Maynard Kong

INTELIGENCIA ARTIFICIAL

PONTIFICIA UNIVERSIDAD CATOLICA DEL PERU
FONDO EDITORIAL 1993

INTELIGENCIA ARTIFICIAL

Maynard Kong

INTELIGENCIA ARTIFICIAL

PONTIFICIA UNIVERSIDAD CATOLICA DEL PERU
FONDO EDITORIAL 1993

Primera edición, setiembre de 1993

Diagramación: José C. Cabrera Zúñiga
Nora O. Cabrera Zúñiga

INTELIGENCIA ARTIFICIAL

ISBN 84-89309-86-8

Impreso en el Perú - Printed in Peru

Maynard Kong. *En 1964 ingresó a la Facultad de Ciencias Físicas y Matemáticas de la Universidad Nacional de Ingeniería. Egresó en 1968 y desde 1969 se ha desempeñado como profesor del Departamento de Ciencias de la Universidad Católica en cursos de Matemáticas de niveles y especialidades variados. Obtuvo el grado de doctor (PhD) en la Universidad de Chicago (Estados Unidos de América) en 1976. Fue profesor visitante en la Universidad de Stuttgart (República Federal de Alemania) en 1979, y al mismo tiempo becario de la Fundación von Humboldt en un programa de posdoctorado, y posteriormente, también en Venezuela, durante cuatro años.*

Ha publicado importantes trabajos de investigación y varios textos de consulta universitaria, entre los que se pueden mencionar: Teoría de Conjuntos (coautor), Cálculo Diferencial, Cálculo Integral, Basic, Lenguaje de Programación Pascal, Lenguaje de Programación C y Lenguaje Ensamblador Macro Assembler

Ha participado en numerosos eventos de Matemáticas, promoción de las Ciencias Básicas e Informática tanto en el país como en el extranjero.

Prólogo

En este texto se exponen los principios fundamentales de la Inteligencia Artificial y sus aplicaciones en el diseño y realización de sistemas computadorizados inteligentes.

El material desarrollado en el libro se puede utilizar en cursos de diferentes niveles. Un curso introductorio, que presente esencialmente las ideas básicas de la Inteligencia Artificial, puede formarse con los temas de los Capítulos 1-6, excluyendo los programas de aplicaciones. Y un curso de nivel avanzado ha de comprender íntegramente los primeros seis capítulos y opcionalmente el Capítulo 7, referente al lenguaje de programación LISP, que en forma independiente también podría emplearse en un curso dedicado a este lenguaje.

Los programas expuestos, siete escritos en lenguaje C y tres en LISP, son sistemas completos, listos para ser aplicados inmediatamente por el lector, que resuelven problemas de distintas áreas: búsqueda de objetivos y rutas, sistemas basados en reglas (sistemas expertos y de generación y prueba) y análisis y comprensión de lenguajes. Desde luego, es muy recomendable que tales programas sean tomados como modelos o prototipos para el desarrollo de las aplicaciones particulares de los interesados.

Maynard Kong

Indice

Capítulo 1

Conceptos Básicos

1.1 INTRODUCCION

La Inteligencia Artificial (IA) es una rama de la Ciencia de la Computación que estudia los fundamentos teóricos y prácticos del diseño de sistemas de computación "inteligentes", esto es, sistemas que exhiben características inteligentes del ser humano, tales como: resolución de problemas, comprensión de lenguajes, aprendizaje, razonamiento, etc.

La IA trata de encontrar las técnicas para diseñar y programar máquinas -computadoras- que emulen y extiendan nuestras capacidades mentales.

La IA está relacionada con otras áreas del conocimiento: Ciencia de la Computación, Lingüística, Ingeniería, Medicina, Filosofía, Sicología, Física, Química, etc.

1.2 HISTORIA

La IA se inició a mediados del presente siglo a través de algunos programas sobre juegos y rompecabezas, en los cuales el computador era capaz de jugar como un jugador humano, es decir, el programa tomaba las decisiones del juego.

Los trabajos de George Boole y Alan Turing, "An Investigation of the Laws of Thought" y "Computing Machinery and Intelligence", respectivamente, constituyen las primeras contribuciones teóricas en IA. Turing estableció las ideas sobre un computador que podría ser programado de modo que muestre una conducta inteligente.

En 1950 Claude Shannon publicó un artículo sobre la posibilidad de que un computador juegue ajedrez.

Oficialmente se reconoce que la IA se originó en la conferencia organizada por John Mc Carthy y Marvin Minsky en el Darthmouth College (Hanover-New Hamphire, EUA), durante el verano de 1956, en la cual se aceptó como una conjetura básica la posibilidad de describir en forma precisa las distintas características y aspectos de la inteligencia de manera que una máquina pueda simularlos.

1.3 TOPICOS DE LA IA

Algunos temas de la IA son:

Búsqueda

Resolución de problemas

Demostración de teoremas

Análisis y comprensión de lenguajes naturales

Aprendizaje

Robótica

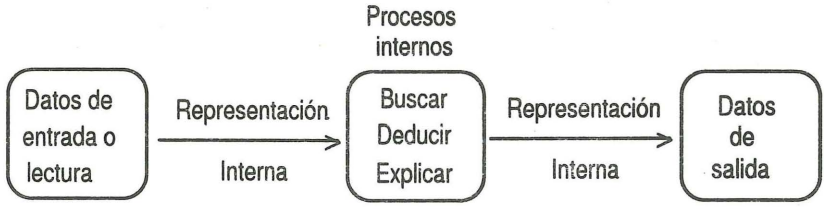
Visión

1.4 REPRESENTACION DE DATOS EN IA

Un *programa* es un conjunto de instrucciones que se suministran al computador para que lea datos, los procese, esto es, realice operaciones con ellos y tome decisiones, y luego proporcione resultados.

Tanto los datos de entrada como los de salida son dados en algún lenguaje natural. Sin embargo, los distintos procesos internos del programa usan y producen datos expresados en una forma especial, denominada *representación interna*. Así, cuando el programa recibe datos, procede a convertirlos a su representación interna, y debe expresar los resultados internos en lenguaje natural para producir datos de salida.

Una representación interna provee una notación adecuada para designar los objetos de un sistema particular y relaciones entre éstos.



Algunos tipos de notaciones para representar datos en los programas son los siguientes:

(1) Tipo *grafo* o *red semántica*, que emplea círculos u óvalos rotulados para representar a los objetos, y arcos rotulados, para las relaciones.

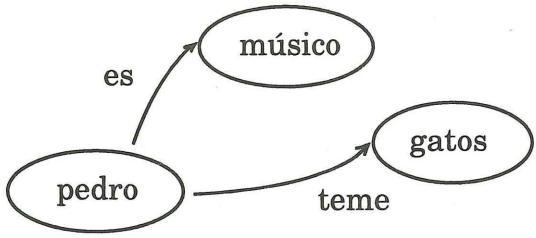
EJEMPLO

El sistema dado por las relaciones

a) pedro es músico

b) pedro teme a los gatos

se representa por:



(2) Tipo *estructura* o *registro*.

En este caso los objetos y relaciones se representan mediante una colección de datos simples llamados *campos*.

$$(\text{campo}_1 \quad \text{campo}_2 \quad \ldots \quad \text{campo}_n)$$

A su vez, un campo puede ser una colección de campos.

EJEMPLO

El sistema del ejemplo anterior se representa por:

($campo_1$ $campo_2$ $campo_3$)

(pedro (es músico) (teme gatos))

(3) Tipo *predicado*.

En esta notación cada relación se representa mediante un predicado o función con valores lógicos:

relación ($objeto_1$, $objeto_2$, ...)

que toma uno de los valores verdadero o falso, según se cumpla o no para los objetos involucrados.

EJEMPLO

Una representación de tipo predicado es dada por:

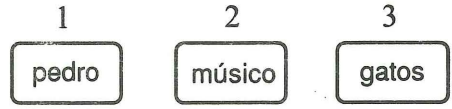
es(pedro, músico) (valor verdadero)

teme(pedro, gatos) (valor verdadero)

es(pedro, escritor) (valor falso)

(4) Tipo *índices* o *punteros*

Las relaciones se expresan mediante arreglos de índices o punteros a los objetos del sistema. Por ejemplo,



en donde los objetos: pedro, músico y gatos son indicados por los números 1, 2 y 3, respectivamente.

En este caso los hechos

"pedro es músico"

"pedro teme a los gatos"

dan lugar a dos arreglos, asociados a las relaciones *es* y *teme*, en los cuales se salvan tales hechos:

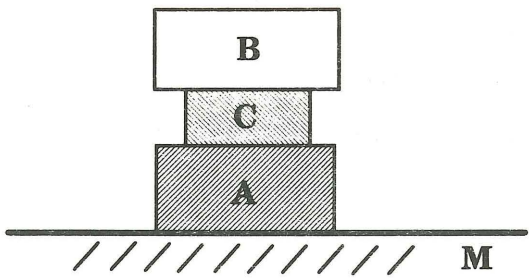
relaciones:
a) es | 1 2 | ;1 es 2
b) teme | 1 3 | ;1 teme 3

1.5 LENGUAJES DE PROGRAMACION EN IA

Los programas de IA se escriben frecuentemente en los lenguajes LISP o Prolog, que ofrecen recursos adecuados para representar o procesar datos de sistemas de IA y por lo tanto se les considera lenguajes orientados a la resolución de problemas de IA. También se utilizan lenguajes de programación de propósito general, como C o Pascal, pero éstos exigen, por lo general, un esfuerzo mayor en el diseño y manejo de los datos del problema.

1.6 EJERCICIOS

1. Se define un sistema por las reglas:
   a) A, B, C son bloques
   b) M es una mesa
   c) A está encima de M, C encima de A y B encima de C
   d) A es de color azul, B es blanco y C es rojo



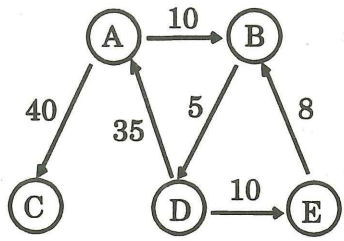
Represente el sistema mediante grafo y predicados

2. Si
   a) Pedro es hijo de Juan y María

y b) Luisa y Fernando son hijos de Pedro

represente estas relaciones usando estructuras e índices.

3. Represente el siguiente mapa de rutas entre las ciudades A, B, C, D y E.



a) mediante predicados de la forma: ruta(A, B, 10)...

b) utilizando estructuras o registros: ((A B 10)...)

4) Problema de los 9 puntos.

Dados 9 puntos como se indica en la figura:

* * *
* * *
* * *

trace con un lápiz, sin levantarlo, 4 líneas (o segmentos de rectas) que los unan.

Suponga que las líneas pueden extenderse fuera de los puntos.

5. Pedro se dirige a una ciudad A recorriendo un camino que en un cierto punto se bifurca. En este lugar habitan dos personas, una que siempre dice la verdad y otra que siempre miente. Pruebe que si uno de ellos responde afirmativamente a la pregunta:

¿O Ud. es veraz y el camino de la izquierda conduce a A,

o bien Ud. es mentiroso y el otro camino conduce a A ?

entonces es cierto que el camino de la izquierda conduce a A.

6. Dos cajas contienen galletas y chocolates y una tercera ambos productos. Las cajas tienen las etiquetas GALLETAS, CHOCOLATES y MIXTO, pero todas en forma errónea. Al extraer un producto de cada una de ellas se obtuvo un chocolate, una galleta y un chocolate, respectivamente. Determine el contenido de cada caja.

Capítulo 2

Problemas de Búsqueda

2.1 INTRODUCCION

La búsqueda de soluciones constituye un tema importante en IA. Sus métodos se aplican en diversos problemas de IA, por ejemplo en la determinación de rutas o caminos, juegos, rompecabezas, análisis de lenguajes, reducción de objetivos, demostración de teoremas, etc.

Un *espacio de búsqueda* consiste de:

(1) un conjunto finito E de objetos

$$e_1, \ldots, e_n$$

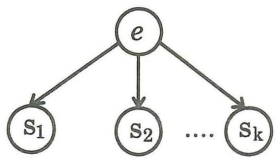
llamados *estados*,

y (2) subconjuntos S_e de E asociados a ciertos estados *e*.

Los estados $s_1, s_2, \ldots, s_k$ de S_e se denominan *sucesores* del estado *e*.

Frecuentemente se representa un espacio de búsqueda mediante un diagrama (grafo dirigido), en el cual a cada estado *e* se le asigna un nodo:

y entre un estado *e* y sus sucesores $s_1, s_2, \ldots, s_k$ se trazan arcos dirigidos:



(sucesores de *e*)

para indicar esta relación.

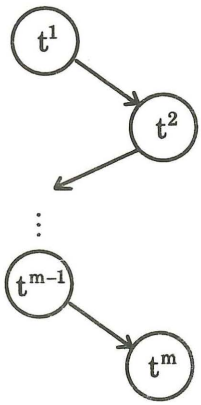
Una ruta del espacio de búsqueda es una sucesión de estados.

$$R = t^1\ t^2 \ldots t^m, \quad m \geq 1,$$

tal que cada estado t^{i+1} es un sucesor del anterior t^i, $i = 1, \ldots, m-1$.

Se dice entonces que R es una ruta entre t^1 y t^m, de t^1 a t^m o que une t^1 con t^m.

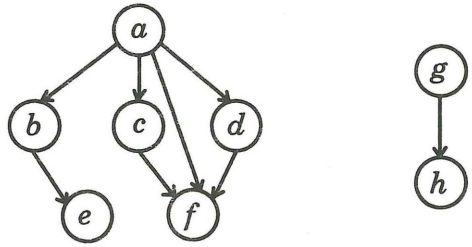
Gráficamente R se representa por:



Una ruta entre dos estados C y F es una manera o forma de unir C con F a través de sucesores de C, los sucesores de éstos, y así sucesivamente.

El problema básico de búsqueda consiste en determinar rutas entre dos estados dados.

EJEMPLO 1. El diagrama:



representa un espacio de búsqueda cuyos estados son: a, b, c, d, e, f, g, h. Los sucesores de algunos estados son:

(1) sucesores de a: b, c, d, f

(2) sucesor de g: h

(3) los estados e, f, h, no tienen sucesores.

Algunas rutas son:

(4) $a\ c\ f$, que une a con f

(5) $g\ h$

y (6) d

Observamos también que no existen rutas entre algunos pares de estados, por ejemplo entre b y f.

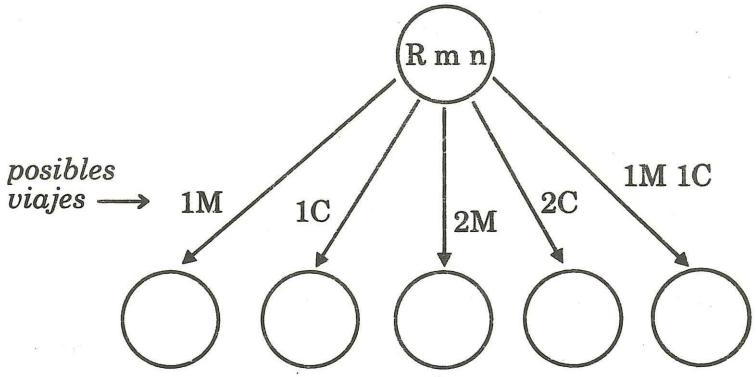
EJEMPLO 2. Problema de los misioneros y caníbales.

En la ribera A de un río se hallan tres misioneros y tres caníbales, quienes desean trasladarse a la ribera B utilizando un bote. Puesto que en el bote sólo pueden viajar dos personas a lo sumo, es preciso realizar varios viajes; sin embargo, en ninguna de las riberas el número de caníbales debe exceder al de misioneros.

Se requiere encontrar una sucesión de viajes de un lugar a otro que finalmente permita transportar a todos a la ribera B.

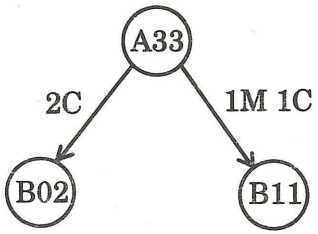
Este problema puede formularse como uno de búsqueda en el cual los estados son ternas de símbolos: R m n, en donde R indica la ribera en la que actualmente se halla el bote y m y n, los números de misioneros y caníbales, respectivamente, presentes allí.

Los estados sucesores de R m n se forman teniendo en cuenta los presentes en la ribera opuesta S y los posibles viajes y restricciones impuestas.



Los números de personas en estos estados se obtienen sumando los números de viajeros a los que actualmente se hallan en S: 3-m y 3-n .

Por ejemplo, los sucesores de A33 son:



Ahora el problema consiste en hallar una ruta que una los estados A33 y B33. (Véase ejemplo 2 de ejecución del programa BUS-RUT.C, Sec. 2.4.1, en donde se obtienen soluciones de este problema).

2.2 PROBLEMAS DE BUSQUEDA. METODOS

Como se ha indicado antes, en general el problema de búsqueda tiene por propósito encontrar rutas -o soluciones- de un estado C a otro F, llamado *objetivo* o *fin*.

Dependiendo del grado de precisión de la solución requerida, se presentan 3 casos de búsqueda: de *objetivos*, *rutas simples* y *rutas óptimas*. En el primer caso, sólo se exige determinar si existen o no rutas que parten de C y llegan a F; en el segundo caso, es necesario hallar rutas de C a F, y, en el último caso, además se requieren que éstas tengan un valor óptimo (máximo o mínimo) con respecto a una función dada.

Existen diversos métodos o procedimientos que se utilizan en la resolución de estos problemas. Para buscar objetivos y rutas simples se tienen los métodos de búsqueda:

a) en profundidad (o primero en profundidad),

b) en anchura (o primero en anchura),

c) escalamiento de la colina,

y d) primero el mejor

y para rutas óptimas:

e) ramificación y acotación.

Los métodos a), b) y c) se desarrollan en las secciones siguientes, y los restantes, en los ejercicios 4 y 5, Sec. 2.3.

2.2.1 BUSQUEDA EN PROFUNDIDAD Y EN ANCHURA

ALGORITMO I (Objetivos)

Para determinar si existe una ruta de C a F (estado objetivo) se procede así:

(p1) Se forman dos listas de estados COLA y VISITADOS, la primera con el estado C y la segunda sin estados:

COLA = (C)
VISITADOS = ( )

(p2)

a) Si COLA es vacía, se indica que no existe ruta y se termina el proceso.

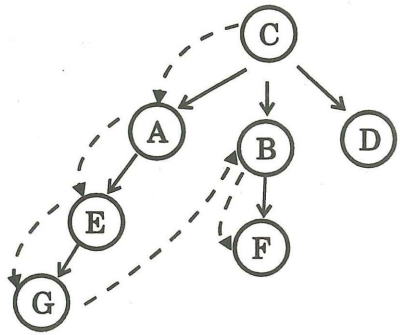
b) Si F es igual a P, el primer elemento de COLA, se indica que existe ruta y se concluye.

c) Se suprime P de COLA y se agrega a VISITADOS

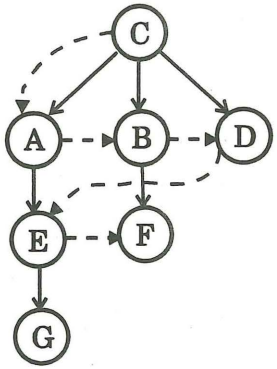
d) Se agregan a COLA por la izquierda (búsqueda en profundidad), o por la derecha (búsqueda en anchura), los sucesores de P que no pertenezcan a VISITADOS. Luego se repite (p2).

OBSERVACIONES

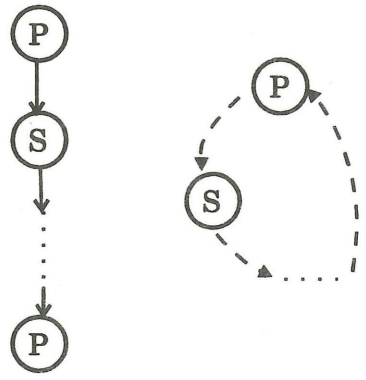
1. El algoritmo de búsqueda en profundidad busca a F "descendiendo" a través de estados sucesores:



En cambio, cuando la búsqueda es en anchura, se visitan los estados en forma "horizontal":



2. Mediante la lista de estados VISITADOS se impide el reingreso a COLA de aquellos estados que son descendientes de sí mismos o que forman rutas cerradas:

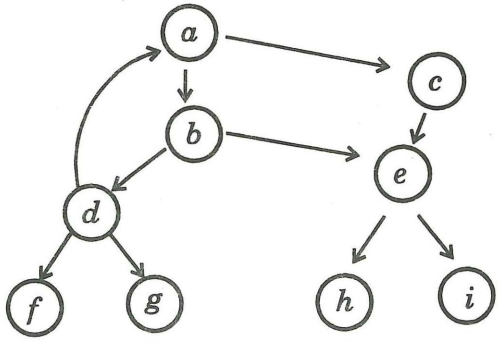


De lo contrario, en estos estados, el algoritmo se repetiría indefinidamente.

Si el espacio de búsqueda carece de esta clase de estados, es claro que la lista VISITADOS puede ser omitida.

EJEMPLO 1

Aplicando el algoritmo de búsqueda en profundidad determinamos si hay una ruta entre los estados a y h del espacio de búsqueda definido por el diagrama:



La siguiente tabla muestra los valores de COLA y VISITADOS al finalizar las sucesivas aplicaciones del procedimiento

	COLA	VISITADOS
1	(a)	()
2	(b, c)	(a)
3	(d, e, c)	(b, a)
4	(f, i, e, c)	(d, b, a)
5	(i, e, c)	(f, d, b, a)
6	(e, c)	(i, f, d, b, a)
7	(h, i, c)	

Explicamos algunos pasos:

PASO 1. Se inician los valores de las dos listas: COLA con el estado a y VISITADOS, sin elementos.

PASO 2. Al comienzo se tiene:

$$\text{COLA} = (a) \text{ y } \text{VISITADOS} = ( )$$

dados por el paso anterior.

Puesto que h es distinto de a, se desplaza a de COLA a VISITADOS y se añaden a COLA por la izquierda todos sus sucesores, b y c, pues ninguno se halla en VISITADOS. Así, se obtiene

$$\text{COLA} = (b,c) \ , \quad \text{VISITADOS} = (a)$$

PASO 4. Por el paso 3:

$$\text{COLA} = (d, e, c) \ , \quad \text{VISITADOS} = (b, a)$$

Como h es diferente de d, se desplaza el estado d a VISITADOS y se agregan a COLA por la izquierda los sucesores f e i, excluyendo al estado a por hallarse enVISITADOS. Luego:

$$\text{COLA} = (f, i, e, c) \ , \quad \text{VISITADOS} = (b, a)$$

PASO 7. El proceso termina indicando que existe una ruta entre a y h, pues el estado objetivo h es el primer elemento de COLA.

ALGORITMO II (Rutas)

El procedimiento para hallar una ruta entre dos estados C y F utiliza una lista COLA cuyos miembros R_1, ... , R_n son rutas que parten de C:

$$\text{COLA} = (R_1 \ , \ R_2 \ , \ \ldots \ , R_n)$$

en donde $R_i = C \ldots P_i$ une C con un estado P_i , al que llamaremos *destino* de R_i .

Los pasos a seguir son:

(p1) Se inicia COLA con la ruta compuesta únicamente por el estado C:

$$\text{COLA} = (C)$$

y la lista de estado VISITADOS, sin elementos:

VISITADOS = ( )

(p2)

(a) Si COLA no contiene rutas, se indica la ausencia de rutas y se termina el proceso

(b) Sea $R = C \ldots P$ la primera ruta de COLA.

Si $P = F$, entonces R es una ruta que une C con F y se concluye.

(c) Se suprime R en COLA y se agrega su destino P a VISITADOS.

(d) A partir de R se construyen rutas extendiendo P a cada sucesor $s_1, \ldots, s_m$ que no esté en VISITADOS:

$$E_1 = C \ldots P s_1$$

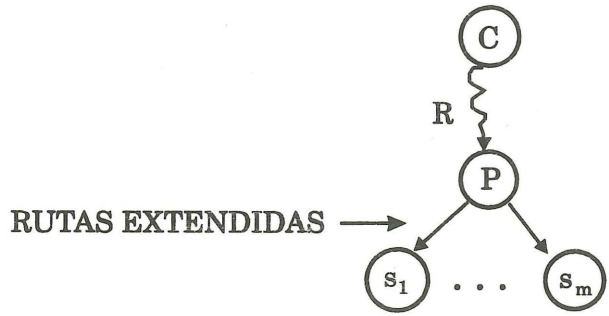
$$\ldots$$

$$E_m = C \ldots P s_m$$

(ver la figura siguiente)

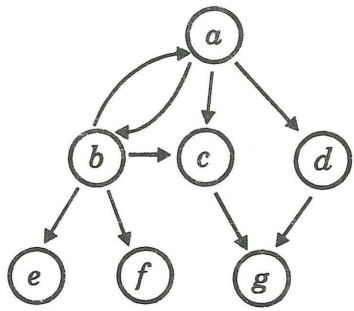
Y se añaden las rutas extendidas a COLA, por la izquierda (búsqueda en profundidad) o por la derecha (búsqueda en anchura).

Luego se repite (p2).



EJEMPLO 2

Aplicamos el método de búsqueda en anchura para hallar una ruta entre *a* y *e* del espacio de búsqueda



	COLA	P	SUC-P	VISITADOS
1	(*a*)	*a*	*b c d*	( )
2	(*ab**, *ac**, *ad**)	*b*	*e f c*	(*a*)
3	(*ac*, *ad*, *abe**, *abf**, *abc**)	*c*	*g*	(*b*, *a*)
4	(*ad*, *abe*, *abf*, *abc*, *acg**)	*d*	*g*	(*c*, *b*, *a*)
5	(*abe*, *abf*, *abc*, *acg*, *abg**)	*e*		(*d*, *c*, *b*, *a*)

En la tabla, P designa el estado de llegada de la primera ruta R_1 de COLA y SUC-P, los sucesores de P que no están en VISITADOS.

Las rutas señaladas por un asterisco, o "*", son las que se agregan a COLA por la derecha y que provienen de extender R_1 a cada estado de SUC-P.

En el paso 1, se asignan los valores iniciales a la lista COLA y VISITADOS.

En los pasos 1-4 se tiene P diferente de *e* y por lo tanto: Se añade P a VISITADOS, se suprime la primera ruta de COLA y se agregan a ésta las rutas extendidas.

El algoritmo termina en el paso 5, ya que se cumple P = *e*, y *abe* es la ruta encontrada.

2.2.2 METODOS DE RAMIFICACION Y ACOTACION

Asumiremos que cada ruta R del espacio de búsqueda tiene asociado un valor numérico v(R) y que estos valores satisfacen la siguiente condición:

v(R) es menor o igual que v(R*)

si R* es una ruta extendida de R, es decir si:

$$R = t^1 \ldots t^m$$

$$\text{y} \qquad R^* = t^1 \ldots t^m \, t^{m+1}$$

siendo t^{m+1} un sucesor de t^m.

En otras palabras, los valores de las rutas crecen o se mantienen iguales cuando se extienden.

Se dice que una ruta R_0 es óptima (de valor mínimo) entre dos estados C y F, si cumple:

(1) R_0 une C en F

y (2) $v(R_0) \leq v(R)$

para toda ruta R entre C y F.

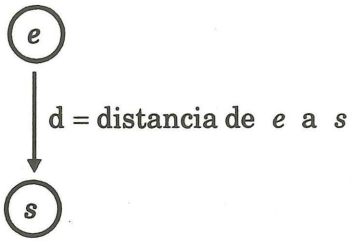
Dos ejemplos típicos de valores funciones de rutas son proporcionados por las longitudes (o distancias) y pesos de rutas.

Si $R = t^1 \, t^2 \ldots t^{m-1} \, t^m$ es una ruta arbitraria, se definen:

(1) $L(R) =$ longitud de R

$$= \begin{cases} 0 , & \text{si } m = 1 \\ d(t^1, t^2) + \ldots + d(t^{m-1}, t^m), & \text{si } m > 1 , \end{cases}$$

si son dados valores no negativos $d = d(e,s)$, distancia de e a s, para cada par (e,s) tal que s es un sucesor de e



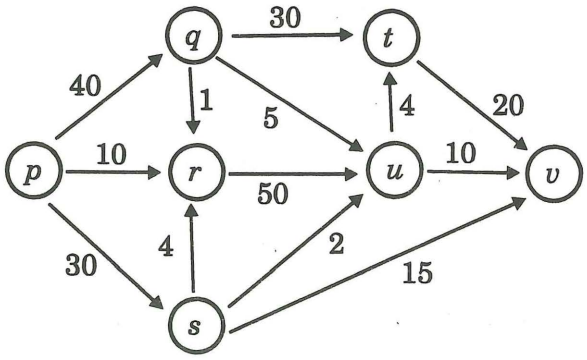
y (2) $P(R)$ = peso de R

$$= p(t^1) + \ldots + p(t^m),$$

si existen valores $p = p(e) \geq 0$, peso de e, para cada estado e.

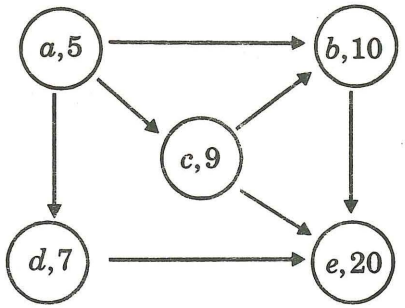
p = peso de e

EJEMPLO 1. Si



se tiene $L(p\,q\,u\,t\,v) = 40 + 5 + 4 + 20 = 69$, por ejemplo, y una ruta de p a v con longitud mínima 42 es $psuv$.

EJEMPLO 2. En el espacio de búsqueda con pesos de estados:



una ruta que une a con e y tiene peso mínimo 32 (= 5 + 7 + 20) es dada por ade.

ALGORITMO DE RAMIFICACION Y ACOTACION

El procedimiento para hallar una ruta de valor mínimo entre dos estados C y F, en un espacio de búsqueda con valores de rutas, utiliza una lista COLA cuyos miembros son rutas con sus respectivos valores:

$$\text{COLA} = (\, R_1 - v_1\,,\; R_2 - v_2\,,\; \ldots\,)$$

en donde v_i es el valor de la ruta R_i.

Los pasos del algoritmo son:

(p1) Se inicia COLA con la ruta C y su valor v_0:

$$COLA = (C - v_0)$$

y la lista VISITADOS sin elementos:

$$VISITADOS = ( )$$

(p2)

a) Si COLA es vacía, se indica la ausencia de rutas entre C y F y el proceso termina.

b) Sea $M = C \ldots P - v$ un miembro de COLA cuyo valor v es el menor posible.

c) Se suprime M en COLA y se agrega P a VISITADOS

Si $P = F$, entonces $C \ldots P$ es una ruta con valor mínimo v y se concluye

d) A partir de $C \ldots P$ se construyen rutas extendiendo P a cada sucesor $s_1, \ldots, s_m$ que no esté en VISITADOS y se añaden a COLA los nuevos objetos

$$C \ldots P s_1 - v_1$$
$$\ldots$$
$$C \ldots P s_m - v_m$$

en donde v_i es el valor de la ruta extendida $C \ldots P s_i$

A continuación se repite (p2)

NOTA

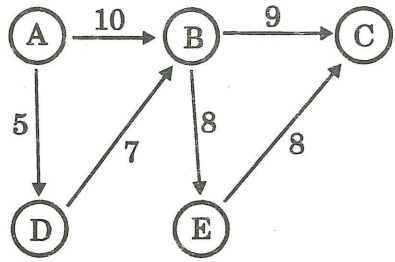
En los casos de valores de longitudes y pesos de rutas, los valores v_i se calculan durante la ejecución del algoritmo.

Así para longitudes, se empieza con $COLA = (C - 0)$ y $v_i = v + d(P, s_i)$

y si se trata de pesos, se inicia con $COLA = (C - p_0)$, $p_0 = p(C) =$ peso de C y $v_i = v + p(s_i)$

EJEMPLO 3

Determinamos una ruta de longitud mínima entre A y E del espacio de búsqueda:



Los sucesivos valores de las listas COLA y VISITADOS son:

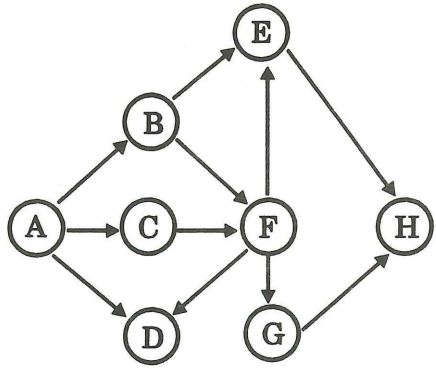
	COLA	VISITADOS
1	A – 0 *	-
2	AB – 10, AD – 5 *	A
3	AB – 10*, ADB – 12	A, B
4	ADB * –12*, ABC – 19, ABE – 18	A, B
5	ABC – 19, ABE – 18*, ADBC – 21, ADBE – 20	A, B, E

En cada paso se elimina la ruta mínima -marcada por "*"- de la lista COLA y se añaden (por la derecha) las rutas extendidas.

El algoritmo termina en el paso 5 pues la ruta marcada ABE alcanza el estado E, y el valor óptimo es 18.

2.3 EJERCICIOS

1. Dado el espacio de búsqueda:

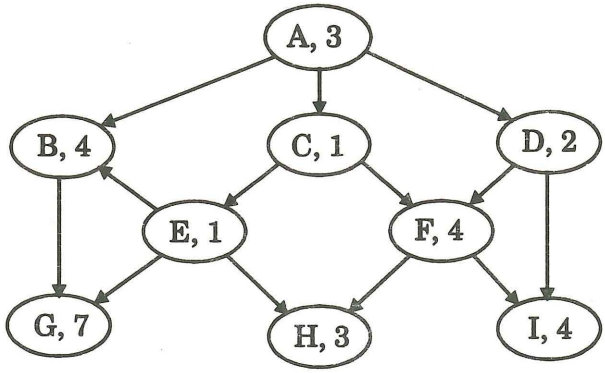


determine si existe una ruta entre C y H aplicando búsqueda.

a) en profundidad
b) en anchura

2. Utilice el algoritmo de búsqueda en profundidad para hallar una ruta entre A y G del espacio de búsqueda del ejercicio 1.

3. Aplicando el algoritmo de ramificación y acotación, encuentre una ruta de peso mínimo entre los estados A y H del espacio de búsqueda:



4. METODO DE BUSQUEDA: Escalamiento de la colina

Este método se utiliza para determinar si existe una ruta entre dos estados C y F de un espacio de búsqueda en el cual hay valores de distancia d(*e*,*s*) para cada par (*e*,*s*) tal que *s* es un sucesor de *e*.

Los pasos del algoritmo son:

(p1) Se inicia la lista COLA con el estado C y la lista VISITADOS sin elementos.

(p2)

a) Si COLA es vacía, se indica la ausencia de rutas entre C y F y se termina.

b) Sea P el primer elemento de COLA. Si P = F, se indica la existencia de rutas y el algoritmo termina.

c) Se suprime P de COLA y se agrega a VISITADOS.

d) Se hallan los sucesores de P que no estén en VISITADOS y se ordena por la distancia en forma creciente:

$$s_1 \; s_2 \; \ldots \; s_k$$

de modo que se cumple

$$d(P,s_1) \leq d(P,s_2) \leq \ldots \leq d(P_1, s_k)$$

Luego se añaden éstos a COLA por la izquierda manteniendo dicho orden:

$$COLA = (s_1 \; s_2 \; \ldots \; s_k \; \ldots)$$

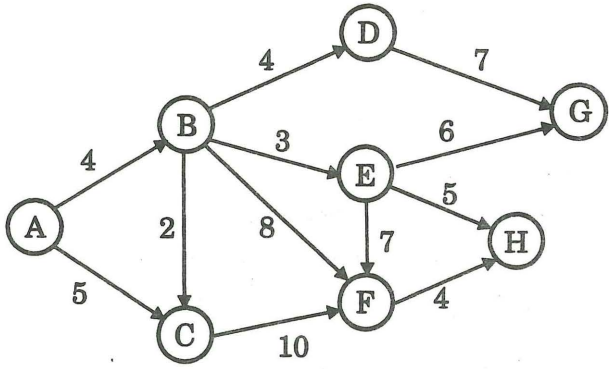
e) Se repite (p2)

NOTA

El procedimiento indicado busca a F a partir de C progresando a través de los sucesores más cercanos.

APLICACION

Utilice el algoritmo indicado para determinar si existe o no una ruta entre A y H del espacio:



5. METODO DE BUSQUEDA: Primero el mejor

El presente método constituye una versión simplificada del algoritmo de ramificación y acotación expuesto en 2.2.2, pues sólo trata de determinar si existe una ruta entre dos estados C y F de un espacio de búsqueda provisto de distancias.

En este caso la lista COLA en lugar de rutas parciales contiene únicamente los estados finales de éstas. Así, COLA es de la forma

$$(P_1 - v_1, P_2 - v_2, \ldots)$$

en donde P_i es un estado descendiente de C y v_i es el valor de la distancia acumulada desde C hasta P_i.

Inicialmente COLA consiste sólo del par C – O.

Modificando 2.2.2 escriba los pasos del nuevo algoritmo y determine si hay una ruta de B a D en el diagrama del ejercicio anterior.

2.4 EJEMPLOS DE PROGRAMAS DE BUSQUEDA ESCRITOS EN LENGUAJE C

A continuación se muestran dos programas en lenguaje C: BUSC-RUT.C y BUSC-MIN.C, para resolver problemas de búsqueda de rutas y rutas de distancia mínima, respectivamente.

En el primer programa, el ingreso de los datos se realiza en la forma:

nodo suc_1 suc_2 ... ;

para cada nodo y su lista de sucesores, con un punto y coma al final de cada lista.

El ingreso de estos datos termina con un asterisco:

*

Luego se leen los nodos de partida y llegada y el método de búsqueda (A = en anchura, o P = en profundidad).

Los datos se ingresan por el teclado, respondiendo a las preguntas, o bien desde un archivo XXX dando la orden:

BUSC - RUT < XXX

En el segundo programa, BUSC-MIN.C, no se presentan mensajes para el ingreso de datos. Para evitar errores se recomienda que los datos se encuentren en un archivo XXX en la forma:

nodo suc_1 d_1 suc_2 d_2 ... ;

...

*

nodo_de_partida

nodo_de_llegada

en donde:

d_i representa la distancia de suc_i a su nodo (predecesor)

* indica fin de lista de nodos

Si se usa un archivo XXX con estos datos, la manera de correr el programa es:

BUSCAR - MIN < XXX

NOTA

Si no se desea utilizar un archivo, la lectura de datos también puede hacerse desde el teclado siempre que se observen las reglas anteriores.

En todos los programas se emplean:

1) un arreglo TabNodo, cuyos componentes TabNodo[0], ..., TabNodo[i], ..., contienen los nombres (máximo 13 caracteres) de los distintos nodos.

2) un arreglo LSuc, en el cual se hallan los grupos de los (índices a TabNodo de los) sucesores de cada nodo. El final de cada grupo es marcado con el valor NODATO (igual a -1).

3) un arreglo IndxSuc de tipo int, cuyo componente IndxSuc[i] es un entero que indica la posición del primer sucesor del nodo TabNodo[i] en LSuc.

Una manera de visualizar estos objetos es la siguiente:

Supongamos que LSuc[9] es el último valor definido de LSuc y que cuando se ingresa una lista como:

LIMA SANTIAGO BOGOTA ;

estos valores se almacenan en las posiciones 3, 7 y 12 del arreglo TabNodo:

```
TabNodo[3] ... TabNodo[7] ... TabNodo[12]
"LIMA"     "SANTIAGO"     "BOGOTA"
```

entonces los índices 7, 12 y -1 (marcador de fin de grupo) se salvan en LSuc[10], LSuc[11] y LSuc[12], respectivamente, y se asigna 10 a IndxSuc[3].

Así, la lista de los sucesores de "LIMA", de índice 3, empieza en la posición 10, dada por IndxSuc[3], de la lista LSuc y continúa hasta antes de -1, esto es, la lista de índices es:

7, 12.

También se emplean dos arreglos que representan las listas:

(4) Cola

y (5) Visitados

de acuerdo a los algoritmos descritos en este capítulo.

Con el propósito de construir rutas, estos arreglos se forman con cada nodo de prueba y (un índice a la lista visitados de) su nodo predecesor.

De este modo, cuando se encuentra el nodo de llegada, se puede obtener la ruta simplemente buscando el nodo predecesor a partir del nodo de llegada, luego el predecesor del anterior y

así sucesivamente, hasta llegar al nodo de partida (cuyo predecesor inicialmente se ha hecho igual a NODATO). La ruta resultante tiene los nodos en orden invertido por lo cual es necesario que éstos se reordenen.

Una ventaja del presente método se refiere a la minimización del tamaño de los datos, pues los arreglos (4) y (5) no se componen de rutas (con todos sus nodos), sino más bien de nodos que se enlazan con su predecesor actual.

2.4.1 PROGRAMA BUSC-RUT.C

```
/* programa de busqueda de rutas:
        A (en anchura) o P (en profundidad) */

#include <stdio.h>
#include <string.h>


#define LongNombre 14
#define MaxNodo   50
#define MaxSuc   200
#define NoDato    -1


typedef char tiponombre[LongNombre];
tiponombre TabNodo[MaxNodo];

int      ContNodo=0;
int      IndxSuc[MaxNodo];
int      LSuc[MaxSuc];
```

```
int       ContSuc=0;
int       Npartida, Nllegada;
char      ModoBusqueda;
typedef struct { int inodo, pred; } tipocola;
tipocola   Cola[MaxSuc];
tipocola   *Comienzo, *MarcaFin;
tipocola   LVisitados[MaxSuc], *pFVis;


main()
{  lee_datos();
   if (ModoBusqueda=='A') buscarA();
      else if (ModoBusqueda=='P') buscarP();
}

lee_datos()
{ tiponombre  temp; int i,c;

      /* inicia tablas */
 for (i=0;i<MaxNodo;i++) IndxSuc[i]=NoDato;
 for (i=0;i<MaxSuc;i++)  LSuc[i]=NoDato;
 while (1)
 {
   printf("Nodo y Sucesores > "); scanf("%s",temp);
   if (strcmp(temp,"*")==0)  break;
   i=loc_nodo(temp);    IndxSuc[i]=ContSuc;

   while (1)
    { scanf("%s",temp);
      if (strcmp(temp,";")==0) { ContSuc++; break;}
      i=loc_nodo(temp);  LSuc[ContSuc++]=i;
    }
```

```
  }

  printf("Nodo de partida "); scanf("%s",temp);
  Npartida=loc_nodo(temp);

  printf("Nodo de llegada "); scanf("%s",temp);
  Nllegada=loc_nodo(temp);
  printf("Metodo de busqueda  (A P)  ? ");
  scanf("%s",&c); c=toupper(c);    ModoBusqueda=c;
}


loc_nodo(n)
char *n;
{ int i;
  for (i=0;i<ContNodo;i++) if (strcmp(TabNodo[i],n)==0) return i;

     /* agregar  nuevo dato */
  strcpy(TabNodo[ContNodo],n);  return ContNodo++;
}


buscarA()

{ int Primero,s,i,pred;
  Comienzo=Cola; MarcaFin=Cola+1;
  Comienzo->inodo=Npartida; Comienzo->pred=NoDato;
  pFVis=LVisitados;
  while (1)
  { if (Comienzo==MarcaFin)
      { printf("\n*** No existe ruta!\n"); return; }
```

```
    pFVis->inodo=Primero=Comienzo->inodo;
        /* agrega a LVisitados */
    pFVis->pred =Comienzo->pred;

    if (Primero==Nllegada) { muestra_ruta(); return; }

    pred=pFVis - LVisitados;            /* marca predecesor */
    pFVis++;

    Comienzo++;                         /* suprime primer dato */

    /* agregar sucesores de Primero al final de la cola */

    i=IndxSuc[Primero];
    if (i==NoDato) continue;

    while (1)
     {  s=LSuc[i++];
        if (s==NoDato) break;
        if (no_visitados(s))
       { MarcaFin->inodo=s;  MarcaFin->pred=pred; MarcaFin++; }
     }

  }

}



buscarP()
```

```
{ int Primero,s,i,pred, j;
  MarcaFin=Cola+MaxSuc; Comienzo=MarcaFin-1;
  Comienzo->inodo=Npartida; Comienzo->pred=NoDato;
  pFVis=LVisitados;

  while (1)

  { if (Comienzo==MarcaFin)
     { printf("\n*** No existe ruta!\n"); return; }

    pFVis->inodo=Primero= Comienzo->inodo;
                    /* agrega a LVisitados */
    pFVis->pred =Comienzo->pred;
    if (Primero==Nllegada) { muestra_ruta(); return; }

    pred=pFVis - LVisitados;          /* marca predecesor */
    pFVis++;
    Comienzo++;                      /* suprime primer dato */

    /* agregar sucesores de Primero al comienzo de la cola */

    i=IndxSuc[Primero];
    if (i==NoDato) continue;
    j=i;  while (LSuc[j++]!=NoDato) ;
    while (1)
     {
      if (--j <i) break;
      s=LSuc[j];
      if (no_visitados(s))
       { --Comienzo; Comienzo->inodo=s; Comienzo->pred=pred;
        }
```

```
    }

  }
}

no_visitados(s)
int s;
{ tipocola *p;
  for (p=LVisitados; p<pFVis;p++) if (s==p->inodo) return 0;
  return 1;
}


muestra_ruta()
{ int ruta[MaxSuc], i,n,s,m; tipocola *p;

  p=pFVis; n=0;
  while (1)
   { ruta[n]=p->inodo;     s=p->pred;
     if (s==NoDato) break;
     p=LVisitados+s;    n++;
   }


  /* reordena datos de ruta */
  m=(n+1)/2;
  for (i=0;i<m;i++)
    { s=ruta[i]; ruta[i]=ruta[n-i]; ruta[n-i]=s; }

  printf("\n*** Ruta encontrada : \n\t");
  for (i=0;i<=n;i++)
    { s=ruta[i];  printf("  %s", TabNodo[s]); }

}
```

EJEMPLOS DE EJECUCION DE BUSC-RUT

(1)

```
Nodo y Sucesores > a b c d ;
Nodo y Sucesores > b e d;
Nodo y Sucesores > *
Nodo de partida  a
Nodo de llegada  d
Método de Búsqueda (A P) ? P
*** Ruta encontrada:
            a b d
```

(2) Para hallar una ruta del problema de los misioneros y caníbales expuesto en el ejemplo 2 de la sección 2.1:

```
Nodo y Sucesores > A33  B11  B02 ;
Nodo y Sucesores > A32  B11  B02  B03 ;
Nodo y Sucesores > A31  B22  B02 ;
Nodo y Sucesores > A22  B22  B31 ;
Nodo y Sucesores > A11  B32  B33 ;
Nodo y Sucesores > A03  B31  B32 ;
Nodo y Sucesores > A02  B32  B33 ;
Nodo y Sucesores > A01  B33 ;
Nodo y Sucesores > B33  A11  A02 ;
Nodo y Sucesores > B32  A11  A02  A03 ;
Nodo y Sucesores > B31  A22  A02 ;
Nodo y Sucesores > B22  A22  A31 ;
Nodo y Sucesores > B11  A32  A33 ;
Nodo y Sucesores > B03  A31  A32 ;
Nodo y Sucesores > B02  A32  A33 ;
```

```
Nodo y Sucesores > B01  A33 ;
Nodo y Sucesores > *
Nodo de partida A33
Nodo de llegada B33
Método de Búsqueda (A P) ? P
*** Ruta encontrada :
A33 B11 A32 B03 A31 B22 A22 B31 A02 B32 A11 B33
```

Si se corre otra vez el programa con los mismos datos pero esta vez aplicando el método de búsqueda A se obtiene el siguiente resultado:

```
*** Ruta encontrada:
A33 B11 A32 B03 A31 B22 A22 B31 A02 B33
```

2.4.2 PROGRAMA BUSC-MIN.C

```
/* programa de busqueda de rutas de  distancia minima */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define LongNombre 14
#define MaxNodo   50
#define MaxSuc   200
#define NoDato    -1


typedef  char tiponombre[LongNombre];
tiponombre  TabNodo[MaxNodo];
```

```
int       ContNodo=0;
int       IndxSuc[MaxNodo];
struct    { int suc, distApred; }  LSuc[MaxSuc];
int       ContSuc=0;
int       Npartida, Nllegada;
typedef struct { int inodo, pred, dist; } tipocola;
tipocola   Cola[MaxSuc];
tipocola   *Comienzo, *MarcaFin;
tipocola   LVisitados[MaxSuc], *pFVis;


main()
{ lee_datos(); buscarMin();
}


lee_datos()
{ tiponombre temp ; int i,d;


      /* inicia tablas */
 for (i=0;i<MaxNodo;i++) IndxSuc[i]=NoDato;
 for (i=0;i<MaxSuc;i++)
   { LSuc[i].suc=NoDato; LSuc[i].distApred=0; }

 while (1)
 {
   scanf("%s",temp);
  if (strcmp(temp,"*")==0)  break;
  i=loc_nodo(temp);    IndxSuc[i]=ContSuc;

  while (1)
   { scanf("%s",temp);
```

```
      if (strcmp(temp,";")==0) { ContSuc++; break;}
      scanf("%d",&d);
      i=loc_nodo(temp);
       LSuc[ContSuc].suc=i; LSuc[ContSuc++].distApred=d;
    }

  }

  scanf("%s",temp);  Npartida=loc_nodo(temp);
  scanf("%s",temp);  Nllegada=loc_nodo(temp);

}

loc_nodo(n)
char *n;
{ int i;
  for (i=0;i<ContNodo;i++) if (strcmp(TabNodo[i],n)==0) return i;

          /* agregar  nuevo dato */
  strcpy(TabNodo[ContNodo],n);
  return ContNodo++;
}

buscarMin()

{ int Primero,s,i,pred,dist;
  int fminorden();

  Comienzo=Cola; MarcaFin=Cola+1;
  Comienzo->inodo=Npartida; Comienzo->pred=NoDato;
  Comienzo->dist = 0;
  pFVis=LVisitados;
  while (1)
  { if (Comienzo==MarcaFin)
```

```
  { printf("\n*** No existe ruta!\n"); return; }

pFVis->inodo=Primero=Comienzo->inodo;
   /* agrega a  LVisitados */
pFVis->pred =Comienzo->pred;
pFVis->dist =Comienzo->dist;

if (Primero==NIlegada) { muestra_ruta(); return; }

pred=pFVis - LVisitados;              /* marca predecesor */
dist=pFVis->dist;
pFVis++;

Comienzo++;                    /* suprime primer dato */

/* agregar sucesores de Primero a cola */

i=IndxSuc[Primero];
if (i==NoDato) continue;

while (1)
 {  s=LSuc[i].suc;
     if (s==NoDato) break;

     if (no_visitados(s))
      {  MarcaFin->inodo=s;  MarcaFin->pred=pred;
         MarcaFin->dist = dist + LSuc[i].distApred;
         MarcaFin++;
      }
     i++;
 }

/* ordena cola en orden ascendente segun dist
desde   numero de datos   long de cada dato  func. de ord. */
```

```
    qsort(Comienzo, MarcaFin-Comienzo, sizeof (tipocola) ,fminorden);

  }

}

fminorden(pa,pb)
tipocola *pa, *pb;
{ return pa->dist - pb->dist; }

no_visitados(s)
int s;
{ tipocola *p;
  for (p=LVisitados; p<pFVis;p++)  if (s==p->inodo) return 0;
  return 1;
}

muestra_ruta()
{ int ruta[MaxSuc], i,n,s,m; tipocola *p;
  int d[MaxSuc];
  p=pFVis; n=0;
    /* determina ruta invertida y distancias acumuladas */
  while (1)
   { ruta[n]=p->inodo; d[n]=p->dist;   s=p->pred;
     if (s==NoDato) break;
     p=LVisitados+s;    n++;
   }
     /* calcula distancias entre nodos a partir de dist. acum. */
  for (i=1;i<=n;i++)  d[i-1] -=d[i];

     /* obtener ruta directa invirtiendo datos  */

  m=(n+1)/2;
```

```
    for (i=0; i<m;i++)
     { s=ruta[i]; ruta[i]=ruta[n-i]; ruta[n-i]=s;
       s=d[i];   d[i]  =d[n-i];   d[n-i]  =s;
     }

    printf("\n*** Ruta encontrada : \n\n\t");
    for (i=0;i<=n;i++)
    { s=ruta[i];  printf(" %s [=%d] ", TabNodo[s],d[i]);
      if (i<n)    printf(" --> ");
    }

    printf("\n\n\*** Distancia Minima =%d\n",pFVis->dist);

  }
```

EJEMPLO DE CORRIDA

El archivo DATOS se compone de:

```
a b 10  c 8  d 5 ;
b  e 5 ;
c  e 7  f 1 ;
d  f 2 ;
e  g 1 ;
f  g 40 ;
*
a
g
```

Al ejecutar

```
BUSC-MIN <DATOS
```

se obtiene:

*** Ruta encontrada:

a [=0] --> c [=8] --> e [=7] --> g [=1]

*** Distancia Mínima = 16

en donde [D] indica la longitud o distancia D del nodo actual al predecesor.

Capítulo 3

Sistemas Expertos

3.1 SISTEMAS DE RESOLUCION DE PROBLEMAS

La resolución de problemas constituye un tópico central en la Inteligencia Artificial. No obstante que muchos de estos problemas pueden ser tratados como problemas de búsqueda, las características y condiciones especiales de los mismos conducen a analizarlos de una manera más precisa y desarrollar sus propios métodos de solución.

Dos clases importantes de sistemas que resuelven problemas son las siguientes:

(1) Sistemas basados en reglas, llamados también sistemas de producción o sistemas expertos

y (2) sistemas de generación y prueba.

En este capítulo trataremos los sistemas basados en reglas y en el siguiente, los sistemas de generación y prueba.

3.2 SISTEMAS EXPERTOS

Un *sistema experto* consiste de:

(1) Un conjunto de datos (o hechos) llamado *base de datos* o *memoria de trabajo*

(2) un conjunto de reglas, frecuentemente denominado *memoria de producción* o *reglas de producción*

y (3) un *intérprete*, que es un programa que decide cómo, cuándo y cúales reglas de producción se aplican.

Adicionalmente, algunas veces se requiere que el sistema proporcione información y explicaciones sobre las reglas aplicadas.

La memoria de trabajo MT es una colección o lista de datos de la forma:

$$\begin{aligned} MT &= [\ (\text{objeto propiedad valor}) \ldots\ ] \\ &= \{\ \text{propiedad}(\text{objeto}, \text{valor}), \ldots\ \} \end{aligned}$$

o por listas o arreglos que designan propiedades y cuyos miembros son los objetos que cumplen las mismas:

$$\begin{aligned} &\text{propiedadA} = (\ a\ \ b\ \ \ldots) \\ &\text{propiedadB} = (\ x\ \ y\ \ \ldots) \\ &\ldots \end{aligned}$$

En un programa particular estos datos pueden estar distribuidos en varios conjuntos o listas.

Las reglas de producción $R_1, R_2, \ldots, R_n$, se expresan usualmente así:

R_i: si P_i entonces Q_i

o R_i: Q_i si P_i

en donde P_i se llama *premisa* o *condición* y Q_i, *conclusión* o *acción*, y significa que si se cumple o es verdadera P_i, entonces se cumple o debe ejecutarse Q_i

Se dice que R_i se *satisface* si P_i se cumple.

NOTA

Tanto la premisa como la conclusión pueden ser expresiones que incluyen los conectivos lógicos: "no" (negación), "y" (conjunción), "o" (disyunción). Por ejemplo:

$$R_i\text{: si } \underbrace{(\text{A "y" B}) \text{ "o" C}}_{P_i} \text{ entonces } \underbrace{\text{D "y" ("no" E)}}_{Q_i}$$

cuya interpretación es:

si se cumplen A y B a la vez, o C,

entonces se cumplen tanto D como la negación de E.

La representación de las reglas en los programas puede hacerse de varias maneras:

(1) mediante listas como $(R_i\ P_i\ Q_i)$

(2) por funciones asociadas a los objetos R_i , P_i y Q_i

o (3) utilizando un espacio de búsqueda con estados o nodos P_i , Q_i , en donde se establece que Q_i es un sucesor de P_i

3.3 ACCION DEL INTERPRETE

Básicamente el intérprete realiza un problema de búsqueda:

Dado un subconjunto de reglas objetivos RO = $\{ R_a , R_b , \ldots \}$, trata de determinar cuáles de ellas se satisfacen o cumplen.

La acción del intérprete depende tanto de la representación de las reglas como del método que se elija.

Si se asume una representación de espacio de búsqueda, simplemente el intérprete recorre los nodos del espacio hasta encontrar los nodos objetivos.

Para las dos primeras clases de representaciones de reglas el intérprete puede actuar según uno de los métodos siguientes:

1) encadenamiento hacia atrás

o 2) encadenamiento hacia adelante.

3.3.1 ENCADENAMIENTO HACIA ATRAS

Este método parte de la conclusión de cada de regla RO - asume que la regla se cumple- y procede a comprobar si su respectiva condición es verdadera.

Los pasos que se siguen aquí son:

(p1) Si RO es vacío el proceso termina

(p2) Se suprime la primera regla R_x de RO y se comprueba si la condición P_x es verdadera o falsa. En el primer caso se indica que R_x se cumple y se aplica, o se salva en MT, su conclusión Q_x. Luego se repite (p1).

3.3.2 ENCADENAMIENTO HACIA ADELANTE

El método utiliza los hechos dados por MT para deducir nuevos datos procediendo desde las premisas a las conclusiones.

Los pasos son:

(p1) Si RO es vacío, termina el proceso

(p2) Si hay reglas de RO cuyas condiciones son ciertas, se elige una de ellas R_x (resolución de conflicto), se aplica su conclusión Q_x o se actualiza MT con Q_x, se indica que R_x se satisface y se suprime de RO. Luego se procede al paso (p1).

(p3) Si hay reglas de RO cuyas condiciones son falsas, se suprimen de RO y se va al paso (p1).

NOTA

1. Si hay reglas de RO cuyas condiciones son indeterminadas, esto es, con los datos actuales de MT no es posible comprobar si sus condiciones son verdaderas o falsas, el intérprete puede requerir el ingreso de

más información para incorporarla a la memoria de trabajo MT y proceder otra vez al paso (p1).

2. En el caso especial de una regla R_x para la que se conoce que la premisa P_x y la conclusión Q_x son equivalentes, esto es, que ambos objetos P_x y Q_x tienen los mismos valores de verdad o falsedad, puede modificarse el último paso de los dos métodos desarrollados a fin de salvar Q_x en MT con el valor falso cuando la condición P_x sea falsa.

3. Algunas aplicaciones utilizan valores numéricos o pesos para establecer un cierto orden de prioridad en la elección de las reglas a aplicar y también valores de probabilidades de ocurrencia asociados a las conclusiones. De esta manera, los sistemas pueden incorporar criterios de preferencia y medidas de incertidumbre.

3.4 CLASES DE SISTEMAS EXPERTOS

Teniendo en cuenta la forma en que cambia la memoria de trabajo MT, los sistemas expertos se clasifican en sistemas de síntesis o de análisis.

En los sistemas de síntesis no cambian la memoria de trabajo ni el conjunto de reglas. Se dice que estos sistemas "no aprenden".

Ejemplo: Sistema de configuración.

Los sistemas de análisis, denominados también sistemas deductivos, durante su ejecución producen nuevos hechos que se añaden a la memoria de trabajo.

Ejemplo: Sistemas de identificación de objetos.

3.5 PROGRAMAS DE TIPO SHELL O CASCARA

Un programa que resuelve problemas de sistemas basados en reglas puede construirse exclusivamente para una aplicación determinada, esto es, para un solo conjunto de reglas, o bien para aplicaciones diferentes.

En el primer caso, a cada regla R: Si P entonces Q, y componentes P, Q, se asocia una función o subprograma:

```
función R
comienzo  si P entonces Q  fin
```

En el segundo caso, el programa se construye mediante un procedimiento de selección de reglas que es independiente de las mismas. Durante la ejecución del programa, se lee un conjunto arbitrario de reglas (generalmente escritas en un archivo) que se almacenan como datos en la memoria y luego se les aplica el procedimiento indicado. La generación de esta clase de programas, llamados programas "shells" o "cáscaras", resulta

muy conveniente pues pueden aplicarse a diversas situaciones. Los ejemplos que se muestran en la siguiente sección corresponden a programas de tipo cáscara.

3.6 EJEMPLOS DE SISTEMAS EXPERTOS ESCRITOS EN C

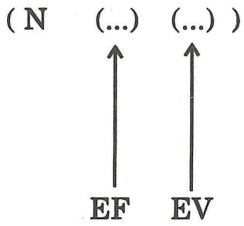
EJEMPLO 1. PROGRAMA ID-BUSC.C

Este programa identifica objetivos usando un método de búsqueda en árboles binarios.

Se ejecuta en la forma:

ID-BUSC archivo-reglas

en donde el archivo de reglas se compone de reglas expresadas mediante un árbol binario:



en donde N es un nombre formado por una o varias palabras y EF (y EV) puede ser a su vez de la misma forma:

(N (...) (...))

o contener solamente a un nombre: (N), en cuyo caso se dice que N es un nodo terminal u objetivo.

El significado de la representación es: Si N es falso, se sigue EF; y si N es verdadero, se sigue EV

EJEMPLOS DE ARBOLES BINARIOS

1. (A (B) (C))

que representa:

Si A entonces C; de lo contrario B

2. (A (B (X) (Y)) (C))

Por ejemplo, si A es falso se sigue:

(B (X) (Y))

y si además B es verdadero: (Y)

Por tanto, se llega al objetivo Y si A es falso y B es verdadero.

Presentamos ahora el archivo ANIMAL.DAT que contiene un árbol binario representando las reglas para identificar un conjunto de animales:

```
(TIENE ESPINAZO
  (PUEDE VOLAR   (GUSANO)   (MOSQUITO) )

  (TIENE SANGRE CALIENTE
   (TIENE AGALLAS Y VIVE EN AGUA
    (AL PRINCIPIO TIENE AGALLAS Y DESPUES RESPIRA
```

```
            (TIENE PIERNAS  (SERPIENTE)   (CROCODILOS) )
            (SAPO)
          )
          (ATUN)
        )
        (AMAMANTA
          (PUEDE VOLAR  (POLLO)  (PETIRROJO) )

          (VIVE EN AGUA
            (ES UN ANIMAL DOMESTICO  (TIGRE) (PERRO) )
            (DELFIN)
          )

        )

      )

    )
```

El texto del programa es:

```
/* Programa ID-BUSC.C, identificador de objetos
   en árboles binarios.
   Recorre el árbol procediendo desde la raíz hasta los nodos
   terminales u objetivos */


#include <stdio.h>
#include <ctype.h>
#define MAX 3000
char texto[MAX],*pt,*marcafin;
int tam;
```

```
FILE * archivo;

main(n,pal)
int n ;
char  *pal[ ];
{
  if (n==2)
    { archivo = fopen(pal[1],"rt");
      if (archivo==NULL) error("No se encuentra archivo");
    }

  tam=fread(texto,1,MAX-1,archivo);
  if (!eof(archivo)) error("Archivo muy grande");
  marcafin=texto+tam;
  fclose(archivo);
  recorre_arbol();
}


respuesta_Si(n)
char *n;
{ int c;

 printf(" %s (S/N) ? ",n);
 while (1)
 { c=getch();
   if (c=='S' || c=='s' ||  c=='N' || c=='n') break;
 }
 if (c>='a') c -=32;
 printf("%c\n",c);      return  c=='S';
```

```
}

 recorre_arbol()
 { char condicion[80],*pt;
   int izq, der;
   pt=texto;
   while (descompone(pt,condicion,&izq,&der))
   {
    if (respuesta_Si(condicion)) pt=texto+der;
    else { pt=texto+izq; marcafin=texto+der; }
   }
   printf("\n*** Objeto puede ser ");
   if(!respuesta_Si(condicion))
      if (respuesta_Si("Desea agregar objeto"))
                    agregar(condicion,izq,der);

}


error(m)
char *m;
{ printf("\nError: %s\n",m); exit(0); }


descompone(pt,condicion,pizq,pder)
char *pt,*condicion;
int *pizq, *pder;
{ int npar;
  *pizq=pt-texto;
  while (*pt!='(') pt++;
  pt++;
  while (*pt!='(' && *pt!=')')  *condicion++ = *pt++;
```

```
  condicion--;
  while (isspace(*condicion)) condicion--;

  ++condicion; *condicion=0;
  if (*pt==')') {*pder=pt-texto; return 0; }
  *pizq=pt-texto; pt++; npar=1;
  while (npar>0)
   { if (pt>marcafin) error("Parentesis no concuerdan");
     if (*pt=='(') npar++;
     else if (*pt==')') npar--;
     pt++;
   }
  *pder=pt-texto;
  return 1;
}

agregar(condicion,i,d)
char *condicion; int i,d;

{ char nombre[30] ,caract[80],  narch[40];

   printf("Ingrese nombre de objeto : ") ; gets(nombre);
   printf("Indique una caracterista que lo distinga : ");
   gets(caract);
   printf("Nombre de archivo con nuevos datos "); gets(narch);
   archivo=fopen(narch,"wt");
   fwrite(texto,1,i,archivo);
   fprintf(archivo,"(%s (%s) (%s) )",caract,condicion,nombre);
   fwrite(texto+d,1, tam -d-1,archivo);
   fclose(archivo);
 }
```

EJEMPLO DE CORRIDA

```
ID-BUSC  ANIMAL.DAT
TIENE ESPINAZO (S/N) ? S
TIENE SANGRE CALIENTE (S/N) ? S
AMAMANTA (S/N) ? N
PUEDE VOLAR (S/N) ? S

***Objeto puede ser PETIRROJO (S/N) S
```

NOTA

Este programa es un sistema experto de tipo cáscara. Puede aplicarse a otro sistema de objetos escribiendo en un archivo las reglas correspondientes en forma similar a las del archivo ANIMAL.DAT.

Si se responde con N en la última pregunta:

```
***Objeto puede ser ... (S/N) N
```

el programa permite agregar un nuevo objeto a la colección actual para lo cual solicita tres datos: el nombre del objeto, una característica que lo distinga y el nombre del archivo que contendrá la colección ampliada de objetos.

EJEMPLO 2. PROGRAMA IDENT.C

A continuación presentamos otro programa identificador de objetos con un intérprete de tipo cáscara. A diferencia del anterior, en este caso las reglas se expresan de una manera más sencilla:

$$(C\ p_1\ p_2\ \ldots\ p_n)$$

en donde C y p_i son nombres formados por una sola palabra o varias palabras encerradas entre comillas.

El significado de la regla es: la conclusión C es verdadera si todas las premisas p_i son verdaderas, esto es: si p_1 y p_2 ... y p_n entonces C. Así cada lista o regla es de tipo "y" o conjunción de premisas cuyo primer elemento es la conclusión.

Opcionalmente, cuando se requiera, se puede escribir $!p_i$ en lugar de p_i, con el símbolo ! para indicar que C depende de la negación de p_i.

El programa IDENT.C se ejecuta en la forma:

ID-BUSC archivo-reglas

El archivo de reglas tiene la siguiente estructura:

$(O_1\ O_2\ \ldots\ O_m)$

$(C_1\ p_1\ \ldots\ p_a)$

...

$(C_k\ q_1\ \ldots\ q_b)$

en donde la primera lista $(O_1\ O_2\ \ldots\ O_m)$ se compone de los objetos a identificar y las siguientes listas son las reglas expresadas según la notación anterior.

Dos o más listas o reglas con la misma conclusión representan alternativas de disyunción o de tipo "o": Para que la conclusión sea verdadera se requiere que al menos en una de tales listas las premisas sean verdaderas; en caso contrario, se establece que la conclusión es falsa.

Un ejemplo de un archivo con reglas expresadas según estas especificaciones es el archivo ANIM.BD:

```
(CHITA TIGRE JIRAFA CEBRA LORO POLLO)
(MAMIFERO "DA LECHE")
(MAMIFERO "TIENE PELOS")
(AVE ! MAMIFERO "TIENE PLUMAS" )
(AVE ! MAMIFERO "PONE HUEVOS" "VUELA" )
(CARNIVORO "COME CARNE")
(CARNIVORO  "TIENE DIENTES AFILADOS" "TIENE GARRAS"
        "TIENE OJOS FRONTALES")

(UNGULADO MAMIFERO "TIENE PEZUÑA")
(UNGULADO MAMIFERO "ES RUMIANTE")

(CHITA MAMIFERO CARNIVORO "COLOR LEONADO" "TIENE MANCHAS
NEGRAS")
(TIGRE MAMIFERO CARNIVORO "COLOR LEONADO" "TIENE RAYAS NEGRAS")

(JIRAFA UNGULADO "TIENE CUELLO LARGO"
                ! "COME CARNE" "TIENE PATAS LARGAS"
                "TIENE MANCHAS NEGRAS")


(CEBRA UNGULADO "TIENE RAYAS NEGRAS")
(LORO AVE "PUEDE REPETIR PALABRAS")
(POLLO AVE "ES UN ANIMAL DOMESTICO")
```

Presentamos el programa:

```
/* Programa IDENT.C */

#include <stdio.h>
#include <ctype.h>
#define MAX_AN 4000
#define MAX_N 200
#define MAX_AR 500

#define PAR_IZQ '('
#define PAR_DER ')'
#define ID_NOM 257
#define NEG '!'

/* valores de la memoria de trabajo */
#define PREMISA 0
#define CONCLUSION 1
#define POSITIVO 2
#define NEGATIVO 3

#define NoDato 0


char AreaNom[MAX_AN],*pN; /* area de nombres */
char *Nom[MAX_N]; /* nombres son: Nom[1],...,
Nom[FinNom] */
int FinNom;

char MT[MAX_N]; /* memoria de trabajo */

int FinObj; /* objetivos son: Nom[1], ...,Nom[FinObj] */
```

```
int AreaReg[MAX_AR],*pR;    /* area de reglas */
int *Reg[MAX_AR];           /* reglas son: Reg[1],...,Reg[FinReg] */
int FinReg;

#define MAXCAR 80
char tempnom[MAXCAR];

FILE *archivo;


main(n, pal)
int n; char *pal[ ];
{  if (n != 2) error("Falta nombre de archivo con reglas ");

   lee_reglas(pal[1]); /* pal[1] contiene nombre de archivo */

   identificar();

}

lee_reglas(na)
char *na;
{ int simbolo;  int i, *pr;

  archivo=fopen(na,"rt");
  if (archivo==NULL) error("No existe archivo ");
  FinNom=FinReg=FinObj=0;

  for (i=1;i<MAX_N;i++) MT[i]=PREMISA;
  pN=AreaNom;  pR=AreaReg;

      /* lee lista de objetivos */
  simbolo=sig_nombre();       comprueba(simbolo,PAR_IZQ);

  simbolo=sig_nombre();
```

```
while (simbolo==ID_NOM) { loc_nom(tempnom);
                          simbolo=sig_nombre(); };

if (FinNom==0)
  error("Lista de objetivos no tiene datos ");

comprueba(simbolo,PAR_DER);

FinObj=FinNom; /* objetivos : 1, ..., FinObj */


    /* lee listas de reglas */

simbolo=sig_nombre(); comprueba(simbolo,PAR_IZQ);

while (simbolo==PAR_IZQ)
{ simbolo=sig_nombre(); comprueba(simbolo,ID_NOM);
  i=loc_nom(tempnom); Reg[++FinReg]=pR;
  *pR++=i; /* Es conclusion de regla: No preguntar */
  MT[i]=CONCLUSION;

  while (1)
  { simbolo=sig_nombre();
    if (simbolo==PAR_DER) { *pR++=NoDato; break; }
    if (simbolo==ID_NOM) {i=loc_nom(tempnom);
                          *pR++=i; continue; }
    if (simbolo==NEG)
          { simbolo=sig_nombre(); comprueba(simbolo,ID_NOM);
            i=loc_nom(tempnom);
            /* con signo menos para indicar condicion negativa */
            *pR++=-i; continue;
          }
     error("Se espera ) o nombre ");
  }
```

```
    simbolo=sig_nombre();
  }

   /* comprueba si objetivos aparecen en reglas */

     for (i=1;i<=FinObj;i++)
       { for (pr=AreaReg; pr<pR; pr++)   if (*pr==i) break;
         if (pr==pR)
         { sprintf(tempnom,"Objetivo %s no aparece en reglas",Nom[ i ]);
           error(tempnom);
         }
       }

}


sig_nombre()
{ int c; char *t; int ncar;

 t=tempnom;  ncar=0;

 while (1)
 { c=getc(archivo);
  if (isspace(c)) continue;
  if  (c==EOF) return EOF;
  if (isalpha(c))
    { do { *t++=c; c=getc(archivo); ncar++; }
              while (isalnum(c) && ncar<MAXCAR);
           ungetc(c,archivo); *t=0;
           if (ncar>=MAXCAR) error("Nombre es muy extenso");
           return ID_NOM;
          }
```

```
    if (c=='\"')
      { c=getc(archivo);
        while (isspace(c)) c=getc(archivo);
        if (!isalpha(c)) error("Nombre debe empezar con letra ");

        do { *t++ = c; c=getc(archivo); ncar++ ;}
          while ( (isalnum(c) || isspace(c)) && ncar<MAXCAR);
        *t=0;
        if (c!='\"' || ncar>=MAXCAR)
          error("Falta comillas de cierre o nombre es muy extenso");
        while (isspace(t[-1]) ) t--; t[0]=0;
       return ID_NOM;
      }

    if (c==PAR_IZQ || c==PAR_DER || c==NEG) return c;

    error("Caracter ilegal");

    }

}


loc_nom(t)
char *t;
{ int i;
  for (i=1;i<=FinNom;i++)
    if (strcmp(Nom[i],t)==0) return i;
  Nom[++FinNom]=pN;    strcpy(pN,t);
  pN +=strlen(pN)+1;   return FinNom;

}
```

```
error(m)
char *m;
{ printf("\nError: %s\n",m); exit(0); }


comprueba(simbolo,id)
int simbolo,id;
{ if (simbolo==id) return;
  switch(id)
  { case ID_NOM:  error("Se espera nombre");
    case PAR_IZQ: error("Se espera parentesis ( ");
    case PAR_DER: error("Se espera parentesis ) ");
  }
}


respuesta_Si(n)
char *n;
{ int c;

  printf(" %s (S/N) ? ",n);
  while (1)
  { c=getch();
    if (c=='S' || c=='s' ||  c=='N' || c=='n') break;
  }
  if (c>='a') c -=32;
  printf("%c\n",c);      return  c=='S';
}


identificar()
{ int i;
  for (i=1;i<=FinObj;i++)
```

```
    if (ident_obj(i)==POSITIVO)
      { printf("\n*** Objeto puede ser %s\n",Nom[i]);
        return;
      }

  printf("*** No es posible identificar a los objetos\n");

}

ident_obj(i)
int i;
{ char valor; int j,k, nr; int LRi[20];
  valor=MT[i];
  if (valor == POSITIVO || valor==NEGATIVO) return valor;
  if (valor == PREMISA)
   { if (respuesta_Si(Nom[i])) return  MT[i]=POSITIVO;
          return   MT[i]=NEGATIVO;
   }


   nr=0;
   for (j=1;j<=FinReg;j++)        /* obtiene todas las reglas j
                                     que tienen conclusion i */
   { pR=Reg[j]; if (*pR==i)  LRi[nr++]=j; }

   for (k=0;k<nr;k++)
   { j=LRi[k];
       /* procesa alternativas "O" */
     if (ident_y(Reg[j])==POSITIVO) return MT[i]=POSITIVO;
   }


   return MT[i]=NEGATIVO;
}
```

```
ident_y(p)
int *p;
{  int i; char valor, res;
   static int En_Prueba[MAX_N];
   static int NA=0;

   /* prueba si conclusion *p esta en lista En_Prueba */


   for (i=0;i<NA;i++)
    if (*p==En_Prueba[i])
    { sprintf(tempnom,"Conclusion %s depende de si misma! ",Nom[*p]);
            error(tempnom);
    }


   En_Prueba[NA++]=*p;

   res=POSITIVO;

   p++; /* desplaza a primer componente de regla "y" */

   while (*p)
   { i=*p++;    valor=ident_obj(abs(i));

     if (i>0  && valor==NEGATIVO || i<0 && valor==POSITIVO)
                    { res= NEGATIVO; break;}
    }

   NA--;   return res;

}
```

EJEMPLO DE CORRIDA

IDENT ANIM.BD

```
DA LECHE (S/N) ? S
COME CARNE (S/N) ? N
TIENE DIENTES AFILADOS (S/N) ? N
TIENE PEZUNA (S/N) ? S
TIENE CUELLO LARGO (S/N) ? S
TIENE PATAS LARGAS (S/N) ? S
TIENE MANCHAS NEGRAS (S/N) ? S

*** Objeto puede ser JIRAFA
```

3.7 EJERCICIOS

1. Escriba un conjunto de reglas en la forma de árbol binario (ver Ejemplo 1, Sección 3.6) para identificar:

   a) enfermedades

   b) vocaciones profesionales

   Indique al menos 5 objetos en cada caso.

2. Escriba un conjunto de reglas expresadas mediante listas de la forma:

$$(C\ p_1\ p_2 \cdots p_n)$$

   como en el ejemplo 2 del programa IDENT.C, Sección 3.6 para los dos casos del ejercicio 1.

3. Diseñe un sistema experto para evaluar los recursos humanos de una empresa.

4. Escriba un programa en el lenguaje de su preferencia para configurar o preparar una bebida que tenga en cuenta:

   a) la ocasión

   b) la clase de comida

   Proponga al menos 5 tipos de bebidas.

5. Escriba un programa para configurar los componentes de un sistema de computación.

6. Diseñe un sistema de configuración EMPACAR que indique el orden en que se debe empacar una lista de productos, para lo cual se puede tener en cuenta:

   a) tamaño del producto: grande, mediano, pequeño

   b) resistencia de producto: frágil y sólido

   Asuma bolsas de un solo tamaño que pueden contener 2 productos grandes, o 6 medianos, o 12 pequeños, o 1 grande y 3 pequeños, etc.

7. Escriba un programa para diseñar pruebas o exámenes de manera que elija preguntas de una base de datos. Cada pregunta tiene asignado un grado de dificultad: 1 (fácil), 2 (regular) y 3 (difícil).

   El programa debe solicitar cuántas preguntas de cada clase se desea elegir.

Capítulo 4

Sistemas de Generación y Prueba

4.1 CONCEPTOS BASICOS

Un sistema de generación y prueba (SGP) tiene dos elementos básicos:

(1) un programa generador de posibles soluciones

y (2) un programa probador que evalúa las soluciones propuestas, aceptándolas o rechazándolas teniendo en cuenta las condiciones presentes y los objetivos a lograr.

El SGP genera un plan de acción, o secuencia de pasos que deben ejecutarse, de acuerdo a la solución actualmente elegida.

Un SGP ejecuta el ciclo o lazo:

actuar - probar - actuar - probar - ...

hasta obtener un objetivo final o indicar fracaso.

4.2 EJEMPLOS DE SGP

1. Un programa para buscar una palabra en un diccionario ejecuta las tareas:

   Elegir una página (Acción)

   Probar si la palabra esta allí

   Si no, elegir otra página (Acción)

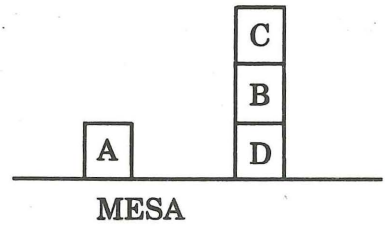
   Probar ...

   ....

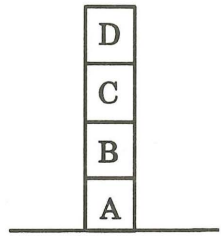
2. Un sistema de robot o computador inteligente que actúa en el mundo.

3. El "mundo de los bloques".

   Inicialmente se dispone de varios bloques sobre una mesa:



   formando una o más pilas y se requiere un programa SGP que logre poner todos los bloques formando la pila:



Para lograr esta disposición de los bloques se puede mover un bloque libre (sin otro encima de él) a la mesa o a otro bloque libre. Desde luego, se asume que la disposición inicial es arbitraria (puede cambiar con cada corrida) y además que el programa resolverá el problema de una manera inteligente; esto es, hará el menor número posible de movimientos para conseguir la disposición final.

4.3 PROGRAMA EN C SOBRE EL PROBLEMA DE LOS BLOQUES

A continuación se desarrolla un programa en C, llamado BLOQUES.C, que resuelve el problema de los bloques.

```
#include <stdio.h>
#define MAX   50
#define MAXOBJ 20
#define MESA  '*'
typedef struct { char E; char D; } tipopar;

tipopar MT[MAX];
tipopar OBJ[MAXOBJ];
int ContPar;
int ContObj;
```

```
main()
{ lee_datos();  resolver(); }


lee_datos()
{ char temp[30];
  int i,n, cp;
  tipopar *pO, *pM;

  ContPar=ContObj=0;   pM=MT;

  while (1)

   { scanf("%s",temp);
     if (strcmp(temp,"*")==0) break;
     cp=ContPar;
     n=strlen(temp);
     for (i=0;i<n; i++, pM++, ContPar++)
       { pM->E=temp[i]; pM->D=temp[i+1]; }
     pM[-1].D=MESA;
   }


  pO=OBJ;  pM--;    /* ultima lista es disposicion final */

  while (ContPar>cp)
   { pO->E=pM->E; pO->D=pM->D;
     pO++; pM--; ContPar--; ContObj++;
   }
}


resolver()
{ int n; char e,d; tipopar *p;
```

```
  p=OBJ;
  for (n=0;n<ContObj; n++, p++)

    { e=p->E; d=p->D;
      if (ubicado(e,d)) continue;
      liberar(e);  liberar(d);
      mover(e,d);
    }
}

ubicado(e,d)
char e,d;
{ int n; tipopar *p;
  p=MT;
  for (n=0;n<ContPar;n++,p++)
    if ( p->E==e && p->D==d) return 1;

  return 0;

}

liberar(x)
char x;
{ int n;  char e; tipopar *p;
  if (x==MESA) return;
  p=MT;
  for (n=0;n<ContPar;n++,p++)
    if  (p->D==x)
       { e=p->E; liberar(e); mover(e,MESA); break; }

}

mover(e,d)
char e,d;
{ int n; tipopar *p;
  p=MT;
  for (n=0;n<ContPar;n++,p++)
```

```
        if (p->E==e)
        { if (p->D != d)
            { p->D=d; printf("Mover %c a ",e);
              if (d==MESA) printf("MESA\n");
                    else printf("%c\n",d);
            }
          break;
        }

    }
```

EJEMPLO DE CORRIDA

```
BLOQUES
AB
CDE
FG
GFEDCAB
*

Mover C a A
Mover D a C
Mover E a D
Mover F a E
Mover G a F
```

EXPLICACION

1. Se inicia la ejecución del programa
2. Se ingresa primera pila
3. Segunda pila
4. Tercera pila
5. Ultima pila: Disposición final
6. Fin de ingreso de datos
7. Se muestra la sucesión de movimientos

La cadena AB representa la primera pila: A sobre B y B sobre MESA; y así sucesivamente. La última pila es la disposición final u objetivo. El símbolo * indica fin de datos

La función resolver() realiza efectivamente el ciclo "probar-actuar" a través de las funciones ubicado(), liberar() y mover().

4.4 EJERCICIOS

1. Indique 4 ejemplos de sistemas de generación y prueba.

2. Modifique el programa de los bloques de manera que admita más de una pila como objetivos.

3. Escriba un programa de generación y prueba que permita graduar la imagen de un televisor. Asuma dos selectores de control para el color y nitidez de la pantalla, respectivamente, que proveen mediciones de valores en dos intervalos 1-10 y 1-3. El nivel de control se realiza por una función de la forma

   $$f = color + 10 * (3 - nitidez), \text{ si color es impar}$$

   y $$f = color + 9 * (4 - nitidez), \text{ si color es par}$$

   Se considera una imagen aceptable si f toma un valor entre 20 y 25.

   El programa debe leer un par de valores correspondientes a los estados iniciales de color y nitidez e imprimir la secuencia a seguir para obtener una imagen satisfactoria.

4. Escriba un programa que mueva el brazo de un robot en un plano desde un punto (x_1, y_1) a otro (x_2, y_2) utilizando solamente movimientos horizontales y verticales.

Suponga que sobre el plano es dada una colección de puntos ocupados, sobre los cuales no es posible moverse.

5. Algunos problemas de generación y prueba pueden ser resueltos usando métodos de búsqueda de rutas. Por ejemplo, el problema de los misioneros y caníbales, ejemplo 2, Sección 2.1, es en realidad un problema de generación y prueba. Modifique el programa BUS-RUT.C, Sección 2.4, de manera que cuando se aplique a este problema, imprima los movimientos en lugar de la ruta encontrada.

Capítulo 5

Análisis y Comprensión de Lenguajes

5.1 INTRODUCCION

El presente capítulo desarrolla los principios básicos de los lenguajes naturales, tales como el español, inglés, etc, y la forma como estos principios se utilizan para construir sistemas de computación que reconocen las estructuras y significados de las palabras de un lenguaje.

Las aplicaciones específicas de este tema son diversas. Los ejemplos mas importantes se encuentran en la construcción de programas:

(a) de interfaz o comunicación con el usuario, que coleccionan información expresada en lenguaje natural para ser usada por otros programas;

(b) que pueden conversar en un lenguaje, realizar deducciones y aprender nuevos hechos; y

(c) traductores, que realizan traducciones de un lenguaje a otro.

El estudio de los lenguajes naturales comprende dos partes:

(a) análisis sintáctico

y (b) análisis semántico

De una manera sencilla el análisis sintáctico tiene por propósito comprobar o verificar si una lista de palabras cumple las reglas del lenguaje para formar una oración o frase y, en caso afirmativo, el análisis semántico trata de determinar los posibles significados que pueden atribuirse a la oración.

Una característica importante de los lenguajes naturales es la ambigüedad, esto es, la presencia de significados múltiples que dependen, en algunos casos, del contexto, momento y entorno en que se usa la oración. Sin lugar a dudas, esta característica constituye la mayor dificultad en la realización de programas de computación que simulen las funciones asociadas al uso de los lenguajes.

EJEMPLOS DE LISTAS DE PALABRAS

1) pedro estudia mucho

2) ana tiró la casa por la ventana

3) juan pagó el pato

4) la por la ana ventana tiró

Las tres primeras listas son evidentemente oraciones, en cambio no lo es la última. Además, las oraciones 2) y 3) admiten significados múltiples.

5.2 ANALISIS SINTACTICO

El objetivo del análisis sintáctico es determinar o reconocer si una lista de palabras dada cumple las reglas de ordenación, o reglas sintácticas, del lenguaje.

Formalmente el concepto de gramática resulta adecuado para expresar las reglas sintácticas. Una *gramática* (libre de contexto) G es definida por:

(a) Un conjunto V de símbolos o nombres, denominado vocabulario.

(b) Un conjunto de reglas o producciones R_1 , R_2 , ... , de la forma:

$$(R_i) \quad N \longrightarrow AB...P$$

en donde N, A, B, ..., y P, son símbolos del vocabulario, y expresa una relación por la cual N "produce" la lista ordenada AB...P

En este caso N y la lista ordenada se denominan *lado izquierdo* y *lado derecho* de la regla R_i, respectivamente.

Todos los símbolos del vocabulario, como N, que son lados izquierdos de reglas se llaman *no terminales*, o *nombres de reglas*, y los restantes, *terminales*.

(c) Un símbolo especial S, no terminal, llamado símbolo *inicial* de la gramática G.

5.2.1 DERIVACION DE LISTAS Y LENGUAJE DE LA GRAMATICA

Una *derivación* de una lista u N v es otra lista u AB...P v que se obtiene reemplazando N por AB...P según una regla $N \longrightarrow AB...P$

Una lista $L = p_1\, p_2 \ldots p_n$, compuesta por símbolos terminales p_i del vocabulario, es *sintácticamente correcta*, *cumple las reglas de la gramática* o *pertenece al lenguaje de la gramática*, si a partir del símbolo inicial S mediante derivaciones sucesivas es posible obtener la lista L.

En este caso la sucesión de reglas asociadas a las derivaciones se denomina un *árbol de derivación* de L.

Equivalentemente, la lista L es sintácticamente correcta si cumple una regla de nombre S, lo cual significa que L se puede descomponer o dividir en grupos de símbolos que satisfacen respectivamente las partes del lado derecho de tal regla.

5.2.2 EJEMPLO DE GRAMATICA

Dadas las reglas:

(R_1) ORACION $\longrightarrow$ SUJETO VERBO

(R_2) ORACION $\longrightarrow$ SUJETO VERBO ADVERBIO

(R_3) SUJETO ⟶ pedro | juan | ana

(R_4) VERBO ⟶ come | estudia | corre

(R_5) ADVERBIO ⟶ mucho | poco

y símbolo inicial ORACION, se obtienen:

vocabulario = { ORACION, SUJETO, VERBO, PREDICADO,
pedro, juan, ana, come, estudia,
corre, mucho, poco }

símbolos terminales = { pedro, juan, ana,
come, estudia, corre,
mucho, poco }

En este ejemplo se usa el símbolo | para designar alternativas.

Así, la regla R_3 nos indica que SUJETO puede ser pedro o juan o ana. Desde luego, esta regla es equivalente a las tres reglas:

SUJETO ⟶ pedro

SUJETO ⟶ juan

SUJETO ⟶ ana

Según esta gramática, una lista de símbolos terminales es sintácticamente correcta, en el presente caso una oración correcta, si cumple una de las dos reglas R_1 o R_2 de nombre

ORACION. La primera exige que la lista esté constituida por un sujeto seguido por un verbo, y la segunda, por éstos seguidos por un adverbio. Las otras reglas establecen qué símbolos son sujetos, verbos y adverbios.

Para la gramática actual son oraciones:

pedro come mucho [cumple la regla R_2]

ana estudia [cumple la regla R_1]

pero no lo son:

fido corre poco

[no cumple R_1 ni R_2 , pues fido no es sujeto]

pedro está cansado

5.2.3 METODOS DE RECONOCIMIENTO DE LISTAS DEL LENGUAJE

Usualmente se utilizan dos métodos para reconocer si una lista de símbolos L es sintácticamente correcta:

(1) método descendente

y (2) método ascendente

En el primer método se comienza con el símbolo inicial S y mediante sustituciones de símbolos lados izquierdos de reglas por sus respectivos lados derechos, se trata de obtener todos los elementos de la lista L.

En cambio en el segundo método se empieza con los elementos de la lista de prueba y se intenta reducirla al símbolo S reemplazando, o reduciendo, sucesivamente grupos de elementos de L que coincidan con lados derechos de reglas por los correspondientes lados izquierdos.

EJEMPLO

Comprobamos si la lista de símbolos

L = pedro come mucho

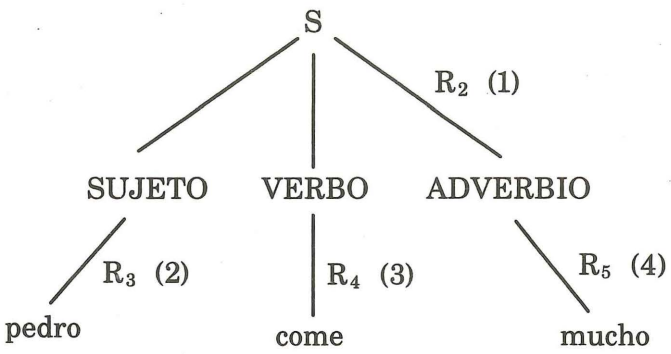
es sintácticamente correcta según la gramática del ejemplo 5.2.2

Aplicamos el método descendente:

PASOS	LISTA EN PROCESO	EXPLICACION
0	ORACION	Lista con símbolo inicial
1	SUJETO VERBO ADVERBIO	se reemplaza ORACION por el lado derecho de la regla R_2
2	pedro VERBO ADVERBIO	se sustituye SUJETO según R_3
3	pedro come ADVERBIO	R_4
4	pedro come mucho	R_5

En el paso 4 se obtiene la lista L y por tanto pertenece al lenguaje.

El árbol de derivación correspondiente a este proceso es dado por la sucesión de reglas R_2, R_3, R_4 y R_5, y es representado por la figura:



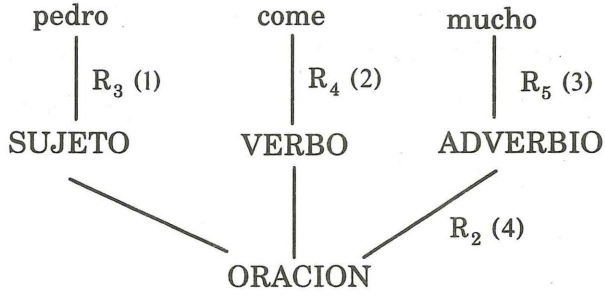
Los números entre paréntesis indican el orden seguido en el proceso de derivación.

Y si se aplica el método ascendente:

PASOS	LISTA EN PROCESO	EXPLICACION
0	pedro come mucho	se inicia con la lista de prueba
1	SUJETO come mucho	se reduce pedro por SUJETO según la regla R_3
2	SUJETO VERBO mucho	se reduce come por VERBO según R_4
3	SUJETO VERBO ADVERBIO	se reduce mucho por el lado derecho de la regla R_5
4	ORACION	se reduce toda la lista por ORACION según la regla R_2

Puesto que en el último paso se obtiene el símbolo inicial ORACION, la lista de prueba satisface las reglas de la gramática.

El árbol de derivación es formado por la sucesión de reglas obtenidas pero dispuestas en orden inverso: R_2 , R_5 , R_4 y R_3 , y el diagrama respectivo es



OBSERVACIONES

En los pasos indicados se han elegido convenientemente las reglas, antes de efectuar las sustituciones o reducciones, a fin de conseguir la lista deseada. Por ejemplo, en el paso 2 del método descendente se eligió la regla R_2 y no R_1. En verdad el proceso de reconocimiento debe probar sucesivamente cada regla posible hasta determinar que la lista pertenece al lenguaje y concluir lo contrario solamente cuando fallen todas las reglas.

5.2.4 ANALIZADORES SINTACTICOS O PARSERS

Un programa que reconoce o determina si una lista de palabras es sintácticamente correcta se llama *analizador sintáctico* o *parser*.

Indicamos un procedimiento para construir un parser que aplica el método descendente (parser descendente):

Para cada símbolo nombre de regla N se escribe una función o subprograma con dos parámetros referidos a listas de símbolos de entrada y salida, respectivamente.

De una manera precisa, si

$$N \longrightarrow A\,B \ldots P$$

entonces la función asociada a N tiene la forma:

```
N(L,LS)

    comienzo

        A(L,LA)
        B(LA,LB)
         ...
        P(LO,LP)

        LS=LP

    fin
```

En el cuerpo de instrucciones de la función, el procesamiento es el siguiente:

Si A es un nombre de regla, A(L,LA) es una llamada de ejecución a la función asociada a este símbolo; y si A no es nombre de regla, A(L,LA) consiste en comprobar si A coincide con el primer símbolo de la lista L; si es cierto, entonces LA indica la lista formada por los símbolos restantes de L.

Y de igual manera se tratan los otros símbolos B ... P.

Al finalizar se hace LS igual a LP.

El programa principal parser contiene instrucciones de llamadas a la función asociada al símbolo inicial S:

```
parser(L)
    comienzo
        S(L,LS)

    fin
```

en donde L es la lista de símbolos a ser reconocida por el parser y LS es la lista de salida producida por la función asociada a S.

La lista L es sintácticamente correcta precisamente cuando LS es vacía, esto es, si no tiene elementos. No obstante, para que el parser sea eficiente y se detenga apenas detecte la presencia de algún error de sintaxis, en el cuerpo de instrucciones se pueden incluir acciones de control, por ejemplo

para la función asociada a la regla de nombre B: si no se cumple B(LA,LB) indicar "error" y terminar.

AGRUPACION DE REGLAS SINTACTICAS

Damos algunas indicaciones que facilitan la construcción de un parser descendente:

1) Todas las reglas con igual nombre se agrupan como alternativas. De esta manera, sólo se tiene una función para cada nombre de regla. Es claro que el cuerpo de la función debe tomar en cuenta las varias posibilidades que resulten de esta agrupación.

2) Si dos o más reglas, con igual nombre, tienen partes iniciales idénticas, es conveniente "factorizarlas" y crear una nueva regla con las partes restantes no comunes.

Por ejemplo, las dos reglas de ORACION dadas en 5.2.2 pueden factorizarse así:

(R_1) ORACION $\longrightarrow$ SUJETO VERBO

(R_2) ORACION $\longrightarrow$ $\underbrace{\text{SUJETO VERBO}}_{\text{parte inicial común}}$ ADVERBIO

se reemplazan por

(R'_1) ORACION $\longrightarrow$ SUJETO VERBO COMP

(R'_2) COMP $\longrightarrow$ ADVERBIO | < nada >

en donde la regla de nombre COMP contiene las alternativas ADVERBIO y el símbolo <nada>, que sirve para indicar que la lista de entrada es vacía. De esta manera, COMP se cumplirá si es ADVERBIO o si la lista no contiene símbolos.

La ventaja de la factorización es evidente: Con las dos reglas R_1 y R_2 , si falla R_1 hay que proceder a probar R_2 repitiendo el proceso de comprobación ya hecho para SUJETO y VERBO; en cambio, si se usan las reglas R_1' y R_2' se evita este retroceso.

5.2.5 EJEMPLO DE UN PROGRAMA PARSER EN C

```
/* PROGRAMA PARSER.C

  constantes y variables:

MAXP=  numero maximo de símbolos o palabras en lista
MAXC=  numero maximo de caracteres por palabra
FIN=   indicador de fin de lista de palabras leidas
Lista= palabras son Lista[0], Lista[1], ..., Lista[npal-1]
npal=  numero actual de símbolos
LS  =  lista de símbolos sujeto
LV  =                verbo
LA  =   adverbio
ns, nv y na : numeros de símbolos en las tres listas anteriores
Cada palabra en las listas ocupa exactamante MAXC caracteres
  */
```

```
#include <stdio.h>

#define MAXP  50
#define MAXC  30
#define FIN   "$"
#define MAX   10

typedef char string[MAXC];

string Lista[MAXP];
string LS[MAX]={"pedro","juan","ana"};
string LV[MAX]={"come","estudia","corre"};
string LA[MAX]={"mucho","poco"};
int ns=3, nv=3,na=2, npal;


main()    /* funcion parser */

{  string temp;
   int ie, is;

   printf("Ingrese lista de palabras (termine con $ ) : ");
  npal=0;
  while (1)
   { scanf("%s",temp);
     if (strcmp(temp,FIN)==0) break;
     strcpy(Lista[npal++],temp);
   }
  ie=is=0;

  ORACION(&ie,&is);   /* llamada a funcion de simbolo inicial */

  if (is==npal) printf("Lista es sintacticamente correcta!\n");
```

```
 else       printf("Error de sintaxis!\n");

}


ORACION(pie,pis)  /* pie = (puntero a) indice de entrada
                       pis = indice de salida  */
int *pie,*pis;
{ int *pS, *pV, *pC;
  pS=pV=pC=pie;

   /* aplica regla: ORACION ---> SUJETO VERBO COMP */

  SUJETO(pie,pS);
  VERBO(pS,pV);
  COMP(pV,pC);
  *pis=*pC;
}


SUJETO(pie,pis)
int *pie, *pis;
{ char *t;
  t=Lista[*pie];

   /* funcion es_miembro prueba si palabra dada por t
                       se encuentra en lista de sujetos LS */
  if (es_miembro(t,LS,ns)) { *pis=*pie+1; return; }

  else error("SUJETO ",t);

}

VERBO(pie,pis)
```

```
int *pie, *pis;
{ char *t;
  t=Lista[*pie];

  if (es_miembro(t,LV,nv)) {  *pis=*pie+1; return; }
  else error("VERBO ",t);

}

COMP(pie,pis)
int *pie, *pis;
{ char *t;
  if (*pie==npal) return;
  t=Lista[*pie];
  if (es_miembro(t,LA,na)) {  *pis=*pie+1; return; }
  else error("ADVERBIO ",t);

}

es_miembro(t,L,n)
string *t, *L;
int n;
{  int i;
   for (i=0;i<n;i++) if (strcmp(t,L[i])==0) return 1;
   return 0;
}


error(m,t)
char *m, *t;
{  printf("Error de sintaxis en %s : %s\n",m,t);
   exit(0);
}
```

EJEMPLOS DE CORRIDAS DEL PROGRAMA:

(1) PARSER

Ingrese lista de palabras (termine con $) : pedro come mucho $
Lista es sintácticamente correcta!

(2) PARSER

Ingrese lista de palabras (termine con $) : juan lee poco $
Error de sintaxis en VERBO: lee

(3) PARSER

Ingrese lista de palabras (termine con $) : ana estudia $
Lista es sintacticamente correcta!!

(4) PARSER

Ingrese lista de palabras (termine con $) : pedro come mucho pan $
Error de sintaxis!

OBSERVACIONES SOBRE EL PROGRAMA

En lugar de las tres listas de símbolos terminales LS, LV y LA se puede formar una sola como un diccionario que contenga estos símbolos o palabras y un indicador de su clase o categoría (SUJETO, VERBO o ADVERBIO) :

PALABRA	CLASE
"pedro"	claseSujeto
"juan"	claseSujeto
"ana"	claseSujeto
"corre"	claseVerbo
...	
"poco"	claseAdverbio

en cuyo caso es necesario modificar la función es_miembro( ) :

es_ miembro(palabra, clase)

de manera que determine si en el diccionario se encuentra dicha palabra con la clase indicada.

5.2.6 EJERCICIOS

1. Dadas las reglas:

   PREGUNTA → VERBO SUJETO ?

   PREGUNTA → VERBO SUJETO COMPLEMENTO ?

   VERBO → es

   SUJETO → pedro

   SUJETO → juan

   COMPLEMENTO → alto

   COMPLEMENTO → bajo

   y símbolo inicial PREGUNTA,

   indique el vocabulario, los símbolos terminales (incluya ?) y no terminales, y determine si son sintácticamente correctas las listas:

   es juan alto ?

   es pedro ?

   es juan

2. a) Para la gramática dada en 5.2.2 escriba todas las listas que cumplan la regla

R_1: ORACION ——→ SUJETO VERBO

b) Escriba 5 listas sintácticamente correctas que satisfacen la regla R_2

3. Aplique el método descendente y obtenga un árbol de derivación para la lista: ana estudia mucho
en relación a la gramatica de 5.2.2

4. Según la gramática:

(1) ORACION → NOMBRE VERBO

(2) ORACION → ARTICULO NOMBRE VERBO

(3) ARTICULO → el | un | la

(4) NOMBRE → pedro | gato | luisa

(5) VERBO → ladra | maúlla | lee | escribe

determine si son sintácticamente correctas la listas

el perro escribe

luisa ladra

un gato maúlla

el juan lee

5. Escriba un conjunto de reglas para definir una gramática de oraciones de tipo implicación que incluya la regla

(R_0) IMPLICA ——→ si ORACION entonces ORACION

y símbolo inicial IMPLICA en la gramática dada en 5.2.2. Observe que "si" y "entonces" son símbolos terminales.

La nueva gramática reconoce listas de la forma:

si pedro estudia mucho entonces ana corre

6. Escriba un parser que reconozca listas sintácticamente correctas para la gramática del ejercicio anterior.

5.3 ANALISIS SEMANTICO

El análisis sintáctico no tiene en cuenta si una lista de palabras o símbolos, aun cuando sea gramaticalmente correcta, pueda carecer de significado alguno en relación al conocimiento que tenemos de ellos. Por ejemplo:

una mesa come mucho

es aceptada por el analizador sintáctico pero se aleja de lo que usualmente atribuimos tanto al objeto mesa como al verbo comer.

El aspecto semántico tiene que ver con el estudio del significado de una lista de símbolos sintácticamente correcta. Este proceso se realiza con un programa capaz de seleccionar y asignar significados al que denominamos analizador o intérprete semántico.

5.3.1 PROCESO DE INTERPRETACION SEMANTICA

Describimos los elementos que participan en el análisis semántico y la forma en que éstos intervienen en dicho proceso.

En primer lugar, se requiere un diccionario de significados o interpretaciones, según un sistema de conocimientos especificado.

El diccionario contiene significados asociados a palabras o ciertos grupos de palabras.

La gramática inicial G, definida por las reglas sintácticas, debe ser ampliada o modificada a fin de obtener significados y almacenarlos, mediante una representación interna, como nuevos hechos de una base de conocimientos y asegurar así su uso posterior.

Durante el procesamiento de una lista de símbolos se reconocen los significados de cada uno y los de algunos grupos de ellos. El significado de la lista íntegra se obtiene combinando los significados de las partes (semántica de composición).

Generalmente, el analizador semántico actúa coordinadamente con el analizador sintáctico: A medida que se reconocen grupos de símbolos se atribuyen significados. Esto se consigue bien incorporando nuevas reglas sintácticas (refinamiento de la gramática) o bien insertando en algunas

de ellas tareas o subprogramas llamados acciones semánticas, por ejemplo en la forma:

$$(R_i)\ N \rightarrow A\ B\ \{\text{acción} - B\}\ C \ldots P$$

en donde {acción – B} es el subprograma encargado de seleccionar significados, y salvarlos en la base de datos, después de aplicar o reconocer B.

Como se muestra aquí, estas acciones se realizan a continuación de ciertos símbolos del lado derecho de una regla.

La determinación de significados por la acción semántica B utiliza las relaciones del símbolo actualmente reconocido con los procesados en A, o con los siguientes símbolos de la lista de prueba.

Se puede afirmar que, salvos casos muy simples, el diseño de un analizador semántico se caracteriza por ser difícil y complejo.

EJEMPLO DE UNA GRAMATICA CON ACCIONES SEMANTICAS

ORACION → FSUJETO FVERBO

FSUJETO → ARTICULO SUJETO {acc – S}

FSUJETO → SUJETO

FVERBO → VERBO {acc – V} COMPLEMENTO

COMPLEMENTO → CD CI

CD	→	< nada > | FSUJETO {acc – CD}
CI	→	< nada > | PREP FSUJETO
ARTICULO	→	el | un | la | una
SUJETO	→	pedro | perro | manzana | tenedor
VERBO	→	come
PREP	→	con

en donde FSUJETO designa "frase sujeto" y FVERBO, "frase verbo"; CD y CI designan complementos directo e indirecto, respectivamente.

La producción CD significa que la siguiente palabra de la lista de prueba puede ser FSUJETO o no -alternativa <nada> -, en cuyo caso se devuelve la lista intacta.

Describimos algunas acciones semánticas:

a) {acc-S} comprueba si al sujeto actual le precede un artículo; por ejemplo:

un perro ...	es semánticamente correcta
la pedro ...	no lo es

b) {acc-V} comprueba si al verbo actual le puede preceder el sujeto anterior; así:

pedro come	es semánticamente correcto
el tenedor come ...	no lo es

Un diccionario bastante simple que expresa estas relaciones semánticas puede formarse con listas de la forma:

SIMBOLO	SIMBOLOS ANTERIORES
tenedor	ARTICULOS: el, un
come	SUJETOS: pedro, perro
manzana	VERBO: come
...	

Las siguientes listas son semánticamente correctas:

pedro come

pedro come la manzana

pedro come una manzana con el tenedor

el perro come la manzana con pedro

5.3.2 EJEMPLO DE UN PROGRAMA QUE DIALOGA

En lugar de presentar un programa ejemplo que realice el análisis sintáctico y semántico para una gramática como la mostrada en 5.3.1, desarrollamos un programa que dialoga con el usuario y es capaz de aprender y efectuar deducciones o inferencias. Una corrida del programa DIALOGA.C, escrito en C, es la siguiente:

```
* prosiga ...
  si pedro estudia mucho entonces pedro aprueba IA
* prosiga ...
  aprueba pedro IA ?
* no lo se, prosiga ...
  si pedro aprueba AI entonces pedro es feliz
* prosiga ...
  pedro estudia mucho
* prosiga ...
  es feliz pedro ?
* es cierto, prosiga ...
  aprueba pedro IA ?
* es cierto, prosiga ...
  fin
```

Las líneas precedidas por * representan las respuestas dadas por el programa. Las otras líneas u oraciones son ingresadas por el usuario (terminan con Enter o Return) y puede ser la palabra fin, con la que se termina la ejecución del programa, o una de los tres tipos siguientes:

1) afirmativa : sujeto verbo complemento
2) pregunta : verbo sujeto complemento ?
3) inferencia : si OA entonces OA

en donde OA es una oración afirmativa y complemento puede contener a lo sumo una palabra.

Para simplificar el programa se ha omitido el parser, de manera que el programa no comprueba si las listas son sintácticamente correctas.

El listado del programa es:

```
    /* PROGRAMA DIALOGA.C */


#include <stdio.h>
#include <string.h>

#define MAX      200
#define MAXC      30
#define FINSIG    -1
#define NODEFINIDO  0
#define POSITIVO    1

typedef char string[MAXC];

typedef struct      { char clase;
                      int  iSig;
                      char npal;
                      string pal[3]; } tdato;

tdato  lista[MAX], *pl;
int    ContLista=0;

string pal[10];
char *vp[3];

int NumPal;

main()
{ char linea[80];
  int i,j,n;
```

```
printf("\n\n*");
 while (1)
 { printf(" prosiga ...\n");
   otra_vez:  gets(linea);
   descompone(linea);
   if (NumPal==0) goto otra_vez;
   if (stricmp(pal[0],"FIN")==0) break;

   if (strcmp(pal[NumPal-1],"?")==0)
    { NumPal--;
       vp[0]=pal[1]; vp[1]=pal[0]; vp[2]=pal[2];
       i=loc_lista(NumPal);
       if (i>=0 && lista[i].clase==POSITIVO)
                  printf("* es cierto, ");
        else      printf("* no lo se, ");

      continue;
    }
   if (stricmp(pal[0],"SI")==0)
    {
      for (i=1;i<NumPal;i++)
      if (stricmp(pal[i],"ENTONCES")==0) break;
      if (i==NumPal) error("Falta entonces");
      n=i;
      vp[0]=pal[1]; vp[1]=pal[2]; vp[2]=pal[3];
      i=loc_lista(n-2);
      if (i<0) i=agregar(n-2);

      vp[0]=pal[n+1]; vp[1]=pal[n+2]; vp[2]=pal[n+3];
      j=loc_lista(NumPal-n);
      if (j<0) j=agregar(NumPal-n);
      lista[i].iSig=j;
      actualizarPos(i);
```

```
            printf("*");
            continue;
        }
        vp[0]=pal[0]; vp[1]=pal[1]; vp[2]=pal[2];

        i=loc_lista(NumPal);
        if (i<0) i=agregar(NumPal);
        actualizarPos(i);
        printf("*");
    }

}


descompone(s)
char *s;
{ char *t;  char *e;
  NumPal=0;  t=pal[NumPal]; e=s+strlen(s);
  while (s<=e && sscanf(s,"%s",t)>0)
  { s +=strlen(t)+1;   t=pal[++NumPal]; }


}

agregar(n)
int n;
{  int i;
   pl=lista+ContLista;        pl->clase=NODEFINIDO;
   pl->iSig = FINSIG;         pl->npal=n;
   for (i=0;i<n; i++) strcpy(pl->pal[i],vp[i]);
   return ContLista++;
}
```

```
loc_lista(n)
int n;

{ int i,j;

  for (i=0,pl=lista; i<ContLista;i++,pl++)
  {  for (j=0;j<n; j++)
      if (stricmp(pl->pal[j],vp[j])) break;
    if (n==pl->npal) return i;
  }

 return -1;

}

actualizarPos(i)
int i;
{
  while (1)
 { pl =lista+i;
   pl->clase=POSITIVO;
   i= pl->iSig;
   if (i== FINSIG) return;
 }

}

error(m)
char *m;
{ printf("\nError : %s\n",m); exit(0); }
```

Capítulo 6

Lógica y Demostración de Teoremas

6.1 INTRODUCCION

El cálculo de predicados de primer orden es un lenguaje formal que es usado en muchos sistemas de Inteligencia Artificial. Los predicados son expresiones que describen relaciones entre datos y simplifican la representación y el procesamiento de éstos, por ejemplo en: La recuperación inteligente de información, problemas de robótica y en la deducción de nuevos hechos (demostración de teoremas).

En el presente capítulo se estudian los conceptos básicos de este lenguaje, suficientes para exponer el principio de resolución completa y sus aplicaciones en la demostración de teoremas.

6.2 ALGEBRA DE PREDICADOS

6.2.1 PREDICADOS

Un *predicado* P es una función definida sobre un conjunto D tal que para cada a en D, P(a) es el valor verdadero (V) o falso (F).

En general el conjunto D es un producto cartesiano de conjuntos $A_1, A_2, \ldots, A_n$, es decir

$$D = A_1 \times A_2 \times \ldots \times A_n$$
$$= \{a = (a_1, a_2, \ldots, a_n) \,/\, a_i \text{ es un elemento de } A_i\}$$

y el valor $P(a_1, a_2, \ldots, a_n)$ depende de $a_1, a_2, \ldots, a_n$.

Como es usual, a menudo emplearemos las notaciones $P(x)$ o $P(x_1, x_2, \ldots, x_n)$ para designar al predicado P, con las letras $x, y, \ldots$, etc., que indican los argumentos del predicado P. Tales símbolos se llaman *variables* y representan elementos arbitrarios del conjunto D. También usaremos la designación *constante* para aludir a un elemento particular de D.

EJEMPLOS

1) El predicado P definido sobre los números naturales 0,1, 2 , ... por

$$P(x): "x \text{ es número par }"$$

es decir, el valor de $P(x)$ es V si es x un número par, y F si es un número impar.

En particular, P(4) es verdadero y P(3) es falso.

2) La relación $H(x, y)$: "x es hermano de y "

determina un predicado entre los miembros x, y de un conjunto de personas A.

En este caso H está definido sobre el conjunto $D = A \times A$. Si, por ejemplo, se sabe que "pedro" es hermano de "maria", entonces H("pedro","maria") tiene el valor V.

6.2.2 OPERACIONES CON LOS PREDICADOS

Si P y Q son predicados se definen nuevos predicados $\neg P$, $P \wedge Q$, $P \vee Q$, $P \Rightarrow Q$ y $P \Leftrightarrow Q$, llamados *negación*, *conjunción*, *disyunción*, *implicación* y *equivalencia*, respectivamente, mediante las siguientes reglas expresadas por las tablas:

$P(x)$	$(\neg P)(x)$
V	F
F	V

o sea $(\neg P)(x)$ es verdadero sólo cuando $P(x)$ es falso

$P(x)$	$Q(x)$	$(P \wedge Q)(x)$
V	V	V
V	F	F
F	V	F
F	F	F

$P(x)$	$Q(x)$	$(P \vee Q)(x)$
V	V	V
V	F	V
F	V	V
F	F	F

$P(x)$	$Q(x)$	$(P \Rightarrow Q)(x)$
V	V	V
V	F	F
F	V	V
F	F	V

lo que significa que $(P \Rightarrow Q)(x)$ es falso sólo cuando la premisa $P(x)$ es verdadera y la conclusión $Q(x)$ es falsa.

$P(x)$	$Q(x)$	$(P \Leftrightarrow Q)(x)$
V	V	V
V	F	F
F	V	F
F	F	V

Si P y Q son predicados, decimos que son *iguales*, y escribimos $P = Q$, si toman los mismos valores para cada x, esto es, si se cumple:

$$P(x) = Q(x), \text{ para cada } x .$$

También se definen dos predicados I, O con valores constantes V y F:

$$I(x) = V$$

$$O(x) = F$$

para cada x.

A continuación presentamos, sin demostración, las propiedades básicas de las operaciones entre predicados.

Se cumplen las siguientes leyes o propiedades:

1) ley de idempotencia:

$$P \vee P = P$$
$$P \wedge P = P$$

2) ley conmutativa:

$$P \vee Q = Q \vee P$$

$$P \wedge Q = Q \wedge P$$

3) ley asociativa:

$$(P \vee Q) \vee R = P \vee (Q \vee R)$$

$$(P \wedge Q) \wedge R = P \wedge (Q \wedge R)$$

4) ley distributiva:

$$P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$$

$$P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$$

5) leyes de la negación:

$$\neg I = O$$

$$\neg O = I$$

$$P \vee \neg P = I$$

$$P \wedge \neg P = O$$

$$\neg(\neg P) = P$$

$$\neg(P \vee Q) = \neg P \wedge \neg Q$$

$$\neg(P \wedge Q) = \neg P \vee \neg Q$$

6) $P \Rightarrow Q = \neg P \vee Q$

$$P \Leftrightarrow Q = (P \Rightarrow Q) \wedge (Q \Rightarrow P)$$

$$P \Leftrightarrow Q = (P \wedge Q) \vee (\neg P \wedge \neg Q)$$

Además de las operaciones estudiadas, a las que se les suele llamar *conectivos lógicos*, hay otras dos dadas por los cuantificadores *universal* y *existencial*. Si $P = P(u,v)$ es un predicado que depende de las variables u y v, se definen los predicados $\forall u\, P(u,v)$ y $\exists u\, P(u,v)$ de manera tal que en cada v sus valores son:

$\forall u\, P(u,v) = V$ si $P(u,v)$ es V para todo valor de u

$= F$ de otra manera

y $\exists u\, P(u,v) = V$ si $P(u_0,v)$ es V para algún u_0 (que depende de v)

$= F$ de otra manera

Es claro que estas definiciones se extienden de una manera simple a cualquier variable de un predicado arbitrario.

En una expresión lógica, una variable *ligada* o asociada a un cuantificador sólo se refiere a la subexpresión afectada por dicho cuantificador -ámbito del cuantificador- y se trata en efecto de una variable distinta de las que aparecen fuera de la subexpresión, aun cuando éstas tengan el mismo nombre. Por lo tanto, cuando sea conveniente destacar esta distinción, dentro del ámbito del cuantificador la variable ligada puede ser cambiada por otra.

EJEMPLOS

1) La expresión $Q(x) \vee (\exists x\, P(x))$ contiene la variable x en dos subexpresiones: $Q(x)$, $(\exists x\, P(x))$. En la segunda, x se encuentra ligada al cuantificador $\exists$, y puede ser reemplazada por otra, tal como y, distinta de x, de modo que la expresión también se escribe así: $Q(x) \vee (\exists y\, P(y))$.

2) De igual modo,

$$\forall x \left(P(x) \Rightarrow (\exists x\, Q(x))\right)$$

puede escribirse

$$\forall x \left(\mathrm{P}(x) \Rightarrow (\exists z\, \mathrm{Q}(z))\right)$$

Las propiedades relacionadas con los cuantificadores son:

7) $\neg(\forall y\ \mathrm{P}) = \exists y\ (\neg \mathrm{P})$

$\neg(\exists y\ \mathrm{P}) = \forall y\ (\neg \mathrm{P})$

8) $\exists y\ (\mathrm{P}(y,z)) = \mathrm{P}(f(z),z)$ en donde f es una función, denominada *función de Skolem*, que asigna a z un valor $y_0 = f(z)$ para el cual $\mathrm{P}(y_0, z)$ es verdadero.

Se asume que si P sólo depende de la variable y, la función f tiene un valor constante, por ejemplo a, entonces (8) se escribe así:

$$(\exists y\ \mathrm{P}(y)) = \mathrm{P}(a)$$

Las siguientes propiedades, en las que se supone que la variable y ligada al cuantificador es distinta de las variables del predicado M (cambiando si es necesario el nombre de y), muestran que el cuantificador se puede desplazar a la izquierda de la expresión:

9) $\mathrm{M} \wedge (\forall y\ \mathrm{P}) = \forall y\ (\mathrm{M} \wedge \mathrm{P})$

10) $\mathrm{M} \wedge (\exists y\ \mathrm{P}) = \exists y\ (\mathrm{M} \wedge \mathrm{P})$

11) $M \vee (\forall y\ P) = \forall y\ (M \vee P)$

12) $M \vee (\exists y\ P) = \exists y\ (M \vee P)$

6.3 FORMA DE CLAUSULAS

A partir de un conjunto inicial de predicados P, Q, ... denominados *predicados atómicos* o simplemente *átomos*, se construyen otros, a los que se llaman *fórmulas bien definidas* (FBD) o expresiones lógicas, aplicando las siguientes reglas:

1) Si P es un predicado atómico y los símbolos x_i representan variables o valores constantes, entonces $P(x_1, x_2, \ldots, x_n)$ es una FBD

2) Si F y G son FBD, entonces también lo son las expresiones que resultan de aplicarles las operaciones lógicas.

Dicho de otro modo, una FBD no es más que una expresión compuesta por predicados atómicos, con argumentos variables o constantes, relacionados mediante conectivos lógicos o afectados por cuantificadores.

EJEMPLO

Si consideramos los predicados atómicos:

$P(x)$: x es un perro

$H(x)$: x tiene hambre

$L(x)$: x ladra

en donde x es una variable que representa un elemento arbitrario de un conjunto de animales, entonces son FBD :

a) P("fido")

b) H("fido")

c) $(\forall x)\big(P(x) \wedge H(x) \Rightarrow L(x)\big)$

que en lenguaje ordinario tiene el enunciado: "Si x es un perro y tiene hambre, entonces ladra"

Una FBD F puede admitir diferentes representaciones cuando se expresa en términos de los átomos. Por ejemplo, según la propiedad 6 de 6.2.2, las expresiones $P \Rightarrow Q$ y $\neg P \vee Q$ representan la misma FBD.

Se llama *literal* L a un predicado atómico P o a su negación $\neg P$. Una *cláusula* C es una FBD formada por disyunciones de literales:

$$L_1 \vee L_2 \vee \ldots \vee L_m$$

En este caso, la cláusula también se designa por:

$$\{L_1, L_2, \ldots, L_m\}.$$

EJEMPLO

La expresión con predicados atómicos P, Q y R:

$P \vee \neg Q \vee R$

o $\{P, \neg Q, R\}$

es una cláusula cuyos literales son: P, $\neg Q$ y R.

Se dice que F tiene la *forma de cláusula plena* (FCP), o *forma normal conjuntiva plena*, si se expresa mediante una conjunción de cláusulas precedida por cuantificadores universales:

$$\cup C_1 \wedge C_2 \wedge \ldots \wedge C_n$$

en donde $\cup$ designa cero o más cuantificadores universales y cada C_i es una cláusula

Así, la expresión que sigue a $\cup$ se compone de conjunciones de grupos de disyunciones de átomos o negaciones de éstos.

6.3.1 REDUCCION DE UNA FBD A FORMA DE CLAUSULAS (FCP)

Una FBD F puede ser transformada en otra que tiene la representación de forma de cláusula plena aplicando el siguiente procedimiento:

(R1) Se eliminan las operaciones $\Rightarrow$ y $\Leftrightarrow$ aplicando la propiedad (6) de 6.2.2 .

(R2) Se desplaza el operador $\neg$ a los átomos según (5), 6.2.2

(R3) Se eliminan los cuantificadores existenciales, por (8), 6.2.2, sustituyendo las variables afectadas por funciones de Skolem, que deben recibir diferentes nombres.

(R4) Se desplazan los cuantificadores universales a la izquierda (propiedades (9) y (11) , 6.2.2), de modo que

ningún cuantificador aparezca en el interior de la expresión.

(R5) Se distribuye $\vee$ respecto de $\wedge$ según la segunda fórmula de (4), 6.2.2.

(R6) Se renombran, si es necesario, las variables de las cláusulas de manera que cláusulas distintas no tengan variables del mismo nombre.

EJEMPLOS

Reducir las siguientes expresiones a formas de cláusulas

1. $\neg(\neg P) \Leftrightarrow Q$
2. $(P \wedge \neg Q) \Rightarrow \big((\neg(\neg P)) \Leftrightarrow Q\big)$
3. $\forall x \, (A(x) \Rightarrow E\, y \; B(x, y))$

Solución

1. Se tiene

   $\neg(\neg P) \Leftrightarrow Q$ [aplicando $\neg$]

   $P \Leftrightarrow Q$ [usando 6, 6.2.2]

   $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$

   $(\neg P \vee Q) \wedge (\neg Q \vee P)$

   con cláusulas $\{\neg P, Q\}$ y $\{P, \neg Q\}$ y sin cuantificadores.

2. Para $(P \wedge \neg Q) \Rightarrow \big((\neg(\neg P)) \Leftrightarrow Q\big)$

eliminamos $\Rightarrow$:

$$\neg(P \wedge \neg Q) \vee ((\neg P \vee Q) \wedge (\neg Q \vee P))$$

aplicamos $\neg$ y designamos $\neg P \vee Q$ y $\neg Q \vee P$ por A y B, respectivamente:

$$A \vee (A \wedge B)$$

distribuimos $\vee$ respecto de $\wedge$:

$$(A \vee A) \wedge (A \vee B)$$

reemplazamos A y B:

$$(\neg P \vee Q) \wedge (\neg P \vee Q \vee \neg Q \vee P)$$

y usamos $\neg P \vee P = I = \neg Q \vee Q$:

$$(\neg P \vee Q) \wedge I$$

$$\neg P \vee Q$$

Luego esta expresión consiste de una sola cláusula $\{\neg P, Q\}$

3. La tercera FBD $\forall x\,(A(x) \Rightarrow \exists\, y\, B(x,y))$ se puede reducir así:

$$\forall x\,(\neg A(x) \vee \exists\, y\, B(x,y))$$

$$\forall x\,(\neg A(x) \vee B(x, f(x)))$$

[en donde $y = f(x)$ es una función de Skolem asociada a la variable y]

que tiene la forma de cláusula plena con un cuantificador, en la variable x, y una cláusula

$$\{\neg A(x), B(x, f(x))\}$$

con dos literales $\neg A$ y B.

6.4 DEMOSTRACION DE TEOREMAS Y PRINCIPIO DE RESOLUCION

En lo que sigue asumimos que A_i y B son fórmulas bien definidas respecto de un conjunto de predicados atómicos que toman valores en elementos de un dominio no vacío.

Decimos que B es una *consecuencia lógica* de $A_1, A_2, \ldots, A_n$, o que éstas *implican* (*lógicamente*) B, si para cada a tal que todos los $A_i(a)$ son verdaderos se cumple que $B(a)$ es verdadero, esto es, si se verifica la igualdad

$$A_1 \wedge A_2 \wedge \ldots \wedge, A_n \Rightarrow B = I \qquad (1)$$

En este caso, B se denomina un *teorema con hipótesis* o *axiomas* A_i y la igualdad indicada expresa el hecho de que B será verdadero siempre que los axiomas lo sean.

Aplicando las propiedades (5) y (6) de 6.2.2 a la igualdad anterior, podemos concluir que B es un teorema en los A_i si y sólo si se cumple:

$$A_1 \wedge A_2 \wedge \ldots \wedge, A_n \wedge \neg B = O \qquad (2)$$

El principio de resolución completa proporciona un método para determinar si B es un teorema mediante una secuencia de pasos, o *demostración*, en los que se utilizan las representaciones de formas de cláusulas plenas de B y los axiomas.

Las igualdades (1) y (2) dan lugar a dos maneras de demostración: *directa* o *indirecta* (o por contradicción). En la demostración directa, a partir de las cláusulas de los axiomas se trata de obtener las cláusulas del teorema. Y en la indirecta, se comienza con las cláusulas de los axiomas y las de la negación de B y se intenta derivar el conjunto vacío de cláusulas.

Independientemente de la forma de demostración, cada paso del método de resolución produce nuevas cláusulas llamadas resolventes.

Se dice que dos cláusulas C_1 y C_2 son *resolubles* si una de ellas contiene a un átomo P y la otra a su negación $\neg P$, con los mismos símbolos en los argumentos de P. En este caso se denomina *resolvente* de C_1 y C_2 a la cláusula R que resulta de unir los literales de ambas excluyendo los átomos complementarios P, $\neg P$. Por ejemplo, para

$$C_1 = \{M, P, \neg N\} \text{ y } C_2 = \{\neg P, Q\}$$

se tiene que $R = \{M, \neg N, Q\}$ es un resolvente de C_1 y C_2 y el resolvente de $C_3 = \{\neg P(x,a)\}$ y $C_4 = \{P(x,a)\}$ es $S = \{\ \}$, la cláusula vacía.

6.4.1 PROCESO DE UNIFICACION

Si C_1 y C_2 contienen a P y $\neg P$ pero no son resolubles, en muchos casos se les puede aplicar el proceso de unificación

antes de hallar un resolvente: Una variable de P asociada a un cuantificador puede ser reemplazada por otro símbolo que aparezca como argumento en la misma posición de P o $\neg$P, siempre que este símbolo sea otra variable, una constante o una función que no dependa de la variable. Por ejemplo, si se tienen las cláusulas:

$$C_1 = \{ \ldots, P(a, x, f(u)), \ldots \}$$

$$C_2 = \{ \ldots, \neg P(u, y, z), Q(u, y), \ldots \}$$

en donde a es una constante, u, x, y, z son variables, entonces se puede hacer: $u = a$, $x = y$, $z = f(u)$, resultando las cláusulas resolubles:

$$C_1' = \{ \ldots, P(a, x, f(a)), \ldots \}$$

$$C_2' = \{ \ldots, \neg P(a, x, f(a)), Q(a, x), \ldots \}$$

6.4.2 ALGORITMO DE DEMOSTRACION POR CONTRADICCION

Para demostrar por contradicción que B es una consecuencia lógica de $A_1, A_2, \ldots, A_n$, empleando el principio de resolución completa, se procede así:

1) Se representan los axiomas A_i y $\neg$B en forma de cláusula plena y se construye una lista $C_1, C_2, \ldots, C_n$ con todas las cláusulas.

2) Se encuentra el resolvente R de un par de cláusulas C_i y C_j, omitiendo los pares de cláusulas ya tratados. Si R es la cláusula vacía, el proceso termina indicando la validez del teorema; en caso contrario, se incrementa el valor de n, se hace $C_n = R$ y se repite 2).

Si no existe ningún resolvente, el proceso termina indicando que B no es una consecuencia lógica de los axiomas.

NOTA

Con el propósito de abreviar la aplicación del algoritmo, en los ejemplos que se desarrollan a continuación, omitiremos la regla (R6) del procedimiento de reducción a forma de cláusula (6.3.1), y por esto, dos cláusulas distintas pueden contener una misma variable. No obstante, como sabemos, la variable representa dos objetos distintos en cada cláusula y su nombre puede ser reemplazado por otro en cualquiera de ellas.

6.4.3 EJEMPLOS

1) Probar que los axiomas (sin cuantificadores):

(a) P

(b) $P \Rightarrow Q$

implican lógicamente Q

Escribiendo en forma de cláusulas P, $P \Rightarrow Q$ y la negación de Q se tiene:

$C_1 = \{P\}$

$C_2 = \{\neg P, Q\}$ pues $P \Rightarrow Q = \neg P \vee Q$

$C_3 = \{\neg Q\}$

Luego, el resolvente de C_1 y C_2 es $C_4 = \{Q\}$ y el resolvente de C_3 y C_4 es $C_5 = \{\ \}$, la cláusula vacía.

Por tanto Q es una consecuencia lógica de los axiomas dados.

2) Probar que $P \Rightarrow Q$ y $Q \Rightarrow R$ implican $P \Rightarrow R$.

En este caso las cláusulas de los axiomas son:

$C_1 = \{\neg P, Q\}$

y $C_2 = \{\neg Q, R\}$

Y para la negación de $P \Rightarrow R$ se tiene:

$$\neg(P \Rightarrow R) = \neg(\neg P \vee R) = P \wedge \neg R$$

de donde resultan las dos cláusulas

$C_3 = \{P\}$

$C_4 = \{\neg R\}$.

Ahora determinamos los resolventes:

$C_5 = \{Q\}$, usando el par C_1 y C_3

$C_6 = \{\neg Q\}$, para C_2 y C_4

$C_7 = \{\ \}$, resolvente de C_5 y C_6

que es la cláusula vacía.

3) Probar que $\forall x\, P(x)$ implica $\exists y\, P(y)$

La cláusula del axioma es $C_1 = \{P(x)\}$, con la variable x, y la negación de $\exists x\, P(x)$ es:

$$\neg(\exists y\, P(y)) = \forall y\, (\neg P(y))$$

y por lo tanto su cláusula es:

$$C_2 = \{\neg P(y)\}\text{ , con una variable } y$$

Unificando las variables x e y del predicado P en C_1 y C_2, (por ejemplo haciendo $x = y$ en $C_2 : \{\neg P(x)\}$) se obtiene el resolvente $C_3 = \{\ \}$

4) Determinar si:

(a) Todos los hombres son animales

implica lógicamente

(b) Algunos hombres son mentirosos

Las dos reglas pueden escribirse así:

$$A : \forall x (H(x) \Rightarrow A(x))$$

$$B : \exists y (H(y) \Rightarrow M(y))$$

en donde los predicados atómicos H, A y M designan las relaciones: es hombre, animal y mentiroso, respectivamente. Luego las cláusulas de A y $\neg$B son:

$C_1 = \{\neg H(x), A(x)\}$

$C_2 = \{\neg H(y), \neg M(y)\}$, pues $\neg B = \forall y\,(\neg H(y) \vee \neg M(y))$

y unificando, $x = y$, vemos que C_1 y C_2 no tienen resolventes.

Por tanto, (b) no es una consecuencia lógica de (a).

5) Determinar si

(a) fido ladra

es una consecuencia lógica de

(b) fido es un perro

(c) fido tiene hambre

(d) si un perro tiene hambre entonces ladra

Representando estas relaciones en términos de predicados atómicos se tiene

B: $ladra(fido)$

A_1: $perro(fido)$

A_2: $thambre(fido)$

A_3: $\forall x\,(perro(x) \wedge thambre(x) \Rightarrow ladra(x))$

y las cláusulas de A_i y $\neg B$ son:

$C_1 = \{perro(fido)\}$

$C_2 = \{thambre(fido)\}$

$C_3 = \{\neg perro(x), \neg thambre(x), ladra(x)\}$

$C_4 = \neg B = \{\neg ladra(fido)\}$

en donde x es una variable y *fido* es una constante.

Aplicando el principio de resolución se deriva la siguiente sucesión de resolventes:

de C_1 y C_3

$C_5 = \{\neg thambre(fido), ladra(fido)\}$, luego de hacer $x = fido$

de C_2 y C_5

$C_6 = \{ladra(fido)\}$

y de C_4 y C_6 , la cláusula vacía.

Por tanto, (a) es una consecuencia lógica de los axiomas (b), (c) y (d).

6) Probar que

(a) si x es padre de y e y es padre de z, entonces x es abuelo de z

y (b) todo y tiene un padre x

implican lógicamente

(c) existe un z que tiene un abuelo x

En este caso se tiene:

$$A_1 : \forall x \forall y \forall z \left(P(x,y) \wedge P(y,z) \Rightarrow A(x,z)\right)$$

$$A_2 : \forall y \left(\exists x\, P(x,y)\right)$$

$$B : \exists x \exists z \left(A(x,z)\right)$$

Representando A_1 en forma de cláusulas:

$$C_1 = \{\neg P(x,y), \neg P(y,z), A(x,z)\}$$

Para A_2, eliminamos el cuantificador existencial asociado a la variable x reemplazando ésta por una función de Skolem $x = p(y)$, que puede interpretarse por "$p(y)$ es padre de y". Luego A_2 se convierte en $\forall y \left(P(p(y), y)\right)$ y su cláusula es

$$C_2 = \{P(p(y), y)\}$$

Finalmente, puesto que

$$\neg B = \forall x\, \forall z \left(\neg A(x,z)\right)$$

la cláusula de $\neg B$ es

$$C_3 = \{\neg A(x,z)\}$$

Obtenemos ahora los siguientes resolventes:

$C_4 = \{\neg P(x,y), P(y,z)\}$, de C_1 y C_3

$C_5 = \{\neg P(y,z)\}$, de C_2 y C_4 haciendo $x = p(y)$, que también es $\{\neg P(u,v)\}$, si se cambian y, z por u, v

$C_6 = \{\ \}$, de C_2 y C_5, haciendo $u = p(y)$, $v = z$

6.5 EJERCICIOS

Resuelva los siguientes ejercicios aplicando el principio de resolución por contradicción.

1. Pruebe que R es una consecuencia lógica de los axiomas

   $P \Rightarrow R$

   $Q \Rightarrow R$

   $P \vee Q$

2. Establezca que los axiomas

   $P \Rightarrow Q \vee R$

   $Q \Rightarrow \neg P$

   $S \Rightarrow \neg R$

   implican lógicamente $P \Rightarrow \neg S$

3. Demuestre que:

   a) si María es una verdadera amiga, entonces Juan dice la verdad

   b) si Juan dice la verdad, Luisa no es una verdadera amiga

   c) si Luisa no es una verdadera amiga, entonces no dice la verdad

d) si Luisa no dice la verdad, entonces María es una verdadera amiga

e) si María es una verdadera amiga, Luisa no dice la verdad

implican lógicamente

f) Luisa no dice la verdad

4. Pruebe que

a) los salarios no subirán

es una consecuencia lógica de:

b) si los salarios o los precios suben, habrá inflación

c) si hay inflación, entonces el congreso debe regularla o la gente sufrirá

d) si la gente sufre, el congreso será impopular

e) el congreso no regulará la inflación y no será impopular

5. Determine si

a) algunos son mentirosos

implica lógicamente

b) algunos son ladrones

6. Indique si

a) ninguna persona inteligente que come en exceso también bebe en exceso

b) algunas personas prudentes comen en exceso

implican lógicamente

c) algunas personas prudentes no son inteligentes

7. Demuestre que los axiomas:

$$\forall x \left(\mathrm{M}(x) \Rightarrow \mathrm{A}(x)\right)$$

$$\mathrm{M}(a)$$

$$\mathrm{T}(a)$$

siendo a una constante, implican:

$$\exists x \left(\mathrm{A}(x) \wedge \mathrm{T}(x)\right)$$

8. Pruebe que

$$\forall x \left(\mathrm{P}(x) \wedge \left(\mathrm{Q}(a) \vee \mathrm{Q}(b)\right)\right)$$

en donde se asume que a y b son constantes, implica

$$\exists x \left(\mathrm{P}(x) \wedge \mathrm{Q}(x)\right)$$

9. Compruebe que

(a) si x es padre de y e y es padre de z, entonces x es abuelo de z

(b) todo y tiene un padre

implican lógicamente:

(c) todo x tiene un abuelo

6.6 SOLUCIONES DE TEOREMAS CON CUANTIFICADORES EXISTENCIALES

Si mediante el principio de resolución se establece que los axiomas $A_1, \ldots, A_n$ implican lógicamente B, y B contiene variables con cuantificadores existenciales, es posible determinar las soluciones o valores particulares de estas variables para los cuales el teorema se cumple. En efecto, a cada cláusula de $\neg B$ se agregan las negaciones de sus literales y con las cláusulas de los axiomas se forma un nuevo conjunto respecto del cual se calculan de nuevo los resolventes efectuando la misma secuencia de selección de pares resolubles encontrada en el procedimiento anterior. La cláusula obtenida en el último paso, correspondiente al que antes produjo la cláusula vacía, contiene las constantes -en el lugar de las variables existenciales- que permiten que B sea satisfecho.

EJEMPLO

Los axiomas:

(a) $P(pedro, maria)$

(b) $P(pedro, juan)$

(c) $\forall x \, \forall y \, \forall z \, \left(P(z, x) \Rightarrow H(x, y)\right)$

y el teorema

(d) $\exists x \, \exists y \; H(x, y)$

en donde $P(x, y)$ y $H(x, y)$ indican las relaciones "x es padre de y" y "x e y son hermanos", respectivamente, deseamos hallar una solución de $H(x, y)$.

$C_1 = \{P(pedro, maria)\}$

$C_2 = \{P(pedro, juan)\}$

$C_3 = \{\neg P(z, x), \neg P(z, y), H(x, y)\}$

$C_4 = \{\neg H(u, v)\}$, cláusula de la negación de (d)

Los pasos del algoritmo de resolución son:

1) de C_3 y C_4 : $C_5 = \{\neg P(z, x), \neg P(z, y)\}$

2) de C_1 y C_5: $C_6 = \{\neg P(pedro, maria), \neg P(pedro, y)\}$

3) de C_2 y C_6: $C_7 = \{\ \}$

Ahora hallamos una solución de $H(x, y)$ aplicando los mismos pasos pero esta vez agregamos la negación de $\neg H(u, v)$ a C_4.

$C_1 = \{P(pedro, maria)\}$

$C_2 = \{P(pedro, juan)\}$

$C_3 = \{\neg P(z, x), \neg P(z, y), H(x, y)\}$

$C_4 = \{\neg H(u, v), H(u, v)\}$

Entonces se obtiene:

1) de C_3 y C_4 : $C_5 = \{\neg P(z, x), \neg P(z, y), H(x, y)\}$

2) de C_1 y C_5:

$C_6 = \{\neg P(pedro, maria), \neg P(pedro, y), H(maria, y)\}$

3) de C_2 y C_6: $C_7 = \{H(maria, juan)\}$

y por tanto $H(maria, juan)$ es una solución del teorema.

Capítulo 7

Lenguaje de Programación LISP

7.1 INTRODUCCION

Este capítulo se dedica a presentar los conceptos fundamentales y aplicaciones de LISP, reconocido como un lenguaje de programación apropiado para escribir programas en Inteligencia Artificial.

El lenguaje LISP, cuyo nombre proviene de "List Processing" o procesamiento de listas, fue creado por John Mc Carthy en 1960.

LISP es un lenguaje orientado a procesar símbolos, y asociarles información, de una manera sencilla. Un ejemplo simple es dado por la siguiente expresión en LISP:

```
(pedro (matematica 12 13) (geografia 15) )
```

para representar las calificaciones de Pedro en los cursos de matemática: 12 y 13, y geografía: 15.

Desde luego, esto también puede hacerse por medio de un lenguaje de programación de propósito general como Pascal o C, pero se requiere crear o definir los datos, establecer las relaciones entre ellos, y escribir las funciones que los manipulen.

No existe una versión de LISP definida en forma única o estándar; en verdad hay varios dialectos de LISP, con algunas diferencias entre ellos, tales como: COMMON LISP, INTERLISP y FRANZ LISP.

La presente exposición es realizada según COMMON LISP.

7.2 DATOS BASICOS EN LISP

Los datos básicos en LISP son los *átomos* y las *listas*.

Los átomos pueden ser:

1) números, como 12.34, -45, etc.

2) símbolos, nombres o identificadores, formados por uno o más caracteres: letras, dígitos, y otros símbolos como *, + , -, etc.

   Se excluyen los caracteres:

   el carácter de blanco,

   paréntesis

   punto y coma (;)

   comillas (")

apóstrofos (')

el carácter de grave (`)

el carácter de barra vertical (|)

EJEMPLOS DE ATOMOS SIMBOLICOS

```
PEDRO
edad
*mensaje*
/
resuelve-ecuacion
VALOR1
*
```

LISP no distingue las letras mayúsculas de las minúsculas. Así, los nombres PEDRO y Pedro son iguales o representan el mismo objeto.

Las listas son expresiones encerradas entre paréntesis que contienen cero o más elementos, separados por blancos o paréntesis. Los elementos de una lista son átomos u otras listas.

EJEMPLOS

```
1)  ()                          ; lista vacía  o sin elementos
2)  (pedro juan)                ; lista con dos elementos:
                                ;   los átomos pedro y juan
3)  (+ 12 40 -20)               ; lista con cuatro elementos:
                                ;   los átomos  +, 12, 40 y -20
4)  (pedro  (matematica 12 13) (geografia 15) )
                                ; lista con tres elementos:
```

```
                                ; el átomo pedro
                                ; la lista (matematica 12 13)
                                ; y la lista (geografia 15)
5)  ( () () )                   ; lista con dos elementos:
                                ;   las dos listas vacías () y ()
6)  ((america))                 ; lista con un único elemento:
                                ;   la lista (america)
```

OBSERVACIONES

1. Si dos elementos consecutivos de una lista son átomos, estos deben estar separados por al menos un carácter de blanco (carácter de espacio, tabulador o cambio de línea).

2. Para permitir el procesamiento de textos de información también se admiten átomos (de tipo string o cadena) compuestos por dos o más palabras encerradas por comillas o barras verticales, por ejemplo:

```
"El resultado es "          ; representa un átomo
|Ingrese datos |
("Pedro estudia mucho")     ; lista con el átomo
                            ; "Pedro estudia mucho"
```

3. Se usa el término *S-expresión*, o *expresión simbólica*, para referirse a los datos básicos. Así, una S-expresión es un átomo o una lista.

7.3 EL INTERPRETE DE LISP

El intérprete de LISP es un programa ejecutable encargado de traducir las S-expresiones a instrucciones de la computadora y proceder inmediatamente a ejecutarlas.

El procesamiento básico del intérprete es un ciclo o lazo por el cual se repiten sucesivamente los pasos:

(1) leer una S-expresión

(2) evaluarla

(3) imprimir el valor obtenido

Una vez que el programa intérprete de LISP, cuyo nombre varía según el fabricante, es puesto en ejecución, muestra un carácter especial o "prompt", tal como: "$", ">", "*", etc., para indicarle al usuario que se espera el ingreso de una S-expresión.

EJEMPLOS

```
LISP                          ; pone en ejecución el
                              ; intérprete de nombre LISP

$                             ; LISP imprime carácter
                              ; indicador $ , por ejemplo

$ Pedro <enter>               ; se ingresa el átomo Pedro
PEDRO                         ; (el intérprete de) LISP
                              ; imprime el valor PEDRO

$ (+ 10 30) <enter>           ; se ingresa la lista (+ 10 30)
40                            ; LISP imprime el valor 40

$ (PRINT Pedro) <enter>       ; se ingresa la lista (PRINT Pedro)
PEDRO                         ; efecto de la función PRINT
PEDRO                         ; LISP imprime el
                              ; valor devuelto por PRINT
```

```
$ (SETQ RES (* 2 3) ) <enter>      ; se ingresa la
                                   ; lista (SETQ RES (* 2 3))
6                                  ; LISP imprime el valor dado por SETQ

$ RES <enter>                      ; se ingresa el átomo RES
6                                  ; LISP imprime 6, el valor
                                   ; actual de RES

$ (SYSTEM)  <enter>                ; (QUIT) o (EXIT), dependiendo de
                                   ; la versión de LISP, termina la
                                   ; ejecución del intérprete
```

COMENTARIOS EN LISP

El carácter de punto y coma (;) es utilizado en LISP para escribir comentarios; en efecto, el intérprete ignora los caracteres de la línea actual que siguen a un punto y coma.

NOTA

1. En lo que sigue se asume que $ es el carácter indicador de LISP para leer S-expresiones.

2. Por defecto la lectura de una S-expresión se realiza por el teclado. En este caso, se ingresa el carácter <enter> o fin de línea para indicar el final de los datos.

   LISP siempre lee una S-expresión completa, esto es, si se trata de una lista, no obstante se haya ingresado el carácter <enter> , espera hasta que la lista se cierre con sus respectivos paréntesis.

   La impresión del valor de la S-expresión, por defecto, se hace en la pantalla.

Tanto la lectura de S-expresiones como la impresión de sus valores puede hacerse desde o hacia un archivo, respectivamente.

7.4 EVALUACION DE S-EXPRESIONES

7.4.1 FORMAS

Algunas S-expresiones pueden ser evaluadas y devolver un valor.

Entre tales S-expresiones, que se denominan *formas*, se encuentran los átomos y las listas de tipo procedimiento o función:

$$(\text{nf } \text{arg}_1 \ \text{arg}_2 \ \ldots \ \text{arg}_n)$$

en donde nf es un átomo nombre de una función previamente definida y los elementos arg_1, arg_2, ..., arg_n, son S-expresiones llamadas *argumentos* o *parámetros*, y constituyen los datos que se pasan a la función nf para ser utilizados por ésta.

LISP emplea la notación de prefijo en el manejo de las funciones: el nombre de una función siempre es el primer elemento de la lista que se desea evaluar, independientemente del número de parámetros.

EJEMPLOS

Son listas de tipo función y por tanto pueden ser evaluadas:

```
1) (+ 20 10 5)              ; + es el nombre
                            ; de la función de adición
2) (* (- 20 10) (/ 40 5) )
```

en donde *, - y / son las funciones de multiplicación, sustracción y división de números.

Nótese que estos símbolos aparecen como primer elemento de la lista más interna que lo contiene.

Las siguientes listas no son de tipo función:

```
3) (10 pedro juan)          ; 10 no es nombre de función
4) (calcular 20 100)        ; si el símbolo calcular no ha sido
                            ; definido como nombre de función
```

7.4.2 REGLA DE EVALUACION DE FORMAS

La regla para determinar el valor de una S-expresión F es la siguiente:

(1) Si F es un número (o átomo numérico), su valor siempre es igual a sí mismo, esto es, al número que representa.

EJEMPLO

```
$ 23
  23        ; el valor de 23 es 23
```

(2) Si F es un símbolo (o átomo simbólico), su valor es igual al último valor asignado (con la función SETQ o SET); y si no ha recibido ningún valor, por defecto, su valor es el mismo nombre.

EJEMPLOS

```
$ RES
  RES                    ; valor de RES si no ha
                         ; recibido ningún  valor

$ (SETQ RES  45)         ; asigna a RES el valor de 45
  45

$ RES                    ; valor actual de RES
  45

$ (SETQ RES  -10)        ; asigna a RES el valor -10
  -10

$ (SETQ TEMP RES)        ; asigna a TEMP el valor
                         ; de RES, o sea -10

$ TEMP                   ; imprime el valor actual
  -10                    ; de TEMP
```

3) Si F es una lista de tipo función:

$$(\text{nf } \text{arg}_1 \ \text{arg}_2 \ \ldots \ \text{arg}_n)$$

esto es, el primer elemento nf es un átomo nombre de una función previamente definida, la manera de hallar su valor depende obviamente de las instrucciones dadas en su definición.

No obstante, según la forma como se tratan los argumentos, se distinguen dos clases de funciones: ordinarias y especiales.

Si nf es una función ordinaria, para determinar el valor de F, primero se evalúan todos los argumentos y luego se aplica nf a los valores o resultados obtenidos.

Si nf es una función especial, no son evaluados todos los argumentos, esto significa que nf recibe algunos argumentos -los que no son evaluados- tal cual aparecen en la lista F.

Tanto las funciones ordinarias como las especiales pueden ser predefinidas por LISP, también llamadas primitivas de LISP, o definidas por el usuario.

7.4.3 EJECUCION DE FUNCIONES EN LISP

La acción de evaluar una lista equivale a ejecutar (las instrucciones que componen) la función indicada por su primer elemento.

Es importante hacer notar que cuando se evalúa una lista de tipo función:

$$(\text{nf } \text{arg}_1 \ \text{arg}_2 \ \ldots \ \text{arg}_n)$$

puesto que nf se aplica a los valores de los argumentos arg_i, todas las funciones contenidas en los arg_i siempre se ejecutan antes que nf.

Por ejemplo:

1) al evaluar

(* (+ 3 5) (- 10 7))

el orden de ejecución es: +, - y *

pues para ejecutar * se requiere conocer los valores de (+ 3 5) y (- 10 7).

2) la evaluación de

(* (+ 3 (- 10 7)) 5)

se realiza en el orden: - , + y *

ya que para evaluar * se requieren los valores de (+ 3 (- 10 7)) y 5; y para evaluar + se necesitan los valores de 3 y (- 10 7).

7.4.4 EJEMPLOS

```
1) $ (* 4 5)              ; evaluar la S-expresión $ (* 4 5)
   20                     ; LISP imprime resultado
```

* es la función (ordinaria) de multiplicación; se aplica a los resultados de los argumentos 4 y 5 , respectivamente

```
2) $ (* (+ 3 1)  (- 7 2) )
   20
```

Primero se evalúan los argumentos (+ 3 1) y (- 7 2), obteniéndose los resultados 4 y 5, respectivamente. Luego se aplica * a 4 y 5, los valores obtenidos.

```
3) $ (SETQ  RES (/ 20 5))
     4
```

SETQ es una función especial: Evalúa el segundo argumento (/ 20 5), cuyo valor es 4, y toma el primer argumento RES, sin evaluarlo, para asignarle dicho valor.

```
4) $ (10 pedro juan)
    error
```

(10 pedro juan) no puede ser evaluada pues no es una lista cuyo primer miembro, 10, es nombre de función

```
5) $ (pedro (* 23 5))
    error
```

si se asume que pedro no es nombre de ninguna función

```
6) $ (QUOTE (10 pedro juan))
    (10 pedro juan)        ;su valor es la misma lista
```

7.4.5 EJERCICIOS

1. Determine si cada S-expresión es átomo o lista; si es lista indique sus elementos.

   SUMA

   LISTA

   ((JUAN 10) (ANA 30))

   (/ (* 2 3) (34 10))

   ((ES-MAMIFERO (TIGRE PERRO)) (ES-AVE (LORO POLLO)))

   (SETQ RESULTADO (+ 5 7 8))

```
'VALOR    ;  o (QUOTE VALOR)
((()))
```

2. **Halle los valores de las S-expresiones:**

```
(SUMAR 12 13 5)  ; SUMAR no es una función
SUCESORES        ; átomo sin valor asignado
(* (+ 5 -2) 10)

(MAX 15 8 7 )          ; MAX y MIN son funciones
                       ; que devuelven el mayor
                       ; y el menor de los
                       ; valores de los argumentos,

(MIN 15 8 7 )          ; respectivamente
(MAX (* 2 3) 10 (- 50 20) )
(+ (MAX 20 50) (MIN 100 70))
(/ (+ 10 15 13) 3 )    ; calcular promedio
'CALCULAR
'(CARLOS ANA LUISA)
'( (CARLOS 12 13 16) (ANA 20 17) (LUISA 16 12 18 19) )
```

7.5 FUNCIONES SETQ, QUOTE, EVAL, PRINT y READ

7.5.1 VARIABLES Y FUNCION SETQ

SETQ se usa para asignar nuevos valores a uno o varios átomos simbólicos $var_1, var_2, \ldots, var_n$. La evaluación o ejecución de

$$(\text{SETQ } var_1\ F_1\ var_2\ F_2 \ldots var_n\ F_n)$$

tiene por efecto asignar secuencialmente a cada var_i el valor de F_i.

En este caso los átomos var_i se denominan variables.

Que el proceso de asignación de valores sea secuencial significa que en cada etapa i se actualiza inmediatamente el valor de la variable var_i, de modo que al momento de evaluar una forma posterior F_j que use dicha variable, $i < j$, ésta ya tiene el último valor recibido.

El valor que devuelve SETQ es igual al último valor asignado.

EJEMPLOS

```
$ (SETQ AREA (* 20 10))   ; evaluar (SETQ ...)
                          ; Efecto: asigna a AREA
                          ; el resultado de (* 20 10)
   200                    ; LISP imprime valor de SETQ

 $ AREA                   ; evaluar AREA
   200

$ (SETQ NUM 5 CUADRADO (* NUM NUM))
  25                      ; al evaluar (* NUM NUM)
                          ; la variable
                          ; NUM ya tiene el valor 5

$ NUM
  5

$ CUADRADO
  25

$  (SETQ LIM 23  MENSAJE "Ingrese valor ")
```

```
|Ingrese valor |          ; valor de SETQ
                         ; asigna 23 a LIM y el texto
                         ; "Ingrese valor" a MENSAJE.
$ LIM                    ; valor de LIM
  23
$ MENSAJE                ; valor de MENSAJE
  |Ingrese valor |
```

7.5.2 QUOTE

El valor de (QUOTE F) es la misma S-expresión F, sin evaluarla, esto es, QUOTE es una función que cita o refiere literalmente la expresión S.

En lugar de (QUOTE F) se suele usar la notación abreviada 'F, con el carácter de apóstrofo precediendo a la S-expresión F. Así, por ejemplo, se escribe:

```
'PEDRO
'(ESTA ES UNA LISTA)
```

en lugar de:

```
(QUOTE PEDRO)
(QUOTE (ESTA ES UNA LISTA))
```

respectivamente.

Por medio de la función QUOTE, a una variable se le puede asignar como valor el nombre de un átomo o una lista, sin evaluarlos:

```
$ (SETQ DATOS '(PEDRO JUAN ANA))
  (PEDRO JUAN ANA)

    ; El valor de DATOS es la lista (PEDRO JUAN ANA)

$ (SETQ LL DATOS)        ; asigna a LL valor
                         ; de DATOS, o sea
  (PEDRO JUAN ANA)       ; la lista (PEDRO JUAN ANA)


$ (SETQ A 34)
  34                     ; valor de A es 34


$ (SETQ B A)
  34                     ; valor de B es igual 34,
                         ; el  valor de A

$ (SETQ C 'A)          ; valor de C es igual al (nombre del)
  A                    ; átomo A

$ C                    ; valor de C
  A
```

7.5.3 EVAL

La función predefinida EVAL es una función especial que halla el valor de una S-expresión F mediante: (EVAL F)

para cuyo efecto aplica la regla de evaluación de formas indicada anteriormente.

EVAL es la función más importante en el intérprete de LISP pues es responsable de la traducción de los datos leídos a instrucciones ejecutables en el ciclo básico (paso 2) del intérprete.

De una manera precisa: Cada vez que se lee una S-expresión F

$ F

el intérprete le aplica EVAL.

Durante la ejecución del intérprete, el uso de $ (EVAL F) equivale a ejecutar EVAL dos veces, o sea a aplicar EVAL al valor producido por (EVAL F)

Así, si C es el átomo del último ejemplo:

```
$ (EVAL C)
34
```

Explicación: El intérprete aplica EVAL a la S-expresión (EVAL C); el valor de (EVAL C) es A, y el valor de aplicar EVAL al símbolo A es 34.

Otro ejemplo de EVAL es el siguiente:

```
$ (SETQ LSUMA '(+ 10 20))    ; asigna lista (+ 10 20) a
                             ; variable LSUMA

$ LSUMA                      ; valor de LSUMA
(+ 10 20)                    ; es la lista (+ 10 20)

$ (EVAL LSUMA)               ; evaluar dos veces
                             ; primera vez, (EVAL LSUMA),
                             ; su valor es (+ 10 20) ;
                             ; segunda vez, aplicar EVAL
                             ; del intérprete a lista anterior
 30                          ; imprime resultado final
```

7.5.4 FUNCIONES DE ENTRADA Y SALIDA DE DATOS: PRINT Y READ

La función (PRINT F) imprime el valor de una S-expresión F y devuelve dicho valor.

EJEMPLOS

```
$ (PRINT PEDRO)        ; evaluar (PRINT ...)
  PEDRO                ; efecto de PRINT
  PEDRO                ; LISP imprime el valor de PRINT

$ (PRINT (* 10 30))
  300
  300

$ (PRINT "MARIA LUISA")
  |MARIA LUISA|
  |MARIA LUISA|
```

La función (READ) lee una S-expresión completa sin evaluarla, es decir, su valor es exactamente la expresión leída.

EJEMPLOS

```
$ (READ)               ; evaluar (READ): leer dato
12                     ; se ingresa 12
12                     ; LISP imprime valor
                       ; devuelto por READ

$ (SETQ LL (READ))     ; evaluar (SETQ ...); primero se evalúa
                       ; (READ) : leer dato

(PEDRO JUAN ANA)       ; se ingresa lista (PEDRO JUAN ANA)
                       ; READ devuelve la lista
                       ; leída y SETQ la
```

```
                              ; asigna a la variable LL
(PEDRO JUAN ANA)              ; LISP imprime valor de LL

$ LL                          ; evaluar LL

(PEDRO JUAN ANA)

$ (SETQ BASE (READ) ALTURA (READ) )
 20 30                        ; datos ingresados
 30                           ; valor de SETQ

$ BASE
 20

$ ALTURA
 30

 $ (* BASE ALTURA)            ; evaluar (* BASE  ALTURA): primero
                              ; se  hallan valores
                              ; de BASE y ALTURA;
                              ; luego se aplica * a tales
                              ; valores: 20 y 30
  600

$ (SETQ AREA (* BASE ALTURA))  ; equivale a
                                    ; AREA = BASE * ALTURA
 600

$ AREA
 600
```

7.5.5 EJERCICIOS

Las siguientes S-expresiones son leídas sucesivamente por el intérprete de LISP:

```
$ (SETQ NUMERO 1)
$ (SETQ NUMERO (+ NUMERO 1) )
```

```
$ NUMERO
$ 'NUMERO
$ (* NUMERO 2)
$ (PRINT (+ 20 10))
$  (PRINT '(+ 20 10))
$ '(PRINT (+ 20 10))
$ (PRINT HOLA)
$ '(PRINT HOLA)
$ (SETQ DISTANCIAS  '((A B 10)  (A C 20) (B D 100)) )
$ (READ)                ; ingrese la lista (LA MESA ES NUEVA)
$ (READ)                        ; ingrese el número 12.30
$ (EVAL (READ))                 ; ingrese la lista (+ 10 20 100)
$ (SETQ LISTA  (READ) )      ; ingrese la lista
                               ;    (+ 10 20 100)
$ LISTA
$ (EVAL LISTA)
```

Explique en cada caso el efecto del proceso de evaluación y halle el valor resultante.

7.6 NOTACION

Para facilitar la descripción de las funciones, en lo que sigue, a menos que se mencione lo contrario, se supone que los argumentos de las funciones siempre se evalúan. Por abuso de lenguaje se dice que un argumento es un número, símbolo, átomo o lista, para indicar que ésta es la clase de valor que dicho argumento ha de tener después de ser evaluado.

A modo de ejemplo, si se especifica que en la función nf

```
(nf  S  L  N)
```

los argumentos son un símbolo S, una lista L y un número N, se sobreentiende que luego de evaluar estas expresiones, los valores resultantes han de ser un símbolo, una lista y un número, respectivamente.

7.7 FUNCIONES MATEMATICAS

Si $n_1, n_2, \ldots, n_m$ son números, las funciones:

(+ n_1 n_2 ... n_m)

(- n_1 n_2 ... n_m)

(* n_1 n_2 ... n_m)

(/ n_1 n_2 ... n_m)

(REM n_1 n_2)

devuelven la suma, diferencia, producto y cociente de tales números, respectivamente, esto es, los valores:

$n_1 + n_2 + \ldots + n_m$

$n_1 - n_2 - \ldots - n_m$

$n_1 * n_2 * \ldots * n_m$

$n_1 / n_2 / \ldots / n_m$

residuo de la división de n_1 entre n_2

EJEMPLOS

```
$ (+ 12 3 2)
  17
```

```
$ (- 12 3 2)                          ; 12 - 3 - 2
  7

$ (* 12 3 2)
  72

$ (/ 12 3 2)                          ;  12/3 es 4 y 4/2 es igual a 2
  2

$ (REM 30 7)                          ; residuo de 30/7
  2

$ (*  (+ 9 1) (- 10 6 2) (/ 20 4) )
  100

$  (SETQ A (- 10 4)  B (* 6 14))     ; A y B reciben los valores
  84                                  ; 6 y 84, respectivamente

$ (/ B A))                            ; se divide 84 entre 6
  14
```

7.8 FUNCIONES BASICAS DE PROCESAMIENTO DE LISTAS

Describimos algunas funciones de LISP que se utilizan para procesar listas.

7.8.1 CAR y CDR

Si L es una lista $(e_1\ e_2 \ldots e_n)$, entonces (CAR L) devuelve el primer elemento e_1 y (CDR L) devuelve la lista $(e_2 \ldots e_n)$, que resulta de suprimir el primer elemento.

EJEMPLOS

```
$ (CAR '(MARIA JUAN CARLOS))
 MARIA

$ (CDR '(MARIA JUAN CARLOS))
  (JUAN CARLOS)

$ (SETQ LL  '(MARIA JUAN CARLOS))
  (MARIA JUAN CARLOS)

$ (SETQ MARIA 15)
  15

$ (CAR LL)
  MARIA

$ (EVAL (CAR LL))
  15

$ (CDR LL)
  (JUAN CARLOS)

$ (SETQ FECHA '(12 5 1990) EDAD 20)
  20

$ (SETQ DATOS '(FECHA EDAD))
  (FECHA EDAD)

$ (CAR DATOS)
   FECHA

$ (EVAL (CAR DATOS))        ; o (EVAL FECHA)
  (12 5 1990)

$ (CDR  '(MANZANA))         ; si la lista objeto
                            ; tiene un solo elemento
  ()                        ; CDR devuelve la lista vacía
```

Aplicando sucesivamente varios CAR y CDR se pueden extraer los siguientes elementos y sublistas. Por ejemplo:

```
$ (CAR (CDR '(a b c d)))        ; obtiene segundo elemento
  B
```

Explicación: CDR devuelve la lista (b c d) y CAR de esta lista es B

```
$ (CDR (CDR '(a b c d))
  (C D)

$ (CAR (CAR DATOS))
  FECHA

$ (CAR (CDR (CAR (CDR DATOS))))
  20
```

Explicación:

(CDR DATOS) es ((EDAD 20)),

CAR de ((EDAD 20)) es (EDAD 20)

CDR de (EDAD 20) es (20)

y CAR de (20) es 20

7.8.2 COMPOSICION DE FUNCIONES CAR Y CDR: CXXR, CXXXR, CXXXXR

LISP provee funciones de nombres CXXR, CXXXR, CXXXXR, en donde X es la letra A (por CAR) o D (por CDR), que equivalen a aplicar sucesivamente varios CAR y CDR, y por tanto simplifican la escritura de esta clase de expresiones.

Utilizando tales funciones, los ejemplos anteriores pueden escribirse así:

```
$ (CADR '(a b c d))                ; equivale a
                                   ; (CAR (CDR '(a b c d)))
  B

$ (CDDR '(a b c d))  ; (CDDR ...) equivale a (CDR (CDR ...) )
  (C D)

$ (CAAR DATOS)
  FECHA

$ (CADADR DATOS)
  20
```

7.8.3 LIST, CONS, APPEND, REVERSE, LAST

La función (LIST e_1 e_2 ... e_n) devuelve la lista (e_1 e_2 ... e_n) formada por los elementos e_i.

EJEMPLOS

```
$ (LIST A B 12)                    ; A y B son símbolos
                                   ; sin valores asignados
  (A B 12)

$ (LIST A)
  (A)

$ (SETQ A 8 B 10)                  ; se asignan valores a A y B

$ (LIST A B 12)                    ; LIST devuelve lista
                                   ; con los valores de A, B y 12
  (8 10 12)
```

```
$ (LIST 'A 'B 12)                    ; ' evita que se evalúen A y B
  (A B 12)

$ (LIST  '(A B)  '(C D) )
  ( (A B) C D))

$ (LIST 5 '(1 2 3) 7)
  (5 (1 2 3) 7)

$ (SETQ TABLA '(3 4 5))
  (3 4 5)                            ; valor de TABLA

$ (LIST TABLA TABLA)
  ((3 4 5) (3 4 5))                  ; lista con dos elementos
```

Si el valor de L es la lista $(e_1\ e_2 \ldots e_n)$, y el valor de X es x, la función (CONS X L) devuelve la lista

$$(x\ e_1\ e_2 \ldots e_n)$$

obtenida agregando x, como primer elemento, a la lista anterior.

EJEMPLOS

```
$ (CONS JUAN '(RAMOS))     ; JUAN es un símbolo sin valor
  (JUAN RAMOS)             ; asignado

$ (CONS '* '(8 12))
  (* 8 12)

$ (CONS (+ 5 7) '(10 20 30))   ; evalúa (+ 5 7)
  (12 20 30))

$ (SETQ DATOS '(10 20 30))
  (10 20 30)
```

```
$ (CONS '+ DATOS)
  (+ 10 20 30)
```

APPEND se utiliza para formar una lista con los elementos de otras listas. De una manera precisa, si los valores de $L_1, L_2, \ldots, L_n$ son listas, entonces el valor de

$$(\text{APPEND } L_1, L_2, \ldots, L_n)$$

es la lista formada por los elementos de la primera lista, seguidos por los elementos de la segunda lista, y así sucesivamente.

EJEMPLOS

```
$ (APPEND '(A B) '(P Q R))
  (A B P Q R)

$ (APPEND '(+) '(4 3 2) '(-5  8) )
  (+  4  3  2  -5  8)

$ (SETQ LNUM '(40 30 10))
  (40 30 10)

$ (APPEND LNUM LNUM)
  (40 30 10 40 30 10)
```

REVERSE invierte el orden de los elementos de una lista: Si L es la lista $(e_1\, e_2 \ldots e_n)$, entonces el valor de (REVERSE L) es la lista $(e_n\, e_{n-1} \ldots e_2\, e_1)$.

Y (LAST L) devuelve la lista (e_n), formada por el último elemento.

EJEMPLOS

```
$ (REVERSE '(A B C))
 (C B A)

$ (LAST '(A B C))
  (C)

$ (SETQ TABLA '(10 8 6 4 2))
 (10 8 6 4 2)

$ (LAST TABLA)
 (2)

$ (REVERSE TABLA)
 (2 4 6 8 10)
```

7.8.4 EJERCICIOS

1. Determine CAR y CDR de cada una de las siguientes listas:

   (A B C)
   ((A 3) (B 4) (C 5))
   (+ (* 10 2 (- 100 89)))
   (()())
   (SETQ V (* NUM 3))

2. Para cada caso siguiente indique una función que al aplicarla a la lista (A B C D) devuelva:

   1) el tercer elemento C
   2) la lista (C D)

3) la lista (B A)
4) ((A) (B C D))

3. Evalúe sucesivamente las siguientes formas:
1) (LIST 'A 'B 'C)
2) (LIST '(A) '(B) 'C)
3) (CONS '(A) '(A B))
4) (APPEND '(1 2 3) (4 5))
5) (SETQ DATO1 '(ANA 17))
6) (SETQ DATO2 '(FRANCISCO 13))
7) (SETQ ACTA (LIST DATO1 DATO2))
8) (LAST ACTA)
9) (CONS '(MARIA 14) ACTA)
10) (APPEND '((MARIA 14)) ACTA)
11) (REVERSE ACTA)
12) (CAAR (REVERSE ACTA))

7.9 DATOS LOGICOS EN LISP

7.9.1 SIMBOLOS ESPECIALES NIL, T

En LISP se usan los símbolos predefinidos NIL y T para representar los valores falso y verdadero, respectivamente.

Los valores de ambos símbolos son sus mismos nombres. En general cualquier valor distinto de NIL se considera verdadero.

Por convenio, NIL y la lista vacía () representan el mismo objeto.

7.9.2 PREDICADOS O FUNCIONES LOGICAS

Las funciones que sirven para comprobar si se cumple o no una determinada condición se denominan funciones lógicas o predicados.

Un predicado devuelve el valor NIL cuando la condición es falsa y un valor distinto de NIL, por ejemplo T, si la condición es verdadera.

A continuación presentamos algunos predicados e indicamos las condiciones que deben cumplirse para que devuelvan un valor verdadero; en caso contrario, retornan el valor NIL.

1) (ATOM F)

devuelve T si F es un átomo

EJEMPLOS

```
$ (ATOM 3)
 T

$ (ATOM 'A)
 T

$ (ATOM '(A B C))
 NIL
```

2) (LISTP F)

su valor es T si F es una lista

EJEMPLOS

```
$ (LISTP '(A B C))
 T
```

```
$ (SETQ X 'A)                    ; valor de X es el átomo A
  A
$ (SETQ L '(A B C))              ; valor de L es la lista (A B C)
  (A B C)
$ (ATOM X)
  T
$ (ATOM L)
  NIL
$ (LISP X)
  NIL
$ (LISTP L)
  T
$ (ATOM NIL)                     ; NIL es átomo o lista (vacía)
  T
$ (LISTP NIL)
  T
```

3) (NULL F)

retorna T si F es la lista vacía

EJEMPLOS

```
$ (NULL '(1 2 3))
  NIL                            ; (1 2 3) no es la lista vacía
$ (SETQ V '())                   ; el valor de V es la lista vacía
  NIL
$ (NULL V)
  T
```

4) (NUMBERP F)

devuelve T si F es un número

EJEMPLOS

```
$ (NUMBERP 5)
  T

$ (SETQ N1 (* 4 3))
  12

$ (NUMBERP N1)
  T
```

5) (EQUAL F G)

su valor es T si F y G tienen iguales valores

EJEMPLOS

```
$ (EQUAL 12 (* 3 4))
  T

$ (SETQ DATOS '(A B C))
  (A B C)

$ (EQUAL DATOS '(A B C))
  T
```

6) (MEMBER E L 'EQUAL)

si E es un elemento de la lista L, esta función devuelve una lista como valor verdadero: la porción final de L que empieza en el primer E.

NOTA

En la presente exposición sólo se hará uso de la función MEMBER con la especificación 'EQUAL, que determina que la función se ejecuta según la descripción dada.

EJEMPLOS

```
$ (MEMBER 'A '(B A B A) 'EQUAL)
  (A B A)                    ; verdadero

$ (MEMBER 'X '(B A B A) 'EQUAL)
  NIL                        ; falso

$ (SETQ ELEMENTO '(1 2))
  (1 2)

$ (SETQ DATOS '( (10 5) 6 (1 2)  7) )
  ((10 5) 6 (1 2) 7)

$ (MEMBER ELEMENTO DATOS 'EQUAL)
  ((1 2) 7)
```

7) (SIMBOLP F)

devuelve T si F es un símbolo

EJEMPLOS

```
$ (SYMBOLP 'A)
  T

$ (SETQ A 3)
  3

$ (SYMBOLP A)
  NIL
```

8) (FUNCTIONP F)

devuelve T si F es un nombre de función

EJEMPLOS

```
$ (FUNCTIONP 'A)
 NIL                          ; si A no es nombre de función

$ (SETQ MAS '+)
 +

$ (FUNCTIONP MAS)
 T

$ (FUNCTIONP 'SETQ)
 T
```

Los siguientes predicados se aplican a argumentos numéricos $n_1, n_2, \ldots, n_m$.

9) (= n_1 n_2 ... n_m)

devuelve T si todos los números son iguales

10) (/= n_1 n_2 ... n_m)

su valor es T si existe un par de números distintos

11) (< n_1 n_2 ... n_m)

devuelve T si cada número es menor que el siguiente

12) (> n_1 n_2 ... n_m)

devuelve T si cada número es mayor que el siguiente

13) (<= n_1 n_2 ... n_m)

retorna T si cada número es menor o igual que el siguiente

14) (>= n_1 n_2 ... n_m)

retorna T si cada número es mayor o igual que el siguiente

EJEMPLOS

```
$ (= 12 (* 3 4) (/ 24 2))
  T

$ (/= 12 (- 15 1))
  T                              ; 12 y 14 son distintos

$ (SETQ A (* 3 2) B (+ 5 8))     ; A vale 6 y B vale 13
  13

$ (< A B 20)
  T                              ; 6 es menor que 13 y 13
                                 ; es menor que 20

$ (>= A 10)
  NIL
```

7.9.3 OPERACIONES LOGICAS

Estas son NOT (negación), AND (conjunción) y OR (disyunción), y se usan para combinar condiciones.

1) (NOT F)

devuelve T (verdadero) si el valor de F es NIL

2) (AND F_1 F_2 ... F_m)

devuelve NIL si hay al menos una F_i con valor NIL; en caso contrario devuelve el valor de F_m.

AND termina en el primer F_i que tenga valor NIL.

3) (OR F_1 F_2 ... F_m)

devuelve NIL si todas las F_i tienen valor NIL; en caso contrario, OR termina y devuelve el resultado del primer F_i que resulte con valor no NIL.

EJEMPLOS

```
$ (NOT 3)
 NIL

$ (NOT NIL)
 T

$ (NOT (ATOM '(A B))
 T                              ; pues (ATOM '(A B))
                                ; tiene valor NIL

$ (SETQ X 10)
 10

$ (AND (> X 9 ) (< X 20))      ; prueba si X es mayor que 9
 T                              ; y menor que 20

$ (AND (NUMBERP X) (* X 3)) ; si X es número,
                                ; multiplicarlo por 3
 30

$ (OR (SYMBOLP X) (LISTP X))
 NIL                            ; pues el valor 10 de X
                                ; no es un símbolo
```

```
                                        ; ni una lista
    $ (SETQ  ANIMAL '(PERRO GATO POLLO))
```

A continuación se prueba si LORO o GATO pertenecen a la lista ANIMAL:

```
$ (OR (MEMBER 'LORO ANIMAL 'EQUAL) (MEMBER 'GATO
ANIMAL 'EQUAL))

  (GATO POLLO)
```

7.9.4 EJERCICIOS

Determine los valores de cada una de las siguientes expresiones

```
1) (SETQ A 10 B 20 C 5)
2) (SETQ LL '(A B (A B)))
3) (ATOM A)
4) (NUMBERP A)
5) (LISTP LL)
6) (SYMBOLP A)
7) (NUMBERP (CAR LL))
8) (MEMBER '(A B) LL 'EQUAL))
9) (MEMBER A (LIST 30 10 5 6) 'EQUAL)
10) (EQUAL A (CAR LL))
11) (EQUAL 'A (CAR LL))
12) (< C A B)
13) (AND (<= A B) (> C B))
14) (AND (LISTP LL) (NOT (NULL LL)) (CAR LL) )
      ; si LL no es la lista vacía, obtener su primer elemento A !
15) (SETQ PRIMERO (AND (LISTP LL) (NOT (NULL LL)) (CARLL) ) )
```

7.10 OTRAS FUNCIONES DE LISP

7.10.1 FUNCIONES COMO ARGUMENTOS: FUNCALL, APPLY, MAPCAR

El nombre nf de una función puede ser usado como argumento de las funciones FUNCALL, APPLY y MAPCAR, de manera que al ser evaluadas ejecutan la función nf.

El valor de

(FUNCALL nf a_1 a_2 ... a_n)

o (APPLY nf lista)

en donde lista se compone de los elementos $a_1, a_2, \ldots, a_n$, es igual al valor de (nf a_1 a_2 ... a_n).

Al evaluar

(MAPCAR nf L_1 L_2 ... L_n)

en donde cada L_i es una lista (p_i s_i ...), se retorna una lista (v_1 v_2 ...) siendo:

v_1 igual al valor de aplicar nf a los primeros elementos de las listas: (nf p_1 p_2 ... p_k),

v_2 igual al valor de nf en los segundos elementos: (nf s_1 s_2 ... s_k),

y así sucesivamente.

EJEMPLOS

```
$ (FUNCALL '+ 1 2 3)
  6
```

```
$ (APPLY '+ '(1 2 3))
  6

$ (MAPCAR '+  '(1 2) (10 20) )
  (11 22)

$ (SETQ SUMAR '+  A 1  B 2  C 3  LL '(1 2 3))
  (A B C)                          ; asigna valores a
                                   ; SUMAR, A, B, C y LL

$ (FUNCALL SUMAR A B C)
  6

$ (APPLY SUMAR LL)
  6

$ (MAPCAR SUMAR LL LL LL)
  (3 6 9)

$ (SETQ LISTA '(A 50 60 B))

$ (MAPCAR 'NUMBERP LISTA)          ; determina los
                                   ; elementos de LISTA
  (NIL T T NIL)                    ; que son números
```

7.10.2 EJECUCION DE UN BLOQUE DE FORMAS: PROGN

Mediante

$$(\text{PROGN } F_1 \; F_2 \ldots F_k)$$

se forma un bloque o grupo de formas $F_1, F_2, \ldots, F_k$, que se evalúan una a continuación de otra.

El valor de PROGN es el de la última forma.

EJEMPLOS

```
$ (PROGN (PRINT 'HOLA) (SETQ A 5))
  HOLA
  5

$ (PROGN
        (PRINC "Ingrese numero ")
        (SETQ NUM (READ))
        (PRINC (* NUM 2))
        (TERPRI)
)

Ingrese numero 20
40
NIL
```

EXPLICACION:

Cuando se ejecuta la primera forma, PRINC imprime el texto "Ingrese numero ", sin cambio de línea; luego la segunda forma efectúa la lectura de un dato, a causa de READ, para asignarlo a NUM; en este caso se ha ingresado 20 y PRINC de la tercera forma imprime el doble de NUM; finalmente TERPRI imprime un cambio de línea y devuelve NIL, que es el valor de PROGN.

En el ejemplo dado se han usado las funciones

TERPRI, que imprime un cambio de línea (avanza la posición de impresión al comienzo de la siguiente línea) y devuelve NIL;

y PRINC F, que al igual que PRINT F imprime el valor de F, pero sin seguirlo por un cambio de línea.

Para imprimir un número n de cambios de líneas se puede usar

(TERPRI n)

7.10.3 LISTAS ASOCIATIVAS: ASSOC

Una lista asociativa es una lista de la forma:

$$((c_1\ a_1)\ (c_2\ a_2)\ldots)$$

cuyos elementos c_i son pares, esto es, listas con dos elementos, en donde v_i representa un valor o atributo particular asociado al objeto c_i, que se denomina campo o llave.

EJEMPLOS DE LISTAS ASOCIATIVAS

1) La lista

((NOMBRE PEDRO) (SEXO MASC) (EDAD 18))

indica los atributos PEDRO, MASC y 18 asociados a los campos NOMBRE, SEXO y EDAD, respectivamente.

2)

((FRUTAS (MANZANA NARANJA)) (ANIMALES (PERRO GATO LORO)))

En este caso el valor del campo FRUTAS es la lista

(MANZANA NARANJA)

3)

((FECHA (12 05 87)) (CURSOS (HISTORIA LENGUAJE)))

4) En la lista

```
( ((PEDRO ANA) (CARLOS LUISA TOMAS) )
  ((JAIME MARIA) (JOSE))
)
```

el campo (PEDRO ANA) tiene el atributo (CARLOS LUISA TOMAS) y puede describir la relación: Los hijos de PEDRO y ANA son CARLOS, LUISA y TOMAS.

La función

(ASSOC c LA)

busca el primer par (c_i v_i) de la lista asociativa LA cuyo campo c_i coincida con el valor de c, y en tal caso devuelve dicho par; de otra manera, retorna NIL.

NOTA

En la definición anterior se asume que el campo c_i es un átomo.

Para buscar campos arbitrarios, sean átomos o listas, se puede emplear

(ASSOC c LA 'EQUAL)

con la especificación 'EQUAL.

EJEMPLOS

1)
```
$ (ASSOC 'NOMBRE '((NOMBRE PEDRO) (SEXO MASC) (EDAD 18) ) )
(NOMBRE PEDRO)
```

2)
```
$ (SETQ LA '((B 3) (A 5) (C 7) (A 10))
    ((B 3) (A 5) (C 7) (A 10))
```

```
3) $ (SETQ CAMPO 'A)
     A

4) $ (ASSOC CAMPO LA)
     (A 5)

5) $ (SETQ LHIJOS
          '( ((PEDRO ANA) (CARLOS LUISA TOMAS) )
            ((JAIME MARIA) (JOSE))
          )
     )

6) $ (SETQ PADRES2 '(JAIME MARIA))

7) $ (ASSOC PADRES2 LHIJOS 'EQUAL)
    ((JAIME MARIA) (JOSE)) ; retorna el par formado por el
                           ; campo y el valor asociado
```

7.11 FUNCIONES DEFINIDAS POR EL USUARIO

7.11.1 DEFINICION DE FUNCIONES: DEFUN

Una función es un nombre nf que se define con DEFUN mediante una lista:

(DEFUN nf (p_1 p_2 ... p_m)

F_1

F_2

...

F_n

)

en donde:

nf es el nombre que se da a la función

$(p_1\ p_2 \ldots p_m)$ es una lista de símbolos $p_1, p_2, \ldots, p_m$, llamados parámetros de nf,

y $F_1, F_2, \ldots, F_n$, son las formas componentes de nf, y constituyen el bloque o cuerpo de instrucciones de la función

Cuando el intérprete de LISP evalúa esta lista pone en memoria la definición de nf y retorna su nombre. A partir de este momento la función está disponible para ser evaluada o ejecutada.

7.11.2 PROCESO DE EVALUACION

Cuando se evalúa (nf $arg_1 \ldots arg_n$), siendo arg_i los argumentos que se pasan a nf, tiene lugar el siguiente proceso:

1) se evalúa cada arg_i y se copia su valor en el respectivo parámetro p_i
2) se ejecuta o evalúa sucesivamente cada forma F_i
3) la función nf retorna el valor de la última forma.

NOTA

1. Usualmente cada F_i es a su vez otra función nf_i

$$(nf_i \ldots)$$

2. Obsérvese que, con excepción del valor de la última forma, no son tomados en cuenta los valores de las

formas. La evaluación de estas formas se hace exclusivamente por sus efectos laterales, o sea por las acciones o tareas que ellas realizan.

3. Los símbolos p_i son variables locales a nf, esto es, desaparecen cuando termina la ejecución de la función. Sin embargo, mientras se ejecuta la función, pueden ser usados y modificados por las formas F_i, inclusive en sus mismos bloques.

   Como se verá más adelante, también se definen variables locales dentro de bloques construidos con las instrucciones LET y DO.

   Las variables no locales, llamadas variables globales, pueden recibir valores en cualquier función, pero a diferencia de las anteriores éstas retienen los últimos valores asignados.

EJEMPLOS DE FUNCIONES

1) La siguiente función calcula el área de un rectángulo en términos de la base y la altura.

```
$ (DEFUN  AREA (B A) (* B A)) ; fin de función AREA
  AREA
```

EXPLICACION: DEFUN define la función AREA con dos parámetros B y A. El cuerpo de la función se compone de la forma (* B A), cuyo valor es el que AREA ha de devolver cuando se evalúe.

```
$ (AREA  10 20)          ; ejecuta la función AREA
  200                    ; valor devuelto
```

2)

```
$ (DEFUN ULTIMO          ; función que devuelve el
                         ; ultimo elemento de una lista X,
   (X)                   ; lista con un parámetro
   (CAR (LAST X))        ; cuerpo de la función
  )                      ; fin de función

ULTIMO

$ (ULTIMO '(A B C))
  C

$ (DEFUN DOBLE ()        ; lista de parámetros es vacía
    (PRINC "Ingrese numero ")     ; bloque
    (SETQ NUM (READ)))
    (SETQ DNUM (* NUM 2) )
    (PRINC "El doble es ")
    (PRINC DNUM)
    (TERPRI)
  )          ; fin de función


  DOBLE

 $ (DOBLE)
  Ingrese numero 10      ; se ingresa 10
  El doble es 20
  NIL
```

Puesto que NUM y DNUM no son variables locales retienen sus valores finales:

```
$ NUM
  10

$ DNUM
  20
```

4)

```
$ (DEFUN PAR2  (X Y) (LIST X Y) ) ; define función
   PAR2                           ; que forma una lista con los
                                  ; elementos

$ (PAR2 'A 'B)
  (A B)
```

5) La función EMPAREJAR retorna la lista de pares $(x_i\ y_i)$:

$$((x_1\ y_1)\ (x_2\ y_2)\ldots)$$

cuando recibe como argumentos dos listas $(x_1\ x_2\ \ldots)$ e $(y_1\ y_2\ \ldots)$.

```
$ (DEFUN EMPAREJAR-LISTAS
    (L1 L2)                       ; parámetros
    (MAPCAR 'PAR2 L1 L2)          ; MAPCAR aplica
                                  ; PAR2 a los
                                  ; respectivos elementos
                                  ; de L1 y L2
  )                               ; fin de función

  EMPAREJAR-LISTAS

$ (EMPAREJAR-LISTAS '(A B C) '(1 2 3))
  ((A 1) (B 2) (C 3))
```

7.11.3 FUNCIONES CUYO PARAMETRO ES UNA LISTA

También se puede definir una función mediante

(DEFUN nf pl

F_1

F_2

...

F_n

)

con un símbolo o nombre pl, en lugar de una lista, para indicar que nf será una función cuyo parámetro es una lista de valores.

En este caso, cuando se evalúa la función

(nf arg_1 arg_2 ... arg_n)

el parámetro pl recibe la lista formada por los valores de los argumentos arg_i.

Esta definición permite considerar funciones que reciben un número arbitrario de argumentos.

EJEMPLOS

```
1) $ (DEFUN  MUESTRA-ARG L
        (PRINC L)
        (TERPRI)
    )    ; fin de función

   MUESTRA-ARG
```

```
$ (MUESTRA-ARG 2 (* 3 4) 4 6)
  (2 12 4 6)
  NIL

2) $ (DEFUN PENULTIMO LARG
      (CADR (REVERSE LARG))
     )

     PENULTIMO

     $ (PENULTIMO 'A 'B '(B C) 'U)
       (B C)
```

7.11.4 VARIABLES LOCALES : LET

LET permite definir un conjunto de variables locales dentro de un bloque de formas:

```
(LET
  ( (v1 s1) (v2 s2) ... (vm sm) ); lista de variables locales
     F1        ; bloque de formas
     F2
     ...
     Fn
)
```

Efecto: Cada v_i recibe simultáneamente (o en paralelo) el valor de la respectiva expresión s_i; luego se ejecutan sucesivamente F_1, F_2, ...

El valor de LET es el de la última forma.

Las variables v_i son locales al bloque asociado a LET.

Si no se requiere que una variable v_i reciba un valor inicial, basta escribir (v_i) en lugar de $(v_i\ s_i)$ en la lista de variables locales.

EJEMPLO

```
$ (DEFUN DOBLE ()              ; lista de parámetros es vacía
   (LET
        ((NUM) (DNUM))         ; lista de variables locales a LET,
                               ; sin valores iniciales

        (PRINC "Ingrese numero ") ; bloque
        (SETQ NUM (READ)))
        (SETQ DNUM (* NUM 2) )
        (PRINC "El doble es ")
        (PRINC DNUM)
   )                           ; fin de LET
   (TERPRI)
)                              ; fin de función

DOBLE

$ (DOBLE)
  Ingrese numero 30
  El doble es 60
  NIL
```

7.12 INSTRUCCIONES DE CONTROL: COND, LOOP, DO

A continuación describimos las funciones COND, LOOP y DO que modifican el proceso de ejecución o evaluación de formas.

7.12.1 COND

Es una instrucción de selección múltiple. Se usa así:

(COND

L_1

...

L_n

)

siendo los L_i listas, en donde se tratan los distintos casos de selección.

El efecto de la función COND es el siguiente:

Selecciona la primera lista L_i cuyo primer miembro es verdadero (o no NIL) y procede a evaluar secuencialmente todos los miembros de L_i devolviendo el valor de su último elemento. Las restantes listas son ignoradas.

COND devuelve NIL si los primeros miembros de todas las listas tienen valores NIL.

Usualmente las listas L_i, con la posible excepción de la última, son expresiones de la forma:

((fp_i ...) $resto_i$)

que empiezan con un predicado de prueba o test : (fp_i ...), o sea una lista de tipo función cuyo valor, verdadero o falso (NIL), se usa para decidir si L_i es la lista a seleccionar:

Si (fp_i ...) es verdadero, ejecutar $resto_i$.

EJEMPLOS DE COND

1)

```
$ (DEFUN MAX2 (A B)     ; define función MAX2
    ( COND
        ( (> A B)  (PRINC A) ) ; si A >B imprime A
        ( 'T       (PRINC B) ) ; de lo contrario, imprime B
    )
  (TERPRI)
) ; fin de MAX2

$ (MAX2 30 20)
 30
 NIL
```

2) Se dice que una función se ejecuta recursivamente, o efectúa llamadas recursivas, si una de las formas de su bloque contiene a dicha función.

Presentamos un ejemplo de una función F que devuelve el valor del factorial de n, igual al producto de los números

1, 2, ... , n

La ecuación F(n) = n * F(n - 1), si n > 0, permite hallar el valor de F(n) en términos de F(n - 1), y da lugar a la si-

guiente versión de la función factorial con llamadas recursivas.

```
$ (DEFUN F (n)
        (COND
          ( (<= n 0) 1 )
          ( 'T
            (* n (F (- n 1) ) )     ; multiplica n por F(n-1)
                                    ; la función F se llama a sí
                                    ; misma pero ahora con el
                                    ; argumento n-1
          )
        ) ; fin de COND             ; F retorna con el valor de COND
 ); fin de F

 F

$ (F 3)
  6

$ (F 4)
  24
```

3) La siguiente función LONGITUD, que también es definida recursivamente, calcula el número de elementos de una lista.

```
$ (DEFUN LONGITUD (L)
      (COND
        ((NULL L) 0)            ; si la lista es vacía,
                                ; devolver 0

        ('T                     ; de otra manera, calcular la
          (+ 1 (LONGITUD (CDR L)))  ; longitud de la lista sin
```

```
        )                          ; el primer elemento (recursión)
                                   ; y sumarle 1
    ) ; fin de COND

) ; fin de LONGITUD

LONGITUD
$ (LONGITUD '(a b c))
 3
```

NOTA:

La función LONGITUD es equivalente a la función LENGTH de LISP, esto es, (LONGITUD L) y (LENGTH L) devuelven los mismos valores.

7.12.2 LOOP y DO

Las intrucciones LOOP y DO ejecutan cero o más veces un bloque de formas.

LOOP se usa así:

```
(LOOP
   L1
   ...
   Ln
)
```

siendo L_i listas $(p_i \ldots)$ miembros del bloque de LOOP.

Efecto: LOOP repite el ciclo de instrucciones

procesar L_1

...

procesar L_n

El procesamiento de una lista L_i, que depende de su cabeza o primer elemento p_i, es el siguiente:

CASO 1: p_i es un símbolo (nombre de función), o sea L_i tiene la forma (función ...)

Simplemente se evalúa L_i

CASO 2: p_i es una lista (predicado de prueba o test), esto es L_i es de la forma ((funcion ...) ...)

Si el valor de p_i es NIL, se ignoran los otros miembros de L_i.

Y si el valor de p_i es verdadero, se procede a evaluar secuencialmente los restantes miembros de L_i, y LOOP termina retornando el valor del último miembro.

Así, LOOP se ejecuta hasta encontrar una lista L_i cuya cabeza es una lista con valor verdadero.

Las listas del caso 2 se denominan listas de control y constituyen los lugares por los cuales LOOP puede salir del ciclo o terminar su ejecución.

La instruccion DO se usa así:

(DO
  LVAR
  L_1
  ...
  L_n
)

en donde LVAR es una lista compuesta por elementos de la forma

$$(var_i\ s_i\ a_i)$$

que definen las variables locales var_i, y L_i son listas $(p_i \ldots)$.

Efecto: Primero DO asigna simultáneamente a cada var_i el valor de s_i, y luego repite el ciclo:

procesar L_1

...

procesar L_n

actualizar var_i con el valor de a_i

DO procesa cada lista L_i de la misma manera en que lo hace LOOP.

Por tanto, cuando halla una forma L_i cuyo primer miembro p_i es una lista con valor verdadero, pasa a evaluar los otros miembros y finaliza retornando el último valor.

NOTA

1. La lista de variables locales puede ser vacía : ()

2. En lugar de (var_i s_i a_i) se admiten las expresiones:

   (var_i s_i) ; variable local con valor inicial
   ; pero sin expresión de actualización

   (var_i) ; variable local sin valor inicial ni
   ; expresión de actualización

3. LOOP es equivalente a DO sin variables locales, es decir, con la lista vacía de variables locales.

EJEMPLOS

1) La siguiente función usa un ciclo LOOP para imprimir los números 1, 2, ... , n, para un valor dado n

```
$ (DEFUN IMPRILOOP (N I)  ; parámetros son N, I
                          ; I se usa como
                          ; variable local
     (SETQ I 1)           ; asignar a I el valor 1
     (LOOP
          ((> I N))       ; si I es mayor que N,
                          ; terminar
          (PRINT I)       ; imprime valor de I
          (SETQ I (+ I 1)) ; incrementa valor

      )                   ; fin de loop

  )                       ; fin de función

  IMPRILOOP

$ (IMPRILOOP 4)
  1
  2
  3
```

```
4
T          ; valor de (>I N) cuando finaliza LOOP
```

2) Una función equivalente a la anterior pero que emplea la orden DO es:

```
$ (DEFUN IMPRIDO (N)
   (DO
      ( (I 1 (+ I 1)) )          ; lista de variables locales:
                                 ; sólo una variable local I, con
                                 ; valor inicial 1 y expresión de
                                 ; actualización para asignar a I
                                 ; su valor incrementado (+ I 1)
      ((> I N))                  ; lista de control, devuelve T
      (PRINT I)
   )
) ; fin de función

IMPRIDO

$ (IMPRIDO 4)
 1
 2
 3
 4
 T
```

3) Una función equivalente a (MEMBER E L 'equal) es:

```
(DEFUN  MIEMBRO  (E L)     ; devuelve resto de lista L
                           ; que empieza con E,
                           ; si E es un elemento de L

  (LOOP
    ((NULL L) NIL)         ; si L es vacía, devolver NIL
    ((EQUAL E (CAR L)) L)  ; si E es primer elemento de L,
                           ; retornar L
```

```
        (SETQ L (CDR L))          ; suprimir primer elemento de L
    ) ; fin de LOOP
) ; fin de función
```

Por ejemplo, el valor de (MIEMBRO 'A '(B A B A 4)) es la lista (A B A 4)

7.13 PROGRAMAS

En general un programa es un conjunto de instrucciones que resuelven un problema particular.

Los programas de LISP se componen de varias funciones encargadas de realizar determinadas tareas y que pueden llamarse unas a otras.

En todo programa se distingue una función especial, a la que se denomina función principal, cuya evaluación equivale a ejecutar el programa.

Usualmente los programas se salvan como archivos de texto que tienen la siguiente disposición:

(definición de función f_1)

...

(definición de función f_n)

Estos archivos se escriben y salvan con un editor de textos cualquiera o con uno proporcionado por el intérprete de LISP. Por ejemplo, algunas versiones de LISP tienen la orden EDIT que permite editar archivos de programas:

$ (EDIT na)

en donde na es el nombre de archivo a editar

Los archivos de programas pueden ser leídos con otra orden, que depende del intérprete usado, de modo que las funciones componentes son inmediatamente incorporadas al sistema para su uso.

7.14 APLICACIONES DE LISP

En esta sección se desarrollan algunos programas de aplicaciones de LISP.

7.14.1 PROGRAMA DE BUSQUEDA EN PROFUNDIDAD O ANCHURA

La siguiente función BUSCAR permite encontrar una ruta entre dos nodos de un espacio de búsqueda, aplicando uno de los métodos de búsqueda en profundidad o anchura (Ver Cap. 2).

```
(DEFUN BUSCAR (METODO INICIO DESTINO ESPACIO)
    ; METODO es 'P (profundidad) o 'A(anchura) para
    ; buscar una ruta de INICIO a DESTINO según  datos del
    ; ESPACIO =( (nodo1 listsuc1) (nodo2 listsuc2) ...)

    (DO
       ( (COLA (LIST (LIST INICIO)))  ; COLA=((INICIO)), es
                                      ; una lista
                                      ; de rutas con nodos en
                                      ; orden inverso
         (VISITADO  NIL)        ; lista de nodos visitados
         (RUTA)                 ; ruta actual
```

```
        (LSUC)                  ; lista de sucesores
        (FIN-RUTA)              ; nodo final de ruta actual
        (NUEVAS-RUTAS)          ; lista de nuevas rutas

      )                         ; lista de variables locales

      ((NULL COLA) (PRINC "No hay rutas ") (TERPRI))
      (SETQ RUTA (CAR COLA))  ; toma primera ruta de cola
      (SETQ FIN-RUTA (CAR RUTA))
                                ; primer elemento =nodo final
      ((EQUAL DESTINO FIN-RUTA) (REVERSE RUTA))
                ; si RUTA llega a DESTINO, terminar
      (SETQ VISITADO (CONS FIN-RUTA VISITADO))
                                        ; no, agregar
      (SETQ COLA (CDR COLA))    ; reducir cola
      (SETQ LSUC (CADR (ASSOC FIN-RUTA ESPACIO)))
                                ; buscar
                                ; sucesores de FIN-RUTA
      (SETQ NUEVAS-RUTAS (HALLAR-NRUTAS))
                                ; y obtener nuevas rutas

      (COND         ; agregarlas a cola según método
       ((EQUAL METODO 'P)
        (SETQ COLA (APPEND NUEVAS-RUTAS COLA))
       )
       ('T (SETQ COLA (APPEND COLA NUEVAS-RUTAS)))
      )

    ) ; fin de DO
)


(DEFUN HALLAR-NRUTAS ()
  ; devuelve ( ruta1 ... ruta2)  o NIL
  ; rutai = se construye agregando a RUTA un
  ; nodo de LSUC que no esté en VISITADO
```

```
(DO
  ((LRUTAS 'NIL) (NODO) (RUTA-AUX))        ; lista de varia-
                                           ; bles de DO

  ((NULL LSUC) LRUTAS)      ; si no quedan suce-
                            ; sores, retornar l lista
  (SETQ NODO (CAR LSUC)) ; tomar siguiente nodo
  (SETQ RUTA-AUX (CONS NODO RUTA)) ; y construir
                                   ; nueva ruta
  (COND
    ( (NOT (MEMBER NODO VISITADO 'EQUAL)) ; probar y
                                          ; agregar ruta
      (SETQ LRUTAS (APPEND LRUTAS (LIST RUTA-AUX)))
    )
  ) ; fin de COND

  (SETQ LSUC (CDR LSUC))   ; reducir lista de sucesores

) ; fin de DO

) ; fin de función
```

EJEMPLOS DE CORRIDAS

```
$ (SETQ LESP  '( (A (B C D)) (B (E D)) ) )
    ; asigna a variable LESP la lista asociativa que representa
    ; el espacio de búsqueda: los sucesores de A y B son
    ; dados por las listas (B C D) y (E D), respectivamente


$ (BUSCAR 'P  'A  'D LESP)  ; buscar en profundidad
                            ; ('P) una ruta
                            ; de 'A a 'D
  (A B D)                   ; ruta encontrada: A-B-D
```

```
$ (BUSCAR 'A  'A  'D LESP)    ; buscar en anchura (primer 'A)
   (A D)
```

7.14.2 PROGRAMA DE BUSQUEDA DE RUTAS DE LONGITUD MINIMA

La función BUSCARMIN encuentra una ruta de longitud mínima entre dos nodos de un espacio de búsqueda.

```
(DEFUN BUSCARMIN (INICIO DESTINO ESPACIO)
   ; busca una ruta mínima de INICIO a DESTINO según
   ; datos del  ESPACIO =( (n1 l1) (n2 l2) ...)
   ; en donde li es una lista de pares (s d), formado por sucesor s
   ; de ni y d=distancia de ni a s
   ; Ejemplo: ESPACIO puede ser
   ;    (
   ;      ( A  ((B 10) (C 5) ) )
   ;      ( B  ((D  3) (E 6) ) )
   ;    )
   ; El programa usa una lista COLA formada por longitudes y
   ; rutas (con elementos en orden invertido) :
   ;       COLA tiene la forma ( (d1 en ...e1) (d2 fm ...f1) ...)
   ;     en donde d1 es la longitud de la ruta (e1 ... en)
   ; Así, para el ejemplo anterior: algunos elementos
   ; de COLA son (10 B A), (13 D B A)

   (DO
      ( (COLA (LIST (LIST 0 INICIO))) ; COLA=((0 INICIO))
        (VISITADO  NIL) ; lista de nodos visitados
        (RUTA)          ; ruta actual
        (LONG)          ; longitud de ruta actual
        (LSUCD)         ; lista de pares de sucesores con
                        ; distancias
        (FIN-RUTA)      ; nodo final de ruta actual
        (NUEVAS-RUTAS) ; lista de nuevas rutas
```

```
        )                    ; lista de variables locales

        ((NULL COLA) (PRINC "No hay rutas ") (TERPRI))
        (SETQ RUTA (CAR COLA))    ; toma primera ruta de cola
        (SETQ FIN-RUTA (CADR RUTA))
                             ; segundo elemento =nodo final

        ((EQUAL DESTINO FIN-RUTA) (REVERSE RUTA))
                             ; si RUTA llega a DESTINO,
        (SETQ VISITADO (CONS FIN-RUTA VISITADO))
                                    ; no, agregar

        (SETQ COLA (CDR COLA))       ; reducir cola
        (SETQ LONG (CAR RUTA))       ; obtener longitud de ruta
        (SETQ LSUCD (CADR (ASSOC FIN-RUTA ESPACIO)))
            ; buscar sucesores de FIN-RUTA
        (SETQ RUTA (CDR RUTA))
           ; suprimir longitud= primer elemento

        (SETQ COLA (APPEND COLA (HALLAR-NRUTAS)) )

        (SETQ COLA (UBICAR-MINIMO)) ; pone mínima ruta
                                    ; al comienzo de COLA

    ) ; fin de DO
) ; fin de función

(DEFUN HALLAR-NRUTAS ()
    ; devuelve ( druta1  druta2)  o NIL
    ; drutai = se construye agregando a RUTA
    ; un nodo de LSUC que no este en
    ; VISITADO, y precedido por su longitud

    (DO
        ( (LRUTAS 'NIL) (NODO) (RUTA-AUX)
        (DISTANCIA) (LONGTEMP)
        )                    ; lista de variables de DO
```

```
        ((NULL LSUCD) LRUTAS) ; si no quedan sucesores,
                              ; retornar lista
        (SETQ NODO (CAAR LSUCD))    ; tomar siguiente nodo
        (SETQ DISTANCIA (CADAR LSUCD))      ; y distancia
        (SETQ LONGTEMP (+ LONG DISTANCIA))
        (SETQ RUTA-AUX (LIST LONGTEMP NODO))
        (SETQ RUTA-AUX (LIST (APPEND RUTA-AUX RUTA)))
                ; construir nueva ruta
        (COND
           ( (NOT (MEMBER NODO VISITADO 'EQUAL))
                ; probar y agregar ruta al final
             (SETQ LRUTAS (APPEND LRUTAS  RUTA-AUX ) )
           )
        ) ; fin de COND

        (SETQ LSUCD (CDR LSUCD))      ; reducir lista de
                              ; pares (dist suc)

    ) ; fin de DO

) ; fin de función


(DEFUN UBICAR-MINIMO ()  ; pone ruta mínima
                         ; como primer elemento de COLA

  (DO  ( (RUTAMIN (CAR COLA))          ; ruta mínima actual
         (LONGMIN (CAAR COLA))         ; longitud mínima actual
         (COLARES (LIST (CAR COLA))) ; COLARES contiene
                                       ; primera ruta
         (COLATEMP (CDR COLA) (CDR COLATEMP)) ; controla
                                       ; proceso
         (RUTATEMP)
         (LONGTEMP)
        ) ; lista de variables de DO
```

```
    ((NULL COLATEMP) COLARES)       ; si COLATEMP es
                                    ; vacía, retornar COLARES
    (SETQ RUTATEMP (CAR COLATEMP)) ; tomar primera
                                        ; ruta de COLARES
    (SETQ LONGTEMP (CAR RUTATEMP)) ; y su longitud
    (COND
      ( (< LONGTEMP LONGMIN)          ; si es menor que
                                      ; mínimo actual:
          (SETQ LONGMIN LONGTEMP)  ; actualizar long. min.
          (SETQ RUTAMIN RUTATEMP)  ; y ruta mínima
          (SETQ COLARES (CONS RUTAMIN COLARES))
              ; y poner  ruta mínima al comienzo de COLARES
      )

      ('T (SETQ COLARES (APPEND COLARES (LIST RUTATEMP))))
            ; en caso contrario, agregar RUTATEMP al final

    ) ; fin de COND

  ) ; fin de DO


) ; fin de función
```

EJEMPLO DE CORRIDA

```
$ (SETQ LESP
    '(  (A  ((B 10) (C 8) (D 5)) )
        (B  ((C 5)) )
        (C  ((E 7) (F 1)) )
```

```
        (D ((F 2)) )
        (E ((G 1)) )
        (F ((G 40)) )
    )
)

$ (BUSCARMIN 'A 'F LESP) ; buscar ruta mínima entre A y F
  (A D F 7)    ; ruta encontrada A-D-F con longitud 7
```

7.14.3 PROGRAMA SHELL DE SISTEMAS EXPERTOS

La función IDENT es un intérprete de tipo cáscara o "shell" de sistemas expertos, que permite identificar un objeto que satisface las reglas del sistema.

Esta función se llama con un argumento al que llamamos base de datos y es una lista de la forma:

$$(\text{lobj}, \text{Regla}_1, \text{Regla}_2, \ldots, \text{Regla}_n)$$

en donde lobj es una lista que contiene los objetos a identificar y cada Regla_i es una lista que se expresa por

(nr *y* $x_1 \ldots x_t$) ; *y* = tipo conjunción

(nr *o* $x_1 \ldots x_t$) ; *o* = tipo disyunción

o (nr *no* x_1) ; *no* = tipo negación

en las cuales:

nr es el nombre de regla

$x_1, \ldots, x_t$ son los componentes o miembros de la regla y pueden ser nombres de otras reglas o listas (tipo $y_1 \ldots y_u$)

El valor del tipo se utiliza para determinar si la $regla_i$ (o su nombre) se cumple o no. Así, si el tipo es *y*, la regla se cumple si todos sus miembros son verdaderos; si el tipo es *o*, la regla se cumple, si al menos uno de sus miembros es verdadero; y finalmente, si el tipo es *no*, la regla se cumple si su único miembro x_1 es falso.

```
(DEFUN IDENT  (BASEDATO)
  (LET

    ; *** variables locales en funciones llamadas dentro de LET
   ((OBJRES) (LOBJ) (LREGLAS) (LPOS) (LNEG) (LACTIVO) (OBJACT) )

    ; *** separar listas de objetivos y reglas
   (SETQ LOBJ (CAR BASEDATO))
   (SETQ LREGLAS (CDR BASEDATO))
   (TERPRI 2)

    ; *** función O-LIST identifica objetivo, resultado en OBJRES
   (SETQ OBJRES (O-LIST LOBJ))
   (TERPRI 2)

         ; *** probar solución encontrada
   (COND
    ((EQUAL OBJRES *ACTIVO*)
      (PRINC " Error , objeto mal definido : ") (PRINC  OBJACT)
    )
    ((NULL OBJRES) (PRINC " No se puede identificar! ") 'T)
```

```
      ('T  (PRINC "   Objeto puede ser un (una) ") (PRINC OBJRES) )
    ); fin COND
  ) ; fin LET
  (TERPRI 2) (PRINC "  Fin de programa ") (TERPRI)
)


(DEFUN O-LIST (L)  ; L = (x1 x2 ... xn)
       ; *** lazo O : detecta primer xi con prueba no nil
   (LOOP
      ((NULL L)  'NIL)
      ((PRUEBA (CAR L)))
      (SETQ L (CDR L))
   )
)


(DEFUN Y-LIST (L)
    ; *** lazo Y : detecta si todos los xi son no nil

    (DO  ((RES))
         ((NULL L)  'T)
         (SETQ RES (PRUEBA (CAR L)))
         ((EQUAL  RES *ACTIVO*) *ACTIVO*)
         ((NULL RES) NIL)
         (SETQ L (CDR L))

    )
)

(DEFUN PRUEBA (X)
  ; *** función de prueba
  (COND
    ((ES-POSITIVO X)  X)
```

```
    ((ES-NEGATIVO X)  NIL)
    ((ESTA-ACTIVO X))
    ('T
      (LET ((RES) (L))
        (SETQ LACTIVO (CONS X LACTIVO) )          ; salva X en
                                                  ; lista LACTIVO
        (SETQ L (LATRIBUTOS X))
        (COND
          ((LISTP X) (SETQ RES (PRUEBA-LIST X))) ; X es lista
          ((NULL L) (SETQ RES (PREGUNTAR X)))    ; X es átomo
                                                  ; sin atributos
                                                  ; solicitar información
          ('T  (SETQ RES (PRUEBA-LIST L))) ; X es átomo
                                                  ; con atributos
        )                                         ; analizar lista de atrib.


        (SETQ LACTIVO (CDR LACTIVO))  ; suprime X en LACTIVO
        (COND              ; salva información en listas LPOS y LNEG
          ((NULL RES) (SETQ LNEG (CONS X LNEG)) NIL)
          ('T       (SETQ LPOS (CONS X LPOS)) X)
        )
      ) ; fin de LET
    ) ; fin de 'T

  ) ; fin de primer COND
)

(DEFUN PRUEBA-LIST (L)
       ; *** analiza lista según el tipo de relación *O*, *Y*, *NO*
  (LET ((X))
       (SETQ X (CAR L))  (SETQ L (CDR L))
```

```
        (COND
            ((EQUAL X '*O*) (O-LIST L))
            ((EQUAL X '*Y*) (Y-LIST L))
            ((AND (EQUAL X '*NO*) ( NULL (CDR L)))
              (NOT (PRUEBA (CAR L))))
        )
  )
)


(DEFUN PREGUNTAR (X)
        ; *** pregunta si propiedad X es verdadera o falsa

  (PRINC " * ") (PRINC X ) (PRINC " (S/N) ? > " )

    (DO  ((RES))
      (SETQ RES (READ))
      ( (MEMBER RES '(S N) 'EQUAL)
        (COND
          ((EQUAL RES 'S) X)
          ('T NIL)
        ) ; fin de COND
      )
    ) ; fin de DO
)

        ; *** retorna lista de atributos asociados a X

(DEFUN LATRIBUTOS (X) (CADR (ASSOC X LREGLAS 'EQUAL) ) )

        ; ***  determina si X esta en lista LPOS

(DEFUN ES-POSITIVO (X) (MEMBER X LPOS 'EQUAL) )
```

```
(DEFUN ES-NEGATIVO (X) (MEMBER X LNEG 'EQUAL) )

        ; *** determina si X esta en pila de pruebas LACTIVO
(DEFUN ESTA-ACTIVO (X)
  (COND
    ((MEMBER X LACTIVO 'EQUAL)  (SETQ OBJACT X) *ACTIVO*)
  )
)
```

EJEMPLO DE CORRIDA

```
$ (SETQ BD          ; definir base de datos en variable BD
   '(
    ; **** lista de objetivos

   (TIGRE CHITA JIRAFA CEBRA)

    ; **** reglas

   (MAMIFERO (*o* "TIENE PELOS" "DA LECHE" ) )
   (AVE   (*o*  "TIENE PLUMAS" (*y* "PONE HUEVOS" "ES VOLADOR") ) )
   (CARNIVORO (*o* "COME CARNE"  (*y* "TIENE DIENTES AFILADOS"
                                          "TIENE GARRAS"
                                    "TIENE OJOS FRONTALES" )))

   (UNGULADO  (*y* MAMIFERO (*o* "TIENE PEZUNA" "ES RUMIANTE")) )

   (CHITA  (*y* (*y* MAMIFERO CARNIVORO "COLOR LEONADO" )
                "TIENE MACHAS NEGRAS"))
```

```
(TIGRE (*y*  (*y* MAMIFERO CARNIVORO "COLOR LEONADO")
             "TIENE RAYAS NEGRAS"))
(JIRAFA (*y* UNGULADO "TIENE CUELLO LARGO" (*no* CARNIVORO)))
      (CEBRA  (*y* UNGULADO "TIENE RAYAS NEGRAS"))

) ; fin de lista '(

) ; fin de (SETQ


$ (IDENT BD)    ; identificar segun BD

 * TIENE PELOS (S/N) ? > N
 * DA LECHE (S/N) ? > S
 * COME CARNE (S/N) > N
 * TIENE DIENTES AFILADOS (S/N) ? > N
 * TIENE PEZUNA (S/N) ? > S
 * TIENE CUELLO LARGO (S/N) ? > S

 Objeto puede ser un (una) JIRAFA

 Fin de programa
 NIL
```

7.15.2 EVALUACION DE MACROS

Cuando se evalúa (nm $arg_1 \ldots arg_n$), siendo arg_i los argumentos que se pasan a la macro nm, tiene lugar el siguiente proceso:

1) se asigna a P_i el símbolo arg_i (sin evaluarlo)
2) se ejecuta o evalúa sucesivamente cada forma F_i
3) se halla la forma intermedia de la macro, que es el valor de la última forma evaluada F_i

y 4) se evalúa la forma intermedia y se retorna su valor

Los parámetros P_i son variables locales de la macro, con valores iniciales arg_i.

7.15.3 CONSTRUCCION DE LA FORMA INTERMEDIA.

Usualmente, cada llamada a una macro es considerada como un proceso de sustitución de la expresión de la macro por su forma intermedia.

De un modo preciso, cada vez que se ejecuta o llama

(nm $arg_1 \ldots arg_n$); llamada a macro nm

se reemplaza esta expresión por la forma intermedia I y luego se evalúa I.

En estos casos, la forma intermedia, o valor de la última forma de la macro, siempre es una lista cuyos elementos son las operaciones o procedimientos que se aplicarán a los argumentos:

7.15 MACROS

Una macro nm es un nombre de procedimiento que se ejecuta o llama de una manera similar a una función ordinaria:

$$(nm\ arg_1 \ldots arg_n)$$

pero sin evaluar los argumentos arg_i.

Puesto que los arg_i se reciben en el cuerpo de la macro tal cual aparecen en la llamada, es posible realizar operaciones sobre ellos mismos que no sólo permiten modificar sus valores, sino también agregar nuevas instrucciones al lenguaje LISP.

7.15.1 DEFINICION DE MACROS

Para definir una macro de nombre nm se ingresa una lista cuyo primer elemento es la palabra DEFMACRO:

(DEFMACRO nm (p_1 p_2 ... p_m))
    F_1
    F_2
    ...
    F_n
)

en donde (p_1 p_2 ... p_m) es una lista de símbolos $p_1, p_2, \ldots, p_m$, llamados parámetros de nm, y $F_1, F_2, \ldots, F_n$, son las formas componentes de nm, y constituyen el bloque o cuerpo de instrucciones de la macro.

```
(nf v ... ) ; nf es un procedimiento
```

y sirve como patrón o molde para construir el valor de la última forma de la macro:

```
(LIST 'nf x ... )
```

en donde cada procedimiento nf sólo aparece indicado y x es una expresión cuyo valor es v, ...

7.15.4 MACROS CON UN NUMERO ARBITRARIO DE ARGUMENTOS

También puede definirse una macro con un número variable de argumentos:

```
(DEFMACRO nm LARG
    F1
    ...
    Fn
)
```

en donde el valor del parámetro LARG es la lista de los argumentos arg_i, sin evaluarlos.

7.15.5 EJEMPLOS

1) La siguiente macro MUESTRA imprime los valores de los parámetros:

```
$ (DEFMACRO MUESTRA (X Y)
  (PRINT X)     ; imprime valor de X
  (PRINT Y)     ; imprime valor de Y
                  ; forma intermedia es el valor de Y
 )

$ (SETQ A 3)

$ (MUESTRA (+ 2 3) A)
  (+ 2 3)
  A             ; forma intermedia es A
  3             ; valor de forma intermedia
```

2) La macro (MSETQ VAR EXPR) asigna a una variable VAR el valor de EXPR.

La forma intermedia que la macro debe producir es
(SETQ VAR EXPR)

y por lo tanto el valor de la (única) forma de la macro ha de ser:
(LIST 'SETQ VAR EXPR)

Ahora se ingresa la macro:

```
$ (DEFMACRO MSETQ (VAR EXPR)
   (LIST 'SETQ VAR EXPR)
  )
```

y se exhibe una evaluación:

```
$ (MSETQ A (* 3 6))
  18

$ A         ; valor de A
  18
```

Explicación: VAR recibe el valor A y EXPR el valor (* 3 6). La forma (LIST `SETQ VAR EXPR) tiene el valor (SETQ A 18), pues LIST evalúa VAR y EXPR. Finalmente se evalúa la forma intermedia (SETQ A 18) y por lo tanto A recibe el valor 18.

3) A continuación presentamos una macro que equivale a una de las dos instrucciones IF de lenguajes como Pascal:

IF condición F_1 F_2 ... F_m

IF condición F_1 F_2 ... F_m *ELSE* G_1 G_2 ... G_n

para ejecutar el bloque F_1 F_2 ... F_m si la condición se cumple, y en caso contrario, siempre que se utilice *ELSE*, ejecutar el segundo bloque de formas.

En el primer caso, la forma intermedia es:

(COND (condición F_1 F_2 ... F_m))

y en el segundo caso:

(COND
(condición F_1 F_2 ... F_m)
('T G_1 G_2 ... G_n)
)

```
$ (DEFMACRO *IF* LARG
  (DO ( (L (CDR LARG) (CDR L))
      (PC (LIST (CAR LARG)))
      (PT)
      (ELEM)
     )  ; variable locales a DO

   ((NULL L ) (LIST 'COND (APPEND PC PT))) ; caso *IF*
                                           ; sin *ELSE*

   (SETQ ELEM (CAR L))
   ((EQUAL *ELSE* ELEM)              ; caso *IF* ... *ELSE*
     (LIST 'COND (APPEND PC PT) (LIST 'T (CDR L))))

   (SETQ PT (APPEND PT (LIST ELEM)))
  ) ; fin de DO
)
```

Explicación:

Se utiliza un ciclo DO para separar los componentes de la lista LARG; PC es la lista formada por la condición, PT contiene las formas del primer bloque y L es la lista de trabajo que se inicia con los elementos de LARG, y en cada iteración se reduce en un elemento.

Si L termina vacía la macro trata el caso a *IF* sin *ELSE* devolviendo la lista

(LIST 'COND (APPEND PC PT))

cuyo valor da la forma intermedia del primer caso.

Y de manera similar si se encuentra *ELSE*.

APLICACION

```
$ (DO ((I 0))
    ((= I 10))
    (*IF* (< I 5) (PRINT I)) ; ejecuta macro *IF*
    (SETQ I (+ I 1))
  )

1
2
3
4
T
```

La forma intermedia producida por la macro *IF* es la lista:

```
(COND
  ( (< I 5) (PRINT I))
)
```

7.15.6 CARACTER DE REFERENCIA CON VALOR

Cuando se utiliza el signo de apóstrofo (') delante de una expresión, se retorna dicha expresión sin evaluarla (Ver 7.5.2). Por ejemplo,

```
'(A B (* 3 2))
```

devuelve (A B (* 3 2))

En muchos casos se requiere no sólo reproducir la expresión, sino que adicionalmente algunos de sus elementos sean evaluados y reemplazados por sus valores. Esto se consigue

usando el signo de apóstrofo invertido o grave (`) ,en lugar de ('), y anteponiendo comas (,) a las partes de la expresión que se deseen evaluar. Por ejemplo, si se precisa que en el sitio de (* 3 2) se ponga su valor:

```
`(A B ,(* 3 2))
```

que al evaluarse retorna la lista (A B 6)

El operador grave es útil para imprimir textos o mensajes como listas con los valores de algunas variables:

```
(PRINT `(texto ,var) )
```

que imprime la lista (texto v)

Aquí el texto esta formado por uno o más átomos que son citados literalmente y v es el valor de var.

EJEMPLOS

```
$ (SETQ A 3 B '(* 4 2))

$ `(A B)
  (A B)

$ `(,A ,B) ; retorna lista con los valores de A y B
  (3 8)

$ (PRINT `(El resultado es ,A))
(El resultado es 3)
```

Otra aplicación importante del signo grave se refiere a la simplificación de la escritura de la expresión que produce la

forma intermedia en el bloque de una macro. Así, para obtener la forma intermedia (Ver 7.15.3) :

(nf v ...)

basta escribir `(nf , x ...)

siendo v el valor de la expresión x.

Por ejemplo, la macro MSETQ del ejemplo 2 de 7.15.5 puede ser dada por:

```
$ (DEFMACRO MSETQ (VAR EXPR)
   `(SETQ ,VAR ,EXPR)
  )
```

de manera que `(SETQ, VAR, EXPR) devuelve la lista compuesta por SETQ y los valores de VAR y EXPR.

7.15.7 MAS EJEMPLOS DE MACROS

Concluimos el presente capítulo presentando varias macros que pueden ser incorporadas y empleadas en el lenguaje LISP como nuevas instrucciones y en las que se observa el mecanismo de construcción de la forma intermedia "por partes", es decir, a partir de los argumentos.

1) La macro (*INC* V) incrementa en 1 el valor de una variable V (con valor entero).

   y se ingresa mediante:

   ```
   $ (DEFMACRO *INC* (V) `(SETQ ,V (+ ,V 1)))
   ```

   Una aplicación es:

```
$ (SETQ I 4)
$ (*INC* I)
$ I          ; nuevo valor de I
  5
```

2) La macro (LPUSH X_1 X_2 ... X_n V) en donde se asume que V es una variable cuyo valor es una lista (a ... z), tiene por efecto agregar los valores x_i de X_i al comienzo de dicha lista, esto es, V tendrá el valor (x_1 ... x_n a ... z).

Se muestra una definición de LPUSH y también una aplicación:

```
$ (DEFMACRO LPUSH LA
  (LET  ((V) (LV))
       (SETQ V (CAR (LAST LA)))          ; V recibe último
                                         ; elemento de LA
    (SETQ LV (MAPCAR 'EVAL (BUTLAST LA)))
          ; ****** LV=lista de valores,
          ; ******* excluyendo último elemento

    `(SETQ ,V (APPEND (QUOTE ,LV) ,V))

  )
  )


$ (SETQ LL '(1 2 3))

$ (LPUSH 'A 'B (* 3 4) LL)

$ LL
  (A B 12 1 2 3)
```

3) La siguiente macro tiene por propósito construir un ciclo de tipo FOR como en el lenguaje Pascal: repite la ejecución del bloque de formas F_1, ... , F_n, de manera que la variable VAR asume sucesivamente cada valor entero entre dos límites dados por las expresiones v_i y v_f:

(*FOR* VAR v_i v_f F_1, ... , F_n)

Se define por:

```
$ (DEFMACRO *FOR* LA
    (LET  ((RES) (VAR) (VI) (VF) (BLOQUE))
     (SETQ VAR (CAR LA))
     (SETQ VI (CADR LA))
     (SETQ VF (CADDR LA))
     (SETQ BLOQUE (CDDDR LA))
     (SETQ RES `(DO ((,VAR ,VI (+ ,VAR 1)))
           ((> ,VAR ,VF))
       )
     )

     (SETQ RES (APPEND RES BLOQUE))
     RES    ; expresión que produce forma intermedia
    )
   )
```

Una aplicación de *FOR* es:

```
$ (*FOR* X 10 (* 3 5) (PRINC X) (PRINC " ") (PRINT (* X X))
  10  100
  11  121
  12  144
  13  169
  14  196
```

```
15 225
T
```

4) La macro

(*DO* F_1 , ... , F_n *WHILE* condición)

crea un ciclo equivalente al de la instrucción do del lenguaje C: Ejecuta el bloque de formas mientras la condición se cumpla.

```
$ (DEFMACRO *DO* LA
      (LET  ((XCOND) (BLOQUE))
           (SETQ XCOND  `((NOT ,(CAR (LAST LA)))))
           (SETQ LA (BUTLAST LA))
           (SETQ BLOQUE (BUTLAST LA))
    (COND
     ((EQUAL '(*WHILE*) (LAST LA))
          (APPEND '(LOOP) BLOQUE  (LIST XCOND)))
     ('T NIL)
    )
  )
)

  $ (SETQ I 1)

  $ (*DO*  (PRINT I) (SETQ I (+ I 1)) *WHILE* (<= I 5))
   1
   2
   3
   4
   5
   T
```

Referencias Bibliográficas

Winston Patrick Henry,
ARTIFICIAL INTELLIGENCE
Ed. Addison-Wesley, 1984

Charniak E. y McDermott D.,
INTRODUCTION TO ARTIFICIAL INTELLIGENCE,
Ed. Addison-Wesley, 1985

Nilsson Nils J.
PRINCIPLES OF ARTIFICIAL INTELLIGENCE
Ed. Springer-Verlag, 1986

Giarratano J. y Ryley G.,
EXPERT SYSTEMS, *Principles and Programming,*
Ed. PWS-KENT, 1989

Winston P. H. y Horn., LISP,
Ed. Addison-Wesley, 1984

Indice Alfabético

Esta obra se terminó de imprimir el mes de setiembre de 1993, en

PERU OFFSET E. I. R. L.

Prolongación Lucanas 278 - La Victoria
Lima - Perú. Telf: 318924
La edición consta de 1000 ejemplares.

PUBLICACIONES RECIENTES

ADOLFO FIGUEROA

Crisis Distributiva en el Perú. 1993. 204 p.

NORMA FULLER

Dilemas de la Femineidad. 1993. 334 p.

GUILLERMO LOHMANN VILLENA

Amarilis Indiana. 1993. 398 p.

ALEJANDRO ORTIZ R.

La Pareja y el Mito en los Andes. 1993. 264 p.

CARLOS AUGUSTO RAMOS

Toribio Pacheco. Jurista Peruano del Siglo XIX. 1993. 312 p.

LILIANA REGALADO DE HURTADO

Sucesión Incaica. 1993. 126 p.

GUILLERMO ROCHABRUN

Socialidad e Individualidad. 1993. 194 p.

JAVIER SOLOGUREN

El Rumor del Origen. 1993. 392 p.

DENIS SULMONT - MARCEL VALCARCEL

Vetas de Futuro. 1993. 288 p.

MAXIMO VEGA-CENTENO

Desarrollo Económico y Desarrollo Tecnológico. 1993. 234 p.

ADOLFO C. WINTERNITZ

Itinerario hacia el Arte. 1993. 118 p.

CELIA WU BRADING

Generales y Diplomáticos. 1993. 282 p.