UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

LISP-PERITO:

UN LENGUAJE CONCURRENTE PARA EL DESARROLLO DE APLICACIONES EN INTELIGENCIA ARTIFICIAL.

DISEÑO E IMPLEMENTACION

T E S I S   P R O F E S I O N A L

PARA OBTENER EL TITULO DE :

M A T E M A T I C O

P R E S E N T A:

MARCO ANTONIO GALVAN JIMENEZ

MEXICO D.F Julio 1989

TESIS CON
FALLA DE ORIGEN

Universidad Nacional
Autónoma de México

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

TABLA DE CONTENIDOS

PROLOGO 1

INTRODUCCIÓN 3

Introduccion 4

1.0 Sistemas de produccion 6

1.1 Arquitectura de los sistemas de producción 7

1.2 Programacion basada en reglas 10

1.3 Apareamiento de patrones 15

1.3.1 Patrones y reconocmiento de patrones 16

1.3.2 Recuperacion de informacion por apareamiento de patrones 17

1.3.3 El apareo de patrones y PERITO 19

1.3.6 Soluciones al problema 20

1.4 Problemas con la implementacion de un lenguaje de sistemas de produccion 23

1.4.1 Uso de redes 24

1.5 Algoritmo de apareamiento 25

Nodos de prueba constante 29

Nodos de dos entradas 30

Nodos terminales 30

1.6 Logros 33

2. PROGRAMACIÓN EN PERITO: UNA INTRODUCCIÓN A LA PROGRAMACIÓN BASADA EN REGLA 34

2.0 Introduccion 35

2.1 El lenguaje de programacion PERITO 35

2.2 Tipos de datos definidos en el lenguaje 39

2.2.1. Tipos de datos primitivos 39

2.3 Hechos o elementos de la memoria de trabajo 40

2.4 Memoria de produccion 40

2.5 Elementos condicion o patrones 42

2.6 Definición de reglas 47

2.7 Declaracion de hechos iniciales 49

2.8 Comparaciones 50

2.9 Prioridad 52

2.10 El lado izquierdo de las reglas de PERITO 53

2.11 Elementos negados 55

2.12 El lado derecho de la reglas de PERITO 55

2.12.1 Acciones de modificacion de la memoria de trabajo 56

2.13 Comando de terminacion de ejecucion 64

2.14 Eliminacion de hechos en bloque 64

2.15 Ejemplos de programacion con PERITO 65

Resumen de la sintaxis de PERITO 70

3. EL ALGORITMO DE APAREAMIENTO DE RED. 74

3.1 Introduccion al algoritmo 75

3.1.1 Redundancia temporal 75

3.1.2 Similitud estructural 76

3.1.3 Compilacion del LI 77

3.1.4 Introducción al algoritmo de red 78

3.1.5 Tipos de cambio de memoria soportados 79

3.2 Elementos condicion no negados 79

3.2.1 Producciones con un elemento condicion 79

3.2.2 Los datos procesados por los nodos 83

3.2.3 Procesamiento de la red para ejemplo_1 85

3.2.4 Producciones con dos elementos condición 85

3.2.5 Eliminacion de los elementos de la memoria de trabajo

3.2.6 Como los nodos de dos entradas manejan las etiquetas 92

3.2.7 Las memorias internas de los nodos de dos entradas 92

3.2.8 Variables que aparecen mas de una vez en un elemento condicioncion 93

3.2.9 Producciones mas complejas 96

3.2.10 Producción de multiples tokens de salida 99

3.2.11 Variables que ocurren multiples veces en un elemento condición 101

3.3 Elementos condicion negados 102

3.3.1 El nodo <NOT> 102

3.3.2 Memorias en el nodo <NOT> 106

3.3.3 Nuevos tokens INVALIDOS que llegan por la derecha 106

3.4 Consideraciones de eficiencia 109

3.4.1 Redundancia Temporal 109

3.4.2 Similitud estructural 109

3.5 Los nodos de programa 114

3.5.1 Los nodos de una entrada que prueban aspectos constantes 114

3.5.2 Los nodos de una entrada que prueban la sustitucion de variables 115
3.5.3 Los nodos ordinarios de dos entradas 115
3.5.4 El nodo <NOT> 118
3.5.5 El nodo que cambia el conjunto conflicto 120
3.5.6 El nodo bus 121

4. EL INTERPRETE PERITO 122
4.1 Reduccion del costo de almacenamiento de los tokens
4.1.1 Nodos de memoria separados 123
4.1.2 Memorias para los nodos <NOT> 127
4.1.3 El procesamiento efectuado por los nodos de memoria 127
4.2 El formato de datos del interprete 135
4.2.1 El formato de los tokens 135
4.2.2 El formato del nodo --preliminares 136
4.3.3 Los nodos de una entrada 136
4.3.5 El nodo memoria 137
4.3.6 El nodo &DOS 138
4.3.7 El nodo ordinario de dos entradas 138
4.3.8 El nodo &NOT 139
4.3.9 El nodo &P 140
4.3.10 El nodo &BUS 141
4.3.11 Una red de ejemplo 141
4.4 El interprete de los nodos 142
4.4.1 Los nodos de una entrada 143
4.4.2 El nodo &VIN 144
4.4.3 El nodo &MEM 144

4.4.4 El nodo &DOS 144

4.4.5 El nodo &VEX 145

4.4.6 El nodo &NOT 146

4.4.7 El nodo &BUS 148

4.4.8 El nodo &P 149

5. TEMAS SELECTOS 150

5.1 Justificacion en terminos de complejidad para el uso de un algoritmo que salva estado. 151

APENDICE

El editor

BIBLIOGRAFÍA

PROLOGO

Los *sistemas de producción* o (*sistemas basados en reglas*) han sido ampliamente usados en Inteligencia Artificial para modelar el comportamiento inteligente y para la construcción de sistemas expertos. Sin embargo, la mayoría de los sistemas de producción, son desde un punto de vista computacional, grandes consumidores de recursos, especialmente tiempo.

Es bastante frecuente que los programas de Inteligencia Artificial gasten el mayor porcentaje del tiempo de ejecución evaluando patrones para localizar, ya sea subprogramas o entradas en una base de datos. Esto es particularmente manifiesto en los *sistemas de produccion* porque a diferencia de otros programas, no existe otra alternativa mas que la evaluación de patrones. Algunos otros programas pueden llamar funciónes por nombre o por algún otro medio, mientras que un sistema de producción usa la evaluación de patrones para seleccionar cada procedimiento que ejecutará y para localizar cada pieza de datos sobre las cúales operan los procedimientos. En esta tesis presentamos un lenguaje de sistemas de produción y un algoritmo altamente eficinte para la evaluación de patrones, que pueda ser utilizado en un contexto completamente diferente del lenguaje aquí presentado.

La presente tesis se divide de la siguiente manera.

Introducción.

El lenguaje de PERITO.

El algoritmo de red.

Consideraciones de eficiencia y estrategia de resolucion de conflictos

Dentro de la virtudes que tiene el algoritmo de apareamiento presentado, podemos mencionar las siguientes:

1. Nota las similitudes en los patrones, de tal forma que evita hacer la misma prueba más de una vez;

2. Saca ventaja del hecho de que tanto el conjunto de patrones y el conjunto de objetos cambia muy lentamente, de tal forma que se pueda almacenar información de una evaluación a las siguientes; y

3. Permite un alto grado de actividad en paralelo durante la evaluación.

Este mtodo involucra el uso de un compilador que traduce los patrones en un programa para una máquina virtual.

INTRODUCCION

Los sistemas de producción han sido utilizados de manera intensiva para entender la naturaleza de la inteligencia humana, por ejemplo, en el modelamiento cognoticitivo. en sistemas de solución de problemas, y en el estudio de sistemas para comprender el aprendizaje. Asimismo, han sido usados para desarrollar un gran cantidad de sistemas expertos en areas como CAD, medicina, tarea de configuración de computadores, exploración geológica, y formación de teorías matemáticas. La mayoría de los probadores automáticos de teormas (por ejem. PROLOG) pueden ser vistos como manipuladores de sistemas de produccion.

Han habido variados intentos por hacer de los sistemas de producción no sólo un formalismo de representación de conocimiento, sino tambin un lenguaje de programación. Entre estos intentos esta el trabajo de OPS5, que en promedio puede activar de 3 a 10 reglas por segundo en una Vax-11/780 .

Veamos algunas razónes para aumentar la velocidad de los sistemas de producción. La actividad cognocitiva de un agente inteligente involucra dos tipos de busqueda :

(1) *Búsqueda de conocimiento*, esto es, búsqueda por un agente inteligente en su base de conociminto para encontrar la información que es relevánte én la solución de un problema dado; y

(2) *Búsqueda en el espacio problema*, esto es, búsqueda dentro del espacio problema de un estado meta.

La *búsqueda en un espacio problema* se manifiesta como una búsqueda AND/OR combinatória. Puesto que la búsqueda en

el espacio problema se manifiesta como explosivamente combinatoria cuando no es podada por el uso del conocimiento, un agente inteligente, sin importar lo que este haciendo, debera realizar una buena cantidad de busqueda de conocimiento antes de efectuar cualquier paso. Esto resulta en búsquedas de conocimiento como parte de un loop interno de la computacion a realizar por el agente inteligente. Mas aun, conforme la inteligencia del agente es incrementada (i.e conforme aumente el tamaño de la base de conocimientos), los recursos necesarios para realizar la búsqueda de conocimiento tambien se incrementán, por lo que es muy importante aumentar la velocidad de la búsqueda de conocimiento tanto como sea posible.

La búsqueda de conocimiento forma un componente esencial en la ejecución de los sistemas de producción. Cada ciclo de ejecución de un sistema de producción involucra un paso en la búsqueda de conocimiento (la fase de *apareamiento*), donde el conocimiento representado en las reglas es apareado contra la memoria global de datos. Puesto que la habilidad de hacer búsquedas de conocimiento de manera eficiente es fundamentál en la construción de agentes inteligentes, se sigue que la habilidad de ejecutar sistemas de producción con conjuntos grandes de reglas a altas velocidades ayudará considerablemente en la construcción de programas inteligentes. En resumen, la fase de apareamiento (*búsqueda de conocimiento*) debe de ser hecha, en una forma u otra, en cualquier sistema inteligente. Por lo que, acelerar esa fáse es una parte esencial en la construcción de agentes altamente inteligentes. Más aun, puesto que los sistemas de producción ofrecen un modelo altamente transparente de búsqueda de conocimiento, los resultados que se obtengan en la aceleración de la ejecución de los mismos tendrán implicaciónes en otros modelos de computaciónes inteligentes

que involucren búsqueda de conocimiento.

Se han desarrollado varios algoritmos para alcanzar este objetivo. La presente tesis describe el diseño e implementación de un lenguaje de reglas de producción, basado en el algoritmo de RED, que hace uso de las caracteristicas de redundáncia temporal y similitud estructural de la base conocimiento para reducir la complejidad del problema.

1.0 SISTEMAS DE PRODUCCION

Un sistema de producción consiste en un conjunto de módulos o procediminto llamadas reglas de producción y una base de datos sobre las cuales las reglas de producción operan. Cada regla de producción denota un par condición-acción, en donde la condicion debe ser satisfecha por un estado en la base de datos antes que la producción pueda ser aplicada a ese estado. Ningún medio explícito de control procedural esta provisto. El sistema es únicamente dirigido a eventos, y la invocación de las producciónes puede ser lograda indirectamente a travs de la base de datos.

Esta falta de estructura de control, resulta en un sistema que es fuertemente modular, flexible y adaptativo, caracteristicas indispensables para sistemas basados en conocimiento.

Los sistemas de producción fueron introducidos en los años 40's por el matemático Post, quien demostró, entre otras muchas cosas, que un lenguaje basado en este sistema es equivalente a una maquina de Turing. Este sistema se ha usado ampliamente en el estudio de los lenguajes formales, de hecho, hoy en día las gramáticas formales se representan

con este formalismo.

1.1 ARQUITECTURA DE SISTEMAS DE PRODUCCION.

Introduzcamos la nocion de *sistema de produccion no deterministico* (SP).

Un sistema de produccion no deterministico es una extension directa de los sistemas de producción de Post. Consiste en conjunto de reglas de produccion llamado *memoria de producciones*, y una *base de datos* o *memoria de trabajo* a la que las reglas de producción son aplicadas. Cada regla de produccion es una expresión o cadena de simbolos, las cuales consisten de dos partes, la primera llamada el *antecedente* o el *lado izquierdo* (LI), y la segunda el *consecuente* o *lado derecho* (LD). Estos, respectivamente, denotan una *condición* que debe de ser satisfecha antes de que la produccion pueda ser aplicada o invocada, y una *acción* que especifica el resultado de aplicar la produccion a la base de datos. Un estado dado s_1 de la base de datos puede ser transformado en otro estado s_2 si el conjunto de producciones contiene una producción cuya condicion es satisfecha en s_1 y cuya accion aplicada a s_1 produce el estado s_2. Si por un secuencia de tales transiciones podemos transformar un estado s_1 a algún estado s_n, entonces podemos decir que s_1 *genera* s_n, y llamaremos a tal secuencia de transiciónes un *camino* desde s_1 a s_n. Por lo que abstractamente, un sistema de produccion es una máquina no determinista que describe conjuntos de secuencias sobre estados de la base de datos, por lo que determina una relación sobre esos estados.

Usualmente, dado un *estado inicial* y *una condicion meta* deseamos determinar que estados que satisfagan la condición meta son generados por un estado inicial. Llamanos a este conjunto de estados el *conjunto solucion* del sistema de

produccion (estrictamente hablando, con respecto al estado inicial dado y la condicion meta). En algunos casos podemos estar solo interesados en encontrar solo un miembro del conjunto solucion, en otros podemos requerir que los caminos del estado inicial a los estados solucion tambien queden determinados.

Por ejemplo, considerémos el siguiente sistema de producción donde los estados de la base de datos son palabras del alfabeto { S,A,B,C,a,b,c } y donde las producciónes (las cuales van a ser interpretadas en el sentido usual como reglas de reescritura) incluyen:

p_1: $S \to ABC$, p_5: $A \to a$
p_2: $A \to aA$, p_6: $B \to b$
p_3: $B \to bB$, p_7: $C \to c$
p_4: $C \to cC$,

Supongámos que el estado inicial es el símbolo S, y que el estado meta es aquel en el que el estado sea una palabra sobre los símbolos "terminales" a, b, c. Entonces, el conjunto solucion es el sistema formado por el conjunto de palabras

$L = \{a^l b^m c^n : l \geq 1,\ m \geq 1,\ y\ n \geq 1\}$

Hay dos formas en que puede ser construido un interprete para un sistemas de producción no determinístico. En el esquema mas simple, las condiciones de cada producción son evaluados con el estado en ese momento de la base de datos, una de las producciónes aplicable es seleccionada, y la accion especificada por la produccion es ejecutada. Este ciclo *reconoce-actua* es repetido para el nuevo estado de la base de datos. La ejecucion termina cuando no haya regla de produccion aplicable o cuando la condicion meta sea

satisfecha. El conjunto de producciónes aplicable en cada estadio de la ejecucion se conoce como el *conjunto conflicto*. Este esquema es no *determinista*, y la implementacion en una máquina determinista necesita especificar el orden en el cual las producciones aplicables son seleccionadas para aplicacion (*la estrategia de resolucion de conflictos*) y el orden en el cual los estados son expandidos (*la busqueda en el grafo*).

Tal esquema de interprete se dice que es de encadenamiénto hacia adelante o dirigido a datos.

La estrategia de resolución de conflictos, y la dirección de evaluación determina la manera en la cual el espacio problema es recorrido, y colectivamente las llamaremos *esquema de evaluación* o *estrategia de búsqueda*. El esquema de evaluacion determina el orden en el cual las soluciónes son generadas, pero de ninguna manera afecta el conjunto de solución.

Muchas arquitectúras de sistemas de producción, sin embargo, no son no-deterministicas en el sentido anterior. Por ejemplo, el esquema de resolución de conflictos puede requerir seleccionar, irrevocablemente, una sola producción del conjunto conflicto y descartar a las restantes. Estos sistemas de produccion son vistos más adecuadamente como generalizaciónes del algoritmo de Markóv que de los sistemas de Post. Por ejemplo, el sistema de producción de Newell y Simon, usa un conjunto ordenado de producciones ordenado de producciónes y solo la primera producción satisfecha por la base de datos es siempre invocada. Similarmente, McDermontt especifica estratégias irrevocables de resolucion de conflictos, en el cual las producciones son seleccionadas en base a uso repetido, generalidad y otros criterios funcionales.

En tales casos, el conjunto solución depende, no solo de las producciónes y de la base de datos, sino tambien del esquéma particular de evaluacion (interprete) usado.

1. 2 PROGRAMACION BASADA EN REGLAS

Un lenguagé de sistemas de produccion es bastante diferente de los lenguages convencionales de programación procedural (por ejemplo PASCAL O C) o de los lenguajes aplicativos (LISP) o de los lenguajes orientados a objetos (tales como SMALLTALK). Una de las principales diferencias consiste en el uso de reglas sin orden de ejecución explicito, (*sensitivos a datos*), mas que a una secuencia de instrucciónes.

Una de las diferências principales con otros lenguajes de programacion es que no existe el concepto de invocación explicita, ni de secuenciacion de llamadas, asi mismo, el modo de operacion de este lenguaje es la concurrencia.

El modelo de programacion de sistemas de produccion no deterministico (SP) tiene los mismos tres componentes fundamentáles que el modelo familiar de los lenguajes procedurales, pero sus componentes difieren en detalles.

Uno de los componentes de ambos modelos es el de *programa*, el cual expresa las computaciones a ser efectuadas; otro componente es el *ejecutánte*, el cual lleva a cabo las computaciónes. Ambos modelos tienen tambien datos, los cuales describen las particularidades del problema a ser resuelto y almacenan los resultados intermedios.

Un programa en el modelo procedural es una secuencia

ordenada de unidades básicas denominadas *instrucciones*. El ejecutante lleva a cabo las instrucciones en el orden en que son dadas. En contraste, un programa de sistemas de producción consisten en un conjunto no ordenado de unidades basicas, las *producciones*

En un sistema basado en reglas, el control esta basado en una frecuente revaluacion del estado de los datos, no en alguna estructura de control estatica del programa. Por lo que decimos que una computación en el modelo de sistemas de producción es dirigido u orientado a datos mas que orientado a instrucciones.

De acuerdo a esta filosofia, las reglas se pueden comunicar entre si solo mediante datos. No existe la transferéncia de control entre reglas como la hay en el modelo de programacion procedural. El nombre de las reglas solo sirve para propositos de documentación; estas no pueden ser referenciádas por otras reglas en ningún llamado a subrutina. A diferencia de las instrucciónes, las reglas no son ejecutadas secuencialmente, por lo que no es posible determinar por inspeccion de un conjunto de reglas cual regla sera ejecutada primero o que regla causará la terminación del programa.

A diferencia del modelo procedural de computacion, en el cual el conocimiento acerca del dominio particular del problema esta mezclado con las instrucciones acerca del flujo de control, el modèlo de sistemas de producción permite una separacion mas completa del conocimiénto (en las reglas) del control (provisto por el ejecutor). En la práctica, sin embargo, esta separacion es dificil de obtener en ciertos modelos de sistemas de produccion. Una consecuencia de esta separación es que el ejecutor en el modelo de sistemas de produccion es mucho mas complejo que

el ejecutor en un modelo procedural.

La fig [1] resume las caracteristicas de una computacion procedural, mientras que un resumen contrastante de las caracteristicas de computacion en el modelo de sistemas de produccion es dado en la fig [2].

Programa o descripcion de la computación procedural.

Una descripcion de una computación es una lista ordenada de instrucciones escritas en un lenguaje con una sintaxis y una semantica bien definida.

La lista tiene un comienzo especificado.

La lista incluye una instruccion de terminacion cuyo significado es cesar la ejecucion del programa.

El ejecutor.

Comienza la ejecucion en una instrucción distinguida denominada la primera instrucción;

Ejecutara a todas las instrucciónes secuencialmente, comenzando con la primera instrucción, a menos de que las instrucciónes especificamente indiquen una transferencia;

Cesa la ejecución una vez que encuentra el simbolo de terminacion.

Figura [1] . El modelo procedural de computacion.

Programa o descripcion de la computacion basada en reglas

Una descripcion de una computacion es un conjunto no ordenado de reglas.
Cada regla en el conjunto de reglas, o memoria de produccion, esta compuesta de dos partes: la parte condicion y la parte de accion.

El ejecutor.

Establece y mantiene la base de datos o memoria de trabajo (memoria de hechos), la cual es una base de datos global de simbolos que representan afirmaciones resultantes de la ejecucion del programa;

Tiene una fase de apareamiento (aparea-reglas) que consta de un algoritmo de apareamiento para comparar las reglas con la memoria de hechos, resultando en un conjunto conflicto constituido por los apareamientos de la reglas.

Tiene una fase de seleccion de reglas, que consta de algoritmos de seleccion, llamados estrategias de resolucion de conflictos, para escoger las reglas dentro del conjunto conflicto;

Tiene una fase de ejecucion de reglas, o interprete de los lados derechos de las mismas, que consiste en algoritmos para ejecutar o interpetar las acciónes de las reglas;

Opera ciclicamente sobre los estados hasta que la fase de apareamiento genere un conjunto conflicto vacio;

Figura [2]. El modelo de computacion de sistemas de produccion.

Un intérprete de sistemas de producción es una computadora que involucra un procesador, más dos memorias disjuntas, la memoria de producciónes o MP y la memoria de trabajo, MT. La memoria de produccion contiene un programa ejecutado por el procesador, y la memoria de trabajo contiene los datos sobre los cuales opera el programa. Los objetos contenidos en la memoria de trabajo son llamados elementos de dato.

A diferencia de programas de computación convencional, un sistema de produccion no incorpora el concepto de flujo secuencial de control a traves del programa. El flujo de control en un sistema de producción esta determinado por el orden en el cual las partes condición de la producción sean satisfechas. El intérprete ejecuta repetidamente los siguientes pasos.

1.Determina que producciónes tienen sus partes condición satisfechas.

2. Si no hay producciónes con partes condición satisfechas, ALTO, de otra manera, selecciona una produccion de aquellas que lo hagan.

3. Realiza las acciónes especificadas por las producciones escogidas.

4. Ve al paso uno.

Esta secuéncia es llamada el ciclo de reconocimiento-acción. El paso (1) del ciclo es llamado el apareamientó. El paso (2) es llamado el de resolución de conflictos y el paso (3) es llamado el de actuación.

Uno de los problemas fundamentales en la implementación

de un lenguaje basado en reglas de produccion es el de la realizacion del apareamiento eficientemente. Los sistemas de produccion han operado historicamente de uno a dos ordenes de magnitud mas lentos que los programas convencionales, debido en gran parte a la dificultad de efectuar el apareamiento. Debido a que los sistemas de producción estan siendo aplicados a tareas cada vez mas complejas, el tamaño del sistema de producción esta incrementandose. Como es tal vez obvio, el costo de realizar el apareamiento depende de tanto el numero de producciones en la memoria de producciones y el número de elementos de datos en la memoria de trabajo. Esta tesis presenta una descripción de lisp-perito un lenguaje de reglas de produccion que hace uso de un algoritmo eficiente para el apareamiento de patrones que reduce el costo de encontrar cuales elementos condicion son satisfechos.

1. 3 APAREAMIENTO DE PATRONES.

El apareamiénto de patrones es un problema central en Inteligencia Artificial. Diversas técnicas de apareamiénto de patrones son utilizádas en muchas areas tales como comprension del lenguaje natural, manipulación simbolica de expresiónes, sistemas expertos, transformación y sintesis automatica de programas, etc, etc.

El aparemiento de patrónes es tan basico en IA que este es provisto como primitiva en algunos lenguajes de inteligencia artificial.

El significádo de la palabra "patrón" en IA no es identico a los muchos significados atribuidos a esta palabra en otras areas de la computación. Por ejemplo, en el area de *reconocimiénto de patrones* un patrón usualmente es un vector de facetas, en el cual cada elemento del mismo representa un

valor númerico de la faceta del patrón que esta siendo descrito. En el area *reconocimiénto sintáctico de patrones* un patrón esta definido como una oracion generada por una gramatica que describe la clase a la cual el patron pertenece. En *procesamiento de texto* un patron es una cadena de caracteres, y el apareo de patrones es la operación que encuentra ocurrencias del patron dentro del texto. El significado de *patrón* y de *reconocimiento de patrones* usado en IA se describe a continuación.

1.3.1 Patrones y reconocimiénto de patrones

Un *patrón* es una descripción parcial de alguno objeto, con algunas piezas que quedan sin especificar. Un patrón puede ser definido recursivamente como sigue:

Un simbolo es un patron.

Una lista de patrónes es un patrón.

Solo las expresiones construidas por las reglas anteriores son patrones.

La notación de lista permite el uso de la misma representación para describir cualquier tipo patrón independientemente de su significado. Esto tiene la ventaja adiciónal de que los mismos algoritmos pueden ser usados independientemente del contenido del patrón (en particular el algoritmo usado en la implementación de nuestro lenguaje).

Un patrón puede contener un simbolo especial que indica *variables de patrón*. Las variables son usadas en el proceso de aparemiento. Un simbolo que no es una variable es un patron constante.

El apareamiento de patrones (o en ingles pattern matching) es el proceso de comparar patrones para ver si ellos son similares. Dos patrones son similares si pueden ser apareados. Un patrón aparea con otro si los dos pueden ser iguales sustituyendo alguna expresion por las variables en el patron. Una variable puede ser apareada con cualquier expresion con la condicion de que cada ocurrencia de la variable sea sustituida por la misma expresion (consistencia en la sustitucion de variables) .

El proceso de apareamiento fallara si es imposible aparear los dos patrones. Este tendrá éxito cuando los dos patrones son identicos o cuando ellos se pueden hacer identicos con sustituciónes apropiadas para las variables.

Por ejemplo, suponga que los nombres de las variables estan indicados prefijandolos por un simbolo de interrogación. El patron (EL ?X ES ?Y) puede ser apareado con (EL CAMINO ES SINUOSO) remplasando ?X con CAMINO y Y con SINUOSO. El mismo patrón pudo haber sido apareado a (LA CASA ES (GRANDE Y VACIA)) remplazando X con CASA y Y con (GRANDE Y VACIA) pero no con (JUAN ES EL JUGADOR) o con (EL) ARBOL ES ALTO. En los ejemplos previos ?X y ?Y son variables de patron mientras que EL, CAMINO, (GRANDE Y VACIO), etc., son constantes de patron.

1.3.4 Recuperacion de información por apareamiento de patrones

La idea básica es que la información debe de ser recuperada por patrónes en lugar de por nombre. Esto significa que el nombre (o la dirección de almacenamiento) de la información almacenada no tiene que ser recordada. La información es recuperada por *contenido* puesto que el patrón describe los contenidos de las piezas de informacion

buscada. Aun cuando el apareamiento de patrones es un proceso sintactico, esto nos permite mantener en una mejor forma el significado semantico de las piezas de datos.

La implementación de recuperacion de informacion dirigida por patrones tiene que ser hecha con cuidado. La exploración en una base de datos completa, y la prueba de cada aspecto contra cada patrón es computacionalmente muy caro cuando la base de datos es no trivial. Para lo cual varios mecanismos de indexamiento son necesarios.

Esta técnica es muy importante en sistemas expertos. Considere por ejemplo, un sistema basado en reglas de produccion que hace encadenamiento hacia adelante (forward chaining). Una regla de producción es activada como resultado de una apareamiento de patrónes entre el contenido de la memoria de trabajo y la parte antecedente de la regla de producción.

En sistemas expertos se ha visto que es necesario aparear muchos patrones contra muchos objetos. El apareador de patrónes tiene que encontrar a cada objeto que aparea cada patrón. Este tipo de aparemiento de patrones puede alentarse considerablemente conforme el numero de patrónes u objetos involucrados crece.

En la presente tesis se presenta un algoritmo para aparear de manera muy eficiente muchos patrones contra muchos objetos. Se da una descripción del algoritmo asi como su uso en un lenguaje de reglas de producción denominado PERITO.

PERITO es un lenguaje de reglas de producción con invocación dirigida por patrónes.

La nocion de programacion dirigída por patrónes se

introdújo en el diseño del lenguaje PLANNER. La idea importante es que los procedimiento son llamados dependiendo de la situacion mas que por una secuencia preprogramada.

Puesto que es posible que exista mas de una regla que sea satisfecha en un momento dado, este método introduce de manera natural el no determinismo.

1.3.5 El apareo de patrones y PERITO

Una produccion es una lista que contiene un separador especial. La parte condicion de una producción es la parte anterior a la flecha; y se denomina el lado izquierdo de la produccion. Tanto la memoria de trabajo y los lados izquierdos de las producciones son representados internamente por medio de listas del tipo de LISP. Los elementos condición son simples patrónes o predicadosaplicadas a elementos obtenidos por estos patrónes. El apareamiénto trata de encontrar instancias de una clase definida por el patrón entre la lista en la memoria de trabajo, un proceso llamado instanciacion del patrón.

Una produccion esta lista a ser ejecutada cuando todos sus elementos condicion son instanciados. (una excepción a esto se expica mas adelante en el caso de los elementos condición negados). El par ordenado de un nombre producción y la colección de elementos de datos que instancian los elementos condicion de la produccion es llamado una instanciacion. La responsabilidad del apareamiento es encontrar el conjunto de todas las instanciaciónes legales (el conjunto conflicto).

1.3.6 Soluciones al problema

Esta sección hace una revisión de las estrategias existentes para mejorar la eficiencia del apareamiento.

La mayoría de las estrategias involucran algún tipo de esquema de indexamiento. En general, un indice es una funcion de los miembros de un conjunto a miembros de otro. En Fortran, por ejemplo, uno puede escribir la expresión X(3) que producirá que la computadora indexe del entero 3 al numero de punto flotante almacenado en el tercer elemento del vector X. Muchos esquemas de indexamiento involucran aplicaciones sucesivas de más de una funcion de indice. En un lenguaje que permite que los vectores sean almacenados como elementos de otros vectores, por ejemplo uno puede escribir X(3,4), lo cual producirá un nivel de indexamiento de 3 a un vector, y luego otro nivel de 4 al elemento almacenado en la cuarta posicion de ese vector. En contraste a estos dos elementos, la funcion indice usada por el intérprete de sistemas de producción no restringe su conjunto de indices a enteros. El indexamiento realizado en los intérpretes de los sistemas de produccion mapea de una expresión simbolica (una lista, por ejemplo) a elementos de algun conjunto. El interprete generalmente extrae algun aspecto de la expresión (posiblemente tomar el primer átomo de la lista) y luego obtiene una asociacion precomputada de ese aspecto a el conjunto.

Uno de los ejemplos de indexamiento mas simple se encuentra en PSNLST. Cuando un elemento entra a la memoria de trabajo de PSNLST, una faceta del elemento de dato es usado para asociar a todas las producciones que puede aparear a ese elemento. Con cada producción el intérprete mantiene una lista de los elementos de dato que han asociado con él. Comparando las listas el interprete determina que

producciónes están mas cerca de tener instanciaciones, y luego trata de instanciar aquellas producciones primero. Dado a las reglas de resolucion de conflicto de PSNLST, el apareamiento puede terminar despues de que la primera produccion instanciada se encuentra. A pesar de la simplicidad del indexamiento, PSNLST tiene uno de los intérpretes de sistemas de produccion mas rápidos, en gran parte porque el lenguaje fue definido de tal forma que el sistema de indexamiento fuera altamente discriminatorio.

Mcdermon, Newel y Mur investigarón varios sistemas de indexamiento. Uno fué un nivel de un solo esquema similar al de PSNLST. Dos fuerón esquemas de dos niveles en los cuales el segundo nivel de indexamiento usa una combinación de aspectos altamente discriminativa, pero computaciónalmente muy cara. Un cuarto esquema que trataron fue bastante diferente a los otros. Este usaba una red de discriminación, la cual en cada ciclo indexaba del contenido de la memoria de trabajo al conjunto entero de las producciónes que debian ser instanciadas. El intérprete que usaba este esquema era menos eficiente que los otros tres.

Rieguer investigó un esquema que utiliza múltiples niveles de indexamiento; el número de niveles usados para cada elemento condición depende del número de aspectos que ocurren en el elemento. Este sistema intenta obtener una eficiencia adiciónal particiónando la memoria de trabajo. Se requiere que el usuario divida la memoria de trabajo en canales y declare a que canal es sensitivo cada elemento condición.

Ryne ha propuesto una extensión al concepto de indexamiento: construir asociaciones de las acciónes en los lados derechos de la producción a los elementos condiciónes que pueden aparear los elementos de datos agregados o

borrados por las acciónes. Cuando un elemento entra o deja la memoria de trabajo, el interprete recupera la asociación almacenada con la accion responsable, en lugar de computar el indice del elemento. Ese esquema parece que nunca ha sido implementado, y por su descripcion aparece que proveería una cantidad util de discriminación solo si los elementos accion estuvieran predominantemente compuestos por constantes.

Los interpretes que usan indexamiento tienen la propiedad de que despues del paso de indexacion, otra operacion tiene que ser realizada para determinar conclusivamente si las produccciónes estan instanciadas. Esta operacion puede ser eliminada usando un mecanismo que ha sido conocido por muchos años. En 1958 Selfriedche presentó el modelo general para la construcción de reconocedores de patrónes en su maquina Pandemonium. La máquina consiste en una reja de unidades de procesamiento llamadas demons. Los demons en la parte baja de la rejilla continuamente monitorean alguna area, esperando un evento particular. En un sistema de produccion esta área sería la memoria de trabajo, y el evento sería una adicion o una eliminación de algún elemento de dato que poseyera un aspecto particular. Cuando el evento ocurre, el demon toma una nota del hecho y pasa la nota a los demons directamente arriba en la red. Estos demons comparan las notas que recibieron de los varios demons inmediatamente debajo de ellos, y cuando uno de estos demons encuentra un patron interesante en las notas, envia una nota a los demons del siguiente nivel. Cuando uno de los demons en la parte alta de la rejilla encuentra un patron interesante en las notas que recibio, toma una accion mas eficaz que enviar simplemente una nota. En otro interprete de sistemas de produccion, estos demons superiores harian cambios al conjunto conflicto. El sistema ACORN, utiliza una red de demons para analizar un subconjunto del inglès, un proceso muy similar al apareamiento de los intérpretes en

los sistemas de producción.

En 1982 Charles Forgy desarrollo otra version del modelo Pandemonium: el algoritmo de apareamiento de RED. Hasta el momento el algoritmo ha sido utilizado para cerca de diez lenguajes de sistemas de produccion.

Nota:

Pandemonium

Desarrollado en 1959 por O.G Selfridge. PANDEMONIUM fue uno de los primeros intentos en aprendizaje de maquina. Ejemplifica una aproximacion al aprendizaje, atravez de redes nueronales antropomorficas con estructura inicial parcialmente Random. La investigacion actual en aprendizaje de maquina ha abandonado este paradigma. (vease O.G Selfridge. Pandemonium: A Paradigm of Learning. en D. Blake y Uttley (eds), Proceedings of the Symposium of mechanization of Thougth Proceess)

1.4 PROBLEMAS CON LA IMPLEMENTACION DE UN LENGUAJE DE SISTEMAS DE PRODUCCION.

Como lo hemos expresado anteriormente, los sistemas de producción son un formalismo adecuado para la expresión de ciertos problemas dentro de la Inteligencia Artificial, sin embargo, no se habia utilizado como lenguaje de programación, porque el problema de determinar cuales reglas son satisfechas en cada ciclo es extramadamente rudo, a modo de ejemplo supóngase un sistema de 1000 reglas de producción con 3 precondiciones cada una, y supongase que se tiene una base de hechos formada por 1000 elementos, el problema de determinar que reglas se satisfacen en cada ciclo es como sigue: Existen 1000^3 ternas ordenadas que hay que probar contra cada regla, y como hay 1000 entonces hay que hacer

1000 x 1000^3 intentos de aparear patrones en cada proceso de seleccion para determinar que reglas tienen satisfecho su antecedente.

1.4.1 USO DE REDES.

La solucion a este problema exponencialmente complejo,esta en el uso de un algoritmo que salve o guarde los apareamientos exitosos previos, el cual consiste en generar una red, consistituida por un conjunto de nodos que son *estimulados* bajo ciertas condiciónes, los cuales propagan el estimulo tambien dependiendo de ciertas condiciónes.

La acción del algoritmo consiste en compilar las reglas de producción a una red, la cual es usada posteriormente en el proceso de determinacion de las reglas candidatas a ser ejecutádas.

La ventaja de hacer uso de una red en este problema es que la resolución del problema se hace de manera concurrente y en paralelo, dando un esquema de pizarrón (blackboard) primitivo, en el que las reglas son agentes oportunistas con una memoria compartida en el que todas ellas pueden escribir, leer y borrar.en el que el módulo resolvedor de conflictos se convierte en un agente que cuida la jerarquia de los componentes Veamos más detalladamente el algoritmo utilizado.

1.5 ALGORITMO DE APAREAMIENTO

El paso que mas consume tiempo en la ejecucion de un sistema de produccion es la *fáse de apareamiento.*

El algoritmo usado en la implementación de PERITO explota los siguientes aspectos:

(1) Solo una pequeña fracción de hechos cambia en cada ciclo, por lo que es necesario almacenar los apareamiéntos exitosos de ciclos previos, que son usados posteriormente en los ciclos subsecuentes.

(2) La similitud entre los elementos condición de las producciónes, realizando pruebas comunes a la vez, lo que nos da un esquema de resolucion en paralelo.

Estos dos aspectos combinados producen un algoritmo muy eficiente.

La estratégia usada para incorporar las consideraciónes anteriores consiste en compilar los lados izquierdos de las producciónes, *L1*, generando una *red de flujo de datos*, la cual es posteriormente utilizada para realizar los apareamiéntos. Para generar la red para una producción, se comienza con los *elementos individuales* del lado izquierdo. Para cada elemeto condicion se encadenan nodos de prueba que determinen lo siguiente:

Si los argumentos del elemento condición que tienen una constante como su valor, son satisfechos.

Si dos ocurrencias de la misma variable dentro de los elementos condicion estan consistentemente acotadas.

Cada nodo de la cadena realiza alguna de las pruebas anteriores. (A estas pruebas se les denomina de *intra-condición*, debido a que ellas se refieren a los elementos condición individuales). Una vez que el algoritmo ha terminado con los elementos condicion individuales, se agregan nodos que prueben la consistencia de las sustituciónes de las variables entre varios elementos

condición de el lado izquierdo de una misma producción. (A estas pruebas se les denomina pruebas inter-condición, debido a que se refieren a multiples elementos condición.) Finalmente el algoritmo agrega un nodo especial terminal que representa la producción correspondiente a esta parte de la red.

La figura 1 muestra tal red para las producciones p1 y p2, las cuales aparecen en la parte superior de la figura. En esta figura, las lineas han sido dibujadas entre nodos que indican los caminos através de los cuales fluye la información. La información fluye del nodo superior hacia bajo. Los nodos con un solo prodecesor (cerca de la parte superior de la figura) son aquellos que tienen que ver con elementos condicion individuales. Los nodos con dos predecesores son aquellos que determinan la consistencia de las sustituciónes en las variables entre varios elementos condición. Los nodos terminales estan en la parte baja de la figura. Note que cuando los LI requieren nodos identicos, el algoritmo comparte porciones de la red, en lugar de construir nodos duplicados.

```
regla p1
(
c1(X 12),
c2(15 X),
c3(X)
-->
escribe( Hola)
)
```

```
regla p2
(
c2(15 Y),
c4(Y)
-->
escribe(Hola 2)
)
```

Agrega a la memoria de trabajo los siguientes hechos.

```
h1: c1(12 12)
h2: c2(12 15)
h3: c2(15 12)
h4: c3(12)
```

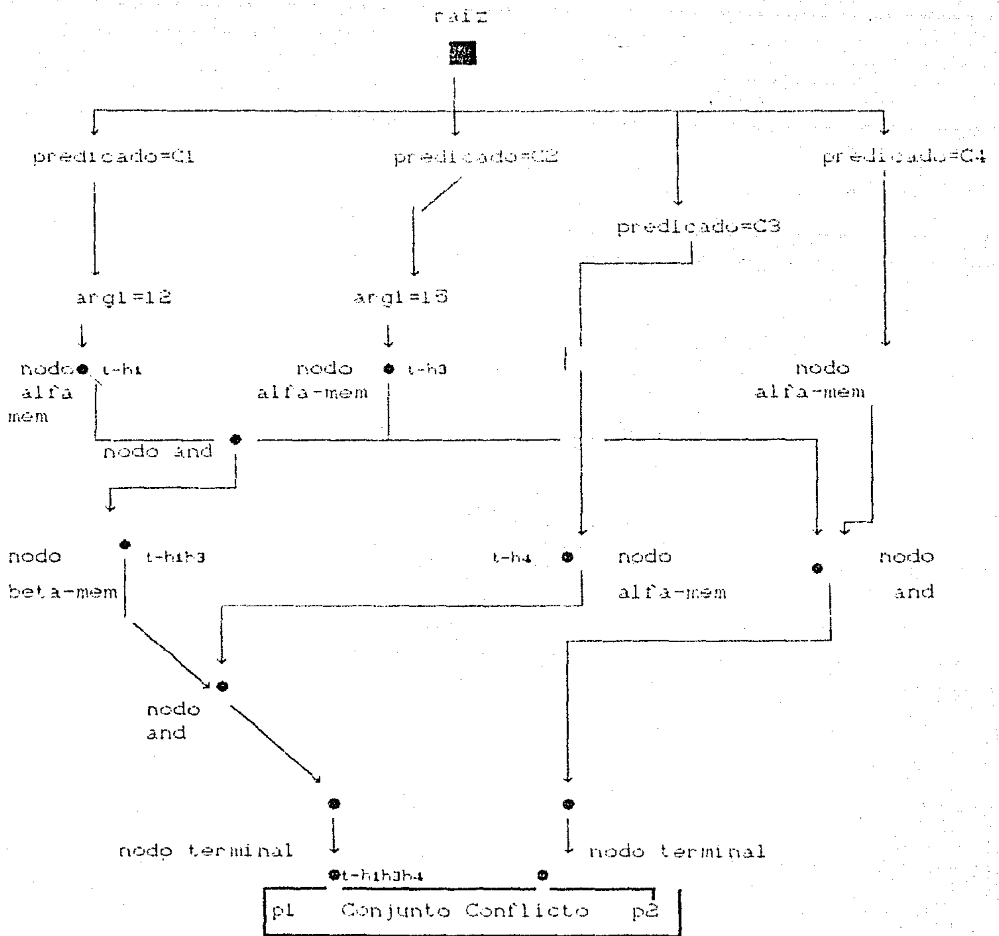


FIGURA 1: Red generada por el algoritmo de red y las producciones p1 y p2

Para evitar realizar la misma prueba mas de una vez, el algoritmo almacena el resultado de los apareamientos anteriores como estados dentro de los nodos. De esta forma, solo los cambios hechos a la memoria de trabajo por las producciones mas recientemente disparadas tienen que ser procesados en cada ciclo. De esta forma, la entrada a la red consta de los cambios hechos a la memoria de trabajo. Estos cambios son filtrados atraves de la red actualizando el estado almacenado dentro de la red. La salida de la red consta de una especificación de los cambios hechos al conjunto conflicto.

Los objetos que son pasados entre los nodos los denominaremos *tokens*, los cuales constan de una marca y una lista ordenada de hechos. La marca puede ser el atomo "VALIDO", la cual indica que algo ha sido incluído en la base de hechos , o un "INVALIDO", indicando que algo ha sido removido de la misma. La lista de hechos asociada con cada token corresponde a la secuencia de aquellos elementos que el sistema esta tratando de aparear, o que han sido apareados contra una subsecuencia de elementos condición en el LI.

La red de flujo de datos producida por este algoritmo consta de cuatro tipos de nodos diferentes. Estos son:

1. **Nodos de prueba-constante:** Estos nodos son utilizados para probar si los componentes de un elemento condicion que tienen un valor constante son satisfechos. Estos nodos siempre aparecen en la parte superior de la red. Tienen una sola entrada, y por consiguiente, son llamados nodos de una sola entrada.

2. **Nodos de memoria:** Estos nodos almacenan el resultado de la fase de apareamiento de ciclos previos como

estados dentro de ellos. El estado almacenado en un nodo memoria consiste de una lista de los tokens que aparean una parte del LI de la produccion asociada. Por ejemplo el nodo mas a la derecha de la figura 1 almacena todos los tokens que aparean el segundo elemento condición de la producción P_2.

A un nivel de mayor detalle, hay dos tipos de nodos de memoria -los nodos de memoria α y los nodos de memoria β-. Los nodos de memoria α almacenan los tokens que aparean elementos condición individuales. Por lo que todos los nodos memoria inmediatamente abajo de los nodos constantes son nodos de memoria α. Los nodos β almacenan los tokens que aparean una secuencia de elementos condiciones en el LI de una producción. Así pues, todos los nodos inmediatamente debajo de los nodos de dos entradas son nodos de memoria β.

3. **Nodos de dos entradas:** Estos nodos prueban la satisfacción conjunta de elementos de condición en el LI de una produccion. Ambas entradas de los nodos de dos entradas vienen de nodos de memoria. Cuando llega un token a una de las entradas, es comparado con cada token almacenado en el nodo de memoria conectado en la otra entrada. Todos los token que tiene una sustitucion de variables consistente son enviados a los sucesores de los nodos de dos entradas.

Existen dos tipos de nodos de dos entradas - los nodos AND y los nodos NOT. Mientras que los nodos AND son responsables de las condiciones positivas de los elementos condición y se comportan de la manera descrita anteriormente, los nodos NOT generan un token a su sucesor(es) si no hay tokens que apareen con el nodo memoria correspondiente a el elemento condición negado.

4. **Nodos terminales:** Hay un sólo nodo asociado con cada

produccion en el programa, como puede ser visto en la parte baja de la figura 1. Cuando un token fluyó hacia un nodo terminal, la produccion correspondiente es o incertada o borrada del conjunto conflicto.

El siguiente ejemplo provee una vision mas detallada del procedimiento que se sigue dentro de la red. El ejemplo corresponde a las producciónes y la red dada en la figura 1. Se muestra el proceso de apareamiento conforme los cuatro hechos de la parte inferior izquierda son secuencialmente añadidos a la memoria de trabajo.

Cuando el primer hecho es agregado, el token t-h1,

<+,c1(12 12)>

es construido y enviado al nodo raiz de la red. El nodo raiz distribuye el token a todos sus sucesores. La prueba asociada falla para todos los sucesores excepto para aquel que está intentando probar "¿ predicado=C1 ?". Este nodo de prueba constante, pasa el token al siguiente nodo que trata de probar si "arg1=12". Puesto que esto es asi, el token es pasado al nodo memoria, el cual almacena el token y pasa una copia del token a el nodo AND debajo de él. El nodo AND compara el token entrante en su entrada izquierda con los tokens en su memoria derecha (la cual en este punto esta vacia), por lo que ningun par puede ser formado. En este punto la red se ha estabilizado -en otras palabras ninguna actividad adicional ocurre-, por lo que veamos que le ocurre al segundo hecho.

El token para el segundo hecho, t-h2,

<+,C2(12 15)>

es construido y enviado al nodo raiz, el cual reparte el token a sus sucesores. El token pasa la prueba 'predicado=C2', pero posteriormente falla la prueba 'argumento1=15', por lo que ningun procesamiento adicional

toma lugar.

El token para el tercer hecho, t-h3.

<+,C2(15 12)>

pasa a traves de la prueba 'predicado=C2' y 'arg1=15', y es almacenado en el nodo de memoria debajo de ellos. El nodo memoria pasa una copia del token a los dos sucesores y los nodos AND debajo de el. El nodo AND en la izquierda prueba la consistencia del token contra el token, t-h1, almacenado en su memoria izquierda. La prueba de consistencia es satisfecha por lo que la variable X está acotada consistentemente.

El nodo AND crea un nuevo token, t-h1h3,

<+,c1(12 12), C2(1512)>

y lo pasa hacia abajo, a el nodo memoria debajo de él. El nodo AND encuentra que su memoria izquierda está vacía, de tal forma que ningún procesamiento adicional toma lugar.

Al añadir el cuarto hecho, t-h4.

<+,C3(12) >

es enviado al nodo raiz, el cual se encarga de repartirlo a sus sucesores. El token pasa la prueba 'predicado=C3' y pasa al nodo memoria debajo de el. El nodo memoria almacena el token y pasa una copia a el nodo AND debajo de él. El nodo AND prueba el acotamiento (binding) consistente en la memoria del lado izquierdo y encuentra que el token que recién arrivo, t-h4, es consistente con el token t-h1h3 almacenado en su memoria izquierda. El nodo AND entonces crea un nuevo token, t-h1h3 almacenado en su memoria izquierda. El nodo AND entonces crea un nuevo token, t-h1h3h4.

< +,C1(12,12),C2(15,12),C3(12) >

y lo envia al nodo terminal correspondiente a pi, abajo. El nodo terminal entonces inserta la instanciación de pi correspondiente a t-h1h3h4 en el conjunto conflicto. Por lo que tenemos una produccion en el conjunto conflicto que puede ser disparada.

LOGROS

PERITO pertenece a una familia de lenguajes al que pertenece OPS5, lenguaje desarrollado por Mcdermont, Allan Newell, Forgy entre otros, en la universidad de Carnegie Mellon. La implementacion en Franz Lisp de su intérprete corre a una velocidad promedio de 8 cambios de la base de hechos por segundo (lo que según ellos mismos reportan, equivale a 3 disparos por segundo) en un a VAX-11/780, mientras que la versión basada en Bliss ejecuta a una velocidad de 40 cbh/s lo que equivale a 14 disparos por segundo.

El resultado obtenido hasta ahora indica que perito trabaja a una velocidad de 30 disparos por segundo en una IBM AT.

C A P I T U L O
1

CAPITULO 1

2.0 INTRODUCCION.

Como lo hemos estado mencionando PERITO es un lenguaje basádos en el modelo de sistemas de produccion. En el presente capitulo se presentara el lenguaje y se vislumbrara el poder declarativo de PERITO, a la vez que se hara evidènte la naturaleza no secuencial de este lenguaje, especificamente, PERITO es un lenguaje no determinista, cuya ejecucion se efectua concurrentemente y en momentos, en paralelo.

2.1 EL LENGUAJE DE PROGRAMACION PERITO.

PERITO es un lenguaje de programación especialmente diseñado para el desarrollo de aplicaciónes en inteligencia artificial y muy particularmente para el desarrollo de sistemas expertos. En este capitulo se presentan los elementos de programacion de PERITO, no se pretende aqui hacer una presentacion de los sistemas expertos, para lo cual remitimos al lector interesado a la bibliografia correspondiente.

Los elementos básicos de PERITO son:

1. La memoria de trabajo. La memoria global de los datos.

2. La base de conocimientos, o memoria de producción la cual contiene a todas las reglas.

3. El motor de inferencia.

Un programa en PERITO consta de un conjunto de hechos y reglas. El motor de inferencia determina cual de las reglas es la que va a ser disparada o activada. Como lo hemos estado mencionando PERITO es un lenguaje dirigido a datos, esto es, los hechos son los datos que estimulan la ejecucion. En PERITO los hechos son indispensables para producir la ejecucion de las reglas y por lo tanto juegan un papel fundamental.

La memoria de trabajo esta formado por un conjunto de elementos de memoria. Un objeto con la siguiente sintaxis denota a los elementos de memoria, esto es, los datos sobre los cuales operan las producciònes :

```
Elemento de memoria = palabra(Lista_de_argumentos)
Lista_de_argumentos= Argumento* [1]
Argumento=          Palabra |
                      Cadena  |
                      Numero
```

ejemplos:

```
persona(jose, av_universidad_12,profesor).
sintoma(fiebre)
```

Es importante hacer notar que a cada elemento de memoria, o hecho, se le asocia un *identificador de hecho*. A cada hecho se le asocia un entero único que indica la **novedad**, estos enteros son asignados en orden creciente.

[1] El simbolo " * " denota una sucesion de uno o mas elementos. Asi pues el argumento* denota un sucesion no vacia de de argumentos.
El simbolo "+" denota sucecion de cero o mas elementos.

esto es, conforme mas reciente séa la presencia del hecho en la memoria de trabajo mayor sera el entero que se le asociara. Es importante notar que en la memoria de trabajo no puede haber duplicados de hechos, porque antes de ser insertado se verifica la existencia del mismo en la ME. Los identificadóres de hecho, aunque se asignan de manera ascendente, no se asignan necesariamente secuencialmente.

Es importante hacer notar que PERITO elimina todos los espacios inecesarios como son los tabs,blancos etc.etc. Y que ademas es sensitivo a mayúsculas y minusculas, esto es las distingue.

En realidad un eleménto de la memoria de trabajo es traducido internamente a una lista del tipo LISP de la siguiente manera:

```
palabra(Lista_de_argumentos)      ----> (palabra argumentos)
```

se ha escogido la primera representación para que el usuário utilice el primer campo como nombre de la relacionentre los otros elementos.

La memoria de produccion esta formada por una colección no ordenada de proposiciones *Si-entonces* llamadas producciónes. Los datos sobre los cuales operan las producciones estan contenidos en una base de datos global llamada la *memoria de trabajo*. Por convencion, la parte del *Si* de una producción se denomina lado izquierdo (en adelante *LI*), y la parte del *entonces* se denomina el lado derecho (*LD*). El interprete de PEPITO ejecuta un sistema de producción realizando las siguientes operaciónes.

(1) *Apareamiento*. Evalua el LI de las producciónes para

determinar cuales son satisfechas dados los contenidos de la memoria de trabajo en ese momento.

(2) *Resolucion de conflictos.* Selecciona una producción de las producciones activadas. Si ninguna produccion esta activada termina la interpretación.

(3) *Actua.* Realiza las acciónes del lado derecho de la producción seleccionada.

(4) Ve al inciso 1

El LI de una producción consiste en una secuencia de patrónes (ver capitulo anterior).

En algunas producciónes, algunos patrónes van precedidos por el simbolo de negación *no(PATRON)*. Un lado izquierdo (LI) es satisfécho cuando

(1) Cada patrón que no es precedido por un *no* aparea con un elemento de la memoria de trabajo, y

(2) Ningun patron que es precedido por un *no* aparea un elemento de la memoria de trabajo.

Es importante notar que la condición anterior implementa un tipo de negación por fracaso, lo cual es una versión restringida de la suposicion del mundo cerrado de Reiter que fue introducida originalmente para base de datos.

Un patron que contiene unicamente constantes se apareara con un eleménto de la memoria de trabajo si cada constante en el patrón aparece en la posición correspondiente en el elemento de la memoria de trabajo.

El lado derecho de una producción consiste en una secuencia de instrucciones, la cuales seran ejecutadas secuencialmente. Algunas de estas instrucciones afectan el contenido de la memoria de trabajo. Más adelante se mencionaran estas, cabe mencionar las basicas que son la que sirve para agregar un elemento a la memoria de trabajo y que se denomina *afirma* y otra que elimina de la memoria de trabajo algunos eleméntos memoria (hechos) y que se denomina *quita*.

El motor de inferencia de PERITO, directamente impleménta una estratégia de encadenamiento hacia adelánte, aunque es posible implementar una estrategia de solucion de encadenamiento hacia atras (backward-chaining). PERITO no impone restriccion al tipo de aplicación que se quiera escribir. Como resultado, el programador tiene que manejar explicitamente tanto la representacion de los datos como el flujo de control por medio de reglas.

1.2 Tipos de datos definidos en el lenguaje.

1.2.1 Tipos de datos primitivos.

Dos tipos de datos primitivos, llamados **tipos escalares**, son usados en PERITO: **numeros** y **atomos simbolicos**. Los tipos de numeros que utiliza perito son los de punto flotante, de hecho todo numero es coercionado a su equivalente en punto flotante.

1.2.1.1 Atomos simbolicos.

Un atomo simbolico es una secuencia de algunos caracteres que no es un numero y que PERITO tratara como una unidad. Existen dos tipos de atomos simbolicos: los identificadores y las cadenas.

Un identificador en PEPITO tiene la misma sintaxis que los identificadores en pascal; esto es, un identificador es una secuencia de caracteres alfanumericos que comienza con un caracter alfabetico y que no contiene blancos. Es importante señalar que PEPITO distingue mayusculas de minúsculas. Asi pues HOLA y hola son identificadores distintos.

2.3 Hechos o elementos de la memoria de trabajo.

En un programa en PEPITO, los datos son colectados y mantenidos en la <u>memoria de trabajo</u> o <u>memoria de hechos</u>. Cada unidad de la memoria de trabajo es una secuencia de simbolos con la sintáxis previamente descrita.

Es importante remarcar que internamente a cada hecho se le ascoia un valor entero llamado **etiqueta del tiempo**, o **atributo de llegada**, o **identificador de hecho**. Este valor entero indica cuando el elemento fue introducido por primera vez en la memoria de trabajo. Durante mayor sea el numero entero, quiere decir que el elemento es mas reciente. La etiquéta del tiempo es usada por el motor de inferencia en la fase de resolucion de conflictos. No puede ser accesada o modificada por el programa directamente.

2.4 MEMORIA DE PRODUCCION

Las reglas de PEPITO tienen un nombre unico, un lado izquierdo que es una secuéncia de uno o mas **elementos condicion**, y un lado derecho que es una secuencia de **acciones**. Cada elemento condición especifica un patrón que va a ser usada para aparear contra la memoria de trabajo; cuando todas las condiciónes sean satisfechas simultaneamente, un apareamiento o unificación para el lado izquierdo como un todo a ha tenido lugar, y la regla se dice que ha sido instanciada. Las acciones que conforman el

lado derecho son declaraciones del tipo imperativas que van a ser ejecutadas en secuencia cuando la regla sea instanciada (es decir cuando vaya a ser disparada).

Y tienen la siguiente sintaxis:

```
regla <Nombre>
 ( LI
    -->
   LD
 )
```

en donde

 LI = patrones*
 LD = accion*

es importante notar, que si en un programa hay dos reglas con el mismo nombre solo la ultima será la que estará en la memoria de producción.

Perito intenta aparear los patrones de la regla contra los hechos en la MT. Si todos los patrones de la regla se pueden aparear con algunos hechos, la regla es **activada** y es puesta en la **agenda**. La agenda es la colección de reglas activadas.

Un punto importante de PERITO es que una vez que un hecho es utilizado para aparear un patrón esto no es "visto" nunca mas por el mismo patrón. Este mecanismo es indispensáble para evitar caer en loops infinitos. Por lo que si se quiere utilizar el mismo hecho para estimular los mismo patrónes es necesario, primero quitar ese hecho y volverlo a agregar. Esto es para cambiar el *identificador de hecho*. Este es uno de los usos del mencionado identificador.

Esta caracteristica en el diseño de PERITO es similar a la caracteristica de las celulas nerviosas, llamadas neuronas. Despues de un que una neurona transmite un impulso nervioso (se dispara), ninguna cantidad de estimulación hara que se vuelva a disparar por algun tiempo. Este fenomeno se denomina **refraccion** y es una caracteristica fundamental en los sistemas expertos.

Sin la **refracción** los sistemas expertos caerian en LOOPS triviales. Esto es, tan pronto una regla es disparada, se mantendria disparandose sobre los mismos hechos una y otra vez. En el mundo real, el estimulo que causa el disparo eventualmente desaparecera. Por lo que repetimos, si uno quiere volver a disparar la misma regla uno tiene que quitar y volver agregar los hechos que la dispararón. Basicamente PERITO recuerda los *identificadores de hecho* que activarón a una regla y no activará a esa regla otra vez con la misma combinacion de identificadores de hechos. Claro esta, una regla puede ser puesta en la agenda con los mismos identificadores de hechos muchas veces. Sin embargo, el ser puesta en la agenda no significa que vaya a ser disparada. Solo aquellas reglas que han sido disparadas "experimentan" refraccion, no las reglas en la agenda.

1.5 ELEMENTOS CONDICION O PATRONES.

Un elemento de condicion es un patrón que especifica las restricciónes del hecho con el cual puede ser apareado. Un hecho aparea con un elemento condicion solo en el caso de que se cumplan todas la restricciónes descritas en el elemento condicion. Un caso simple es el siguiente,

por ejemplo:

persona(marta).

este elemento condicion apareara con cualquier elemento de la base de datos que sea de la misma forma.

Es importante señalar que los argumentos de elemento condicion son las restricciones que deben de ser satisfechas por todo aquel hecho que aparee con el. Siendo la restriccion mas particular la constante, donde el programador debera saber su valor de antemano. Sin embargo, existen otros formas de expresar aquellos valores que pueden aparear en una posición dada. Entre ellas podemos menciónar a las siguientes:

1. <u>Variables anónimas</u>

2. <u>Variables con nombre</u>

3. <u>Variables restringidas a un conjunto dado</u>

4. <u>Valores que satisfagan una fórmula proposiciónal especificada.</u>

Una <u>variable anonima</u> en PERITO se denota por la raya denominada 'underscore', esta es " _ " y se utiliza para denotar que cualquier objeto puede aparear con ella y que además no interesa el valor del objeto con el cual se aparee.

Ejemplos de elementos condición que tiene una variable anonima:

Patrón del lado izquierdo	Hecho en la lista	Aparea
padre_de(_ juan)	padre_de(pedro juan)	si

padre_de(_ juan)   padre_de(juan luis)   No

En el caso de una variable con nombre en PERITO su sintáxis es la siguiente :

Es un identificador tipo pascal que comienza con una mayuscula.

Este tipo de argumento puede aparear con cualquier objeto y además se crea una asociación (un binding) del objeto con el que se apareo y el nombre de la variable, esta asociación una vez establecida debera de ser consisténte con todas las otras ocurrencias de la misma variable dentro del mismo elemento condicion o dentro de los otros elementos condición dentro de una misma regla de producción.

Ejemplo:

Patron del lado izquierdo	Hecho en MT	Apareamiento.
padre_de(X juan)	padre_de(pedro juan)	X/pedro
padre_de(Y Z)	padre_de(juan luis)	Y/juan,Z/luis

En el caso de <u>variable restringida a un conjunto</u> nos referimos a una variable que satisface la condicion lógica que define al conjunto, de hecho en el caso de PERITO, aquella expresión lógica es una fórmula proposiciónal unicaménte. La sintáxis para una <u>variable restringida a un conjunto</u> es la siguiente:

<<variable> : <formula proposiciónal>>

ejemplo:

(Nombre: pedro | juan)

La cual usada en un elemento condicional quedaria de la siguiente manera:

```
padre_de(juan <Nombre (pedro | juan)>)
```

en donde dicho elemento condicional tendria el siguiente significado:

Determina si hay elemento de la memoria de trabajo cuyo primer componente sea *padre_de* cuyo primer argumento sea *juan* y cuyo segundo argumento sea *pedro o juan* y si lo hay asocia *Nombre* con el objeto con el que hayas apareado .

En el caso de <u>valores que satisfagan una formula proposicional</u> lo que se expresa es de hecho la fórmula proposiciónal, el objeto que vaya a unificar con ese argumento debe de satisfacer aquella formula.

ejemplo:

```
padre_de(juan ~luis)
```

queremos indicar que el segundo argumento puede unificar con cualquier cosa, siempre y cuando sea distinta de luis.

Existen tres operadores lógicos para restringir los valóres que puede tener un argumento. Estos tres operadores son suficientes para generar todos los demas operadores binarios.

Esto son:

operador	**sintaxis**
y	&
o	|
not	~

Como deciamos estos operadores puden combinarse en cualquier manera o numero para generar cualquier formula proposicional que restringa los valores que un argumento puede tomar, o el conjunto de valores al que puede pertenecer el objeto que vaya a ser apareado durante la fase de apareo de patrones.

Es importante señalar que la evaluacion se hace de izquierda a derecha.

Ejemplos:

<table>
<tr><th>Elemento condicion</th><th>Hecho en la base</th><th>Aparea</th></tr>
<tr><td>padre(juan ~ luis)</td><td>padre(juan pedro)</td><td>Si</td></tr>
<tr><td>padre(juan ~luis)</td><td>padre(juan luis)</td><td>No</td></tr>
<tr><td>padre(juan (luis|pedro))</td><td>padre(juan pedro)</td><td>Si</td></tr>
<tr><td>padre(juan (luis|moy|pedro))</td><td>padre(juan pedro)</td><td>Si</td></tr>
<tr><td>padre(juan (X:~luis))</td><td>padre(juan pedro)</td><td>Si</td></tr>
</table>

además el valor de "X" en el ultimo ejemplo tiene la asociación "X/pedro".

Ahora bien en el ejemplo anterior, si "X" hubiéra tenido un valor previamente asignádo, se considera un restricción extra que debera de satisfacer la variable "X".

O B E R V A C I O N: SI USTED TRATA DE USAR UNA VARIABLE NO ASIGNADA (UNA QUE NO TENGA UN VALOR QUE RESULTA DE UN APAREAMIENTO) EN EL LADO DERECHO DE UNA REGLA SE INDICARA LA UN MENSAJE DE ERROR Y SE TERMINARA LA EJECUCION DEL PROGRAMA.

2.6 DEFINICION DE REGLAS

El componente básico de un programa en la filosofía de sistemas de producción es la regla, que de hecho puede verse como un procedimiento que es invocado cuando todas las precondiciones necesarias, han sido satisfechas.

Una regla en PERITO consta de varias partes: encabezado,lado izquierdo, y lado derecho.

La sintáxis es la siguiente

```
regla <nombre_de_la_regla>
 {
   [<prioridad>]
    <lado izquierdo>
     -->
     <lado derecho>
  }
```

En donde <nombre_de_la_regla> es un identificador.

ejemplo:

```
 regla escribe_abuelos
 {
  padre_de(X Y),
  padre_de(Y Z)
  -->
  escribeln(Z, " es abuelo")
 }
```

1.7 DECLARACION DE LOS HECHOS INICIALES

Como mencionabamos anteriormente, PERITO es un lenguaje de reglas que tratan de ser activadas dependiendo de los hechos en la memoria de trabajo, por lo que es importante contar con una forma de dar de alta hechos en la memoria de trabajo, por lo cual PERITO cuenta con la construccion denominada **hechos**, y tiene como finalidad agrupar un conjunto de hechos y asociarles un nombre, de tal forma que nos podamos referir a ese conjunto de hechos unicamente por su nombre.

La sintaxis de esta construcion es la siguiente:

```
hechos <nombre>
        {
         < hecho>
         [  <,hecho>* ]
        }
```

Ejemplo:

```
hechos relaciónes_familiares
          {
           padre_de(juan luis),
           padre_de(pedro juan),
           padre_de(maria juan)
          }
```

Cuando uno comienza la ejecución de un programa en PERITO, todos los hechos son introducidos en la base de hechos. Es importante hacer notar que el uso de esta construcion es opciónal en cuanto que un programa no necesariamente tiene hechos en su memoria de trabajo al comenzar la ejecución. Dado esto parece surguir una pregunta importante, dado que no hay hechos iniciales

¿ Como es que comiénza la ejecución de un programa?, porque siguiendo nuestro esquema descrito anteriormente debe de existir al menos un hecho con el cúal pueda comenzar la ejecucion. La solución es simple, se ha provisto por default,siempre que se comienza la ejecucion de un programa, un hecho distinguido que se denomina **hecho_inicial**, y que se encontrara siempre en la memoria de trabajo, de tal forma que uno puede tener reglas que unicamente tengan como precondición la existencia del **hecho_inicial** y estas tendran su precondicion satisfecha y por lo tanto seran candidatas a ser disparadas.

2.8 COMPARACIONES

Ademas de las restricciónes de argumento mencionadas anteriormente, existen otro tipo de precondiciones que nos permitiran comparar las asociaciones de las variables encontradas para determinar si satisfacen mas restricciónes adicionales. Entre los tipos de comparaciones que podemos hacer son las matematicas (por ejemplo determinar si la suma de dos variables es mayor que alguna tercera etc. etc) y tambien se pueden hacer comparaciónes logicas mas complejas por ejemplo:

$$X+Y<Z \;\#\; X+Z>Y.$$

A continuación se da una lista de las funciónes predefinidas y que pueden usarse:

Logicas

Funcion	Sintaxis	Uso
not	!	lógico
and	&	lógico
or	#	lógico

Comparasiones

Función	Significado
=	igualdad
/=	desigual
>=	mayor o igual a
<=	menor o igual
<	menor que
>	mayor que

ARITMETICO

/	division
*	multiplicacion
+	suma
-	resta

ejemplo:

La siguiente regla determina si la diferencia entre X y Y es menor que 2

```
    regla  ejemplo
  {
    temperatura(hoy X),
     temperatura(ayer Y),
      X-Y>2
     -->
      escribe("la temperatura del paciente ha variado mucho")
    }
```

2.8 PRIORIDAD

Resulta claro de las explicaciones anteriores que varias reglas pueden dispararse simultaneamente (en paralelo) de tal forma que para determinar que regla se disparara (se ejecutara) debera de pasarse a una fase de resoluciónde conflictos. Para sesgar la elección de una regla sobre otras, podra asignarse a cada regla una prioridad que sera un numero que indique la predileccion de esa regla, esto es cuando existan varias reglas en la agenda, aquella con la mayor prioridad es la que se disparara primero.

Uno de los problemas fundamentales con el uso de la prioridad es la tendencia a abusar de la misma de tal forma que se quiera programar secuencialmente, que es completamente contrario a la filosofia de la programación basada en reglas.

Sintáxis:

```
prioridad( <numero> )
```

dónde numero es un entero entre -1000 y 1000. Cuando la declaracion de la prioridad es omitida por default se asume que la prioridad de la regla es cero.

Veamos un ejemplo en dónde el uso de la prioridad es deseable para controlar la ejecución:

Suponga que tiene un programa para controlar a un robot que trata de cruzar la calle. Puede haber muchas reglas que son satisfechas por los hechos en un situacion determinada. Puede haber un hecho, por ejemplo *luz(verde)* mientras que puede haber un policia que indique alto, aun cuándo la luz

este en verde.

```
   regla luz_verde
    {
      luz( verde )
-->
      escribeln( "Avanza")
}

   regla policia
     {
       policia( alto)
       -->
       escribeln( " Alto")
     }

hechos condiciones
   {
    luz(verde),
    policia(alto)
    }
```

Al correr este programa se encuentra con que las dos reglas se van a disparar, dando una respuesta válida pero incorrecta, el problema es que ambas tienen igual prioridad, prioridad 0, por lo que es nesesario ponderar la segunda régla y esto se hace a travez de la declaracion de prioridad. Ademas para prevenir que la primera regla se disparé uno tiene que quitar el hecho que la dispararia.

De lo anterior tambien se desprende que a travez del uso de la prioridad uno puede crear partes que se ejecutén secuencialmente.

2.9 El lado izquierdo de las reglas de PERITO

El lado izquierdo de las reglas en PERITO ha sido definido como una secuencia de elementos condición separados por comas que indica la conjuncion de condiciones que deben de ser satisfechas.

Los elementos condicion pueden de ser de varios tipos:

1). El que hemos visto hasta ahora.

2) El elemento condiciónvariable que se define a continuacion:

Definimos un <u>elemento variable</u> como una asignaciondе la forma:

(Elemento condicion usual) ^ <Variable>

y el significado de tal asignacion es que el hecho (elemento memoria) que aparee con el <u>elemento condicion usual</u>, sea asignado como un todo a la variable llamada *<Variable>*.

El proposito de tal asignacion es comunicarse con el lado derecho de la regla. El lado derecho utilizara esta variable asignada como una indicaciondе que elemento de la base de hechos ha tenido un apareamiento exitoso con el elemento condicion. El alcance de un elemento variable es unicamente la regla en la que esta siendo usada. Es importante señalar que esta variable es muy diferente de las anteriores, porque esta es acotada con una direcciondе memoria, de la base de hechos, donde se encuentra el hecho con el que patron apareo, mientras que las otras variables definidas anteriormente aparean unicamente con tipos escalares.

Ejemplo:

padre_de(juan luis) ^ Variable

El significado es el siguiente: si hay un hecho en la base de datos que aparee con *padre_de(juan luis)*, asigna la dirección de la base de datos donde se encuentra a *Variable*.

De esta forma, *Variable* tiene la direccion donde se encuentra el elemento con el que hubo apareamiento, de esta forma el lado derecho de la regla podra, mediante una instruccionprovista para ello (que se vera mas adelante),quitar o modificar ese hecho (elemento memoria) de la memoria de trabajo.

2.10 ELEMENTOS NEGADOS

Ahora bien los elementos condició n del LI pueden estar negados, teniendo la siguiente sintaxis:

no(<Elemento condicion>)

ejemplo:

no (padre_de(juan luis))

En dónde esta expresión es satisfecha en el caso de que no haya en la memoria de trabajo un hecho con el que padre_de(juan luis) pueda unificar. En otras palabras, se utiliza *negacion por regla de fallo.*

2.11 EL LADO DERECHO DE LAS REGLAS DE PERITO.

El lado derecho de las reglas de PERITO consisten en una secuéncia de instrucciónes imperativas que van a ser ejecutadas en el orden en que aparécen, vamos a clasificar las acciones por el tipo de acción que realizan:

Modificacion de la memoria

Condicionales

Iteración

Asignacion

Entrada y salida

Terminación de programa

2.11.1 ACCIONES DE MODIFICACION DE LA MEMORIA DE TRABAJO

Afirma

Una de las acciones mas importantes en un programa en PERITO es la de inclusion de nuevos hechos en la memoria de hechos, y esto lo hacemos a travès de la instrucción **afirma** siendo su sintàxis la siguiente;

```
afirma(HECHO)
```

En donde el HECHO en caso de tener variables debe de tener todas sus variables acotadas. Si una copia del hecho existe previamente en la base de hechos, el hecho no sera incluido.

Veamos un ejemplo del uso de esta instrucción en una regla

```
regla tiene_gripa
 {
  temp(alta),
  mucosidad(alta)
  -->
  afirma(enfermdedad(gripa) )
 }
```

o utilizando variables previamente acotadas

```
regla cierra_valvula
 {
  temp(X alta),
  valvula(Nombre abierta)
   -->
   afirma(pon(Nombre cerrada) )
```

}

Quita.

Es claro que siendo PERITO un lenguaje dirigido a datos es necesario contar con una operacion que nos permita quitar hechos de la MT. Antes de que un hecho pueda ser quitado, este debe de ser especificado. Para extraer un hecho desde una regla, el *identificador de hecho* debe de ser acotado a una variable en el LHS.

Existe una gran diferencia entre acotar una variable a los contenidos de un hecho y acotar una variable a la *direccion del hecho*. La dirección del hecho es asignada mediante un *elemento variable*, visto anteriormente. Esto es la direccion del hecho a ser removido es especificada utilizando el operador " ^ " a la derecha del patron con el que el hecho se pretende aparear.

Una vez que se conoce la dirección, se usa la instruccion **quita** para quitar el hecho de la memoria de trabajo, cuya sintaxis es la siguiente :

quita (<**VARIABLE** de un elemento variable>)

Solo un elemento memoria puede quitarse con una instrucción *quita*. La accion **quita** tiene el efecto de no solo quitar el hecho denotado por la variable , si no que remueve a todas las reglas que se encuentren en la agenda que dependieron del hecho para su activacion.

veamos un ejemplo:

```
regla cambia_estado_paciente
 {
   temp(Nombre alta)^Var1,    pon(temp baja)^Var2
```

```
--> quita(Var1):quita(Var2):afirma(temp(Nombre_baja)) }
```

Veamos un ejemplo de como disparar una misma regla mas de una vez. Como deciamos PERITO tiene el mecanismo de refraccion para evitar caer en loops triviales. Por lo que, deciamos, para disparar una regla mas de una vez se deben de modificar los *identificadores de hechos* de los hechos que la activan, y como hemos visto esto se hace a travez de quitar y poner hechos. Veamos el siguiente programa que, cuya ejecucion sera la escritura "infinita" del mensaje correspondiente.

```
regla extasis
{
   nombre(Nombre Apellido) ^ Persona
    -->
    escribeln( Nombre, " ", Apellido,  " es una persona feliz);
    quita(Persona);
    afirma( nombre( Nombre Apellido))
  }
```

Defina algun hecho en la MT usando la declaracion de hechos y compruebe la ejecucion infinita de este programa.

Afirma_cadena

afirma_cadena es una acción que toma como argumento una cadena y lo convierte al formato de un hecho (elemento memoria), rompiendo la cadena usando las blancos como los delimitadores de argumento.

Sintaxis:

```
afirma_cadena(cadena)
```

De igual manera que con **afirma** si existe una copia del hecho que se pretende meter, el hecho no sera incluído. Es importante hacer notar que una cadena puede tener una subcadena entrecomillada siempre y cuando las comillas interiores vayan precedidas por un slash '\'. Esto va a ser particularmente util con la función de lectura **leelinea**.

Veamos algunos ejemplos:

instrucción	hecho afirmado
afirma_cadena("estado juan grave")	estado(juan,grave)
afirma_cadena("luz \"roja\"")	luz("roja")

Añade_regla y quita_regla.

Estas acciones son de suma importancia, debido a que un sistema experto puede descartar ciertas reglas durante su proceso de inferencia, a la vez que puede incluir otras que antes no se encontraban. Es claro que un programa con aprendizaje requerira este tipo de acciones.

En esta version de PERITO (version 1.0) éstas acciones no están disponibles, pero lo estarán en las siguientes.

Condicionales

Una construcción imperativa fundamentál es la del condicional, que nos permite tomar acciones dependiendo del resultado de una condición lógica, en PERITO tenemos la construcción *Si entonces else*, o la de *Si entonces*.

Con la siguiente sintáxis:

```
Si <condicion_logica>
    entonces
        <accion>
        [ <;accion>+ ]
    [else
    <accion>   [ <;acción>+ ]     ]
finif
```

Cualquier número de acciones puden ser usadas dentro de la sección **Si entonces else** del condicional, inclusive otro condicional. Como se ve en la sintáxis en BNF la parte **elsees** opcional. Todo condicional debe de terminar con un **finif** que demarca el alcance del mismo.

Iteraciones

Otra construccion imperativa importante es la iteracion, que en este caso tambien es provista en PERITO, en este caso en forma de un **while** tipo pascal, la sintáxis es la siguiente:

```
mientras <condición>  haz
    <accion>
     <;acción>*
    fin
```

Todo tipo y número de acciones pueden ponerse dentro del cuerpo de un mientras, incluído otros **mientras** y **si entonces else**. Como en pascal la prueba de la condición es hecha antes de ejecutar la iteración.

Asignacion

Frecuentemente es necesario crear nuevas variables o modificarles el valor a las ya definidas, por esto en PERITO

contamos con una instruccion que nos permite realizar las dos funciones mencionadas anteriormente, este comando se denomina asignacion y su sintaxis es la siguiente:

<Variable> := <VALOR>

<Variable> := <funcion predefinida>

en donde variable debe de ser el nombre de una variable (que pudo haber sido definida previamente) en donde valor puede ser una valor constante, otra variable que haya sido previamente acotada o puede ser el resultado de llamar a una funcion predefinida.

ejemplos:

```
Entrada:=lee
Valor:= X+Y
Viejo:=Nuevo
Variable:=10
```

en el primero de los ejemplos se deja el resultado de la funcion predefinida **lee** , en la variable denominada Entrada.

Funciónes de entrada y salida.

Entre las primeras funciónes de entrada tenemos a **abre**. que tiene como efecto abrir un archivo y asignarle un alias. Toma dos argumentos:

1) El nombre del archivo a abrirse.

2) El alias.

ejemplo:

```
abre("c:\\experto.exp",mi_experto)
```

el alias no debe de haber sido utilizado antes.

Otra funcion de E/S es la de **cierra** que es el complemento de la anterior, es decir cierra el archivo abierto con el comando abre:

Sintaxis:

```
cierra(<alias>)
```

ejemplo:

```
cierra(mi_experto)
```

Otra instrucción es la de **cierra** sin argumentos , y su acción es la de cerrar todos los archivos que esten abiertos hasta el momento.

Lee

Además del apareamiento de patrónes **PERITO**, existe otra forma de obtener información desde el exterior, por medio del teclado, para este proposito contamos con la **funcion lee.** Esta funcion es equivalente a un readln en pascal. Es importante hacer notar que el resultado de la funcion tiene que ser asignado a una variable mediante la instruccion de asignación.

Ejemplo:

```
regla lee_información
      {
       hecho_inicial
           -->
      escribeln("Dame un nombre entre luis/juan/pedro");
      Nombre:=lee;
      afirma( nombre(Nombre) )
```

```
    }

regla verifica_entrada1
    {
     nombre( {Nombre:juan | luis | pedro} ) ^ Hecho
    -->
    quita(Hecho);
    escribe(" Bien hecho ")
    }


regla verifica_entrada2
    {
    hecho_inicial  Variable
     not(nombre( juan | luis | pedro) ))
    -->
    quita(Hecho);
    escribe(" Te equivocaste ")
    }
```

Este miniprograma es iniciado con la primera regla puesto que el hecho_inicial siempre se encuentra en la memoria de trabajo al comenzar la ejecución de un programa. La intencion de este programa trivial, es la de recibir información del teclado, incluirla en un hecho y despues tratar de verificar que la información fue la que se pedia.

Una restriccion importante que hay que señalar es que la función **lee** solo puede leer un tipo escalar a la vez, es decir un número, un identificador o una cadena.

Leeln

Si uno quiere leer varios valores a la vez, contamos con una función para ello, **leelinea**, que lee todos los valores en una linea y los convierte a una cadena. Ahora bien si uno quiere convertir la cadena leeida a un hecho es necesario utilizar la instrucción, **convierte_afirma**.

COMANDO DE TERMINACION DE EJECUCION DE UN PROGRAMA.

Mencionabamos al principio de este capitulo que el flujo de control de un programa en el modelo orientado a reglas de produccion no siempre es del todo claro, tampoco es claro en que momento el programa terminara, sin embargo hay situaciones en que dadas ciertas circunstancias nos gustaria terminar la ejecucion del programa, para esto proveemos a PERITO de un comando que una vez que es interpretado produce la terminacion del programa.

El comando en cuestión tiene la siguiente sintáxis:

alto

es decir es un comando sin argumentos.

ELIMINACION DE HECHOS EN BLOQUE

La acción denominada **quita_hechos** quita todos los hechos que hayan sido incluidos en una declaración de hechos. la sintáxis es la siguiente:

quita_hechos(*nombre*)

en dónde **nombre** es el nombre de la declaración *hechos* que se quiere quitar.

1.14 EJEMPLOS DE PROGRAMACION CON PERITO.

Ejemplo 1:

Veamos un programa para generar el sucesor en una sucesión de numeros hasta un limite dado.

```
regla lee_limite
    {
```

```
                hecho_inicial
                -->
                escribe("Numero de iteraciones? ");
                Limite:=lee;
                afirma( itera(0  Limite));
               }


regla escribe_sucesor
                {
                itera(Cuenta Maximo),
                Cuenta <= Maximo
                -->
                Sucesor:=Cuenta+1;
                 Cuenta:=Cuenta+1;/* incrementa el contador */
                 escribeln(Sucesor);
                 afirma( itera(Cuenta,Maximo))
                }
```

En este caso el hecho itera lleva el contador de los números que se han escrito y tambien lleva el número de maximo de iteraciones . Este programa funciona, sin embargo los hechos que una vez son usados se tornan inutiles y consumen memoria, un buen ejercicio consite en quitar los hechos una vez que se vuelven inutiles.(hint Utilize la instrucción quita en cada en la regla).

Ejemplo 2:

```
   regla suma
   {
     numeros(X Y)
     -->
     Res:=X+Y;
     escribeln("La suma de ",X, " y ", Y, " es ", Res)
    }
```

declare algunos hechos del tipo

```
    hechos prueba
      {
```

```
    numeros(2 3),
    numeros(5 6)
   }
```

Ejemplo 3:

Veamos un ejemplo de como obtener informacion mas haya del apareamiento de patrónes con el uso de la accion lee.

```
regla lee_entrada
    {
       empieza()
       -->
       escribeln(" DAme el nombre de un color primario ");
       Resp:=lee;
       afirma ( color(Resp))
     }

regla intenta_otra_vez
 {
  empieza()^ H1.
  respuesta(si) ^ H2.
  -->
  quita(H1);
  quita(H2);
  afirma( empieza())
  }
regla no_repita
  {
    repuesta( ~si) ^ H1
    -->
    quita(H1)
  }


regla checa_entrada
{
     color( rojolamarillolazul ) ^ Patron
     -->
     quita( Patrón);
     escribeln(" Tu respuesta fue correcta felicidades")
 }


regla checa_entrada_2
{
```

```
    color( ~rojo &~amarillo &~azul)^Patron
    -->
    quita( Patron);
    escribeln(" tu respuesta fue incorrecta Quieres intentarlo
                    otra vez (si/no) ");
     Resp:=lee;
    afirma(respuesta(si))
 }
```

Observece en este caso como es que interactuan las reglas, en dónde unas proven los datos que otras necesitan. En este caso producimos un comportamiento de ciclo. Este es un metodo bastante comun para producir ciclos en sistemas expertos. Es claro que como el flujo no es secuencial usted debe de programar la correcta interaccion de las reglas cuando usted quiere un comportamiento secuencial.

Ejemplo 4:

Veamos un uso de las comparaciones en el LI de la regla.

Suponga que quiere hacer un programa para generar el cuadrado de unos numeros hasta un cierto limite. El siguiente programa hace uso de la comparacion en el LI para determinar cuando se debe de parar la impresion de los cuadrados.

```
 regla lee_maximo
   (
     hecho_inicial
     -->
     escribe(" Numero de loops ");
     Limite:=lee;
     afirma( loop(0 Limite))
    }

regla escribe_cuadrados
   (
    loop( Contador Limite),
    Contador <= Limite
```

```
-->
Cuadrado:=Contador * Contador;
escribeln(" El cuadrado de", Contador, " es ", Cuadrado);
Contador:=Contador+1;
afirma( loop(Contador Limite))
}
```

Recordatorio: Tenga presenta que la acción lee unicamente lee un campo.

Ejercicio 1:

Desarrollar un miniexperto que auxilie a una mujer convencional de principios de siglo, a elegir marido, un ejemplo de regla que podria tener su experto es la siguiente;

```
regla  el_soltero_ideal
      {
      cantidad_de_dinero(mucha Nombre),
      edad(viejo Nombre)
      -->
       escribe("El ideal a atrapar es ", Nombre)
      }
```

RESUMEN DE LA SINTAXIS DE PERITO

Acontinuacion se presenta un resumen de la sintáxis de PERITO, en dónde se usa la siguiente notación.

Cuando despues de un nombre aparezca un asterisco seindica que es una sucesión de uno o mas eleméntos.

Cuando despúes de un nombre aparezca un " + " indica una sucesión de cero o mas eleméntos.

El caracter " | " indica un "o".

Y si despúes del "+" o del "*" aparece la palabra <u>separador</u> se indica que es una sucesión en donde es nesesario usar un separador entre los eleméntos de la misma.

```
PROGRAMA=LISTA_REGLAS LISTA_HECHOS

LISTA_REGLAS=REGLA+

LISTA_HECHOS=LISTA_DE_HECHOS*


LISTA_DE_HECHOS =hechos
                   {
                         LISTA_DE_ELEMENTOS_DE_MEMORIA
                   }


REGLA=             regla PALABRA
                         {
                         IZQ --> DERECHA
                         }


LISTA_DE_ELEMENTOS_DE_MEMORIA=
                              ELEMENTO_DE_MEMORIA*

ELEMENTO_DE_MEMORIA=PALABRA ( LISTA_DE_CONSTANTES )

LISTA_DE_CONSTANTES=CONSTANTE*
```

```
CONSTANTE=
                    PALABRA
                    CADENA
                    NUMERO

HECHO=  PALABRA ( LISTA_ARGUMENTOS )   ;
        hecho_inicial


LISTA_ARGUMENTOS= ARGUMENTO* separador coma

ARGUMENTO=
                    PALABRA
                    CADENA
                    NUMERO
                    VARIABLE
                    COMPUESTO


VARIABLE=
                    VARIABLE_ANONIMA
                    VARIABLE_SIMPLE


COMPUESTO=            VALOR
                    (  VARIABLE : VALOR  )


VALOR=              VALOR  #  VALOR
                    --
                    VALOR  &  VALOR
                    --
                    PALABRA
                    CADENA
                    NUMERO
                    ( VALOR )


CHECA=  CHECA or_ CHECA
        --
        CHECA and_ CHECA
        --
        EXP  <  EXP
        EXP  >  EXP
        EXP  <= EXP
        EXP  >= EXP
        EXP  <> EXP
        EXP   = EXP
```

```
        ! CHECA
        ( CHECA )


EXP=
        EXP  +  EXP
        EXP  -  EXP
        --

        EXP * EXP
        EXP / EXP

        --

        VARIABLE_SIMPLE
         - EXP
        NUMERO
        ( EXP )


PRIORIDAD=        prioridad( ENTERO_PRIORIDAD)


ENTERO_PRIORIDAD = ENTERO        /*  -10000 < ENTERO <10000 */


ASIGNA_PATRON=  HECHO ^  VARIABLE_SIMPLE


HECHO_NEGADO=   no(HECHO)


PRECONDICIONES= PATRONES+ separador coma


PATRON  =        ASIGNA_PATRON
                 HECHO_NEGADO
                 HECHO
                 CHECA


IZQ=             PRIORIDAD ,PRECONDICIONES
                 PRECONDICIONES

DERECHA=         ACCIONES+ separador semicolon


ACCIONES=         afirma (  HECHO )

                  quita ( VARIABLE )
                 convierte_afirma( CADENA )
                 VARIABLE_SIMPLE := DER_ASIGNA
                 alto
```

CAPITULO 3

3. EL ALGORITMO DE APAREAMIENTO DE RED.

Este capitulo describe el algoritmo utilizado en la implementacion de PERITO. Este algoritmo es una variante del algoritmo de RED desarrollado por Charles Forgy en su tesis doctoral en la Universidad de Carnegie Mellon. Las primeras cuatro secciones tratan de motivar el algoritmo. Estas secciones explican cómo es que el algoritmo trata de ganar eficiencia. La siguiente sección contiene una descripción concisa del algoritmo desarrollado en las primeras cuatro secciones.

3.1 INTRODUCCION AL ALGORITMO

Esta seccion introduce el algoritmo de apareamiento, describiendo las propiedades de los sistemas de producción que el algoritmo trata de explotar, y se explica la organizacion básica del algoritmo para ayudar a explotarlas. Estas propiedades de los sistemas de producción son llamadas redundancia temporal y similitud estructural.

3.1.1 Redundancia Temporal:

En la mayoria de los sistemas de produccion, los cambios hechos a la memoria de trabajo son realmente muy lentos de ciclo a ciclo. Los tamaños de la memoria de trabajo típicamente están en un rango de 15 a 500 elementos. El disparo de una produccion típicamente cambiará únicamente de 2 a 5 elementos. Este hecho es importante en el diseño de un algoritmo de apareamiento porque la mayoria de la información usada por el apareamiento en un ciclo podria ser usado por el apareamiento en el siguiente ciclo. Los cambios realizados por la produccion que sea disparada puede

provocar que menos producciones sean instanciadas y que algunas otras pierdan sus instanciaciones, pero la mayoria de las producciones ni perderan ni ganaran ninguna instanciacion. Más aún, cualquier produccion que llegue a estar instanciada probablemente estaba cerca de estarlo en los ciclos anteriores, posiblemente tenia todas las instanciaciones, excepto tal vez para algun elemento condicion. La rutina de apareamiento puede sacar ventaja de esto almacenando de un ciclo al siguiente una indicacion de qué producciones estan instanciadas y cuales elementos condición de las otras producciones están instanciados, actualizando de manera creciente la información para reflejar los cambios hechos a la memoria de trabajo. Si esto se hace así, el esfuerzo requerido para realizar el apareamiento dependerá mas del pequeño numero de elementos que cambiaron que en el numero mas grande de elementos en la memoria de trabajo.

3.1.2 SIMILITUD ESTRUCTURAL

Los lados izquierdos de un sistema de producción usualmente tienen una gran cantidad de similitud. Elementos condición identicos frecuentemente ocurren o aparecen en diferentes producciones. Diferentes elementos condicion pueden tener longitudes idénticas o pueden contener constantes identicas. los primeros tres elementos condicion de EJEMPLO1, por ejemplo, son idénticos a los tres primeros elementos condicion de EJEMPLO2.

```
regla EJEMPLO1
  (
  meta( agarra  Obj),
  objeto( Obj  Lugar  ligero  techo  _  _),
  no( meta( mover  escalera  Lugar))
  -->
  afirma( meta( mover  escalera  Lugar))
  )
```

```
regla EJEMPLO2
  (
  meta( agarra  Obj),
  objeto( Obj  Lugar  ligero  techo  _  _),
  objeto( escalera  Lugar   _  piso  _  _),
  no( meta( sobre  escalera))
  -->
  afirma( meta( sobre  escalera))
  )
```

Si la rutina de apareamiento puede reconocer la ocurrencia de aspectos idénticos, puede evitar hacer la misma prueba varias veces. Después de determinar, por ejemplo, que EJEMPLO1 no tiene ninguna instanciacion legal, no gastará el tiempo tratando de instanciar EJEMPLO2.

3.1.3 COMPILACION DE LOS LADOS IZQUIERDOS

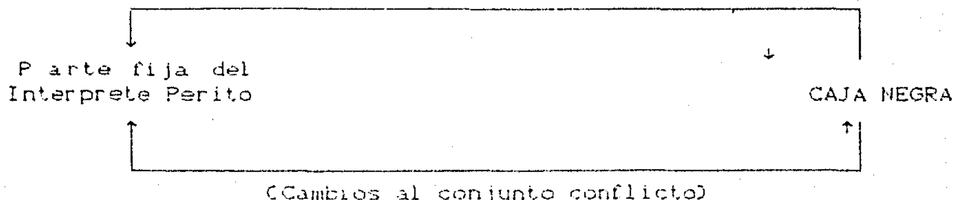
Para sacar ventaja de la similitud estructural, un intérprete que utilice el algoritmo de aparemiento deberá procesar los lados izquierdos de un sistema de producción antes de comenzar la ejecución del sistema. Un paso de preprocesamiento es necesario porque es en este paso en el que las similitudes son encontradas.

El paso de preprocesamiento es una compilación que construye una parte del intérprete. Un interprete que utilice el algoritmo de red es incompleto hasta que el conjunto de producciones es compilado. El intérprete contiene programas para realizar la resolución de conflictos, y para ejecutar los lados derechos de las producciones; pero no contiene programas para interpretar los lados izquierdos de las producciones y para realizar el apareamiento. Del sistema de producción a ser ejecutado, el compilador de los lados izquierdos construye un programa para realizar los apareamientos para ese sistema de producción en particular. Puesto que el programa construido

por un compilador de lados izquierdos contiene toda la informacion que es necesaria para realizar el apareamiento. los lados izquierdos pueden ser desacartados despues de que es construido. La mayor parte de este capitulo esta dedicada a la descripcion de la estructura de estos programas y de la forma en que el control es pasado entre sus partes.

3.1.4 INTRODUCCION AL ALGORITMO DE APAREAMIENTO

Posiblemente el aspecto más inusual de una rutina de apareamiento basada en el algoritmo de apareamiento de red es que nunca examina la memoria de trabajo. En lugar de ello monitorea los cambios hechos a la memoria de trabajo y mantiene información interna que es equivalente a aquella en la memoria de trabajo. Al principio de un ciclo la rutina de apareamiento nota los cambios hechos a la memoia de trabajo en el ciclo anterior. De esta información la rutina de apareamiento computa si algún cambio debe ser hecho al conjunto conflicto. Si hay cambios que hacer, se envía una lista de los cambios a la parte fija del interprete, donde se mantiene al conjunto conflicto. Con el estado de la memoria de trabajo constantemente cambiando, una producción no puede permanecer satisfecha indefinidamente, por lo que los cambios al conjunto conflicto incluye remover las viejas instanciaciones como insertar las nuevas. Para el resto del intérprete, la rutina de apareamiento parece ser una caja negra con una sola entrada y una sola salida:



3.1.5 TIPOS DE CAMBIOS EN LA MEMORIA DE TRABAJO QUE SE REALIZAN

Para mantener este capitulo lo mas breve posible, asumiremos aqui que solo dos tipos de cambios se pueden hacer a la memoria de trabajo: añadir un elemento y borrar un elemento. Estos cambios son escogidos debido a que ellos comprenden un conjunto completo minimo del cual todas las otras acciones pueden obtenerse.

La modificación de elementos, por ejemplo, que se provee en muchos lenguajes, puede ser simulada por un simple borra y pon. La acción de modifica puede aparecer en el lado derecho de una producción como si ésta fuera soportada directamente por el apareamiento. Cuando una instruccion de modifica es ejecutada, una rutina en el intérprete (invisible para el usuario) la intercepta y la convierte en las operaciones primitivas de borrar y luego añadir.

3.2 ELEMENTOS CONDICION NO NEGADOS

La descripción del algoritmo sera más fácil si los elemento condición negados (aquellos que deben satisfacer la regla de fallo), son postergados a una sección posterior.

3.2.1 PRODUCCIONES CON UN SOLO ELEMENTO CONDICION

Los programas de apareamiento construidos por el compilador de lados izquierdos son redes tipo Pandemonium de simples reconocedores de facetas debido a que esta organización ayuda a sacar ventaja de la similitud estructural. Como lo demostraremos mas adelante, es particularmente fácil localizar una estructura común en

ESTA 'S NO DEBE
SALIR DE LA BIBLIOTECA

estas redes y eliminarlas. Los detalles de estos programas serán ilustrados compilando varias producciones. Las producciones mas simples, aquellas con un solo elemento condicion, son consideradas primero.

La produccion EJEMPLO3 es un ejemplo de este tipo de reglas de produccion.

```
regla EJEMPLO3
  {
   meta(mono cerca X)
   -->
   afirma(meta(mono sobre piso)
  }
```

que por cierto, es representada internamente de la siguiente manera, en una estructura LISP.

Traducción interna:

```
(regla ejemplo1 ( (meta (mono cerca X)) --> (afirma (meta (mono sobre p
```

La definicion de lenguaje de PERITO especifica como el lado izquierdo de esta producción va a ser interpretado. El lado izquierdo puede ser instanciado por cualquier elemento en la memoria de trabajo que satisfaga lo siguiente:

1. Tiene la constante meta como su primer subelemento.
2. Tiene una lista de tres argumentos como su segundo elemento.
3. Tiene la constante mono como su primer argumento.
4. Tiene la constante cerca como el segundo argumento.

Los nodos en la red prueban la ocurrencia de aspectos como estos en los elementos de datos.

Cada uno de los nodos tiene un solo vertice hacia él y

uno o más vértices que parten del el. Los elementos de dato son enviados a un nodo privilegiado en la parte superior. Una vez que llegan los elementos de dato, el nodo es activado para probar la presencia de un aspecto particular. Si la prueba tiene éxito el elemento de dato es enviado a lo largo de cada vértice que parte del nodo. Si la prueba falla se termina la exploracion por esa rama de la red. Puesto que la regla de producción de EJEMPLO3 tiene cuatro aspectos que probar, existen cinco nodos en su red. Estos nodos estan arreglas en una secuencia lineal; la salida de cada nodo es la entrada del el siguiente:

↓

Es el primer elemento meta?

↓

Es el segundo subelemento
una lista de tres subelementos?

↓

Es el primer subelemento
del segundo subelemento
mono?

↓

Es el segundo subelemento
del segundo subelemento cerca?

↓

Ninguna entrada para el primer vértice o ninguna salida para el último se muestra debido a que los nodos no prueban elementos de dato. Simplemente conectan esta red dentro de la caja negra.

El nodo que provee las entradas para el primer nodo arriba recibe el nombre de <u>bus entrada</u> de la caja negra.

Este nodo acepta una descripcion de los cambios hechos a la memoria de trabajo y distribuye la informacion a cada primer nodo en la red de cada elemento condición. La red contiene solo un bus de entrada sin importar el número de producciones en el sistema.

El último vértice en la secuencia de arriba se conecta a un nodo que esta encargado de mantener al conjunto conflicto. Cuando este nodo del conjunto conflicto recibe un elemento de dato, envia el elemento de dato y el nombre la producción a las rutinas que mantiene el conjunto conflicto. Puesto que los nodos de este tipo tienen que saber los nombres de las producciones, un nodo por separado tiene que ser construído para cada producción en el sistema.

La red completa para EJEMPLO3 es

Distribuye las descripciónes
de los cambios en la memoria
de trabajo.

Es el primer elemento meta?

Es el segundo subelemento
una lista de tres subelementos?

Es el primer subelemento
del segundo subelemento
mono?

Es el segundo subelemento
del segundo subelemento cerca?

Reporta que la produccion
ejemplo1 ha sido satisfecha.

3.2.2 LOS DATOS PROCESADOS POR LOS NODOS

Los paquetes enviados entre los nodos los llamaremos <u>tokens</u> o <u>estafetas</u>. Cada token tiene dos componentes, una parte de etiqueta y una parte de dato. La parte de dato del token contiene un elemento de la memoria de trabajo o una secuencia de elementos de la memoria de trabajo. El uso de la parte de etiqueta no puede ser explicado hasta que se describan más tipos de nodos. En los nodos del tipo mostrado hasta ahora, la parte de etiqueta contiene solo una indicación del tipo de cambio que ha sido hecho a la memoria de trabajo (con la parte de dato conteniendo el elemento de dato afectado). Si solo dos tipos de acción van a ser manejados por el apareamiento, solo se requerirán dos etiquetas, un signo '+' para elementos recien añadidos a la memoria de trabajo y un signo '-' para elementos que se acaban de borrar de la memoria. En este capitulo los tokens serán presentados como pares ordenados. La primera parte del par es la etiqueta y la segunda parte es la parte de datos.

EJEMPLO.

```
< +,  meta(agarra  escalera) >
< +,  meta(come  platanos)   >
< +,  meta(sobre   escalera) >
```

3.2.3 PROCESANDO LA RED PARA EJEMPLO3

Suponga que EJEMPLO3 se encuentra en programa en el que los siguientes hechos son incluidos;

```
meta(agarra  platanos)
meta(come  platanos)
```

Cuando estos entran a la memoria de trabajo el interprete construye tokens con etiquetas '+', y el primer nodo en la red pasa copias de los tokens a sus sucesores inmediatos.

Es el primer subelemento meta?

Este nodo prueba que el primer subelemento en la parte de dato, y puesto que es *meta* pasa el token a sus sucesores.

Es el segundo subelemento una lista de tres subelementos?

Este nodo prueba la longitud de su segundo subelemento y pasa el token a sus sucesores.

Es el primer subelemento del segundo subelemento mono?

Este nodo muestra que el primer subelemento del segundo subelemento, y encuentra que es mono y pasa el token a sus sucesores.

Es el segundo subelemento del segundo subelemento cerca?

Y este nodo rechaza el token, prueba el segundo subelemento del segundo subelemento y encuentra que vale a *agarrar* el lugar de *cerca*. Por lo que no envia nada a su sucesor, y el procesamiento de este token en la red de EJEMPLO3 se termina. El resto de los tokens procesados son rechazados más rápido que el primero. El token

<+, mono(cerca,baul)>,

por ejemplo es rechazado después de que se le aplica la primera prueba.

En otra parte de la red, sin embargo, otros nodos puden aceptar estos tokens, y el conjunto conflicto puede tener instanciaciones añadidas. Al ejecutarse estas

instanciaciones, resulta que nuevos elementos se añaden a la memoria de trabajo y viejos elementos son borrados. Ninguno de los tokens creados para representar estos cambios pasan completamente a traves de la red de EJEMPLO3 hasta que el token < +, meta(mono cerca) > es añadido. El token pasa todas las pruebas en la red de EJEMPLO3 y causa que el último nodo añada una instanciación de EJEMPLO3 al conjunto conflicto.

3.2.4 PRODUCCIONES CON DOS ELEMENTOS CONDICION

Un LHS con dos elementos condición es compilado en una red que contiene dos secuencias lineales de nodos (uno para cada elemento condicion) más un nodo que combina los tokens pasados por estas dos secuencias. Esta sección esta avocada en describir este ultimo nodo. Por motivo de simplicidad, se comienza considerando los lados izquierdos en los cuales los elementos condición no comparten variables.

La produccion de EJEMPLO4 es un ejemplo tipico de las producciones consideradas en esta sección.

```
regla EJEMPLO4
   (
   meta(vacia_mano  mono),
   mono(agarra  X)   -->
   escribe("El  mono  tira  suelta""  X)
   )
```

que internamente es representado de la siguiente manera;

Traducción interna:

```
(regla ejemplo2
 ( (meta (vacia_mano mono X )) (mono (agarra ))    --> (escribe "El
mono suelta" X)))
```

El primer elemento condicion es instanciado por

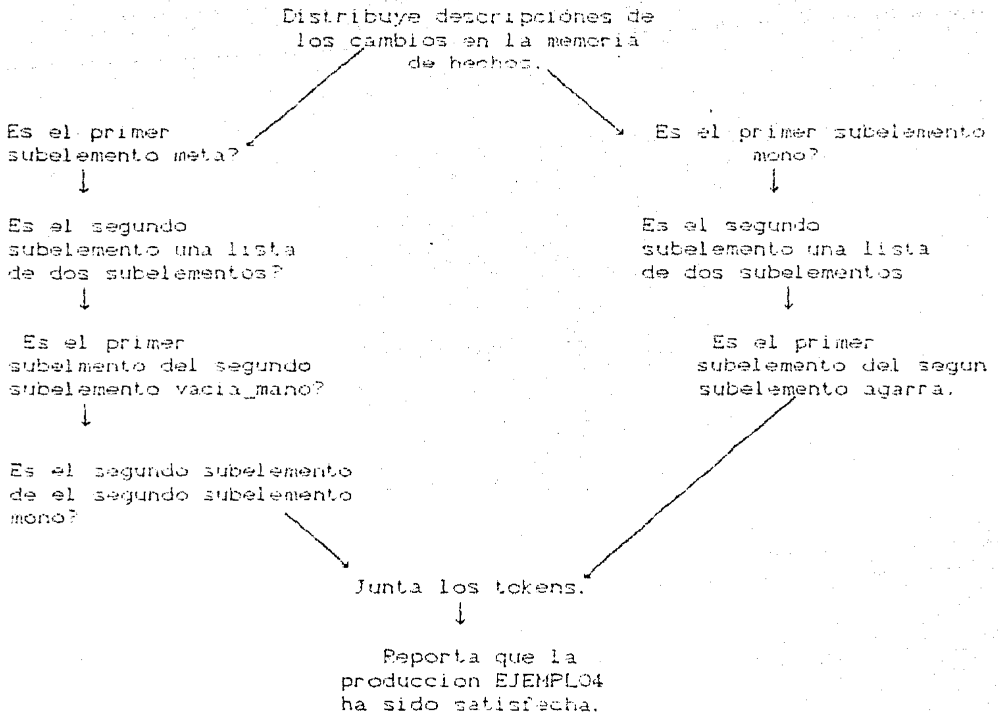
cualquier elemento de datos que satisfaga las siguientes condiciones

1. Tiene a la constante meta como su primer subelemento.
2. Tiene una lista de dos subelementos como su segundo subelemento.
3. Tiene la constante *mano_vacia* como su primer subelemento del segundo subelemento.
4. Tiene la constante mono como el segundo subelemento de su segundo subelemento.

El segundo elemento condición es instanciado por cualquier elemento de dato que satisfaga lo siguiente:

1. Tenga la constante mono como su primer subelemento.
2. Tiene su segundo subelemento longitud 2.
3. Tenga la constante agarrar como su primer subelemento de su segundo subelemento.

Como la red para EJEMPLO4 cada uno de estos aspectos es compilado en una secuencia de nodos por separado.



La responsabilidad del nodo con dos entradas es juntar pares de elementos de datos - un elemento de la rama izquierda de la red y uno de la derecha - en una lista de dos elementos. Puesto que los elementos usualmente no llegan simultáneamente, si es que va a realizar esta función, debemos almacenar el estado de una activación. Supóngase como por ejemplo, que un sistema de producción esta corriendo, pero que ningun elemento de dato se encuentra en la memoria de trabajo que aparee con alguno de los dos elementos condición. Entonces el nodo de dos entradas en la red en EJEMPLO4 no ha sido activado, y por lo que no se mantiene ningún estado. En algún tiempo durante la ejecución, el elemento *meta(agarra,escalera)* entra a la

memoria de trabajo. El token

<+.*meta(agarra,escalera)*>

es procesado, y se las arregla para pasar todos los nodos en la rama derecha de esta red. El token llega a la entrada derecha del nodo de dos entradas y se detiene. Puesto que ningun token ha sido recibido tokens por la otra entrada, no se pueden construir listas. Para prepararse para una posible llegada de un token en su entrada izquierda en un tiempo futuro, se almacena en un memoria interna a si mismo que dice "He recibido el token <+,*meta(agarra,escalera)*> en mi entrada derecha". EL nodo no hace mas en esta activacion. El procesamiento del sistema continua, y eventualmente la meta entra a la memoria de trabajo. El token

<+,meta(vacia_mano,mono)>

es procesado por la red, pasando atraves de la rama izquierda de la red de EJEMPLO4, y llega finalmente a la entrada izquierda del nodo de dos entradas. El nodo examina su memoria, y encuentra la nota que se hizo anteriormente, y construye el token

< +,*meta(vacia_mano,mono) mono(agarra,escalera)* >.

Este token es enviado al último nodo donde causa que una nueva instanciacion de EJEMPLO4 sea añadida al conjunto conflicto. Antes de que el procesamiento del nodo de dos entradas termino, agrega una nota a su memoria interna, "He recibido el token <+,meta(vacia_mano,mono) en mi entrada izquierda". Esto sera necesario si recibe otro token sobre su entrada derecha.

3.2.5 ELIMINACION DE ELEMENTOS DE LA MEMORIA DE TRABAJO

Cuando los elementos son borrados de la memoria de

trabajo, la rutina de apareamiento debe actualizar tanto el conjunto conflicto como aquellos nodos memoria que hallan almacenado al elemento. Los ejemplo de procesamiento en la red de **EJEMPLO3** (seccion 3.2.3) muestran como las etiquetas son usadas en la actualización del conjunto conflicto. Esta sección muestra como son usados en la actualización de los nodos memoria. En particular, muestra como el uso de datos etiquetados permite la actualización de la memoria de los nodos sin examinar cada nodo en el sistema.

Considere el siguiente ejemplo

```
regla ejemplo_t
   (
   meta(vacia_mano  mono)`Var,
   mono(agarra  X)
    -->
   escribe("El  mono  tira  suelta"  X);
   quita(Var)
   )
```

Despues de que el nodo de dos entradas de ejemplo_t termina el procesamiento del segundo token, ha almacenado

<+,*meta(vacia_mano mono)*>

recibido por la izquierda y

<+,mono(agarra escalera)>

por la derecha. ejemplo_t posiblemente se disparara después de la segunda activación de este nodo. Cuando lo hace, borra *meta(vacia_mano mono)* y *mono(agarra escalera)* . Supóngase que los tokens resultantes son procesados en este orden. Entonces

<-,*meta(vacia_mano mono)*>

pasa através de la rama izquierda de la red y llega a la entrada izquierda del nodo de dos entradas. La marca o etiqueta "-" de este token causa que el nodo de dos entradas borre la nota que hizo anteriormente. El nodo de dos entradas entonces construye un nuevo token bastante similar al que construyó cuando un token valido o '+' arribo; toma el elemento *mono(agarra escalera)* de su memoria derecha y los une al elemento de dato que recién ha arribado para producir el siguiente token

<-,*meta(vacia_mano mono) mono(agarra escalera)*>.

Este token es enviado al ultimo nodo y produce que la instanciación de *EJEMPLO4* sea ramovida del conjunto conflicto.

Uno puede preguntarse por qué el nodo de dos entradas etiquetó al nodo que construyó con la etiqueta del token que recien arribó ('-') en lugar de la etiqueta del token en su memoria ('+'). Para contestar esto es necesario entender la razón por la cual el token se construyó. El token

<-,*meta(vacia_mano mono)*>

no pudo haber arribado en la entrada izquierdad del nodo a menos de que el token

<+,*meta(vacia_mano mono)*>

halla arribado anteriormente, esto es un elemento de dato no puede ser borrado de su memoria de trabajo a menos de que esté presente. Puesto que habia un nota en la memoria del nodo de que el token

<+,*mono(agarra escalera)*>

había llegado en su entrada derecha, tiene que ser verdad que, en algún momento en el pasado, el nodo ha producido como salida el token

<+, *meta(vacia_mano mono) mono(agarra escalera)* >

Este nodo fue construido porque, en el momento de su construcción, era necesario añadir en su parte de datos a las memorias que seguían a este nodo. Pero después del arribo de

<-,*meta(vacia_mano mono)* >

no seguía siendo válido mantener la parte de datos en las memorias. El nodo tuvo entonces que crear el token

<-,*meta(vacia_mano mono) mono(agarra escalera)* >

para informar a sus sucesores que la situación cambió.

En otras palabras, en términos de redes neuronales, lo que estamos haciendo es enviar un impulso inhibitorio al resto de la red.

Después del procesamiento del token resultante de la eliminación de *meta(vacia_mano mono)*, el apareamiento procesa el que resulta de la eliminación de *mono(agarra escalera)*:

<-, *mono(agarra,escalera)* >

Este token pasa por la rama derecha de la red y llega a la entrada derecha del nodo de dos entradas. Este nodo es

excitado para buscar en su memoria y borrar la nota, "He recibido el token <+,*monoCagarra escalera)* > sobre mi entrada derecha". Puesto que el nodo había borrado la nota cerca del único token válido que arribó desde su izquierda, no produce salida.

3.2.6 COMO LOS NODOS DE DOS ENTRADAS MANEJAN LAS ETIQUETAS

Para hacer un resumen de la sección anterior, los nodos de dos entradas observan las siguientes tres reglas para manejar las etiquetas.

1. Cuando un token que llega esta marcado como válido ('+'), almacena ese token en su memoria interna.
2. Cuando un token llega y esta marcado como inválido('-'), borra de su memoria interna un token con una parte de datos idéntica y
3. Cuando un token de salida es creado, se copia la etiqueta de el token que recién ha llegado.

Algunos nodos de dos entradas prueban los elementos de datos antes de construir los tokens que van a ser enviados a sus sucesores (ver sección 3.2.8).

3.2.7 LA MEMORIA INTERNA DE LOS NODOS DE DOS ENTRADAS

Las notas en los ejemplos anteriores ("He recibido el token en mi entrada derecha") son más grandes de los que se requiren. Ellas muestran a los nodos de dos entradas almacenando tres piezas de información acerca de cada token: la etiqueta del token, la parte de datos del token, y una indicación de si ha arribado por su izquerda o por su derecha. Puesto que, como lo muestran las reglas para manejar etiquetas, solo los tokens marcados como validos son almacenados, las etiquetas pueden omitirse. La indicación de

si el token llegó por su entrada izquierda o derecha no puede ser omitida, pero puede ser sacada fuera del nodo para ahorrar espacio. Cada nodo puede tener dos memorias en lugar de una. Los tokens que arriban por la izquierda pueden ser almacenados en una memoria, y los tokens que arriban por la derecha en otra. Por lo que solo la parte de datos de los tokens necesita ser almacenada explicitamente.

3.2.8 LAS VARIABLES QUE OCURREN EN MAS DE UN ELEMENTO CONDICION

Esta seccion comienza con la discusion de las redes para producciones en las cuales las variables aparaecen más de una vez. La produccion EJEMPLO5 por ejemplo, con la variable X ocurriendo en sus dos elementos condiciones, es un ejemplo tipico.

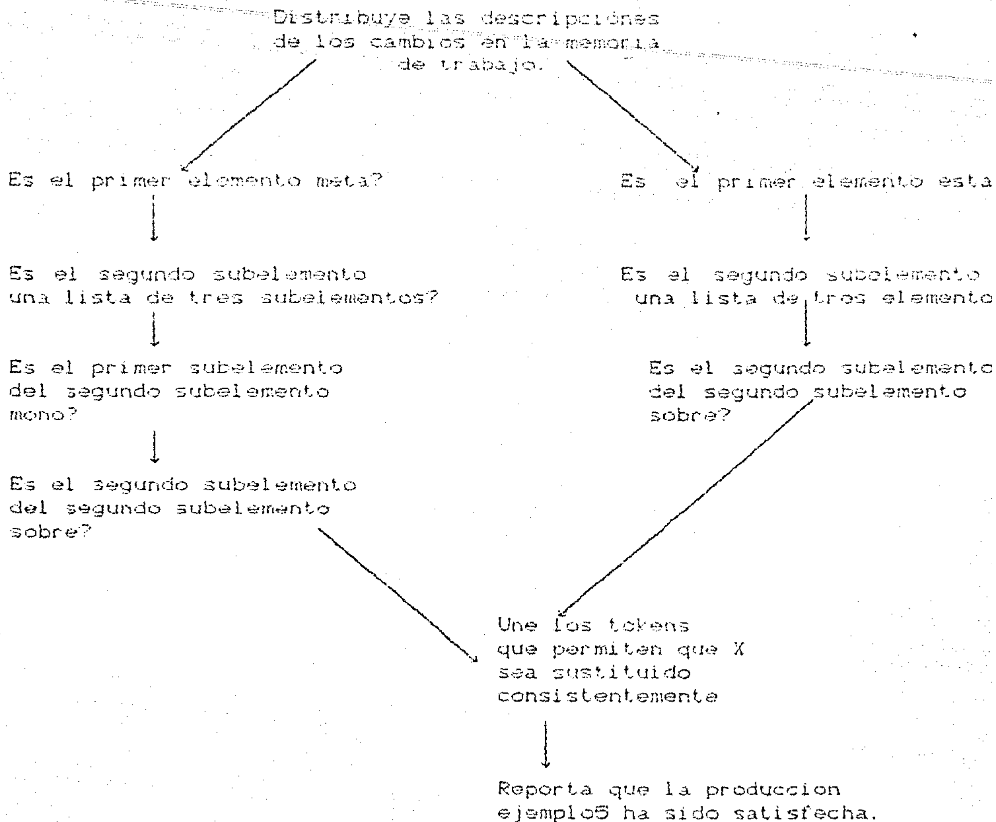
```
regla EJEMPLO5
        {
         meta(mono sobre X),
         estado(X cerca Y)
          -->
         afirma(mono cerca X)
        }
```

Instanciar esta producción involucra probar los elementos de datos aislados como pares de elementos de datos. Las pruebas de los elementos aislados son hechas, como en los ejemplos previos, para probar aspectos como las longitudes de las listas y las constantes en posiciones particulares. Estas pruebas son utilizadas por ejemplo para localizar para el primer elemento condición un elemento de dato que

1. Es el primer elemento meta ?
2. Es el segundo subelemento una lista de tres subelementos?
3. Es el primer subelemento del segundo subelemento mono?
4. Es el segundo subelemento del segundo subelemento sobre?

La prueba de pares de elementos de datos son hechas para asegurar que todas las ocurrencias de la variable X estan acotadas de manera consistente es decir que están acotadas al mismo subelemento de datos.

En el algoritmo utilizado por perito, los nodos de dos entradas realizan la prueba de consistencia en la sustitución de las variables. La red para EJEMPLOS, incluyendo la prueba para X es.



Para ver como trabaja este nodo de dos entradas, supóngase que los siguientes tres elementos de datos

mono(cerca baul)

meta(mono cerca escalera)

estado(escalera cerca cojin)

entran a la memoria de trabajo en este orden.

Suponga también que no hay algo mas en la memoria de

trabajo que pueda pasar por alguncanjunto de nodos de una entrada para esta condición. Cuando el primero de los elementos de datos entra a la memoria de trabajo, su token pasa a través de la rama derecha de la red, y la parte de datos del token es almacenado en la memoria derecha del nodo de dos entradas. Puesto que el nodo no ha recibido nada en su entrada izquierda, no produce salida. Cuando el segundo elemento entra a la memoria de trabajo, su token pasa a traves de la rama izquierda de la red. El nodo de dos entradas almacena la parte de datos del token en su memoria de entrada izquierda y entonces examina su memoria de entrada derecha. Encuentra el token

mono(cerca baul)

ahi, pero puesto que la unión de estos dos elementos produce que X sea sustituida simultaneamente tanto a escalera y mono, el nodo de dos entradas todavia no produce salida. Finalmente *estado(escalera cerca cojin)* entra a la memoria de trabajo. Cuando el token para este elemento llega a la entrada izquierda del nodo de dos entradas, el nodo almacena de inmediato su parte de datos y examina la memoria de entrada izquierda, encontrando *meta(mono sobreescalera)*. La variable X puede entonces ser sustituida consistentemente, por lo que el nodo produce como salida el token

<+, meta(mono sobre escalera) estado(escalera cerca cojin)>.

3.2.9 PRODUCCIONES MAS COMPLEJAS

Muchas producciones son más complejas que EJEMPLO5, conteniendo ya sea mas elementos condición o más variables. La producción EJEMPLO8, por ejemplo, contiene tres elementos condición y dos variables. Una de las variables ocurre en los tres elementos condición, y dos de los elementos

condición contienen a ambas ocurrencias de las variables.

```
regla EJEMPLOS
 (
  meta(X cerca Y),
  ligero(X),
  estado(X  cerca ~Y)
  -->
  afirma( meta(mono agarra X)
  )
```

Habiendo mas elementos condición se hace necesario tener más nodos de dos entradas en la red; en general, un LHS con K elementos condición es compilado en una red con K-1 nodos de dos entradas. Teniendo más de una variable en algún elemento condición sehace necesario tener mas de una prueba en algunos nodos de dos entradas. La red para EJEMPLOS, por ejemplo es

Distribuye descripciones de
los cambios en la memoria de hechos

Es el primer subelemento mota?

Es el segundo subelemento una lista de tres subelementos?

Es el segundo subelemento del segundo subelemento cerca?

Es el segundo subelemento del segundo subelemento cerca?

Es el primer subelemento ligero?

Es el primer subelemento estado?

Junta aquellos token que permiten que X sea sustituida consistentemente?

Junta aquellos tokens que permiten que X sea acotada consistentemente pero que no permiten que X sea acotada.

Reporta que la producción ejemplo5 ha sido satisfecha.

Como un ejemplo en el procesamiento en esta red, considere que los siguientes elementos entran en la memoria de trabajo

estado(escalera cerca baul)

ligera(escalera)

Los tokens que representan estos dos elementos son procesados, pasando a traves de las ramas mas a la derecha y la central, respectivamente. El token para *estado(escalera cerca baul)* es almacenado en su entrada de la memoria derecha del primer nodo de dos entradas.Posteriormente, el elemento *meta(escalera cerca baul)* entra a la memoria de trabajo, y su token pasa por la rama izquierda de la red de **ejemplo5**. Cuando el primer nodo de dos entradas recibe el token, construye

<+, *meta(escalera,cerca,baul) ligero(escalera)* >

y lo envia al otro nodo de dos entradas. El nodo construye

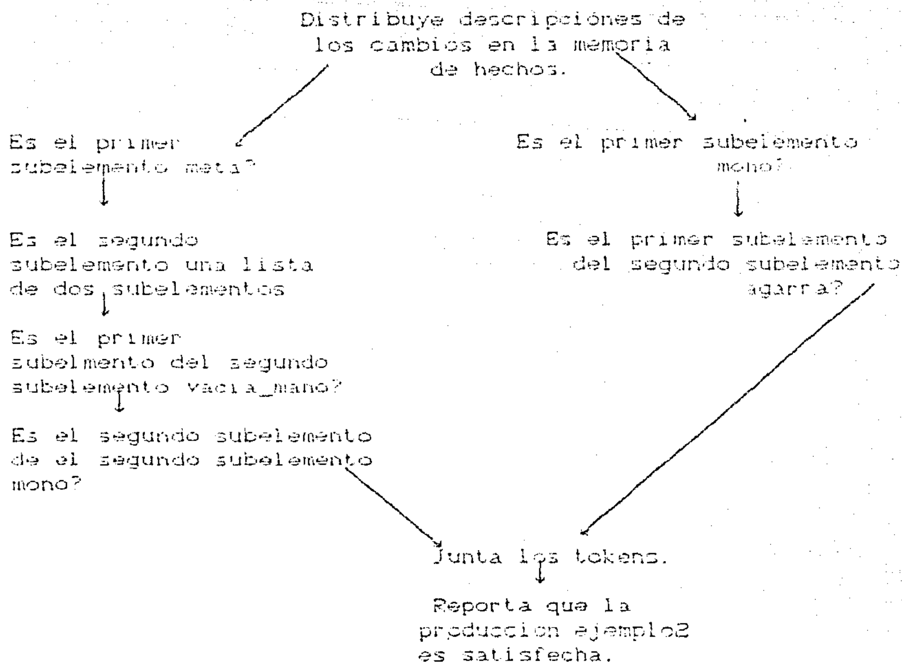
< +, *meta(escalera,cerca,baul) ligero(escalera) estado(escalera,cerca,cojin)* >

y lo envía al último nodo.
Este último nodo añade la instanciación de **ejemplo5** al conjunto conflicto.

3.2.10 PRODUCCIENDO TOKENS DE SALIDA MULTIPLE

En los ejemplos presentada arriba, el arribo de un token a un nodo de dos entradas resultá en a lo más un token de salida. En la práctica un nodo frecuentemente produce más de un token de salida.

Considere la red para la produccion **EJEMPLO5** otra vez.



Si los siguientes elementos entran en la memoria de trabajo vacia, el nodo de dos entradas no producirá salidas, pero hará una marca en la memoria de entrada derecha para cada elemento de dato.

```
mono(agarra  escalera)
mono(agarra  naranja)
mono(agarra  vidrio)
mono(agarra  caja)
```

Si *meta(vacia_mano mono)* entra a la memoria de trabajo posteriormente, su token alcanzara la entrada izquierda del nodo de dos entradas, y provocará procesamiento que resultará en cuatro tokens de salida.

<+, meta(vacia_mano mono) mono(agarra escalera)>

<+, meta(vacia_mano mono) mono(agarra naranja)>

```
<+, meta(vacia_mano mono) mono(agarra vidrio)>
<+, meta(vacia_mano mono) mono(agarra caja)>
```

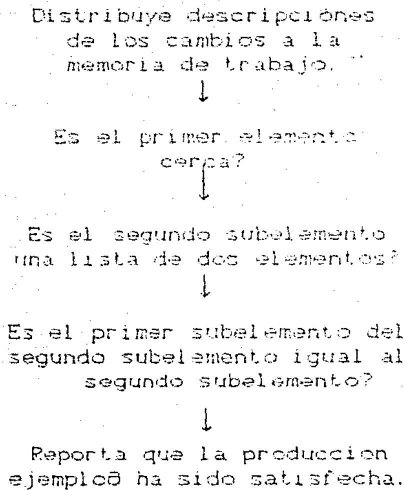
3.2.11 VARIABLES QUE OCURREN VARIAS VECES EN UN ELEMENTO CONDICION

Algunas veces una variable puede ocurrir más de una vez en un elemento condición.

Si la relacion cerca se satisficiera o valiera entre dos objetos, y si el sistema incorporara reglas acerca de la transitividad de la cercania (si A esta cerca de B y B cerca de C entonces A esta cerca de C) puede ser necesario incorporar reglas para reconocer aplicaciones incorrectas de la transitividad. Sería facil de cometer el error de afirmar que un objeto está cerca de sí mismo. La produccion para corregir este error podria simplemente borrar el elemento de dato:

```
regla ejemplo6
  {
  cerca(X  X)^H1 -->quita(H1)
  }
```

puesto que no hay nodos de dos entradas en la red para **ejemplo6**, las sustituciones para X deben ser probadas para certificar consistencia por un nodo de una sola entrada.



Aún si una producción contiene más de un elemento condición - y por lo tanto tiene nodos de dos entradas en su red que deben de probar todas las sustituciones de variables - es mejor probar tanto como sea posible en nodos de una sola entrada. Poniendo las pruebas antes de los nodos de dos entradas se reduce el número de tokens que los alcanzan, o que llegan a ellos, y que tienen que ser almacenados en sus memorias de entrada.

3.3 ELEMENTOS CONDICION NEGADOS

Un nuevo tipo de nodo es necesario para compilar los lados izquierdos que contienen elementos condición negados. Este nodo tiene dos entradas, pero es bastante diferente del nodo de dos entradas visto anteriormente en este capítulo. Esta sección describe este nuevo nodo de dos entradas.

3.3.1 EL NODO <NOT>

Los ejemplos en esta sección utilizan la producción

ejemplo7.

```
regla ejemplo7
 {
   meta(vacia_mano X),
   no(estado(X agarra  _ )
    -->
   afirma(meta(agarra platanos)
  }
```

Puesto que el lado izquierdo es interpretado diferentemente de los otros lados izquierdos discutidos en este capitulo, es valioso establcer claramente que es lo que tiene que hacer la rutina de apareamiento para instanciar ejemplo7. Primero el apareamiento debe encontrar elementos de datos individuales que instancian los dos elementos condicion. El primer elemento condición puede ser instanciado por cualquier elemento de datos que

1. Tenga la constante *meta* como su primer subelemento.
2. Tiene una lista de dos subelementos como su segundo subelemento.
3. Tiene la constante vacia_mano como su primer subelemento de su segundo subelemento.

El segundo elemento condicion puede ser instanciado por cualquier elemento de datos que

1) Tenga la constante *estado* como su primer elemento.
2) Su segundo subelemento tiene longitud 3.
3) Tiene a la constante agarra como su segundo subelemento de su segundo subelemento.

Las acciones tomadas posteriormente dependen de si los elementos de datos que se encontraron instancian uno, ambos

o ninguno de los elementos condicion.

- Si hay instanciaciones para el primer elemento condicion y ninguno para el segundo elemento condicion, el lado izquierdo es satisfecho. Por lo que nada mas es necesario hacer en este caso.

- Si no hay elemento de dato que instancie el primer elemento condicion, el lado izquierdo no es satisfecho; si hay instanciación para el segundo elemento condición es irrelevante. Por lo que es este caso nada más es necesario hacer.

- Si hay instanciaciones para ambos elementos condición, el apareamiento debe probar las sustituciones para X. Cada posible instanciacion del primer elemento condicion es probado contra cada posible instanciación del segundo. Si hay una instanciación para el primero que no tenga una sustitucion consistente con alguno de las instanciaciones del segundo, el lado izquierdo es satisfecho. De otra forma no lo es.

La prueba para los aspectos individuales son compilados en los nodos usuales de una entrada. Los nuevos nodos de dos entradas hacen la prueba de consistencia de sustitucion de variables. La red para el ejemplo anterior es:

Distribuya descripciones
de los cambios a la
memoria de trabajo.

Es el primer elemento
meta?

Es el segundo una
lista de dos subelementos?

Es el primer subelemento
del segundo subelemento
vacia_mano?

Es el primer subelemento
estado?

Es el segundo subelemento
del segundo subelemento
agarra?

Existen tokens de la
izquierda que no permiten
una sustitucion consistente
para X?. Si es asi, pasalos.

Reporta que la producción
ejemplo7 ha sido satisfecha.

3.3.3 MEMORIAS EN EL NODO <NOT>

Una de las pocas similitudes de los dos tipos de nodos de dos entradas es que ambos contienen una memoria interna. Ambos almacenan informaciones en las memorias cuando los tokens marcados como *válidos* llegan y ambos borran la informacion si el mismo token marcado como inválido llega posteriormente. Ambos tienen una memoria para tokens que arriban por la entrada izquierda y otro para tokens que arriban por la derecha, y ambos leen de la memoria de una entrada cuando un token arriba por la otra. Sin embargo, a pesar de estas similitudes, los dos tipos de nodos difieren algo en el uso de las memorias, como lo muestra la siguiente sección.

3.3.3 NUEVOS TOKENS INVALIDOS QUE LLEGAN POR LA DERECHA

La descripción del procesamiento del nodo <NOT> es más simple si se da un tratamiento individual a las varias situaciones con las que se tiene que enfrentar el nodo. La llegada por la derecha de un token marcado INVALIDO es potencialmente la mas dificil de las situaciones. Porque el procesamiento del nodo en los otros casos esta determinado parcialmente por la necesidad de hacer este caso más fácil, por lo que este caso es considerado primero.

Un ejemplo mostrará el problema que debe ser resuelto para manejar este caso correctamente.

Supóngase que el sistema que hemos estado usando está trabajando con dos monos al mismo tiempo. El sistema de producción a estado corriendo un cierto numero de ciclos, y los cambios hechos en la memoria de trabajo han causado que

el nodo <NOT> reciba los tokens

<VALIDO, estado(mono1,agarra, escalera) >
<VALIDO, estado(mono1,agarra,naranja) >
<VALIDO, estado(mono2,agarra,naranja) >

sobre su entrada derecha y el token

<VALIDO, meta(mano_vacia, mono1) >

sobre su entrada izquierda. La produccion ejemplo_p

```
regla ejemplo_p
 (
  meta(mano_vacia,X),
  no(estado(X,agarra, _)
  -->
  afirma(meta(agarra,platanos)
  )
```

no es instanciada. Si el token

< invalido, estado(mono1,agarra,naranja) >

arriba por su entrada derecha, ejemplo_p todavia no se instancia. Pero si

< inválido, estado(mono1,agarra,escalera) >

arriba después, ejemplo_p es instanciado. La entrada restante en la memoria derecha - para

estado(mono2 agarra platanos)>

- no previene la instanciación porque X no puede ser acotada tanto a mono1 como a mono2 simultaneamente. Si los dos tokens INVALIDOShan llegado en el orden opuesto, ejemplo_p se hubiera instanciado solo después del arribo de

<inválido, estado(mono agarra naranja) >

Sin importar el orden en el cual arriben los tokens marcados como **INVALIDOS**, el nodo no debe de enviar nada despues del arribo del primero, sino despues del arribo del segundo debe de enviar el siguiente token

<VALIDO,meta(mano_vacia,mono) >.

El problema a ser resuelto en el diseño de los nodos <NOT> está en como determinar cuando enviar tokens como este sin un esfuerzo excesivo.

La solución adoptada en el algoritmo de apareamiento escogido para la implementacion de perito es almacenar un contador junto con cada entrada en la memoria izquierda. Este contador indica cuantas entradas en la memoria derecha permiten que las variables sean sustituidas consistentemente. Cuando un token marcado como **INVALIDO** llega por la entrada derecha causa que un elemento de la memoria derecha sea borrado, las pruebas de sustitución de variables son repetidas para cada elemento en la memoria izquierda. Cuando se encuentra uno que tiene una sustitucion de variables consistentes el contador se decrementa por uno. Si un contador llega a ser cero, el nodo construye un token marcado como **VALIDO** para la entrada correspondiente en la memoria izquierda. En el ejemplo anterior, el contador para meta(mano_vacia,mono1) era inicialmente dos. Cuando llegó el primero token por la derecha, el contador disminuyó a uno. Cuando el siguiente token llego, se disminuyó a cero, resultando en el envio del siguiente token

< VALIDO, meta(mano_vacia mono1) >.

Una razon para escoger esta solucion al problema es mantener razonablemente pequeño la cantidad de estado almacenado en los nodos <NOT>. Las memorias derechas almacenan la misma información que las memorias izquierdas de otros tipos de nodos de dos entradas: la parte de datos

de los token VALIDOs. Las memorias izquierdas almacenan un entero junto con cada parte de dato.

3.4 CONSIDERACIONES DE EFICIENCIA

Esta seccion está principalmente interesada en analizar el costo del algoritmo de apareamiento de red. Las primeras dos secciones explican como el algoritmo saca ventaja de la redundancia temporal y de la similitud estructural.

3.4.1 REDUNDANCIA TEMPORAL.

La respuesta del algoritmo a la redundancia temporal debe estar claro ahora. La red procesa los elementos de datos solo cuando ellos cambian; almacena la informacion acerca de los elementos de datos en tanto ellos permanezcan sin cambiar.

3.4.2 SIMILITUD ESTRUCTURAL

El algoritmo de apareamiento saca ventaja de la similitud estructural compartiendo nodos entre las producciones. Cuando dos lados izquierdos se compilan en secuencias de nodos que contienen subsecuencias iniciales idénticas, las secuencias iniciales son compartidas. El compartimiento es mas pesado, claro está, cuando las producciones son muy similares. Por ejemplo:

```
regla ejemplo_x
   {
   meta(mono sobre X),
   cerca(X Y)

   -->

   afirma( cerca(mono Y))
   }
```

```
regla ejemmplo_y
   {
   meta(mono sobre X),
   cerca(X Y),
   cerca(mono Y)
   -->
   afirma(meta(vacia_mano mono))
   }
```

Las producciones ejemplo_x y ejemplo_y son un ejemplo, que contenien dos elementos condicion identicos. Cuando estas dos producciones se compilan, la mayoría de los nodos en la red son compartidos.

Distribuye descripciones
de los cambios a la
memoria de trabajo.

Es el primer
subelemento
meta?

Es el segundo
subelemento una
lista de tres
subelementos?

Es el primer
subelemento del
segundo subelemento
mono?

Es el segundo
subelemento del
segundo subelemento
sobre?

Es el primer subelemento
cerca?

Tiene el segundo subelemento
longitud 2?

Es el prime
subelemento
del segundo
subelemento
mono?

Junta aquellos
tokens que
permitan a X
sea sustituido
consistentemente.

Junta aquellos
tokens que permitan
que Y sea sustituida
consistentemente.

Reporta que la
produccion ejemplo_x
ha sido satisfecha

Reporta que la producci_
ejemplo_y esta satisfecha

Aún producciones muy disimbolas, frecuentemente comparten ciertas cosas: la red para ejemplo_x y ejemplo_y tiene 17 nodos; contendría 20 nodos si estos no fueran compartidos.

```
regla ejemplo_x
  {
  meta(mono sobre X),
  cerca(X Y)
  -->
  afirma(cerca(mono Y))
  }

 regla ejemplo_y
  {
   meta(vacia_mano mono),
   estado(mono agarra X)
   -->
   escribe("El mono tira " X)
   }
```

Distribuye descripciones de los cambios a la memoria de trabajo.

Es el primer subelemento meta?

Es el primer subelement cerca?

Tiene el segundo subelemento longitud 2?

Es el primer subelemento estado?

Es el segundo subelemento una lista de tres subelementos?

Es el primer subelemento del segundo subelemento vacia_mano?

Es el primer subelemento del segundo subelem mono?

Es el primer subelemento del segundo subelemento mono?
subelemento

Es el segundo subelemento del segundo subelemento

mono?

Es el segundo subelemento del segundo

agarra?

Es el segundo subelemento del segundo subelemento sobre?

Junta aquellos tokens que permitan a X sea sustituido consistentemente.

Junta los tokens

Reporta que la produccion ejemplo_x ha sido satisfecha

Reporta que la producci ejemplo_y es satisfecha

3.5 LOS NODOS DEL PROGRAMA.

La sección pasada a levantado una serie de discusiones y descrito la respuesta del algoritmo de red a cada uno. El resultado es una descripción completa aunque larga de los aspectos escenciales del algoritmo. Esta seccion contiene una descripcion más concisa.

La version del algoritmo de apareamiento simple descrito en este capitulo usa seis clases de nodos. Estos son el nodo BUS (el nodo que reporta los cambios hechos a la memoria de trabajo a el resto de los nodos); los nodos de una entrada que prueban la sustitución de una o mas variables; los nodos de una entrada que prueban los aspectos constantes (por ejemplo la longitud de una lista o la identidad de un atomo); los nodos de dos entradas para elementos condición no negados; los nodos <NOT>; y los nodos que reportan cambios hechos al conjunto conflicto. Cada una de estas clases se describe abajo listando dos cosas:la información que está construida permanentemente en el nodo de ese tipo, y el procesamiento realizado en un nodo cuando éste es activado.

3.5.1 LOS NODOS DE UNA ENTRADA QUE PRUEBAN ASPECTOS CONSTANTES

Construidos dentro de los nodos de una entrada que prueban aspectos constantes están

- El indice de los subelementos a probar.
- Una constante con la cual comparar el subelemento (o su longitud).
-Una descripción de la prueba a ser realizada.
- Y una lista de los vertices que parten del nodo.

Cuando el nodo es activado por la llegada de un token, se realizan tres pasos y luego se detiene, apaciguándose en espera del siguiente token:

1) extraer el subelemento a ser probado.

2) realiza la prueba.

3) si la prueba tiene éxito, envia el token a sus sucesores.

4) PARA.

3.5.2 NODOS DE UANA ENTRADA PARA PROBAR SUSTITUCIONES DE VARIABLES

Construidos dentro de los nodos de una entrada que prueban sustituciones de variables estan

- Los indices de los dos subelementos a probar.
- Una descripcion de la prueba a realizar.
- Y una lista de los vértices que parten del nodo.

Cuando el nodo es activado realiza los siguientes pasos:

1) extrae el primer subelelemento.

2) extrae el segundo subelemento.

3) realiza la prueba.

3.5.3 LOS NODOS ORDINARIOS DE DOS ENTRADAS

La cantidad de información construida en un nodo de dos entradas depende del número de variables probadas por el nodo. Para cada variable a ser probada, tiene lo siguiente:

- Dos indices de subelementos.

- Una descripcion de la prueba a realizarse en los dos subelementos.

Además, tiene las ligas usuales a sus sucesores:

- Una lista de los vertices que parten al nodo.

El procesamiento realizado por el nodo depende del vértice sobre el cual el token haya arribado. Si éste llegó por la izquierda: entonces realizamos lo siguiente:

1) Si el token está marcado como "VALIDO", almacena la parte de datos en la memoria izquierda; de otra forma encuentra y borra una parte de datos identica de la memoria izquierda.

2) Para cada parte de datos de la memoria derecha

COMIENZA.

3) Falla:=0.

4) para cada variable a probar hasta que Falla=1

COMIENZA.

5) Usa el primer indice para extraer un subelemento de la parte de datos izquierda.

6) Usa el segundo indice para extraer un subelemento de la parte de datos derecha.

7) Realiza la prueba.

8) Si la prueba falló, Falla=1.

9) Si Falla=0,

COMIENZA.

10) Pon D igual a la parte de datos del token de la izquierda concatenado con la parte de datos de token de la derecha.

11) Construye un token usando D y la parte de etiqueta del token que recien ha llegado.

12) Envia el nuevo token a los sucesores.

TERMINA.

TERMINA.

13) ALTO.

El procesamiento realizado cuando los tokens llegan por la derecha es similar, excepto que en algunos lugares (pero no en todos, vea los pasos 5,6 y 10) los roles de la izquierda y la derecha son invertidos.

1) Si el token está marcado como VALIDO, almacena la parte de datos en la memoria derecha; de otra forma encuentra o borra una parte de datos idéntica a la memoria derecha.

2) Para cada parte de datos en la memoria izquierda
COMIENZA.
3) Falla:=0.
4) Para cada variable a probar hasta que Falla=1
COMIENZA.
5) Usa el primer indice para extraer un subelemento de la parte de datos izquierda.
6) Usa el segundo indice para extraer un segundo subelemento de la parte de datos derecha.
7) Realiza la prueba.
8) Si la prueba falla, Falla=1.
TERMINA.
9) si Falla=0,
COMIENZA.
10) Pon D igual a parte de datos del token de la izquierda concatenados con la parte de datos del token de la derecha.
11) Construye un token usando D y la parte de etiqueta del token que recién ha arribado.
12) Envía el nuevo token a los sucesores.
Termina.
TERMINA.
13) ALTO.

3.5.4 EL NODO <NOT>

Un nodo <NOT> contiene la misma informacion que un nodo regular de dos entradas. Tiene las ligas usuales a sus sucesores:
- Una lista de los vertices que parten del nodo.

Para cada variable a ser probada, tienen lo siguiente:

- Dos indices de subelementos.
- Una descripción de la prueba a realizar en estos dos.

Cuando un nodo <NOT> recibe un token en su memoria izquierda realiza lo siguiente:

1) Contador:=0.
2) Para cada parte de datos en la memoria izquierda COMIENZA .
   3) Falla:=0.
   4) Para cada variable a probar hasta que Falla=1 COMIENZA.
      5) Usa el primer indice para extraer un subelemento de la parte de datos izquierda.
      6) Usa el segundo indice para extraer un subelemento de la parte de datos derecha.
      7) Realiza la prueba.
      8) Si la prueba falla, Falla:=1.
      TERMINA.
   9) Si Falla=0, Contador:=Contador + 1.
   termina.
10) Si el token que recién ha llegado está marcado como"VALIDO", almacena Contador y la parte de datos en sumemoria izquierda; de otra forma borra una parte de datosidéntica y su contador.

11) Si Contador=0, envia el token que recién ha llegado a sus sucesores.

12) TERMINA.

El procesamiento realizado cuando un token llega por la derecha es bastante diferente:

1) Si el token está marcado como VALIDO, almacena la parte de datos en la memoria derecha; de otra forma encuentra y borra una parte de datos identica de su memoria derecha.

2) Si el token está marcado como VALIDO, Incremento:=1 ; de otra forma.Incremento:=-1.

3) Para cada parte de datos en la memoria izquierda

COMIENZA.

4) CONTADOR_NUEVO:=contador almacenado con la parte de datos.

5) Falla:=0.

6) Para cada variable a probar hasta que Falla:=1

COMIENZA.

7) Use el primer indice para extraer un subelemento de la parte de datos izquierda.

8) Use el segundo indice para extraer un subelemento de la parte de datos derecha.

9) Realice la prueba.

10) Si la prueba falla, Falla:=1.

Termina.

11) Si la Falla=0,

COMIENZA.

12) Pon CONTADOR_NUEVO= CONTADOR_NUEVO+ Incremento.

13) Reemplaza el contador en la memoria izquierda con CONTADOR_NUEVO.

14) Si (CONTADOR_NUEVO=0 e Incremento=-1) o (CONTADOR_NUEVO=1 e Incremento=1).

COMIENZA.

15) Si Incremento=-1, Etiqueta:="VALIDO"; de otra forma, Etiqueta:="INVALIDO".

16) Construye un token utilizando la etiqueta y la parte de datos de la izquierda.

17) Envia el nuevo token a los sucesores.

Termina.

TERMINA

TERMINA.

18)ALTO.

3.5.5 EL NODO QUE CAMBIA EL CONJUNTO CONFLICTO

El nodo que cambia el conjunto conflicto solo tiene un fragmento de información construido dentro de el que es

- El nombre de la produccion que representa.

Y el procesamiento que realiza cuando llega un token depende de la etiqueta del token:

1) Si el token está marcado como "VALIDO", agrega la parte de datos del token y el nombre de la producción alconjunto conflicto; de otra forma remueve una entrada identica del conjunto conflicto.
2) Termina.

3.5.6 EL NODO BUS

El nodo que reporta cambios hechos a la memoria de trabajo al resto de la red tiene construido solo una cosa:

- Una lista de los vértices que parten del nodo.

Este nodo no puede ser activado de la misma forma que los otros nodos. Ningún token puede ser enviado a él puesto que es en este nodo es donde los tokens son primeramente construidos. Es activado despues de recibir cada cambio hecho a la memoria de trabajo. Si los únicos cambios hechos a la memoria de trabajo son adiciones de nuevos elementos y eliminaciones de elementos existentes, el procesamiento realizado por su activacion es:

1) Si la memoria de trabajo cambió por una introducción. Etiqueta:="VALIDO",de otra manera Etiqueta:= "INVALIDO".
2) Construye un token usando el elemento de memoria afectado y etiqueta.
3) Envía el token a sus sucesores.
4) TERMINA.

C A P I T U L O 4

4. MODIFICACIONES AL ALGORITMO DE APAREAMIENTO DE RED UTILIZADOS POR PERITO

Este capitulo contiene una información detallada del interprete de PERITO. Este capitulo explica las dos diferencias fundamentales con el algoritmo descrito en el capitulo 3 y el algoritmo utilizado por PERITO. Después se describe el formato de los nodos y de los tokens en PERITO.

Finalmente se describe el procesamiento realizado por los nodos. Esta descripción es suficientemente detallada de tal forma que cualquiera podria reproducir, en principio, este algoritmo y este intérprete.

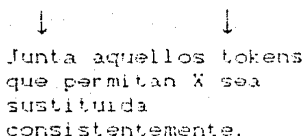
4.1 REDUCIENDO EL COSTO DE ALMACENAR TOKENS

Debido a que el intérprete de PERITO fue diseñado para correr en un uniprocesador, el algoritmo del capitulo anterior debe ser cambiado en una forma que ligeramente reduzca el numero de tokens almacenados en la red. Esta sección describe los cambios.

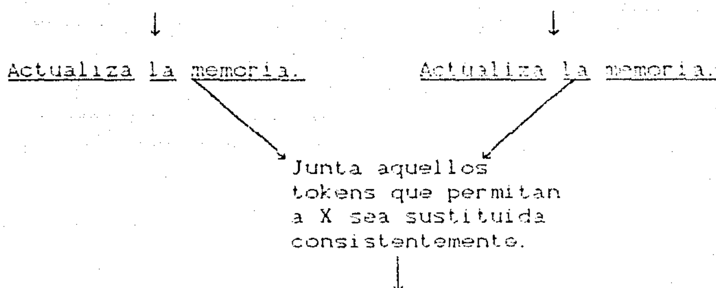
4.1.1 NODOS DE MEMORIA SEPARADOS.

Los nodos de dos entradas descritos en el último capitulo realizan dos funciones relacionadas pero distintas. Mantienen sus memorias internas que contienen las partes de datos de los tokens, y comparan la parte de datos para determinar cuales permiten sustituciones consistentes de variables. Aunque la mayoria de los intérpretes que usan el algoritmo de red tienen nodos de dos entradas de este tipo, algunas versiones utilizan nodos más simples. En el intérprete PERITO, tres nodos simples son usados para realizar estas funciones de uno de los más complejos nodos

de dos entradas. Un nodo mantiene la memoria izquierda, y el segundo mantiene la memoria derecha, y un tercero realiza la prueba de sustitucion de variables. Un nodo tipico de el ultimo capitulo es



lo cual convertido a PERITO tiene la siguiente forma:



En lugar de dos memorias internas, un nodo de dos entradas en PERITO contiene apuntadores hacia atrás a sus predecesores. Los predecesores son nodos de memoria.

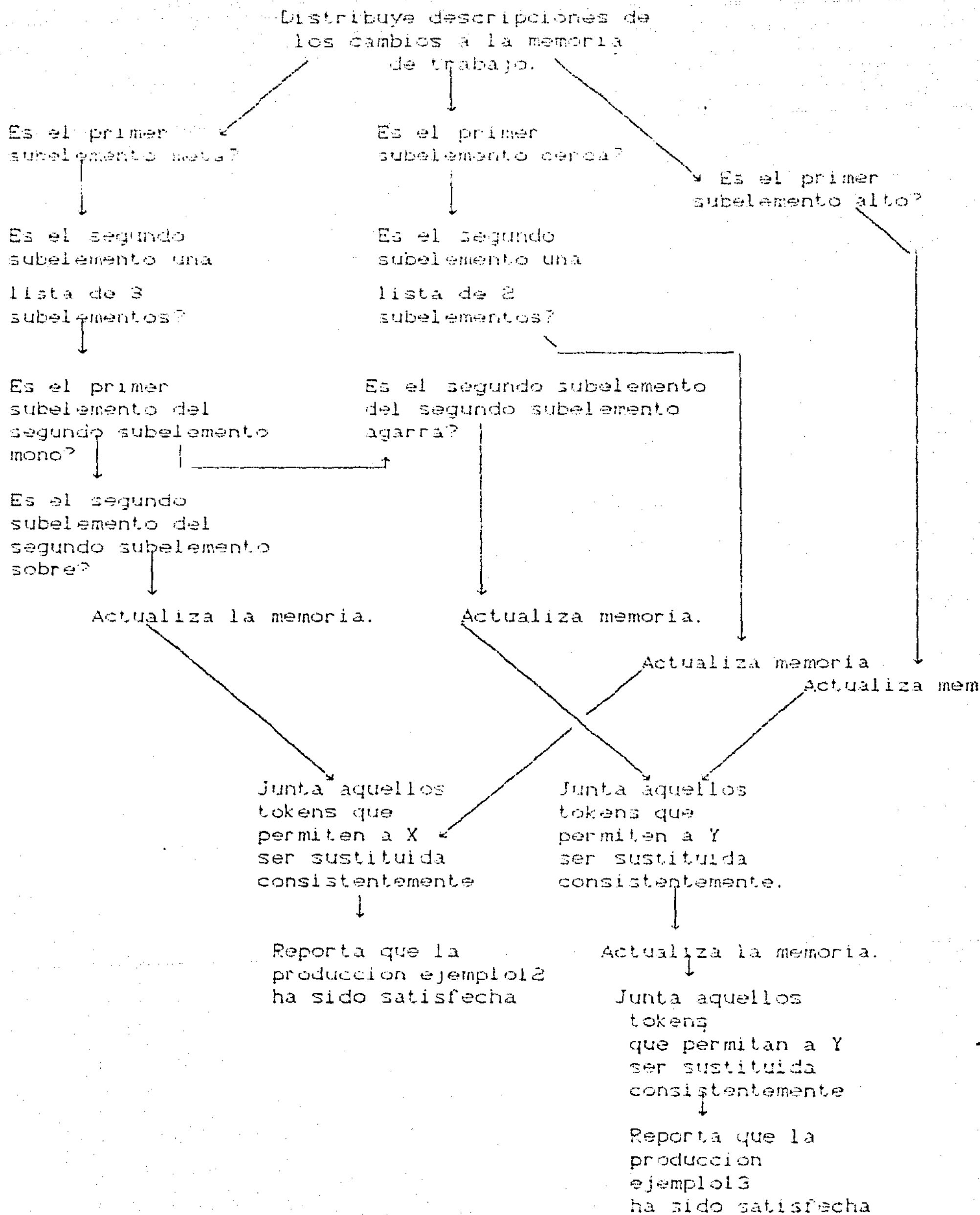
Las memorias han sido separadas de los nodos de dos entradas porque esto incrementa el potencial de compartimiento. Los nodos de memoria pueden ser compartidos entre dos producciones si las producciones tienen elementos condiciones que son idénticos excepto por las variables. El tercer elemento condición en ejemplo12 y el segundo elemento

condición de ejemplo13, por ejemplo, permiten compartir un nodo de memoria.

```
regla ejemplo12
  (
   meta(mono  agarra  X),
   alto(X),
   cerca(X  Y)
   -->
   afirma(cerca(escalera  Y)
  )

 regla ejemplo13
   (
   meta(mono  sobre  X),
   cerca(X  Y)
   -->
   afirma( cerca(mono  Y))
   )
```

La red para estas dos funciones contiene cinco memorias; seis se hubieran requerido si las memorias fueran internas a los nodos de dos entradas.


Distribuye descripciones de los cambios a la memoria de trabajo.
Es el primer subelemento meta?
Es el primer subelemento cerca?
Es el primer subelemento alto?
Es el segundo subelemento una lista de 3 subelementos?
Es el segundo subelemento una lista de 2 subelementos?
Es el primer subelemento del segundo subelemento mono?
Es el segundo subelemento del segundo subelemento agarra?
Es el segundo subelemento del segundo subelemento sobre?
Actualiza la memoria.
Actualiza memoria.
Actualiza memoria
Actualiza mem
Junta aquellos tokens que permiten a X ser sustituida consistentemente
Junta aquellos tokens que permiten a Y ser sustituida consistentemente.
Reporta que la produccion ejemplo12 ha sido satisfecha
Actualiza la memoria.
Junta aquellos tokens que permitan a Y ser sustituida consistentemente
Reporta que la produccion ejemplo13 ha sido satisfecha

Rompiendo los nodos de dos entradas mejora la eficiencia del algoritmo en algunas maneras, pero degrada la eficiencia en una. Tanto el tiempo requerido para actualizar la red en cada ciclo comonúmero de tokens almacenados en la red son reducidos rompiendo los nodos. Pero aun cuando se comparten nodos de memoria, el numero total de nodos en la red se incrementa.

4.1.2 MEMORIAS PARA LOS NODOS <NOT>

Los nodos <NOT> no comparten sus memorias izquierdas. Recordemos que la memoria izquierda de un nodo <NOT> almacena un contador junto con cada parte de dato. Puesto que estos contadores tienen significado solo para el nodo que los generó, ellos presentarían problemas si fueran almacenados en una estructura compartida. Por lo que los nodos <NOT> deben almacenar los contadores internamente. Los nodos <NOT> pueden almacenar solo los contadores, utilizando los nodos de memoria para almacenar las partes de datos. Pero cuando se diseñó a PERITO, parecía más simple tener la parte de datos almacenados juntos. Por lo que la memoria izquierda del nodo <NOT> se mantuvo interna al nodo. Y puesto que la memoria derecha de un nodo <NOT> almacena solo parte de datos del token, puede ser compartido como las memorias de un nodo regular de dos entradas.

4.1.3 EL PROCESAMIENTO EFECTUADO POR LOS NODOS DE MEMORIA

Los nodos de memoria efectúan solo tres funciones. Mantienen sus memorias internas, pasan los tokens a lo largo de sus vrtices llegando a las entradas derechas de los nodos de dos entradas, y pasan tokens a lo largo de vrtices que conducen a las entradas izquierdas de los nodos de dos entradas.

El orden en que estas operaciones se efectue es critica. Considresa la siguiente produccion:

```
regla ejemplox
    {
    menor_igual(X  Y),
    menor_igual(Y  X)
    -->
    afirma(igual(X  Y))
    }
```

Debido a que los dos elementos condicion son idnticos excepto en el orden de las variables, el nodo dos entradas en la red de EX1 comparte un nodo de memoria consigo mismo.

Distribuye descripciones de los cambios a la memoria de trabajo.

↓

Es el primer subelemento menor_igual?

↓

Es el segundo subelemento una lista de longitud 2?

↓

Actualiza la memoria.

↓

Junta aquellos tokens que permitan a X y Y ser sustituidos consistentemente.

↓

Reporta que la produccion ejemplox ha sido satisfecha.

Si el elemento *menor_igual(p p)* entrara a la memoria de trabajo vacía, EX1 tendria una instanciacion legal, y

esta red la encontraria. Pero si el nodo de memoria efectuara su operación en el orden incorrecto, la red podria fallar de añadir su instanciación al conjunto conflicto o añadirla dos veces.

Supongase que el nodo de memoria modifica su memoria interna antes de enviar el token a sus sucesores. Por lo que cuando el token

< valido, menor_igual(p p) >

arriva al nodo de memoria, el nodo debe de almacenar de inmediato la parte de datos del token y posteriormente enviar el token a cada vrtice que parte del nodo. Estos dos vrtices conectan la memoria de trabajo a el nodo de dos entradas, el toke arrivaria al nodo de dos entradas dos veces. Si arrivará por la izquierda primero, el nodo veria a a su memoria izquierda y encontraria *menor_igual(p p)*. Puesto que las variables pueden ser acotadas consistentemente, el nodo enviaria el token

< valida, menor_igual(p p) menor_igual(p p) >.

Cuando el token

< valida, menor_igual(p p >

arriva por su entrada derecha, el nodo checaria su memoria izquierda, encontrando *menor_igual(p p)* , y otra vez envía el token

< valido, menor_igual(p p) menor_igualp p) >.

El último nodo recibiria dos tokens idnticos y añadiria dos instanciaciones de EX1 al conjunto conflicto.

Puesto que el apareamiento procede incorrectamente si los nodos memorias modifican sus memorias internas antes de enviar los tokens, supongase que los nodos modifican sus memorias despus de enviar sus tokens. Si esto va a ser deterministico, debe suponerse que los nodos memoria permiten a sus sucesores terminar el procedimiento antes de modificar sus memorias. Entonces cuando el nodo de memoria recibe el token

<valido, menor_igual(p p) >

inmediatamente lo envia a lo largo de sus vrtices que parten del nodo. Si el token llegó orimero por la entrada izquierda del nodo dos entradas, el nodo checará su memoria derecha, no encontrando nada ahi, y por lo tanto no produciendo salidas. Cuando el token llega por su entrada derecha, el nodo checara su memoria izquierda, encontrando que tambin se encuentra vacia, y por lo tanto tampoco produciendo ninguna salida. Finalmente el nodo de memoria modificará su memoria, bastante tarde para que el nodo dos entradas produzca una salida.

El apareamiento procederá correctamente solo si el nodo de memoria distingue los vrtices que alcanzan entradas derechas de los nodos de los vrtices que alcanzan entradas izquierdas de los nodos. Una secuencia correcta es enviar primero el token a lo largo de los vrtices que conducen a entradas izquierdas de los nodos de dos entradas, esperar que todos los nodos terminen procesamiento, despus modifocar el nodo memoria, y finalmente enviar el token a los vrtices que llevan a las entradas derechas. Si el nodo memoria hiciera sto , cuando

<valido, menor_igual(p p) >

arriva, enviaria primero el token a lo largo del vrtice que conduce a la entrada izquierda del nodo de dos entradas. El nodo de dos entradas checaria su memoria derecha y no encontrando nada ahi, no produce salida. Cuando el nodo de dos entradas termine el procesamiento, el nodo de memoria añadiria

< *valido, menor_igual(p p) menor_igual(p p)* >

a la memoria y enviaria el token a cada uno de los vrtices que conducen a la entrada derecha del nodo de dos entradas. El nodo de dos entradas checaria su memoria izquierda y encontraría *menor_igual(p p)* ahi. Puesto que las variables pueden ser acotadas consistentemente, sacaria el token

<*valido, menor_igual(p p) menor_igual(p p)* >.

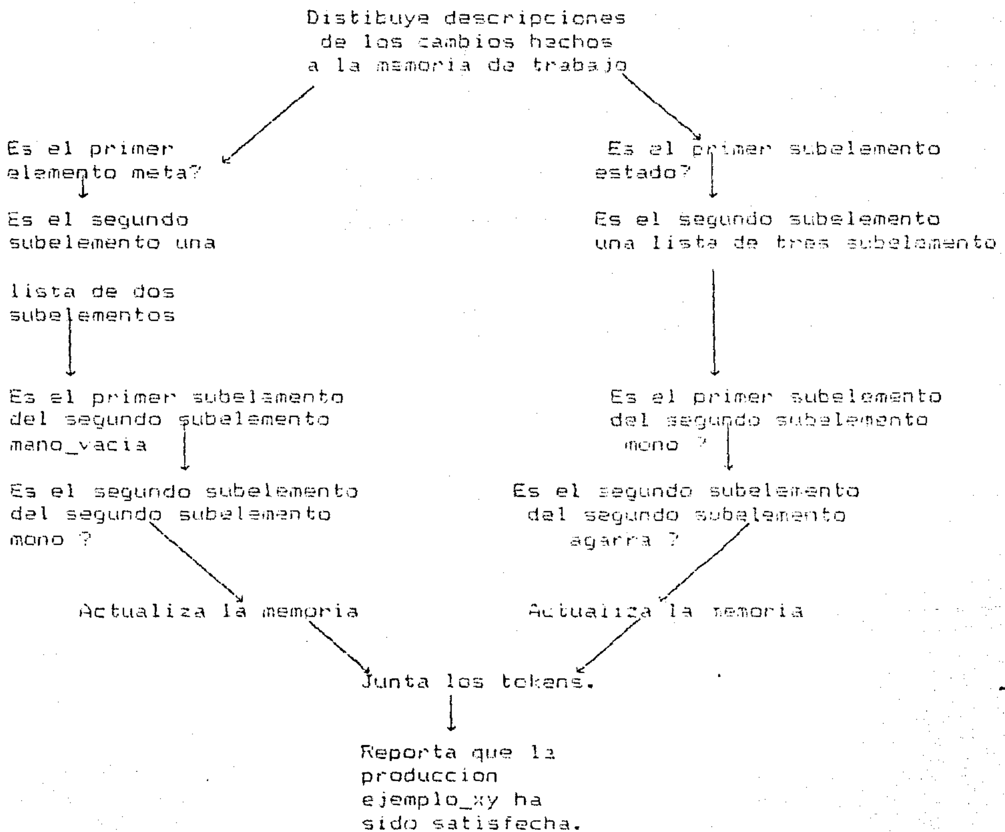
Puesto que el último nodo de la red recibiría este único token, añadiría la instanciación de EX1 al conjunto conflicto solo una vez.

4.1.4 Sincronizando los nodos de dos entradas divididos.

Los nodos de memorias separados no fueron descritos en el último capitulo debido a que su uso hace al algoritmo sensitivo al orden en el cual los nodos son ejecutados. Esta sensibilidad presenta poco problema para una implementación en un uniprocesador como PERITO, pero complicaría el diseño en un intrprete en paralelo. Para ver un ejemplo del problema, considrese la producción ejemplo_xy

```
regla ejemplo_xy
   {
    meta(vacia_mano  mono),
    estado(mono  agarra  X)
    -->
    escribe("El mono lleva "  X)
    }
```

El compilador de PERITO traduciría el lado izquierdo de ejemplo_xy en:



Supongan que *meta(mano_vacia mono)* y *estado(mono agarra escalera)* entran en la memoria vacia al mismo tiempo, y suponga que el apareamiento procesa ambos cambios en paralelo. Si ambos tokens alcanzan los nodos de dos entradas simultaneamente, el orden en el cual los tokens son procesados determinara cual es la salida por el nodo. Si el nodo de dos entradas acepta el token de la izquierda primero, produciria dos tokens de salida. Cuando procesa

< válido, meta(vacia_mano mono) >

checará su memoria derecha y encontrará la parte de dato de

< valido,estado(mono agarra escalera) >.

La parte de datos estara ahi porque, si el nodo de memoria usa el ordende procedimiento decidido en la última seccion, habrá modificado su memoria antes de enviar el token a su entrada derecha del nodo. Puesto que encuentra la parte de datos en la memoria derecha, el nodo dos entradas construira y sacará el token

<valido,meta(vacia_mano mono) estado(mono agarra escalera)>.

Despus de que el nodo dos entradas termina, dos cosas suceden en paralelo. El nodo dos entradas acptará el otro token, y el nodo de memoria en la rama izquierda de la red modificará su memoria. Si la memoria izquierda es modificada ahi antes de que el nodo de dos entradas la lea, el nodo dos entradas encontrará ahi la parte de datos de

<válido,meta(vacia_mano mono) >.

esto causará que el nodo dos entradas saque otra vez el token

válido,meta(vacia_mano mono) estado(mono agarra escalera) >.

Puesto que el intrprete de PERITOse ejecuta de una manera serial no es difícil ordenar el procesamiento de los nodos de tal forma que este problema se evita. Si se realizara una ejecución en paralelo, sin embargo sincronizaciones explicita de los nodos sería necesaria.

4.2. EL FORMATO DE LOS DATOS DEL INTERPRETE.

Esta sección describe los dos tipos de datos procesados por el intrprete de PERITO los tokens y los nodos.

4.2.1 EL FORMATO DE LOS TOKENS

Los tokens en PERITO son representados por medio de listas. El primer elemento de la lista es la etiqueta del token, y los elementos restantes son apuntadores a los elementos de dato. El token

<+,meta(vacia_mano mono)>

se representa en PERITO de la siguiente manera

(valida (meta (vacia_mano mono)))

La parte de los datos del token están en el orden inverso. El token

<válida, meta(vacia_mano mono) estado(mono agarra escalera))>.

está representado como.

(valida (meta (vacia_mano mono)) (estado (mono agarra escalera))).

El elemento de dato debe de estar almacenado en la memoria de trabajo. (La parte de datos no contiene copias de los elementos, sino apuntadores hacia slots apropiádos en la memoria de trabajo).

Puesto que el compilador de PERITO arregla los nodos de dos entradas en una secuencia lineal, unidos por sus entradas izquierdas, la parte de datos

((estado (plátanos cerca baul)) (alto (platanos)) (estado (mono agarra plátanos)))

no estará en una memoria de trabajo a menos que la parte de datos

((alto (platanos)) (estado (mono agarra plátanos)))

estuviera en la memoria izquierda del nodo que construyó el token más grande. Esta parte de datos de dos elementos, en turno, no estaría en memoria a menos de que

((estado (mono agarra plátanos)))

fuera almacenado en una memoria anterior.

4.2.2 EL FORMATO DE LOS NODOS

LOS NODOS DE UNA ENTRADA

Tres de los nodos de una entrada solo contienen cuatro campos. Estos son los nodos &len, &atomo y &Vin, los cuales son descritos en esta subsección.

Los nodos &len prueban la longitud de los elementos de dato y de los subelementos. El primer campo en uno de estos nodos contiene el tipo de nodo (que en este caso es &len). El segundo campo contienen la lista de los sucesores del nodo. El tercer campo contiene un vector de índices; y el cuarto contiene una constante entera. El nodo

¿ es el segundo subelemento de una lista de tres subelementos ?
está codificado en PERITO de la siguiente manera

(&len (. . .) (1 2) 3).

Los puntitos representan los apuntadores a los sucesores del nodo.

El nodo &atomo prueba la ocurrencia de una constante particular. El nodo

¿es el primer subelemento meta?

se codifica de la siguiente manera

(&atomo (. . .) (1 1) meta)

Los nodos &VIN prueban las variables que ocurren mas de una vez en un elemento condición. Su nombre fue utilizado para indicar que prueba variables que son internas a un elemento condición.

El nodo

es el primer subelemento igual al tercer subelemento?

se codifica de la siguiente manera

(&VIN (. . .) (VARIABLE 1 1) (VARIABLE 1 3)).

Este nodo muestra la codificación de variables cuando ambos son variables ordinarias .

4.2.4 EL NODO MEMORIA

El nodo memoria contiene cuatro campos. El primer campo como siempre, contiene el tipo del nodo en este caso &MEM.

El segundo y tercer campo contiene una lista de los sucesores del nodo. Recordamos que un nodo de memoria debe distinguir los sucesores del cual desempeña el rol de "memoria izquierda" de aquellos que desempeñan el rol de "memoria derecha". Esto lo hace así teniendo un campo para lo que puede ser llamado los "sucesores izquierdos" y uno para los "sucesores derechos". El último campo contiene una lista de un elemento. Este elemento es la memoria de el nodo. Puesto que las memorias de PERITO son listas simples de partes de datos, un nodo de memoria con memoria vacía esta representado de la siguiente manera

(&MEM (. . .) (. . .) (())).

4.2.5 EL NODO &DOS

Aunque si bien la memoria izquierda de un nodo <NOT> es interna la entrada izquierda de un nodo no puede venir directamente del nodo que le anteceda de una entrada o de dos entradas. Todos los nodos de dos entradas requieren que sus predecesores inmediatos distingan entre sucesores izquierdos y derechos. Puesto que los nodos de una y dos entradas no hacen esta distinción, ellos no pueden ser predecesores inmediatos de un nodo <NOT>. Los nodos de memoria pueden preceder a los nodos <NOT>, pero sería demasiado dispendioso poner un nodo de memoria solo para este propósito. En PERITO un tipo de nodo ha sido definido que no tiene el propósito sino de distinguir sus sucesores izquierdos y derechos. Estos nodos tienen tres campos, uno para el tipo del nodo (&DOS) y dos para los sucesores del nodo. Estos son muy parecidos a los nodos &MEM sin memorias:

(&DOS (. . .) (. . .)).

4.2.6 EL NODO ORDINARIO DE DOS ENTRADAS

Los nodos de dos entradas excluyendo los nodos <NOT> tienen seis campos. El primer campo contiene el tipo de nodo, &VEX. (Este nombre fue escogido para indicar que el nodo prueba Variables que están externamente acotadas). El segundo campo contiene una lista de los sucesores del nodo. El tercer y cuarto campo contienen apuntadores a los dos predecesores del nodo. Estos apuntadores son necesarios debido a que los predecesores contienen las memorias leidas por el nodo &VEX. El quinto y sexto campo son listas de vectores indice. Cada variable probada por el nodo tiene un vector de indice en cada lista. El vector en el quinto campo apunta subelementos en los datos de la izquierda. Los vectores en el sexto campo apuntan a subelementos en la parte de datos de la derecha. La producción ejemplo_j

```
ejemplo_j
     {
      meta(mono sobre X),
      estado(X cerca Y)
      -->
      afirma(meta(estado(mono X))
     }
```

tiene una variable común a ambos elementos condición. Su nodo &VEX por lo tanto tiene solo una prueba que realizar.

```
(&VEX (. . .) (. . .) (. . .) ((VARIABLE 1 2 3)) ((VARIABLE 1 1)))
```

4.2.7 EL NODO &NOT

Los nodos <NOT> contienen seis campos, la mayoria de ls cuales tienen los mismos propositos que los campos correspondientes de los nodos de dos entradas. El primer

campo contiene el tipo del nodo, este caso &NOT. El segundo campo contiene una lista de los sucesores de el nodo. El tercer campo contiene un apuntador a la memoria derecha del nodo. El cuarto campo contiene la memoria izquierda del nodo (no un apuntador a la memoria). Este campo es idntico al cuarto campo del nodo &MEM. El quinto y sexto campo contienen una lista de vectores indice como las listas en el nodo &VEX. La producción en EJEMPLO_P

```
regla ejemplo_p
  {
   meta(vacia_mano X),
   not(estado(X agarra _ )
   -->
   afirma(estado(X  agarra  platanos))
   }
```

tiene un nodo <NOT> en su red.

```
(&not (. . .) ( . . .) (()) (variable 1 2 2)) (variable 1 2)))
```

4.2.8 EL NODO &P

Los nodos que reportan los cambios que son hechos al conjunto conflicto son los nodos &P. (Ese nombre fue escogido para indicar que el nodo va a representar a la producción).

4.2.9 EL NODO &BUS

El primer nodo en la red (la raíz en la red completa) solo tiene dos campos. El primero contiene el tipo del nodo, &BUS. El segundo contiene una lista de los sucesores de el nodo.

(&BUS (. . .))

4.2.10 UNA RED DE EJEMPLO

La red para EJEMPLO_k contiene ejemplos de cada tipo de nodo excepto para &VIN.

```
regla ejemplo_k
     {
      meta(mono  agarra  X),
      no(alto(X)),
      estado(X  cerca  Y)
      -->
      afirma(meta(mono  cerca  Y))
      }
```

Cuando los nodos memoria están vacíos, la red es

```
                              (&BUS ( ) )
                                   |
   ↓                       ↓                          ↓
(&ATOMO ( ) ( 1 1) meta) (&ATOMO ( ) (1 1) alto) (&ATOMO ( )(1 1) cerc

(&LON ( ) (1 2) 3)

(&ATOMO ( ) ( 1 2 1) mono)

(&ATOMO ( ) ( 1 2 2) agarra

(&DOS ( ) ( ))                     (&MEM ( ) ( ) (()))

       (&NO ( ) ( ) (()) ((VARIABLE 1 2 3) ((NOVARIABLE 1 2)))

            (&MEM ( ) ( ) (()))            (&MEM ( ) ( ) (()))

            (&VEX ( ) ( ) ( )  ((VARIABLE 1 2 3)) ((VARIABLE 1 1)))

                   (&P ejemplo_k)
```

4.3 EL INTERPRETE DE LOS NODOS

Los nodos en una red de PERITO pueden ser consideradas instrucciones para una maquina PERITO. De la misma manera que en las computadoras convencionales se interpretan instrucciones del tipo

MOV R5,0
ADD R1,R2

la máquina PERITO interpreta instrucciones del tipo

(&VIN (. . .) (VARIABLE 1 3) (NOVARIABLE 1 4)).

Aunque las instrucciones para una computadora convencional estan almacenadas como vectores de bit's y los nodos para la maquina de PERITO estan almacenados como listas, ambos contienen información similar. El nombre en el campo del tipo de nodo corresponde a los contenidos en el campo del código OP en una instruccion convencional. El campo sucesor es similar en funcion a la campo siguiente instrucción en las viejas computadoreas de cuatro direcciones. Los otros campos son usados como campos de datos de instrucciones inmediatas.

La màquina actual de PERITO incluye un intrprete escrito en el LISP. Funciona de una menera similar aun interprete microm programado para una arquitectura convencional y el programa LISP desempeñando el rol del microcodigo. El intrprete de PERITO va por un nodo de la misma manera en que un intrprete microprogramado va por una instruccion. Examina el nombre del campo y lo despacha a la subrutina apropiada como otros que codifican el campo de codigo OP y lo mandan a las microsubrutina apropiada. Las siguiente sección explica como funciona este interprete; explicando como escoge que nodo protesar cuando hay varios esperando, y se describen las subrutinas para los varios tipos de nodos, y se explican como los datos son pasados a las subrutinas.

3.4.2 LOS NODOS DE UNA ENTRADA

El procesamiento efectuado por los nodos de una entrada en una red de PERITO es similar a la efectuada por los nodos de una entrada en el capítulo anterior. El programa que se muestra aquí es diferente solo porque la información

contenida en los nodos de PERITO ha sido descrita en mayor detalle.

Puesto que los programas para &len y %atom son idénticos excepto por las pruebas realizadas, sólo el programa para &len se muestra aquí.

1. Usando el vector indice, extrae el subelemento a ser probado de la parte de datos de el token.

2. Prueba si la longitud del subelemento es igual al parámetro constante del nodo.

3. Si la prueba tiene xito, enviar el token a los sucesores.

4. Termina.

4.3.4 EL NODO &MEM

1. Envia el token a los sucesores izquierdos.
2. Si el token está marcado como válido almacena la parte de datos en el nodo memoria; de otra manera si está marcado como inválido, encuentra y borra una parte de datos idntica.
3. Envia el token a los sucesores derechos.
4. Termina.

4.3.5 EL NODO &DOS

Los nodos dos es tambin algo nuevo en el algoritmo de PERITO.

1. envia el token a los sucesores izquierdos.

2. envía el token a los sucesores derechos.

3. termina.

4.3.6 EL NODO &VEX

Los nodos ordinarios de dos entradas difieren de aquellos descritos en el capitulo II solo en que no contienen memorias. Cuando un nodo &VER recibe un token por la izquierda, realiza el siguiente procesamiento.

1. Para cada parte de datos en la memoria derecha

Comienza.
2. Falla =0
3 Para cada variable a probar hasta que Falla=1

Comienza
4. Usa el primer vector indice para extraer un subelemento de la parte de datos izquierda.
5. Usa el segundo vector índice para extraer un subelemento de la parte de datos derecha.
6. Usando la parte variable del tipo de ambos vectores indices, escoge que prueba efectuar.
7. Efectúa la prueba.
8. Si la prueba falla, pon Falla= 1.
Termina
9. Si Falla =0.
Comienza
10. Pon D igual a la parte de datos del token de la izquierda concatenados con la parte de datos del token de la derecha.
11. Construye un token usando D y la etiqueta del token que recien ha llegado.
12. Envia el nuevo token a los sucesores.
Termina.
Termina.

13. Para.

Cuando el nodo recibe un token por la derecha, hace lo siguiente:

1. Para cada parte de datos en la memoria izquierda

Comienza

2. Falla:=0.

3. Para cada variable a probar hasta que Falla=1

Comienza.

4. Usa el primer vector indice para extraer un subelemento de la parte de datos izquierda.

5. Usa el segundo vector indice para extraer un subelemento de la parte de datos derecha.

6. Usando la parte tipo variable de ambos vectores indices, escoge que prueba efectuar.

7. Efectúa la prueba.

8. Si la prueba falla,pon Falla=1.

Termina.

9. Si falla=0

Comienza.

10. Pon D igual a la parte de datos de el token de la izquierda concatenado con la parte de datos de la derecha.

11. Construye un token usando d y la etiqueta que recin haya llegado.

12. Envia un nuevo token a los sucesores.

Termina.

Termina.

13 Para.

4.3.7 EL NODO &NOT

El nodo &NOT difiere de los nodos &NOT del capitulo anterior. Un nodo &NOT contiene solo una memoria interna en lugar de dos.

Cuando un token llega por la izquierda, el nodo &NOT realiza los siguientes pasos:

1. Contador:=0.
2. Para cada parte de datos en la memoria derecha
   Comienza
   3. Falla:=0.
   4. Para cada variable a probar hasta que Falla=1
      Comienza.
      5. Use el primer vector indice para extraer un subelemento de la parte de datos izquierda.
      6. Utilice el segundo vector indice para extraer un subelemento de la parte de datos derecha.
      7. Usando la parte del tipo de variable de ambos vectores indice, escoga que prueba realizar.
      8. Realice la prueba.
      9. Si la prueba falla, Falla:=1
      Termina.
   10. Si falla=0, Contador:= Contador+1
   Termina.
11. Si el token que recin ha llegado está marcado como VALIDO, almacena CONTADOR y la parte de datos en la memoria izquierda; de otra forma si el token está marcado como INVALIDO, borra la parte de datos idntica y su cuenta.

12. Si Cuenta=0, envia el token que recin ha llegado a sus sucesores.
13. Termina.

Cuando un nodo &NOT recibe un token por la derecha, realiza los siguientes pasos.

1. Si el token está marcado como VALIDO, Incremento:=1; de otra forma Incremento:= -1.
2. Para cada parte de datos en la memoria izquierda
   Comienza.
3. Pon Nueva_cuenta igual a la cuenta almacenada con la parte de dato.
4. Falla:=0.
5. Para cada variable a probar hasta que Falla=1
   Comienza.
   6. Usa el primer vector indice para extraer un subelemento de la parte de datos izquierda.
   7. Use el segundo vector indice para extraer un subelemento de la parte de datos derecha.
   8. Usando la parte tipo variable de ambos vectores indices, escoge que prueba realizar.
   9. Efectúa la prueba.
   10. Si la prueba falla, Falla:=1
   Termina.
11. Si Falla:=0 ,
   Comienza.
   12. Nuevo_contador:=Nuevo_contador + Incremento.
   13. Reemplace la cuenta en la memoria izquierda con Nueva_cuenta.
14 Si (Nueva_cuenta=0 y Incremento= -1) o (Nueva_cuenta=1 y Incremento=1),
   Comienza.
   15. Si Incremento=-1, Etiqueta:=VALIDO, de otra forma Etiqueta:=INVALIDO.
   16. Construye un token usando Etiqueta y la parte de datos de la izquierda.
17. Envía el nuevo token a los sucesores.
   Termina.
   Termina
Termina
18. ALTO.

4.3.8 EL NODO BUS

1. Si el cambio a la memoria de trabajo fue una eliminación, Etiqueta:='INVALIDO'; de otra forma, si fue una introducción Etiqueta:=VALIDO;
2. Construye un token usando etiqueta y el elemento de dato afectado.
3. Envía el token a los sucesores.
4. Termina.

4.3.10 PASANDO INFORMACION A LOS PROGRAMAS DE LOS NODOS

Los programas de los nodos deben de recibir tres piezas de información antes de que puedan comenzar su ejecución: la descripción del nodo a ser interpretado, el token a ser probado, y (solo en los nodos de dos entradas), una indicación de si fue llamado por su predecesor izquierdo o por su predecesor derecho. Esta sección explica como esta información es pasada a las funciones de LISP que sirven como programas de nodos.

La descripción del nodo es pasada usando el mecanismo de llamado de funciones de LISP. Los nombres de los varios programas de nodo son los nombres que aparecen en el primer campo de los nodos. Cuando la función LISP APPLY es llamada con ese nombre y el resto del nodo como su argumento, localiza al programa del nodo, sustituye los valores que aparecen en los otros campos a variables locales del programa del nodo, y pasa el control al programa.

El token y la indicación de cúal predecesor llamó al nodo son pasados en variables globales. El intrprete de perito contiene una función que mantiene un recordatorio de todos los nodos en espera más la información a ser pasada a ellos. Cada vez que uno determina su ejecución, esta función escoje otro nodo, acota los tokens y la identidad del predecesor a las dos variables globales, y luego llama APPLY para cambiar al pasar el control al programa del nodo.

C A P I T U L O
5

5.1 JUSTIFICACION EN TERMINOS DE COMPLEJIDAD PARA HACER USO DE UN ALGORITMO QUE GUARDA ESTADO.

A continuacion vamos a presentar una justificacion en terminos de complejidad que justifican el uso de un algoritmo tan sofisticado como lo es el algoritmo de RED

5.1.1 Algoritmos que guardan estado vs los que no lo salvan

Es posible dividir a los algoritmos de apareamiento (matching) para sistemas de produccion en dos categorias:

1) Los algoritmos que guardan estado.

2) Los algoritmos que no salvan estado.

Los algoritmos que salvan estado, almacenan los resultados de la ejecucion del apareamiento en los ciclos previos de *reconoce-actua*, por lo que solo los cambios hechos a la memoria de trabajo por los disparos de las producciones mas recientes necesitan ser procesados en cada ciclo. En contraste, los algoritmos salvan estado comienzan desde cero cada vez, esto es, intentan aparear la memoria completa de trabajo contra las producciones en cada ciclo.

En un algoritmo que salva estado el trabajo hecho incluye dos pasos:

(1) Computacion del estado fresco, correspondiente a los nuevos elementos memoria insertados y el almacenamiento de este nuevo estado en la estructura de datos correspondiente;

(2) Identificacion del estado correspondiente a los elementos de la memoria de trabajo y borrar este estado de la estructura de datos que la almacena.

En un algoritmo que no salva estado el trabajo incluye un solo paso, que consiste en la computación del estado del apareamiento entre la memoria completa de producciones y la memoria de trabajo. (Note que esto puede involucrar el almacenamiento temporal de algo del estado parcial que es generado) Tanto en los algoritmos que salvan estado como el los que no lo salvan, el estado se refiere a los apareamientos entre los elementos condicion y los elementos de la memoria de trabajo que son computados como un paso intermedio en el proceso de computación del apareamiento entre el *LI* de las producciones y los elementos de la memoria de trabajo.

El que sea ventajoso o no almacenar el estado depende de dos consideraciones:

1. La fraccion de la memoria de trabajo que cambia en cada ciclo, y

2. La cantidad de estado que es almacenada por el algoritmo que salva estado.

Para evaluar las ventajas y desventajas mas concretamente, considere el siguiente modelo simple. Considere un programa de sistemas de produccion para el cual el tamaño estable de la memoria de trabajo es s, el numero promedio de inserciones en la memoria de trabajo es i, y el numero promedio de eliminaciones de elementos de memoria sea d. Sea c_1 el costo para una simple insercion a la memoria de trabajo y c_2 el costo para elpara una eliminacion de un elemento de memoria. Mas aun, suponga que el costo promedio del estado temporal computado y almacenado por un algoritmo que no guarda estado sea c_3 para cada elemento de la memoria de trabajo. Por lo que el costo promedio por ciclo de ejecución es:

$$C_{guarda\text{-}estado} = i*c_1 + d*c_2.$$

Mientras que el costo promedio por ciclo de ejecucion en un algoritmo que no salva estado es

$$C_{no\text{-}salva} = s*c_3$$

Para evaluar las ventaja de los algoritmos que salvan estado, considere la desigualdad

$$C_{guarda\text{-}estado} < C_{no\text{-}guarda\text{-}estado}$$

Para la implementacion que esta siendo considerada del algoritmo de red, el costo de insertar un elemento de memoria es el mismo que el costo de borrarlo de la memoria de elementos. Como resultado, sustituyendo $c_1=c_2$ en la desigualdad obtenemos

$$(i+d)/s < c_3/c_1.$$

Estimaciones basadas en simulaciones hechas para el algoritmo de red (vease Gupta) indican que c_1 es aproximadamente igual a 1800 instrucciones de maquina, y c_3 es aproximadamente igual a 1100 instrucciones de maquina. Usando estas estimaciones obtenemos la condicion de que los algoritmos que salvan estado son mas eficientes cuando

$$(i+d)/s < 0.61,$$

esto es, el algoritmo que salva estado es mejor si el numero de inserciones mas eliminaciones por ciclo es menor al 61% del tamaño estable de la memoria de trabajo. Medidas hechas en varios programas de OPS5 muestran que el numero de inserciones mas las eliminaciones por ciclo constituyen menos un 0.5% del tamaño estable de la memoria de trabajo.

Por lo que un algoritmo que no guarda estado tendra que recuperar un factor de ineficiencia de 120 antes de que iguale a un algoritmo como el que hemos usado en esta tesis.

El siguiente ejemplo ilustra algunos de los puntos señalados en le parrafo previo. Considere un programa de sistemas de producción cuyo tamaño estable de memoria de trabajo es de cerca de 1000 elementos y en el que cada disparo de cada produccion produce de 2 a 3 cambios en la memoria de trabajo. Este es un escenario comun para los programas del tipo de los de OPS. Es bastante obvio en este caso que puesto que el 99.8% de la memoria de trabajo no cambia seria absurdo usar un algoritmo que no salve los estados previos. Pero ahora considere un programa cuyo tamaño estable de la memoria de trabajo es otra vez 1000, pero en el que cada produccion cambia 750 de los 1000 elementos de la memoria de trabajo. En este caso un algoritmo que salva estado tendra primero que identificar y luego borrar el estado correspondiente a los 750 elementos borrados y luego recomputar y almacenar el estado para los nuevos 750 elementos. Los unicos almacenamientos corresponden a los 250 elementos de memoria sin cambiar. En este caso el algoritmo que salva estado esta contraindicado.

En resumen, los algoritmos que salvan estado son apropiados cuando los elementos de memoria cambian lentamente y cuando cuesta mas recomputar el estado para la parte estable de la memoria de trabajo que borrar el estado para los elementos quitados de la memoria de trabajo.

Se han hecho estudios intensivos sobre la eficiencia del algoritmo de apareamiento de RED. Se han hecho tanto estudios analiticos (que determinan la complejidad en espacio y tiempo del algoritmo) como empiricos. Esta seccion presenta algunos resultados de los estudios analiticos.

La tabla 1 resume los resultados de los estudios analiticos en el algoritmo. La notacion usual para complejidad asimptotica es usada en la tabla. Si aparece que el costo es O(f(x)) indica que el costo varia conforme f(x) mas posiblemente un termino pequeño en x. Los terminos mas pequeños son ignorados porque el termino f(x) domina cuando x es grande. Si aparece que el costo es O(1) indica que el costo no es afectado por el factor que esta siendo considerado.

Tabla 1. Complejidad en tiempo y espacio.

Medidas de complejidad	Mejor caso	Peor caso
Efecto de la memoria de trabajo en el numero de tokens	$O(1)$	$O(W^{c})$
Efecto del tamaño de la memoria de produccion en el numero de nodos.	$O(P)$	$O(P)$
Efecto del tamaño de la memoria do produccion en el numero de tokens.	$O(1)$	$O(P)$
Efecto del tamaño de la memoria de trabajo en el tiempo de un disparo.	$O(1)$	$O(W^{2c-1})$
Efecto del tamaño de la memoria de produccion en el tiempo de un disparo.	$O(\log_2 P)$	$O(P)$

C es el numero de patrones en una produccion.

P es el numero de producciones en la memoria de produccion.

W es el numero de elementos en la memoria de trabajo.

BIBLIOGRAFÍA

[Abelson y Sussman 1984]
H. Abelson y G. J. Sussman, *Structure and Interpretation of Computer Programs*, M.I.T Press, Cambridge ,MA, 1884.

[Allen 82]
Allen,L., *YAPS:Yet another production system, technical report 1146*, Department of computer science, U.Maryland. febrero 1982.

[Anderson 76]
Anderson,J.R., *Language, Memory, and thought*, Hillsdale, NJ: Lawrence Erlbaum Associates, 1976.

[Anderson 83]
J.R Anderson. *The architecture of cognition*. Hardvard University Press.

[Barr y Feigebaum 81]
Handbook of Artificial Intelligence,Vol 1,2,3.Los altos,CA:William Kaufmann, 1981.

[Daniel 1984]
L.Daniel, Planning and Operation research, in *Artificial Intelligence: Tools,Techniques, and applications*, Harper and Row, New York,pp 423-452,1984.

[Davies 1980]

R Davis, *Metarules: Reasoning about control. Artificial Intelligence 15 (3), 179-222 (1980)*

[Davis,Lenat 1980]

Davis, Lenat 80 Davis.R y D.B Lenat, *Knowledge-Based Systems in Artificiall Intelligence.* New York: Mc Graw Hill, 1980.

[Kibler 1985]

Dennis F. Kibler y John Conery. *Parallelism in AI Programs.* IJCIA, 1985.

[Fenell 1987.]

Richard D. Fennell y Victor Lesser. *Parallelism in Artificial Intelligence Problem Solving.* IEEE Transactions on Computer, Febrero 1987.

[Forgy 1982]

C.L. Forgy, *Rete: A fast algortihm for many patterns/many object pattern match problem.* Artificial Intelligence 19,17-37 (1982).

[Georgeff 82]

Georgeff,M.P., *Procedural Control in Production Systems,* Artificial Intelligence 18, Abril 1982.

[Hayes-Roth,Waterman 83]

Hayes-Roth,Waterman,Lenat, *Building Expert systems, Reading,* MA: Addison-Wesley, 1983.

[Laird John 1984.]
John Laird. *Universal Subgoaling.* PhD thesis, Carnegie Mellon University, 1984.

[Langley 83]
Langley,P. *Exploring the space of cognotive architectures.* Beheivor Research Methods and Instrumentation, 1983.

[McDermontt 80]
McDermott J. *R1: A rule based configurer of computer systems.* Technical Reoort . Carnegie Mellon University 1980.

[McDermott, Forgy 78.]
McDermott,J., C Forgy 78 .*Production System Conflict Resolution Strategies*;in Watterman and Hayes-Roth (eds), Pattern Directed Inference. New York: Academic Press, 1978.

[Michalski y Carbonell 83.]
Michalski,R.S y J.G. Carbonell, *Machine Learning.*Palo Alto, CA: Tioga Publishing Co., 1983.

[Michalski y Carbonell 88.]
Michalski,R.S y J.G. Carbonell, *Machine Learning vol 2.3.* Palo Alto, CA: Tioga Publishing Co., 1983.

[Daniel Miranker.]
Treat: *A new and efficient Match Algorithm for AI production Systems.* PhD thesis, Columbia University, New York, 1987.

[Newell y Simon 1980.]
Human Problem Solving. Englewood Cliffs,NJ:Prentice Hall.

[Nilsson 80]
Nilsson, N.J., *Principles of Artificial Intelligence*. Palo Alto, CA: Tioga Publishing Co., 1980.

[Post 43]
Post,E.L *Formal Reductions of the General Combinatorial Desicion Problem*, American Journal Of Mathematics 65, 1943.

[Raphael 1980]
B. Raphael. *SIR: A computer Program for Semantic Information Retrieval*, in M.Minsky (ed), en *Semantic INformation Processing, MIT Press, Cambridge, MA. pp 33-145*.

[Rich 1983.]
Rich, E. *Artificcial Intelligence*. New York: McGraw Hill, 1983.

[Rosembloom 83.]
Rosembloom, P.S., *The chunking of Goal Hierarchies:A model of practice and stimulus-response compatibility*, PhD Thesis, Department of Computer Science, Carnegie Mellon University, 1983.

[Sacerdoti 1979.]
E. Sacerdoti, *Problemm solving Tactics.Procedings of the sixth* IJCAI, Tokio Japon, pp 1077-1085, 1979.

[Stefik 1985]

M Stefik. *Planning with constraints*. *Artificial Intelligence*. *16 (a)*.

[Tate 1977]

A. Tate. *Generating Project Networks*, Proceddings of the fifth IJCAI. Cambridge, MASS. pp 888-893, 1977.

[Watterman 1978]

Watterman y Hayes Roth. *Pattern-Directed Inference Systems*. New York: Academic Press. 1978.

[Winston 1984]

H.P Winston y B.K Horn. *LISP*, *2nd ed.*, Addison-Wesley, Readings. MA. 1984.