LIDIA

UMA LINGUAGEM PARA DESENVOLVIMENTO

DE SISTEMAS ESPECIALISTAS

COM TRATAMENTO DE INCERTEZAS:

PROJETO E IMPLEMENTAÇÃO PARCIAL

JOSé ULISSES FERREIRA JUNIOR

LIDIA - UMA LINGUAGEM PARA DESENVOLVIMENTO DE SISTEMAS ESPECIALISTAS COM TRATAMENTO DE INCERTEZAS: PROJETO E IMPLEMENTAÇÃO PARCIAL

JOSé ULISSES FERREIRA JUNIOR

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

ÁREA DE CONCENTRAÇÃO: CIÊNCIA DA COMPUTAÇÃO

PEDRO SERGIO NICOLLETTI

Orientador

HéLIO DE MENEZES SILVA

Co-Orientador

CAMPINA GRANDE

JUNHO - 1990

F3831 Ferreira Junior, Jose Ulisses
LIDIA : uma linguagem para desenvolvimento de sistemas especialistas com tratamento de incertezas - projeto e implementacao parcial / Jose Ulisses Ferreira Junior. - Campina Grande, 1990.
168 f. : il.

Dissertacao (Mestrado em Informatica) - Universidade Federal da Paraiba, Centro de Ciencias e Tecnologia.

1. Linguagem de Programacao de Computador 2. Sistemas Especialistas 3. Dissertacao I. Nicolletti, Pedro Sergio, M.Sc. II. Silva, Helio de Menezes, M.Sc. III. Universidade Federal da Paraiba - Campina Grande - Campina Grande (PB) IV. Título

CDU 004.43(043)

LIDIA: UMA LINGUAGEM PARA DESENVOLVIMENTO DE SISTEMAS COM TRATAMENTO DE INCERTEZAS: PROJETO E IMPLEMENTAÇAO PARCIAL.

JOSE ULISSES FERREIRA JÜNIOR

DISSERTAÇAO APROVADA EM 08.06.1990

PEDRO SERGIO NICOLLETTI - M.Sc
Presidente

HELIO DE MENEZES SILVA - M.Sc
Componente da Banca

SERGIO DE MELLO SCHNEIDER - Dr.
Componente da Banca

GIUSEPPE MONGIOVI - M.Sc
Componente da Banca

Campina Grande, 08 de junho de 1990

AGRADECIMENTOS

Aos Orientadores Pedro Sergio Nicolletti (o Peter) e Hélio de Menezes Silva pela atenção e amizade, bem como pelas idéias e contribuições incorporadas ao nosso trabalho.

A Isaque Alves de Lyra pela paciência em tirar algumas dúvidas e explicar-me o funcionamento do seu trabalho (IntForBayes) tão interessante, bem como pelas horas dedicadas a discussões referentes aos nossos trabalhos.

À baiana Claudete M. S. Alves (CPD/UFBA), pela competência, carinho e amizade, sem medo de ser feliz. Além de toda ajuda que tem me dado durante esse período, desde a obtenção de uma caixa de formulários até os maiores incentivos e apoios técnicos.

Um "obrigadão" a Marta Chianca (UFPB), pela paciência em revisar todo este texto, e a Luiz Cláudio (CPD/UFBA) pela revisão do *abstract*.

A Tatiana Bellini Rolemberg, da Alliance Française de Salvador, pela atenção, amizade e pela qualidade do suporte dado em tudo que se refere à língua francesa. Merci, Tata, ça va bien.

Ao professor e pesquisador Rodolfo Miguel Baccarelli (CPqD/Telebrás) que sempre foi o meu "Guru". Agradeço a você, Bacca, pela amizade, espírito de cooperação, honestidade, integridade, competência e simplicidade: qualidades de um grande educador.

Aos companheiros do CPD/UFBA que me deram a oportunidade de morar em outro estado a fim de aprimorar o meu conhecimento técnico-científico, o que trará benefícios inestimáveis ao nosso Centro e à comunidade local. Também àqueles que lutam para ampliar o espaço da pesquisa no CPD/UFBA. Aos colegas do CPD que ajudaram, de uma forma ou de outra, na realização deste trabalho.

Aos amigos de Campina Grande que me acolheram muito calorosamente nesta cidade, uma das muitas coisas que os nordestinos sabem muito bem fazer. Aquele abraço a Simone, Salomé, Norma, Ivanilde, Neuma, Linda, Ed, Marcos & Marcia, Fátima & Bruno, Aninha & Jacques, Antão, Joberto e toda a turma. Aos colegas da UFPB que contribuíram, de alguma forma, para a realização deste projeto.

Finalmente, não poderia deixar de agradecer a minha família, em particular a minha mãe Ana Maria. Com certeza, sem esta ajuda, dificilmente chegaria a estudar em uma Universidade.

SUMÁRIO

ÍNDICE DE FIGURAS

LISTA DE ABREVIATURAS

AL	Analisador Léxico
AS	Analisador Sintático
CA	Chance Atual
CP	Chance *a priori*
Fig	Figura
FN	Fator do Não
FS	Fator do Sim
GC	Grau de Certeza
LOO	Linguagem Orientada a Objetos
Mexel	Máquina de Execução LIDIA
MI	Máquina de Inferências
NL	Número de Variáveis Livres
POO	Programação Orientada a Objetos
PA	Probabilidade Atual
PP	Probabilidade *a priori*
Prob.	Probabilidade
SE	Sistema Especialista
SEs	Sistemas Especialistas
SO	Sistema Operacional
Somat	Somatório
TPR	Tabela de Palavras Reservadas

RESUMO

Este texto tanto consiste da definição de uma linguagem de programação (LIDIA) orientada para Sistemas Especialistas, como da descrição da implementação do seu primeiro compilador. LIDIA é uma linguagem de fácil programação que fornece tratamento de incertezas inspirado no modelo bayesiano e permite a definição de procedimentos, expressões booleanas e aritméticas, bem como a ativação de funções e procedimentos externos. LIDIA possui, em seu Mecanismo de Inferências, encadeamento progressivo e regressivo. Possibilita ao usuário atribuir e reatribuir, livremente, valores a qualquer variável do sistema (evidência). O compilador LIDIA foi escrito tendo como objetivo a produção de sistemas legíveis, portáteis, independentes e eficientes. Atualmente, ele se encontra em versão acadêmica, gerando código em C, linguagem na qual será desenvolvida a Máquina de Inferência.

RÉSUMÉ

Ce texte non seulement consiste de la définition d'un langage de programmation (LIDIA) orientée pour des Systèmes Spécialistes, mais aussi, de la description du développement fait par son premier compilateur. LIDIA est un langage de programmation facile qui fournit un traitement d'incertudes, selon la logique Bayesiane et permet la définition de procédés, expressions booleanes et aritmétiques; aussi bien que l'activation de fonctions et procédés externes. LIDIA a dans son Mécanisme d'Inférences un enchainement progressif et regressif. Permet à l'usuaire d'attribuer librement des valeurs a n'importe quelle variable du système (évidence). Le compilateur LIDIA a été écrit ayant comme but (objectif) la production de systèmes lisibles, portables, indépendents et efficients. Il produit un code en C, langage dans lequel sera développer la Machine d'Inférence.

ABSTRACT

This text comprises the definition of an Expert System Oriented Programming Language (LIDIA,) and also describes the implementation of its first compiler. LIDIA is an easy-to-program language that provides uncertainty management proceeding from the bayesian model, and allows procedure definitions, boolean and arithmetic expressions, as well as external function/procedure calls. LIDIA has forward and backward chaining in its inference machine. The user is free to assign and reassign values to any system variable (called evidence). This LIDIA compiler was written to produce readable, portable and efficient stand-alone systems. It's currently implemented as an academic version which is ready to use and generates code in C, language in which the Inference Machine will be written.

CAPÍTULO I

INTRODUÇÃO

Um usuário que pretenda desenvolver um sistema especialista (SE) deverá fazê-lo através de uma linguagem de programação ou através de uma ferramenta para construção de SEs. Normalmente, uma linguagem de programação requer que seus usuários sejam mais especializados que os das ferramentas. Por outro lado, as ferramentas geralmente oferecem menos recursos que as linguagens de programação.

Alguns sistemas especialistas foram desenvolvidos em linguagens de propósitos gerais de terceira geração, como é o caso da linguagem C [KERN 78]. Isto, devido ao fato de que não havia ainda implementações eficientes de linguagens Prolog e LISP, que são mais apropriadas para IA. Até 1986, por exemplo, programas escritos em linguagens para IA eram executados de forma muito mais eficiente em sistemas operacionais escritos em uma linguagem para IA. Assim, LISP era muito bem aceita em uma máquina LISP onde o SO era escrito na mesma linguagem. Os japoneses têm desenvolvido recentemente máquinas Prolog que destacam sistemas

operacionais Prolog e que são especialmente projetados para executar Prolog [HARM 88].

LISP é quase tão antiga quanto a linguagem FORTRAN e é amplamente usada nos Estados Unidos. Foi inventada por John McCarthy. As idéias básicas do LISP são:

1. Computação envolvendo expressões simbólicas em vez de números.

2. Processamento de listas; isto é, representação de dados como estruturas de listas encadeadas na máquina e como listas de vários níveis para o programador.

3. Estrutura de controle baseada na composição de funções para formar funções mais complexas.

4. Recursão, como um método de descrição de processos e problemas.

5. Representação de programas LISP internamente como listas encadeadas e externamente como listas de vários níveis, isto é, na mesma forma em que os dados são representados.

6. A função EVAL, escrita no próprio LISP, serve como um interpretador para o LISP e como uma definição formal da linguagem.

LISP não faz distinção entre programas e dados, o que permite facilmente que um SE modifique linhas do seu próprio código durante a sua execução.

A linguagem Prolog foi desenvolvida por volta de 1970 por Alain Colmerauer e sua equipe da Universidade de Marselha. O Prolog implementa uma versão simplificada do cálculo de predicados. De forma simplificada, pode-se afirmar que programar em Prolog é programar em Lógica. O Prolog é amplamente utilizado em programas de simulação de inteligência, incluindo os sistemas especialistas, bancos de dados dedutivos, processamento de linguagem natural e controle de robôs.

Outra classe de linguagens que vem ganhando em popularidade na construção de Sistemas Especialistas é a das linguagens orientadas a objetos [BYTE 86]. O conceito de *programação orientada a objetos* (POO) foi introduzido através da linguagem Smalltalk [BYTE 81]. Esta linguagem foi desenvolvida no inicio dos anos 70, por Alan Kay e Dan Ingalls da Xerox, no Centro de Pesquisas Palo Alto (PARK), Califórnia. As linguagens orientadas a objetos são mais conceituais que LISP e Prolog. Elas possuem *abstração de dados*, *amarração dinâmica (dynamic binding)* [GHEZ 85] e *herança*. Alguns exemplos, além da Smalltalk, são: Object Pascal, Object Logo, Objective-C, C++ e Neon.

As ferramentas de *software* para construção de SEs, podem ser classificadas como *indutivas*, *baseadas em regras simples*, *em regras estruturadas*, *ferramentas híbridas* e *de domínios específicos* [HARM 88].

As ferramentas indutivas geram regras a partir de exemplos. Estas ferramentas são derivadas de experimentos em aprendizado automático (*machine learning*). Elas operam em mini e microcomputadores.

As ferramentas baseadas em regras simples usam regras do tipo *se-então* para representar o conhecimento. Estas ferramentas são muito eficientes para o usuário que pretenda desenvolver Sistemas Especialistas que contenham menos de 500 regras. Elas são simples, comparadas aos sistemas de regras estruturadas, e não constam de *árvores de contexto* nem outras facilidades de edição, que normalmente fazem parte das ferramentas baseadas em regras estruturadas.

As ferramentas baseadas em regras estruturadas tendem a oferecer *árvores de contexto*, *instanciação múltipla*, *fatores de certeza* semelhantes aos do EMYCIN e editores mais poderosos. Estas ferramentas usam regras *se-então* agrupadas em *conjuntos*. Estes conjuntos são como bases de conhecimento separadas. Um conjunto de regras pode herdar uma informação

que foi adquirida quando o outro conjunto de regras estava sendo examinado.

As ferramentas híbridas representam os ambientes de desenvolvimento de Sistemas Especialistas mais complexos, atualmente disponíveis. Até recentemente, estas ferramentas eram disponíveis apenas em LISP e eram executadas apenas em máquinas LISP e estações de trabalho VAX/UNIX, configuradas para LISP. Estas ferramentas usam técnicas de POO para representar elementos de cada problema, que o sistema tratará como objetos. Um objeto, por sua vez, contém fatos, regras *se-então* ou apontadores para outros objetos. Estas ferramentas são muito mais difíceis de serem usadas e, tipicamente, necessitam de algum conhecimento do LISP, além de um computador de maior porte.

As ferramentas de domínios específicos são especialmente projetadas para serem usadas apenas para desenvolver sistemas especialistas para um domínio particular. Elas podem incorporar qualquer das técnicas listadas acima e, assim, podem ser classificadas dentro das categorias expostas anteriormente. Contudo, estas ferramentas provêem métodos e interfaces que possibilitam o desenvolvimento de um sistema especialista, em um domínio particular, mais rapidamente que as ferramentas listadas acima. Por isso, elas têm uma categoria especial. Pode-se

esperar que esta categoria se expandirá rapidamente, nos próximos anos.

Este trabalho trata do projeto de uma linguagem de programação para Sistemas Especialistas - LIDIA - e sua primeira implementação. Eventualmente, a implementação de LIDIA poderá ser ampliada de modo que se tenha uma ferramenta com edição de texto; compilação e depuração de programas; árvores de contexto e explanação integrados em um único ambiente.

A idéia de fazer este trabalho nasceu da fusão de duas vontades: a primeira, conhecer ferramentas para Sistemas Especialistas e Inteligência Artificial, em termos mais abrangentes. A segunda, desenvolver um compilador. A idéia inicial era implementar uma ferramenta semelhante ao VP-EXPERT [HARM 88], implementando melhorias onde elas fossem necessárias. Surgiram, porém, novas idéias que mudariam significativamente a estrutura da linguagem. Essas idéias vieram da proposta de se desenvolver uma ferramenta que usasse a Máquina de Inferências do IntForBayes [LYRA 89] (um *shell* desenvolvido no DSC/UFPB para Sistemas Especialistas Bayesianos de caminhamento progressivo e que busca fazer perguntas ao usuário segundo um critério "inteligente"). A proposta foi aceita, ampliando-se esta máquina de modo que a mesma pudesse trabalhar também com procedimentos,

expressões, etc [FERR 88]. Verificou-se que uma linguagem de programação era mais expressiva e poderosa que uma ferramenta interativa, devido ao fato de ela possuir, entre outros recursos, seqüências de comandos e chamadas a procedimentos. Assim, nasceu LIDIA (uma linguagem de nível profissional orientada para sistemas com base em conhecimento) e o seu primeiro compilador.

Um dos fatores que motivaram o projeto foi o fato de já se ter desenvolvido na UFPB alguns sistemas especialistas e ferramentas interativas para desenvolvimento de SEs. Entendemos que este trabalho é, para nós, um passo a frente no domínio da tecnologia do desenvolvimento de SEs. LIDIA utilizou-se da experiência do Sindromus [NICO 87], um Sistema Especialista bayesiano para diagnóstico de síndromes de malformações congênitas, e do IntForBayes.

A linguagem LIDIA tem aplicação em diversas áreas do conhecimento: nas *Ciências da Saúde*, através do diagnóstico e monitoração de pacientes em uma UTI; na *Indústria*, através do controle de processos; na *Geologia*, através da identificação de terrenos propícios a um certo mineral; na *Biologia*, através da identificação de animais e plantas; na *Computação*, através da identificação e correção de falhas numa rede de computadores; na *Meteorologia*, através da previsão do tempo; na *Literatura*, através da identificação

de estilos e autores; na *Psicologia*, através da análise da grafia; na *Educação*, através do estudo de bases de conhecimentos especializados, etc.

A linguagem LIDIA possui várias características que a distinguem das demais. Os parágrafos seguintes expõem tais características e fazem comparações com outras linguagens.

A maioria dos sistemas especialistas foram desenvolvidos em LISP, Prolog, linguagens orientadas a objetos (como SmallTalk e C++), linguagens procedimentais (como Pascal e C) e ferramentas específicas para construção de SEs [WATE 86]. LIDIA é mais simples que LISP e Prolog [FERR 88]. Assemelha-se, em alguns aspectos, às linguagens orientadas a objetos, mas ainda é bem mais simples. Por outro lado, LIDIA não se assemelha a linguagens como Pascal e C, mesmo permitindo definições de procedimentos. Pode ser classificada como no nível das ferramentas específicas para SEs, embora haja diferenças em muitos aspectos quanto às existentes no mercado.

LIDIA, além de prover diversos procedimentos padronizados, permite a ligação com outros programas do usuário, escritos em qualquer linguagem. Essa ligação é feita através de chamadas a procedimentos e funções externas

[GHEZ 85]. O programador deve declarar os procedimentos externos com suas respectivas listas de tipos de parâmetros.

Uma importante característica da linguagem LIDIA (existente também em linguagens como Arity e Turbo Prolog) é a possibilidade da geração de sistemas especialistas independentes de um ambiente integrado de execução. Dessa forma, podemos ter maior eficiência quanto ao tamanho do programa a ser executado.

Linguagens como Prolog [CASA 87] e LISP [WINS 81] são muito poderosas mas não possuem, em si, recursos específicos para tratamento de incertezas [GRAH 88], a exemplo das linguagens orientadas para Sistemas Especialistas. LIDIA é uma linguagem que, apesar de não ser tão flexível como aquelas duas, incorpora importantes facilidades para o programador do SE, entre as quais destaca-se o tratamento de incertezas.

A maioria das linguagens e ferramentas orientadas para sistemas especialistas (incluindo Prolog, EXSYS, Insight 2 e VP-EXPERT [HARM 88]) utilizam *regras de inferência* do tipo "se *antecedente* então *consequente*". LIDIA não usa regras, embora se possa fazer uma correspondência. O conhecimento do especialista é representado através de *redes de inferência* [FIRE 88], que são grafos acíclicos. O modelo envolve

conceitos relativos de *nó-pai* e *nó-filho*. Numa rede de inferências LIDIA, um nó também é chamado de objeto.

Em linguagens como Prolog e ferramentas como VP-EXPERT, a escolha da regra que possui o antecedente a ser provado é feita de forma determinista: escolhe-se a primeira regra, de cima para baixo. LIDIA faz a escolha equivalente (escolha do objeto-pai a ser provado) baseando-se em graus dinâmicos de importância dos antecedentes (objetos-filhos do objeto a ser provado) [FERR 89].

LIDIA faz uso de objetos globais (*variáveis*), expressões booleanas, aritméticas e de inferência, operadores de inferência, comandos, módulos e agrupamento de objetos em níveis hierárquicos. Em LIDIA, porém, não há casamento de padrões (*resolução*) [GENE 87] como há em Prolog, por exemplo. Não há *recursão* [ABEL 85], como nas linguagens LISP, Pascal e C, e nem há o conceito de *herança*, existente nas linguagens orientadas a objetos.

Como será visto na seção 3.8, LIDIA possui o conceito de *símbolo de escopo dinâmico* [GHEZ 85]. Além da possibilidade de declarar o tipo do objeto (no caso do objeto ser uma evidência), pode-se declarar o domínio dos valores que podem ser atribuídos ao mesmo. Ao contrário de algumas ferramentas como VP-EXPERT, não há em LIDIA multi-

instanciamento. Uma evidência ou está livre (não instanciada) ou está instanciada com apenas um valor.

Comparada a outras linguagens para SEs, LIDIA faz o caminhamento nos dois sentidos (*progressivo* e *regressivo*). Uma explanação detalhada sobre este assunto encontra-se nas seções 2.1.1 e 2.1.2. O conceito de *caminhamento* aqui empregado é similar, mas não segue o mesmo algoritmo de *encadeamento* [NILS 82] usado na grande maioria dos outros sistemas e ferramentas. Em Prolog, EXSYS, VP-EXPERT e Insight 2, no entanto, só é provido internamente o encadeamento regressivo.

As ferramentas EXSYS, VP-EXPERT e Insight 2, inspiraram-se no modelo do EMYCIN [BUCH 85]. LIDIA não trabalha com este modelo: utiliza o modelo bayesiano com algumas modificações.

Os SEs gerados podem ser executados em microcomputadores do tipo PC, o que facilita a sua disseminação. Além disso, estão disponíveis três conjuntos de palavras reservadas: em língua portuguesa, francesa e inglesa. Isto também visa facilitar a disseminação da ferramenta entre usuários não programadores. Nos exemplos desta dissertação, utiliza-se palavras reservadas do Inglês para facilitar a distinção entre as palavras da linguagem e

as definidas pelo programador, que estão escritas em português.

Os programas em LIDIA podem conter até quatro módulos especiais: inicialização, consulta, voluntariamento e término. Esses módulos são apresentados detalhadamente nas seções 2.3 (módulos especiais) e 3.3 (estrutura de programa). Com exceção do voluntariamento, esses módulos são ativados automaticamente durante a execução do SE. Estruturas hierárquicas de menus são facilmente definidas pelo programador e apresentadas de forma padrão para o usuário.

É mostrado no capítulo 2 o modelo conceitual, incluindo uma descrição do mecanismo de inferência e de todo o tratamento de incertezas da linguagem. O leitor que queira conhecer a filosofia da linguagem sem entrar nos detalhes (expostos no capítulo 3), poderá ler apenas o capítulo 2.

A linguagem é descrita no capítulo 3. São mostrados seus elementos básicos e como eles se relacionam. A sintaxe é definida formalmente enquanto que os outros componentes, como a descrição léxica dos símbolos e a semântica dos comandos, são definidos em Português corrente. O leitor que queira aprender a programar em LIDIA, deverá ler o segundo capítulo, antes de ler o terceiro.

O capítulo 4 apresenta a especificação do código objeto LIDIA, necessária para a compreensão da implementação da mesma. São mostradas as estruturas geradas pelo compilador e suas ligações.

O Sistema de Execução LIDIA é especificado no capítulo 5. Este sistema ainda não está implementado, de modo que a sua especificação é particularmente útil para um trabalho posterior.

Os aspectos gerais do compilador, incluindo a sua estrutura geral; o funcionamento do Analisador Léxico; a análise sintática; e as principais ações semânticas do compilador são apresentados no sexto capítulo, que mostra também a estrutura das tabelas de símbolos do compilador.

Finalmente, o capítulo 7 é reservado às conclusões e sugestões acerca do trabalho.

No apêndice "A", é mostrado um pequeno exemplo de programa escrito em LIDIA. Este programa tem como objetivo apresentar o uso de algumas definições e comandos da linguagem.

No apêndice "B", é apresentado como ilustração o código gerado pelo exemplo citado no apêndice "A".

O apêndice "C" consiste de um glossário de termos técnicos usados no decorrer da dissertação.

OBSERVAÇÃO IMPORTANTE:

Em todos os exemplos contidos nesta dissertação, não houve nenhuma preocupação em corresponder o conhecimento representado com o mundo real. Os exemplos têm como objetivo apenas ajudar o leitor na compreensão do texto.

CAPÍTULO II

CONCEITOS DA LINGUAGEM LIDIA

Este capítulo descreve conceitos da linguagem LIDIA: tratamento de incertezas, operadores e funções de inferência, estruturas de grupos, módulos especiais, etc.

2.1 TRATAMENTO DE INCERTEZAS

Em sistemas baseados em conhecimento, a incerteza pode ser inerente aos dados ou ao processo de inferência. Entre as diversas técnicas conhecidas de inferência sob incertezas, podemos citar [NEWT 87] a teoria de Dempster-Shafer sobre funções de crença, o modelo MYCIN de confirmação, a teoria dos conjuntos nebulosos de Zadeh e a teoria da probabilidade bayesiana, que é um dos assuntos desta seção. Como exemplo de um sistema especialista bem sucedido, que usa este modelo, podemos citar o PROSPECTOR [FIRE 88] desenvolvido para resolver problemas de exploração mineral.

Um *objeto* em LIDIA pode ser um *fato* ou um *operador*. Um fato pode ser uma *hipótese* ou uma *evidência*. Diz-se que

evidência é um fato perguntável, enquanto que hipótese é um fato dedutível a partir de outro(s) fato(s). Veja a figura 2.1:

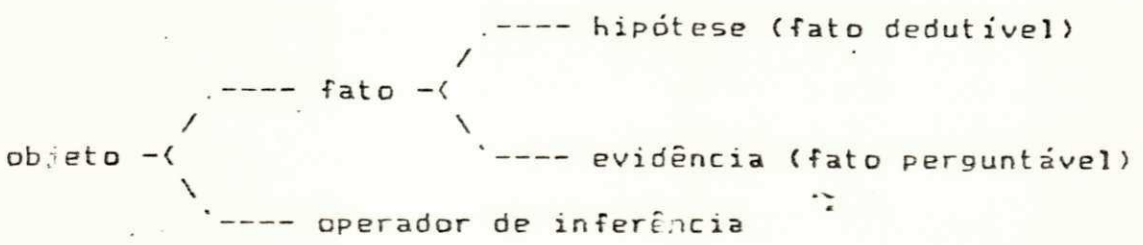


fig. 2.1 - classificação dos objetos LIDIA

Associado a um fato, tem-se o seu *grau de certeza*, que é representado por um valor real entre -1 (negação absoluta) e +1 (confirmação absoluta). Ao iniciar a execução de um programa, os graus de certeza das hipóteses assumem o valor 0 (*zero*) para indicar que nada se sabe a respeito das mesmas. Quanto às evidências, seus graus de certeza são fornecidos diretamente pelo usuário (ou instrumento medidor). O programa em LIDIA, no início da sua execução, "marcam" esses graus de certeza como indefinidos para indicar que as evidências ainda não foram instanciadas. Assim, à proporção em que o sistema vai obtendo graus de certeza das evidências, as hipóteses consequentes vão tendo os seus graus de certeza atualizados, segundo o *caminhamento progressivo*.

Conceitualmente, toda hipótese em LIDIA tem duas probabilidades associadas: *a priori* e *atual*. No início da execução de um programa, a probabilidade atual de cada hipótese é igual à probabilidade *a priori* da mesma, pois não se tem nenhum conhecimento novo acerca das hipóteses.

Internamente, porém, é mais eficiente trabalhar em termos de *chance* do que probabilidade, como pode ser observado na apresentação das fórmulas deste capítulo.

Nem o programador LIDIA nem o usuário dos sistemas precisam conhecer o conceito de *chance*. Este conceito é utilizado apenas pelo implementador do compilador LIDIA e da máquina de inferência. A chance e a probabilidade de uma mesma hipótese se relacionam de acordo com as seguintes equações [LYRA 89]:

chance = probabilidade / (1 - probabilidade)

probabilidade = chance / (1 + chance)

(equação 2.1)

É fácil notar que a chance tem um valor positivo na faixa de zero a infinito. Assim como tem-se, conceitualmente, a probabilidade *a priori* e a probabilidade atual, tem-se internamente, armazenadas em cada hipótese, a sua chance *a priori* (CP) e a chance atual (CA).

2.1.1 Caminhamento Progressivo

O grau de certeza (GC) de uma hipótese é sempre calculado no caminhamento progressivo, segundo a equação 2.2.a (em função das chances) e 2.2.b (em função das probabilidades), cujo gráfico é apresentado na figura 2.1.a [LYRA 89]:

GC = (CA - CP) / (1 + CA), se CA >= CP

GC = (CA - CP) / (CP * (1 + CA)), se CA < CP

(equação 2.2.a)

GC = (PA - PP) / (1 - PP), se PA >= PP

GC = (PA - PP) / PP, se PA < PP

(equacao 2.2.b)

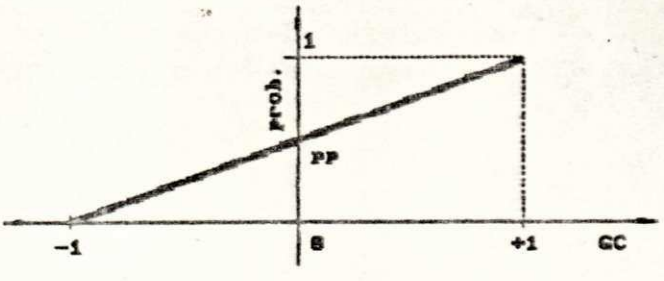


fig 2.1.a - grafico da equacao 2.2.b

O modelo da linguagem LIDIA trabalha com redes de inferências bayesianas [PEAR 86], usando o conceito de *arco*

que representa uma relação de dependência entre um objeto e outro. Assim, diz-se que o fato A é *pai* do fato B se e somente se existe um arco ligando esses dois objetos, que indica a relação "A depende de B". Neste caso, diz-se também que B é *filho* de A. Tal relação é apresentada através da forma B -> A.

A figura 2.2 mostra um trecho de uma rede de inferências que corresponde ao conjunto de regras "se B então A", "se C então A" e "se D então A". Cada arco tem seus atributos que permitem definir, entre outras informações, quanto o filho contribui para o aumento ou diminuição do grau de certeza do pai. Isto é dado pelo

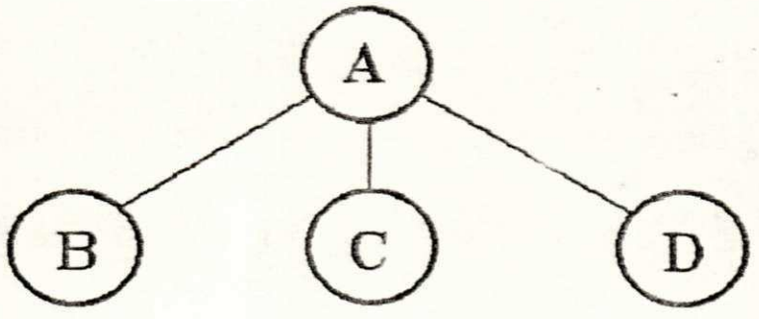


fig. 2.2 - trecho de uma rede de inferências

fator do sim (FS) e pelo *fator do não* (FN). O fator do sim é o valor que será usado (atribuído ao *fator atual*) se o objeto-filho tiver o grau de certeza +1. Por outro lado, se o objeto-filho tiver o grau de certeza -1, usa-se o fator do

não. Para um objeto-filho com grau de certeza intermediário, o mecanismo de inferências LIDIA faz a interpolação linear, de acordo com a função abaixo [LYRA 89], cujo gráfico é apresentado na figura 2.3.

$$fator\_atual = (FS - 1) * GC + 1, \text{ se } GC >= 0;$$

$$fator\_atual = (1 - FN) * GC + 1, \text{ se } GC < 0.$$

(equação 2.3)

Assim, do exemplo da figura 2.2, a chance atual da hipótese A é a sua chance *a priori* multiplicada pelos fatores atuais de todos os arcos relacionados com seus filhos, no caso, B, C e D.

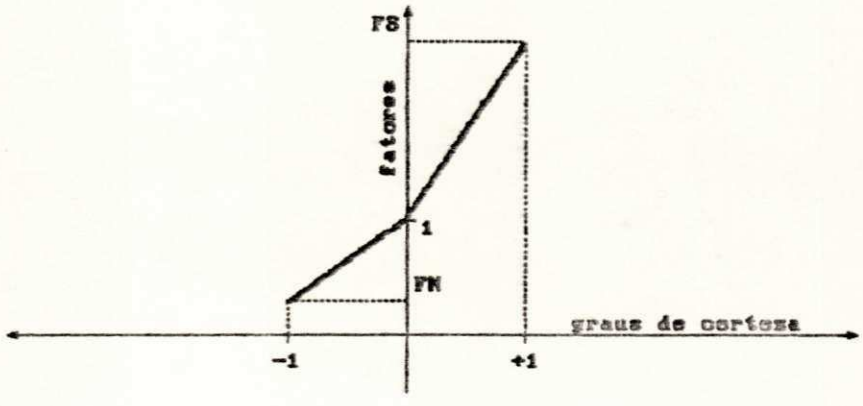


Fig 2.3 - gráfico da equação 2.3

Supondo-se que o fato B tenha o seu grau de certeza alterado (instanciado com um novo valor), como o fato A depende do fato B, é necessário atualizar o grau de certeza de A. Para tanto, divide-se a sua chance atual pelo fator

atual da relação B -> A, e recalcula-se o fator atual da mesma relação, através da equação 2.3. Feito isto, atualiza-se a chance de A, através do comando de atribuição "CA := CA * fator_atual". Finalmente, obtem-se o novo grau de certeza de A através da equação 2.2. Toda vez que se instancia ou altera o grau de certeza de uma evidência, repete-se este processo recursivamente para todos os pais dos objetos que tiverem os seus respectivos graus de certeza atualizados, até chegar às *raizes*, ou seja, aos fatos que não possuem pais.

Os fatores do sim e do não de uma relação entre os objetos A (pai) e B (filho) são constantes fornecidas pelo programador do SE, em função das seguintes probabilidades condicionais [GENE 88]:

$$FS(B,A) = P(B:A) / P(B:\tilde{}A)$$

$$FN(B,A) = (1 - P(B:A)) / (1 - P(B:\tilde{}A))$$

(equação 2.4)

onde P(B:A) é a probabilidade de B dado A; e P(B:~A) é a probabilidade de B dado não A.

O motor de inferências LIDIA, além de calcular e armazenar as chances atuais das hipóteses, também faz o mesmo com os valores mínimos e máximos que essas chances podem alcançar. Esse tipo de previsão permite ao sistema

antecipar decisões, diminuindo consideravelmente, o número de perguntas e, consequentemente, o tempo de resposta a uma consulta. Chamamos então de *piso alcançável*, ou simplesmente *piso*, o menor valor que a chance da hipótese poderá alcançar no futuro e de *teto alcançável*, ou *teto* simplesmente, o maior valor que essa mesma chance poderá alcançar. Para se calcular o teto de uma hipótese, basta supor que todas as evidências não instanciadas, relacionadas a ela diretamente ou não, passem a ter graus de certeza máximos (+1). O mesmo é feito para o piso, porém, supondo que aquelas evidências passem a ter valores mínimos (-1). Assim, ao instanciar uma evidência, serão calculados ascendentemente todos os tetos e pisos das hipóteses dependentes (direta ou indiretamente) daquela evidência, juntamente com as chances atuais, até alcançar as raízes. Naturalmente, o teto sempre tem um valor maior ou igual ao piso de uma hipótese.

Para cada hipótese, o programador LIDIA pode estabelecer o *limiar de aceitação* (ou simplesmente *limiar*), um valor de probabilidade a partir do qual a hipótese é considerada verdadeira. A esse valor, convertido em chance, de acordo com a equação 2.1, chamamos de *limiar interno de aceitação*, ou simplesmente *limiar interno*. Assim, uma hipótese pode estar em *disputa* (se o limiar interno estiver entre o piso e o teto alcançável, inclusive), *classificada* (se o piso for maior ou igual que o limiar interno) ou *desclassificada* (se o teto for menor que o limiar interno).

A figura 2.4 mostra os 3 estados possíveis. Se o limiar de aceitação não for especificado pelo usuário, assume-se o valor 1, que corresponde à probabilidade de 50%.

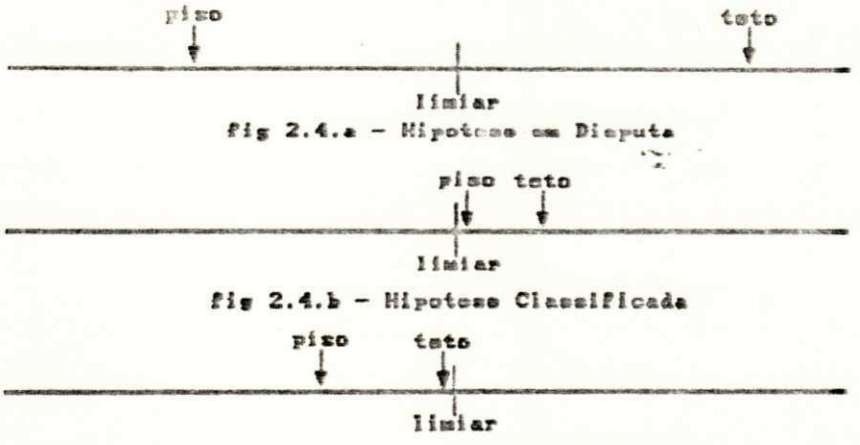


fig 2.4.a – Hipotese em Disputa

fig 2.4.b – Hipotese Classificada

fig 2.4.c – Hipotese Desclassificada

Para se obter o teto e o piso de uma hipótese, deve-se ter um cálculo paralelo ao da chance atual da mesma. Para isto, têm-se também o *fator do piso* e o *fator do teto*, associados aos arcos-filhos da hipótese. Esses fatores são calculados segundo a equação 2.3, sendo que em vez de serem calculados em função do grau de certeza, são calculados em função do *mínimo* e do *máximo graus de certeza*, respectivamente. O mínimo grau de certeza. é calculado, aplicando a equação 2.2, em função do piso alcançável do objeto-filho. O máximo grau de certeza é calculado, através dessa mesma fórmula, em função do teto alcançável do mesmo objeto-filho.

2.1.2 Caminhamento Regressivo

Após o *voluntariamento* (seção 2.3.2), em que é feito o caminhamento progressivo, os graus de certeza das hipóteses raízes podem ser apresentados ao usuário, que poderá a qualquer momento apontar qual dessas hipóteses será explorada ou deixar que o sistema escolha a hipótese que tiver em disputa com o maior GC. Feita esta escolha, faz-se agora o caminhamento regressivo até escolher qual das evidências será pedida ao usuário. Esse caminhamento pode ser feito escolhendo-se sempre ou o filho ou o arco mais importante (a medida *global* ou *local*, respectivamente), a depender da estratégia a ser adotada pelo programador do sistema (esta última, a padrão). A importância de um fato-filho F é uma medida heurística, calculada pela seguinte função [LYRA 89]:

$$imp(F) = Somat[\ P(pai:teto(F)) - P(pai:piso(F))\ ]$$

(equação 2.5)

onde Somat é o somatório para todos os pais de F, em disputa. P(pai:teto(F)) é a probabilidade do pai dado que o fato F atinja o seu teto. P(pai:piso(F)) é a probabilidade do pai dado que o fato F atinja o seu piso. O seguinte algoritmo explica de uma outra forma o cálculo da importância de F, representada pela variável imp.

```
imp := 0;
para cada nó em-disputa(pai(F)), faça:
    p := piso-alcançável(F);
    t := teto-alcançável(F);
    pp := probabilidade(pai(F),p);
    pt := probabilidade(pai(F),t);
    imp := imp + (pt - pp);
fim-para.
```

Para a estratégia de escolha do arco mais importante, temos o conceito de *importância do arco*, que é também uma medida heurística, definida a seguir:

imp = 0, se NL = 0

imp = (FS * piso(pai) - FN * teto(pai)) / NL, se NL > 0

(equação 2.6)

onde FS e FN são os fatores do sim e do não, respectivamente, para o arco em questão. piso(pai) e teto(pai) são o piso e o teto, respectivamente, do objeto-pai do mesmo arco. NL é o número de evidências livres (não instanciadas) que interferem neste arco, isto é, o número de evidências livres subordinadas ao mesmo.

Na equação 2.6, se NL é 0 (zero), então não há mais o que explorar abaixo do arco e, por isso, a importância do arco se torna 0 (zero). Caso contrário (NL > 0), imp cresce

na medida em que é menor NL (menos perguntas a fazer) e em que é maior a *sensibilidade* do objeto-pai em relação ao arco. Esta sensibilidade está representada pelo numerador da fração. Quanto maior for FS, maior se tornará o piso do objeto-pai e, consequentemente, a sua sensibilidade e a importância do arco. Por outro lado, quanto menor for FN, menor se tornará o teto do objeto-pai e, consequentemente, maior será a sua sensibilidade e a importância do arco. Pela mesma equação, quanto maior estiver o piso atual do objeto-pai e menor estiver o teto atual do mesmo, mais facilmente o objeto-pai deixará de estar em disputa e, portanto, maior será a importância do arco.

A importância do fato-filho é tomada como base na capacidade que tem este filho de influenciar todos os seus fatos-pais e, por isto, é uma medida global. Esta medida é importante quando se quer que os fatos alterem, de forma rápida, as raízes da rede de inferências. Neste caso, há uma tendência do sistema a fazer perguntas um tanto desconexas, isto é, fazer uma pergunta relacionada a uma sub-árvore e, logo em seguida, fazer uma outra pergunta relacionada a uma outra sub-árvore. Esta estratégia pode ser útil e deve ser usada quando a entrada de dados for obtida através de equipamentos.

Um exemplo de aplicação em que deve ser adotada a estratégia de escolha dos fatos-filhos mais importantes, é o

monitoramento simultâneo de vários pacientes em um hospital. Cada quarto corresponde a um paciente e, consequentemente, a uma ou mais hipóteses-raízes do sistema. A cada hipótese, corresponde uma ação específica (por exemplo, uma mensagem ao terminal da sala de controle). As evidências são obtidas a partir de aparelhos de medição, instalados nos pacientes, e influenciam diretamente as hipóteses. Estas medidas são obtidas em intervalos regulares de tempo, previamente estabelecidos. Cada vez que uma evidência é instanciada (uma medição feita em um paciente), o processamento é interrompido e faz-se a atualização da rede progressivamente, da mesma forma que, em um outro sistema, um instanciamento pode ser feito voluntariamente pelo usuário a partir do teclado (seção 2.3.2). Contudo, entre duas medições regulares, o sistema pode verificar qual o paciente que necessita de maior atenção, isto é, qual a medição extraordinária que deve ser feita. Se, no nosso modelo, uma medida pode influenciar mais de uma hipótese, então é melhor que se faça uso da estratégia global, da escolha do filho mais importante.

A medida do arco mais importante é tomada como base na capacidade de um filho influenciar o seu próprio pai em avaliação. Portanto é uma medida local e tende a escolher o caminho que faz menos perguntas, uma vez que é considerado o número de evidências livres subordinadas ao arco.

Um exemplo de aplicação em que esta última escolha é mais adequada, é um sistema que sugere ao usuário uma cidade para onde ele irá passar as férias. Para efeito de simplificação, não serão apresentados, no exemplo, os graus de certeza das respostas.

Em um determinado instante, o sistema resolve explorar a hipótese "Salvador" (que têm uma alta probabilidade *a priori*) e faz as seguintes perguntas:

- *O usuário gosta de cidade grande?*
- sim!
- *O usuário gosta de ir à praia?*
- sim!!
- *Gosta de comer frutos do mar?*
- sim!!
- *Gosta de tomar água-de-côco?*
- sim!!!

As respostas positivas do usuário aumentam a chance da hipótese "Salvador". Então, o sistema continua a sua investigação:

- *Deseja conhecer belas igrejas?*
- Sim!
- *Aprecia a arquitetura colonial?*
- Sim!!!

As duas últimas respostas aumentam a chance da hipótese "Salvador" e tudo leva a crer que o usuário está mesmo decidido a conhecê-la. Mas estas mesmas respostas aumentam ainda mais a hipótese "Ouro Preto" que é uma cidade inteiramente tombada como Patrimônio Histórico da Humanidade. Desse modo, a hipótese "Ouro Preto" passa a ter uma chance maior que a da hipótese "Salvador". O sistema que adota a estratégia da medida local jamais perguntará, em seguida, se o usuário quer conhecer a história da Inconfidência Mineira ou se ele quer fazer passeios de trem, a menos que o usuário interfira, pedindo para que o sistema explore esta nova opção. O sistema, se bem projetado, pode utilizar outros recursos para descartar hipóteses e perguntas.

2.1.3 Operadores de Inferência

LIDIA tem um operador especial, o *operador de inferência*, que opera diretamente na rede de inferências. Sua função é acentuar ou abrandar a influência coletiva dos filhos de um objeto.

A figura 2.5 mostra um trecho de uma rede com um operador de inferência entre a hipótese A e os fatos B, C e D. O operador funciona da seguinte maneira: se todos os fatos filhos do operador forem *verdadeiros*, a hipótese A verá tanto os fatores do sim de B, C e D como o fator do sim

do operador e, neste caso, diz-se que o resultado da operação é *verdadeiro*. Por outro lado, se todos os fatos filhos do operador forem *falsos*, a hipótese A terá tanto os fatores do não de B, C e D como o fator do não do operador. Neste caso, diz-se que o resultado da operação é *falso*. Numa condição intermediária, não entrarão no cálculo da chance de A os fatores do operador, mas apenas dos seus filhos, como se o operador não estivesse entre os objetos pai e filho(s). Neste caso, diz-se que o resultado da operação é *intermediário*. Chama-se de *fato verdadeiro* a uma hipótese classificada, ou a uma evidência cujo grau de certeza é maior ou igual a 0.7, ou ainda a um operador de inferência cujo resultado é verdadeiro. Por outro lado, um *fato falso* pode ser uma hipótese desclassificada, uma evidência cujo grau de certeza é menor ou igual a -0.7, ou ainda um operador de inferência cujo resultado é falso.

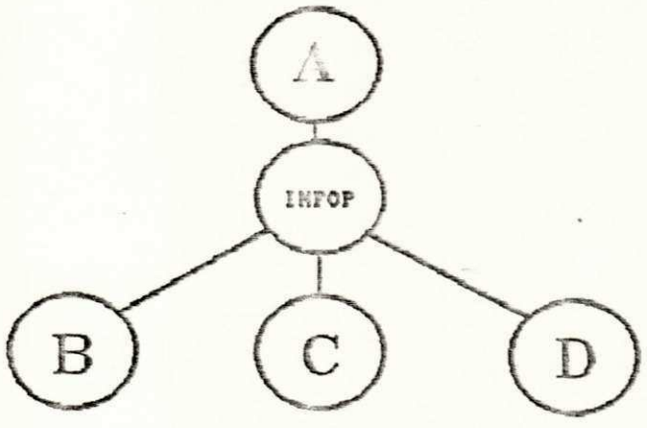


Fig. 2.5 - operador de inferência

Na condição especial de um operador de inferência ter o fator do não igual a um e o fator do sim maior que um (por exemplo, a figura 2.6), diz-se que este operador é um AND-de-inferência e, para facilitar a clareza do programa, o usuário pode usar a palavra reservada IAND.

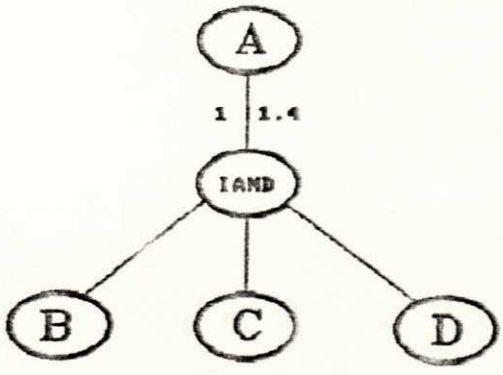


fig 2.6 - Operador IAND

O OR-de-inferência é constituído quando se tem um operador cujo fator do sim é 1 e o fator do não é menor que 1 (por exemplo, a figura 2.7). Neste caso, o programador pode utilizar a palavra reservada IOR.

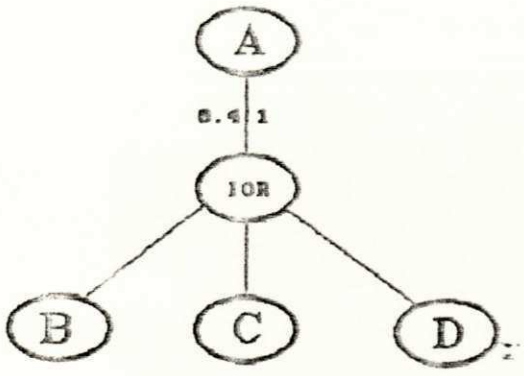


fig 2.7 - Operador IOR

As palavras INFOP, IAND e IOR são sinônimas, isto é, possuem o mesmo efeito na execução do programa. A escolha do nome apropriado facilita a leitura e o entendimento dos programas.

2.1.4 Outros Operadores

LIDIA possui um conjunto de operadores de expressão lógica, relacional e aritmética, além de funções pré-definidas e possibilidade de chamadas a funções externas. No entanto, estas operações não manipulam incertezas, sendo, portanto, uma limitação da linguagem. Remover esta limitação é, porém, uma tarefa muito complexa. Eis um exemplo simples e típico:

Seja o arco representado pela expressão "se altura >= 1.80 então homem_alto". Supõe-se agora que a evidência

altura receba o valor 1.79 com o grau de certeza 0.5. Naturalmente, isto não é suficiente para se calcular o grau de certeza do homem ser alto. Para isto, seria necessário introduzir na linguagem o conceito de distribuição de probabilidade. Em vez de se ter fatores do sim e do não associados ao arco, ter-se-ia uma *função de probabilidade*. Devido à simplicidade do exemplo dado, esta função aparenta ser simples, porém, se torna extremamente complicada quando se trata de expressões mais complexas. Por esta razão foi eliminado, em primeira instância, o uso de incertezas com variáveis numéricas. As expressões relacionais e/ou lógicas, quando satisfeitas, passam a ter grau de certeza +1 e, quando não satisfeitas, passam a ter grau de certeza -1.

Embora não seja provida pela linguagem a manipulação da incerteza com operadores lógicos, relacionais e aritméticos, o programador LIDIA pode (através de funções externas que recebam, como parâmetro, endereço(s) de objeto(s) da rede de inferências) construir operadores que tratem incerteza da forma desejada.

2.1.5 Arquivo de Contexto

É um arquivo de formato texto em que são armazenados os estados internos (variáveis) dos objetos da rede, tais como graus de certeza ou conteúdos das evidências; graus de certeza, chances e probabilidades de hipóteses e indicação

de quais hipóteses e/ou evidências foram descartadas. Este arquivo tem como objetivo permitir que o conhecimento obtido em uma consulta seja utilizado posteriormente. Exemplo de arquivo de contexto:

```
nome=Anjolina Grisi de Oliveira
idade=24
motivo=pendencia: foi fazer exames.
dor_de_cabeca=-0.3
febre=-1.0
coriza=0.4
dor_de_dente=-1.0
```

2.1.6 Funções de Inferência

O Sistema de Execução LIDIA provê algumas funções de inferência, disponíveis para o programador. A saber, zap, loadfacts, savefacts, discard e free, apresentados nesta seção.

a) zap()

Esta função coloca a rede de inferências no seu estado inicial. As hipóteses passam a possuir graus de certeza 0 (zero) e probabilidades (chances) atuais iguais às probabilidades (chances) *a priori*. As evidências passam a ser indicadas como não instanciadas.

b) loadfacts(char *nome_arquivo)

Esta função recebe como parâmetro um nome de arquivo de contexto. Ao ser ativada, ela lê do arquivo as evidências com seus respectivos valores, os estados internos das hipóteses e dos operadores.

c) savefacts(char *nome_arquivo)

savefacts grava, no arquivo de contexto especificado como argumento, os graus de certeza e/ou conteúdos das evidências instanciadas, os estados internos das hipóteses e operadores de inferência. A rede de inferências não sofre nenhuma alteração.

d) discard(char *endereco)

Esta função deve ser usada sempre que o programador precisar afastar uma determinada hipótese ou negar uma evidência. A função recebe como parâmetro uma evidência ou hipótese. No primeiro caso, a evidência é instanciada com grau de certeza -1. No segundo, a função força a desclassificação da hipótese atribuindo, à mesma, um grau de certeza -1. Em ambos os casos, a função atualiza a rede, seguindo o caminhamento progressivo, e retorna o controle ao comando seguinte à chamada desta função.

e) free(char *evid)

Ao ser chamada, esta função recebe como parâmetro uma evidência e marca a mesma como não instanciada. A rede é atualizada progressivamente, e o controle da execução é retornado ao comando seguinte à chamada desta função. Esta é comumente usada antes de se fazer a leitura de uma evidência já instanciada.

2.2 ESTRUTURAS DE GRUPOS

Em LIDIA, podemos agrupar evidências e nomear esses grupos, formando assim, uma hierarquia de grupos e objetos. Esta hierarquia é usada na apresentação de menus para o usuário, visando uma melhor organização durante a escolha da(s) evidência(s) a ser(em) instanciadas. Exemplo:

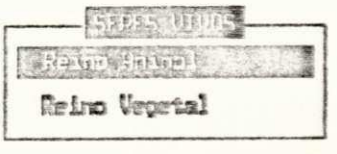
```
...
group vertebrados: peixe, reptil, anfibio, ave, mamifero;
group invertebrados;
...
group animal = "Reino Animal": vertebrados, invertebrados;
group vegetal = "Reino Vegetal";
...
group vivos = "Seres Vivos": animal, vegetal;
...
```

Neste exemplo, os grupos são vivos, animal, vegetal, vertebrados, invertebrados e provavelmente outros objetos cujas definições não foram mostradas no exemplo.

Após definir os grupos, o programador pode ativar o sistema de menus através do comando

menu vivos

que provoca o aparecimento da janela de menu-principal, mostrada na figura 2.8.a.



2.8.a - Menu Principal

2.8.b - Menu Secundario

"Seres Vivos" é o rótulo do menu e "Reino Animal" e "Reino Vegetal" são as opções do mesmo. O usuário escolhe a sua opção e confirma com a tecla [Enter]. Supondo que ele escolha a primeira opção deste exemplo, aparecerá um novo menu com rótulo "Reino Animal", como mostra a figura 2.8.b.

A qualquer momento, o usuário pode pressionar a tecla [Esc] para retornar ao menu anterior ou, caso esteja no nível raiz, concluir o instanciamento das evidências. Assim, o usuário escolhe as opções "navegando" pelos menus. Sempre que ele chega a uma folha, isto é, a uma evidência, o sistema pede o valor e/ou o grau de certeza dessa evidência escolhida.

O usuário tem total liberdade para instanciar várias evidências da estrutura dos menus. No entando, ele terá acesso apenas aos objetos que estejam subordinados ao grupo raiz, referenciado no comando MENU.

2.3 MÓDULOS ESPECIAIS

LIDIA possui quatro módulos especiais. A saber, *inicialização*, *voluntariamento*, *consulta* e *término*. Esses módulos são únicos dentro de um programa, isto é, não pode haver mais de um módulo de consulta, de inicialização, etc. Os módulos especiais são apresentados nesta seção.

2.3.1 Inicialização

Neste módulo, o usuário especifica os comandos que devem ser executados quando o SE é ativado. Este módulo inicializa automaticamente todos os objetos do programa e

passa o controle para o seu primeiro comando. Após a execução do módulo, o controle é passado para o módulo de consulta.

O módulo de inicialização é ativado automaticamente pelo programa logo após o mesmo ter sido carregado pelo Sistema Operacional da máquina.

2.3.2 Voluntariamento

Voluntariamento é um módulo destinado a fazer o instanciamento de evidências. Ele pode ser ativado através de uma chamada pelo programa ou de uma tecla de interrupção que o usuário tem ao seu dispor. Neste módulo, podemos ter comandos, tipicamente chamadas a procedimentos e menus. Sempre que um objeto é instanciado, é feita a atualização da rede de inferências, segundo o caminhamento progressivo.

2.3.3 Consulta

O módulo de consulta é o módulo principal do programa e sua definição é obrigatória. Ele contém comandos para chamar o módulo de voluntariamento, o módulo de inferências, procedimentos, etc. O módulo de consulta é ativado automaticamente, logo após a execução do módulo de inicialização ou através do comando RECONSULT.

2.3.4 Término

Este módulo é ativado automaticamente pelo programa, após a conclusão do módulo de consulta. Em geral, o módulo de término contém comandos para fechar arquivos, imprimir totais de relatórios e tudo que for necessário ser processado antes do término da execução do programa.

CAPÍTULO III

DEFINIÇÃO DA LINGUAGEM

Este capítulo, tem como objetivo apresentar os elementos léxicos, a sintaxe e a semântica da linguagem LIDIA. Para uma maior compreensão do mesmo, o leitor deve ler o capítulo 2, que apresenta conceitos da linguagem.

3.1 META-LINGUAGEM

Para se definir formalmente uma linguagem, necessita-se de uma outra, denominada *meta-linguagem*. Na definição da LIDIA, será usada a notação BNF (Backus-Naur Form), com as seguintes considerações:

* o símbolo ":-" será usado para denotar uma regra de produção. Ele separa o *não terminal* à esquerda da produção, da *forma sentencial* à direita da mesma.

* o símbolo "&" será usado para indicar a ausência de símbolos em uma derivação (uma forma sentencial vazia).

* o símbolo "|" será utilizado para simplificar a notação, quando houver mais de uma produção para um mesmo símbolo à esquerda do sinal ":-". Assim, define-se este último símbolo apenas uma vez, e coloca-se à direita do sinal ":-", as formas sentenciais separadas por "|" para indicar as possíveis alternativas.

* as palávras reservadas da LIDIA aparecerão com todas as suas letras maiúsculas. Embora LIDIA utilize palavras de três idiomas, na descrição da sintaxe, serão usadas apenas as palavras de idioma inglês para representar as correspondentes dos três idiomas. A tabela do item 3.2.1.1.b mostra as correspondências entre palavras.

* os símbolos não terminais da gramática aparecerão em minúsculas e negrito. Os símbolos especiais da linguagem aparecerão entre apóstrofos. Símbolos terminais especificados através de um padrão léxico (*identificador*, *constante_inteira*, etc.) são apresentados em itálico.

* comentários aparecerão no meio das regras entre os símbolos "(" e ")". Os comentários ajudarão ao leitor a entender as definições.

3.2 ELEMENTOS BÁSICOS

3.2.1 Palavras

Uma palavra é uma sequência de letras, dígitos e sublinhas. Ela não pode, porém, ser iniciada com um dígito.

3.2.1.1 Palavras Reservadas e Vocabulários

Em LIDIA, há três conjuntos, chamados *vocabulários*, de palavras que são reservadas como símbolos da linguagem. Cada vocabulário, possui palavras em um dos três idiomas: Português, Francês e Inglês. As palavras reservadas, como o nome já diz, não podem ser utilizadas como outra classe de símbolo: identificador, por exemplo.

a) Palavras Reservadas do Inglês (ordem de tamanho)

OR	IF	DO	END	IOR
ASK	MOD	DIV	AND	REF
REAL	SELF	TRUE	IAND	CHAR
VOID	MENU	THEN	ELSE	CALL
PROC	GROUP	WRITE	FALSE	WHILE
INFER	INFOP	ACTION	YES_NO	STRING
VOLUNT	INTEGER	CONSULT	ENGLISH	EVIDENCE
INITPROC	FEEDBACK	EXTERNAL	CHILDREN	REQUISITE
THRESHOLD	TERMINATE	RECONSULT	HYPOTHESIS	VOLUNTPROC

b) Palavras Reservadas do Português e do Francês

A tabela 3.1 apresenta a correspondência entre as palavras, nos três idiomas, por ordem de tamanho.

português	francês	inglês
EE	ET	AND
OU	OU	OR
SE	SI	IF
ATO	ACTION	ACTION
FIM	FIN	END
MOD	MOD	MOD
DIV	DIV	DIV
REF	REF	REF
EINF	ETINF	IAND
ALFA	CORDE	STRING
EXEC	FAIS	DO
REAL	REAL	REAL
NULO	RIEN	VOID
MENU	MENU	MENU
PROC	PROC	PROC
OUINF	OUINF	IOR
GRUPO	GROUPE	GROUP
FALSO	NON	FALSE
CARAC	CARAC	CHAR
ENTAO	ALORS	THEN

SENAO	SINON	ELSE
CHAME	APPELLE	CALL
OPINF	OPINF	INFOP
FILHOS	FILS	CHILDREN
VOLUNT	VOLONT	VOLUNT
LIMIAR	LIMITE	THRESHOLD
INFIRA	INFERE	INFER
ESCREVA	ECRIS	WRITE
SIM_NAO	OUI_NON	YES_NO
INTEIRO	ENTIER	INTEGER
EXTERNO	EXTERNE	EXTERNAL
TERMINO	CONCLUSION	TERMINATE
PROPRIA	PROPRE	SELF
VERDADE	OUI	TRUE
HIPOTESE	HYPOTHESE	HYPOTHESIS
ENQUANTO	TANDIS	WHILE
CONSULTA	CONSULT	CONSULT
PROCINIC	PROCINI	INITPROC
PERGUNTE	DEMANDE	ASK
FEEDBACK	FEEDBACK	FEEDBACK
EVIDENCIA	EVIDENCE	EVIDENCE
REQUISITO	CONNEC	REQUISITE
PORTUGUES	FRANCAIS	ENGLISH
PROCVOLUNT	PROCVOLONT	VOLUNTPROC
RECONSULTE	NOUVCONS	RECONSULT

tabela 3.1 - palavras reservadas

3.2.1.2 Palavras Reservadas Ativas

O programador especifica o idioma (vocabulário) do seu programa através da primeira palavra reservada do mesmo. A partir daí, ele só poderá usar as palavras reservadas desse idioma.

Define-se *palavra reservada ativa* como sendo uma palavra pertencente ao vocabulário do idioma selecionado.

No caso, porém, da primeira palavra ser PROC ou EVIDENCE, que constam em mais de um vocabulário, ainda não se pode saber qual o idioma desejado. O programador simplesmente continua escrevendo seu programa até que uma palavra reservada pertença a um único vocabulário. A partir daí, define-se o conjunto das palavras reservadas ativas.

Outra forma de se resolver o problema da ambiguidade de idiomas (necessário quando se quer usar, no início do programa, uma palavra de outro idioma como identificador), é usar a palavra reservada PORTUGUES, FRANCAIS ou ENGLISH como a primeira do programa, para especificar o idioma no qual se pretende comunicar.

3.2.2 Identificadores

Um identificador é uma palavra que não é reservada ativa. Pode ser usado como nome de *evidência*, *hipótese*, *grupo*, *função* ou *procedimento*. Todo identificador deve ser definido antes de ser referenciado.

3.2.3 Espaços e Caracteres Especiais

Os programas em LIDIA possuem formato livre, isto é: os caracteres "espaço", TAB, CR, LF são ignorados na linguagem, servindo apenas como delimitadores de símbolos.

3.2.4 Comentários

Os comentários aparecem entre os símbolos "/*" e "*/", não podendo haver aninhamento de comentários. A primeira ocorrência do símbolo "*/" fecha o comentário, ficando o restante do programa para os outros símbolos da linguagem.

3.2.5 Constantes

As constantes, em LIDIA, podem ser inteiras, reais, *strings* ou booleanas. Uma constante inteira é uma sequência de um ou mais dígitos. Como exemplo, são constantes inteiras: 30, 11 e 1963.

Uma constante real é uma sequência de um ou mais dígitos, seguida de um ponto, seguido de um ou mais dígitos. Além disso, uma constante real pode ser expressa em forma de potência. Para isso, a letra "E" separa a base do expoente inteiro. O expoente pode conter o sinal "-", quando negativo. Exemplos de constantes reais:

3.141592 0.0001 13.56 1.4E-8

Uma constante *string* é uma sequência de caracteres delimitada por aspas ou apóstrofos. Ela não pode ser aberta com um delimitador e fechada com outro. Exemplos de constantes do tipo *string*:

"Jose' e Maria" '*' 'gripe' "Luiz Ignácio"

Uma constante *string* pode conter o delimitador. Para isto, o programador repete, imediatamente, o caractere delimitador. Exemplo:

'Jose'' e Ana'

Uma constante booleana é uma das palavras reservadas YES ou NO, para representar um valor verdadeiro ou falso, respectivamente.

3.2.6 Símbolos Especiais

LIDIA possui outros símbolos que são usados como operadores, símbolos de pontuação, de agrupamento de expressões e símbolos de outras funções. Um símbolo especial pode ser formado por um ou dois caracteres. Eis os símbolos:

```
<   <=   =   <>   >=   >   :=   +   -   *   /

,   ;   :   [   ]   #   %   &   (   )   {   }
```

3.3 ESTRUTURA DE PROGRAMA

Define-se *programa fonte* como sendo um programa escrito na linguagem LIDIA. Um programa fonte é compilado por um compilador LIDIA, gerando o *programa objeto*, que por sua vez é *ligado* juntamente com os procedimentos externos, produzindo o *código executável*.

Um programa fonte é formado por declarações, definições e comandos. O programador declara o idioma a ser utilizado e as funções externas, caso existam; define procedimentos, evidências, hipóteses, grupos e os módulos de inicialização, solucionariamento, consulta e término. Estas declarações e definições seguem esta mesma ordem. As definições de

hipóteses, evidências e consulta são obrigatórias, enquanto que as outras podem ocorrer ou não no programa.

Os comandos aparecem dentro das definições de procedimentos, evidências, hipóteses, inicialização, voluntariamento, consulta e término.

Assim, um programa fonte possui as seguintes partes:

```
programa :-    dec_procs_ext   ( declara procs externos )
               defprocs        ( definição de procedimentos )
               evidências      ( definição das evidências )
               hipóteses       ( definição das hipóteses )
               agrupamentos    ( definição dos grupos )
               inicialização   ( módulo de inicialização )
               voluntariamento ( módulo de voluntariamento )
               consulta        ( módulo de consulta )
               término         ( módulo de termino )
```

Se o programador inclui, em seu programa fonte, uma chamada a um procedimento externo, naturalmente deve declará-lo como tal. O programador pode ainda definir procedimentos no próprio programa fonte. É obrigatória a definição de pelo menos uma evidência e uma hipótese, para que haja rede. A declaração de grupos e dos módulos de inicialização, voluntariamento e término são facultativas,

enquanto que a declaração do módulo de consulta é obrigatória.

3.4 Declaração de Procedimentos Externos

Em um programa fonte, esta declaração pode existir ou não, a depender da existência ou não de chamadas externas.

```
dec_procs_ext :- & | EXTERNAL lista_dec_procs ';'
lista_dec_procs :- dec_proc | lista_dec_procs ',' dec_proc
```

Em cada declaração de procedimento externo, tem-se opcionalmente, o tipo do procedimento. Caso o tipo não esteja especificado explicitamente, assume-se o tipo inteiro. Caso o programador utilize a palavra VOID para o tipo, assume-se que o procedimento não retorna nenhum valor. Assim, este procedimento não poderá ser usado em uma expressão, como uma função. Em vez disso, ele deve ser usado como um comando isolado.

Em seguida, o programador deve especificar o nome do procedimento, seguido da lista de parâmetros, de acordo com as seguintes regras:

```
dec_proc :- dec_tipo2 identificador '(' dec_tipos ')'
dec_tipo2 :- & | VOID | tipo2
dec_tipos :- & | VOID | lista_de_tipos
lista_de_tipos :- passagem tipo2
                | lista_de_tipos passagem tipo2
passagem :- & | REF
tipo2 :- CHAR | INTEGER | REAL | YES_NO | STRING
```

Observa-se que, entre os parênteses, poderá aparecer uma lista de tipos de parâmetro. Um tipo pode ser inteiro, real, yes_no ou *string*. Associado a cada parâmetro, está a forma de passagem do mesmo. Em LIDIA, existem duas formas de passagem de parâmetros: por *valor* e por *referência*[GHEZ 85]. A forma padrão é a passagem por valor. Mas, caso o programador queira passar parâmetro por referência, poderá usar a palavra REF antes do tipo correspondente ao parâmetro em questão. Exemplo de declaração de procedimentos externos:

```
EXTERNAL  real mede_temperatura(integer),
          void ler_pressao(ref real, ref yes_no),
          yes_no feriado(integer, integer, integer),
          void abrir_comporta();
```

Uma das características, que fez de C uma excelente linguagem é o seu conjunto amplo e previamente definido de funções e operadores padrão. Então, as funções definidas nos arquivos math.h, string.h e time.h da linguagem C também

estão disponíveis ao programador LIDIA, por motivo de simplicidade. Ele não precisa (nem deve) declarar estas funções como externas. Estas funções incluem funções matemáticas (seno, cosseno, etc.), de manipulação de *strings* (concatenação, pesquisa, etc.) e rotinas para obtenção da data e hora do sistema. Ver detalhes em qualquer manual da linguagem C.

3.5 Definição de Procedimentos

O usuário pode definir zero ou mais procedimentos em seu programa. As definições de procedimentos são feitas uma após a outra e devem ficar após as declarações de procedimentos externos e antes das definições de evidências. Veja a construção abaixo:

```
defprocs :- & | defprocs defproc

defproc :- PROC identificador ';'
             comandos
           END ';'
```

onde *identificador* representa o nome do procedimento. Este nome não pode ter sido declarado anteriormente. Um procedimento definido internamente não possui parâmetros nem variáveis locais. Trata-se apenas de uma sequência de comandos LIDIA, podendo ser ativada a partir de um ou mais pontos do programa (veja comando CALL, seção 3.11.13). Os comandos estão definidos na seção 3.11. Exemplo de definições de procedimentos:

```
proc troca;
     temp := a;
     a := b;
     b := temp
end;
proc cabecalho;
     clrscr();
     write "MENU PRINCIPAL"
     gotoxy(10,10);
     write "ponto 10,10"
end;
```

3.6 Definição de Evidências

Uma ou mais classes de evidências devem ser especificadas dentro de um programa fonte, de acordo com a seguinte sintaxe:

```
evidências :- classe_evid | evidências classe_evid
classe_evid :- EVIDENCE dec_nomes dec_tipo domínio ';'
                 requisito
                 comandos
                 ação
               END ';'
```

Onde dec_nomes representa uma ou mais declarações de nomes separadas por vírgula:

dec_nomes :- dec_nome | dec_nomes ',' dec_nome

dec_nome :- *identificador* texto

texto :- & | '=' *constante_string*

Em cada declaração de nome, pode-se ter, associado ao identificador, uma constante *string*. Sempre que se fizer necessária a interação com o usuário, o texto associado ao nome da evidência será apresentado. Caso o texto não esteja especificado, o nome da evidência será mostrado em seu lugar. Exemplo de declaração de nomes:

dor_cab = "Dor de Cabeça", coriza, febre_alta

Associada à declaração de nomes, tem-se a declaração do tipo da(s) evidência(s):

dec_tipo :- & | ':' tipo

tipo :- INTEGER | REAL | YES_NO | STRING

Caso o tipo não esteja especificado explicitamente, assume-se o tipo YES_NO, para as evidências da classe em definição.

O domínio refere-se ao conjunto de valores que poderão ser assumidos pela(s) evidência(s) da classe em definição. A especificação do domínio é importante, pois, automatiza a

verificação das informações introduzidas pelo usuário do SE. O domínio pode ser explícito, formado por duas constantes numéricas ou por uma lista de constantes do tipo *string*. Caso o domínio não esteja especificado (domínio implícito), apenas a verificação do tipo será feita. Caso o domínio seja formado por duas constantes numéricas, o sistema de execução verificará se a informação digitada pelo usuário é um número e se este número está entre as duas constantes especificadas, inclusive. Caso o domínio esteja especificado como uma lista de constantes do tipo *string*, o sistema de execução apresentará, ao usuário, um menu com a lista de constantes, para que ele selecione uma delas, a qual será atribuída a evidência. Obviamente, o tipo do domínio deve ser *compatível* com o tipo da classe de evidência. Eis a definição formal:

domínio :- & | '[' *número* ':' *número* ']' | '[' cadeias ']'

cadeias :- *string* | cadeias ',' *string*

Exemplos de domínio:

[49: 134]

[3.14 : 4]

['verde', 'vermelho', "amarelo"]

Um requisito é uma condição necessária para que uma evidência seja instanciada. é formado por uma expressão booleana, que é sempre calculada antes de se tentar obter o

valor da evidência. Se a expressão for verdadeira, a evidência será instanciada, através de comandos da evidência. Caso contrário, o controle é retornado ao pai que chamou a evidência, indicando que a mesma não foi instanciada. Veja a seguinte definição:

requisito :- & | REQUISITE ':' expressão ';'

Caso o requisito não seja especificado, a evidência é instanciada incondicionalmente, sempre que for pedido o seu valor.

ação é uma sequência de comandos da linguagem que são executados quando a evidência é instanciada como verdadeira (Grau de Certeza maior que zero). A ação é facultativa, se aplica apenas à evidência do tipo YES_NO e sua declaração está subordinada à seguinte sintaxe:

ação :- & | ACTION ':' comandos

Exemplo de uma definição de classe de evidências:

```
evidence otimista, animado, sorridente,
                adaptavel = "adaptável",
                distraido = "distraído",
                humano, honesto, intelectual;
        requisite: posso_perguntar();
            write "Você é ", #self, "? ";
            ask;
        action: sag := sag + 1
    end;
```

No exemplo acima, são definidas as evidências otimista, animado, sorridente, adaptavel, distraido, humano, honesto e intelectual, em apenas uma classe (um único uso da palavra evidence). Observe que, como a linguagem não reconhece os caracteres acentuados, foi necessário definir textos associados às evidências adaptavel e distraido para que os caracteres acentuados (existentes em diversos computadores, mas não pertencentes à codificação padronizada ASCII) sejam apresentados ao usuário. Como o tipo e o domínio não foram escritos, assume-se que estas evidências são do tipo YES_NO e que seus domínios consistem das duas constantes YES e NO. Toda vez que uma dessas evidências for requisitada, o programa ativa a função externa posso_perguntar, que retorna um valor booleano. Se este valor for verdadeiro, a evidência faz uma pergunta ao usuário do tipo

Você é distraído?

e instancia a evidência requisitada, no caso distraido. Se a resposta for afirmativa, o comando

```
sag := sag + 1
```

será executado. Uma vez instanciada a evidência, a rede de inferências é atualizada pelo caminhamento progressivo. O emprego da palavra self está descrito na seção 3.8.

3.7 Definição de Hipóteses

O programador LIDIA deve especificar pelo menos uma hipótese, de acordo com a seguinte regra:

hipóteses :- hipótese | hipóteses hipótese

Cada hipótese é definida dentro da seguinte sintaxe:

```
hipótese :- HYPOTHESIS dec_nome '(' número ')'
                requisito
                limiar
                filhos
                ação
            END ';'
```

Onde dec_nome, requisito e ação foram definidos na seção 3.6. *número* é uma constante inteira ou real, que representa a probabilidade *a priori* da hipótese, em percentagem. Esta constante deve estar na faixa de 0 a 100, inclusive. Por exemplo, se a síndrome x ocorre a cada 20 milhões de nascimentos, podemos declará-la da seguinte forma:

```
hypothesis x(5.0e-6) ...
```

O requisito é a condição necessária para que a hipótese seja avaliada. Ao se fazer o caminhamento regressivo, verifica-se se esta condição é satisfeita. Caso positivo, o caminhamento é continuado normalmente, escolhendo-se o filho ou arco mais importante da hipótese. Caso negativo, retorna-

se ao pai da hipótese a informação de que a hipótese é falsa. Um exemplo típico de aplicação é o caso do diagnóstico "câncer no ovário". Esta hipótese só pode ser considerada verdadeira, caso o paciente seja do sexo feminino. Eis um exemplo, em LIDIA, desta situação:

```
hypothesis can_ov = "Câncer no ovário" (3.0e-4)
     requisite: sexo = "feminino";
     treshold 50;
     children
          dores_no_ovario[0.01,40],
          teste_positivo[1e-10,1e10];
     action: trata_cancer(ovario)
end /* can_ov */;
```

Neste exemplo, os objetos dores_no_ovario e teste_positivo serão avaliados apenas se a evidência sexo tiver o seu conteúdo "feminino".

O limiar é uma constante real, em percentagem, que representa o limite de probabilidade acima do qual a hipótese será considerada verdadeira. Se esta declaração for omitida, assume-se o limiar de 50% para esta hipótese. Sua declaração obedece à seguinte sintaxe:

limiar :- & | THRESHOLD *número* ';'

filhos é a declaração da *lista de filhos* da hipótese. Descreve os relacionamentos da hipótese com outros objetos, dentro da sintaxe abaixo:

```
filhos :- CHILDREN lista_de_filhos ';'
lista_de_filhos :- filho | lista_de_filhos ',' filho
filho :-    identificador fatores
          | expressão fatores
          | inferop '(' lista_de_filhos ')' fatores
fatores :- '[' número ',' número ']'
inferop :- IAND | IOR | INFOP
```

O *identificador* que aparece na definição de filho refere-se a um identificador de hipótese ou evidência. expressão é uma expressão booleana, como definida na seção 3.12. fatores é um par de números que representam os fatores do sim e do não, existente entre a hipótese declarada e um filho da mesma. O menor destes números é tomado como o fator do não, enquanto que o maior destes é atribuído ao fator do sim, independentemente da ordem em que eles aparecem. Exemplo de uma declaração de filhos com três níveis de hierarquia:

```
hypothesis sagitario ...
  children     ior(  otimista[0.1,10],
                     iand( animado[0.4,1.2],
                           sorridente[0.4,2]
                         ) [1, 1.1]
                    ) [0.3, 1],
               iand( gosta_liberdade[1.8,0.2],
                     gosta_aventura[0.80,3],
                     iand( adaptavel[0.6,1.05],
                           tropeca_bastante[1,1.2],
                           distraido[1,1.4]
                         ) [1, 2],
                    ) [1, 1.1],
               gosta_viajar[0.3,1.1],
               iand( humano[0.8,1.5],
                     honesto[0.7,1.3],
                     intelectual[1,1.4]
                    ) [1, 2.5];
  action ...
end;
```

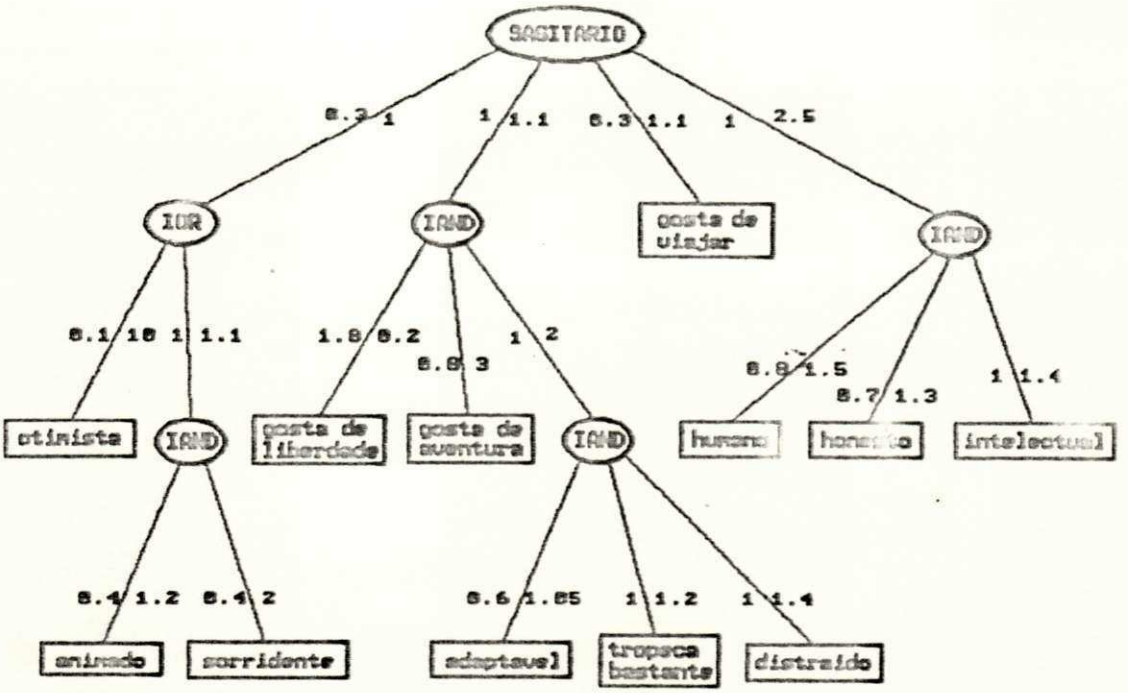


figura 3.1 - Reconhecimento da personalidade tipo sagitário

Supondo que os identificadores não declarados no exemplo sejam evidências, esta lista de filhos corresponde ao trecho da rede de inferências da figura 3.1. Observa-se que o fator do não relacionado à evidência intelectual é 1. Isto significa que a negação desta evidência não altera o grau de certeza da hipótese relacionada.

Da mesma forma que para evidência (seção 3.6), a ação é uma sequência de comandos da linguagem que são executados quando a hipótese passa a ser considerada como classificada.

A ação é facultativa e sua sintaxe está descrita na seção anterior.

3.8 A PALAVRA SELF

SELF é uma palavra reservada que representa o fato atualmente em avaliação durante o processo de inferência. Ela é usada numa expressão contida no corpo da definição de um fato ou em algum procedimento interno, chamado a partir de um ou mais fatos. Na teoria de linguagens, diz-se que SELF corresponde a um identificador de escopo dinâmico, uma vez que o objeto só é identificado durante a sua execução [GHEZ 85].

Quando SELF representa uma evidência, podemos usá-la do lado esquerdo de um comando de atribuição. Neste caso, é a evidência que receberá o valor da expressão. Exemplo:

```
evidence base, altura: real;
     self := 0;
     while self <= 0 do {
       write "valor de ", #self, ": ";
       ask; newline();
       if self <= 0 then
         write #self, " deve ser positiva!", nl
     }
end;
```

```
        evidence triang1, triang2: real;
             free(base); free(altura);
             self := base * altura / 2;
        end;

        evidence conta10: integer;
             self := self + 1;
             if self = 10 then self := 0
        end;
```

3.9 DEFINIÇÃO DE GRUPOS

Define-se grupos simplesmente como uma sequência de zero ou mais símbolos grupo:

```
grupos :- & | grupos grupo
```

Cada grupo é definido da seguinte forma:

```
grupo :- GROUP dec_nome ':' lista_de_ident ';'
lista_de_ident :- identificador |
                  lista_de_ident ',' identificador
```

onde lista_de_ident é uma lista de um ou mais nomes de grupos ou evidências separados por vírgula. Como todo identificador, um grupo deve ser definido antes de ser referenciado. Com isso, temos que, na definição de uma

estrutura de menus, os grupos são definidos das folhas para as raízes. Exemplo:

```
group impr = "Defeitos na impressora":
     impressora_apitando,
     caracteres_extranhos_na_impressora,
     impressora_escreve_fraco,
     caracteres_incompletos,
     impressora_nao_salta_linha,
     impressora_nao_salta_folha;
group tecl = "Defeitos no teclado":
     tecla_nao_funciona,
     tecla_prendendo;
group video = "Defeitos no video":
     imagem_nao_aparece,
     regiao_escurecida,
     caracteres_extranhos_no_video;
group e_s = "Defeitos de Entrada/Saida":
     impr, tecl, video;
```

3.10 DEFINIÇÃO DOS MÓDULOS ESPECIAIS

Os módulos especiais de um programa em LIDIA são inicialização, voluntariamento, consulta e término. Desses, apenas a definição do módulo de consulta é obrigatória, como mostra a seguinte definição:

inicialização :- & | INITPROC ';' comandos END ';'

voluntariamento :- & | VOLUNTPROC ';' comandos END ';'

consulta :- CONSULT ';' comandos END ';'

término :- & | TERMINATE ';' comandos END ';'

Vide seção 2.3, para uma melhor compreensão dos módulos especiais e exemplo, no apêndice "A".

3.11 COMANDOS

Definimos comandos como uma sequência de um ou mais comandos da linguagem separados pelo ponto e vírgula, como mostra a seguinte definição:

comandos :- comando | comandos ';' comando

comando pode ser o comando nulo, ASK, VOLUNT, INFER, FEEDBACK, RECONSULT, MENU, WRITE, IF-THEN, IF-THEN-ELSE, WHILE, CALL, um comando de atribuição, uma chamada a procedimento externo e comando composto. Nas próximas seções, tem-se as definições desses comandos individualmente.

3.11.1 Comando Nulo

Este comando tem efeito apenas na definição da linguagem, visando manter a consistência e elegância da definição. Sintaxe:

comando :- &

3.11.2 Comando ASK

Este é o comando de interação que permite ao usuário instanciar uma evidência. Deve ser usado dentro da evidência a qual se quer instanciar. Seu efeito é esperar que o usuário entre com o valor da evidência. Este comando verifica se o tipo e o domínio da evidência pedida permite aceitar o dado fornecido. Caso negativo, uma mensagem de erro é apresentada ao usuário e a execução do comando é repetida, até que o usuário tenha entrado com o dado corretamente. Neste momento, se o tipo da evidência for YES_NO, o sistema ainda pedirá o grau de certeza da informação fornecida. Este valor numérico deve estar no intervalo real de -1.0 a 1.0. Caso não esteja neste intervalo, o comando ASK é mais uma vez executado. Após receber o dado correto, o sistema atualiza a rede de inferências, segundo o caminhamento progressivo, e o comando ASK é dado como executado. Sua sintaxe é bastante simples:

comando :- ASK

Exemplo de uso do comando ASK:

```
evidence x: real;
  write "qual o valor de x? ";
  ask
end;
```

3.11.3 Comando VOLUNT

Este comando ativa o módulo de voluntariamento. Após a execução deste módulo, o controle é passado ao comando seguinte ao VOLUNT. Este comando deve ser único dentro do módulo de consulta, pois define o ponto de salto do comando FEEDBACK. Sintaxe:

comando :- VOLUNT

3.11.4 Comando INFER

Este comando ativa a máquina de inferência. Os caminhamentos regressivo e progressivo são executados, escolhendo as evidências a serem perguntadas ao usuário ou colhidas através de algum instrumento medidor. A máquina de inferências mantém sempre, no vídeo, as hipóteses-raízes da rede, com as suas respectivas probabilidades (ou graus de certeza). O processamento é repetido até que o usuário se dê por satisfeito e pressione a tecla de término da inferência. Feito isto, o controle é passado ao comando seguinte ao

comando INFER. A sintaxe deste comando constitui apenas da palavra INFER, tal qual a definição:

comando :- INFER

3.11.5 Comando FEEDBACK

Este comando, usado no módulo de consulta, tem como função transferir o controle do programa para o comando VOLUNT. Antes, porém, o programa pergunta ao usuário se ele deseja introduzir novas informações ao sistema. Caso positivo, o salto é efetivado. Caso negativo, o controle é passado ao comando seguinte ao comando FEEDBACK. A sintaxe deste comando é simples:

comando :- FEEDBACK

3.11.6 Comando RECONSULT

Este comando tem como finalidade iniciar uma nova consulta. Sua sintaxe é a seguinte:

comando :- RECONSULT

O comando RECONSULT deve ser usado no módulo de consulta, provocando o seguinte efeito: o SE pergunta ao usuário se ele deseja efetuar uma nova consulta. Se o usuário responder afirmativamente, o comando reinicializa todos os objetos do programa, isto é, da rede de

inferências, e transfere o controle da execução ao início do módulo de consulta. Se o usuário responder negativamente (não quiser efetuar mais nenhuma consulta), nenhum salto é feito e o controle é passado ao comando seguinte ao RECONSULT.

3.11.7 Comando MENU

O comando MENU ativa o sistema de menus, que é responsável pelo instanciamento de uma ou mais evidências. Sua sintaxe é mostrada a seguir:

comando :- MENU *identificador*

onde *identificador* é um nome de grupo. Este comando permite que o usuário instancie uma, várias ou até todas as evidências subordinadas ao grupo especificado, escolhendo as opções através de menus.

3.11.8 Comando de atribuição

Podemos atribuir um valor a uma evidência. Esse valor pode ser proveniente de uma constante, de uma evidência, de uma chamada a uma função, ou ainda de uma expressão mais complexa, envolvendo operadores, etc. A sintaxe deste comando é mostrada a seguir:

comando :- *identificador* ':=' expressão

| SELF ':=' expressão

onde *identificador* representa o identificador da evidência que receberá o valor da expressão. Com o uso da palavra SELF, na segunda produção, fica implícito que a evidência a receber o valor será a que estiver sendo avaliada, durante o processo de inferências. Por isso, a segunda forma só deve ser usada no corpo de uma evidência ou em um procedimento que é ativado a partir de uma evidência. Caso esta condição não seja verificada, a execução do programa é cancelada, emitindo uma mensagem de erro no terminal.

O tipo da expressão deve ser compatível com o tipo da evidência. A figura 3.2 mostra as relações de compatibilidade de tipos, para transferência de dados:

destino	origem	conversão
integer	integer	
real	integer	sim
real	real	
string	string	
yes_no	yes_no	

figura 3.2 - compatibilidade de tipos de dados

Exemplo:

```
evidence a, b, c: integer;
  write "> "; ask
end;
evidence media: real;
  media := (a + b + c) / 3
end;
```

3.11.9 Comando WRITE

Este comando provoca a saída de uma informação na tela, a partir de uma lista de expressões. O seu formato é mostrado a seguir:

```
comando :- WRITE lista_exp
lista_exp :- expressão | lista_exp ',' expressão
```

lista_exp é uma lista de uma ou mais expressões, que podem ser de quaisquer tipos. Os valores resultantes das expressões são mostrados na tela, na medida em que elas vão sendo avaliadas, da esquerda para a direita. Exemplo de comando WRITE:

```
write "saidas: ",sqrt(x)," ",x+y," ",pot(y,2)
```

3.11.10 Chamadas a procedimento externo

Um comando pode ser também uma chamada a um procedimento definido externamente, em uma linguagem qualquer, de acordo com a seguinte sintaxe:

comando :- *identificador* '(' lista_args ')'
lista_args :- & | lista_exp

onde *identificador* é o nome do procedimento e deve ser especificado na declaração de procedimentos externos (veja seção 3.4). lista_args é uma lista de zero ou mais argumentos. A lista de argumentos deve ser compatível em número e tipo com a lista de parâmetros formais do procedimento. Se um parâmetro é especificado como sendo passado por referência, a expressão do argumento correspondente deve ser apenas um identificador de objeto.

3.11.11 Comando IF

Aqui, tem-se o comando IF em suas duas formas, chamadas IF-THEN e IF-THEN-ELSE, respectivamente:

comando :- IF expressão THEN comando
comando :- IF expressão THEN comando ELSE comando

onde expressão é uma expressão booleana (seção 3.12). Durante a execução deste comando, a expressão é avaliada. Caso o resultado seja verdadeiro, o comando associado (seguinte) ao THEN é executado. Caso o resultado seja falso e o comando IF for da forma IF-THEN, o controle é simplesmente passado ao comando seguinte ao IF-THEN. Caso o resultado seja falso e o comando IF for da forma IF-THEN-ELSE, será executado o comando associado (seguinte) ao ELSE e o controle continua com a execução do comando seguinte ao IF-THEN-ELSE. Observa-se que comando pode ser um comando composto (seção 3.11.14), permitindo assim que se tenha mais de um comando subordinado ao comando IF. Exemplo:

```
if a < b then (
  temp := a;
  a := b;
  b := temp
) else
    i := i + 1
```

3.11.12 Comando WHILE

O comando WHILE é um comando de repetição, que possui a seguinte sintaxe:

comando :- WHILE expressão DO comando

onde expressão é uma expressão booleana (seção 3.12). Ao iniciar a execução deste comando, avalia-se a expressão. Se o valor resultante for verdadeiro, o comando é executado. Sempre após a execução do comando, verifica-se o resultado da expressão e, sempre que o resultado for verdadeiro, o comando é mais uma vez executado. A execução do comando **WHILE** se dá por encerrada quando a expressão tiver o valor falso. Neste caso, o controle é passado ao comando imediatamente posterior ao comando WHILE. Exemplo:

```
evidence resposta: string ["NAO", "SIM"];
     resposta := "NAO";
     while resposta = "NAO" do (
       write "Diga sim: ";
       ask;
       resposta := upercase(resposta)
     )
end;
```

3.11.13 Chamada a procedimento interno, comando CALL

Os procedimentos definidos internamente, são chamados de forma diferente dos procedimentos externos. Para tanto, usa-se a palavra CALL. Sua sintaxe é mostrada a seguir:

comando :- CALL *identificador*

onde *identificador* é o identificador do procedimento a ser executado, que deve estar definido no programa fonte. A execução deste comando, dá-se através de um desvio do controle para o início do procedimento. Os comandos do procedimento são executados e, ao término da execução do mesmo, o controle é retornado ao comando seguinte ao comando CALL. Durante a execução do procedimento, a palavra SELF, nele contida, passa a representar o mesmo objeto-SELF do escopo da chamada. Exemplo:

```
proc auto_incremento;
  self := self + 1
end;

evidence a: integer;
  call auto_incremento
end;

evidence b: real;
  call auto_incremento
end;
```

No exemplo acima, tanto a evidência inteira A como a evidência real B chamam o procedimento interno auto_incremento. A palavra SELF, contida no procedimento, representa a evidência que chamou o mesmo.

3.11.14 O comando composto

Em LIDIA, pode-se agrupar mais de um comando, formando assim um único comando composto. Isto permite inserir uma lista de comandos onde é sintaticamente permitido apenas um comando (no WHILE, por exemplo). Para se agrupar comandos, coloca-os simplesmente entre chaves:

comando :- '{' comandos '}'

onde comandos está definido no início da seção 3.11. Após a execução dos comandos, o controle é passado ao comando seguinte ao comando composto.

3.12 EXPRESSÕES

Uma expressão pode conter operadores, identificadores de evidência, a palavra SELF, constantes e chamadas a funções externas. Dentro de uma expressão pode haver sub-expressões de níveis inferiores, de acordo com a seguinte sintaxe:

expressão :- sub_exp | expressão OR sub_exp

Observa-se que uma sub-expressão (sub_exp) também é uma expressão, de nível mais baixo. Ao se avaliar a expressão "OR", avalia-se primeiro a expressão do lado esquerdo do operador. Se esta resultar verdadeira, a expressão completa é dita verdadeira, sem que o lado direito seja avaliado. Se a expressão do lado esquerdo for falsa, o lado direito é avaliado e o resultado da expressão completa é igual ao resultado obtido da expressão do lado direito. A seguir, a definição formal de sub-expressão:

sub_exp :- exp_rel | sub_exp AND exp_rel

Uma sub-expressão é uma expressão que não contém o operador OR, isto é, é uma expressão de segundo nível. Esta quebra de expressões em níveis diferentes é útil para determinar sintaticamente as precedências entre os operadores (no caso, o AND tem precedência sobre o OR).

Sub-expressão pode ser uma expressão relacional (exp_rel) ou uma conjunção (AND) entre uma sub-expressão e uma expressão relacional. Na conjunção, inicialmente, o sistema avalia a sub-expressão. Se o valor resultante for *falso*, a sub-expressão é dita ser falsa. Se o valor resultante for verdadeiro, o resultado da sub-expressão completa é igual ao resultado da avaliação da expressão relacional.

```
exp_rel :- exp_desig |
           exp_desig '=' exp_desig |
           exp_desig '<>' exp_desig
exp_desig :- exp_aritm |
             exp_aritm '<' exp_aritm |
             exp_aritm '>' exp_aritm |
             exp_aritm '>=' exp_aritm |
             exp_aritm '<=' exp_aritm
```

Uma expressão relacional possui dois níveis de operadores: nível *igual-diferente* e nível de *desigualdade*. Isto permite indicar que os operadores igual ('=') e diferente ('<>') são avaliados após os operadores menor do que ('<'), maior do que ('>'), menor ou igual a ('<='), maior ou igual a ('>=').

```
exp_aritm :- termo |
             exp_aritm '+' termo |
             exp_aritm '-' termo
```

Uma expressão aritmética, exp_aritm, pode ser um termo, uma soma (operador '+') ou uma subtração (operador '-'). termo refere-se a uma expressão aritmética de nível inferior e, por isso, seus operadores possuem precedência sobre os de soma e subtração.

```
termo :- sinal_fator |
         termo '*' sinal_fator |
         termo '/' sinal_fator |
         termo DIV sinal_fator |
         termo MOD sinal_fator
sinal_fator :- fator | '-' fator
```

Um termo pode ser um fator com sinal (sinal_fator) ou uma operação de produto (operador '*'), divisão (operador '/'), divisão inteira (operador DIV) ou módulo entre um termo e um fator com sinal (operador MOD).

Como mostra a definição a seguir, fator é uma constante da linguagem (seção 3.2.5), a palavra SELF (seção 3.8), um identificador (seção 3.2.2), uma expressão qualquer entre parênteses, uma chamada a função externa ou uma operação unária.

```
fator :- constante |
         SELF |
         identificador |
         '(' expressão ')' |
         chamada_função_externa |
         op_unária
```

onde op_unária é uma das seguintes operações unárias:

op_unária :- '#' SELF |
'#' *identificador* |
'&' SELF |
'&' *identificador*

O operador '#' chama-se *texto*. Ele opera sobre uma hipótese ou evidência e retorna o texto associado à mesma, do tipo *string*. O operador de endereço, '&', também trabalha com hipótese ou evidência e sua função é retornar o endereço do operando, permitindo, entre outras coisas, passar objetos como parâmetros de procedimentos externos.

3.12.1 Chamadas a funções externas

Um subprograma externo que retorna um valor é dito *função externa*. Ele é utilizado dentro de uma expressão como um fator. Sua sintaxe é:

chamada_função_externa :- *identificador* '(' lista_args ')'

onde *identificador* é o identificador da função e lista_args é a sua lista de argumentos, definida na seção 3.11.10.

Toda função externa, antes de ser utilizada, deve ser declarada como externa (seção 3.4). Durante a sua chamada, o controle é passado à função e, após o término da sua

execução, um valor é retornado como fator e a expressão, na qual a chamada está inserida, continua sendo avaliada. O tipo da função externa deve ser compatível com o tipo do operando pedido pela expressão. Por exemplo: é um erro chamar uma função externa que retorna um caractere em uma expressão aritmética.

CAPÍTULO IV

ESPECIFICAÇÃO DO CÓDIGO OBJETO LIDIA

Os três primeiros capítulos descrevem toda a linguagem. Aqui, neste capítulo, tem-se a especificação do código objeto, como uma introdução à implementação da linguagem. Vale a pena observar que esta especificação é um pré-requisito para a construção do compilador, que é assunto abordado no capítulo 6.

Como linguagem para apresentação de estruturas de dados, escolheu-se C por ter sido a linguagem usada na implementação do compilador.

4.1 POOL DE LITERAIS

Todos os literais de um programa em LIDIA são armazenados no vetor *pool*. Cada elemento desse vetor corresponde a um literal (constante *string* ou identificador).

Descrição:

```
typedef  unsigned short word;
typedef char *string;
word  npool = NPOOL;
string pool[NPOOL] = { ...
```

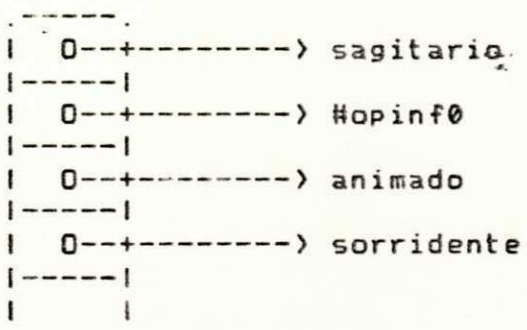


fig. 4.1 - pool de literais

Se um literal ocorre várias vezes no programa fonte apenas uma entrada é criada em pool para esse literal. Qualquer referência a um literal é feita através do índice da entrada correspondente ao mesmo. As ocorrências dos operadores de inferência também são incluídas no pool, pois é como se cada ocorrência desse operador fosse um novo identificador. Nesse caso, os nomes dos identificadores são "#opinf0", "#opinf1", etc.

4.2 TABELA DE OBJETOS

Essa tabela armazena todos os objetos do programa. Um objeto pode ser uma *hipótese*, uma *evidência*, um *grupo*, uma

função externa, uma ocorrência de um *operador de inferência* ou ainda um *procedimento*. Está implementada como um vetor de registros (*regobj*), como mostrado a seguir:

```
typedef  struct regobj {
                word nome, texto;
                char classe;
                word contin;
              } tipregobj;
word nobj = NOBJ;
tipregobj obj[NOBJ] = { ...
```

nome	texto	classe	contin

fig. 4.2 - registro de um objeto

O tipo word normalmente é usado para representar índices de tabelas. Para se indicar que uma variável desse tipo não está "apontando" para nenhuma tabela, simplesmente atribui-se a constante NIL (que equivale ao -1, um número formado de 16 bits ligados). Uma variável do tipo word também pode conter uma informação qualquer, inteira não negativa.

Os campos nome e texto representam o identificador e seu texto associado, respectivamente. São índices do *pool de*

literais. texto é especialmente usado na interação com o usuário.

classe é um caractere que especifica a classe do objeto, que pode ser

'e'	evidência
'h'	hipótese
'g'	grupo
'f'	função (externa)
'y'	operador de inferência
'p'	procedimento

Como o registro do tipo tipregobj representa objetos de classes diferentes, por uma questão de economia de memória, a parte específica desses objetos ficam em tabelas específicas de suas classes. Assim, temos tabelas de evidências, de hipóteses, de grupos, de funções e de operadores de inferência. O campo contin indica a entrada da tabela específica, referente à classe do objeto. No caso de procedimento, contin contém o índice da entrada, na tabela de comandos, correspondente ao primeiro comando do procedimento.

4.3 TABELA DE EVIDÊNCIAS

Esta tabela aglutina todas as evidências do programa fonte. Ela tem a seguinte descrição:

```
typedef char boolean;
union uval {
        int     i;
        char    c;
        float   r;
        boolean b;
        string  s;
    };

typedef unsigned char byte;
typedef struct {
          char       tipo;
          int        valmin, valmax;
          union uval valor;
          word       req, cmd, acao;
          float      gc, imp;
          word       pais;
        } tipregevid;

word       nevid = NEVID;
tipregevid evid[NEVID] = { ...
```

Antes de se descrever os componentes de tipregevid, convém ressaltar que a união uval é usada para armazenar o conteúdo de uma variável. Como existem vários tipos de dados, usa-se cada tipo existente como um componente de uval.

O componente tipo representa o tipo da evidência e pode ser:

'i'	integer
'r'	real
'b'	boolean (yes_no)
's'	string

valmin e valmax representam os valores mínimos e máximos que a evidência pode assumir. Esse intervalo de valores é chamado de *domínio*. Para os tipos inteiros e booleanos, essas variáveis contém os próprios valores mínimos e máximos, respectivamente. Para o domínio real, existe uma tabela de constantes reais cr (descrita na seção 4.5). Nesse caso, valmin aponta para o valor mínimo contido na tabela cr. Para o tipo *string*, porém, não há dois valores limitantes, há uma lista de *strings* que podem ser assumidos pela evidência. Nesse caso, existe uma tabela de domínios *strings* ds (descrita na seção 4.4). valmin e valmax apontam, respectivamente, para o primeiro e último *string*, elementos da lista contida em ds.

valor é o conteúdo da evidência. Esse valor sempre deve estar no domínio determinado por valmin e valmax. A constante booleana NO corresponde ao valor 0, a constante YES corresponde ao valor 1.

req, cmd, acao são índices da tabela de comandos (descrita na seção 4.8). req aponta para o requisito da evidência. cmd aponta para a lista de comandos, enquanto que acao aponta para a lista de ações. Qualquer um desses ponteiros pode assumir o valor NIL para indicar a ausência de expressões, já que elas são opcionais na linguagem.

O componente real gc armazena o *grau de certeza* da evidência. Quando a evidência é do tipo inteiro, real ou *string*, o gc sempre tem o valor máximo (+1) para indicar a certeza absoluta. Embora gc esteja na faixa [-1,+1], o grau de certeza será apresentado ao usuário como um número na faixa de [-100,100]. gc é inicializado com 2.0 para indicar que a evidência ainda não foi instanciada (respondida).

imp é a importância da evidência E, vide seção 2.1.2.

pais é o índice do primeiro elemento da lista de pais. Ele aponta para a tabela de *arcos*, descrita na seção 4.10. Isto se refere aos relacionamentos da evidência com todos os seus ancestrais imediatos, na rede de inferências.

4.4 TABELA DE DOMÍNIOS DE STRINGS (ds)

Essa tabela é usada apenas para armazenar os domínios de *strings* das evidências. Descrição:

```
word nds = NDS;
word ds[NDS] = { ...
```

Cada elemento de ds é um índice para o *pool* de literais.

Como exemplo, suponha que uma evidência do tipo *string* tenha o seguinte domínio: ["Amarelo", "Verde", "Vermelho"] e que estes *strings* tenham as entradas 1, 15 e 22, respectivamente, no *pool* de literais. Suponha também que os componentes da evidência valmin e valmax tenham valores 7 e 9 respectivamente. Então,

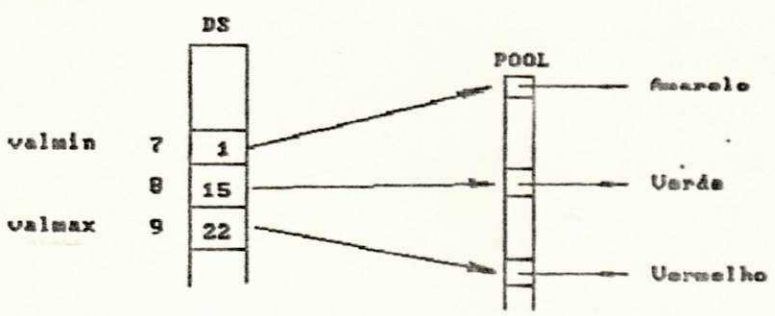


fig. 4.3 - Exemplo de tabela de domínios de string

Dessa forma, pode-se percorrer sequencialmente parte do vetor ds para mostrar na tela, em forma de menu, os valores possíveis de uma evidência.

4.5 TABELA DE CONSTANTES REAIS

É semelhante à tabela anterior, porém armazena números reais. Descrição:

```
word  ncr = NCR;
float cr[NCR] = {...
```

Cada elemento de cr é uma constante real. Este vetor também é usado para armazenar os valores limitantes (esquerdo e direito) das evidências do tipo real.

4.6 TABELA DE CONSTANTES

Esta tabela permite armazenar todas as constantes que aperecem no programa fonte. Sua descrição é a seguinte:

```
typedef struct {
          char  tipo; /* i,c,r,b,s */
          int   valor;
        } regconst;
word maxconst = MAXCONST;
regconst const[MAXCONST] { ...
```

1	2	3	4	5	6
tipo	tipo	tipo	tipo	tipo	tipo
valor	valor	valor	valor	valor	valor

fig. 4.4 - Tabela de Constantes

Cada elemento da tabela const pode ser inteiro, caractere, real, booleano ou *string*. O campo valor é usado como ponteiro no caso dos tipos real e *string*. Caso o tipo da constante seja real, o componente valor contém o índice para a tabela de constantes reais (seção 4.5). Caso o tipo da constante seja *string*, o componente valor contém o índice para o *pool* de literais. Para os tipos inteiro, caractere e booleano, o valor da constante é armazenado diretamente no componente valor.

4.7 TABELA DE HIPóTESES

Essa tabela armazena todas as hipóteses do programa fonte. Sua descrição é:

```
typedef struct {
            float   gc, chance0, chance;
            float   lim;
            word    req, filhos, pais, acao;
            float   imp, piso, teto;
            char    estado;
          } tipreghip;
word        nhip = NHIP;
tipreghip hip[NHIP] = {...
```

O componente gc armazena o *grau de certeza* da hipótese. Tem a faixa real de -1.0 a +1.0. O valor 0.0 (zero) indica a ignorância total sobre a possibilidade da ocorrência ou não da hipótese. No caso das hipóteses, gc é inicializado com 0 (zero).

chance0 e chance são as *chances iniciais e atuais*, respectivamente. Estão na faixa de 0 a MAXREAL (o maior número real positivo), que representa o valor "mais infinito". lim é o *limiar de aceitação* e está na mesma faixa. req aponta para o requisito, contido na tabela de comandos ou contém o valor NIL. filhos aponta para a lista dos arcos filhos da hipótese (tabela de arcos). pais aponta para a lista de arcos pais da hipótese (mesma tabela) ou é NIL, se a hipótese for uma raíz. acao simplesmente aponta para uma entrada na tabela de comandos, onde estará o

fator é o fator atual, fruto da multiplicação de todos os fatores dos arcos-filhos do operador. Esse resultado será usado no caminhamento progressivo. filhos é o apontador que referencia a lista dos arcos-filhos (tabela de arcos) do operador de inferência. Enquanto que pais desempenha o mesmo papel para a lista dos arcos-pais (mesma tabela). result é o valor resultante da expressão de inferência ('v', 'f' ou ' ' para não instanciado). result é usado para indicar se a máquina de inferências deve multiplicar ('v' -> o fator sim; 'f' -> o fator não) ou não (' ') pelo fator de cada nó-pai.

4.9 TABELA DE COMANDOS

Nesta tabela, estão armazenadas as expressões, que são formadas de *comandos*, *operadores* (exceto IAND e IOR, que são convertidos em INFOP, no nível léxico), *constantes*, *referências a hipóteses e evidências*, *chamadas a funções* e *referências ao símbolo* SELF. Uma expressão é representada na tabela através de uma sequência de instruções da máquina de execução LIDIA (*mexel*). A estrutura da tabela é a seguinte:

```
word    ncmd = NEXP;
byte    cmd[NEXP] = { ...
```

4.10 TABELA DE ARCOS

A tabela de arcos tem como objetivo manter as relações entre os nós da rede de inferências. Cada entrada nesta tabela corresponde a uma relação entre um *nó-pai* e um *nó-filho*, com os atributos inerentes a essa relação. Uma entrada desta tabela pertence, ao mesmo tempo, a duas listas. Por exemplo, a relação A é pai de B implica em inserir uma entrada na tabela de arcos. No caso, a entrada pertencerá tanto à lista dos filhos de A como à lista dos pais de B.

Descrição:

```
typedef struct {
            float fnao, fsim, fator;
            float imp;
            word  filho, pai, proxfilho, proxpai;
            char  tpfilho;
          } tipregarco;
word narco = NARCO;
tipregarco arco[NARCO] = { ...
```

fnao e fsim são os fatores do não e do sim, respectivamente. fator é o multiplicador atual, um fator a ser utilizado durante o processo de inferência, e tem valor inicial 1.

imp é a importância do arco para o nó pai. É usado durante o caminhamento regressivo, no processo de inferências. O cálculo de imp é apresentado na seção 2.1.2.

NL, o número de evidências livres mostrado na equação 2.6 (seção 2.1.2), não é um campo que faz parte da estrutura. Ele é calculado, através de um caminhamento na sub-árvore, toda vez que for necessário se calcular a importância de um arco.

filho aponta para o nó filho e pai aponta para o nó pai. No caso do filho, este pode estar na tabela cmd (se for uma expressão booleana, neste caso tpfilho = 'x') ou na obj (se for uma hipótese ou evidência, neste caso tpfilho = 'o').

proxfilho aponta, na tabela de arcos, para o próximo elemento da lista de filhos do objeto referenciado por pai. O inverso ocorre com proxpai que aponta, para a mesma tabela, para o próximo elemento da lista de pais do objeto referenciado por filho.

4.11 A FUNÇÃO callfunc

Esta função é usada para as chamadas existentes no programa fonte, cujas funções estão definidas externamente, ou a chamadas a funções pré-definidas.

Descrição:

```
callfunc(n)
  int n;
{
  switch (n) {
     ...
  }
}
```

Esta função recebe um parâmetro e chama uma outra função que é selecionada a partir do valor deste parâmetro. Na chamada à função callfunc, deve-se usar como argumento o índice da entrada da tabela obj correspondente à função que se quer executar.

CAPÍTULO V

A MÁQUINA DE EXECUÇÃO LIDIA

Este texto descreve todo o funcionamento da Máquina de Execução LIDIA (*Mexel*). A sua função é interpretar e executar os comandos contidos nos procedimentos, nas ações ou nas listas de comandos.

5.1 ARQUITETURA DA MEXEL

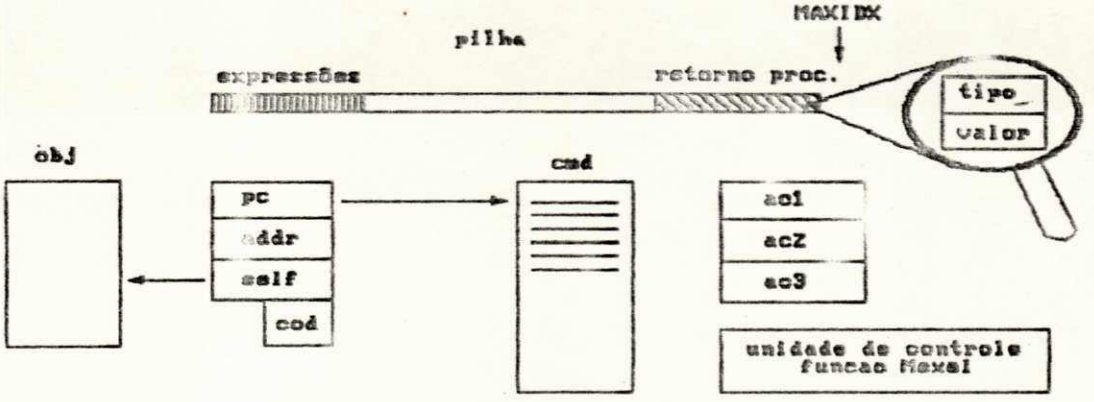


fig 5.1 - Arquitetura da Mexel

Como mostra a figura 5.1, Mexel é formada por três acumuladores, uma pilha de expressões, uma pilha de retorno de procedimentos, uma área (tabela cmd) onde residem as instruções a serem executadas pela mexel, um contador de

instruções (pc), um registrador de endereços (addr) um registrador de código de instrução (cod), o registrador SELF, que contém o endereço do atual objeto em avaliação na tabela de objetos, além das tabelas descritas no capítulo 4.

O ciclo de busca [ZUFF 78] de uma instrução se processa da seguinte forma: inicialmente obtém-se o byte de código da instrução, colocando-o no registrador cod; incrementa-se 1 ao contador de instruções. Verifica-se através de uma tabela se há operando (e quantos bytes este requer) para o código recém obtido. Caso positivo, obtém-se o operando colocando o seu valor no registrador addr e incrementando o pc de maneira a apontar para o início da próxima instrução.

O ciclo de execução de uma instrução da mexel é feito através da escolha da função que executa a instrução, a partir do código da mesma, contido no registrador cod. Mexel é uma função que está implementada da seguinte maneira:

```
typedef char boolean;
typedef char *string;
typedef unsigned short word;
typedef union {  short   i;
                 word w;
                 float   r;
                 boolean b;
                 char    c;   string    s;
              } uval;
```

```
typedef struct {
          char      tipo; /* i,c,r,b,s,w */
          uval      valor;
        } regpilha;
mexel(ende)
  int ende;
{
  word pc, addr, self;
  regpilha ac1, ac2, ac3;
  regpilha pilha[MAXIDX];
  byte cod;
  ...
}
```

onde ende representa o endereço da instrução, na tabela cmd, de onde será iniciada a execução. Mexel executa instrução a instrução, até encontrar a instrução RET que lhe ordena encerrar o seu trabalho. Daí, a função Mexel retorna o controle à função que a chamou.

As variáveis ac1, ac2 e ac3 representam os acumuladores da máquina. Um acumulador tem o seu tipo e seu conteúdo. Sempre que um elemento é retirado da pilha é colocado em ac1 ou ac2. O acumulador ac3 guarda os resultados das operações da máquina e é usado quando este valor for empilhado.

A *tabela de instruções* (cmd) é um espaço formado por uma sequência de bytes, de onde são alocadas sequencialmente

as instruções. Cada instrução pode ocupar 1, 2 ou 3 bytes consecutivos.

A pilha é um espaço definido, localmente, da seguinte forma:

```
regpilha  pilha[MAXIDX];
```

O vetor *pilha*, na verdade, armazena duas pilhas, uma em cada extremidade, e são *inicializadas* da seguinte forma:

```
word      tamanho = MAXIDX,
          topo = 0,  ret = MAXIDX;
```

Onde MAXIDX é o maior índice do vetor *pilha*. A *pilha de expressões*, que vai do elemento de índice 1 ao elemento de índice *topo*, armazena os operandos das expressões aritméticas, lógicas e resultados de funções. A outra pilha, formada pelos elementos entre *ret* e MAXIDX-1, armazenam apenas valores inteiros (short) que representam os endereços (índices) de retorno dos procedimentos chamados.

O termo *pilha* será usado para representar a *pilha de expressões*. A outra pilha sempre será referenciada pelo termo "*pilha de retorno*".

5.2 CÓDIGOS DE OPERAÇÃO

Cada instrução é formada por até dois campos: o código de operação e, opcionalmente, o operando. A seguir, estão listados os códigos de operações das instruções existentes na *máquina de execução*:

00	NOP	18	NE
01	ASK	19	LT
02	VOLUNT	20	GT
03	INFER	21	GE
04	FEEDBACK	22	LE
05	RECONSULT	23	IDIV
06	CALLFUNC	24	MOD
07	WRITE	25	MINUS
08	MENU	26	ADD
09	CALLPROC	27	SUB
10	JPZ	28	MUL
11	JPNZ	29	DIV
12	JUMP	30	FCALL
13	RET	31	CONST
14	STEV	32	OBJ
15	OR	33	SELF
16	AND	34	TEXT
17	EQ	35	ADDR

5.3 OPERANDOS

Uma instrução da linguagem objeto, pode ter 0, 1 ou 2 bytes reservados ao operando. A maioria das instruções não possui operando, portanto tem apenas um byte. Algumas instruções possuem 2 bytes para o operando, como é o caso dos endereços. A instrução WRITE possui apenas um byte de operando, que representa o número de parâmetros que serão usados pela instrução.

5.4 REPERTÓRIO DE INSTRUÇÕES

A seguir será descrito em detalhe todo o funcionamento de cada instrução da Mexel.

5.4.1 Instrução NOP (1 byte)

Esta instrução não faz efeito durante a sua execução. É usada apenas para facilitar a estruturação de programas gerados pelo compilador. Exemplo:

```
...
if idade > 20 then {
     tempo(self,false);
     tempo(idade,true)
};
tempo(nfilhos,false);
...
```

Os comandos acima podem ser compilados, gerando o seguinte código objeto:

```
...
42:  OBJ,      0,   40,      ; empilha idade
45:  CONST,    0,    4,      ; empilha constante 20
48:  GT,                     ; idade > 20?
49:  JZP,      0,   68,      ; não, salta p/ 68
```

```
52:  SELF,                      ; empilha self
53:  CONST,     0,     5,       ; empilha constante false
56:  CALLFUNC,  0,     0,       ; chama a função tempo
59:  OBJ,       0,    40,       ; empilha idade
62:  CONST      0,     6,       ; empilha constante true
65:  CALLFUNC,  0,     0,       ; chama função tempo
68:  NOP,                       ; end-if
69:  OBJ,       0,    39,       ; empilha objeto nfilhos
72:  CONST,     0,     5,       ; empilha constante false
75:  CALLFUNC,  0,     0,       ; chama a função tempo
...
```

5.4.2 Instrução ASK (1 byte)

Formato: ASK

Descrição:

Verifica se a evidência atualmente em avaliação já foi ou não instanciada. Caso positivo, esta instrução não faz nada. Caso a variável não esteja instanciada, esta instrução providencia uma entrada de dados pelo teclado. Esta entrada de dados pode ser através de um menu (caso a evidência seja do tipo *string* e o seu domínio esteja explícito) ou uma pergunta. Neste caso, o valor fornecido deve pertencer ao domínio. Caso não pertença, um aviso deve ser fornecido ao

usuário, deve-se mostrar o domínio e repetir o pedido até que o valor fornecido pertença ao domínio estabelecido.

5.4.3 Instrução VOLUNT (1 byte)

Formato: VOLUNT

Descrição:

Empilha o endereço da próxima instrução, na pilha de retorno, e chama o procedimento padrão VOLUNTPROC, que se encarregará de fazer o voluntariamento.

5.4.4 Instrução INFER (1 byte)

Formato: INFER

Descrição:

Empilha o endereço da próxima instrução, na pilha de retorno, e chama o procedimento padrão, a máquina de inferências LIDIA.

5.4.5 Instrução FEEDBACK (1 byte)

Formato: FEEDBACK

Descrição:

Esta instrução pergunta ao usuário se ele quer aumentar ou alterar alguma informação na base de

conhecimentos. Caso a resposta seja negativa, o controle é passado para a instrução seguinte. Caso a resposta seja positiva, o controle é passado para a instrução VOLUNT.

5.4.6 Instrução RECONSULT (1 byte)

Formato: RECONSULT

Descrição:

Esta instrução pergunta ao usuário se ele quer fazer uma nova consulta. Caso positivo, a rede é *inicializada* e o controle é passado para o início da consulta. Caso negativo, nada é feito e o controle é passado à instrução seguinte.

5.4.7 Instrução CALLFUNC (3 bytes)

Formato: CALLFUNC ende

onde *ende* é o endereço (índice) da função, na tabela obj.

Descrição:

Esta instrução chama a função *callfunc* com o parâmetro *ende*. Depois o controle é passado à instrução seguinte. Nenhuma operação é feita na pilha.

5.4.8 Instrução WRITE (2 bytes)

Formato: WRITE num

onde *num* é um byte que representa o número de parâmetros da instrução WRITE. Esses parâmetros estão na pilha, os *num* últimos elementos incluídos na mesma.

Descrição:

Esta instrução mostra na console os últimos *num* valores da pilha, na ordem em que eles foram inseridos e desempilha esses *num* valores que estão mais ao topo da mesma.

5.4.9 Instrução MENU (3 bytes)

Formato: MENU ende

onde *ende* representa o endereço (índice) do grupo, contido na tabela obj.

Descrição:

Esta instrução mostra na tela um ou mais menus hierárquicos, a partir da estrutura declarada dos grupos de evidências, em que *ende* é o grupo raiz. Após montar o menu, o usuário toma o controle e escolhe uma ou mais evidências e atribui valores a elas. Para cada valor introduzido, a rede é ré-calculada (pisos, graus de certezas, chances e tetos das hipóteses e os fatores dos arcos). Para indicar que o usuário já terminou o seu voluntariamento, ele pressiona a [End].

5.4.10 Instrução CALLPROC (3 bytes)

Formato: CALLPROC ende

onde *ende* é o endereço (índice) da tabela obj, onde está o procedimento a ser executado.

Descrição:

Esta instrução empilha, na pilha de retorno, o endereço da próxima instrução, consulta a tabela obj, obtém o endereço do início do procedimento (na tabela de instruções) e passa o controle para esse endereço, através da atribuição pc = ende.

5.4.11 Instrução JPZ (Jump if Zero - 3 bytes)

Formato: JPZ ende

onde *ende* é o endereço (índice) da tabela de instruções.

Descrição:

Esta instrução retira o elemento do topo da pilha e verifica se o seu conteúdo é zero. Caso positivo, o controle é passado para a instrução do endereço *ende*. Caso contrário, o controle é passado para a instrução seguinte.

5.4.12 Instrução JPNZ (Jump if Not Zero - 3 bytes)

Formato: JPNZ ende

onde *ende* é o endereço (índice) da tabela de instruções.

Descrição:

Esta instrução retira o elemento do topo da pilha e verifica se o seu conteúdo é diferente de zero. Caso positivo, o controle é passado para a instrução do endereço *ende*. Caso contrário, o controle é passado para a instrução seguinte.

5.4.13 Instrução JUMP (3 bytes)

Formato: JUMP ende

onde *ende* é o endereço (índice) da tabela de instruções.

Descrição:

Esta instrução desvia, incondicionalmente, para a instrução de endereço *ende*.

5.4.14 Instrução RET (Return - 1 byte)

Formato: RET

Descrição:

Este procedimento verifica se a pilha está vazia. Caso positivo, a máquina de execução pára de executar a instrução e o controle é passado para a função que chamou a *mexel*. Caso negativo, desempilha o endereço contido no topo da pilha e passa o controle para este endereço.

5.4.15 Instrução STEV (Store Evidence - 3 bytes)

Formato: STEV ende

onde *ende* é o endereço da evidência, na tabela de objetos ou o valor NULL.

Descrição:

Esta instrução retira o elemento do topo da pilha e armazena no campo *valor* da evidência referenciada por *ende*. Se *ende* for o valor NULL, a instrução armazena o elemento no campo *valor* da evidência referenciada pelo registrador SELF.

5.4.16 Instrução OR (1 byte)

Formato: OR

Descrição:

Esta instrução retira os dois últimos elementos da pilha, efetua a operação lógica "ou" e coloca o resultado da operação na pilha.

5.4.17 Instrução AND (1 byte)

Formato: AND

Descrição:

Esta instrução retira os dois últimos elementos da pilha, efetua a operação lógica "e" e coloca o resultado da operação na pilha.

5.4.18 Instrução EQ (Equal to - 1 byte)

Formato: EQ

Descrição:

Esta instrução retira os dois últimos elementos da pilha, compara-os e, se os valores forem igüais, empilha o valor 1 (TRUE). Caso contrário, empilha o valor 0 (FALSE).

5.4.19 Instrução NE (Not Equal to - 1 byte)

Formato: NE

Descrição:

Esta instrução retira os dois últimos elementos da pilha e faz uma comparação entre eles. Se tiverem valores diferentes, empilha o 1 (TRUE), caso contrário empilha o valor 0 (FALSE).

5.4.20 Instrução LT (Less Than - 1 byte)

Formato: LT

Descrição:

Esta instrução desempilha dois valores e compara-os. Caso, o último elemento retirado (topo-1) seja menor que o primeiro retirado (topo), empilha o valor 1 (TRUE). Caso contrário, empilha o valor 0 (FALSE).

5.4.21 Instrução GT (Greater Than - 1 byte)

Formato: GT

Descrição:

Esta instrução desempilha dois valores e compara-os. Caso, o último elemento retirado (topo-1) seja maior que o primeiro retirado (topo), empilha o valor 1 (TRUE). Caso contrário, empilha o valor 0 (FALSE).

5.4.22 Instrução GE (Greater than or Equal to - 1 byte)

Formato: GE

Descrição:

Esta instrução desempilha dois valores e compara-os. Caso, o último elemento retirado (topo-1, antes da retirada) seja maior ou igual que o primeiro retirado (topo, antes da retirada), empilha o valor 1 (TRUE). Caso contrário, empilha o valor 0 (FALSE).

5.4.23 Instrução LE (Less than or Equal to - 1 byte)

Formato: LE

Descrição:

Esta instrução desempilha dois valores e compara-os. Caso, o último elemento retirado (topo-1, antes da retirada) seja menor ou igual que o primeiro retirado (topo, antes da retirada), empilha o valor 1 (TRUE). Caso contrário, empilha o valor 0 (FALSE).

5.4.24 Instrução IDIV (Integer Division - 1 byte)

Formato: IDIV

Descrição:

Esta instrução desempilha dois elementos da pilha e efetua a divisão inteira "pilha[topo-1]/pilha[topo]". O quociente é colocado na pilha.

5.4.25 Instrução MOD (1 byte)

Formato: MOD

Descrição:

Esta instrução desempilha dois elementos da pilha e efetua a divisão inteira do penúltimo elemento retirado pelo último. O resto da divisão é armazenado nesta pilha.

5.4.26 Instrução MINUS (1 byte)

Formato: MINUS

Descrição:

Esta instrução desempilha o elemento do topo, faz a troca de sinal do seu valor e repõe o resultado da operação na pilha.

5.4.27 Instrução ADD (1 byte)

Formato: ADD

Descrição:

Esta instrução desempilha dois elementos da pilha, efetua a soma entre eles e empilha o resultado.

5.4.28 Instrução SUB (1 byte)

Formato: SUB

Descrição:

Esta instrução desempilha dois elementos da pilha, subtrai o primeiro elemento retirado do último e coloca o resultado de volta na pilha.

5.4.29 Instrução MUL (1 byte)

Formato: MUL

Descrição:

Esta instrução desempilha dois elementos da pilha, e empilha o produto desses dois valores.

5.4.30 Instrução DIV (1 byte)

Formato: DIV

Descrição:

Esta instrução desempilha dois elementos da pilha, faz a divisão real entre o último elemento retirado e o primeiro e coloca o resultado da operação na pilha.

5.4.31 Instrução FCALL (2 bytes)

Formato: FCALL ende

onde *ende* é o endereço (índice) da função, na tabela obj.

Descrição:

Esta instrução chama a função *callfunc* com o parâmetro *ende*. O resultado da função é armazenado na pilha. Após isto, o controle é passado à instrução seguinte.

5.4.32 Instrução CONST (3 bytes)

Formato: CONST ende

onde *ende* é o endereço da entrada na tabela de constantes.

Descrição:

Esta instrução empilha o valor da constante endereçada por *ende*.

5.4.33 Instrução OBJ (3 bytes)

Formato: OBJ ende

onde *ende* é o endereço do objeto, na tabela de objetos (obj).

Descrição:

Esta instrução empilha o valor (e o tipo) da evidência endereçada por *ende*.

5.4.34 Instrução SELF (1 byte)

Formato: SELF

Descrição:

Esta instrução empilha o conteúdo do objeto endereçado pelo registrador SELF e o seu respectivo tipo.

5.4.35 Instrução TEXT (1 byte)

Formato: TEXT

Descrição:

Esta instrução retira da pilha o índice do identificador e coloca na mesma o índice da constante *string* obtido a partir do campo *texto* da respectiva entrada na tabela de identificadores. Para se obter o endereço do *string*, obtem-se o apontador *texto* e, em seguida, acessa-se o *pool* de literais.

5.4.36 Instrução ADDR (1 byte)

Formato: ADDR

Descrição:

Esta instrução retira da pilha o índice de um identificador (tabela de identificadores) e empilha o endereço da entrada referente ao mesmo.

CAPÍTULO VI

IMPLEMENTAÇÃO DO COMPILADOR

A descrição do projeto e da implementação do compilador LIDIA é matéria deste capítulo. Nele, encontram-se as técnicas utilizadas na implementação dos analisadores léxico e sintático, no manuseio das tabelas de símbolos, nas ações semânticas, tratamento de erros e na geração do código objeto.

6.1 INTRODUÇÃO

O compilador LIDIA é um programa que recebe como entrada um programa fonte nesta mesma linguagem e produz o *programa-objeto* correspondente, um módulo que contém a base de conhecimento, comandos e chamadas a funções externas. Esse programa-objeto está escrito em C. O programador compila o programa-objeto e liga-o com o Sistema de Execução e, possivelmente, com os seus módulos de funções externas já compilados. Deste processo, resulta um programa executável independente, como mostra a figura 6.1.

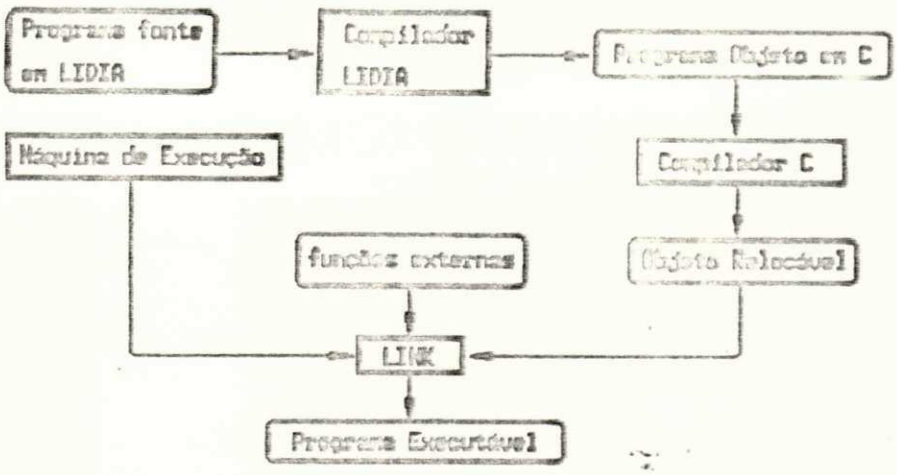


fig. 6.1 - Passos na produção do programa executável

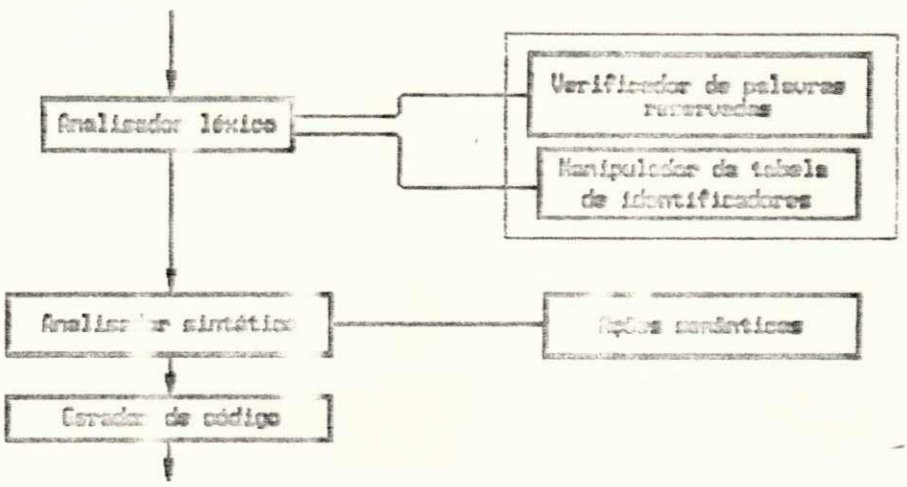


fig. 6.2 - partes do compilador

Como mostra a figura 6.2, as partes do compilador são o analisador léxico, o manipulador da tabela de identificadores, o analisador sintático, as ações semânticas e o gerador de código.

O compilador foi escrito utilizando YACC. Para se fazer a recuperação de erros sintáticos com esta ferramenta, adiciona-se diversas regras de produção, uma para cada erro,

o que é uma tarefa muito simples. Tendo em vista que a linguagem ainda está em caráter experimental e, portanto, algumas modificações lhe serão necessárias (vide capítulo 7 - Conclusões e sugestões), optou-se por não incluir ainda a recuperação de erros sintáticos neste trabalho. Apenas mensagens de erros são apresentadas ao usuário.

Sempre que necessário, o analisador sintático pede um símbolo ao analisador léxico. Este varre o programa fonte, identifica o símbolo lido e retorna-o para o analisador sintático num formato apropriado.

Eventualmente, para identificar um símbolo, o analisador léxico interage com as funções de verificação de palavras reservadas e, se for o caso, com o manipulador da tabela de identificadores a fim de incluir um símbolo.

Após analisar uma sentença, o analisador sintático executa ações semânticas. Estas ações incluem inclusão de informações em estruturas de dados.

Após analisar todo o programa-fonte e gerar internamente a estrutura do programa, o compilador chama, então, o gerador do código objeto, se nenhum erro foi encontrado.

6.2 ANALISADOR LÉXICO

O analisado léxico LIDIA é uma função que lê os caracteres do programa fonte e, ao encontrar uma sequência de caracteres que formam um padrão léxico da linguagem, retorna o símbolo num formato mais apropriado de ser manipulado pelo analisador sintático. Opcionalmente, o analisador léxico retorna o apontador de uma entrada da tabela de símbolos ou o valor de uma constante.

Na implementação do analisador léxico, foi utilizada a ferramenta LEX [AHO 87]. Para utilizar esta ferramenta, o programador simplesmente especifica as expressões regulares que formam os padrões léxicos dos símbolos da linguagem. Para alguns símbolos, porém, foi necessária a inserção de trechos de código em C.

Para o LEX, não foi especificado o comentário como um símbolo, mas apenas o seu início. Assim, foi especificado, como símbolo, o "abre-comentário" e a função de tratamento de comentário se encarrega de ler os outros caracteres, contar linhas (no caso do comentário ocupar mais de uma linha), e verificar a condição de fim do comentário. Um comentário sem fim ocasiona uma mensagem de erro, após o final do arquivo-fonte. No caso de comentário, o analisador léxico o ignora: um outro símbolo é lido e retornado, na mesma chamada.

Ao identificar uma palavra, o analisador léxico verifica se é uma palavra reservada, chamando a função de manipulação da tabela de palavras reservadas. Se a palavra for reservada, o analisador verifica se trata-se do operador de inferência (INFOP). Caso positivo, inclui na tabela de símbolos um nome formado pela palavra "#opinf" seguida de um número sequencial. Este nome indica a ocorrência do operador, já que cada ocorrência possui comportamento específico e corresponde a um objeto, na linguagem.

Se a palavra lida pelo analisador não for reservada, ela é entregue à função de manuseio da tabela de símbolos que, por sua vez, retorna o apontador da entrada deste símbolo na tabela. Neste caso, o símbolo é retornado pelo analisador léxico como um identificador.

O analisador léxico possui 105 linhas de especificação lex, que produz o mesmo analisador em C com 680 linhas.

6.3 TABELAS DE SÍMBOLOS

O compilador possui quatro tabela de símbolos, sendo que três delas armazenam palavras reservadas e a quarta armazena os identificadores.

6.3.1 Tabela de Palavras Reservadas

São três tabelas do mesmo formato que armazenam palavras reservadas do Português, Francês e Inglês. Como essas tabelas são fixas, puderam ser organizadas de forma eficiente, usando uma técnica similar a *hashing*.

Cada Tabela de Palavras Reservadas (TPR) está dividida em diversos pequenos trechos como mostra a figura 6.3. As palavras reservadas são agrupadas pelo seu tamanho e cada grupo de palavras é alocado dentro de um trecho. Assim, a função verificadora de palavras reservadas recebe, como parâmetro de entrada, a palavra e o seu tamanho. Esta função consulta uma tabela de índices, a partir do tamanho recebido e obtém diretamente o endereço do trecho correspondente ao tamanho da palavra. O acesso dentro do trecho é sequencial.

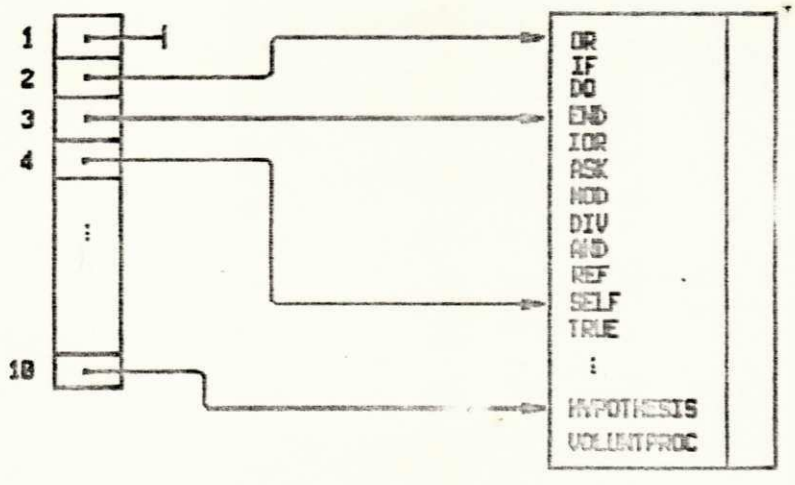


fig. 6.3 - Organização da TPR

As palavras, dentro de cada trecho, devem estar organizadas por ordem decrescente de frequência em que as ocorrem nos programas-fonte. Neste trabalho, não houve esta preocupação. Porém, com a utilização efetiva da linguagem, poder-se-á fazer uma análise das frequências das palavras e assim fazer esta otimização.

A função verificadora de palavra reservada, faz o acesso à TPR e caso a palavra recebida esteja na tabela, retorna o código da palavra no formato inteiro. No caso da palavra não ser encontrada, é sinal de que ela não é reservada e por isso, retorna o código FALSE.

Como existem três idiomas e três tabelas, a função que identifica palavras reservadas possui uma variavel seletora que indica o idioma. Inicialmente, esta variavel tem o valor ANY para indicar que a função ainda não "sabe" qual é o idioma. Assim que a primeira palavra reservada é recebida, sabe-se qual o idioma. A partir daí, apenas a tabela referente ao idioma selecionado é consultada.

4.3.2 Pool de Literais

Antes de se entender a estrutura da tabela de identificadores, é importante conhecer o *pool* de literais. Trata-se de uma arvore binária não balanceada, de alocação dinâmica onde estão armazenadas todas as constantes do tipo

string e os nomes dos identificadores encontrados no programa-fonte e dos pré-definidos. A figura 6.4 apresenta a estrutura do pool de literais. O campo idx armazena o endereço da entrada do literal a ser alocado no vetor *pool* do código objeto (seção 4.1). O campo st simplesmente aponta para o literal, propriamente dito.

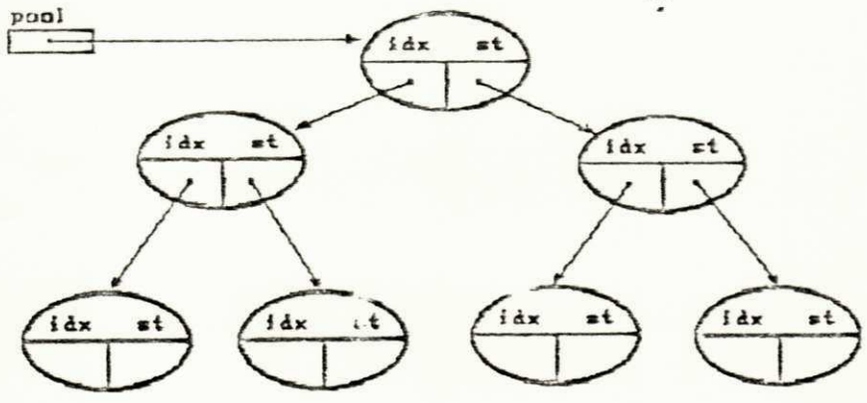


fig. 6.4 – pool de literais

6.3.3 Tabela de Identificadores

Esta é a tabela de símbolos, propriamente dita. Esta tabela é uma árvore de alocação dinâmica endereçada pela variável id, como mostra a figura 6.5. Cada nodo da árvore é um registro do tipo reg_id, detalhado na figura 6.6. Ao mesmo tempo em que esta tabela é uma árvore, é também uma lista linear, tendo como elo o campo proxid. A árvore é organizada pelo nome do identificador enquanto que a lista é classificada pela ordem de entrada dos símbolos na tabela.

Esta lista tem como objetivo, otimizar a geração do código objeto. Assim, cada identificador tem o seu endereço idx a ser usado no código objeto.

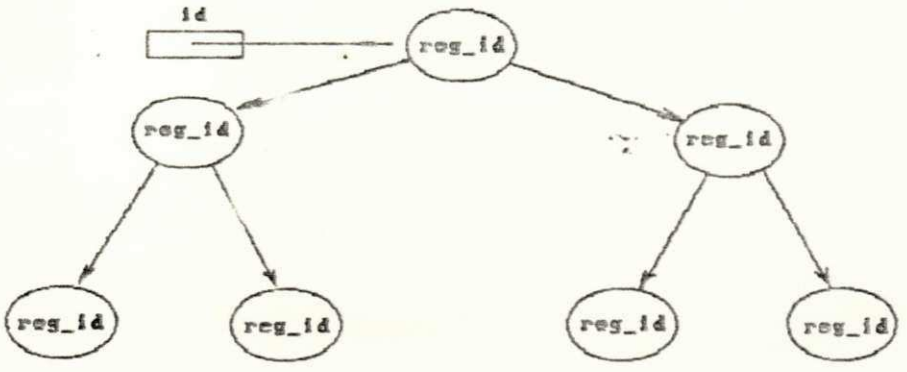


fig. 6.5 - tabela de identificadores

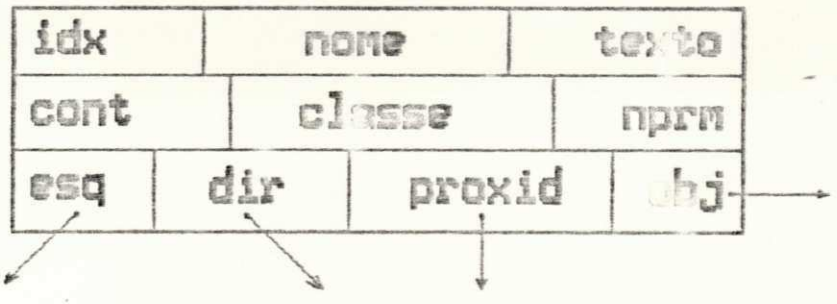


fig. 6.6 - nodo da tabela de identificadores

No reg_id o campo cont é usado para contar o número de ocorrências de um identificador. Ao incluir um símbolo, este campo tem valor 1. Toda vez que este símbolo for consultado, o campo é incrementado. Isto serve para o analisador

semântico acusar erros como "Identificador não declarado" ou "Identificador declarado, mas não usado".

Os campos nome e texto são apontadores para entradas do pool de literais. Esses apontadores representam o nome e o texto do identificador respectivo. Se o identificador não for um objeto que tenha texto declarado, texto aponta para a mesma entrada que nome, no pool de literais. texto também é usado quando se trata de funções pré-definidas. Neste caso, nome armazena o nome da função e texto, o arquivo-biblioteca onde a mesma é encontrada.

o campo classe é um caractere que indica a classe do identificador, segundo a tabela 6.1:

conteúdo	classe de identificador
h	hipótese
e	evidência
g	grupo
f	função externa
y	operador de inferência
p	procedimento interno
x	expressão

tabela 6.1 - classes de identificadores

As classes "operador de inferência" e "expressão" não existem para o programador LIDIA. São usadas para controle interno sempre que se encontra uma ocorrência de operador de

inferência ou uma expressão booleana dentro de uma lista de filhos.

O campo nprm é usado apenas quando a classe é uma função ou um grupo. No primeiro caso, nprm armazena o número de parâmetros da função. No segundo, armazena o número de identificadores (de grupo ou evidência) subordinados ao grupo.

O campo obj é um apontador para um registro específico da classe e que completa as informações do identificador. Os formatos desses registros estão descritos na seção 6.5, que trata das ações semânticas.

Antes de iniciar a tradução, o compilador LIDIA insere na tabela de símbolos os identificadores pré-definidos. Dessa forma, o programador não precisa (nem deve) declarar as funções de bibliotecas da linguagem C, por exemplo. Atualmente, se o programador declara algum identificador com o mesmo nome de um identificador pré-definido, uma mensagem de erro é dada, indicando "identificador duplamente declarado". Isto pode ser melhorado (vide capítulo 7 - Conclusões e Sugestões).

6.4 O ANALISADOR SINTÁTICO

O analisador sintático foi escrito usando a ferramenta YACC [AHO 87], o que tornou bastante simples o seu desenvolvimento. Assim, o analisador sintático, juntamente com as ações semânticas, torna-se uma coleção de regras de produção em BNFs. A cada produção, temos um trecho de programa em C que corresponde a uma ação semântica.

O YACC produz analisadores ascendentes, do tipo LALR(1). Sempre que uma produção é reduzida, a ação correspondente é executada.

Existem algumas produções em que foi necessário introduzir alguma ação semântica entre dois simbolos da forma sentencial (por exemplo, verificar se um identificador foi declarado). Para isto, um símbolo não-terminal fictício foi inserido na forma sentencial, no local da ação. Além disso, incluiu-se uma produção nula com este símbolo, contendo apenas a ação desejada.

O analisador sintático contém cerca de 1100 linhas em YACC, que gera o mesmo analisador em C que contém cerca de 1900 linhas.

6.5 AÇÕES SEMÂNTICAS

A rigor, LIDIA é uma linguagem *sensível ao contexto*, na medida em que sua gramática gera consistência de tipos de dados, identificadores declarados, proíbe que se passe uma constante como argumento cuja passagem seja declarada por referência, etc. Porém, é muito mais adequado simplificar a formalização da linguagem, expandindo-a ao ponto de se tornar uma linguagem *livre de contexto*, deixando verificações desse tipo para as ações semânticas. No restante dessa seção, são apresentadas as principais ações semânticas, referentes às classes dos identificadores da linguagem.

6.5.1 Declaração de Funções externas

Sempre que o analisador sintático reduz uma declaração de função externa, cria-se um nodo na tabela de identificadores, ligado, através do apontador obj (figura 6.6), a uma lista de *nodos da classe função*, que é mostrado na figura 6.7. No primeiro nodo da lista, armazena-se o tipo da função no campo tipo. Nos nodos restantes, armazena-se os tipos dos parâmetros e suas respectivas formas de passagem. Para isto, os campos tipo e passagem são utilizados.

(dom) da evidência; requisito (req), comandos e ação (acao) apontando para a tabela de comandos; e pais apontando para os arcos-pais da evidência.

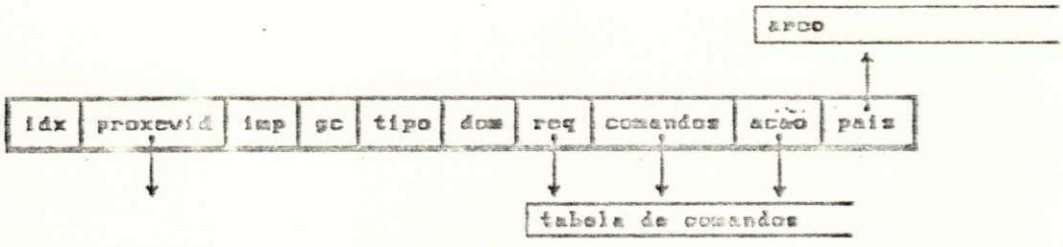


fig. 6.9 – Nodo da classe evidência

6.5.5 Definição de Grupo

Após reduzir uma definição de grupo, um nodo da tabela de identificadores é criado, contendo a classe = 'g' e nprm o valor dos grupos e/ou evidências subordinadas ao grupo definido. Através do campo obj, tem-se uma lista de *nodos da classe grupo*, como mostra a figura 6.10. Em cada nodo da lista, temos os campos idx (que corresponde ao endereco do símbolo no código objeto), id (que aponta para um identificador já declarado e contido na tabela de identificadores) e prox (que aponta para o próximo elemento da lista).

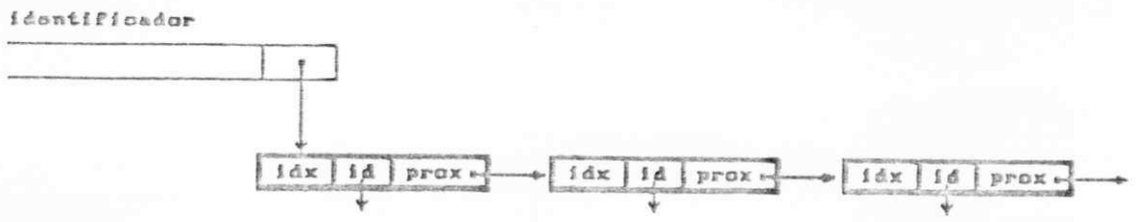


Fig. 6.10 - lista de nodos da classe grupo

6.5.6 Operador de Inferência

Assim que o analisador sintático reconhece um operador de inferência, com os seus filhos e fatores, cria-se um nodo da tabela de identificadores com o nome e texto igual a "#opinf" seguido de um número sequencial. A classe possui valor 'y' e obj aponta para um novo *nodo da classe operador*.

O nodo desta classe é mostrado na figura 6.11 e consta dos seguintes campos: idx é o endereço do nodo, a ser usado na geração do código; prox aponta para o próximo nodo da mesma classe; imp representa a importância do nodo de inferência; fator é o fator calculado juntamente com os fatores do piso (fpiso) e do teto (teto); filhos aponta para a lista de filhos do operador (seção 6.5.7); pais aponta para a lista de pais do mesmo nodo; e result indica o resultado da operação ('v' se todos os filhos forem

verdadeiros, 'f' se todos os filhos forem falsos, ' ' com o resultado intermediário).

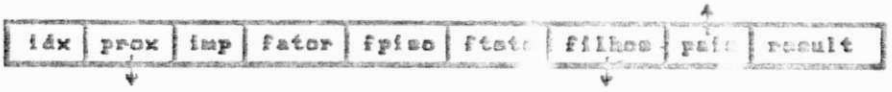


fig. 6.11 – o nodo da classe operador

6.5.7 Arcos

Como pode-se observar, existe ligação entre elementos da tabela de identificadores. Uma relação típica entre elementos são "identificador A *é pai* do identificador B". Para interligar dois nodos, e assim construir a rede de inferências, é necessária a estrutura *arco*. Como mostra o nodo da figura 6.12, idx é o endereco do nodo na tabela de arcos do código objeto; prox é um apontador usado para manter todos os arcos "costurados" em uma única lista linear p otimizar a busca sequencial; filho e pai apontam para os identificadores filho e pai, respectivamente; fnao e fsim contém os fatores do não e do sim, respectivamente; fator é o fator atual do arco; fpiso e fteto são os fatores do piso

e do teto, respectivamente; proxfilho aponta para o próximo arco da lista de filhos do identificador apontado por pai; e, finalmente, proxpai aponta para o próximo arco da lista de pais do identificador apontado por filho.

fig. 6.12 - o nodo arco

Um exemplo: uma evidência pode ser filha de várias hipóteses e cada uma dessas, pode conter várias evidências filhas. Isso quer dizer que, fixando uma hipótese e uma evidência relacionadas, um mesmo arco pertence a duas listas: a dos filhos da hipótese e a dos pais da evidência. Isto sem contar a lista única, encadeada pelo campo prox.

6.6 MENSAGENS DE ERRO

Nesta primeira versão, não está implementada a recuperação de erros sintáticos. Porém, os erros são detectados nos níveis léxico, sintático e semanticamente.

Uma mensagem de erro é apresentada ao usuário após o número da linha, na qual foi detectado o erro, e o código do erro. A tabela 6.2a e 6.2b apresentam as listas dos avisos e mensagens de erros apresentados pelo compilador.

Código	Mensagem
01	Evidência xxxxxxxx não usada
02	Hipótese xxxxxxxx classificada
03	Hipótese xxxxxxxx desclassificada

Tabela 6.2a - Avisos

Código	Mensagem
1	Identificador não é da classe apropriada
2	Identificador não declarado
3	Esperado um identificador de função
4	Tipo do operador incompatível com o dos operandos
5	Número de argumentos não coincide com a definição
6	Tipo(s) de argumento(s) diferente(s) da definição
7	Identificador duplamente declarado
8	Domínio incompatível com o tipo
9	Constantes devem aparecer na ordem
10	Simbolo "Self" não é permitido aqui
11	Comentário não fechado
12	Argumento deve ser uma variável
13	Símbolo ilegal
14	Memória insuficiente para se compilar o programa
15	Comando VOLUNT usado mais de uma vez
16	Caractere ilegal
99	Erro de Sintaxe

Tabela 6.2b - Mensagens de erros

6.7 GERAÇÃO DE CÓDIGO

A geração de código consiste de um módulo (geracod.c) que, a partir da tabela de símbolos devidamente preenchida,

cria o programa objeto em C. Basicamente, este módulo gera tabelas, no programa objeto, que correspondem às classes de identificadores. É impressa, também, a tabela de comandos em um formato apropriado.

O módulo gerador de código escreve as declarações das funções externas, cria listas invertidas de arcos, calcula teto, piso e fatores das hipóteses e dos operadores de inferência, graus de certeza das hipóteses e imprime as chamadas das funções externas.

O apêndice "A" apresenta um exemplo de programa fonte e o apêndice "B" mostra o seu código objeto correspondente, gerado pelo compilador.

CAPÍTULO VII

CONCLUSÕES E SUGESTÕES

Com este trabalho, acredita-se que é possível se desenvolver, com facilidade e rapidez, sistemas especialistas úteis, portáteis, eficientes, independentes e legíveis nesta nova linguagem.

Para tanto, porém, é necessário dar continuidade ao trabalho através do desenvolvimento do sistema de execução, incluindo o motor de inferências e o módulo de explanações. Neste sistema, alguns sub-sistemas podem ser identificados *a priori*, como o módulo de manipulação de menus, o módulo gráfico, de *voluntariamento*, de depuração, etc. Além disso, o sistema de execução deve prover outras bibliotecas de funções externas, disponíveis ao programador LIDIA e *interface* com *softwares* de bancos de dados e planilhas eletrônicas.

Uma vez concluído o sistema de execução, pode-se dar início ao desenvolvimento de vários sistemas especialistas para validar a linguagem e testar o compilador e o sistema de execução de forma mais efetiva. A partir daí, pode-se fazer um estudo da frequência de utilização de cada palavra

reservada, de cada comando, etc. Este estudo visa deixar o compilador mais eficiente e efetuar possíveis alterações na linguagem.

Desde já, algumas sugestões podem ser apresentadas, em relação à linguagem. Ei-las:

- Introduzir dois níveis de identificadores: os pre-definidos (nível 0) e os definidos pelo programador (nível 1). Assim, o programador poderá definir identificadores como *acos*, *abs*, *pow*, *strchr*, etc, que já estão pré-definidos. O compilador verifica se o identificador foi definido no nível 1 e, só no caso negativo, verifica o nível 0.

- Incluir parâmetros nos procedimentos internos. Podem ser passados por valor ou referência, da mesma forma que para os procedimentos externos.

- Eliminar a palavra reservada CALL das chamadas aos procedimentos internos. Assim, o procedimento interno será chamado da mesma maneira que são chamados os procedimentos externos.

- Incorporar recursão de procedimentos, tanto direta como indireta. Assim, um procedimento "A" poderá fazer chamada a um outro "B" definido posteriormente. Para isto,

basta declarar o procedimento "A" antes da definição do "B" e definir o procedimento "A" após o procedimento "B".

- Incluir, na linguagem, o conceito de hipótese-default. Esta hipótese pode ser definida pelo programador, antes das hipóteses reais do seu programa. Para cada declaração (limiar de aceitação, por exemplo) omitida dentro de uma definição de hipótese, será observada a hipótese *default*. Neste caso, o valor existente nesta hipótese será atribuído à hipótese em definição.

- Permitir ao usuário que ele defina suas funções de cálculo de importância, caso ele não queira usar o algoritmo padrão da linguagem.

- Permitir que o usuário especifique o *grau de certeza a priori*, como alternativa à probabilidade *a priori* de uma hipótese. Permitir, também, que o usuário especifique o limiar de aceitação, como um grau de certeza.

- Incorporar formas alternativas de tratamento de incerteza na linguagem, de modo que o programador possa utilizar também outros modelos similares, como o do MYCIN.

- Implementar *multi-instanciamento* de evidências, através de uma forma alternativa de uso dos *grupos*. Assim, pode-se ter uma evidência do "tipo grupo".

- Alterar as definições atuais dos operadores de inferência. INFOP, IAND e IOR não serão sinônimas. Quando se usar IAND, a gramática só deve aceitar o *fator do sim*, já que o *fator do não* deve ser assumido 1. Por outro lado, quando se usar o IOR, a gramática só deve aceitar o *fator do não*, já que o *fator do sim* deve ser assumido 1. INFOP só deve ser usado no caso em que ambos os fatores são diferentes de 1.

- Implementar a recuperação de erros sintáticos no compilador. Isto deve ser feito, após uma validação mais completa da sintaxe da linguagem pois, para cada erro, deve ser incorporada uma ou mais regras (do YACC) à especificação do analisador sintático.

Atualmente, o compilador LIDIA está com cerca de 6 mil linhas de código fonte (em 21 módulos) e 100 kbytes de código objeto, podendo ser executado, tanto em um microcomputador do tipo IBM PCxT com 640 kbytes de memória RAM, como em qualquer computador de porte igual ou mair que tenha um compilador C disponível.

O projeto completo da ferramenta LIDIA, na qual fará arte o presente compilador, consta de um ambiente integrado de edição, compilação, execução e depuração de programas. Estima-se que a ferramenta completa tenha cerca de 15 mil

linhas e possa ser executada numa configuração igualmente simples.

A implementação deste projeto (definição da linguagem LIDIA e construção do compilador) foi feita com um esforço equivalente a 8 meses de trabalho (8 horas por dia, 5 dias por semana) por um analista de sistemas com experiência de 5 anos na profissão.

A escolha de C como linguagem de desenvolvimento do projeto não foi feita considerando-se a simplicidade e rapidez no desenvolvimento, mas sim a portabilidade e eficiência do compilador. Praticamente, Hoje tem-se um compilador disponível em qualquer máquina, o que não seria tão fácil de se afirmar caso a linguagem escolhida fosse Prolog ou LISP, por exemplo, que possuem vários dialetos. O mesmo argumento é usado para a escolha da linguagem objeto. Um programa em LIDIA, não só pode ser compilado, como também executado em qualquer máquina, com maiores esforços.

REFERÊNCIAS BIBLIOGRÁFICAS

[ABEL 85] ABELSON, Harold et alii. *Structure and Interpretation of Computer Programs*. USA, McGrawHill Book Company, 1985, 542p.

[AHO 73] AHO, Alfred V. & ULLMAN, Jeffrey D. *The Teory of Parsing, Translation, and Compiling*. VOL II, USA, Prentice-Hall, 1973, 1002p.

[AHO 87] AHO, Alfred V. et alii. *Compilers, principles, techniques, and tools*. USA, Addison-Wesley, 1987, 796p.

[BUCH 84] BUCHANAN, Bruce G. & SHORTLIFFE, Edward H. *Rule-Based Expert Systems - The MYCIN Experiments of the Stanford Heuristic Programming Project*. USA, Addison-Wesley Publishing Company, 1984, 738p.

[BYTE 81] BYTE - Suplemento especial sobre Smalltalk, BYTE 6, nº8, USA, agosto 1981.

[BYTE 86] BYTE - Suplemento especial sobre Linguagens Orientadas a Objetos, BYTE 11, nº8, USA, agosto 1986.

[CASA 87] CASANOVA, Marco A. et alii. *Programação em Lógica e a linguagem Prolog*. São Paulo, Editora Edgard Bhucher, 1987, 461p.

[CUNH 87] Cunha, Horácio & Ribeiro, Sousa. *Introdução aos Sistemas Especialistas*, Rio de Janeiro, Livros Técnicos e Científicos Editora S.A. 1987, 137p.

[DIGI 87] DIGITALK. *Smalltalk/V - Tutorial and Programming Handbook*. Los Angeles, Digitalk, 1987, 514p.

[FERR 88] FERREIRA JUNIOR, J. Ulisses et alii. *LIDIA: Uma Linguagem para Desenvolvimento de Sistemas Especialistas Bayesianos*. Anais do 5º SBIA, SBC, Natal, 1988, p.271-280.

[FIRE 88] FIREBAUGH, Morris W. Artificial Inteligence - A Knowledge Based Approach. Boston, Boyd & Fraser Publishing Company, 1988, 740p.

[GENE 87] GENESERETH, Michael & NILSON, Nils J. *Logical Foundations of Artificial Inteligence*, USA, Morgan Kaufmann Publishers, 1987, 405p.

[GHEZ 85] GHEZZI, Carlo & JAZEYERI, Mehdi. *Programming Language Concepts (Conceito de Linguagens de Programação)*, VELOSO, Paulo A. S. Rio de Janeiro, CAMPUS, 1985, 306p.

[GRAH 88] GRAHAM, Ian & JONES, Peter Llewelyn. *Expert Systems Knowledge, Uncertainty and Decision*. New York, Chapman and Hall, 1988, 363p.

[HARM 88] HARMON, Paul et alii. *Expert Systems Tools and Applications*. San Francisco, John Wiley & Sons, Inc. 1988, 289p.

[KERN 78] KERNIGHAN, Brian W. & RITCHIE, Dennis M. *The C Programming Language*. New Jersey, Prentice-Hall, 1978, 228p.

[LYRA 89] LYRA, Isaque Alves et alii. *Núcleo de INTFORBAYES, uma Ferramenta para S.E.'s Bayesianos*. Anais do 6º SBIA, SBC, Rio de Janeiro, 1989, p.195-205.

[NEWT 87] NEWTON, S. Lee et alii. *Quantitative Models for Reasoning under Uncertainty in Knowledge-Based Expert Systems*. International Journal of Intelligent Systems, vol. II, 1987, p.15-38.

[NILS 82] NILSSON, Nils J. *Principles of Artificial Intelligence*, New York, Tioga Publishing Company, 1982, 476p.

[PEAR 86] PEARL, Judea. *Bayes Decision Methods*. Technical Report CSD-850023, Computer Science Dep., University of California, Los Angeles, 1986.

[WATE 86] WATERMAN, Donald A. *A Guide to Expert Systems*. New York, Addison-Wesley Publishing Company, 1986, 419p.

[WINS 81] WINSTON, Patrick Henry. *Inteligência Artificial*, Rio de Janeiro, Livros Técnicos e Científicos, 1987, 498p.

[ZUFF 78] ZUFFO, João Antônio. *Fundamentos da Arquitetura e Organização dos Microprocessadores*. São Paulo, Edgard Blucher, 1978.

APENDICE A

UM EXEMPLO DE PROGRAMA EM LIDIA

```
/*
   Este programa em LIDIA nao faz nada de util.
   Apenas serve de exemplo de construcoes sintaticas
*/

external
   void tempo( ref integer, yes_no),
   imp(real),
   cadastra();

proc pergunta;
  write "O paciente tem ",#Self,"? ";
  ask;
end;

evidence dor_cab     = "Dor de Cabeca",
         coriza,
         febre,
         tosse,
         dor_garg    = "dor na garganta",
         dor_ouvido  = "dor no ouvido",
         tontura,
         desmaio,
         falta_mem   = "falta de memoria",
         manchas     = "manchas na pele",
         vomito,
         rugas,
         fome        = "fome excessiva": yes_no;
  call pergunta;
end;

evidence nfilhos: integer[0:10];
  write "Quantos filhos?   ";
  ask;
  tempo(nfilhos,dor_cab = dor_garg);
end;


evidence idade: integer[0:90];
  write "Qual a sua Idade? ";
  ask;
  if idade > 20 then (
    tempo(self,false),
    tempo(idade,true)
  );
  tempo(nfilhos,false)
end;
```

```
evidence peso: real[0.0 : 200.0];
  requisite: idade > 25;
    write "Qual o seu peso? ";
    ask;
    while peso >= 0.5 do
      peso := sin(sqrt(sqrt(peso)));
end;

evidence altura: real[1.0 : 2.5];
  write "Qual a sua altura? ";
  ask;
  action:
    imp(altura);
    write strcat(#altura,"terminacao");
end;

evidence hora: string;
end;

hypothesis gripe(10)
  threshold 10;
  children
    coriza[0.5,2], dor_cab[0.5,2], febre[0.5,2],
    tosse[0.5,2],
    dor_garg[0.5,2];
  action:
    strset(#self,'*');
    _strtime(hora);
end;

hypothesis labirintite(10)
  threshold 50;
  children
    band(tontura[0.5,4], desmaio[0.1,1])[1,3];
end;

hypothesis gravidez(23)
  threshold 60;
  children
    (peso > 60)[0.3,4], tontura[0.3,7], desmaio[0.2,2],
          vomito[0.4,3], labirintite[1,3];
end;

group medidas = "Medidas do corpo: ":
      peso, altura, nfilhos;

group dores: dor_cab, dor_garg, dor_ouvido;

group cabeca = "Sintomas na cabeca: ":
     dores, coriza, tosse, tontura, desmaio, falta_mem;

group sintomas: medidas, cabeca;
```

```
initproc;
  cadastra();
end;

voluntproc;
  menu sintomas;
end;

consult;
  volunt;
  infer,
  feedback;
  reconsult;
end;

terminate;
end;
```

APÊNDICE B

UM EXEMPLO DE CÓDIGO OBJETO

```
#include "time.h"
#include "math.h"
#include "string.h"

typedef unsigned char byte;
typedef unsigned char boolean;
typedef unsigned short word;
typedef unsigned char* string;

#define NIL       -1

word     inicializacao =  172, volunt =   176,
consulta =   180, termino =   NIL;


/* funcoes externas */

extern  tempo(short*, char);
extern short imp(float);
extern short cadastra();

#define FALSE      0
#define TRUE       1

#define  NPOOL  73
word npool = NPOOL;
string pool[NPOOL] = {
/*    0 */  "#opinf0",
/*    1 */  "#expr0",
/*    2 */  "*",
/*    3 */  "? ",
/*    4 */  "Dor de Cabeca",
/*    5 */  "Medidas do corpo: ",
/*    6 */  "O paciente tem ",
/*    7 */  "Qual a sua Idade? ",
/*    8 */  "Qual a sua altura? ",
/*    9 */  "Qual o seu peso? ",
/*   10 */  "Quantos filhos?   ",
/*   11 */  "Sintomas na cabeca: ",
/*   12 */  "_strdate",
/*   13 */  "_strtime",
/*   14 */  "abs",
/*   15 */  "acos",
/*   16 */  "altura",
/*   17 */  "asin",
/*   18 */  "atan",
/*   19 */  "cabeca",
```

```
/*  20 */  "cadastra",
/*  21 */  "coriza",
/*  22 */  "cos",
/*  23 */  "cosh",
/*  24 */  "desmaio",
/*  25 */  "dor na garganta",
/*  26 */  "dor no ouvido",
/*  27 */  "dor_cab",
/*  28 */  "dor_garg",
/*  29 */  "dor_ouvido",
/*  30 */  "dores",
/*  31 */  "exp",
/*  32 */  "fabs",
/*  33 */  "falta de memoria",
/*  34 */  "falta_mem",
/*  35 */  "febre",
/*  36 */  "fome",
/*  37 */  "fome excessiva",
/*  38 */  "gravidez",
/*  39 */  "gripe",
/*  40 */  "hora",
/*  41 */  "idade",
/*  42 */  "imp",
/*  43 */  "labirintite",
/*  44 */  "log",
/*  45 */  "log10",
/*  46 */  "manchas",
/*  47 */  "manchas na pele",
/*  48 */  "math.h",
/*  49 */  "medidas",
/*  50 */  "nfilhos",
/*  51 */  "pergunta",
/*  52 */  "peso",
/*  53 */  "pow",
/*  54 */  "rugas",
/*  55 */  "sin",
/*  56 */  "sinh",
/*  57 */  "sintomas",
/*  58 */  "sqrt",
/*  59 */  "stdlib.h",
/*  60 */  "strcat",
/*  61 */  "strchr",
/*  62 */  "strdup",
/*  63 */  "string.h",
/*  64 */  "strset",
/*  65 */  "tan",
/*  66 */  "tanh",
/*  67 */  "tempo",
/*  68 */  "terminacao",
/*  69 */  "time.h",
/*  70 */  "tontura",
/*  71 */  "tosse",
/*  72 */  "vomito"
};
```

```
typedef struct regobj {
                    word nome, texto;
                    char classe;
                    word contin;
                  } tipregobj;

#define   NOBJ    53
word      nobj    = NOBJ;
tipregobj obj[NOBJ] = {
/*     0 */   {   67,   67, 'f',   -1 }   /* tempo */,
/*     1 */   {   42,   42, 'f',   -1 }   /* imp */,
/*     2 */   {   20,   20, 'f',   -1 }   /* cadastra */,
/*     3 */   {   15,   48, 'f',   -1 }   /* acos */,
/*     4 */   {   17,   48, 'f',   -1 }   /* asin */,
/*     5 */   {   18,   48, 'f',   -1 }   /* atan */,
/*     6 */   {   22,   48, 'f',   -1 }   /* cos */,
/*     7 */   {   23,   48, 'f',   -1 }   /* cosh */,
/*     8 */   {   31,   48, 'f',   -1 }   /* exp */,
/*     9 */   {   32,   48, 'f',   -1 }   /* fabs */,
/*    10 */   {   14,   59, 'f',   -1 }   /* abs */,
/*    11 */   {   44,   48, 'f',   -1 }   /* log */,
/*    12 */   {   45,   48, 'f',   -1 }   /* log10 */,
/*    13 */   {   53,   48, 'f',   -1 }   /* pow */,
/*    14 */   {   55,   48, 'f',   -1 }   /* sin */,
/*    15 */   {   56,   48, 'f',   -1 }   /* sinh */,
/*    16 */   {   58,   48, 'f',   -1 }   /* sqrt */,
/*    17 */   {   65,   48, 'f',   -1 }   /* tan */,
/*    18 */   {   66,   48, 'f',   -1 }   /* tanh */,
/*    19 */   {   62,   63, 'f',   -1 }   /* strdup */,
/*    20 */   {   60,   63, 'f',   -1 }   /* strcat */,
/*    21 */   {   64,   63, 'f',   -1 }   /* strset */,
/*    22 */   {   61,   63, 'f',   -1 }   /* strchr */,
/*    23 */   {   12,   69, 'f',   -1 }   /* _strdate */,
/*    24 */   {   13,   69, 'f',   -1 }   /* _strtime */,
/*    25 */   {   51,   51, 'p',    0 }   /* pergunta */,
/*    26 */   {   27,    4, 'e',    0 }   /* dor_cab */,
/*    27 */   {   21,   21, 'e',    1 }   /* coriza */,
/*    28 */   {   35,   35, 'e',    2 }   /* febre */,
/*    29 */   {   71,   71, 'e',    3 }   /* tosse */,
/*    30 */   {   28,   25, 'e',    4 }   /* dor_garg */,
/*    31 */   {   29,   26, 'e',    5 }   /* dor_ouvido */,
/*    32 */   {   70,   70, 'e',    6 }   /* tontura */,
/*    33 */   {   24,   24, 'e',    7 }   /* desmaio */,
/*    34 */   {   34,   33, 'e',    8 }   /* falta_mem */,
/*    35 */   {   46,   47, 'e',    9 }   /* manchas */,
/*    36 */   {   72,   72, 'e',   10 }   /* vomito */,
/*    37 */   {   54,   54, 'e',   11 }   /* rugas */,
/*    38 */   {   36,   37, 'e',   12 }   /* fome */,
/*    39 */   {   50,   50, 'e',   13 }   /* nfilhos */,
/*    40 */   {   41,   41, 'e',   14 }   /* idade */,
/*    41 */   {   52,   52, 'e',   15 }   /* peso */,
/*    42 */   {   16,   16, 'e',   16 }   /* altura */,
```

```
/*    43 */    {    40,    40,  'e',    17 }    /* hora */,
/*    44 */    {    39,    39,  'h',     0 }    /* gripe */,
/*    45 */    {    43,    43,  'h',     1 }    /* labirintite */,
/*    46 */    {     0,     0,  'y',     0 }    /* Hopinf0 */,
/*    47 */    {    38,    38,  'h',     2 }    /* gravidez */,
/*    48 */    {     1,     1,  'x',     0 }    /* Hexpr0 */,
/*    49 */    {    49,     5,  'g',     0 }    /* medidas */,
/*    50 */    {    30,    30,  'g',     4 }    /* dores */,
/*    51 */    {    19,    11,  'g',     8 }    /* cabeca */,
/*    52 */    {    57,    57,  'g',    15 }    /* sintomas */,
};

#define  NGRUPO  18
word     ngrupo  = NGRUPO;
int grupo[NGRUPO] = {
        /*     0 */     41,
        /*     1 */     42,
        /*     2 */     39,
        /*     3 */    NIL,
        /*     4 */     26,
        /*     5 */     30,
        /*     6 */     31,
        /*     7 */    NIL,
        /*     8 */     50,
        /*     9 */     27,
        /*    10 */     29,
        /*    11 */     32,
        /*    12 */     33,
        /*    13 */     34,
        /*    14 */    NIL,
        /*    15 */     49,
        /*    16 */     51,
        /*    17 */    NIL
};

union uval {
    short   i;
    char    c;
    float   r;
    boolean b;
    string  s;
};

typedef struct {
          char    tipo;
          union uval valor;
        } regpilha;

#define MAXPILHA 100
regpilha  pilha[MAXPILHA];
int       topo = -1;

typedef struct {
          char    tipo;
```

```
          union uval valor;
        } tipregconst;

#define  NCONST 14
word nconst = NCONST;
tipregconst cst[NCONST] = {
          /*    0  */    { 's', 6},
          /*    1  */    { 's', 3},
          /*    2  */    { 's', 10},
          /*    3  */    { 's', 7},
          /*    4  */    { 'i', 20},
          /*    5  */    { 'b', FALSE},
          /*    6  */    { 'b', TRUE},
          /*    7  */    { 'i', 25},
          /*    8  */    { 's', 9},
          /*    9  */    { 'r', 0.500000},
          /*   10  */    { 's', 8},
          /*   11  */    { 's', 68},
          /*   12  */    { 's', 2},
          /*   13  */    { 'i', 60}
};


typedef struct {
          float imp;
          word  filho, pai;
          float fnao, fsim, fator, fpiso, fteto;
          word  proxfilho, proxpai;
        } tipregarco;

#define  NARCO  13
word     narco  = NARCO;
tipregarco arco[NARCO] = {
   /*    0 */ {  0.00,   27,     44,      0.500,    2.000,
1.000,    0.500,    2.000,    1,    NIL} ,
   /*    1 */ {  0.00,   26,     44,      0.500,    2.000,
1.000,    0.500,    2.000,    2,    NIL} ,
   /*    2 */ {  0.00,   28,     44,      0.500,    2.000,
1.000,    0.500,    2.000,    3,    NIL} ,
   /*    3 */ {  0.00,   29,     44,      0.500,    2.000,
1.000,    0.500,    2.000,    4,    NIL} ,
   /*    4 */ {  0.00,   30,     44,      0.500,    2.000,
1.000,    0.500,    2.000,  NIL,    NIL} ,
   /*    5 */ {  0.00,   32,     46,      0.500,    4.000,
1.000,    0.500,    4.000,    6,      9} ,
   /*    6 */ {  0.00,   33,     46,      0.100,    1.000,
1.000,    0.100,    1.000,  NIL,     10} ,
   /*    7 */ {  0.00,   46,     45,      1.000,    3.000,
1.000,    0.050,   12.000,  NIL,    NIL} ,
   /*    8 */ {  0.00,   48,     47,      0.300,    4.000,
1.000,    0.300,    4.000,    9,    NIL} ,
   /*    9 */ {  0.00,   32,     47,      0.300,    7.000,
1.000,    0.300,    7.000,   10,    NIL} ,
```

```
    /*   10 */ (  0.00,    33,     47,      0.200,     2.000,
1.000,     0.200,     2.000,    11,    NIL) ,
    /*   11 */ (  0.00,    36,     47,      0.400,     3.000,
1.000,     0.400,     3.000,    12,    NIL) ,
    /*   12 */ (  0.00,    45,     47,      1.000,     3.000,
1.000,     1.000,     2.048,   NIL,    NIL) ,
);

typedef struct {
                    char    tipo;
                    int     valmin, valmax;
                    union uval valor;
                    word    req, com, acao;
                    float   gc, imp;
                    word    pais;
                  } tipregevid;

#define     NEVID  18
word        nevid  = NEVID;
tipregevid evid[NEVID] = {
      /*    0 */   ( 'b',      0,      1,    0,    NIL,     12,
15, 2.0, 0.077703,     1) ,
      /*    1 */   ( 'b',      0,      1,    0,    NIL,     12,
15, 2.0, 0.077703,     0) ,
      /*    2 */   ( 'b',      0,      1,    0,    NIL,     12,
15, 2.0, 0.077703,     2) ,
      /*    3 */   ( 'b',      0,      1,    0,    NIL,     12,
15, 2.0, 0.077703,     3) ,
      /*    4 */   ( 'b',      0,      1,    0,    NIL,     12,
15, 2.0, 0.077703,     4) ,
      /*    5 */   ( 'b',      0,      1,    0,    NIL,     12,
15, 2.0, 0.000000,   NIL) ,
      /*    6 */   ( 'b',      0,      1,    0,    NIL,     12,
15, 2.0, 0.227290,     5) ,
      /*    7 */   ( 'b',      0,      1,    0,    NIL,     12,
15, 2.0, 0.227290,     6) ,
      /*    8 */   ( 'b',      0,      1,    0,    NIL,     12,
15, 2.0, 0.000000,   NIL) ,
      /*    9 */   ( 'b',      0,      1,    0,    NIL,     12,
15, 2.0, 0.000000,   NIL) ,
      /*   10 */   ( 'b',      0,      1,    0,    NIL,     12,
15, 2.0, 0.227290,    11) ,
      /*   11 */   ( 'b',      0,      1,    0,    NIL,     12,
15, 2.0, 0.000000,   NIL) ,
      /*   12 */   ( 'b',      0,      1,    0,    NIL,     12,
15, 2.0, 0.000000,   NIL) ,
      /*   13 */   ( 'i',      0,     10,    0,    NIL,     16,
35, 2.0, 0.000000,   NIL) ,
      /*   14 */   ( 'i',      0,     90,    0,    NIL,     36,
78, 2.0, 0.000000,   NIL) ,
      /*   15 */   ( 'r',      0,      1,    0,     79,     86,
121, 2.0, 0.000000,   NIL) ,
      /*   16 */   ( 'r',      2,      3,    0,    NIL,    122,
128, 2.0, 0.000000,   NIL) ,
```

```
     /*  17 */   ( 's',     -1,     -1,    0,    NIL,    NIL,
148, 2.0, 0.000000,  NIL)
};

#define  NCR  4
word     ncr  = NCR;
float cr[NCR] = {
                 /*    0 */ 0.000000,  200.000000,
                 /*    2 */ 1.000000,  2.500000
};

typedef struct {
           float        gc, chance0, chance;
           float        lim;
           word         req, filhos, pais, acao;
           float        imp, piso, teto;
           char         estado;
        } tipreghip;

#define   NHIP  3
#define   MAXREAL 339999995214436425000000000000000000000.0
word      nhip  = NHIP;
tipreghip hip[NHIP] = {
/* 0 */  {0.000000, 0.111111, 0.111111, 0.111111,  NIL,
0,   NIL,    149,     MAXREAL, 0.003472, 3.555556, 'D'},
/* 1 */  {0.000000, 0.111111, 0.111111, 1.000000,  NIL,
7,     12,   NIL,        0.23, 0.005556, 1.333333, 'D'},
/* 2 */  {0.000000, 0.298701, 0.298701, 1.500000,  NIL,
8,   NIL,    NIL,     MAXREAL, 0.002151, 102.753258, 'D'}
};

typedef struct {
   char result;
   word pai, ini, fim;
} tipregexp;

#define NEXPR 1
word nexpr = 1;
tipregexp expr[NEXPR] = {
       /*    0 */  {' ',      8,    164,    171}
};

typedef struct {
           float  imp;
           float  fator, fpiso, fteto;
           word   filhos, pais;
           char   result;
        } tipreginf;

#define   NINF  1
word      ninf  = NINF;
tipreginf inf[NINF] = {
/* 0 */  {     0.057,      1.000,      0.050,      4.000,
5,     7,  ' '}
```

```
};


#define         NOP             0
#define         ASK             1
#define         VOLUNT          2
#define         INFER           3
#define         FEEDBACK        4
#define         RECONSULT       5
#define         CALLFUNC        6
#define         WRITE           7
#define         MENU            8
#define         CALLPROC        9
#define         JPZ            10
#define         JPNZ           11
#define         JUMP           12
#define         RET            13
#define         STEV           14
#define         OR             15
#define         AND            16
#define         EQ             17
#define         NE             18
#define         LT             19
#define         GT             20
#define         LE             21
#define         GE             22
#define         IDIV           23
#define         MOD            24
#define         MINUS          25
#define         ADD            26
#define         SUB            27
#define         MUL            28
#define         DIV            29
#define         FCALL          30
#define         CONST          31
#define         OBJ            32
#define         SELF           33
#define         TEXT           34
#define         ADDR           35

#define         NCMD  185
word            ncmd  = NCMD;
byte            cmd[185] = {
        /*    0 */        CONST, 0, 0  /* 0 */,
        /*    3 */         SELF,
        /*    4 */         TEXT,
        /*    5 */        CONST, 0, 1  /* 1 */,
        /*    8 */        WRITE, 3,
        /*   10 */          ASK,
        /*   11 */          RET,
        /*   12 */    CALLPROC, 0, 25  /* 25 */,
        /*   15 */          RET,
        /*   16 */        CONST, 0, 2  /* 2 */,
        /*   19 */        WRITE, 1,
```

```
/*   21 */          ASK,
/*   22 */          OBJ, 0, 39  /* 39 */,
/*   25 */          OBJ, 0, 26  /* 26 */,
/*   28 */          OBJ, 0, 30  /* 30 */,
/*   31 */           EQ,
/*   32 */    CALLFUNC, 0, 0  /* 0 */,
/*   35 */          RET,
/*   36 */       CONST, 0, 3  /* 3 */,
/*   39 */       WRITE, 1,
/*   41 */          ASK,
/*   42 */          OBJ, 0, 40  /* 40 */,
/*   45 */       CONST, 0, 4  /* 4 */,
/*   48 */          GT,
/*   49 */          JPZ, 0, 68  /* 68 */,
/*   52 */         SELF,
/*   53 */       CONST, 0, 5  /* 5 */,
/*   56 */    CALLFUNC, 0, 0  /* 0 */,
/*   59 */          OBJ, 0, 40  /* 40 */,
/*   62 */       CONST, 0, 6  /* 6 */,
/*   65 */    CALLFUNC, 0, 0  /* 0 */,
/*   68 */          NOP,
/*   69 */          OBJ, 0, 39  /* 39 */,
/*   72 */       CONST, 0, 5  /* 5 */,
/*   75 */    CALLFUNC, 0, 0  /* 0 */,
/*   78 */          RET,
/*   79 */          OBJ, 0, 40  /* 40 */,
/*   82 */       CONST, 0, 7  /* 7 */,
/*   85 */          GT,
/*   86 */       CONST, 0, 8  /* 8 */,
/*   89 */       WRITE, 1,
/*   91 */          ASK,
/*   92 */          OBJ, 0, 41  /* 41 */,
/*   95 */       CONST, 0, 9  /* 9 */,
/*   98 */          GE,
/*   99 */          JPZ, 0, 120  /* 120 */,
/*  102 */          OBJ, 0, 41  /* 41 */,
/*  105 */       FCALL, 0, 16  /* 16 */,
/*  108 */       FCALL, 0, 16  /* 16 */,
/*  111 */       FCALL, 0, 14  /* 14 */,
/*  114 */        STEV, 0, 41  /* 41 */,
/*  117 */        JUMP, 0, 92  /* 92 */,
/*  120 */          NOP,
/*  121 */          RET,
/*  122 */       CONST, 0, 10  /* 10 */,
/*  125 */       WRITE, 1,
/*  127 */          ASK,
/*  128 */          RET,
/*  129 */          OBJ, 0, 42  /* 42 */,
/*  132 */    CALLFUNC, 0, 1  /* 1 */,
/*  135 */          OBJ, 0, 42  /* 42 */,
/*  138 */         TEXT,
/*  139 */       CONST, 0, 11  /* 11 */,
/*  142 */       FCALL, 0, 20  /* 20 */,
/*  145 */       WRITE, 3,
```

```
        /*  147 */          RET,
        /*  148 */          RET,
        /*  149 */         SELF,
        /*  150 */         TEXT,
        /*  151 */        CONST, 0, 12  /* 12 */,
        /*  154 */     CALLFUNC, 0, 21  /* 21 */,
        /*  157 */          OBJ, 0, 43  /* 43 */,
        /*  160 */     CALLFUNC, 0, 24  /* 24 */,
        /*  163 */          RET,
        /*  164 */          OBJ, 0, 41  /* 41 */,
        /*  167 */        CONST, 0, 13  /* 13 */,
        /*  170 */           GT,
        /*  171 */          RET,
        /*  172 */     CALLFUNC, 0, 2  /* 2 */,
        /*  175 */          RET,
        /*  176 */         MENU, 0, 52  /* 52 */,
        /*  179 */          RET,
        /*  180 */       VOLUNT,
        /*  181 */        INFER,
        /*  182 */     FEEDBACK,
        /*  183 */   RECONSULT,
        /*  184 */          RET
);

callfunc(n)
  int n;
{
  switch (n) {
    case 23:  _strdate( pilha[topo].valor.s);  topo -= 1;
break;
    case 24:  _strtime( pilha[topo].valor.s);  topo -= 1;
break;
    case 10:  pilha[topo].valor.i =
abs(pilha[topo].valor.i); break;
    case 3:  pilha[topo].valor.r =
acos(pilha[topo].valor.r); break;
    case 4:  pilha[topo].valor.r =
asin(pilha[topo].valor.r); break;
    case 5:  pilha[topo].valor.r =
atan(pilha[topo].valor.r); break;
    case 2:  pilha[topo-(-1)].valor.i = cadastra(),
topo -=1; break;
    case 6:  pilha[topo].valor.r = cos(pilha[topo].valor.r),
break;
    case 7:  pilha[topo].valor.r =
cosh(pilha[topo].valor.r); break;
    case 8:  pilha[topo].valor.r = exp(pilha[top ].valor.r);
break;
    case 9:  pilha[topo].valor.r =
fabs(pilha[topo].valor.r); break;
    case 1:  pilha[topo].valor.i = imp(pilha[topo].valor.r);
break;
    case 11:  pilha[topo].valor.r =
log(pilha[topo].valor.r); break;
```

```
    case 12:  pilha[topo].valor.r =
log10(pilha[topo].valor.r); break;
    case 13:  pilha[topo-(1)].valor.r =
pow(pilha[topo-1].valor.r, pilha[topo].valor.r);  topo -= 1;
break;
    case 14:  pilha[topo].valor.r =
sin(pilha[topo].valor.r); break;
    case 15:  pilha[topo].valor.r =
sinh(pilha[topo].valor.r); break;
    case 16:  pilha[topo].valor.r =
sqrt(pilha[topo].valor.r); break;
    case 20:  pilha[topo-(1)].valor.s =
strcat(pilha[topo-1].valor.s, pilha[topo].valor.s);
topo -= 1; break;
    case 22:  pilha[topo-(1)].valor.s =
strchr(pilha[topo-1].valor.s, pilha[topo].valor.c);
topo -= 1; break;
    case 19:  pilha[topo].valor.s =
strdup(pilha[topo].valor.s); break;
    case 21:  pilha[topo-(1)].valor.s =
strset(pilha[topo-1].valor.s, pilha[topo].valor.c);
topo -= 1; break;
    case 17:  pilha[topo].valor.r =
tan(pilha[topo].valor.r); break;
    case 18:  pilha[topo].valor.r =
tanh(pilha[topo].valor.r); break;
    case 0:
tempo(&pilha[topo-1].valor.i, pilha[topo].valor.b);
topo -= 2; break;
  }
}
```

APÊNDICE C

GLOSSÁRIO DE TERMOS TÉCNICOS

apontar *v.t.* 1. Indicar através de um endereço ou índice __para 2. Conter o endereço de.

apontador *sm.* Aquilo que aponta.

caminhamento *sm.* Acesso aos nodos de uma estrutura de dados, segundo uma certa ordem.

dedutível *adj.* Que é instanciado através de inferência.

default 1 *adj.* Padrão. 2. *sm.* valor assumido quando o mesmo não é especificado explicitamente, valor padrão. Palavra oriunda da expressão francesa *de fault*.

dígito *sm.* Algarismo decimal.

inicialização *sf.* 1. Fase da execução de um programa em LIDIA em que são feitas atribuições de valores iniciais a uma ou mais evidências da linguagem. Módulo de _____ 2. Módulo especial da linguagem em que são feitas inicializações.

inicializar *v.t.* Atribuir valores iniciais a uma ou mais evidências da linguagem.

instanciar *v.t.* Atribuir valor a. Refere-se a evidência ou objeto.

interface *sf.* Região física ou lógica em que se faz ligação de dois dispositivos, procedimentos ou máquinas.

literal *sm.* Caractere ou sequência de caracteres.

livre *adj.* Diz-se da evidência não instanciada.

multi-instanciamento *sm.* Ato ou efeito de instanciar com múltiplos valores.

nodo *sm.* Registro de uma estrutura encadeada alocado dinamicamente.

perguntável *adj.* Que é *instanciado* através de uma pergunta.

pool *sm.* Agrupamento de objetos. ___ de literais. Estrutura de dados usada pelo compilador LIDIA em que se agrupam os literais do programa fonte.

portátil *adj.* Que é passível de transporte de uma máquina para outra; aproximação do termo inglês *portable*.

progressivo *adj.* 1. Para frente. 2. Encadeamento ___. Do termo inglês *forward chaining*; Encadeamento para frente.

reinicializar *v.t.* Inicializar de novo.

regressivo *adj.* 1. Para trás. 2. Caminhamento ___. Tradução do termo inglês *backward chaining*; Encademento para trás.

software *sm.* 1. Programa ou informação existente no computador; parte lógica do computador. 2. Programa de suporte. 3. Programa de computador.

string *sm.* 1. Cadeia de caracteres, Sequência de uma ou mais letras, algarismos decimais ou outros símbolos usados pelo computador. 2. Tipo de dado usado na linguagem LIDIA. 3. Tipo de dado usado no compilador LIDIA constituído de um apontador para o início de uma sequência de caracteres.

voluntariamento *sm.* 1. Ato ou efeito de instanciar um conjunto de evidências por vontade do usuário. 2. Fase da execução de um programa em LIDIA em que evidências são instanciadas através da escolha das mesmas pelo usuário. Módulo de ___ 3. Módulo especial da linguagem destinado a ... er o voluntariamento.