WILHELMUS MARIA JOHANNES GERARDUS RUBERG

ESPROLOG:

UM PROLOG PARA DESENVOLVIMENTO DE SISTEMAS ESPECIALISTAS

Dissertação apresentada ao Curso de MESTRADO EM SISTEMAS E COMPUTAÇÃO da Universidade Federal da Paraíba, em cumprimento às exigências para obtenção do Grau de Mestre.

HELIO DE MENEZES SILVA

Orientador

ESPROLOG:

UM PROLOG PARA DESENVOLVIMENTO DE SISTEMAS ESPECIALISTAS

WILHELMUS MARIA JOHANNES GERARDUS RUBERG

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DO CURSO DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO DA UNIVERSIDADE FEDERAL DA PARAIBA, COMO PARTE DOS REQUISITOS NECESSARIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS

Aprovado por:

HÉLIO DE MENEZES SILVA

- Presidente -

MARIA CAROLINA MONARD

- Examinadora -

PEDRO SERGIO NICOLLETTI

- Examinador -

JOSE HAMURABI NÓBREGA DE MEDEIROS

- Examinador -

CAMPINA GRANDE
ESTADO DA PARAIBA - BRASIL
DEZEMBRO - 1989

R895e Ruberg, Wilhelmus Maria Johannes Gerardus.
ESPROLOG : um PROLOG para desenvolvimento de sistemas especialistas / Wilhelmus Maria Johannes Gerardus Ruberg. - Campina Grande, 1989.
85 f.

Dissertação (Mestrado em Sistemas e Computação) - Universidade Federal da Paraíba, Centro de Ciências e Tecnologia, 1989.
"Orientação : Prof. Hélio de Menezes Silva".
Referências.

1. Sistemas Especialistas - Desenvolvimento. 2. PROLOG. 3. Modelo Computacional - Raciocínio Humano. 4. Dissertação - Sistemas e Computação. I. Silva, Hélio de Menezes. II. Universidade Federal da Paraíba - Campina Grande (PB). III. Título

CDU 004.891(043)

ESPROLOG:

UM PROLOG PARA DESENVOLVIMENTO DE SISTEMAS ESPECIALISTAS

WILHELMUS MARIA JOHANNES GERARDUS RUBERG

DISSERTAÇÃO APROVADA EM 07 / 12 / 1989

HÉLIO DE MENEZES SILVA

- Orientador -

MARIA CAROLINA MONARD

- Componente da Banca -

PEDRO SERGIO NICOLLETTI

- Co-orientador -

JOSE HAMURABI NÓBREGA DE MEDEIROS

- Componente da Banca -

AGRADECIMENTOS

Ao Prof. Hélio de Menezes Silva, pela orientação dessa dissertação, repleta de dedicação e amizade.

Ao Prof. Pedro Sergio Nicolletti, pelas valiosas discussões e sugestões oferecidas

À Prof. Dra Maria Carolina Monard, do Instituto de Ciências Matemáticas, USP-São Carlos, e ao Prof. José Hamurabi Nobrega de Medeiros, do Departamento de Sistemas e Computação da UFPb, que muito me honraram aceitando compor a banca dessa dissertação.

À IBM do Brasil pela ajuda financeira concedida.

Para Angela, Nicolaas, Adriana e Claudia

ESPROLOG

UM PROLOG PARA DESENVOLVIMENTO DE SISTEMAS ESPECIALISTAS

RESUMO

Embora crescentemente usada no desenvolvimento de Sistemas Especialistas (S.E.), PROLOG só faz inferências "quantitativas", obrigando-nos a escrever módulos de inferência "quantitativa" que lidam com "graus de certeza" entre FALSO e VERDADEIRO e com "fatores de atenuação" associados às regras. No presente trabalho é descrita a implementação de um PROLOG que trata as incertezas diretamente, passando a constituir uma ferramenta para desenvolvimento de S.E.

ABSTRACT

Althoug increasingly used in developing Expert Systems, PROLOG only does "qualitative" inference, forcing us to write "qualitative" inference engines, diling with "certainty values" between FALSE and TRUE, and with "attenuation factors" associated to the rules. In this work is described a PROLOG implementation that deals with uncertainties directly, starting to be a tool for Expert Systems development.

SUMARIO

INDICE DE FIGURAS

1. INTRODUÇÃO

1.1 Sistemas Especialistas

Um sistema especialista (S.E.) é aquele que trabalha com problemas complexos do mundo real que requerem a interpretação de um especialista. Os S.E.'s solucionam problemas através do uso de um modelo computacional do raciocínio humano, chegando idealmente às mesmas conclusões que um especialista chegaria caso se defrontasse com um problema comparável [WEIS 88].

O Especialista ou Perito é a fonte do conhecimento para os S.E.'s. Essa fonte de conhecimento é, em geral, constituida de uma ou mais pessoas do mais alto grau de conhecimento e experiência em uma área. O Engenheiro de Conhecimento é o profissional da área da Ciência da Computação que concebe e constroi um S.E., colocando nele todo o conhecimento do Perito, obtido através de entrevistas e proposições de problemas [FEIG 81].

O conhecimento é a informação, acrescida de sua forma de utilização, que um S.E. manipula para resolver problemas

A representação do conhecimento de um S.E. é feita através de diversos formalismos. Os mais usuais [FEIG 81] são:

1) regras do tipo : SE - ENTÃO

que são adequadas para representar recomendações, diretivas e/ou estratégias.

2) redes semânticas, que são grafos compostos de nós (representando conceitos, objetos ou eventos) e arcos (representando as relações entre os nós), adequadas para o conhecimento mais naturalmente expresso em termos de classificações e taxonomias.

3) quadros, que são estruturas estáticas que guardam informação do que fazer sob certas situações/comandos ou mesmo a atitude a tomar se não se sabe o que fazer.

A Figura 1.1 mostra os componentes envolvidos num S.E.

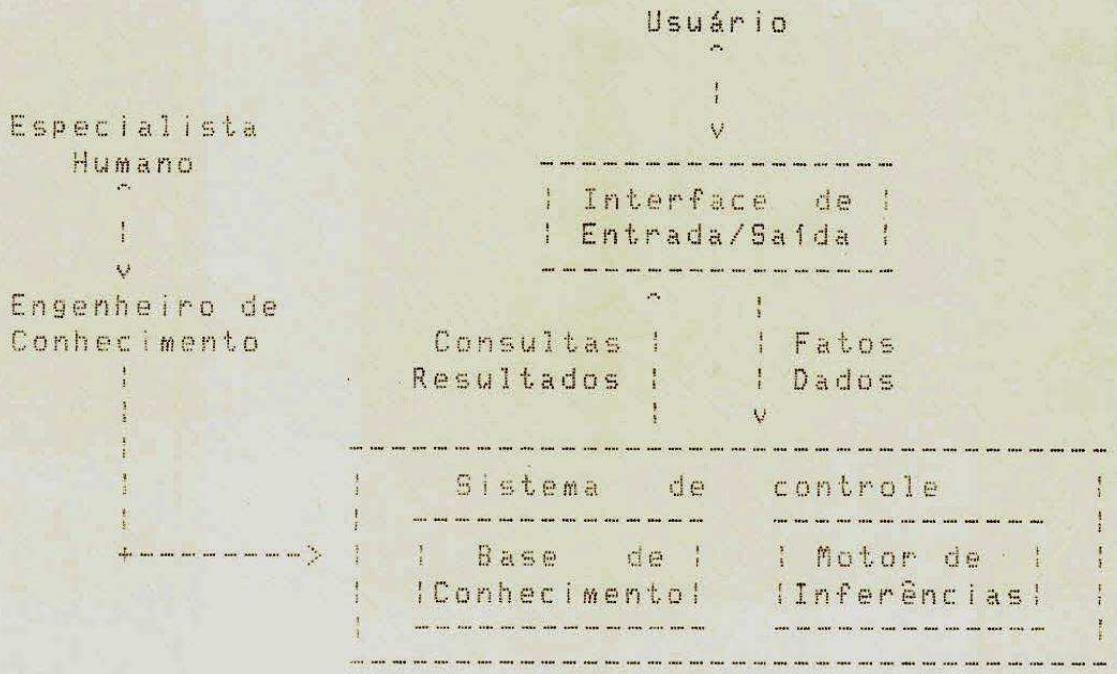

FIG 1.1 Estrutura de um S.E.

1.2 Ferramentas para a construção de S.E.'s.

O grande gargalo na construção de S.E.'s está na aquisição do conhecimento, ou seja, na interação existente entre o engenheiro de conhecimento e o perito [WATE 85]. Para minimizar esse gargalo são muito importantes as ferramentas específicas para a construção de S.E.'s, principalmente se elas permitem a construção de protótipos que possam ser facilmente validados pelo perito ou mesmo serem construídos por ele.

As primeiras ferramentas desenvolvidas foram do tipo "esqueleto", derivadas de S.E.'s já existentes de onde foi removido o conhecimento. As ferramentas do tipo "esqueleto" provêem estruturas e facilidades de construção que tornam o desenvolvimento de um S.E. mais fácil e mais rápido, perdendo, por outro lado, em generalidade e flexibilidade, pois só podem trabalhar com uma classe restrita de problemas [WATE 85]. Como exemplo de alguns "esqueletos" podemos citar: o EMYCIN (derivado do MYCIN) [MELL 74], KAS (derivado do PROSPECTOR) [DUDA 81], EXPERT (derivado do CASNET) [WEIS 79].

As linguagens de representação de propósito geral são linguagens desenvolvidas especialmente para o engenheiro de conhecimento. Provêem maior controle nos acessos aos dados e na inferência do que os sistemas tipo "esqueleto", mas também são mais difíceis de usar. Como exemplos desse tipo de ferramentas temos a linguagem ROSIE [FAIN 82], OPS5 [FORG 81], RLL [GREI 80].

A linguagem de programação PROLOG tem sido dia a dia mais

utilizada no desenvolvimento de S.E.'s. Isto se deve ao fato de ser denotacional, ter mecanismos de inferência usando "backtracking" e casamento de padrões, além de ter uma eficiência comparável a LISP. No entanto, PROLOG só faz inferências "qualitativas" (que envolvem apenas os valores "verdadeiro" e "falso", e implicação no sentido da lógica clássica). Os S.E.'s, em sua maioria, necessitam de inferências "quantitativas" (que envolvem um contínuo de valores entre "verdadeiro" e "falso", e implicações com fatores de atenuação associados a elas). Desta forma, os S.E.'s, escritos em PROLOG exigem que um módulo de inferência quantitativa seja escrito nessa linguagem, com prejuízo de tempo de programação e execução [RUBE 85].

1.3 Objetivos da dissertação

ESPROLOG é um PROLOG que lida diretamente com inferências "quantitativas", tendo a inferência "qualitativa" como caso particular. Isso torna ESPROLOG uma ferramenta adequada para o desenvolvimento de S.E.'s.

Principais vantagens de ESPROLOG:

1) Como PROLOG é uma linguagem popular entre os estudiosos de I.A., não há necessidade de investir tempo ou mesmo enfrentar dificuldades como as existentes nas linguagens de propósito geral para representar o conhecimento. ESPROLOG permite que se construam protótipos que poderão ser validados pelo perito. Além

do mais, ESPROLOG permite a construção de interfaces que permitem que as regras e os fatos sejam digitados numa linguagem muito próxima à natural, permitindo que o próprio perito use a ferramenta diretamente.

2) As otimizações existentes nos principais interpretadores PROLOG também estão presentes em ESPROLOG. A otimização de **submetas determinísticas**, a de **recursão de cauda** e a **de última chamada**, têem a função de recuperar espaço na pilha de variáveis. O coletor de lixo permite recuperar espaço na estrutura de armazenamento de termos simples e compostos. Além das otimizações existentes no PROLOG, ESPROLOG apresenta mais uma denominada **poda por limiar de aceitação.** Esta otimização abandona um caminho na busca de uma solução, se esta não puder ser resolvida com o grau de certeza estipulado pelo usuário. Todas as otimizações serão detalhadas no capítulo 4.

3) ESPROLOG apresenta um bom conjunto de predicados pré-definidos destinados a facilitar o trabalho do usuário. Além dos predicados normalmente existentes nos principais interpretadores, ESPROLOG apresenta outros predicados específicos para sistemas especialistas. Destacamos que, com o uso de consultas, do predicado resolva, do predicado NOT e da otimização de poda por limiar de aceitação, a geração de soluções é feita de maneira racional e rápida.

O trabalho de construção do interpretador ESPROLOG a partir do nada pareceu-nos grande demais e desnecessário. Assim, tomamos partido do oferecimento do código fonte de um PROLOG desenvolvido

por Allan Roger CROGE 85]. O mesmo foi estudado, testado, algumas rotinas foram reescritas e alguns "bugss" e omissões foram sanados. Em seguida foram introduzidas as modificações que permitem ao interpretador resolver inferências quantitativas.

1.4 Esboço da dissertação

O capítulo 2 contém uma descrição informal da linguagem e de seu uso. Esse capítulo é destinado aos usuários que não têem o conhecimento prévio da linguagem PROLOG.

O capítulo 3 contém a sintaxe da linguagem, com exemplos quando necessário, além da descrição dos predicados pré-definidos.

As estruturas de dados e a descrição dos principais algoritmos do motor de inferência estão no capítulo 4.

Finalmente, no capítulo 5, são apresentadas as conclusões e sugestões para trabalhos futuros. É também feita uma avaliação crítica do funcionamento do interpretador e a comparação do tempo de resposta com o PROLOG que deu origem ao ESPROLOG e com o ARITY-PROLOG 4.0. Também foram feitos testes comparativos usando dois pequenos Sistemas Especialistas.

O anexo 1 consta das listagens dos programas ESPROLOG usados nos testes comparativos.

2. TUTORIAL BASICO

2.1 Introdução a ESPROLOG

ESPROLOG é, essencialmente, uma linguagem para representação e manipulação de conhecimento. Um programa ESPROLOG consiste de informações sobre um certo domínio. É permitida a formulação de perguntas, que o interpretador tenta responder usando o conhecimento e os fatos disponíveis e, caso estes últimos sejam insuficientes, o próprio interpretador solicita novos fatos ao usuário.

O interpretador utiliza as técnicas de **encadeamento para trás** ("back chain") a de **rastreamento para trás** ("backtracking") [NILS 82]. No encadeamento para trás uma consulta ou meta inicial é assumida como verdadeira. A partir da meta inicial, utilizando-se um banco de regras e fatos, geram-se novas metas que possam culminar com a prova da meta inicial. Caso uma das metas intermediárias não possa ser tornada verdadeira, mesmo perguntando-se pelos fatos faltantes, o rastreamento para trás é utilizado para desfazer essa meta, substituindo-a por uma nova. Essas características fazem com que ESPROLOG forneça uma maneira natural para implementar sistemas especialistas baseados em regras [BUCH 84].

2.2 Regras ESPROLOG

As regras ESPROLOG correspondem às cláusulas PROLOG [CASA 87], e podem ser: **Cláusulas Fato**, **Cláusulas Meta** e **Cláusulas Implicação**.

2.2.1 Cláusula fato

Uma cláusula fato tem a forma:

((p t1 ... tn) GC)

Onde **p** é uma relação e **t1 ... tn** (n >= Ø) são termos, que podem ser números , constantes , variáveis ou funções.

Uma cláusula fato representa um fato relativo a um determinado domínio de conhecimento. O grau de certeza GC representa a confiança que se tem nesse fato. GC pode assumir valores no intervalo fechado -1ØØ a 1ØØ, sendo que GC = -1ØØ significa a negação do fato, GC = Ø quer dizer que nada se sabe a respeito e GC = 1ØØ representa a confirmação absoluta do fato.

Exemplo : ((gosta João queijo) 9Ø)

Significado: João gosta muito de queijo, uma vez que 9Ø está muito próximo de 1ØØ, que é a confirmação absoluta desse fato.

Se numa cláusula fato, o grau de certeza não estiver presente, o interpretador assumirá que seu valor é 1ØØ.

2.2.2 Cláusula meta ou consulta

Uma cláusula meta ou consulta tem a forma:

?(LA (p1 t11 ... t1m)(p2 t21 ... t2m) ... (pj tj1 ... tjp))

com j >= 1, -100 <= LA <= 100 e n, m, p >= 0 .

Como o próprio nome indica, a cláusula meta corresponde a uma consulta (atômica ou conjuntiva). A consulta terá sucesso se o interpretador conseguir resolvê-la com GC maior ou igual ao limiar de aceitação LA, estipulado pelo usuário. Caso o usuário omita LA, o interpretador assumirá que seu valor é 20.

Exemplo: ?(40 (gosta _alguem queijo))

Significado: Quem gosta de queijo com grau de certeza maior ou igual a 40 ?

2.2.3 Cláusula implicação

A cláusula implicação permite expressar uma implicação entre relações existentes num domínio. Ela pode representar o conhecimento, em geral, tal como definições, dependências de causa e efeito, heurísticas, restrições, etc.

Uma cláusula implicação tem a forma :

((A t1 ... tn) FA (B1 t11 ... t1p) ... (Bj tj1 ... tjq))

com j >= 1, n, p, ... ,q >= 0 e -100 <= FA <= +100 .

Lê-se: (A ...) com fator de atenuação FA, se (B1 ...), (B2

conclusão da cláusula e os Bi's (i>=1) representam uma conjunção de condições. FA fornece a confiança que se terá na conclusão da regra quando as condições da mesma forem satisfeitas. Assim sendo, um FA negativo indica uma predominância de evidências contrárias à conclusão da cláusula; um FA positivo reforça essa conclusão. Os valores de FA, tal qual os graus de certeza, variam de -100 a 100. Se o FA não estiver presente na cláusula, o interpretador assumirá que seu valor é 100.

Exemplo : ((queijo montrachet) 80 (finalidade aperitivo)
(preferência suave)(consistência macio))

O grau de certeza resultante da cláusula implicação acima é dado pela expressão :

GC(A) = FA * mínimo (GC(Bi)) [BUCH 84]

A conclusão de uma regra será considerada verdadeira se a conjunção de suas condições for verdadeira. Deve-se salientar que o conceito de verdadeiro não é o da lógica clássica. Uma conclusão será considerada verdadeira se tiver GC maior ou igual ao limar de aceitação LA estipulado pelo usuário, lembrando que, se o usuário omitir o valor de LA, este será considerado como 20.

2.3 Programa ESPROLOG

Um programa ESPROLOG é um conjunto finito de grupos de regras. Cada grupo de regras é armazenado automaticamente numa tabela "HASH" para que possa ser recuperado de maneira eficiente.

Um grupo de regras é um conjunto de cláusulas de mesma conclusão. As regras de um mesmo grupo são armazenadas automaticamente em ordem decrescente de acordo com o fator de atenuação (ou GC no caso de fatos). Como veremos mais adiante essa, ordenação faz com que o interpretador não tente provar hipóteses que de antemão se saiba que terá GC < LA.

Como já foi visto, o GC de uma cláusula fato, o FA de uma cláusula implicação e LA de uma cláusula meta, não precisam ser especificados. Nesse caso o interpretador atribui o valor 1Ø0 tanto para GC como para FA e atribui 2Ø para LA. Isso faz com que ESPROLOG aceite e processe programas PROLOG.

2.4 Alguns exemplos

A sintaxe e o funcionamento do interpretador podem ser melhor compreendidos a partir de exemplos simples.

O fato de João gostar pouco de queijo pode ser representado pela cláusula:

((gosta João queijo) 3Ø)

Com esse programa pode-se perguntar ao interpretador se João

gosta de queijo, digitando:

?((gosta João queijo))

Feita a consulta, o interpretador exibirá na saida padrão:

SIM
GRAU DE CERTEZA = 30

Se a pergunta for:

?(60(gosta Pedro queijo))

O interpretador não poderá responder, e emitirá a seguinte mensagem:

DIGITE O GRAU DE CERTEZA DO FATO

(gosta Pedro queijo)

Se não se sabe se Pedro gosta de queijo digita-se "0", e tem-se a resposta:

NÃO

Caso Pedro goste muito de queijo, pode-se digitar, por exemplo, 95. Agora o interpretador responderá afirmativamente e colocará esse fato no banco de dados.

Perguntas tais como: "quem gosta de queijo?" , podem ser feitas digitando:

?((gosta _x queijo))

O fato de x vir precedido de "_" significa que x é uma variável. Sempre que uma consulta contiver variáveis, ESPROLOG tentará associar um valor às mesmas. A consulta acima terá como resposta:

SIM

_x = Pedro

GRAU DE CERTEZA = 95

DESEJA OUTRA RESPOSTA? (S/N)

Digitando S, o interpretador responderá:

SIM

_x = João

GRAU DE CERTEZA = 30

DESEJA OUTRA RESPOSTA? (S/N)

Mais uma vez digitando S tem-se a mensagem:

NÃO HA MAIS RESPOSTAS

2.5 Resolução

As consultas em ESPROLOG são respondidas com base no processo de resolução desenvolvido por Robinson [ROBI 65].

Seja o seguinte programa (a numeração serve para referência posterior):

(1) ((queijo montrachet) 80 (finalidade aperitivo)
(preferencia suave)(consistencia macio))

(2) ((queijo gorgonzola) 75 (finalidade aperitivo)
(preferencia picante)(consistencia macio))

(3) ((preferencia picante) 90)

(4) ((preferencia suave) 60)

(5) ((preferencia aromatico) 50)

(6) ((consistencia macio) 95)

Considere a meta (M-0), chamada de meta inicial.

(M-0) ?(60 (queijo _tipo))

A construção por etapas da árvore de provas [FERG 81] do programa exemplo mostra como a meta inicial vai se transformando até atingirmos uma solução.

A árvore de prova é iniciada com a criação do nó raiz, ao qual está unida a meta inicial (nó incompleto). A cada nó completo tem-se associada uma variável teto (T) [SILV 86] que guarda o valor máximo que o GC dessa meta pode atingir. Assim sendo, o teto associado ao nó raiz é 100 (Figura 2.1).

FIG 2.1 Arvore de prova

A cabeça da cláusula (1) (queijo montrachet) pode ser associada ao nó incompleto (queijo _tipo). Essa associação recebe o nome de instanciação. Para que a instanciação de M-Ø e a cláusula (1) seja possível, é necessário fazer a associação da variável "_tipo" a "montrachet".

O teto do novo nó completado é dado pelo teto do nó que lhe deu origem multiplicado pelo FA da cláusula (1) e dividido por 1ØØ.

O corpo da cláusula (1) (finalidade aperitivo) (preferencia suave) (consistencia macio) passa a ser a nova meta(M-1). (Figura 2.2)

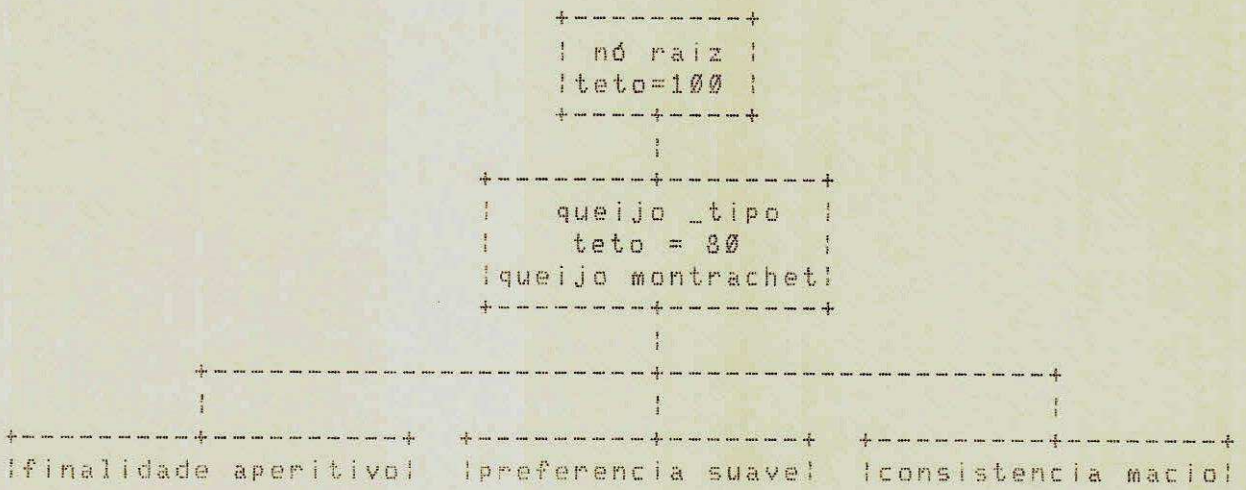

FIG 2.2 Árvore de prova

Como a meta (M-1) tem três submetas, o interpretador tentará resolver primeiro a submeta mais à esquerda. Como não existe nenhuma cláusula que possa casar com essa submeta, teremos a mensagem:

DIGITE O GRAU DE CERTEZA DO FATO

(finalidade aperitivo)

Se o queijo for para ser servido como aperitivo, o usuário deverá digitar 100. O interpretador colocará esse fato no banco de dados (cláusula (7)), e completará a prova da submeta (Figura 2.3).

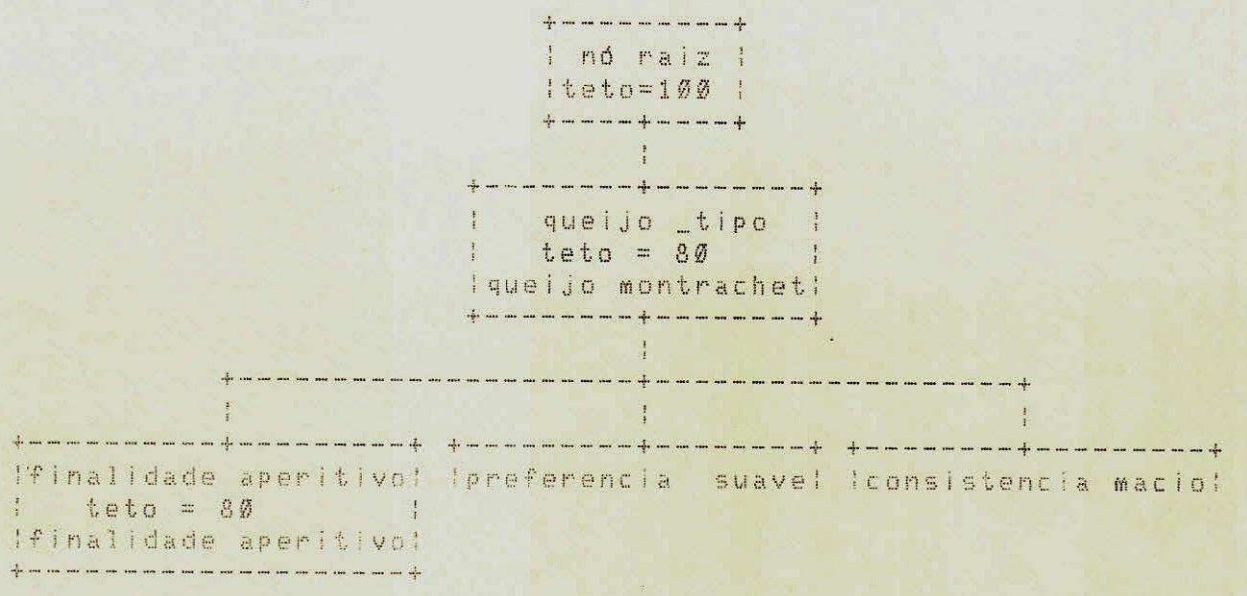

FIG 2.3 Arvore de prova

Para provar a segunda submeta de (M-1), recorre-se às cláusulas (3), (4) e (5). A regra (3) não poderá ser feita idêntica à submeta atual, porque **suave** não unifica com **picante.** O interpretador tenta a unificação com a cláusula (4) que, apesar de ser idêntica à meta, também falha por ter teto dado por 80*60/100 = 48 (inferior ao limiar de aceitação LA = 60 estipulado pelo usuário).

As regras são ordenadas decrescentemente de acordo com seu

fator de atenuação (se forem cláusulas implicação) e, em ordem decrescente com o GC (se forem fatos). Por esse motivo, sempre que houver uma falha por LA, mesmo havendo ainda cláusulas para serem tentadas, elas são abandonadas, pois também terão tetos inferiores ao limiar de aceitação.

O interpretador fará um rastreamento para trás (backtracking), até atingir uma meta que possa ser instanciada com outra cláusula (Figura 2.4).

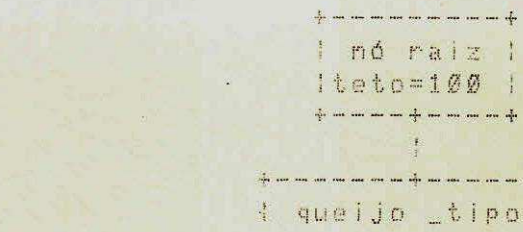

FIG 2.4 Arvore de prova

E tentada a cláusula (2) - (Figura 2.5)

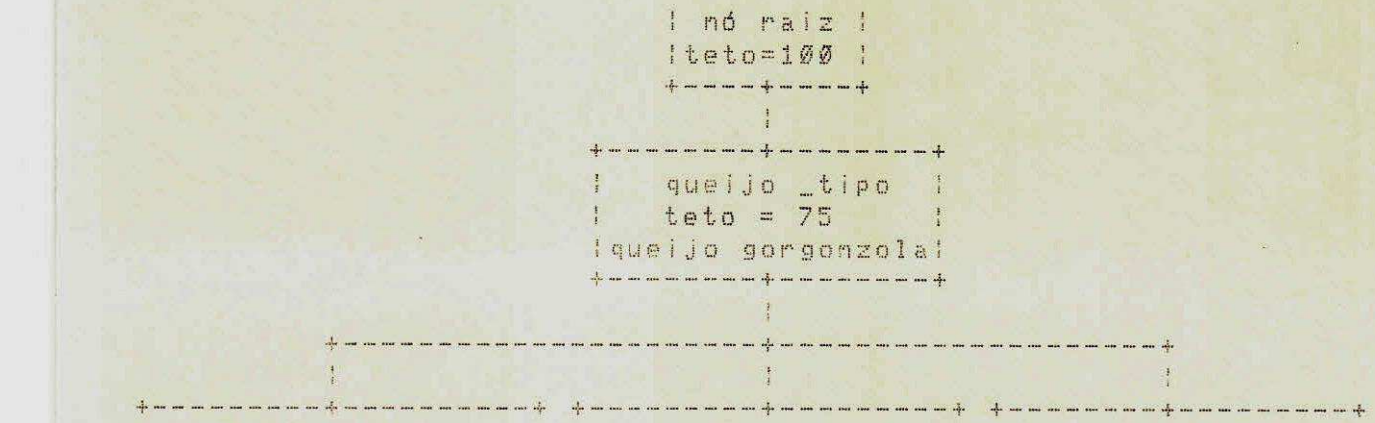

FIG 2.5 Arvore de prova

As submetas (finalidade aperitivo) (preferência picante) (consistência macio) poderão ser instanciadas com as claúsulas (7), (3) e (6), respectivamente, completando a árvore de prova (Figura 2.6).

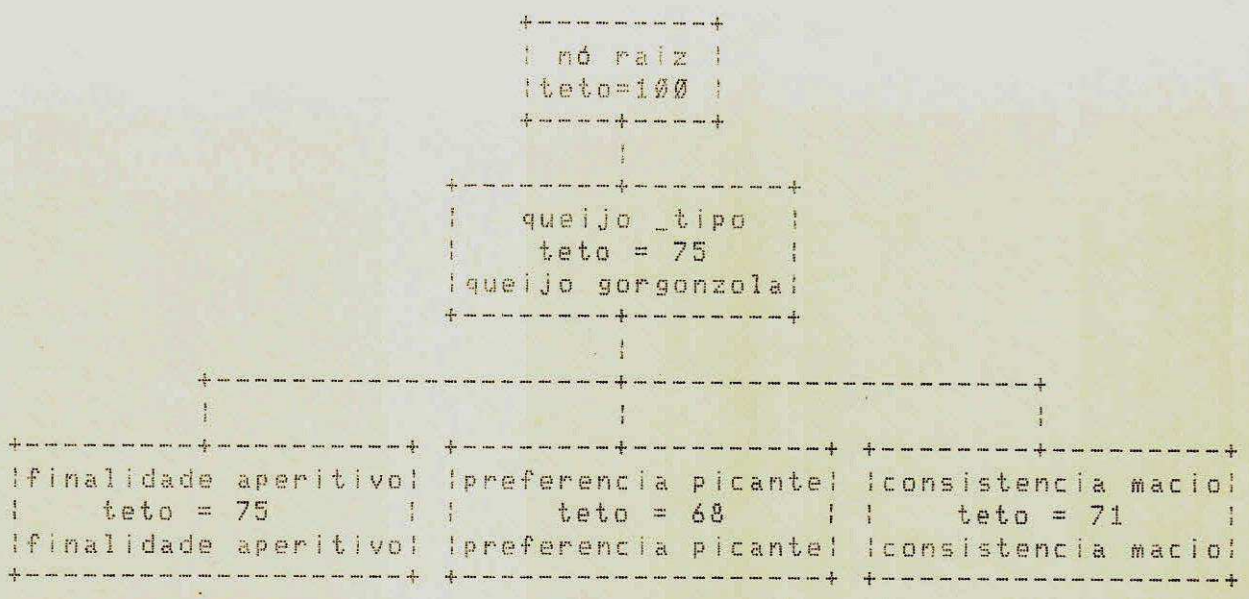

FIG 2.6 Arvore de prova

A meta inicial foi resolvida com sucesso, uma vez que não há mais metas a serem provadas. Como o grau de certeza de uma consulta é dado pelo menor dos tetos, tem-se na saída padrão:

```
SIM
_tipo = gorgonzola
GRAU DE CERTEZA = 60
DESEJA OUTRA RESPOSTA (S/N)?
```

2.6 Utilização do interpretador

Digitando-se ESPROLOG e a tecla return, o programa executável é carregado. Na saída padrão aparecerá a mensagem:

ESPROLOG - UM PROLOG PARA DESENVOLVIMENTO
DE SISTEMAS ESPECIALISTAS.

O interpretador passa a analisar as cláusulas que vierem a ser introduzidas entrada padrão. Sempre que o programa estiver pronto para receber informações, posicionará o cursor no início da linha precedido do símbolo "$".

O usuário deverá teclar "Enter" após ter digitado uma cláusula ou preenchido uma linha. O interpretador verificará se existe erro no que foi digitado, emitindo mensagem apropriada se for o caso. Não havendo erro, o interpretador verificará se foi digitada uma regra completa. Em caso afirmativo, esta será armazenada ou resolvida (se for uma consulta). Completada a tarefa, teremos na tela "$", seguido do cursor, indicando que o interpretador está apto a receber outra cláusula. Se apenas uma parte da cláusula foi digitada, aparecerá na tela do terminal o número de parênteses que faltam para completar a cláusula, seguido do "$", indicando que o interpretador está pronto para receber o restante da cláusula.

3. MANUAL DO USUARIO

3.1 Sintaxe

Os programas ESPROLOG são constituídos a partir de termos. Termos podem ser constantes, variáveis e termos compostos. Cada termo é formado por uma sequência de caracteres, obedecendo certas regras. Esses caracteres estão divididos em quatro grupos:

Letras maiúsculas

A B C D .....Z

Letras minúsculas

a b c d .....z

Dígitos

0 1 2 3 9

Símbolos especiais

+ - * / \ ? . $ % & ( ) _ + ¦ / \ [ ] { } ! # @ : < >

3.1.1 Constantes

Constantes são usadas para especificar objetos. Há dois tipos de constantes: átomos, inteiros.

Destacamos dois tipos de átomos: aqueles formados por letras e dígitos e aqueles formados por símbolos especiais.

O primeiro tipo é iniciado por uma letra maiúscula ou minúscula seguida de letras e/ou dígitos. O caracter especial "_" pode ser inserido no interior de um átomo para melhorar sua legibilidade.

Exemplos: aux1 dia_mes Pai S1S2

Os átomos constituídos por símbolos especiais são formados por um único desses símbolos.

Exemplos: + | < ( ) [ ] ?

Às vezes, por questões de legibilidade ou para evidenciar associações mneumônicas, os átomos não obedecem às regras acima. Isso é permitido desde que o átomo seja iniciado e terminado por aspas (").

Exemplos: "dia mes ano" "a+b"

O numero máximo de caracteres permitido num átomo é treze. Caso o usuário ultrapasse esse limite, os símbolos excedentes serão ignorados.

A constante inteira é usada para representar números. Os números inteiros podem variar de -32768 a 32767.

3.1.2 Variáveis

Uma variável é uma cadeia formada por uma letra seguida de letras e/ou dígitos precedidos pelo caracter "_" que também

pode aparecer no interior da variável.

O escopo de uma variável se reduz à cláusula em que ela aparece. Se duas ou mais cláusulas contêem variáveis de mesmo nome, elas são tratadas independentemente.

São exemplos de variáveis:

_data _dia

_var1 _dia_mes

O símbolo "_" isoladamente representa uma variável chamada de variável anônima. ESPROLOG não atribui nenhum valor a essa variável [CLOC 81].

Exemplo:

```
((funcionario Joao_Alves solteiro (12 04 49) pintor    ))
((funcionario Jose_Dias  casado   (23 11 51) motorista ))
     .              .              .              .
     .              .              .              .
     .              .              .              .
((funcionario Pedro_Silva casado  (09 02 48) engenheiro))
```

No exemplo acima, para separar os funcionários casados, digita-se:

```
?((funcionario _nome casado _ _))
```

O interpretador não atribuirá nenhum valor às variáveis anônimas, o que na prática significa uma economia em tempo de execução pois o interpretador não perderá tempo em tentar

associar um valor a essa variável. Além disso também não são alocados espaços para a variável anônima na pilha de variáveis.

3.1.3 Termos compostos

Os termos compostos ajudam a organizar os dados de um programa, porque permitem que um grupo de informações seja tratado como um único elemento. Os termos compostos podem ser listas, predicados e cláusulas.

Uma lista inicia-se por um abre parêntese seguido de zero ou mais elementos da lista e termina com um fecha parêntese. Os elementos de uma lista podem ser termos simples ou compostos. Um ou mais espaços em branco são utilizados para separar os elementos de uma lista.

São exemplos de listas:

```
( _nome _cargo _data_nasc )
(Joao_Santos Mestre_obras ( 23 10 1953 ))
```

A lista é uma estrutura de dados muito comum na programação não numérica. Uma lista é uma sequência ordenada de elementos, que pode ter qualquer comprimento. As listas podem representar praticamente todos os tipos de estruturas necessárias na computação simbólica. Listas são largamente empregadas para representar gramáticas, mapas de cidades, programas de computador e entidades matemáticas tais como grafos, fórmulas e funções.

O operador infixo "|" é definido para permitir a manipulação de listas de tamanho variável e lê-se "seguido pela lista..." [CLAR 84].

(x1 x2 .... xn|_z) é uma lista cujos n primeiros elementos são x1 ... xn (com n >= 1) seguidos pelo resto da lista _z. Como _z pode estar associado à lista vazia "()" (x1 .... xn|_z) tem no mínimo n elementos.

Um predicado é uma lista cujo primeiro termo é um átomo, o qual dá nome ao predicado. O número de argumentos (termos simples ou compostos) do predicado fornecem sua aridade. Dois predicados de mesmo nome deverão ter a mesma aridade. O predicado é utilizado para especificar relações.

São exemplos de predicados:

(gosta Joao vinho)

(soma 2 4 6)

(preferencia _x)

As cláusulas são clasificadas em: cláusula implicação, cláusula fato e cláusula meta.

A cláusula implicação é uma lista cujo primeiro elemento é um predicado chamado cabeça da cláusula. À cabeça segue-se o fator de atenuação (FA). Os predicados restantes constituem o corpo da cláusula. Se o FA de uma cláusula for igual a 1Ø0 poderá ser omitido.

Exemplos:

```
((queijo gorgonzola) 75 (finalidade aperitivo)
    (preferencia picante)(consistencia macio))
((sucessor _x _y)(add _x 1 _y))
```

Uma cláusula fato não tem corpo. Após a cabeça segue-se o grau de certeza (GC) (que pode ser omitido caso seja igual a 100).

Exemplos:

```
((finalidade aperitivo) 90)
((preferencia picante))
```

Cláusula meta é uma cláusula sem cabeça. O limiar de aceitação (LA) precede o primeiro predicado do corpo da cláusula, podendo ser omitido caso seja igual a 20. A clásula meta é identificada por uma interrogação colocada no seu início.

Exemplos:

```
?(80 (queijo _tipo))
?((add 2 3 _x))
```

3.2 Predicados pré-definidos

O interpretador ESPROLOG fornece uma série de predicados pré-definidos, que visam facilitar o trabalho do usuário. Eles também têem a função de executar tarefas que normalmente seriam difíceis ou mesmo impossíveis de serem concretizadas a partir de

cláusulas definidas pelo usuário.

3.2.1 Predicados de entrada e saída

(consult arg1)

Permite ler um arquivo de cláusulas, colocando-as na tabela de cláusulas. Caso o arquivo contenha cláusulas metas, essas serão resolvidas.

O argumento "arg1" desse predicado é o nome do arquivo. O nome de um arquivo é composto de um prefixo seguido de um ponto e de um sufixo (terminologia do MS-DOS). O prefixo fornece o nome do arquivo e o sufixo informa o tipo do arquivo.

A referência a um dado arquivo de cláusulas é feita digitando seu prefixo, ou seu nome completo entre aspas.

(output arg1)

Constrói um arquivo de saída, no qual são colocadas todas as mensagens do interpretador para o usuário. O arquivo de saída deve terminar com o sufixo "out".

(see arg1)

Abre um arquivo de entrada.

(tell arg1)

Abre um arquivo de saída.

(seeing arg1)

Retorna no argumento de chamada o nome do arquivo que está aberto para entrada.

(telling arg1)

Retorna no argumento de chamada o nome do arquivo que está aberto para saída.

(seen)

Fecha o arquivo de entrada.

(told)

Fecha o arquivo de saída.

(ttyget arg1)

Lê um caracter da console colocando-o em arg1.

(ttygetØ arg1)

Lê um caracter não branco da console.

(ttyput arg1)

Escreve no terminal o caracter associado a arg1.

(get arg1)

Lê um caracter do arquivo de entrada.

(getØ arg1)

Lê um caracter não branco do arquivo de entrada.

(put arg1)

Escreve um caracter no arquivo de saída.

(read_term arg1)

Lê um termo do arquivo de entrada.

(put_terms arg1 arg2 ... argn)

Escreve os termos argi (1 <= i <= 12) no arquivo de saída.

3.2.2 Predicados de execução e controle

(fail)

Retorna fracasso.

(true)

Retorna sucesso.

(NOT arg1)

Verifica se a negação de arg1 é verdadeira. Isso ocorre se arg1 for um fato que tenha -GC >= LA (onde LA é o limiar de aceitação) ou se arg1 puder ser unificado a uma cláusula de -FA >= LA.

Exemplo: seja o seguinte banco de fatos e regras:

((salar_minimo Joao))

significado: Joao ganha um salário mínimo com GC 100.

((outro_rend Joao)-100)

Significado: Joao tem outros rendimentos com GC -100.

((rico _x) -90 (salar_minimo _x)(NOT(outro_rend _x)))

Significado: x é rico com FA -90 se x ganha um salário mínimo e não tem outros rendimentos.

Se fizermos a consulta ?((Not (rico Joao))

Teremos como resposta SIM com GC = 90, isso porque a hipótese inicial (NOT (rico Joao)) pode ser unificado com sucesso a (rico _x). A nova hipótese ((salar_minimo Joao)(NOT (outro_rend Joao)) também será unificada com sucesso.

(resolva arg1 arg2)

Esse predicado obtém o GC resultante do predicado associado a arg1, colocando seu valor na variável arg2. Com uma cláusula meta é possível obter as soluções de um problema. Quanto maior o GC da solução, tanto mais confiável ela deve ser. Por outro lado, às vezes é possível chegar à mesma solução, por caminhos distintos e, se esses caminhos forem independentes, a confiança que se tem na solução deve aumentar.

Com o predicado "resolva" , todos os caminhos que levam à mesma solução são levantados e os graus de certeza desta são acumulados. Esse predicado também levanta as evidências contrárias a essa solução fazendo diminuir a confiança na mesma.

Obtidos dois graus de certeza (X e Y) usando-se duas cláusulas de mesma conclusão, eles serão compostos em um só, segundo os critérios desenvolvidos por Shortliffe [BUCH 84].

X + (100 - X) * Y / 100 se X > 0 e Y > 0

X + Y se X < 0 e Y > 0 ou X > 0 e Y < 0

-(-X (100 + X) * (-Y) /100 se X<0 e Y<0

(!)

Impede o rastreamento para trás. O predicado corte pode ser usado por três motivos distintos [CLOC 84] :

1) Para indicar ao interpretador que um bom caminho foi encontrado, não adiantando explorar outros.

Exemplo:

```
(1)  ((merge () _x _x)(!))
(2)  ((merge _x () _x)(!))
(3)  ((merge (_x|_y1)(_x|y2)(_x _x|y))(!)
          (merge _y1 _y2 _y))
(4)  ((merge (_x1|_y1)(_x2|_y2)(_x1|_y))(less _x1 _x2)(!)
          (merge _y1 (_x2|y2) y))
(5)  ((merge (_x1|_y1)(_x2|_y2)(_x2|_y))(less _x2 _x1)(!)
          (merge _y2 (_x1|_y1) _y))
```

Esse programa recebe duas listas ordenadas em ordem decrescente e, a partir da união delas, cria outra lista também em ordem decrescente.

É fácil verificar que, se qualquer uma das cláusulas do programa merge for instanciada com sucesso, as outras não o serão, pois elas são mutuamente exclusivas. Nesse caso o corte é útil pois evita o armazenamento de pontos de retrocesso desnecessários. O tempo de processamento diminui pois não há perda de tempo utilizando submetas que não levam à solução.

2) Um outro uso do corte é para indicar ao interpretador que o caminho explorado deve falhar imediatamente, e a busca de novas soluções interrompida. Nesse caso o corte é usado em conjunção com o predicado fail.

Exemplo:

```
(1)  ((not _x)(_x)(!)(fail))
(2)  ((not _x))
```

Esse programa representa a negação lógica [CLAR 78]. Se _x for satisfeito então ((not _x)) deve falhar e vice-versa. No caso em que _x for satisfeito, o corte é executado e, em seguida, o é o predicado fail fazendo ((not _x)) falhar. Caso _x falhe é feito o rastreamento para trás e ((not _x)) terá sucesso ao ser instanciada à cláusula (2).

3) O terceiro refere-se aos pontos em que se deseja terminar a geração de soluções através do rastreamento para trás, principalmente se as tentativas levaram a ciclos infinitos.

Exemplo:

```
(1)  ((soma 1 1)(!))

(2)  ((soma _n _res)(subtract _n 1 _n1)(soma _n1 _res1)
                    (add _res _n res1))
```

Ao se perguntar ?((soma 15 _res)), o programa efetua a soma de 1 até 15 colocando o resultado em _res. Quando o controle chega a (soma 1 1), executa o corte que impede a exploração de outras alternativas.

3.2.3 Predicados aritméticos

São do tipo (op arg1 arg2 arg3) onde op é uma operação binária. O resultado de arg1 op arg2 é colocado em arg3, se este for uma variável. Caso arg3 não seja variável, o predicado terá sucesso se arg3 for igual a arg1 op arg2.

Os predicados aritméticos são:

(add arg1 arg2 arg3)

Adição de números inteiros.

(subtract arg1 arg2 arg3)

Subtração de números inteiros.

(multiply arg1 arg2 arg3)

Multiplicação de inteiros.

(divide arg1 arg2 arg3)

Divisão inteira.

(modulo arg1 arg2 arg3)

Fornece o resto da divisão inteira de arg1 por arg2.

3.2.4 Predicados relacionais

Os predicados relacionais verificam a relação de ordem entre constantes. As regras usadas para determinar a ordem são:

1) Inteiros são ordenados de acordo com o seu valor numérico.

2) Um átomo é sempre menor que um inteiro.

3) Atomos são ordenados lexicograficamente e de acordo com o código ASCII. Assim sendo, letras maiúsculas são menores que minúsculas.

(less arg1 arg2)

Verifica se arg1 é menor que arg2; arg1 e arg2 podem ser dígitos, letras, inteiros ou átomos.

(greater arg1 arg2)

Verifica se arg1 é maior que arg2. É definido pela cláusula:

((greater _x _y)(less _y _x))

(le arg1 arg2)

Verifica se arg1 é menor ou igual a arg2. E definido pelas cláusulas:

```
((le _x _x)(!))
((le _x _y)(less _x _y))
```

(ge arg1 arg2)

Verifica se arg1 é maior ou igual a arg2. E definido pelas cláusulas:

```
((ge _x _x)(!))
((ge _x _y)(less _y _x))
```

(equal arg1 arg2)

Verifica se arg1 é igual a arg2. E definido pela cláusula:

```
((equal _x _x))
```

(neq arg1 arg2)

Verifica se arg1 é igual a arg2. E definido pelas cláusulas:

```
((neq _x _x)(!)(fail))
((neq _x _y))
```

(greatest arg1 arg2 arg3)

Se arg1 for maior que arg2 seu valor é colocado em arg3; caso contrário, é atribuido o valor de arg2 a arg3. E definido pelas cláusulas:

```
((greatest _x _y _z)(less _x _y)(!)(eq _z _y))
((greatest _x _y _z)(eq _z _x))
```

3.2.5 Predicados para manipulação de cláusulas

(clreg arg1)

Deleta todas as cláusulas de nome arg1.

(clregp arg1)

Deleta todas as cláusulas de nome arg1 que tenham FA (ou GC) positivos.

(clregn arg1)

Idêntico a clregp para cláusulas de FA ou GC negativos.

(delclp arg1 arg2)

No grupo de cláusulas com FA ou GC positivo e de nome arg1, deleta a cláusula apontada por arg2.

(delcln arg1 arg2)

Idêntico a delclp para cláusulas de FA ou GC negativos.

(retract arg1)

Deleta a primeira cláusula cuja cabeça unificar com arg1.

(asserta arg1)

Armazena a cláusula arg1 mantendo o grupo ordenado decrescentemente segundo o FA (ou GC). Caso nesse grupo haja uma ou mais cláusulas com o mesmo FA (ou GC) de arg1, ela é colocada antes das já existentes.

(assertz arg1)

Idêntico ao predicado asserta só que a cláusula é colocada após as cláusulas de mesmo FA (ou GC).

(addcl arg1 arg2)

Adiciona a cláusula dentro do grupo de cláusulas com a mesma cabeça que arg1, na posição indicada por arg2. Se a cláusula a ser colocada na posição indicada por arg2 não mantiver o grupo em ordem decrescente com FA ou GC, o predicado adcl falha.

3.2.6 Predicados de tipo de dados

(is_alpha arg1)

Retorna sucesso se arg1 for uma letra.

(is_uper arg1)

Verifica se arg1 é uma letra maiúscula.

(is_lower arg1)

Verifica se arg1 é uma letra minúscula.

(is_digit arg1)

Verifica se arg1 é um dígito.

(is_space arg1)

Verifica se arg1 é um espaço em branco.

(is_punct arg1)

Verifica se arg1 é um ponto.

(is_print arg1)

Verifica se arg1 é um caracter ASCII com representação gráfica.

(to_upper arg1 arg2)

Converte arg1 num caracter maiúsculo colocando-o em arg2.

(to_lower arg1 arg2)

Converte um caracter maiúsculo para um minúsculo.

(predicate arg1)

Verifica se arg1 é um predicado.

(atom arg1)

Verifica se arg1 é um átomo.

(intgr arg1)

Verifica se arg1 é um inteiro.

(float arg1)

Verifica se arg1 é um número real.

3.2.7 Predicados de depuração e acompanhamento de programas

(enable TR)

Ativa a rotina "trace" [CLOC 81]. São mostradas as metas atingidas na resolução de um programa e a cabeça da cláusula instanciada a cada meta. Comparando-se uma meta e a cabeça da cláusula instanciada pode-se ver as unificações feitas.

Como ESPROLOG é uma linguagem não determinística, o mecanismo de rastreamento para trás é acionado sempre que uma meta não puder ser resolvida. Uma nova sequência de metas é gerada e a sistemática se repete até que se chegue a uma solução ou à impossibilidade de atingir uma solução.

(enable OC)

Ativa a rotina "occour check" [COLM 82]. Ela tem a função de evitar que uma variável seja unificada a uma estrutura que contenha essa variável, pois unificações desse tipo podem gerar ciclos infinitos.

Como essa rotina pode comprometer a eficiência do

interpretador, só deve ser usada na fase de depuração de programas.

(enable LCO)

Ativa rotinas para otimização de recursão de cauda [BRUY-82].

(desable arg1)

Desativa as rotinas de depuração. Arg1 pode assumir os valores TR, OC ou LCO.

(gc)

Ativa a rotina de coleta de lixo, que recupera os espaços para armazenamento de cláusulas, átomos e inteiros.

(statistics)

Essa rotina mostra a situação das pilhas de ambientes locais e globais, de variáveis locais e globais e a de falhas e substituições.

4. MANUAL DO IMPLEMENTADOR

4.1 Introdução

O interpretador ESPROLOG foi construído a partir do interpretador PROLOG desenvolvido por Allan Roger, no INSTITUTO NACIONAL DE PESQUISA ESPACIAIS [ROGE_86]. Essa escolha se deve ao fato desse interpretador ter sido escrito na linguagem "C", o que garante sua portabilidade. Além disso, à época do início de nosso trabalho, o interpretador já estava disponível com as otimizações de cauda, de última chamada e de submetas determinísticas implementadas.

O interpretador ESPROLOG é constituído de três módulos principais: o analisador léxico, o analisador sintático e o motor de inferência.

O analisador léxico recebe uma sequência de caracteres como entrada, agrupando-os em unidades fundamentais chamadas constantes (vide secção 3.1.1). O analisador sintático verifica se a sequência de constantes recebidas do léxico forma sentenças corretas de acordo com a gramática da linguagem. É nessa fase que é gerado e armazenado o código das cláusulas. A análise semântica é feita pelo motor de inferência.

O motor de inferência é responsável pela resolução de cláusulas metas. É usada a técnica de compartilhamento de estruturas desenvolvido por Boyer e Moor [BOYE 72]. Como nas

várias instanciações de uma cláusula só são mutáveis as variáveis, sempre que se fizer referência a essa cláusula é usado o mesmo apontador. Para guardar o valor assumido pelas variáveis locais e globais, em cada instanciação é construída uma pilha para cada tipo de variável.

Uma alternativa para o compartilhamento de estruturas seria a cópia [MELL 82] [CHEN 84]. Sabe-se que a cópia é mais lenta que o compartilhamento, apesar de oferecer uma economia de espaço. Como o tempo de resposta é um fator importante e o problema de espaço é minimizado pelas otimizações, o compartilhamento foi preferido à cópia.

4.2 Estrutura de armazenamento de termos simples

Os átomos são armazenados numa matriz bidimensional (Figura 4.1). LASTATM define o número máximo de átomos que podem ser armazenados. WL define o comprimento máximo das cadeias. Como as cadeias na matriz terminam com um caracter nulo, o comprimento máximo de um átomo é dado por WL - 1. As locações, de zero até NEWATOMS - 1, são inicializadas com os átomos utilizados nos predicados pré-definidos. O restante das locações está disponível para os átomos definidos pelo usuário.

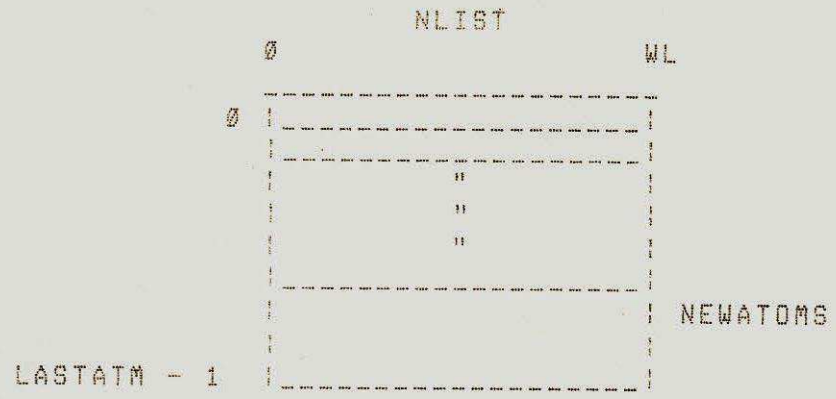

FIG 4.1 Matriz NLIST

A rotina STATM armazena um átomo, se ele não estiver presente na matriz, devolvendo um apontador para o mesmo. Se o átomo já estiver presente, STATM apenas devolve o apontador.

Os números inteiros são armazenados no vetor INT cujo comprimento é dado pela constante LASTINT. As locações livres em INTG estão encadeadas. A variável PINTG aponta para a primeira locação livre. Na alocação de um inteiro , PINTG é atualizado com o conteúdo da posição para a qual ele aponta. A rotina STNUM armazena números inteiros devolvendo um apontador.

As variáveis ESPROLOG são separadas em dois grupos: variáveis globais, que aparecem em listas, e as demais que são locais. Essa divisão é necessária para que se possa fazer otimizações.

As variáveis são armazenadas numa matriz. Essa matriz é gerenciada de maneira análoga à matriz de átomos. Ao encontrar uma dada variável, o interpretador não sabe se adiante ela fará

parte de uma lista ou não e, por esse motivo, cria uma estrutura para atualizar o seu tipo e também para gerar o código interno dessa variável. Esses conceitos serão detalhados no armazenamento de cláusulas.

4.3 Estrutura de armazenamento de termos compostos

Uma lista é um par cabeça-cauda armazenado numa estrutura denominada célula. Uma célula contém duas locações "tcar" e "tcdr". Na primeira é colocada a cabeça da lista e na segunda um apontador para sua cauda. Para manipular células, utilizam-se as funções básicas semelhantes às da linguagem LISP: CAR, CDR, RPLCAR, RPLCDR e CONS.

A lista :

(Joao_Silva motorista (31 Ø4 49))

é armazenada internamente de acordo com a figura 4.2.

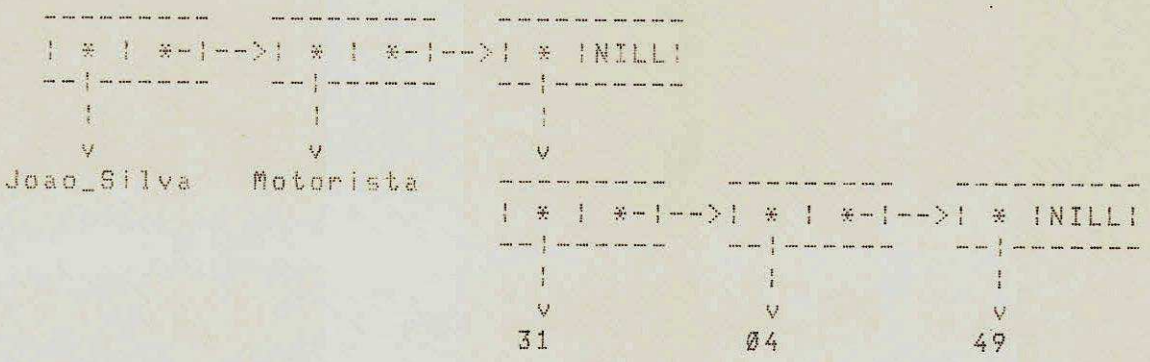

FIG 4.2 Representação interna de uma lista

Os apontadores para inteiros, listas e átomos são

inteiros. Eles são formados pela soma do inteiro que dá a sua posição no vetor ou matriz correspondente, com uma constante que permite a identificação do apontador.

Um predicado é armazenado como uma lista e, para diferenciá-lo das listas propriamente ditas, sua cabeça-cauda é armazenada numa célula marcada que se encontra num determinado espaço das células. A função CONS1 gerencia as células nesse intervalo.

Uma cláusula é armazenada como uma lista de predicados acrescida de outras informações.

Sejam as cláusulas:

C1 ((append () _x _x))

C2 ((append (_xx!_yy) _z (_xx!_z1)
(append _yy _z _z1))

Elas têm uma representação interna mostrada nas figuras 4.3 e 4.4.

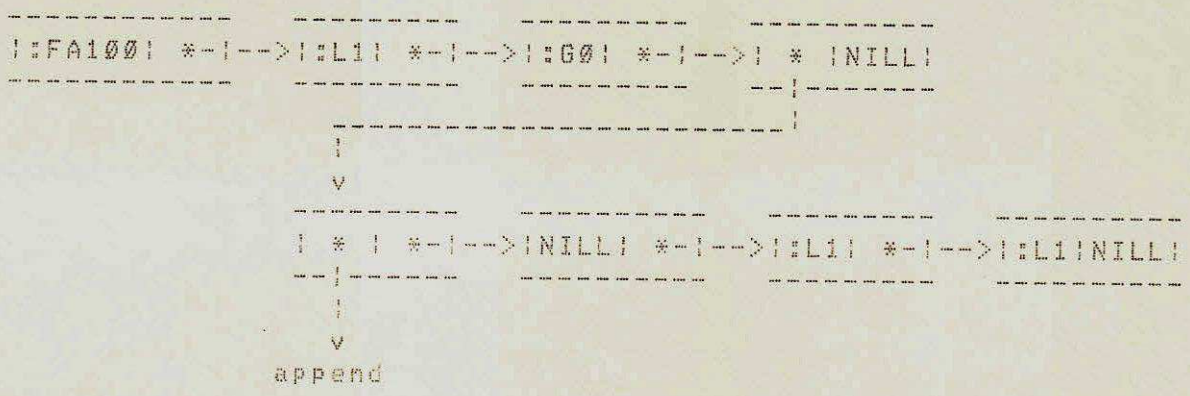

FIG 4.3 Representação interna da cláusula C1

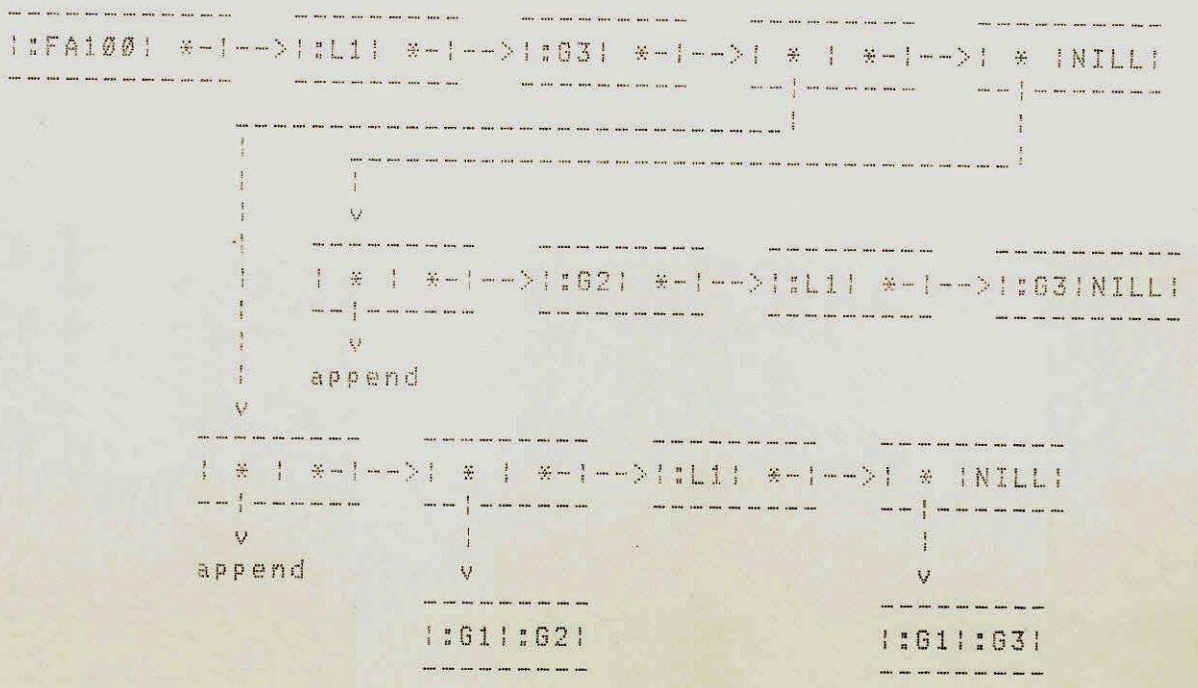

FIG 4.4 Representação interna da cláusula C2

O primeiro elemento da lista é o FA se for uma cláusula implicação, o GC se for uma cláusula fato e LA se for uma meta. Em seguida vêm, respectivamente, o número de variáveis locais e globais seguido da lista de predicados que compõem a cláusula.

Como pode ser visto na figura, os nomes das variáveis são trocados por números precedidos pelo símbolo :L se forem locais e :G se forem globais. O número indica a ordem de ocorrência da variável na cláusula, ou seja, a primeira variável local da cláusula é trocada por :L1, a segunda por :L2 e assim por diante. O código das variáveis globais segue o mesmo esquema das locais.

As células são armazenadas numa matriz de acordo com a figura 4.5

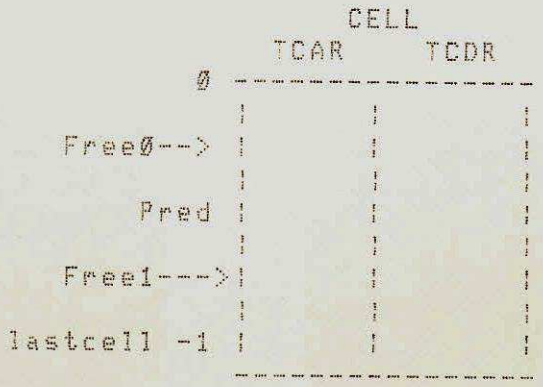

FIG 4.5 Matriz CELL

A constante "lastcell" fornece o número de locações na matriz CELL. "FreeØ" aponta para a primeira locação livre. As demais locações livres estão encadeadas no CDR das células. A constante "Pred" dá o início das células marcadas destinadas a armazenar predicados. O apontador "Free1" aponta para a primeira célula livre nesse intervalo.

4.4 Armazenamento de cláusulas

As cláusulas são armazenadas numa tabela. A localização de uma cláusula é dada por uma função simples aplicada ao apontador do átomo que dá nome ao predicado. As cláusulas de mesmo nome ficam encadeadas em duas listas. A primeira contém todas as

cláusulas de FA ou GC positivo em ordem decrescente. A segunda contém, em ordem crescente, as cláusulas de FA ou GC negativos.

Uma locação na tabela de cláusulas contém um apontador para uma célula; o CAR desta aponta para a primeira lista e o CDR para a segunda (figura 4.6).

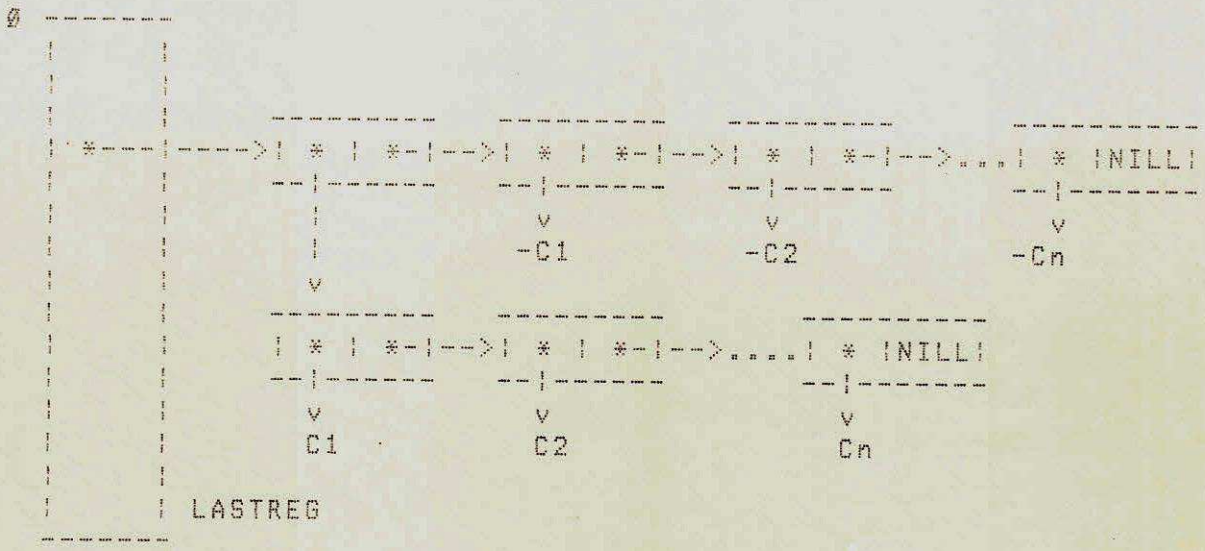

FIG 4.6 Encadeamento de cláusulas no vetor NREG

4.5 Coleta de lixo

No decorrer da leitura de cláusulas e na resolução de cláusulas metas, as listas que representam essas estruturas são modificadas e algumas células podem ser dealocadas. Essas células são chamadas de lixo. Se for necessário um número maior de células do que as existentes na lista de células livres, é preciso recuperar as células perdidas. O mesmo também ocorre com o espaço reservado para átomos e números.

A recuperação é feita mediante um coletor de lixo simples como o coletor descrito por Winston [WINS 84].

```
MARK(p)
 {
     se p for um átomo ou um número então
          marque-o;
     senão {
          enquanto p for uma lista não marcada {
               MARK(car(p));
               se car(p)>Ø então
                    multiplique o seu valor por -1;
               p=cdr(p);
             }
      }
  }
```

As células são marcadas trocando-se o conteúdo do CAR das mesmas por seu valor negativo. Os átomos e números são marcados pela anotação em uma tabela binária.

Todas as estruturas não marcadas são incorporadas às listas livres correspondentes, e as marcadas são desmarcadas.

A coleta de lixo é feita automaticamente sempre que a estrutura para armazenamento de listas, átomos ou inteiros estiver cheia, ou ao final de uma consulta. Além disso, o usuário poderá acionar o coletor sempre que achar conveniente, com o

predicado (gc).

4.6 A unificação

Dados dois predicados, o algoritmo de unificação verifica se existe um casamento entre os termos correspondentes de cada um. O algoritmo, para cada par de termos, verifica uma das seguintes condições:

1) Os dois termos são constantes iguais.

2) Um termo é uma variável e o outro constante.

3) Os dois termos são variáveis.

4) Um termo é uma variável e o outro uma lista.

5) Os dois termos são listas e os termos correspondentes unificam.

Caso não se verifique uma dessas condições o algoritmo de unificação fracassa.

```
UNIFY(h, r)
{
enquanto h e r não forem ambos listas vazias {
 x <-- car(h);
 y <-- car(r);
 se x for uma variável então
      associe x a y;
 senão se y for uma variável então
      associe y a x;
```

```
        senão se x e y forem átomos diferentes então
            retorne fracasso;
        senão se não UNIFY( car(x), car(y)) então
            retorne fracasso;
        h <-- cdr(h);
        r <-- cdr(r);
      }
    retorne sucesso
  }
```

4.7 Pilha de ambientes e pilha de variáveis

A anotação dos valores das variáveis é feita utilizando-se duas pilhas: uma de ambientes e outra onde são anotados os valores das variáveis. Como temos dois tipos de variáves teremos um total de quatro pilhas.

As duas pilhas de ambientes são armazenadas numa mesma matriz. A constante AMBILIM representa o número de locações na matriz (Figura 4.7). Há dois apontadores: TAMBGLOB e TAMBLOC. Eles servem para apontar a próxima locação livre para os ambientes local e global respectivamente. Uma locação na pilha de ambientes consta de duas partes: NVAR e VAR1. Em NVAR é colocado o número de variáveis desse ambiente e em VAR1 o endereço da primeira variável na pilha de variáveis.

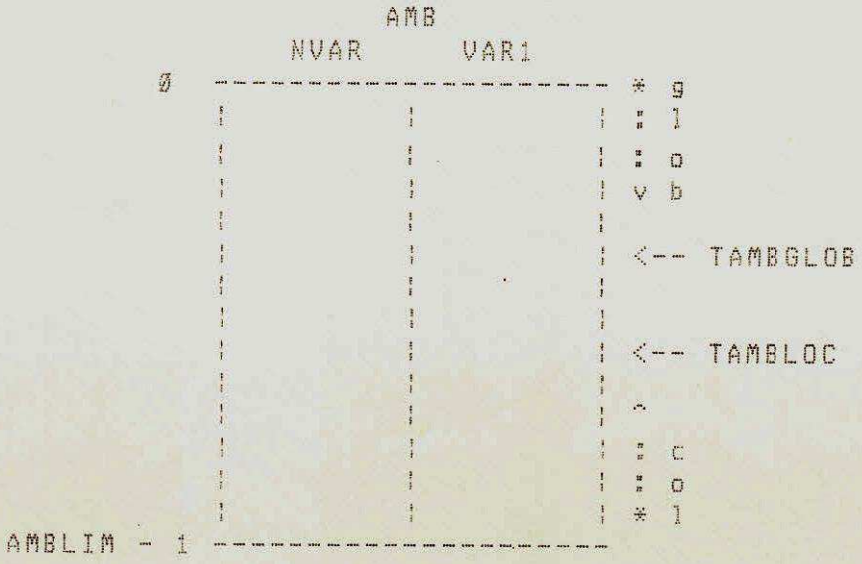

FIG 4.7 As duas pilhas de ambientes

As duas pilhas de variáveis também são armazenadas em uma matriz (Figura 4.8). O número de locações na matriz é dado pela constante VAVLIM. Os apontadores TGLOBVAV e TLOCVAV indicam as posições livres para as variáveis globais e locais respectivamente. Cada locação na matriz consta de duas partes: AP_AMB e VAL. Em AP_AMB é colocado o endereço do ambiente em que a variável assume um valor e em VAL esse valor. Caso essa variável esteja associada a outra, para localizá-la é necessário ter, além do valor do ambiente, a sua posição nesse ambiente que será colocada em VAL. Se a variável estiver associada a um valor,o ambiente não será necessário mas, mesmo assim, seu valor é anotado pois essa informação auxilia nas otimizações.

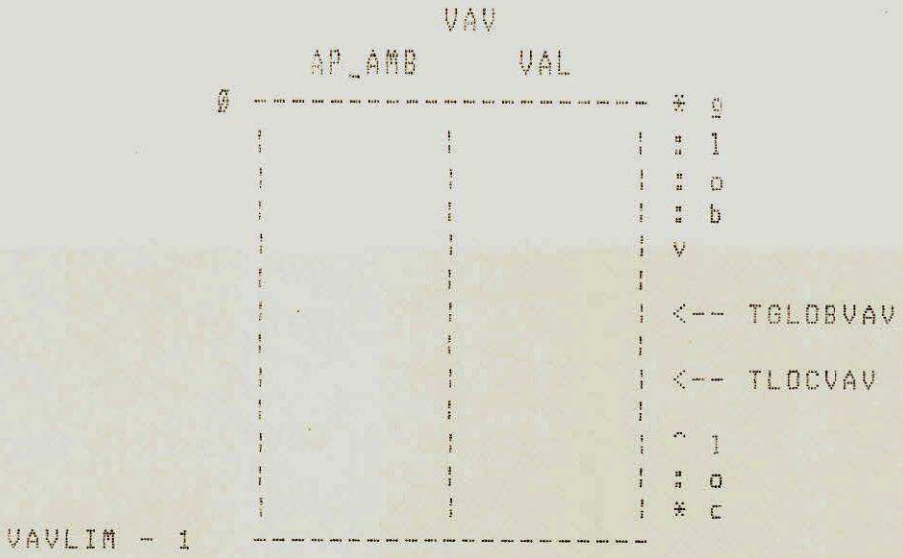

FIG 4.8 As duas pilhas de variáveis

4.8 Avaliação e associação de variáveis

O algoritmo AVAL obtém o valor de uma variável.

```
AVAL(termo, amb_loc, amb_glob)
{
enquanto verdadeiro {
   se o termo não for variável então
     retorne(termo);
   senão obtenha o endereço da variável
        apontado por termo;
  }
}
```

O algoritmo pára quando o termo for uma constante ou se o

termo ainda não recebeu um valor.

O procedimento ASSIGN é usado para associar duas variáveis ou uma variável a uma constante.

```
ASSIGN(x, local_x, global_x, y, local_y, global_y)
                                    .
{
se y e x estiverem associados à mesma variável
   retorne(verdade);
calcule a posição i da variável  x  na
      pilha  de  variáveis (VAV);
VAV[i].val <-- y;
se y for uma variável local
   VAV[i].amb <-- local_y;
senão se y for uma variável global
   VAV[i].amb <-- global_y;
coloque  na  pilha  de substituições
   a posição alterada em VAV;
}
```

O algoritmo associa à variável x o valor y e o valor do ambiente de y também é anotado.

As posições alteradas na pilha VAV serão anotadas na pilha de substituições. Com isso VAV poderá ser restaurada se houver um rastreamento para trás.

4.9 A resolução

As cláusulas metas são resolvidas pelo procedimento RESOL.

A resolução trabalha sobre uma estrutura chamada pilha de falha. Ela compartilha a estrutura SF com a pilha de substituições. As duas pilhas (Figura 4.9) possuem crescimentos opostos. A pilha de falha possue um apontador para seu topo chamado TFAIL. O apontador para o topo da pilha de substituições chama-se TSUB. A condição de pilha cheia é dado por TFAIL > TSUB.

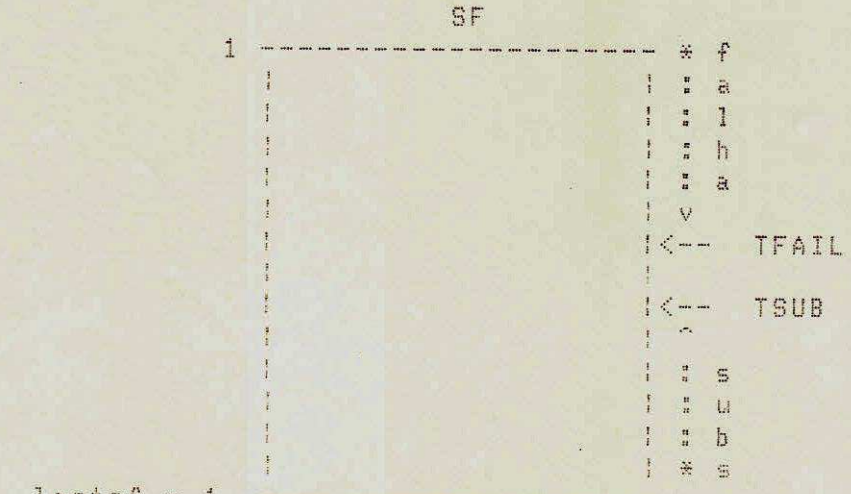

FIG 4.9 A pilha de falhas e de substituições

Se a cláusula que unificou com a hipótese for única ou for a última cláusula do grupo, tem-se um nó determinístico. Para um nó determinístico não é necessário colocar informações na pilha de

falhas.

Toda vez que uma hipótese unificar com a cabeça de uma cláusula e esta não for a única candidata, tem-se um nó não determinístico. Havendo necessidade de fazer um rastreamento para trás, a pilha de falhas deve conter as informações para recomeçar desse ponto. Na pilha de falhas são armazenados os seguintes dados:

1) Próximas cláusulas candidatas à unificação
2) Topo da pilha de ambientes globais (TAMBGLOB).
3) Topo da pilha de ambientes locais (TAMBLOC).
4) Topo da pilha de substituições (TSUB).
5) A hipótese.

Caso haja fracasso, os dados são desempilhados do topo da pilha de falha e reinicia-se com a próxima cláusula candidata.

```
RESOL(tfail)
{
START:
se a lista de cláusulas candidatas estiver vazia
     vá_para BACKTRACK;
REDO:
a hipótese atual é o primeiro predicado da hipótese;
se a hipótese for um predicado pré_definido {
     se o predicado foi resolvido com sucesso
        vá_para NEXT;
     se não
        vá_para BACKTRACK;
```

```
}
NEXTCAND:
separe a cláusula candidata da lista;
construa os ambientes locais e globais para a cláusula;
se a hipótese não unificar com a cláusula {
     desfaça as ligações feitas na pilha de variáveis;
     desfaça os ambientes locais e globais da cláusula;
     se não houver mais cláusulas candidatas
          vá_para BACKTRACK;
     se não
          vá_para REDO;
}
atualize a hipótese com o corpo da cláusula que unificou;
se   houver   mais   cláusulas   candidatas
     atualize a pilha de falhas;
se a hipótese for vazia
     retorne(sucesso);
se não {
     obtenha a lista de cláusulas que possam unificar com
          o primeiro predicado da hipótese;
     vá_para START;
}
BACKTRACK:
se a pilha de falhas estiver vazia
     retorne(falha);
atualize os topos das pilhas de ambientes, de variáveis,
     de substituições e de falhas;
```

```
        desfaça as ligações nas pilhas de variáveis entre o
            topo antigo e o novo;
        obtenha a lista de cláusulas candidatas para o
            primeiro predicado da hipótese;
        va_para REDO;
}
```

4.10 O grau de certeza de uma consulta

Como já foi visto, à cláusula fato temos associado um grau de certeza (GC) que representa a confiança que se tem nesse fato. O grau de certeza de uma cláusula implicação é dado pela expressão:

GC(A) = FA * mínimo(GC(Bi))

onde A é o átomo que dá nome à cláusula, FA é o fator de atenuação dessa cláusula e os "Bi" são os predicados que constituem o corpo da cláusula.

O grau de certeza de uma consulta só fica determinado no final da resolução. No início sabe-se que a resolução poderá ter GC no máximo igual a 100. A grandeza que representa o valor máximo que o GC pode assumir recebe o nome de teto.

A cada unificação feita um novo teto é obtido multiplicando o teto associado à hipótese anterior (no caso 100) pelo FA ou GC da cláusula unificada dividido por 100. As variáveis teto são armazenadas na pilha de variáveis globais. Ao fim da resolução o

menor dos tetos será o grau de certeza da consulta.

4.11 Otimizações

Alguns programas podem apresentar problemas de consumo excessivo na pilha de variáveis. Serão descritos critérios para recuperar espaço nessa pilha, minimizando o problema.

4.11.1 Sucesso de submetas determinísticas

Ao finalizar a resolução de uma submeta com sucesso, se esta for determinística, poderemos dispensar todos os ambientes locais até o primeiro nó não determinístico [BRUY 82].

A otimização só é aplicada às variáveis locais, uma vez que as variáveis globais envolvem listas e estas podem estar associadas a variáveis de um ambiente anterior, impedindo a otimização.

4.11.2 Otimização de recursão de cauda

Essa otimização transforma um programa recursivo em um programa iterativo [BRUY 82][HOGG 84].

As condições a serem satisfeitas para aplicarmos a otimização são:

1) A meta a ser resolvida deve ser a última do corpo de uma cláusula.

2) Todas as outras metas da cláusula já resolvidas devem ser determinísticas.

3) A cláusula deve ser recursiva.

Se as condições forem satisfeitas, o ambiente local da meta a ser resolvida é colocado no lugar do ambiente local da cabeça da cláusula e os demais ambientes locais das metas do corpo da cláusula podem ser eliminados.

4.11.3 Otimização da última chamada

A otimização de última chamada é uma extensão da anterior. As condições a serem satisfeitas são as mesmas, exceto que a cláusula não precisa ser recursiva [BRUY 82][CHEH 84].

4.11.4 Otimização de poda por limiar de aceitação

Como já vimos, a associação de um predicado da meta com a cabeça de uma cláusula é precedida pelo cálculo da variável teto. Se esta variável assumir um valor menor que LA é feito um rastreamento para procurar uma solução que tenha GC >= LA. Essa otimização impede que se explorem caminhos que não levem à solução.

4.12 Detalhes sobre a resolução

Os exemplos subsequentes visam mostrar como funciona o algoritmo de resolução com compartilhamento de estruturas.

O primeiro exemplo evidencia como é calculada e armazenada a variável teto. Dadas as cláusulas e a consulta:

```
((queijo montrachet) 80 (finalidade aperitivo)
                        (preferencia suave))

((queijo gorgonzola) 75 (finalidade aperitivo)
                        (preferencia picante))

((preferencia picante) 90)

((preferencia suave) 60)

((preferencia aromatico) 50)

((finalidade aperitivo) 90)

?(60(queijo _tipo))
```

O intrpretador transforma essas cláusulas em listas, no formato:

```
C1- ( FA80 L0 G0 (queijo montrachet)(finalidade aperitivo)
                 (preferencia suave))

C2- ( FA75 L0 G0 (queijo gorgonzola)(finalidade aperitivo)
                 (preferencia suave))
```

C3- (GC90 L0 G0 (preferencia picante))

C4- (GC80 L0 G0 (preferencia suave))

C5- (GC50 L0 G0 (preferencia aromatico))

C6- (GC90 L0 G0 (finalidade aperitivo))

O primeiro elemento das listas que representam a cláusula é o FA ou o GC desta; os dois elementos seguintes fornecem o número de variáveis locais e globais, e o último elemento é uma lista que representa a cláusula propriamente dita.

A consulta é transformada numa lista denominada hipótese inicial .

H1 - ((queijo :L1) Amb_G1 Amb_L1)

O primeiro elemento da lista é a hipótese e os dois elementos seguintes são apontadores para a pilha de ambientes locais e globais (fig 4.10).

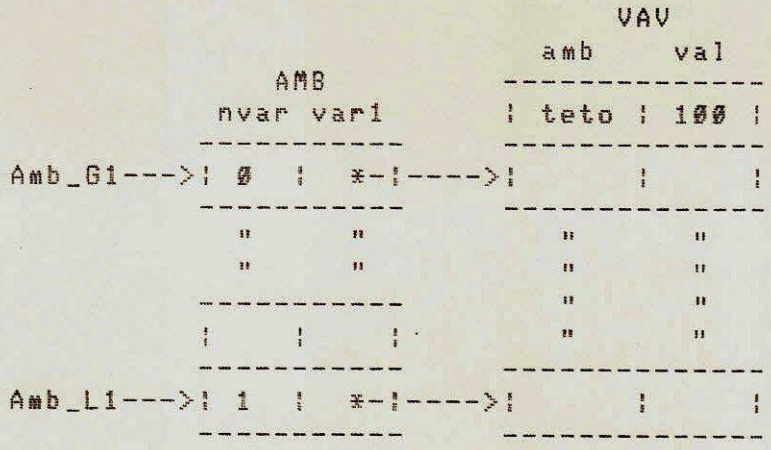

FIG 4.10 Pilhas de ambientes e variáveis

Como se vê na figura 4.10 **Amb_G1** aponta para o ambiente global da hipótese **H1**. Nesse ambiente é anotado o número de variáveis globais e o endereço da primeira variável desse ambiente na pilha **VAV**. Na primeira posição **VAV** é colocado o teto inicial e a posição apontada por **amb.var1** é reservado para o valor do próximo teto.

Analogamente, **Amb_L1** aponta para o ambiente local onde temos anotado o número de variáveis locais e o endereço da primeira variável local desse ambiente.

O interpretador tentará instanciar a cabeça de uma cláusula à hipótese **H1**. As cláusulas **C1** e **C2** são as candidatas. A primeira a ser tentada é **C1**. A rotina **NEWNV** se encarrega de criar o ambiente local **Amb_L2** e global **Amb_G2** para a instanciação, além de reservar o espaço necessário às variáveis locais e globais. O novo teto é calculado (100*80/100) e, como esse é maior que o **LA**, seu valor colocado em **VAV**. No próximo passo o casamento de todos os termos é verificado e a pilha **VAV** atualizada fig 4.11.

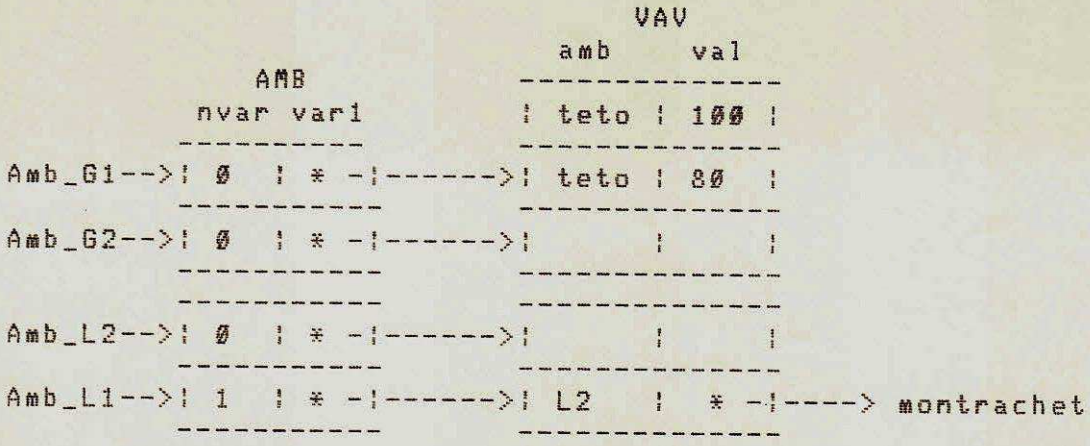

FIG 4.11 Pilhas de ambientes e variáveis

Após a instanciação, a hipótese H2 passa a ser o corpo da cláusula C1. Como ainda existe uma cláusula candidata, a pilha de falha é atualizada.

H2 ((finalidade aperitivo)(preferencia suave) G2 L2))

O primeiro predicado da hipótese H2 será instanciado à cláusula C6 (fig 4.11).

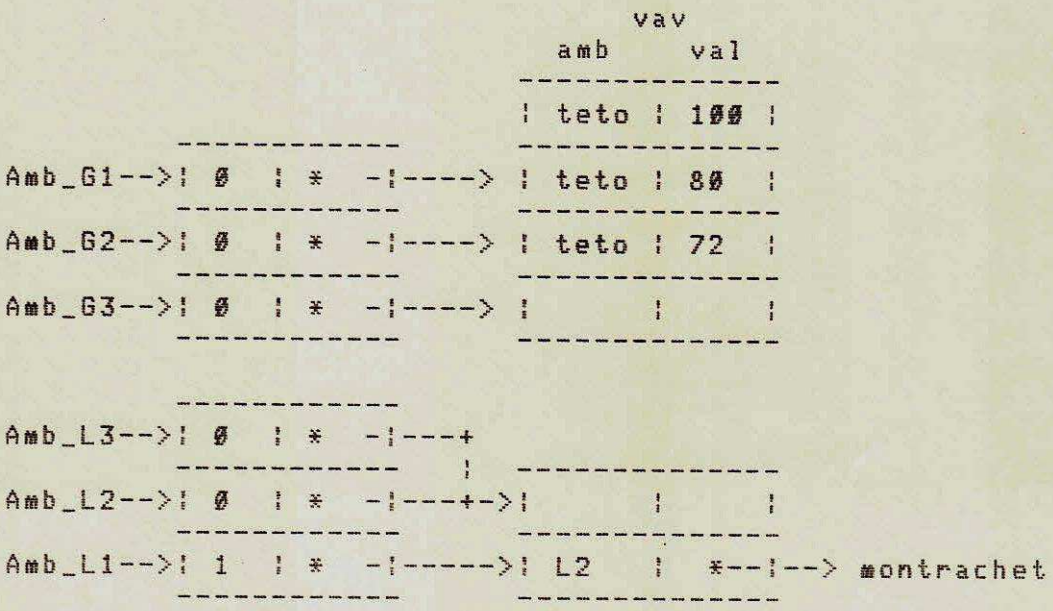

FIG 4.12 Pilhas de ambientes e variáveis

Como a cláusula C7 não tem corpo, a hipótese H3 passa a ser:

H3 ((preferencia suave) Amb_L2 Amb_G2)

Essa nova hipótese poderá ser instanciada às cláusulas C3, C4 ou C5. A tentativa de unificar C3 com H3 falha pois **picante** não unifica com **suave**. A cláusula C4 poderá ser instanciada com sucesso (fig 4.13).

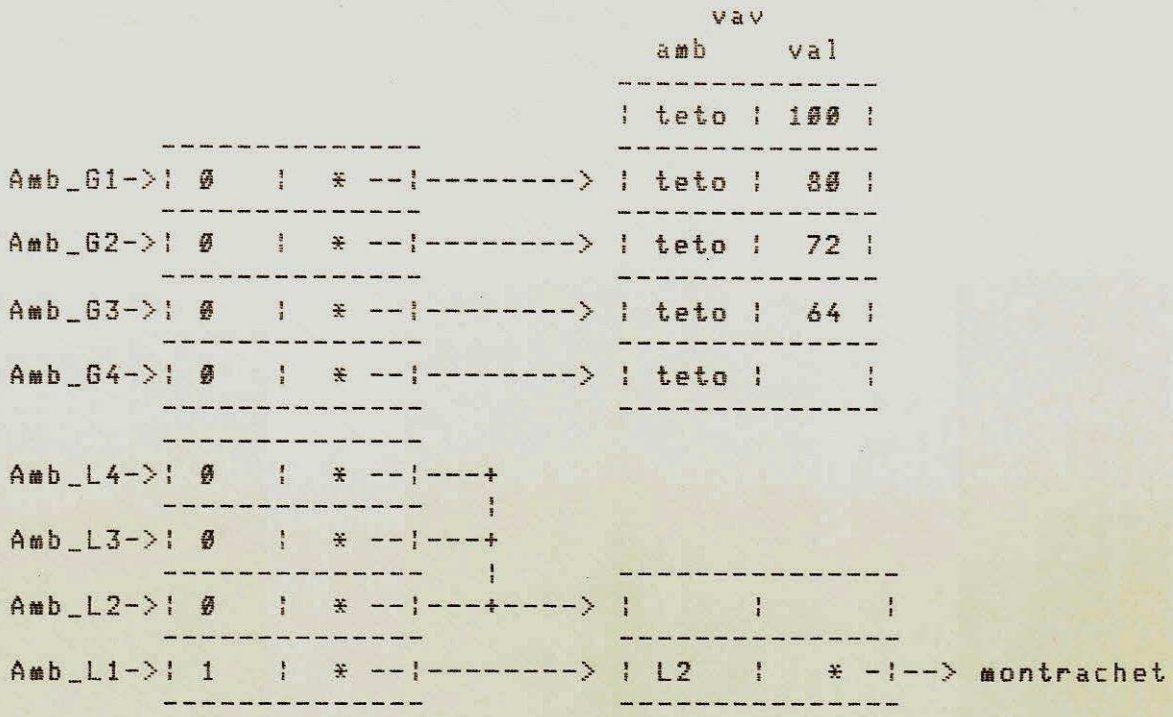

FIG 4.13 Pilhas de ambientes e variáveis

Como a hipótese agora é uma lista vazia, a consulta foi resolvida com sucesso e o **GC** dessa solução é 64 (a menor das variáveis teto).

O segundo exemplo ilustra como o algoritmo de resolução atribui valores às variáveis na pilha **VAV**. Sejam as seguintes cláusulas e a hipótese, apresentadas na forma de representação interna:

```
C1 (FA100 L0 G0 (NREV NIL NIL))

C2 (FA100 L2 G2 (NREV (:G1 :G2) :L1)(NREV :G2 :L2)
                (APPEND  :L2 (:G1) :L1))

C3 (FA100 L1 G0 (APPEND () :L1 :L1))
```

C4 (FA1ØØ L1 G3 (APPEND (:G1¦:G2) :L1 (:G1¦:G3))

(APPEND :G2 :L1 :G3))

H1 ((NREV (a b) L1) :L1 :G1) Amb_L1 Amb_G1)

Esse programa recebe a lista (a b) e devolve a lista na ordem reversa.

A hipótese **H1** (NREV (a b) :L1) pode ser instanciada às cláusulas **C1** ou **C2**. É fácil ver que a cabeça de **C1** não pode ser feita idêntica a **H1**. A figura 4.14 mostra as pilhas **AMB** e **VAV** após a cabeça da cláusula **C2** ter sido instanciada com **H1**.

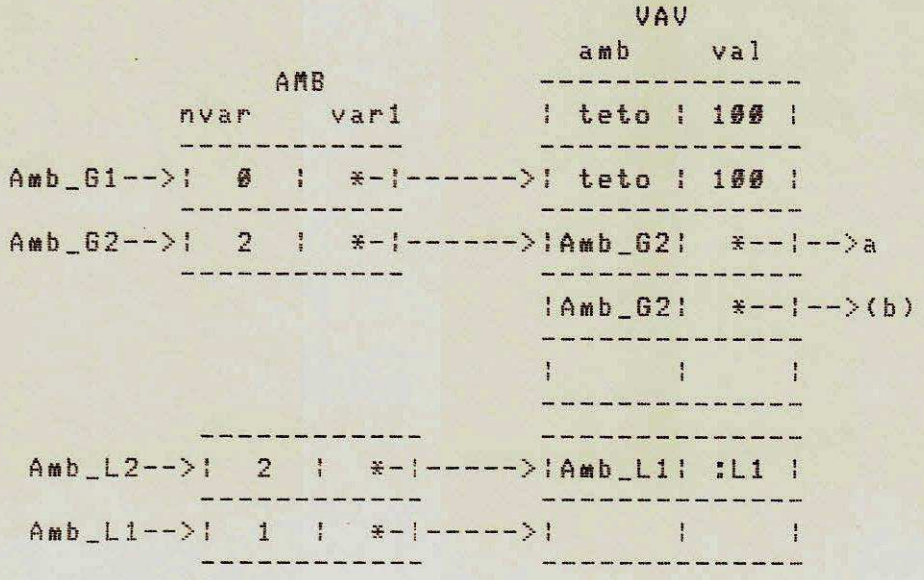

FIG 4.14 Pilhas de ambientes e variáveis

Como pode ser visto, o algoritmo de unificação atribuiu à primeira variável do ambiente **Amb_G1** o átomo **a** e à segunda

variável desse ambiente é atribuída à lista (b). A primeira variável do ambiente local Amb_L2 receberá o valor da primeira variável do ambiente Amb_L1. A primeira variável local de Amb_L1 ainda não recebeu valor.

A nova hipótese H2 a ser provada será:

H2 ((NREV (b) :L2)(APPEND :L2 (a) :L1) Amb_L2 Amb_G2)

A figura 4.15 mostra as pilhas após o primeiro predicado de H2 ter sido instanciado à cláusula C2.

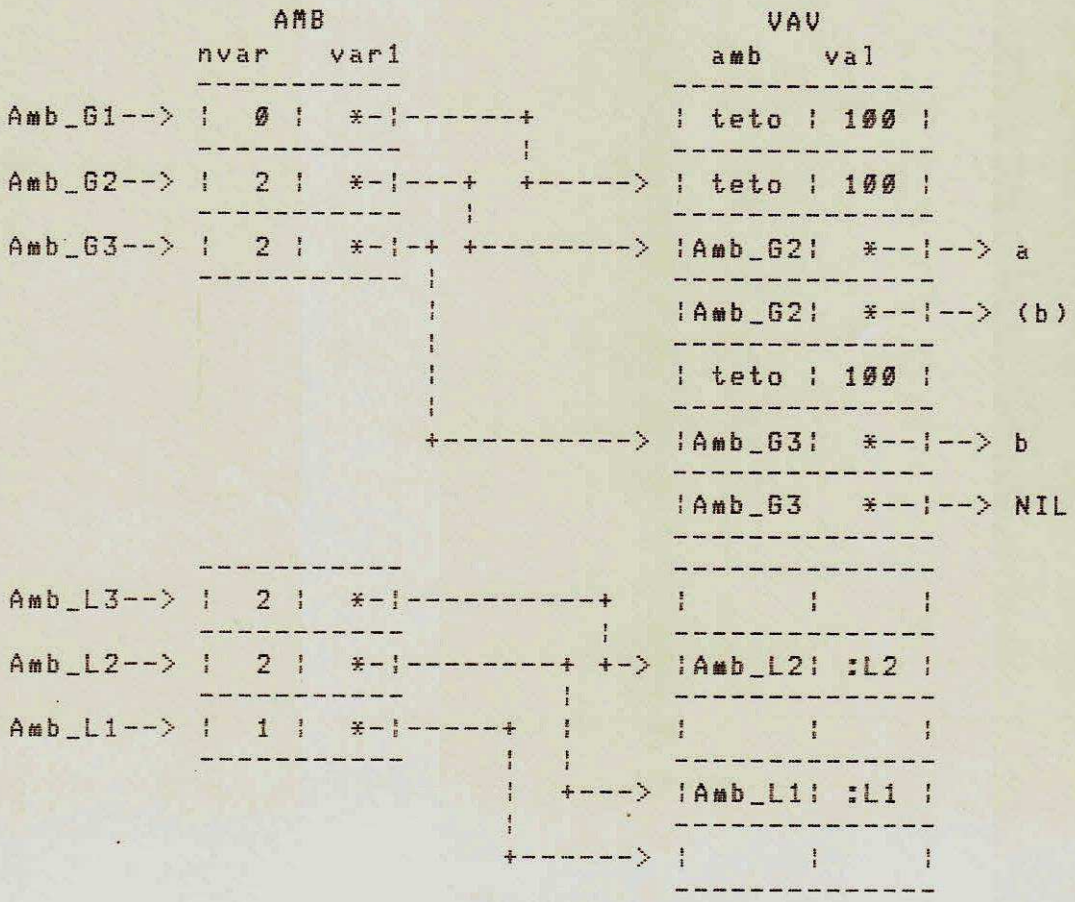

FIG 4.15 Pilhas de ambientes e variáveis

H3 (((NREV NIL :L2)(APPEND :L2 (b) :L1) Amb_L3 Amb_G3)

(APPEND :L2 (a) :L1) Amb_L2 Amb_G2))

O primeiro predicado da hipótese é instanciado à cláusula C1 (fig 4.16).

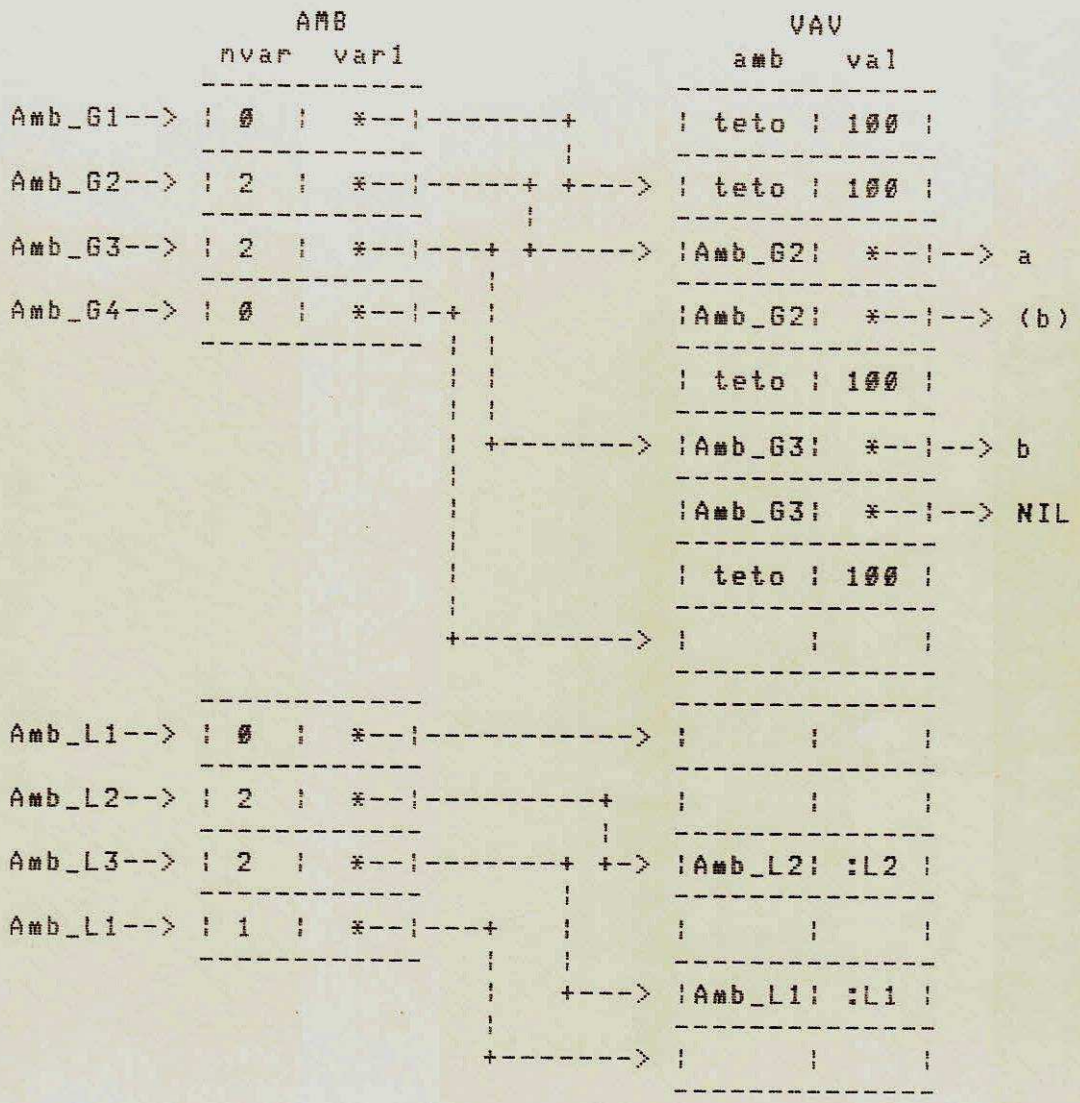

FIG 4.16 Pilha de ambientes e variáveis

H4 (((APPEND :L2 (b) :L1) Amb_L3 Amb_G3)
(APPEND :L2 (a) :L1) Amb_L2 Amb_G2))

O primeiro predicado de H4 pode ser instanciado com sucesso

com a cláusula C3 (fig 4.17), dando origem à hipótese H5.

H5 ((APPEND a (b) :L1) Amb_L2 Amb_G2)

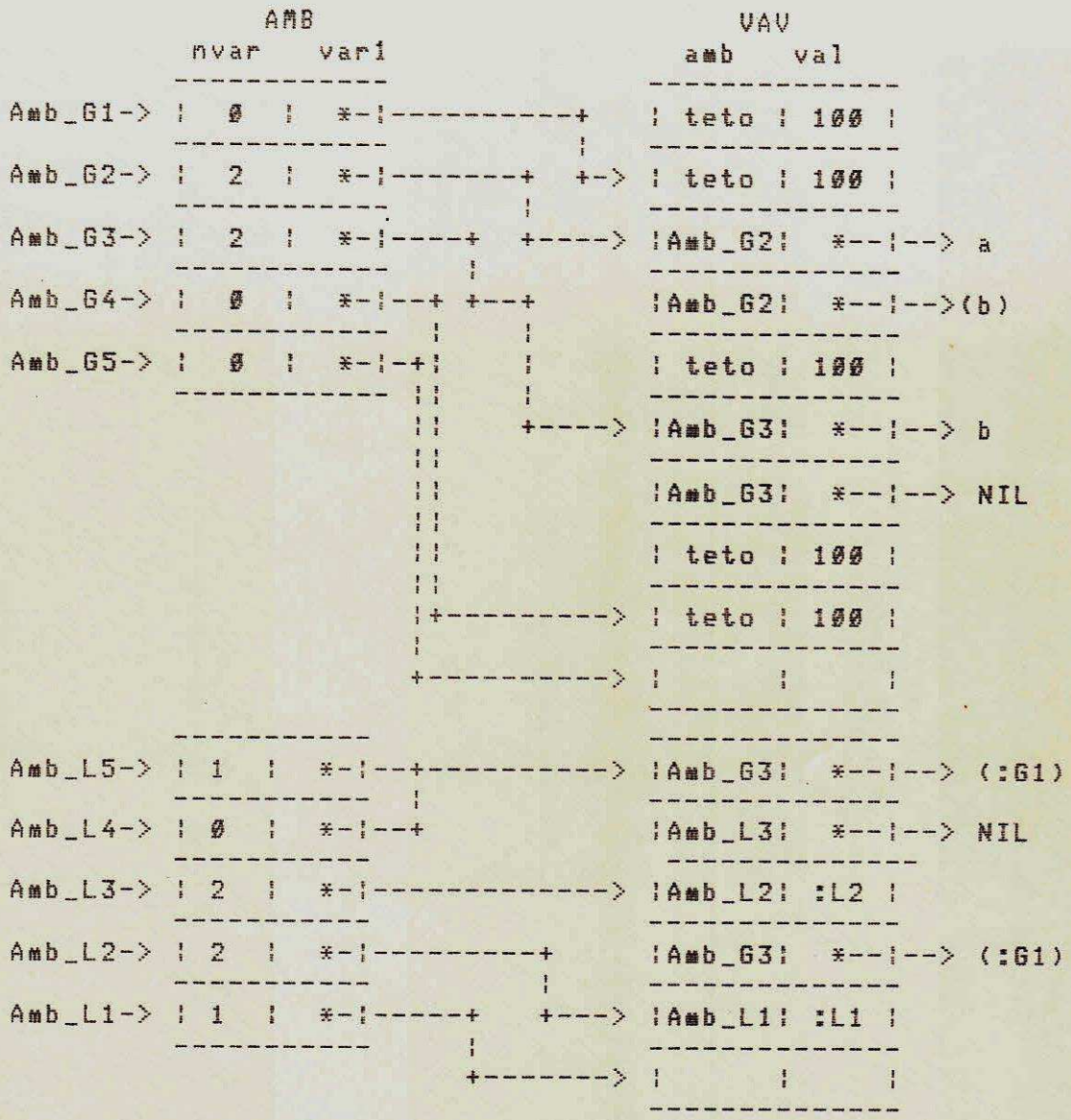

FIG 4.17 Pilhas de ambientes e variáveis

A hipótese H5 será instanciada com sucesso à cláusula C4 dando origem a H6 ((APPEND NIL (a) :G3) L6 G6). A hipótese H6 por sua vez, será instanciada à cláusula C3 terminando a consulta com

sucesso (fig 4.18).

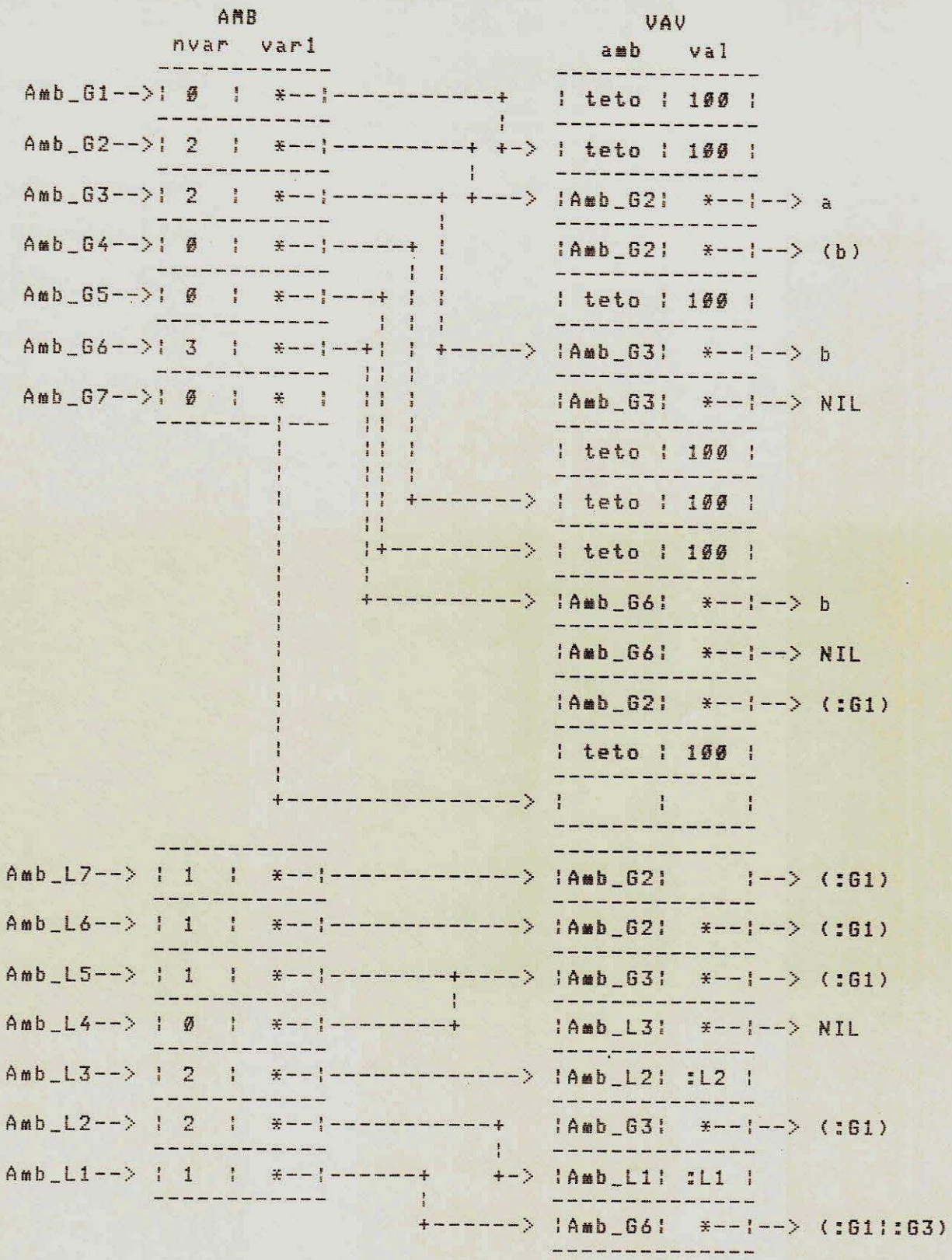

FIG 4.18 Pilha de ambientes e variáveis

A variável **:L1** do ambiente **Amb_L1** fornece a solução da consulta. A pilha de variáveis mostra que ela é dada pela lista **(:G1!:G3)** na qual as variáveis pertencem ao ambiente **Amb_G6**. No **Amb_G6** a variável **:G1** está associada ao átomo **b**, e a variável **:G3** esta associada à variável **:G1** do ambiente **Amb_G2**, que por sua vez está associada ao valor **a**.

5. CONCLUSÕES E SUGESTÕES

ESPROLOG, implementado na linguagem C (favorecendo sua portabilidade) sob um sistema operacional compatível com MSDOS, apresenta um bom desempenho, tanto na pouca ocupação de memória como em velocidade de execução. O sistema necessita de aproximadamente 90 KBYTES para ser executado.

Quanto às vantagens de ESPROLOG, vcide a seção 1.3.

Quanto à performace em tempo de execução, primeiramente foi medido o tempo gasto para executar três programas PROLOG no interpretador desenvolvido por A. Roger, depois no ESPROLOG e finalmente no ARITY-PROLOG 4.0 (vide resultados na tabela 1 e programas no anexo 1).

PROGRAMA	ESPROLOG	PROLOG(A.Roger)	ARITY PROLOG 4.0
conte	7.08s	8.49s	3.00s
NREV	4.01s	3.96s	1.71s
qsort	1.21s	1.10s	0.33s

TABELA 1

A comparação com o PROLOG de A. Roger permite avaliar o tempo gasto no cálculo do grau de certeza que, no ESPROLOG, é feito inclusive nos programas PROLOG. Pelos resultados podemos notar que esse tempo não compromete a eficiência do interpretador. Chama a atenção que um dos programas testados é mais rápido em ESPROLOG que no PROLOG de Roger, o que mostra que,

deixando-se de lado o tratamento de incertezas, as demais correções e modificações introduzidas tornaram ESPROLOG mais eficiente.

A comparação de ESPROLOG com o ARITY-PROLOG 4.Ø foi feita porque o ARITY PROLOG:

a) é atualmente um dos melhores, mais profissionalmente implementado e mais eficiente, interpretador PROLOG para máquinas PC's;

b) tal como ESPROLOG, faz a coleta de lixo, tem otimização de recursão de cauda e de última chamada.

A tabela 1 mostra que ESPROLOG é apenas cerca de 2,3 a 3,7 vezes mais lento que ARITY-PROLOG. Esse resultado é bastante aceitável, pois este é um trabalho acadêmico preocupado em explorar incertezas diretamente no interpretador, e não um trabalho de muitos homens-ano, preocupados apenas em obter o máximo de eficiência para o PROLOG. A detecção e remoção dos pontos de estrangulamento de ESPROLOG no tempo de execução deverá ser feita numa etapa futura.

Também foi feito um teste comparativo de ESPROLOG, BRODESE e VPX. BRODESE é uma biblioteca integrada para construção de S.E.'s baseados em regras com incertezas, implementado em ARITY PROLOG [TOJA 89]. VPX, tal qual ESPROLOG, é uma linguagem cujo interpretador incorpora um motor de inferências [EXPE 87]. O teste foi feito primeiramente usando um banco de conhecimento adaptado do PROSPECTOR [GASE 82], e depois um sistema

especialista com pouco mais de 100 regras para o diagnóstico de hipertensão arterial (vide resultados na tabela 2 e bancos de conhecimentos no anexo 1).

S.E	ESPROLOG	BRODESE	VPX
prospector	1.71s	4,23s	
hipert	0,11s	4,83s	1,10s

TABELA 2

A comparação VPX-ESPROLOG não pode ser feita com o programa PROSPECTOR, pelo fato de VPX ter a desvantagem de não aceitar FA negativos. No outro sistema especialista, ESPROLOG foi 10 vezes mais rápido.

A análise dos resultados mostra que, em média, ESPROLOG é 20 vezes mais rápido que BRODESE. Isso comprova a nossa hipótese inicial: embutir o tratamento de incertezas no interpretador (enfoque "núcleo") seria melhor que construir um "shell" interpretador das regras e a ser, por sua vez, interpretado em PROLOG (enfoque "casca"). Como BRODESE foi implementado em ARITY-PROLOG, que é em média 3 vezes mais rápido que ESPROLOG, e esse último é 20 vezes mais rápido que BRODESE, o enfoque "núcleo" demonstra ser, em média, 3 x 20 = 60 vezes mais rápido que o enfoque "casca". Isto é, ESPROLOG poderia ser 60 vezes mais rápido que BRODESE. Para isso ocorrer, ESPROLOG teria que ter a eficiência do ARITY PROLOG ou então o interpretador ARITY-PROLOG deveria ser modificado de modo a tratar incertezas diretamente. Outra fato

que chama a atenção é que o sistema especialista PROSPECTOR em BRODESE é somente 2,5 vezes mais lento quando comparado com ESPROLOG, enquanto o sistema de hipertensão arterial é 44 vezes mais lento. Essa diferença pode ser explicada pelo fato de BRODESE armazenar o resultado da resolução de uma cláusula e, se esse resultado for necessário novamente, o cálculo não é refeito. Essa otimização, surtindo efeito no primeiro mas não no segundo banco de conhecimento (por causa de suas peculiaridades), deverá ser feita também no interpretador ESPROLOG.

Um outro aspecto que deve ser salientado é que, para obter o GC de uma consulta, ESPROLOG necessita guardar uma locação na pilha de variáveis para cada inferência lógica feita. Dependendo do número de inferências feitas para se chegar a uma solução, a pilha de variáveis pode se esgotar. Para contornar esse problema pode-se colocar a variável teto na pilha de variáveis locais, de tal forma que ela seja atingida pelas otimizações. A otimização recursão de cauda e a de última chamada não apresentam problemas. Como as variáveis teto são sempre decrescentes ou, na melhor das hipóteses, permanecem constantes, nada impede que um novo ambiente de variáveis seja colocado sobre o anterior. Já na otimização de sucesso de submetas determínisticas, a variável teto só poderá ser abandonada se for idêntica à variável teto do ambiente anterior. Essas otimizações também ficam como sugestão para um trabalho futuro.

6. REFERÊNCIAS BIBILIOGRAFICAS

[BRUY 82] Bruynooghe, M.

The memory manegement of PROLOG implementations

Logic programming, Clark, K. L.; Tarnlund, S. A., eds, New York, Academic, 1982

[BUCH 84] Buchanan, B. G. and Shortliffe, E. A.

Rule-based expert systems

Addison Wesley, 1984

[CASA 87] Casanova, M. A.; Giorno, F. A. C.; Furtado, A. L.

Pogramação em lógica e a linguagem PROLOG

Editora Edgard Blucher Ltda, 1987

[CHEN 84] Cheng, M. H. D.

Design and implementation of the Waterloo Unix PROLOG environment

M. Math. Thesis, Department of Computer Science, University of Waterloo, 1984

[CLAR 78] Clark, K. L.

Negation as Failure

Logic and data bases, Gallaire, H.; Minker, J., eds, New York, Plenum, 1978

[CLAR 84] Clark, K. L. and McCabe, F. G.

Micro-PROLOG: programming in logic

Prentice-Hall, Englewood Cliffs, NJ, 1984

[CLOC 84] Clocksin, W. F. and Mellish, C. S.

Programming in PROLOG

Springer Verlag, 1984

[COLM 82] Colmerauer, A.

PROLOG and infinite trees

Logic programming, Clark, K. L.; Tarnlund, S. A. eds, New York, Academic, 1982

[DONA 89] Donato Jr, E. T.; Cirne Filho, W.

BRODESE: uma bibilioteca para a construção de sistemas especialistas

6 Simpósio Brasileiro em Inteligência Artificial, Rio de Janeiro, 1989

[DUDA 81] Duda, R. O.

The Knowledge Acquisition System

Artificial Intelligence Center, SRI International, Tecnical Report, 1979

[EXPE 87] VP-Expert

Reference manual

[FAIN 82] Fain, J.; Hayes-Roth, F.; Sowizral, H. and Waterman, D.

Programing in ROSIE: an introduction by means of examples

Santa Monica, CA, Rand, 1982

[FEIG 81] Ban, A, and Feigenbaum, E.

The handbook of Artificial Intelligence, Vol 1.

Heuristech Press, 1981

[FERG 81] Fergunson, R. J.

A PROLOG Interpreter for the UNIX Operating System

M. Math. thesis, Department of Computer Science,

University of Watherloo, 1981

[FORG 81] Forgy, C. L.

The OPS 5 user's manual

Computer Science Department, Carnegie-Mellon

University, Pittsburg, PA, 1981

[GREI 80] Greiner, R.

RLL-1: A representational language

Heuristic Programming Project, Computer Science

Department, Stanford University, Stanford CA, 1980

[KERN 78] Kernigan, B. W., and Ritchie, D. M.

The C programming Language

Prentice-Hall, Inc., 1978

[MELL 74] van Melle, W.

Do you wish advice another horn?

Stanford Computer Science Department, Stanford

University, 1794

[MELL 82] Mellish, C. S.

An alternative to structure sharing on the implementation of a PROLOG interpreter

Logic programming, Clark, K. L.; Tarlund, S. A., eds, New York, Academic, 1982

[NILS 82] Nilsson, N. J.

Principles of Artificial Intelligence

Addison Wesley, 1979

[ROGE 86] Roger, A.

Implementação de um interpretador PROLOG com otimizações

Dissertação de mestrado, Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 1986

[RUBE 85] Ruberg, W. M. G. e Silva, H. M.

Proposta de uma ferramenta para sistemas especialistas

VI Congresso da Sociedade Brasileira de Computação, anais, Olinda, 1986

[SILV 86] Silva, H. M. e Ruberg, W. M. J. G.

Proposta de transformação de PROLOG em uma ferramenta de desenvolvimento de sistemas especialistas

6 Congresso Brasileiro de Automação, Belo Horizonte, 1986

[WEIS 79] Weis, S. M. and Kulikowiski, C. A.

EXPERT: a system for developing consultation models

International Joint Conference on Artificial Intelligence, 6., Tokio, 1979

[WEIS 88] Weiss, S. M.; Kulikowsky, C. A.

Guia prático para projetar sistemas especialistas

Livros Técnicos e Científicos Editora S.A. São Paulo, 1988

[WINS 84] Winston, P. H.; Horn, B. K. P.

LISP

Reading, MA., Addison Wesley, 1984

ANEXO 1

O programa **conte**, usado para comparar o tempo de resposta do ESPROLOG com o PROLOG do A. Roger e com o ARITY-PROLOG 4.Ø:

```
((conte Ø Ø))

((conte _n )(subtract _n 1 _n1)(conte _n1))
```

O programa **NREV:**

```
((APPEND nil _a _a)(!))
((APPEND (_a!_b)_c (_a!_d))(APPEND _b _c _d))

((NREV (_x!_1Ø) _1)(NREV _1Ø _11)(APPEND _11 (_x) _1))
((NREV () ()))

((list3Ø (1 2 3 4 5 6 7 8 9 1Ø 11 12 13 14 15 16 17 18 19
          2Ø 21 22 23 24 25 26 27 28 29 3Ø)))

?((list3Ø _X)(NREV _X _Y))
```

O programa **qsort:**

```
((qsort (_x!_l) _r _rØ)(part _l _x _l1 _l2)
     (qsort _l2 _r1 _rØ)
     (qsort _l1 _r (_x!_r1)))

((qsort () _r _r))

((part ( _x!_l) _y (_x!_l1) _l2)(le _x _y)(!)
     (part ( -l _y _l1 _l2)

((part (_x!_l) _y _l1) (_x!_l2)(part _l _y _l1 _l2))

((part () _l () ()))
```

A tabela 1 do capítulo 5 traz os tempos de execução em ESPROLOG, PROLOG (A. Roger) e ARITY-PROLOG 4.Ø, das seguintes

consultas:

```
?((conte 750)

?((list 30 _x)(NREV _x _y))

?((qsort ( 27 74 17 33 94 10 46 83 65 2 32 53 28 85  99)  _k
() ))
```

Subconjunto do banco de dados do PROSPECTOR usado para testes comparativos com BRODESE [DONA 89] e VPX [EXPE 87]:

REGRAS

```
((amb_reg_fav_c_por ) 20 (falhas_ext_pre_int ) )
((amb_reg_fav_c_por ) -10 (NOT (falhas_ext_pre_int ) ) )
((amb_reg_fav_c_por ) 100 (nivel_ero_fav ) )
((amb_reg_fav_c_por ) -100 (NOT (nivel_ero_fav ) ) )

((nivel_ero_fav ) 90 (hiper_ab ) )
((nivel_ero_fav ) -100 (NOT (hiper_ab ) ) )
((nivel_ero_fav ) 100 (roc_vul_coe ) )

((hiper_ab ) 80 (tex_roc_ign ) )
((hiper_ab ) -80 (NOT (tex_roc_ign ) ) )
((hiper_ab ) 100 (morf_roc_ign ) )
((hiper_ab ) -100 (NOT (morf_roc_ign ) ) )

((tex_roc_ign ) 5 (grao_fino_med ) )
((tex_roc_ign ) 95 (grao_fino_med )(tex_porf ) )
((tex_roc_ign ) -100 (NOT (grao_fino_med ) ) )
((tex_roc_ign ) -100 (NOT (tex_porf ) ) )

((morf_roc_ign ) 5 (plugs_int ) )
((morf_roc_ign ) 10 (brechs_int ) )
((morf_roc_ign ) 50 (diques_int ) )
((morf_roc_ign ) 100 (estoqs_int ) )
((morf_roc_ign ) -100 (NOT (plugs_int ) )(NOT (diques_int ) )
                      (NOT (estoqs_int ) ) )
```

FATOS

```
((falhas_ext_pre_int ) 70 )
((roc_vul_coe ) -70 )
((grao_fino_med ) 100 )
((tex_porf ) -50 )
((plugs_int ) 20 )
((brechs_int ) 0 )
((diques_int ) 80 )
```

((estoqs_int) 30)

Os tempos da consulta ?((hiper_ab)) estão na tabela 2 do capítulo 5.

Significado dos mneumônicos do programa Prospector

```
sinonimo(falhas_ext_pre_int,'Sistema de Falhas Extensas
                                    e Pre-Intrusivas').
sinonimo(roc_vul_coe,        'Rochas Vulcanicas Coevais').
sinonimo(grao_fino_med,      'Grao de Fino a Medio').
sinonimo(tex_porf,           'Textura Porfiritica').
sinonimo(plugs_int,          'Plugs Vulcanicos').
sinonimo(brechs_int,         'Brechas Intrusivas').
sinonimo(diques_int,         'Diques Intrusivos').
sinonimo(estoqs_int,         'Estoques Intrusivos').
sinonimo(amb_reg_fav_c_por, 'Ambiente Regional Favoravel
                                a Cobre Porfiridico').
sinonimo(nivel_ero_fav,      'Nivel de Erosao Favoravel').
sinonimo(hiper_ab,           'Ambiente Regional Hiper Abissal').
sinonimo(tex_roc_ign,        'Textura Sugestiva de Rochas Igneas').
sinonimo(morf_roc_ign,       'Morfologia Sugestiva de Rochas Igneas').
```

Banco de conhecimento do sitema especialista de diagnóstico de hipertensão arterial.

```
((hleve) 98 (radiald)(radiale))
((hleve) 21 (idade))
((hleve) 21 (ocup))
((hleve) 21 (colest))
((hleve) 21 (anthip))
((hleve) 21 (peso))
((hleve) 21 (cef))
((hleve) 2 (NOT (radiald))(NOT (radiale)))
((hmod) 98 (radiald)(radiale))
((hmod) 21 (anthip))
((hmod) 21 (antagor))
((hmod) 21 (cef))
((hmod) 12 (idade))
((hmod) 12 (ocup))
((hmod) 24 (avc))
((hmod) 12 (ang))
((hmod) 12 (colest))
((hmod) 12 (antavc))
((hmod) 12 (peso))
((hmod) 24 (terap)(doses))
```

```
((hmod) 12 (nic))
((hmod) 12 (ictus))
((hmod) 12 (tont))
((hmod) 12 (taqui))
((hmod) 12 (hemat))
((hmod) 2 (NOT (radiald))(NOT (radiale)))
((hmod) 50 (NOT (radiald))(NOT (radiale))(terap)(doses))
((hsev) 90 (radiald)(radiale).)
((hsev) 20 (sopro))
((hsev) 5 (anthip))
((hsev) 5 (cef))
((hsev) 26 (avc))
((hsev) 26 (icc))
((hsev) 13 (ang))
((hsev) 13 (colest))
((hsev) 26 (antavc))
((hsev) 26 (terap)(doses))
((hsev) 26 (bul3))
((hsev) 13 (fraq))
((hsev) 13 (hemat))
((hsev) 7 (idade))
((hsev) 7 (cor))
((hsev) 7 (rim))
((hsev) 7 (alcool))
((hsev) 7 (antrim))
((hsev) 7 (peso))
((hsev) 7 (fa))
((hsev) 7 (disp))
((hsev) 7 (nic))
((hsev) 7 (tont))
((hsev) 7 (taqui))
((hsev) 7 (insonia))
((hsev) 2 (NOT (radiald))(NOT (radiale)))
((hsev) 50 (NOT (radiald))(NOT (radiale))(terap)(doses))
((diamel) 40 (antdiab))
((diamel) 30 (peso))
((diamel) 80 (antdiab)(peso))
((diamel) 30 (ang))
((diamel) 10 (anthip))
((diamel) 50 (rim))
((diamel) 10 (alcool))
((diamel) 20 (colest))
((diamel) 26 (radiald)(radiale))
((diamel) 26 (fa))
((diamel) 13 (fraq))
((diamel) 10 (idade))
((diamel) 10 (avc))
((diamel) 30 (antrim))
((diamel) 30 (ortmax)(ortmin)(supmax)(supmin))
((diamel) 80 (terap)(doses))
((diamel) 10 (emagr))
((diamel) 30 (nic))
((diamel) 15 (edema))
((diamel) 10 (tont))
((diamel) 10 (sono))
```

```
((diamel) 10 (impo))
((hmal) 50 (avc))
((hmal) 50 (radiald)(radiale))
((hmal) 45 (anthip))
((hmal) 45 (cef))
((hmal) 40 (icc))
((hmal) 20 (ang))
((hmal) 30 (rim))
((hmal) 15 (colest))
((hmal) 30 (terap)(doses))
((hmal) 40 (fa))
((hmal) 26 (bul3))
((hmal) 26 (hemat))
((hmal) 10 (femurd)(femure))
((hmal) 10 (tposd)(tpose))
((hmal) 10 (pediod)(pedioe))
((hmal) 10 (sopro))
((hmal) 10 (antavc))
((hmal) 10 (disp))
((hmal) 10 (emagr))
((hmal) 2 (tont))
((hmal) 2 (fraq))
((hmal) 3 (idade))
((hmal) 10 (cor))
((hmal) 10 (antrim))
((hmal) 3 (peso))
((hmal) 10 (ortmax)(ortmin)(supmax)(supmin))
((hmal) 10 (nic))
((hmal) 7 (edema))
((hmal) 7 (ictus))
((hmal) 7 (boca))
((hmal) 7 (taqui))
((hmal) 7 (insonia))
((hmal) 50 (NOT (radiald))(NOT (radiale))(terap)(doses)(avc))
```

FATOS

```
((radiald) 100)
((radiale) 100)
((idade) 100)
((ocup) 100)
((colest) 10)
((anthip) 100)
((peso) 100)
((cef) 100)
((antagor) 100)
((avc) 100)
((ang) 100)
((antavc) 10)
((terap) 10)
((doses) 10)
((nic) 10)
((ictus) 100)
((tont) 10)
```

```
((taqui) 10)
((hemat) 10)
((sopro) 100)
((icc) 100)
((bul3) 100)
((fraq) 100)
((cor) 100)
((rim) 10)
((alcool) 10)
((antrim) 10)
((fa) 10)
((disp) 10)
((insonia) 10)
((antdiab) 100)
((ortmax) 100)
((ortmin) 100)
((supmax) 10)
((supmin) 10)
((emagr) 10)
((edema) 10)
((sono) 10)
((impo) 10)
((femurd) 100)
((femure) 100)
((tposd) 100)
((tpose) 100)
((pediod) 100)
((pedioe) 100)
((boca) 100)
```

A consulta ?((resolva(hleve) _GC1)(resolva(hmod) _GC2)

(resolva(hsev) _GC3)(resolva(diamel) _GC4)(resolva(hsev) _GC5))

tem como resposta:

SIM

_GC1 = 99

_GC2 = 98

_GC3 = 98

_GC4 = 94

_GC5 = 96

Os tempos dessa consulta estão na tabela 2 do capítulo 5.