Lej. 5

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

LISP-PERITO:

APLICACIONES DE UN EXPERTO ARTIFICIAL, CONSTRUIDO CON EL LENGUAJE FUNCIONAL, LISP

T E S I S

Que para obtener el título de

ACTUARIO

PRESENTA:

EDITH ARIZA GOMEZ

México, D.F.

1983

I N D I C E

1. INTRODUCCION

El hombre en el transcurso de la historia, ha utilizado parte de su vida en crear instrumentos que le ayuden a realizar sus actividades en un menor tiempo y con un mínimo esfuerzo.

Un ejemplo de ello es la construcción de máquinas -- computadoras las cuales se han desarrollado en un lapso relativamente corto, ya que la construcción de éstas fue iniciada por Alan Mathison Turin en el año de 1937.

En el año de 1957 surge el término de Inteligencia - Artificial asociado al propósito de lograr mediante artefactos, que se realicen tareas que de ser hechas por seres humanos, necesitan de un cierto grado de razonamiento (1).

Sin embargo, esta inquietud del hombre se remonta -- quizá a los años en que nuestros ancestros utilizan piedras para contar. Todo este trabajo previo puede ser considerado como la base de lo que se conoce como Inteligencia Artificial aunque hoy en día se le asocie con la idea moderna - de computadoras.

Las aplicaciones de la Inteligencia Artificial hasta la fecha han sido entre otras:

La construcción de robots
El desarrollo de juegos
La implementación de procesadores del lenguaje natural
El análisis de imágenes
Así como la construcción de Expertos Artificiales.

Con el gran auge que en el área de cómputo se ha logrado, Feigenbaun (2) señala que nos encontramos a un paso de la construcción de lo que el llama "las computadoras de la quinta generación", para lo cual se requiere de algunas innovaciones técnicas, científicas y sociales, ya que para su construcción no es necesario desarrollar el equipo existente hasta la fecha (Hardware), sino que se necesita desarrollar en particular el llamado Software de la Ingeniería del Conocimiento, el cual tiene como principal objetivo lograr programar funciones inteligentes.

Se señala también que dos de las características que deben tener estas computadoras son: que sean sistemas que permitan la manipulación simbólica y que sean procesadores o por lo menos operadores del conocimiento. Se espera además que la demanda comercial en los años 1990 y 2000, de este tipo de computadoras, esté basado en la construcción de Sistemas Expertos. A continuación se explicará brevemente-

lo que es un Sistema Experto.

Un Sistema Experto está formado por un conjunto de - programas que logran un alto nivel de eficiencia, relativo- al trabajo que realiza un especialista de una determinada - área del conocimiento.

Al realizar la construcción de un Sistema Experto no se hace con la idea de buscar información en una base de datos, lo cual se logra con un programa de computación común, sino que se trata de realizar una búsqueda inteligente so-- bre una base de conocimientos, la cual está formada por una colección de hechos, suposiciones, creencias y reglas heu-- rísticas.

Un Sistema Experto debe estar formado fundamentalmente por:

Un Conocimiento específico de un área del conocimiento
Un sistema de consulta
Un sistema de adquisición del conocimiento
Un subsistema de inferencia en donde se determine el orden- y el tipo de estrategia a seguir
Y por una interface amigable que ayude al usuario para que- el experto pueda ser consultado.

Dentro de las estrategias se cuenta con dos, las cuales establecen relaciones entre los hechos y las hipótesis-

o diagnósticos, una de ellas denominada "hacia adelante" -- (forward) que trata de encontrar aquella hipótesis que se- pueda adecuar a un conjunto de hechos los cuales inicialmen te son preguntados aleatoriamente y la otra denominada - - "hacia atrás" (backward) la cual trata de validar una hipó- tesis buscando confirmar aquellos hechos que la apoyen.

Las áreas de aplicación de los Expertos han sido:

El diagnóstico y la terapéutica médica
La configuración de sistemas de cómputo
La enseñanza y el aprendizaje
La comprobación de teoremas
La implementación del lenguaje natural
El procesamiento de imágenes
Así como la exploración de minerales

En la siguiente tabla se muestran algunos ejemplos - de Sistemas Expertos donde se señala el nombre y su función.

N O M B R E	F U N C I O N
AQ11 (3)	Diagnóstico de enfermedades en - plantas
CASNET (4)	Diagnóstico Médico
DENDRAL (5)	Explora las estructuras molecula res de las masas
DIPMETER ADVISOR (6)	Exploración del petróleo
EL (7)	Análisis de circuitos eléctricos
INTERNIST (8)	Diagnóstico Médico
KMS (9)	Diagnóstico Médico

N O M B R E	F U N C I O N
MACSYMA (10)	Manipulación de fórmulas matemáticas
MDV (11)	Diagnóstico Médico
MOLGEN (12)	Planear experimentos de DNA
MYCIN (13)	Diagnóstico Médico
PROSPECTOR (14)	Exploración de Minerales
PUFF (15)	Diagnóstico Médico
R1 (16)	Configuración de computadoras

Se espera que en un futuro próximo, los Sistemas Expertos sean ampliamente usados para:

el diseño de circuitos,
automatización de oficinas,
inversiones y finanzas,
el control de procesos,
la interpretación de señales,
en usos militares y
en la producción y mantenimiento del Software

Para la construcción de un Sistema Experto se necesita del uso de un lenguaje de programación. El más comunmente usado en la llamada Inteligencia Artificial ha sido el lenguaje LISP (17), el cual a diferencia de otros lenguajes de programación, cuenta con una amplia capacidad para el manejo de expresiones simbólicas y para la definición de funciones recursivas.

Otra de las capacidades de LISP es que se pueden generar nuevas funciones construidas a partir de las funciones

ya existentes (función de función), ésto permite la elaboración de programas o funciones de un alto grado de complejidad, las cuales pueden ser usadas de inmediato por el usuario debido al carácter interactivo de LISP.

Con motivo de familiarizarse un poco con los Sistemas Expertos a continuación se resume el funcionamiento de uno de ellos que puede ser considerado como clásico y es MYCIN, el cual está enfocado al Diagnóstico Médico.

MYCIN es un Sistema Experto que está programado en un dialecto de LISP denominado Interlisp (18). Este Experto usa criterios de decisión basados en conocimientos de diferentes Expertos humanos, para aconsejar a los fisiólogos sobre la selección de una terapéutica antimicrobiana.

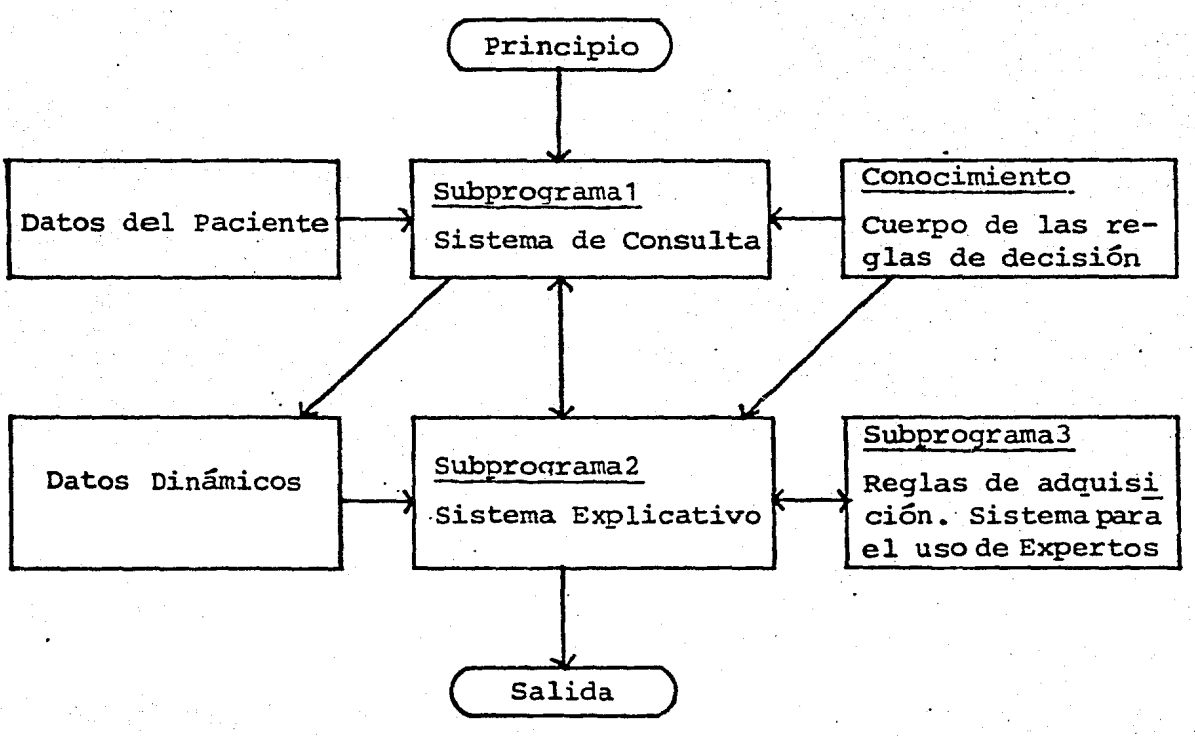
Este Experto usa la estrategia denominada "hacia atrás" o razonamiento regresivo y la forma en la que se encuentra representado el conocimiento, es en forma de reglas de producción las cuales están formadas por premisas y conclusiones o cursos de acción a seguir.

Este Sistema Experto contiene además tres subprogramas, el primero de ellos es un sistema de consulta y usa los datos del paciente y la base de conocimientos para generar una terapéutica adecuada.

El subprograma 2 o Sistema Explicativo, es un sistema que puede usarlo el usuario para pedir información adicional sobre cómo y porqué se sugieren ciertas terapéuticas médicas

El subprograma 3 está formado por un sistema de adquisición de conocimientos al cual sólo tienen acceso un grupo de expertos para modificar la base de conocimientos del Sistema Experto.

La dinámica de este Sistema Experto se representa a continuación en un diagrama.



MYCIN es un Sistema que para su consulta requiere de una sesión que dura de 15 a 20 minutos, en donde aproximadamente se hacen 45 preguntas con las cuales se generan posibles diagnósticos.

MYCIN empieza por hacer preguntas iniciales sobre el nombre, sexo y edad del paciente y en seguida adopta la estrategia de razonamiento regresivo para seleccionar otras -- preguntas hasta llegar a emitir un diagnóstico final.

Una vez que el Sistema identifica al posible o posibles organismos, se recomienda una terapéutica adecuada a - cada uno de ellos.

Si el usuario demanda una explicación de porqué o cómo se llegó a un determinado diagnóstico, MYCIN le puede -- proporcionar este tipo de información.

Por último mencionaremos que MYCIN cuenta con un factor de creencias y descreencias asociado al diagnóstico final emitido por éste.

Como resultado de recapacitar sobre una de las necesidades futuras dentro de la Ingeniería del Conocimiento, - que es el desarrollo de los Sistemas Expertos, se plantea -

como objetivo de esta investigación, el ilustrar como a partir de mínimas modificaciones hechas en un Sistema Experto-básico, se pueden abarcar diferentes áreas del conocimiento.

El Sistema Experto usado en este trabajo fue desarrollado por Winston (19) y cumple con los requisitos mínimos-mencionados anteriormente para funcionar como un buen experto. La estrategia con la que trabaja este Sistema es la denominada "hacia atrás", la cual parte de la hipótesis a los hechos.

En este trabajo se ofrecen dos de las posibles aplicaciones de este Sistema Experto, una de ellas muestra a un Experto "ortografista" y la otra a un Experto "estadístico".

El lenguaje usado para la construcción del Sistema - Experto es una versión de LISP denominada APP-L-LISP (20) que se encuentra implementada en una computadora Apple II Plus. En el apéndice se encuentra la información relacionada a la sintáxis y uso de las principales funciones usadas en el -- lenguaje LISP.

2. REPRESENTACION DEL CONOCIMIENTO

2.1 INTRODUCCION

La representación del conocimiento surge como una necesidad del hombre desde el origen de él mismo, con el uso de los primeros símbolos, señales y sonidos que empleó para comunicarse con los demás. Estas representaciones se fueron afinando hasta llegar a la época de aquel hombre de las cavernas, que pintaba sus cuevas con imágenes y símbolos. Entre otras cosas para representar algunos aspectos importantes de su vida.

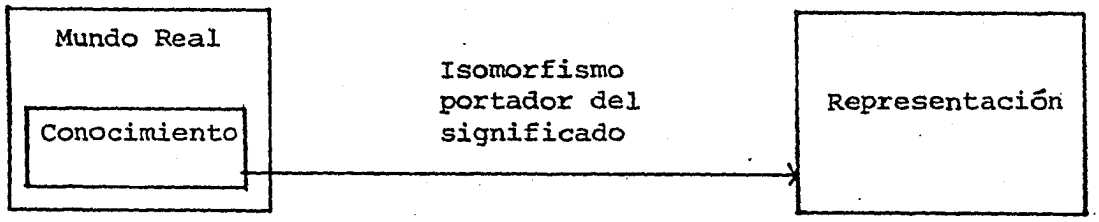
Hoy en día se cuenta con una gran variedad de representaciones y algunas de ellas son: pinturas, letras, números, listas, reglas, símbolos, diagramas, gráficas, etc.

Aunque existen representaciones que pueden ser usadas en diversas áreas del conocimiento como por ejemplo: las ecuaciones tan usadas en matemáticas, física, química, biología, etc.; existen otras representaciones que sólo son usadas por áreas del conocimiento específicas; ejemplos de ello son los símbolos para escribir los idiomas, como el Japonés, Chino, Tahi entre otros, los cuales sólo para ello son usados.

Las diversas representaciones han servido esencial-- mente entre otras cosas, para transmitir, comprender y aún- manipular el conocimiento.

Entre la representación y el conccimiento, existe un lazo de unión, al cual se le considera un isomorfismo forma do por un conjunto de reglas de correspondencia (21).

Este isomorfismo es el portador del significado del- conocimiento, el cual queda plasmado en la propia represen- tación. Para entender esta idea se muestra el siguiente es quema:

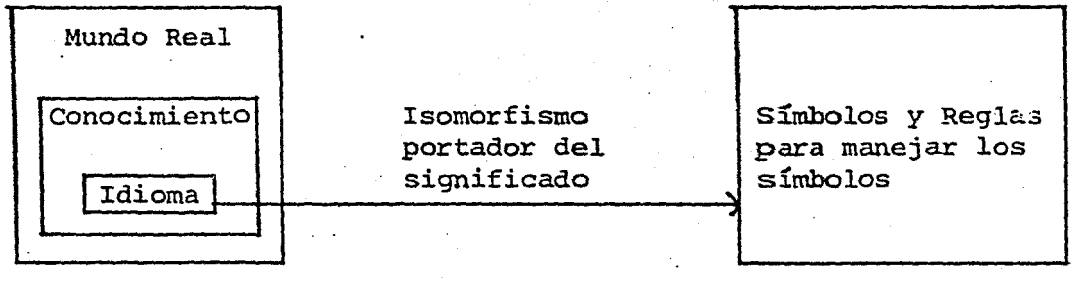


Las representaciones del conocimiento casi siempre - van acompañadas de una serie de reglas, las cuales rigen su correcto uso e interpretación.

Por ejemplo, para escribir un idioma se cuenta con - un conjunto de símbolos o letras, las cuales se manejan a - través de una serie de reglas gramaticales. En este ejem--

plo el conocimiento es el idioma, la representación de éste se logra mediante símbolos que tienen reglas gramaticales y el vínculo de unión entre el conocimiento y su representa-- ción es el isomorfismo portador del significado que se le - da a los propios símbolos.

Si representamos el ejemplo anterior en un diagrama, éste queda de la siguiente forma:



Por último hay que señalar que el conocimiento que - se tenga de cierta área del conocimiento puede ser represen tado de diversas formas, pero hay que encontrar la represen tación adecuada que logre transmitirlo o que permita que -- pueda ser manipulado de una forma eficiente.

2.2 REPRESENTACION DEL CONOCIMIENTO QUE USA EL EXPERTO ARTIFICIAL

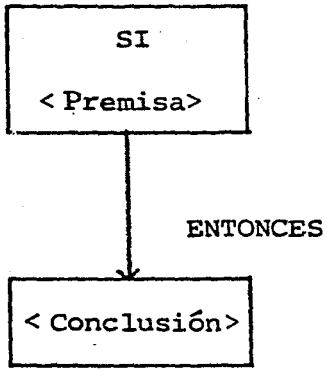
En la construcción de Expertos se han estudiado prin cipalmente dos formas de representar el conocimiento, una -

de ellas se refiere a listas de asociación usadas en los -- marcos de MINSKY (22,23,24), y la otra está basada en reglas de producción (19); ésta última es la que será nuestro objeto de estudio.

En las llamadas reglas de producción o reglas SI-ENTONCES, el tipo de conocimiento que representan, es aquél - que puede ser expresado en una colección de reglas con la - siguiente estructura:

SI < premisa> ENTONCES < conclusión >

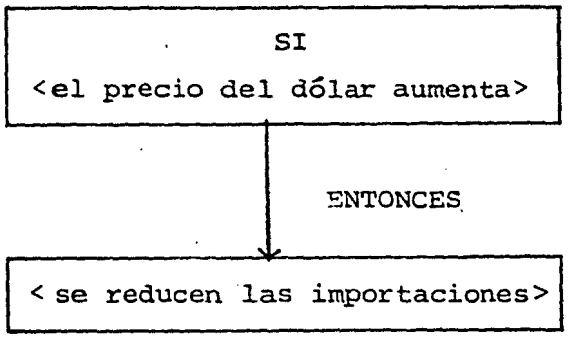
A manera de comparación, se representa la regla anterior en una forma más comunmente conocida que es la repre-- sentación en forma de "árbol" y es la siguiente:



Para familiarizarse con la representación en forma - de reglas de producción se muestra el siguiente ejemplo:

SI <el precio del dólar aumenta>
ENTONCES <se reducen las importaciones>

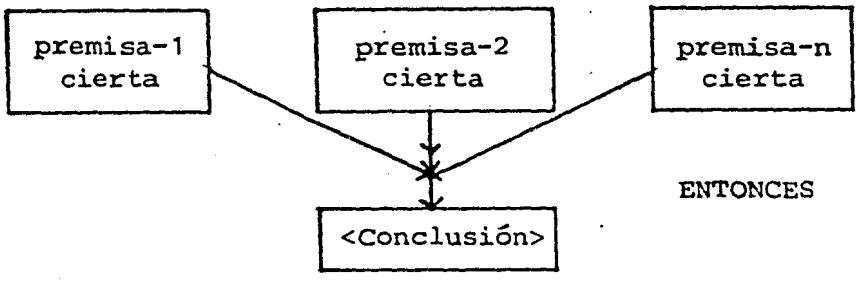
lo cual representado en forma de "arbol" quedaría de la si-- guiente forma:



La estructura anterior de las reglas de producción - puede complicarse un poco aumentando el número de premisas. De esta forma se obtendrá una conclusión, solo si se cum- - plen todas las premisas de la regla, quedando el esquema anterior de la siguiente manera:

SI [< premisa-1 es cierta >
< premisa-2 es cierta >
.
.
.
< premisa-n es cierta >] ENTONCES < conclusión >

lo que en forma de árbol quedaría así:



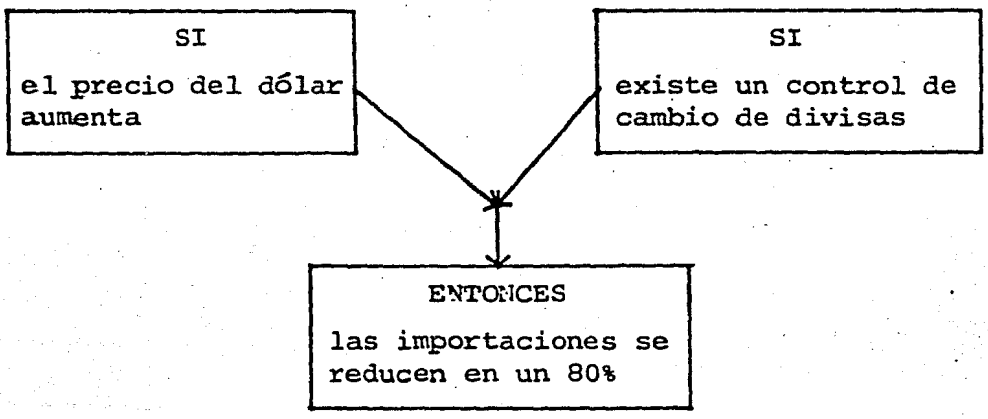
Continuando con el ejemplo dado anteriormente, le aumentaremos otra premisa más, quedando el problema expresado en la siguiente forma:

Las importaciones se reducen en un 80%; si el precio del dólar aumenta y si además existe un control de cambio de divisas.

Esto representado en forma de reglas de producción - queda de la siguiente forma:

SI [< el precio del dólar aumenta> y
   [< existe un control de cambios de divisas>]
ENTONCES < las importaciones se reducen en un 80%>

que representado en forma de "árbol" queda así:



Como por lo general el conocimiento no se representa por una sóla regla, se necesita de una colección de reglas, con las cuales se represente el conocimiento, cuyas conclu-

siones puedan pasar a formar parte de las premisas de otra- regla, quedando entonces éstas como conclusiones parciales, o como conclusiones finales.

El esquema es el siguiente:

regla 1: SI <premisas> ENTONCES <conclusiones>
regla 2: SI <premisas> ENTONCES <conclusiones>
•
•
•
regla n: SI <premisas> ENTONCES <conclusiones>

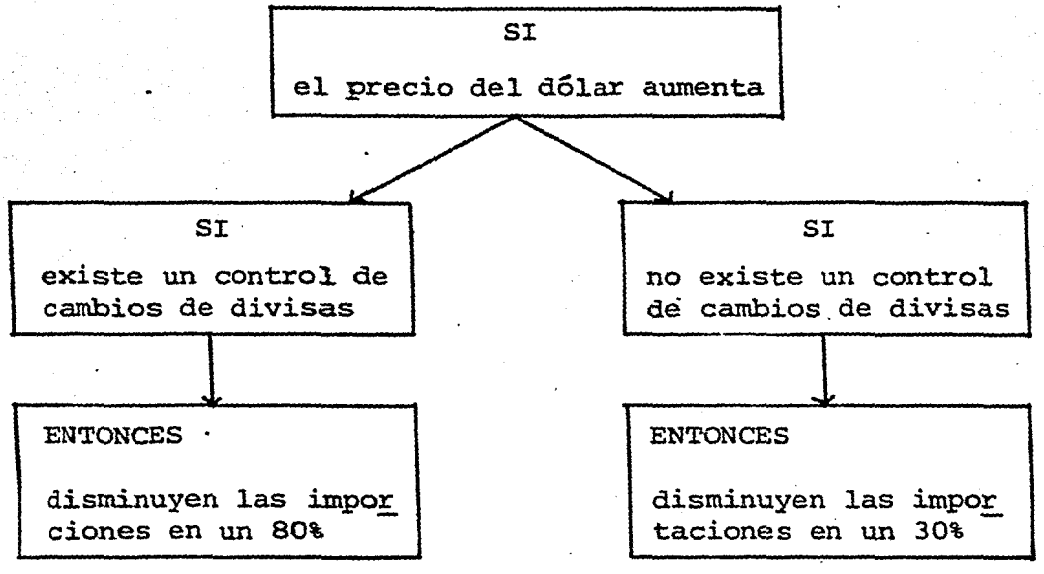
Siguiendo con el ejemplo anterior, se añadirá otra - regla más la cual señala como condición adicional que si el precio del dólar aumenta y no existe un control de cambio - de divisas, entonces las importaciones solo se reducen en - un 30%.

Esto en forma de reglas de producción se representa- de la siguiente manera:

Regla 1: SI <el precio del dólar aumenta> ENTONCES "pendiente"
REGLA 2: SI "pendiente" y <existe un control de cambios de divisas> ENTONCES <disminuyen las importaciones en un 80%>
Regla 3: SI "pendiente" y <si no existe un control de cambios de divisas> ENTONCES <disminuyen las importaciones en un - 30%>

Se observa que en la regla 1 la conclusión es par- - cial y pasa a formar parte de las premisas de las reglas 2 y 3, las cuales contienen las dos posibles conclusiones finales.

El ejemplo anterior representado en forma de "árbol" es el siguiente:

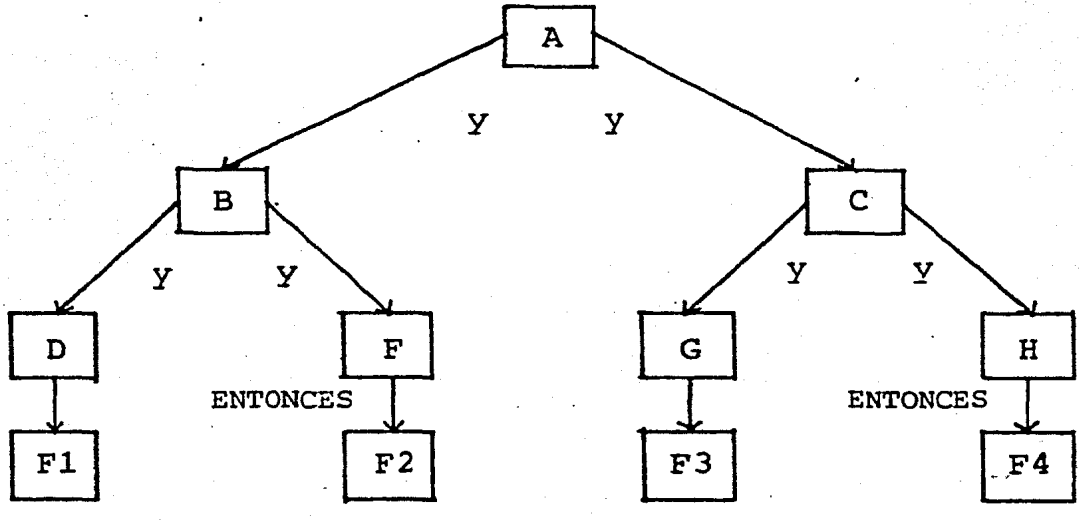


Se puede observar que el conocimiento puede expresarse tanto en reglas de producción como en forma de árbol, el cual tiene la característica de que para poder continuar -- con determinada ramificación, debe de cumplirse el nodo o - condición siguiente, así como también para poder llegar a - un nodo terminal o conclusión final es necesario que todos- los nodos o conclusiones anteriores que lleven a ésta se --

cumplan.

Para poder lograr una mejor comprensión de estos dos tipos de representación, se muestra un ejemplo donde se presenta el conocimiento inicialmente en forma de árbol, el -- cual posteriormente se representa en forma de reglas de producción.

La representación en forma de árbol es:



En esta representación en forma de árbol, A, B, C, D, F, G y H son las premisas y F1, F2, F3 y F4 son las conclusiones.

Puede observarse que B debe de cumplirse para poder- continuar con D o con F, al igual que C debe de cumplirse -

para poder seguir con G o con H.

La representación del árbol anterior en forma de reglas de producción es la siguiente:

```
Regla 1:  SI   A        ENTONCES  "pendiente 1"
Regla 2:  SI  "pendiente 1"  y  B  ENTONCES "pendiente 2"
Regla 3:  SI  "pendiente 2"  y  D  ENTONCES F1
Regla 4:  SI  "pendiente 2"  y  F  ENTONCES F2
Regla 5:  SI  "pendiente 1"  y  C  ENTONCES "pendiente 3"
Regla 6:  SI  "pendiente 3"  y  G  ENTONCES F3
Regla 7:  SI  "pendiente 3"  y  H  ENTONCES F4
```

Puede observarse en las reglas anteriores que los diagnósticos o conclusiones finales son F1, F2, F3 y F4, mientras que el resto de las conclusiones son parciales.

La forma de implementar estas reglas en el sistema experto, se hará de acuerdo a la siguiente sintáxis:

```
(REGLA <identificador 1> (SI ( <premisas> ) (ENTONCES  <conclusiones>)))
(REGLA <identificador 2> (SI ( <premisas> ) (ENTONCES  <conclusiones>)))
                         :
(REGLA <identificador n> (SI ( <premisas> ) (ENTONCES  <conclusiones>)))
```

Observése que el identificador sirve para diferenciar entre una y otra regla.

Para indicar al Experto cuando se trata de una conclusión final, se crea una lista de listas denominada en este caso DIAGNOSTICOS, en la cual se encuentran reunidas las diferentes hipótesis o diagnósticos finales.

La estructura de esta lista es la siguiente:

((<hipótesis 1>) (< hipótesis 2>) . . . (< hipótesis n >))

Al usar un sistema basado en este tipo de reglas de producción, lo que se busca es encontrar la regla que valide la hipótesis, la cual debe de estar incluída en la lista denomínada DIAGNOSTICO.

Al contrastar la representación en forma de árbol -- contra la representación en forma de reglas de producción, - se puede observar que estas reglas no siempre tienen como - conclusión de ellas una conclusión final, es decir, estas - reglas pueden estar conectadas indirectamente unas con - - otras, mientras que en la representación en forma de árbol- la conexión es necesariamente directa.

Este Sistema Experto maneja el conocimiento representado en forma de reglas de producción, lo que da como resultado una gran versatilidad en el momento de accesar o modificar el conocimiento, ya que se le pueden añadir en cualquier orden tantas reglas como sean necesarias, ya que el - mismo Experto tiene como tarea conectar y ordenar unas con- otras en el momento requerido.

3. SISTEMAS EXPERTOS

3.1 INTRODUCCION.

La construcción de Expertos Artificiales ha sido una de las tareas que se han realizado dentro de la Ingeligencia Artificial y en la actualidad lo que se ha dado en llamar la Ingeniería del Conocimiento está enfocada a la construcción de estos Sistemas Expertos.

Los sistemas Expertos están formados por una colección de rutinas que al funcionar tratan de igual el nivel de competencia de un Experto Humano dentro de cierta área del conocimiento.

La idea central que se maneja en la construcción de un Experto, es tratar de poner el conocimiento a trabajar, ya que la primera tarea de éste es reponder preguntas acerca del conocimiento.

Generalmente al realizar un programa tradicional de computación, se organiza el conocimiento en dos niveles: - uno formado por los datos y el otro por un programa o función.

Sin embargo, los Sistemas Expertos en su mayoría organizan el conocimiento en tres niveles: los datos, la base del conocimiento y el control (25).

En el nivel de los datos se encuentra definido el conocimiento específico del problema a tratar, éste es llamado también conocimiento declarativo, el cual puede organizarse en forma de reglas de producción, Listas de asociación (Frames) o mezclas de ellas.

La estructura de las reglas de producción se dicuten el capítulo 2 de este trabajo y presentan la siguiente estructura:

SI <premisas> ENTONCES <conclusiones>

Las listas de asociación llamadas también Marcos de Minsky, fueron ideadas por él, en el año de 1974, como otra forma de representar el conocimiento.

Cada Marco es una lista que contiene un nombre asociado a él y una colección de sublistas donde se encuentra definido el conocimiento de cierta área. A cada una de estas sublistas le antecede una identificación o etiqueta.

La estructura general de los Marcos de Minsky es la siguiente:

(nombre-del-marco (N1( ) N2( )... Nn( )))

donde N1, N2... Nn son etiquetas de cada una de las sublistas.

Como ejemplo de este tipo de representación se presenta un marco donde se encuentran definidas las capitales de dos de los estados de la República Mexicana:

(Capitales (DF(Ciudad de México) Puebla (Puebla)...etc.))

En el nivel de la base del conocimiento se encuentra definido como manipular los datos para finalmente resolver un problema.

El nivel de control se encuentra formado por la estrategia que se debe seguir en la manipulación del conocimiento para obtener la solución de un problema.

Esta estrategia de trabajo puede estar orientada en dos diferentes formas: una de ellas denominada "hacia adelante" (Forward) y la otra denominada "hacia atrás" (backward).

El sistema que trabaja "hacia adelante" maneja los hechos conocidos hasta obtener una conclusión o diagnóstico final, es decir, empieza a analizar los hechos hasta en-

contrar aquélla que pueda ser aplicada. En este tipo de -- Sistemas se revisan primero aquellas reglas que dependen de los hechos conocidos.

Para ilustrar este tipo de estrategia se da el si- - guiente ejemplo:

Ejemplo: Queremos clasificar el tipo de animal dependiendo del tipo de alimentación de éste. Para ello se tienen las siguientes reglas de producción:

(Regla No. 1
(SI (el animal sólo come carne))
(ENTONCES (el animal es un carnívoro)))

(Regla No. 2
(SI (El animal sólo come hierbas))
(ENTONCES (el animal es herbívoro))

Como se observa los diagnóstico finales son:

((animal es carnívoro) (animal es hervívoro))

La forma en la que el Experto "hacia adelante" analiza estas reglas es la siguiente.

Se revisa la premisa de la primera regla y en caso - de ser cierta se emite como diagnóstico final que el animal es un carnívoro. En caso contrario se revisa la premisa de la siguiente regla y si ésta es cierta se concluye que el - animal es un herbívoro. Si al revisar todas las reglas no-

se llega a emitir ningún diagnóstico, entonces el Sistema - Experto debe de generar un mensaje que indique este hecho - como diagnóstico final.

La otra forma de funcionar de un Sistema Experto que es la que se usa en este trabajo es "hacia atrás", la cual- parte de una hipótesis inicial a los hechos.

La estrategia que sigue este tipo de Experto es revisar aquellas reglas que validen la hipótesis seleccionada - en turno y para ello se revisan aquellas reglas que la apoyen. Ilustraremos la forma en la que trabaja este tipo de- Experto retomando el ejemplo anterior pero modificando el - orden de los diagnósticos finales de la siguiente manera:

((animal es herbívoro) (animal es carnívoro))

La forma en la que trabaja un Experto de este tipo - es la siguiente.

Selecciona aquellas reglas que apoyen la primera hipótesis que en este caso es (animal es herbívoro), la regla que apoya esta hipótesis es la No. 2, si su premisa es cierta entonces se concluye que el animal es un herbívoro. En- caso contrario se trata de probar la siguiente hipótesis -- que en este caso es (animal es carnívoro) y si la premisa - de la regla No. 1 se cumple, entonces se concluye que el --

animal es carnívoro.

En el caso de que ninguna hipótesis se pueda confirmar, se emite un mensaje que indique este hecho como tarea-final del Sistema Experto.

Al comparar estos dos tipos de estrategias no se puede decir que alguna de ellas sea mejor que la otra, sin embargo, dependiendo del objetivo final del Sistema Experto,- el uso de alguna de ellas puede ser más eficiente.

Una de las estrategias a seguir para decidir cual de estas dos formas es la adecuada en la construcción de un -- Sistema Experto es la siguiente.

Si lo que interesa saber es qué Hipótesis o Diagnóstico se puede obtener a partir de ciertos hechos conocidos- entonces la estrategia recomendada es "hacia adelante".

Pero si lo que interesa es validar una hipótesis o - Diagnóstico, la estrategia recomendada es "hacia atrás".

Como la finalidad de los Expertos usados en este trabajo consiste en validar una hipótesis, la forma que se consideró pertinente para el funcionamiento del Sistema Experto fue "hacia atrás".

3.2 ESTRUCTURA DEL EXPERO ARITIFICIAL

El Sistema Experto está formado por una colección de funciones, donde cada una de ellas resuelve una parte del problema. Como se mencionó anteriormente, en LISP se pueden dividir problemas en subproblemas creando funciones especiales para resolver cada caso.

Esto es parte de las ideas de Henderson (26) el cual señala que una forma de plantear un problema se logra mediante la construcción de una función o esqueleto general, el cual resuelve una parte del problema y deja tareas pendientes por resolver, lo cual se logra mediante otras funciones.

El Sistema Experto está formado por una función o esqueleto general denominado *DIAGNOSTICA*, el cual emplea diferentes funciones para poder emitir un diagnóstico final, éstas son:

VERIFICA, EN-ENTONCES, RECUERDAS, MEMORIZA, ENSAYA-REGLA, ENSAYA-PREMISA, ENTONCES-USA, ENSAYA-REGLA+ y ENSAYA PREMISA+

La forma en la que funciona el Sistema Experto es la siguiente: La función que llama al Sistema Experto es *DIAGNOSTICA* la cual revisa cada una de las diferentes hipótesis hasta encontrar aquélla que pueda ser validada, para ello hace uso de la función denominada *VERIFICA*.

Con la función *VERIFICA* se trata de validar la hipótesis en turno seleccionada por *DIAGNOSTICA*, revisando aquellas reglas con las cuales se pueda confirmar la hipótesis. Primero se buscan aquellas reglas que infieran la hipótesis directamente con la función *ENSAYA-REGLA*, y en caso de que no se confirme la hipótesis entonces con la función *ENSAYA-REGLA+* se buscan aquellas reglas que infieran la hipótesis indirectamente.

Tanto la función *ENSAYA-REGLA* como la función *ENSAYA-REGLA+* generan un diagnóstico afirmativo si todas las premisas de la regla se cumplen y al menos una de sus conclusiones no es ya un hecho conocido.

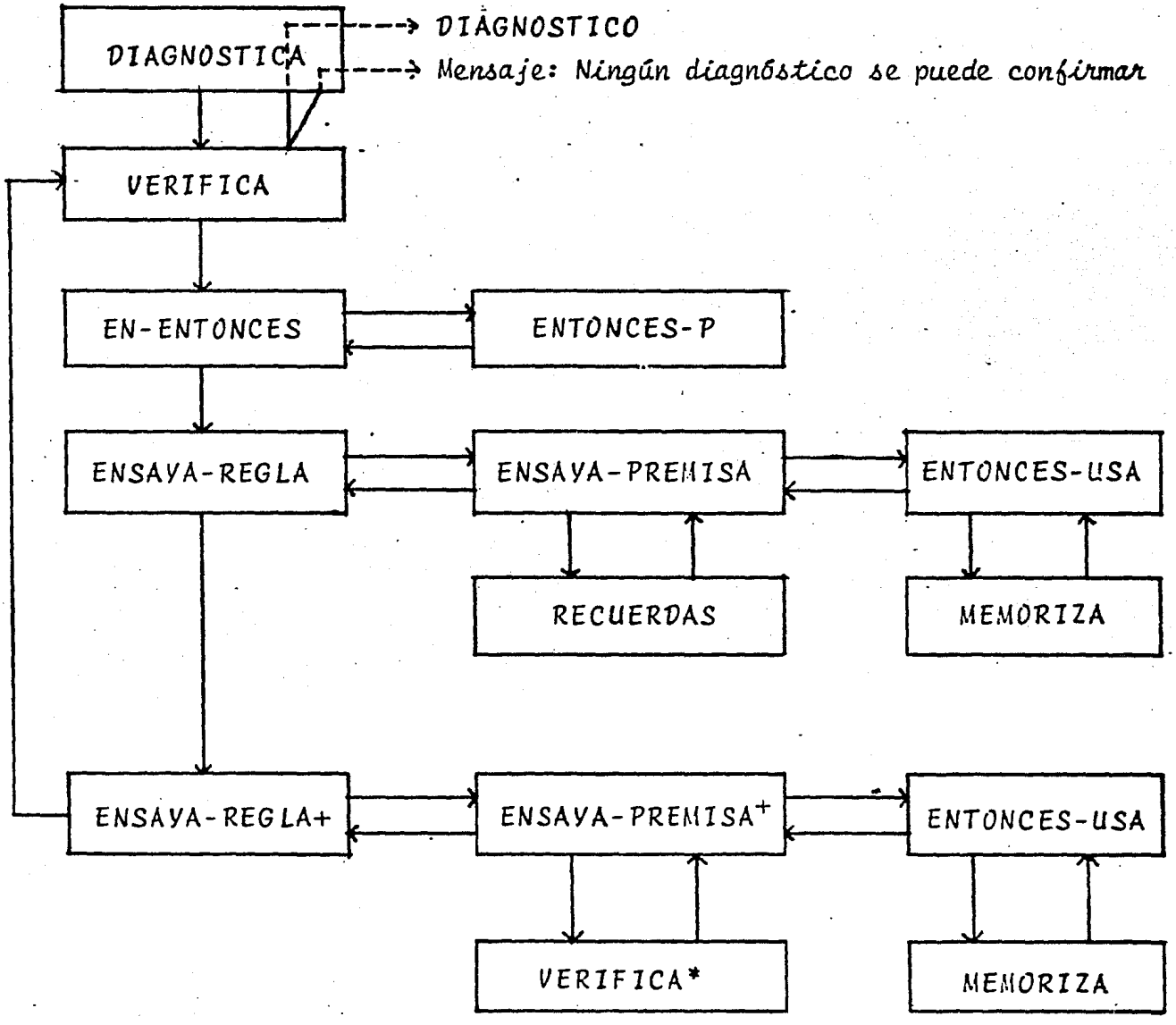
Para revisar las premisas de las reglas se usan las funciones *ENSAYA-PREMISA* y *ENSAYA-PREMISA+* y para revisar las conclusiones se usa la función *ENTONCES-USA*.

Para checar si un hecho es conocido o no *ENSAYA-PREMISA* usa la función *RECUERDAS* y *ENSAYA-PREMISA+* usa la función *VERIFICA*.

Para checar si alguna de las conclusiones ya es un hecho conocido o en caso contrario anexarla a la lista de hechos se usa la función *MEMORIZA*.

A continuación se muestra un diagrama que ilustra la dinámica de las funciones que integran al Sistema Experto.

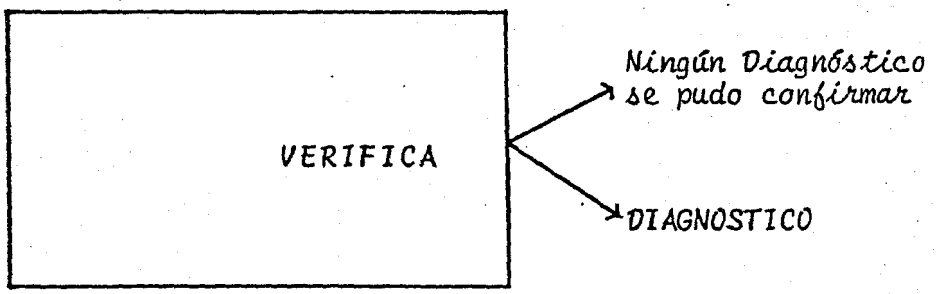
Nótese que la función *VERIFICA** señala la recursión- que la función *VERIFICA* hace sobre ella misma.



(*DIAGNOSTICA*):

La función general llamada *DIAGNOSTICA*, es una fun-- ción que no necesita argumentos y busca validar alguno de - los diferentes diagnóstico. Para ello usa la función *VERI-FICA*. El resultado de la función *DIAGNOSTICA*, puede ser el diagnóstico confirmado o la emisión de un mensaje que indica que ningún diagnóstico se pudo confirmar.

En el siguiente esquema se representa la función - - *DIAGNOSTICA*:



PROGRAMACION

```
LAMBDA NIL
 (PROG (POS PREGUNTADOS)
       (SETQ POS DIAGNOSTICOS)
       ETIQUETA
       (COND ((EQ POS)
              (PRINT (QUOTE NINGUN-DIAGNOSTICO-SE-PUEDE-CONFIRMAR)) (RETURN NIL
)
             ((VERIFICA (CAR POS))
              (PRIN1 (QUOTE EL-DIAGNOSTICO)) (PRIN1 (CAR POS)) (PRINT (QUOTE ES
EL-CORRECTO)) (RETURN (CAR POS))))
       (SETQ POS (CDR POS))
       (GO ETIQUETA)))
```

(VERIFICA):

Esta función tiene como argumento una de las hipótesis denominada *HECHO* y emite como resultado el valor *T*, si se puede validar esta hipótesis con la ayuda de las reglas de producción, las cuales forman parte del conocimiento del Experto, o se genera el valor *NIL* en caso contrario.

El funcionamiento de la función *VERIFICA* puede explicarse con los siguientes pasos:

paso 1: Si *HECHO* o *HIPOTESIS* que queremos demostrar, ya se encuentra en la lista de Hechos, termina la función generando como resultado *T*.

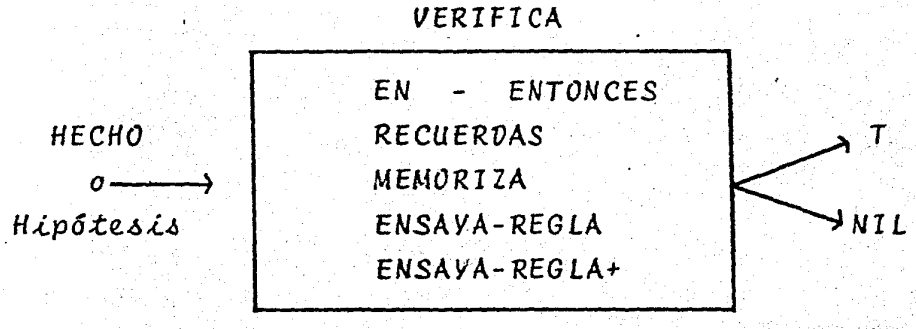
paso 2: Se construye una lista de reglas seleccionando aquellas que -- tengan a este *HECHO* en su conclusión.

paso 3: Si el paso 2 no se realiza se le pregunta al usuario.

paso 4: Se usa *ENSAYA-REGLA* para ver si el *HECHO* es deducible directamente.

paso 5: Se usa *ENSAYA-REGLA+* para ver si *HECHO* es deducible indirectamente.

La función *VERIFICA* podemos representarla en un diagrama de la siguiente manera:



PROGRAMACION

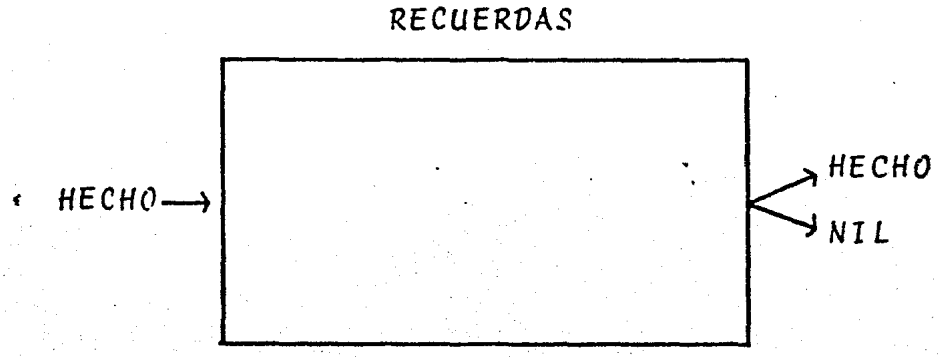
```
(LAMBDA (HECHO)
  (PROG (RELEVANTE1 RELEVANTE2)
        (COND ((RECUERDAS HECHO)
               (RETURN T)))
        (SETQ RELEVANTE1 (EN-ENTONCES HECHO))
        (SETQ RELEVANTE2 RELEVANTE1)
        (COND ((EQ RELEVANTE1)
               (COND ((MEMBL HECHO PREGUNTADOS)
                      (RETURN NIL))
                     ((AND (PRIN1 (QUOTE (ES ESTE HECHO CIERTO?)))
                           (PRIN1 HECHO)
                           (READ))
                      (MEMORIZA HECHO) (RETURN T))
                     (T (SETQ PREGUNTADOS (CONS HECHO PREGUNTADOS)) (RETURN NIL
)))))
```

```
ETIQUETA1
(COND ((EQ RELEVANTE1)
       (GO ETIQUETA2))
      ((ENSAYA-REGLA (CAR RELEVANTE1))
       (RETURN T)))
(SETQ RELEVANTE1 (CDR RELEVANTE1))
(GO ETIQUETA1)
ETIQUETA2
(COND ((EQ RELEVANTE2)
       (GO SALIDA))
      ((ENSAYA-REGLA+ (CAR RELEVANTE2))
       (RETURN T)))
(SETQ RELEVANTE2 (CDR RELEVANTE2))
(GO ETIQUETA2)
SALIDA
(RETURN NIL)))
```

(RECUERDAS):

Esta función tiene como entrada el *HECHO* y revisa si éste es miembro de la lista denominada *HECHOS*, en cuyo caso el valor que genera la función *RECUERDAS* es el valor del -- *HECHO* mismo o en caso contrario el valor es *NIL*.

La representación de *RECUERDAS* es la siguiente:



PROGRAMACION

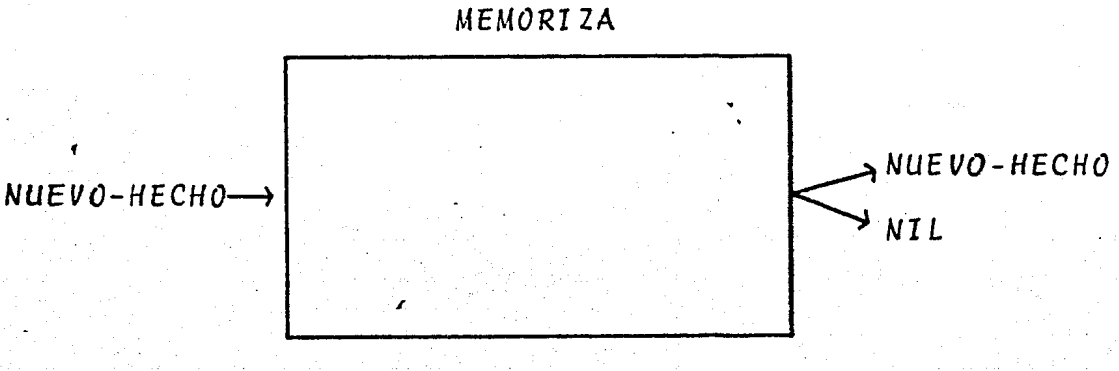
```
<LAMBDA <HECHO>
  <COND <<MEMBL HECHO HECHOS>
          HECHO>
         <T NIL>>>
```

(MEMORIZA):

Esta función acepta como argumento una hipótesis denominada nuevo-hecho y mediante esta función se implementan nuevos hechos en la lista denominada *HECHOS* y se genera como resultado el *HECHO* mismo si éste se implementó como un nuevo elemento. Se genera el valor *NIL* en caso de que este

hecho ya se encuentre dentro de la lista.

Esta función se representa en el siguiente diagrama:



PROGRAMACION

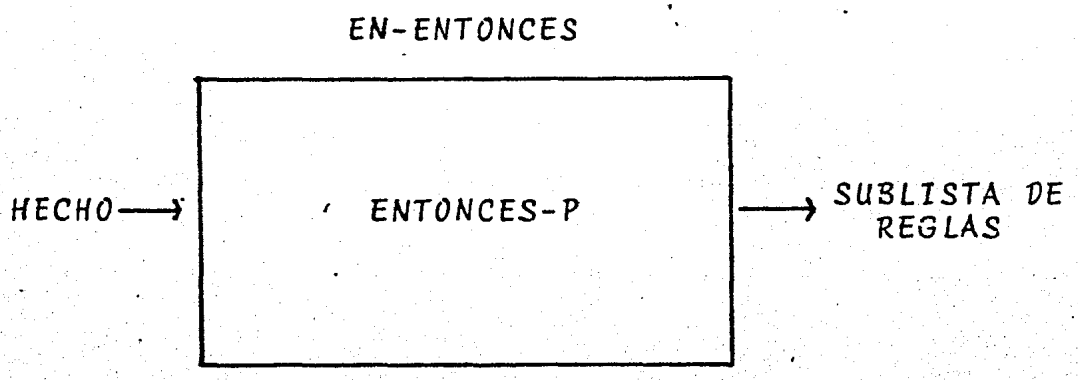
```
(LAMBDA (NUEVOS-HECHOS)
  (COND ((MEMBL NUEVOS-HECHOS HECHOS)
         NIL)
        (T (SETQ HECHOS (CONS NUEVOS-HECHOS HECHOS)) NUEVOS-HECHOS)))
```

(EN-ENTONCES):

Esta función revisa cada una de las reglas de producción para comprobar si el *HECHO* que se da como argumento se encuentra entre sus conclusiones. Para ello se usa la función *ENTONCES-P* y se genera como resultado la colección de reglas que cumplen con esta condición. Es decir, se genera-

una lista con las reglas que contengan a la hipótesis denominada *HECHO*.

Se representa esta función en el siguiente diagrama:



PROGRAMACION

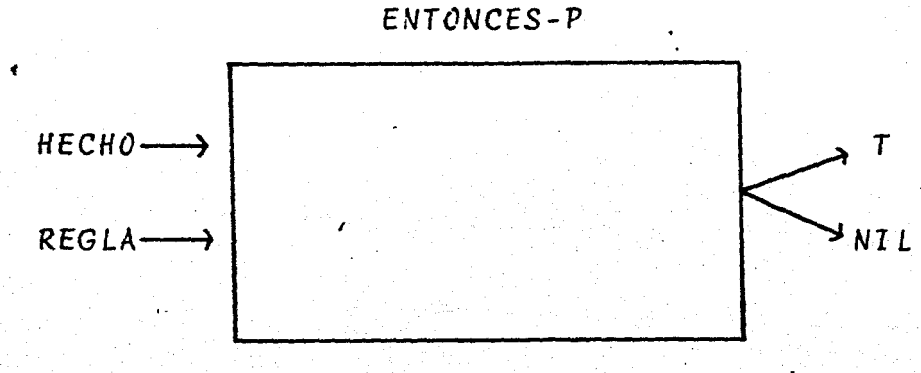
```
<LAMBDA <HECHO>
  <MAPCA <QUOTE <LAMBDA <R>
    <COND <<ENTONCES-P HECHO R>
           R>>>> REGLAS>>
```

(ENTONCES-P):

Esta función pregunta si el *HECHO* que se da como primer argumento se encuentra en la conclusión de la regla que-se da como segundo argumento, generando el valor *T* si ésto -

se cumple o el valor *NIL* en caso contrario.

El esquema de esta función es el siguiente:



PROGRAMACION

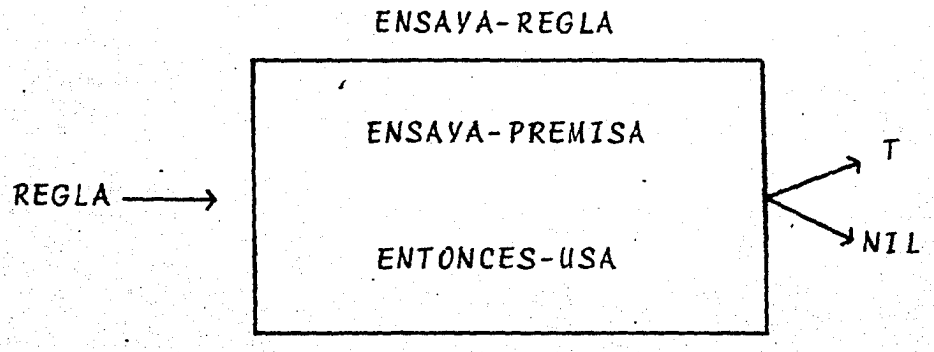
```
(LAMBDA (HECHO REGLA)
  (MEMBL HECHO (CDR (CAR (CDR (CDR (CDR REGLA)))))))
```

(ENSAYA-REGLA):

ENSAYA-REGLA es una función que verifica, con la ayuda de la función *ENSAYA-PREMISA*, si todas las premisas de la regla que se da como argumento se encuentran en la lista de *HECHOS*, y mediante la función *ENTONCES-USA* se revisa que almenos una de las conclusiones de esta regla no se encuentra-

en la lista de *HECHOS*. Si este es el caso se genera como - resultado el valor *T* y el valor *NIL* en el caso contrario. - Esta función está formada por dos funciones *ENSAYA-PREMISA* y *ENTONCES-USA*.

El esquema de esta función es el siguiente:



PROGRAMACION

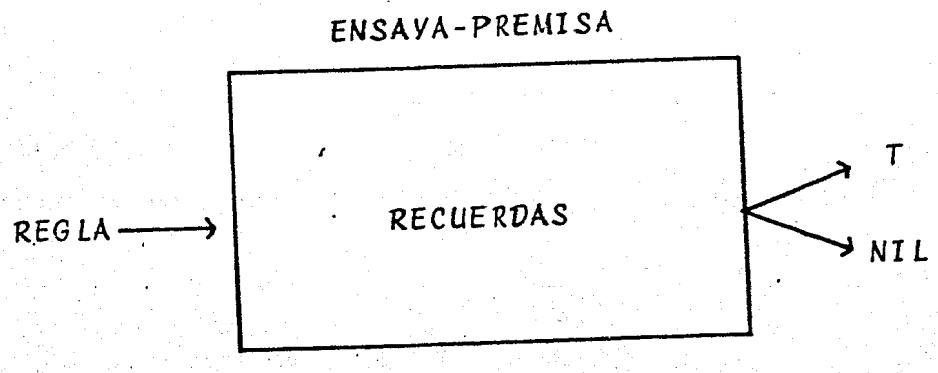
```
(LAMBDA (REGLA)
  (COND ((AND (ENSAYA-PREMISA REGLA)
              (ENTONCES-USA REGLA))
         (SETQ USADAS (APPEND USADAS REGLA)) T)
        (T NIL)))
```

(ENSAYA-PREMISA):

Esta función prueba si todas las premisas de la re-- gla que se da como argumento se cumplen, usando la función-

RECUERDAS para ello. Si todas ellas se cumplen, se genera- como resultado el valor *T*, o el valor *NIL* en caso contra- - rio.

La representación de esta función es la siguiente:



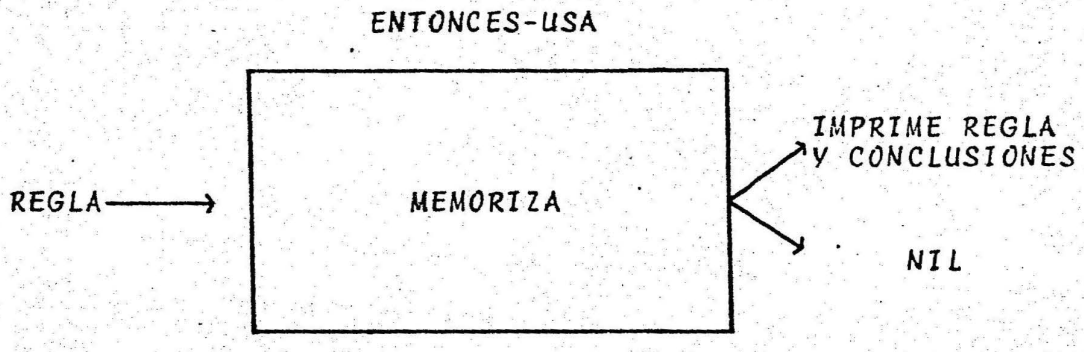
PROGRAMACION

```
(LAMBDA (REGLA)
  (PROG (IFS)
        (SETQ IFS (CDR (CAR (CDR (CDR REGLA)))))
        ETIQUETA
        (COND ((EQ IFS)
               (RETURN T))
              ((RECUERDAS (CAR IFS))
              )
              (T (RETURN NIL)))
        (SETQ IFS (CDR IFS))
        (GO ETIQUETA)))
```

(ENTONCES-USA):

Esta función prueba si todas las premisas de la re-- gla que se da como argumento se encuentran ya añadidas, - - usando la función *MEMORIZA* para ello. Si este es el caso,- se genera como resultado la impresión de la regla junto con las conclusiones de ésta, o se genera el valor *NIL* en caso- contrario.

La representación de esta función es la siguiente:



PROGRAMACION

```
(LAMBDA (REGLA)
  (PROG (ENTONCES-S SUCESORES)
        (SETQ ENTONCES-S (CDR (CAR (CDR (CDR (CDR REGLA))))))
        ETIQUETA
        (COND ((EQ ENTONCES-S)
               (RETURN SUCESORES))
              ((MEMORIZA (CAR ENTONCES-S))
               (PRIN1 (QUOTE (LA-REGLA))) (PRIN1 (CAR (CDR REGLA))) (PRIN1 (QUO
E (DEDUCE-QUE))) (PRINT (CAR ENTONCES-S)) (SETQ SUCESORES T)))
        (SETQ ENTONCES-S (CDR ENTONCES-S))
        (GO ETIQUETA)))
```

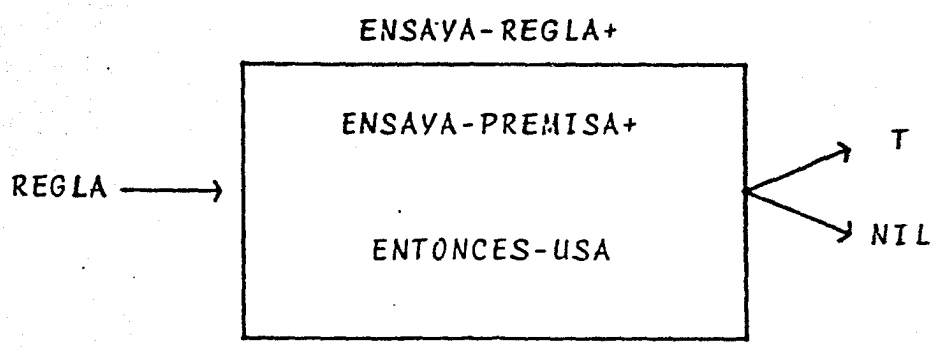
(ENSAYA-REGLA+):

Esta función trabaja igual que *ENSAYA-REGLA*, pero su diferencia radica en que se usa la función *ENSAYA-PREMISA+* en lugar de la función *ENSAYA-PREMISA*.

La función *ENSAYA-PREMISA+* usa a la función *VERIFICA* en donde la función *ENSAYA-PREMISA* usa la función *RECUERDAS* para revisar si todas las premisas de la regla que se da como argumento se cumplen y es aquí donde la función *VERIFICA* se usa en forma recursiva.

El argumento de la función *ENSAYA-REGLA+* es la regla y el resultado final al igual que *ENSAYA-REGLA* es *T*, si se cumplen las premisas de la regla o *NIL* en caso contrario.

El esquema de la función *ENSAYA-REGLA+* es el siguiente:



PROGRAMACION

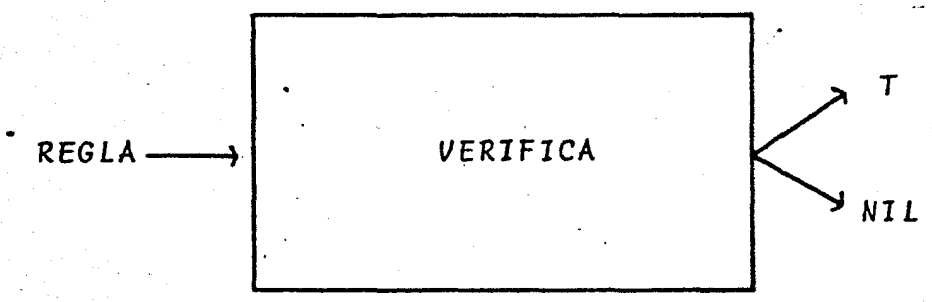
```
(LAMBDA (REGLA)
  (COND ((AND (ENSAYA-PREMISA+ REGLA)
              (ENTONCES-USA REGLA))
         (SETQ USADAS (APPEND USADAS REGLA)) T)
        (T NIL)))
```

(ENSAYA-PREMISA+):

Esta función al igual que ENSAYA-PREMISA prueba si - todas las premisas de la regla que se da como argumento se - cumplen, usando para ello la función VERIFICA, generando como resultado el valor T si este es el caso o el valor NIL - en caso contrario.

La representación de esta función es la siguiente:

ENSAYA-PREMISA+



PROGRAMACION

```
(LAMBDA (REGLA)
  (PROG (IFS)
        (SETQ IFS (CDR (CAR (CDR (CDR REGLA)))))
        ETIQUETA
        (COND ((EQ IFS)
               (RETURN T))
              ((VERIFICA (CAR IFS))
              )
              (T (RETURN NIL)))
        (SETQ IFS (CDR IFS))
        (GO ETIQUETA)))
```

4. RESULTADOS

Al Sistema Experto mostrado en el capítulo 3, se le hicierom modificaciones mínimas para aplicarlo en dos diferentes áreas del conocimiento, creando con ello a dos tipos de Expertos, uno de ellos un Experto Ortografista y el otro un Experto Estadístico.

A continuación se explica el funcionamiento y la estructura de cada uno de ellos, así como algunos ejemplos que muestran su eficiencia.

4.1 EXPERTO ORTOGRAFISTA

Este experto ante una palabra dada por el usuario le corrige los errores ortográficos, basándose en las reglas que rigen el uso correcto de las letras b y v, de las palabras del idioma Español.

Este Sistema Experto puede ser aplicado en el área educativa donde existen grandes problemas sobre Ortografía los cuales persisten en el estudiantado y aún en los profesionistas; tal vez ésto es debido a la falta de un cierto orden en cuanto a la presentación de las reglas ortográficas en los libros de texto correspondientes, lo cual se re

fleja en la falta de comprensión y retención de éstas.

Con la revisión de las reglas que usa el Sistema Experto el estudiante puede tener un enfoque diferente en el estudio de estas reglas y además al hacer uso del Sistema-Experto puede el mismo llevar una contabilidad sobre sus - posibles fallas y reincidencia en ellas.

Así el Sistema Experto juega un doble papel, como informante y como revisor.

Las reglas Ortográficas que forman el conocimiento - del Experto son tomadas de un libro de ortografía (27), que se usa como parte de un curso de Redacción.

Para hacer uso del Experto fue necesario representar estas reglas en forma de reglas de producción de las cua-- les se habló en el capítulo 2.

Estas reglas se aplican sobre cada una de las letras que integran la palabra "problema", usando para ello ciertas funciones que checan la sintáxis de cada letra en relación a las demás letras y a la posición que ésta ocupa en- la palabra. Para ello se construyeron las premisas de las reglas ortográficas, en forma de funciones *LISP*, donde el- primer elemento es la función que checa la sintáxis y el -

resto corresponde a sus argumentos. De este modo para obtener conclusiones o emitir diagnósticos deben de cumplirse todas las funciones que integren las premisas de la regla.

Por ejemplo si el usuario tiene duda sobre la palabra *CAMBIO*, la regla que se aplica es la siguiente:

"después de M se escribe B"

La cual le informa al Sistema Experto que como en efecto la letra B se encuentra precedida de *M* entonces debe de concluir que la forma correcta de escribir la palabra es con *B*. Lo cual no modificaría la forma en la que el usuario escribió la palabra.

La función que hace funcionar al Experto Ortografista es:

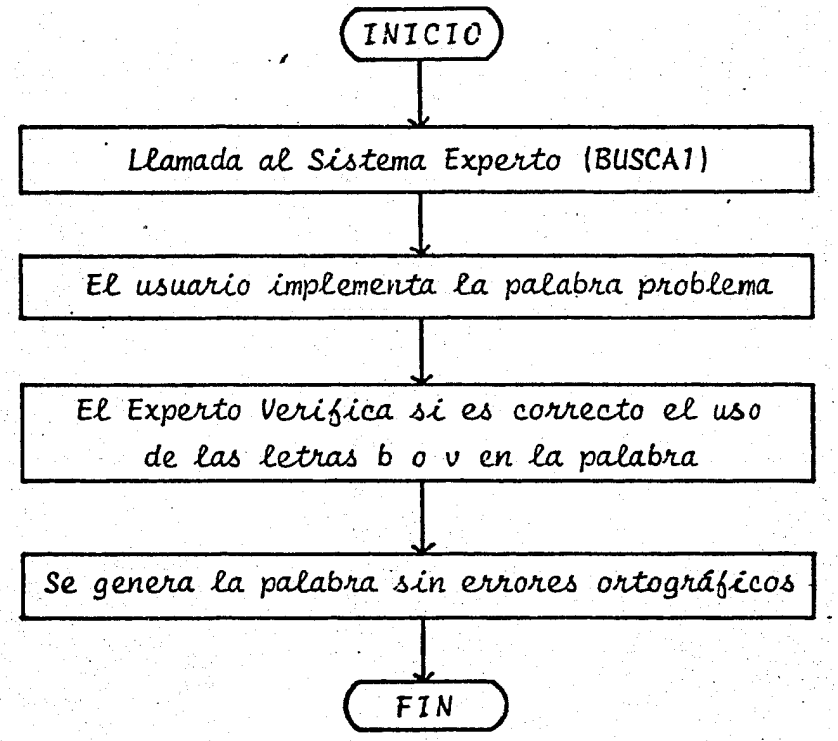
(BUSCA1)

la cual se explicará posteriormente en forma detallada.

Una vez que el Sistema Experto es activado se requiere de la palabra problema, la cual debe ser dada por el usuario. El Experto se consulta varias veces para verificar la sintáxis de cada una de las letras que integren la palabra y genera como resultado final la palabra sin errores ortográficos. Cuando no existe ninguna función prede-

finida que pueda resolver algún hecho, el experto pregunta al usuario si éste es cierto o falso, para así implementar lo dentro de su información.

La representación del funcionamiento del Experto Ortografista en un diagrama es el siguiente:



Como el Experto Ortografista tiene un razonamiento - regresivo se verifica primero, en este caso, si la palabra- se escribe con "b" y si ésto no es lo correcto entonces se- verifica si la palabra se escribe con la letra "v". En ca-

so contrario se emite un mensaje que informa al usuario que ningún diagnóstico se puede confirmar.

Este Experto Ortografista cuenta con una lista de -- acepciones donde se revisa la forma de escribir una palabra, la cual cambia en alguna de sus letras dependiendo tan solo- del significado de ésta.

Así el Experto pregunta al usuario el significado -- asociado a la palabra que éste escribió y si éste es el correcto la labor del Experto termina al emitir la palabra -- sin cambio alguno, pero si el significado no corresponde a- lo que el usuario quería escribir entonces el Sistema Experto cambia la forma de escribir la palabra reemplazando la - letra correspondiente.

Por ejemplo:
Si el usuario escribe *TUBO*, el Experto revisa las siguien-- tes acepciones:

((*TUBO CILINDRO*) (*TUVO VERBO*))

Preguntando al usuario si se trata de un cilindro, - en caso afirmativo el Experto genera finalmente la palabra- sin cambio alguno; en caso negativo, el Experto genera la - acepción de la palabra que en este caso es *TUVO*.

A continuación se muestra el conocimiento que maneja este Experto en forma de Reglas de Producción, algunas funciones que se implementaron para el manejo de estas reglas, las modificaciones hechas al Experto Básico y algunos ejemplos del funcionamiento de este Experto Ortografista.

Las reglas de producción que forman la base del conocimiento del Experto Ortografista tienen la siguiente estructura:

SI <función 1 argumento 1> <función 2 argumento 2 >...<función N argumento N >

ENTONCES <conclusiones o Diagnósticos >

Las reglas que usa este Experto "ortografista son -- las siguientes:

```
<R B1 <SI <SEGUIDA-DE B/C/D/F/G/H/J/K/L/M/N/R/S/T/U/>> <ENTONCES <B>>>
:A
<R B.1 <SI <PRINCIPIO-DE-PALABRA *> <ALGUN <BON BUE>>> <ENTONCES <B2.1>>>
:A
<R B0 <SI <PRECEDIDA-DE M>> <ENTONCES <B>>>
```

```
:A
(R B2 (SI (SEGUIDA-DE U)) (ENTONCES (B2.1)))

:A
(R B3 (SI (B2.1) (NINGUN (VUESCENCIA VULGO VUESTRO VULNERAR VUELCO VUELTA))) (EN
TONCES (B)))

:A
(R B4 (SI (VERBO *) (COPRETERITO *)) (ENTONCES (B)))

:A
(R B5 (SI (PRINCIPIO-DE-PALABRA *) (ALGUN (BI BIS BIBL))) (ENTONCES (B)))

:A
(R B6 (SI (PRINCIPIO-DE-PALABRA *) (ALGUN (AL GO UR TA NU RA RO RU SA SO SU))) (
ENTONCES (B6.1)))

:A
(R B7 (SI (B6.1) (NINGUN (ALVEO ALVEOLO ALVO SAVIA))) (ENTONCES (B)))

:A
(R B8 (SI (TERMINADAS-EN (BILIDAD BUNDO BUNDA))) (ENTONCES (B8.1)))

:A
(R B9 (SI (B8.1) (NINGUN (MOVILIDAD))) (ENTONCES (B)))

:A
(R B10 (SI (VERBO *) (TERMINADAS-EN (BER BIR))) (ENTONCES (B10.1)))

:A
(R V1 (SI (VERBO *) (INFINITIVO-SIN B/V)) (ENTONCES (V1.1)))

:A
(R V2 (SI (VERBO *) (V1.1) (NO-FUTURO *)) (ENTONCES (V)))
```

```
:A
<R V3 <SI <PRINCIPIO-DE-PALABRA *> <ALGUN <AD CLA DI JO LE EXTRA IN PRA PRI PER
POR VEN NOV VIA LLA LLE LLO LLU SAL>>> <ENTONCES <V3.1>>>

:A
<R V4 <SI <V3.1> <NINGUN <DIBUJO BENCINA BENDECIR BENJAMIN>>> <ENTONCES <V>>>

:A
<R V5 <SI <ADJETIVO *> <TERMINADAS-EN <AVA AVE AVO AVU EVA EVE EVO EVU IVA IVE I
VO IVU>>> <ENTONCES <V5.1>>>

:A
<R V6 <SI <V5.1> <NINGUN <ARABE SILABA>>> <ENTONCES <V>>>

:A
<R V7 <SI <PRECEDIDA-DE B/N/D>> <ENTONCES <V>>>

:A
<R V8 <SI <PRINCIPIO-DE-PALABRA *> <ALGUN <VICE VILLA>>> <ENTONCES <V>>>

:A
<R V9 <SI <TERMINADAS-EN <VIRO VIRA IBORO IVORA IVO IVA>>> <ENTONCES <V9.1>>>

:A
<R V10 <SI <V9.1> <NINGUN <VIVORA>>> <ENTONCES <V>>>

:A
<R B11 <SI <B10.1> <NINGUN <VER HERVIR SERVIR VIVIR>>> <ENTONCES <B>>>

:A
*EOL*
```

Las conclusiones finales de este Experto se guardan en la lista denominada *DIAGNOSTICO*, la cual en nuestro caso tiene dos posibles alternativas o diagnósticos finales b o v, es decir, el Experto decide si la palabra se escribe correctamente con alguna de estas dos letras, las cuales se muestran en la siguiente lista:

((b) (v))

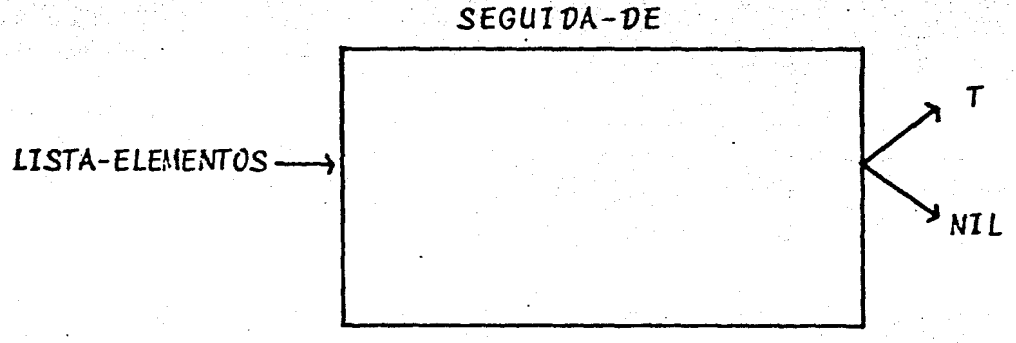
FUNCIONES QUE SE IMPLEMENTARON PARA EL USO DE LAS REGLAS

Para este tipo de reglas de producción se requirió - de la implementación de algunas funciones para permitir el- uso correcto de estas reglas. Estas funciones son las si-- guientes: *SEGUIDA-DE*, *PRECEDIDA-DE*, *ALGUN*, *NINGUN*, *TERMINA-DAS-EN*, *ACEPCIONES* y algunas funciones adicionales que forman parte de estas funciones. A continuación se presenta - la sintáxis de cada una de estas funciones acompañada de su programación correspondiente.

(SEGUIDA-DE):

Esta función revisa si la letra de la palabra está - seguida de alguno de los elementos que se dan como argumento denominado *LISTA-ELEMENTOS* en cuyo caso se genera el valor "T" o el valor *NIL* en caso contrario.

La representación de esta función es la siguiente:



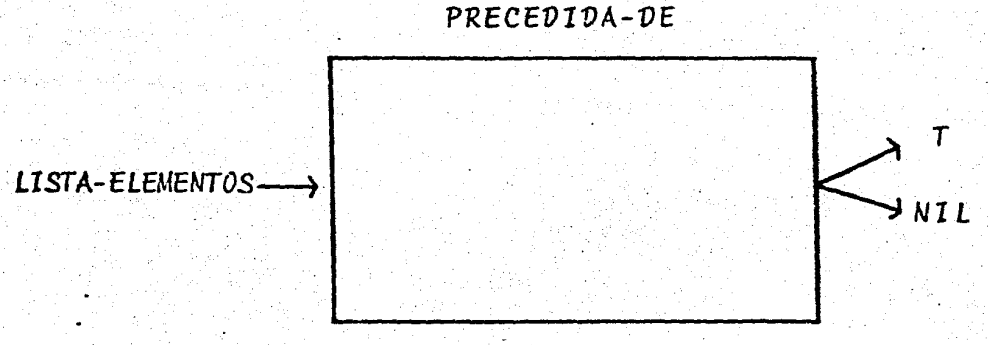
PROGRAMACION DE SEGUIDA-DE

```
(LAMBDA (LISTA-ELEMENTOS)
  (MEMBER (CAR (CDR PALABRA)) (UNPACK LISTA-ELEMENTOS)))
```

(PRECEDIDA-DE):

Esta función revisa si la letra en turno está precedida de alguno de los elementos que integran al argumento de esta función denominado *LISTA-ELEMENTO*, en cuyo caso se genera el valor "*T*" o el valor *NIL* en caso contrario.

El esquema de esta función es el siguiente:



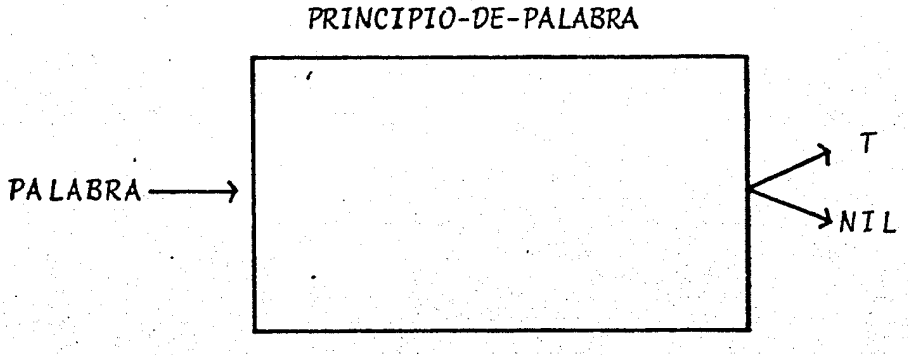
PROGRAMACION DE PRECEDIDA-DE

```
(LAMBDA (LISTA-ELEMENTOS)
  (MEMBER (CAR (@ REFERENCIA (SUB (SUB LREF (LEN PALABRA))))) (UNPACK LISTA-ELEM
ENTOS)))
```

(PRINCIPIO-DE-PALABRA):

Esta función checa si se está analizando el primer - elemento de la palabra en cuyo caso se genera el valor *T* o- el valor *NIL* en cáso contrario.

El esquema de esta función es el siguiente:



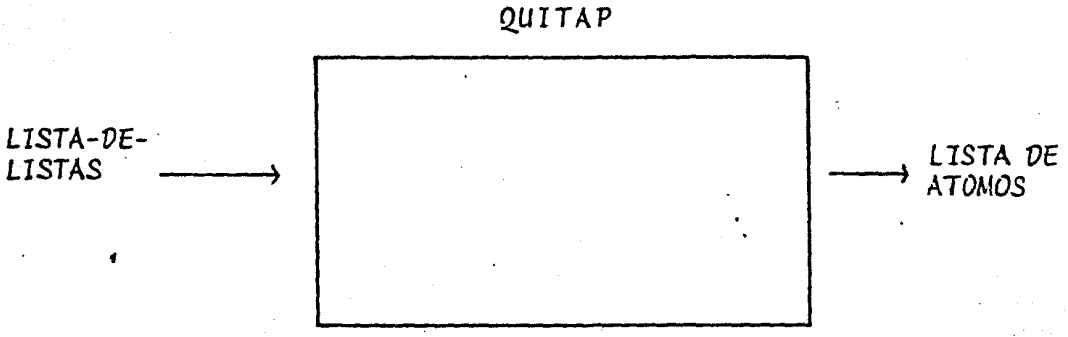
PROGRAMACION DE PRINCIPIO-DE-PALABRA

```
(LAMBDA (PALABRA)
  (COND ((EQ (LEN PALABRA) LREF)
         T)
        (T NIL)))
```

(QUITAP):

Esta función forma una lista de átomos con las lis-- tas del argumento denominadas *LISTA-DE-LISTAS*, es decir, -- quita los paréntesis posibles del argumento.

El esquema de esta función es el siguiente:



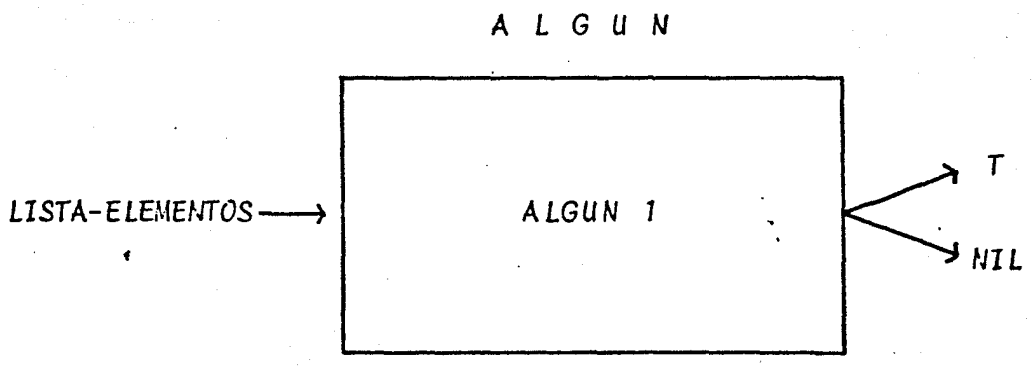
PROGRAMACION DE QUITAP

```
<LAMBDA <LISTA-DE-LISTAS>
  <COND <<EQ LISTA-DE-LISTAS>
         NIL>
        <<EQ <LEN <CAR LISTA-DE-LISTAS>> 0>
         <CONS <CAR LISTA-DE-LISTAS> <QUITAP <CDR LISTA-DE-LISTAS>>>>
        <T <CONS <CAR <CAR LISTA-DE-LISTAS>> <QUITAP <CDR LISTA-DE-LISTAS>>>>>>
```

(ALGUN):

Esta función tiene como argumento una lista de ele-- mentos denominada *LISTA-ELEMENTOS*. Mediante la función *ALGUN1* se checa si alguno de los elementos de esta forma parte de la palabra denominada *REFERENCIA*, que es una variable global en cuyo caso se genera *T* o el valor *NIL* en caso contrario.

El esquema de la función es:



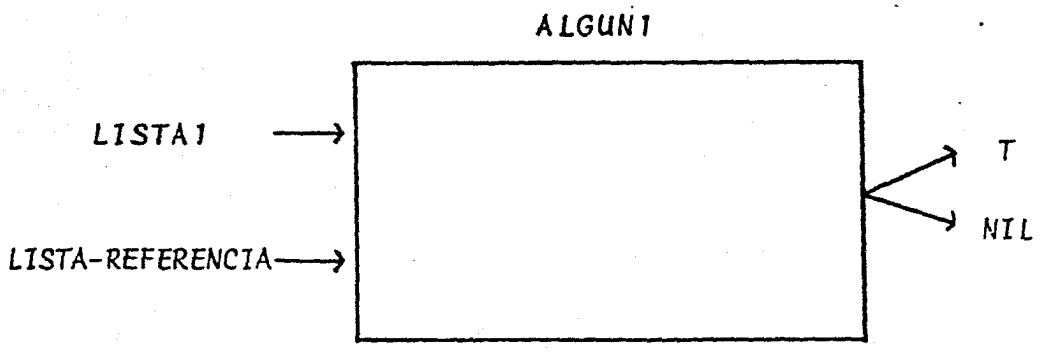
PROGRAMACION DE ALGUN

```
(LAMBDA (LISTA-ELEMENTOS)
  (ALGUN1 LISTA-ELEMENTOS (QUITAP REFERENCIA)))
```

(ALGUN1):

Esta función tiene como argumentos a las listas denominadas *LISTA* y *LISTA-REFERENCIA*. Esta función verifica, mediante la función *ESTA*, si alguno de los elementos de la primera lista forma parte de la segunda. Si éste es el caso el valor de la función es *T*, o se genera el valor *NIL* en caso contrario.

El esquema de esta función es el siguiente:



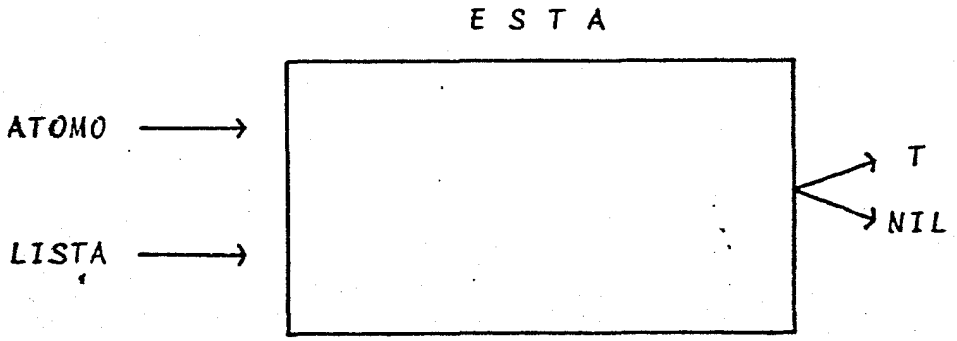
PROGRAMACION DE ALGUN1

```
(LAMBDA (LISTA LISTA-REFERENCIA)
  (COND ((EQ LISTA)
         NIL)
        ((ESTA (CAR LISTA) LISTA-REFERENCIA)
         T)
        (T (ALGUN1 (CDR LISTA) LISTA-REFERENCIA))))
```

(ESTA):

Esta función tiene como argumento un átomo y una lista. Mediante la función *JUNTA* se verifica si este átomo es parte constitutiva de la lista. Si éste es el caso el valor de la función es el átomo mismo, o se genera el valor *NIL* en caso contrario.

El esquema de la función es el siguiente:

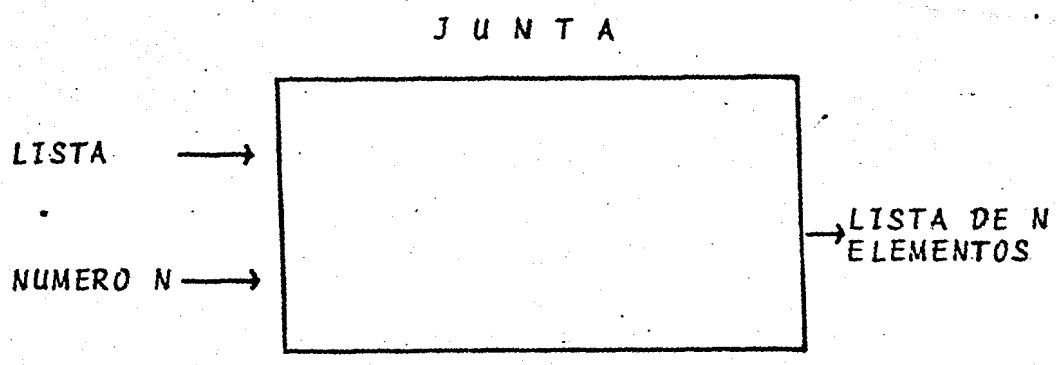


PROGRAMACION DE ESTA

```
<LAMBDA <ATOMO LISTA>
  <COND <<> <LEN <UNPACK ATOMO>> <LEN LISTA>>
         NIL>
        <<EQ <LEN <UNPACK ATOMO>> <LEN LISTA>>
         <COND <<EQUAL ATOMO <PACK LISTA>>
                 <RETURN <PACK LISTA>>>
                <T NIL>>>
        <T <EQUAL ATOMO <PACK <JUNTA LISTA <LEN <UNPACK ATOMO>>>>>>>>
```

(JUNTA):

La función *JUNTA* tiene dos argumentos, el primero de ellos es una lista y el otro es un número n: Esta función - genera como valor una lista con los primeros n elementos de la lista que se dió como primer argumento. El esquema de - esta función es el siguiente:



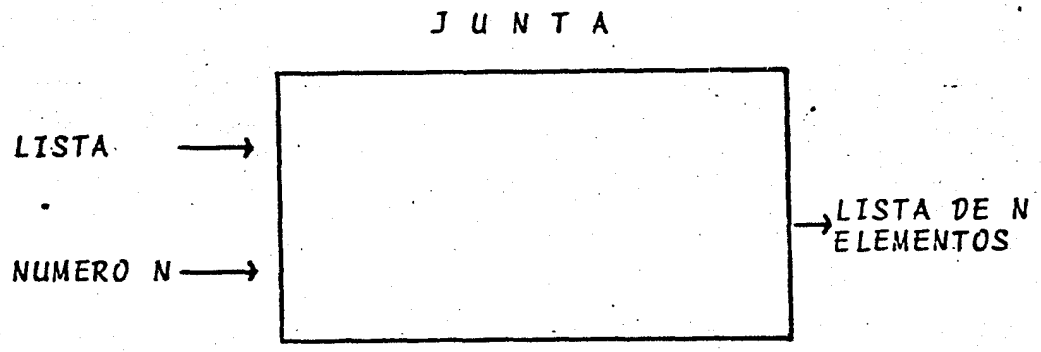
PROGRAMACION DE JUNTA

```
(LAMBDA (LISTA NUMERO-N)
  (COND ((EQ NUMERO-N 0)
         NIL)
        (T (CONS (CAR LISTA) (JUNTA (CDR LISTA) (SUB NUMERO-N 1))))))
```

(NINGUN):

Esta función tiene como argumento una lista de elementos denominada *LISTA-ELEMENTOS*. Mediante la función *NINGUN1* se revisa si ninguno de estos elementos forman parte de la palabra denominada *REFERENCIA*, la cual es una variable global. Si este es el caso el valor de la función es *T*, o se genera el valor *NIL* en caso contrario.

El esquema de esta función es el siguiente:



PROGRAMACION DE JUNTA

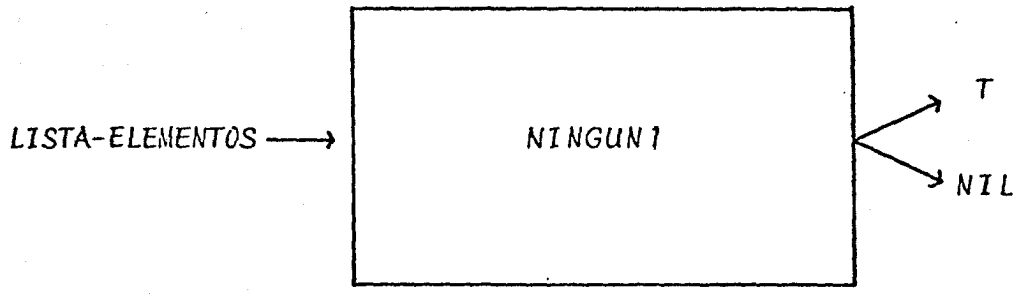
```
(LAMBDA (LISTA NUMERO-N)
  (COND ((EQ NUMERO-N 0)
         NIL)
        (T (CONS (CAR LISTA) (JUNTA (CDR LISTA) (SUB NUMERO-N 1))))))
```

(NINGUN):

Esta función tiene como argumento una lista de ele-- mentos denominada *LISTA-ELEMENTOS*. Mediante la función *NIN GUN1* se revisa si ninguno de estos elementos forman parte - de la palabra denominada *REFERENCIA*, la cual es una varia-- ble global. Si este es el caso el valor de la función es *T*, o se genera el valor *NIL* en caso contrario.

El esquema de esta función es el siguiente:

NINGUN



PROGRAMACION DE NINGUN

```
(LAMBDA (LISTA-ELEMENTOS)
  (NINGUN1 LISTA-ELEMENTOS (QUITAP REFERENCIA)))
```

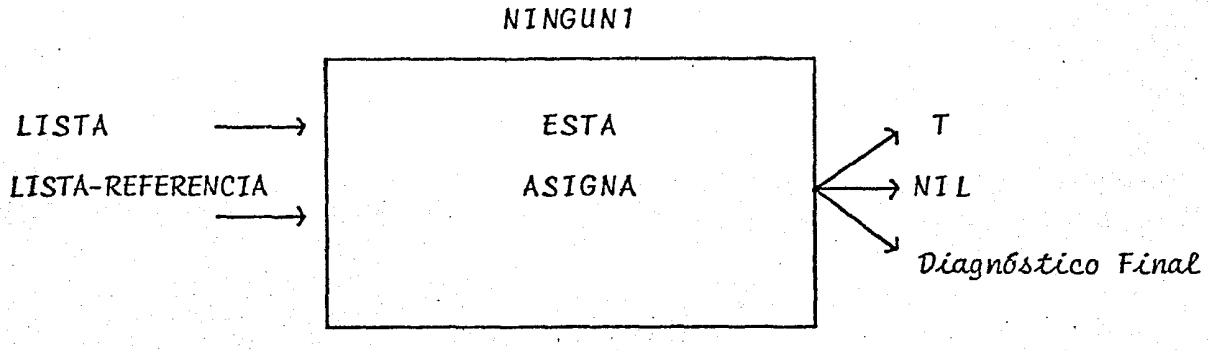
(NINGUN1):

Esta función tiene como argumentos a dos listas denominadas *LISTA* y *LISTA-REFERENCIA*. Mediante la función *ESTA* se verifica si ninguno de los elementos de la primera lista forman parte de la segunda lista. Si este es el caso el valor de la función es *T* o se genera el valor *NIL* en caso contrario.

Si alguno de los elementos de la primera lista constituyen a toda la segunda *LISTA-REFERENCIA*, el valor que la función toma es esta segunda lista, la cual mediante la fun

ción *ASIGNA* se genera como el Diagnóstico final del Experto, terminando con ello su trabajo.

El esquema de esta función es el siguiente:



PROGRAMACION DE NINGUN1

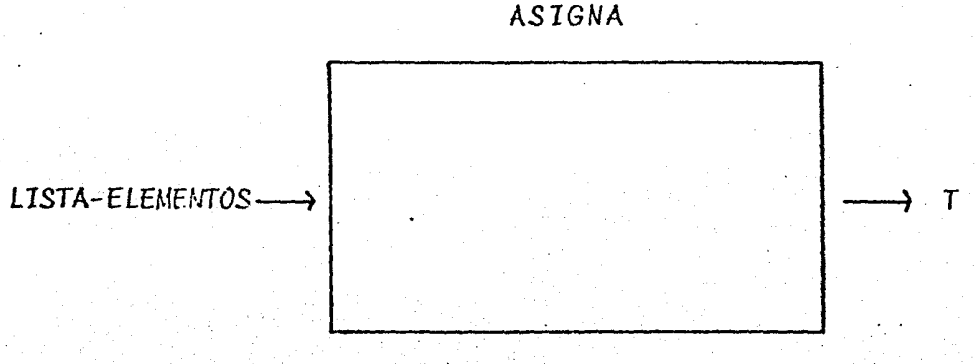
```
(LAMBDA (LISTA LISTA-REFERENCIA)
  (COND ((EQ LISTA)
         T)
        ((ESTA (CAR LISTA) LISTA-REFERENCIA)
         (COND ((EQ (LEN (UNPACK (CAR LISTA))) (LEN LISTA-REFERENCIA))
                (ASIGNA (CAR LISTA)))
               (T NIL)))
        (T (NINGUN1 (CDR LISTA) LISTA-REFERENCIA))))
```

(ASIGNA):

Esta función acepta un solo argumento, el cual está-formado por una lista de elementos denominada *LISTA-ELEMEN-*

TOS. El valor que genera esta función es siempre el valor *T* el cual es asignado a una variable global denominada *N1*.

El esquema de esta función es el siguiente:



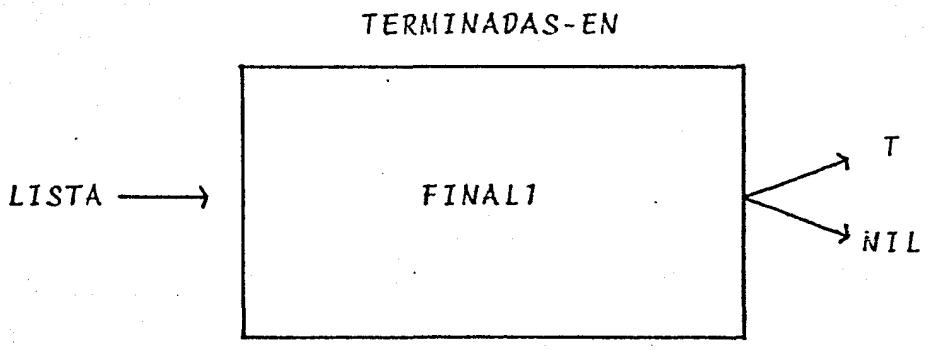
PROGRAMACION DE ASIGNA

```
(LAMBDA (LISTA-ELEMENTOS)
  (AND (PRINT (QUOTE (LA PALABRA SE ESCRIBE ASI :)))
       (PRINT LISTA-ELEMENTOS)
       (SETQ N1 T)))
```

(TERMINADAS-EN):

Esta función verifica mediante la función *FINAL1* si la palabra de referencia, denominada *REFERENCIA*, que es una variable global, termina con alguno de los elementos de la lista que se tiene como argumento, en cuyo caso se genera el valor *T* o *NIL* en caso contrario.

El esquema de esta función es el siguiente:



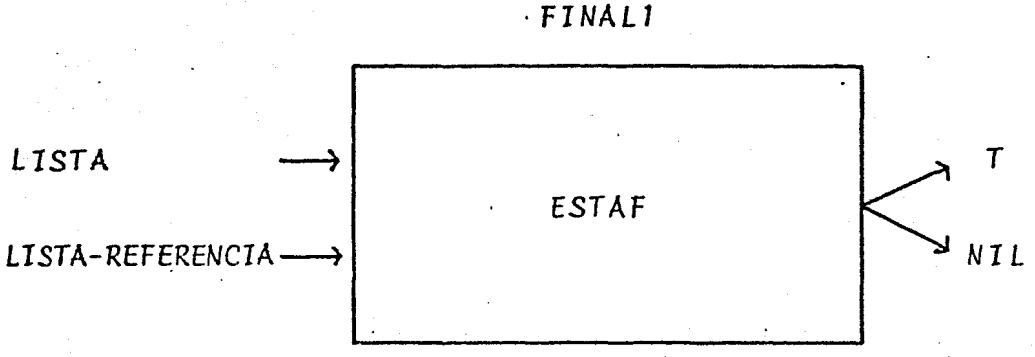
PROGRAMACION DE TERMINADAS-EN

```
<LAMBDA <LISTA>
  <FINAL1 LISTA <QUITAP REFERENCIA>>>
```

(FINAL1):

Esta función acepta como argumentos a dos listas denominadas *LISTA* y *LISTA-REFERENCIA*. Mediante la función -- *ESTAF* se verifica si la terminación de la *LISTA-REFERENCIA* forma parte de los elementos de la lista que se dió como -- primer argumento, en cuyo caso se genera el valor *T*, o el valor *NIL* en caso contrario.

El esquema de esta función es el siguiente:



PROGRAMACION DE FINAL1

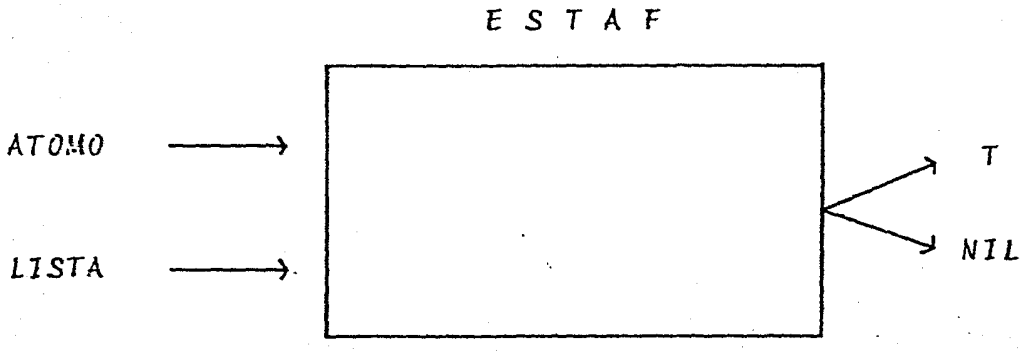
```
<LAMBDA <LISTA LISTA-REFERENCIA>
  <COND <<EQ LISTA>
         NIL>
        <<ESTAF <CAR LISTA> LISTA-REFERENCIA>
         T>
        <T <FINAL1 <CDR LISTA> LISTA-REFERENCIA>>>>
```

(ESTAF):

Esta función acepta dos argumentos, el primero es un átomo y el segundo una lista.

Esta función pregunta si los últimos componentes de la lista que se dió como argumento son iguales al valor del átomo, en cuyo caso se genera el valor *T*, o el valor *NIL* en caso contrario.

El esquema de esta función es el siguiente:



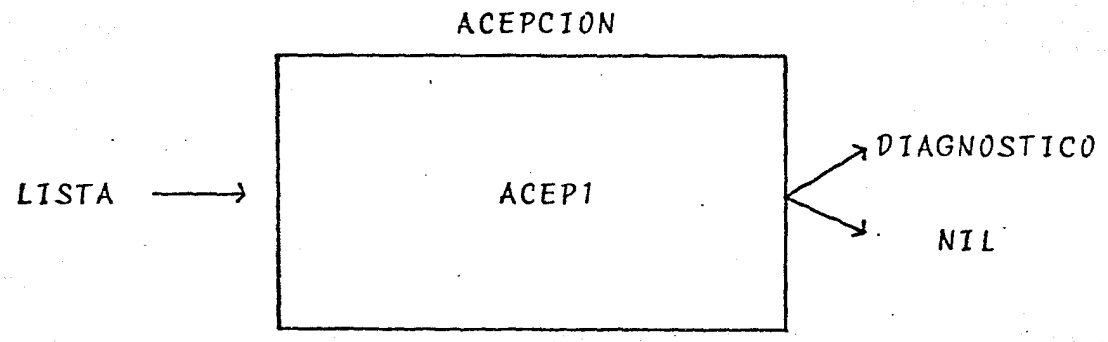
PROGRAMACION DE ESTAF

```
<LAMBDA <ATOMO LISTA>
  <COND <<> <LEN <UNPACK ATOMO>> <LEN LISTA>>
         NIL>
        <T <EQUAL ATOMO <PACK <@ LISTA <SUB <LEN LISTA> <LEN <UNPACK ATOMO>>
>>>>>>
```

(ACEPCION):

Esta función acepta como argumento una lista de elementos. Mediante la función *ACEP1* se checa si la palabra de referencia, que es una variable global denominada *REFERENCIA*, es igual a alguno de estos elementos en cuyo caso se genera un diagnóstico, o el valor *NIL* en caso contrario.

El esquema de esta función es el siguiente:



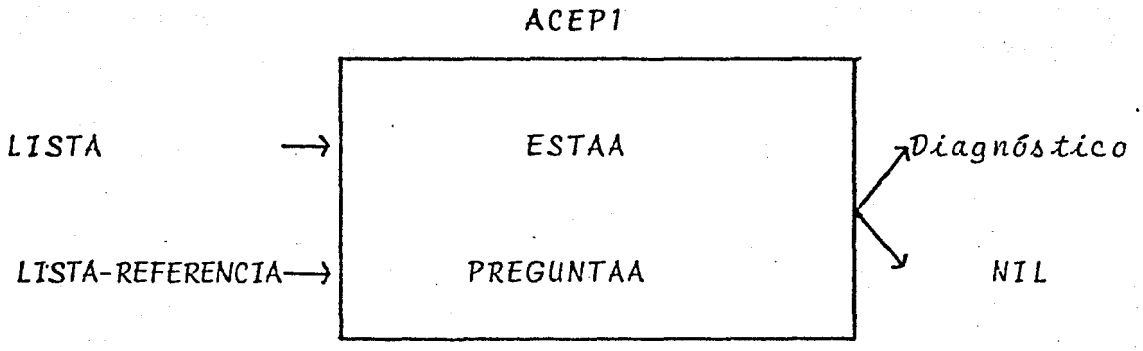
PROGRAMACION DE ACEPCION

```
(LAMBDA (LISTA)
  (ACEP1 LISTA (QUITAP REFERENCIA)))
```

(ACEP1):

Esta función acepta como argumentos a dos listas denominadas *LISTA* y *LISTA-REFERENCIA*. Mediante la función ESTAA se verifica si la lista de referencia se encuentra en alguno de los elementos de la primera lista, en cuyo caso mediante la función *PREGUNTA* se emite un diagnóstico, o se genera el valor *NIL* en caso contrario.

El esquema de esta función es el siguiente:



PROGRAMACION DE ACEP1

```
<LAMBDA <LISTA LISTA-REFERENCIA>
  <COND <<EQ LISTA>
         NIL>
        <<ESTAA <CAR LISTA> LISTA-REFERENCIA>
         <PREGUNTAA <CAR LISTA>>>
        <T <ACEP1 <CDR LISTA> LISTA-REFERENCIA>>>>
```

(ESTAA):

Esta función acepta como argumentos dos listas denominadas *LISTA* y *LISTA-REFERENCIA*. Verifica si la lista dereferencia está formada por alguno de los elementos de la primera lista, asignándosele a una variable global *R* el valor *1* si se encuentra en el primer elemento, el valor *2* si-

se encuentra en el segundo elemento o el valor *NIL* en caso de que no se encuentre en ninguno de ellos. El esquema de esta función es el siguiente:

ESTAA

PROGRAMACION DE ESTAA

```
<LAMBDA <LISTA LISTA-REFERENCIA>
  <COND <<EQUAL <PACK LISTA-REFERENCIA> <CAR <CAR LISTA>>>
         <SETQ R 1>>
        <<EQUAL <PACK LISTA-REFERENCIA> <CAR <CDR LISTA>>>
         <SETQ R 2>>
        <T NIL>>>
```

(PREGUNTAA):

Esta función tiene un sólo argumento que está formado por una lista que presenta la siguiente estructura:

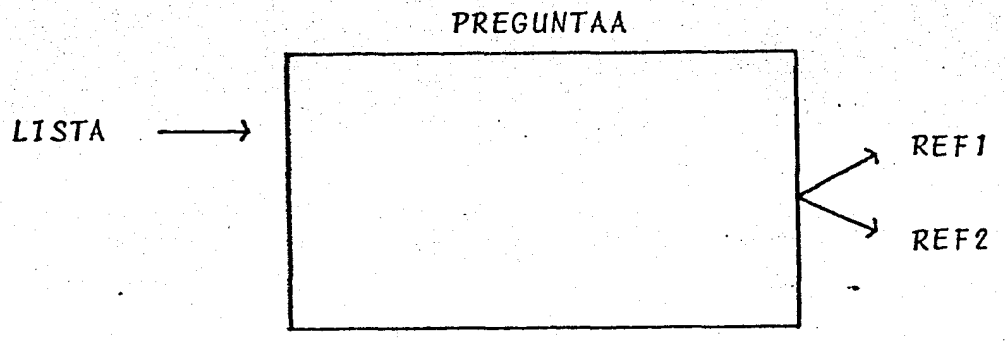
((REF1 SIG1) REF2 SIG2)

donde *REF1* y *REF2* son palabras que sólo difieren en la es-- critura de alguna de sus letras dependiendo del significado *SIG1* y *SIG2* de cada una de ellas.

Si el valor de la variable global *R*, el cual fue generado por la función *ESTAA*, es igual a *1* se pregunta al -- usuario si el significado *SIG1* de la palabra es el correcto, - si la respuesta de éste es *T*, entonces el diagnóstico final es la palabra escrita como se indica en *REF1*, o es la palabra escrita como se indica en *REF2* si la respuesta fue *NIL*.

De igual manera si el valor de *R* es igual *2*, se pregunta el significado *SIG2* de la palabra y si la respuesta - del usuario es *T* el diagnóstico será la palabra guardada en *REF2* o la palabra guardada en *REF1* en caso contrario.

El esquema de esta función es el siguiente:



PROGRAMACION DE PREGUNTAA

```
(LAMBDA (LISTA)
  (COND ((EQ R 1)
         (COND ((AND (PRIN1 (QUOTE (ES ESTE HECHO CIERTO ?)))
                     (PRIN1 (CDR (CAR LISTA)))
                     (READ))
                (PRINT (CAR (CAR LISTA))))
               (T (PRINT (CAR (CDR LISTA))))))
        (T (COND ((AND (PRIN1 (QUOTE (ES ESTE HECHO CIERTO ?)))
                       (PRIN1 (CDR (CDR LISTA)))
                       (READ))
                  (PRINT (CAR (CDR LISTA))))
                 (T (PRINT (CAR (CAR LISTA))))))))
```

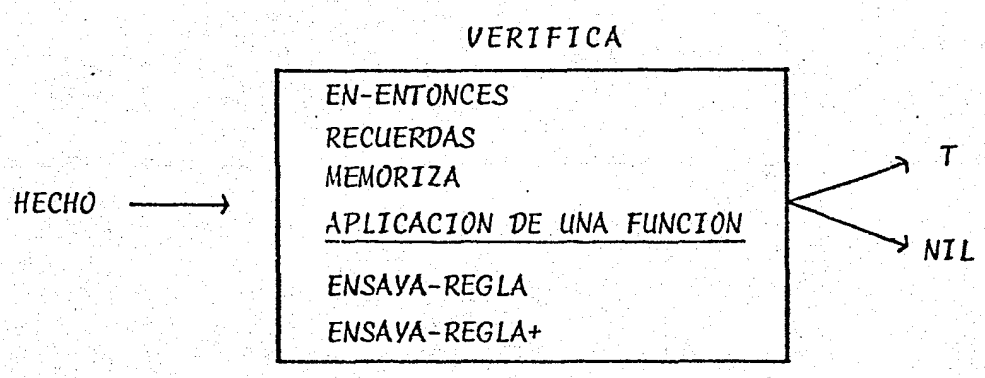
La lista de acepciones que usa este experto es la - siguiente:

```
(   ( (ACERBO  ASPERO)  ACERVO  CONJUNTO)
    ( (BACA  APUESTA) VACA ANIMAL )
    ( (BACILO  BACTERIA) VACILO  TITUBEO)
    ( (BARON  TITULO) VARON  MASCULINO)
    ((BIENES  PROPIEDAD)  VIENES  VERBO)
    ((BOTAR  ARROJAR)  VOTAR  ELECCIONES)
    ((TUBO  CILINDRO) TUVO  VERBO)   )
```

MODIFICACIONES HECHAS AL EXPERTO ORIGINAL

La única modificación que se le hizo al Experto Artificial mostrado en el capítulo 3 fue sobre la función *VERIFICA*, donde se reemplazó la instrucción que pregunta al usuario sobre alguna premisa no preguntada anteriormente, por la instrucción que aplica una función sobre sus argumentos. Tanto la función usada como sus argumentos están contenidos en las premisas de las reglas de producción que forman parte del conocimiento de este *EXPERTO ORTOGRAFISTA*.

El esquema de la función *VERIFICA* modificado es el siguiente; la modificación se encuentra subrayada.



La *programación* de esta función modificada es la siguiente:

```
(LAMBDA (HECHO)
  (PROG (RELEVANTE1 RELEVANTE2)
        (COND ((RECUERDAS HECHO)
               (RETURN T)))
        (SETQ RELEVANTE1 (EN-ENTONCES HECHO))
        (SETQ RELEVANTE2 RELEVANTE1)
        (COND ((EQ RELEVANTE1)
               (COND ((MEMBL HECHO PREGUNTADOS)
                      (RETURN NIL))
                     ((SETQ FUNCION (EVAL (CAR HECHO)))
                      (COND ((APPLY* FUNCION (CAR (CDR HECHO)))
                             (MEMORIZA HECHO) (RETURN T))
                            (T (RETURN NIL))))
                     ((AND (PRIN1 (QUOTE (ES ESTE HECHO CIERTO?)))
                           (PRIN1 HECHO)
                           (READ))
                      (MEMORIZA HECHO) (RETURN T))
                     (T (SETQ PREGUNTADOS (CONS HECHO PREGUNTADOS)) (RETURN NIL
)))))
```

FUNCIONAMIENTO Y EJEMPLOS DEL EXPERTO ORTOGRAFISTA

El Experto empieza a funcionar por medio de la fun-- ción denominada *BUSCA1*, la cual se activa de la siguiente- manera:

(BUSCA1)

Lo primero que realiza esta función es bajar del -- disco a la memoria de la máquina las funciones que inte- - gran al Sistema Experto, mediante la función *CARGA-ARCHI-- VOS*. En seguida se pregunta por la palabra problema y és- ta se guarda en una lista denominada *REFERENCIA*, que fun-- ciona como variable global del sistema, cuyos elementos --

son las letras que forman a la palabra.

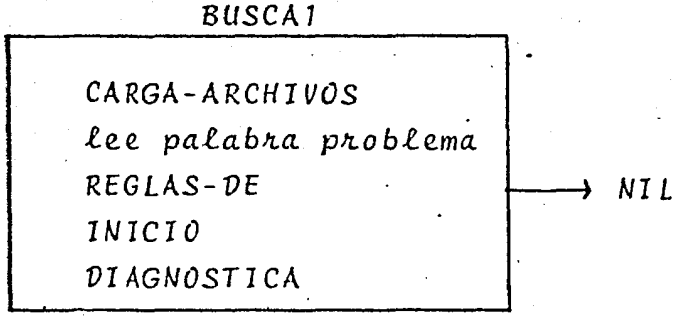
El Experto es consultado en varias ocasiones para revisar la sintáxis de cada una de las letras de la palabra, aplicando para ello las reglas ortográficas correspondientes, las cuales se obtienen con la función *REGLASD*, hasta agotar la palabra.

De este modo se reemplaza la letra que cumple con éstas reglas, por el diagnóstico que es asignado por medio de la función *INICIO*.

En caso de que no se cuente con reglas ortográficas para determinada letra se deja ésta en su forma original y se pasa a revisar la siguiente letra.

La función *BUSCA1* termina emitiendo el valor *NIL* cuando ya se han revisado todas las letras de la palabra original.

El esquema de la función *BUSCA1* es el siguiente:



A continuación se muestra la *programación* de la función *BUSCA1* y de las funciones que la integran: *CARGA-ARCHIVO*, *REGLASD* e *INICIO*.

PROGRAMACION DE BUSCA1

```
(LAMBDA NIL
  (PROG NIL
        (CARGA-ARCHIVOS)
        (PRINT (QUOTE (DAME LA PALABRA-PROBLEMA)))
        (SETQ PALABRA (UNPACK (READ)))
        (SETQ REFERENCIA PALABRA)
        (SETQ N1 NIL)
        (SETQ LREF (LEN REFERENCIA))
        ETIQUETA
        (PRINT REFERENCIA)
        (COND ((EQ PALABRA)
               (RETURN NIL))
              ((REGLASD (QUOTE (B U)))
               (INICIO (QUOTE %REGLAS/B/U) (QUOTE REGLAS/B/U) ACEP/B/U (QUOTE ((
B)
                (U)))))
              (T (GO CONTINUA)))
        (COND ((ACEPCION LISTA)
               (RETURN NIL)))
        (SETQ HECHOS NIL)
        (COND ((DIAGNOSTICA)
               (COND ((EQ N1)
                      (REEMPLAZA))
                     (T (RETURN NIL)))))
        CONTINUA
        (SETQ PALABRA (CDR PALABRA))
        (GO ETIQUETA)))
```

PROGRAMACION DE CARGA-ARCHIVOS

```
(LAMBDA NIL
  (PROG NIL
        (COND ((EQ (CAR %ABDUCCION1))
               (LOAD (QUOTE ABDUCCION1))))
        (COND ((EQ (CAR %ORTOGRAFIA))
               (LOAD (QUOTE ORTOGRAFIA))))
        (RETURN NIL)))
```

PROGRAMACION DE REGLAS-DE

```
(LAMBDA (X)
  (COND ((EQ (CAR X))
         NIL)
        ((EQ (CAR PALABRA) (CAR X))
         T)
        (T (REGLASD (CDR X)))))
```

PROGRAMACION DE INICIO

```
(LAMBDA (W X Y Z)
  (PROG NIL
        (COND ((EQ (CAR (EVAL W)))
               (LOAD X)))
        (SETQ DIAGNOSTICOS Z)
        (SETQ LISTA Y)
        (RETURN NIL)))
```

A continuación se muestran algunos ejemplos del funcionamiento de este *EXPERTO ORTOGRAFISTA*. Las instrucciones que están precedidas del signo de pesos representan las intervenciones del usuario.

Ejemplo (1)

```
$(BUSCA1)

(DAME LA PALABRA-PROBLEMA)

$TUBO
(T U B O)
(T U B O)
(T U B O)
(ES ESTE HECHO CIERTO ?)(CILINDRO)
$T
TUBO
NIL
```

Ejemplo (2)

```
$(BUSCA1

$)

(DAME LA PALABRA-PROBLEMA)

$CAMVIO
(C A M V I O)
(C A M V I O)
(C A M V I O)
(C A M V I O)
(LA-REGLA)BB(DEDUCE-QUE)(B)
EL-DIAGNOSTICO
(B)
ES-EL-CORRECTO
(C A M (B) I O)
(C A M (B) I O)
(C A M (B) I O)
NIL
```

Ejemplo (3)

```
$(BUSCA1)

(DAME LA PALABRA-PROBLEMA)

$TUBO
(T U B O)
(T U B O)
(T U B O)
(ES ESTE HECHO CIERTO ?)(CILINDRO)
$NIL
TUVO
NIL
```

Ejemplo (4)

```
$(BUSCA1)

(DAME LA PALABRA-PROBLEMA)

$ACERVO
(A C E R V O)
(A C E R V O)
(A C E R V O)
(A C E R V O)
(A C E R V O)
(ES ESTE HECHO CIERTO ?)(CONJUNTO)
$T
ACERVO
NIL
```

Ejemplo (5)

```
$(BUSCA1)

(DAME LA PALABRA-PROBLEMA)

$BONDADOSO
```

```
<B O N D A D O S O>
<LA-REGLA>B.1<DEDUCE-QUE><B2.1>
<LA-REGLA>B3<DEDUCE-QUE><B>
EL-DIAGNOSTICO
<B>
ES-EL-CORRECTO
<<B> O N D A D O S O>
<<B> O N D A D O S O>
<<B> O N D A D O S O>
<<B> O N D A D O S O>
<<B> O N D A D O S O>
<<B> O N D A D O S O>
<<B> O N D A D O S O>
<<B> O N D A D O S O>
<<B> O N D A D O S O>
NIL
```

Ejemplo (6)

```
$<BUSCA1>

<DAME LA PALABRA-PROBLEMA>

$AMBIGUO
<A M B I G U O>
<A M B I G U O>
<A M B I G U O>
<LA-REGLA>B0<DEDUCE-QUE><B>
EL-DIAGNOSTICO
<B>
ES-EL-CORRECTO
<A M <B> I G U O>
<A M <B> I G U O>
<A M <B> I G U O>
<A M <B> I G U O>
<A M <B> I G U O>
NIL
```

Ejemplo (7)

```
$(BUSCA1)

(DAME LA PALABRA-PROBLEMA)

$AMUAR
(A M U A R)
(A M U A R)
(A M U A R)
<LA-REGLA>B0<DEDUCE-QUE>(B)
EL-DIAGNOSTICO
(B)
ES-EL-CORRECTO
(A M (B) A R)
(A M (B) A R)
(A M (B) A R)
NIL
```

Ejemplo (8)

```
$(BUSCA1)

(DAME LA PALABRA-PROBLEMA)

$CAULE
(C A U L E)
(C A U L E)
(C A U L E)
<LA-REGLA>B1<DEDUCE-QUE>(B)
EL-DIAGNOSTICO
(B)
ES-EL-CORRECTO
(C A (B) L E)
(C A (B) L E)
(C A (B) L E)
NIL
```

Ejemplo (9)

```
$<BUSCA1>

<DAME LA PALABRA-PROBLEMA>

$VAGAVUNDO
<V A G A V U N D O>
<ES ESTE HECHO CIERTO?><VERBO *>
$NIL
<ES ESTE HECHO CIERTO?><ADJETIVO *>
$NIL
NINGUN-DIAGNOSTICO-SE-PUEDE-CONFIRMAR
<V A G A V U N D O>
<V A G A V U N D O>
<V A G A V U N D O>
<V A G A V U N D O>
<LA-REGLA>B2<DEDUCE-QUE><B2.1>
<LA-REGLA>B3<DEDUCE-QUE><B>
EL-DIAGNOSTICO
<B>
ES-EL-CORRECTO
<V A G A <B> U N D O>
<V A G A <B> U N D O>
<V A G A <B> U N D O>
<V A G A <B> U N D O>
<V A G A <B> U N D O>
NIL
```

4.2 EXPERTO ESTADISTICO

Con el uso de este Experto se obtiene información -- acerca de las pruebas estadísticas que se recomiendan usar- dependiendo de los objetivos del usuario y de las caracte-- rísticas de sus datos.

El uso que se le de a este Experto puede tener dife- rentes fines. Uno de ellos es el uso de este Sistema por - personas dedicadas a las Ciencias Sociales, las cuales por- regla general cuentan con una gran cantidad de información- y desean saber que pruebas estadísticas aplicar dependiendo del problema en particular que se tenga.

Otro uso puede ser con fines didácticos considerando al Experto como un curso inicial para aprender en forma pro gramada algunas características que deben de tener los da-- tos para que sea posible la aplicación de cierta prueba es- tadística.

La forma en la que se "llama" al Experto Estadístico para que empiece a trabajar es haciendo uso de la función

(DAME-PRUEBA)

y en seguida el experto comienza a hacer una serie de pre-- guntas sobre las características de los datos y sobre lo --

que se quiere hacer con ellos.

A cada pregunta del Experto el usuario contesta "T"- si la respuesta es afirmativa o *"NIL"* si la respuesta es negativa.

Las preguntas que hace el Experto siguen la estrategia del razonamiento regresivo, es decir, se trata de validar alguna de las diferentes pruebas estadísticas preguntando al usuario si son verdaderas las características asociadas a éstas, por el orden en que se encuentran las pruebas- dentro de la lista de Diagnósticos primero se tratan de probar las pruebas no-paramétricas y posteriormente las paramétricas.

Como tarea final el Sistema Experto emite las pruebas estadísticas que se sugieren usar o en caso contrario - un mensaje que indica que no existe ninguna prueba estadística que pueda ser aplicada con las características que presentan los datos.

A continuación se muestra un ejemplo del funcionamiento de este Experto.

El paso inicial consiste en llamar al Sistema Experto con la ayuda de la función *(DAME-PRUEBA)*.

La primera hipótesis que tratará de validar será el-uso de la prueba Binomial, por lo que las siguientes preg--guntas denotadas por P.E. son las correspondientes a esta - prueba.

Las respuestas del usuario se denotan con R.U. y son todas afirmativas.

P.E. Se quiere usar una prueba Estadística?

R.U. T

P.E. Se quiere usar una prueba no-Paramétrica?

R.U. T

P.E. Los datos provienen de una muestra?

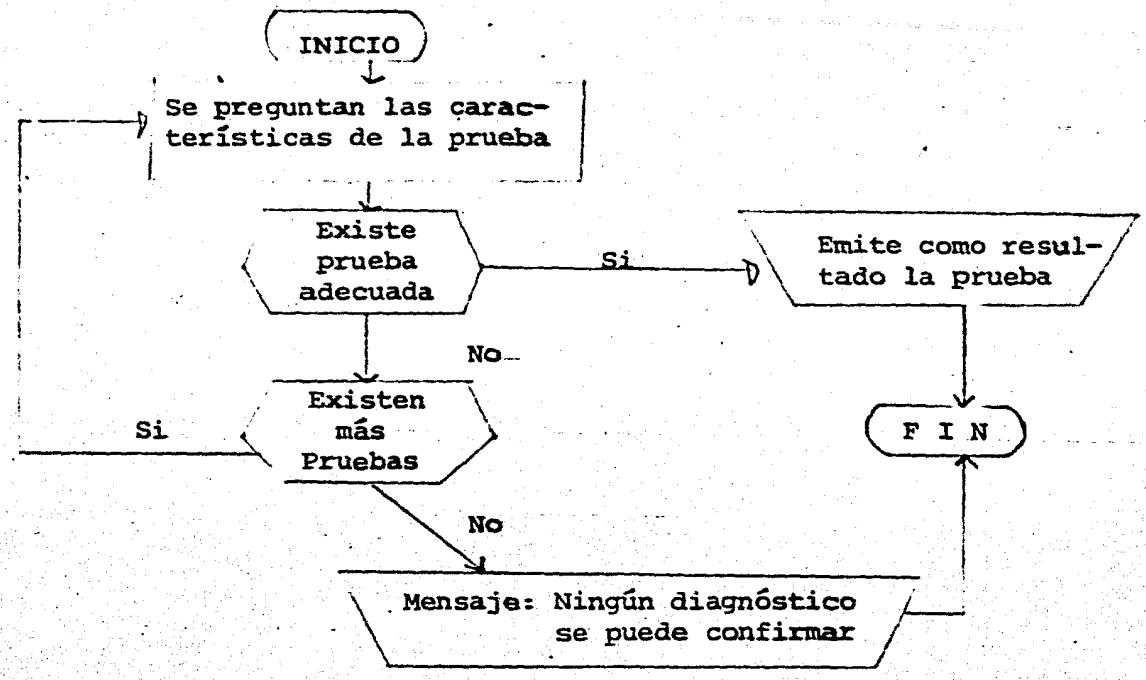
R.U. T

P.R. El nivel de medición es nominal?

R.U. T

DIAGNOSTICO FINAL.- PRUEBA BINOMIAL

La representación del funcionamiento del Experto Ex-tadístico en un diagrama es la siguiente:



A continuación se muestran las reglas de producción que usa este Experto Estadístico, así como las modificaciones hechas al Experto original y por último algunos ejemplos que muestran su competencia.

REGLAS DE PRODUCCION

El conocimiento de este Experto se basa en las características que presenta cada prueba estadística para poder ser utilizadas, las cuales se encuentran en cualquier libro de texto de estadística. Este conocimiento se encuentra representado en forma de reglas de producción, cuyas premisas presentan las características correspondientes a cada prueba y como conclusiones finales se obtiene una clave que se encuentra asociada a una prueba estadística.

Las reglas que usa este Experto Estadístico son las-
siguientes:

```
(R N1 (SI (C0) (SE QUIERE USAR UNA PRUEBA ESTADISTICA NO PARAMETRICA ?)) (ENTON
ES (C1)))

:A
(R N0 (SI (SE QUIERE USAR UNA PRUEBA ESTADISTICA?)) (ENTONCES (C0)))

:A
(R N2 (SI (C1) (LOS DATOS PROVIENEN DE UNA MUESTRA?)) (ENTONCES (C2)))

:A
(R N3 (SI (C1) (LOS DATOS PROVIENEN DE DOS MUESTRAS)) (ENTONCES (C3)))

:A
(R N4 (SI (C1) (LOS DATOS PROVIENEN DE K MUESTRAS ?)) (ENTONCES (C4)))

:A
(R N5 (SI (EL NIVEL DE MEDICION ES NOMINAL)) (ENTONCES (C5)))

:A
(R N6 (SI (EL NIVEL DE MEDICION ES ORDINAL)) (ENTONCES (C6)))

:A
(R N7 (SI (QUOTE (EL NIVEL DE MEDICION ES INTERVALAR ?))) (ENTONCES (C7)))

:A
(R N8 (SI (C2) (C5)) (ENTONCES (D1)))

:A
(R N9 (SI (C2) (C6)) (ENTONCES (D2)))

:A
(R N10 (SI (LAS MUESTRAS ESTAN RELACIONADAS ?)) (ENTONCES (C8)))

:A
(R N11 (SI (LAS MUESTRAS SON INDEPENDIENTES ?)) (ENTONCES (C9)))

:A
(R N12 (SI (C3) (C8) (C5)) (ENTONCES (D3)))

:A
(R N13 (SI (C3) (C8) (C6)) (ENTONCES (D4)))

:A
(R N14 (SI (C3) (C8) (C7)) (ENTONCES (D5)))
```

```
:A
(R N15 (SI (C3) (C9) (C5)) (ENTONCES (D6)))

:A
(R N16 (SI (C3) (C9) (C6)) (ENTONCES (D7)))

:A
(R N17 (SI (C3) (C9) (C7)) (ENTONCES (D8)))

:A
(R N18 (SI (C4) (C8) (C5)) (ENTONCES (D9)))

:A
(R N19 (SI (C4) (C8) (C6)) (ENTONCES (D10)))

:A
(R N20 (SI (C4) (C9) (C5)) (ENTONCES (D11)))

:A
(R N21 (SI (C4) (C9) (C6)) (ENTONCES (D12)))

:A
(R N22 (SI (C0) (SE DESEA UTILIZAR UNA PRUEBA ESTADISTICA PARAMETRICA ?)) (ENTON
CES (C10)))

:A
(R N23 (SI (C10) (LOS DATOS PROVIENEN DE UNA MUESTRA ?)) (ENTONCES (C12)))

:A
(R N24 (SI (C10) (LOS DATOS PROVIENEN DE DOS MUESTRAS ?)) (ENTONCES (C13)))

:A
(R N25 (SI (C10) (LOS DATOS PROVIENEN DE K MUESTRAS?)) (ENTONCES (C14)))

:A
(R N26 (SI (C12) (SE QUIERE ESTIMAR LA MEDIA ?)) (ENTONCES (C11)))

:A
(R N27 (SI (C12) (SE QUIERE ESTIMAR LA VARIANZA?)) (ENTONCES (D13)))

:A
(R N28 (SI (C11) (SE CONOCE LA DESVIACION ?)) (ENTONCES (D14)))

:A
(R N29 (SI (C11) (SE DEZCONOCE LA DESVIACION ?)) (ENTONCES (D15)))

:A
(R N30 (SI (C13) (SE QUIEREN COMPARAR LAS MEDIAS?)) (ENTONCES (C15)))

:A
(R N31 (SI (C13) (SE QUIEREN COMPARAR LAS VARIANZAS?)) (ENTONCES (D16)))
```

```
:A
:R N32 (SI (C14)) (ENTONCES (D20))

:A
:R N33 (SI (C15) (SE TIENEN VARIANZAS CONOCIDAS IGUALES?)) (ENTONCES (D17))

:A
:R N34 (SI (C15) (VARIANZAS DESCONOCIDAS?)) (ENTONCES (D18))

:A
:R N35 (SI (C15) (SE TIENEN VARIANZAS CONOCIDAS Y DESIGUALES?)) (ENTONCES (D19))
)
```

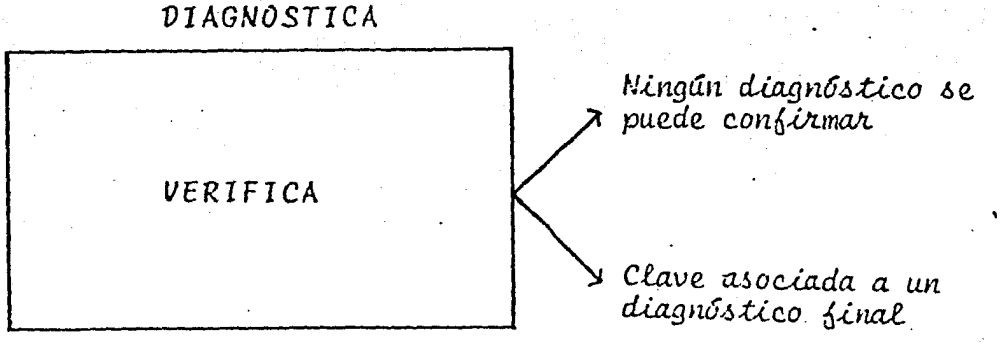
Las conclusiones finales de este Experto se guardan- al igual que en el Experto Ortografista en una lista denominada *DIAGNOSTICO*, la cual en este caso presenta la siguiente estructura:

((D1) (D2) (D3) (D4) (D5) (D6) (D7) (D8) (D9)
(D10) (D11) (D12) (D13) (D14) (D15) (D16) (D17)
(D18) (D19) (D20))

MODIFICACIONES HECHAS AL EXPERTO ORIGINAL

La única modificación que se le hizo al Experto original fue en la forma de trabajar con el resultado que genera la función *DIAGNOSTICA*, la cual en lugar de emitir un diagnóstico final emite una clave que posteriormente se le asocia a la prueba estadística correspondiente.

El esquema de la función *DIAGNOSTICA* es el siguiente:



PROGRAMACION DE DIAGNOSTICA

```
(LAMBDA NIL
  (PROG (POS PREGUNTADOS)
        (SETQ POS DIAGNOSTICOS)
        ETIQUETA
        (COND ((EQ POS)
               (PRINT (QUOTE NINGUN-DIAGNOSTICO-SE-PUEDE-CONFIRMAR)) (RETURN NI
))
              ((VERIFICA (CAR POS))
               (RETURN (CAR POS))))
        (SETQ POS (CDR POS))
        (GO ETIQUETA)))
```

FUNCIONAMIENTO Y EJEMPLOS DEL EXPERTO ESTADISTICO

Este experto empieza a funcionar al hacer uso de la-función denominada *DAME-PRUEBA*, la cual tiene como primera-tarea cargar las funciones que integran al Experto Artifi-- cial mediante una función denominada *CARGA-EXPERTO*. En seguida se cargan las reglas de producción mediante la fun- - ción denominada *CARGA-REGLAS* en donde se encuentra represen tado el conocimiento sobre las pruebas estadísticas paramé- tricas y no-paramétricas que son usadas por este Experto.

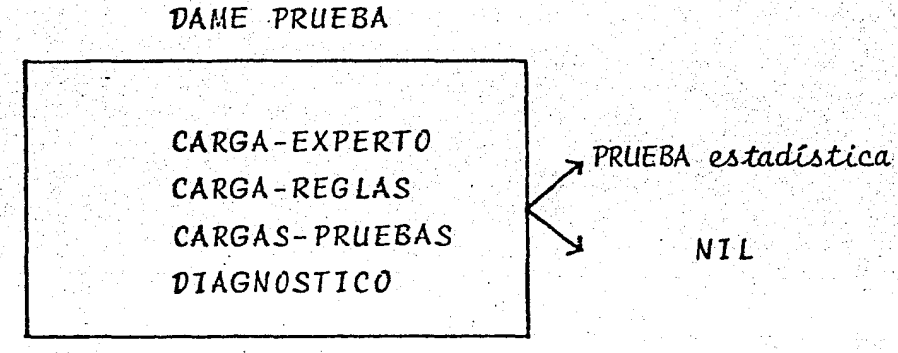
Estas reglas de producción tienen como conclusiones-finales claves que están asociadas a diferentes pruebas es- tadísticas las cuáles se cargan con la función *CARGA-PRUE--*

BAS.

El Experto original es consultado mediante la función *DIAGNOSTICA* y el valor que se genera como diagnóstico-final es asignado a una variable denominada *VALOR*, para posteriormente buscar la prueba estadística asociada a él dentro de una lista denominada *PRUEBAS*, la cuál contiene todas las posibles pruebas estadísticas que maneja este Experto.

Esta prueba o pruebas estadísticas se guardan en una variable denominada *SUGIERE*, la cual será el Diagnóstico final que se emite. En caso de no encontrarse una prueba estadística adecuada se generará como resultado el valor *NIL*.

El esquema de esta función es el siguiente:



A continuación se muestra la *programación* de las funciones: *DAME-PRUEBA*, *CARGA-EXPERTO*, *CARGA-REGLAS* y *CARGA-PRUEBAS*

(1) PROGRAMACION DE DAME-PRUEBA

```
(LAMBDA NIL
  (PROG NIL
        (CARGA-EXPERTO)
        (CARGA-REGLAS)
        (CARGA-PRUEBAS)
        (SETQ DIAGNOSTICOS (QUOTE ((D1)
         (D2) (D3) (D4) (D5) (D6) (D7) (D8) (D9) (D10) (D11) (D12) (D13) (D14) (
D15) (D16) (D17) (D18) (D19) (D20))))
        (SETQ FACTS NIL)
        (SETQ VALOR (CAR (DIAGNOSE)))
        (COND ((EQ VALOR)
               (RETURN NIL)))
        (SETQ SUGUIERE (CDR (ASSOC VALOR PRUEBAS)))
        (PRINT (QUOTE (LA PRUEBA QUE SE SIGUIERE APLICAR ES :)))
        (RETURN SUGUIERE)))
```

(2) PROGRAMACION DE CARGA-EXPERTO

```
(LAMBDA NIL
  (COND ((EQ (CAR %ABDUCCION))
         (LOAD (QUOTE ABDUCCION)))))
```

(3) *PROGRAMACION DE CARGA-REGLAS*

```
<LAMBDA NIL
  <COND <<EQ <CAR %REGLAS-ESTADISTA>>
         <LOAD <QUOTE REGLAS-ESTADISTA>>>>>
```

(4) *PROGRAMACION DE CARGA-PRUEBAS*

```
<LAMBDA NIL
  <COND <<EQ <CAR %PRUEBAS-NO-PARAMETRICAS>>
         <LOAD <QUOTE PRUEBAS-NO-PARAMETRICAS>>>>>
```

A continuación se muestran algunos ejemplos que ilustran el funcionamiento de este *EXPERTO ESTADISTICO*. Las instrucciones precedidas del signo de pesos representan las instrucciones del usuario.

Ejemplo (1)

```
$<DAME-PRUEBA>

<ES ESTE HECHO CIERTO?><SE QUIERE USAR UNA PRUEBA ESTADISTICA?>
$T
```

```
<LA-REGLA>N0<DEDUCE-QUE><C0>
<ES ESTE HECHO CIERTO?><SE QUIERE USAR UNA PRUEBA ESTADISTICA NO PARAMETRICA ?>
$T
<LA-REGLA>N1<DEDUCE-QUE><C1>
<ES ESTE HECHO CIERTO?><LOS DATOS PROVIENEN DE UNA MUESTRA?>
$NIL
<ES ESTE HECHO CIERTO?><LOS DATOS PROVIENEN DE DOS MUESTRAS>
$NIL
<ES ESTE HECHO CIERTO?><LOS DATOS PROVIENEN DE K MUESTRAS ?>
$T
<LA-REGLA>N4<DEDUCE-QUE><C4>
<ES ESTE HECHO CIERTO?><LAS MUESTRAS ESTAN RELACIONADAS ?>
$T
<LA-REGLA>N10<DEDUCE-QUE><C8>
<ES ESTE HECHO CIERTO?><EL NIVEL DE MEDICION ES NOMINAL>
$T
<LA-REGLA>N5<DEDUCE-QUE><C5>
<LA-REGLA>N18<DEDUCE-QUE><D9>
EL-DIAGNOSTICO<D9>ES-EL-CORRECTO
<LA PRUEBA QUE SE SIGUIERE APLICAR ES :>
<<PRUEBA DE CHOCHRAN@>>
```

Ejemplo (2)

```
$<DAME-PRUEBA>

<ES ESTE HECHO CIERTO?><SE QUIERE USAR UNA PRUEBA ESTADISTICA?>
$T
<LA-REGLA>N0<DEDUCE-QUE><C0>
<ES ESTE HECHO CIERTO?><SE QUIERE USAR UNA PRUEBA ESTADISTICA NO PARAMETRICA ?>
$T
<LA-REGLA>N1<DEDUCE-QUE><C1>
<ES ESTE HECHO CIERTO?><LOS DATOS PROVIENEN DE UNA MUESTRA?>
$T
<LA-REGLA>N2<DEDUCE-QUE><C2>
<ES ESTE HECHO CIERTO?><EL NIVEL DE MEDICION ES NOMINAL>
$T
<LA-REGLA>N5<DEDUCE-QUE><C5>
<LA-REGLA>N8<DEDUCE-QUE><D1>
EL-DIAGNOSTICO<D1>ES-EL-CORRECTO
<LA PRUEBA QUE SE SIGUIERE APLICAR ES :>
<<PRUEBA BINOMIAL> <PRUEBA CHI CUADRADA PARA UNA MUESTRA>>
```

Ejemplo (3)

```
$(DAME-PRUEBA)

(ES ESTE HECHO CIERTO?)(SE QUIERE USAR UNA PRUEBA ESTADISTICA?)
$T
(LA-REGLA)N0(DEDUCE-QUE)(C0)
(ES ESTE HECHO CIERTO?)(SE QUIERE USAR UNA PRUEBA ESTADISTICA NO PARAMETRICA ?)
$NIL
(ES ESTE HECHO CIERTO?)(SE DESEA UTILIZAR UNA PRUEBA ESTADISTICA PARAMETRICA ?)
$T
(LA-REGLA)N22(DEDUCE-QUE)(C10)
(ES ESTE HECHO CIERTO?)(LOS DATOS PROVIENEN DE UNA MUESTRA ?)
$T
(LA-REGLA)N23(DEDUCE-QUE)(C12)
(ES ESTE HECHO CIERTO?)(SE QUIERE ESTIMAR LA VARIANZA?)
$T
(LA-REGLA)N27(DEDUCE-QUE)(D13)
EL-DIAGNOSTICO(D13)ES-EL-CORRECTO
(LA PRUEBA QUE SE SIGUIERE APLICAR ES :)
((PRUEBA CHI CUADRADA))
```

Ejemplo (4)

```
$(DAME-PRUEBA)

(ES ESTE HECHO CIERTO?)(SE QUIERE USAR UNA PRUEBA ESTADISTICA?)
$T
(LA-REGLA)N0(DEDUCE-QUE)(C0)
(ES ESTE HECHO CIERTO?)(SE QUIERE USAR UNA PRUEBA ESTADISTICA NO PARAMETRICA ?)
$NIL
(ES ESTE HECHO CIERTO?)(SE DESEA UTILIZAR UNA PRUEBA ESTADISTICA PARAMETRICA ?)
$T
(LA-REGLA)N22(DEDUCE-QUE)(C10)
(ES ESTE HECHO CIERTO?)(LOS DATOS PROVIENEN DE UNA MUESTRA ?)
$T
(LA-REGLA)N23(DEDUCE-QUE)(C12)
(ES ESTE HECHO CIERTO?)(SE QUIERE ESTIMAR LA VARIANZA?)
$NIL
(ES ESTE HECHO CIERTO?)(SE QUIERE ESTIMAR LA MEDIA ?)
$T
(LA-REGLA)N26(DEDUCE-QUE)(C11)
(ES ESTE HECHO CIERTO?)(SE CONOCE LA DESVIACION ?)
$T
(LA-REGLA)N28(DEDUCE-QUE)(D14)
EL-DIAGNOSTICO(D14)ES-EL-CORRECTO
(LA PRUEBA QUE SE SIGUIERE APLICAR ES :)
((PRUEBA Z= (MEDIA-MU) / (SIGMA/ (RAIZ CUADRADA DE N))))
```

Ejemplo (5)

```
$(DAME-PRUEBA)

(ES ESTE HECHO CIERTO?)(SE QUIERE USAR UNA PRUEBA ESTADISTICA?)
$NIL
NINGUN-DIAGNOSTICO-SE-PUEDE-CONFIRMAR
NIL
```

5. DISCUSION Y CONCLUSIONES

Se utilizó un Sistema Experto básico al cual se le hicieron ligeras modificaciones para lograr la construcción de dos diferentes Expertos, uno de ellos un Experto "Ortografista y el otro un Experto "Estadístico".

Con la implementación de ellos se ilustra como se puede variar el uso y las aplicaciones de este Sistema Experto para de este modo abarcar diferentes áreas del conocimiento.

Los usos que se le pueden dar a ambos Expertos dentro del área educativa son dos. Uno de ellos tomando al Experto como un medio de difusión de conocimientos, ya que la revisión de las reglas de producción por los usuarios ilustra la dinámica del conocimiento el cual puede ser consultado y discutido por ellos.

Otro de sus usos puede ser tomando al Experto como un Sinodal, el cual genera el diagnóstico a ser comparado por el del usuario y en caso de que estos sean diferentes, remitir al usuario a revisar una vez más el conocimiento del Experto representado en reglas de producción.

Estas formas de usar al Sistema Experto serían de -- gran utilidad para los estudiosos de cierta área del conocimiento, permitiéndole al usuario lograr una mejor comprensión de la dinámica del conocimiento, así como proporcionarle un nuevo método de estudio parecido al programado.

Como se observa, el poder de cada uno de los Expertos está en relación directa con la calidad y cantidad de conocimiento que estos manejen.

El conocimiento en este trabajo se representa en forma de reglas de producción y dada esta estructura el conocimiento estará limitado potencialmente hasta el momento en que se le aumenten o modifiquen las reglas de producción -- que forman la base del conocimiento del Sistema Experto.

En el caso del Sistema Ortografista se encuentran incorporadas hasta el momento solo las reglas ortográficas sobre el uso de la "b" y "v". Sin embargo, para ampliar el conocimiento de este Experto, sólo se necesita incorporar en forma de reglas de producción, las reglas ortográficas correspondientes a las demás letras "problema" del idioma Expañol, sin ser necesaria ninguna modificación adicional sobre la estructura básica de este Experto "Ortografista".

De igual forma se puede ampliar el conocimiento del-

Experto "Estadístico", con sólo incluir otros diferentes -- análisis estadísticos en forma de reglas de producción en - la base de conocimientos de este Experto.

Una de las limitaciones que se pueden encontrar con- el uso de este Experto es que no cuenta hasta el momento -- con la capacidad de "aprender" directamente.

Donde por aprender se entenderá el incorporar una -- nueva regla de producción con la ayuda del usuario a la base del conocimiento del Experto, en el momento en que el -- Sistema esté funcionando.

Para incorporar un nuevo conocimiento, se reemplazaría el mensaje de "ningún diagnóstico se puede confirmar" - por uno donde se le preguntaría al usuario si en ese momento se desea agregar una nueva regla, donde su premisa estaría formada por los hechos conocidos hasta el momento y su- conclusión la formaría el diagnóstico que daría el usuario- en ese momento.

Si la respuesta es afirmativa con la ayuda de una -- nueva función que se crearía para tal caso, se le pediría - el diagnóstico al usuario y la nueva regla se anexaría a la base de conocimientos del Sistema Experto.

En cuanto al mecanismo que usa el Sistema Experto para emitir un diagnóstico, se observa que su forma de trabajar, la cual usa un razonamiento regresivo, tiene como finalidad tratar de validar una hipótesis reuniendo y analizando para ello todas las posibles reglas de producción que son necesarias para ello.

Este tipo de razonamiento puede ser considerado como contrario al usado por el Método Científico. Existe, sin embargo, la posibilidad de modificar el contenido de las reglas de producción de tal modo que las conclusiones o diagnósticos finales, lleven a rechazar un diagnóstico o hipótesis inicial, simulando con ello un razonamiento "científico".

Así en el caso del Experto "Ortografista un razonamiento de este tipo llevaría a suponer inicialmente que el usuario tiene errores ortográficos en la escritura de su palabra, por lo que se trataría de probar si la palabra se escribe en forma diferente.

En el caso del Experto "Estadístico" un razonamiento "científico" trataría de buscar aquellas características que invalidaran a la prueba estadística en turno.

Aunque en la actualidad los Sistemas Expertos no han

sido ampliamente utilizados, debido tal vez al miedo de ser reemplazados o superados por ellos, se espera que en un futuro próximo exista gran demanda de Sistemas Expertos en diferentes áreas del conocimiento.

El Sistema Experto aquí mostrado puede servir como una base sólida de la cual partir para construir nuevos Expertos en diferentes áreas del conocimiento, los cuales superen algunas de las deficiencias ya mencionadas de éste, las cuales puede observarse que son mínimas.

APENDICE

1) L I S P

1.1) Introducción

LISP es un lenguaje de programación que fue diseñado por John Mc. Carthy en el año de 1960 (28, 29, 30). Este lenguaje consta de tan solo siete funciones denominadas primitivas: *CAR, CDR, CONS, ATOM, EQ, COND* y *LAMBDA* y a partir de estas funciones puede generarse el resto del interprete *LISP*.

El formato y el uso de estas funciones se trata posteriormente, así como se dan a conocer algunas funciones adicionales construidas en base a éstas, que se consideran que son las más usuales.

El funcionamiento en general de *LISP* se asemeja al concepto matemático de función $f(x)$, el cual representa la aplicación de una función f sobre su argumento x.

Existen dos notaciones diferentes de *LISP* donde una de ellas, que es la *EVAL-QUOTE* preserva este tipo de notación:

$$< función > \ (\ < argumento > \ )$$

La otra notación llamada *EVAL*, que es la que se usa-

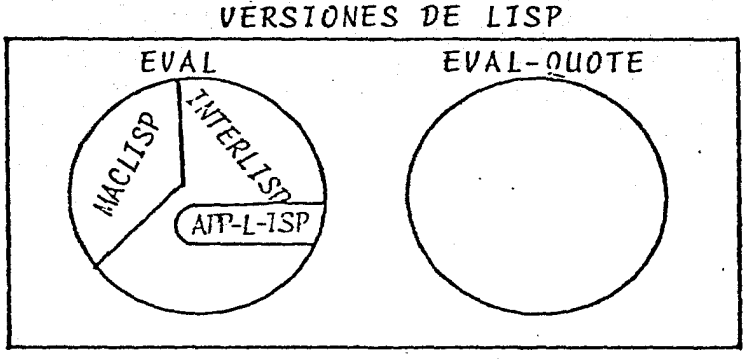
en este trabajo, incluye tanto la función como el argumento dentro de paréntesis, donde el primer elemento es siempre - la función que se va a ejecutar y el resto es el argumento. A este tipo de notación se le conoce como Polaca y es la siguiente:

(< *función* > < *argumento* >)

La versión *EVAL* de *LISP* evalúa tanto la función como su argumento en el momento de su ejecución, salvo en algu-- nos casos especiales.

Los principales dialectos de *LISP* que trabajan con - la notación *EVAL* son: el *MACLISP* y el *INTERLISP* los cuales - presentan diferencias mínimas en cuanto a la sintáxis y - - existencia de algunas de sus funciones.

El dialécto usado en este trabajo es el *APP-L-ISP* -- que es una implementación del *INTERLISP*. En el siguiente - diagrama se muestran las diferentes notaciones y algunos -- dialectos de *LISP*.



El uso de un lenguaje como el *LISP* presenta ciertas ventajas y una de ellas es que a partir de funciones primitivas, se pueden generar nuevas funciones, las cuales pasan a formar inmediatamente parte de las funciones disponibles para usar, ésto permite trabajar con problemas divididos en subproblemas, creando una función especial para cada uno de ellos para finalmente integrar a todas estas funciones mediante una función general. Un lenguaje como éste, que presente este tipo de programación está enfocado a lograr una programación estructurada, donde la revisión y aún la generalización de un problema se realiza en una forma sencilla.

Por otra parte en *LISP* se pueden manejar funciones recursivas, es decir, funciones que están definidas en términos de ellas mismas; ésto representa una gran ventaja, en relación con otros lenguajes de programación, ya que la naturaleza de muchos problemas matemáticos y quizá aún de la vida diaria son recursivos.

Por último, otra de las ventajas que se tienen en el uso del lenguaje *LISP* es que cuenta con una gran capacidad para la manipulación de cadenas simbólicas.

A continuación se muestra la sintáxis de las funciones más usuales en *LISP*, las cuales irán acompañadas de algunos ejemplos que ilustran su uso y manejo.

El nombre de las funciones se expresará con mayúsculas y a manera de notación los argumentos de ellas se representarán dentro de los siguientes símbolos "< >" y se escribirán con letras minúsculas.

1.2 *Atomos y Listas*

En *LISP* se manejan expresiones simbólicas que están formadas por elementos denominados átomos, o por conjunto de elementos denominados listas.

A continuación se explicarán cada uno de estos conceptos.

ATOMOS:

Un átomo puede considerarse como la expresión más pequeña e indivisible que puede ser usada. Un átomo no puede descomponerse en partes, ya que él en sí representa la unidad.

Cada átomo puede ser un número o un carácter alfanumérico. En *LISP* se cuenta con una función creada para probar si una expresión es un átomo o no, la cual genera el valor "*T*" si la expresión es un átomo o "*NIL*" en caso contrario. A esta función se le denomina *ATOM*.

Ejemplos:

```
$(ATOM A)
T
$(ATOM ABCD)
T
$(ATOM 1)
T
$(ATOM 139)
T
```

```
$(ATOM 'A)
T
$(ATOM (A BC))
EVAL ERR
A
(A BC)
>(RESET)

$(ATOM '(A BC))
NIL
```

LISTAS:

Las listas están formadas por colecciones de elementos, los cuales se delimitan por medio de paréntesis. Los elementos de una lista pueden ser átomos, listas o combinaciones de éstas.

Una característica de *LISP* es que puede trabajar con listas "sin elementos" o listas vacías, las cuales se denotan por medio de paréntesis sin elementos intermedios "( )".

En *LISP* se cuenta con la función *LIST*, la cual admi-

te uno o más argumentos con los cuales se construye una lista. La forma general de esta función es:

(*LIST* < *argumentos* >)

Donde los argumentos pueden ser átomos o listas.

Ejemplos:

```
$<LIST '<A B> '<C D> F>
<<A B> <C D> NIL>

$<LIST A B C>
<<NIL> HOLA <HOLA>>
$<LIST 'A 'B 'C>
<A B C>
```

1.3 *Funciones que permiten realizar operaciones aritméticas*

En el lenguaje *LISP* se cuenta con una colección de funciones, las cuales permiten la realización de las operaciones fundamentales de suma, resta, multiplicación, división y de una función que permite obtener el residuo de la división. A continuación presentamos estas funciones, junto

con algunos ejemplos sobre su uso.

SUMA: (+)

La función "+" admite dos argumentos, y da como re-- sultado la suma de ellos, la sintáxis general de esta fun- - ción es:

(+ < *argumento-1* > < *argumento-2* >)

Ejemplos:

```
$(+ 2 4)
6

$(+ 34 -54)
-20

$(+ A B)
0

$(+ -34 -54)
-88
```

DIFERENCIA: (SUB)

La función usada para realizar la diferencia es: *SUB*, esta función admite dos argumentos y genera como resultado - la diferencia del primer argumento menos el segundo.

La sintáxis general de esta función es:

(*SUB* < *argumento-1* > < *argumento-2* >)

Ejemplos:

```
$(SUB 45 69)
-24
$(SUB -45 69)
-114
```

```
$(SUB 69 -45)
114
$(SUB -69 -45)
-24
```

PRODUCTO: (*)

La función "*" realiza el producto de dos argumentos. La sintáxis general de esta función es:

(* < *argumento-1* > < *argumento-2* >)

Ejemplos:

```
$(* 2 5)
10
$(* 2 -5)
-10

$(* -2 -5)
10

$(* -2 5)
-10
```

DIVISION: (/)

La función "/", realiza la división del primer argumento entre el segundo argumento. La sintáxis general es:

(/ <*argumento-1*> <*argumento-2*>)

Ejemplos:

```
$(/ 10 5)
2
$(/ 10 -5)
-2
$(/ -10 -5)
2
$(/ -10 5)
-2
```

RESIDUO: (MOD)

Esta función admite dos argumentos y genera como resultado el residuo generado por la división del primer argumento entre el segundo argumento. La sintáxis general de esta función es:

(MOD <*argumento-1*> <*argumento-2*>)

Ejemplos:

```
$(MOD 20 5)
0
```

```
$(MOD 21 5)
1

$(MOD 24 5)
4
$(MOD -22 5)
2
```

Las operaciones anteriores se pueden combinar para -- realizar operaciones tan complicadas como se requiera.

Ejemplos:

```
$(+ 2 (/ 10 2))
7
$(SUB 100 (* 2 (/ 6 2)))
94
$(+ 20 (SUB 100 (* 2 (/ 6 2))))
114
```

1.4 *Asignaciones:* (SETQ) *y* (SET)

Mediante las asignaciones podemos darle valor a una- variable. En *LISP* se cuenta con dos tipos de asignaciones:

SETQ y SET, en seguida explicaremos cada una de ellas:

(SETQ):

La forma general de esta función es:

(SETQ < variable > < argumento >)

donde la variable es un átomo y a ella se le asigna el va-- lor del argumento, el cual puede ser un átomo o una lista.

Ejemplos:

```
$(SETQ X 2)
2
$(SETQ Y 3)
3
```

```
$(SETQ X '(A B C))
(A B C)
$(SETQ Y '(D E))
(D E)
```

(SET):

Mediante la función *SET* se le asigna al valor del -- primer argumento, el valor del segundo argumento. La diferencia principal entre *SET* y *SETQ* radica en que *SET* evalúa el primer argumento y *SETQ* no lo hace, la sintáxis general de esta función es:

(SET < argumento-1 > < argumento-2 >)

Ejemplos:

```
$<SET 'A 'ATOMO>
ATOMO
$A
ATOMO

$<SET 'Z '<LISTA>>
<LISTA>
$Z
<LISTA>
```

1.5 *Funciones Predicativas de Comparación:* (EQ), (EQUAL) *y* (>)

En el lenguaje *LISP* se cuenta con dos funciones que prueban la igualdad entre dos elementos, éstas son: *EQ y EQUAL*. Se cuenta también con la función (>) que compara si un elemento es mayor que otro.

A continuación se explica el funcionamiento de cada una de estas funciones:

(EQ):

La función *EQ* admite dos argumentos, los cuales pueden ser átomos o apuntadores que estén apuntando a la misma lista o a los mismos átomos. Si los argumentos son iguales el resultado de la función es "T" y en caso contrario el va

lor de la función es *NIL*.

La sintáxis general de esta función es:

(*EQ* < *argumento-1* > < *argumento-2* >)

Ejemplos:

```
$(SETQ B 'HOLA)
HOLA
$(SETQ A 'HOLA)
HOLA

$(EQ A B)
T
```

```
$(SETQ D '(HOLA))
(HOLA)
$(SETQ C '(HOLA))
(HOLA)

$(EQ C D)
NIL
```

Cuando esta función se usa con un sólo argumento se-sobreentiende que el segundo argumento es el átomo *NIL*, es-decir, se pregunta si el primer argumento tiene el valor de *NIL*.

(EQUAL):

La función *EQUAL* admite dos argumentos, éstos pueden ser átomos, listas o apuntadores que están apuntando a las-mismas o a diferentes listas o átomos ya que en este caso - se revisa el contenido del apuntador. El valor de la fun-- ción es *T* si los argumentos son iguales, o *NIL* en caso contrario.

La sintáxis general de esta función es:

(EQUAL < *argumento-1* > < *argumento-2* >)

EQUAL al igual que la función *EQ* pueden tener un sólo argumento en cuyo caso se pregunta, si el argumento tiene el valor *NIL*, si éste es un átomo, o si es igual a la -- lista *(NIL)*, si el argumento es una lista.

Ejemplos:

```
$(EQUAL '(A B C) '(A B C))
T
$(EQUAL 2 2)
T
```

```
$(EQUAL 'A 'A)
T
```

(>)

Otra de las funciones para relacionar dos elementos- es la función >, la cual acepta dos argumentos y genera el- valor "*T*" si el primer argumento es mayor que el segundo, y "*NIL*" en caso contrario. La sintáxis general de esta fun-- ción es:

(> < *argumento-1* > < *argumento-2* >)

Ejemplos:

```
$(SETQ A 8)
8
$(SETQ B 6)
6
$(> A B)
T
$(> B A)
NIL
```

```
$(> 5 2)
T
$(> 2 5)
NIL
```

1.6 *Funciones Predicativas Lógicas: (AND) y (OR)*

Las funciones predicativas lógicas que se usan en -- *LISP* son *AND* y *OR*. Mediante este tipo de funciones se pueden probar una serie de condiciones a la vez. A continuación se indica la sintáxis de cada una de ellas:

(AND):

La función *AND* admite dos o más argumentos, y cada uno de ellos expresan una o varias condiciones. El valor de la función *AND* es *T* si todos sus argumentos son verdaderos, es decir, si todas las condiciones dentro de los argu-

mentos se cumplen, generando el valor *NIL* en caso contrario.

La sintáxis general de esta función es:

(*AND* < *argumento-1* > < *argumento-2* > ... < *argumento-n* >)

donde cada argumento representa una condición o un conjunto de condiciones.

Ejemplos:

```
$(AND (EQUAL 'A 'A) (EQUAL 1 1))
T
$(AND (EQUAL 'A 'B) (EQUAL 1 1))
NIL
$(AND (EQUAL 1 1) (EQUAL 2 2) (EQUAL 3 3))
T
```

(OR):

La función *OR* admite dos o más argumentos, y al igual que la función *AND*, cada uno de los argumentos representa una condición, el valor de la función *OR* es *T* o verdadero, si alguno de sus argumentos es verdadero, generándose el valor *NIL* en caso contrario.

La sintáxis general de la función *OR* es:

(*OR* < *argumento-1* > < *argumento-2* > .... < *argumento-n* >)

Ejemplos:

```
$<OR <EQUAL 'A 'A> <EQUAL 1 1>>
T
$<OR <EQUAL 'A 'B> <EQUAL 1 1>>
T
$<OR <EQUAL 2 1> <EQUAL 3 2>>
NIL
```

Las funciones *AND* y *OR* pueden combinarse entre sí, para formar listas de condiciones tan complejas como se requieran, para ello solo se debe tener la precaución de delimitar por medio de paréntesis las listas de una manera adecuada, formando siempre un balance entre los paréntesis izquierdos y derechos.

Ejemplos:

```
$<AND <EQUAL 1 1> <OR <EQ 2 1> <EQ 'A 'A>>>
T
```

```
$(OR (EQUAL 2 3) (AND (EQ 1 1) (EQ 2 2)))
T
```

1.7 *Funciones que modifican las evaluaciones: (EVAL) y (QUOTE)*

LISP es un lenguaje donde por lo general, sus funciones están diseñadas para evaluar tanto la función como sus argumentos, pero cuenta con dos funciones con las cuales el usuario puede determinar cuando evaluar una expresión y cuándo no. Estas funciones son *EVAL* y *QUOTE*, las cuales se detallan a continuación, mostrándose su estructura y algunos ejemplos de su funcionamiento.

(EVAL):

La función *EVAL* admite un argumento, el cual debe ser un átomo. Esta función genera como reusltado el valor que esté asociado al argumento. El formato general de esta función es:

(*EVAL* < *argumento* >)

Ejemplo:

```
$(SETQ A 5)
5
                    $(EVAL A)
                    5
```

(QUOTE):

La función *QUOTE* admite un argumento, el cual puede ser un átomo o una lista, el valor que genera esta función es el argumento mismo, sin ser evaluado.

El formato general de esta función es:

(*QUOTE* < *argumento* >) *o bien* '< *argumento* >

Ejemplos:

```
$(QUOTE (A B C))
(A B C)
$(QUOTE (A) (B) (C))
(A)
$(QUOTE ( (A) (B) (C)))
((A) (B) (C))
```

```
$'A
A
$'(A B C)
(A B C)
$'((A) (B) (C))
((A) (B) (C))
```

1.8 *Funciones que manejan listas:* (CAR), (CDR), (CONS), (APPEND), (LAST) *y* (LEN)

En *LISP* se cuenta con un conjunto de funciones, las cuales tienen la capacidad de manejar listas de una manera más eficiente que cualquier otro lenguaje de programación.

Estas son: *CAR*, *CDR*, *CONS*, *LAST* y *LEN*. Estas funciones pueden combinarse y formar funciones de un grado de complejidad mayor. A continuación se muestran estas funciones, las cuales irán acompañadas de algunos ejemplos.

(CAR):

La función *CAR* acepta un sólo argumento, el cual debe ser una lista, esta función genera como resultado el primer elemento de la lista, que puede estar formado por un átomo o una lista. El formato general de esta función es:

(CAR < *argumento* >)

Ejemplos:

```
$(CAR '(A B C))
A

$(SETQ A '(1 2 3))
(1 2 3)
$(CAR A)
1
```

(CDR):

La función *CDR* acepta un sólo argumento, el cual debe ser una lista. Al aplicar esta función se obtiene como resultado una lista, que está formado por todos los elemen-

tos del argumento menos el primero, la forma general de esta función es:

(*CDR* < *argumento* >)

Ejemplos:

```
$(CDR '(A B C))
(B C)

$(SETQ A '(1 2 3))
(1 2 3)
$(CDR A)
(2 3)
```

(CONS):

La función *CONS* admite dos argumentos, el primero de ellos puede ser un átomo o una lista y el segundo debe ser una lista. El resultado que se genera al aplicar esta función es una lista, cuyo primer elemento es el primer argumento y el resto está formado por todos los elementos de la lista que integran al segundo argumento. La sintáxis general de esta función es:

(*CONS* < *argumento-1* > < *argumento-2* >)

Ejemplos:

```
$(CONS 'A '(B C))
(A B C)
$(CONS '(A B) '(D E F))
((A B) D E F)
```

(APPEND):

Esta función admite como argumentos a dos listas. El resultado que se generá al aplicar esta función es una lista en donde se reúnen los elementos de las dos listas que actúan como argumentos. La sintáxis de esta función es:

(APPEND *< lista-1 >* *< lista-2 >*)

A continuación se muestran algunos ejemplos:

```
$ (APPEND '(A  B)  '(C  D) )
(A  B  C  D)

$ (APPEND '((A))  '((B)) )
(  (A)  (B)  )
```

(LAST):

La función LAST acepta un sólo argumento, el cual de

be ser una lista. El valor que resulta al aplicar esta función, es una lista cuyo elemento está formado por el último elemento del argumento, el cual puede ser un átomo o una -- lista. El formato general de esta función es:

(*LAST* < *argumento* >)

Ejemplos:

```
$<LAST '(A B C)>
<C>

$<SETQ A '(B C D E F)>
<B C D E F>
$<LAST A>
<F>
```

(LEN):

La función *LEN* acepta un sólo argumento, si éste es una lista, la función genera como resltado el número de elementos que la integran; si el argumento es un átomo, se genera como resultado el valor cero. La sintáxis general de esta función es:

(*LEN* < *argumento* >)

Ejemplos:

```
$(LEN '(A B C))
3


$(SETQ A '(1 2 3 4 50))
(1 2 3 4 50)
$(LEN A)
5
```

COND.

La función *COND* es una de las más usuales en *LISP* ya que con el uso de ella se logra la construcción de estructuras del tipo *SI-ENTONCES*. Esta función acepta cualquier número de argumentos, los cuales son de la forma:

(*si* < *premisa* > *entonces* < *conclusión* >)

La forma general de esta función es la siguiente:

```
(COND
      ( < premisa1 >   < conclusión1 > )
      ( < premisa2 >   < conclusión1 > )
      .
      .
      .
      ( < premisaN >   < conclusiónN > ))
```

El valor de la función *COND* está dado por la conclusión correspondiente a la primera premisa que se satisfaga en orden descendente.

Ejemplos:

```
$<COND <<> X Y> <+ X Y>>
$      <T <SUB X Y>>>
```

```
$<COND <<EQ X Y> 0>
$      <<> X Y> 1>
$      <T -1>>
```

1.9 *Funciones que permiten poner como variable a una función (APPLY*) y (MAPCA)*

Otra de las ventajas que se logran con el uso del lenguaje *LISP*, es el poder construir una función cuyo argumento sea también una función, ésto es de gran utilidad, ya que procedimientos que son iguales y que solo varían por las funciones que éstos aplican, no es necesario duplicarlos, sino tan solo generar una función donde uno de sus ar-

gumentos sea una función, la cual puede ser variada para ca da caso en especial.

Las funciones de *LISP* que pueden usarse para este -- propósito es *APPLY** y *MAPCA*, las cuales se muestran a conti nuación:

(APPLY):*

La función *APPLY** acepta como argumento el nombre de la función y los argumentos que de ella se requieran, el -- formato general es:

(APPLY* < *función* > < *argumentos* >)

Ejemplos:

```
$(APPLY* '+ 2 3)
5

$(APPLY* 'AND (EQUAL 1 1) (EQUAL 2 2))
T


$(SETQ FUNCION '+)
+
$(APPLY* FUNCION 2 3)
5
```

(MAPCA):

Esta función acepta dos argumentos, el primero de -- ellos es una función y el segundo una lista de elementos. - Esta función aplica el primer argumento sobre cada uno de - los elementos que integran al segundo elemento y cada valor resultante se guarda en una lista.

Esta función se usa cuando una misma función va a a- plicarse a todos los elementos de una lista.

La sintáxis de esta función es:

(MAPCA < *función* > < *lista* >)

A continuación se muestran algunos ejemplos:

```
$(MAPCAR ÁTOM ´(2 A B (A) ))
( T T T NIL )

$( MAPCAR ´EQUAL ´( 1 2 3 ) ´(5 1 3 ) )
( NIL NIL T )

$( MAPCAR ´LIST ´(2 3 (A) (B) ) )
( (2) (3) ((A)) ((B)) )
```

1.10 Función de Asociación: (ASSOC)

En *LISP* se cuenta con una función que permite buscar información asociada a una clave dentro de un archivo de datos, el cual está formado por una lista que contiene a su vez sublistas.

Esta función es *ASSOC*, la cual genera como resultado la sublista en la que se encuentra la clave como primer elemento, o el valor *NIL* en caso de no encontrar esta clave dentro de los datos. A continuación explicamos el formato de esta función:

(ASSOC):

Admite dos argumentos, el primero de ellos puede ser un átomo o una lista, el segundo es una lista que contiene información guardada en sublistas cuya estructura es la siguiente:

```
( < clave-1 >   < lista-1 > )
( < clave-2 >   < lista-2 > )
.
.
.
( < clave n >   < lista n > )))
```

Esta función busca el primer argumento o clave dentro del primer elemento de cada una de las sublistas que forman al segundo argumento, si esta clave se encuentra co-

mo primer elemento de alguna de las sublistas, esta sublista es el valor que genera la función; en caso contrario el valor que se genera es *NIL*. La sintáxis general de esta -- función es:

(ASSOC < *elemento* > < *lista* >)

Ejemplos:

```
$(ASSOC 'MEXICO '((MEXICO (DF))
$                 (MICHOACAN (MORELIA))
$                 (PUEBLA (PUEBLA))))
(MEXICO (DF))

$(ASSOC 'NOMBRE '((RFC (AIGE550708))
$                 (NOMBRE (EDITH))))
(NOMBRE (EDITH))
```

1.11 *Funciones de Empacamiento:* (PACK) *y* (UNPACK)

En *LISP* se cuenta con dos funciones que permiten modificar la estructura de los átomos y de las listas, desempacando o empacando su información. Estas funciones son -- *UNPACK* y *PACK* mediante las cuales se pueden convertir áto-- mos a listas y viceversa.

Enseguida se explicará su funcionamiento y se darán algunos ejemplos sobre el uso de estas funciones.

(PACK):

La función PACK acepta un sólo argumento, el cual debe ser una lista, esta función concatena los elementos de una lista y forma con todos ellos un átomo. Si el argumento contiene sublistas éstas son ignoradas. El formato general de esta función es:

(PACK < *lista* >)

Ejemplos:

```
$(PACK '(E J E M P L O))
EJEMPLO
$(SETQ A '(T A R E A))
(T A R E A)
$(PACK A)
TAREA
```

(UNPACK);

La función UNPACK acepta un sólo argumento, el cual debe ser un átomo. Esta función convierte el argumento en una lista cuyos elementos son las partes que lo integran.

El formato general de esta función es:

(UNPACK < *átomo* >)

Ejemplos:

```
$<UNPACK 'EJEMPLO>
<E J E M P L O>
$<SETQ B 'RESUMEN>
RESUMEN
$<UNPACK B>
<R E S U M E N>
```

1.12 *Funciones para generar nuevas funciones (LAMBDA), (NLAMBDA), (DEFINEQ), (DEFINE) y (PROG)*

Otra de las cpacidades de *LISP* es poder construir -- nuevas funciones, en base a las ya existentes, las cuales - pasan de inmediato a formar parte de las funciones disponibles a usar. Para lograr ésto se tienen las funciones: *LAMBDA, NLAMBDA, DEFINEQ, DEFINE* y *PROG*. A continuación se explica el funcionamiento de cada una de ellas y se dan algunos ejemplos ilustrativos.

(LAMBDA):

Mediante la función *LAMBDA*, se logra la construcción

de una nueva función, en la cual los argumentos de las variables se encuentran definidas localmente. El formato de esta función es:

(LAMBDA (< *argumento-1*> <*argumento-2*>...<*argumento-n*>*cuerpo*>))

donde el, o los argumentos son usados dentro de la función que está definida por el cuerpo.

Ejemplos:

```
$(LAMBDA (X Y) (SETQ Y X))
(LAMBDA (X Y) (SETQ Y X))

$(LAMBDA (X) (SETQ Y X))
(LAMBDA (X) (SETQ Y X))

$(LAMBDA (X Y) (> X Y))
(LAMBDA (X Y) (> X Y))

$(LAMBDA (X Y) (+ X Y))
(LAMBDA (X Y) (+ X Y))
```

(NLAMBDA):

La función NLAMBDA está formado por un sólo argumen-

to y el cuerpo de la función. Dicha función a diferencia - de *LAMBDA*, no evalúa su argumento. La sintáxis general de- esta función es:

(*NLAMBDA* (< *argumento*>) (< *cuerpo* >))

Ejemplos:

```
$(NLAMBDA (Y) (CONS Y (' (A B))))
```

```
(NLAMBDA (Y) (CONS Y ((QUOTE (A B)))))
```

```
$(NLAMBDA (X) (SETQ A X))
```

(DEFINEQ):

La función *DEFINEQ* admite dos argumentos, el primero de ellos es el nombre al cual estará asociada la función, - que se define con el segundo argumento. La sintáxis gene-- ral de esta función es:

(*DEFINEQ* <*argumento-1*> (<*argumento-2*>))

Ejemplos:

```
$(DEFINEQ PRIMER (LAMBDA (X) (CAR X)))
(LAMBDA (X) (CAR X))
```

```
$(DEFINEQ MEDIA (LAMBDA (S N) (/ S N)))
(LAMBDA (S N) (/ S N))
```

(DEFINE):

La función *DEFINE*, al igual que la función *DEFINEQ* - admite dos argumentos y la principal diferencia entre ambas radica en que la función *DEFINE* evalúa el primer argumento y al valor asociado a éste se le asigna la función definida - por el segundo argumento, mientras que *DEFINEQ* no evalúa su primer argumento. El formato general de esta función es:

(*DEFINE* < *argumento-1* > (< *argumento-2* >))

Ejemplos:

```
$(DEFINE 'CUADRADO (LAMBDA (N) (* N N)))
(LAMBDA (N) (* N N))
```

```
$(DEFINE 'DOBLE (LAMBDA (N) (+ N N)))
(LAMBDA (N) (+ N N))
```

(PROG):

La función PROG es una función que simula la forma - de programación de otros lenguajes como ALGOL, PASCAL, FORTRAN, BASIC, etc., en donde se requiere de la ejecución devarias instrucciones, en este caso funciones, en orden se-- cuencial. La función PROG admite uno o más argumentos, y - aún se permite la programación de PROG sin argumentos, indicándolo con NIL. Dentro de esta función se pueden ejecutar todas las funciones LISP y aún generar nuevas funciones, todas las instrucciones dentro de PROG se ejecutarán en orden- descendente.

El valor de la función PROG será el que se genere -- con una instrucción que indica el final del programa, esta- función es (RETURN < valor final >), en caso de que no se - encuentre esta instrucción, se genera un mensaje de error - como parte final del programa.

El formato general de la función PROG es el siguiente:

```
( PROG ( < argumentos > )
       ( < instrucción-1 > )
       ( < instrucción-2 > )
       .
       .
       .
       ( < instrucción-n > ) )
```

Para lograr los procesos interactivo dentro de un -- programa se cuenta con la instrucción *"GO TO"*, la cual permite la transferencia incondicional del sistema al lugar -- marcado por una etiqueta, la cual debe de ser un átomo y debe de encontrarse dentro del programa. La sintáxis de esta función es la siguiente:

(*GO TO < etiqueta >*)

Si la etiqueta a la que se refiere la instrucción -- *"GO TO"* no se encuentra dentro de la función *PROG*, se generará un mensaje de error.

A continuación se dan algunos ejemplos del uso de la función *PROG*:

1) Programa que calcula el doble de un número

```
$(PROG (N)
$       (SETQ N (READ))
$       (SETQ DOBLE (+ N N))
$       (RETURN DOBLE))
```

2) Programa que suma dos números

```
$(PROG (X Y)
$       (RETURN (+ X Y)))
```

1.13 *Funciones de lectura e impresión* (READ), (PRINT), (PRINT1)

En *LISP* se cuenta con una colección de funciones que permiten la lectura y la impresión de listas.

Como *LISP* es un lenguaje interactivo, las lecturas se hacen en el momento en que se esté ejecutando la función que requiera datos del usuario. Las impresiones se pueden lograr en una pantalla o en una impresora. A continuación se explican estas instrucciones de lectura e impresión.

(READ):

Mediante la función *READ*, se pueden leer listas las cuales una vez leídas serán también el valor que la función *READ* tenga al ejecutarse.

La sintáxis de esta instrucción es:

(READ)

Ejemplos:

```
$(READ)

$LEE-UN-ATOMO
LEE-UN-ATOMO


$(READ)

$'(LEE UNA LISTA)
(QUOTE (LEE UNA LISTA))
```

(PRINT):

La función *PRINT* acepta un sólo argumento, el cual puede ser un átomo o una lista. Esta función realiza dos tareas a la vez, una de ellas es imprimir el argumento y la otra es generar como valor final el mismo argumento.

El formato general de esta función es:

(*PRINT* < *argumento* >)

Ejemplos:

```
$(PRINT 'IMPRIME-UN-ATOMO)

IMPRIME-UN-ATOMO
IMPRIME-UN-ATOMO


$(PRINT '(IMPRIME UNA LISTA))

(IMPRIME UNA LISTA)
(IMPRIME UNA LISTA)
```

(PRINT1):

La función *PRINT1* acepta un sólo argumento y la principal diferencia entre *PRINT* y *PRINT1* es que al ejecutarse ésta última, imprime el valor del argumento, pero no salta al siguiente renglón, permitiendo con ello imprimir mayor información en un mismo renglón.

El formato general de esta función es:

(*PRINT1* < *argumento* >)

Ejemplo:

```
$(PRINT1 '(IMPRIME Y NO SALTES RENGLON))
```

1.14 Recursividad

La recursión es otra de las capacidades con las que cuenta *LISP*, y se define como la capacidad de definir una función en términos de ella misma.

En toda función recursiva debe de existir una condición que indique el fin de todas las recursiones, ya que ésta se estará ejecutando mientras no se encuentre con una condición que al cumplirse indique lo contrario.

En algunas funciones recursivas se pueden dejar tareas pendientes de realizar hasta encontrar un valor final que indique el regreso para realizar todas aquellas tareas que quedaron pendientes.

Dentro de una función recursiva, se pueden definir también funciones que a su vez sean recursivas.

A continuación se muestran algunos ejemplos de funciones con naturaleza recursiva.

Ejemplo [1]

Cálculo del Factorial:

El factorial de un número entero y positivo "*n*" se-

expresa como $n!$, y el valor de éste se encuentra multiplicando (n) por ($n-1$).... , es decir, se decrementa el valor de n en la unidad y se realizan los productos sucesivos hasta que n sea igual a la unidad. La programación de la función factorial en *LISP* es la siguiente:

```
$(DEFINEQ FACTORIAL
$  (LAMBDA (N)
$   (COND ((EQ N 0) 1)
$    (T (* (FACTORIAL (SUB N 1))N)))))
(LAMBDA (N) (COND ((EQ N 0) 1) (T (* (FACTORIAL (SUB N 1)) N))))
```

Ejemplo (2)

Cálculo de la media aritmética:

Esta función se llama *MEDIA* y tiene como argumento una lista de valores a los cuales se les sacará su promedio con la ayuda de la función *MEDIAA*.

La programación de ambas funciones es la siguiente:

```
$(DEFINEQ MEDIA
$  (LAMBDA (DATOS)
$   (MEDIAA DATOS 0 0)))
(LAMBDA (DATOS) (MEDIAA DATOS 0 0))
```

```
$(DEFINEQ MEDIAA
$(LAMBDA (L S C)
$ (COND ((EQ L) (/S C))
$    (T (MEDIAA (CDR L) (+ S (CAR L)) (+ C 1)))))))
```

Ejemplo (3)

Suma de dos vectores:

La función *SUMA* acepta dos argumentos donde cada uno de ellos es una lista que representa un vector; el valor -- que genera esta función es la suma de ambos vectores, a los cuales se les considera de las mismas dimensiones. La programación en *LISP* de esta función es la siguiente:

```
$(DEFINEQ SUMA
$ (LAMBDA (X Y)
$  (COND ((EQ X)NIL)
$    (T (CONS (+ (CAR X) (CAR Y)) (SUMA (CDR X) (CDR Y))))))))
```

1.15 *Funciones que modifican la estructura de una lista. (RPLACA) y (RPLACD)*

En *LISP* se tienen dos funciones que se les puede denominar quirúrgicas, ya que con ellas el usuario puede modificar la estructura de una lista, reemplazando parte de ella por nuevos elementos. Estas funciones son: *RPLACA* y *RPLACD*, a continuación se explica la sintáxis de cada una de ellas y se dan ejemplos ilustrativos.

(RPLACA):

La función *RPLACA* admite dos argumentos, el primero de ellos es una lista a la cual se le va a sustituir su elemento *CAR* por el valor asociado al segundo argumento. La estructura general de esta función es:

(RPLACA < *argumento-1* > < *argumento-2* >)

Ejemplos:

```
$(RPLACA '(A B C) '(D (E)))
((D (E)) B C)
$(RPLACA '((A B C) D) '(D (E)))
'((D (E)) D)

  $(RPLACA '(A B C) 'Z)
  (Z B C)
  $(RPLACA '((A) B C) 'Z)
  (Z B C)
```

(RPLACD):

La función *RPLACD* admite dos argumentos, a el primer - argumento que es una lista se le va a reemplazar su *CDR* por el valor indicado en el segundo argumento, la sintáxis de - esta función es:

(*RPLACD* < *argumento-1* > < *argumento-2* >)

a continuación se muestran algunos ejemplos:

```
$(RPLACD '(A (B)) '(D C))
(A D C)

$(RPLACD '(A B C) '(D (E)))
(A D (E))
```

1.16 Función que genera números "Aleatorios" (RND)

En algunas ocasiones se requiere trabajar con núme-- ros que sean generados en forma aleatoria, en *LISP* se cuenta con la función *RND*, la que permite la generación de números "aleatorios". Enseguida explicaremos su sintáxis y daremos algunos ejemplos:

(RND):

RND es una función sin argumentos, la cual genera números "aleatorios" entre -32768 y 32767. La sintáxis general de esta función es:

(*RND*)

a continuación se muestran algunos ejemplos:

```
$(RND)
-29986
$(RND)
14283
$(RND)
7935
```

1.17 Notación punto

La notación punto surge cuando se encuentra ante el problema de construir una lista mediante la función (*CONS X Y*) cuando *Y* es un átomo diferente de *NIL*. El valor generado por esta función se representa por:

(*X* . *Y*)

y se le denomina notación punto.

El sistema *LISP* al trabajar internamente con cual- - quier lista maneja la notación punto. De esta forma pode-- mos representar a una lista ya sea como (A) o bien como - - (A . *NIL*) indistintamente ya que ambas notaciones generan - el mismo valor.

Ejemplos:

```
$(CONS  'A 'B)
(A . B)
$(CONS '(A B) 'C)
((A B) . C)
$(CONS '( (A) B) 'C)
(((A) B) . C)
```

REFERENCIAS

1. Y. Gloess, Paul. (1981). "Understanding Artificial Intelligence", Alfred Publishing Co., Inc. California, U.S.A.

2. Feigenbaum, Edward A. (1982). "Innovation and Symbol manipulation in fifth generation Computer System". Fifth Generation Computers Metting. Japan.

3. R. Chilausky, B. Jacobsen and R.S. Michalski. (1976). "An application of Variable-Valued Logic to Inductive Learning of Plant Disease Diagnostic Rules", Proc. Sixth Annual Int'l Symp. Multiple-Valued Logic.

4. S.M. Weiss et. al. (1978). "A Model-Based Method for Computer-Aided Medical Decision-making". Artificial Intelligence, Vol. 11, No. 2, pp. 145-172.

5. E. Feigenbaum, G. Buchanan and J. Lederberg. (1971)."Generality and Problem Solving: A Case Study Using the DENDRAL Program". Machine Intelligence G.D. Meltzer and D. Michie, eds., Edinburg University Press, - - pp. 165-190.

6. R. Davis, et. al. (1981). "The Dipmeter Advisor: Interpretation of Geological Signals". Proc. Seventh Int'l Joint Conf. Artificial Intelligence. Aug. 1981.

7. R.M. Stallman and G.J. Susman. (1977). "Fordward Reasoning and Dependency-Directed Backtraing in a System for Computer-Aided Circuit Analysis," Vol. 9, Artificial Intelligence, 1977, pp. 135-196.

8. H.E. Pople. (1977). "The Formation of Composite Hypotheses in Diagnostic Problem Solving; An Excersice in Synthetic Reasoning". Proc. Fifth Int'l Joint Conf. Artificial Intelligence, pp. 1030-1037.

9. J. Reggia, et. al. (1980). "Towards an Intelligent Textbook of Neurology". Proc. Fourth Annual Symp. Computer Aplications in Medical Care, pp. 190-199.

10. J. Moses. (1971). "Symbolic Integration: The Stormy Decade". Comm. ACM, Vol. 14, No. 8, pp. 548-560.

11. B. Chandrasekaran, et. al. (1979). "An Approach to Medical Diagnosis Based on Conceptual Structures". Proc. Sixth Int'l Joint Conf. Artificial Intelligence, pp. 134-142.

12. N. Martín, et. al. (1977). "Knowledge-Based Management for Experiment Planning in Molecular Genetics". Proc. Fifth Int'l Joint Conf. Artificial Intelligence, pp. 882-887,

13. R. Davis, B. Buchanan and E. Shortliffe. (1977). "Production Rules as a representation for a Knowledge Base Consultation Program". Artificial Intelligence, Vol. 8, No. 1, pp. 15-45.

14. P.E. Hart, R.O. Duda and M.T. Einaudi. (1978). "A computer Based Consultation System for Mineral Exploration". Tech. Report, SRI International, Menlo Park, Calif.

15. J. Osborn, et. al. (1979). "Managind the Data from Respiratory Measurements". Medical Instrumentation, Vol. 13, No. 6, Nov.

16. J. McDemott and B. Steele. (1981). "Extending a knowledge Based System to Deal with ad hoc Constraints". Proc. Seventh Int'l Joint Conf. Artificial Intelligence, pp. 824-828.

17. Tracton, Ken. (1980). Programer's Guide to LISP". Copyright. by TAB BOOKS Inc.

18. Teitelman, W. (1974). "INTERLISP Reference Manual". Xerox Corporation (palo Alto Research Center), Palo Alto, CA and Bolt Beranek and Newman, Cambridge, Ma.

19. Winston, Patrick Henry. (1981). "LISP". Addison-Wesley Publishing Company, U.S.A..

20. J. Bonar and S. Levitan.(1982). "APP-L-ISP". BYTE Publications Inc. June, pp. 220-230.

21. Hofstadter, Douglas R. (1980. "Godel, Escher, Bach an Eternal Golden Braid". Vintage Books, New York, USA.

22. Minsky, Marvin. (1975). "A Framework for Representing Knowledge". In the Psychology of Computer Vision, edited by Patrick H. Winston, pp. 211-277.

23. Roberts, R. Bruce and Ira P. Goldstein. (1977). "The FRL Primer". Memo No. 408, Artificial Intelligence Laboratory, MIT, Cambridge, Massachussetts, July.

24. Roberts, R. Bruce and Ira P. Goldstein. (1977). "The FRL Manual". Memo No. 409, Artificial Intelligence Laboratory, MIT, Cambridge, Massachussetts, June.

25. Dana, S. Nau.(1983). "Expert Computer Systems". Computer, pp. 63-88, February.

26. Henderson, Peter.(1980). "Functional Programming, Application and Implementation". Prentice-Hall International, Inc. London.

27. S. Galofre Llanos y Sara Escobar. (1980). "El arte de escribir correctamente". Psicología Técnica Aplicada S. C., México.

28. Marling, D. Park and S. Russell. (1960). "LISP 1 Programer's Manual". Artificial Intelligence group, Computation Center and Research Laboratory of Electronics. MIT, Cambridge, Massachussetts, March.

29. McCarthy, John, P.W. Abrahams, D.J. Edwards, T.P. Hart and M.I. Levin (1962). "LISP 1'5 Programer's Manual". The MIT Press, Cambridge, Massachussetts.

30. McCarthy, John. (1978). "History of LISP". ACM SIGPLAN Notice, Vol. 13, No. 8, pp. 217-223, August.