión anterior se escribe:

```
(Y)  /  (X) > 0
```

La forma de prefijo para la división es entonces:

```
(/  (Y) (X))
```

Ya que en el prefijo el operador está antes del argumento. Ahora debe probarse el resultado de la división para determinar si es mayor que 0. De esta manera, la forma de prefijo es:

```
(>  (/  (Y) (X))  0)
```

En forma infija, Y = y2 – y1, pero necesita usarse la forma de prefijo porque se está creando una expresión este tipo, así que (–y2 y1) se usará para (Y) y (–x2 x1) para (X). Al reemplazar (Y) y (X) por sus formas de prefijo se producirá la expresión final para determinar si los dos puntos tienen una pendiente positiva:

```
(>  (/  (- y2 y1)  (- x2 x1))  0)
```

La forma más sencilla de evaluar una expresión numérica (así como de cualquier otra expresión) de CLIPS es evaluar la expresión en el indicador de comandos de nivel superior. Por ejemplo, al introducir (+ 2 2) en el indicador de CLIPS se produciría la siguiente salida:

```
CLIPS> (+ 2 2)↵
4
CLIPS>
```

La salida muestra la respuesta correcta, 4. En general, es posible introducir al nivel superior cualquier expresión de CLIPS que habrá de evaluarse. Muchas funciones, como la suma, tienen un **valor de devolución,** éste puede ser un entero, una variable, un símbolo, una cadena o incluso un valor de campo múltiple. Otras funciones, como los comandos *facts* y *agenda,* no tienen valor de devolución, en cuyo caso suelen tener lo que se llama un ***efecto lateral.*** El efecto lateral del comando *facts* consiste en incluir los hechos en la lista de hechos.

Las otras funciones aritméticas también funcionan en el nivel superior. La siguiente salida demuestra la evaluación de algunas otras expresiones:


```
CLIPS> (+ 2 3)↵
5
CLIPS> (- 2 3)↵
-1
CLIPS> (* 2 3)↵
6
CLIPS> (/ 2 3)↵
0.66666667
CLIPS>
```

Observe que la respuesta de la división probablemente muestre un error de redondeo en el último dígito, resultado que puede variar de máquina a máquina.

El valor de devolución de las funciones +, – y * es un entero si todos sus argumentos son enteros, si alguno es un número de punto flotante, entonces el valor de devolución también lo es. El primer argumento para la función / siempre se convierte en flotante, de modo que su valor de devolución es siempre un flotante. Por ejemplo:

```
CLIPS> (+ 2 3.0)↵
5.0
CLIPS> (+ 2.0 3)↵
5.0
CLIPS> (+ 2 3)↵
5
CLIPS>
```

Números variables de argumentos

La notación de prefijo permite representar diferentes números de argumentos de forma muy sencilla, por lo que muchas funciones de CLIPS aceptan un número variable de argumentos. Los argumentos de una expresión numérica pueden ser más de dos para las funciones +, – y *; en ellas se realiza la misma secuencia de cálculos aritméticos para más de dos argumentos. Los siguientes ejemplos, introducidos en el nivel superior, muestran cómo se usan tres argumentos; la evaluación va de izquierda a derecha.

```
CLIPS> (+ 2 3 4) ↵
9
CLIPS> (- 2 3 4) ↵
-5
CLIPS> (* 2 3 4) ↵
24
CLIPS> (/ 2 3 4) ↵
0. 16666667
CLIPS>
```

Conviene recordar de nuevo que la respuesta para la división puede variar un poco, dependiendo de la máquina que se utilice.

Expresiones de precedencia y anidación

Un hecho importante relativo a los cálculos con CLIPS y LISP es que no existe precedencia integrada a las operaciones aritméticas. En otros lenguajes de computadora, la multiplicación y la división tienen mayor jerarquía que la suma y la resta, y la computadora realiza primero las operaciones de más alto rango. En LISP y CLIPS todo se calcula simplemente de izquierda a derecha, y los paréntesis determinan la precedencia.

Pueden efectuarse cálculos combinados con notación de prefijo. Suponiendo como ejemplo que se va a evaluar la siguiente expresión infija :

```
2 + 3 * 4
```

En la evaluación acostumbrada se multiplica 3 por 4 y después se le suma dos al resultado. Sin embargo, en CLIPS debe escribirse la precedencia de forma explícita, la expresión podría evaluarse introduciendo en el nivel superior lo siguiente:

```
CLIPS> (+ 2 (* 3 4))↵
14
CLIPS>
```

En este regla se evalúa primero la expresión situada en el paréntesis interno; por tanto, se multiplica 3 por 4 y al resultado se le suma después 2. Si se desea evaluar (2 + 3) * 4, donde se hace primero la suma, la expresión de nivel superior sería:

```
CLIPS> (* (+ 2 3) 4)↵
20
CLIPS>
```

En general, las expresiones pueden incrustarse libremente dentro de otras, de esta manera es posible colocar una expresión dentro de un comando *assert*, como lo muestra el siguiente ejemplo:

```
CLIPS> (clear) ↵
CLIPS> (assert (answer (+ 2 2))) ↵
<Fact-0>
CLIPS> (facts) ↵
f-0     (answer 4)
For a total of 1 fact.
CLIPS>
```

Además, como los nombres de las funciones también son símbolos, pueden utilizarse como se haría con otros símbolos. Por ejemplo, puede usarse como campos en un hecho:

```
CLIPS> (clear)↵
CLIPS> (assert (expression 2 + 3 * 4))↵
<Fact-0>
CLIPS> (facts)↵
f-0     (expression 2 + 3 * 4)
For a total of 1 fact.
CLIPS>
```


Sin embargo, CLIPS utiliza los paréntesis como delimitadores, de modo que no es posible usarlos como se usarían otros símbolos. Para emplearlos en los hechos o como el argumento de una función, es necesario encerrarlos entre comillas para convertirlos en cadenas.

8.10 SUMA DE VALORES UTILIZANDO REGLAS

Consideremos el problema de sumar el área de un grupo de rectángulos como un sencillo ejemplo del uso de las funciones para realizar cálculos. Sus alturas y bases pueden especificarse con la siguiente deftemplate:

```
(deftemplate rectángulo (slot altura) (slot base))
```

y la suma de las áreas del rectángulo podrían especificarse usando un hecho ordenado como el siguiente:

```
(suma 20)
```

Un deffacts que contiene información de muestra es:

```
(deffacts información-inicial
   (rectángulo (altura 10) (base 6))
   (rectángulo (altura 7) (base 5))
   (rectángulo (altura 6) (base 8))
   (rectángulo (altura 2) (base 5))
   (suma 0))
```

Un intento inicial de producir una regla para sumar el rectángulo podría ser

```
(defrule sumar-rectángulos
   (rectángulo (altura ?altura) (base ?base))
   ?suma <- (suma ?total)
   =>
   (retract ?suma)
   (assert (suma (+ ?total (* ?altura ?base)))))
```

Sin embargo, esta regla producirá un lazo infinito, por que la retractación del hecho *suma* y su reafirmación producirá un lazo con un solo hecho *rectángulo*. Una solución para resolver este problema sería retractar el hecho *rectángulo* después de sumar su área al hecho *suma*, lo que evita que la regla se dispare por un mismo hecho *rectángulo* con un hecho *suma* diferente. Si necesita conservarse el hecho *rectángulo*, se necesita un método diferente, en el que para cada hecho *rectángulo* se crea un hecho temporal que contiene el área que se agrega a la suma, luego este hecho temporal puede retractarse para evitar un lazo infinito. El programa modificado es como sigue:

```
(defrule sumar-rectángulos
   (rectángulo (altura ?altura) (base ?base))
   =>
   (assert (agregar-a-suma (* ?altura ?base))))
```

```
(defrule sumar-áreas
   ?suma <- (suma ?total)
   ?área-nueva <- (agregar-a-suma ?área)
   =>
   (retract ?suma ?área nueva)
   (assert (suma (+ ?total ?área))))
```

La siguiente salida muestra cómo estas dos reglas interactúan para sumar las áreas de los rectángulos:

```
CLIPS> (unwatch all)↵
CLIPS> (watch rules)↵
CLIPS> (watch facts)↵
CLIPS> (watch activations)↵
CLIPS> (reset)↵
==> f-0     (hecho-inicial)
==> f-1     (rectángulo (altura 10) (base 6))
==> Activation 0       sumar-rectángulos: f-1
==> f-2     (rectángulo (altura 7) (base 5))
==> Activation 0       sumar-rectángulos: f-2
==> f-3     (rectángulo (altura 6) (base 8))
==> Activation 0       sumar-rectángulos: f-3
==> f-4     (rectángulo (altura 2) (base 5))
==> Activation 0       sumar-rectángulos: f-4
==> f-5     (sumar 0)
CLIPS> (run)↵
FIRE    1 sumar-rectángulos: f-4
==> f-6     (agregar-a-suma 10)
==> Activation 0       sumar-áreas: f-5, f-6
FIRE    2 sumar-áreas: f-5, f-6
<== f-5     (sumar 0)
<== f-6     (agregar-a-suma 10)
==> f-7     (sumar 10)
FIRE    3 sumar-rectángulos: f-3
==> f-8     (sumar 10)
==> Activation 0       sumar-áreas: f-7, f-8
FIRE    4 sumar-áreas: f-7, f-8
<== f-7     (sumar 0)
<== f-8     (agregar-a-suma 48)
==> f-9     (sumar 58)
FIRE    5 sumar-rectángulos: f-2
==> f-10    (agregar-a-suma 35)
==> Activation 0       sumar-áreas: f-9, f-10
FIRE    6 sumar-áreas: f-9, f-10
<== f-9     (sumar 58)
<== f-10    (agregar-a-suma 35)
==> f-11    (sumar 93)
FIRE    7 sumar-rectángulos: f-1
```



```
==> f-12    (agregar-a-suma 60)
==> Activation 0       sumar-áreas: f-11, f-12
FIRE    8 sumar-áreas: f-11, f-12
<== f-11    (sumar 93)
<== f-12    (agregar-a-suma 60)
==> f-13    (sumar 153)
CLIPS> (unwatch all)↵
CLIPS>
```

La regla *sumar-rectángulos* se activa cuatro veces cuando los hechos *rectángulo* se afirman como resultado del comando *reset*, cada vez que se dispara afirma un hecho que activa la regla *sumar-áreas*. Ésta suma el área al total actual y elimina el hecho *agregar-a-suma*. Debido a que la regla *sumar-rectángulos* no coincide con el patrón del hecho *suma*, no se reactiva cuando se afirma un nuevo hecho *suma*.

8.11 FUNCIÓN BIND

A menudo es útil almacenar un valor en una variable temporal para evitar calcularlo de nuevo, lo que es crucial cuando las funciones producen efectos secundarios. La función **bind** puede usarse para unir el valor de una variable con el de una expresión y su sintaxis es:

```
(bind <variable> <valor>)
```

La variable limitada, <variable>, usa la sintaxis de una variable de campo simple; el nuevo valor, <valor>, debe ser una expresión que evalúe un valor simple o de campo múltiple. Por ejemplo, la regla *sumar-áreas* podría imprimir la suma total y el área que se le agregó para cada rectángulo.

```
(defrule sumar-áreas
   ?suma <- (suma ?total)
   ?área-nueva <- (agregar-a-suma ?área)
   =>
   (retract ?suma ?área-nueva)
   (printout t "Sumar " ?área " a " ?total crlf)
   (printout t "Nueva suma es " (+ ?total ?área) crlf)
   (assert (suma (+ ?total ?área))))
```

Observe que la expresión (+ ?total ?área) se usa dos veces en el lado derecho de la regla, pero reemplazar las dos evaluaciones separadas con una función *bind* elimina los cálculos innecesarios. Escribiendo nuevamente la regla, utilizando ahora la función *bind* tendremos:

```
(defrule sumar-áreas
   ?suma <- (suma ?total)
   ?área-nueva <- (agregar-a-suma ?área)
   =>
   (retract ?suma ?área-nueva)
   (printout t "Sumar " ?área " a " ?total crlf)
   (bind ?nuevo-total (+ ?total ?área)
```

```
(printout t "Nueva suma es " ?nuevo-total crlf)
(assert (suma ?nuevo-total)))
```

Además de crear nuevas variables para usarlas en el lado derecho de una regla, la función *bind* también puede utilizarse para volver a unir el valor de una variable empleada en el lado izquierdo de una regla.

8.12 FUNCIONES DE ENTRADA/SALIDA

Función read

Los sistemas expertos a menudo necesitan que el usuario del programa les proporcione una entrada. CLIPS permite la lectura de información introducida mediante el teclado usando la función **read.** La sintaxis básica de esta función no requiere de argumentos; el siguiente ejemplo muestra la manera en que se usa la función *read* para la introducción de datos:

```
CLIPS> (clear)↵
CLIPS>
(defrule obtener-nombre
   =>
    (printout t "¿Cuál es tu nombre? ")
    (bind ?respuesta (read))
    (assert (nombre-usuario ?respuesta)))
CLIPS> (reset)↵
CLIPS> (run)↵
¿Cuál es tu nombre? Gabriel↵
CLIPS> (facts)↵
f-0       (hecho-inicial)
f-1       (nombre-usuario Gabriel)
For a total of 2 facts.
CLIPS>
```

Observe que la función *read* necesita un enter antes de leer la señal (token) introducida. Esta función sólo puede usarse para introducir un campo a la vez, todos los demás caracteres que se introducen entre el primer campo y el enter se descartan. Por ejemplo, si la regla obtener-nombre trata de leer el nombre y el apellido con la siguiente entrada:

```
Gabriel Rojas ↵
```

sólo se leerá el primer campo, Gabriel; para leer toda la entrada, ambos campos deben encerrarse entre comillas. Desde luego, una vez que la entrada se encierra entre comillas, es un campo literal simple, mientras que los campos *individuales Gabriel* y *Rojas* no se pueden introducir con facilidad.

La función read permite la introducción de campos que no son símbolos, cadenas, enteros ni puntos flotantes, como los paréntesis; dichos campos se colocan entre comillas y se tratan como cadenas. El siguiente diálogo de línea de comando demuestra esta capacidad:


```
CLIPS> (read)↵
(↵
"("
CLIPS>
```

Función open

Además de la entrada de teclado y la salida a terminal, CLIPS también puede leer y escribir archivos. Antes de introducir un archivo para lectura o escritura, debe abrirse con la función **open.** El número de archivos que pueden abrirse de manera simultánea depende del sistema operativo y el hardware particulares.

La sintaxis de la función *open* es

```
(open <nombre-archivo> <ID-archivo> [<acceso-archivo>])
```

Por ejemplo:

```
(open "entrada.dat" datos "r")
```

El primer argumento de *open,* <nombre-archivo>, es una cadena que representa el nombre del archivo en la computadora, en el ejemplo se usa el nombre de "entrada.dat". El nombre del archivo también puede incluir información de ruta (el directorio donde se encuentra el archivo); aunque las especificaciones de ruta suelen variar de un sistema operativo a otro, de modo que habrá que estar un tanto familiarizado con el sistema operativo de la computadora para especificar una ruta.

El segundo argumento, <ID-archivo>, es el **nombre lógico** que CLIPS asocia con el archivo, y que es un nombre global mediante el cual CLIPS puede ingresar el archivo desde cualquier regla o del indicador de nivel superior; aunque el nombre lógico puede ser igual al nombre del archivo, es conveniente emplear un nombre diferente para evitar confusiones. Para el ejemplo que se muestra, se usó el nombre lógico *datos* para hacer referencia al archivo "entrada.dat", pero pueden emplearse otros nombres significativos, como entrada o datos de archivo.

Una ventaja del uso de un nombre lógico es que se puede sustituir con facilidad un nombre de archivo diferente sin hacer cambios importantes al programa. Puesto que el nombre del archivo sólo se usa en la función *open* y más adelante se le consulta mediante su nombre lógico, sólo se necesita cambiar la función *open* para leer un archivo diferente.

El tercer argumento, <acceso-archivo>, es una cadena que representa uno de cuatro posibles métodos de acceso de archivo, que se enumeran en la tabla 8.2.

Modo	Acción
"r"	Acceso de solo lectura
"w"	Acceso de solo escritura
"r+"	Acceso de lectura y escritura
"a"	Acceso de solo añadir

Tabla 8.2 Métodos de acceso a un archivo

Si <acceso-archivo> no está incluido como argumento, se utilizará "r" como valor predeterminado. Algunos métodos de acceso no son significativos para ciertos sistemas operativos, pues aunque la mayor parte de éstos soportan acceso de lectura (para entrada de datos) y escritura (para salida de datos), los accesos de lectura y escritura y añadir (para añadir una salida de datos al final de una archivo) quizá no siempre estén disponibles.

Es importante recordar que las consecuencias de abrir un archivo pueden variar de una máquina a otra. Por ejemplo, los sistemas operativos que no soportan varias versiones de

archivo, como MS-DOS en una PC o en UNIX, reemplazarán el archivo existente al abrirlo con el acceso de escritura. Por el contrario VMS en un VAX creará una versión nueva del archivo si éste ya existe y se abre con el acceso de escritura.

Open actúa como una función de afirmación (descrita en la sección 9.4), que devuelve el símbolo *TRUE* si el archivo se abre con éxito; de lo contrario, devuelve el símbolo *FALSE*. El valor de devolución puede usarse para revisar errores; por ejemplo, si el usuario proporciona el nombre de un archivo que no existe y se intenta abrirlo mediante el acceso de lectura, la función *open* devolverá el símbolo *FALSE*. La regla que abre el archivo podría detectar esto y realizar las acciones apropiadas para pedir al usuario otro nombre de archivo.

Función close

Un archivo debe cerrarse cuando ya no se necesita utilizarlo, de otra manera no hay garantía de que se guarde la información escrita en él. Además, cuánto más tiempo esté abierto un archivo, mayor posibilidad habrá de que una pérdida de energía u otra disfunción eviten que la información se guarde.

La forma general de la función **close** es:

```
(close [<ID-archivo>])
```

donde el argumento opcional <ID-archivo> especifica el nombre lógico del archivo que habrá de cerrarse. Si no se especifica <ID-archivo>, se cerrarán todos los archivos abiertos. Por ejemplo:

```
(close datos)
```

cerrará el archivo conocido por CLIPS con el nombre lógico *datos;* las instrucciones:

```
(close entrada)
(close salida)
```

cerrarán los archivos asociados con los nombres lógicos *entrada* y *salida*. Observe que es necesario proporcionar instrucciones separadas para cerrar archivos específicos.

Es importante recordar que cada archivo abierto debe cerrarse con la función *close*, sobre todo por que si no se emite la orden de cerrar un archivo, lo datos escritos en él pueden perderse. CLIPS no le pedirá cerrar un archivo abierto, la única medida de protección de CLIPS para cerrar archivos que se dejan abiertos de manera inadvertida es que todos los archivos abiertos se cerrarán cuando se emita un comando *exit*.

Lectura y escritura en un archivo

En los ejemplos precedentes, todas las entradas se han introducido mediante el teclado y todas las salidas se han enviado a la terminal, pero el uso de nombres lógicos permite la entrada desde otras fuentes y la salida a éstas. En el capítulo 7, la función *printout* utiliza el nombre lógico t para enviar la salida a la pantalla, pero pueden usarse otros nombres lógicos con esta función para enviar la salida a otros destinos.

Cuando se usa como nombre lógico para una función de salida, el nombre lógico t escribe la salida para el dispositivo normal de salida, a menudo la terminal. Del mismo modo, cuando se usa como nombre lógico para una función de entrada, el nombre t lee la entrada del **dispositivo normal de entrada,** que por lo general es el teclado.

En el siguiente ejemplo se ilustra el uso de nombres lógicos para escribir en un archivo:


```
CLIPS> (open "ejemplo.dat" ejemplo "w")↵
TRUE
CLIPS> (printout ejemplo "verde" crlf)↵
CLIPS> (printout ejemplo 7 crlf)↵
CLIPS> (close ejemplo)↵
TRUE
CLIPS>
```

En primer lugar, el archivo "ejemplo.dat" se abre con acceso de escritura, de modo que es posible escribir valores en él. La función open ahora tiene asociado el nombre lógico ejemplo con el archivo "ejemplo.dat"; luego los valores *verde* y 7 se escriben en el archivo "ejemplo.dat" a través del uso del nombre lógico *ejemplo* como primer argumento para la función *printout;* una vez que se han escrito los valores en el archivo, éste se cierra usando el comando *close*.

Ahora que se han escrito los valores en el archivo, es posible abrirlo con acceso de lectura y recuperar los valores mediante el uso de la función *read,* cuyo formato general es:

```
(read [<nombre-lógico>])
```

La función *read* establece como opción predeterminada la lectura del dispositivo normal de entrada, t, si no se proporcionan argumentos; el ejemplo anterior de la función *read* hace uso de este nombre lógico predeterminado. El siguiente ejemplo muestra la manera en que puede usarse un nombre lógico con la función read para recuperar los valores del archivo "ejemplo.dat":

```
CLIPS> (open "ejemplo.dat" ejemplo "r") ↵
TRUE
CLIPS> (read ejemplo) ↵
verde
CLIPS> (read ejemplo) ↵
7
CLIPS> (read ejemplo) ↵
EOF
CLIPS> (close ejemplo) ↵
TRUE
CLIPS>
```

Primero se abre el archivo "ejemplo.dat", pero esta vez con el acceso de lectura. (Observe que no tuvo que usarse la opción "r" para abrir el archivo "ejemplo.dat" con la segunda llamada a la función *open*, porque ésta es la opción predeterminada). Una vez que se abre el archivo, se usa la función *read* para recuperar los valores *verde* y 7 del archivo asociado con el nombre lógico *ejemplo*. Observe que la tercera llamada de la función *read* devuelve el símbolo **EOF** (End Of File, final del archivo), CLIPS devuelve este valor para funciones de entrada cuando se intenta leer más allá del final del archivo. Al verificar el valor de devolución de la función *read* (u otra entrada) es posible determinar el momento en que ya no hay más datos en el archivo.

Función format

A veces es deseable dar formato a la salida de un programa de CLIPS, tal como cuando se organizan datos en tablas. Aunque la función *printout* es útil, hay una función diseñada de

manera específica para el formato, llamada *format,* que proporciona una amplia gama de estilos. La sintaxis de la función *format* es

```
(format <nombre-lógico> <cadena-control>
        <parámetros>*)
```

La función *format* tiene varias partes: el <nombre-lógico> es el nombre lógico al que se envía la salida; el dispositivo normal de salida predeterminado puede especificarse con el nombre lógico t; a continuación viene una **cadena de control,** que debe estar entre comillas y consta de **marcas de formato,** que indican la manera en que deben imprimirse los parámetros para la función *format;* luego sigue una lista de parámetros; el número de marcas de formato de la cadena de control determinará la cantidad de parámetros que deben especificarse, y éstos pueden ser valores constantes o expresiones. El valor de devolución de la función *format* es la cadena formateada. Si se usa el nombre lógico nil con el comando format, entonces no se imprime salida (para la terminal o un archivo), pero aún se devuelve la cadena formateada.

La función *format* es muy útil para desplegar columnas de datos, un ejemplo se muestra mediante el siguiente diálogo, que crea una cadena formateada que contiene el nombre de una persona (que reserva 15 espacios para el nombre) seguido por su edad. Observe como se alinean en columnas los nombres y las edades.

```
CLIPS> (format nil "Nombre: %-15s Edad: %3d"
                   "Gerardo Rojo" 35)↵
"Nombre: Gerardo Rojo       Edad:  35"
CLIPS> (format nil "Nombre: %-15s Edad: %3d"
                   "Rafael Muñoz" 32)↵
"Nombre: Rafael Muñoz       Edad:  32"
CLIPS>
```

Las marcas de formato siempre empiezan con un signo "%". Las cadenas ordinarias como "Nombre:" también se pueden poner en la cadena de control y se imprimirán en la salida. Algunas marcas de formato no forman parámetros, por ejemplo, "%n" se usa para poner un retorno de carro/avance de línea/enter de carro en la salida, de manera muy semejante a como se usó el símbolo *crlf* con el comando *printout.*

En este ejemplo, el indicador de formato "%-15s" se usa para imprimir el nombre en una columna que tiene 15 caracteres de ancho. El signo – indica que la salida debe dejarse justificada y el carácter s indica que se imprimirá una cadena o un símbolo. El indicador de formato "%3d" indica que se imprimirá un número alineado a la derecha como un valor entero en una columna de tres caracteres. Si se proporciona el valor 5.25 como parámetro para el indicador de formato, entonces tendría que imprimirse el número 5, porque en un formato de enteros no se permite la parte fraccionaria de un número.

Tome en cuenta que cuando se emplea la función *format* en el lado derecho de una regla, a menudo se ignora su valor de devolución. En estos casos, el nombre lógico se asociará con un archivo o con t para enviar la salida a la pantalla. La especificación general de un indicador de formato es:

```
%-M.Nx
```


donde "–" es opcional y significa **alinear a la izquierda,** mientras que la opción predeterminada es **alinear a la derecha,** éstas se utilizan cuando la cantidad de espacio proporcionado para imprimir un número excede lo necesario para imprimirlo. Si se presenta esta situación, la alineación del número a la izquierda provoca que se imprima del lado izquierdo del espacio libre mientras que la alineación a la derecha provoca que lo haga en el lado derecho.

M especifica el campo de la anchura de las columnas, la salida tendrá al menos M caracteres. Los espacios suelen usarse para rellenar la salida y completar las columnas. Si la salida excede M columnas, la función format extenderá el campo todo lo necesario.

N es una especificación opcional del número de dígitos que se imprimirá después del punto decimal. La opción predeterminada consta de seis dígitos después del punto decimal.

x precisa la especificación del formato de despliegue. En la tabla 8.3 se proporcionan las especificaciones de formato de despliegue.

Carácter	Significado
e	Entero
f	Con punto decimal
e	Exponencial (en formato de potencia de diez)
g	General (numérico); el despliegue en cualquier formato es más corto
o	Octal; número sin signo (especificador *N* no aplicable)
x	Hexadecimal; número sin signo (especificador *N* no aplicable)
s	Cadena; a las cadenas entrecomilladas se les quitarán las comillas
n	Retorno de carro /Avance de línea enter
%	El propio carácter "%"

Tabla 8.3 Especificaciones del formato de despliegue

Función readline

La función **readline** puede utilizarse para leer una línea completa de entrada, su sintaxis es:

```
(readline [<nombre-lógico>])
```

Como en el caso de la función *read,* el nombre lógico es opcional, pero si no se proporciona un nombre lógico o se usa el nombre lógico t, la entrada se leerá a partir del dispositivo normal de entrada. La función *readline* devolverá como cadena la siguiente línea de entrada desde la fuente origen de entrada asociado con el nombre lógico hasta el enter. La función *readline* devolverá el símbolo *EOF* si se alcanza el final del archivo cuando el nombre lógico que emplea está asociado con un archivo. El siguiente diálogo ilustra el uso de la función *readline:*

```
CLIPS> (clear) ↵
CLIPS>
(defrule obtener-nombre
```

```
   =>
   (printout t "¿Cuál es tu nombre? ")
   (bind ?respuesta (readline))
   (assert (nombre-usuario ?respuesta)))
CLIPS> (reset)↵
CLIPS> (run)↵
¿Cuál es tu nombre? Gabriel Rojo↵
CLIPS> (facts)↵
f-0     (hecho-inicial)
f-1     (nombre-usuario "Gabriel Rojo")
For a total of 2 facts.
CLIPS>
```

En este ejemplo, el nombre "Gabriel Rojo" se almacenó en el hecho *nombre-usuario* como un campo simple, por ello no es posible recuperar directamente el nombre y apellido a partir del campo utilizando variables de patrón. Con el uso de la función **explode$**, que acepta un argumento de cadena simple y lo convierte a valor de campo múltiple, es posible convertir la cadena devuelta por la función *readline* en un valor de campo múltiple, que se afirmará como una serie de campos en el hecho *nombre-usuario*. El siguiente diálogo ilustra el uso de *readline* junto con la función *explode$:*

```
CLIPS>
(defrule obtener-nombre
   =>
   (printout t "¿Cuál es tu nombre? ")
   (bind ?respuesta (explode$ (readline)))
   (assert (nombre-usuario ?respuesta)))↵
CLIPS> (reset)↵
CLIPS> (run)↵
¿Cuál es tu nombre? Gabriel Rojo
CLIPS> (facts) ↵
f-0     (hecho-inicial)
f-1     (nombre-usuario Gabriel Rojo)
For a total of 2 facts.
CLIPS>
```

En el apéndice E se encuentra una lista de otras funciones útiles para crear y manipular cadenas y valores de campo múltiple.

8.13 RESUMEN

En este capítulo se introdujo el concepto de variables, que se usan para recuperar información de los hechos y para restringir los valores de ranura cuando se comparan patrones del lado izquierdo de una regla. Las variables pueden almacenar las direcciones de hecho de los patrones en el lado izquierdo de una regla, de modo que el hecho enlazado con un patrón puede retractarse en el lado derecho de la regla. Los comodines de campo simple se pueden utilizar en vez de variables cuando el campo que se habrá de comparar puede ser cualquier cosa y su valor no se necesitará más tarde **en el lado izquierdo o lado derecho** de la regla. Las variables y comodines de campo múltiple permiten la comparación de más de un campo en un patrón.

Las restricciones de campo permiten la negación y combinación de más de una restricción para un campo dado. La restricción de campo *not* se utiliza para evitar la comparación con ciertos valores, la restricción de campo *and* se usa para garantizar que toda una serie de condiciones de comparación es verdadera y la restricción de campo *or* se utiliza para garantizar que cuando menos una de las de condiciones de comparación que conforman la serie es verdadera.

Las funciones se introducen en el lazo del comando de nivel superior de CLIPS o se usan en el lado izquierdo o lado derecho de una regla. Muchas funciones, como las aritméticas, pueden tener un número variable de argumentos. Las llamadas a funciones pueden anidarse dentro de otras. El comando bind permite enlazar variables en el lado derecho de una regla.

CLIPS proporciona varias funciones de E/S: las funciones *open* y *close* se usan para abrir y cerrar archivos, los archivos abiertos se asocian con un nombre lógico, los nombres lógicos se usan en la mayor parte de las funciones que efectúan entradas y salidas para más de un tipo de dispositivo físico. Las funciones *printout* y *read* usan los nombres lógicos, la función *printout* puede producirse en la terminal y en los archivos, la función *read* puede introducirse con el teclado y mediante archivos. Las funciones *format* y *readline* también aceptan nombres lógicos, la primera permite tener mayor control sobre el aspecto de la salida, la segunda puede emplearse para leer una línea completa de datos. La función *explode$* convierte una cadena en un valor de campo múltiple.

PROBLEMAS

8.1 Dadas las siguientes deftemplates para hechos que describen un árbol genealógico:

```
(deftemplate padre-de (slot padre) (slot hijo))
(deftemplate madre-de (slot madre) (slot hijo))
(deftemplate masculino (slot persona))
(deftemplate femenino (slot persona))
(deftemplate esposa-de (slot esposa) (slot esposo))
(deftemplate esposo-de (slot esposo) (slot esposa))
```

escriba reglas que infieran las siguientes relaciones, describiendo las deftemplates que usa para resolver el problema.

(a) Tío, tía
(b) Primo
(c) Abuelos
(d) Abuelo, abuela
(e) Hermana, hermano
(f) Antepasado

8.2 Una planta industrial tiene diez sensores con números de identificación del 1 a 10. Cada sensor tiene un estado "bueno" o "malo". Construya una deftemplate para representar los sensores y escriba una o más reglas que imprimirán un mensaje de advertencia si tres o más sensores están en mal estado. Pruebe sus reglas con los sensores 3 y 5 malos; los sensores 2, 8 y 9 malos; y los sensores 1, 3, 5 y 10 malos. ¿Qué debe hacerse para evitar que el mensaje de advertencia se despliegue varias veces?

8.3 Construya un programa CLIPS basado en las reglas SI-ENTONCES desarrollado en el problema 3.5 de la página 158. El programa debe solicitar el método de pago del viajero y el interés de viaje y debe producir los posibles recorridos con base en estas dos entradas.

8.4 En una pila, el primer valor agregado es el último valor eliminado y el último valor agregado es el primer valor eliminado. Si una fila de espera funciona de manera opuesta (el primer valor agregado es el primero que se elimina y el último valor agregado es el último que se elimina), escriba las reglas que agregarán y eliminarán valores en ella. Suponga que sólo existe una fila de espera.

8.5 Escriba una o más reglas que generarán todas las permutaciones de un hecho base e imprímalas. Por ejemplo, el hecho:

```
(hecho-base rojo verde azul)
```

debe generar la salida

```
Permutación es (rojo verde azul)
Permutación es (rojo azul verde)
Permutación es (verde rojo azul)
Permutación es (verde azul rojo)
Permutación es (azul rojo verde)
Permutación es (azul verde rojo)
```

8.6 Dada una series de hechos que describen figuras utilizando las siguientes deftemplates:

```
(deftemplate cuadrado
   (slot id) (slot longitud-lado))
(deftemplate rectángulo
   (slot id) (slot base) (slot altura))
(deftemplate círculo
   (slot id) (slot radio))
```

escriba las reglas que calcularán la suma de

(a) las áreas de las figuras.
(b) los perímetros de las figuras.

Pruebe las salidas de las reglas con los siguientes deffacts:

```
(deffacts prueba-8-8
   (cuadrado (id A) (longitud-lado 3))
   (cuadrado (id B) (longitud-lado 5))
   (rectángulo (id C) (base 5) (altura 7))
   (círculo (id D) (radio 2))
   (círculo (id E) (radio 6)))
```


8.7 Dados el nombre, el color de ojos, el color de cabello y la nacionalidad de una persona de un grupo utilizando la siguiente deftemplate:

```
(deftemplate persona (slot nombre)
                     (slot color-ojos)
                     (slot color-cabello)
                     (slot nacionalidad))
```

escriba la regla que identificará;

(a) cualquier persona con ojos azules o verdes que tiene cabello castaño y es francesa.
(b) cualquier persona que no tiene ojos azules ni cabello negro y no tiene el mismo color de ojos y cabello.
(c) dos personas, la primera tiene ojos castaños o azules, no tiene cabello rubio y es alemana; la segunda tiene ojos verdes, el mismo color de cabello que la primera persona y es de cualquier nacionalidad. Los ojos de la segunda persona podrían ser café si el color de cabello de la primera persona es castaño.

8.8 Convierta las siguientes expresiones de infijo a expresiones de prefijo.

```
(a) (3 + 4) * (5 + 6) + 7
(b) (5 * (5 + 6 + 7)) - ((3 * (4 / 9) + 2) / 8)
(c) 6 - 9 * 8 / 3 + 4 - (8 - 2 - 3) * 6 / 7
```

8.9 Escriba las reglas que llevarán a una máquina de estado finito de su estado actual a su siguiente estado dado un hecho de la forma

```
(entrada <valor>)
```

La máquina de estado y sus arcos deben representarse como hechos. El hecho de entrada debe retractarse cuando la máquina pase a su siguiente estado. Pruebe sus reglas y representaciones de hecho en las máquinas de estado finito que se muestran en las figuras 3.5 y 3.6 de las páginas 101 y 102.

8.10 Considerando la siguiente información acerca de un equipo de béisbol: Andrés siente antipatía por el cátcher; la hermana de Eduardo está comprometida con el segunda base; el jardinero central es más alto que el jardinero derecho; Humberto y el tercera base viven en el mismo edificio; Pablo y Alejandro le ganaron $20 cada uno al pitcher en los naipes; Eduardo y los jardineros juegan póker durante su tiempo libre; la esposa del pitcher es la hermana del tercera base; toda la batería (el pitcher y el catcher) y los jugadores del cuadro, excepto Alejandro, Humberto y Andrés, son más bajos que Pedro; Pablo, Andrés y el parador en corto perdieron $50 cada uno en el hipódromo; Pablo, Humberto, Beto y el cátcher vapulearon en el billar al segunda base; Pedro se está divorciando; el cátcher y el tercera base tienen dos hijos. Eduardo, Pablo, Julio y los jardineros derecho y central son solteros, los demás son casados. El parador en corto, el tercera base y Beto perdieron $100 apostando en una pelea; uno de los jardineros es Miguel o Andrés; Julio es más alto que Beto; Miguel es más bajo que Beto y cada uno de ellos es más pesado que el tercera base; Pedro, el cátcher, y el tercera base son zurdos; Eduardo, Pedro y el parador en corto fueron juntos a la secundaria. Escriba un programa CLIPS para determinan quién juega cada posición.

8.11 Convierta el árbol de decisión que se muestra en la figura 3.3 de la página 99 en una serie de reglas de CLIPS. Elabore patrones para comparar hechos utilizando la siguiente deftemplate:

```
(deftemplate pregunta
   (slot cadena-consulta)
   (slot respuesta))
```

Por ejemplo, si la respuesta a la pregunta representada por el nodo raíz del árbol fue "no", el hecho que representa esta información sería

```
(pregunta (cadena-consulta "¿Es muy grande?")
          (respuesta no))
```

Utilice las funciones *printout* y *read* en el lado derecho de las reglas para formular las preguntas que se muestran en la figura y afirmar hechos de *pregunta* con la respuesta del usuario.

8.12 Escriba un programa CLIPS que agregará dos números binarios sin usar ninguna función aritmética. Represente los números binarios utilizando la siguiente deftemplate:

```
(deftemplate #-binario
   (multislot nombre)
   (multislot dígitos))
```

Dado un hecho que indica cuál par de números binarios conocidos habrá de agregarse, el programa debe crear un nuevo número binario conocido que contenga la suma. Por ejemplo, los hechos:

```
(#-binario (nombre A) (dígitos 1 0 1 1 1))
(#-binario (nombre B) (dígitos 1 1 1 0))
(agregar-#-binarios (nombre-1 A) (nombre-2 A))
```

deben hacer que se agregue el siguiente hecho a la lista de hechos.

```
(#-binario (nombre ( A + B )) (dígitos 1 0 0 1 0 1))
```

8.13 Escriba un programa CLIPS que solicite los tipos de sangre de un paciente en caso de que haya necesidad de una transfusión sanguínea y un donador. Luego el programa debe determinar si la transfusión debe llevarse a cabo con base en los tipos de sangre. La sangre tipo O sólo puede transfundirse con sangre tipo O, la sangre tipo A puede transfundirse con sangre tipo A o tipo O, la sangre tipo B puede transfundirse con tipo B u O y la sangre tipo AB puede transfundirse con tipo AB, A, B u O.

8.14 Escriba un programa CLIPS que proporcione información sobre la manera en que puede cocinarse un corte de res específico o los cortes de res que pueden cocinarse de una manera específica. El programa primero debe indicar si se seleccionará un corte de res o un método de preparación y luego debe señalar la selección apropiada. Utilice las siguientes directrices para determinar los cortes o métodos de preparación apropiados: el filete de lomo debe cocerse a fuego lento o asarse; el bistec de solomillo debe asarse a la parrilla, asarse a la sartén o freírse a la sartén; el filete de espinazo debe asarse a la parrilla, asarse a la sartén o freírse a la sartén; las costillas deben asarse; la carne molida debe asarse, asarse a la parrilla, asarse al sartén o freírse al sartén; el filete de espaldilla debe cocerse a fuego lento; y el bistec de rueda debe cocerse a fuego lento.

8.15 Modifique el programa desarrollado en el problema 7.14 de la página 363, de modo que la entrada se determine al formular a los usuarios una serie de preguntas respecto a las características necesarias de un arbusto. La salida del programa debe ser la misma.

8.16 Las bacterias pueden clasificarse de acuerdo con sus características, incluyendo sus formas básicas (cocos, bacilos, espiroquetas o filamentosas), los resultados de un estudio de laboratorio con colorante (positivo, negativo o ninguno) y si requieren oxígeno para sobrevivir (aeróbicas o anaeróbicas). Escriba un programa que identifique el tipo bacteriano con base en la información que contiene la siguiente tabla. El programa debe preguntar la forma, el resultado del colorante y las necesidades de oxígeno de la bacteria, pero el usuario debe tener la posibilidad de especificar que cualquiera ellas es desconocida; La salida deben ser todos los tipos de bacterias posibles, con base en las respuestas proporcionadas.

Tipo	Forma	Colorante	Necesidad de oxígeno
Actinomiceto	bacilo o filamentosa	positivo	aeróbica
Cocácea	esférica	positivo	aeróbica y anaeróbica
Corineforme	bacilo	positivo	aeróbica
Formadora de endósporas	bacilo	positivo o negativo	aeróbica o anaeróbica
Entérica	bacilo	negativo	aeróbica
Deslizante	bacilo	negativo	aeróbica
Micobacteria	coco	ninguno	aeróbica
Micoplasma	coco	ninguno	aeróbica
Pseudomonada	bacilo	negativo	aeróbica
Ricketsia	coco o bacilo	negativo	aeróbica
Envainada	filamentosa	negativo	aeróbico
Espirilo	espiroqueta	negativo	aeróbico
Espiroqueta	espiroqueta	negativo	anaeróbica
Vibrión	bacilo	negativo	aeróbica

8.17 Electrónica Acme construye un dispositivo llamado Tingamabob 2000, dispositivo que está disponible en cinco modelos diferentes que se distinguen por el chasis. Cada chasis proporciona varios paneles para algunos accesorios opcionales y es capaz de generar cierta cantidad de energía. En la siguiente tabla se resumen los atributos de cada chasis:

Chasis	Páneles para accesorios	Energía proporcionada	Precio ($)
C100	1	4	2000.00
C200	2	5	2500.00
C300	3	7	3000.00
C400	2	8	3000.00
C500	4	9	3500.00

Cada accesorio que puede instalarse en el chasis requiere cierta cantidad de energía para funcionar, en la siguiente tabla se resumen los atributos de cada uno:

Aparato	Energía empleada	Precio ($)
Zaptron	2	100.00
Yatmizer	6	800.00
Fenerador	1	300.00
Malcifer	3	200.00
Escudo-Zeta	4	150.00
Warnosincronizador	2	50.00
Dinoseparador	3	400.00

Dando como hechos de entrada los que representan el chasis y cualquier accesorio que ha sido seleccionado, escriba un programa que genere hechos que representen el número de aparatos empleados, la cantidad total de energía requerida por el aparato y el precio total del chasis y todos los aparatos seleccionados.

8.18 Utilizando la tabla de gemas del problema 7.13 de la página 363, escriba reglas para identificar las siguientes gemas: diamante, corindón, crisoberilio, espilena, cuarzo y turmalina. Incluya reglas para preguntar al usuario la dureza, la densidad y el color de la gema. Se consultará al usuario hasta que responda con uno de los colores enumerados en la tabla.

BIBLIOGRAFÍA

(Firebaug 88). Morris W. Firebaugh, *Artificial Intelligence: A. Knowledge-Based Approach*, Boyd & Fraser Publishing Co., pp. 224-226, 1988.

CAPÍTULO 9

Comparación avanzada de patrones

9.1 INTRODUCCIÓN

En este capítulo se comentan varias técnicas para controlar la ejecución de reglas, ya que se utilizan grupos de reglas para elaborar programas más complejos y con más opciones que los que se han analizado en los capítulos anteriores; se muestran las técnicas para introducir, comparar los valores y generar lazos, además de las técnicas para especificar el conocimiento de control utilizando reglas. Aquí también se explican otros tipos de elementos condicionales aparte del patrón EC, éstos permiten que una sola realice la función de varias reglas y proporcionan la posibilidad de comparar con la ausencia de hechos. Por último, se presentan algunos comandos útiles en la práctica.

9.2 EL JUEGO DE LOS PALILLOS

Para demostrar varias técnicas de control en un lenguaje basado en reglas, en este capítulo se utilizará un sencillo juego para dos jugadores llamado Palillos. El objeto del juego es no verse forzado a tomar el último palillo de un cúmulo; cada jugador debe tomar 1, 2 o 3 palillos por turno. El truco (o la heurística) para ganar en este juego radica en darse cuenta de que puede obligar a su oponente a perder si es su turno y quedan 2, 3 o 4 palillos. Un jugador al que le restan 5 palillos cuando le toca mover, habrá perdido, para obligar al otro jugador a quedarse con cinco palillos, siempre deje cinco palillos más algún múltiplo de 4 al final de su turno, es decir, al final de su turno, haga que la pila quede con un número de palillos que sea igual a 5, 9, 13, etcétera. Si usted mueve primero y el cúmulo queda con uno de los números "perdedores", no podrá ganar a menos que su oponente cometa un error, pero si mueve primero y la pila no tiene un número "perdedor" de palillos, entonces siempre puede ganar.

9.3 TÉCNICAS DE ALIMENTACIÓN DE DATOS

Antes de empezar a jugar Palillos, el programa debe determinar cierta información; en primer lugar, como el programa jugará contra un oponente humano debe establecerse quién hará el primer movimiento; también, debe determinarse el tamaño inicial del cúmulo. Esta información podría colocarse en una construcción deffacts, sin embargo, es muy sencillo solicitar al oponente del programa que la introduzca con el teclado. En el siguiente ejemplo se muestra cómo se utiliza la función *read* para introducir los datos:

```
(deffacts fase-inicial
   (phase elegir-jugador))

(defrule jugador-selecciona
   (phase elegir-jugador)
   =>
   (printout t "¿Quién mueve primero (Computadora: c "
           "Humano: h)? ")
   (assert (jugador-selecciona (read))))

(defrule buena-elección-jugador
   ?phase <- (phase elegir-jugador)
   ?elección <- (jugador-selecciona ?jugador&c | h)
   =>
   (retract ?phase ?elección)
   (assert (jugador-mueve ?jugador)))
```

Ambas reglas emplean el patrón (phase elegir-jugador) para indicar su aplicabilidad sólo cuando un hecho específico se encuentra en la lista de hechos, a éste se le llama ***patrón de control*** porque se utiliza de manera específica para controlar cuándo es aplicable la regla. Un **hecho de control** se usa para disparar un patrón de control, y como el patrón de control para estas reglas sólo contiene campos literales, el hecho de control debe coincidir exactamente con el patrón; en este caso, el hecho de control que dispara estas reglas debe ser el hecho (phase elegir-jugador). Este hecho de control será útil para corregir errores cuando la entrada recibida por la función read no coincida con los valores esperados de "c" o "h" para "computadora" y "humano". Al retractar y reafirmar el hecho de control, puede dispararse nuevamente la regla *jugador-selecciona*.

La siguiente salida muestra como funcionan las reglas *jugador-selecciona* y *buena-elección-jugador* para determinar quién debe mover primero:

```
CLIPS> (unwatch all)↵
CLIPS> (watch facts)↵
CLIPS> (reset)↵
==> f-0     (hecho-inicial)
==> f-1     (phase elegir-jugador)
CLIPS> (run)↵
¿Quién mueve primero (Computadora: c Humano: h)? c↵
==> f-2     (jugador-selecciona c)
<== f-1     (phase elegir jugador)
<== f-2     (jugador-selecciona c)
==> f-3     (jugador-mueve c)
CLIPS>
```

Estas reglas funcionan de manera apropiada si se introduce la respuesta correcta de *c* o *h*, pero si se introduce una respuesta incorrecta no se realiza la verificación de errores. Existen muchas situaciones en las que debe repetirse una solicitud de entrada para corregir entradas defectuosas. En el siguiente ejemplo se muestra una manera de programar un lazo de solicitud de entrada. La siguiente regla, utilizada en conjunto con las reglas *jugador-selecciona* y *buena-elección-jugador*, proporciona una verificación de errores.


```
(defrule mala-elección-jugador
   ?phase <- (phase elegir-jugador)
   ?elección <- (jugador-selecciona ?jugador&~c&~h)
   =>
   (retract ?phase ?elección)
   (assert (phase elegir-jugador))
   (printout t "Elegir c o h." crlf))
```

Observemos una vez más que el uso del patrón de control (phase elegir–jugador) proporciona el control básico para el lazo de entrada y evita además que este grupo de reglas se dispare durante otras fases de la ejecución del programa.

Note cómo las dos reglas, *jugador-selecciona* y *mala-elección-jugador*, funcionan juntas porque cada regla proporciona los hechos necesarios para activar a la otra. Si la respuesta del jugador es incorrecta, la regla *mala-elección-jugador* retractará el hecho de control (phase elegir-jugador) y lo reafirmará, haciendo que se reactive la regla *jugador-selecciona*.

9.4 FUNCIONES DE PREDICADO

Una **función de predicado** se define como cualquier función que devuelve el símbolo *TRUE* o el símbolo *FALSE*. En realidad, cuando se trabaja con la lógica de predicado, CLIPS trata cualquier valor diferente al símbolo FALSE como TRUE; también puede considerarse que una función de predicado posee un valor de devolución boleano. Las funciones de predicado pueden ser **funciones predefinidas** o **definidas por el usuario**; las funciones predefinidas son aquellas que CLIPS proporciona mientras las **funciones externas,** o definidas por el usuario, son funciones diferentes a las predefinidas que se escriben en C u otro lenguaje y se vinculan con CLIPS. El apéndice E contiene una lista de las funciones de predicado predefinidas de CLIPS para realizar operaciones boleanas lógicas, comparar valores y pruebas para un tipo específico. Algunas de estas funciones se muestran en el siguiente diálogo:

```
CLIPS>  (and (> 4 3)  (> 4 5))↵
FALSE
CLIPS>  (or  (> 4 3)  (>  4 5)↵
TRUE
CLIPS>  (>  4 3)↵
TRUE
CLIPS>  (<  6 2)↵
FALSE
CLIPS>  (integerp 3)↵
TRUE
CLIPS>  (integerp 3.5)↵
FALSE
CLIPS>
```

9.5 EL ELEMENTO CONDICIONAL TEST

Existen muchos casos en los que conviene repetir un cálculo o algún otro procesamiento de la información, la manera habitual de hacerlo es configurando un lazo. En el ejemplo anterior, se configuró un lazo para que se repitiera hasta que el usuario respondiera correctamente a una

pregunta, pero existen muchas situaciones en las que es necesario que un lazo se termine automáticamente como resultado de la evaluación de una expresión arbitraria.

El **elemento condicional (EC) test** proporciona una forma eficaz de evaluar expresiones del lado izquierdo de la regla; pues en lugar de comparar patrones con un hecho de la lista de hechos, evalúa una expresión. (La última función de la expresión debe ser una función de afirmación.) Si esta evaluación arroja cualquier valor diferente al símbolo *FALSE*, el EC test se satisface; pero si tiene como resultado el símbolo *FALSE*, no se satisface. Sólo se disparará una regla si se satisfacen todos sus elementos condicionales test junto con otros patrones, la sintaxis del EC test es

```
(test <función-predicado>)
```

Por ejemplo, suponiendo que es el turno del jugador humano, si sólo queda un palillo en el cúmulo, el jugador humano ha perdido; si hay más de un palillo, se le debe preguntar cuántos palillos quedan en el cúmulo. Como la regla que pregunta al humano cuántos palillos deben eliminarse del cúmulo necesita verificar que hay más de uno en la pila, puede utilizarse la función de predicado > para expresar esta restricción, como se muestra en el siguiente EC test, donde ?tamaño es el número de palillos restantes en el cúmulo:

```
(test (> ?tamaño 1))
```

Una vez que el jugador humano ha establecido el número de palillos que habrá de eliminarse, debe verificarse que la respuesta sea válida, es decir, el número de palillos tomados debe ser un entero y debe ser mayor o igual a uno y menor o igual a tres, porque un jugador no puede tomar más palillos de los que hay en el cúmulo y debe ser forzado a tomar el último. La función de predicado *and* puede utilizarse para expresar todas estas restricciones, como se muestra en el siguiente EC test, donde ?elección contendría el número de palillos por tomar y ?tamaño es el número de palillos restantes en la pila:

```
(test (and (integerp ?elección)
           (>= ?elección 1)
           (<= ?elección 3)
           (< ?elección ?tamaño)))
```

Del mismo modo, un número no válido de palillos tomados sería un valor que no es un entero, menor que uno o mayor que tres, o mayor o igual al número de palillos restantes. Esto puede expresarse utilizando la función de predicado *or*, como se muestra en el siguiente EC test, donde ?elección contendría el número de palillos que habrán de tomarse y ?tamaño es el número de palillos que queda en el cúmulo:

```
(test (or  (not (integerp ?elección))
           (< ?elección 1)
           (> ?elección 3)
           (>= ?elección ?tamaño)))
```

Las siguientes reglas usan los EC test anteriores para verificar que el jugador humano ha tomado un número válido de palillos. El hecho *tamaño-pila* utilizado en ellas almacena la información sobre el número de palillos restantes en el cúmulo.


```
(defrule obtener-movimiento-humano
   (movimiento-humano h)
   (tamaño-pila ?tamaño)
   ; El jugador humano sólo tiene una opción cuando hay
   ; más de un palillo en el cúmulo
   (test (> ?tamaño 1))
   =>
   (printout t
         "¿Cuántos palillos desea tomar?")
   (assert (turno-humano (read))))

(defrule buen-movimiento-humano
   ?turno <- (jugador-mueve h)
   (tamaño-cúmulo ?tamaño)
   ?número-tomado <- (turno-humano ?elección)
   (test (and (intergerp ?elección)
              (>= ?elección 1)
              (<= ?elección 3)
              (< ?elección ?tamaño)))
   =>
   (retract ?turno ?número-tomado)
   (printout t "El humano hizo un movimiento válido"
crlf))

(defrule mal-movimiento-humano
   ?turno <- (jugador-mueve h)
   (tamaño-cúmulo ?tamaño)
   ?número-tomado <- (turno-humano ?elección)
   (test (or (not (intergerp ?elección)
              (< ?elección 1)
              (> ?elección 3)
              (>= ?elección ?tamaño)))
   =>
   (printout t "El humano hizo un movimiento no válido"
crlf)
   (retract ?turno ?número-tomado)
   (assert (jugador-mueve h)))
```

Este programa terminará cuando se haya introducido una opción válida para el número de palillos tomados. Una vez más, se utiliza un hecho de control para volver a disparar una regla que permita que se reintroduzcan respuestas no válidas, la regla *mal-movimiento-humano* reafirmará el hecho de control (jugador-mueve h) para reactivar la regla *obtener-movimiento-humano*, y entonces el jugador humano puede reeintroducir el número de palillos que se toman.

9.6 LA RESTRICCIÓN DEL CAMPO DE PREDICADO

La **restricción del campo de predicado**, :, es útil para realizar pruebas de predicado directamente dentro de los patrones, en muchas maneras es similar a realizar un EC test directamente después de un patrón, aunque en algunos casos, como se analiza en el capítulo 11, es más eficaz. La restricción del campo de predicado puede utilizarse al igual que una restricción de campo literal, puede colocarse por sí sola en un campo o usarse como parte de un campo más complejo utilizando las restricciones de campo conectivo ~, &, y |, y siempre va seguida de una función para evaluación. Al igual que con un EC test, esta función debe ser una función de predicado.

Por ejemplo, la regla *obtener-movimiento-humano* de la sección anterior utilizó los siguientes dos patrones para verificar que el cúmulo contenía más de un palillo:

```
(tamaño-pila ?tamaño
(test (> ?tamaño 1))
```

Estos dos patrones podrían reemplazarse con el siguiente patrón sencillo:

```
(tamaño-pila ?tamaño&:(> ?tamaño 1))
```

La restricción de campo de predicado puede utilizarse por sí sola; sin embargo, son pocas las situaciones en que esto es útil, pues por lo general una variable se delimita y luego se prueba utilizando esta restricción. Cuando se lee un patrón, es útil considerar la restricción de campo de predicado como si significara *tal que*, por ejemplo, la restricción de campo mostrada:

```
?tamaño&:(> ?tamaño 1)
```

podría leerse como "delimitar ?tamaño tal que ?tamaño sea mayor que 1".

Una aplicación de la restricción del campo de predicado es la verificación de errores en los datos. Por ejemplo, la siguiente regla verifica que un elemento de los datos sea numérico, antes de sumarlo a un total en funcionamiento actual:

```
(defrule agregar-suma
   (elemento-datos ?valor&:(númerop ?valor))
   ?total-anterior <- (total ?total)
   =>
   (retract ?total-anterior)
   (assert (total (+ ?total ?valor))))
```

El segundo campo del primer patrón puede leerse como "delimitar ?valor tal que ?valor sea un número". Las siguientes dos reglas realizan el mismo tipo de verificación de errores y demuestran el uso de las restricciones de campo conectivo ~ y | con la restricción de campo de predicado:


```
(encontrar-tipo-de-datos-1
   (datos ?elemento&:(stringp ?elemento)|: (symbolp ?elemento))
   =>
   (printout t ?elemento " es una cadena o un símbolo "
            crlf))

(defrule encontrar-datos-tipo-2
   (datos ?elemento&~:(intergerp ?elemento))
   =>
   (printout t ?elemento " no es un entero " crlf))
```

Al utilizar las funciones de predicado tipo *stringp* y *symbolp* la regla *encontrar-datos-tipo-1* verifica que un elemento de los datos sea una cadena o un símbolo. La regla *encontrar-datos-tipo-2* verifica que un elemento de los datos no sea un entero utilizando la función *integerp* y luego la restricción conectiva ~ para negar el valor.

9.7 LA RESTRICCIÓN DEL CAMPO DEL VALOR DEVUELTO

La **restricción del campo de valor devuelto**, =, permite que el valor devuelto por una función se utilice para hacer comparaciones dentro de un patrón. La restricción del campo del valor devuelto puede utilizarse junto con las restricciones del campo conectivo ~, & y |, además de la restricción del campo de predicado. Al igual que ésta, la restricción del campo del valor devuelto debe ir seguida por una función, pero no tiene que ser una función de predicado. La única restricción es que la función debe tener un solo valor de campo devuelto. La siguiente regla muestra cómo puede utilizarse la restricción del valor devuelto en el programa Palillos para determinar el número de palillos que el jugador debe eliminar del cúmulo:

```
(deftemplate tomar-palillos
   (slot cuántos)
   (slot quedan))

(deffacts tomar-palillos-información
   (tomar-palillos (cuántos 1) (quedan 1))
   (tomar-palillos (cuántos 1) (quedan 2))
   (tomar-palillos (cuántos 2) (quedan 3))
   (tomar-palillos (cuántos 3) (quedan 0)))

(defrule computadora-mueve
   ?turno <- (jugador-mueve c)
   ?cúmulo <- (tamaño-cúmulo ?tamaño)
   (test (< ?tamaño 1))
   (tomar-palillos (cuántos ?número)
                   (quedan =(mod ?tamaño 4)))
   =>
   (retract ?turno ?cúmulo)
   (assert (tamaño-cúmulo (- ?tamaño ?número)))
   (assert (jugador-mueve h)))
```

La regla *computadora-mueve* determina el número apropiado de palillos que la computadora tomará en su turno; el primer elemento condicional de patrón asegura que la regla sólo es aplicable cuando le toca mover a la computadora, el segundo y tercero elementos verifican que el número restante de palillos sea mayor que uno, si sólo queda un palillo, la computadora estaría forzada a tomarlo y perder. El patrón final funciona junto con la deffacts *tomar-palillos-información* para determinar el número apropiado de palillos por tomar. La función **mod** (abreviatura de módulo) devuelve el entero resultante de dividir su primer argumento entre su segundo. Cuando se lee un patrón, es útil considerar la restricción del campo del valor devuelto como si significara *es igual a*; por ejemplo, la restricción de campo:

```
=(mod ?tamaño4)
```

podría leerse como "El campo es igual a ?tamaño módulo 4".

La computadora tratará de tomar suficientes palillos para que cuando el número total de palillos se divida entre cuatro, quede un residuo de 1; si esto ocurre resta uno cuando la computadora ha comenzado su movimiento, entonces habrá perdido a menos que su oponente cometa un error; en este caso, sólo toma una palillo para que el juego dure más (esperando que el jugador humano cometa un error). En todos los demás casos, la computadora puede tomar el número apropiado de palillos para que su oponente pierda.

Para demostrar más detalladamente cómo funciona la restricción del campo del valor de devolución, supongamos que el tamaño del cúmulo es 7 y que es el turno de la computadora. Los primeros tres elementos condionales de la regla *computadora-mueve* serán satisfechos con el último elemento restante. El número 7 substituirá a ¿tamaño en la expresión de la función (mod ?tamaño 4) quedando la expresión como (mod 7 4), que toma el valor 3. Sólo hay un hecho tomar-palillos con un valor de ranura de quedan 3 y el valor de ranura cuántos para este hecho es 2, de modo que la computadora terminará tomando 2 palillos, dejando en el cúmulo 5 palillos. Después del siguiente movimiento del jugador humano, la computadora puede forzar una derrota.

Observe que la restricción de campo =, que sólo puede utilizarse dentro de un elemento condicional de patrón, no es la misma que la función de predicado =, que puede utilizarse dentro de una restricción de campo :, el EC test, el lado derecho de la regla o en la parte superior del indicador de nivel. Nuevamente, al igual que con la restricción del campo de predicado, es posible generar restricciones de campo más complejas utilizando la restricción del valor devuelto con las restricciones de campo conectivo ~, & y |.

9.8 EL PROGRAMA PALILLOS

Como ya se han analizado todas las técnicas básicas necesarias para completar el programa Palillos, a continuación se muestra una ejecución de muestra del programa, luego de que ha sido cargado.

```
CLIPS> (reset)↵
CLIPS> (run) ↵
¿Quién mueve primero (Computadora: c Humano: h)? c↵
¿Cuántos palillos quedan en el cúmulo? 15↵
La computadora toma 2 palillo(s).
quedan 13 palillo(s) en el cúmulo.
¿Cuántos palillos desea tomar? 3↵
quedan 10 palillo(s) en el cúmulo.
La computadora toma 1 palillo(s).
```



```
Quedan 9 palillo(s) en el cúmulo.
¿Cuántos palillos desea tomar? 2↵
quedan 7 palillo(s) en el cúmulo.
La computadora toma 2 palillo(s).
Quedan 5 palillo(s) en el cúmulo.
¿Cuántos palillos desea tomar? 1↵
quedan 4 palillo(s) en el cúmulo.
La computadora toma 3 palillo(s).
Quedan 1 palillo(s) en el cúmulo.
¡Debe tomar el último palillo!
¡Usted pierde!
CLIPS>
```

9.9 EL ELEMENTO CONDICIONAL *OR*

Hasta ahora, todas las reglas mostradas tienen un **elemento condicional *and* implícito** entre los patrones, es decir, una regla no se disparará a menos que todos los patrones sean verdaderos. Con CLIPS también se pueden especificar un **elemento condicional *and* explícito** y un **elemento condicional *or* explícito** en el lado izquierdo.

Consideremos como ejemplo de un elemento condiconal *or* a las siguientes reglas que utilizará el sistema de vigilancia de la planta industrial (como se analiza en el capítulo 7) que primero se escriben sin un elemento condicional *or*, y luego se verá cómo reescribirlas con él.

```
(defrule cortar-electricidad-1
   (emergencia (type inundación))
   =>
   (printout t "Cortar la electricidad" crlf))

(defrule cortar-electricidad-2
   (extintor (type contraincendio)
             (status encendido))
   =>
   (printout t "Cortar la electricidad" crlf))
```

En lugar de escribir dos reglas separadas, puede un elemento condicional combinarse en la siguiente regla, usando el EC *or*:

```
(defrule cortar-electricidad
   (or (emergencia (type inundación))
       (extintor (type contraincendios)
                 (status encendido))
   =>
   (printout t "Cortar la electricidad" crlf))
```

que es equivalente a las dos reglas procedentes. La afirmación de los dos hechos que coinciden con los patrones de esta regla provocaría que la regla se disparara dos veces, una para cada uno de los hechos.

Aparte del elemento *or*, pueden incluirse otros EC y serán parte del elemento *and* implícito para todo el lado izquierdo de la regla. Por ejemplo, la regla

```
(defrule cortar-electricidad
   (energía-eléctrica (status encendido))
   (or (emergencia (type inundación))
       (extintor (type contraincendio)
                 (status encendido)
   =>
   (printout t "Cortar la electricidad" crlf))
```

es equivalente a estas dos reglas:

```
(defrule cortar-electricidad-1
   (energía-eléctrica (status encendido))
   (emergencia (type inundación))
   =>
   (printout t "Cortar la electricidad" crlf))

(defrule cortar-electricidad-2
   (energía-eléctrica (status encendido))
   (extintor (type contraincendio)
                 (status encendido))
   =>
   (printout t "Cortar la electricidad" crlf))
```

Como un elemento condicional *or* genera el equivalente de varias reglas, es posible que una regla se active más de una vez por los patrones contenidos con él. Por tanto, la pregunta natural es ¿cómo evitar el problema de más de un disparo?; por ejemplo, son innecesarias las múltiples salidas de "Cortar la electricidad" que aparecen para varios hechos, después de todo, sólo es necesario cortar la energía una vez, sin importar el número de razones para hacerlo.

Quizá la manera más apropiada de evitar que se disparen las otras reglas sea cambiando la regla para actualizar la lista de hechos, indicando que ya se ha cortado la electricidad. A continuación se muestra la regla modificada. (Observe que al escribir estas reglas se ha supuesto que en la planta hay un operador de alarma que está observando la salida del sistema de vigilancia y emprende las acciones. En la realidad, el sistema experto puede tener el control directo para encender y apagar varios sistemas o el operador de planta puede tener un mecanismo para indicar al sistema de vigilancia que ya se han emprendido varias acciones.)

```
(defrule cortar-electricidad
   (?electricidad <- (energía-eléctrica (status
                     encendido))
   (or (emergencia (type inundación))
       (extintor (type contraincendio)
                 (status encendido))
   =>
   (modify ?electricidad (status apagado)))
   (printout t "Cortar la electricidad" crlf))
```

Ahora la regla sólo se dispara una vez debido a que el hecho que contiene la información acerca de la energía eléctrica se modifica cuando la regla se dispara; esto eliminará otras

activaciones de la regla hechas por otros patrones. Si también es necesario eliminar la razón que sustenta el corte de energía en la lista de hechos, entonces la regla tendría que escribirse:

```
(defrule cortar-electricidad
   (?electricidad <- (energía-eléctrica (status
                     encendido))
   (or ?razón <- (emergencia (type inundación))
       ?razón <- (extintor
                     (type contraincendios)
                     (status encendido))
   =>
   (retract ?razón)
   (modify ?electricidad (status apagado))
   (printout t "Cortar la electricidad" crlf))
```

Observe que todos los elementos condicionales de patrón dentro del elemento *or* se unen a la misma variable, ?razón; en principio esto parecería un error, porque normalmente no se asignan las mismas variables a los diferentes patrones de una regla ordinaria, pero una regla que utiliza un elemento condicional *or* es diferente, pues éste siempre producirá "varias reglas", como la regla anterior que es equivalente a las dos siguientes reglas:

```
(defrule cortar-electricidad-1
   ?electricidad <- (energía-eléctrica (status encendido))
   ?razón <- (emergencia (type inundación))
   =>
   (retract ?razón)
   (modify ?electricidad (status apagado))
   (printout t "Cortar la electricidad" crlf))

(defrule cortar-electricidad-2
   ?electricidad <- (energía-eléctrica (status encendido))
   ?razón <- (extintor
                (type contraincendio)
                (status encendido))
   =>
   (retract ?razón)
   (modify ?electricidad (status apagado))
   (printout t "Cortar la electricidad" crlf))
```

Observando estas reglas, se puede ver que fue necesario emplear el mismo nombre de variable para igualar la acción (retract ?electricidad ?razón) en el lado derecho.

9.10 EL ELEMENTO CONDICIONAL *AND*

El concepto del elemento condicional *and* es opuesto al *or*, pues en lugar de que cualquiera de los diferentes condicionales dispare una regla, *and* requiere que se satisfagan todos ellos. CLIPS coloca automáticamente un elemento *and* implícito en dirección contraria al lado derecho de la regla. Una regla como:

```
(defrule cortar-electricidad
   ?electricidad <- (energía-eléctrica (status encendido))
   (emergencia (type contraincendios))
   =>
   (modify ?electricidad (status apagado))
   (printout t "Cortar la electricidad" crlf))
```

también podría escribirse con un elemento condicional *and* explícito como

```
(defrule cortar-electricidad
   (and ?electricidad <- (energía-eléctrica (status
                          encendido))
        (emergencia (type inundación)))
   =>
   (modify ?electricidad (status apagado))
   (printout t "Cortar la electricidad" crlf))
```

Por supuesto, no existe ventaja alguna en escribir una regla con un *and* explícito en dirección contraria a todo el lado izquierdo. Este elemento se emplea para que pueda usarlo junto con otros elementos condicionales para crear patrones más complejos. Por ejemplo, puede utilizarse con un *or* que requiera grupos de varias condiciones para ser verdadero, como se muestra en el siguiente ejemplo:

```
(defrule utilizar-extintor-dióxido-carbono
   ?sistema <- (extintor
                (type dióxido-carbono)
                (status apagado))
   (or (emergencia (type incendio-clase-B))
       (and (emergencia (type incendio-clase-C))
            (energía-eléctrica (status apagado))))
   =>
   (modify ?sistema (status encendido))
   (printout t "Utilizar extintor de dióxido de carbono"
      crlf))
```

Esta regla se activará si existe una emergencia de incendio clase B (como aceite o grasa en combustión) o si hay una clase C (de equipo eléctrico) y la energía eléctrica ya se ha cortado. En efecto, siempre se emplea el extintor de dióxido de carbono para un incendio de clase B, pero sólo se utiliza para uno clase C si la interrupción de la electricidad no extinguió el fuego. La defrule *utilizar-extintor-dióxido-carbono* es equivalente a las dos reglas siguientes:

```
(defrule utilizar-extintor-dióxido-carbono-1
   ?sistema <- (extintor
                (type dióxido-carbono)
                (status apagado))
   (emergencia (type incendio-clase-B))
   =>
   (modify ?sistema (status encendido))
   (printout t "Utilizar extintor de dióxido de carbono"
      crlf))
```



```
(defrule utilizar-extintor-dióxido-carbono-2
   ?sistema <- (extintor
                (type dióxido-carbono)
                (status apagado))
   (emergencia (type incendio-clase-C))
   (energía-eléctrica (status apagado))
   =>
   (modify ?sistema (status encendido))
   (printout t "Utilizar extintor de dióxido de carbono"
      crlf))
```

9.11 EL ELEMENTO CONDICIONAL *NOT*

Algunas veces es útil tener la posibilidad de activar reglas con base en la ausencia de un hecho particular en la lista de hechos. CLIPS permite especificar la ausencia de un hecho en el lado izquierdo de una regla utilizando el **elemento condicional *not***. Por ejemplo, un sistema experto de vigilancia puede tener dos reglas para informar su status:

```
SI   se va a informar el estado de la vigilancia y
     se está manejando una emergencia
ENTONCES  informar el tipo de emergencia

SI  se va a informar el estado de la vigilancia y
    no se está manejando una emergencia
ENTONCES  informar que no se está manejando una emergencia
```

EL elemento condicional *not* puede aplicarse de manera conveniente a estas sencillas reglas simples ya expuestas:

```
(defrule informar-emergencia
   (informar-estado)
   (emergencia (tipo ?tipo))
   =>
   (printout t "Manejando " ?tipo " emergencia"
        crlf))

(defrule no-hay-emergencia
   (informar-estado)
   (not (emergencia))
   =>
   (printout t "No se están manejando emergencias" crlf))
```

Observe que estas dos reglas son mutuamente excluyentes. Es decir, no pueden estar ambas en la agenda al mismo tiempo, ya que el segundo patrón de cada regla no puede satisfacerse de manera simultánea.

También pueden utilizarse variables dentro de un patrón negado para producir algunos efectos interesantes. Consideremos la siguiente regla, que busca el número más grande de un grupo de hechos que representan números:

```
(defrule número-mayor
   (número ?x)
   (not (número ?y&:(> ?y ?x)))
   =>
   (printout t "El número mayor es " ?x crlf))
```

El primer patrón unirá a todos los hechos *número*, pero el segundo no permitirá que ningún hecho active la regla, con excepción del hecho con el mayor valor para ?x.

Conviene advertir que las variables que primero se unen dentro de un elemento condicional *not* conservan su valor solamente dentro del ámbito del propio *not*. Por ejemplo, la regla:

```
(defrule no-hay-emergencia
   (informar-status)
   (not (emergencia (tipo ?tipo)))
   =>
   (printout t "No hay emergencia de tipo " ?tipo crlf))
```

generará un error porque la variable ?tipo se utilizó del lado derecho de la regla, pero sólo se unió dentro del condicional *not* en el lado izquierdo de la regla.

El alcance de las variables también se aplica al lado izquierdo de una regla; por ejemplo, la siguiente regla determina que no hay personas conocidas que tengan su cumpleaños en un día determinado:

```
(defrule no-hay-cumpleaños-en-fecha-específica
   (verificar-que-no-hay-cumpleaños (date ?fecha))
   (not (persona (cumpleaños ?fecha)))
   =>
   (printout t "No hay cumpleaños en " ?fecha crlf))
```

Si los primeros dos elementos condicionales se intercambian como sigue,

```
(defrule no-hay-cumpleaños-en-fecha-específica
   (not (persona (cumpleaños ?fecha)))
   (verificar-que-no-hay-cumpleaños (date ?fecha))
   =>
   (printout t "No hay cumpleaños en " ?fecha crlf))
```

la regla ya no funcionará de manera adecuada, porque el primer condicional no quedará satisfecho si sólo hay hechos *cualquier persona*. El valor que se une a la variable ?fecha en el primer elemento condicional no tiene efecto en los valores permitidos para la variable ?fecha en el segundo. Esto contrasta con la versión original de la regla *no-hay-cumpleaños-en-fecha-específica*, donde el valor unido a la variable ?fecha en el primer condicional restringió la búsqueda de hechos *persona* en el segundo a aquellas personas que cumplen años en una fecha específica. A diferencia de los condicionales de patrón, el orden de colocación de un elemento condicional *not* en el lado izquierdo de una regla puede afectar la activación de la regla.

Not puede utilizarse junto con otros elementos condicionales. Por ejemplo, la siguiente regla determina que no hay dos personas que cumplan años en la misma fecha:


```
(defrule no-hay-cumpleaños-idénticos
   (not (and (persona (nombre ?nombre)
                      (cumpleaños ?fecha))
             (persona (nombre ~?nombre)
                      (cumpleaños ?fecha))))
   =>
   (printout t
      "No hay dos personas que tengan el mismo
      cumpleaños" crfl)
```

and se utiliza dentro de *not* porque éste puede contener cuando mucho un elemento condicional. Observe que las variables ?nombre y ?fecha se vuelven a utilizar dentro de *not* para restringir correctamente la búsqueda de dos personas con el mismo cumpleaños.

Debido al algoritmo utilizado por CLIPS, el patrón (hecho-inicial) se agrega al principio de cualquier *and* (implícito o explícito) cuyo primer condicional es un *not* o un *test*; por tanto, la regla:

```
(defrule no-hay-emergencias
   (not (emergencia))
   =>
   (printout t "No hay emergencias" crlf))
```

se convierte en:

```
(defrule no-hay-emergencias
   (hecho-inicial)
   (not (emergencia))
   =>
   (printout t "No hay emergencias" crlf))
```

Resulta útil comprender esta conversión cuando se examina la salida del comando *matches*. También conviene considerar que no se despliega el índice de hechos para los condicionales *not* en comparaciones parciales o activaciones de reglas. Por tanto, la activación "f-5,,f-3" indica que el primer elemento condicional coincidió con el hecho que tiene un índice 5, el segundo elemento condicional era un *not* que no coincidió con ningún hecho y, por tanto, se satisfizo; y el tercer elemento condicional coincidió con el hecho que tiene un índice 3.

9.12 EL ELEMENTO CONDICIONAL *EXISTS*

El **elemento condicional** *exists* permite comparar patrones con base en la existencia de cuando menos un hecho que coincida con un patrón, sin importar el número total de hechos que en realidad coincidan con él. Esto permite que se genere una sola coincidencia o activación parcial para una regla, con base en la existencia de un hecho entre una clase de hechos. Por ejemplo, suponiendo que cada vez que se presenta una emergencia debe imprimirse un mensaje informativo que indique a los operadores de planta que deben estar alerta, la regla podría escribirse como se muestra a continuación:

```
(deftemplate emergencia (slot tipo))
(defrule aviso-de-emergencia-al-operador
   (emergencia)
   =>
   (printout t "Emergencia: aviso al operador" crlf)
   (assert (aviso-operador)))
```

Observe lo que sucede cuando se afirma más de un *hecho de emergencia* (el mensaje a los operadores se imprime más de una vez):

```
CLIPS> (reset) ↵
CLIPS> (assert (emergencia (type incendio))) ↵
<Fact-1>
CLIPS> (assert (emergencia (type inundación))) ↵
<Fact-2>
CLIPS> (run) ↵
Emergencia: aviso al operador
Emergencia: aviso al operador
CLIPS>
```

La regla *aviso-de-emergencia-al-operador* podría modificarse de la siguiente manera para evitar que vuelva a dispararse la regla para una emergencia adicional:

```
(defrule aviso-de-emergencia-al-operador
   (emergencia)
   (not (alerta-operador))
   =>
   (printout t "Emergencia: aviso al operador" crlf)
   (assert (aviso-operador)))
```

El condicional (not (alerta-operador) evita que la regla se vuelva a disparar; sin embargo, al cambiar la regla de esta manera se supone que si ya hay un aviso para el operador, lo habrá disparado la regla *aviso-de-emergencia-al-operador*. Es posible que los operadores pudieran ponerse en alerta para un simulacro de emergencia, como lo muestra la siguiente regla:

```
(defrule aviso-de-simulacro-al-operador
   (simulacro-operador)
   (not (aviso-operador))
   =>
   (printout t "Simulacro: aviso al operador" crlf)
   (assert (alerta-operador)))
```

Observe que si se dispara primero la regla de aviso de simulacro, la regla de aviso de emergencia no podrá dispararse más adelante porque el hecho *aviso-operador* se habrá afirmado. Para corregir el problema, podría modificarse el hecho *aviso-operador* para almacenar la causa del aviso, pero el uso excesivo de hechos de control para evitar que las reglas se disparen aumenta la complejidad de las reglas y disminuye su capacidad de mantenimiento al agregar dependencias de control entre ellas.

Por fortuna, la regla *aviso-de-emergencia-al-operador* puede modificarse para utilizar el condicional *exists*, lo que eliminará la necesidad de recurrir a los hechos de control. Independientemente de cuántos hechos coincidan con los condicionales dentro de *exists*, éste sólo generará una coincidencia parcial. Por tanto, si un *exist* es el enésimo condicional de una regla y los N – 1 condicionales previos han generado M coincidencias parciales, el número más grande de coincidencias parciales que los primeros N condicionales pueden generar es M. La regla *aviso-de-emergencia-al-operador* reescrita para utilizar el condicional *exists* es


```
(defrule aviso-de-emergencia-al-operador
   (exists (emergencia))
   =>
   (printout t "Emergencia: Aviso al operador" crlf)
   (assert (aviso-operador)))
```

Esta regla sólo generará una activación y, por tanto, el mensaje para el operador sólo se imprimirá una vez, como se muestra en el siguiente diálogo:

```
CLIPS> (reset)↵
CLIPS> (assert (emergencia (type incendio)))↵
<Fact-1>
CLIPS> (assert (emergencia (type inundación)))↵
<Fact-2>
CLIPS> (agenda)↵
0      aviso-de-emergencia-al-operador: f-0,
For a total of 1 activation.
CLIPS> (run)↵
Emergencia: aviso al operador
CLIPS>
```

El condicional *exists* se implanta utilizando una combinación de condicionales *and* y *not*. Los condicionales dentro de *exists* se colocan dentro de un *and* y luego dentro de dos *not*. Por tanto, la regla *aviso-de-emergencia-al-operador* se convierte a lo siguiente al rodear todo el lado izquierdo con un *and* y luego reemplazar el *exists* como se establece:

```
(defrule aviso-de-emergencia-al-operador
   (and (not (not (and (emergencia)))))
   =>
   (printout t "Emergencia: aviso al operador" crlf)
   (assert (aviso-operador)))
```

Como el condicional *and* que se encuentra al lado izquierdo de la regla tiene un *not* como primer condicional, el patrón condicional (hecho-inicial) se coloca al principio. Haciendo esta adición y eliminando el *and* externo que rodea al patrón de emergencia se produce la siguiente regla:

```
(defrule aviso-de-emergencia-al-operador
   (and (hecho-inicial)
        (not (not (emergencia))))
   =>
   (printout t "Emergencia: aviso al operador" crlf)
   (assert (aviso-operador)))
```

El patrón (hecho-inicial) del principio de la regla explica el índice de hechos f-0 que se desplegó en la regla cuando se emitió el comando *agenda* en el diálogo previo. Si no hay hechos de *emergencia*, entonces el EC *not* más interno será satisfecho y, por tanto no lo será el más externo y no se activará la regla. Por el contrario a la inversa, si hay hechos de emergencia, no quedará satisfecho el *not* más interno, lo será por tanto, más externo y se activará la regla.

9.13 EL ELEMENTO CONDICIONAL *FORALL*

El **elemento condicional** *forall* permite comparar patrones con base en un conjunto de condicionales que se satisface para *cada* aparición de otro. Por ejemplo, supongamos que hay varios sitios (como edificios) que podrían incendiarse en una planta industrial, y se desea determinar si se ha evacuado cada edificio que está en llamas y si una cuadrilla de bomberos está tratando de extinguir el fuego. El condicional *forall* puede utilizarse para verificar esto; el hecho *emergencia* se modificará como se muestra a continuación para comprender la ubicación y el tipo de emergencia. Se utilizarán otras dos deftemplates para indicar la ubicación de las cuadrillas de bomberos y si se ha evacuado un edificio, la regla *todo-incendio-se-está-controlando* utiliza estas deftemplates para determinar si se han satisfecho las condiciones apropiadas.

```
(deftemplate emergencia
   (slot tipo)
   (slot ubicación))

(deftemplate cuadrilla-bomberos
   (slot nombre)
   (slot ubicación))

(deftemplate evacuado
   (slot edificio)

(defrule todo-incendio-se-está-controlando
   (forall (emergencia (type incendio)
                       (ubicación ?dónde))
           (cuadrilla-bomberos (ubicación ?dónde))
           (evacuado (edificio ?dónde)))
   =>
   (printout t
             "Todos los edificios que están en llamas" crlf
             "han sido evacuados y" crlf
             "hay bomberos en el lugar"
```

Por cada hecho que coincida con el patrón (emergencia (type incendio) (ubicación ?dónde)), también debe haber hechos que igualan el patrón (escuadra-bomberos (ubicación ?dónde)) y el patrón (evacuado (edificio ?dónde)). Cuando se cargan las deftemplates y la defrule y se aplica el comando *reset*, la regla debe satisfacerse porque no hay emergencias de incendio.


```
CLIPS> (watch activations)↵
CLIPS> (reset)↵
==> Activation 0     todo-incendio-se-está-controlando: f-0
CLIPS>
```

Una vez que se afirma un hecho *emergencia*, la regla se desactiva hasta que se afirman los hechos *escuadra-bomberos* y *evacuado* apropiados.

```
CLIPS>
(assert (emergencia (type incendio)↵
                    (ubicación edificio-11)))↵
<== Activation 0   todo-incendio-se-está-controlando: f-0,
<Fact-1>
CLIPS>
(assert (evacuado (edificio edificio-11)))↵
<Fact-2>
CLIPS>
(assert (cuadrilla-bomberos (nombre A)↵
                            (ubicación edificio-11)))↵
==> Activation 0   todo-incendio-se-está-controlando: f-0,
<Fact-3>
CLIPS>
(assert (cuadrilla-bomberos (nombre B)↵
                            (ubicación edificio-1)))↵
<Fact-4>
CLIPS>
(assert (emergencia (type incendio)↵
                    (ubicación edificio-1)))↵
<== Activation 0   todo-incendio-se-está-controlando: f-0,
<Fact-5>
CLIPS>
(assert (evacuado (edificio edificio-1)))↵
==> Activation 0   todo-incendio-se-está-controlando: f-0,
<Fact-6>
CLIPS>
```

Si se elimina el hecho *escuadra-bomberos* para el edificio 1, entonces la regla se desactiva, pero la eliminación del hecho *emergencia* para este edificio la reactivará.

```
CLIPS> (retract 4)↵
<== Activation 0   todo-incendio-se-está-controlando: f-0,
CLIPS> (retract 5)↵
==> Activation 0   todo-incendio-se-está-controlando: f-0,
CLIPS> (run)↵
Todos los edificios que están en llamas
Han sido evacuados y
Hay bomberos en el lugar
CLIPS>
```

A continuación se muestra el formato general del condicional *forall*.

```
(forall <primer-EC>
      <ECs-restantes>+)
```

Para satisfacer el elemento condicional *forall*, cada hecho que coincide con el <primer-EC> también debe tener hechos que coinciden con todos los <EC-restantes>. El formato general del condicional *forall* se reemplaza con combinaciones de los condicionales *and* y *not*, utilizando el siguiente formato:

```
(not (and <primer-EC>
          (not (and <EC-restantes>+))))
```

9.14 EL ELEMENTO CONDICIONAL LOGICAL

El **elemento condicional** ***logical*** permite especificar que la existencia de un hecho depende de la existencia de otro hecho o grupo de hechos. Este condicional es el medio que proporciona CLIPS para la preservación de la verdad. Consideremos como ejemplo la siguiente regla, que indica que los bomberos deben utilizar máscaras de oxígeno cuando un incendio produce humos tóxicos:

```
(defrule humos-tóxicos-presentes
   (emergencia (type incendio))
   (humos-tóxicos-presentes)
   =>
   (assert (usar-máscaras-oxígeno)))
```

Como se muestra en el siguiente diálogo, la regla anterior afirmará el hecho *usar-máscaras-oxígeno* cada vez que haya una emergencia de incendio de este tipo:

```
CLIPS> (unwatch all)↵
CLIPS> (reset)↵
CLIPS> (watch facts)↵
CLIPS> (assert (emergencia (type incendio))
                  (humos-tóxicos-presentes)↵
==> f-1      (emergencia (type incendio))
==> f-2      (humos-tóxicos-presentes)
<Fact-2>
CLIPS> (run)↵
==> f-3      (usar-máscaras-oxígeno)
CLIPS>
```

¿Qué sucede cuando el incendio se extingue y ya no hay ningún humo tóxico? Como se muestra en el siguiente diálogo, la retracción del hecho *emergencia* o el hecho *humos-tóxicos-presentes* no afecta al hecho *usar-máscaras-oxígeno*.

```
CLIPS> (retract 1 2)↵
<== f-1      (emergencia (type incendio))
<== f-2      (humos-tóxicos-presentes)
CLIPS> (facts) ↵
f-0      (hecho-inicial)
f-3      (usar-máscaras-oxígeno)
```



```
For a total of 2 facts.
CLIPS>
```

Como ya se mencionó, logical es el condicional que CLIPS utiliza como mecanismo de conservación de la verdad para crear dependencias entre hechos. La modificación de la regla *humos-tóxicos-presentes*, como se muestra a continuación, permite que se cree una dependencia entre los hechos que coinciden con los patrones en el lado izquierdo de una regla y los hechos que se afirman de su lado derecho.

```
(defrule humos-tóxicos-presentes
   (logical (emergencia (type incendio))
            (humos-tóxicos-presentes))
   =>
   (assert (usar-máscaras-oxígeno)))
```

Cuando se ejecuta la regla *humos-tóxicos-presentes*, se crea un vínculo entre los hechos que coinciden con los patrones contenidos dentro del EC logical del lado izquierdo de una regla y los hechos afirmados en su lado derecho. En esta regla, si se retraen los hechos *emergencia* o *humos-tóxicos-presentes*, entonces también se retraerá la regla *usar-máscaras-oxígeno*, como se muestra en el siguiente diálogo:

```
CLIPS> (unwatch all)↵
CLIPS> (reset)↵
CLIPS> (watch facts)↵
CLIPS> (assert (emergencia (type incendio))
                  (humos-tóxicos-presentes)↵
==> f-1      (emergencia (type incendio))
==> f-2      (humos-tóxicos-presentes)
<Fact-2>
CLIPS> (run)↵
==> f-3      (usar-máscaras-oxígeno)
CLIPS> (retract 1) ↵
<== f-1      (emergencia (type incendio))
<== f-3      (usar-máscaras-oxígeno)
CLIPS>
```

El hecho *usar-máscaras-oxígeno* recibe **apoyo lógico** de los hechos y *humos-tóxicos-presentes*, por tanto es **dependiente** de ellos. Por su parte, estos dos hechos son **dependencias** del hecho usar-máscaras-oxígeno.

El condicional *logical* no tiene que incluirse rodeando a todos los patrones del lado izquierdo de una regla, pero si se utiliza, debe encerrar al primer condicional del lado izquierdo de una regla y no puede haber espacios entre los condicionales encerrados por él, por ejemplo, los *logical* no pueden colocarse rodeando al segundo y cuarto o el primero y tercer condicionales de una regla, ya que esto dejaría espacios. Esta limitante del condicional *logical* es resultado la forma en que se ha implantado. También es posible hacer que los hechos sean dependientes de la no existencia de hechos utilizando el condicional *not* dentro del *logical*, incluso pueden usarse condiciones más complejas que emplean los condicionales *exists* y *forall* u otras combinaciones de ellos dentro de logical. Excepto por la creación de de-

pendencias entre grupos de hechos, el condicional *logical* actúa en todos los aspectos con un *and*.

Para modificar la regla *humos-tóxicos-presentes* de modo que el hecho *usar-máscaras-oxígeno* sólo sea dependiente del hecho *humos-tóxicos-presentes*, sería necesario reordenar los patrones de la siguiente manera:

```
(defrule humos-tóxicos-presentes
   (logical (humos-tóxicos-presentes))
   (emergencia (type incendio))
   =>
   (assert (usar-máscaras-oxígeno)))
```

Con la regla modificada como se muestra, el hecho *usar-máscaras-oxígeno* no se retraería automáticamente si se retrae el hecho *emergencia* (que es un método más seguro porque es posible que aún haya humos tóxicos aunque se haya extinguido el incendio).

En condiciones normales, la afirmación de un hecho que ya está en la lista de hechos no tiene efecto, pero un hecho lógicamente dependiente que se deriva de más de un origen no se retrae de manera automática hasta que se elimina el apoyo lógico de todos sus orígenes. Por ejemplo, supongamos que se agrega otra regla para indicar que deben utilizarse máscaras de oxígeno si se están empleando extintores de gas:

```
(defrule extintores-gas-en-uso
   (logical (extintores-gas-en-uso))
   (emergencia (type incendio))
   =>
   (assert (usar-máscaras-oxígeno)))
```

La ejecución del sistema ahora provoca que dos reglas separadas afirmen el mismo hecho con base en diferentes razones.

```
CLIPS> (unwatch all)↵
CLIPS> (reset)↵
CLIPS> (watch facts)↵
CLIPS> (watch rules)↵
CLIPS> (assert (emergencia (type incendio))
            (humos-tóxicos-presentes)
                (extintores-gas-en-uso))↵
==> f-1      (emergencia (type incendio))
==> f-2      (humos-tóxicos-presentes)
==> f-3      (extintores-gas-en-uso)
<Fact-3>
CLIPS> (run) ↵
FIRE    1 extintores-gas-en-uso: f-3, f-1
==> f-4      (usar-máscaras-oxígeno)
FIRE    2 humos-tóxicos-presentes: f-1, f-2
CLIPS>
```

La retracción del hecho *humos-tóxicos-presentes* no es suficiente para provocar la retracción automática del hecho *usar-máscaras-oxígeno* porque hay otro apoyo lógico para el uso de las máscaras de oxígeno, pues también el hecho *extintores-gas-en-uso* debe retraerse antes de que se retraiga el hecho *usar-máscaras-oxígeno*.


```
CLIPS> (retract 2) ↵
<== f-2      (humos-tóxicos-presentes)
CLIPS> (retract 3) ↵
<== f-3      (extintores-gas-en-uso)
<== f-4      (usar-máscaras-oxígeno)
CLIPS>
```

Un hecho que se afirma desde el indicador de nivel superior o desde el lado derecho de una regla y que no tiene ningún condicional logical recibe **apoyo condicional**. Un hecho con apoyo incondicional nunca se retraerá automáticamente al retraerse otro hecho. Cualquier apoyo lógico previo para un hecho se descarta una vez que recibe apoyo condicional.

CLIPS proporciona dos comandos para ver los **dependientes** y las **dependencias** asociadas con un hecho. La sintaxis de estos comandos es

```
(dependents <índice-o-dirección-hecho>)
(dependencies <índice-o-dirección-hecho>)
```

En el último ejemplo, antes de que se retraigan los hechos *humos-tóxicos-presentes* y *extintores-gas-en-uso*, estos comandos habrían producido la siguiente salida:

```
CLIPS> (facts)↵
f-0     (hecho-inicial)
f-2     (emergencia (type incendio))
f-2     (humos-tóxicos-presentes)
f-3     (extintores-gas-en-uso)
f-4     (usar-máscaras-oxígeno)
For a total of 5 facts.
CLIPS> (dependents 1)↵
None
CLIPS> (dependents 2)↵
f-4
CLIPS> (dependents 3)↵
f-4
CLIPS> (dependents 4)↵
None
CLIPS> (dependencies 1)↵
None
CLIPS> (dependencies 2)↵
None
CLIPS> (dependencies 3)↵
None
CLIPS> (dependencies 4)↵
f-2
f-3
CLIPS>
```

9.15 COMANDOS DE USO GENERAL

Comando system

El comando **system** permite la ejecución de comandos del sistema operativo desde el interior de CLIPS, su sintaxis es

```
(system <expresión>+)
```

Por ejemplo, la siguiente regla proporcionará una lista de directorio para un directorio específico en una máquina que utiliza el sistema operativo UNIX:

```
(defrule lista-directorio
   (lista-directorio ?directorio)
   =>
   (system "ls " ?directorio))
```

En este ejemplo, el primer argumento para el comando *system*, "ls", es el comando de UNIX para enumerar un directorio. Observe que se incluye un espacio después de los caracteres. El comando *system* simplemente coloca todos sus argumentos juntos como cadenas antes de permitir que el sistema operativo procese el comando. Cualquier espacio necesario para el comando del sistema operativo debe incluirse como parte de un argumento en la llamada al comando *system*.

Los efectos del comando *system* pueden variar de un sistema operativo a otro, y no todos los sistemas operativos tienen la funcionalidad para utilizar el comando *system*, de modo que no puede dependerse de que el comando esté disponible en CLIPS. El comando *system* no devuelve un valor, por lo que es imposible devolver uno directamente a CLIPS después de ejecutar un comando del sistema operativo.

Comando batch

El comando **batch** permite comandos y respuestas que en condiciones normales tendrían que introducirse en el indicador de nivel superior para leerse directamente desde un archivo; su sintaxis es:

```
(batch (<nombre-archivo>)
```

Por ejemplo, supongamos que el siguiente diálogo muestra los comandos y las respuestas que deben introducirse para ejecutar un programa de CLIPS (recuerde que las letras en negritas indican las claves que introduce):

```
CLIPS> (load "reglas1.clp")↵
*************
CLIPS> (load "reglas2.clp")↵
****************
CLIPS> (load "reglas3.clp")↵
*************
CLIPS> (reset)↵
CLIPS> (run)↵
```



```
How many iterations? 10 ↵
Starting value? 1 ↵
End value? 20 ↵
Completed
CLIPS>
```

Los comandos y las respuestas que se necesitan para ejecutar el programa podrían almacenarse en un archivo como se muestra a continuación:

```
(load "reglas1.clp") ↵
(load "reglas2.clp") ↵
(load "reglas3.clp") ↵
(reset) ↵
(run) ↵
10 ↵
1 ↵
20 ↵
```

Si el archivo con los comandos y las respuestas se le llama "comandos.bat", en el siguiente diálogo se muestra cómo puede utilizarse el comando *batch* (otra vez observe que las letras en negritas indican claves que usted introduce):

```
CLIPS> (batch "comandos.bat") ↵
CLIPS> (load "reglas1.clp") ↵
*************
CLIPS> (load "reglas2.clp") ↵
****************
CLIPS> (load "reglas3.clp") ↵
*************
CLIPS> (reset) ↵
CLIPS> (run) ↵
How many iterations? 10 ↵
Starting value? 1 ↵
End value? 20 ↵
Completed
CLIPS>
```

Una vez que se han leído todos los comandos y las respuestas de un archivo de procesamiento por lotes, la interacción del teclado en el indicador de comandos de nivel superior regresa a lo normal.

Cuando se ejecuta en sistemas operativos que soportan argumentos de línea de comandos para programas ejecutables (como UNIX), CLIPS puede ejecutar comandos automáticamente de un archivo de procesamiento por lotes al iniciar. Suponiendo que el programa ejecutable de CLIPS puede ejecutarse al escribir "clips", la sintaxis para ejecutar un archivo de procesamiento por lotes al inicio es

```
clips -f <nombre-archivo>
```

El uso de la opción –f es equivalente a introducir el comando (batch <nombre-archivo>) una vez que se ha iniciado CLIPS. Las llamadas al comando *batch* pueden anidarse.

Comandos dribble-on y dribble-off

El comando **dribble-on** puede utilizarse para almacenar un registro de todas las salidas a la terminal y de todas las entradas del teclado, su sintaxis es

```
(dribble-on <nombre-archivo>)
```

Una vez que se ha ejecutado este comando, todas las salidas se envían a la terminal y todas las entradas introducidas con el teclado se transmiten al archivo especificado mediante <nombre-archivo>, así como a la terminal.

Los efectos del comando *drible-on* pueden eliminarse con el comando **dribble-off**, cuya sintaxis es:

```
(dribble-off)
```

9.16 RESUMEN

En este capítulo se introdujeron varios conceptos para controlar el flujo de ejecución: la función *read* se emplea para demostrar cómo podría crearse un ciclo de control simple para introducir datos utilizando hechos de control que se retraen y luego se vuelven a afirmar: los condicionales *test* junto con las funciones de predicado pueden usarse en el lado izquierdo de una regla para proporcionar opciones de comparación de patrones más completas; además, los *test* pueden utilizarse para mantener un lazo de control. La restricción del campo de predicado permite que se coloquen pruebas de predicado directamente dentro de un patrón. La restricción del campo de igualdad se utiliza para comparar un campo con un valor devuelto por una función. (El programa Palillos demuestra varias de estas técnicas de control.)

El condicional *or* se utiliza para expresar varias reglas como una sola; el condicional *not* permite la comparación de patrones con la ausencia de un hecho en la lista de hechos; el condicional *and* se usa para agrupar condicionales y, junto con los *or* y *not*, puede anidarse de manera arbitraria para expresar condiciones complejas necesarias para satisfacer una regla; el condicional *exists* se utiliza para determinar la existencia de cuando menos un grupo de hechos que satisface un condicional o una combinación de ellos; el condicional *forall* se utiliza para determinar que un conjunto de condicionales se satisface por *cada* aparición de otro; el condicional *logical* proporciona un mecanismo de conservación de la verdad; la existencia de hechos puede hacerse dependiente de la existencia o no existencia de otros hechos.

El comando *system* permite que se ejecuten comandos del sistema operativo desde CLIPS; el comando *batch* permite almacenar una serie de comandos y respuestas en un archivo para reemplazar entradas del teclado normal; los comandos *dribble-on* y *dribble-off* permiten almacenar un registro de salida de terminal en un archivo.

PROBLEMAS

9.1 Agregue al programa Palillos las reglas que preguntarán si el jugador humano desea volver a jugar después de que ha terminado una partida.

9.2 Modifique el programa Palillos para permitir que dos jugadores humanos jueguen entre sí, además de que la computadora juegue contra un humano.

9.3 Rescriba las siguientes reglas como una sola utilizando los condicionales *and* y *not*.

```
(defrule regla-1
   (hecho-a)
   (hecho-d)
   =>)

(defrule regla-2
   (hecho-b)
   (hecho-c)
   (hecho-e)
   (hecho-f)
   =>)

(defrule regla-3
   (hecho-a)
   (hecho-e)
   (hecho-f)
   =>)

(defrule regla-4
   (hecho-b)
   (hecho-c)
   (hecho-d)
   =>)
```

9.4 Escriba un programa que utilice los condicionales *and* y *or* para el árbol Y/O que se muestra en la figura 3.10 de la página 107. Pruébelo en todas las ramas.

9.5 Determine si se hace referencia de manera apropiada a la variable x en cada una de la siguientes reglas. Explique sus respuestas.

```
(a)  (defrule ejemplo-1
        (not (hecho ?x))
        (test (> ?x 4))
        =>)

(b)  (defrule ejemplo-2
        (not (hecho ?x&:(> ?x 4)))
         =>)

(c)  (defrule ejemplo-3
        (not (hecho ?x))
        (hecho ?y&:(> ?y ?x)))
   =>)

(d)  (defrule ejemplo-4
        (not (hecho ?x))
         (hecho ?x&:(> ?x 4)))
        =>)
```

9.6 Rescriba el programa del modelo de bloques de la sección 8.6, página 371, para que pueda reorganizar los bloques de cualquier estado inicial de bloques apilados a cualquier estado de destino de bloques apilados. Por ejemplo si el estado inicial de los bloques era

```
(pila A B C)
(pila D E F)
```

Un estado de destino posible sería

```
(pila D C B)
(pila A)
(pila F E)
```

9.7 Escriba un programa CLIPS en el que un usuario consultará valores de color y luego imprimirá una lista de todos los países con banderas que contienen todos los colores especificados. A continuación se presenta una lista de colores de bandera para varios países.

País	Colores de bandera
Estados Unidos	Rojo, blanco y azul
Bélgica	Negro, amarillo y rojo
Polonia	Blanco y rojo
Mónaco	Blanco y rojo
Suecia	Amarillo y azul
Panamá	Rojo, blanco y azul
Jamaica	Negro, amarillo y verde
Colombia	Amarillo, azul y rojo
Italia	Verde, blanco y rojo
Irlanda	Verde, blanco y naranja
Grecia	Azul y blanco
Botswana	Azul, blanco y negro

9.8 Dada la siguiente deftemplate que describe un conjunto,

```
(deftemplate conjunto
   (multislot nombre)
   (multislot miembros))
```

escriba una o más reglas que

(a) Calculen la unión de dos conjuntos específicos dado un hecho que utiliza la siguiente deftemplate.

```
(deftemplate unión
   (multislot nombre-conjunto-1)
   (multislot nombre-conjunto-2))
```

(b) Calcule la intersección de dos conjuntos específicos dado un hecho que utiliza la siguiente deftemplate.

```
(deftemplate intersección
   (multislot nombre-conjunto-1)
   (multislot nombre-conjunto-2))
```


Tome en cuenta que cuando se calcula la unión y la intersección, no debe permitirse que aparezcan elementos duplicados en ellas. El resultado final de (a) y (b) debe ser un nuevo hecho de *conjunto* que contenga la unión o la intersección de los dos conjuntos especificados y los hechos *unión* e *intersección* deben retraerse cuando se completen las operaciones.

9.9 Escriba un conjunto de reglas para clasificar formas silogísticas por ambiente y figura. Por ejemplo, la forma silogística

```
Ningún M es P
Algunos M no son S
-----------------
∴ Algunos S son P
```

es del tipo EOI-3. La entrada para las reglas debe ser un solo hecho que represente las premisas mayor y menor y la conclusión. La salida debe ser una instrucción impresa del ambiente y la figura.

9.10 Escriba un programa que lea un archivo de datos que contiene una lista de nombres y edades de personas y elabore un nuevo archivo que contenga la lista en orden ascendente por edad; el programa debe pedir los archivos de entrada y salida, por ejemplo, el archivo de entrada:

```
Felipe Solís 37
Roberto Domínguez 35
Juan González 36
Gerardo García 34
```

debe crear el archivo de salida

```
Gerardo García 34
Roberto Domínguez 35
Juan González 36
Felipe Solís 37
```

9.11 Escriba un programa para calcular el valor de 13 cartas de la mano de un jugador de bridge utilizando el método de conteo de puntos. Los ases valen cuatro puntos, los reyes valen tres, las reinas dos y las sotas uno. Un conjunto vacío (cero cartas de un conjunto) vale tres puntos, un conjunto sencillo (una carta de un conjunto) vale dos puntos y un conjunto doble (dos cartas de un conjunto) vale un punto.

9.12 Escriba un programa que indique la acción que se emprenderá cuando alguien ingiera veneno. El programa debe tener conocimiento de los siguientes venenos: ácidos (como removedor de óxido y yodo), alcalinos (como amoniaco y blanqueador) y derivados del petróleo (como gasolina y trementina), los demás venenos deben agruparse en la categoría *otros*.

En caso de envenenamiento, debe llamarse a un médico o centro de control de envenenamiento. Y, para ácidos, alcalinos y otros tipos de veneno (pero no derivados del petróleo), el veneno se diluye haciendo que la víctima beba un líquido como agua o leche, se induce el vómito con otros tipos de veneno, pero no en ácidos, alcalinos o derivados del petróleo. No se proporcionan líquidos ni se induce el vómito si la víctima está inconsciente o tiene convulsiones.

9.13 Escriba un programa que, dados dos puntos, calcule la pendiente de la línea que los une. Su programa debe verificar que los puntos proporcionados contengan números y que no se especifique dos veces un mismo punto. Trate las líneas verticales como si tuvieran una pendiente infinita.

9.14 Un triángulo escaleno tiene tres lados diferentes, un triángulo isósceles tiene dos lados con la misma longitud y un triángulo equilátero tiene tres lados iguales. Escriba un programa que, cuando se den los tres puntos que forman un triángulo, determine el tipo de triángulo. El programa debe tomar en cuenta posibles errores de redondeo (se asume que dos lados son iguales si la diferencia entre sus longitudes es menor a .00001). Pruebe su programa con los siguientes triángulos.

(a) Puntos (0,0), (2,4) y (6,0)
(b) Puntos (1,2), (4,5) y (7,2)
(c) Puntos (0,0), (3,5.196152) y (6,0)

9.15 Escriba un programa para resolver el problema de las Torres de Hanoi, en ellas se debe mover un conjunto de anillos de una clavija a otra sin poner nunca un anillo más grande encima de un anillo más pequeño. Hay tres clavijas para su uso. El número de anillos debe ser una entrada al programa. En la configuración inicial, todos los anillos están en la primera clavija por tamaño descendente de arriba abajo. El objetivo inicial debe ser mover todos los anillos de la primera clavija a la tercera.

9.16 Escriba un programa para determinar los valores de las letras que hacen que el siguiente problema criptoaritmético sea correcto. Cada una de las letras H, O, C, U, S, P, R, E y T corresponde sólo a un dígito en el rango de 0 a 9.

```
 HOCUS
+ POCUS
-------
= PRESTO
```

9.17 Escriba un programa que proporcione asesoramiento sobre inversión en mutualidades. La salida del programa debe indicar el porcentaje de dinero que se invertirá en fondos con ingresos fijos, que se invierten principalmente en bonos y acciones preferentes, y en fondos accionarios, con un riesgo más elevado pero con posibilidades de obtener mayores ganancias. Los porcentajes se determinarán al "clasificar" la cantidad de riesgo que el inversionista está dispuesto a tomar con base en las respuestas a varias preguntas. Si el inversionista tiene 29 años o menos, se suma 4 a la calificación; de 30 a 39 años, se suma 3; de 40 a 49, se suma 2; de 50 a 59, se suma 1; y de 60 o más, se suma 0. Si al inversionista le restan de 0 a 9 años para el retiro, se suma 0; de 10 a 14, se suma 1; de 15 a 19, se suman 2; de 20 a 24, se suman 3; y 25 o más, se suman 4. Si el inversionista está dispuesto a soportar pérdidas de sólo 5% o menos, se suma 0 a la calificación; de 6 a 10%, se suma 1; de 11 a 15% se suman 2; y de 16% o más, se suman 3. Si el inversionista conoce mucho de inversiones y el mercado accionario, se suma 4 a la calificación; si conoce un poco, se suman 2; si no sabe nada, se suma 0. Si el inversionista está dispuesto a correr un riesgo significativo para obtener ganancias más elevadas, se suman 4 a la calificación; si está dispuesto a correr cierto riesgo, se suman 2; y si el riesgo es mínimo, se suma 0. Si el inversionista cree que sus objetivos de retiro quedarán satisfechos con sus ingresos y activos actuales, se suman 4 a la clasificación; si los objetivos tal vez se cumplan, se suman 2; y si es improbable que se cumplan, se suma 0. Si la calificación inicial es mayor a 20 puntos, entonces debe colocarse 100% de las inversiones en fondos accionarios; si es de 16 a 20 puntos, se debe colocar 80% en fondos accionarios y 20% en fondos con ingresos fijos; si es de 11 a 15 puntos, 60% debe colocarse en fondos accionarios y 40% en fondos fijos; si es de 6 a 10 puntos, entonces 40% debe colocarse en fondos accionarios y 60% en fondos de ingresos fijos; si es de 0 a 5 puntos, entonces 20% debe colocarse en fondos accionarios y 80% en fondos de ingresos fijos.

9.18 Modifique el problema desarrollado en el problema 8.15 de la página 403 de modo que sólo se enumeren los arbustos que tengan todas las características necesarias. Por ejemplo, si el usuario especifica que la planta debe tolerar frío y sequía, entonces sólo deben enumerarse el laurel del norte y el junípero común; si ninguno de los arbustos cumple con todos los requisitos, entonces debe imprimirse un mensaje que indique esto.

9.19 Modifique el programa desarrollado en el problema 7.2 de la página 362, de modo que la información sobre las estrellas se represente utilizando hechos. La salida del programa debe estar en el mismo orden: todas las estrellas con la clase espectral especificada, todas las estrellas con la magnitud especificada y al final todas las estrellas comparando la clase espectral y la magnitud junto con su distancia a la Tierra en años-luz.

9.20 Modifique el programa desarrollado en el problema 8.17 de la página 403 de modo que se imprima el precio total de la configuración. Debe imprimirse un mensaje de advertencia cuando se seleccionen más accesorios de los que sea posible conectar o si la energía requerida por ellos excede la proporcionada por el chasis.

9.21 Modifique el programa desarrollado en el problema 8.18 de la página 404 para incluir las joyas restantes encontradas en la tabla de joyas del problema 7.13 de la página 363. En joyas que tienen un solo valor numérico por su dureza o densidad, modifique las reglas de modo que sea aceptable cualquier valor dentro de .01 del valor especificado. Incluya reglas para verificar y volverle a indicar al usuario si la dureza no está comprendida entre 1 y 10 y la densidad no está comprendida entre 1 y 6.

CAPÍTULO 10

Diseño modular y control de ejecución

10.1 INTRODUCCIÓN

En este capítulo se introducen varias funciones de CLIPS útiles para el desarrollo y mantenimiento de los sistemas expertos, como las características de deftemplate permiten el seguimiento de las restricciones de valor para sus valores de ranura, pues cuando se carga una regla, estas cualidades pueden detectar los errores semánticos que impiden que se compare el lado izquierdo de esa regla. Utilizando la prominencia, que proporciona un método para jerarquizar las reglas, y los hechos que representan el flujo de conocimiento de control, en este capítulo se demuestran las técnicas para controlar la ejecución de un programa de CLIPS. Además se analiza la construcción defmodule, que permite la división de una base de conocimiento y proporciona un método más explícito para controlar la ejecución de un sistema.

10.2 ATRIBUTOS DE DEFTEMPLATE

CLIPS proporciona varios atributos de ranura que pueden especificarse cuando se definen las ranuras de una deftemplate, estos atributos ayudan al desarrollo y mantenimiento de un sistema experto porque confirman la escritura y verifican las restricciones. Es posible definir los tipos y valores permitidos que pueden almacenarse en una ranura, para los valores numéricos, puede especificarse el rango de números permitido, las ranuras múltiples pueden especificar el número mínimo y máximo de campos que pueden contener, y por último, el atributo predeterminado proporciona un valor de ranura que habrá de utilizarse en las ranuras que se especifican en un comando *assert*.

Atributo type

El **atributo type** define los tipos de datos que pueden colocarse en una ranura, su formato general es (type <especificación-tipo>), donde <especificación-tipo> es ?VARIABLE o uno o más de los símbolos SYMBOL, STRING, LEXEME, INTERGER, FLOAT o NUMBER. Si se usa ?VARIABLE, la ranura puede contener cualquier tipo de datos (que es el comportamiento predeterminado para todas las ranuras). Si se utilizan una o más de las especificaciones simbólicas, la ranura se limita al tipo especificado; la especificación LEXEME es equiva-

lente a especificar SYMBOL y STRING, la especificación NUMBER es equivalente a especificar INTEGER y FLOAT.

La siguiente deftemplate *persona* restringe los valores almacenados en la ranura *nombre* a símbolos y los valores almacenados en la ranura de *edad* a enteros:

```
(deftemplate persona
   (multislot nombre (type SYMBOL))
   (slot edad (type INTEGER)))
```

Una vez que se ha definido esta deftemplate, CLIPS hace valer automáticamente las restricciones de cualquier atributo de ranura, por ejemplo, la asignación del símbolo *cuatro* a la ranura edad en lugar del entero 4 provocará un error, como se muestra a continuación:

```
CLIPS> (assert (persona (nombre Alfredo Salas)
                        (edad cuatro)))↵

[CSTRNCHK1] A literal slot value found in the assert
command does not match the allowed types for slot edad.
CLIPS>
```

CLIPS también verificará la congruencia de los enlaces de variable en ambos lados de la regla. Por ejemplo, si hay una regla que actualiza la ranura *edad* de una persona cada vez que cumple años, la deftemplate para el hecho de control que indica que una persona acaba de cumplir años es

```
(deftemplate tuvo-cumpleaños
   (slot nombre (type STRING)))
```

Los tipos permitidos de las ranuras nombre para las dos deftemplates son incongruentes por lo que un intento por comparar directamente las dos ranuras generará un error, como se ilustra en el siguiente diálogo:

```
CLIPS>
(defrule actualizar-cumpleaños
   ?f1 <- (tuvo-cumpleaños (nombre ?nombre))
   ?f2 <- (persona (nombre ?nombre) (edad ?edad))
   =>
   (retract ?f1)
   (modify ?f2 (edad (+ ?edad 1))))↵

[RULECSTR1] Variable ?nombre in CE #2 slot name has constraint
conflicts which make the pattern unmatchable.

ERROR:
(defrule MAIN::actualizar-cumpleaños
   ?f1 <- (tuvo-cumpleaños (nombre ?nombre))
   ?f2 <- (persona (nombre ?nombre) (edad ?edad))
   =>
```



```
   (retract ?f1)
   (modify ?f2 (edad (+ ?edad 1))))
CLIPS>
```

Aquí la ranura *nombre* para el hecho *tuvo-cumpleaños* debe ser una cadena y la ranura *nombre* para el hecho *persona* debe ser un símbolo, entonces no es posible que la variable ?nombre satisfaga ambas restricciones y, por tanto, el lado izquierdo de la regla nunca puede satisfacerse.

Verificación de restricción estática y dinámica

CLIPS proporciona dos niveles de verificación de restricciones. El primer nivel, la **verificación de restricción estática**, se realiza como opción predeterminada cuando CLIPS analiza una expresión o restricción y se ilustra con los ejemplos de la violación de restricciones que utilizan el atributo type ya expuestos. La verificación de restricción estática puede desactivarse al llamar a la función **set-static-constraint-checking** y convertirlo a *FALSE*, por el contrario, la llamada de esta función con el símbolo *TRUE* la activará, el valor que devuelva es el valor de estado previo (el símbolo *FALSE* si se inhabilitó previamente y el símbolo *TRUE* en caso contrario). Se puede determinar el estado actual de verificación de restricción estática al llamar a la función **get-static-constraint-checking** (que devuelve el símbolo *TRUE* si está habilitada y el símbolo *FALSE* en caso contrario).

No siempre es posible determinar todos los errores de restricción durante el análisis gramatical (o sintáctico). Por ejemplo, en la regla *crear-persona* que se muestra aquí, las variables ?edad y ?nombre pueden unirse a valores ilícitos:

```
(defrule crear-persona
   =>
   (printout t "¿Cuál es su nombre? ")
   (bind ?nombre (explode$ (readline)))
   (printout t "¿Cuál es su edad? ")
   (bind ?edad (read))
   (assert (persona (nombre ?nombre) (edad ?edad))))
```

La función *readline* se utiliza para introducir el nombre completo de una persona como una cadena, luego la función *explode$* la convierte en un valor de campo múltiple que puede colocarse en la ranura *nombre*. La función *read* se utiliza para introducir la edad de una persona, que puede colocarse en la ranura de *edad*, pero en ambos valores de entrada, es posible recibir valores ilícitos. Por ejemplo, puede introducirse el símbolo *cuatro* para la edad de la persona, como se muestra a continuación:

```
CLIPS> (reset)↵
CLIPS> (run)↵
¿Cuál es tu nombre? Alfredo Salas↵
¿Cuál es tu edad? cuatro↵
CLIPS> (facts)↵
f-0     (hecho-inicial)
f-1     (persona (nombre Alfredo Salas) (edad cuatro))
For a total of 2 facts
CLIPS>
```

Observe que el mismo hecho *persona* para Alfredo Salas, que antes violó una restricción y no se agregó a la lista de hechos, ahora se agrega. Esto se debe a que el segundo nivel de verificación de restricción efectuado por CLIPS, **verificación de restricción dinámica**, está inhabilitado como opción predeterminada. La verificación de restricción dinámica se realiza sobre hechos cuando se afirman en realidad, con lo que se detectan errores que no pueden localizarse en el momento del análisis sintáctico.

La verificación de restricción dinámica puede habilitarse o inhabilitarse utilizando la función **set-dynamic-constraint-checking** y es posible determinar su estado actual mediante la función **get-dynamic-constraint-checking**. En el siguiente diálogo se muestra como se maneja la violación de restricción cuando está habilitada la verificación de restricción dinámica. El hecho *persona* de Alfredo Salas todavía está incluido en la lista de hechos, pero se detecta la violación a la restricción y se detiene la ejecución.

```
CLIPS> (set-dynamic-constraint-checking TRUE)↵
FALSE
CLIPS> (reset)↵
CLIPS> (run)↵
¿Cuál es tu nombre? Alfredo Salas↵
¿Cuál es tu edad? cuatro↵

[CSTRNCHK1] Slot value (Alfredo Salas) found in fact f-1
does not match the allowed types for slot age.
[PRCCODE4] Execution halted during the actions of defrule
crear-persona.
CLIPS> (facts)↵
f-0     (hecho-inicial)
f-1     (persona (nombre Alfredo Salas) (edad cuatro))
For a total of 2 facts.
CLIPS>
```

Atributos allowed value

Además de restringir los tipos permitidos con el atributo type, CLIPS también permite especificar una lista de valores permitidos para un tipo específico, por ejemplo, si se agrega una ranura *género* a la deftemplate *persona*, los símbolos permitidos para esa ranura pueden restringirse a *masculino* y *femenino*:

```
(deftemplate persona
   (multislot nombre (type SYMBOL))
   (slot edad (type INTEGER))
   (slot género (type SYMBOL)
                (allowed-symbols masculino femenino)))
```

Hay siete diferentes atributos de valor permitido que proporciona CLIPS: **allowed-symbols, allowed-strings, allowed-lexems, allowed-integers, allowed-floats, allowed-numbers** y **allowed-values**. Cada uno de ellos debe ir seguido por ?VARIABLE (que indica que cualquier valor del tipo especificado es válido) o por una lista de valores de tipo seguido del prefijo *allowed-*. Por ejemplo, el atributo *allowed-lexemes* debe ir seguido por ?VARIABLE, por una lista de símbolos o cadenas, o ambas. El atributo allowed-value predeterminado para las ranuras es (allowed-values ?VARIABLE).

Observe que los atributos de valor permitido no restringen los tipos permitidos de una ranura, por ejemplo (allowed-symbols masculino femenino) no restringe el tipo de la ranura *género* a un símbolo, sólo indica que si el valor de esa ranura es un símbolo, entonces éste debe ser: *masculino* o *femenino*. Si se elimina el atributo (type SYMBOL) cualquier cadena, entero o número con punto decimal sería un valor lícito para la ranura *género*.

El atributo allowed-values puede utilizarse para restringir por completo una lista específica del conjunto de valores permitidos para una ranura. Por ejemplo, el cambio de la deftemplate *persona* como se muestra a continuación limita los tipos permitidos para la ranura *género* a símbolos:

```
(deftemplate persona
   (multislot nombre (type SYMBOL))
   (slot edad (type INTEGER))
   (slot género (allowed-values masculino femenino)))
```

Atributo range

El **atributo range** permite especificar los valores numéricos mínimo y máximo, su formato general es (range <límite-inferior> <límite superior>), donde <límite inferior> y <límite-superior> son ?VARIABLE o un valor numérico. El término <límite-inferior> indica el valor mínimo para la ranura, <límite-superior> indica el valor máximo y ?VARIABLE indica que no hay valor mínimo ni máximo (dependiendo de si es primero o segundo). Por ejemplo, la ranura *edad* de la deftemplate *persona* puede modificarse para evitar que se coloquen valores negativos en ella:

```
(deftemplate persona
   (multislot nombre (type SYMBOL))
   (slot edad (type INTEGER) (range 0 ?VARIABLE)))
```

si se está dispuesto a asumir que nadie vive más de 125 años, entonces el atributo range cambiaría a (range 0 125). Al igual que con los atributos de valor permitido, range no restringe el tipo de un valor de ranura a un valor numérico, sólo restringe los valores numéricos permitidos de una ranura al rango especificado si el valor de la ranura es numérico. El atributo de range predeterminado para las ranuras es (range ?VARIABLE ?VARIABLE).

Atributo cardinality

El **atributo cardinality** permite especificar el número mínimo y máximo de valores que pueden almacenarse en una ranura múltiple, su formato general es (cardinality <límite-inferior> <límite-superior>), donde <límite-inferior> y <límite-superior> son ?VARIABLE o un entero positivo. El término <límite-inferior> indica el número mínimo de valores que puede contener la ranura y el término <límite-superior> indica el número máximo. ?VARIABLE indica que no hay un número mínimo o máximo de valores (dependiendo de si se ubican primero o segundo); el atributo de cardinality predeterminado para una multirranura es (cardinality ?VARIABLE ?VARIABLE). La siguiente deftemplate podría utilizarse para representar los

equipos de volibol de una empresa, que deben tener seis jugadores y pueden incluir hasta dos jugadores de reserva. (Observe que los atributos type, allowed value y range se aplican a cada valor contenido en una ranura múltiple.)

```
(deftemplate equipo-volibol
   (slot nombre (type STRING))
   (multislot jugadores (type STRING)
                        (cardinality 6 6))
   (multislot alternos (type STRING)
                       (cardinality 0 2)))
```

Atributo default

En los capítulos anteriores, cada hecho de deftemplate que se afirmó siempre tuvo un valor explícito establecido para cada ranura, pues a menudo es conveniente tener especificado automáticamente un valor almacenado en una ranura si no se establece ningún valor de manera explícita en un comando *assert*. El **atributo default** permite especificar el valor predeterminado, su formato general es (default <especificación-predeterminada>), donde <especificación-predeterminada> es ?DERIVE, ?NONE, una expresión sencilla (para una ranura de un solo campo) o cero o más expresiones (para una ranura de varios campos).

Si se especifica ?DERIVE en el atributo default, entonces se deduce para la ranura un valor que satisfaga todos sus atributos; si no se especifica entonces se supone que es (default ?DERIVE). Para una ranura de un solo campo, esto significa que se selecciona un valor que satisface los atributos type, range y allowed values para la ranura. El valor predeterminado derivado para una ranura de varios campos será una lista de valores idénticos, que son la cardinalidad mínima permitida para la ranura (cero como opción predeterminada). Si están contenidos uno o más valores en el valor predeterminado para una ranura de varios campos, entonces cada valor cumplirá con los atributos type, range y allowed values. A continuación se muestra un ejemplo de valores derivados:

```
CLIPS> (clear)↵
CLIPS>
(deftemplate ejemplo
   (slot a)
   (slot b (type INTEGER))
   (slot c (allowed-values rojo verde azul))
   (multislot d)
   (multislot e (cardinality 2 2)
                (type FLOAT)
                (range 3.5 10.0)))↵
CLIPS> (assert (ejemplo))↵
<Fact-0>
CLIPS> (facts)↵
f-0     (ejemplo (a nil)
                 (b 0)
                 (c rojo)
                 (d)
                 (e 3.5 3.5))
For a total of 1 fact.
CLIPS>
```


CLIPS sólo garantiza que el valor predeterminado derivado para una ranura satisface los atributos de restricción para ella, en otras palabras, sus programas no deben depender de los valores derivados específicos (como el símbolo *nil* para la ranura *a* o el entero 0 para la ranura *b* en el ejemplo anterior) que se colocan en las ranuras. Si un programa depende de un valor predeterminado específico, debe utilizarse una expresión para el atributo default (que se explicará más adelante).

Si se especifica ?NONE en el atributo default, debe proporcionarse un valor para la ranura cuando se afirme el hecho, es decir, no hay un valor predeterminado. Por ejemplo:

```
CLIPS> (clear)↵
CLIPS>
(deftemplate ejemplo
  (slot a)
   (slot b (default ?NONE)))↵
CLIPS> (assert) (ejemplo))↵

[TMPLTRHS1] Slot b requires a value because of its
(default ?NONE) attribute.
CLIPS> (assert (ejemplo (b 1)))↵
<Fact-0>
CLIPS> (facts)↵
f-0     (ejemplo (a nil) (b 1))
For a total of 1 fact.
CLIPS>
```

Si se utilizan una o más expresiones con el atributo default, las expresiones se evalúan cuando se examina la ranura y, cada vez que no se especifica en un comando *assert*, este valor se almacena en ella. El atributo default para una ranura de un solo campo debe contener exactamente una expresión. Si no se especifican expresiones en el atributo *default* para una ranura de campo múltiple, entonces se utiliza un campo múltiple de longitud cero para el valor predeterminado. De lo contrario, los valores de devolución de todas las expresiones se agrupan para formar un valor de campo múltiple. El siguiente es un ejemplo en el que se utilizan expresiones con el atributo *default*:

```
CLIPS> (clear)↵
CLIPS>
(deftemplate ejemplo
   (slot a (default 3))
   (slot b (default (+ 3 4)))
   (multislot c (default a b c))
   (multislot d (default (+ 1 2) (+ 3 4))))↵
CLIPS> (assert) (ejemplo))↵
<Fact-0>
CLIPS> (facts)↵
f-0     (ejemplo (a 3) (b 7) (c a b c) (d 3 7))
For a total of 1 fact.
CLIPS>
```

Atributo default-dynamic

Cuando se utiliza el atributo default, se determina el valor predeterminado para una ranura cuando se examina la definición de la misma. También es posible hacer que se genere el valor predeterminado cuando se afirme el hecho que utilizará tal valor con el **atributo default-dynamic**. Cuando se deja sin especificar el valor de una ranura que utiliza el atributo default-dynamic en un comando de assert, se evalúa la expresión especificada con el atributo default-dynamic y se emplea para el valor de la ranura.

Por ejemplo, consideremos el problema de eliminar hechos después de que ha transcurrido una determinada cantidad de tiempo: en primer lugar, es necesario contar con alguna manera de saber cuándo se afirmaron los hechos, por lo que se utilizará la función **time** proporcionada por CLIPS para etiquetarlos con su momento de creación; esto también devolverá el número de segundos que han transcurrido desde un momento de referencia dependiente del sistema. El valor de devolución de la función time no es significativo por sí mismo, sólo es útil cuando se le compara con otros valores devueltos por la función. Se utilizará la siguiente deftemplate para este ejemplo; contiene una ranura *hora-creación*, y una ranura *valor*, para almacenar un valor asociado con el hecho.

```
(deftemplate datos
   (slot hora-creación (default-dynamic (hora)))
   (slot valor))
```

Cada vez que se crea un hecho *datos* y no se especifica la ranura *hora-creación*, se llama a la función *time* y el valor se almacena en esa ranura.

```
CLIPS> (watch facts)↵
CLIPS> (assert (datos (valor 3)))↵
==> f-0     (datos (hora-creación 12002.45)
            (valor 3))
<Fact-0>
CLIPS> (assert (datos (valor b))) ↵
==> f-1     (datos (hora-creación 12010.25)
            (valor b)
<Fact-1>
CLIPS> (assert (datos (valor c))) ↵
==> f-2     (datos (hora-creación 12018.65)
            (valor c)
<Fact-2>
CLIPS>
```

Suponiendo que se afirmó un hecho *hora-actual* y se actualizó mediante otras reglas para contener la hora actual del sistema, la siguiente regla retraerá los hechos *datos* que se afirmaron hace más de un minuto:

```
(defrule retraer-hechos-datos-después-un-minuto
   ?f <- (datos (hora-creación ?t1))
   (hora-actual ?t2)
   (test (> (- ?t2 ?t1) 60))
   =>
   (retract ?f))
```


Observe que el siguiente cambio en la regla *retraer-hechos-datos-después-un-minuto* no produce los mismos resultados.

```
(defrule retraer-hechos-datos-después-un-minuto
   ?f <- (datos (hora-creación ?t1))
    (test (> (- (time) ?t1) 60))
   =>
   (retract ?f))
```

La función *time* del elemento condicional (EC) *test* sólo se verificará cuando un hecho *datos* coincida con el primer patrón. Como el valor devuelto será casi la misma hora que el valor de la ranura *hora-creación*, la regla no será satisfecha. CLIPS no verifica de manera continua los condicionales *test* para determinar si se evalúan a un valor diferente; sólo se verifican cuando ocurren cambios en los condicionales precedentes; esto se debe a que el hecho *hora-actual* de la versión original de la regla debe actualizarse para verificar de manera periódica el condicional *test*.

Atributos de ranura conflictivos

CLIPS no permite especificar atributos conflictivos para una ranura, por ejemplo, un valor predeterminado de ranura debe satisfacer los atributos type, allowed-..., range y cardinality de la misma. Si se especifica un atributo allowed-..., el tipo asociado con el atributo debe satisfacer el tipo de atributo de la ranura. Los atributos allowed-numbers, allowed-integers y allowed-floats pueden dejar de utilizarse con el atributo range.

10.3 PROMINENCIA

Hasta este punto se han utilizado hechos de control para controlar indirectamente la ejecución de programas. CLIPS proporciona dos técnicas explícitas para controlar la ejecución de reglas: prominencia y módulos. El control de ejecución de reglas utilizando módulos se analizará más adelante en este mismo capítulo. El uso de la palabra clave ***salience*** permite especificar de manera explícita la prioridad de las reglas. En condiciones normales, la agenda actúa como una pila. Es decir, la activación de más reciente ubicación en la agenda es la primera en dispararse. La prominencia permite que las reglas más importantes permanezcan en la parte superior de la agenda, independientemente del momento en que se agregaron, de esta manera las reglas de menor prominencia se colocan debajo de las reglas de mayor prominencia dentro de la agenda.

La prominencia se establece utilizando un valor numérico que va del valor más pequeño de −10,000 al mayor de 10,000. Si el programador no ha asignado de manera explícita la prominencia de una regla, CLIPS supone una prominencia de 0. Observe que una prominencia de 0 es el punto medio entre los valores de prominencia más grande y más pequeño, esto no significa que la regla no tenga prominencia, sino que más bien tiene un nivel de prioridad inmediata. Una regla recién activada se coloca en la agenda antes que todas las reglas con igual o menor prominencia y después de todas las reglas con mayor prominencia.

Uno de los usos de la prominencia es la de forzar a las reglas para que se disparen en forma secuencial. Consideremos el siguiente conjunto de reglas, en donde no se declararon valores de prominencia:

```
(defrule disparar-primero
   (primera prioridad)
   =>
   (printout t "Imprimir primero crlf))
```

```
(defrule disparar-segundo
   (segunda prioridad)
   =>
   (printout t "Imprimir segundo" crlf))

(defrule disparar-tercero
   (tercera prioridad)
   =>
   (printout t "Imprimir tercero" crlf))
```

El orden en que se disparan las reglas depende del orden en que se afirman los hechos que satisfacen los condicionales del lado izquierdo de las reglas. Por ejemplo, si se introducen las reglas, los siguientes comandos producirán la salida que se muestra a continuación:

```
CLIPS> (unwatch all)↵
CLIPS> (reset)↵
CLIPS> (assert (primera prioridad))↵
<Fact-1>
CLIPS> (assert (segunda prioridad))↵
<Fact-2>
CLIPS> (assert (tercera prioridad))↵
<Fact-3>
CLIPS> (run)↵
Imprimir tercero
Imprimir segundo
Imprimir primero
CLIPS>
```

Observe el orden de las instrucciones de salida: primero se imprime "Imprimir tercero", luego "Imprimir segundo" y, por último, "Imprimir primero". El primer hecho (primera prioridad) activa la regla *disparar-primero*, cuando se afirma el segundo hecho, se activa la regla *disparar-segundo*, que se apila en la parte superior de la activación de la regla *disparar-primero*, y por último, se afirma el tercer hecho y su regla activada, *disparar-tercero*, se apila en la parte superior de la activación de la regla *disparar-segundo*.

En CLIPS, el orden de pila de hechos sirve para asignar prioridades a las reglas de igual prominencia que se disparan a partir de diferentes patrones. En la agenda, las reglas se disparan de la parte superior de la pila hacia abajo, de modo que primero se dispara la regla *disparar-tercero* porque está en la parte superior de la pila, luego la regla *disparar-segundo* y al final la regla *disparar-primero*. Si se invierte el orden en que se afirman los hechos, también se invertirá el orden en que se disparan las reglas, lo que se observa en la siguiente salida:

```
CLIPS> (reset)↵
CLIPS> (assert (tercera prioridad))↵
<Fact-1>
CLIPS> (assert (segunda prioridad))↵
<Fact-2>
CLIPS> (assert (primera prioridad))↵
<Fact-3>
CLIPS> (run) ↵
Imprimir primero
```



```
Imprimir segundo
Imprimir tercero
CLIPS>
```

Es importante conocer que si el mismo hecho activa dos o más reglas con la misma prominencia, no se garantiza el orden en que las reglas se colocarán en la agenda.

La prominencia puede utilizarse para forzar que las reglas se disparen en el orden *disparar-primero, disparar-segundo, disparar-tercero*, a pesar del orden en que se afirman los hechos de activación; esto puede realizarse al declarar valores de prominencia:

```
(defrule disparar-primero
   (declare (salience 30))
   (primera prioridad)
   =>
   (printout t "Imprimir primero" crlf))

(defrule disparar-segundo
   (declare (salience 20))
   (segunda prioridad)
   =>
   (printout t "Imprimir segundo" crlf))

(defrule disparar-tercero
   (declare (salience 10))
   (tercera prioridad)
   =>
   (printout t "Imprimir tercero" crlf))
```

Independientemente del orden en que se afirman los hechos de prioridad, la agenda siempre se ordenará de la misma manera. Si se aplica el comando *agenda* después de asegurar los hechos de prioridad, se produciría la siguiente salida:

```
CLIPS> (reset)↵
CLIPS> (assert (segunda prioridad)
       (primera prioridad)
       (tercera prioridad) ↵
<Fact-3>
CLIPS> (agenda)↵
30     disparar-primero: f-2
20     disparar-segundo: f-1
10     disparar-tercero: f-3
For a total of 3 activations.
CLIPS>
```

Observe la manera en que los valores de prominencia han reorganizado la prioridad de las reglas en la agenda. Cuando se ejecute el programa, el orden de disparo de las reglas siempre será *disparar-primero, disparar-segundo, disparar-tercero*.

10.4 FASES Y HECHOS DE CONTROL

El concepto más puro de un sistema experto basado en reglas es que las reglas actúan de manera oportuna cada vez que son aplicables. Sin embargo, la mayor parte de ellos tienen algún aspecto orientado a procedimientos. El programa Palillos, por ejemplo, tenía diferentes reglas que se aplicaban dependiendo de si se trataba del movimiento del humano o de la computadora. El control de este programa se manejó mediante hechos que indicaban de quién era turno. Estos hechos de control permiten que se inserte información de la estructura de control del programa en las reglas de conocimiento de dominio, lo que representa una desventaja: el conocimiento sobre el control de las reglas se entremezcla con el conocimiento sobre cómo jugar el juego. Esto no es un inconveniente importante en el caso del programa Palillos porque es pequeño, pero, en programas que incluyen cientos o miles de reglas, el entremezclado del conocimiento de dominio y el conocimiento de control hace que el desarrollo y mantenimiento representen un problema importante.

Por ejemplo, consideremos el problema de realizar la **detección, aislamiento** y **recuperación de fallas** de un dispositivo electrónico. La *detección de fallas* es el proceso de reconocer que el dispositivo electrónico no está funcionando de manera apropiada; el *aislamiento* es el proceso de determinar los componentes del dispositivo que han causado la falla y la *recuperación* es el proceso de determinar los pasos necesarios para corregirlo, si es posible. Por lo general, en este tipo de problema el sistema experto tendrá reglas para determinar si ha ocurrido un fallo, otras reglas para aislar la causa y otras más para determinar cómo solucionarlo, luego el ciclo se repetirá. En la figura 10.1 se muestra un ejemplo del flujo de control en este tipo de sistema.

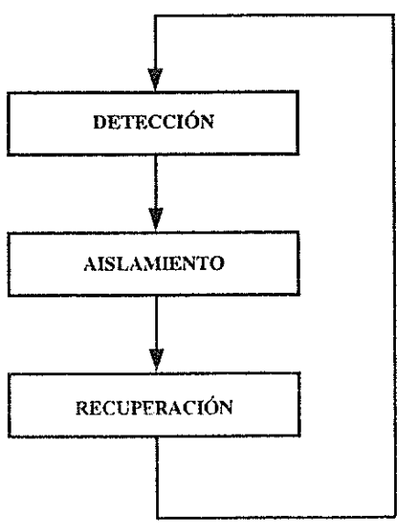


Figura 10.1 Las diferentes fases de un problema para detección, aislamiento y recuperación de fallas

Establecer el flujo de control en este sistema puede realizarse a través de cuando menos cuatro formas, los primeros tres métodos utilizan la prominencia y se analizarán en esta sección; el cuarto utiliza módulos y se analizará más adelante en este mismo capítulo.

El primer método para establecer el flujo de control es colocar el conocimiento de control directamente en las reglas, por ejemplo, las reglas de detección incluirían aquellas que indiquen el momento en que debe introducirse la fase de aislamiento, a cada grupo de reglas se le asignaría un patrón que indicara en qué fase sería aplicable. Esta técnica tiene dos desventajas: en primer lugar, como ya se mencionó, el conocimiento de control se coloca en las reglas de conocimiento de dominio, lo que hace que sean más difíciles de comprender; en segundo lugar, no siempre es fácil determinar el momento en que se completa una fase, pues por lo general esto requiere la escritura de una regla que sólo sea aplicable cuando se hayan disparado todas las demás.

El segundo método consiste en utilizar la prominencia para organizar las reglas, como se muestra en la figura 10.2. Este método también tiene dos desventajas importantes. En primer lugar, el conocimiento de control también se coloca en las reglas que utilizan la prominencia; en segundo lugar, este método no garantiza el orden correcto de la ejecución, pues las reglas de detección siempre se dispararán antes de las reglas de aislamiento. Sin embargo, cuando se disparan las reglas de aislamiento pueden provocar la activación de una regla de detección y su disparo inmediato debido a su mayor prominencia.

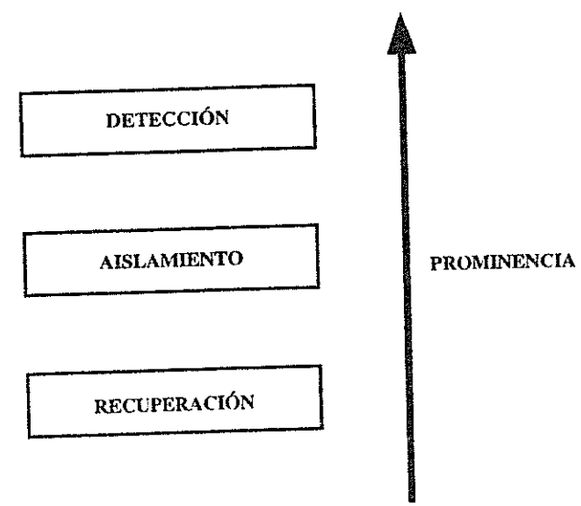


Figura 10.2 Asignación de prominencia para las diferentes fases

Un tercer y mejor método para controlar el flujo de ejecución consiste en separar el conocimiento de control del conocimiento de dominio, como se muestra en la figura 10.3. Utilizando este método, a cada regla se le asigna un patrón de control que indica su fase aplicable, luego se escriben reglas de control para transferir el control entre las diferentes fases, como se muestra a continuación:

```
(defrule detección-a-aislamiento
   (declare (salience -10))
   ?fase <- (fase de detección)
   =>
   (retract ?fase)
   (assert (fase de aislamiento)))

(defrule aislamiento-a-recuperación
   (declare (salience -10))
   ?fase <- (fase de aislamiento)
   =>
   (retract ?fase)
   (assert (fase de recuperación)))

(defrule recuperación-a-detección
   (declare (salience -10))
   ?fase <- (fase de recuperación)
   =>
   (retract ?fase)
   (assert (fase de detección)))
```

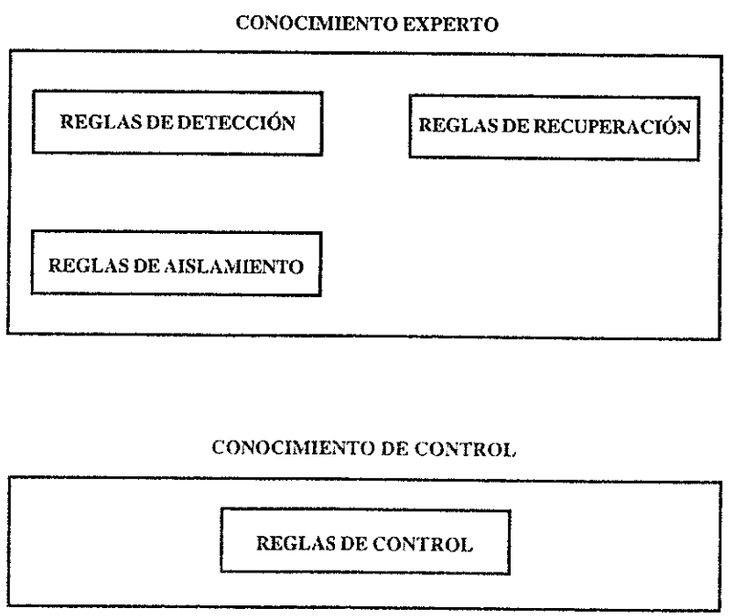


Figura 10.3 Separación del conocimiento experto del conocimiento de control

Luego, a cada una de las reglas aplicables para una fase particular se les proporciona un patrón de control que confirma que está presente el hecho apropiado. Por ejemplo, una regla de recuperación puede ser semejante a esto:

```
(defrule encontrar-ubicación-fallas-y-recuperación
   (fase de recuperación)
   (solución-recuperación dispositivo-conmutación
                          ?reemplazo on)
   =>
   (printout t "Dispositivo de conmutación " ?reemplazo
   " encendido "
            crlf
```

Una **jerarquía de prominencia** es una descripción de los valores de prominencia utilizados por un sistema experto; cada nivel de una jerarquía corresponde a un conjunto específico de reglas a cuyos miembros se les proporciona la misma prominencia. Si a las reglas de detección, aislamiento y recuperación se les proporciona una prominencia predeterminada de cero, entonces la jerarquía de prominencia es como se muestra en la figura 10-4. Observe que mientras el hecho (fase de detección) está en la lista de hechos, la regla *detección-a-aislamiento* estará en la agenda, y como tiene una prominencia menor que las reglas de detección, no se disparará hasta que todas las reglas de detección hayan tenido oportunidad de dispararse. La siguiente salida muestra una ejecución de las tres reglas de control anteriores:

```
CLIPS> (reset)↵
CLIPS> (assert (fase de detección))↵
<Fact-1>
CLIPS> (watch rules)↵
CLIPS> (run 10)↵
FIRE    1 detección-a-aislamiento: f-1
FIRE    2 aislamiento-a-recuperación: f-2
FIRE    3 recuperación-a-detección: f-3
FIRE    4 detección-a-aislamiento: f-4
FIRE    5 aislamiento-a-recuperación: f-5
FIRE    6 recuperación-a-detección: f-6
FIRE    7 detección-a-aislamiento: f-7
FIRE    8 aislamiento-a-recuperación: f-8
FIRE    9 recuperación-a-detección: f-9
FIRE   10 detección-a-aislamiento: f-10
CLIPS>
```

Observe que las reglas de control sólo se mantienen disparándose en secuencia porque no hay reglas de conocimiento de dominio que se apliquen durante cualquiera de las fases; si las hubiera, entonces las reglas de conocimiento de dominio se aplicarían en reglas activadas durante las fases apropiadas.

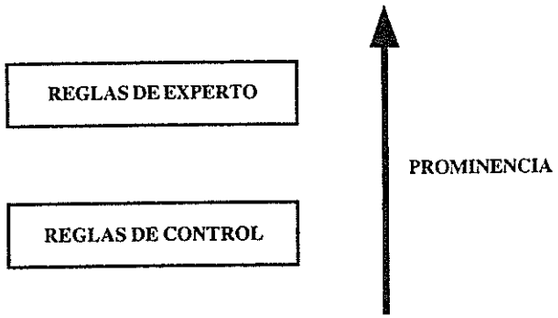


Figura 10.4 Jerarquía de prominencia utilizando reglas de experto y de control

Las anteriores reglas de control podrían escribirse de manera más genérica con una construcción de deffacts y una regla sencilla como:

```
(deffacts información-control
   (fase de detección)
   (fase-después detección aislamiento)
   (fase-después aislamiento recuperación)
   (fase-después recuperación detección))

(defrule cambiar-fase
   (declare (salience -10))
   ?fase <- (fase ?fase-actual)
   (fase-después ?fase-actual ?fase-siguiente)
   =>
   (retract ?fase)
   (assert (fase ?fase-siguiente)))
```

o podrían escribirse utilizando una secuencia de fases para producirse en ciclos como:

```
(deffacts información-control
   (fase de detección)
   (fase-secuencia aislamiento)
   (fase-después aislamiento recuperación detección))

(defrule cambiar-fase
   (declare (salience -10))
   ?fase <- (fase ?fase-actual)
   ?lista <- (fase-secuencia ?fase-siguiente
                             $?otra-fase)
   =>
   (retract ?fase ?lista)
   (assert (fase ?fase-siguiente))
   (assert (fase-secuencia ?otra-fases
                           ?fase-siguiente)))
```

Pueden agregarse con facilidad niveles adicionales a la jerarquía de prominencia: en la figura 10.5 se muestra una jerarquía con dos niveles adicionales. Las reglas de restricción

representan reglas que detectan estados ilícitos o improductivos que puedan ocurrir en el sistema experto, por ejemplo, un sistema experto que asigna gente a varias tareas puede producir un programa que viole una restricción. En lugar de permitir que las reglas de menor prominencia sigan funcionando en el programa, las reglas de restricción eliminarán de inmediato las violaciones al programa. Otro ejemplo sería si el usuario introduce, en respuesta a una serie de preguntas una serie de valores válidos que dan como resultado la generación de un valor ilícito, entonces las reglas de restricción podrían utilizarse para detectar estas violaciones.

Las reglas de consulta que se muestran en el diagrama representan reglas que formulan preguntas particulares a los usuarios para ayudar al sistema experto a determinar una respuesta, estas reglas tienen menor prominencia que las reglas de experto porque no es deseable formular al usuario una pregunta que pueda determinarse por las reglas del experto. Por tanto, las reglas de consulta sólo se disparan cuando las reglas del experto no pueden deducir más información.

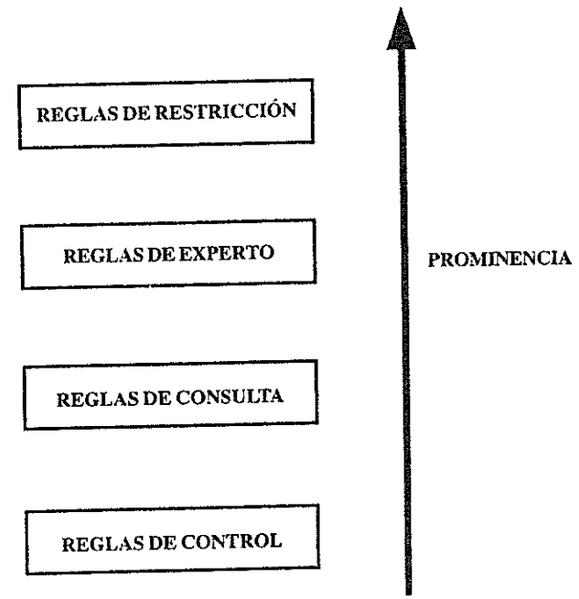


Figura 10.5 Jerarquía de prominencia de cuatro niveles

10.5 USO INAPROPIADO DE LA PROMINENCIA

Aunque la prominencia es una herramienta eficaz para controlar la ejecución, es posible abusar de ella con facilidad. En particular, las personas que apenas están aprendiendo programación basada en reglas tienden a usar en exceso la prominencia porque les proporciona control explícito de la ejecución. Es mejor utilizar la programación de procedimientos, en la que las instrucciones se ejecutan de manera secuencial.

El uso excesivo de la prominencia produce un programa codificado de manera deficiente. La principal ventaja de un programa basado en reglas es que el programador no tiene que preocuparse por la ejecución de control, un programa basado en reglas bien diseñado tiene un modo natural de ejecución que permite que el mecanismo de inferencia guíe el disparo de reglas de manera óptima.

La prominencia debe emplearse de manera primordial como un mecanismo para determinar el orden en que se disparan las reglas; esto significa que, en general, una regla que se coloca en la agenda eventualmente se disparará. La prominencia no debe usarse como un método para elegir una sola regla de un grupo de reglas cuando pueden utilizarse patrones que expresen los criterios de selección; tampoco debe utilizarse como una "solución rápida" para hacer que las reglas se disparen en un orden apropiado.

En general, *cualquier* valor de prominencia utilizado en una regla debe corresponder a un nivel en la jerarquía de prominencia del sistema experto; de hecho el rango de valores de prominencia de –10,000 a 10,000 es un poco confuso, pues solo ocasionalmente serán necesarios más de siete valores de prominencia para codificar un sistema experto. La mayor parte de los sistemas expertos bien codificados no necesitan más de tres o cuatro valores de prominencia. En los sistemas expertos grandes, se recomienda en gran medida que los programadores utilicen módulos para controlar el flujo de ejecución (como en analiza en la siguiente sección) y que no se utilicen más de dos o tres valores de prominencia.

Como un ejemplo de cómo evitar el uso de la prominencia, a continuación se muestra un conjunto simple de reglas que sugiere los cuadros que hay que marcar en un juego de gato, las reglas se enumeran en el orden en que deben aplicarse.

```
SI está abierto un cuadro ganador, ENTONCES tomarlo.
SI está abierto un cuadro bloqueador, ENTONCES tomarlo.
SI está abierto un cuadro, ENTONCES tomarlo.
```

Si el hecho *elegir-movimiento* indica que el movimiento que debe realizarse y los hechos *cuadro-abierto* indican si está abierto un cuadro ganador, bloqueador, medio, de la esquina o lateral, las siguientes reglas seleccionarán el movimiento apropiado:

```
(defrule elegir-para-ganar
   (declare (salience 10))
   ?fase <- (elegir-movimiento)
   (cuadro-abierto gana)
   =>
   (retract ?fase)
   (assert (mover-para ganar)))

(defrule elegir-para-bloquear
   (declare (salience 5))
   ?fase <- (elegir-movimiento)
   (cuadro-abierto bloquea)
   =>
   (retract ?fase)
   (assert (mover-para bloquear)))

(defrule elegir-cualquiera
   ?fase <- (elegir-movimiento)
   (cuadro-abierto ?cualquiera&esquina|medio|lateral)
   =>
   (retract ?fase)
   (assert (mover-para ?cualquiera)))
```


Observe que si está disponible más de un tipo de cuadro, las tres reglas se colocarán en la agenda. Cuando se dispara la regla *elegir-para-ganar*, *elegir-para-bloquear* o *elegir-cualquiera*, la retracción del hecho de control eliminará de la agenda a las demás. Por tanto, las tres reglas se interrelacionan de manera muy estrecha, no es posible comprender el propósito de cada una de ellas sin ver todas las demás reglas en conjunto. Esto viola un concepto básico de la programación basada en reglas, porque en la medida de lo posible, una regla debe representar una heurística expresada por completo y, en este caso, la prominencia se utiliza para expresar una relación implícita entre estas reglas, que podría establecerse de manera explícita con patrones adicionales en las reglas. Las reglas pueden reescribirse sin prominencia como sigue:

```
(defrule elegir-para-ganar
   ?fase <- (elegir-movimiento)
   (cuadro-abierto gana)
   =>
   (retract ?fase)
   (assert (mover-para ganar)))

(defrule elegir-para-bloquear
   ?fase <- (elegir-movimiento)
   (cuadro-abierto bloquea)
   (not (cuadro-abierto gana))
   =>
   (retract ?fase)
   (assert (mover-para bloquear)))

(defrule elegir-cualquiera
   ?fase <- (elegir-movimiento)
   (cuadro-abierto ?cualquiera&esquina|medio|lateral)
   (not (cuadro-abierto gana))
   (not (cuadro-abierto bloquea))
   =>
   (retract ?fase)
   (assert (mover-para ?cualquiera)))
```

El hecho de agregar los patrones adicionales a las reglas establece de manera explícita las condiciones bajo las que son aplicables, eliminando la fuerte interacción entre ellas se permite que actúen de manera oportuna. La reescritura de las reglas también demuestra que la heurística original podría haberse expresado con mayor claridad.

```
SI está abierto un cuadro ganador, ENTONCES tomarlo.
SI está abierto un cuadro bloqueador, y
   no está abierto un cuadro ganador, ENTONCES tomarlo.
SI está abierto un cuadro de esquina, medio o lateral, y
   No está abierto un cuadro ganador, y
   No está abierto un cuadro bloqueador, ENTONCES tomarlo.
```

10.6 ELEMENTO CONDICIONAL DEFMODULE

Hasta ahora, todas las defrules, deftemplates y deffacts han estado contenidas en un solo espacio de trabajo. CLIPS utiliza la construcción **defmodule** para dividir una base de conocimiento mediante la definición de los diferentes módulos, la sintaxis básica de esta construcción es

```
(defmodule <nombre-módulo> [<comentario>])
```

CLIPS define un módulo llamado MAIN como opción predeterminada. En los ejemplos anteriores hemos visto que el nombre de módulo MAIN aparece con la representación impresa de las construcciones. Por ejemplo:

```
CLIPS> (clear)↵
CLIPS> (deftemplate sensor (nombre de ranura))↵
CLIPS> (ppdeftemplate sensor)↵
(deftemplate MAIN: :sensor
   (nombre de ranura))
CLIPS>
```

Al símbolo :: del nombre MAIN::sensor se le llama ***separador de módulo***, a su derecha se encuentra el nombre de la construcción, a su izquierda está el nombre del módulo dentro del que se encuentra la construcción. Puesto que todas las construcciones que se han definido hasta ahora se han colocado como opción predeterminada en el módulo MAIN, este nombre de módulo aparece con la representación de impresión de las construcciones.

Ahora que se conoce la sintaxis de defmodule, es posible definir nuevos módulos; utilizando el ejemplo del sistema para detectar fallas, se definirán módulos que correspondan a las fases DETECCIÓN, AISLAMIENTO y RECUPERACIÓN.

```
CLIPS> (defmodule DETECCIÓN)↵
CLIPS> (defmodule AISLAMIENTO)↵
CLIPS> (defmodule RECUPERACIÓN)↵
```

Una vez que se han definido más módulos, surge la pregunta "¿en qué módulo se colocan las nuevas construcciones?". Como opción predeterminada, CLIPS coloca las construcciones recién definidas en el módulo actual, que cuando CLIPS se inicia por primera vez o se limpia, es automáticamente el módulo MAIN. Por tanto, como en los ejemplos anteriores sólo había un módulo y éste era el módulo actual, todas las construcciones definidas se colocaron en el módulo MAIN.

Cada vez que se define un nuevo módulo, CLIPS lo hace módulo actual. Puesto que en el diálogo anterior se definió al último el módulo RECUPERACIÓN, éste es el módulo actual y las reglas recién definidas se colocarán en él.

```
CLIPS> (defrule ejemplo1 =>)↵
CLIPS> (ppdefrule ejemplo1)↵
(defrule RECUPERACIÓN::ejemplo1
   =>)
CLIPS>
```


El módulo en el que se coloca una construcción puede especificarse en el nombre de la construcción. En el nombre, primero se especifica el módulo, seguido por el separador de módulo, y luego el nombre de la construcción.

```
CLIPS> (defrule AISLAMIENTO::ejemplo2 =>)↵
CLIPS> (ppdefrule ejemplo2)↵
(defrule AISLAMIENTO::ejemplo2
   =>)
CLIPS>
```

El módulo actual se cambia cuando se especifica el nombre de módulo en un nombre de construcción. El módulo actual puede determinarse con la función **get-current-module**; esta función no requiere argumentos y devuelve el nombre del módulo actual. La función **set-current-module** se utiliza para cambiar el módulo actual. Requiere un argumento sencillo, el nombre del nuevo módulo actual, y devuelve el nombre del módulo actual previo.

```
CLIPS> (get-current-module)↵
AISLAMIENTO
CLIPS> (set-current-module DETECCIÓN)↵
AISLAMIENTO
CLIPS>
```

Especificación de módulos en los comandos

Como opción predeterminada, la mayor parte de los comandos de CLIPS que operan en construcciones sólo funcionan en las que se encuentren dentro del módulo actual. Por ejemplo, el comando *list-defrules* no producirá una salida si el módulo actual es el módulo DETECCIÓN porque ese módulo no contiene reglas:

```
CLIPS> (list-defrules)↵
CLIPS>
```

Si se desea ver las defrules contenidas en el módulo AISLAMIENTO, habría que establecerlo como módulo actual y luego ejecutar otro comando *list-defrules*:

```
CLIPS> (set-current-module AISLAMIENTO)↵
DETECCIÓN
CLIPS> (list-defrules)↵
ejemplo2
For a total of 1 defrule.
CLIPS>
```

Como alternativa, el comando *list-defrules* acepta un nombre de módulo como un argumento opcional, que especifica cuál módulo tendrá enumeradas sus reglas:

```
CLIPS> (list-defrules RECUPERACIÓN)↵
ejemplo1
For a total of 1 defrule.
CLIPS>
```

Si usa el símbolo * como un argumento para *list-defrules*, entonces se enumeran las reglas de todos los módulos. Cada lista va precedida por el nombre del módulo seguido por la lista de las reglas contenidas en ese módulo.

```
CLIPS> (list-defrules)↵
MAIN:
DETECCIÓN:
AISLAMIENTO:
   ejemplo2
RECUPERACIÓN
   ejemplo1
For a total of 2 defrules.
CLIPS>
```

Las funciones *list-deftemplates* y *list-deffacts* funcionan de manera similar y el comando *show-breaks* también permite especificar el módulo que desplegará sus puntos de división. La sintaxis modificada para estas funciones es:

```
(list-defrules [<nombre-módulo>])
(list-deftemplates [<nombre-módulo>])
(list-deffacts [<nombre-módulo>])
(show-breaks [<nombre-módulo>])
```

Las reglas que operan en construcciones específicas también permiten que se especifique un módulo. Por ejemplo, si no especificó un nombre de módulo *ppdefrule* sólo buscará el módulo actual:

```
CLIPS> (ppdefrule ejemplo2)↵
(defrule AISLAMIENTO::ejemplo2
   =>)
CLIPS> (ppdefrule ejemplo1)↵
[PRNTUTIL1] Incapaz de encontrar defrule ejemplo1.
CLIPS>
```

La regla *ejemplo2* tiene desplegado su forma de impresión, ya que está dentro del módulo AISLAMIENTO, pero como la regla *ejemplo1* no lo está, el comando *ppdefrule* no la encuentra.

El módulo en el que se busca una construcción puede especificarse colocando el nombre de módulo, seguido por el separador de módulo, al frente del nombre de construccion. Por ejemplo:

```
CLIPS> (ppdefrule RECUPERACIÓN::ejemplo1)↵
(defrule RECUPERACIÓN::ejemplo1
   =>)
CLIPS>
```

Es posible tener dos construcciones con el mismo nombre en diferentes módulos. El uso del especificador de módulo antes del nombre de la construcción le permite distinguir entre ambas en un comando:


```
CLIPS> (defrule DETECCIÓN::ejemplo1 =>)↵
CLIPS> (list-defrules *)↵
MAIN:
DETECCIÓN:
   ejemplo1
AISLAMIENTO:
   ejemplo2
RECUPERACIÓN
   ejemplo1
For a total of 3 defrules.
CLIPS> (ppdefrule RECUPERACIÓN::ejemplo1)↵
(defrule RECUPERACIÓN::ejemplo1
   =>)
CLIPS> (ppdefrule DETECCIÓN::ejemplo1)↵
(defrule DETECCIÓN::ejemplo1
   =>)
CLIPS>
```

Los siguientes comandos permiten especificar el módulo como parte del nombre de la construcción: *ppdefrule, undefrule, ppdeftemplate, undeftemaplate, ppdeffacts, undeffacts, matches, refresh, remove-break* y *set-break*.

10.7 IMPORTACIÓN Y EXPORTACIÓN DE HECHOS

Ya se ha visto como dividir construcciones colocándolas en módulos separados y que los propios hechos también pueden dividirse. La afirmación de hechos se asocia automáticamente con el módulo en el que están definidas sus deftemplates correspondientes, por ejemplo:

```
CLIPS>
(deftemplate DETECCIÓN::falla
   (componente de ranura))↵
CLIPS> (assert (falla (componente A)))↵
<Fact-0>
CLIPS> (facts)↵
f-0     (falla (componente A))
For a total of 1 fact.
CLIPS>
(deftemplate AISLAMIENTO::falla-posible
   (componente de ranura))↵
CLIPS> (assert (falla-posible (componente B)))↵
<Fact-1>
CLIPS> (facts)↵
f-1     (falla-posible (componente B))
For a total of 1 fact.
CLIPS> (set-current-module DETECCIÓN)↵
AISLAMIENTO
CLIPS> (facts)↵
f-0     (falla (componente A))
```

```
For a total of 1 fact.
CLIPS>
```

Observe que el único hecho enumerado en el módulo AISLAMIENTO es *falla-posible*, cuya deftemplate correspondiente se encuentra contenida en el módulo AISLAMIENTO; lo mismo es cierto para el hecho *falla* en el módulo DETECCIÓN.

El comando *facts*, al igual que el comando *list-defrules* y otros similares, puede aceptar un nombre de módulo como un argumento opcional, su sintaxis es:

```
(facts [<nombre-módulo>]
       [<inicio> [<final> [<máximo>]]])
```

Al igual que el comando *list-defrules*, la especificación de un nombre de módulo sólo enumera los hechos contenidos en el módulo especificado. Si se utilizó * para el nombre de módulo, entonces se enumeran todos los hechos:

```
CLIPS> (facts DETECCIÓN)↵
f-0     (falla (componente A))
For a total of 1 fact.
CLIPS> (facts AISLAMIENTO)↵
f-1     (falla-posible (componente B))
For a total of 1 fact.
CLIPS> (facts RECUPERACIÓN)↵
CLIPS> (facts *)↵
f-0     (falla (componente A))
f-1     (falla-posible (componente B))
For a total of 2 fact.
CLIPS>
```

A diferencia de las construcciones de defrule y deffacts, las construcciones de deftemplate (y todos los hechos que utilicen esa deftemplate) pueden compartirse con otros módulos. Un hecho es "propiedad" del módulo en que está contenida su deftemplate, pero el módulo propietario puede **exportar** la deftemplate asociada con el hecho, lo que hace que todos los hechos que la utilizan sean visibles para los otros módulos. No es suficiente tan sólo exportar la deftemplate para hacer que un hecho sea visible para otro módulo, para emplear una deftemplate definida en otro módulo, un módulo también debe **importar** la definición de deftemplate.

Los módulos que exportan deftemplates deben usar el atributo export en su definición de defmodule. El atributo export debe utilizar uno de los siguientes formatos:

```
(export ?TODO)
(export ?NADA)
(export deftemplate ?TODO)
(export deftemplate ?NADA )
(export deftemplate <nombre-deftemplate>+)
```

El primer formato exportará todas las construcciones exportables de un módulo. De las construcciones que se analizan en este libro, sólo son exportables las deftemplates; también pueden exportarse algunas de las otras construcciones de programación de procedimientos y orientada a objetos de CLIPS. El segundo formato indica que no se exporta ninguna construcción y es la opción predeterminada para un defmodule. El tercer formato indica que se exportan todas las deftemplates de un módulo (para las construcciones que se analizan en este libro, este formato es igual que el primero). De manera similar, el cuarto formato indica que no se exporta ninguna construcción de deftemplate. El segundo y cuarto formatos se proporcionan principalmente para que las construcciones exportadas por un módulo puedan establecerse en forma explícita. Por último, el quinto formato proporciona una lista específica de deftemplates exportadas por un módulo. El atributo export puede utilizarse más de una vez en una definición de defmodule para especificar diferentes tipos de construcciones exportadas, pero ya que las deftemplates son las únicas construcciones exportables que hemos analizado, no habrá necesidad de emplear más de una instrucción del atributo export.

El formato import también tiene cinco formatos posibles:

```
(import <nombre-módulo> ?TODO)
(import <nombre-módulo> ?NADA)
(import <nombre-módulo> deftemplate ?TODO)
(import <nombre-módulo> deftemplate ?NADA )
(import <nombre-módulo> deftemplate
                        <nombre-deftemplate>+)
```

Cada uno de ellos tiene el mismo significado que su contraparte de export, sólo que las construcciones especificadas se importan, y también debe especificarse el módulo del que provienen. Al igual que el atributo export, una definición de defmodule puede tener más de un atributo import.

Una construcción debe estar definida antes de que pueda especificarse en una lista de importación, pero no tiene que estar definida antes de que pueda especificarse en una lista de exportación (para colocar una construcción en un módulo, éste debe estar definido, por lo que en realidad es imposible definir una construcción antes de definir el módulo que exporta). Debido a esta limitante, es imposible que dos módulos importen entre sí (es decir, si el módulo A importa del módulo B, no es posible que éste se importe del A).

Para ilustrar la importación y exportación de hechos, supongamos que el módulo RECUPERACIÓN desea importar la deftemplate *falla* del modulo DETECCIÓN y la deftemplate *posible-falla* del módulo AISLAMIENTO. A diferencia de otras construcciones, una vez definido un defmodule no puede volverse a definir, por tanto, para cambiar los atributos import y export de un defmodule, primero debe emitirse un comando clear, con excepción del módulo MAIN, que es predefinido y puede redefinirse una vez para incluir diferentes atributos import y export (el módulo MAIN, como opción predeterminada, no exporta ni importa nada). *Observe* que la definición predeterminada del módulo MAIN *no* exporta la deftemplate *hecho-inicial*. Recuerde que en el capítulo 9 se menciona que el patrón (hecho-inicial) se agrega al lado izquierdo de una regla bajo ciertas circunstancias (como cuando el primer elemento condicional es un *not*), si esta regla se coloca en un módulo que no importa la deftemplate *hecho-inicial* del módulo MAIN, no puede activarse. También observe que no se obtendrá un error al definir la regla porque el patrón (hecho-inicial) provocará la creación de una deftemplate *hecho-inicial* implícita en el módulo actual.

Las nuevas definiciones para los módulos DETECCIÓN, AISLAMIENTO y RECUPERACIÓN junto con sus deftemplates (que deben introducirse después de un comando *clear*) son las siguientes:

```
(defmodule DETECCIÓN
   (export deftemplate falla))
```

```
(deftemplate DETECCIÓN::falla
   (componente de ranura))

(defmodule AISLAMIENTO
   (export deftemplate falla-posible))

(defmodule AISLAMIENTO::falla-posible
   (componente de ranura))

(defmodule RECUPERACIÓN
   (import DETECCIÓN deftemplate falla)
   (import AISLAMIENTO deftemplate falla-posible))
```

Con los defmodules y deftemplates definidos, ahora es posible afirmar hechos *falla* en los módulos DETECCIÓN y AISLAMIENTO y hechos *falla-posible* en los módulos AISLAMIENTO y RECUPERACIÓN.

```
CLIPS>
(deffacts DETECCIÓN::inicio
   (falla (componente A)))↵
CLIPS>
(deftemplate AISLAMIENTO::inicio
   (falla-posible (componente B)))↵
CLIPS>
(deftemplate RECUPERACIÓN::inicio
   (falla (componente C))↵
   (falla-posible (componente D)))↵
CLIPS> (reset)↵
CLIPS> (facts DETECCIÓN)↵
f-0     (falla (componente A))
f-2     (falla (componente C))
For a total of 2 facts.
CLIPS> (facts AISLAMIENTO) ↵
f-1     (falla-posible (componente B))
f-3     (falla-posible (componente D))
For a total of 2 facts.
CLIPS> (facts RECUPERACIÓN) ↵
f-0     (falla (componente A))
f-1     (falla-posible (componente B))
f-2     (falla (componente C))
f-3     (falla-posible (componente D))
For a total of 4 facts.
CLIPS>
```

Observe que los módulos DETECCIÓN y RECUPERACIÓN ven los hechos *falla* afirmados por cada uno de los módulos, lo mismo es cierto para los hechos *falla-posible* afirmados por los módulos AISLAMIENTO y RECUPERACIÓN.

10.8 MÓDULOS Y CONTROL DE EJECUCIÓN

Además de controlar cuáles deftemplates pueden importar y exportar un módulo, la construcción defmodule puede utilizarse para controlar la ejecución de reglas. En lugar de sólo ser parte de una agenda general, cada módulo definido en CLIPS tiene su propia agenda, así la ejecución puede controlarse entonces al decidir cuál agenda de módulo se selecciona para ejecutar las reglas. Por ejemplo, las siguientes defrules deben activarse por los hechos falla y falla-posible afirmados en el ejemplo anterior:

```
(defrule DETECCIÓN::regla-1
      (falla (componente A | C))
      =>)

   (defrule AISLAMIENTO::regla-2
      (falla-posible (componente B | D))
      =>)

   (defrule RECUPERACIÓN::regla-3
      (falla (componente A | C))
      (falla-posible (componente B | D))
      =>)
```

Si se emite el comando *agenda* después de que se cargan estas reglas, se desplegará la agenda del módulo RECUPERACIÓN porque la última regla definida se colocó en ese módulo:

```
CLIPS> (get-current-module) ↵
RECUPERACIÓN
CLIPS> (agenda) ↵
0       regla-3: f-0, f-3
0       regla-3: f-0, f-1
0       regla-3: f-2, f-3
0       regla-3: f-2, f-1
For a total of 4 activations.
CLIPS>
```

Al igual que los comandos *list-defrules* y *facts*, el comando agenda acepta un argumento opcional que, si se especifica, indica el módulo cuya agenda se desplegará:

```
CLIPS> (agenda DETECCIÓN)↵
0       regla-1: f-2
0       regla-1: f-0
For a total of 2 activations.
CLIPS> (agenda AISLAMIENTO)↵
0       regla-2: f-3
0       regla-2: f-1
For a total of 2 activations.
CLIPS> (agenda RECUPERACIÓN)↵
0       regla-3: f-0, f-3
```

```
0       regla-3: f-0,f-1
0       regla-3: f-2,f-3
0       regla-3: f-2,f-1
For a total of 4 activations.
CLIPS> (agenda *)↵
MAIN:
DETECCIÓN:
   0       regla-1: f-2
   0       regla-1: f-0
AISLAMIENTO:
   0       regla-2: f-3
   0       regla-2: f-1
RECUPERACIÓN:
   0       regla-3: f-0,f-3
   0       regla-3: f-0,f-1
   0       regla-3: f-2,f-3
   0       regla-3: f-2,f-1
For a total of 8 activations.
CLIPS>
```

Comando focus

Ahora que hay reglas en tres agendas separadas, ¿qué sucede cuando se emite un comando *run*?

```
CLIPS> (unwatch all)↵
CLIPS> (watch rules)↵
CLIPS> (run)↵
CLIPS>
```

¡No se disparan las reglas! Además del módulo actual, que CLIPS utiliza para determinar dónde se agregan nuevas construcciones y cuáles construcciones utilizan los comandos o se ven afectadas por ellos, CLIPS también mantiene un **enfoque actual**, que determina la agenda que usa el comando *run* durante la ejecución. Los comandos *reset* y *clear* establecen automáticamente el enfoque actual en el módulo MAIN. El enfoque actual *no* cambia cuando se cambia el módulo actual, por tanto, en el ejemplo, cuando se emite el comando *run* se utiliza la agenda asociada con el módulo MAIN para seleccionar las reglas que se habrán de ejecutar y, como esta agenda está vacía, no se disparan reglas.

El comando **focus** se utiliza para cambiar el enfoque actual y su sintaxis es

```
(focus <nombre-módulo>+)
```

En el sencillo caso de que sólo se especifique un nombre de módulo, el enfoque actual se establece en el módulo especificado. Al establecer el enfoque actual en el módulo DETECCIÓN y luego emitir un comando run, se dispararán las reglas de la agenda del módulo DETECCIÓN:

```
CLIPS> (focus DETECCIÓN)↵
TRUE
CLIPS> (run)↵
```



```
FIRE    1 regla-1: f-2
FIRE    2 regla-1: f-0
CLIPS>
```

El uso del comando *focus* no sólo cambia el enfoque actual, también recuerda su valor previo. El enfoque actual es en realidad el valor superior de una estructura de datos apilada llamada ***pila de enfoque***. Cada vez que el comando *focus* cambia el enfoque actual, el nuevo enfoque actual se coloca en la parte superior de la pila de enfoque, desplazando a los previos. Mientras la regla se ejecuta, cuando la agenda del enfoque actual queda vacía, el enfoque actual se extrae (elimina) de la pila de enfoque y el siguiente módulo se convierte en el enfoque actual; entonces se ejecutan las reglas de la agenda del nuevo enfoque actual hasta que se enfoca a un nuevo módulo o hasta que ya no hay reglas en la agenda del enfoque actual. Las reglas se seguirán ejecutando hasta que ya no haya módulos en la pila de enfoque o se emita el comando *halt*.

Continuando con el mismo ejemplo, el enfocar primero el módulo AISLAMIENTO y luego el módulo RECUPERACIÓN provocará que se disparen todas las reglas de la agenda del módulo RECUPERACIÓN, lo que será seguido por el disparo de las reglas de la agenda del módulo AISLAMIENTO. El comando **list-focus-stack** (que no requiere argumentos) se utiliza para desplegar los módulos de la pila de enfoque:

```
CLIPS> (focus AISLAMIENTO)↵
TRUE
CLIPS> (focus RECUPERACIÓN)↵
TRUE
CLIPS> (list-focus-stack)↵
RECUPERACIÓN
AISLAMIENTO
CLIPS> (run)↵
FIRE    1 regla-3: f-1,f-4
FIRE    2 regla-3: f-1,f-2
FIRE    3 regla-3: f-3,f-4
FIRE    4 regla-3: f-3,f-2
FIRE    5 regla-2: f-4
FIRE    6 regla-2: f-2
CLIPS> (list-focus-stack)↵
CLIPS>
```

El uso de dos comandos focus para empujar a los módulos AISLAMIENTO y RECUPERACIÓN provoca la ejecución de las reglas de RECUPERACIÓN antes que las de AISLAMIENTO, pero, cuando se especifica más de un módulo en un solo comando *focus*, los módulos se empujan en la pila de enfoque de derecha a izquierda, ejemplo:

```
CLIPS> (focus AISLAMIENTO RECUPERACIÓN)↵
TRUE
CLIPS> (list-focus-stack)↵
AISLAMIENTO
RECUPERACIÓN
CLIPS> (focus AISLAMIENTO)↵
TRUE
```

```
CLIPS> (list-focus-stack)↵
AISLAMIENTO
RECUPERACIÓN
CLIPS> (focus RECUPERACIÓN)↵
TRUE
CLIPS> (list-focus-stack)↵
RECUPERACIÓN
AISLAMIENTO
RECUPERACIÓN
CLIPS>
```

Observe que el mismo módulo puede estar en la pila de enfoque más de una vez, pero el enfoque en un módulo que ya es el enfoque actual no tiene efecto.

Manejo y examen de la pila de enfoque

CLIPS proporciona varios comandos y funciones para manejar el enfoque actual y la pila de enfoque: el comando **clear-focus-stack** elimina todos los módulos de la pila de enfoque. la función **get-focus** devuelve el nombre de módulo del enfoque actual o el símbolo FALSE si la pila de enfoque está vacía. La función **pop-focus** elimina el enfoque actual de la pila de enfoque (y devuelve el nombre de módulo o el símbolo FALSE si la pila de enfoque está vacía) y la función **get-focus-stack** devuelve un valor de campo múltiple que contiene los módulos de la pila de enfoque.

```
CLIPS> (get-focus-stack)↵
(RECUPERACIÓN AISLAMIENTO RECUPERACIÓN)
CLIPS> (get-focus)↵
RECUPERACIÓN
CLIPS> (pop-focus)↵
RECUPERACIÓN
CLIPS> (clear-focus-stack)↵
CLIPS> (get-focus-stack)↵
()
CLIPS> (get-focus)↵
FALSE
CLIPS> (pop-focus)↵
FALSE
CLIPS>
```

Puede utilizarse el comando *watch* para ver los cambios hechos en la pila de enfoque empleando la palabra clave *focus* como el argumento del comando:

```
CLIPS> (watch focus)↵
CLIPS> (focus DETECCIÓN AISLAMIENTO RECUPERACIÓN)↵
==> Focus RECUPERACIÓN
==> Focus AISLAMIENTO from RECUPERACIÓN
==> Focus DETECCIÓN from AISLAMIENTO
TRUE
CLIPS> (run)↵
<== Focus DETECCIÓN to AISLAMIENTO
==> Focus AISLAMIENTO to RECUPERACIÓN
==> Focus RECUPERACIÓN
CLIPS>
```


En caso de que se proporcione el comando *run* y la pila de enfoque esté vacía, el módulo MAIN se coloca automáticamente en la pila de enfoque. Esta característica se proporciona de manera primordial como un mecanismo útil en caso de que se agreguen nuevas activaciones después de que se ha terminado el programa porque no hay módulos en la pila de enfoque, por ejemplo:

```
CLIPS> (clear)↵
CLIPS> (watch focus)↵
CLIPS> (watch rules)↵
CLIPS> (defrule ejemplo-1 =>)↵
CLIPS> (reset)↵
<== Focus MAIN
==> Focus MAIN
CLIPS> (run)↵
FIRE    1 ejemplo-1: f-0
<== Focus MAIN
CLIPS> (defrule ejemplo-2 =>)↵
CLIPS> (agenda)↵
0      ejemplo-2: f-0
For a total of 1 activation.
CLIPS> (list-focus-stack)↵
CLIPS>
```

La regla *ejemplo-2* está en la agenda, pero no hay módulos en la pila de enfoque, sin embargo, la activación de un comando *run* coloca el módulo MAIN en la pila de enfoque, de modo que la regla *ejemplo-2* se disparará de cualquier manera.

```
CLIPS> (run)↵
<== Focus MAIN
FIRE    1 ejemplo-2: f-0
==> Focus MAIN
CLIPS>
```

Comando return

Una de las desventajas de controlar el flujo de ejecución utilizando hechos de control para representar fases, como se analizó en la sección 10.4, es que no es posible disparar algunas activaciones en una fase en particular, salir de ella, regresar más tarde y ejecutar las activaciones restantes de la agenda. Esto sucede porque una vez que se retrae el hecho de control que representa la fase, todas las activaciones de esa fase se eliminan de la agenda. Cuando el hecho de control se reafirma más adelante, se reactivarán *todas* las activaciones previas para esa fase y no sólo aquellas que no se dispararon (desde luego, esto suponiendo que solo se afirmó o retrajo el hecho de control y no algún otro hecho).

Si se utilizan módulos para controlar el flujo de ejecución, es posible detener de manera prematura la ejecución de las activaciones de una agenda de módulo específica (es decir, antes de que la agenda del módulo esté vacía). El comando **return** puede utilizarse para detener

inmediato la ejecución del lado derecho de una regla y eliminar el enfoque actual de la pila de enfoque (lo que coloca el control de ejecución en el siguiente módulo de la pila de enfoque). Cuando se utiliza en el lado derecho de una regla, el comando *return* debe pasarse sin argumentos (este comando también puede utilizarse en construcciones de programación de procedimientos proporcionadas por CLIPS). El siguiente ejemplo ilustra el uso del comando *return*:

```
CLIPS> (clear)↵
CLIPS>
(defmodule MAIN
   (export deftemplate hecho-inicial))↵
CLIPS>
(defmodule DETECCIÓN
   (import MAIN deftemplate hecho-inicial))↵
CLIPS>
(defrule MAIN::inicio
   =>)
   (focus DETECCIÓN))↵
CLIPS>
(defrule DETECCIÓN::ejemplo-1
   =>)
   (prinout t "¡No imprimir!" crlf))↵
CLIPS>
(defrule DETECCIÓN::ejemplo-2
   =>)
   (return)
   (printout t "¡No imprimir!" crlf))↵
CLIPS> (watch rules)↵
CLIPS> (watch focus)↵
CLIPS> (reset)↵
<== Focus MAIN
==> Focus MAIN
CLIPS> (run)↵
FIRE    1 inicio: f-0
==> Focus DETECCIÓN de MAIN
FIRE    2 ejemplo-1: f-0
<== Focus DETECCIÓN a MAIN
<== Focus MAIN
CLIPS>
```

Hay dos puntos de este ejemplo que vale la pena destacar, en primer lugar, para que se activen las reglas del módulo DETECCIÓN utilizando el patrón predeterminado *hecho-inicial*, la deftemplate *hecho-inicial* debe exportarse mediante el módulo MAIN e importarse mediante el módulo DETECCIÓN. En segundo lugar, observe que el comando return detiene *de inmediato* la ejecución del lado derecho de la regla, el comando *printout* que sigue al comando *return* no se ejecuta en la regla *ejemplo-1* (o en la regla *ejemplo-2*, porque no tiene la oportunidad de ejecutarse). Observe que la funcionalidad del comando *return* es similar, aunque no igual, al comando *pop-focus*, que elimina el enfoque actual de la pila de enfoque, pero permitirá que continúe la ejecución de las accior [illegible] lado derecho de [illegible] si se hubiera [illegible] ado el

comando *pop-focus* en lugar del comando *return* en este ejemplo, la cadena "¡No imprimir!" se habría impreso cuando se ejecutaran las acciones de la regla *ejemplo-1*.

Característica auto-focus

Además de tener la posibilidad de enf[illegible] módulos de manera explícita usando el comando *focus*, también es posible enfocar aut[illegible]amente un módulo cuando se activan reglas específicas de ese módulo. Como opció[illegible]terminada, el módulo de una regla no se enfoca automáticamente cuando esa regla [illegible]. Esto puede cambiarse utilizando el atributo **auto-focus**, que se especifica en la inst[illegible] declare junto con el atributo salience. La palabra clave *auto-focus* se especifica se[illegible]or TRUE (para habilitar la función) o FALSE (para inhabilitarla). No es necesario q[illegible] las reglas de un módulo tengan habilitada la función auto-focus para que algunas de [illegible] utilicen. Del mismo modo, no es necesario declarar tanto el atributo salience como [illegible]uto auto-focus si se emplea la instrucción declare para una regla. En el siguiente ejem[illegible] ilustra el uso de la función auto-focus:

```
CLIPS> (clear)↵
[illegible]PS>
   [illegible]fmodule MAI[illegible]
   (export def[illegible]ate hecho-inicial
   [illegible]
                [illegible]IÓN
                [illegible]eftemplate hecho-inicial))↵
[illegible]
(defrule DETEC[illegible]CIÓN::[illegible]
   (declare (auto-focus TRUE))
   =>) ↵
CLIPS> (focus)↵
CLIPS> [illegible]eset)↵
[illegible]
   [illegible]cus MA[illegible]
   [illegible] Focus DE [illegible]
CLIPS>
```

Cuando se emite el comando *reset*, se enfoca automáticamente el [illegible] MAIN porque la pi[illegible] de enfoque está vacía, la regla *ejemplo* se activa por la afirmac[illegible]cho *hecho-inicial* y puesto que el atributo auto-focus está habilitado para esta regla, [illegible] DETECCIÓN, se [illegible]máticamente en la pila de enfoque. La función auto-focus es útil sobre todo en las [illegible]ctan violaciones a las restricciones, como el módulo de la regla de restricción se [illegible] inmediato en el enfoque actual, es posible emprender acciones cuando ocurre alguna violación en lugar de tener una fase explícita para detectarlos.

Reemplazo d[illegible]ases y hechos de control

A través del uso de defmodules, los comandos focus y return, y la función autofocus, ahora es posible reemplazar el uso de fases y hechos de control con un mecanismo mucho más explícito para controlar el flujo de ejecución de reglas.

Las construcciones descritas en la sección 10.4 para controlar la ejecución pueden reemplazarse con las siguientes construcciones:

```
(defmodule DETECCIÓN)
(defmodule AISLAMIENTO)
(defmodule RECUPERACIÓN)


(deffacts MAIN::información-control
   (fase-secuencia DETECCIÓN AISLAMIENTO RECUPERACIÓN))

(defrule MAIN::cambiar-fase
   ?lista <- (fase-secuencia ?fase-siguiente
                             $?otras-fases)
   =>
   (focus ?fase-siguiente)
   (retract ?lista)
   (assert (fase-secuencia ?otras fases
                           ?fase-siguiente)))
```

El control de la ejecución se realizará en ciclos del módulo DETECCIÓN al módulo AISLAMIENTO al módulo RECUPERACIÓN, y vuelve a empezar. Todas las reglas de cada módulo se dispararían antes de permitir que lo hagan las reglas del siguiente módulo (a menos que se haya emitido un comando return o se haya enfocado un nuevo módulo como resultado de la función auto-focus).

```
CLIPS> (unwatch all)↵
CLIPS> (reset)↵
CLIPS> (watch facts)↵
CLIPS> (watch focus)↵
CLIPS> (run 5)↵
FIRE    1 cambiar-fase: f-1
==> Focus DETECCIÓN de MAIN
<== Focus DETECCIÓN a MAIN
FIRE    2 cambiar-fase: f-2
==> Focus AISLAMIENTO de MAIN
<== Focus AISLAMIENTO a MAIN
FIRE    3 cambiar-fase: f-3
==> Focus RECUPERACIÓN de MAIN
<== Focus RECUPERACIÓN a MAIN
FIRE    4 cambiar-fase: f-4
==> Focus DETECCIÓN de MAIN
<== Focus DETECCIÓN a MAIN
FIRE    5 cambiar-fase: f-5
==> Focus AISLAMIENTO de MAIN
<== Focus AISLAMIENTO a MAIN
CLIPS>
```


10.9 RESUMEN

En este capítulo se introdujeron varias funciones de CLIPS para auxiliar en el desarrollo de sistemas expertos robustos: los atributos de deftemplate permiten la aplicación de las restricciones de tipo y de valor, lo que puede evitar errores tipográficos y semánticos; la verificación de las restricciones puede realizarse de manera estática (cuando se definen expresiones o construcciones) o dinámica (cuando se evalúan expresiones). El atributo type posibilita restringir los tipos válidos permitidos para una ranura. Los atributos de valor permitidos restringen los valores válidos de una ranura a una lista especificada; el atributo range permite restringir valores numéricos a un rango especificado; el atributo cardinality hace posible restringir las cantidades mínima y máxima de campos almacenados en una ranura de campo múltiple. Otros dos atributos de deftemplate, los atributos default y default-dynamic, no restringen valores de ranura pero permiten especificar el valor inicial de una ranura de deftemplate.

La prominencia proporciona un mecanismo para elaborar estructuras de control aún más complejas y se utiliza para priorizar reglas; por ejemplo, que primero se dispare la regla activada con mayor prominencia, la prominencia puede combinarse con hechos de control para separar el conocimiento de experto del conocimiento de control.

La construcción defmodule permite dividir una base de conocimiento, al establecer de manera explícita cuáles deftemplates se importan de otros módulos y se exportan a los mismos (un módulo puede controlar cuáles hechos son visibles para él). Utilizando el comando focus, es posible controlar la ejecución de un programa sin el uso de prominencia al dividir reglas en grupos y colocarlas en módulos separados.

PROBLEMAS

10.1 Modifique el programa Palillos del capítulo 9 de modo que las reglas de control queden separadas de las reglas para jugar el juego. Utilice la prominencia para proporcionar menor prioridad a las reglas de control.

10.2 Agregue una regla al programa del modelo de bloques de la sección 8.6, página 371, que eliminaría un objetivo de movimiento si éste ya se ha satisfecho.

10.3 Elabore reglas para establecer el procedimiento de decisión para determinar si un silogismo es válido. Pruebe su programa con el silogismo del problema 9.9, página 433.

10.4 Escriba un programa para determinar los factores primarios de un número. Por ejemplo, los factores primarios de 15 son 3 y 5.

10.5 Dadas las siguientes distancias entre ciudades de Texas, resuelva el problema del agente viajero (página 43, sección 1.13) y escriba un programa que encuentre la ruta más corta para visitar todas las ciudades. La entrada al programa debe ser la ciudad de partida y la lista de ciudades que habrán de visitarse. Para probar su programa, determine la ruta más corta, empezando en Houston.

	Houston	Dallas	Austin	Abilene	Waco
Houston	—	241	162	351	183
Dallas	241	—	202	186	97
Austin	162	202	—	216	106
Abilene	351	186	216	—	186
Waco	183	97	106	186	—

10.6 Dada la siguiente información, escriba un programa que pregunte el tipo de nubes visibles y la dirección del viento, y luego proporcione un pronóstico de las probabilidades de lluvia.

Las nubes con cúmulos indican buen clima, pero pueden cambiar a nubes con nimbos si el viento sopla de noreste a sur; las nubes con cirrocúmulos indican lluvia en el transcurso de una día si la dirección del viento es de noreste a sur, si el viento sopla de norte al occidente, entonces se predice cielo nublado. Las nubes con estratocúmulos pueden cambiar a nubes con cumulonimbos si la dirección del viento es de noreste a sur; las nubes con estratos indican una llovizna ligera, si el viento sopla de noreste a sur, entonces puede ocurrir una lluvia prolongada; las nubes con nimboestratos indican una lluvia breve si la dirección del viento es sur a norte, se indica una lluvia larga si el viento sopla de noreste a sur; las nubes con cumulonimbos indican chubascos si son visibles antes de la noche; las nubes con cirroestratos indican lluvia en el transcurso de 15 a 24 horas si la dirección del viento es norte a sur; las nubes con altoestratos indican lluvia en el transcurso del día si el viento es de noreste a sur, de lo contrario se nublará el cielo. Las nubes con altocúmulos indican lluvia en el transcurso de 15 a 20 horas si el viento es de noreste a sur.

10.7 Escriba un programa para convertir un mensaje de código Morse a su serie equivalente de caracteres alfabéticos. Primero observe los ejemplos de entrada y salida al programa (donde los símbolos * y – son puntos y guiones y el carácter / se utiliza para delimitar los caracteres de código Morse).

```
Introduzca un mensaje (<CR> a final): * * * / - - - / * * *↵
El mensaje es SOS
Introduzca un mensaje (<CR> a final): ↵
CLIPS>
```

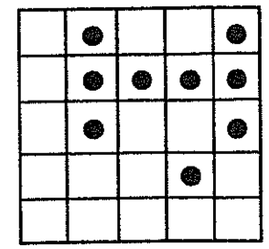
A	*—	H	****	O	— — —	V	***—
B	—***	I	**	P	*— — *	W	*— —
C	—*—*	J	*— — —	Q	— —*—	X	—**—
D	—**	K	—*—	R	*—*	Y	—*— —
E	*	L	*—**	S	***	Z	— —**
F	**—*	M	— —	T	—		
G	— —*	N	—*	U	**—		

Los códigos y sus equivalentes de carácter son los siguientes:

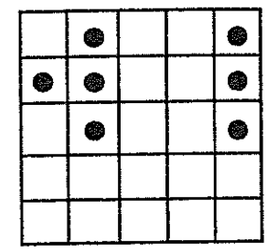
10.8 Escriba un programa que cuando se proporcione una expresión compuesta de números y unidades, convertirá las unidades de la expresión en un conjunto de unidades de base (como metros, segundos, kilogramos, libras y amperios). Los siguientes son ejemplos de entrada y salida al programa.

```
Introduzca una expresión (<CR> a final): 30 metros / minuto↵
La conversión es 0.5 m / s
Introduzca una expresión (<CR> a final): ↵
CLIPS>
```

10.9 Escriba un programa para jugar Life (un programa común que simula autómatas celulares). Dada una serie bidimensional de células en la que cada célula está muerta o viva, los valores para la siguiente generación se basan en las siguientes reglas: cualquier célula viva que es adyacente a exactamente dos o tres células vivas diferentes continúa viva, cualquier célula viva que se encuentra a cualquiera menor de dos o cualquiera mayor que tres células vivas diferentes morirá. Cualquier célula muerta que es adyacente a exactamente tres células vivas diferentes llegará a vivir. Por ejemplo, si la primera generación era una serie de cinco por cinco con las células vivas representadas con puntos como se muestra



entonces la siguiente generación aparecería como sigue:



Utilizando la configuración inicial que se mostró antes, calcule las siguientes cuatro generaciones.

10.10 Un cuadrilátero es una figura de cuatro lados. Un cuadrilátero es un romboide si tiene dos pares distintos de lados consecutivos de la misma longitud. Un cuadrilátero es un trapezoide si tiene cuando menos un par de lados paralelos, es un rombo si sus cuatro lados son de igual longitud, es un rectángulo si tiene cuatro ángulos rectos, es un cuadrado si tiene cuatro lados iguales y cuatro ángulos rectos. Observe que un rombo es un romboide y un paralelogramo, un paralelogramo es un trapezoide, un rectángulo es un paralelogramo y un cuadrado es un rombo y un rectángulo.

Escriba un programa que, cuando se proporcionan los cuatros puntos que forman un cuadrilátero, determine el tipo de cuadrilátero. El programa debe dar razón de un posible error de redondeo (se asume que dos lados son iguales si la diferencia entre sus longitudes es menor que .00001). Pruebe su programa con los siguientes cuadriláteros:

(a) Puntos (0,0), (2,4), (6,0) y (3,2).
(b) Puntos (0,3), (2,5), (4,3) y (2,0).
(c) Puntos (0,0), (3,2), (4,2) y (9,0).
(d) Puntos (0,0), (1,3), (5,3) y (4,0).
(e) Puntos (0,0), (3,5), (3,5.196152), (9,5.196152) y (6,0).
(f) Puntos (0,0), (0,4), (2,4) y (2,0).
(g) Puntos (0,2), (4,6), (6,4) y (2,0).
(h) Puntos (0,2), (2,4), (4,2) y (2,0).

Pista: si el lado 1 contiene los puntos (x1,y1) y (x2,y2) y el lado 2 contiene los puntos (x3,y3) y (x4,y4), entonces el lado 1 es paralelo al lado 2 si (x2 – x1)*(y4 – y3) es igual a (x3 – x3)*(y2 – y1).

10.11 Escriba un programa CLIPS que pueda determinar si una oración simple es correcta desde el punto de vista gramatical. La oración debe utilizar el siguiente formato BNF (Backus Naus Form).

```
<oración> ::= <verbo> <objeto-directo>
                    [<objeto-indirecto>]
<objeto-directo> ::= [<determinador>] <adjetivo>* <sustantivo>
<objeto-indirecto> ::= <preposición> <objeto-directo>
<determinador> ::= un | uno | la
<adjetivo> ::= roja | brillante | pesado
<sustantivo> ::= pelota | llave | pistola | tenazas
<preposición> ::= con | en | dentro
<verbo> ::= obtener | lanzar | disparar
```

Por ejemplo:

```
Introduzca una oración (<CR> a final): disparar la roja brillante
   pistola en las tenazas
OK
Introduzca una oración (<CR> a final): disparo
No entiendo
Introduzca una oración (<CR> a final):
CLIPS>
```

10.12 Modifique el programa desarrollado en el problema 9-18, página 435, de modo que si ninguno de los arbustos cumple con todos los requisitos, se enumeren los arbustos que cumplen con la mayor parte de ellos. El número de requisitos satisfechos debe imprimirse.

10.13 Dados un número de generadores que proporciona energía y un número de dispositivos que la consumen, escriba un programa que una los dispositivos con generadores para reducir el número de generadores usados y la cantidad de energía no utilizada para cada generador usado. Por ejemplo, si hay cuatro generadores que proporcionan 5, 6, 7 y 10 watts de energía y cuatro dispositivos que consumen 4, 5, 6 y 7 watts de energía, la conexión del dispositivo de 5 watts al generador de 5 watts, el dispositivo de 7 watts al generador de 7 watts, y los dispositivos de 4 y 6 watts al generador de 10 watts reduciría el número de generadores usados y la energía no utilizada. La entrada y salida del programa puede ser una serie de hechos. Pruebe su programa con el ejemplo anterior y, también, para el caso en el que se utilizan los mismos generadores anteriores pero con dispositivos que consumen 1, 3, 4, 5 y 9 watts de energía.

10.14 Escriba un programa que funcione como el sistema operativo de una computadora y determine ubicaciones adecuadas en la memoria para cargar aplicaciones con cantidades fijas de memoria asignadas en ellas. La entrada al programa será una serie de hechos como los siguientes:

```
(lanzar (aplicación procesador-texto) (memoria-necesaria 2))
(lanzar (aplicación hoja-cálculo) (memoria-necesaria 6))
(lanzar (aplicación juego) (memoria-necesaria 1))
(terminar (aplicación procesador-texto))
(lanzar (aplicación juego) (memoria-necesaria 1))
(terminar (aplicación hoja-cálculo))
(terminar (aplicación juego))
```


Suponga que la computadora tiene ocho megabytes de memoria y que el número de megabytes que requiere una aplicación es entero. Si existe un ajuste exacto de memoria para una aplicación, ésta debe utilizarse, por ejemplo, si había dos bloques de memoria libre, uno de seis y otro de cuatro megabytes, y se lanzó una aplicación que requería cuatro megabytes, entonces la aplicación debe colocarse en el bloque libre de cuatro megabytes. Cuando se procesa un comando de lanzamiento o de terminación, debe imprimirse un mensaje. Si no hay un bloque de memoria lo suficientemente grande para lanzar un programa, entonces debe imprimirse un mensaje para este efecto. Pruebe su programa afirmando los hechos precedentes y emitiendo un comando *run* después de afirmar cada hecho. La salida debe ser similar a la siguiente:

```
La ubicación de memoria de la aplicación procesador de texto es 1
   a 2.
La ubicación de memoria de la aplicación procesador de texto es 3
   a 8.
Incapaz de lanzar aplicación juego.
Terminar procesador de texto.
La ubicación de memoria de la aplicación juego es 1 a 1.
Terminar hoja de cálculo.
Terminar juego.
```

10.15 Desarrolle una interfaz de menú de texto que esté autocontenida dentro de un módulo que sea adecuada para volverla a utilizar en otros programas. Los elementos de menú deben representarse como hechos. Deben soportarse dos tipos de acciones para elementos de menú, uno es salir de la ejecución del programa. Si se selecciona este tipo de elemento, el programa debe detener la ejecución, el otro es afirmar un hecho y enfocarse en un módulo específico. Para esta acción los hechos de elemento de menú deben contener ranuras que especifiquen el módulo de enfoque y el valor de hecho que habrán de afirmarse cuando se seleccione el elemento. Por ejemplo, en el siguiente diálogo, el resultado de seleccionar "Opción A" sería enfocarse en el módulo A y afirmar el hecho (seleccionar-menú (valor opción-a)) donde A y *opción-a* son los valores especificados en el hecho elemento-menú. El módulo A contiene una regla que coincide con el hecho *seleccionar-menú* que provoca el mensaje "Ejecutar Opción A" que habrá de imprimirse.

```
CLIPS> (run)
Seleccione una de las siguientes opciones:

1 -- Opción A
2 -- Opción B
9 -- Salir del programa

Su elección: 1

Ejecutar opción A

Seleccione una de las siguientes opciones:

1 -- Opción A
2 -- Opción B
9 -- Salir del programa

Su elección: 9
CLIPS>
```

10.16 Modifique el programa desarrollado en el problema 9-19, página 435 de modo que los módulos separados se utilicen para imprimir la información de todas las estrellas que tienen la clase espectral especificada, todas las estrellas que tienen la magnitud especificada y todas las estrellas que igualan la clase espectral y la magnitud junto con su distancia de la Tierra en años-luz.

10.17 Modifique el programa desarrollado en el problema 9-21, página 435 para utilizar módulos. Si no pueden encontrarse joyas que coinciden con el color, la dureza y la densidad especificados, el programa debe indicarlo. Después de especificar una joya, el programa debe proporcionar al usuario la oportunidad de identificar otra.

10.18 La tabla de abajo enumera los cursos disponibles en una preparatoria, el nombre del instruc-tor y los periodos de clase en los que está disponible el curso. Dando como entrada un hecho que indica una clase que se programará y las preferencias que indican los instructores y periodos preferidos y no preferidos, escriba un programa que sugiera un instructor y periodo apropiados para tomar la clase. Para determinar el "mejor" instructor y periodo, clasifique cada candidato posible como sigue: la clasificación de inicio es cero; un instructor o periodo preferido por persona suma un punto a la calificación; un instructor o periodo no preferido por per-sona resta un punto de la calificación; la falta de preferencia para un instructor o periodo deja intacta la calificación; el instructor y periodo con la mayor calificación son los "mejores".

Curso	Instructor	Periodos ofrecidos
Álgebra	Jiménez	1, 2, 3
Álgebra	Suárez	3, 4, 5, 6
Historia de América	Vale	5
Historia de América	Hernández	1, 2
Arte	Xavier	1, 3, 5
Biología	Duarte	1, 2, 5
Química	Duarte	3, 6
Química	Vargas	6
Francés	Baca	2, 4
Geología	Vargas	1
Geometría	Jiménez	5, 6
Geometría	Suárez	1
Alemán	Baca	5
Literatura	Haro	2, 3, 4, 5, 6
Literatura	Delgado	1, 2, 3, 4, 5
Música	Xavier	2, 4
Educación física	Martínez	1, 2, 3, 4, 5
Educación física	Reyes	1, 2, 3, 4, 6
Educación física	Saldívar	2, 3, 4, 5, 6
Física	Vargas	2, 3, 5
Español	Baca	1, 3
Historia Regional	Vale	2, 3, 4
Historia Regional	Hernández	5, 6
Historia Universal	Vale	2, 3, 4
Historia Universal	Hernández	4

CAPÍTULO 11

Eficiencia de los lenguajes basados en reglas

11.1 INTRODUCCIÓN

En este capítulo se proporcionan varias técnicas para incrementar la eficiencia de un sistema experto basado en reglas que usa el **algoritmo de comparación de patrones de Rete ok**. Primero se exponen las razones por las que es necesario un algoritmo eficaz para la comparación de patrones y luego se explica el algoritmo de Rete. También se exponen varias técnicas para escribir reglas con mayor eficiencia.

11.2 EL ALGORITMO DE COMPARACIÓN DE PATRONES DE RETE

Los lenguajes basados en reglas como CLIPS, ART, OPS5 y OPS83 usan un algoritmo muy eficaz para comparar los hechos con los patrones de reglas y determinar cuáles de ellas han satisfecho sus condiciones. A este algoritmo se le llama ***algoritmo de comparación de patrones de Rete*** (Forgy 79; Forgy 85; Brownston 85). Para escribir reglas eficientes en CLIPS no es necesario comprender el algoritmo de Rete, pero la comprensión del algoritmo que se usa en CLIPS y otros lenguajes basados en reglas facilita que se entienda por qué es más eficiente escribir reglas de una manera específica.

Para comprender la razón por la que es eficiente el algoritmo de Rete, resulta útil observar el problema de la comparación de hechos para reglas en general y luego examinar otros algoritmos que no son tan eficientes. En la figura 11.1 se muestra el problema atendido mediante el algoritmo de Rete.

Si sólo tiene que ocurrir una vez el proceso de comparación, entonces la solución del problema es sencilla; el mecanismo de inferencia puede examinar cada regla y luego buscar el grupo de hechos para determinar si se han satisfecho los patrones de la regla. En este caso, entonces es posible colocar la regla en la agenda, como se muestra en la figura 11.2.

Sin embargo, en los lenguajes basados en reglas, el proceso de comparación tiene lugar varias veces; en condiciones normales la lista de hechos se modificará durante cada ciclo de ejecución, y pueden agregársele los nuevos hechos o eliminarse los hechos antiguos. Estos cambios pueden provocar que algunos patrones no satisfechos previamente se satisfagan, o viceversa y el problema de la comparación ahora se vuelve un proceso continuo. Durante

cada ciclo, a medida que se agregan y se eliminan hechos, el conjunto de reglas debe mantenerse y actualizarse.

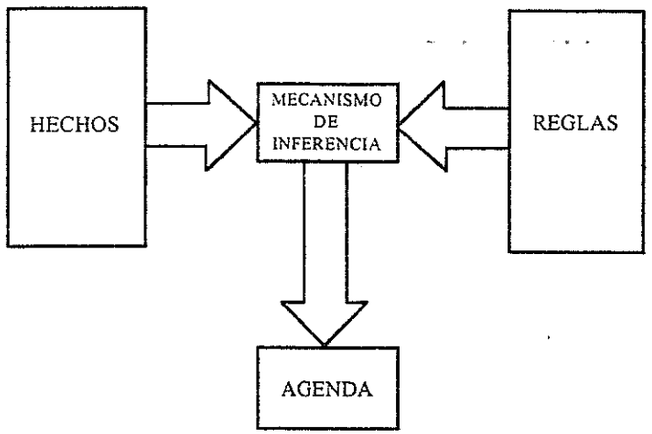


Figura 11.1 Comparación de patrones: reglas y hechos

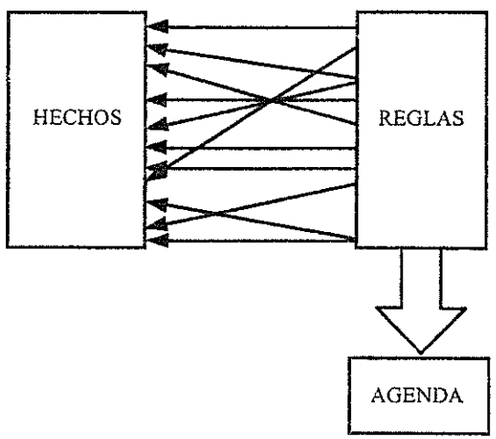


Figura 11.2 Búsqueda de reglas para hechos

Si se hace que el mecanismo de inferencia verifique cada regla para dirigir la búsqueda de hechos después de cada ciclo de ejecución, se tendrá una técnica simple y directa para resolver este problema, pero este método puede ser muy lento. La mayor parte de los sistemas expertos basados en reglas exhiben una propiedad llamada ***redundancia temporal.*** Por lo general, las acciones de una regla sólo cambiarán unos cuantos hechos de la lista es decir, los hechos del sistema experto cambian lentamente con el tiempo. Cada ciclo de ejecución sólo

puede verificar un pequeño porcentaje de los agregados o cambiados y, por tanto, sólo un pequeño porcentaje de reglas suele verse afectado por los cambios de la lista de hechos. De esta manera, hacer que las reglas dirijan la búsqueda de los hechos necesarios requiere muchos cálculos innecesarios, porque es probable que la mayor parte de las reglas encuentren los mismos hechos del ciclo actual que ya se encontraron en el último ciclo. La ineficiencia de este método se muestra en la figura 11.3, en la que el área sombreada representa los cambios que se hicieron a la lista de hechos. Pero es posible evitar el recálculo innecesario si se recuerda lo que ya se ha comparado de un ciclo a otro y luego calcular sólo los cambios necesarios para los hechos recién agregados o eliminados, como se muestra en la figura 11.4. Las reglas permanecen estáticas y los hechos cambian, de modo que los hechos deben encontrar las reglas y no al revés.

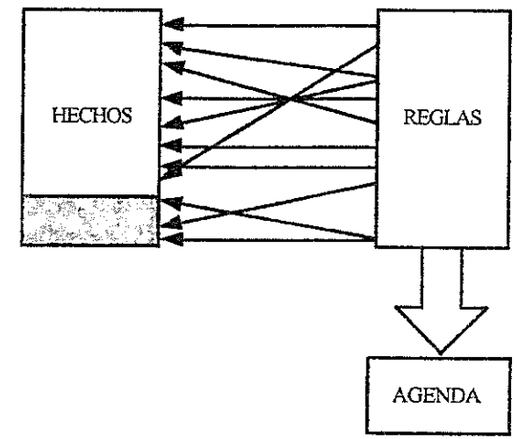


Figura 11.3 Cálculos innecesarios cuando las reglas buscan hechos

El algoritmo de comparación de patrones de Rete está diseñado para aprovechar la redundancia temporal exhibida por sistemas expertos basados en reglas, lo que hace al guardar el estado del proceso de comparación de un ciclo a otro y luego calculando de nuevo los cambios en este estado sólo para los cambios que suceden en la lista de hechos. Es decir, si un conjunto de patrones encuentra dos de los tres hechos necesarios en un ciclo, no se requiere de una verificación durante el siguiente ciclo para los dos hechos que ya se encontraron, sólo el tercer hecho es de interés. El estado del proceso de comparación sólo se actualiza a medida que se agregan y se eliminan hechos; si el número de hechos agregados y eliminados es pequeño en comparación con el número total de hechos y patrones, el proceso de comparación avanzará con rapidez. En el peor de los casos, si se cambiaran todos los hechos, entonces el proceso de comparación funcionaría como si se compararan todos los hechos con todos los patrones.

Si sólo se procesan las actualizaciones a la lista de hechos, cada regla debe recordar lo que ya se ha comparado; es decir, si un nuevo hecho ha coincidido con el tercer patrón de una regla, la información sobre las comparaciones con los dos primeros patrones debe estar disponible para finalizar el proceso de comparación. Al tipo de información de estado que indica los hechos que tienen patrones coincidentes en una regla se le llama ***coincidencia parcial.*** Una coincidencia parcial para una regla es cualquier conjunto de hechos que satisface los patrones de una regla, empezando con el primero y terminando con cualquiera, incluyendo el último,

por tanto, una regla con tres patrones tendría coincidencias parciales para el primer patrón, para primero y segundo, y para primero, segundo y tercero y una coincidencia parcial de todos los patrones de una regla también sería una activación. Al otro tipo de información de estado guardada se le llama ***coincidencia de patrón***, que se presenta cuando un hecho ha satisfecho un patrón individual de cualquier regla sin considerar las variables de otros patrones que puedan restringir el proceso de coincidencia.

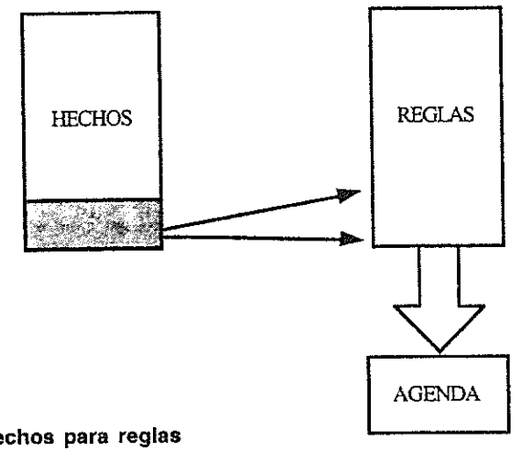


Figura 11.4 Búsqueda de hechos para reglas

La principal desventaja del algoritmo de comparación de patrones de Rete es que utiliza mucha memoria, en tanto que la comparación simple de todos los hechos con todos los patrones no necesita memoria; el solo guardado del estado del sistema usando coincidencias de patrón y coincidencias parciales puede consumir cantidades considerables de memoria. En general, vale la pena este intercambio de memoria por velocidad; sin embargo, es importante recordar que una regla escrita de manera deficiente no sólo puede ejecutarse con lentitud, sino que puede llegar a utilizar cantidades considerables de memoria.

Al aprovechar la **similitud estructural,** de las reglas y el algoritmo de Rete también mejora la eficiencia de los sistemas basados en reglas. La *similitud estructural* se refiere al hecho de que muchas reglas a menudo contienen patrones o grupos de patrones similares. El algoritmo de Rete utiliza esta función para aumentar la eficiencia mediante la unión de componentes comunes, de modo que no tienen que calcularse más de una vez.

11.3 LA RED PATRÓN

El problema de la comparación de hechos con reglas puede dividirse en dos pasos; primero, cuando se agregan y se eliminan hechos debe determinarse cuáles patrones han coincidido y segundo, debe verificarse la comparación de las uniones de variables mediante patrones para determinar las coincidencias parciales de un grupo de patrones.

El proceso para determinar cuáles hechos han coincidido con qué patrones se realiza en la **red patrón;** para abreviar, a continuación se explica solamente la comparación de patrones con ranuras de campo simple de hechos de deftemplates. Todas las coincidencias que no implican comparación contra variables ligadas con otros patrones pueden realizarse en la **red patrón**, éste tiene una estructura arborescente donde la primera restricción de ranura de todos

los patrones son los nodos conectados con la raíz del árbol, la segunda restricción de ranura de todos los patrones son los nodos conectados con esos nodos, y así sucesivamente hasta las últimas restricciones de ranura de un patrón, que son las hojas del árbol. A los nodos de la red patrón se les conoce como ***nodos de una entrada,*** porque cada uno recibe información únicamente del nodo anterior y algunas veces también se les llama ***nodos patrón;*** los nodos hoja también se conocen como ***nodos terminales.*** Cada nodo patrón contiene una especificación usada para determinar si el valor de ranura de un hecho ha coincidido con la restricción de ranura de un patrón; por ejemplo, el patrón:

```
(datos (x 27))
```

se representaría con un nodo porque sólo hay una restricción de ranura. La especificación de comparación para el primer nodo sería entonces la siguiente.

```
El valor x de ranura es igual a la constante 27
```

En realidad, también debe verificarse si el hecho tiene el nombre de relación apropiado para el patrón (por ejemplo, no se desea que los hechos *foobar* coincidan con los patrones *datos* sólo porque tienen los mismos nombres de ranura). Para efectuar esta prueba, CLIPS mantiene redes patrón separadas para cada deftemplate, de modo que la verificación del nombre de relación se realiza a medida que se crea el hecho, pero antes de que ocurra la coincidencia de patrón.

Las especificaciones de comparación incluyen toda la información para coincidir con una sola ranura y pueden efectuarse varias pruebas juntas; por ejemplo, el patrón:

```
(datos (x ~rojo&~verde))
```

generaría la siguiente especificación de comparación para la ranura *x*:

```
El valor x de ranura no es igual a la constante rojo
                    y no es igual a la constante verde
```

En condiciones normales, las uniones de variables no se verifican en la red patrón a menos que la variable se use más de una vez en el patrón; por ejemplo, el patrón:

```
(datos (x ?x) (y ?y) (z ?x))
```

no generaría especificaciones de comparación para la ranura *x* ni *y*, porque el primer enlace de una variable con un valor de ranura no tiene efecto si el patrón coincide con el hecho. Sin embargo, la ranura *z* debe tener el mismo valor que la ranura *x*, de modo que la especificación de comparación para la ranura *z* sería la siguiente:

```
El valor de ranura z es igual al valor de ranura x
```

Las expresiones que contienen variables que se encuentran dentro del patrón pueden revisarse dentro de la red patrón; por ejemplo, el patrón:

```
(datos (x ?x&: (> ?x ?y)))
```

no generaría una especificación de comparación para la ranura x porque la variable ?y no está dentro del patrón (suponiendo, por supuesto, que la variable ?y se definió en un patrón anterior), pero la ranura x del patrón

```
(datos (x ?x&:(> ?x 4)))
```

tendría la especificación de comparación:

```
El valor de ranura x es mayor que la constante 4
```

debido a que la única variable encontrada en la expresión, *?x*, también está dentro del patrón.

Como ya se estableció, la red patrón se organiza de manera jerárquica, colocando en la parte superior los nodos patrón que corresponden a las primeras restricciones de ranura de los patrones. Cuando se afirma un hecho, se revisan los nodos patrón para las primeras restricciones de ranura de la red patrón. Cualquiera de ellos cuya especificación de comparación se satisface, activará los nodos patrón que se encuentran directamente debajo de él. Este proceso continúa hasta alcanzar un nodo terminal de la red, los nodos terminales representan el final de un patrón y una coincidencia de patrón exitosa. Cada nodo terminal tiene una memoria **alfa** o **derecha** asociada a él, esta memoria contiene el conjunto de todos los hechos que han coincidido con el patrón asociado con el nodo terminal. En otras palabras, la memoria alfa almacena el conjunto de comparaciones de patrón para un patrón particular.

La red patrón aprovecha la similitud estructural al compartir nodos patrón comunes entre los diferentes patrones. Debido a que los nodos patrón se almacenan de manera jerárquica, dos patrones pueden compartir sus primeros *N* nodos patrón si las especificaciones de comparación para sus primeras *N* restricciones de ranura son idénticas. Por ejemplo, los patrones

```
(datos (x rojo) (y verde))
(datos (x rojo) (y azul))
```

pueden compartir nodos patrón para la ranura *x*. Observe que las especificaciones de comparación deben ser idénticas, y no necesariamente las restricciones de ranura que se encontraron en el patrón; Por ejemplo, los patrones

```
(datos (x ?x) (y ?x))
(datos (x ?y) (y ?y))
```

pueden compartir el nodo patrón para la ranura *y* aunque las variables usadas en ambos patrones sean diferentes. Las ranuras *x* no generan una especificación de comparación y, por tanto, se ignoran para efectos de compartir nodos. Sin embargo, cambiar el orden de las ranuras a

```
(datos (x ?x) (y ?x))
(datos (y ?y) (x ?y))
```

evitaría que los patrones compartan nodos, porque el primer patrón generaría una especificación de comparación para la ranura *y* y el segundo generaría una especificación de comparación para la ranura *x*.

En la figura 11.5 se muestra la red patrón generada mediante las siguientes reglas:

```
(defrule regla-1-Rete
   (comparar (un rojo))
   (datos (x ?x) (y ?x))
   =>)
(defrule regla-2-Rete
```



```
   (comparar (a ?x) (b rojo))
   (datos (x ~verde) (y ?x))
   (datos (x ?x) (y ?x))
   =>)
```

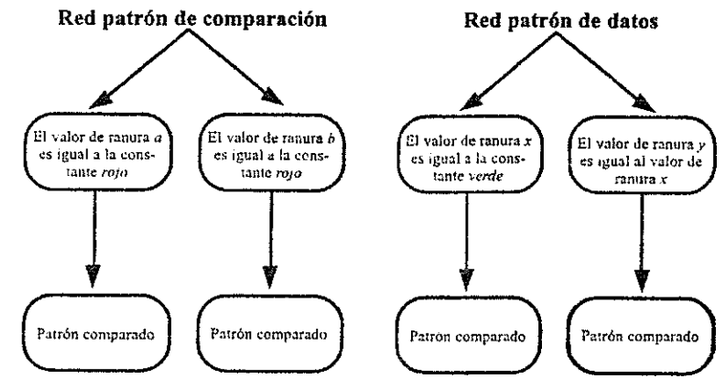


Figura 11.5 Red de patrón para dos reglas

11.4 LA RED DE UNIÓN

Una vez que se han determinado los patrones que han coincidido con los hechos, debe verificarse la comparación de los enlaces de variables a través de los patrones para cerciorarse que las variables usadas en más de un patrón tienen valores congruentes. Esta comparación se efectúa en la **red de unión,** cada nodo terminal de la red patrón actúa como una entrada a una **unión,** o **nodo de dos entradas,** de la red de unión. Cada unión contiene una especificación de comparación para las coincidencias de la memoria alfa asociada con su nodo terminal y para el conjunto de coincidencias parciales que han comparado patrones anteriores. Las coincidencias parciales para patrones previos se almacenan en la **memoria beta** o **izquierda** de la unión. Una regla con *N* patrones tendría $N-1$ uniones (aunque CLIPS en realidad usa *N* uniones para simplificar el algoritmo de Rete, lo que permite introducir cada unión mediante un solo patrón).

La primera unión compara los primeros dos patrones y el resto de las uniones compara un patrón adicional con las coincidencias parciales de la unión previa; por ejemplo, dada la defrule *Regla-2-Rete* utilizada en el ejemplo anterior:

```
(defrule regla-2-Rete
   (comparar (a ?x) (b rojo))
   (datos (x ~verde) (y ?x))
   (datos (x ?x))
   =>)
```

la primera unión contendría la especificación de comparación

```
El valor de ranura a del hecho
   limitado al primer patrón es igual a
el valor de ranura y del hecho
   limitado al segundo patrón.
```

La segunda unión recibiría como entrada el conjunto de coincidencias parciales de la primera unión y contendría la siguiente especificación de comparación:

```
El valor de ranura x del hecho
   limitado al tercer patrón es igual a
el valor de ranura y del hecho
   limitado al segundo patrón.
```

Observe que la variable *?x* del tercer patrón podría tener su valor comparado con la variable *?x* del primer patrón, en lugar de compararse con la variable *?x* del segundo patrón. La segunda aparición de *?x* en el tercer patrón no tiene que verificarse en la red de unión porque es posible hacerlo en la red patrón. Las redes patrón y de unión para la regla *Regla-2-Rete* se muestran en la figura 11.6.

La red de unión aprovecha la similitud estructural al compartir uniones entre reglas, las uniones de esta red pueden compartirse en dos reglas, si tienen patrones idénticos y comparaciones de unión para dos o más patrones, empezando con el primer patrón; por ejemplo, las reglas

```
(defrule compartido-1
   (comparar (a ?x) (b rojo))
   (datos (x ~verde) (y ?x))
   (datos ( x ?x) (y ?x))
   (otro (q ?z))
   =>)

(defrule compartido-2
   (comparar (a ?y) ( b rojo))
   (datos (x ~verde) (y ?y))
   (datos (x ?y) (y ?y))
   (otro (q ?y))
   =>)
```

compartirían todos sus patrones en la red patrón y las uniones creadas para sus primeros tres patrones de la red de unión, en tanto que la unión para el cuarto patrón no puede compartirse porque las especificaciones de comparación para las dos reglas son diferentes. La especificación de comparación para el cuarto patrón de la regla *compartido-1* no tiene que realizar ninguna comparación porque la variable *?z* no se usa en otros patrones, pero la especificación de comparación para el cuarto patrón de la regla *compartido-2* debe comparar la variable *?y* con la variable *?y* de otro patrón para g[illegible]tizar que las uniones de variable sean congruentes. Observe otra vez que los nombres de la v[illegible]le no tienen que ser idénticos para aprovechar la similitud estructural, lo importante es que [illegible]ecificaciones de comparación sean idénticas.

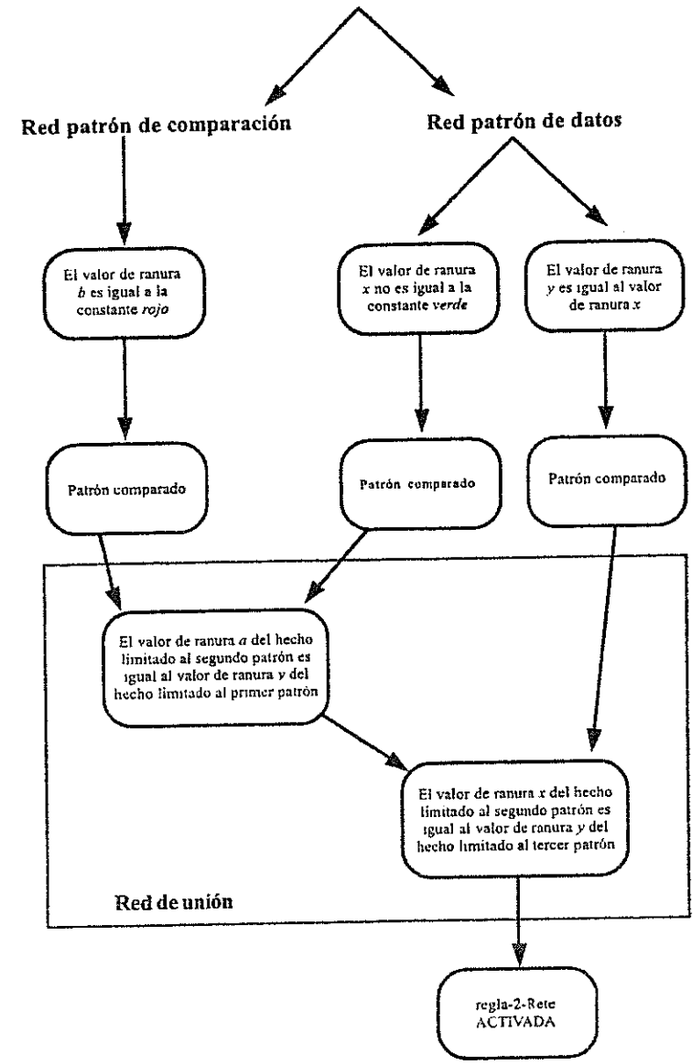


Figura 11.6 Redes patrón y de unión para la regla *regla-2-Rete*

CLIPS proporciona información útil para compartir una unión si se ha aplicado el comando *watch compilations*. Por ejemplo, los siguientes comandos ilustran cómo se despliega la información sobre elementos compartidos, suponiendo que las reglas *compartido-1* y *compartido-2* están en el archivo "reglas.clp".

```
CLIPS> (watch compilations)↵
CLIPS> (load "reglas.clp")↵
Defining defrule: compartido-1 +j+j+j+j
Defining defrule: compartido-2 =j=j=j+j
TRUE
CLIPS>
```

El *+j* en la salida indica que se está agregando una unión, mientras que *=j* indica que se está compartiendo; por tanto, cuando se agrega la regla *compartido-1* se crean cuatro uniones nuevas. Cuando se agrega la regla *compartido-2*, comparte sus primeras tres uniones con la regla *compartido-1* y crea una unión nueva para su último patrón. Observe que CLIPS ha utilizado cuatro uniones para representar la regla, contra las tres que se requerirían en condiciones normales. Para establecerlas, es conveniente tener sólo un patrón por unión; de modo que en lugar de usar una unión para los primeros dos patrones, CLIPS emplea dos.

11.5 LA IMPORTANCIA DEL ORDEN DE LOS PATRONES

Los nuevos programadores de lenguajes basados en reglas a menudo tienen conceptos erróneos sobre la importancia del correcto ordenamiento de los patrones en reglas para tener una velocidad y uso de memoria eficientes. Debido a que el algoritmo de Rete guarda el estado de un ciclo a otro, es muy importante asegurarse de que las reglas no generan grandes cantidades de coincidencias. Por ejemplo, considere el siguiente programa simple:

```
(deffacts información
   (encontrar-comparación a c e g)
   (elemento a)
   (elemento b)
   (elemento c)
   (elemento d)
   (elemento e)
   (elemento f)
   (elemento g))

(defrule comparación-1
   (encontrar-comparación ?x ?y ?z ?w)
   (elemento ?x)
   (elemento ?y)
   (elemento ?z)
   (elemento ?w)
   =>
   (assert (comparación-encontrada ?x ?y ?z ?w)))
```

Se restablecerá muy rápidamente; esto se verificará con un comando *watch facts* seguido por un comando *reset*. Ahora bien, en el siguiente programa:

```
(deffacts información
   (encontrar-comparación a c e g)
   (elemento a)
   (elemento b)
```



```
   (elemento c)
   (elemento d)
   (elemento e)
   (elemento f)
   (elemento g))

(defrule comparación-2
   (elemento ?x)
   (elemento ?y)
   (elemento ?z)
   (elemento ?w)
   (encontrar-comparación ?x ?y ?z ?w)
   =>
   (assert (comparación-encontrada ?x ?y ?z ?w)))
```

Un comando *watch facts* seguido por uno *reset* mostrarán un tiempo de reinicio mucho más lento. A medida que se realiza el reinicio, los primeros pocos hechos de la construcción deffacts se afirmarán con rapidez y los hechos subsecuentes necesitarán un tiempo cada vez mayor para afirmarse. Las reglas *comparación-1* y *comparación-2* tienen los mismos patrones, pero la segunda requiere mucho más tiempo para reiniciarse; en realidad, en algunas computadoras la regla *comparación-2* podría provocar que CLIPS se quede sin memoria. Puede demostrarse una diferencia aún mayor en cuanto a velocidad al agregar más hechos elemento al deffacts *información* (en realidad, en algunas computadoras llega a ser necesario agregar hechos *elemento* adicionales al deffacts *información* para ver una diferencia notable).

Conteo en la regla *comparación-1*

El conteo de las coincidencias de patrón y parciales en la regla *comparación-1* proporciona información útil. Se utilizará el identificador del hecho en lugar de todo el hecho de la lista de coincidencias de patrón y parciales. Además, las coincidencias parciales se encerrarán en llaves. Los identificadores de hecho son:

```
f-1 (encontrar-comparación a c e g)
f-2 (elemento a)
f-3 (elemento b)
f-4 (elemento c)
f-5 (elemento d)
f-6 (elemento e)
f-7 (elemento f)
f-8 (elemento g)
```

La regla *comparación-1* tiene las siguientes coincidencias parciales:

```
Patrón 1: f-1
Patrón 2: f-2, f-3, f-4, f-5, f-6, f-7, f-8
Patrón 3: f-2, f-3, f-4, f-5, f-6, f-7, f-8
Patrón 4: f-2, f-3, f-4, f-5, f-6, f-7, f-8
Patrón 5: f-2, f-3, f-4, f-5, f-6, f-7, f-8
```

y las siguientes coincidencias de patrón:

```
Patrón   1:   [f-1]
Patrones 1-2: [f-1,f-2]
Patrones 1-3: [f-1,f-2,f-4]
Patrones 1-4: [f-1,f-2,f-4,f-6]
Patrones 1-5: [f-1,f-2,f-4,f-6,f-8]
```

en total, la regla *comparación-1* tiene 29 coincidencias de patrón y 5 coincidencias parciales.

Conteo en la regla *comparación-2*

La regla *comparación-2* tiene las siguientes coincidencias de patrón:

```
Patrón 1: f-2, f-3, f-4, f-5, f-6, f-7, f-8
Patrón 2: f-2, f-3, f-4, f-5, f-6, f-7, f-8
Patrón 3: f-2, f-3, f-4, f-5, f-6, f-7, f-8
Patrón 4: f-2, f-3, f-4, f-5, f-6, f-7, f-8
Patrón 5: f-1
```

Las reglas *comparación-1* y *comparación-2* tienen el mismo número de coincidencias de patrón. Ahora consideremos sólo las coincidencias parciales para el patrón 1:

```
[f-2],[f-3],[f-4],[f-5],[f-6],[f-7],[f-8]
```

El patrón 1 tiene 7 coincidencias parciales, lo que no es sorprendente porque las coincidencias de patrón y parciales serán las mismas para el primer patrón. Sin embargo, las coincidencias parciales para los patrones 1 y 2 son numerosas:

```
[f-2],[f-2] ,  [f-2,f-3],[f-2,f-4],[f-2,f-5],
               [f-2,f-6],[f-2,f-7],[f-2,f-8],
[f,3],[f-2] ,  [f-3,f-3],[f-3,f-4],[f-3,f-5]
               [f-3,f-6],[f-3,f-7],[f-3,f-8],
[f-4],[f-2] ,  [f-4,f-3],[f-4,f-4],[f-4,f-5]
               [f-4,f-6],[f-4,f-7],[f-4,f-8],
[f-5],[f-2],   [f-5,f-3],[f-5,f-4],[f-5,f-5],
               [f-5,f-6],[f-5,f-7],[f-5,f-8],
[f-6],[f-2] ,  [f-6,f-3],[f-6,f-4],[f-6,f-5],
               [f-6,f-6],[f-6,f-7],[f-6,f-8],
[f-7],[f-2] ,  [f-7,f-3],[f-7,f-4],[f-7,f-5],
               [f-7,f-6],[f-7,f-7],[f-7,f-8],
[f-8],[f-2],   [f-8,f-3],[f-8,f-4],[f-8,f-5],
               [f-8,f-6],[f-8,f-7],[f-8,f-8]
```

En todas hay 49 coincidencias parciales para los patrones 1 y 2 (siete coincidencias de patrón para el patrón 1 multiplicadas por siete coincidencias de patrón para el patrón 2). El espacio limitará con rapidez las coincidencias parciales que pueden incluirse para los otros patrones porque habrá 343 coincidencias parciales para los patrones del 1 al 3 y 2401 coincidencias parciales para los patrones del 1 al 4. Igual que en la regla *comparación-1*, sólo existe una coincidencia parcial para los patrones 1 a 5.

```
[f-2,f-4,f-6,f-8,f-1]
```


Observe que a pesar de que el número de coincidencias de patrón y de activaciones son los mismos para ambas reglas, la regla *comparación-1* sólo tiene cinco coincidencias parciales, mientras que la regla *comparación-2* tiene 2801, una diferencia que seguirá creciendo a medida que se agreguen más hechos elemento. El hecho (elemento h) no creará coincidencias parciales nuevas para la regla *comparación-1*; sin embargo, creará 1880 coincidencias parciales nuevas para la regla *comparación-2*. Con este ejemplo se demuestra que el número de coincidencias parciales creadas puede afectar en gran medida el desempeño de un programa. Un conjunto eficiente de reglas no sólo debe tratar de reducir la creación de coincidencias parciales nuevas, sino también la supresión de coincidencias parciales antiguas, en efecto, es necesario tratar de reducir el cambio de estado del sistema de un ciclo al siguiente. Las técnicas específicas para disminuir el cambio de estado se explican más adelante en este mismo capítulo.

Comando matches

CLIPS tienen un comando de depuración llamado **matches** que desplegará las coincidencias de patrón, parciales y activaciones de una regla. Este comando es útil para encontrar reglas que generan grandes cantidades de coincidencias parciales y para depurar casos en los que una regla parece satisfacer todos sus patrones pero no se activa. La sintaxis del comando *matches* es:

```
(matches <nombre-regla>)
```

El argumento del comando *matches* es el nombre de la regla para la que se despliegan las coincidencias; el siguiente diálogo ilustra la salida este comando:

```
CLIPS> (clear)↵
CLIPS>
(defrule comparación-3)
   (encontrar-coincidencia ?x, ?y)
   (elemento ?x)
   (elemento ?y)
  =>
   (assert (comparación-encontrada ?x ?y)))↵
CLIPS>
(assert (encontrar-coincidencia a b)
        (encontrar-coincidencia c d)
        (encontrar-coincidencia e f)
        (elemento a)
        (elemento b)
        (elemento c)
        (elemento f))↵
<Fact-6>
CLIPS> (facts)↵
f-0     (encontrar-coincidencia a b)
f-1     (encontrar-coincidencia c d)
f-2     (encontrar-coincidencia e f)
f-3     (elemento a)
f-4     (elemento b)
f-5     (elemento c)
```

```
f-0     (elemento f)
For a total of 7 facts.
CLIPS> (matches comparación-3)↵
Matches for Pattern 1
f-0
f-1
f-2
Matches for Pattern 2
f-3
f-4
f-5
f-6
Matches for Pattern 3
f-3
f-4
f-5
f-6
Partial matches for CEs 1 - 2
f-1,f-5
f-0,f-3
Partial matches for CEs 1 - 3
f-0,f-3,f-4
Activations
f-0,f-3,f-4
CLIPS>
```

El primer patrón de la regla *comparación-3* tiene tres coincidencias de patrón, una para cada hecho *encontrar-coincidencia;* del mismo modo, el segundo y tercer patrones tienen cuatro coincidencias de patrón, una para cada hecho *elemento;* los primeros dos patrones tienen dos coincidencias parciales: una para los hechos (encontrar-coincidencia c d) y (elemento c), y otra para los hechos (encontrar comparación c d) y (elemento c), no hay coincidencia parcial asociada con el hecho (encontrar-coincidencia e f), ya que el hecho (elemento e) no existe; sólo hay una coincidencia parcial para los tres patrones: la coincidencia parcial para los hechos (comparar-hecho a b), (elemento a) y (elemento b), y también hay una activación para esta coincidencia parcial. Una vez que se ha disparado la regla *comparación-3* para esta activación ya no se desplegará mediante el comando *matches.*

Observación del cambio de estado

Aunque el comando *matches* proporciona una manera útil de examinar las coincidencias parciales de una regla, otra forma de observarlas es considerando activaciones parciales de una regla. Si se considera a la regla *comparación-1* como reglas separadas que tratan de calcular coincidencias parciales, es posible usar el comando *watch activations* para ver sus coincidencias parciales a medida que se generan. La regla *comparación-1* se puede dividir, como sigue:

```
(defrule c1-cp-1
        "Coincidencias parciales para el patrón 1"
   (encontrar-coincidencia ?x ?y ?z ?w)
   =>)
```



```
(defrule c1-cp-1-a-2
        "Coincidencias parciales para los patrones 1 y 2"
   (encontrar-coincidencia ?x ?y ?z ?w)
   (elemento ?x)
   =>)

(defrule c1-cp-1-a-3
        "Coincidencias parciales para los patrones 1 a 3"
   (encontrar-coincidencia ?x ?y ?z ?w)
   (elemento ?x)
   (elemento ?y)
   =>)

(defrule c1-cp-1-a-4
        "Coincidencias parciales para los patrones 1 a 4"
   (encontrar-coincidencia ?x ?y ?z ?w)
   (elemento ?x)
   (elemento ?y)
   (elemento ?z)
   =>)

(defrule comparación-1 "Activaciones para la regla de
coincidencia"
   (encontrar-coincidencia ?x ?y ?z ?w)
   (elemento ?x)
   (elemento ?y)
   (elemento ?z)
   (elemento ?w)
   =>)
   (assert (comparación-encontrada ?x ?y ?z ?w)))
```

Si estas reglas y el deffacts *información* están cargados, las coincidencias parciales pueden observarse a medida que se crean usando, el siguiente diálogo de comando:

```
CLIPS> (watch activations)↵
CLIPS> (watch facts)↵
CLIPS> (reset)↵
==> f-0     (hecho-inicial)
==> f-1     (encontrar-coincidencia a c e g)
==> Activation 0     c1-cp-1: f-1
==> f-2     (elemento a)
==> Activation 0     c1-cp-1-a-2: f-1,f-2
==> f-3     (elemento b)
==> f-4     (elemento c)
==> Activation 0     c1-cp-1-a-3: f-1,f-2,f-4
==> f-5     (elemento d)
==> f-6     (elemento e)
==> Activation 0     c1-cp-1-a-4: f-1,f-2,f-4,f-6
==> f-7     (elemento f)
==> f-8     (elemento g)
==> Activation 0     comparación-1: f-1,f-2,f-4,f-6,f-8
CLIPS>
```

El uso de la misma técnica en la regla *comparación-2* produciría cientos de activaciones parciales. Debe quedar muy claro que no es posible considerar al lado izquierdo de una regla como un todo para mejorar la eficiencia, cada lado izquierdo de una regla debe considerarse como varias reglas separadas, y cada una producirá un conjunto de coincidencias parciales para toda regla. La escritura de una regla eficiente se reduce a limitar no sólo el número total de activaciones para ella, sino también las coincidencias parciales para cada una de las subreglas separadas que componen su lado izquierdo.

11.6 ORDENAMIENTO DE PATRONES POR EFICIENCIA

Para limitar el número de coincidencias parciales que se realizan es necesario tener en mente varias directrices durante el ordenamiento de patrones; puede ser difícil encontrar la mejor manera de ordenar los patrones porque algunas directrices pueden estar en conflicto con otras. En general, las directrices para el ordenamiento se emplean para evitar ineficiencias básicas que pudieran aparecer en un sistema basado en reglas. El afinamiento de un sistema experto llega a requerir que se experimente mucho, mediante ensayo y error, el reordenamiento de patrones para determinar cuántos cambios hacen que el sistema se ejecute con mayor rapidez. A menudo, probar un método completamente diferente en lugar de tratar de afinar los patrones puede producir mejores resultados.

Los patrones más específicos van primero

Los patrones más específicos deben colocarse al frente del lado izquierdo de una regla, pues por regla general, un patrón específico tendrá el menor número de hechos de comparación en la lista de hechos y el mayor número de uniones de variable que restringen otros patrones. El patrón (comparar ?x ?y ?z ?w) utilizado en las reglas *comparación-1* y *comparación-2* es específico porque restringe los hechos permitidos para generar coincidencias parciales para los otros cuatro patrones de la regla.

Los patrones que comparan hechos volátiles van al final

Los patrones que se comparan con patrones que se agregan y se eliminan con frecuencia de la lista de hechos deben colocarse al final del lado izquierdo de una regla, esto produce un menor número de cambios en las coincidencias parciales de la regla. Es importante tomar en cuenta que los patrones que igualan hechos volátiles suelen ser los patrones más específicos de la regla, lo que llega a crear un dilema al tratar de organizar los patrones de la regla para alcanzar una mayor eficiencia. Por ejemplo, suele ser ventajoso colocar hechos de control como el patrón de inicio de una regla, de otra manera no se generarán coincidencias parciales para la regla. Sin embargo, si los hechos de control se afirman y se *retraen* con frecuencia, y un número mayor de coincidencias parciales se recalculan de manera constante, tal vez resulte mejor colocar el hecho de control al final de la regla.

Los patrones que igualan menos hechos van primero

La colocación de patrones que compararán muy pocos hechos de la lista de hechos cerca del frente de la regla reducirá el número de coincidencias parciales que puede generarse. Una vez más, el uso de esta directriz podría entrar en conflicto con otras directrices, por que una coincidencia de patrón que coincide con muy pocos hechos no es necesariamente el patrón más específico, o los hechos comparados por el patrón pueden ser volátiles.

11.7 LAS VARIABLES DE CAMPO MÚLTIPLE Y LA EFICIENCIA

Los comodines y las variables de campo múltiple proporcionan funciones eficaces de comparación de patrones, pero cuando se usan de manera inadecuada, pueden resultar poco eficientes. Para el uso de comodines y variables de campo múltiple deben aplicarse dos reglas: en primer lugar, no deben usarse a menos que se necesiten; en segundo, cuando se emplean, debe tenerse cuidado de limitar el número de comodines y variables de campo múltiple en una sola ranura de un patrón. La siguiente regla demuestra que los comodines y las variables de campo múltiple pueden ser útiles, pero también son muy costosas:

```
(defrule producir-pares
   (list (elementos $?b $?m $?e))
   =>
   (assert (frontal ?b))
   (assert (media ?m))
   (assert (posterior ?e)))
```

Dado un hecho como (list (elementos a 4 z 2)), esta regla producirá los hechos que representen las partes frontal, media y posterior de la lista, cuyas longitudes variarán desde cero hasta el total de la lista. Esta regla es fácil de establecer empleando variables de campo múltiple, sin embargo, es una operación de comparación de patrones muy costosa. En la tabla 11.1 se muestran todas las comparaciones que se intentaron y, también, los comodines y las variables de campo múltiple que pueden realizar una gran parte del proceso de comparación de patrones. En general, para los N campos contenidos en el hecho *elementos*, ocurrirán $(N^2 + 3N + 2)/2$ coincidencias para la regla *producir-pares*.

Intento de comparación	Campos comparados por $?b	Campos comparados por $?m	Campos comparados por $?e
1			a4z2
2		a	4z2
3		a4	z2
4		a4z	2
5		a4z2	
6	a		4z2
7	a	4	z2
8	a	4z	2
9	a	4z2	
10	a4		z2
11	a4	z	2
12	a4	z2	
13	a4z		2
14	a4z	2	
15	a4z2		

Tabla 11.1 Intentos de comparación para tres variables de campo múltiple

11.8 EL ELEMENTO CONDICIONAL TEST Y LA EFICIENCIA

Cualquier elemento condicional test dentro de una regla debe colocarse lo más cerca posible de la parte superior de la misma; por ejemplo, la siguiente regla trata de encontrar tres puntos distintos:

```
(defrule tres-puntos-distintos
   ?punto-1 <- (punto (x ?x1) (y ?y1))
   ?punto-2 <- (punto (x ?x2) (y ?y2))
   ?punto-3 <- (punto (x ?x3) (y ?y3))
   (test (and (neq ?punto-1 ?punto-2)
              (neq ?punto-2 ?punto-3)
              (neq ?punto-1 ?punto-3)))
   =>
   (assert (puntos-distintos (x1 ?x1) (y1 ?y1)
                             (x2 ?x2) (y2 ?y2)
                             (x3 ?x3) (y3 ?y3))))
```

El elemento condicional test que determina que la dirección del hecho ?punto-1 no es la misma que la dirección del hecho ?punto-2 puede colocarse inmediatamente después del segundo patrón. La ubicación del elemento condicional test en este punto reducirá el número de coincidencias parciales creadas.

```
(defrule tres-puntos-distintos
   ?punto-1 <- (punto (x ?x1) (y ?y1))
   ?punto-2 <- (punto (x ?x2) (y ?y2))
   (test (neq ?punto-1 ?punto-2)
   ?punto-3 <- (punto (x ?x3) (y ?y3))
   (test (and (neq ?punto-2 ?punto-3)
              (neq ?punto-1 ?punto-3)))
   =>
   (assert (puntos-distintos (x1 ?x1) (y1 ?y1)
                             (x2 ?x2) (y2 ?y2)
                             (x3 ?x3) (y3 ?y3))))
```

Los elementos condicionales test en el lado izquierdo de una regla siempre se evalúan cuando se generan las coincidencias parciales en la red de unión y las expresiones usadas con la restricción del campo de predicado o de igualdad pueden evaluarse durante el proceso de comparación de patrones si se cumplen ciertas condiciones. La evaluación de expresiones durante el proceso de comparación de patrones en la red patrón produce mayor eficiencia, una expresión utilizada por las restricciones del campo de predicado o del valor de devolución se evaluará durante el proceso de comparación de patrones si todas las variables a las que se hace referencia en la expresión se encuentran dentro del patrón que la incluye.

La expresión de la siguiente regla se evaluará durante la generación de coincidencias parciales porque está dentro de un elemento condicional test:

```
(defrule puntos-que-comparten-valor-x-o-y-común
   (punto (x ?x1) (y ?y1))
   (punto (x ?x2) (y ?y2))
   (test (or (= ?x1 ?x2) (= ?y1 ?y2)))
   =>
   (assert (valor-x-o-y-común
              (x1 ?x1) (y1 ?y1)
              (x2 ?x2) (y2 ?y2))))
```


La colocación de la expresión dentro del patrón no provoca su evaluación durante la comparación de patrón debido a que las variables *?x1* y *?y1* no están dentro del segundo patrón.

```
(defrule puntos-que-comparten-valor-x-o-y-común
   (punto (x ?x1) (y ?y1))
   (punto (x ?x2) (y ?y2&:(or (= ?x1 ?x2)
                              (= ?y1 ?y2))))
   =>
   (assert (valor-x-o-y-común
              (x1 ?x1) (y1 ?y1)
              (x2 ?x2) (y2 ?y2))))
```

Una vez más, la expresión de la siguiente regla se evaluará durante la generación de coincidencias parciales porque está dentro de un condicional test:

```
(defrule punto-no-en-diagonales-x-y ""
   (punto (x ?x1) (y  ?y1))
   (test (and (<> ?x1 ?y1) (<> ?x1 (- 0 ?y1))))
   =>
   (assert (punto-no-diagonal (x ?x1) (y ?y1))))
```

Sin embargo, esta vez la colocación de la expresión dentro del patrón permite que se evalúe durante la coincidencia del patrón porque las dos variables *?x1* y *?y1* están dentro del patrón que incluye la expresión:

```
(defrule punto-no-en-diagonales-x-y
   (punto (x ?x1)
          (y  ?y1&:(and (<> ?x1 ?y1)
                        (<> ?x1 (- 0 ?y1)))))
   =>
   (assert (punto-no-diagonal (x ?x1) (y ?y1))))
```

11.9 RESTRICCIONES INTEGRADAS DE COMPARACIÓN DE PATRONES

Las restricciones integradas de comparación de patrones siempre son más eficientes que una expresión equivalente que debe evaluarse; por ejemplo, una regla como

```
(defrule color-primario
   (color ?x&:(or (eq ?x rojo)
                  (eq ?x verde)
                  (eq ?x azul)))
   =>
   (assert (color-primario ?x)))
```

no debe usarse cuando es posible emplear las restricciones de comparación de patrones para obtener los mismos resultados, como se muestra aquí:

```
(defrule color-primario
   (color ?x&rojo|verde|azul)
   =>
   (assert (color-primario ?x)))
```

11.10 COMPARACIÓN DE REGLAS GENERALES CON REGLAS ESPECÍFICAS

No siempre es posible conocer si muchas reglas específicas funcionarán de manera más eficiente que menos reglas más generales, las reglas específicas tienden a aislar de la red patrón una gran parte del proceso de comparación de patrones, lo que reduce la cantidad de trabajo en la red de unión y con frecuencia proporcionan una mayor posibilidad de compartir en las redes patrón y de unión; mientras que una regla simple también puede ser más sostenible que un grupo más grande de reglas específicas. Sin embargo, las reglas generales necesitan escribirse con cuidado. Debido a que tienden a realizar el trabajo de varias reglas, es mucho más fácil escribir una regla general ineficiente que una regla específica ineficiente. Para ilustrar la diferencia entre las dos técnicas, consideremos la siguiente deftemplate y las cuatro reglas que actualizarán un hecho que contiene las coordenadas de un objeto que puede moverse hacia el norte, el sur, el este o el oeste, este hecho *ubicación* contiene las coordenadas cartesianas *x* y *y* del objeto.

```
(deftemplate ubicación (slot x) (slot y))

(defrule movimiento-norte
   (movimiento norte)
   ?ubicación-anterior <- (ubicación (y ?y-anterior))
   =>
   (modificar ?ubicación-anterior (y (+ ?y-anterior 1))))

(defrule movimiento-sur
   (movimiento sur)
   ?ubicación-anterior <- (ubicación (y ?y-anterior))
   =>
   (modificar ?ubicación-anterior (y (- ?y-anterior 1))))

(defrule movimiento-este
   (movimiento este)
   ?ubicación-anterior <- (ubicación (x ?x-anterior))
   =>
   (modificar ?ubicación-anterior (x (+ ?x-anterior 1))))

(defrule movimiento-oeste
   (movimiento oeste)
   ?ubicación-anterior <- (ubicación (x ?x-anterior))
   =>
   (modificar ?ubicación-anterior (x (- ?x-anterior 1))))
```

Estas cuatro reglas pueden reemplazarse con una regla general más simple, una deftemplate adicional y una construcción deffacts:


```
(deftemplate dirección
   (slot cuál-dirección)
   (slot desplazamiento-x)
   (slot desplazamiento-y)

(deffacts información-dirección
   (dirección (cuál-dirección norte)
          (desplazamiento-x 0) (desplazamiento-y 1))
   (dirección (cuál-dirección sur)
          (desplazamiento-x 0) (desplazamiento-y -1))
   (dirección (cuál-dirección este)
          (desplazamiento-x 1) (desplazamiento-y 0))
   (dirección (cuál-dirección oeste)
          (desplazamiento-x -1) (desplazamiento-y 0)))

(defrule dirección-movimiento
   (movimiento ?dir)
   (dirección (cuál-dirección ?dir)
              (desplazamiento-x ?dx)
              (desplazamiento-y ?dy))
   ?ubicación-anterior <- (ubicación (x ?x-anterior)
                                     (y ?y-anterior))

   =>
   (modificar ?ubicación-anterior (x (+ ?x-anterior ?dx))
                                  (y (+ ?y-anterior ?dy))))
```

Las variables *?dx* y *?dy* son los valores de *desplazamiento* de *x* y de *y*, que necesitan sumarse a los valores *x* y *y* de la ubicación anterior para obtener los valores *x* y *y* para la nueva ubicación.

Esta nueva regla requiere más trabajo al crear coincidencias parciales para determinar los valores de desplazamiento de *x* y de *y* para sumarlos a la ubicación actual, al tiempo que proporciona un nivel de abstracción que facilita la adición de más direcciones. Para el movimiento al noreste, sureste, noroeste y suroeste se requeriría la adición de cuatro nuevas reglas específicas, el ejemplo general sólo requiere la adición de cuatro nuevos hechos en la construcción deffacts, como se muestra a continuación:

```
(deffacts información-dirección
   (dirección (cuál-dirección norte)
          (desplazamiento-x 0) (desplazamiento-y 1))
   (dirección (cuál-dirección sur)
          (desplazamiento-x 0) (desplazamiento-y -1))
   (dirección (cuál-dirección este)
          (desplazamiento-x 1) (desplazamiento-y 0))
   (dirección (cuál-dirección oeste)
          (desplazamiento-x -1) (desplazamiento-y 0))
   (dirección (cuál-dirección noreste)
          (desplazamiento-x 1) (desplazamiento-y 1))
   (dirección (cuál-dirección sureste)
          (desplazamiento-x 1) (desplazamiento-y -1))
```

```
(dirección (cuál-dirección noroeste)
           (desplazamiento-x -1) (desplazamiento-y 1))
(dirección (cuál-dirección suroeste)
           (desplazamiento-x -1) (desplazamiento-y -1))
```

11.11 FUNCIONES DE PROCEDIMIENTOS

En ocasiones resulta útil (y más eficiente) realizar algunas operaciones mediante un paradigma de programación basado en procedimientos en lugar de uno basado en reglas. CLIPS proporciona algunas funciones para controlar el flujo de ejecución de acciones en el lado derecho de una regla, dos de éstas son las funciones ***while*** e ***if***, que también se encuentran como estructuras de control en lenguajes modernos de alto nivel como Ada, Pascal y C. Además, la función ***halt*** permite detener la ejecución de reglas desde el lado derecho de una regla.

CLIPS está diseñado para que sea un lenguaje basado en reglas eficiente, por lo que las funciones de procedimientos *if* y *while* sólo están destinadas para que se les use aplicando el buen juicio, pues la escritura de un programa de procedimientos extenso en el lado derecho de una regla anula todo el propósito del uso de un lenguaje basado en reglas. En general, las funciones *if* y *while* deben usarse para realizar pruebas y ciclos simples en el lado derecho de una regla y debe evitarse su anidado complejo en el lado derecho de una regla.

La función *if*

La sintaxis de la función *if* es

```
(if <expresión-predicado>
   then <expresión>+
   [else <expresión>+])
```

donde <expresión-predicado> es una expresión simple (como una función o una variable de predicado) y la <expresión>+ que sigue a las palabras clave *if* y *then* son una o más expresiones que se evalúan con base en el valor de devolución de la evaluación de <expresión-predicado>. (Observe que la cláusula *else* es opcional.)

Cuando se evalúa una función *if*, la condición que representa <expresión-predicado> se evalúa primero para determinar si se ejecutan las acciones de la cláusula *then* o de la cláusula *else*. Si la condición evalúa cualquier símbolo diferente a FALSE, se ejecutan las acciones de la cláusula *then* de la función, pero si evalúa el símbolo FALSE, se ejecutan las acciones de la cláusula *else*. Si no se ha incluido esta cláusula, entonces no se toma una acción bajo una condición falsa y una vez que se termina la ejecución de la función *if*, CLIPS continúa con la siguiente acción, si la hay, en el lado derecho.

La función *if* puede ser útil para revisar valores en el lado derecho de una regla en lugar de hacer que otras reglas realicen una prueba. Por ejemplo, la siguiente regla podría utilizarse para determinar si un programa debe continuar:

```
(defrule continuar-verificación
   ?fase <- (fase verificar-continuar)
   =>
   (retract ?fase)
   (printout t "¿Continuar? ")
   (bind ?respuesta (read))
```



```
   (if (or (eq ?respuesta y) (eq ?respuesta sí))
      (then (assert (fase continue))
      else (assert (fase halt))))
```

Observe que la función *if* se emplea para convertir la respuesta si o no del usuario en un hecho que indica el tipo de acción que habrá de tomarse, que en este caso, es continuar o parar.

El valor de devolución de esta función es la última expresión evaluada en las porciones *then* o *else* de la función. Si la <expresión-predicado> se evalúa como FALSE y no hay porción *then*, se devuelve el símbolo FALSE.

La función *while*

La sintaxis de la función *while* es

```
(while <expresión-predicado> [do]
    <expresión>*)
```

donde <expresión-predicado> es una expresión simple (como una función de predicado o una variable) y la <expresión>* que sigue a la palabra clave *do* son cero o más expresiones que se evalúan con base en el valor de devolución de la <expresión-predicado> evaluada. Estas expresiones componen el **cuerpo** del ciclo.

La parte de la función *while* representada por <expresión-predicado> se evalúa antes de que se ejecuten las acciones del cuerpo, y si se evalúa como cualquier valor diferente al símbolo FALSE, se ejecutarán las expresiones del cuerpo, pero si se evalúa con el símbolo FALSE, la ejecución continúa con la siguiente instrucción después de la función *while*, si es que la hay. La condición de la función *while* se revisará cada vez que se ejecute el cuerpo para determinar si debe ejecutarse otra vez.

La función *while* puede usarse con la función *if* para establecer una verificación de errores de entrada en el lado derecho de una regla. La siguiente modificación a la regla *continuar-verificación* emplea la función *while* para continuar el ciclo hasta que se recibe una respuesta apropiada:

```
(defrule continuar-verificación
   ?fase <- (fase verificar-continuar)
   =>
   (retract ?fase)
   (printout t "¿Continuar? ")
   (bind ?respuesta (read))
   (while (and (neq ?respuesta sí) (neq ?respuesta no))
      do
      (printout t "¿Continuar? ")
      (bind ?respuesta (read)))
   (if (eq ?respuesta sí)
      then (assert (fase continuar))
      else (assert (fase detener))))
```

La función *halt*

La función *halt* se emplea en el lado derecho de una regla para detener la ejecución de las reglas en agenda y no necesita argumentos. Cuando se le llama, no se ejecutarán otras accio-

nes del lado derecho de la regla que se dispara y el control regresará al indicador de nivel superior. La agenda contendrá cualquier otra regla activada cuando se llamó a la función *halt*.

Por ejemplo, la regla *continuar-verificación* podría reemplazar la acción:

```
(assert (fase halt))
```

con la siguiente acción:

```
(halt)
```

lo que detendría la ejecución de las reglas.

Esta función es útil sobre todo para detener la ejecución cuando el usuario trata de reiniciar más adelante la ejecución recurriendo al comando *run*. Considerando la siguiente modificación a la regla *continuar-verificación:*

```
(defrule continuar-verificación
   ?fase <- (fase verificar-continuar)
   =>
   (retract ?fase)
   (printout t "¿Continuar? ")
   (bind ?respuesta (read))
   (while (and (neq ?respuesta sí) (neq ?respuesta no))
       do
      (printout t "¿Continuar? ")
      (bind ?respuesta (read)))
   (assert (fase continuar))
   (if (neq ?respuesta sí)
      then (halt)))
```

Se puede observar que la regla afirma el hecho (fase continuar) sin tener en cuenta la respuesta del usuario. Con la afirmación de este hecho colocará las reglas apropiadas en la agenda para continuar la ejecución. Si la contestación a la pregunta sobre continuar no fue sí, la ejecución se detendrá mediante la función *halt;* luego el usuario podría examinar las reglas y los hechos, reiniciando más tarde la ejecución donde se quedó al emitir un comando *run*.

11.12 REGLAS SIMPLES COMPARADAS CON REGLAS COMPLEJAS

Los lenguajes basados en reglas permiten expresar muchos problemas de manera sencilla e incluso elegante. Aunque CLIPS no está diseñado específicamente para problemas algorítmicos o computacionales, puede emplearse con facilidad para encontrar el número más grande de un grupo. La siguiente regla y los deffacts asociados afirmarán un grupo de una muestra de números que se utilizará como datos de prueba mediante la regla que encuentra el número más grande:

```
(deffacts núm-máx
   (ciclo-máx 100))

(defrule afirmar-ciclo
   (ciclo-máx ?n)
   =>
   (bind ?i 1)
   (while (<= ?i ?n) do
      (assert (número ?i))
      (bind ?i (+ ?i 1))))
```


La manera más simple de encontrar el número más grande se demostró en el capítulo 9 y como se muestra a continuación, puede emplearse una regla para encontrar el número más grande:

```
(defrule número-más-grande
   (número ?número1)
   (not (número ?número2&:(< ?número2 ?número1)))
   =>
   (printout t "El número más grande es " ?número1 crlf))
```

Este regla, aunque simple, no es el método más rápido para encontrar el número más grande. Si *N* representa el número de hechos con la relación de *número,* entonces el primero y segundo patrones tendrán un número de coincidencias de patrón iguales a *N*, y aunque sólo habrá una coincidencia parcial para los primeros dos patrones, habrá que realizar *N comparaciones* al cuadrado para encontrar esa coincidencia parcial. El aumento de valor del hecho *ciclo-máx* a 200, 300, 400, etc., demostrará que el tiempo para ejecutar el problema es proporcional al cuadrado de *N*.

Este tipo de comparación es muy ineficiente porque a medida que se agrega cada número se compara con *todos* los demás para determinar si es el más grande; por ejemplo, si ya se han afirmado los hechos que representan del número 2 al 100 y se afirma el hecho que representa el número 1, se harán 199 comparaciones para determinar si 1 es el número más grande. El hecho (número 1) comparará el primer patrón, de modo que se comparará con los 99 hechos que coinciden con el segundo patrón para determinar si es el más grande; la primera comparación fallará (debido a que cualquiera de los números entre 2 y 100 será más grande), pero aún se comparará con el resto de los números. Del mismo modo, el hecho (número 1) comparará el segundo patrón, por lo que se comparará con los 100 hechos que coinciden con el primer patrón (que ahora también incluye el hecho (número 1)).

La clave para acelerar el programa es evitar que ocurran comparaciones innecesarias, esto puede lograrse con un hecho adicional para mantener el registro del número más grande y compararlo con los hechos *número*. Las siguientes reglas muestran cómo se hace esto:

```
(defrule probar-número
   (número ?n)
   =>
   (assert (probar-número ?n)))

(defrule más-grande-desconocido
   ?tratar <- (probar-número ?n)
   (not (más grande ?))
   =>
   (retract ?tratar)
   (assert (más grande ?n)))
```

```
(defrule más-pequeño-mayor
   ?anterior-más-grande <- (más grande ?n1)
   ?tratar <- (probar-número ?n2&:(> ?n2 ?n1))
   =>
   (retract ?anterior-más-grande ?tratar)
   (assert (más-grande ?n2)))

(defrule más-grande-mayor
   (más-grande ?n1)
   ?tratar <- (probar-número ?n2&:(<= ?n2 ?n1))
   =>
     (retract ?tratar))

(defrule imprimir-más-grande
   (declare (salience -1))
   (más-grande ?número)
   =>
   (printout t "El número más grande es " ?número crlf))
```

La ejecución de este programa para valores de *ciclo-máx* de 100, 200, 300, 400, etc., demostrará que el tiempo para ejecutar este programa es proporcional a *N*, lo que es aún más interesante considerando que el primer programa sólo disparará dos reglas, mientras que el segundo disparará aproximadamente 2*N* reglas. El segundo grupo de reglas demuestra que una regla debe tratar de limitar no sólo el número de coincidencias parciales que tiene, sino también el número de comparaciones que se necesitan para determinarlas. Como se demostró con la primera regla, si el primer patrón tiene *N* coincidencias y el segundo también tiene *N* coincidencias, entonces la regla hará *N* comparaciones al cuadrado para determinar las coincidencias parciales para los primeros dos patrones. Aunque no se generen coincidencias parciales, el tiempo utilizado para los cálculos será más o menos equivalente a haber generado *N* coincidencias parciales al cuadrado. El segundo grupo de reglas limita el número de comparaciones que se harán con *N*, y debido a que sólo existen a la vez un hecho *más grande* y uno *probar-número*, únicamente puede existir una coincidencia parcial en cualquier momento para las reglas *más-grande-desconocido, más-pequeño-mayor* y *más-grande-mayor*. Cómo se generan *N* hechos probar-número, esto limitará el tiempo de cálculo para la generación de *N* coincidencias parciales. A primera vista podría parecer que es posible generar más de un hecho *probar-número* (lo que causaría *N* coincidencias parciales al cuadrado), pero hay que recordar, sección 10.3, que la agenda funciona como una pila; esto significa que todas las activaciones *probar-número* colocadas en la agenda a través de la regla *afirmar-ciclo* siempre se dispararán después de cualquier activación de las reglas *más-grande-desconocido, más-pequeño-mayor* y *más-grande-mayor*. Puesto que estas reglas siempre eliminan el hecho actual *probar-número*, nunca habrá más de un hecho de este tipo.

Con este ejemplo se demuestran dos conceptos importantes: en primer lugar, la manera más fácil de codificar un problema en un lenguaje basado en reglas no es necesariamente la mejor; y, en segundo, a menudo es posible reducir el número de comparaciones si se usan hechos temporales para almacenar datos. El hecho *más-grande* se usa en este problema para almacenar el valor y hacer comparaciones posteriores, de manera que no tienen que buscarse todos los hechos *número* cuando se lleva a cabo la comparación.

11.13 CARGA Y GUARDADO DE HECHOS

La rapidez de un programa de CLIPS puede aumentarse al reducir el número de hechos de la lista de hechos. Un método para reducir el número de hechos es cargarlos en CLIPS sólo cuando se necesiten; por ejemplo, un programa para diagnosticar problemas con automóviles puede preguntar primero la marca y el modelo del carro y después cargar la información específica para ese automóvil. CLIPS proporciona las funciones **load-facts** y **save-facts** para permitir la carga de hechos a partir de un archivo o su guardado en él, la sintaxis de estas dos funciones es:

```
(load-facts <nombre-archivo>)

(save-facts <nombre-archivo>
            [<ámbito-guardado> <deftemplate-nombres>*])
```

donde <ámbito-guardado> se define como

```
visible | local
```

La función *load-facts* cargará un grupo de hechos almacenados en el archivo <nombre-archivo>, cuyos hechos deben estar en el formato estándar de un hecho ordenado o de deftemplate; por ejemplo, si el archivo "hechos.dat" contenía:

```
(datos 34)
(datos 89)
(datos 64)
(datos 34)
```

entonces el comando:

```
(load-facts "hechos.dat")
```

cargaría los hechos contenidos en ese archivo.

La función *save-facts* se usa para guardar hechos de la lista de hechos en el archivo <nombre-archivo>, los hechos se almacenan en el formato que requiere la función *load-facts*. Si <ámbito-guardado> no se especifica o se especifica como **local**, entonces sólo se guardarían en el archivo aquellos hechos que corresponden a deftemplates definidas en el módulo actual; si se especifica como **visible**, entonces todos los hechos que corresponden a deftemplates que son visibles en el módulo actual se guardarían en el archivo. Si se especifica <ámbito-guardado>, también pueden especificarse uno o más nombres de deftemplates y sólo se guardarán los hechos correspondientes a las deftemplates especificadas (sin embargo, los nombres de deftemplate aún deben satisfacer la especificación *local* o *visible*).

Tanto load-facts como save-facts devuelven TRUE si el archivo de hechos se abre y luego se carga o guarda con éxito; de lo contrario, devuelven FALSE. Es responsabilidad del programador cerciorarse de que las deftemplates correspondientes a los hechos de deftemplate de un archivo de hechos estén visibles en el módulo actual en el que se está ejecutando el comando *load-facts*.

11.14 RESUMEN

En este capítulo se demuestra la importancia de comparar de manera eficiente los hechos contra las reglas. El algoritmo de Rete es muy eficaz para este proceso de comparación porque aprovecha la redundancia temporal y la similitud estructural exhibidas en los sistemas expertos basados en reglas.

Las reglas se convierten en estructuras de datos en una red de reglas, que consta de una red patrón y una red de unión; la primera compara hechos con patrones y la segunda verifica que las uniones de variable a través de los patrones sean congruentes. El ordenamiento de patrones puede tener un efecto significativo sobre el desempeño de una regla; por lo general, los patrones más específicos y los que comparan menos hechos deben colocarse en primer lugar en el lado izquierdo de una regla y los patrones volátiles deben colocarse al final del lado izquierdo de una regla. El comando *matches* se utiliza para desplegar las coincidencias de patrón, coincidencias parciales y activaciones de una regla.

Hay otras técnicas para mejorar la eficiencia de reglas, incluyendo el uso apropiado de las variables de campo múltiple, la colocación de patrones test y el uso de restricciones integradas de comparación de patrones. También hay intercambios de eficiencia asociados con la escritura general en comparación con reglas específicas y reglas simples en comparación con reglas complejas.

Las funciones *if* y *while* se usan para flujo de control en el lado derecho de una regla y la función *halt* para detener la ejecución de reglas; pero el uso excesivo de estas funciones en el lado derecho de una regla se considera un estilo de programación deficiente. Las funciones *save-facts* y *load-facts* se usan para guardar hechos en un archivo y para cargar hechos desde un archivo.

PROBLEMAS

11.1 Enumere las especificaciones de **nodo patrón** generadas por las ranuras de los siguientes patrones:

```
(a) (dirigible (altitud 100))
(b) (dirigible (altitud ?x&:(> ?x 100)))
(c) (detener-luz (color ~rojo))
(d) (globo (color azul | blanco))
```

11.2 Elabore la red patrón para el siguiente grupo de patrones.

```
(datos (x rojo) (y ?y) (z ?y))
(datos (x ~rojo))
(elemento (b ?y) (c ?x&:(> ?x ?y)))
(elemento (a rojo) (b azul|amarillo))
```

11.3 Dibuje el patrón y la red de unión para las siguientes reglas, enumerando las expresiones evaluadas en cada nodo.

```
(defrule regla1
   (fase (nombre prueba))
   (datos (x ?x) (y ?y))
   (datos (x ?y) (y ?x&:(> ?x ?y)))
   =>)
```

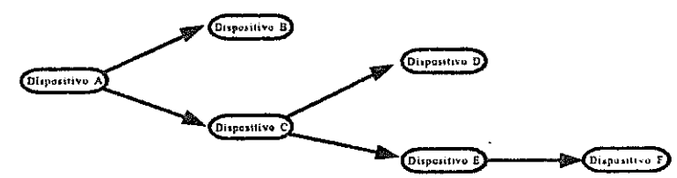


```
(defrule regla2
   (fase (nombre prueba))
   (datos (x ?x) (y ?y))
   (datos (x ?y&~rojo) (y ?x&~verde))
   =>)
```

11.4 Reescriba la siguiente regla para hacerla más eficiente:

```
(defrule regla-mala
   (elementos ( x ?x1) (y ?y1) (z ?z1))
   (elementos (x ?x2) (y ?y2) (z ?z2))
   (elementos (x ?x3) (y ?y3) (z ?z3))
   (rest (and (or (eq ?x3 verde)
                  (eq ?x3 rojo))
              (eq ?z2 ?y3)
              (> ?y1 ?x1)
              (< ?z1 ?x1)
              (neq ?z3 ?x1)))
=>)
```

11.5 En la siguiente figura se muestra una red de falla para una pieza hipotética de equipo; las flechas indican la dirección en que se propagará la falla, es decir si el componente A tiene una falla, entonces los componentes B y C también tendrán fallas:



Las siguientes reglas propagarán fallas por toda la red:

```
(defrule propagar-falla-dispositivo-A
   (falla dispositivo-A)
   =>
   (assert (falla dispositivo-B))
   (assert (falla dispositivo-C)))

(defrule propagar-falla-dispositivo-C
   (falla dispositivo-C)
   =>
   (assert (falla dispositivo-D))
   (assert (falla dispositivo-E)))

(defrule propagar-falla-dispositivo-E
   (falla dispositivo-E)
   =>
   (assert (falla dispositivo-F)))
```

Reescriba estas tres reglas como una regla general y una construcción de deffacts que propagará las fallas conforme lo hagan las tres reglas. Explique los cambios que deben hacerse para cada método cuando se agrega un dispositivo nuevo a la red de falla.

11.6 Reescriba la siguiente regla en una o más reglas que no usen las funciones *if* y *while*. Verifique que sus reglas realizan las mismas acciones mediante la comparación de la salida y la lista final de hechos de sus reglas con esta regla.

```
(defrule continuar-verificación
   ?fase <- (fase verificar-continuar)
   =>
   (retract ?fase)
   (printout t "¿Continuar? ")
   (bind ?respuesta (read))
   (while (and (neq ?respuesta sí) (neq ?respuesta no)) do
      (printout t "¿Continuar? ")
      (bind ?respuesta (read)))
(if (eq ?respuesta sí)
   (then (assert (fase continuar))
   else (assert (fase detener))))
```

11.7 Dado un tablero de ajedrez de $N x N$, donde N es un entero, escriba un programa que colocará N *reinas* en el tablero de manera que una reina no pueda capturar a otra reina. Sugerencia: desarrolle su programa usando 4 como número de filas y columnas. Éste es el número más pequeño para el que hay una solución (con excepción del caso trivial de un tablero de 1 x 1).

11.8 Modifique el programa desarrollado en el problema 10.12 de la página 474 para proporcionar la posibilidad de explicación. Después de que se imprima la salida para el mejor arbusto, el programa debe pedir al usuario que determine si desea una explicación para un arbusto específico. Si se oprime la tecla enter, el programa debe detenerse, si se introduce el nombre de un arbusto, el programa debe enumerar los requisitos satisfechos, los que no satisfizo y otros que podría satisfacer. Una vez que se imprima la explicación, se le debe pedir una vez más al usuario que determine si desea otra explicación.

11.9 Combine los programas desarrollados en los problemas 10.14 y 10.15 de las páginas 474 y 475. El programa resultante debe tener una interfaz de menú textual con elementos para el lanzamiento de una aplicación, la terminación de una aplicación y la salida del programa. Cuando se selecciona el elemento de menú de lanzamiento, se le debe pedir al usuario el nombre y los requisitos de memoria de la aplicación. Cuando se selecciona el elemento de terminación del menú, se le debe pedir al usuario el nombre de la aplicación que habrá de terminarse.

11.10 Modifique el programa desarrollado en el problema 10.15 de la página 475 para soportar submenús. Por ejemplo:

```
CLIPS> (run)

Seleccione una de las siguientes opciones:

1 -- Opción A
2 -- Opción B
3 -- Submenú 1
9 --  Salir del programa
```



```
Su elección: 3

Seleccione una de las siguientes opciones:

1 -- Opción C
2 -- Opción D
9 -- Menú anterior

Su elección: 9

Seleccione una de las siguientes opciones:

1 -- Opción A
2 -- Opción B
3 -- Submenú 1
9 -- Salir del programa

Su elección: 9
CLIPS>
```

11.11 Modifique el programa desarrollado en el problema 10.18 de la página 476 para programar seis cursos para un solo estudiante. El programa debe tratar de aumentar la calificación total (como se describió en el problema 10.18) para los seis cursos. La entrada al programa deben ser seis hechos que indiquen los cursos que se programarán y la salida debe ser la lista de cursos programados en orden de periodo, del uno al seis. Pruebe su programa con las siguientes selecciones de curso:

Curso	Instructores preferidos	Periodos preferidos	Instructores no preferidos	Periodos no preferidos
Historia regional	Hernández	2,5	-	1,3
Álgebra	Suárez	1,2	Jiménez	6
Educación física	-	-	Martínez, Reyes	1
Química	Dolby	5,6	-	-
Literatura	-	3,4	-	1,6
Alemán	-	-	-	-

11.12 Un jugador de póker ha repartido cinco cartas. Dando como entrada hechos que representan estas cinco cartas, escriba un programa que imprimirá el tipo de mano del jugador: flor imperial, flor de color, póker, full, flor, escalera, tercia, dos pares, un par o nada.

11.13 Escriba un programa que simplificará una ecuación algebraica moviendo todas las constantes al lado derecho de la ecuación y todas las variables al lado izquierdo de la ecuación y luego reduciendo los términos comunes. Por ejemplo, la ecuación

```
2 x + y + 5 + 3  y - 2 z - 8 = 3 z - 4 y + 4
```

puede simplificarse a:

```
2 x + 8 y - 5 z = 7
```

Puesto que el símbolo = tiene un significado especial en los patrones, tal vez desee representar la ecuación en un formato donde se el símbolo = esté implícito, por ejemplo:

```
(ecuación (lado izquierdo 2 x + y + 5 + 3 y - 2 z - 8)
          (lado derecho 3 z - 4 y + 4))
```

11.14 Combine los programas desarrollados en los problemas 9.20 de la página 435 y 11.10 para crear una interfaz controlada por menús para el programa de configuración. Las opciones del menú principal deben permitir al usuario seleccionar un chasis, agregar y quitar accesorios a la configuración e imprimir el costo de la configuración. Se deben emplear submenús para permitir la selección del chasis y para agregar o eliminar accesorios de la configuración. Después de la selección de un chasis o la adición o eliminación de un accesorio, deben imprimirse mensajes de advertencia que indiquen si hay conflictos entre todos los accesorios y el número de bahías y la energía eléctrica que proporciona el chasis. El control debe regresar al menú principal y la se-lección de la opción de menú para imprimir la configuración debe enumerar el chasis y los accesorios seleccionados junto con sus precios individuales y el precio total.

BIBLIOGRAFÍA

(Brownston 85). Lee Brownston *et al., Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*, Addison-Wesley Pub., 1985.

(Forgy 85), Charles L. Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", en *Artificial Intelligence*, **19**, pp. 17-37, 1985.

(Forgy 79). Charles L. Forgy, "On the Efficient Implementation of Production Systems", Ph. D. thesis, Carnegie-Mellon University, 1979.

CAPÍTULO 12

Ejemplos de diseño de sistemas expertos

12.1 INTRODUCCIÓN

En este capítulo se proporcionan varios ejemplos de programas de CLIPS. El primer ejemplo demuestra cómo se puede representar la incertidumbre en CLIPS. Los dos siguientes demuestran la manera en que se pueden emular otros paradigmas de representación de conocimiento empleando CLIPS: uno muestra la manera en que se pueden representar árboles de decisión y el otro reglas de encadenamiento hacia atrás. El cuarto y último ejemplo construye la estructura de un sistema experto simple que vigila un grupo de sensores.

12.2 FACTORES DE CERTIDUMBRE

CLIPS no tiene funciones integradas para el manejo de la incertidumbre, pero se le puede incorporar incertidumbre mediante la colocación de información que trate con incertidumbre directamente en hechos y reglas. A manera de ejemplo, en esta sección se emulará con CLIPS el mecanismo de incertidumbre de MYCIN, y se demostrará la manera en que se puede reescribir la siguiente regla de MYCIN (Firebaugh 88) en CLIPS:

```
SI
   La coloración del microorganismo es gramnegativo y
   La morfología del microorganismo es bacilo y
   El paciente es un huésped afectado
ENTONCES
   Hay evidencia que sugiere (0.6) de que la
     identidad del microorganismo es pseudomonas
```

MYCIN representa la información objetiva en forma de ternas objeto-atributo-valor OAV, que se pueden representar en CLIPS usando la siguiente construcción de deftemplate (que se colocará en su propio módulo para crear un componente de software reutilizable):

```
(defmodule OAV (exportar deftemplate oav))
(deftemplate OAV::oav
   (multislot objeto (type SYMBOL))
   (multislot atributo (type SYMBOL))
   (multislot valor))
```

Empleando esta deftemplate, algunos de los hechos que necesita la porción SI de la regla precedente de MYCIN serían

```
(oav (objeto microorganismo)
     (atributo coloración)
     (valor gramnegativo))

(oav (objeto microorganismo)
     (atributo morfología)
     (valor bastoncillo))

(oav (objeto paciente)
     (atributo es un)
     (valor huésped afectado))
```

MYCIN también asocia con cada hecho un factor de certidumbre (FC) que representa un grado de creencia en el hecho. El factor de certidumbre tiene un rango de –1 a 1; –1 significa que se sabe que el hecho es falso, 0 significa que no se tiene información (incertidumbre completa), y 1 significa que se sabe que es verdadero.

Debido a que CLIPS no maneja factores de certidumbre automáticamente, esta información también debe mantenerse; con este fin, se usará una ranura adicional en cada hecho para representar el factor de certidumbre y ahora la deftemplate *oav* para cada hecho se convierte en:

```
(deftemplate OAV::oav
   (multislot objeto (type SYMBOL))
   (multislot atributo (type SYMBOL))
   (multislot valor)
   (slot FC (type FLOAT) (range -1.0 +1.0)))
```

y los hechos de ejemplos serían:

```
(oav (objeto microorganismo)
     (atributo coloración)
     (valor gramnegativo)
     (FC 0.3))

(oav (objeto microorganismo)
     (atributo morfología)
     (valor bacilo)
     (FC 0.7))

(oav (objeto paciente)
     (atributo es un)
     (valor huésped afectado)
     (FC 0.8))
```


Todavía debe hacerse otra modificación en CLIPS para que los hechos *oav* funcionen de manera apropiada. MYCIN permite que la misma terna OAV se derive de reglas separadas, luego las ternas OAV se combinan para producir una sola que combine los factores de certidumbre de ambas. La deftemplate *oav* actual permitirá que sólo se afirmen dos ternas OAV idénticas si tienen diferentes factores de certidumbre (ya que CLIPS normalmente no permite que se afirmen dos hechos duplicados). Para permitir que se afirmen ternas OAV idénticas con los mismos factores de certidumbre, se emplea el comando **set-fact-duplication** para desactivar el funcionamiento de CLIPS que evita la afirmación de hechos duplicados. El comando

```
(set-fact-duplication TRUE)
```

desactiva el funcionamiento. De igual modo, el comando

```
(set-fact-duplication FALSE)
```

evitará la afirmación de hechos duplicados.

Como se dijo, MYCIN combinará dos ternas OAV idénticas en una sola con una certidumbre combinada. Si los factores de certidumbre de los dos hechos (representados por FC_1 y FC_2) son mayores o iguales a cero, MYCIN usa la siguiente fórmula para calcular el nuevo factor de certidumbre:

$$\text{Nueva certidumbre} = (FC_1 + FC_2) - (FC_1 * FC_2)$$

Por ejemplo, suponiendo que los siguientes hechos están en la lista de hechos:

```
(oav (objeto microorganismo)
     (atributo morfología)
     (valor bacilo)
     (FC 0.7))

(oav (objeto microorganismo)
     (atributo morfología)
     (valor bacilo)
     (FC 0.5))
```

y que $FC_1 = 0.7$ y $FC_2 = 0.5$; entonces la nueva certidumbre para la combinación de los dos hechos se calcula como:

$$\begin{aligned}\text{Nueva certidumbre} &= (0.7 + 0.5) - (0.7 * 0.5)\\ &= 1.2 - 0.35\\ &= 0.85\end{aligned}$$

y el nuevo hecho para reemplazar los dos hechos originales sería:

```
(oav (objeto microorganismo
     (atributo morfología)
     (valor bacilo)
     (FC 0.85))
```

Puesto que CLIPS no maneja automáticamente factores de certidumbre para hechos, se infiere que tampoco combina automáticamente dos ternas OAV derivadas de reglas diferentes; pero este tipo de combinación puede manejarse con facilidad mediante una regla que busque en la lista de hechos las ternas OAV idénticas que habrán de combinarse. La siguiente regla demuestra cómo se puede lograr esto cuando los factores de certidumbre de ambas ternas son mayores o iguales a cero:

```
(defrule OAV::combinar-dos-certidumbres-positivas
   (declare (auto-focus TRUE))
   ?hecho1 <- (oav (objeto $?o)
                   (atributo $?a)
                   (valor $?v)
                   (FC ?C1&: (>= ?C1 0)))
   ?hecho2 <- (oav (objeto $?o)
                   (atributo $?a)
                   (valor $?v)
                   (FC ?C2&: (>= ?C2 0)))
   (test (neq ?hecho1 ?hecho2))
   =>
   (retract ?hecho1)
   (bind ?C3 (- (+ ?C1 ?C2) (* ?C1 ?C2)))
   (modify ?Fact2 (FAC ?C3))
```

Observe que los identificadores de hecho *?hecho1* y *?hecho2* se comparan entre sí en el elemento condicional, *test*, esto es para garantizar que la regla no se compare utilizando exactamente el mismo hecho para los primeros dos patrones. Las funciones *eq* y *neq* son capaces de comparar direcciones de hecho. También observe que se ha desactivado el atributo auto-focus para la regla. Esto asegurará que se combinen dos ternas OAV antes de que se disparen otras reglas satifechas por ambas ternas.

El siguiente paso en el establecimiento de factores de certidumbre es vincular los factores de certidumbre de los hechos que igualan el lado izquierdo de la regla con los factores de certidumbre de los hechos afirmados por el lado derecho de la regla. En MYCIN los factores de certidumbre asociados con el lado izquierdo de una regla se derivan usando las siguientes fórmulas:

$$FC\ (P_1 \text{ o } P_2) = \text{máx}\ \{\ FC\ (P_1),\ FC(P_2)\ \}$$
$$FC\ (P_1 \text{ y } P_2) = \text{mín}\ \{\ FC\ (P_1),\ FC(P_2)\ \}$$
$$FC\ (\text{no } P) = -\ FC\ (P)$$

Donde P, P_1 y P_2 representan patrones del lado izquierdo. Además, si el factor de certidumbre del lado izquierdo es menor que 0.2, se considera que la regla es inaplicable y no se disparará.

El factor de certidumbre de un hecho afirmado del lado derecho de una regla se deriva mediante la multiplicación del factor de certidumbre de la afirmación por el factor de certidumbre del lado izquierdo de la regla. La siguiente traducción de la regla de MYCIN, a regla de CLIPS que se introdujo al principio de esta sección, demuestra el cálculo de los factores de certidumbre del lado izquierdo y del lado derecho. La regla se coloca en el módulo *IDENTIFICAR*, que importa la deftemplate *oav* del módulo *OAV:*


```
(defmodule IDENTIFICAR (import deftemplate oav de OAV))
(defrule IDENTIFICAR::MYCIN-traducción-CLIPS
   (oav (objeto microorganismo)
        (atributo coloración)
        (valor gramnegativo)
        (FC ?C1))
   (oav (objeto microorganismo)
        (atributo morfología)
        (valor bacilo)
        (FC ?C2))
   (oav (objeto paciente)
        (atributo es un)
        (valor huésped afectado)
        (FC ?C3))
   (test (> (mín ?C1 ?C2 ?C3) 0.2))
   =>
   (bind ?C4 (* (mín ?C1 ?C2 ?C3) 0.6))
   (asssert (oav (objeto microorganismo)
                 (atributo identidad)
                 (valor pseudomonas)
                 (FC ?C4))))
```

Se necesita un paso final para completar la emulación del factor de certidumbre de MYCIN. La regla *combinar-dos-certidumbres-positivas* sólo maneja un caso de combinación del factor de certidumbre, los casos de combinación restantes son los siguientes:

$$\text{Nueva certidumbre} = (FC_1 + FC_2) + (FC_1 * FC_2)$$
$$\text{si } (FC_1 \leq 0 \text{ y } FC_2 \leq 0$$

$$\text{Nueva certidumbre} = \frac{FC_1 + FC_2}{1 - \text{mín}\ \{\ |CF_1|\ ,\ |CF_2|\ \}}$$
$$\text{si } -1 < FC_1 * FC_2 < 0$$

12.3 ÁRBOLES DE DECISIÓN

En el capítulo 3 se menciona que los árboles de decisión proporcionan un paradigma útil para resolver ciertos tipos de problemas de clasificación, pues deducen soluciones al reducir el conjunto de posibles soluciones a una serie de decisiones o preguntas que acortan su espacio de búsqueda. Los problemas que son adecuados para su solución mediante árboles de decisión se tipifican con la característica de que proporcionan la respuesta a un problema a partir de un conjunto predeterminado de respuestas posibles. Por ejemplo, quizá un problema de taxonomía requiera la identificación de una gema del conjunto de todas las gemas conocidas o un problema de diagnóstico tal vez requiera la selección de un remedio posible de un conjunto de remedios o la selección de la causa de una falla de un conjunto de causas posibles. Puesto que debe predeterminarse el conjunto de respuestas, los árboles de decisión no funcionan bien para resolver problemas de programación, planeación o síntesis y, en general, problemas que deben producir soluciones además de seleccionar entre ellas.

Recuerde que un árbol de decisión se compone de nodos y ramas, los nodos representan ubicaciones en el árbol y las ramas conectan nodos padre con nodos hijo cuando se mueven

de la parte superior a la base y conectan nodos hijos con nodos padres cuando se mueven de la base a la parte superior; al nodo de la parte superior del árbol, que no tiene padre, se le llama nodo raíz y a los nodos sin hijos se les llama hojas. Observe que en un árbol, cada nodo sólo tiene un padre, con excepción del nodo raíz, que no tiene ninguno.

Los nodos hoja de un árbol de decisión representan todas las soluciones posibles que se derivan del árbol, a esos nodos se les conoce como *nodos de respuesta*, y a los demás como *nodos de decisión*. Cada nodo de decisión representa un cuestionamiento o decisión que, al preguntarse o decidirse, determina la rama apropiada que debe seguirse en el árbol de decisión. En los árboles de decisión simples esta pregunta podría tener respuestas de sí o no como "¿Es el animal de sangre caliente?", la rama izquierda del nodo representaría la ruta que se seguiría si la respuesta es sí y la rama derecha del nodo representaría la ruta que se seguiría si la respuesta es no. En general, un nodo de decisión puede usar cualquier criterio para seleccionar la rama a seguir, suponiendo que el proceso de selección siempre produce una sola rama. Por tanto, los nodos de decisión pueden seleccionar una rama que corresponde a un conjunto o rango de valores, una serie de casos o funciones que correlacionan el estado del nodo de decisión con las ramas del nodo de decisión. Los nodos de decisión sofisticados podrían permitir incluso razonamiento hacia atrás o probabilístico.

Para ilustrar la operación de un árbol de decisión, consideremos las siguientes heurísticas en la selección del vino apropiado para servir con un alimento:

```
SI el plato principal es carne roja
ENTONCES servir vino tinto

SI el plato principal es ave y es pavo
ENTONCES servir vino tinto

SI el plato principal es ave y no es pavo
ENTONCES servir vino blanco

SI el plato principal es pescado
ENTONCES servir vino blanco.
```

En la figura 12.1 se muestra la representación de las heurísticas del vino como un árbol de decisión binario, donde los nodos de decisión suponen que cada pregunta sólo puede responderse con sí o no y se ha agregado un nodo de respuesta predeterminado que contiene la respuesta "El mejor color es desconocido" al conjunto de heurísticas para el caso en que el plato principal no sea carne roja, ave, o pescado.

El procedimiento para recorrer el árbol y llegar a un nodo de respuesta es muy simple. El proceso de inferencia se inicia al establecer la ubicación actual del nodo raíz en el árbol de decisión; si la ubicación actual es un nodo de decisión, entonces la pregunta asociada con el nodo de decisión debe responderse de alguna manera (con frecuencia lo debe hacer la persona que consulta el árbol de decisión), si la respuesta es sí, entonces la ubicación actual se establece en el nodo hijo conectado a la rama sí (o izquierda) de la ubicación actual, si la respuesta es no, entonces la ubicación actual se establece en el nodo hijo conectado a la rama no (o derecha) de la ubicación actual. Si en cualquier punto un nodo de respuesta se vuelve la ubicación actual, entonces el valor del nodo de respuesta es la respuesta inferida mediante la consulta con el árbol de decisión, de lo contrario, el procedimiento para el manejo de un nodo de decisión se repite hasta que se alcance un nodo de respuesta. El pseudocódigo para este algoritmo es:


```
procedure Resolver_árbol_binario
   Establecer la ubicación del nodo raíz en el árbol
      al nodo raíz.
   while la ubicación actual es un nodo de decisión do
      Preguntar en el nodo actual.
      If la respuesta a la pregunta es sí
         Establecer el nodo actual en la rama sí.
      else
         Establecer el nodo actual en la rama no.
      end if
   end do
   Devolver la respuesta al nodo actual.
end procedure
```

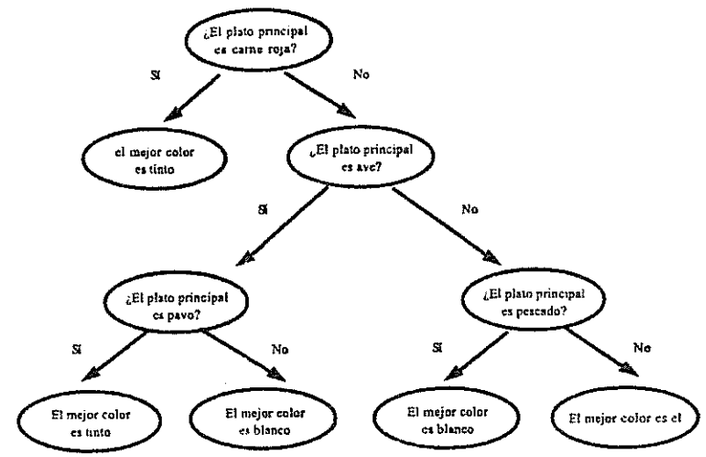


Figura 12.1 Árbol de decisión binario

Árboles de decisión con varias ramas

Los árboles de decisión que sólo permiten ramas binarias de un nodo de decisión dificultan la representación de una decisión que permita un conjunto de respuestas o una serie de casos. El árbol de decisión binario construido para el ejemplo del vino proporciona un buen ejemplo de esta ineficiencia, pues en caso de que el plato principal sea pescado, se deben tomar tres decisiones antes de que pueda determinarse que el mejor color es el blanco: deben hacerse las preguntas "¿El plato principal es carne roja?", "¿El plato principal es ave?" y ¿El plato principal es pescado?" Una pregunta mucho más directa que permitiría expresar el nodo de decisión de manera concisa es "¿Cuál es el plato principal?", y un nodo de decisión capaz de manejar esta pregunta debe permitir ramas múltiples, dada una serie de decisiones posibles (en este caso carne roja, ave, pescado y otros). En la figura 12.2 se muestra el árbol de decisión modificado de la figura 12.1, permitiendo ahora ramas múltiples realizadas mediante una modificación simple del algoritmo *Resolver_árbol_binario:*

```
procedure Resolver_árbol
   Establecer la ubicación actual del nodo raíz en el
árbol en el nodo raíz.
   while la ubicación actual es un nodo de decisión do
      Preguntar en el nodo actual hasta que
         se proporcione una respuesta válida para el
         conjunto de opciones de este nodo.
      Establecer el nodo actual en el nodo hijo de
         la rama asociada con la opción
         seleccionada.
   end do
   Devolver la respuesta al nodo actual.
end procedure
```

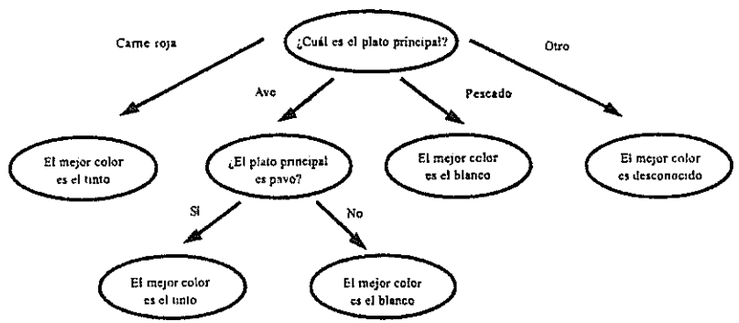


Figura 12.2 Árbol de decisión con ramas múltiples

Árboles de decisión que aprenden

En ocasiones es útil agregar conocimiento nuevo a un árbol de decisión a medida que se aprende, como en el ejemplo comúnmente usado del árbol de decisión de identificación de animales. Una vez que el árbol de decisión alcanza una respuesta, pregunta si la respuesta es correcta y, si es así, entonces no se hace nada más; pero si la respuesta es incorrecta, se modifica para ajustarse a la respuesta correcta. El nodo de respuesta se reemplaza con un nodo de decisión que contiene una pregunta que diferenciará entre la respuesta anterior que estaba en el nodo y la respuesta que no se dedujo correctamente. En la figura 12.3 se muestra un árbol de decisión que clasificará un animal conforme a sus características. Este árbol es un poco ingenuo (sólo conoce tres animales) y tiene necesidad de aprender.

El ejemplo de una sesión de deducción con el árbol podría darse como sigue:

```
¿El animal es de sangre caliente? (sí or no) sí↵
¿El animal ronronea? (sí or no) no↵
Deduzco que es un perro
¿Estoy en lo correcto? (sí or no) no↵
¿Cuál es el animal? pájaro↵
¿Qué pregunta distinguirá entre un pájaro y un perro
   cuando se responda sí? ¿El animal vuela?↵
```



```
Ahora puedo deducir que es un pájaro
¿Intentar otra vez? (sí or no) no↵
```

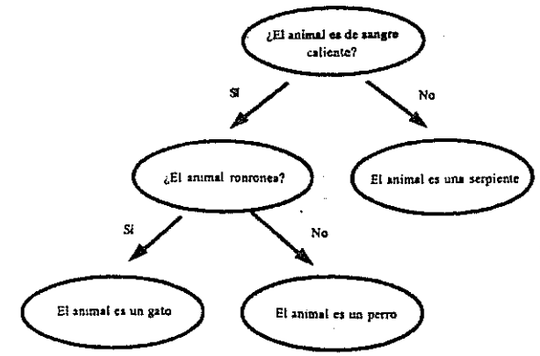


Figura 12.3 Árbol de decisión para identificar animales

La sesión puede continuar sin cesar, con el árbol de decisión aprendiendo cada vez más información. En la figura 12.4 se muestra la representación del árbol de decisión después de la sesión anterior, aunque una desventaja de este tipo de aprendizaje es que el árbol de decisión quizá no termine muy estructurado jerárquicamente o no sea muy eficiente en el descubrimiento del animal apropiado. Un árbol de decisión eficiente debe tener casi el mismo número de ramas del nodo raíz a los de respuesta para todas las rutas.

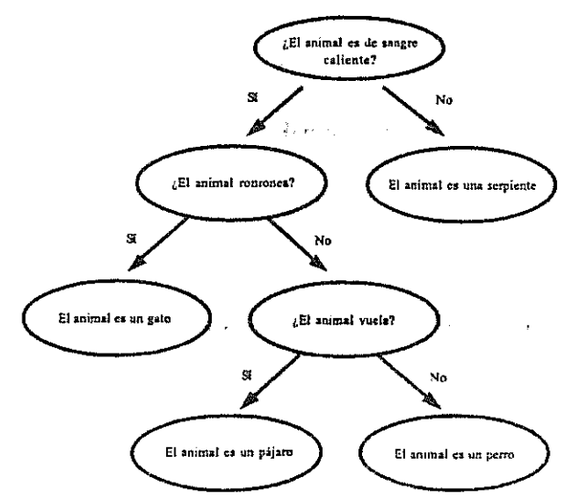


Figura 12.4 Árbol de decisión de identificación de animales después de aprender *pájaro*

El pseudocódigo para modificar el algoritmo *Resolver_árbol* e incorporar aprendizaje es:

```
procedure Resolver_árbol_ y_aprender
   Establecer la ubicación actual del nodo raíz
      en el árbol
   while la ubicación actual es un nodo de decisión do
      Preguntar en el nodo actual.
      if la respuesta a la pregunta es sí
         Establecer el nodo actual en la rama sí.
      else
         Establecer el nodo actual en la rama no.
      end if
   end do
   Preguntar si la respuesta en el nodo actual es
      correcta.
   if la respuesta es correcta
      Devolver la respuesta correcta.
   else
      Determinar la respuesta correcta.
      Determinar una pregunta que cuando se responda sí
         distinguirá entre la respuesta para el nodo
         actual de la respuesta correcta.
      Reemplazar el nodo de respuesta con un nodo de decisión
         que tiene como su rama no el nodo
         de respuesta actual y como su rama sí un
         nodo de respuesta con la respuesta correcta.
         La pregunta del nodo de decisión debe ser
         la pregunta que distingue los
         dos nodos de respuesta.
   end if
end procedure
```

Un programa de árbol de decisión basado en reglas

El primer paso para determinar cómo se puede establecer un árbol de decisión de aprendizaje en CLIPS es decidir cómo debe representarse el conocimiento. Debido a que el árbol de decisión debe aprender, tal vez valdrá la pena representarlo como hechos en lugar de reglas, porque los primeros se pueden agregar y eliminar con facilidad para actualizar el árbol a medida que aprende. Puede usarse un conjunto de reglas de CLIPS para recorrer el árbol de decisión estableciendo el algoritmo *Resolver_árbol_y_aprendizaje* con un método basado en reglas.

Cada nodo del árbol de decisión se representará con un hecho; la siguiente deftemplate se empleará para representar nodos de respuesta y de decisión:

```
(deftemplate nodo
   (slot nombre)
   (slot tipo)
   (slot pregunta)
   (slot nodo-sí)
   (slot nodo-no)
   (slot respuesta))
```


donde la ranura *nombre* es el nombre único para el nodo y la ranura *tipo* es el tipo del nodo y contiene el valor *respuesta* o *decisión*; las ranuras *pregunta, nodo-sí* y *nodo-no* sólo se usan para nodos de decisión. La ranura *pregunta* es la pregunta que se hace cuando se recorre un nodo de pregunta. La ranura *nodo-sí* es el nodo que habrá de proceder si la pregunta se responde de manera afirmativa, y la ranura *nodo-no* es el nodo que habrá de proceder si la pregunta se responde de manera negativa. La ranura *respuesta* sólo se emplea en nodos de respuesta y es la respuesta al árbol de decisión cuando se *recorre* un nodo de respuesta.

Debido a que el programa del animal aprenderá, será necesario almacenar información sobre lo que se ha aprendido de una ejecución del programa a la siguiente. Puesto que el árbol de decisión se estructurará como una colección de hechos, será útil almacenarlos en un archivo con el formato de comando *load-facts* y afirmarlos usando el comando *load-facts* cuando inicie el programa y guardarlos con el comando *save-facts* cuando finalice el programa. En este programa los hechos se almacenarán en un archivo llamado "animal.dat". Si se usa la figura 12.3 para el árbol de decisión inicial, el archivo "animal.dat" debe contener el siguiente texto. Observe que se ha rotulado el nodo raíz y que a cada uno de los otros nodos se le ha asignado un nombre único, también observe que algunas ranuras (como las ranuras de *decisión, nodo-sí* y *nodo-no* para los nodos de respuesta) permanecen sin especificar, ya que se les asignarán valores predeterminados (y nuestro programa no se ocupará de cuáles valores se colocan en las ranuras):

```
(nodo (nombre raíz) (type decisión)
      (pregunta "¿El animal es de sangre caliente?)
      (nodo-sí nodo1) (nodo-no nodo2))
(nodo (nombre nodo1) (type decisión)
      (pregunta "¿El animal ronronea?")
      (nodo-sí nodo3) (nodo-no nodo4))
(nodo (nombre nodo2) (type respuesta) (respuesta serpiente))
(nodo (nombre nodo3) (type respuesta (respuesta gato))
(nodo (nombre nodo4) (type respuesta (respuesta perro))
```

Ahora deben escribirse las reglas para recorrer el árbol de decisión; la siguiente regla iniciará el programa de árbol de decisión de aprendizaje:

```
(defrule iniciar
   (no (nodo (nombre raíz)))
   =>
   (load-facts "animal.dat")
   (assert (raíz de nodo-actual)))
```

Esta regla *iniciar* se disparará si el nodo raíz no está presente en la lista de hechos, las acciones de la regla *iniciar* cargan la representación del árbol de decisión en la lista de hechos y afirman un hecho que indica que el nodo actual de interés es el nodo raíz.

La siguiente regla hace la pregunta asociada con un nodo de decisión y luego afirma un hecho que contiene la respuesta a la pregunta:

```
(defrule hacer-pregunta-nodo-decisión
   ?nodo <- (nodo-actual ?nombre)
   (nodo (nombre ?nombre)
         (type decisión)
         (pregunta ?pregunta))
```

```
   (not (respuesta ?))
   =>
   (printout t ?pregunta " (sí or no) ")
   (assert (respuesta (read))))
```

El segundo patrón coincide con el nodo actual sólo si es un nodo de decisión, mientras que el tercer patrón revisa que la pregunta aún no se haya respondido, este patrón se empleará junto con la siguiente regla para que se reactive la regla *hacer-pregunta-nodo-decisión* si la respuesta a la pregunta es diferente a sí o no:

```
(defrule mala-respuesta
   ?respuesta <- (respuesta ~sí&~no)
   =>
   (retract ?respuesta))
```

Si la respuesta fue sí o no, entonces se disparará una de las dos siguientes reglas. El hecho *nodo actual* se retraerá y luego se actualizará con una nueva afirmación que depende de la respuesta a la pregunta y el hecho respuesta se retraerá, de modo que la regla *hacer-pregunta-nodo-decisión* se activará de nuevo:

```
(defrule proceder-a-rama-sí
   ?nodo <- (nodo-actual ?nombre)
   (nodo (nombre ?nombre)
         (type decisión)
         (nodo-sí ?rama-sí))
   ?respuesta <- (respuesta sí)
   =>
   (retract ?nodo ?respuesta)
   (assert (nodo-actual ?rama-sí)))

(defrule proceder-a-rama-no
   ?nodo <- (nodo-actual ?nombre)
   (nodo (nombre ?nombre)
         (type decisión)
         (nodo-no ?rama-no))
   ?respuesta <- (respuesta no)
   =>
   (retract ?nodo ?respuesta)
   (assert (nodo-actual ?rama-no)))
```

La siguiente regla pregunta si un nodo de respuesta ha hecho la deducción correcta y actúa del mismo modo que la regla *hacer-pregunta-nodo-decisión*.

```
(defrule preguntar-si-nodo-respuesta-es-correcto
   ?nodo <- (nodo-actual ?nombre)
   (nodo (nombre ?name) (type respuesta)
         (respuesta ?valor))
   (not (respuesta ?))
   =>
```



```
   (printout t " Deduzco que es un " ?valor crlf)
   (printout t "¿Estoy en lo correcto? (sí or no) ")
   (assert (respuesta (read))))
```

Si la respuesta fue diferente a sí o no, entonces la regla mala-respuesta provocará otra activación de la regla *preguntar-si-nodo-respuesta-es-correcto*; pero si la respuesta fue sí o no, entonces se disparará una de las dos reglas que se muestran a continuación. Si se verifica el valor del nodo de respuesta, entonces se afirma el hecho *preguntar-por-nuevo-intento* para indicar que se le debe pedir al usuario que continúe. Si la respuesta es incorrecta, entonces se presentará el aprendizaje y se afirmará el hecho *reemplazar-nodo-respuesta* para indicar el nombre del nodo que se reemplazará. En cualquier caso, los hechos *nodo-actual* y *respuesta* se retraen:

```
(defrule deducción-nodo-respuesta-es-correcta
   ?nodo <- (nodo-actual ?nombre)
   (nodo (nombre ?nombre) (type respuesta))
   ?respuesta <- (respuesta sí)
   =>
   (assert (preguntar-por-nuevo-intento))
   (retract ?nodo ?respuesta))

(defrule deducción-nodo-respuesta-es-incorrecto
   ?nodo <- (nodo-actual ?nombre)
   (nodo (nombre ?nombre) (type respuesta))
   ?respuesta <- (respuesta no)
   =>
   (assert (reemplazar-nodo-respuesta))
   (retract ?nodo ?respuesta))
```

Las tres reglas siguientes se usan para determinar si el usuario desea continuar, la regla preguntar-por-nuevo-intento hace la pregunta "¿Intentar otra vez?" y de nuevo se disparará la regla mala-respuesta si se proporciona otra respuesta diferente a sí o no; si la respuesta es sí, la regla una-vez-más restablecerá el hecho nodo-actual al nodo raíz para iniciar de nuevo el proceso de deducción; si la respuesta es no, los hechos que representan el árbol de decisión se guardan en el archivo "animal.dat" con el comando *save-facts*.

```
(defrule preguntar-por-nuevo-intento
   (preguntar-por-nuevo-intento)
   (not (respuesta ?))
   =>
   (printout t "¿Intentar otra vez? (sí or no) ")
   (assert (respuesta (read)))

(defrule una-vez-más
   ?fase <- (preguntar-por-nuevo-intento)
   ?respuesta <- (respuesta sí)
   =>
   (retract ?fase ?respuesta)
   (assert (raíz de nodo-actual)))
```

```
(defrule ya-no
   ?fase <- (preguntar-por-nuevo-intento)
   ?respuesta <- (respuesta no)
   =>
   (retract ?fase ?respuesta)
   (save-facts "Animal.dat" local node))
```

Por último, si la respuesta fue incorrecta, la siguiente regla agregará un nuevo nodo de decisión que permite que el árbol de decisión aprenda:

```
(defrule reemplazar-nodo-respuesta
   ?fase <- (reemplazar-nodo-respuesta ?nombre)
   ?datos <- (nodo (nombre ?nombre)
                   (type respuesta)
                   (respuesta ?valor))
   =>
   (retract ?fase)
   ; Determinar cuál debe haber sido la deducción
   (printout t "¿Cuál es el animal? ")
   (bind ?nuevo-animal (read))
   ; Obtener la pregunta para la deducción
   (printout t "Cuál pregunta cuando se responde sí ")
   (printout t "distinguirá " crlf "    un ")
   (printout t ?nuevo-animal " de un " ?valor "? ")
   (bind ?pregunta (readline))
   (printout t "Ahora puedo deducir " ?nuevo-animal crlf)
   ; Crear los nuevos nodos aprendidos
   (bind ?nuevonodo1 (gensym*))
   (bind ?nuevonodo2 (gensym*))
   (modify ?datos (type decisión)
                  (pregunta ?pregunta)
                  (nodo-sí ?nuevonodo1)
                  (nodo-no ?nuevonodo2))
   (assert (nodo  (nombre ?nuevonodo1)
                  (type respuesta)
                  (respuesta ?nuevo-animal)))
   (assert (nodo  (nombre ?nuevonodo1)
                  (type respuesta)
                  (respuesta ?valor)))
   ; Determinar si el jugador desea intentarlo de nuevo
   (assert (preguntar-por-nuevo-intento)))
```

La regla *reemplazar-nodo-respuesta* solicita la identificación correcta del animal y una pregunta que distinguirá entre este animal y el que el árbol de decisión identificó como la respuesta correcta. El nodo de respuesta previo se convierte en un nodo de decisión y se crean dos nodos de respuesta como respuestas a la pregunta recién aprendida, y luego se afirma el hecho *preguntar-por-nuevo-intento* para determinar si se debe ejecutar otra vez el programa. La función **gensym*** (que genera un símbolo único cada vez que se llama) se usa para proporcionar un nombre a cada uno de los dos nuevos nodos de respuesta.

Un registro gradual del programa de árbol de decisión

Es posible observar el comportamiento del programa de árbol de decisión al seguir su ejecución. Suponiendo que se han cargado las reglas del árbol de decisión y que se ha creado el archivo "animal.dat" que contiene los hechos que representan el árbol de decisión inicial, el siguiente diálogo muestra el estado del sistema después de que se ha aplicado un comando *reset:*

```
CLIPS> (watch facts)↵
CLIPS> (watch rules)↵
CLIPS> (reset)↵
==> f-0     (hecho-inicial)
CLIPS> (agenda)↵
0      inicializar:
For a total of 1 activation.
CLIPS>
```

La regla *inicializar* se activó por la ausencia del hecho *nodo raíz;* si se permite que se dispare la regla *inicializar*, se cargará en el árbol de decisión. Observe que una parte de la salida de los siguientes registros tiene sangría para mejorar su legibilidad.

```
CLIPS> (run 1)↵
FIRE    1 inicializar:
==> f-1     (nodo (nombre raíz) (type decisión)
        (pregunta "¿El animal es de sangre caliente?")
             (nodo-sí nodo1) (nodo-no nodo2)
             (respuesta nada))
==> f-2     (nodo (nombre nodo1) (type decisión)
             (pregunta "¿El animal ronronea?")
             (nodo-sí nodo 3) (nodo-no nodo4)
             (respuesta nada))
==> f-3     (nodo (nombre nodo2) (type respuesta)
             (pregunta nil) (nodo-sí nil)
             (nodo-no nil) (respuesta serpiente)
==> f-4     (nodo (nombre nodo3) (type respuesta)
             (pregunta nil) (nodo-sí nil)
             (nodo-no nil) (respuesta gato)
==> f-5     (nodo (nombre nodo4) (type respuesta)
             (pregunta nil) (nodo-sí nil)
             (nodo-no nil) (respuesta perro))
==> f-6     (raíz de nodo-actual)
CLIPS> (agenda)↵
0      hacer-pregunta-nodo-decisión: f-6, f-1,
For a total of 1 activation.
CLIPS>
```

La regla *inicializar* emplea la función load-facts para cargarse en el árbol de decisión. El hecho *nodo-actual* se establece en el hecho *nodo-raíz* y puesto que el nodo raíz es un nodo de decisión, se activa la regla *hacer-pregunta-nodo-decisión*. Si se permite que se disparen esta regla y la regla asociada *proceder-a-rama-sí*, se produce el siguiente diálogo:

```
CLIPS> (run 2)↵
FIRE    1 hacer-pregunta-nodo-decisión: f-6, f-1,
¿El animal es de sangre caliente? (sí or no) sí↵
==> f-7     (respuesta sí)
FIRE    2 proceder-a-rama-sí: f-6, f-1, f-7
<== f-6     (nodo raíz-actual)
<== f-7     (respuesta sí)
==> f-8     (nodo-actual nodo1)
CLIPS> (agenda)↵
0      hacer-pregunta-nodo-decisión: f-8, f-2,
For a total of 1 activation.
CLIPS>
```

Como la pregunta asociada con el nodo de decisión raíz es "¿El animal es de sangre caliente?" y la respuesta es sí, el nodo izquierdo del nodo de decisión (nodo1) se convierte en el nodo actual a través de la regla *proceder-a-rama-sí*. Debido a que el nodo1 también es un nodo de decisión, la regla *hacer-pregunta-nodo-decisión* también se activa. Si se permite que se disparen otra vez las dos reglas siguientes, se produce el diálogo siguiente:

```
CLIPS> (run 2)↵
FIRE    1 hacer-pregunta-nodo-decisión: f-8, f-2
¿El animal ronronea? (sí or no) no↵
==> f-9     (respuesta no)
FIRE    2 proceder-a-rama-no: f-8, f-2, f-9
<== f-8     (nodo-actual nodo1)
<== f-9     (respuesta no)
==> f-10    (nodo-actual nodo4)
CLIPS> (agenda)↵
0      preguntar-si-nodo-respuesta-es-correcto: f-10, f-5,
For a total of 1 activation
CLIPS>
```

Debido a que la pregunta asociada con el nodo de decisión nodo1 es "¿El animal ronronea?" y la respuesta es no, el nodo derecho del nodo de decisión, nodo4, se convierte en el nodo actual mediante la regla *proceder-a-rama-no*, y puesto que se trata de un nodo de respuesta, se activa la regla *preguntar-si-nodo-respuesta-es-correcto*. Si se permite que se disparen esta regla y la siguiente, se produce el diálogo siguiente:

```
CLIPS> (run 2)↵
FIRE  1 preguntar-si-nodo-respuesta-es-correcto: f-10, f-5,
Creo que es un perro
¿Estoy en lo correcto? (sí or no) no↵
==> f-11     (respuesta no)
FIRE  2 deducción-nodo-respuesta-es-incorrecta: f-10, f-5,
                                                f-11
```



```
==> f-12     (reemplazar-nodo-respuesta nodo4)
<== f-11     (respuesta no)
<== f-10     (nodo-actual nodo4)
CLIPS> (agenda)↵
0      reemplazar-nodo-respuesta: f-12, f-5
For a total of 1 activation.
CLIPS>
```

La deducción asociada con el nodo de respuesta es perro, pero como es incorrecta, se activa la regla *reemplazar-nodo-respuesta* para determinar la respuesta correcta. Si se permite que se dispare esta regla, se produce el siguiente diálogo:

```
CLIPS> (run 1)↵
FIRE    1 reemplazar-nodo-respuesta: f-12, f-5
<== f-12    (reemplazar-nodo-respuesta nodo4)
¿Qué animal es? pájaro↵
¿Qué pregunta distinguirá entre un pájaro y un perro
   cuando se responda sí? ¿El animal vuela?↵
Ahora puedo deducir que es un pájaro
<== f-5     (nodo (nombre nodo4) (type respuesta)
                  (pregunta nil)
                  (nodo-sí nil) (nodo-no nil)
                  (respuesta perro))
<== f-13    (nodo (nombre nodo4) (type decisión)
                  (pregunta "¿El animal vuela?")
                  (nodo-sí gen1) (nodo-no gen2)
                  (respuesta perro))
<== f-14    (nodo (nombre gen1) (type respuesta)
                  (pregunta nil)
                  (nodo-sí nil) (nodo-no nil)
                  (respuesta pájaro))
<== f-15    (nodo (nombre gen2) (type respuesta)
                  (pregunta nil)
                  (nodo-sí nil) (nodo-no nil)
                  (respuesta perro))
==> f-16    (preguntar-por-nuevo-intento)
CLIPS> (agenda)↵
0      preguntar-por-nuevo-intento: f-16,
For a total of 1 activation.
CLIPS>
```

En primer lugar, se retrae el hecho de control (reemplazar-nodo-respuesta nodo4), luego se determina la deducción correcta junto con la pregunta que la especificará. El nodo de respuesta incorrecta se modifica para que sea un nodo de pregunta y después se generan dos nodos de respuesta para el nuevo nodo de pregunta; por último, se afirma el hecho de control *preguntar-por-nuevo-intento* para determinar si se hace otra identificación. Si se permite que se dispare la regla *preguntar-por-nuevo-intento* y luego la regla *ya-no*, se produce el siguiente diálogo:

```
CLIPS> (run 2)↵
FIRE    1 preguntar-por-nuevo-intento: f-16,
¿Intentar otra vez? (sí or no) no↵
==> f-17     (respuesta no)
FIRE    2 ya-no: f-16, f-17
<== f-16     (preguntar-por-nuevo-intento)
<== f-17     (respuesta no)
CLIPS> (agenda)↵
CLIPS>
```

Mediante la regla *preguntar-por-nuevo-intento*, se le pregunta al usuario si se hace otra identificación, y como la respuesta es no, la regla *ya-no* guarda el árbol de decisión en el archivo "animal.dat". Después de esta sesión, la forma final del archivo "animal.dat" es

```
(nodo (nombre raíz) (type decisión)
      (pregunta "¿El animal es de sangre caliente?")
      (nodo-sí nodo1) (nodo-no nodo2)
      (respuesta nil)
(nodo (nombre nodo1) (type decisión)
      (pregunta "¿El animal ronronea?")
      (nodo-sí nodo3) (nodo-no nodo4)
      (respuesta nil)
(nodo (nombre nodo2) (type respuesta)
      (pregunta nil)
      (nodo-sí nil) (nodo-no nil) (respuesta serpiente))
(nodo (nombre nodo3) (type respuesta)
      (pregunta nil)
      (nodo-sí nil) (nodo-no nil) (respuesta gato))
(nodo (nombre nodo4) (type decisión)
      (pregunta "¿El animal vuela?")
      (nodo-sí gen1) (nodo-no gen2) (respuesta perro))
(nodo (nombre gen1) (type respuesta)
      (pregunta nil)
      (nodo-sí nil) (nodo-no nil) (respuesta pájaro))
(nodo (nombre gen2) (type respuesta)
      (pregunta nil)
      (nodo-sí nil) (nodo-no nil) (respuesta perro))
```

El nodo de respuesta nodo4 se ha reemplazado con un nodo de decisión conectado a dos nuevos nodos de respuesta. Además, el valor predeterminado *nil*, asignado automáticamente para algunas de las ranuras de la deftemplate, está ahora establecido explícitamente cuando se guardan los hechos.

12.4 ENCADENAMIENTO HACIA ATRÁS

CLIPS no aplica directamente el encadenamiento hacia atrás como parte de su mecanismo de inferencia, pero puede emularse usando reglas para el encadenamiento hacia delante de CLIPS, lo que se demostrará en esta sección es la manera en que se construye un sistema simple de encadenamiento hacia atrás. Debe notarse que CLIPS está diseñado para usarse como lenguaje de encadenamiento hacia adelante; si un método de encadenamiento hacia atrás es más

conveniente para resolver un problema, entonces debe usarse un lenguaje que establezca directamente el encadenamiento hacia atrás dentro de su mecanismo de inferencia, como PROLOG.

El sistema de encadenamiento hacia atrás de CLIPS se construirá con las siguientes capacidades y limitantes:

- Los hechos se representarán como pares atributo-valor.
- El encadenamiento hacia atrás iniciará con la afirmación de un atributo de objetivo inicial simple.
- Sólo se probará la igualdad de un atributo para un valor específico como una condición del antecedente de una regla.
- La única acción del antecedente de una regla será asignar el valor de un atributo simple.
- Si el valor del atributo objetivo no puede determinarse con reglas, el encadenador hacia atrás pedirá que se le proporcione tal valor. No es posible asignar un valor desconocido a los atributos.
- Un atributo sólo puede tener un valor simple. No se respaldará el razonamiento hipotético sobre diferentes valores de atributo de diferentes reglas.
- No se representará la incertidumbre.

Un algoritmo para el encadenador hacia atrás

Antes de escribir un mecanismo de inferencia de encadenamiento hacia atrás e intentar un método basado en reglas usando CLIPS, se debe considerar un algoritmo basado en procedimientos. El siguiente pseudocódigo de procedimiento puede utilizarse para determinar el valor de un atributo objetivo usando un método de encadenamiento hacia atrás (con las capacidades y limitantes ya explicadas:

```
procedure Resolver-objetivo (objetivo)
   objetivo: el objetivo actual a resolver

   if se conoce el valor del atributo objetivo
      Devolver el valor del atributo objetivo.
   end if

   for cada regla cuya consecuencia es el atributo
   objetivo do
     call Intentar_regla con la regla
     if Intentar_regla tiene éxito then
         Asignar al atributo objetivo el valor
         indicado mediante la consecuencia de la regla.
         Devolver el valor del atributo objetivo.
      end if
   end do

   Preguntar al usuario por el valor del atributo
   objetivo.
   Establecer el atributo objetivo al valor proporcionado
   por el usuario.
   Devolver el valor del atributo de objetivo.

end procedure
```

Se pasa un atributo objetivo al procedimiento *Resolver_objetivo* como argumento, éste determinará y devolverá el valor del atributo objetivo; verifica primero si ya se conoce el valor del atributo objetivo, quizá el valor ya se haya asignado como la consecuencia de otra regla o lo haya proporcionado el usuario del encadenador hacia atrás, si ya se conoce, se devuelve.

Si no se conoce, tratará de determinar el valor al encontrar una regla que asigne un valor al atributo objetivo como su consecuente. El procedimiento *Resolver_objetivo* probará cada regla que asigna un valor al atributo de objetivo como su consecuente hasta que una de las reglas tenga éxito. El procedimiento *Intentar_regla* (que se explica posteriormente) se da para probar cada una de las reglas con el atributo objetivo deseado, si se satisface el antecedente de la regla intentada, la regla tendrá éxito; de lo contrario fallará. Si ocurre lo primero, el valor de atributo en la consecuencia de la regla se asigna al atributo objetivo y este valor es devuelto mediante *Resolver_objetivo;* si la regla no tiene éxito, entonces se intenta la siguiente regla que asigna un valor al atributo objetivo como su consecuente.

Si ninguna de las reglas tiene éxito, se debe consultar al usuario para determinar el valor del atributo objetivo y el valor dado por el usuario se devuelve mediante el procedimiento *Resolver_objetivo.*

Intentar_regla se emplea para determinar si se satisface el antecedente de una regla; si es así, se puede usar la consecuencia para asignar el valor de un atributo objetivo. El pseudocódigo para este procedimiento es:

```
procedure Intentar_regla (regla)
   regla: regla que habrá de probarse para resolver el
objetivo
   for cada condición del antecedente
   de la regla do
      call Resolver_objetivo con atributo de condición
      if el valor devuelto por resolver_objetivo no es
      igual al valor requerido por la condición
      then
         Devolución sin éxito.
      end if
   end for

   Devolución con éxito

end procedure
```

Intentar_regla iniciará en la primera condición de una regla e intentará probarla antes de intentarlo en las condiciones subsecuentes. Para determinar si se satisface una condición, *Intentar_regla* debe conocer el valor del atributo probado en la condición y para determinarlo se llama recursivamente a *Resolver_objetivo.* Si el valor que devuelve este procedimiento no es igual al valor requerido por la condición, *Intentar_regla* regresa con el valor de devolución sin éxito (recuerde que sólo se prueba la igualdad en condiciones); de lo contrario, se prueba la siguiente condición de la regla. Si se satisfacen todas las condiciones de la regla, el procedimiento *Intentar_regla* regresa con el valor de devolución con éxito.

Representación de las reglas para encadenamiento hacia atrás con CLIPS

Una vez más, el primer paso en la solución de este problema es determinar cómo debe representarse el conocimiento. Puesto que CLIPS no realiza automáticamente el encadenamiento hacia atrás, será útil representar reglas de encadenamiento hacia atrás como hechos para que los antecedentes y las consecuencias se pueden examinar mediante reglas que actuarán como un motor de inferencia con encadenamiento hacia atrás. A continuación se muestra la deftemplate para representar reglas de encadenamiento hacia atrás que se almacenará en el defmodule *BC* (que se definirá al final de esta subsección, una vez que se hayan identificado todas las deftemplates necesarias para el de encadenamiento *hacia atrás*):

```
(deftemplate BC::regla
   (multislot si)
   (multislot entonces))
```

Las ranuras *si* y *entonces* almacenarán el antecedente y la consecuencia, respectivamente, de cada regla. Cada antecedente contendrá un par simple atributo-valor del formato

```
<atributo> es <valor>
```

o una serie de pares atributo-valor conectados por medio el símbolo *y.* Se permitirá que la consecuencia de cada regla sólo contenga un par atributo-valor sencillo.

Como ejemplo de representación de reglas usando este formato, consideremos el árbol de decisión de la figura 12.2. Es posible convertir con facilidad este árbol a reglas usando pares AV, como se explicó anteriormente; el pseudocódigo para las reglas convertidas es:

```
SI plato-principal es carne-roja
ENTONCES mejor-color es tinto

SI plato-principal es ave y
    comida-es-pavo es sí
ENTONCES mejor-color es rojo

SI plato-principal es ave y
     comida-es-pavo es no
ENTONCES mejor-color es blanco

SI plato-principal es pescado
ENTONCES mejor-color es blanco
```

Los atributos que se usan en las reglas son *plato-principal, comida-es-pavo* y *mejor-color,* el primero corresponde a la respuesta determinada por la pregunta del árbol de decisión "¿Cuál es el plato principal?", el segundo corresponde a la respuesta determinada por la pregunta "¿El plato principal es pavo?". Observe que la rama del árbol de decisión que determina que el mejor color es desconocido no se representa como una regla, ya que una de las limitantes de nuestro encadenador hacia atrás es que desconoce valores que no están representados. Si el plato principal no es ninguno de los que produce una respuesta, en este caso se preguntará al usuario el valor del atributo *mejor-color.*

La siguiente deffacts muestra como pueden representarse las reglas del vino con un formato de regla de encadenamiento *hacia atrás*. Debido a que esos hechos *regla* no son parte intrínseca del mecanismo de encadenamiento hacia atrás, se colocarán en el módulo *MAIN* (recuerde que este módulo se importa de los otros módulos, de modo que la deftemplate *regla* será visible para él).

```
(deffacts MAIN::reglas-vino
   (regla (if plato-principal es carne-roja)
          (then mejor-color es tinto))

   (regla (if plato-principal es pescado)
          (then mejor-color es blanco))

   (regla (if plato-principal es ave y
              comida-es-pavo es sí)
          (then mejor-color es tinto))

   (regla (if plato-principal es ave y
              comida-es-pavo es no)
          (then mejor-color es blanco)))
```

Esta representación proporciona una gran cantidad de flexibilidad cuando se manipulan las reglas de encadenamiento *hacia atrás*. Por ejemplo, si se determina el atributo *plato-principal* para que tenga el valor *ave*, los hechos:

```
(regla (if plato-principal es carne-roja)
       (then mejor-color es tinto))
```

y

```
(regla (if plato-principal es pescado)
       (then mejor-color es blanco))
```

pueden eliminarse de la lista de hechos para indicar que esas reglas no son aplicables, y los hechos:

```
(regla (if plato-principal es ave y
           comida-es-pavo es sí)
       (then mejor-color es tinto))
```

y

```
(regla (if plato-principal es ave y
           comida-es-pavo es no)
       (then mejor-color es blanco)))
```

pueden modificarse respectivamente como:

```
(regla (if comida-es-pavo es sí)
       (then mejor-color es tinto))
```


y

```
(regla (if comida-es-pavo es no)
       (then mejor-color es blanco))
```

para indicar que se ha satisfecho la primera condición de estas dos reglas.

A medida que avanza el encadenamiento *hacia atrás*, se generarán subobjetivos para determinar el valor de los atributos. Se necesitará un hecho para representar información sobre atributos de objetivo y se utilizarán hechos ordenados para representar los atributos objetivo y su formato será:

```
(deftemplate BC::objetivo
   (slot atributo))
```

En principio, el atributo de objetivo es *mejor-color*, lo que puede representarse con un deffacts como se muestra aquí:

```
(deffacts MAIN::objetivo-inicial
   (objetivo (atributo mejor-color)))
```

Cuando se determinan los valores para atributos, será necesario almacenarlos, lo que puede hacerse con la siguiente deftemplate:

```
(deftemplate BC::atributo
   (slot nombre)
   (slot valor))
```

Ahora que se han identificado todas las deftemplates, podemos dar la definición del módulo *BC*. Recuerde que cuando está cargando un archivo de construcciones, debe definirse el defmodule que contendrá otras construcciones antes de definir las construcciones que contiene.

```
(defmodule BC
   (exportar atributo objetivo de regla deftemplate))
```

El mecanismo de inferencia de encadenamiento hacia atrás de CLIPS

Este mecanismo se puede establecer con dos conjuntos de reglas. El primer grupo generará objetivos para los atributos y cuando éstos no se puedan determinar mediante reglas pedirá al usuario que los proporcione; el segundo grupo de reglas realizará operaciones de actualización, éstas incluyen modificación de reglas cuando sus condiciones han sido satisfechas y eliminación de objetivos cuando se han satisfecho. El primer conjunto de reglas es el siguiente:

```
(defrule BC::intentar-regla
   (objetivo (atributo ?nombre-o))
   (regla (if ?nombre-a $?)
          (then ?nombre-o $?)
   (not (atributo (nombre ?nombre-a)))
   (not (objetivo (atributo ?nombre-a)))
   =>
```

```
   (assert (objetivo (atributo ?nombre-a))))

(defrule BC::pedir-valor-atributo
   ?objetivo <- (objetivo (atributo ?nombre-o))
   (not (atributo (nombre ?nombre-o)))
   (not (regla (then ?nombre-o $?)))
   =>
   (retract ?objetivo)
   (printout t "Cuál es el valor de "
               ?nombre-o "? ")
   (assert (atributo (nombre ?nombre-o)
                     (valor (read)))))
```

La regla *intentar-regla* busca reglas cuyos antecedentes proporcionarán el valor para un atributo objetivo; el primer patrón se igualará con un hecho *objetivo*, el segundo busca todas las reglas cuyos antecedentes asignen un valor al atributo objetivo, el tercer patrón verificará que el valor del atributo de objetivo aún no se haya determinado y el cuarto confirma que aún no hay un objetivo para determinar el valor del atributo. Por cada regla encontrada, el lado derecho de la regla *intentar-regla* afirmará un objetivo para determinar el valor del atributo probado por la primera condición de la regla.

La regla *pedir-valor-atributo* es muy similar a la regla *intentar-regla*, sus primeros dos patrones son idénticos y su tercer patrón verifica que no queden reglas que puedan usarse para determinar el valor del atributo objetivo, en cuyo caso, se le pide al usuario que proporcione el valor del atributo; Se afirma un hecho que representa el valor de atributo y se retrae el hecho *objetivo* para el atributo.

Las cuatro reglas siguientes se usan para actualizar las reglas y objetivos de encadenamiento *hacia atrás* que se representan como hechos, a estas reglas se les ha asignado una prominencia de 100, de modo que las actualizaciones sucederán antes de hacer cualquier intento por generar objetivos nuevos o pedir al usuario los valores del atributo.

```
(defrule BC::objetivo-satisfecho
   (declare (salience 100))
   ?objetivo <- (objetivo (atributo ?nombre-o))
   (atributo (nombre ?nombre-o))
   =>
   (retract ?objetivo))

(defrule BC::regla-satisfecha
   (declare (salience 100))
   (objetivo (atributo ?nombre-o))
   (atributo (nombre ?nombre-a)
             (valor ?valor-a))
   ?regla <- (regla (if ?nombre-a es ?valor-a)
                    (then ?nombre-o es ?valor-o))
   =>
   (retract ?regla)
   (assert (atributo (nombre ?nombre-o)
                     (valor ?valor-o))))
```



```
(defrule BC::eliminar-regla-no-coincidente
   (declare (salience 100)
   (objetivo (atributo ?nombre-o))
   (atributo (nombre ?nombre-a) (valor ?valor-a))
   ?regla <- (regla (if ?nombre-a es ~?valor-a)
                    (then ?nombre-o es ?valor-o))
   =>
   (retract ?regla))

(defrule BC::modificar-regla-coincidente
   (declare (prominencia 100))
   (objetivo (atributo ?nombre-o))
   (atributo (nombre ?nombre-a) (valor ?nombre-a))
   ?regla <- (regla (if ?nombre-a es ?valor-a y
                        $?restar-si)
                    (then ?nombre-o es ?valor-o))
   =>
   (retract ?regla)
   (modify ?regla (if $?restar-si)))
```

La regla *objetivo-satisfecho* elimina cualquier objetivo para el que se ha determinado el valor de atributo.

La regla *regla-satisfecha* busca cualquier regla que tenga una condición restante; si existe un atributo que satisface esta condición restante y hay un objetivo para determinar el valor de este atributo, entonces se agrega el valor de atributo de la consecuencia de la regla a la lista de hechos.

La regla *eliminar-regla-no-coincidente* busca reglas cuyos antecedentes puedan proporcionar el valor del atributo objetivo y contengan una o más condiciones de la cuales la primera entre en conflicto con un valor asignado a un atributo. Si éste es el caso, la regla se elimina de la lista de hechos porque no es aplicable.

La regla *modificar-regla-coincidente* busca reglas cuyos antecedentes puedan proporcionar el valor para un atributo objetivo y contengan dos o más condiciones de las cuales la primera se satisface mediante un valor asignado a un atributo. Si se encuentra una regla de este tipo, se elimina la primera condición de la regla para dejar las condiciones restantes, que deben probarse.

Ahora que se han proporcionado todas las reglas de encadenamiento hacia atrás, lo único que se necesita para iniciar el proceso de encadenamiento hacia atrás es concentrase en el módulo *BC*, lo que se consigue agregando la siguiente regla al módulo *MAIN:*

```
(defrule MAIN::inicializar-BC
   =>
   (focus BC))
```

Un registro gradual del encadenador hacia atrás

Es posible observar el comportamiento del mecanismo de inferencia de encadenamiento *hacia atrás* de CLIPS establecido mediante reglas al observar su ejecución. Suponiendo que se han cargado los deffacts *reglas-vino* y *objetivo-inicial* junto con las reglas del mecanismo de inferencia de encadenamiento *hacia atrás*, el estado inicial del sistema después de un comando *reset* es como sigue (otra vez se ha colocado sangría a una parte de la salida para mejorar la legibilidad):

```
CLIPS> (unwatch all)↵
CLIPS> (reset)↵
CLIPS> (facts)↵
f-0     (hecho-inicial)
f-1     (objetivo (atributo mejor-color))
f-2     (regla (if plato-principal es carne-roja)
               (then mejor-color es tinto))
f-3     (regla (if plato-principal es pescado)
               (then mejor-color es blanco))
f-4     (regla (if plato-principal es ave y
                   comida-es-pavo es sí)
               (then mejor-color es tinto))
f-5     (regla (if plato-principal es ave y
                   comida-es-pavo es no)
               (then mejor-color es blanco))
For a total of 6 facts.
CLIPS> (agenda)↵
0       iniciar-BC: f-0
For a total of 1 activation.
CLIPS>
```

La regla *iniciar-BC* se concentra únicamente en el módulo *BC* y, una vez que lo dispara, se vuelve el punto de enfoque actual.

```
CLIPS> (run 1)↵
CLIPS> (agenda)↵
0       intentar-regla: f-1,f-5,,
0       intentar-regla: f-1,f-4,,
0       intentar-regla: f-1,f-3,,
0       intentar-regla: f-1,f-2,,
For a total of 4 activations.
CLIPS>
```

Observe que la agenda contiene cuatro activaciones de la regla *intentar-regla*. Como el objetivo inicial es determinar el valor del atributo *mejor-color* como se especificó con el hecho f-1 y debido a que las consecuencias de los hechos *reglas* f-2, f-3, f-4 y f-5 le asignan uno, es necesario intentar cada una de estas reglas para satisfacer el objetivo de ese atributo.

El siguiente paso de la ejecución es disparar la primera activación de la regla *intentar-regla*. Los elementos de observación de reglas y de hechos se activan antes de que se dispare la regla:

```
CLIPS> (watch rules)↵
CLIPS> (watch facts)↵
CLIPS> (run 1)↵
FIRE    1 intentar-regla: f-1,f-5,,
==> f-6      (objetivo (atributo plato-principal))
CLIPS> (agenda)↵
0      pedir-valor-atributo: f-6,,
For a total of 1 activation.
CLIPS>
```


La regla *intentar-regla* se dispara basándose parcialmente en el hecho f-5, que representa la regla de encadenamiento *hacia atrás* que se muestra aquí:

```
SI plato-principal es ave y
     comida-es-pavo es no
ENTONCES  mejor-color es blanco
```

Antes de que pueda aplicarse esta regla para asignar valor al atributo *mejor-color*, deben satisfacerse los elementos condicionales de sus antecedentes. La primera condición requiere el valor del atributo plato-principal; debido a que este atributo es desconocido, se crea un objetivo para él y se representa con el hecho f-6; y como no hay reglas que asignen el valor del atributo *plato-principal*, se activa la regla *pedir-valor-atributo*.

Siguiendo con la ejecución, la regla *pedir-valor-atributo* se dispara para determinar el valor del atributo *plato-principal*.

```
CLIPS (run 1)↵
FIRE   1 pedir-valor-atributo: f-6,,
<== f-6    (objetivo (atributo plato-principal))
¿Cuál es el valor de plato-principal?) pollo↵
==> f-7    (atributo (nombre plato-principal)
                     (valor pollo))
CLIPS> (agenda)↵
100    eliminar-regla-no-coincidente: f-1,f-7,f-3
100    eliminar-regla-no-coincidente: f-1,f-7,f-2
100    modificar-regla-coincidente: f-1,f-7,f-5
100    modificar-regla-coincidente: f-1,f-7,f-4
For a total of 4 activations.
CLIPS>
```

Puesto que se le pide al usuario que proporcione el valor del atributo *plato-principal*, se modifica el objetivo para este atributo f-6 y el valor proporcionado por el usuario se afirma como el hecho f-7 de *atributo*, esto provoca que cuatro nuevas activaciones se coloquen en la agenda. Las reglas representadas por los hechos f-4 y f-5 requieren como primera condición que el atributo *plato-principal* sea *ave*, pero como el valor del atributo es *pavo*, ambos hechos *regla* deben modificarse para representar que se ha satisfecho la primera condición; de esta manera, ambos hechos provocan activaciones de la regla *modificar-regla-coincidente*. Las reglas representadas por los hechos f-2 y f-3 requieren como primera condición que el atributo *plato-principal* sea algo diferente a *pollo*; por tanto, ya no es aplicable ninguno de esos hechos y la regla *eliminar-regla-no-coincidente* se activa para eliminarlos.

Si se permite que se disparen las dos activaciones de *eliminar-regla-no-coincidente*, se produce la siguiente salida:

```
CLIPS> (run 2)↵
FIRE    1 eliminar-regla-no-coincidente: f-1,f-7,f-3
<== f-3     (regla (if plato-principal es pescado)
                   (then mejor-color es blanco))
FIRE    2 eliminar-regla-no-coincidente: f-1,f-7,f-2
```

```
<== f-2      (regla (if plato-principal es carne-roja)
                    (then mejor-color es tinto))
CLIPS> (agenda)↵
100     modificar-regla-coincidente: f-1,f-7,f-5
100     modificar-regla-coincidente: f-1,f-7,f-4
For a total of 2 activations.
CLIPS>
```

Los hechos f-2 y f-3 se eliminan de la lista de hechos, lo que indica que las reglas representadas por ellos ya no son aplicables y sus activaciones de *intentar-regla* se eliminan de la agenda.

Si ahora se pasa a la ejecución de las dos activaciones *modificar-regla-coincidente*, se produce la siguiente salida:

```
CLIPS> (run 2)↵
FIRE    1 modificar-regla-coincidente: f-1,f-7,f-5
<== f-5      (regla (if plato-principal es ave y
                        comida-es-pavo es no)
                    (then mejor-color es blanco))
==> f-8      (regla (if comida-es-pavo es no)
                    (then mejor-color es blanco))
FIRE    2 modificar-regla-coincidente: f-7,f-4
<== f-4      (regla (if plato-principal es ave y
                        comida-es-pavo es sí)
                    (then mejor-color es tinto))
==> f-9      (regla (if comida-es-pavo es sí)
                    (then mejor color es tinto))
CLIPS> (agenda)↵
0       intentar-regla: f-1,f-9,,
0       intentar-regla: f-1,f-8,,
For a total of 2 activations.
CLIPS>
```

En parte, el primer disparo de la regla *modificar-regla-coincidente* se basa en el hecho f-5, que representa la siguiente regla de encadenamiento *hacia atrás*:

```
Si el plato-principal es ave y
   comida-es-pavo es no
ENTONCES mejor-color es blanco
```

Las acciones de la regla *modificar-regla-coincidente* modifican esta regla de encadenamiento hacia atrás como:

```
SI comida-es-pavo es no
ENTONCES  mejor-color es blanco
```

El nuevo hecho que representa la regla modificada es f-8, y describe las condiciones de la regla inicial que permanecen luego que se ha satisfecho la primera condición y provoca otra activación de la regla *intentar-regla* para esta regla de encadenamiento *hacia atrás*. Esta nueva activación afirmará un objetivo nuevo para determinar el valor de *comida-es-pavo*, de

modo que puede aplicarse la consecuencia de la regla para asignar el valor del atributo *mejor-color*.

El segundo disparo de la regla *modificar-regla-coincidente* es similar al primero, el hecho que representa la regla

```
SI plato-principal es ave y
   comida-es-pavo es rojo
ENTONCES mejor-color es tinto
```

queda como:

```
SI comida-es-pavo es sí
ENTONCES mejor-color es tinto
```

que se representa mediante el hecho f-9. Del mismo modo, este nuevo hecho provoca una activación de la regla *intentar-regla* para reemplazar la activación perdida cuando se retrajo el hecho que representa la regla.

Al permitir que se dispare la primera activación de *intentar-regla*, se produce la siguiente salida:

```
CLIPS> (run 1)↵
FIRE    1 intentar-regla: f-1,f-9,,
==> f-10    (objetivo (atributo comida-es-pavo))
CLIPS> (agenda)↵
0      pedir-valor-atributo: f-10,,
For a total of 1 activation.
CLIPS>
```

El hecho f-10 se afirma representando un objetivo para determinar el valor del atributo *comida-es-pavo*, pero como ninguna regla asigna el valor de este atributo, se activa la regla *pedir-valor-atributo* para determinarlo.

Al permitir que se dispare la regla *pedir-valor-atributo*, se produce la siguiente salida:

```
CLIPS> (run 1)↵
FIRE    1 pedir-valor-atributo: f-10,,
<== f-10    (objetivo (atributo comida-es-pavo))
¿Cuál es el valor de comida-es-pavo? sí↵
==> f-11    (atributo (nombre comida-es-pavo)
                      (valor sí))
CLIPS> (agenda)↵
100     regla-satisfecha: f-1,f-11,f-9
100     eliminar-regla-no-coincidente: f-1,f-11,f-8
For a total of 2 activations
CLIPS>
```

Un hecho *atributo* que representa el valor del atributo *comida-es-pavo* se afirma como resultado del disparo de esta regla y también se elimina el hecho *objetivo* que determina el valor de este atributo. El nuevo hecho *atributo* provoca dos activaciones, la primera es para la regla *eliminar-regla-no-coincidente*, y como a la primera condición del hecho f-8 no concuerda con el valor del nuevo atributo y la regla que representa el hecho ya no es aplicable, es

necesario eliminar este hecho; la segunda activación es para la regla *regla-satisfecha*, puesto que se satisface la condición restante del hecho f-8 a través del nuevo hecho *atributo*, es posible aplicar la consecuencia de este hecho.

Las demás reglas que habrán de dispararse finalizan el proceso de encadenamiento hacia atrás.

```
CLIPS> (run)↵
FIRE    1 regla-satisfecha: f-1,f-11,f-9
<== f-9     (regla (if comida-es-pavo es sí)
                   (then mejor-color es tinto))
==> f-12    (atributo (nombre mejor-color) (valor tinto))
FIRE    2 objetivo-satisfecho: f-1,f-12
<== f-1     (objetivo (atributo mejor-color))
CLIPS> (agenda)↵
CLIPS> (facts)↵
f-0     (hecho-inicial)
f-7     (atributo (nombre plato-principal) (valor ave))
f-8     (regla (if comida-es-pavo es no)
               (then mejor-color es blanco))
f-11    (atributo (nombre comida-es-pavo) (valor sí))
f-12    (atributo (nombre mejor-color) (valor tinto))
For a total of 5 facts.
CLIPS>
```

La regla *regla-satisfecha* se dispara para asignar el valor del atributo *mejor-color* como parte de la consecuencia de la regla representada por el hecho f-9. El hecho *atributo* afirmado por esta regla satisface el hecho *objetivo* restante de la lista de hechos, la regla *objetivo-satisfecho* se activa y luego se dispara para eliminar el hecho *objetivo* restante.

El comando *agenda* muestra que no hay otras reglas para disparar, el comando *facts* a los atributos a los que se les ha asignado valores y el hecho f-12 que al atributo objetivo inicial *mejor-color* se le asignó el valor *tinto*.

12.5 UN PROBLEMA DE MONITOREO

En esta sección se presenta el desarrollo paso a paso de un programa de CLIPS como la solución de un problema simple. Los pasos a desarrollar son: descripción inicial del problema, suposiciones hechas sobre la naturaleza del problema y definiciones iniciales para la representación del conocimiento del problema, seguidas de una construcción en aumento de las reglas para resolver el problema.

Planteamiento del problema

Este problema ejemplifica un sistema de monitoreo simple, que tiende a ser más adecuado para lenguajes basados en reglas de encadenamiento hacia atrás debido a su naturaleza controlada por datos. Por lo general, se lee un conjunto de valores de entrada o de sensor durante cada ciclo de programa y la inferencia se presenta hasta que se alcanzan todas las conclusiones posibles que pueden derivarse de los datos de entrada, lo que es compatible con un método controlado por datos, donde el razonamiento circula de los datos a las conclusiones.

En este ejemplo, el tipo de monitoreo que habrá de realizarse será de naturaleza genérica. Una planta de procesamiento hipotética contiene varios dispositivos que deben monitorearse, algunos dependerán de otros para su operación y cada uno tendrá uno o más sensores que facilitará las lecturas numéricas que indican su estado general. Cada sensor tendrá valores de línea de seguridad baja (LBS), línea de advertencia baja (LBR), línea de seguridad alta (LAS) y línea de advertencia alta (LAR). Una lectura entre las líneas de seguridad baja y alta se considerará normal, una lectura por arriba de la línea de seguridad alta pero debajo de la línea de advertencia alta o debajo de la línea de seguridad baja pero por arriba de la línea de advertencia baja se considerará aceptable, aunque es una indicación de que el dispositivo podría descomponerse dentro de poco. Una lectura por arriba de la línea de advertencia alta o por debajo de la línea de advertencia baja indicará un dispositivo descompuesto que deberá apagarse. Cualquier dispositivo dentro de la región de seguridad debe emitir avisos; además, debe apagarse cualquier dispositivo que se mantenga en una región de seguridad durante un tiempo excesivo. En la tabla 12.1 se resumen las acciones que se emprenderán para valores de sensor determinados.

Valor de sensor	Acción
Menor o igual a la línea de advertencia baja	Apagar dispositivo
Mayor que línea de advertencia baja y menor o igual que línea de seguridad baja	Emitir aviso o apagar dispositivo
Mayor que línea de seguridad baja y menor que línea de seguridad alta	Ninguna
Mayor o igual que línea de seguridad alta y menor que línea de advertencia alta	Emitir aviso o apagar dispositivo
Mayor o igual que línea de advertencia alta roja	Apagar dispositivo

Tabla 12.1 Acciones de valores de sensor

El programa de monitoreo debe tener la capacidad de leer los datos y evaluar las lecturas, emitir avisos y apagar dispositivos con base en las evaluaciones y tendencias del sensor. El modelo de salida para el programa de monitoreo podría tener el siguiente aspecto:

```
Ciclo 20 - sensor 4 en línea de seguridad alta
Ciclo 25 - sensor 4 en línea de advertencia alta
   Apagando dispositivo 4
Ciclo 32 - sensor 3 en línea de seguridad baja
Ciclo 38 - sensor 1 en línea de seguridad alta para 6 ciclos
   Apagando dispositivo 1
```

Para este ejemplo, en la figura 12.5 se muestran las conexiones entre los dispositivos y los sensores que se monitorean y en la tabla 12.2 se enumeran los atributos de cada uno de los sensores.

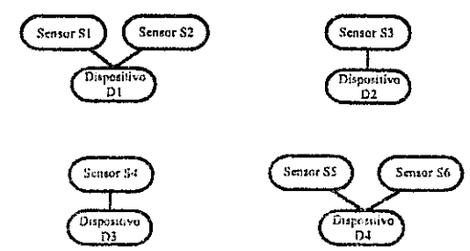


Figura 12.5 Dispositivos y sensores de un sistema monitoreado

Sensor	Línea de advertencia baja	Línea de seguridad baja	Línea de seguridad alta	Línea de advertencia alta
SI	60	70	120	130
S2	20	40	160	180
S3	60	70	120	130
S4	60	70	120	130
S5	65	70	120	125
S6	110	115	125	130

Tabla 12.2 Atributos de los sensores

Parte de la solución de este problema comprende el uso de la descripción general del problema para determinar los detalles específicos que deben resolverse antes de establecer toda la solución. Por lo general, este proceso incluiría la consulta reiterativa de especialistas con amplia información sobre el dominio del problema, pero no necesariamente aptos para especificar por completo la tarea que se proyecta circundar mediante un sistema experto. Los prototipos del problema se desarrollarían para señalar los detalles omitidos durante la especificación del problema, estos detalles se determinarían mediante consultas con especialistas y se desarrollaría otro prototipo que pudiera revelar otros detalles pasados por alto. A la postre, la especificación del problema circundaría por completo mediante esta filosofía de programación con desarrollo iterativo.

Muchos detalles de especificación de este problema permanecen sin respuesta: ¿cómo se representará la información sobre los sensores y los dispositivos? ¿Qué tan generales o específicos deben hacerse los hechos y las reglas? ¿Cómo se reunirán los datos del sensor? ¿Cuánto tiempo debe estar un sensor en una región de seguridad antes de que se apague el dispositivo correspondiente? y, ¿qué acciones se emprenden cuando el monitor detecta dispositivos descompuestos?

Los detalles necesarios para iniciar

Uno de los principales problemas que se encuentran en la construcción de un sistema experto es que casi siempre el problema está mal especificado y aun cuando la idea es emular a un especialista sólo éste tiene el conocimiento suficiente para especificar los detalles. Por lo general, se conoce el funcionamiento esperado del sistema, pero no la manera en que se generará este funcionamiento y un perito podría tener dificultades para comunicar los pasos exactos que se emprenden para llegar a una solución. Debido a las técnicas de desarrollo iterativo que el paradigma del sistema experto soporta de manera natural, es relativamente fácil construir sistemas expertos que resuelvan problemas mal especificados, sin embargo, esto no significa que los sistemas expertos pueden resolver problemas que nunca antes se han resuelto o problemas que no se entienden.

Son necesarias varias especificaciones del problema antes de que se pueda construir un sistema experto para este ejemplo de monitoreo; en primer lugar debe especificarse el funcionamiento esperado del sistema experto, incluyendo la información inicial y la información que el sistema produce pero no significa que esta especificación no pueda cambiarse. El desarrollo del programa puede indicar que su ámbito debe reducirse o quizá ampliarse, y esto afectaría las entradas iniciales y las salidas finales, deben hacerse algunas suposiciones y deben determinarse detalles iniciales sobre la manera en que el sistema experto debe realizar sus tareas, estos tipos de decisiones tampoco son permanentes. Es común que los expertos puedan mostrar cómo fueron resolviendo un problema, pero luego no logran formular con facilidad las reglas que usaron, pues para describirlas pueden pasar por alto detalles que resultan obvios para ellos o quizá olviden excepciones.

El problema de monitoreo requiere la toma de muchas decisiones iniciales, la primera es acerca del arranque. ¿La solución debe ajustarse de manera específica a los detalles exactos del problema o debe incorporar suficientes generalidades para permitir que se actualice o modifique con facilidad? Es posible escribir reglas específicas para cada uno de los dispositivos que se monitorean, o escribir una regla general para monitorear todos los dispositivos. En este problema, la escritura de reglas generales parece más apropiada porque los dispositivos y sensores individuales que habrán de vigilarse no tienen características únicas; este tipo de generalidad también debe permitir la fácil adición de más dispositivos y sensores.

Como también se pasan por alto detalles sobre el flujo de control para el sistema, en este problema se usará un ciclo de monitoreo simple que tendrá tres fases: durante la primera, los valores se leerán a partir de los sensores, durante la segunda se hará un análisis de los valores de sensor y en la tercera se emprenderá la acción apropiada.

Asimismo, se deben hacer suposiciones sobre la manera en que se recuperan los datos del sensor. ¿Se leerán directamente de los sensores? ¿Será necesario manejar la simulación de los datos del sensor? ¿Estarán disponibles los datos del sensor siempre que se soliciten? ¿Son confiables los datos del sensor o están sujetos a error? En situaciones normales de prototipo, se podría entrevistar a los especialistas y obtener esta información, pero para los propósitos de este problema, se harán suposiciones para completar los detalles.

Todas estas preguntas subrayan las lagunas que podrían encontrarse en un problema mal definido. Durante el desarrollo, debe mantenerse una lista de suposiciones, preguntas y posibles incongruencias relacionadas con la especificación del problema. En el desarrollo iterativo del programa, esta lista debe ser el punto focal de estudio, con especialistas que se aseguren que la especificación del problema coincida con las concepciones del especialista acerca de la manera de resolver el problema. Las siguientes suposiciones inician esta lista para el problema de monitoreo:

- Los datos de sensor son siempre confiables y están disponibles siempre que se solicitan.
- Los valores del sensor deben ser legibles directamente desde los sensores. También deben soportarse valores de sensor simulados.
- Los valores del sensor no se monitorearán para una máquina que se ha apagado.
- Se supone que se ejecutarán las acciones especificadas por el sistema de monitoreo (es decir, suponemos que tenemos un operador alerta o que el programa controla directamente los dispositivos).
- El problema se dividirá en tres fases: lectura de valores de sensor, análisis de valores de sensor y realización de acciones apropiadas, como el apagado de un dispositivo.
- Además de los detalles de especificación del problema, también deben decidirse los detalles de la especificación del problema, como la manera en que se representa la información disponible, el flujo de control y la prueba del sistema experto.

Definiciones de conocimiento

Nuevamente se comienza a resolver el problema determinando cómo representarse el conocimiento. Un buen inicio para este problema es codificar el conocimiento de la figura 12.5 y de la tabla 12.2; se empleará la siguiente deftemplate para describir cada uno de los dispositivos:

```
(defmodule MAIN (export ?TODO))
(deftemplate MAIN::dispositivo
   (slot nombre (type SYMBOL))
   (slot estado (valores-permitidos on off)))
```

donde la ranura *nombre* es el nombre del dispositivo y la ranura *estado* indica si el dispositivo está encendido o apagado. Usando la figura 12.5 y suponiendo que todos los dispositivos están encendidos inicialmente, describimos el estado inicial de los dispositivos usando el siguiente deffacts:

```
(deffacts MAIN::información-dispositivo
   (dispositivo (nombre D1) (estado on))
   (dispositivo (nombre D2) (estado on))
   (dispositivo (nombre D3) (estado on))
   (dispositivo (nombre D4) (estado on)))
```

En la figura 12.5 también se indica qué sensores se asocian con cuáles dispositivos. Se utilizará la siguiente deftemplate para representar esta relación:

```
(deftemplate MAIN::sensor
   (slot nombre (type SYMBOL))
   (slot dispositivo (type SYMBOL))
   (slot valor-primario (type SYMBOL NUMBER)
                        (símbolos-permitidos none)
                        (predeterminado none))
   (slot estado (valores-permitidos línea-advertencia-baja-
                                    línea-seguridad-baja
                                    normal
                                    línea advertencia-alta
                                    línea-seguridad-alta)
            (predeterminado normal)
   (slot línea-advertencia-baja- (type NUMBER))
   (slot línea-seguridad-baja (type NUMBER))
   (slot línea-seguridad-alta (type NUMBER))
   (slot línea-advertencia-alta (type NUMBER))
```

donde la ranura *nombre* es el nombre del sensor y la ranura *dispositivo* es el nombre del dispositivo al que está conectado; la ranura *valor-primario* contiene el valor de los datos leído directamente del sensor antes de haberse procesado; la ranura estado indica el estado actual del sensor (por ejemplo, normal, línea de seguridad baja, línea de advertencia alta, etc.) y las ranuras *línea-advertencia-baja, línea-seguridad-baja, valor-promedio-esperado, línea-seguridad-alta y línea-advertencia-alta-* se usan para contener la información descrita en la tabla 12.2. El siguiente deffacts puede emplearse para describir los sensores de la figura 12.5:

```
(deffacts MAIN::información-sensor
   (sensor (nombre S1) (dispositivo D1)
           (línea-advertencia-baja 60) (línea-seguridad-baja 70)
           (línea-seguridad-alta 120)
           (línea-advertencia-alta 130))
   (sensor (nombre S2) (dispositivo D1)
           (línea-advertencia-baja 20) (línea-seguridad-baja 40)
```



```
           (línea-seguridad-alta 160))
           (línea-advertencia-alta 180))
   (sensor (nombre S3) (dispositivo D2)
           (línea-advertencia-baja 60) (línea-seguridad-baja 70)
           (línea-seguridad-alta 120))
           (línea-advertencia-alta 130))
   (sensor (nombre S4) (dispositivo D3)
           (línea-advertencia-baja 60) (línea-seguridad-baja 70)
           (línea-seguridad-alta 120))
           (línea-advertencia-alta 130))
   (sensor (nombre S5) (dispositivo D4)
           (línea-advertencia-baja 65) (línea-seguridad-baja 70)
           (línea-seguridad-alta 120))
           (línea-alta 125))
   (sensor (nombre S6) (dispositivo D4)
           (línea-advertencia-baja 110) (línea-seguridad-baja 115)
           (línea-seguridad-alta 125))
           (línea-advertencia-alta 130))
```

Puesto que el sistema de monitoreo será cíclico, se necesitará un hecho que represente el ciclo actual, el primer ciclo puede iniciar en 1 y después aumentar 1 para cada nuevo ciclo. El formato del hecho ordenado para esta información será

```
(ciclo <número>)
```

donde <número> es el valor del ciclo actual. Además, debido a que los valores del sensor podrían leerse de más de un origen (por ejemplo, un origen para simulación y otro para la operación real), será útil tener un hecho que indique el origen de los datos de sensor. Este hecho puede representarse con el siguiente formato:

```
(origen-datos <origen>)
```

donde <origen> es un nombre que representa el origen de donde se leen los datos: los posibles orígenes serían los sensores, un simulador, un archivo de texto, un conjunto de hechos o una entrada de usuario.

Un deffacts que contiene esta información inicial es:

```
(deffacts MAIN::iniciar-ciclo
   (origen-datos usuario)
   (ciclo 0))
```

Observe que el usuario proporcionará los datos para los sensores. En este ejemplo será más conveniente la introducción de datos con el teclado que la lectura de datos desde un archivo.

Control de ejecución

El establecimiento de las suposiciones del problema indica que habrá tres fases específicas para el proceso de monitoreo: en la primera se leerán valores proporcionados por el usuario, simulados o reales de los sensores y en la segunda se asociarán condiciones de línea de seguridad y de advertencia para los valores del sensor y se determinará cualquier tendencia. Una vez que se hayan establecido tendencias, el sistema de monitoreo emitirá el aviso apropiado,

apagará el equipo de funcionamiento defectuoso y reiniciará el equipo que pueda restablecerse. Después de que se hayan emprendido las acciones apropiadas, iniciará un nuevo ciclo mediante la lectura de los nuevos valores de sensor.

El control de este sistema experto de monitoreo se manejará usando técnicas similares a las descritas en el capítulo 10, se crearán tres módulos separados: *ENTRADA, TENDENCIAS* y *AVISOS*, en cada ciclo se actualizará el valor del hecho *ciclo* y los módulos en funcionamiento se concentrarán en el orden apropiado. La siguiente regla realizará estas acciones:

```
(defrule MAIN::inicializar-siguiente-ciclo
   ?f <- (ciclo ?ciclo-actual)
   =>
   (reatract ?f)
   (assert (ciclo (+ ?ciclo-actual 1)))
   (focus ENTRADA TENDENCIAS AVISOS))
```

La regla *inicializar-siguiente-ciclo* se reactiva a sí misma, de modo que será necesario usar el comando *halt* del lado derecho de una regla para detener la ejecución.

Lectura de los valores de sensor primarios

El siguiente paso en la construcción del sistema de monitoreo es la lectura del sensor. El lugar lógico para leer sus valores es cuando el enfoque actual está en el módulo *ENTRADA*. Para propósitos de depuración y prueba, será conveniente tener la posibilidad de leer los valores de sensor desde varios orígenes. En esta sección se mostrará la manera de leer tales valores directamente de los sensores, de deffacts, de un archivo y del usuario. El hecho origen-datos ya descrito se emplea para indicar el origen de los datos para el sistema de monitoreo y los valores de sensor, leídos como datos "primarios", se almacenarán directamente en la ranura *valor-primario* de los hechos *sensor* y se procesará y analizará de este valor primario durante la fase de análisis.

Bajo condiciones de operación reales, los valores se extraerían directamente de los sensores, requiriendo funciones externas para hacer este trabajo. Supongamos que la función *obtener-valor-sensor* devolverá el valor de sensor actual, dado un argumento que representa el identificador del sensor para el que se desea el valor. (Para llamar a una función escrita en C, Ada, FORTRAN o algún otro lenguaje de programación, es necesario compilar el código de origen de CLIPS). La regla para la lectura directa de valores de sensor es la siguiente:

```
(defrule ENTRADA::Leer-valores-sensor-de-sensores
   (origen-datos sensores)
   ?s <- (sensor (nombre ?nombre)
                 (valor-primario ninguno)
                 (dispositivo ?dispositivo))
   (dispositivo (nombre ?dispositivo) (estado on))
   =>
   (modify ?s (valor-primario (obtener-valor-sensor
                                          ?nombre))))
```

El primer patrón de la regla (origen-datos sensores) limita el uso de esta regla a las ocasiones en que el origen de datos proviene de los sensores y los siguientes dos patrones forman la parte principal del lado izquierdo de la regla. La regla se activará para cada hecho de sensor de la lista de hechos que tiene un dispositivo correspondiente que está encendido. La restricción literal *ninguno* asegura que los valores de sensor sólo se leerán en los sensores que no

tienen ya un valor primario (de lo contrario, el ciclo de la regla se repetiría de manera infinita). La ranura *valor-primario* tendrá que restringirse al valor none luego del análisis del sensor y luego el lado derecho de la regla hace una llamada de función externa a la función *obtener-valor-sensor* y coloca el valor de devolución en la ranura *valor-primario* del hecho *sensor*.

Podría ser difícil usar esta regla en particular durante el prototipo real del sistema experto porque el programa tendría que comunicarse con los sensores y dispositivos reales, pero una técnica para leer los valores de sensor sería consultarlos directamente con el usuario, lo que permitiría la flexibilidad máxima de depuración ya que se consultaría directamente con el usuario cada valor de sensor:

```
(defrule ENTRADA::Obtener-valor-sensor-de-usuario
   (origen-datos usuario)
   ?s <- (sensor (nombre ?nombre)
                 (valor-primario ninguno)
                 (dispositivo ?dispositivo))
   (dispositivo (nombre ?dispositivo) (estado on))
   =>
   (printout t "Valor de entrada para sensor " ?nombre ": ")
   (bind ?valor-primario (read))
   (if (not (númerop ?valor-primario))
       (then (halt)
       else (modify ?s (valor-primario ?valor-primario))))
```

Todos los patrones de la regla *Obtener-valor-sensor-del-usuario* son idénticos a la regla *Leer-valores-sensor-de-sensores* con excepción del primero (origen-datos usuario), lo que limita las activaciones de la regla a situaciones en que el origen de los datos es el usuario. El lado derecho de la regla es similar, sólo que se consulta con el usuario el valor primario del sensor; observe también la expresión *if* que sigue a la unión de la variable *?valor-primario*, esta expresión detendrá el programa si se devuelve una ranura no numérica a través de la función *read*. Esta prueba es un método conveniente para que el usuario detenga la ejecución del programa cuando se estén introduciendo valores de sensor.

Para casos de pruebas o para evitar una gran cantidad de entradas por parte del usuario, sería más deseable leer datos desde un "script" en lugar de leerlos realmente de un sensor o preguntando al usuario por cada valor, una técnica para lograr esto almacenaría los datos en hechos que las reglas podrían introducir después. La siguiente deftemplate describe un hecho empleado para almacenar valores de datos:

```
(deftemplate ENTRADA::datos-hecho-para-sensor
   (slot nombre)
   (multislot datos))
```

donde la ranura *nombre* es el nombre del sensor asociado con los valores de datos y la ranura *datos* es la lista real de valores de datos para el sensor.

Utilizando esta deftemplate, un deffacts que contiene valores de prueba tendría el siguiente aspecto:

```
(deffacts ENTRADA::valores-datos-hecho-sensor
   (datos-hecho-para-sensor (nombre S1)
      (datos 100 100 110 110 115 120))
   (datos-hecho-para-sensor (nombre S2)
      (datos 110 120 125 130 130 135))
```

```
(datos-hecho-para-sensor (nombre S3)
   (datos 100 120 125 130 130 125))
(datos-hecho-para-sensor (nombre S4)
   (datos 120 120 120 125 130 135))
(datos-hecho-para-sensor (nombre S5)
   (datos 100 120 125 130 135 135))
(datos-hecho-para-sensor (nombre S6)
   (datos 115 120 125 135 130 135))
```

La siguiente regla podría usarse para introducir los valores de datos de los hechos *datos-hecho-para-sensor:*

```
(defrule ENTRADA::Leer-valores-sensor-de-hechos
   (origen-datos hechos)
   ?s <- (sensor (nombre ?nombre)
                 (valor primario ninguno))
   ?f <- (datos-hecho-para-sensor)
                 (nombre ?nombre)
                 (datos ?valor-primario $?resto))
   =>
   (modify ?s (valor-primario ?valor-primario))
   (modify ?f (datos ?resto)))
```

Esta regla es muy parecida a las dos reglas de entrada anteriores, pero se ha omitido el patrón (dispositivo (nombre ?dispositivo) (estado on)); esto permite que el *enésimo* valor de un hecho *datos-hecho-para-sensor* se corresponda directamente con el valor para un sensor en el *enésimo* ciclo de monitoreo. También es digno de comentar el último patrón en el lado izquierdo: el hecho *datos-hecho-para-sensor* contiene los valores de datos que usa el sensor con el nombre limitado a la variable *?nombre;* el primer valor de datos del hecho se limita a la variable *?valor-primario* y los demás valores se limitan a la variable *?resto;* entonces, el primer valor de datos se almacena en la ranura *valor-primario* del sensor correspondiente y el hecho *datos-hecho-para-sensor* se modifica luego con el valor de datos recién-usados pasado por alto. Resta un problema en el que la regla no se disparará si el hecho *datos-hecho-para-sensor* no tiene valores de datos. Ahora supongamos que si no hay valores de datos dejados por un sensor, el sistema debe detenerse; la siguiente regla maneja esta situación:

```
(defrule ENTRADA::No-dejar-valores-datos-hecho
   (origen-datos hechos)
   (sensor (nombre ?nombre)
           (valor-primario ninguno))
   (datos-hecho-para-sensor (nombre ?nombre) (datos))
   =>
   (printout t "No hay datos de hecho para sensor "
               ?nombre crlf)
   (printout t "Detener sistema de monitoreo" crlf)
   (halt))
```


La técnica de entrada final que habrá de explicarse implica la lectura de la información desde un archivo de datos, lo que es más complicado que los ejemplos anteriores porque deben manejarse varios aspectos. El archivo de datos debe abrirse inicialmente y los datos leerse del archivo de manera secuencial, mientras que las anteriores técnicas de entrada no dependían de la lectura secuencial de datos, sino que podían introducir en cualquier orden los nuevos valores de datos para los sensores. Tampoco es necesario saber cuántos valores de sensor tienen que leerse, y es deseable evitar la codificación compleja de dicha información en las reglas. Para agregar más complejidad, se supondrá que los valores de datos de sensor quizá se hayan dejado sin especificar, en cuyo caso asumirán los valores de datos primarios del sensor durante el ciclo anterior.

Lo primero que debe manejarse es la apertura inicial del archivo de datos. Para indicar que se ha abierto se utilizará un hecho para que actúe como indicador y el hecho (archivo-datos-abierto) indicará que se ha abierto un archivo de datos utilizando el nombre lógico *archivo-datos;* la ausencia de este hecho indicará la necesidad de abrirlo. La siguiente regla abrirá el archivo de datos, si aún no se ha abierto:

```
(defrule INPUT::Abrir-archivo-con-valores-sensor
   (origen-datos archivo)
   (not (apertura-archivo-datos))
   =>
   (bind ?indicador archivo-cerrado)
   (while (eq ?indicador archivo cerrado)
     (printout t
              "¿Cuál es el nombre del archivo de datos? ")
     (bind ?nombre-archivo (readline))
     (if (open ?nombre-archivo archivo-datos "r")
        then (bind ?indicador true)))
   (assert (apertura-archivo-datos)))
```

Esta regla tiene dos condiciones simples: el origen de los datos debe ser un archivo y éste aún no debe estar abierto; si se cumplen estas condiciones, entonces el lado derecho de la regla abrirá el archivo de datos cuyo nombre lo proporciona el operador del sistema de monitoreo. El ciclo *while,* junto con la variable *?indicador,* establece un método conveniente para asegurarse que el archivo que proporcionó el operador se abrió con éxito. Al operador se le preguntará repetidamente el nombre de un archivo de datos de origen, siempre y cuando la función *open* no haya abierto con éxito el archivo proporcionado. Una vez que se ha abierto un archivo, se afirma el indicador del hecho *archivo-datos-abierto.*

La siguiente decisión tienen que ver con el formato del archivo para almacenar los datos de sensor. Como es conveniente tener la posibilidad de especificar sólo aquellos valores de sensor que han cambiado desde el último ciclo, no se sabe cuántos valores de sensor se leerán para un ciclo determinado y tampoco puede determinarse de antemano el orden en que se leerán los valores de sensor; estas suposiciones establecen un formato de datos donde se mantendrá el nombre del sensor junto con el valor de sensor primario que habrá de leerse. Debido a que no se conocer el número de valores de datos que se leerán, los valores de fin-de-datos para un ciclo se indicarán por medio de la palabra clave *fin-de-ciclo* y el formato de datos tendrá el siguiente aspecto:

```
S1 100
S2 110
S3 100
S4 120
S5 110
S6 115
fin-de-ciclo
S2 120
S3 120
S5 120
S6 120
fin-de-ciclo
S1 110
S2 125
S3 125
S5 125
S6 125
fin-de-ciclo
...
```

Primero se necesita una regla que permita leer todos los valores primarios del archivo de datos para el ciclo actual. La regla que realiza esta operación es:

```
(defrule ENTRADA::Leer-valores-sensor-de-archivo
   (origen-datos archivo)
   (apertura-archivo-datos)
   (ciclo ?tiempo)
   =>
   (bind ?nombre (read archivo-datos))
   (if (eq ?nombre EOF) then (halt))
   (while (and (neq ?nombre fin-de-ciclo)
               (neq ?nombre EOF))
       (bind ?valor-primario (read archivo-datos))
       (if (eq ?valor-primario EOF) then (halt))
       (assert (valor-sensor-primario ?nombre ?valor-
                                                 primario))
       (bind ?nombre (read archivo-datos))
       (if (eq ?nombre EOF) then (halt))))
```

Esta regla lee todos los valores de datos del archivo hasta que se llega a la palabra clave *fin-de-ciclo;* si se llega al final del archivo, entonces se detiene el sistema de monitoreo, los valo-res de datos leídos se afirman como hechos con el nombre de relación *valor-sensor-primario* y las dos ranuras restantes del hecho especifican el nombre del sensor para el que se aplica el valor de datos y el valor de sensor primario.

La siguiente regla eliminará el hecho valor de datos primario para cualquier sensor cuyo dispositivo se haya apagado, de modo que el razonamiento sólo se limita a aquellos sensores cuyos dispositivos están en:


```
(defrule ENTRADA::Eliminar-valores-para-sensores-inactivos
   (origen-datos archivo)
   (apertura-archivo-datos)
   (ciclo ?tiempo)
   (sensor (nombre ?nombre) (dispositivo ?dispositivo))
   (dispositivo (nombre ?dispositivo) (estado off))
   ?datos <- (valor-sensor-primario ?nombre ?valor-primario)
   =>
   (reatract ?datos))
```

La siguiente regla almacena la lectura del valor primario del sensor obtenida del archivo en la ranura *valor-primario* de los sensores. Durante esta fase de ejecución, no se emprenden acciones en los sensores que no tienen cambios en su valor de datos primario.

```
(defrule ENTRADA::Transferir-valores-sensor-a-sensores
   (origen-datos archivo)
   ?s <- (sensor (nombre ?nombre)
                 (valor-primario ninguno)
                 (dispositivo ?dispositivo))
   (dispositivo (nombre ?dispositivo) (estado on))
   ?f <- (valor-sensor-primario ?nombre ?valor-primario)
   =>
   (modify ?s (valor-primario ?valor-primario))
   (retract ?f))
```

Detección de una tendencia

La siguiente fase de actividad, la fase de *tendencias,* determina el estado actual de los sensores y calcula las tendencias que podrían desarrollarse. El estado actual del sensor (normal, línea de seguridad baja o alta o línea de advertencia baja o alta) debe determinarse a partir del valor de sensor primario afirmado durante la fase de *entrada* y se almacena en la ranura *estado* de la deftemplate *sensor.* La siguiente regla determina si un sensor está en el estado normal:

```
(defrule TENDENCIAS::Estado-normal
   ?s <- (sensor (valor-primario ?valor-primario&~ninguno)
                 (línea-seguridad-baja ?lbs)
                 (línea-seguridad-alta ?hgl))
   (test) (and (> ?valor-primario ?lbs)
               (< ?valor-primario ?hgl)))
   =>
   (modify ?s (estado normal) (valor-primario ninguno)))
```

El primer patrón, después del elemento condicional test, busca cualquier sensor que se encuentre en el estado normal. La ranura *valor-primario* del patron *sensor* se compara con el valor ninguno para comprobar que no se compare un símbolo con valores numéricos en el elemento condicional test de la regla. El rango necesario para probar si un sensor se encuentra en estado normal se ubica entre los valores de las líneas de seguridad baja y alta. El elemento condicional del test restringe la regla a las situaciones donde el valor de sensor primario está en el rango de estado normal. La única acción de esta regla es afirmar el estado derivado del sensor. A la ranura *va-lor-primario* también se le asigna el valor *ninguno,* de modo que se leerá en la siguiente fase de *entrada.*

Se necesitan cuatro reglas más para analizar los cuatro posibles estados restantes de un sensor, cada una de ellas será similar a la regla *Estado-normal*, con excepción de los valores recuperados del hecho *sensor* y el elemento condicional test usado para determinar el estado actual. Las cinco reglas de estado podrían escribirse como una regla simple con una expresión *if* en su lado derecho para determinar el estado apropiado del sensor, pero este tipo de codificación anula el propósito de un sistema controlado por datos. Además, con una regla simple y poco manejable sería mucho más difícil realizar otras modificaciones que comprenden acciones, condiciones, o ambas, de un subconjunto de los estados posibles. Las cuatro reglas de análisis de estado restantes son las siguientes:

```
(defrule TENDENCIAS::Estado-línea-seguridad-alta
   ?s <- (sensor (valor-primario ?valor-primario&~ninguno)
                 (línea-alta-seguridad ?hgl)
                 (línea-advertencia-alta ?hrl))
   (test (and (>= ?valor-primario ?hgl)
              (< ?valor-primario ?hrl)))
   =>
   (modify ?s (estado línea-seguridad-alta)
              (valor-primario ninguno)))

(defrule TENDENCIAS::Estado-línea-advertencia-alta)
   ?s <- (sensor (valor-primario ?valor-primario&~ninguno)
                 (línea-advertencia-alta ?hrl))
   (test (>= ?valor-primario ?hrl))
   =>
   (modify ?s (estado línea-advertencia-alta)
              (valor-primario ninguno)))

(defrule TENDENCIAS::Estado-línea-seguridad-baja
   ?s <- (sensor (valor-primario ?valor-primario&~ninguno)
                 (línea-seguridad-baja ?lgs)
                 (línea-advertencia-baja ?lgr))
   (test (and (> ?valor-primario ?lbr)
        (<= ?valor-primario ?lbs)))
   =>
   (modify ?s (estado línea-seguridad-baja)
              (valor-primario ninguno)))

(defrule TENDENCIAS::Estado-línea de advertencia-baja
   ?s <- (sensor (valor-primario ?valor primario&~ninguno)
                 (línea-advertencia-baja ?lbr))
   (test (<= ?valor-primario ?lbr))
   =>
   (modify ?s (estado línea-advertencia-baja)
              (valor-primario ninguno)))
```

Las cinco reglas anteriores determinan el estado de un sensor para el ciclo actual. Puesto que la detección de tendencias en los sensores es uno de los objetivos del módulo *TENDENCIAS*, será necesario conservar información sobre el último estado de un sensor, por lo que se

utilizará la siguiente deftemplate para almacenar esta información; y como los módulos *TENDENCIAS* y *AVISOS* también la emplearán, se colocará en el módulo PRINCIPAL.

```
(deftemplate PRINCIPAL::tendencia-sensor
   (slot nombre)
   (slot estado (predeterminado normal))
   (slot inicio (predeterminado 0))
   (slot final (predeterminado 0))
   (slot duración-interrupción (predeterminado 3)))
```

La ranura *nombre* es el nombre del sensor, la ranura *estado* corresponde al estado más actual del sensor, la ranura *inicio* es el primer ciclo durante el que el sensor estuvo en su estado actual, la ranura *final* es el ciclo actual y la ranura *duración-interrupción* es el tiempo que un sensor debe estar en una región de línea de seguridad antes de apagar el dispositivo asociado.

Las reglas que actualizan la información de tendencia dependerán de un hecho *tendencia-sensor* existente en la lista de hechos para cada sensor. Por ello, las tendencias del sensor inicial se definirán en una construcción de deffacts.

```
(deffacts PRINCIPAL::tendencias-iniciales
   (tendencia-sensor (nombre S1) (duración-interrupción 3))
   (tendencia-sensor (nombre S2) (duración-interrupción 5))
   (tendencia-sensor (nombre S3) (duración-interrupción 4))
   (tendencia-sensor (nombre S4) (duración-interrupción 4))
   (tendencia-sensor (nombre S5) (duración-interrupción 4))
   (tendencia-sensor (nombre S6)  (duración-interrupción 2)))
```

Con esta información, pueden definirse dos reglas para monitorear las tendencias de un sensor; una monitoreará una tendencia que *no* ha cambiado desde el último ciclo y la otra una tendencia que *ha* cambiado desde el último ciclo:

```
(defrule TENDENCIAS::Estado-no-ha-cambiado
   (ciclo ?tiempo)
   ?tendencia <- (tendencia-sensor (nombre ?sensor)
                                   (estado ?estado)
                                   (final ?ciclo-
                                          final&~?tiempo))

   (sensor (nombre ?sensor) (estado ?estado)
           (valor-primario ninguno))

   =>
   (modify ?tendencia (final ?tiempo)))

(defrule TENDENCIAS::Estado-ha-cambiado
   (ciclo ?tiempo)
   ?tendencia <- (tendencia-sensor (nombre ?sensor)
                                   (estado ?estado)
                                   (final ?ciclo-
                                          final&~?tiempo))

   (sensor (nombre ?sensor)
           (estado ?nuevo-estado&~?estado)
           (valor-primario ninguno))
```

```
=>
(modify ?tendencia (inicio ?tiempo)
                   (final ?tiempo)
                   (estado ?nuevo-estado)))
```

El primer patrón de ambas reglas establece el ciclo; el siguiente patrón encuentra el hecho *tendencia-sensor* para el ciclo previo, y la restricción en la ranura *final* asegura que las reglas no repitan el ciclo de manera infinita. Para la regla *Estado-no-ha-cambiado,* el siguiente patrón revisa que el estado del ciclo anterior es el mismo que el estado para el ciclo actual. La restricción ?nuevo estado&~?estado de la regla *Estado-ha-cambiado* realiza exactamente la prueba opuesta, asegurándose de que el estado ha cambiado desde el último ciclo. La revisión de la ranura *valor-primario* es *none* evita determinar la tendencia antes de que se determine el estado presente del sensor. En ambas reglas se actualiza el tiempo del ciclo final, si el estado ha cambiado, entonces el valor de estado y el tiempo de ciclo inicial también deben actualizarse.

Emisión de avisos

La fase final del ciclo es la fase de *avisos,* que debe manejar tres tipos de acciones: los sensores que han entrado a una región de línea de advertencia tendrán apagados sus dispositivos asociados, los sensores que han permanecido en la región de línea de seguridad por un número de ciclos especificado tendrán apagados sus dispositivos asociados y los sensores que están en la región de línea de seguridad y no tenían apagados sus dispositivos asociados recibirán un aviso.

La siguiente regla apaga los sensores que han entrado a una región de línea de advertencia:

```
(defrule AVISOS :: Interrupir-en-región-advertencia
   (ciclo ?tiempo)
   (tendencia-sensor)
      (nombre ?sensor)
      (estado ?estado&línea-advertencia-alta | línea-
                                  advertencia-baja))
   (sensor (nombre (?sensor) (dispositivo ?dispositivo))
   ?encendido <- (dispositivo (nombre ?dispositivo)
(estado on))
   =>
   (printout t "Ciclo " ?tiempo " - ")
   (printout t "Sensor " ?sensor " activo " ?estado crlf)
   (printout t " Apagar dispositivo " ?dispositivo
             crlf)
   (modify ?encendido (estado off)))
```

El estado de un sensor se verifica con el hecho *tendencia-sensor,* si el sensor está en la región de línea de advertencia, se apaga el dispositivo asociado. Como los dispositivos se apagan de inmediato cuando alguno de sus sensores entra en la región de línea de advertencia, cumple pequeños propósitos para verificar cuánto tiempo ha estado un sensor en este estado.

La regla *Interrumpir-en-región-seguridad* es parecida a la regla anterior, con la diferencia de que el sensor debe haber estado en la región de línea de seguridad durante un periodo (especificado por la ranura duración-interrupción del hecho tendencia-sensor) antes de apagarse. El tiempo que un sensor ha estado en un estado particular puede determinarse al restar la ranura *inicio* de la ranura *final* del hecho *tendencia-sensor.*


```
(defrule AVISOS::Interrupir-en-región-seguridad
   (ciclo ?tiempo)
   (tendencia-sensor
       (nombre ?sensor)
       (estado ?estado&línea-seguridad-alta|
                     línea-seguridad-baja)
       (duración-interrupción ?longitud)
       (inicio ?inicio) (final ?final))
   (test (>= (+ (- ?final ?inicio) 1) ?longitud))
   (sensor (nombre ?sensor) (dispositivo ?dispositivo))
   ?encendido <- (dispositivo (nombre ?dispositivo)
                                  (estado on))

   =>
   (printout t "Ciclo " ?tiempo " - ")
   (printout t "Sensor " ?sensor " activo " ?estado " ")
   (printout t "por " ?longitud " ciclos "
             crlf)
   (printout t "    Interrumpir dispositivo " ?dispositivo
             crlf)
   (modify ?encendido (estado off)))
```

La adición de la ranura *duración-interrupción* en el patrón *tendencia-sensor* y el elemento condicional test asociado es la única diferencia importante entre esta regla y la regla *Interrumpir-en-región-advertencia,* esos dos patrones determinan si un patrón ha estado suficiente tiempo en una región para apagar el dispositivo asociado con el sensor.

La regla final para el sistema de monitoreo emitirá un aviso para sensores que están en una región de seguridad pero que no han estado allá el tiempo suficiente para garantizar el apagado de sus dispositivos asociados:

```
(defrule AVISOS::Sensor-en-región-seguridad
   (ciclo ?tiempo)
   (tendencia-sensor
       (nombre ?sensor)
       (estado ?estado&línea-seguridad-alta|
                     línea-seguridad-baja)
       (duración-interrupción ?longitud)
       (inicio ?inicio) (final ?final))
   (test (< (+ (- ?final ?inicio) 1) ?longitud))
   =>
   (printout t "Ciclo " ?tiempo " - ")
   (printout t "Sensor " ?sensor " activo "
             ?estado crlf))
```

Esta regla actúa como complemento de la regla *Interrumpir-en región-seguridad.* El elemento condicional test se ha modificado para verificar que el sensor haya estado en la región de línea de seguridad menos tiempo que el número de ciclos que se necesitan para apagar su dispositivo asociado. Puesto que éste no necesita apagarse, no se incluyen patrones para determinar el dispositivo asociado.

Esta regla final completa la base de un sistema de monitoreo simple, sería necesario establecer reglas adicionales para cubrir situaciones específicas que deben monitorearse o proporcionar un modelo genérico para el manejo de situaciones complejas de monitoreo y representación de relaciones sensor/dispositivo.

12.6 RESUMEN

En este capítulo se mostró una técnica para representar factores de certidumbre con el estilo MYCIN en CLIPS. Los hechos se usan para representar ternas objeto-atributo-valor. Una ranura adicional en cada hecho representa el factor de certidumbre del hecho. Las reglas se usan para calcular los valores de certidumbre para hechos recién afirmados en el lado derecho de una regla mediante el uso de valores de certidumbre limitados en el lado izquierdo de la regla, luego se utiliza una regla para combinar dos ocurrencias de una terna OAV en una ocurrencia simple con un nuevo valor de certidumbre calculado a partir de los factores de certidumbre del par original.

También es posible representar árboles de decisión usando el paradigma de encadenamiento hacia adelante de CLIPS. Hay varios algoritmos para recorrer árboles de decisión, incluyendo árboles de decisión binarios, de ramas múltiples y de ramas múltiples de aprendizaje; el algoritmo para un árbol de decisión de ramas múltiples que aprende está incluido en CLIPS.

CLIPS también puede emular una estrategia de inferencia de encadenamiento hacia atrás. Un mecanismo de inferencia con encadenamiento hacia atrás se integra utilizando reglas de CLIPS y se basa en parte en un algoritmo para realizar encadenamiento hacia atrás de un lenguaje de procedimientos. Las reglas de encadenamiento hacia atrás se representan como hechos y se guían por las reglas de inferencia de encadenamiento hacia atrás de CLIPS. Un registro paso a paso de una sesión de muestra de encadenamiento hacia atrás demuestra la manera en que se representa este paradigma en CLIPS.

El ejemplo final de este capítulo es el sistema experto de monitoreo simple. Para iniciar el proceso de escritura del sistema experto, se usa una instrucción inicial del problema de monitoreo, se hacen suposiciones y se agregan detalles adicionales al sistema de monitoreo, junto con más reglas. La ejecución del sistema de monitoreo se divide en tres fases: la fase de *entrada* que reúne valores de datos primarios para los sensores (se demostraron varios métodos diferentes para proporcionar valores de datos); la fase *tendencias* analiza los valores de sensor primarios y verifica el desarrollo de tendencias y, por último, la fase *avisos* emite avisos y realiza acciones apropiadas con base en el análisis de la fase *tendencias*.

PROBLEMAS

12.1 Escriba reglas de CLIPS para combinar factores de certidumbre de MYCIN, como se muestra en la sección 12.2 para los dos casos siguientes

$$\text{Nueva certidumbre} = (FC_1 + FC_2) + (FC_1 * FC_2)$$

$$\text{si } (FC_1 \leq 0 \text{ y } FC_2 \leq 0$$

$$\text{Nueva certidumbre} = \frac{FC_1 + FC_2}{1 - \text{mín}\ \{\ |CF_1|\ ,\ |CF_2|\ \}}$$

$$\text{si } -1 < FC_1 * FC_2 < 0$$

12.2 Muestre cómo podría incorporarse la probabilidad clásica en CLIPS usando los tipos de técnicas demostradas en la sección 12.2. Enumere las posibles ventajas y desventajas del uso de probabilidad clásica en reglas.

12.3 El algoritmo *Resolver_árbol_y_aprender* descrito en la sección 12.3 empleando un lenguaje de procedimientos como LISP, C o PASCAL. Pruébelo con el ejemplo de identificación de animales.

12.4 *Implante* los algoritmos *Resolver_objetivo* e *Intentar_regla* de la sección 12.4 usando un lenguaje de procedimientos como LISP, C o PASCAL. Pruébelos empleando el ejemplo de la selección de vino.

12.5 Modifique la regla *ENTRADA::Obtener-valor-sensor-de usuario* de modo que sólo permita una entrada numérica del usuario. La regla debe causar una detención si se introduce la palabra *halt* durante la entrada.

12.6 Modifique la regla *ENTRADA::Obtener-valor-sensor-de-usuario* y agregue reglas, si es necesario para permitir al usuario emplear un enter que indique que se debe conservar el valor previo de cada sensor. Por ejemplo, si los valores anteriores para los sensores 1 y 2 fueron 100 y 120, entonces las siguientes respuestas

```
¿Cuál es el valor para el sensor 1?↵
¿Cuál es el valor para el sensor 2? 130↵
```

deben asignar el valor primario de 100 al sensor 1 y de 130 al sensor 2. ¿Cuáles suposiciones deben hacerse y cuáles acciones deben emprenderse si se desconoce el valor previo del sensor?

12.7 Modifique el programa de modo que si un dispositivo se ha apagado continúe monitoreando sus sensores. Si todos los sensores para el dispositivo regresan a un estado normal, entonces el dispositivo debe volverse a encender.

12.8 La regla *Sensor-en-región-seguridad* emitirá un aviso para un sensor aunque haya otra regla en la agenda, después de lo cual apagaría su dispositivo asociado. ¿Cuál cambio debe hacerse en esta regla para evitar que emita un aviso, si otra regla del módulo *AVISOS* apagara su dispositivo asociado?

12.9 Agregue reglas para imprimir un mensaje en la fase *avisos* para indicar que un sensor ha estado en estado normal durante cuando menos *n* ciclos, donde el número *n* se especifica en un hecho. Imprima el mensaje únicamente cada *n* ciclo.

12.10 ¿Cómo podrían modificarse las reglas de encadenamiento hacia atrás de la sección 12.4 para permitir el encadenamiento hacia adelante?

12.11 Modifique el programa desarrollado en el problema 11.8 de la página 506, de modo que los hechos que representan los arbustos se carguen desde un archivo usando la función load-facts. Después de proporcionar las explicaciones, se debe pedir al usuario que determine si se desea otra selección de arbusto, y, si es así, el programa debe volver a ejecutarse.

12.12 Modifique el programa desarrollado en el problema 11.11 de la página 507, de modo que pueda registrar varios estudiantes. La entrada al programa debe leerse desde un archivo. Usted tiene la libertad de determinar el formato de los datos de entrada; sin embargo, deben contener cuando menos el nombre de cada estudiante, los cursos programados y las preferencias de los estudiantes por instructores y periodos. La salida el programa debe escribirse para un archivo. El archivo de salida debe contener el nombre de cada estudiante seguido de los cursos programados para él.

12.13 Modifique el programa desarrollado para el problema 11.10 de la página 506 para mostrar menús reconfigurables dinámicamente. Por ejemplo, la selección del elemento de menú 1 del submenú 1 provoca el despliegue de dos elementos de menú en el submenú 2, pero la selección del elemento de menú 2 del submenú 1 ocasiona el despliegue de cuatro elementos de menú en el submenú 2.

12.14 Usando el programa desarrollado para el problema 12.13, modifique el programa desarrollado para el problema 11-14 de la página 508, de modo que sólo los dispositivos que no se han seleccionado se muestren en el submenú "Agregar dispositivo" y sólo los dispositivos que se han seleccionado se enumeren en el submenú "Eliminar dispositivo". En el menú principal, la opción de menú "Agregar dispositivo" sólo debe aparecer si hay dispositivos no seleccionados, y la opción de menú "Eliminar dispositivo" sólo debe aparecer si hay dispositivos seleccionados.

12.15 Usando el programa desarrollado para el problema 11.10 de la página 506, modifique el programa que se desarrolló para el problema 10.17 de la página 476 para usar una interfaz controlada por menús. Se deben proporcionar al usuario opciones de menú separadas para especificar el color, la dureza y la densidad de la gema que se está identificando. La especificación del color debe hacerse usando un submenú que contenga los colores válidos. Debe proporcionarse una opción de menú que enumere las gemas que satisfacen el criterio especificado en la actualidad. Por ejemplo, si el usuario sólo ha especificado que la gema es negra, la selección de esta opción de menú sólo enumerará las gemas que pueden ser negras. Deben proporcionarse otras dos opciones de menú, una que enumere los valores actuales especificados para el criterio y otra que reinicie todos los criterios para un estado no especificado.

BIBLIOGRAFÍA

(Firebaug 88). Morris W. Firebaug, *Artificial Intelligence: A Knowlegde-Based Approach*, Boyd & Fraser Publishing, p. 309, 1988.

APÉNDICE A

Algunas equivalencias útiles

$\sim\sim p \equiv p$

$p \rightarrow q \equiv \sim p \vee q \equiv \sim q \rightarrow \sim p$

$\sim(p \wedge q) \equiv \sim p \vee \sim q$
$\sim(p \vee q) \equiv \sim p \wedge \sim q$

$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$
$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$

$(p \wedge q) \wedge r) \equiv p \wedge (q \wedge r)$
$(p \vee q) \vee r) \equiv p \vee (q \vee r)$

$p \wedge q \equiv q \wedge p$
$p \vee q \equiv q \vee p$

$\sim(\forall x)\ P(x) \equiv (\exists x)\ \sim P(x)$
$\sim(\exists x)\ P(x) \equiv (\forall x)\ \sim P(x)$

$(\forall x)\ P(x) \wedge (\forall x)\ Q(x) \equiv (\forall x)\ (P(x) \wedge Q(x))$
$(\exists x)\ P(x) \vee (\exists x)\ Q(x) \equiv (\exists x)\ (P(x) \vee Q(x))$

Nota: el $\forall$ no se distribuye sobre un $\vee$, y $\exists$ no se distribuye sobre $\wedge$, de modo que se necesita una nueva variable modelo, z, en las siguientes dos equivalencias.

$$(\forall x)\ P(x) \vee (\forall x)\ Q(x) \equiv (\forall x)\ (P(x) \vee (\forall z)\ Q(z)$$
$$\equiv (\forall x)\ (\forall z)\ (P(x) \vee Q(z))$$

$$(\exists x)\ P(x) \wedge (\exists x)\ Q(x) \equiv (\exists x)\ (P(x) \wedge (\exists z)\ Q(z)$$
$$\equiv (\exists x)\ (\exists z)\ (P(x) \wedge Q(z))$$

APÉNDICE B

Algunos cuantificadores elementales y sus significados

Fórmula	Significado
$(\forall x)\ (P(x) \rightarrow Q(x))$	Para toda *x*, todas las *P* son *Q*
$(\forall x)\ (P(x) \rightarrow \sim Q(x))$	Para toda *x*, las *P* no son *Q*
$(\exists x)\ (P(x) \wedge Q(x))$	Para algunas *x*, *x* son *P* y *Q*
$(\exists x)\ (P(x) \wedge \sim Q(x))$	Para algunas *x*, *x* no son *P* y *Q*
$(\forall x)\ (P(x)$	Para todas las *x*, *x* es *P*
$(\exists x)\ (P(x)$	Algunas *x* son *P* (o hay *P*)
$\sim(\forall x)\ (P(x)$	No todas las *x* son *P* (o algunas *x* son *P*)
$(\forall x)\ \sim(P(x)$	Todas las *x* no son *P*
$(\forall x)\ (\exists y)\ P(x,y)$	Para todas las *x*, hay una *y* tal que *P*
$(\exists x)\ \sim P(x)$	Algunas *x* no son *P*

APÉNDICE C

Algunas propiedades de conjunto

Comunicatividad	$A \cup B = B \cup A$ $A \cap B = B \cap A$
Asociatividad	$A \cup (B \cup C) = (A \cup B) \cup C$ $A \cap (B \cap C) = (A \cap B) \cap C$
Idempotencia	$A \cup A = A$ $A \cap A = A$
Distributividad	$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
Ley del medio excluido	$A \cup A' = U$
Ley de contradicción	$A \cap A' = \varnothing$
Identidad	$A \cup \varnothing = A$ $A \cap U = A$
Absorción	$A \cup (A \cap B) = A$ $A \cap (A \cup B) = A$
Leyes de De Morgan	$(A \cap B)' = A' \cup B'$ $(A \cup B' = A' \cap B'$
Involución	$(A')' = A$
Equivalencia	$(A' \cup B) \cap (A \cup B') = (A' \cap B') \cup (A \cap B)$
Diferencia simétrica	$(A' \cap B) \cup (A \cap B') = (A' \cup B') \cap (A \cup B)$

APÉNDICE D

Información de soporte de CLIPS

En la página principal de CLIPS, en http://www.jsc.nasa.gov/~clips/CLIPS.html o como alternativa en http://www.ghg.net/clips/CLIPS.html, puede encontrarse información sobre solución de fallas, actualizaciones y otros temas relacionados. Las preguntas sobre CLIPS pueden enviarse mediante correo electrónico a clips@ghg.net. Los usuarios de Usenet pueden encontrar información y enviar preguntas sobre CLIPS al grupo de noticias comp.ai.shells.

También está disponible una sala de conferencia electrónica para usuarios de CLIPS. Los suscriptores a esta sala pueden enviar preguntas, observaciones, respuestas, editoriales, etc., en forma de correo electrónico a la conferencia. Todos los suscriptores tendrán una copia de estos mensajes en sus respectivas direcciones de correo electrónico. Para suscribirse, envíe un mensaje de una sola línea a listserv@cosmic.uga.edu que diga SUBSCRIBE CLIP-LIST (el guión es obligatorio) SU NOMBRE. El campo de asunto se ignora, pero la dirección encontrada en los campos "Responder:", "Responder a:" o "De:" se introducirá en la lista de distribución. Durante la suscripción recibirá un mensaje de correo que le indica cómo participar en la conferencia desde ese momento. Guarde este mensaje de correo porque puede necesitar las instrucciones más adelante, si desea eliminar su subscripción del servidor de lista.

APÉNDICE E

Resumen de comandos y funciones de CLIPS

Funciones de predicado

`(and <expresión>+)`
Devuelve TRUE si cada uno de sus argumentos se evalúa a TRUE, de lo contrario devuelve FALSE.

`(eq <expresión> <expresión>+)`
Devuelve TRUE si su primer argumento es igual en tipo y valor a todos sus argumentos subsecuentes, de lo contrario devuelve FALSE.

`(evenp <expresión>)`
Devuelve TRUE si <expresión> es un entero par, de lo contrario devuelve FALSE.

`(floatp <expresión>)`
Devuelve TRUE si <expresión> es un punto flotante, de lo contrario devuelve FALSE.

`(integerp <expresión>)`
Devuelve TRUE si <expresión> es un entero, de lo contrario devuelve FALSE.

`(lexemep <expresión>)`
Devuelve TRUE si <expresión> es una cadena o un símbolo, de lo contrario devuelve FALSE.

`(multifieldp <expresión>)`
Devuelve TRUE si <expresión> es un valor de campo múltiple, de lo contrario devuelve FALSE.

`(neq <expresión> <expresión>+)`
Devuelve TRUE si su primer argumento no es igual en tipo y valor a todos sus argumentos subsecuentes, de lo contrario devuelve FALSE.

`(not <expresión>)`
Devuelve TRUE si su argumento se evalúa a FALSE, de lo contrario devuelve TRUE.

(numberp <expresión>)
Devuelve TRUE si <expresión> es un punto flotante o un entero, de lo contrario devuelve FALSE.

(oddp <expresión>)
Devuelve TRUE si <expresión> es un entero impar, de lo contrario devuelve FALSE.

(or <expresión>+)
Devuelve TRUE si cualquiera de sus argumentos se evalúa a TRUE, de lo contrario devuelve FALSE.

(stringp <expresión>)
Devuelve TRUE si <expresión> es una cadena, de lo contrario devuelve FALSE.

(symbolp <expresión>)
Devuelve TRUE si <expresión> es un símbolo, de lo contrario devuelve FALSE.

(= <expresión-numérica> <expresión-numérica>+)
Devuelve TRUE si su primer argumento es de un valor numérico igual al de todos sus argumentos subsecuentes, de lo contrario devuelve FALSE.

(<> <expresión-numérica> <expresión-numérica>+)
Devuelve TRUE si su primer argumento no es de un valor numérico igual al de todos sus argumentos subsecuentes, de lo contrario devuelve FALSE.

(> <expresión-numérica> <expresión-numérica>+)
Devuelve TRUE si, para todos sus argumentos, el argumento $n-1$ es mayor que el argumento n, de lo contrario devuelve FALSE.

(>= <expresión-numérica> <expresión-numérica>+)
Devuelve TRUE si, para todos sus argumentos, el argumento $n-1$ es mayor o igual que el argumento n, de lo contrario devuelve FALSE.

(< <expresión-numérica> <expresión-numérica>+)
Devuelve TRUE si, para todos sus argumentos, el argumento $n-1$ es menor que el argumento n, de lo contrario devuelve FALSE.

(<= <expresión-numérica> <expresión-numérica>+)
Devuelve TRUE si, para todos sus argumentos, el argumento $n-1$ es menor o igual que el argumento n, de lo contrario devuelve FALSE.

Funciones de campo múltiple

(create$ <expresión>*)
Añade cero o más expresiones juntas para crear un valor de campo múltiple.

(delete$ <expresión-campo-múltiple>
<expresión-entero-inicial>
<expresión-entero-final>)

Elimina todos los campos del rango especificado (<expresión-entero-inicial> y <expresión-entero-final>) de <expresión-campo-múltiple> y devuelve el resultado.

(explode$ <expresión-cadena>))
Devuelve un valor de campo múltiple creado de los campos contenidos en una cadena.

(first$ <expresión-campo-múltiple>)
Devuelve el primer campo de <expresión-campo-múltiple>.

(implode$ <expresión-campo-múltiple>)
Devuelve una cadena que contiene los campos de un valor de campo múltiple.

(insert$ <expresión-campo-múltiple>
<expresión-entero>
<expresión-campo-simple-o-múltiple>+)
Inserta todos los valores de <expresión-campo-simple-o-múltiple> en la <expresión-campo-múltiple> antes del *enésimo* valor (<expresión-entero>) de <expresión-campo-múltiple>.

(length$ <expresión-campo-múltiple>)
Devuelve el número de campos de un valor de campo múltiple.

(member$ <expresión-campo-simple>
<expresión-campo-múltiple>)
Devuelve la posición del primer argumento en el segundo argumento o FALSE si el primer argumento no está contenido en el segundo argumento.

(nth$ <expresión-entero> <expresión-campo-múltiple>)
Devuelve el *enésimo* campo (<expresión-entero>) contenido en <expresión-campo-múltiple>.

(replace$ <expresión-campo-múltiple>
<expresión-entero-inicial>
<expresión-entero-final>
<expresión-campo-simple-o-múltiple>+)
Reemplaza los campos del rango especificado (<expresión-entero-inicial> o <expresión-entero-final>) en <expresión-campo-múltiple> con todos los valores de <expresión-campo-simple-o-múltiple> y devuelve el resultado.

(rest$ <expresión-campo-múltiple>)
Devuelve un valor de campo múltiple que contiene todos los campos menos el primero de <expresión-campo-múltiple>.

(subseq$ <expresión-campo-múltiple>
<expresión-entero-inicial>
<expresión-entero-final>)
Extrae los campos del rango especificado (<expresión-entero-inicial> a <expresión-entero-final>) de <expresión-campo-múltiple> y los devuelve en un valor de campo múltiple.

(subsetp <expresión>)
Devuelve TRUE si el primer argumento es un subconjunto del segundo argumento, de lo contrario devuelve FALSE.

Funciones de cadena

(lowcase<expresión-cadena-o-símbolo>)
Devuelve su argumento con todas las letras mayúsculas reemplazadas con letras minúsculas.

(str-cat <expresión>*)
Devuelve todos sus argumentos concatenados como una cadena.

(str-compare <expresión-cadena-o-símbolo>
<expresión-cadena-o-símbolo>)
Devuelve cero si ambos argumentos son iguales, de lo contrario devuelve un valor diferente a cero.

(str-index <expresión-lexema> <expresión-lexema>)
Devuelve la posición del entero del primer argumento dentro del segundo argumento si el primero es una subcadena del segundo, de lo contrario devuelve FALSE.

(str-length <expresión-cadena-o-símbolo>)
Devuelve la longitud de una cadena en caracteres.

(sub-string <expresión-entero-inicial>
<expresión-entero-final>
<expresión-cadena>)
Devuelve la subcadena de <expresión-cadena> del rango especificado (<expresión-entero-inicial> a <expresión-entero-final>).

(sym-cat <expresión>*)
Devuelve todos sus argumentos concatenados como un símbolo.

(upcase <expresión-cadena-o-símbolo>)
Devuelve su argumento con todas las letras minúsculas reemplazados con letras mayúsculas.

Funciones de E/S

(close [<nombre-lógico>])
Cierra el archivo asociado con el nombre lógico <nombre-lógico> (o todos los archivos si no están especificados). Devuelve TRUE si el archivo se cerró con éxito, de lo contrario devuelve FALSE.

(format <nombre-lógico> <expresión-cadena>
<expresión>*)
Evalúa e imprime como salida formateada al nombre lógico <nombre-lógico> cero o más expresiones formateadas por <expresión-cadena>. Consulte la sección 8.12 para conocer detalles sobre indicadores de formato.

(open <nombre-archivo> <nombre-lógico> [<modo>])
Abre el archivo <nombre-archivo> en el modo especificado ("r", "w", "r+" o "a") y asocia el nombre lógico <nombre-lógico> con el archivo. Devuelve TRUE si el archivo se abrió con éxito, de lo contrario devuelve FALSE.

(printout <nombre-lógico> <expresión>*)
Evalúa e imprime como salida sin formato al nombre lógico <nombre-lógico> cero o más expresiones.

(read [<nombre-lógico>])
Lee un campo simple del nombre lógico especificado (*stdin*, si no se especificó). Devuelve el campo si tiene éxito o EOF si no hay entrada disponible.

(readline [<nombre-lógico>])
Lee una línea completa del nombre lógico especificado (*stdin*, si no se especificó). Devuelve una cadena si tiene éxito o EOF si no hay entrada disponible.

(remove <nombre-archivo>)
Elimina el archivo <nombre-archivo>.

(rename <nombre-archivo-antiguo> <nombre-archivo-nuevo>)
Cambia el nombre del archivo <nombre-archivo-antiguo> por <nombre-archivo-nuevo>.

Funciones matemáticas básicas

(abs <expresión-numérica>)
Devuelve el valor absoluto de su único argumento.

(div <expresión-numérica> <expresión-numérica>+)
Devuelve el valor del primer argumento dividido entre cada uno de los argumentos subsecuentes. La división se realiza utilizando cálculos aritméticos con enteros.

(float <expresión-numérica>)
Devuelve su único argumento convertido a punto flotante de tipo.

(integer <expresión-numérica>)
Devuelve su único argumento convertido a entero de tipo.

(max <expresión-numérica> <expresión-numérica>+)
Devuelve el valor de su argumento más grande.

(min <expresión-numérica> <expresión-numérica>+)
Devuelve el valor de su argumento más pequeño.

(+ <expresión-numérica> <expresión-numérica>+)
Devuelve la suma de sus argumentos.

(- <expresión-numérica> <expresión-numérica>+)
Devuelve el valor del primer argumento menos la suma de todos los argumentos subsecuentes.

(* <expresión-numérica> <expresión-numérica>+)
Devuelve el producto de sus argumentos.

(/ <expresión-numérica> <expresión-numérica> +)
Devuelve el valor del primer argumento dividido por cada uno de los argumentos subsecuentes.

Funciones trigonométricas

(acos <expresión-numérica>)
Devuelve el coseno del arco de su argumento (en radianes).

(acosh <expresión-numérica>)
Devuelve el coseno del arco hiperbólico de su argumento (en radianes).

(acot <expresión-numérica>)
Devuelve la tangente del arco de su argumento (en radianes).

(acoth <expresión-numérica>)
Devuelve la cotangente del arco hiperbólico de su argumento (en radianes).

(acsc <expresión-numérica>)
Devuelve la cosecante del arco de su argumento (en radianes).

(acsch <expresión-numérica>)
Devuelve la cosecante del arco hiperbólico de su argumento (en radianes).

(asec <expresión-numérica>)
Devuelve la secante del arco de su argumento (en radianes).

(asech <expresión-numérica>)
Devuelve la secante del arco hiperbólico de su argumento (en radianes).

(asin <expresión-numérica>)
Devuelve el seno del arco de su argumento (en radianes).

(asinh <expresión-numérica>)
Devuelve el seno del arco hiperbólico de su argumento (en radianes).

(atan <expresión-numérica>)
Devuelve la tangente del arco de su argumento (en radianes).

(atanh <expresión-numérica>)
Devuelve la tangente del arco hiperbólico de su argumento (en radianes).

(cos <expresión-numérica>)
Devuelve el coseno de su argumento (en radianes).

(cosh <expresión-numérica>)
Devuelve el coseno hiperbólico de su argumento (en radianes).

(cot <expresión-numérica>)
Devuelve la cotangente de su argumento (en radianes).

(coth <expresión-numérica>)
Devuelve la cotangente hiperbólica de su argumento (en radianes).

(csc <expresión-numérica>)
Devuelve la cosecante de su argumento (en radianes).

(csch <expresión-numérica>)
Devuelve la cosecante hiperbólica de su argumento (en radianes).

(sec <expresión-numérica>)
Devuelve la secante de su argumento (en radianes).

(sech <expresión-numérica>)
Devuelve la secante hiperbólica de su argumento (en radianes).

(sin <expresión-numérica>)
Devuelve el seno de su argumento (en radianes).

(sinh <expresión-numérica>)
Devuelve el seno hiperbólico de su argumento (en radianes).

(tan <expresión-numérica>)
Devuelve la tangente de su argumento (en radianes).

(tanh <expresión-numérica>)
Devuelve la tangente hiperbólica de su argumento (en radianes).

Funciones de conversión

(deg-grad <expresión-numérica>)
Devuelve el valor de su argumento convertido de unidades de grados a unidades de gradientes.

(deg-rad <expresión-numérica>)
Devuelve el valor de su argumento convertido de unidades de grados a unidades de radianes.

(exp <expresión-numérica>)
Devuelve el valor de *e* elevado a la potencia de su único argumento.

(grad-deg <expresión-numérica>)
Devuelve el valor de su argumento convertido de unidades de gradientes a unidades de grados.

(log <expresión-numérica>)
Devuelve el logaritmo base *e* de su argumento.

(log10 <expresión-numérica>)
Devuelve el logaritmo base 10 de su argumento.

(mod <expresión-numérica> <expresión-numérica>)
Devuelve el sobrante del resultado de la división de su primer argumento entre su segundo argumento.

(pi)
Devuelve el valor de π.

(rad-deg <expresión-numérica>)
Devuelve el valor de su argumento convertido de unidades de radianes a unidades de grados.

(round <expresión-numérica>)
Devuelve el valor de su argumento redondeado al entero más cercano.

(sqrt <expresión-numérica>)
Devuelve la raíz cuadrada de su argumento.

(** <expresión-numérica> <expresión-numérica>)
Devuelve el valor de su primer argumento elevado a la potencia de su segundo argumento.

Funciones de procedimiento

(bind <variable> <valor>)
Une una variable a un valor especificado.

(if <expresión-predicado>
then <expresión> + [else <expresión>+])
Evalúa la expresión contenida en la porción *else* de la función si la <expresión-predicado> se evalúa al símbolo FALSE, de lo contrario evalúa las expresiones contenidas en la porción *then* de la función.

(while <expresión-predicado> [do] <expresión>*)
Evalúa <expresión>* siempre y cuando la <expresión-predicado> no evalúe al símbolo FALSE.

Otras funciones

(gensym)
Devuelve un símbolo secuenciado de la forma genX donde *X* es un entero.

(gensym*)
Devuelve un símbolo secuenciado de la forma genX donde *X* es un entero. A diferencia de la función gensym, gensym* produce un símbolo único que no existe en la actualidad dentro del ambiente de CLIPS.

(random)
Devuelve un valor entero "aleatorio".

(seed <expresión-entero>)
Establece la semilla usada por el generador del número aleatorio para la función *random*.

(setgen <expresión-entero>)
Establece el índice de secuencia usado por *gensym* y *gensym**.

(time)
Devuelve un valor de punto flotante que representa el número de segundos transcurridos desde un tiempo de referencia del sistema.

Comandos de ambiente

(apropos <lexema>)
Despliega todos los símbolos definidos actualmente en el ambiente de CLIPS que contiene la subcadena <lexema> especificada.

(batch <nombre-archivo>)
Permite el procesamiento "en lotes" de comandos interactivos de CLIPS al reemplazar entrada estándar con el contenido del archivo <nombre-archivo>. Devuelve TRUE si el archivo se ejecutó con éxito, de lo contrario devuelve FALSE.

(clear)
Elimina todas las construcciones del ambiente de CLIPS.

(exit)
Sale del ambiente de CLIPS.

(get-dynamic-constraint-checking)
Devuelve el valor actual de la acción de verificación dinámica de restricción.

(get-static-constraint-checking)
Devuelve el valor actual de la acción de verificación estática de restricción.

(load <nombre-archivo>)
Carga las construcciones almacenadas en el archivo especificado por <nombre-archivo> en el ambiente de CLIPS. Devuelve TRUE si el archivo se cargó con éxito, de lo contrario devuelve FALSE.

(reset)
Reinicia el ambiente de CLIPS.

(save <nombre-archivo>)
Guarda todas las construcciones del ambiente de CLIPS en el archivo especificado por <nombre-archivo>.

(set-dynamic-constraint-checking
<expresión-booleana>)
Inhabilita la verificación dinámica de restricción si <expresión-booleana> es FALSE, de lo contrario habilita la verificación dinámica de restricción. Se devuelve el valor antiguo de la acción de verificación de restricción.

(set-static-constraint-checking
<expresión-booleana>)
Inhabilita la verificación estática de restricción si <expresión-booleana> es FALSE, de lo contrario habilita la verificación estática de restricción. Se devuelve el valor antiguo de la acción de verificación de restricción.

(system <expresión-lexema>*)
Concatena sus argumentos como una cadena y pasa la cadena como un comando que habrá de ejecutarse por el sistema operativo.

Comandos de depuración

(dribble-off)
Detiene el envío de salida al archivo de registro abierto con la función *dribble-on*. Devuelve TRUE si el archivo de registro se cerró con éxito, de lo contrario devuelve FALSE.

(dribble-on <nombre-archivo>)
Envía al archivo de registro <nombre-archivo> toda la salida que normalmente va a la pantalla. Devuelve TRUE si el archivo de registro se abrió con éxito, de lo contrario devuelve FALSE.

(unwatch <elemento-observación>)
Inhabilita el despliegue de mensajes informativos cuando suceden ciertas operaciones de CLIPS.

(watch <elemento-observación>)
<elemento-observación> ::= activations | all |
compilations | facts | focus |
rules | statistics
Habilita el despliegue de mensajes informativos cuando suceden ciertas operaciones de CLIPS.

Comandos de deftemplate

(list-deftemplates [<nombre-módulo>])
Enumera las deftemplates en el módulo especificado (o en el módulo actual si no se especifica <nombre-módulo>).

(ppdeftemplate <nombre-deftemplate>)
Despliega el texto de la deftemplate especificada.

(undeftemplate <nombre-deftemplate>)
Elimina la deftemplate especificada.

Funciones y comandos de hecho

(assert <patrón-lado-derecho>)
Agrega uno o más hechos a la lista de hechos. Se devuelve la dirección de hecho del último hecho agregado.

(assert-string <expresión-cadena>)
Convierte una cadena en un hecho y lo afirma. Devuelve la dirección de hecho del hecho recién afirmado.

(dependencies <índice-hecho-o-dirección-hecho>)
Enumera todas las igualaciones parciales de las que el hecho especificado recibe soporte.

(dependents <índice-hecho-o-dirección-hecho>)
Enumera todos los hechos que reciben soporte lógico del hecho especificado.

(duplicate <índice-hecho-o-dirección-hecho>)
<ranura-lado-derecho>*)
Afirma una copia duplicada de un hecho de deftemplate con uno o más valores de ranura cambiados.

(hechos [<nombre-módulo>]
[<expresión-entero-inicial>
[<expresión-entero-final>
[<expresión-entero-máxima>]]])
Despliega los hechos en la lista de hechos. Si se especifica <nombre-módulo>, sólo se enumeran aquellos hechos visibles para el módulo especificado, de lo contrario se enumeran los hechos visibles para el módulo actual. Los hechos con índices de hecho menores que <expresión-entero-inicial> o mayores que <expresión-entero-final> no se enumeran. Si se especifica <expresión-entero-máxima>, entonces no se enumerarán más hechos además de ese valor.

(fact-index <expresión-dirección-hecho>)
Devuelve el índice de hecho asociado con una dirección de hecho.

(get-fact-duplication)
Devuelve el valor actual de la acción de duplicación de hecho.

(load-facts <nombre-archivo>)
Afirma los hechos contenidos en el archivo <nombre-archivo> del módulo actual. Devuelve TRUE si tiene éxito, de lo contrario devuelve FALSE.

(modify <índice-hecho-o-dirección-hecho> <ranura-RHS>*)
Cambia uno o más valores de ranura de un hecho de deftemplate.

(retract <índice-hecho-o-dirección-hecho>+)
Elimina uno o más hechos de la lista de hechos.

(save-facts <nombre-archivo>
[visible | local <nombres-deftemplate>*])
Guarda los hechos especificados al archivo <nombre-archivo>. Devuelve TRUE si tiene éxito, de lo contrario devuelve FALSE.

(set-fact-duplication <expresión-booleana>)
Permite afirmar hechos idénticos si <expresión-booleana> es FALSE, de lo contrario evita agregar hechos duplicados a la lista de hechos. Se devuelve el valor antiguo de la acción de duplicación.

Comandos de deffacts

(list-deffacts [<nombre-módulo>])
Enumera los deffacts del módulo especificado (o del módulo actual si <nombre-módulo> no se especifica).

(ppdeffacts <nombre-deffacts>)
Despliega el texto del deffacts especificado.

(undeffacts <nombre-deffacts>)
Elimina el deffacts especificado.

Comandos de defrule

(list-defrules [<nombre-módulo>])
Enumera las defrules del módulo especificado (o del módulo actual si <nombre-módulo> no se especifica).

(matches <nombre-defrule>)
Despliega la lista de hechos e igualaciones parciales que igualan los patrones de la regla especificada.

(ppdefrule <nombre-defrule>)
Despliega el texto de la defrule especificada.

(refresh <nombre-defrule>)
Actualiza la defrule especificada. Las activaciones de la regla que ya se han disparado pero aún son válidas se colocan en la agenda.

(remove-break [<nombre-defrule>])
Elimina un punto de división para la regla especificada. Si la regla no se especifica, se eliminan todos los puntos de división.

(set-break <nombre-defrule>)
Establece un punto de división para la regla especificada. Esto provocará la detención de la ejecución de la regla antes de que se dispare.

(show-breaks [<nombre-módulo>])
Enumera las reglas del módulo especificado que tienen puntos de división establecidos (o del módulo actual si <nombre-módulo> no se especifica).

(undefrule <nombre-defrule>)
Elimina la defrule especificada.

Funciones y comandos de agenda

(agenda [<nombre-módulo>])
Enumera las activaciones en la agenda del módulo especificado (o del módulo actual si <nombre-módulo> no se especifica).

(clear-focus-stack)
Devuelve todos los nombres de módulo de la pila de concentración.

(focus <nombre-módulo>+)
Pone uno o más módulos en la pila de concentración. Los módulos especificados se colocan en la pila de concentración en el orden inverso de la lista de argumentos.

(get-focus)
Devuelve el nombre de módulo de la concentración actual.

(get-focus-stack)
Devuelve todos los nombres de módulo de la pila de concentración como un valor de campo múltiple.

(halt)
Detiene la ejecución de reglas.

(list-focus-stack)
Enumera todos los nombres de módulo en la pila de concentración.

(pop-focus-stack)
Devuelve el módulo nombre de la concentración actual y elimina la concentración actual de la pila de concentración.

(run [<expresión-entero>])
Inicia la ejecución de reglas de la concentración actual. Si <expresión-entero> se especifica, entonces sólo se ejecuta ese número de reglas, de lo contrario la ejecución se detiene cuando la agenda está vacía.

Comandos de defmodule

(get-current-module)
Devuelve el módulo actual.

(list-defmodules)
Enumera todos los defmodules del ambiente de CLIPS.

(ppdefmodule <nombre-defmodule>)
Despliega el texto del defmodule especificado.

(set-current-module <nombre-defmodule>)
Establece el módulo actual para el módulo especificado y devuelve el módulo actual previo.

APÉNDICE F

CLIPS BNF

Programa CLIPS

```
<CLIPS-program>   ::= <construct>*

<construct>       ::= <deftemplate-construct> |
                      <deffacts-construct> |
                      <defrule-construct> |
                      <defmodule-construct>
```

Deftemplate Construct

```
<deftemplate-construct>      ::= (deftemplate <name>
                                   [<comment>]
                                   <slot-definition> *)

<slot-definition>            ::= <single-slot-definition> |
                                 <multislot-definition>

<single-slot-definition>     ::= (slot <slot-name>
                                       <slot-attribute>*)

<slot-name>                  ::= <symbol>

<multislot-definition>       ::= (multislot <slot-name>
                                   <slot-attribute>*)

<slot-attribute>        ::= <type-attribute> |
                            <allowed-constant-attribute> |
                            <range-attribute> |
                            <cardinality-attribute> |
                            <default-attribute>

<type-attribute>        ::= (type <type-specification>)

(type-specification>    ::= <allowed-type>+ | ?VARIABLE
```

```
<allowed-type>          ::= SYMBOL | STRING | LEXEME |
                            INTEGER | FLOAT | NUMBER

<allowed-constant-attribute>
                   ::= (allowed-symbols <symbol-list>) |
                       (allowed-strings <string-list>) |
                       (allowed-lexemes <lexeme-list> |
                       (allowed-integers <integer-list>) |
                       (allowed-floats <float-list>) |
                       (allowed-numbers <number-list>) |
                       (allowed-values <value-list>)

<symbol-list>           ::= <symbol>+ | ?VARIABLE

<string-list>           ::= <string>+ | ?VARIABLE

<lexeme-list>           ::= <lexeme>+ | ?VARIABLE

<integer-list>          ::= <integer>+ | ?VARIABLE

<float-list>            ::= <float>+ | ?VARIABLE

<number-list>           ::= <number>+ | ?VARIABLE

<value-list>            ::= <constant>+ | ?VARIABLE

<range-attributes>      ::= (range <range-specification>
                                   <range-specification>

<range-specification>   ::= <number>+ | ?VARIABLE

<cardinality-attribute>
            ::= (cardinality <cardinality-specification>
                             <cardinality-specification>)

<cardinality-specification>  ::= (integer> | ?VARIABLE

<default-attribute>     ::= (default <default-item>) |
                            (default-dynamic <expression>*)

<default-item>          ::= ?DERIVE | ?NONE | <expression>*
```

Deffacts Construct

```
<deffacts-construct>    ::=(deffacts <name> [<comment>]
                             <RHS-pattern>*)
```


Defrule Construct

```
<defrule-construct>     ::= (defrule <name> [<comment>]
                                [<declaration>]
                                <conditional-element>*
                                =>
                                <expression>*)

<declaration>           ::= (declare <rule-property>+)

<rule-property>   ::= (salience <integer-expression>)
                      (auto-focus <boolean-symbol>) |

<boolean-symbol>        ::= TRUE | FALSE

<conditional-element>   ::= <pattern-CE> |
                            <assigned-pattern-CE> |
                            <test-CE> |
                            <not-CE> |
                            <and-CE> |
                            <or-CE> |
                            <logical-CE> |
                            <exists-CE> |
                            <forall-CE>

<pattern-CE>            ::= <ordered-pattern-CE> |
                            <template-pattern-CE>

<assigned-pattern-CE>
            ::= <single-field-variable> <- <pattern-CE>

<test-CE>               ::= (test <function-call>)

<not-CE>                ::= (not <conditional-element>)

<and-CE>                ::= (and <conditional-element>+)

<or-CE>                 ::= (or <conditional-element>+)

<logical-CE>      ::= (logical <conditional-element>+)

<exists-CE>       ::= (exists <conditional-element>+)

<forall-CE>       ::= (forall <conditional-element>
                              <conditional-element>+)

<ordered-patterns-CE>  ::= (<symbol> <constraint>+)
```

```
<template-pattern-CE>
                    ::= (<deftemplate-name <LHS-slot>*)

<LHS-slot>              ::= <single-field-LHS-slot>
                            <multifield-LHS-slot>

<single-field-LHS-slot>  ::= (<slot-namte> <constraint>)

<multifield-LHS-slot>   ::= (<slot-name> <constraint>*)

<constraint>            ::= ? | $? | <connected-constraint>

<connected-constraint>
      ::= <single-constraint> |
          <sinle-constraint> & <connected-constraint> |
          <sinle-constraint> | <connected-constraint>

<single-constraint>     ::= <term> | ~<term>

<term>                  ::= <constant> |
                            <variable> |
                            :<function-call> |
                            =<function-call>
```

Fact Specification

```
<RHS-pattern>           ::= <ordered-RHS-pattern> |
                            <template-RHS-pattern>

<ordered-RHS-pattern>   ::= (<symbol> <RHS-field>+)

<template-RHS-pattern>
                    ::= (<deftemplate-name> <RHS-slot>*)

<RHS-slot>              ::= <single-field-RHS-slot> |
                            <multifield-RHS-slot>

<single-field-RHS-slot>::=(<slot-name> <RHS-field>)

<multifield-RHS-slot>   ::= (<slot-name> <RHS-field>*)

<RHS-field>             ::= <variable> |
                            <constant> |
                            <function-call>

<deftemplate-name>      ::= <symbol>
```

Variables and Expressions

```
<single-field-variable>::= ?<variable-symbol>
```



```
<multifield-variable>   ::= $?<variable-symbol>

<variable>              ::= <single-field-variable> |
                            <multifield-variable>

<function-call>   ::= (<function-name> <expression>*) |
                      <special-function-call>

<special-function-call>
            ::= (assert <RHS-pattern>+) |
                (modify <expression> <RHS-slot>+) |
                (duplicate <expression> <RHS-slot>+) |
                (bind <sinle-field-varible> <expression>)

<expression>            ::= <constant> |
                            <variable> |
                            <function-call>
```

Data types

```
<symbol>  ::= A valid symbol as specified in Section 7.4

<string>  ::= A valid string as defined in Section 7.4

<float>  ::= A valid float as defined in Section 7.4

<integer>
            ::= A valid integer as specified in Section 7.4

<number>    ::= <float> | <integer>

<lexeme>    ::= <symbol> | <string>

<constant>  ::= <number> | <lexeme>

<comment>   ::= <string>

<variable-symbol>       ::= A <symbol> beginning with an
                            alphabetic character

<function-name>         ::= <symbol>

<name>                  ::= <symbol>
```

Índice

Índice
= 61
" 87
~ 81, 331, 382
>> 80
" 331
"hágalo dos veces" 319
$ 358
$? 331, 377
% 358, 396
& 331, 383
(331
() 36
) 331
* 329, 358, 358, 570
** 572
+ 330, 334, 358, 569
+j 358, 486
- 570
/ 385, 570
< 328, 329, 331, 566
<- 367
<= 566
<== 342, 348
<> 61, 566
= 411, 566
==> 342, 348
=> 62, 346
=j 486
> 328, 386, 566
>, 329
>= 566
? 331
?DERIVE 442
?NONE 443
?VARIABLE 437
[328
\ 332
] 328
{ 328
{} 35
| 35, 86, 330, 331, 383
} 328
' 81
Π-función 263
∫ 35
ª, operador 249
- 385
∅ 86
(E 79
"a" 393
"r" 393
"r+" 393
"w" 393
+ 126

a posteriori 57
a posteriori, probabilidad 174
a priori 57
a priori, probabilidad 170
abajo 71
abducción 151
abierto, enunciado 83
abrir 393, 569
abs 569
absoluta, probabilidad 178
acción, marco de 77
acciones 27, 334
acíclico 98
acos 570
acosh 570
acot 570
acoth 570
acsc 570
acsch 570
activación 480
activación, función de 45
activado 12, 26, 346
acto 27, 182
actual, enfoque 464
Ada 328, 333
adecuado 86

aditiva, Ley 177
afirmación 125, 209
afinnativo en calidad 111
agenda 23, 346, 347, 463, 577
AKO 64
aleatoria, variable 293
aleatorio, error 167
alfa, memoria 482
algorítmica, programación 33
algoritmo 33
alternativa, negación 86
alto 265
ambiente 243
ambigüedad 166
ámbito 124
amplitud, primera, búsqueda 69
análisis, árbol de 62
analogía 8, 149
ancho de banda 263
and 565
and, elemento condicional 355, 415
and, restricción 383
and, restricción conectiva 385
ángulo, corchetes de 61
ANS 43
antecedente 25, 110
APL 36
aplicación, operaciones de 36
apropiado, subconjunto 79, 267
apropos 573
aproximado, razonamiento 282
árbol 97
arcos 63
ARE 65
argumento(s) 109, 385, 387
aritméticos, operadores 385
ART 328, 477
artificiales, sistemas neuronales 43
ART-IM 328
asec 570
asech 570
asignación 272
asignación de nombre, procedimientos de 36
asin 570
asinh 570
asociación 35
asociativa, memoria 46
asociativas, redes 64
assert 337, 437, 575
assert-string 575
atan 570
atanh 571
atenuación, factor de 240
atómico 63
átomos 36
atributos 65
atributo-valor, pares 66, 527
autoaprendizaje 99
autociclo 98
autoepistémico, razonamiento 155
auto-focus, atributo 469
automatizado, conocimiento, adquisición 100
automatizado, razonamiento, sistemas 78
AV 66
Axiomas 83, 125

barra 35, 61
básica, asignación 246
básica, probabilidad, asignación de 246
Bayes, Teorema de 181
bayesiana, toma de decisiones 183
Bel 250
beta, memoria 483
bicondicional 84
bien formadas, fórmulas 125
binaria, variable 272
bivalente, lógica 257
bloques, mundo de 371, 381
boca abajo 34,
boca abajo, razonamiento 145
bold, intersección 269
bold, unión 268
bordes 63, 97
bpa 246
búsqueda, espacio de 513
C 328, 333
cadena 61, 143, 330, 332, 437
cadena de razonamiento 110
cálculo de restricciones indistintas 288
calidad 316
calificación, reglas de 296
campo, restricción de 382, 384
campos 330
canónica, forma 293
cantidad 112
característica, función 256
cardinalidad 251
cardinalidad, atributo de 441
cargar 357, 574
cartesiano, producto 271
caso 39
caso, dirección de 330, 333
caso, nombre de 330, 333
categórico, silogismo 110

causal, cadena 134
causal, conocimiento 7
causal, conocimiento, marcos de 77
causales, asociaciones 153
CAUSE 66
centro de gravedad 291
cerrada, palabra, suposición de 152
cerrado, enunciado 83
cerrar 394, 568
certidumbre, factores de 210, 233, 236, 510
ciclo(s) 27, 98
cierto, evento 173
cilíndricas, extensiones 275
circuito 98
clase 39, 64
clásica, probabilidad 170
cláusulas, forma de 129
clear-focus-stack 466, 577
CLIPS 327, 477
CLIPS, lenguaje orientado a objetos 328
CLOS 328
codificar y solucionar, modelo 319
código y verificación, etapa de 324
codominio 35
Cognate 328
cognición 11
cognoscitiva, ciencia 10
cognoscitivo, procesador 12
comando 334
combinada, masa 249
combinatoria, explosión 44
comentario(s) 345, 358
Common Lisp Object System 328
comodines 376
compatibilidad de reglas 229
compatibilidad, función de 257
compensaciones 277
complemento 80
completa, evidencia 196
completar 128
composición 273
composición, operador de 273
composición, reglas de 295
composicional, regla 299
comprobable 127
compuesta, evidencia 201
compuesto, enunciado 84
compuesto, evento 171
computacional, lingüística 276
concentración 269
concepto, esquema 73
conclusión 78
condicional 84
condicional y conjuntiva, composición 295
condicional, composición 295
condicional, elemento 25
condicional, parte 25
condicional, probabilidad 178
conectado, gráfico 98
conectiva, restricción 382
conexionismo 43
confianza, calificación de 296
confianza, mantenimiento de 154, 424
confianza, valor de 285
confianza, valor de, del antecedente 286
configuración 16
conflictos, resolución de 27
conjunción 84
conjuntiva, composición 295
conjuntiva, forma normal 129
conjunto 35
conjunto de soporte 143
conjunto, complemento de 266
conjunto, contención de 266
conjunto, equivalencia de 266
conjunto, intersección de 267
conjunto, producto de 268
conjunto, unión de 267
conocimiento, abstracción de 21
conocimiento, adquisición, cuello de botella en 8
conocimiento, adquisición, tarea de 322
conocimiento, adquisición, utilería de 23
conocimiento, análisis, tarea de 322
conocimiento, base de 327
conocimiento, dominio de 3
conocimiento, estructura del 73
conocimiento, etapa de definición del 321
conocimiento, etapa de diseño del 323
conocimiento, etapa de verificación del 324
conocimiento, fuente 313
conocimiento, ingeniería del 6
conocimiento, programación basada en 14
conocimiento, sistema experto basado en 2, 6
conocimiento, tarea de extracción de 322
consecuencias 193
consecuente 27, 110
consenso 249
consistente 127
construcciones 335
consulta 70
consultables, nodos 209
contador, propagación de 45
contexto, sensible al 155
contextos 213

contingente 84
contingente, enunciados de 92
continuo 265
contradicción 84
contradicción de reglas 229
contrapositivo 119
control, estrategia de 30, 231
control, cadena de 396
control, estructura de 19
control, hecho 406, 448, 469, 525
control, patrón 406
convexas, funciones 300
convexo, conjunto bayesiano 300
COOL 328
cópula 111
cortar 68
cortes a 264
corto plazo, memoria a 12
cos 571
cosh 571
cot 571
coth 571
create$ 566
creencia, función de 250
creencia, medida de 251
crfl 353
crisp, conjuntos 257
cruce, punto de 260
csc 571
csch 571
cuantificación, reglas de 296
cuantificador 86, 112
cuantificador, libre de 129
cuerpo 499

datos, abstracción de 21
datos, fusión de 233
datos, objetos de 36
Dbt 253
decidible 127
decisión, árboles de 98
decisión, procedimiento 112
decisión, rejillas de 98
declarativo, conocimiento 58
declarativo, paradigma 38
deductiva, lógica 109
deffacts 342
definitivo, conocimiento 72
defmodule 456
defrule 345
deftemplate 335
    explícita 336
    implicada 336
degeneración, árbol de 98
deg-grad 571
deg-rad 571
delete$ 566
delimitador 331
Dempster, regla de combinación de 249, 254
Dempster-Shafer, teoría de 243
dependencias 425, 427, 575
dependiente(s) 425, 427, 575
depresión, prueba de 289
derecha, justificar a la 397
derecha, memoria 482
derecho, lado 25
derivación, árbol de 62
derrotable 155
descendente 34
descendente, razonamiento 145
desdifusión, problema de 292
desesperado 183
detallado, diseño, etapa de 323
detener 498, 499, 577
determinista 105, 171
devolución, restricción de campo del valor de 411
devolución, valor de 334, 386
diagrama 98
dilación 270
dinámico, conocimiento, estructura de 216
directa, suma 249
directo, razonamiento 115
discriminación, función de 256
disparar 26, 47
disparos 346
disposición 112, 298
distribución, problema de 37, 312
disyunción 84
disyuntiva, composición 295
disyuntiva, forma 129
div 569
doble, flecha 62
dominio 1, 35
dominio, variable de 87
dos entradas, nodo de 483
dos valores, lógica de 257
dribble-off 430, 574
dribble-on 430, 574
duda 253
duplicate 341, 575

Eclipse 328
efectiva, probabilidad, razón de 205
elástica, restricción 287

elementos 78
emular 2
enfoque 464, 577
enfoque, pila de 465
enrejado 98
entero 330, 437, 569
entrada, resolución de 143
enunciado 83
enunciados, cálculo de 83
enunciados, orientado a 33
envenenado 185
EOF 395, 397
epistémica, probabilidad 236
epistemología 57
eq 565
equivalencia, axioma de 128
errático 167
error 166
errores de medición 167
espacio(s) 214, 332, 365
especificidad 230
esperada, utilidad 185
espiral, modelo de 319
esquemas 73
esquematas 73
estado, espacio de 101
estado(s) 101, 193
estándar, cuadro 150
estándar, dispositivo de entrada 394, 397
estándar, dispositivo de salida 353
estándar, forma 111
estática, estructura de conocimiento 216
estática, verificación de restricción 439
estereotipos 73
estocástico, proceso 187
estocásticos, eventos independientes 176
estructura 98
estructural, semejanza 480
evenp 565
evento, árbol de 171
evento(s) 170, 171, 182
evidencia 144, 192
evidencia, medida de 246
evidencial, conflicto 255
evidencial, intervalo 250, 252
evidencial, razonamiento 243
exactitud 167
exacto, razonamiento 165
exclusivo, OR 45
existencial, cuantificador 88
existencial, importación 82
exists, elemento condicional 419
exp 572
experiencia 2, 3
experimental, probabilidad 173
experimental, conocimiento 20, 133
experto 2
experto, errores de conocimiento del 313
expertos, sistemas 1
explicación, utilería de 6, 23
explícita, prioridad 229
explícito, elemento condicional and 413
explícito, elemento condicional or 413
explode$ 398, 567
exportar 460
expresión 385, 388, 391
extensión, principio de 280
externa, dirección 330, 333
externas, funciones 407
extralógicas, características 129

fact 327, 334
    deftemplate 336
    ordenado 336, 341
facts 338, 575
falacia 118
falacia de la inversión 118
falso negativo 167
falso positivo 167
faltantes, reglas 233
fallos, detección, aislamiento y recuperación de 448
fallos, tolerancia a 46
fijo, matriz de estado 190
finito, estado, máquina de 102
first$ 567
FL 295
flecha 346
floatp 565
focal, elemento 247
forall, elemento condicional 422
formal, lógica 81
formal, prueba, tarea de 324
formato 396, 568
formato, marcas de 396
fraccionada, red semántica 214
fragmento 12
fuerte, escritura 323
función 334
funcional, programación 34
funcionales, formas 36
funciones 35

ganancias 185

garantía 10
General Problem Solver 11
generalizada, ley multiplicativa 179
generalizada, regla delta 48
generalizado, *modus ponens* 300
generar y probar 150
genérico 65
gensym* 522, 573
get-current-module 457, 578
get-dynamic-constraint-checking 440, 573
get-fact-duplication 576
get-focus 466, 577
get-focus-stack 466, 577
get-static-constraint-checking 439, 573
global, base de datos 23
grad-deg 572
grado de confirmación 236
grado de creencia 191
grado de membresía 258
gráfico 98
gramática 61
granularidad 13

hacia atrás, inducción 185
hacia atrás, razonamiento 26, 328, 526
hacia delante, razonamiento 26, 328
hecho, dirección de 367, 512
hecho, identificador de 338, 339, 512
hecho, índice de 338
hecho, lista de 327, 337
hecho-índice 575
herencia 39, 66
herramienta 22
heurística 72
heurístico, conocimiento 7
hijo 97
hipótesis 9, 167, 192
hipotético, razonamiento 6
hojas 97

if 498, 572
if-added 76
if-needed 75
if-removal 76
ignorancia 246, 253
Igr 253
igualaciones 27, 419, 459, 489, 576
imágenes 35
imperativo 33
implicaciones 85
implícita, prioridad 229
implícito, elemento condicional and 413
implode$ 567
importar 460
imposible, creencia 212
imposible, evento 173
impresión 353, 394, 395, 569
incertidumbre 165, 509
incompleto 166
incondicional, probabilidad 178
incondicional, soporte 427
inconsciente, conocimiento 58
inconsistente 127
incorrección 166
increíble 247
incremental, modelo de cascada 319
incrementalmente 10
independiente 181
independientes, eventos 176
indiferente 192
indirecto, razonamiento 119
indistinta, gramática 279
indistinta, lógica 204, 256
indistinta, probabilidad 295
indistinta, producción 279
indistinta, proposición 258
indistinta, relación 272
indistinta, restricción 287
indistinto, calificador 258, 295
indistinto, predicado 293
indistinto, razonamiento 282
indistinto, sistema experto 280
indistintos, cuantificadores 258
individuo 65
inducción 168
inducción, basado en 43
inducir 49
inductivo, aprendizaje 49
inexacto, razonamiento 166, 227
inferencia, errores de cadena de 315
inferencia, errores del motor de 314
inferencia, motor de 3, 13, 23, 327
inferencia, red de 208, 213
inferencia, redes 42
inferencias 4
inferior, límite 250
infija, forma 385
infinito, valor, lógica de 281
inicio, símbolo de 61
initial-fact 343, 346, 351, 461
insert$ 567
instanciado 26
integridad 128
intensificación 270

intergerp 270
interiores, probabilidades 179
interpretación 127
interrupt, comando 369
intersección 79
intervalo 257
intrusas, brechas 213
inválida, inducción 168
inválido 127
inversa, probabilidad 181
inversa, propagación 45, 48
inverso 119, 182
invertir 119
ir-removal 76
IS 66
ISA 64
izquierda, justificar a la 397
izquierda, memoria 483
izquierdo, lado 25

jerarquía 97
Johnson, centro de espacio de 328
justo 171

Kyburg 300
L1 281
largo plazo, memoria a 12
length$ 567
lenguaje 22
LEX 231
LEXEME 437
lexemp 565
lexicográfico 231
ley de contraposición 119
ley de separación, 115
leyes de inferencia 119
LHS 25, 346
libre 124
literales 129
limitada, diferencia 269
limitada, suma 268
limitado, producto 169
limitar 124, 365
límites de errores de ignorancia 315
límites de ignorancia 7
limpiar 344, 352, 573
lineal, modelo 320
lingüísticas, variables 276
LISP 328, 333
listas 36
list-deffacts 350, 458, 576
list-defmodules 578
list-defrules 350, 457, 576
list-deftemplates 350, 458, 575
list-focus-stack 465, 577
LN 281
load-facts 503, 519, 524
local 503
log 572
log10 572
lógica 78
lógica L 281
lógica, equivalencia 85
lógica, programación 78
logical, elemento condicional 424
lógicas, combinaciones 211
lógicas, variables, 83
lógico L 281
lógico, AND 80
lógico, nombre 353, 393
nulo 396
lógico, NOT 81
lógico, soporte 425
lote 428, 573
lowcase 568
Lukasiewics, operadores lógicos de 281
llaves 35

mágico, cuadro 149
MAIN, module 456
mal estructurados, problemas 18
marco de discernimiento 245
marco, problema de 154
marco(s) 42, 74
marginales, probabilidades 179
Markov, algoritmo de 30
Markov, proceso de cadena de 190
más general, cláusula 142
más general, unificador 142
masa 246
material, equivalencia 85
material, implicación 84
matriz 139, 213
max 569
máximo, método 289
máximo, producto 274
max-min 273
max-min, producto matriz de 273
max-min, regla composicional de inferencia 286
mayúsculas y minúsculas, sensible a 331
MEA 231
medición de aumento de creencia 236
medición de aumento de incredulidad 236
medio, término 111

medio, tiempo, entre fallos 316
medios, términos, análisis de 231
meet, operador 267
member$ 567
membresía, función de 257
menor, premisa 111
menor, término 111
menores, incrementos 319
meta(s) 38, 60, 68
metaconocimiento 60
metalenguaje 60
metametodología 318
metaregla 10, 154
metasímbolo 126
métrica 316
mgu 142
microincremento 319
min 569
min, función 204
mínimo, límite superior 299
misión 315
misión crítica 313
mitad del ancho 262
mix 289
mod 412, 572
modelo 127, 135, 207
modificación, reglas de 295
modificar 340, 576
modificar y duplicar 341
módulo 351
módulo, separador de 456
modus ponens 115
modus tollens 119
momentos, método de 289
monitoreo 538
monotonicidad 153
monotónicos, sistemas 153
muestra, espacio de 171
muestra, punto de 171
multifield, comodín 376
multifield, ranuras 336
multifield, valor 333
multifield, variables 493
multifieldp 565
multiplicativa, ley 179
multislot 335
mundo 154
mutual, independencia 176
mutualmente exclusivos, eventos 173
MYCIN 509

nada 36, 131
NAND 86
NASA 328
negación 84
negativa, calidad 111
neq 565
neurona 26
no comprobable 127
no confiable 142
no creíble 247
no determinista 171
no monotónico, sistema 153
no terminal 61
nodos 63, 97
nones 192
nones, forma de probabilidad de 194
NOR 86
normal 129
normalización 254, 271
normalizado 265
not 565
not, elemento condicional 417
not, restricción 382
novedad de hechos 231
nth$ 567
nula, cadena 30
nula, hipótesis 167
nulo 131
nulo, conjunto 80
numberp 566
numérico, campo 330
número 330, 437

OAV 66
objetivo, teoría de probabilidad 173
objeto, diseño orientado al 38
objeto, orientado al 38
objeto, programación orientada al 38, 327
objeto-atributo-valor, triple 66, 509
objetos 6
ocultas, capas 48
oddp 566
operandos 385
OPS5 328, 477
OPS83 477
or 566
or, restricción 383
or, restricción de campo 385
oración 61
oraciones, cálculo de 83
órdenes 128, 150
orientado, árbol 97
ortogonal, suma 249

padre 97
padre, cláusulas 130
pairwise independiente 176
paradigma 14
parcial, evidencia 196
parcial, igualación 479
particular 112
Pascal 328, 335
pasos 190
patrón 25, 345
patrón, CE 345
patrón, coincidencia de 480
patrón, igualación de 71
patrón, nodos de 481
patrón, parte 25
patrón, red de 480
patrón, unión de 367
perceptrón 47
permitido, atributo de cadenas 440
permitido, atributo de enteros 440
permitido, atributo de lexemas 440
permitido, atributo de números 440
permitido, atributo de puntos flotantes 440
permitido, atributo de símbolos 440
permitido, atributo de valores 440
personal, probabilidad 193
personal, teoría de construcción 323
peso(s) 44, 98
pi 572
pila 380
plan 151
planeación 513
planear-generar-probar 151
plasticidad 47
plausibilidad 250
plausible, creencia 212
plausible, inferencia 155
plausibles, relaciones 211
PLs 250
poblaciones 172
podado 185
poder de un conjunto 268
poder, conjunto de 93
ponderadas, combinaciones 211
pop-focus 466
pop-focus-stack 577
porfirítico 213
portabilidad 328
posibilidad 293
posibilidad, calificación de 296
posibilidad, distribución de 293
posibilidad, ecuación de asignación de 293
posible, creencia 212
Poss 293
posterior, probabilidad 174
postulados 125
ppdeffacts 350, 459, 576
ppdefmodule 578
ppdefrule 345, 350, 459, 576
ppdeftemplate 350, 459, 575
precedencia de operaciones aritméticas 388
precisión 167
predefinidas, funciones 407
predeterminado 75
predeterminado, atributo 442
predeterminado, dinámico, atributo 444
predeterminado, razonamiento 154
predicado 111
predicado, base de datos de 69
predicado, expresión de 67
predicado, función de 407, 498
predicado, función o variable de 499
predicado, funciones 87
predicado, lógica de 86
predicado, restricción de campo de 410
predicados 36
prefijo, forma de 385
premisas 78
prenex, forma 139
primario, término 278
primer momento de inercia 291
primer orden 86, 128
primitivas, funciones 36
primitivos, tipos de datos 330
principal, premisa 111
principal, término 111
principales, conectivos 117
principales, incrementos 319
principio de indiferencia 246
probabilidad 169, 192
probabilidad de suficiencia 194
probabilidad, calificación de 296
probabilidad, distribución de 293
probabilidad, razón de 193
probabilidad, vector de 187
probabilística, suma 268
probable, creencia 213
probado 127
problema, dominio de 3
problema, espacio de 102
procedimiento, archivos adjuntos de 75
procedimientos, conocimiento de 58
procedimientos, programa de 33

procedimientos, programación de 327
proceso, modelo de 318
producción, memoria de 24
producción, reglas de 11, 13
producción, sistemas de 13, 328
profundidad, primera, búsqueda de 69
profundo 73
profundo, conocimiento 7
profundo, razonamiento 154
programación 513
PROLOG 527
prominencia 346, 445, 454
prominencia, jerarquía de 451
pronóstico 9
propiedad 65
proposición 83
proposicional, cálculo 83
proposicional, función 124
proposicional, red 63
prototipo 76
proyección 274
prueba, revisión de preparación de 324
prueba, tareas de análisis de 324
punto flotante 330, 437, 569
puntos de interrupción 355

rad-deg 572
raíz, nodo 97
ramas 97
random 573
rango 35
rango de creencia 250
rango, atributo de 441
ranura(s) 74, 334, 335
rápido, prototipo 10
read 392, 395, 397, 569
readline 397, 569
real, tiempo 187
reconocer-actuar, ciclo 27
reducción al absurdo 130
redundantes, reglas 232
reescritura, reglas de 29
Referencia manual viii
referencia, transparente en 35
refracción 347
refresh 348, 498, 577
refutación 132
refutación completa 132
refutado 128
regla de sustitución 121
regla, inducción de 7
regla, modelo de 157
reglas 5
reglas de inferencia 119
reglas, programación basada en 327
regresar 41
regular, matriz de transición 190
relación 271
relación, nombre de 334
relacional, unión 274
rellenados 74
remove 569
remove-break 356, 459, 577
rename 569
repertorio, cuadrículas de 323
replace$ 567
reproducible, incertidumbre 227
reset 343, 574
resolución 128
resolución, árbol de refutación de 133
resolución, refutación de 132
resolvente 130
rest$ 567
restricción, relación de 299
Rete, algoritmo 32
Rete, algoritmo de igualación de patrones 477
retracción 340
retract 340, 367, 576
return 467
RHS 27, 346
round 572
run 346, 467, 578
run, comando 500
ruta 98

S, función 261
sabiduría 60
salir 333, 334, 573
save 358, 574
save-facts 503, 519
script 74
sec 571
secuencial 33
secundario, efecto 386
sech 571
seed 573
segundo orden 128
seleccionar-ejecutar, ciclo 27
semántica 60
semántica, red 63
semántica, regla 278
semánticos, errores 314
semidecidible 127
sensorimotor 73

sentido común, conocimiento de 109, 299
sentido común, razonamiento 155, 281
señales 330
set-break 355, 459, 577
set-current-module 457, 578
set-dynamic-constraint-checking 440, 574
set-fact-duplication 511, 576
setgen 573
set-static-constraint-checking 439, 574
shell 14, 23
show-breaks 356, 458, 577
singleton, conjunto 244
sigmoidea, función 45
significado 278
silogismo 78
simbólica, lógica 82
simbólicas, expresiones 36
símbolo 330, 331, 437
simple, evento 171
simple, evidencia 201
sin 571
sin procedimientos 33
sinapsis 47
single-field, comodín 370
single-field, ranuras 335
singleton 86
sinh 571
sintáctica, regla 278
sintaxis 60
sintaxis, errores de 314
síntesis 513
sísmica, encuesta 183
sistema, etapa de evaluación del 325
sistemático, error 167
situación-acción, ciclo 27
situacional, marco 77
situación-respuesta, ciclo 27
Skolem, funciones de 138
SmallTalk 328
software, metodología de ingeniería de 309
sonido 128
soporte 250, 263, 264
sospecha 253
Spt 250
sqrt 572
str-cat 568
str-compare 568
str-index 568
stringp 566
str-length 568
subclase 39, 65
subconjunto 79, 267
subjetiva, probabilidad 174
submetas 40, 68
subseq$ 567
substep 568
sub-string 568
subsunción de reglas 230
sujeto 111
sup 299
superclase(s) 39, 65
superficial 73
superficial, conocimiento 7
superficial, razonamiento 134
superior, límite 250
superior, nivel 28, 333, 334
superior, nivel, hipótesis de 209
superior, nivel, indicador de, 386
superior, parte 71
suposición del antecedente 115
suposiciones, mantenimiento de confianza basada en 154
supp 264
supremo 299
sustitución, casos de 140
symbolp 566
sym-cat 568
system 428, 574

tácito, conocimiento 58
tan 571
tanh 571
tareas 320
tautología 84
taxonomía 32, 513
temporal, razonamiento 186
temporal, redundancia 478
temporales, eventos 186
teorema 125, 126
terminal 61
terminales, nodos 481
término, conjunto de 277
test, elemento condicional 408, 493
textura 213
tiempo, restricciones de 8
tiempo, rótulo de 27, 231
time 444, 573
tipo 1, subconjunto indeterminado 265
tipo 2, subconjunto indeterminado 266
tipo I, error 167
tipo II, error 167
tipo N, subconjunto indeterminado 266
tipo, atributo 437
total, ancho 263

trabajo, memoria de 23
trabajo, plan de 321
traducción, reglas de, 295
transiciones 101
trazo 86
tres valores, lógica de 281
triplet 66
trivalente 281

umbral, valor de 45, 239
una entrada, nodos de
undeffacts 351, 459, 576
undefrule 351, 459, 577
undeftemplate 351, 459, 575
unidad, preferencia de 143
unificable 142
unificación 41, 140
unificación, algoritmo de 142
unificador 142
unión 80, 483
unión signo de 383
unión, operador de 267
unión, operador de negación de 86
unión, red de 483
unir 365, 391
unitario, sistema numérico 127
universal, conjunto 79
universal, cuantificador 87
UN-TIPO-DE 64
unwatch 342, 574
upcase 568
usuario, funciones definidas por el 333, 407
usuario, Guía del ix
usuario, interfaz de 23

V&V 157
vacía, lista 36
vacío, conjunto 80
validación 157
válido 82, 127
valor 65
variable(s) 365, 376, 391
Venn, diagrama de 78
verificación 157
verificación de detención 27
vértices 97
vida, ciclo de 317
vínculos 63, 97
visible 503
vuelo, técnicas, paneles de 313

watch 341, 466, 574
    activaciones 341, 348
    todos 341
    compilaciones 341, 357, 485
    hechos 341
    enfoque 341
    reglas 341, 349
    estadísticas 341, 349
wffs 125
while 498, 499 572

XOR 45

28 ENERO
9:30

Esta obra se teminó de imprimir en el mes de
Agosto de 2001, en los talleres de
Editorial Color, S.A de C.V. Naranjo 96 bis p.b.
Col. Santa María la Ribera C.P. 06400
México D.F. Tel. 5547-1542