

INTRODUÇÃO À PROGRAMAÇÃO **PROLOG**

Luiz A. M. Palazzo

Editora da Universidade Católica de Pelotas / UCPEL
Rua Félix da Cunha, 412 - Fone (0532)22-1555 - Fax (0532)25-3105
Pelotas - RS - Brasil

EDUCAT

Editora da Universidade Católica de Pelotas
Pelotas, 1997

© 1997 LUIZ A. M. PALAZZO

SUMÁRIO

1. LÓGICA E PROGRAMAÇÃO DE COMPUTADORES	1
1.1 AS RAÍZES	1
1.2 PROGRAMAÇÃO EM LÓGICA	2
1.3 APLICAÇÕES	4
1.4 A QUINTA GERAÇÃO	6
1.5 PORQUE ESTUDAR PROLOG	8
RESUMO	9
2. A LINGUAGEM PROLOG	11
2.1 FATOS	11
2.2 REGRAS	14
2.3 CONSTRUÇÕES RECURSIVAS	17
2.4 CONSULTAS	19
2.5 O SIGNIFICADO DOS PROGRAMAS PROLOG	21
RESUMO	22
EXERCÍCIOS	22
3. SINTAXE E SEMÂNTICA	24
3.1 OBJETOS	24
3.2 UNIFICAÇÃO	27
3.3 SEMÂNTICA DECLARATIVA E SEMÂNTICA PROCEDIMENTAL	28
3.4 SEMÂNTICA OPERACIONAL	30
RESUMO	30
EXERCÍCIOS	31
4. OPERADORES E ARITMÉTICA	33
4.1 OPERADORES	33
4.2 ARITMÉTICA	36
RESUMO	38
EXERCÍCIOS	39
5. PROCESSAMENTO DE LISTAS	41
5.1 REPRESENTAÇÃO DE LISTAS	41
5.2 OPERAÇÕES SOBRE LISTAS	42
5.3 OUTROS EXEMPLOS	48
RESUMO	49
EXERCÍCIOS	50
6. CONTROLE	51
6.1 BACKTRACKING	51
6.2 O OPERADOR "CUT"	52
6.3 APLICAÇÕES DO CUT	56
6.4 NEGAÇÃO POR FALHA	57
6.5 CUIDADOS COM O CUT E A NEGAÇÃO	58
RESUMO	60
EXERCÍCIOS	60
7. ESTRUTURAS DE DADOS	62
7.1 RECUPERAÇÃO DE INFORMAÇÕES	62
7.2 ABSTRAÇÃO DE DADOS	64
7.3 UM AUTÔMATO FINITO NÃO-DETERMINÍSTICO	65
7.4 PLANEJAMENTO DE ROTEIROS AÉREOS	67
RESUMO	69
EXERCÍCIOS	69
8. ENTRADA E SAÍDA	71
8.1 ARQUIVOS DE DADOS	71
8.2 PROCESSAMENTO DE ARQUIVOS DE TERMOS	73
8.3 PROCESSAMENTO DE CARACTERES	77
8.4 CONVERSÃO DE TERMOS	78
8.5 LEITURA DE PROGRAMAS	79
RESUMO	80
EXERCÍCIOS	80
9. PREDICADOS EXTRALÓGICOS	82
9.1 TIPOS DE TERMOS	82
9.2 CONSTRUÇÃO E DECOMPOSIÇÃO DE TERMOS	84
9.3 EQUIVALÊNCIAS E DESIGUALDADES	85
9.4 PROGRAMAS OU BASES DE DADOS?	86
9.5 RECURSOS PARA O CONTROLE DE PROGRAMAS	89

9.6 BAGOF, SETOF E FINDALL	89
RESUMO	91
EXERCÍCIOS	91
10. LÓGICA E BASES DE DADOS	93
10.1 BASES DE DADOS RELACIONAIS	93
10.2 RECUPERAÇÃO DE INFORMAÇÕES	95
10.3 ATUALIZAÇÃO DA BASE DE DADOS	96
10.4 MODELAGEM DE DADOS	97
10.5 ALÉM DO MODELO RELACIONAL	99
10.6 REDES SEMÂNTICAS	99
RESUMO	103
EXERCÍCIOS	103
11. PROGRAMAÇÃO SIMBÓLICA	105
11.1 DIFERENCIAÇÃO SIMBÓLICA	105
11.2 MANIPULAÇÃO DE FÓRMULAS	105
11.3 OS OPERADORES REVISITADOS	105
11.4 AVALIAÇÃO DE FÓRMULAS	106
11.5 SIMPLIFICAÇÃO ALGÉBRICA	107
11.6 INTEGRAÇÃO	109
RESUMO	109
EXERCÍCIOS	110
12. METODOLOGIA DA PROGRAMAÇÃO EM LÓGICA	111
12.1 PRINCÍPIOS GERAIS DA BOA PROGRAMAÇÃO	111
12.2 COMO PENSAR EM PROLOG	112
12.3 ESTILO DE PROGRAMAÇÃO	114
12.4 DEPURAÇÃO DE PROGRAMAS	116
12.5 EFICIÊNCIA	117
12.6 PROGRAMAÇÃO ITERATIVA	122
RESUMO	123
EXERCÍCIOS	124
13. OPERAÇÕES SOBRE ESTRUTURAS DE DADOS	125
13.1 CLASSIFICAÇÃO EM LISTAS	125
13.2 REPRESENTAÇÃO DE CONJUNTOS	127
13.3 DICIONÁRIOS BINÁRIOS	129
13.4 INSERÇÃO E REMOÇÃO DE ITENS EM DICIONÁRIOS BINÁRIOS	130
13.5 APRESENTAÇÃO DE ÁRVORES	133
13.6 GRAFOS	133
RESUMO	138
EXERCÍCIOS	139
14. ESTRATÉGIAS PARA A SOLUÇÃO DE PROBLEMAS	140
14.1 CONCEITOS BÁSICOS	140
14.2 PESQUISA EM PROFUNDIDADE	143
14.3 PESQUISA EM AMPLITUDE	146
14.4 PESQUISA EM GRAFOS, OTIMIZAÇÃO E COMPLEXIDADE	150
RESUMO	151
EXERCÍCIOS	151
15. PESQUISA HEURÍSTICA	153
15.1 BEST-FIRST SEARCH	153
15.2 UMA APLICAÇÃO DA PESQUISA HEURÍSTICA	158
RESUMO	160
EXERCÍCIOS	161
16. REDUÇÃO DE PROBLEMAS E GRAFOS E/OU	162
16.1 REPRESENTAÇÃO DE PROBLEMAS	162
16.2 EXEMPLOS DE REPRESENTAÇÃO DE PROBLEMAS EM GRAFOS E/OU	165
16.3 PROCEDIMENTOS BÁSICOS DE PESQUISA EM GRAFOS E/OU	167
16.4 PESQUISA HEURÍSTICA EM GRAFOS E/OU	170
RESUMO	178
EXERCÍCIOS	178
APÊNDICE A	179
A.2 SEMÂNTICA MODELO-TEORÉTICA	182
A.3 SEMÂNTICA PROVA-TEORÉTICA	189
BIBLIOGRAFIA	191

Tanto pelo privilégio da amizade de vários anos, como pela condição de colega profissional do prof. Luiz Antonio Palazzo, já há muito acompanho sua contribuição à cultura em Ciência da Computação na região em que trabalhamos (zona sul do Rio Grande do Sul). Com graduação e pós-graduação pela UFRGS, vem marcando sua atuação desde a época de estudante, tanto no meio acadêmico como na comunidade em geral, por uma postura de vanguarda na busca de tecnologias para um uso racional e eficiente da computação. Sem dúvida, este livro permitirá que um número maior de pessoas se beneficiem de sua larga experiência no ofício de ensinar.

A estrutura do livro mescla o contexto histórico da Inteligência Artificial (IA) com o estudo do Prolog, uma das mais difundidas linguagens para Programação em Lógica. O conteúdo, por sua vez, tem como ponto alto contemplar uma rigorosa conceituação formal, cujo emprego é caracterizado por exemplos claros e significativos.

O emprego das linguagens para Programação em Lógica ganhou significativo impulso com o projeto Japonês de Sistemas Computacionais de Quinta Geração (1982-1992), o qual investiu alternativas de hardware e software para atender o desenvolvimento de aplicações que contemplavam metas ambiciosas, tais como reconhecimento de imagens, processamento da linguagem natural, processamento de conhecimento, etc.

As linguagens para Programação em Lógica, a exemplo do Prolog, outrora empregadas principalmente na prototipação, já podem ser utilizadas para resolver, com bom desempenho, complexos problemas reais de IA. Isto se tornou possível pela disponibilidade de processadores poderosos a custos reduzidos, bem como pela disseminação do uso de arquiteturas paralelas.

Neste trabalho são ressaltadas, com muita propriedade, as vantagens do emprego da lógica clausal para programação de computadores, resgatando a "elegância" das linguagens para Programação em Lógica, nas quais o programador tem como principal preocupação a especificação em Prolog do problema a ser resolvido, ficando a cargo do sistema computacional a gerência dos mecanismos de busca das possíveis soluções.

Esta obra moderna, das poucas em português no seu estilo, vem preencher uma lacuna editorial, trazendo a estudantes e profissionais da ciência da computação uma abordagem ampla, porém não menos crítica e objetiva, das perspectivas do uso da Programação em Lógica.

Adenauer Corrêa Yamin
Pelotas, RS

1. LÓGICA E PROGRAMAÇÃO DE COMPUTADORES

A lógica é a *ciência do pensamento correto*¹. Esta declaração não implica contudo em afirmar que ela seja a *ciência da verdade*. Mesmo que tudo o que se permita afirmar dentro da lógica seja supostamente verdadeiro em determinado contexto, as mesmas afirmações podem resultar falsas se aplicadas ao mundo real. Os filósofos da lógica afirmam que, "para entender o que *realmente* acontece no mundo, precisamos entender *o que não acontece*", isto é, as propriedades *invariantes* das entidades ou objetos que o compõem. Com essa idéia em mente, podemos considerar lógicos os conjuntos de declarações que possuem a propriedade de ser verdadeiros ou falsos independentemente do tempo ou lugar que ocupam no universo considerado. Este *insight* inicial costuma ser de grande valia para entender como a lógica pode ser empregada na programação de computadores com grande vantagem sobre as linguagens convencionais. O cálculo proposicional, que é o subconjunto da lógica matemática mais diretamente envolvido nesse processo, formaliza a estrutura lógica mais elementar do discurso definindo precisamente o significado dos conectivos *e*, *ou*, *não*, *se...então* e outros. No presente capítulo esboça-se a forma como evoluiu a idéia de empregar a lógica como linguagem de programação de computadores, comenta-se os principais usos e aplicações das linguagens baseadas na lógica, relata-se os resultados mais significativos obtidos ao longo dos dez anos do controvertido projeto japonês para o desenvolvimento dos denominados "Computadores de Quinta Geração" e, por fim, se tenta antecipar as perspectivas mais promissoras da pesquisa neste ramo do conhecimento científico.

1.1 AS RAÍZES

O uso da lógica na representação dos processos de raciocínio remonta aos estudos de Boole (1815-1864) e de De Morgan (1806-1871), sobre o que veio a ser mais tarde chamado "Álgebra de Boole". Como o próprio nome indica, esses trabalhos estavam mais próximos de outras teorias matemáticas do que propriamente da lógica. Deve-se ao matemático alemão Gottlob Frege no seu "Begriffsschrift" (1879) a primeira versão do que hoje denominamos cálculo de predicados, proposto por ele como uma ferramenta para formalizar princípios lógicos. Esse sistema oferecia uma notação rica e consistente que Frege pretendia adequada para a representação de todos os conceitos matemáticos e para a formalização exata do raciocínio dedutivo sobre tais conceitos, o que, afinal, acabou acontecendo.

No final do século passado a matemática havia atingido um estágio de desenvolvimento mais do que propício à exploração do novo instrumento proposto por Frege. Os matemáticos estavam abertos a novas áreas de pesquisa que demandavam profundo entendimento lógico assim como procedimentos sistemáticos de prova de teoremas mais poderosos e eficientes do que os até então empregados. Alguns dos trabalhos mais significativos deste período foram a reconstrução axiomática da geometria abstrata por David Hilbert, a aritmética proposta por Giuseppe Peano e a exploração intuitiva da teoria geral dos conjuntos, por Georg Cantor, que também produziu a iluminada teoria dos números transfinitos. O relacionamento entre lógica e matemática foi profundamente investigado por Alfred North Whitehead e Bertrand Russell, que em "Principia Mathematica" (1910) demonstraram ser a lógica um instrumento adequado para a representação formal de grande parte da matemática.

Um passo muito importante foi dado em 1930, em estudos simultâneos, porém independentes, realizados pelo alemão Kurt Gödel e o francês Jacques Herbrand. Ambos, em suas dissertações de doutorado, demonstraram que o mecanismo de prova do cálculo de predicados poderia oferecer uma prova formal de toda proposição logicamente verdadeira. O resultado de maior impacto foi entretanto produzido por Gödel, em 1931, com a descoberta do "teorema da incompleteza dos sistemas de formalização da aritmética". A prova deste teorema se baseava nos denominados *paradoxos de auto-referência* (declarações do tipo: "Esta sentença é falsa", que não podem ser provadas nem verdadeiras

¹ Na realidade, de uma certa *classe* de pensamento correto.

nem falsas). Em 1934, Alfred Tarski produziu a primeira teoria semântica rigorosamente formal do cálculo de predicados, introduzindo conceitos precisos para "satisfatibilidade", "verdade" (em uma dada interpretação), "consequência lógica" e outras noções relacionadas. Ainda na década de 30, diversos outros estudos - entre os quais os de Alan Turing, Alonzo Church e outros - aproximaram muito o cálculo de predicados da forma com que é hoje conhecido e estudado.

No início da Segunda Guerra Mundial, em 1939, toda a fundamentação teórica básica da lógica computacional estava pronta. Faltava apenas um meio prático para realizar o imenso volume de computações necessárias aos procedimentos de prova. Apenas exemplos muito simples podiam ser resolvidos manualmente. O estado de guerra deslocou a maior parte dos recursos destinados à pesquisa teórica, nos EUA, Europa e Japão para as técnicas de assassinato em massa. Foi somente a partir da metade dos anos 50 que o desenvolvimento da então novíssima tecnologia dos computadores conseguiu oferecer aos pesquisadores o potencial computacional necessário para a realização de experiências mais significativas com o cálculo de predicados.

Em 1958, uma forma simplificada do cálculo de predicados denominada *forma clausal* começou a despertar o interesse dos estudiosos do assunto. Tal forma empregava um tipo particular muito simples de sentença lógica denominada *cláusula*. Uma cláusula é uma (possivelmente vazia) disjunção de literais. Também por essa época, Dag Prawitz (1960) propôs um novo tipo de operação sobre os objetos do cálculo de predicados, que mais tarde veio a ser conhecida por *unificação*. A unificação se revelou fundamental para o desenvolvimento de sistemas simbólicos e de programação em lógica.

A programação em lógica em sistemas computacionais, entretanto, somente se tornou realmente possível a partir da pesquisa sobre prova automática de teoremas, particularmente no desenvolvimento do *Princípio da Resolução* por J. A. Robinson (1965). Um dos primeiros trabalhos relacionando o Princípio da Resolução com a programação de computadores deve-se a Cordell C. Green (1969) que mostrou como o mecanismo para a extração de respostas em sistemas de resolução poderia ser empregado para sintetizar programas convencionais.

A expressão "programação em lógica" (logic programming, originalmente em inglês) é devido a Robert Kowalski (1974) e designa o uso da lógica como linguagem de programação de computadores. Kowalski identificou, em um particular procedimento de prova de teoremas, um procedimento computacional, permitindo uma interpretação procedimental da lógica e estabelecendo as condições que nos permitem entendê-la como uma linguagem de programação de uso geral. Este foi um avanço essencial, necessário para adaptar os conceitos relacionados com a prova de teoremas às técnicas computacionais já dominadas pelos programadores. Aperfeiçoamentos realizados nas técnicas de implementação também foram de grande importância para o emprego da lógica como linguagem de programação. O primeiro interpretador experimental foi desenvolvido por um grupo de pesquisadores liderados por Alain Colmerauer na Universidade de Aix-Marseille (1972) com o nome de Prolog, um acrônimo para "Programmation en Logique". Seguindo-se a este primeiro passo, implementações mais "práticas" foram desenvolvidas por Battani e Meloni (1973), Bruynooghe (1976) e, principalmente, David H. D. Warren, Luís Moniz Pereira e outros pesquisadores da Universidade de Edimburgo (U.K.) que, em 1977, formalmente definiram o sistema hoje denominado "Prolog de Edimburgo", usado como referência para a maioria das atuais implementações da linguagem Prolog. Deve-se também a Warren a especificação da WAM (Warren Abstract Machine), um modelo formal empregado até hoje na pesquisa de arquiteturas computacionais orientadas à programação em lógica.

1.2 PROGRAMAÇÃO EM LÓGICA

Uma das principais idéias da programação em lógica é de que um algoritmo é constituído por dois elementos disjuntos: a lógica e o controle. O componente lógico corresponde à definição do que deve ser solucionado, enquanto que o componente de controle estabelece como a solução pode ser obtida. O programador precisa somente descrever o componente lógico de um algoritmo, deixando o controle

da execução para ser exercido pelo sistema de programação em lógica utilizado. Em outras palavras, a tarefa do programador passa a ser simplesmente a *especificação* do problema que deve ser solucionado, razão pela qual as linguagens lógicas podem ser vistas simultaneamente como linguagens para especificação formal e linguagens para a programação de computadores.

Um *programa em lógica* é então a representação de determinado problema ou situação expressa através de um conjunto finito de um tipo especial de sentenças lógicas denominadas *cláusulas*. Ao contrário de programas em Pascal ou C, um programa em lógica *não é* a descrição de um procedimento para se obter a solução de um problema. Na realidade o sistema utilizado no processamento de programas em lógica é inteiramente responsável pelo procedimento a ser adotado na sua execução. Um programa em lógica pode também ser visto alternativamente como uma base de dados, exceto que as bases de dados convencionais descrevem apenas *fatos* tais como "Oscar é um avestruz", enquanto que as sentenças de um programa em lógica possuem um alcance mais genérico, permitindo a representação de *regras* como em "Todo avestruz é um pássaro", o que não possui correspondência em bases de dados convencionais. Na figura abaixo se procura explicitar as principais diferenças entre programação convencional e programação em lógica.

PROGRAMAS CONVENCIONAIS	PROGRAMAS EM LÓGICA
Processamento Numérico	Processamento Simbólico
Soluções Algorítmicas	Soluções Heurísticas
Estruturas de Controle e Conhecimento Integradas	Estruturas de Controle e Conhecimento Separadas
Difícil Modificação	Fácil Modificação
Somente Respostas Totalmente Corretas	Incluem Respostas Parcialmente Corretas
Somente a Melhor Solução Possível	Incluem Todas as Soluções Possíveis

Figura 1.1 Programas Convencionais x Programas em Lógica

O paradigma fundamental da programação em lógica é o da *programação declarativa*, em oposição à programação procedimental típica das linguagens convencionais. A programação declarativa engloba também a programação funcional, cujo exemplo mais conhecido é a linguagem Lisp. Lembrando entretanto que Lisp data de 1960, a programação funcional é um estilo conhecido há bastante tempo, ao contrário da programação em lógica, que só ganhou ímpeto a partir dos anos 80, quando foi escolhida como a linguagem básica do projeto japonês para o desenvolvimento dos denominados computadores de quinta geração. O ponto focal da programação em lógica consiste em identificar a noção de *computação* com a noção de *dedução* . Mais precisamente, os sistemas de programação em lógica reduzem a execução de programas à pesquisa da refutação das sentenças do programa em conjunto com a negação da sentença que expressa a consulta, seguindo a regra: "*uma refutação é a dedução de uma contradição*".

Pode-se então expressar conhecimento (programas e/ou dados) em Prolog por meio de cláusulas de dois tipos: fatos e regras². Um fato denota uma verdade incondicional, enquanto que as regras definem as condições que devem ser satisfeitas para que uma certa declaração seja considerada verdadeira. Como fatos e regras podem ser utilizados conjuntamente, nenhum componente dedutivo adicional precisa ser utilizado. Além disso, como regras recursivas e não-determinismo são permitidos, os programadores podem obter descrições muito claras, concisas e não-redundantes da informação que desejam representar. Como não há distinção entre argumentos de entrada e de saída, qualquer combinação de argumentos pode ser empregada.

Os termos "programação em lógica" e "programação Prolog" tendem a ser empregados indistintamente. Deve-se, entretanto, destacar que a linguagem Prolog é apenas uma particular abordagem da programação em lógica. As características mais marcantes dos sistemas de programação em lógica em geral - e da linguagem Prolog em particular - são as seguintes:

² Ver o Apêndice A para uma abordagem mais formal.

- **Especificações são Programas:** A linguagem de especificação é entendida pela máquina e é, por si só, uma linguagem de programação. Naturalmente, o refinamento de especificações é mais efetivo do que o refinamento de programas. Um número ilimitado de cláusulas diferentes pode ser usado e predicados (procedimentos) com qualquer número de argumentos são possíveis. Não há distinção entre o programa e os dados. As cláusulas podem ser usadas com grande vantagem sobre as construções convencionais para a representação de tipos abstratos de dados. A adequação da lógica para a representação simultânea de programas e suas especificações a torna um instrumento especialmente útil para o desenvolvimento de ambientes e protótipos.
- **Capacidade Dedutiva:** O conceito de computação confunde-se com o de (passo de) inferência. A execução de um programa é a prova do teorema representado pela consulta formulada, com base nos axiomas representados pelas cláusulas (fatos e regras) do programa.
- **Não-determinismo:** Os procedimentos podem apresentar múltiplas respostas, da mesma forma que podem solucionar múltiplas e aleatoriamente variáveis condições de entrada. Através de um mecanismo especial, denominado "backtracking", uma sequência de resultados alternativos pode ser obtida.
- **Reversibilidade das Relações:** (Ou "computação bidirecional"). Os argumentos de um procedimento podem alternativamente, em diferentes chamadas representar ora parâmetros de entrada, ora de saída. Os procedimentos podem assim ser projetados para atender a múltiplos propósitos. A execução pode ocorrer em qualquer sentido, dependendo do contexto. Por exemplo, o mesmo procedimento para inserir um elemento no topo de uma pilha qualquer pode ser usado, em sentido contrário, para remover o elemento que se encontrar no topo desta pilha.
- **Tríplice Interpretação dos Programas em Lógica:** Um programa em lógica pode ser semanticamente interpretado de três modos distintos: (1) por meio da semântica declarativa, inerente à lógica, (2) por meio da semântica procedimental, onde as cláusulas dos programas são vistas como entrada para um método de prova e, (3) por meio da semântica operacional, onde as cláusulas são vistas como comandos para um procedimento particular de prova por refutação. Essas três interpretações são intercambiáveis segundo a particular abordagem que se mostrar mais vantajosa ao problema que se tenta solucionar.
- **Recursão:** A recursão, em Prolog, é a forma natural de ver e representar dados e programas. Entretanto, na sintaxe da linguagem não há laços do tipo "for" ou "while" (apesar de poderem ser facilmente programados), simplesmente porque eles são absolutamente desnecessários. Também são dispensados comandos de atribuição e, evidentemente, o "goto". Uma estrutura de dados contendo variáveis livres pode ser retornada como a saída de um procedimento. Essas variáveis livres podem ser posteriormente instanciadas por outros procedimentos produzindo o efeito de atribuições implícitas a estruturas de dados. Onde for necessário, variáveis livres são automaticamente agrupadas por meio de referências transparentes ao programador. Assim, as variáveis lógicas um potencial de representação significativamente maior do que oferecido por operações de atribuição e referência nas linguagens convencionais.

A premissa básica da programação em lógica é portanto que "computação é inferência controlada". Tal visão da computação tem se mostrado extremamente produtiva, na medida em que conduz à idéia de que computadores podem ser projetados com a arquitetura de máquinas de inferência. Grande parte da pesquisa sobre computação paralela, conduzida hoje nos EUA, Europa e Japão, emprega a programação em lógica como instrumento básico para a especificação de novas arquiteturas de hardware e o desenvolvimento de máquinas abstratas não-convencionais.

1.3 APLICAÇÕES

Um dos primeiros usos da programação em lógica foi a representação e análise de subconjuntos da linguagem natural. Esta foi inclusive a aplicação que motivou Alain Colmerauer a desenvolver a pri-

meira implementação da linguagem Prolog. Logo em seguida, outros pesquisadores da área da inteligência artificial propuseram diversas novas aplicações para o novo instrumento. Alguns dos primeiros trabalhos com Prolog envolviam a formulação de planos e a escrita de compiladores, por Pereira e Warren (1977), prova de teoremas em geometria por R. Welhan (1976) e a solução de problemas de mecânica, por Bundy *et al.* (1979). As aplicações relatadas desde então, multiplicaram-se velozmente.

Concentraremos aqui a atenção em um conjunto das principais áreas investigadas com o concurso da programação em lógica.

- **Sistemas Baseados em Conhecimento (SBCs):** Ou *knowledge-based systems*, são sistemas que aplicam mecanismos automatizados de raciocínio para a representação e inferência de conhecimento. Tais sistemas costumam ser identificados como simplesmente "de inteligência artificial aplicada" e representam uma abrangente classe de aplicações da qual todas as demais seriam aproximadamente subclasses. A tecnologia dos SBCs foi identificada na Inglaterra pelo Relatório Alvey (1982) como uma das quatro tecnologias necessárias à completa exploração dos computadores de quinta geração. As outras seriam: interface homem-máquina (MMI), integração de circuitos em ultra-grande escala (ULSI) e engenharia de software (SE). O relacionamento entre SBCs e a nova geração de computadores é, na verdade, altamente simbiótica, cada uma dessas áreas é necessária para a realização do completo potencial da outra.
- **Sistemas de Bases de Dados (BDs):** Uma particularmente bem definida aplicação dos SBCs são bases de dados. BDs convencionais tradicionalmente manipulam dados como coleções de relações armazenadas de modo extensional sob a forma de tabelas. O modelo relacional serviu de base à implementação de diversos sistemas fundamentados na álgebra relacional, que oferece operadores tais como junção e projeção. O processador de consultas de uma BD convencional deriva, a partir de uma consulta fornecida como entrada, alguma conjunção específica de tais operações algébricas que um programa gerenciador então aplica às tabelas visando a recuperação de conjuntos de dados (n-tuplas) apropriados, se existirem. O potencial da programação em lógica para a representação e consulta à BDs foi simultaneamente investigado, em 1978, por van Emden, Kowalski e Tärnlund. As três pesquisas estabeleceram que a recuperação de dados - um problema básico em BDs convencionais - é intrínseca ao mecanismo de inferência dos interpretadores lógicos. Desde então diversos sistemas tem sido propostos para a representação de BDs por meio de programas em lógica.
- **Sistemas Especialistas (SEs):** Um sistema especialista é uma forma de SBC especialmente projetado para emular a especialização humana em algum domínio específico. Tipicamente um SE irá possuir uma *base de conhecimento* (BC) formada de fatos, regras e *heurísticas* sobre o domínio, juntamente com a capacidade de entabular comunicação interativa com seus usuários, de modo muito próximo ao que um especialista humano faria. Além disso os SEs devem ser capazes de oferecer sugestões e conselhos aos usuários e, também, melhorar o próprio desempenho a partir da experiência, isto é, adquirir novos conhecimentos e heurísticas com essa interação. Diversos sistemas especialistas foram construídos com base na programação em lógica, como por exemplo o sistema ORBI, para a análise de recursos ambientais desenvolvido por Pereira *et al.* na Universidade Nova de Lisboa.
- **Processamento da Linguagem Natural (PLN):** O PLN é da maior importância para o desenvolvimento de ferramentas para a comunicação homem-máquina em geral e para a construção de interfaces de SBCs em particular. A implementação de sistemas de PLN em computadores requer não somente a formalização sintática, como também - o grande problema - a formalização semântica, isto é, o correto *significado* das palavras, sentenças, frases, expressões, etc. que povoam a comunicação natural humana. O uso da lógica das cláusulas de Horn³ para este propósito foi inicialmente investigado por Colmerauer, o próprio criador do Prolog (1973), e posteriormente por Kowalski (1974). Ambos mostraram (1) que as cláusulas de Horn eram adequadas à representação de qualquer *gramática livre-de-contexto* (GLC), (2) permitiam que ques-

³ Assim denominadas em homenagem a Alfred Horn, que primeiro lhes estudou as propriedades, em 1951.

tões sobre a estrutura de sentenças em linguagem natural fossem formuladas como objetivos ao sistema, e (3) que diferentes procedimentos de prova aplicados a representações lógicas da linguagem natural correspondiam a diferentes estratégias de análise.

- **Educação:** A programação em lógica poderá vir a oferecer no futuro uma contribuição bastante significativa ao uso educacional de computadores. Esta proposta foi testada em 1978 quando Kowalski introduziu a programação em lógica na Park House Middle School em Wimbledon, na Inglaterra, usando acesso on-line aos computadores do Imperial College. O sucesso do empreendimento conduziu a um projeto mais abrangente denominado "Lógica como Linguagem de Programação para Crianças", inaugurado em 1980 na Inglaterra com recursos do Conselho de Pesquisa Científica daquele país. Os resultados obtidos desde então tem mostrado que a programação em lógica não somente é assimilada mais facilmente do que as linguagens convencionais, como também pode ser introduzida até mesmo a crianças na faixa dos 10 a 12 anos, as quais ainda se beneficiam do desenvolvimento do pensamento lógico-formal que o uso de linguagens como o Prolog induz.
- **Arquiteturas Não-Convencionais:** Esta área vem se tornando cada vez mais um campo extremamente fértil para o uso da programação em lógica especialmente na especificação e implementação de máquinas abstratas de processamento paralelo. O paralelismo pode ser modelado pela programação em lógica em variados graus de atividade se implementado em conjunto com o mecanismo de unificação. Duas implementações iniciais nesse sentido foram o *Parlog*, desenvolvido em 1984 por Clark e Gregory, e o *Concurrent Prolog* (CP), por Shapiro em 1983. O projeto da Quinta Geração, introduzido na próxima seção, foi fortemente orientado ao uso da programação em lógica em sistemas de processamento paralelo.

Muitas outras aplicações poderiam ainda ser citadas, principalmente na área da inteligência artificial, que tem no Prolog e no Lisp as suas duas linguagens mais importantes. Novas tecnologias de hardware e software tais como sistemas massivamente paralelos, redes de computadores, assistentes inteligentes, bases de dados semânticas, etc., tornam o uso do Prolog (e de outras linguagens baseadas em lógica) cada vez mais atraentes

1.4 A QUINTA GERAÇÃO

Em 1979 o governo japonês iniciou estudos para um novo, ambicioso e único projeto na área da computação normalmente denominado *Sistemas Computacionais de Quinta Geração* cujo objetivo principal era o desenvolvimento, no espaço de uma década, de hardware e software de alto desempenho, caracterizando uma nova geração de computadores. O projeto iniciou em 1982 e foi oficialmente encerrado em maio de 1992. Muito foi dito e escrito sobre o projeto, que produziu inúmeros resultados e diversos subprodutos ao longo desses dez anos. Um de seus principais méritos, entretanto, parece ter sido chamar a atenção da comunidade científica mundial para as potencialidades da lógica como linguagem de programação de computadores. Sistemas de processamento lógico paralelo derivados do Prolog foram desenvolvidos para servir como linguagens-núcleo (kernel languages) dos novos equipamentos que seriam produzidos a partir dos resultados do projeto. Considerado um sucesso por seus dirigentes, o projeto foi entretanto criticado por não haver conseguido colocar as tecnologias desenvolvidas à disposição do grande público. Em outras palavras: ainda não dispomos hoje (1994) de microcomputadores pessoais de quinta geração - denominados máquinas PSI (Personal Sequential Inference machines) - comercialmente viáveis para o grande público. Os resultados teóricos obtidos e os protótipos construídos foram entretanto de grande valia para que num futuro próximo isso venha a ser possível. Nestas novas máquinas o papel da linguagem assembly será desempenhado por um dialeto do Prolog orientado ao processamento paralelo.

Um relatório sobre o projeto, organizado por Ehud Shapiro e David Warren em 1993, reuniu as opiniões de diversos pesquisadores dele participantes, entre os quais Kazuhiro Fuchi, seu líder, Robert Kowalski, Koichi Furukawa, Kazunori Ueda e outros. Todos os depoimentos foram unânimes em

declarar que os objetivos do projeto foram plenamente atingidos. Na Figura 1.2 é mostrada uma adaptação em português do diagrama "de intenções" apresentado por Fuchi, no *Fifth Generation Computer Systems Congress* de 1981 (FGCS'81), o congresso que deu a conhecer ao mundo um dos mais ambiciosos projetos da história da computação.

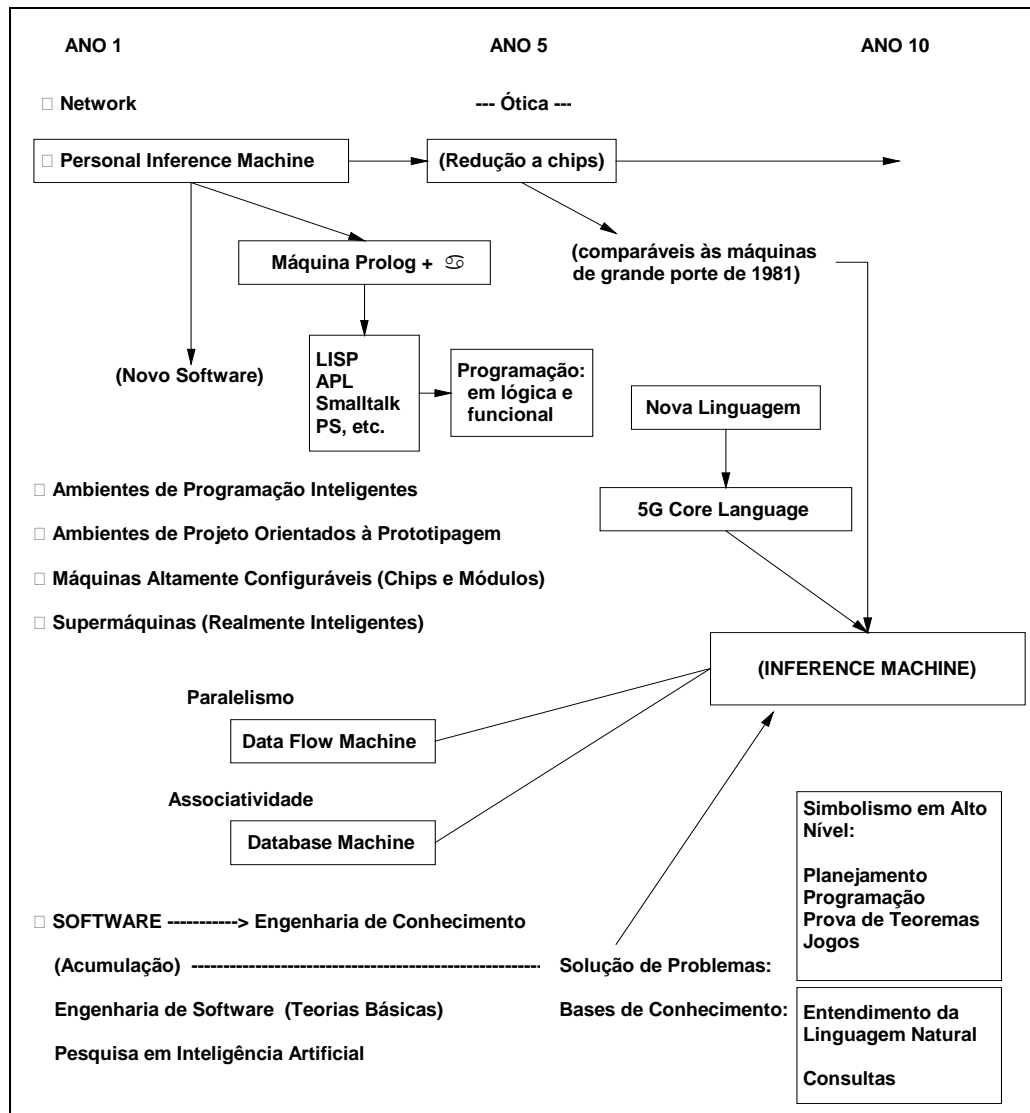


Figura 1.2 Diagrama Conceitual do Projeto do Computador de Quinta Geração

Segundo o relatório de Shapiro e Warren, um dos primeiros passos do projeto consistiu em definir uma linguagem de programação em lógica que ao mesmo tempo fosse adequada ao paralelismo do hardware e aos requisitos sofisticados especificados para o software. Baseada no Parlog e no Concurrent Prolog, uma equipe de pesquisadores liderada por Kazunori Ueda desenvolveu a linguagem GHC (Guarded Horn Clauses), que deu origem à KL0 (Kernel Language Zero). Um refinamento dessa versão⁴, realizado pela equipe de Takashi Chikayama produziu, em 1987, a linguagem KL1. Todos os sub-projetos do FGCS foram revistos para trabalhar com essa linguagem. Em 1988 os primeiros protótipos do computador de quinta geração foram construídos, recebendo o nome genérico de *Parallel Inference Machines* (PIMs). Tais computadores possuíam arquitetura massivamente paralela e tinham velocidade de processamento calculada em MLIPS (milhões de inferências lógicas por segundo). Uma dessas máquinas, denominada Multi-PSI foi apresentada com grande sucesso no FGCS'88.

⁴ Uma versão distribuída a grupos selecionados de usuários para teste e depuração.

A linguagem KL1 foi empregada para escrever o sistema operacional PIMOS (Parallel Inference Machine Operating System), em 1988. É importante ressaltar aqui que a linguagem KL1 é uma linguagem de muito alto nível⁵ e, ao mesmo tempo, uma *linguagem de máquina*, isto é, adequada à programação a nível de registradores, posições de memória e portas lógicas. As versões mais recentes do PIMOS provam definitivamente que KL1 (agora já KL2) é uma linguagem muito mais adequada do que as linguagens convencionais para a construção de software básico em máquinas paralelas. Outras linguagens de programação foram - e ainda vem sendo - pesquisadas. Por exemplo, uma linguagem de programação em lógica com restrições denominada GDCC foi projetada em um nível ainda mais alto que a KL1. Uma outra linguagem, denominada "Quixote" foi produzida para lidar com bases de dados dedutivas e orientadas a objetos. Para o gerenciamento de sistemas paralelos distribuídos foi especificada a linguagem Kappa-P. Todas essas linguagens, com as quais - ou com seus dialetos - todos certamente estaremos em contato num futuro próximo, estão baseadas nos conceitos e resultados da pesquisa em programação em lógica.

Tecnicamente considera-se que o projeto atingiu a primeira parte de seus objetivos: diversos computadores paralelos foram construídos. Tais computadores são denominados coletivamente de *máquinas de inferência paralela* (PIMs), incorporam a linguagem KL1 e o sistema operacional PIMOS. Além disso as máquinas PIM mais recentemente construídas lograram atingir um pico de desempenho da ordem de 1 *gigalips* (1 bilhão de inferências lógicas por segundo), o que era um dos objetivos concretos do projeto considerados mais difíceis de atingir.

A segunda parte do projeto, entretanto, a construção de máquinas orientadas à bases de dados (database machines) foi menos claramente abordada. Tal objetivo foi reformulado a partir do sucesso obtido com a construção de linguagens de programação em lógica concorrente para a construção de implementações baseadas em KL1 na mesma plataforma de hardware das máquinas PIM.

De um modo geral, entretanto, considera-se que o projeto demonstrou ser a tecnologia PIM bem sucedida em novas aplicações envolvendo paralelismo em diversas áreas, especialmente computação não-numérica e inteligência artificial. Em suma, segundo o relatório Shapiro-Warren:

"(...) uma ponte foi construída entre a computação paralela e as aplicações envolvendo inteligência artificial. Entretanto, as duas extremidades finais da ponte ainda se encontram por concluir e a ponte em si é mais frágil do que poderia ter sido. É sem dúvida ainda muito cedo para se esperar que a ponte seja inaugurada recebendo uma grande aclamação."

1.5 PORQUE ESTUDAR PROLOG

Normalmente há um *gap* de 10 a 20 anos entre o estágio básico de uma pesquisa tecnológica e o momento em que esta é colocada à disposição da sociedade consumidora. Na área de informática esse intervalo costuma ser menor, entretanto, estamos assistindo a uma completa transformação: do paradigma da quarta geração, ora em fase de esgotamento⁶ para arquiteturas inovadoras, contemplando sistemas de processamento paralelo, a concorrência de processos e *layers* baseados em lógica. A grande explosão da informática atualmente persegue conceitos tais como interoperabilidade, conectividade, orientação a objetos, sistemas multimídia, agentes inteligentes cooperativos, hiperdocumentos, realidade virtual, inteligência de máquina e outros, cuja evolução irá determinar nos próximos anos uma mudança tão radical quanto foi a das carruagens para os veículos automotores - mais ainda, segundo alguns autores, - terminando por transformar completamente a própria estrutura social.

A programação Prolog é uma excelente porta de entrada para a informática do futuro, tendo em vista

⁵ Quanto mais alto o nível de uma linguagem, mais próxima da linguagem natural ela se encontra.

⁶ As atuais tecnologias de integração de circuitos (VLSI/ULSI) tendem a atingir os limites físicos além dos quais se tornam economicamente inviáveis.

que, entre outras vantagens:

- (1) É de aprendizado muito mais fácil e natural do que as linguagens procedimentais convencionais, podendo inclusive ser ministrada a estudantes entre o final do primeiro e o início do segundo grau com grande aproveitamento;
- (2) Implementa com precisão todos os novos modelos surgidos nos últimos anos, inclusive redes neurais, algoritmos genéticos, sociedades de agentes inteligentes, sistemas concorrentes e paralelos;
- (3) Permite a implementação de extensões, inclusive em nível *meta*, e a definição precisa de *sistemas reflexivos* (essenciais, por exemplo, à robótica);
- (4) Libera o programador dos problemas associados ao controle de suas rotinas, permitindo-lhe concentrar-se nos aspectos lógicos da situação a representar.

Tem sido observada a tendência de substituição paulatina no mercado de trabalho dos serviços de *programação* pelos de *especificação*. Isso ocorre por várias razões, dentre elas porque as especificações podem ser formalmente *provadas* corretas, o que não ocorre com facilidade nos programas convencionais. Essa transição - da *arte* de programar à *ciência* de especificar - vem estimulando o aparecimento de linguagens como o Prolog, que pode ser visto como sendo simultaneamente uma linguagem de programação e de especificação (ou, como querem alguns, como uma linguagem de especificações diretamente executáveis em computadores).

Vem também ocorrendo aceleradamente a popularização de ambientes e interfaces cada vez mais próximos do usuário final e oferecendo recursos muito poderosos para a personalização de programas de acordo com as preferências individuais. Isso permite supor que, num futuro próximo, qualquer pessoa, mesmo sem formação específica em programação, poderá interagir facilmente com computadores, em níveis muito elevados⁷, dispensando em grande parte a programação, tal como é hoje conhecida. Por outro lado, a *construção* de tais ambientes irá depender de profissionais bem mais preparados do que um programador em Pascal, por exemplo. Deverão, tais profissionais, possuir um currículo muito mais rico, abrangendo a teoria da computação, lógica matemática, álgebra relacional, filosofia, arquiteturas concorrentes e paralelas, etc. Serão necessários entretanto em número muito maior do que se imaginava no início dos anos 80, quando essa tendência ainda não se apresentava perfeitamente delineada, uma vez que praticamente todo software colocado no mercado deverá ser produzido a partir de suas especificações formais.

Um último motivo - não menos importante que os demais já apresentados - deve ainda ser considerado: A *expressividade* herdada da lógica torna a linguagem Prolog um instrumento especialmente poderoso, adequado para a descrição do mundo real com todos os seus contornos, nuances e sutilezas. Nos poucos casos em que a representação se torna mais difícil - na representação temporal, por exemplo - a flexibilidade do Prolog em aceitar o desenvolvimento de extensões semanticamente precisas e incorporá-las ao seu mecanismo de produção de inferências, remove qualquer impedimento para o seu emprego em virtualmente qualquer área do conhecimento.

RESUMO

- A programação em lógica, tal como a conhecemos hoje, tem suas raízes no cálculo de predicados, proposto por Frege em 1879. Diversos estudos posteriores foram de grande importância para sua evolução, com destaque para as investigações de Herbrand, Gödel, Tarski, Prawitz, Robinson e Green;
- A primeira implementação da linguagem Prolog foi realizada por Alain Colmerauer e sua equi-

⁷ Ao nível da linguagem coloquial falada ou escrita, por exemplo.

pe, na Universidade de Aix-Marseille em 1972. A formalização semântica da programação com cláusulas de Horn é devida a Kowalski (1974) e a especificação do primeiro "standard" - o Prolog de Edimburgo - foi realizada por Warren e Pereira em 1977;

- As principais características que diferenciam os *programas em lógica* dos programas convencionais são as seguintes:
 - (1) Processamento simbólico,
 - (2) Soluções heurísticas,
 - (3) Estruturas de controle e conhecimento separadas,
 - (4) Fácil modificação,
 - (5) Incluem respostas parcialmente corretas, e
 - (6) Incluem todas as soluções possíveis;
- Além disso, os sistemas de programação em lógica em geral e a linguagem Prolog em particular possuem as seguintes propriedades:
 - (1) Funcionam simultaneamente como linguagem de programação e de especificação,
 - (2) Possuem capacidade dedutiva,
 - (3) Operam de forma não-determinística,
 - (4) Permitem a representação de relações reversíveis,
 - (5) Permitem interpretação declarativa, procedimental e operacional, e
 - (6) São naturalmente recursivos;
- As principais aplicação da programação em lógica são:
 - (1) Sistemas Baseados em Conhecimento (SBCs),
 - (2) Sistemas de Bases de Dados (BDs),
 - (3) Sistemas Especialistas (SEs),
 - (4) Processamento da Linguagem Natural (PLN),
 - (5) Educação, e
 - (6) Modelagem de Arquiteturas Não-Convencionais;
- O projeto japonês para o desenvolvimento de Sistemas Computacionais de Quinta Geração iniciou em 1982 e foi oficialmente concluído em maio de 1992. Apesar de ficarem aquém do esperado, os resultados produzidos permitem claramente antever o papel preponderante que a programação em lógica deverá representar nos futuros sistemas computacionais;
- A crescente necessidade de garantir a qualidade do software substituindo programas por especificações formais diretamente executáveis, aliada à evolução das características do hardware, que passam a explorar cada vez mais os conceitos de concorrência e paralelismo, tornam a linguagem Prolog uma excelente porta de entrada para a informática do futuro.

2. A LINGUAGEM PROLOG

A principal utilização da linguagem Prolog reside no domínio da programação simbólica, não-numérica, sendo especialmente adequada à solução de problemas, envolvendo objetos e relações entre objetos. O advento da linguagem Prolog reforçou a tese de que a lógica é um formalismo conveniente para representar e processar conhecimento. Seu uso evita que o programador descreva os procedimentos necessários para a solução de um problema, permitindo que ele expresse declarativamente apenas a sua estrutura lógica, através de fatos, regras e consultas. Algumas das principais características da linguagem Prolog são:

- É uma linguagem orientada ao processamento simbólico;
- Representa uma implementação da lógica como linguagem de programação;
- Apresenta uma semântica declarativa inerente à lógica;
- Permite a definição de *programas reversíveis*, isto é, programas que não distinguem entre os argumentos de entrada e os de saída;
- Permite a obtenção de respostas alternativas;
- Suporta código recursivo e iterativo para a descrição de processos e problemas, dispensando os mecanismos tradicionais de controle, tais como *while*, *repeat*, etc;
- Permite associar o processo de especificação ao processo de codificação de programas;
- Representa programas e dados através do mesmo formalismo;
- Incorpora facilidades computacionais extralógicas e metalógicas.

No presente capítulo introduz-se informalmente os conceitos essenciais da linguagem Prolog, visando conduzir rapidamente o leitor ao domínio da sintaxe e a um entendimento intuitivo da semântica associada aos programas.

2.1 FATOS

Considere a árvore genealógica mostrada na Figura 2.1. É possível definir, entre os objetos (indivíduos) mostrados, uma relação denominada *progenitor* que associa um indivíduo a um dos seus progenitores. Por exemplo, o fato de que João é um dos progenitores de José pode ser denotado por:

```
progenitor(joão, josé).
```

onde *progenitor* é o nome da relação e *joão* e *josé* são os seus argumentos. Por razões que se tornarão claras mais tarde, escreve-se aqui nomes de pessoas (como João) iniciando com letra minúscula. A relação *progenitor* completa, como representada na figura acima pode ser definida pelo seguinte programa Prolog:

```
progenitor(maria, josé).  
progenitor(joão, josé).  
progenitor(joão, ana).  
progenitor(josé, júlia).  
progenitor(josé, íris).  
progenitor(íris, jorge).
```

O programa acima compõe-se de seis *cláusulas*, cada uma das quais denota um fato acerca da relação *progenitor*. Se o programa for submetido a um sistema Prolog, este será capaz de responder algumas questões sobre a relação ali representada. Por exemplo: "*José é o progenitor de Íris?*". Uma consulta como essa deve ser formulada ao sistema precedida por um "?-". Esta combinação de sinais denota que se está formulando uma pergunta. Como há um fato no programa declarando explicitamente que

José é o progenitor de Íris, o sistema responde "sim".

```
?-progenitor(josé, íris).  
sim
```

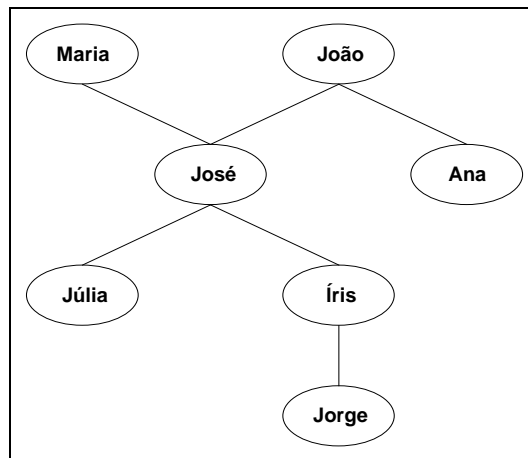


Figura 2.1 Uma árvore genealógica

Uma outra questão poderia ser: "Ana é um dos progenitores de Jorge?". Nesse caso o sistema responde "não", porque não há nenhuma cláusula no programa que permita deduzir tal fato.

```
?-progenitor(ana, jorge).  
não
```

A questão "Luís é progenitor de Maria?" também obteria a resposta "não", porque o programa nem sequer conhece alguém com o nome Luís.

```
?-progenitor(luís, maria).  
não
```

Perguntas mais interessantes podem também ser formuladas, por exemplo: "Quem é progenitor de Íris?". Para fazer isso introduz-se uma variável, por exemplo "X" na posição do argumento correspondente ao progenitor de Íris. Desta feita o sistema não se limitará a responder "sim" ou "não", mas irá procurar (e informar caso for encontrado) um valor de X que torne a assertiva "X é progenitor de Íris" verdadeira.

```
?-progenitor(X, íris).  
X=josé
```

Da mesma forma a questão "Quem são os filhos de José?" pode ser formulada com a introdução de uma variável na posição do argumento correspondente aos filhos de José. Note que, neste caso, mais de uma resposta verdadeira pode ser encontrada. O sistema irá fornecer a primeira que encontrar e aguardar manifestação por parte do usuário. Se este desejar outras soluções deve digitar um ponto-e-vírgula (;), do contrário digita um ponto (.), o que informa ao sistema que a solução fornecida é suficiente.

```
?-progenitor(josé, X).  
X=júlia;  
X=íris;  
não
```

Aqui a última resposta obtida foi "não" significando que todas as soluções válidas já foram fornecidas. Uma questão mais geral para o programa seria: "Quem é progenitor de quem?" ou, com outra formulação: "Encontre X e Y tal que X é progenitor de Y". O sistema, em resposta, irá fornecer (enquanto se desejar, digitando ";") todos os pares progenitor-filho até que estes se esgotem (quando então responde "não") ou até que se resolva encerrar a apresentação de novas soluções (digitando "."). No exemplo a seguir iremos nos satisfazer com as três primeiras soluções encontradas.

```
?-progenitor(X, Y).
```



```

X=maria Y=josé;
X=joão Y=josé;
X=joão Y=ana.

```

Pode-se formular questões ainda mais complicadas ao programa, como "*Quem são os avós de Jorge?*". Como nosso programa não possui diretamente a relação *avô*, esta consulta precisa ser dividida em duas etapas, como pode ser visto na Figura 2.2. A saber:

(1) Quem é progenitor de Jorge? (Por exemplo, Y) e

(2) Quem é progenitor de Y? (Por exemplo, X)

Esta consulta em Prolog é escrita como uma sequência de duas consultas simples, cuja leitura pode ser: "*Encontre X e Y tais que X é progenitor de Y e Y é progenitor de Jorge*".

```

?-progenitor(X, Y), progenitor(Y, jorge).
X=josé Y=íris

```

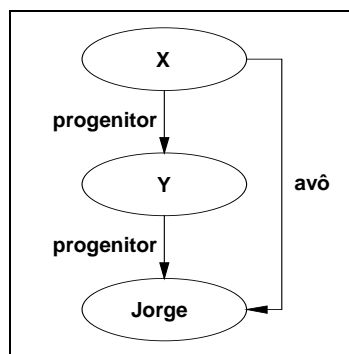


Figura 2.2 A relação *avô* em função de *progenitor*

Observe que se mudarmos a ordem das consultas na composição, o significado lógico permanece o mesmo, apesar do resultado ser informado na ordem inversa:

```

?-progenitor(Y, jorge), progenitor(X, Y).
Y=íris X=josé

```

De modo similar podemos perguntar: "*Quem é neto de João?*":

```

?-progenitor(joão, X), progenitor(X, Y).
X=josé Y=júlia;
X=josé Y=íris.

```

Ainda uma outra pergunta poderia ser: "*José e Ana possuem algum progenitor em comum?*". Novamente é necessário decompor a questão em duas etapas, formulando-a alternativamente como: "*Encontre um X tal que X seja simultaneamente progenitor de José e Ana*".

```

?-progenitor(X, josé), progenitor(X, ana).
X=joão

```

Por meio dos exemplos apresentados até aqui acredita-se ter sido possível ilustrar os seguintes pontos:

- Uma relação como *progenitor* pode ser facilmente definida em Prolog estabelecendo-se as *tu-
plas* de objetos que satisfazem a relação;
- O usuário pode facilmente consultar o sistema Prolog sobre as relações definidas em seu programa;
- Um programa Prolog é constituído de *cláusulas*, cada uma das quais é encerrada por um ponto (.);
- Os argumentos das relações podem ser objetos concretos (como júlia e íris) ou objetos genéricos (como X e Y). Objetos concretos em um programa são denominados *átomos*, enquanto que os objetos genéricos são denominados *variáveis*;

- Consultas ao sistema são constituídas por um ou mais *objetivos*, cuja seqüência denota a sua conjunção;
- Uma resposta a uma consulta pode ser *positiva* ou *negativa*, dependendo se o objetivo correspondente foi alcançado ou não. No primeiro caso dizemos que a consulta foi *bem-sucedida* e, no segundo, que a consulta *falhou*;
- Se várias respostas satisfizerem a uma consulta, então o sistema Prolog irá fornecer tantas quantas forem desejadas pelo usuário.

2.2 REGRAS

O programa da árvore genealógica pode ser facilmente ampliado de muitas maneiras interessantes. Inicialmente vamos adicionar informação sobre o sexo das pessoas ali representadas. Isso pode ser feito simplesmente acrescentando os seguintes fatos ao programa:

```
masculino(joão).
masculino(josé).
masculino(jorge).

feminino(maria).
feminino(júlia).
feminino(ana).
feminino(íris).
```

As relações introduzidas no programa são *masculino* e *feminino*. Tais relações são unárias, isto é, possuem um único argumento. Uma relação binária, como *progenitor*, é definida entre pares de objetos, enquanto que as relações unárias podem ser usadas para declarar propriedades simples desses objetos. A primeira cláusula unária da relação *masculino* pode ser lida como: "*João é do sexo masculino*". Poderia ser conveniente declarar a mesma informação presente nas relações unárias *masculino* e *feminino* em uma única relação binária *sexo*:

```
sexo(joão, masculino).
sexo(maria, feminino).
sexo(josé, masculino). ... etc.
```

A próxima extensão ao programa será a introdução da relação *filho* como o inverso da relação *progenitor*. Pode-se definir a relação *filho* de modo semelhante à utilizada para definir a relação *progenitor*, isto é fornecendo uma lista de fatos, cada um dos quais fazendo referência a um par de pessoas tal que uma seja filho da outra. Por exemplo:

```
filho(josé, joão).
```

Entretanto podemos definir a relação "*filho*" de uma maneira muito mais elegante, fazendo o uso do fato de que ela é o inverso da relação *progenitor* e esta já está definida. Tal alternativa pode ser baseada na seguinte declaração lógica:

```
Para todo X e Y
  Y é filho de X se
  X é progenitor de Y.
```

Essa formulação já se encontra bastante próxima do formalismo adotado em Prolog. A cláusula correspondente, com a mesma leitura acima, é:

```
filho(Y, X) :- progenitor(X, Y).
```

que também pode ser lida como: "*Para todo X e Y, se X é progenitor de Y, então Y é filho de X*".

Cláusulas Prolog desse tipo são denominadas *regras*. Há uma diferença importante entre regras e fatos. Um fato é *sempre* verdadeiro, enquanto regras especificam algo que "*pode ser verdadeiro se algumas condições forem satisfeitas*".

- Uma parte de *condição* (o lado direito da cláusula).

O símbolo ":-" significa "se" e separa a cláusula em *conclusão*, ou *cabeça* da cláusula, e *condição* ou *corpo* da cláusula, como é mostrado no esquema abaixo. Se a condição expressa pelo corpo da cláusula - *progenitor*(X, Y) - é verdadeira então, segue como consequência lógica que a cabeça - *filho*(Y, X) - também o é. Por outro lado, se não for possível demonstrar que o corpo da cláusula é verdadeiro, o mesmo irá se aplicar à cabeça.

```
filho(Y, X) :- progenitor(X, Y)
```

A maioria dos sistemas Prolog, na ausência de caracteres ASCII adequados, emprega o símbolo composto ":-" para denotar a implicação "→". Aqui, por uma questão de clareza, adotaremos este último símbolo, que é o normalmente empregado na programação em lógica com cláusulas definidas.

A utilização das regras pelo sistema Prolog é ilustrada pelo seguinte exemplo: vamos perguntar ao programa se José é filho de Maria:

```
?-filho(josé, maria).
```

Não há nenhum fato a esse respeito no programa, portanto a única forma de considerar esta questão é aplicando a regra correspondente. A regra é genérica, no sentido de ser aplicável a quaisquer objetos X e Y. Logo pode ser aplicada a objetos particulares, como José e Maria. Para aplicar a regra, Y será substituído por José e X por Maria. Dizemos que as variáveis X e Y se tornaram *instanciadas* para:

```
X=maria e Y=josé
```

A parte de condição se transformou então no objetivo *progenitor*(maria, José). Em seguida o sistema passa a tentar verificar se essa condição é verdadeira. Assim o objetivo inicial, *filho*(José, maria), foi substituído pelo sub-objetivo *progenitor*(maria, José). Esse novo objetivo apresenta-se como trivial, uma vez que há um fato no programa estabelecendo exatamente que Maria é um dos progenitores de José. Isso significa que a parte de condição da regra é verdadeira, portanto a parte de conclusão também é verdadeira e o sistema responde "sim".

Vamos agora adicionar mais algumas relações ao nosso programa. A especificação, por exemplo, da relação *mãe* entre dois objetos do nosso domínio pode ser escrita baseada na seguinte declaração lógica:

```
Para todo X e Y
  X é mãe de Y se
  X é progenitor de Y e
  X é feminino.
```

que, traduzida para Prolog, conduz à seguinte regra:

```
mãe(X, Y) :- progenitor(X, Y), feminino(X).
```

onde a vírgula entre as duas condições indica a sua *conjunção*, significando que, para satisfazer o corpo da regra, ambas as condições devem ser verdadeiras. A relação *avô*, apresentada anteriormente na Figura 2.2, pode agora ser definida em Prolog por:

```
avô(X, Z) :- progenitor(X, Y), progenitor(Y, Z).
```

Neste ponto é interessante comentar alguma coisa sobre o *layout* dos programas Prolog. Estes podem ser escritos quase que com total liberdade, de modo que podemos inserir espaços e mudar de linha onde e quando melhor nos aprouver. Em geral, porém, desejamos produzir programas de boa aparência, elegantes e sobretudo fáceis de ser lidos. Com essa finalidade, normalmente se prefere escrever a cabeça da cláusula e os objetivos da condição cada um em uma nova linha. Para destacar a conclusão, indentamos os objetivos. A cláusula *avô*, por exemplo, seria escrita:

```
avô(X, Z) :-
    progenitor(X, Y),
    progenitor(Y, Z).
```

Adicionaremos ainda uma última relação ao nosso programa para exemplificar mais uma particularidade da linguagem Prolog. Uma cláusula para a relação *irmã* se embasaria na seguinte declaração lógica:

```
Para todo X e Y
  X é irmã de Y se
  X e Y possuem um progenitor comum e
  X é do sexo feminino.
```

Ou, sob a forma de regra Prolog:

```
irmã(X, Y) :-
  progenitor(Z, X),
  progenitor(Z, Y),
  feminino(X).
```

Deve-se atentar para a forma sob a qual o requisito "*X e Y possuem um progenitor comum*" foi expressa. A seguinte formulação lógica foi adotada: "*Algum Z deve ser progenitor de X e esse mesmo Z deve também ser progenitor de Y*". Uma forma alternativa, porém menos elegante, de representar a mesma condição seria: "*Z1 é progenitor de X e Z2 é progenitor de Y e Z1 é igual a Z2*". Se consultarmos o sistema com "*Júlia é irmã de Íris?*", obteremos, como é esperado, um "*sim*" como resposta. Poderíamos então concluir que a relação *irmã*, conforme anteriormente definida, funciona corretamente, entretanto, há uma falha muito sutil que se revela quando perguntamos: "*Quem é irmã de Íris?*". O sistema irá nos fornecer *duas* respostas:

```
?-irmã(X, iris).
X=júlia;
X=iris
```

dando a entender que Íris é irmã de si própria. Isso não é certamente o que se tinha em mente na definição de *irmã*, entretanto, de acordo com a regra formulada, a resposta obtida pelo sistema é perfeitamente lógica. Nossa regra sobre irmãs não menciona que X e Y não devem ser os mesmos para que X seja irmã de Y. Como isso não foi requerido, o sistema, com toda razão, assume que X e Y podem denotar a mesma pessoa e irá achar que toda pessoa do sexo feminino que possui um progenitor é irmã de si própria.

Para corrigir esta distorção é necessário acrescentar a condição de que X e Y devem ser diferentes. Isso pode ser feito de diversas maneiras, conforme se verá mais adiante. Por enquanto vamos assumir que uma relação *diferente(X, Y)* seja reconhecida pelo sistema como verdadeira se e somente se X e Y não forem iguais. A regra para a relação *irmã* fica então definida por:

```
irmã(X, Y) :-
  progenitor(Z, X),
  progenitor(Z, Y),
  feminino(X),
  diferente(X, Y).
```

Os pontos mais importantes vistos na presente seção foram:

- Programas Prolog podem ser ampliados pela simples adição de novas cláusulas;
- As cláusulas Prolog podem ser de três tipos distintos: fatos, regras e consultas;
- Os fatos declaram coisas que são incondicionalmente verdadeiras;
- As regras declaram coisas que podem ser ou não verdadeiras, dependendo da satisfação das condições dadas;
- Por meio de consultas podemos interrogar o programa acerca de que coisas são verdadeiras;
- As cláusulas Prolog são constituídas por uma cabeça e um corpo. O corpo é uma lista de objetivos separados por vírgulas que devem ser interpretadas como conjunções;
- Fatos são cláusulas que só possuem cabeça, enquanto que as consultas só possuem corpo e as regras possuem cabeça e corpo;

- Ao longo de uma computação, uma variável pode ser substituída por outro objeto. Dizemos então que a variável está instanciada;
- As variáveis são assumidas como universalmente quantificadas nas regras e nos fatos e existencialmente quantificadas nas consultas

2.3 CONSTRUÇÕES RECURSIVAS

Iremos adicionar agora ao programa a relação *antepassado*, que será definida a partir da relação *progenitor*. A definição necessita ser expressa por meio de duas regras, a primeira das quais definirá os antepassados diretos (imediatos) e a segunda os antepassados indiretos. Dizemos que um certo X é antepassado indireto de algum Z se há uma cadeia de progenitura entre X e Z como é ilustrado na Figura 2.3. Na árvore genealógica da Figura 2.1, João é antepassado direto de Ana e antepassado indireto de Júlia.

A primeira regra, que define os antepassados diretos, é bastante simples e pode ser formulada da seguinte maneira:

Para todo X e Z
 X é antepassado de Z se
 X é progenitor de Z.

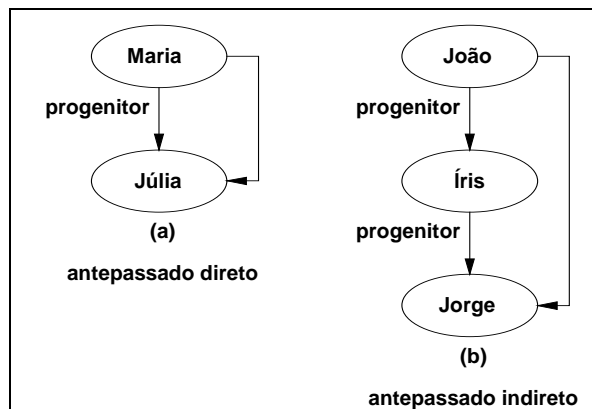


Figura 2.3 Exemplos da relação *antepassado*

ou, traduzindo para Prolog:

```
antepassado(X, Z) :-
    progenitor(X, Z).
```

Por outro lado, a segunda regra é mais complicada, porque a cadeia de progenitores poderia se estender indefinidamente. Uma primeira tentativa seria escrever uma cláusula para cada posição possível na cadeia. Isso conduziria a um conjunto de cláusulas do tipo:

```
antepassado(X, Z) :-
    progenitor(X, Y),
    progenitor(Y, Z).
antepassado(X, Z) :-
    progenitor(X, Y1),
    progenitor(Y1, Y2),
    progenitor(Y2, Z).
antepassado(X, Z) :-
    progenitor(X, Y1),
    progenitor(Y1, Y2),
    progenitor(Y2, Y3),
    progenitor(Y3, Z). ... etc.
```

Isso conduziria a um programa muito grande e que, de qualquer modo, somente funcionaria até um determinado limite, isto é, somente forneceria antepassados até uma certa profundidade na árvore

genealógica de uma família, porque a cadeia de pessoas entre o antepassado e seu descendente seria limitada pelo tamanho da maior cláusula definindo essa relação. Há entretanto uma formulação elegante e correta para a relação *antepassado* que não apresenta qualquer limitação. A idéia básica é definir a relação em termos de si própria, empregando um estilo de programação em lógica denominado *recursivo*:

```
Para todo X e Z
  X é antepassado de Z se
    existe um Y tal que
      X é progenitor de Y e
      Y é antepassado de Z.
```

A cláusula Prolog correspondente é:

```
antepassado(X, Z) :-
  progenitor(X, Y),
  antepassado(Y, Z).
```

Assim é possível construir um programa completo para a relação *antepassado* composto de duas regras: uma para os antepassados diretos e outra para os indiretos. Reescrevendo as duas juntas tem-se:

```
antepassado(X, Z) :-
  progenitor(X, Z).
antepassado(X, Z) :-
  progenitor(X, Y),
  antepassado(Y, Z).
```

Tal definição pode causar certa surpresa, tendo em vista a seguinte pergunta: Como é possível ao definir alguma coisa empregar essa mesma coisa se ela ainda não está completamente definida? Tais definições são denominadas recursivas e do ponto de vista da lógica são perfeitamente corretas e inteligíveis, o que deve ficar claro, pela observação da Figura 2.4. Por outro lado o sistema Prolog deve muito do seu potencial de expressividade à capacidade intrínseca que possui de utilizar facilmente definições recursivas. O uso de recursão é, em realidade, uma das principais características herdadas da lógica pela linguagem Prolog.

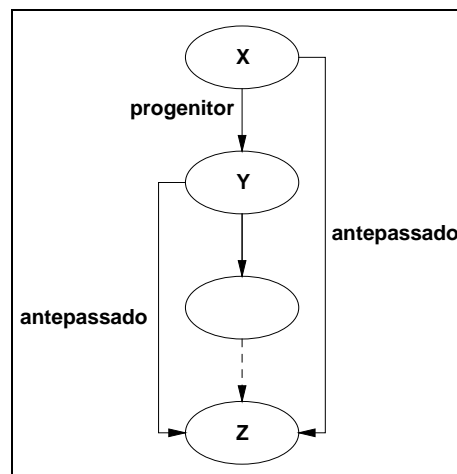


Figura 2.4 Formulação recursiva da relação *antepassado*

Há ainda uma questão importante a ser respondida: Como *realmente* o sistema Prolog utiliza o programa para encontrar as informações procuradas? Uma explicação informal será fornecida na próxima seção, antes porém vamos reunir todas as partes do programa que foi sendo gradualmente ampliado pela adição de novos fatos e regras. A forma final do programa é mostrada na Figura 2.5. O programa ali apresentado define diversas relações: *progenitor*, *masculino*, *feminino*, *antepassado*, etc. A relação *antepassado*, por exemplo, é definida por meio de duas cláusulas. Dizemos que cada uma delas é *sobre* a relação *antepassado*. Algumas vezes pode ser conveniente considerar o conjunto completo de cláusulas sobre a mesma relação. Tal conjunto de cláusulas é denominado um *predicado*.

Na Figura 2.5, as duas regras sobre a relação antepassado foram distinguidas com os nomes [pr1] e [pr2] que foram adicionados como comentários ao programa. Tais nomes serão empregados adiante como referência a essas regras. Os comentários que aparecem em um programa são normalmente ignorados pelo sistema Prolog, servindo apenas para melhorar a legibilidade do programa impresso. Os comentários se distinguem do resto do programa por se encontrarem incluídos entre os delimitadores especiais "/*" e "*/". Um outro método, mais conveniente para comentários curtos, utiliza o caracter de percentual "%": todo o texto informado entre o "%" e o final da linha é interpretado como comentário. Por exemplo:

```

/* Isto é um comentário. */
% E isto também.

progenitor(maria, josé).    % Maria é progenitor de José.
progenitor(joão, josé).
progenitor(joão, ana).
progenitor(josé, júlia).
progenitor(josé, íris).
progenitor(íris, jorge).
masculino(joão).           % João é do sexo masculino.
masculino(josé).
masculino(jorge).
feminino(maria).           % Maria é do sexo feminino.
feminino(ana).
feminino(júlia).
feminino(íris).
filho(Y, X) :-             % Y é filho de X se
    progenitor(X,Y).        % X é progenitor de Y.
mãe(X,Y) :-                % X é mãe de Y se
    progenitor(X, Y),       % X é progenitor de Y e
    feminino(X).            % X é do sexo feminino.
avô(X, Z) :-               % X é avô de Z se
    progenitor(X, Y),       % X é progenitor de Y e
    progenitor(Y, Z).       % Y é progenitor de Z.
irmã(X, Y) :-              % X é irmã de Y se
    progenitor(Z, X),       % X tem um progenitor, Z que
    progenitor(Z, Y),       % é também progenitor de Y e
    feminino(X),            % X é do sexo feminino e
    diferente(X, Y).        % X e Y são diferentes.
antepassado(X, Z) :-       % X é antepassado de Z se
    progenitor(X, Z).        % X é progenitor de Z.           [pr1]
antepassado(X, Z) :-       % X é antepassado de Z se
    progenitor(X, Y),       % X é progenitor de Y e
    antepassado(Y, Z).      % Y é antepassado de Z.           [pr2]

```

Figura 2.5 Um programa Prolog

2.4 CONSULTAS

Uma consulta em Prolog é sempre uma seqüência composta por um ou mais objetivos. Para obter a resposta, o sistema Prolog tenta satisfazer todos os objetivos que compõem a consulta, interpretando-os como uma conjunção. Satisfazer um objetivo significa demonstrar que esse objetivo é verdadeiro, assumindo que as relações que o implicam são verdadeiras no contexto do programa. Se a questão também contém variáveis, o sistema Prolog deverá encontrar ainda os objetos particulares que, atribuídos às variáveis, satisfazem a todos os sub-objetivos propostos na consulta. A particular instanciação das variáveis com os objetos que tornam o objetivo verdadeiro é então apresentada ao usuário. Se não for possível encontrar, no contexto do programa, nenhuma instanciação comum de suas variáveis que permita derivar algum dos sub-objetivos propostos então a resposta será "não".

Uma visão apropriada da interpretação de um programa Prolog em termos matemáticos é a seguinte: O sistema Prolog aceita os fatos e regras como um conjunto de axiomas e a consulta do usuário como um teorema a ser provado. A tarefa do sistema é demonstrar que o teorema pode ser provado com base nos axiomas representados pelo conjunto das cláusulas que constituem o programa. Essa visão

será ilustrada com um exemplo clássico da lógica de Aristóteles. Sejam os axiomas:

Todos os homens são falíveis.
Sócrates é um homem.

Um teorema que deriva logicamente desses dois axiomas é:

Sócrates é falível

O primeiro axioma pode ser reescrito como: "*Para todo X, se X é um homem então X é falível*". Nessa mesma linha o exemplo pode ser escrito em Prolog como se segue:

```
falível(X) :-  
    homem(X).  
  
homem(sócrates).  
  
?-falível(X).  
X=sócrates
```

Um exemplo mais complexo, extraído do programa apresentada na Figura 2.5 é:

?-antepassado(joão, íris).

Sabe-se que *progenitor(josé, íris)* é um fato. Usando esse fato e a regra [pr1], podemos concluir *antepassado(josé, íris)*. Este é um fato derivado. Não pode ser encontrado explícito no programa, mas pode ser derivado a partir dos fatos e regras ali presentes. Um *passo de inferência* como esse pode ser escrito em uma forma mais complexa como:

$\text{progenitor}(\text{josé}, \text{íris}) \Rightarrow \text{antepassado}(\text{josé}, \text{íris})$

que pode ser lido assim: "*de progenitor(josé, íris) segue, pela regra [pr1] que antepassado(josé, íris)*". Além disso sabemos que *progenitor(joão, josé)* é fato. Usando este fato e o fato derivado, *antepassado(josé, íris)*, podemos concluir, pela regra [pr2], que o objetivo proposto, *antepassado(joão, íris)* é verdadeiro. O processo completo, formado por dois passos de inferência, pode ser escrito:

$\text{progenitor}(\text{josé}, \text{íris}) \Rightarrow \text{antepassado}(\text{josé}, \text{íris})$

e

$\text{progenitor}(\text{joão}, \text{josé}) \text{ e } \text{antepassado}(\text{josé}, \text{íris}) \Rightarrow \text{antepassado}(\text{joão}, \text{íris})$

Mostrou-se assim o que pode ser uma seqüência de passos de inferência usada para satisfazer um objetivo. Tal seqüência denomina-se *seqüência de prova*. A extração de uma seqüência de prova do contexto formado por um programa e uma consulta é obtida pelo sistema na ordem inversa da empregada acima. Ao invés de iniciar a inferência a partir dos fatos, o Prolog começa com os objetivos e, usando as regras, substitui os objetivos correntes por novos objetivos até que estes se tornem fatos.

Dada por exemplo a questão: "*João é antepassado de Íris?*", o sistema tenta encontrar uma cláusula no programa a partir da qual o objetivo seja consequência imediata. Obviamente, as únicas cláusulas relevantes para essa finalidade são [pr1] e [pr2], que são sobre a relação *antepassado*, porque são as únicas cujas cabeças podem ser unificadas com o objetivo formulado. Tais cláusulas representam dois caminhos alternativos que o sistema pode seguir. Inicialmente o Prolog irá tentar a que aparece em primeiro lugar no programa:

$\text{antepassado}(X, Z) \text{ :- } \text{progenitor}(X, Z).$

uma vez que o objetivo é *antepassado(joão, íris)*, as variáveis na regra devem ser instanciadas por $X=\text{joão}$ e $Y=\text{íris}$. O objetivo inicial, *antepassado(joão, íris)* é então substituído por um novo objetivo:

$\text{progenitor}(\text{joão}, \text{íris})$

Não há, entretanto, nenhuma cláusula no programa cuja cabeça possa ser unificada com *progenitor(joão, íris)*, logo este objetivo *falha*. Então o Prolog retorna ao objetivo original (*backtracking*) para tentar um caminho alternativo que permita derivar o objetivo *antepassado(joão, íris)*. A regra [pr2] é então tentada:

$\text{antepassado}(X, Z) \text{ :- } \text{progenitor}(X, Y),$


```
antepassado(Y, Z).
```

Como anteriormente, as variáveis *X* e *Z* são instanciadas para *joão* e *íris*, respectivamente. A variável *Y*, entretanto, não está instanciada ainda. O objetivo original, `antepassado(joão, íris)` é então substituído por dois novos objetivos derivados por meio da regra [pr2]:

```
progenitor(joão, Y), antepassado(Y, íris).
```

Encontrando-se agora face a dois objetivos, o sistema tenta satisfazê-los na ordem em que estão formulados. O primeiro deles é fácil: `progenitor(joão, Y)` pode ser unificado com dois fatos do programa: `progenitor(joão, josé)` e `progenitor(joão, ana)`. Mais uma vez, o caminho a ser tentado deve corresponder à ordem em que os fatos estão escritos no programa. A variável *Y* é então instanciada com *josé* nos dois objetivos acima, ficando o primeiro deles imediatamente satisfeito. O objetivo remanescente é então:

```
antepassado(josé, íris).
```

Para satisfazer tal objetivo, a regra [pr1] é mais uma vez empregada. Essa segunda aplicação de [pr1], entretanto, nada tem a ver com a sua utilização anterior, isto é, o sistema Prolog usa um novo conjunto de variáveis na regra cada vez que esta é aplicada. Para indicar isso iremos *renomear* as variáveis em [pr1] nessa nova aplicação, da seguinte maneira:

```
antepassado(X', Z') :-  
    progenitor(X', Z').
```

A cabeça da regra deve então ser unificada como o nosso objetivo corrente, que é `antepassado(josé, íris)`. A instanciação de *X'* e *Y'* fica: *X'*=*josé* e *Y'*=*íris* e o objetivo corrente é substituído por:

```
progenitor(josé, íris)
```

Esse objetivo é imediatamente satisfeito, porque aparece no programa como um fato. O sistema encontrou então um caminho que lhe permite provar, no contexto oferecido pelo programa dado, o objetivo originalmente formulado, e portanto responde "sim".

2.5 O SIGNIFICADO DOS PROGRAMAS PROLOG

Assume-se que um programa Prolog possua três interpretações semânticas básicas. A saber: **interpretação declarativa, interpretação procedimental, e interpretação operacional**.

Na interpretação declarativa entende-se que as cláusulas que definem o programa descrevem uma *teoria de primeira ordem*. Na interpretação procedimental, as cláusulas são vistas como entrada para um método de prova. Finalmente, na interpretação operacional as cláusulas são vistas como comandos para um procedimento particular de prova por refutação.

Tais alternativas semânticas são valiosas em termos de entendimento e codificação de programas Prolog. A interpretação declarativa permite que o programador modele um dado problema através de assertivas acerca dos objetos do universo de discurso, simplificando a tarefa de programação Prolog em relação a outras linguagens tipicamente procedimentais como Pascal ou C. A interpretação procedimental permite que o programador identifique e descreva o problema pela redução do mesmo a subproblemas, através da definição de uma série de chamadas a procedimentos. Por fim, a interpretação operacional reintroduz a idéia de controle da execução (que é irrelevante do ponto de vista da semântica declarativa), através da ordenação das cláusulas e dos objetivos dentro das cláusulas em um programa Prolog. Essa última interpretação é semelhante à semântica operacional de muitas linguagens convencionais de programação, e deve ser considerada, principalmente em grandes programas, por questões de eficiência. É interessante notar que o programador pode comutar de uma interpretação para outra, produzindo um efeito sinérgico que facilita consideravelmente a codificação dos programas Prolog.

Essa habilidade específica do Prolog, de trabalhar em detalhes procedimentais de ação sobre o seu próprio domínio de definição, isto é, a capacidade de ser *meta-programado*, é uma das principais vantagens da linguagem. Ela encoraja o programador a considerar a semântica declarativa de seus programas de modo relativamente independente dos seus significados procedimental e operacional. Uma vez que os resultados do programa são considerados, em princípio, pelo seu significado declarativo, isto deveria ser, por decorrência, suficiente para a codificação de programas Prolog. Isso possui grande importância prática, pois os aspectos declarativos do programa são em geral mais fáceis de entender do que os detalhes operacionais. Para tirar vantagem dessa característica o programador deve se concentrar principalmente no significado declarativo e, sempre que possível, evitar os detalhes de execução. A abordagem declarativa, na realidade, torna a programação em Prolog mais fácil do que nas linguagens convencionais. Infelizmente, entretanto, essa interpretação nem sempre é suficiente. Como deverá ficar claro mais adiante, em problemas de maior complexidade os aspectos operacionais não podem ser ignorados. Apesar de tudo, a atribuição de significado declarativo aos programas Prolog deve ser estimulada, na extensão limitada por suas restrições de ordem prática.

RESUMO

- A programação em Prolog consiste em estabelecer relações entre objetos e em formular consultas sobre tais relações.
- Um programa Prolog é formado por cláusulas. Há três tipos de cláusulas: fatos ou *assertivas*, regras ou *procedimentos* e consultas;
- Uma relação pode ser especificada por meio de fatos, que estabelecem as tuplas de objetos que satisfazem a relação, por meio de regras, que estabelecem condições para a satisfação das relações, ou por meio de combinações de fatos e regras descrevendo a relação;
- Denomina-se *predicado* ao conjunto de fatos e regras empregados para descrever uma determinada relação;
- Interrogar um programa acerca de suas relações por meio de uma *consulta* corresponde a consultar uma base de conhecimento. A resposta do sistema Prolog consiste em um conjunto de objetos que satisfazem as condições originalmente estabelecidas pela consulta;
- Em Prolog, estabelecer se um objeto satisfaz a uma consulta é freqüentemente um problema de certa complexidade, que envolve inferência lógica e a exploração de caminhos alternativos em uma *árvore de busca* ou de *pesquisa*, com a possível utilização de mecanismos especiais de retorno (backtracking). Tudo isso é feito automaticamente pelo sistema, de forma transparente ao usuário;
- Três tipos de semântica são atribuídas aos programas Prolog: declarativa, procedimental e operacional. O programador deve empregá-las conforme o problema a ser resolvido, tirando proveito da situação apresentada.

EXERCÍCIOS

- 2.1 Amplie o programa apresentado na Figura 2.5 para representar as relações tio, prima, cunhado e sogra.
- 2.2 Programe a relação descendente(X, Y), onde X é descendente de Y.
- 2.3 Escreva um programa Prolog para representar o seguinte:
João nasceu em Pelotas e Jean nasceu em Paris.
Pelotas fica no Rio Grande do Sul.
Paris fica na França.
Só é gaúcho quem nasceu no Rio Grande do Sul.

2.4 Escreva um programa Prolog para representar o seguinte:

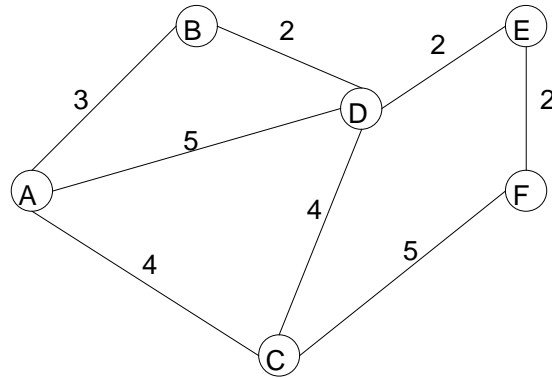
Os corpos celeste dignos de nota são as estrelas, os planetas e os cometas.

Vênus é um corpo celeste, mas não é uma estrela.

Os cometas possuem cauda quando estão perto do sol.

Vênus está perto do sol, mas não possui cauda.

2.5 Assuma que os arcos em um grafo expressem custos, como no exemplo abaixo:



e sejam descritos através de assertivas da forma

`arco(R, S, T)`

significando que há um arco de custo T entre os nodos R e S. Por exemplo, `arco(A, B, 3)` descreve um arco de custo 3 entre os nodos A e B. Assuma também que o relacionamento `mais(X, Y, Z)` vale quando $X+Y=Z$. Defina o relacionamento `custo(U, V, L)` de forma a expressar que existe um caminho de custo L entre os nodos U e V.

3. SINTAXE E SEMÂNTICA

Prolog é um nome comum para uma família de sistemas que implementam a lógica de predicados como linguagem de programação. Algumas destas implementações, como o Prolog de Edimburgo e o IC-Prolog, são bastante conhecidas nos meios acadêmicos. Outras, como o microProlog, o Quintus-Prolog e o Arity Prolog ganharam popularidade em diferentes segmentos. No presente texto se adota, visando maior clareza, uma sintaxe genérica, capaz de ser facilmente adaptada a qualquer ambiente Prolog.

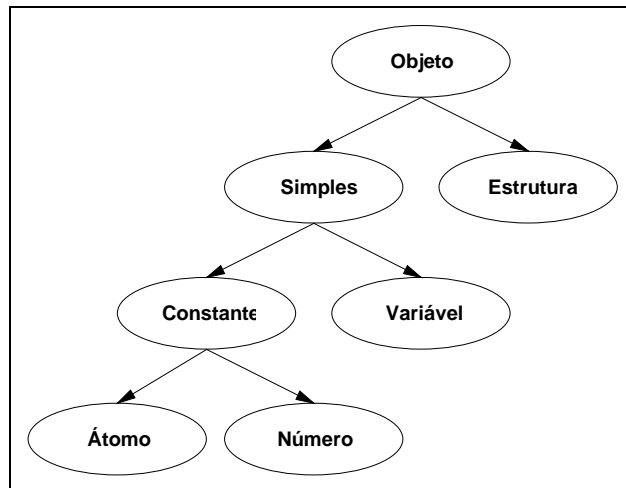


Figura 3.1 Classificação dos Objetos Prolog

3.1 OBJETOS

Na Figura 3.1 apresenta-se uma classificação dos objetos em Prolog. O sistema reconhece o tipo de um objeto no programa por meio de sua forma sintática. Isso é possível porque a sintaxe do Prolog especifica formas diferentes para cada tipo de objeto. Na sintaxe aqui adotada, comum à maioria das implementações, variáveis sempre irão iniciar com letras maiúsculas, enquanto que as constantes não-numéricas, ou átomos, iniciam com letras minúsculas. Nenhuma informação adicional, tal como *tipos* de dados precisa ser fornecida para que o sistema reconheça a informação com a qual está lidando.

3.1.1 ÁTOMOS E NÚMEROS

No capítulo anterior viu-se informalmente alguns exemplos simples de átomos e variáveis. Em geral, entretanto, estes podem assumir formas mais complexas. O alfabeto básico adotado aqui para a linguagem Prolog consiste dos seguintes símbolos:

- Pontuação: () . ' "
- Conetivos: , (conjunção)
; (disjunção)
:- (implicação)
- Letras: a, b, c, ..., z, A, B, C, ..., Z
- Dígitos: 0, 1, 2, ..., 9
- Especiais: + - * / < > = : _ ... etc.

Os *átomos* podem ser construídos de três maneiras distintas:

- a. Como cadeias de letras e/ou dígitos, podendo conter o caracter especial *sublinhado* (), iniciando obrigatoriamente com letra minúscula. Por exemplo:

```
socrates      x_y
nil           mostraMenu
x47           a_b_1_2
```

- b. Como cadeias de caracteres especiais. Por exemplo:

```
<----->   ::=   =/=   =====>   .'.   ++++
```

- c. Como cadeias de caracteres quaisquer, podendo inclusive incluir espaços em branco, desde que delimitados por apóstrofes ('). Por exemplo:

```
'D. Pedro I'
'representação de conhecimento'
'13 de outubro de 1993'
'Robert Kowalski'
```

Um certo cuidado é necessário na formação de átomos do tipo (b.) porque algumas cadeias de caracteres especiais podem possuir um significado pré definido para o sistema Prolog subjacente, como costumava acontecer, por exemplo, com as cadeias '== ' e '=| '.

Os números usados em Prolog compreendem os números inteiros e os números reais. A sintaxe dos números inteiros é bastante simples, como pode ser visto nos exemplos abaixo:

```
1 1812 0 -273
```

Nem todos os números inteiros podem ser representados em um computador, portanto o escopo de variação dos números inteiros está limitado a um intervalo entre algum menor e algum maior número, dependendo da implementação. Normalmente a variação permitida nas implementações correntes é suficiente para atender todas as necessidades do usuário.

O tratamento dos números reais também varia de implementação para implementação. Será adotada aqui a sintaxe natural e consagrada, que faz uso do ponto decimal explícito.

```
3.14159 0.000023 -273.16
```

Os números reais não são, na verdade, muito utilizados em programas Prolog típicos. A razão disso é que o Prolog é uma linguagem orientada ao processamento simbólico, não-numérico, em oposição às linguagens "devoradoras de números", como por exemplo o Fortran. Na computação simbólica, números inteiros são frequentemente empregados, por exemplo, para contar os itens em uma lista, mas a necessidade de números reais é bastante pequena, virtualmente inexistente.

3.1.2 VARIÁVEIS

Variáveis Prolog são cadeias de letras, dígitos e do caracter sublinhado (), devendo iniciar com este ou com uma letra *maiúscula*. O caracter " ", sozinho, representa uma *variável anônima*, isto é, sem interesse para um determinado procedimento. Exemplos de variáveis são:

```
X
Resultado
Objeto2
Lista_de_Associados
_var35
_194
_ (variável anônima)
```

O *escopo léxico* de nomes de variáveis é apenas uma cláusula. Isso quer dizer que, por exemplo, se o nome X25 ocorre em duas cláusulas diferentes, então ele está representando duas variáveis diferentes. Por outro lado, toda ocorrência de X25 dentro da mesma cláusula quer significar a mesma variável.

Essa situação é diferente para as constantes: o mesmo átomo sempre significa o mesmo objeto ao longo de todo o programa.

3.1.3 ESTRUTURAS

Objetos estruturados, ou simplesmente *estruturas*, são objetos que possuem vários componentes. Os próprios componentes podem, por sua vez, ser também estruturas. Por exemplo, uma data pode ser vista como uma estrutura com três componentes: dia, mês e ano. Mesmo que sejam formadas por diversos componentes as estruturas são tratadas no programa como objetos simples. Para combinar os componentes em uma estrutura é necessário empregar um *functor*. Um functor é um símbolo funcional (um nome de função) que permite agrupar diversos objetos em um único objeto estruturado. Um functor adequada ao exemplo dado é *data*, então a data correspondente a 13 de outubro de 1993, cuja estrutura está presente na Figura 3.2, pode ser escrita como:

`data(13, outubro, 1993)`

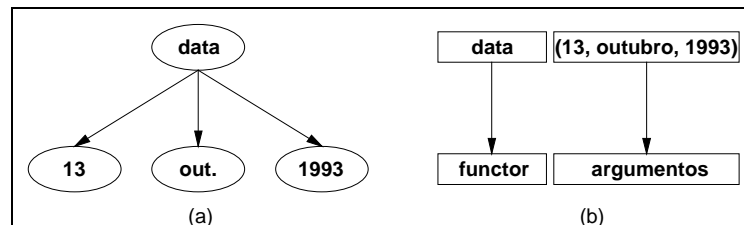


Figura 3.2 Uma data como exemplo de objeto estruturado

Na figura acima, em (a) temos a representação de data sob a forma de árvore e em (b) a forma como é escrita em Prolog. Todos os componentes no exemplo são constantes (dois inteiros e um átomo), entretanto, podem também ser variáveis ou outras estruturas. Um dia qualquer de março de 1996, por exemplo, pode ser representado por:

`data(Dia, março, 1996)`

Note que "Dia" é uma variável e pode ser instanciada para qualquer objeto em algum ponto da execução.

Sintaticamente todos os objetos em Prolog são denominados *termos*. O conjunto de *termos Prolog*, ou simplesmente *termos*, é o menor conjunto que satisfaz às seguintes condições:

- Toda constante é um termo;
- Toda variável é um termo;
- Se t_1, t_2, \dots, t_n são termos e f é um átomo, então $f(t_1, t_2, \dots, t_n)$ também é um termo, onde o átomo f desempenha o papel de um *símbolo funcional n -ário*. Diz-se ainda que a expressão $f(t_1, t_2, \dots, t_n)$ é um termo funcional Prolog.

Todos os objetos estruturados podem ser representados como árvores. A raiz da árvore é o functor e os ramos que dela partem são os argumentos ou componentes. Se algum dos componentes for também uma árvore, então ele passa a constituir uma sub-árvore do objeto estruturado completo. Por exemplo, na Figura 3.3 é mostrada a estrutura em árvore correspondente à expressão:

`(a + b) * (c - 5)`

De acordo com a sintaxe dos termos Prolog, anteriormente apresentada, e tomando os símbolos "+", "-" e "*" como funtores, a expressão dada pode ser escrita:

`*(+(a, b), -(c, 5))`

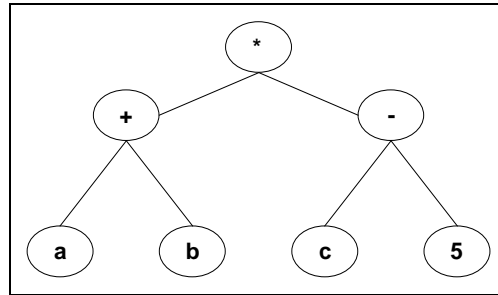


Figura 3.3 Uma expressão aritmética estruturada em árvore

Este é, naturalmente, um termo legal em Prolog, entretanto, não é a forma trivial com a qual estamos acostumados. Normalmente se irá preferir a notação usual, infixa, como é utilizada na matemática. Na verdade a linguagem Prolog admite as duas formas, prefixa e infixa, para a escrita de expressões aritméticas. Detalhes sobre operadores e definição de operadores especiais serão abordados mais adiante.

3.2 UNIFICAÇÃO

Na seção anterior foi visto como os objetos podem ser utilizados na representação de objetos de dados complexos. A operação mais importante entre dois termos Prolog é denominada *unificação*. A unificação pode, por si só, produzir alguns resultados interessantes. Dados dois termos, diz-se que eles *unificam* se:

- (1) Eles são idênticos, ou
- (2) As variáveis de ambos os termos podem ser instanciadas com objetos de maneira que, após a substituição das variáveis por esses objetos, os termos se tornam idênticos.

Por exemplo, os termos `data(D, M, 1994)` e `data(X, março, A)` unificam. Uma instanciação que torna os dois termos idênticos é:

D é instanciada com X;
M é instanciada com março;
A é instanciada com 1994.

Por outro lado, os termos `data(D, M, 1994)` e `data(X, Y, 94)` não unificam, assim como não unificam `data(X, Y, Z)` e `ponto(X, Y, Z)`. A unificação é um processo que toma dois termos como entrada e verifica se eles podem ser unificados. Se os termos não unificam, dizemos que o processo *falha*. Se eles unificam, então o processo é bem-sucedido e as variáveis dos termos que participam do processo são instanciadas com os valores encontrados para os objetos, de modo que os dois termos participantes se tornam idênticos. Vamos considerar novamente a unificação entre duas datas. O requisito para que essa operação se efetue é informada ao sistema Prolog pela seguinte consulta, usando o operador "=":

`?-data(D, M, 1994) = data(X, março, A)`

Já foi mencionada a instanciação `D=X`, `M=março` e `A=1994`, que obtém a unificação. Há, entretanto, outras instanciações que também tornam os termos idênticos. Duas delas são:

`D=1, X=1, M=março, A=1994`

`D=terceiro, X=terceiro, M=março, A=1994`

Essas duas instanciações são consideradas *menos gerais* do que a primeira, uma vez que restringem o valor das variáveis `D` e `X` mais fortemente do que seria necessário.. Para tornar os dois termos do exemplo idênticos, basta que `D` e `X` tenham o mesmo valor, seja qual for esse valor. A unificação em Prolog sempre resulta na instanciação *mais geral*, isto é, a que limita o mínimo possível o escopo de valores das variáveis, deixando a maior liberdade possível às instanciações posteriores. As regras

gerais que determinam se dois termos S e T unificam são as seguintes:

- Se S e T são constantes, então S e T unificam somente se ambos representam o mesmo objeto;
- Se S é uma variável e T é *qualquer coisa*, então S e T unificam com S instanciada com T. Inversamente, se T é uma variável, então T é instanciada com S;
- Se S e T são estruturas, unificam somente se: (1) S e T tem o mesmo functor principal, e (2) todos os seus componentes correspondentes também unificam. A instanciação resultante é determinada pela unificação dos componentes.

Essa última regra pode ser exemplificada pelo processo de unificação dos termos

```
triângulo(ponto(1, 1), A, ponto(2, 3))
```

com

```
triângulo(X, ponto(4, Y), ponto(2, Z))
```

cujas representação em árvore é apresentada na Figura 3.4.

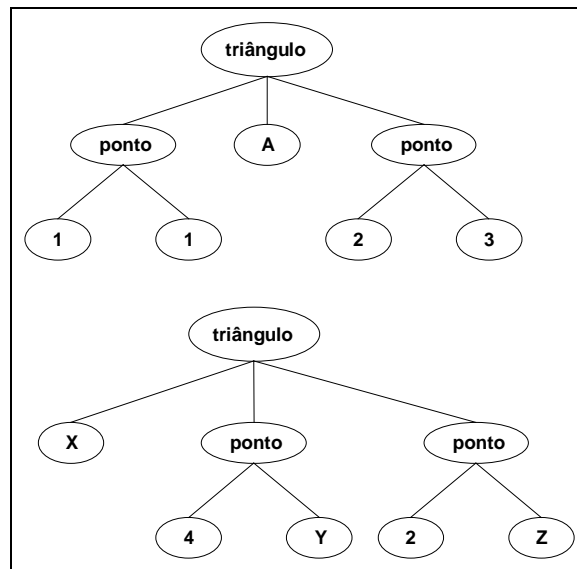


Figura 3.4 Termos representados em árvore

O processo de unificação começa pela raiz (o functor principal). Como ambos os funtores unificam, o processo parte para a unificação dos argumentos, onde a unificação dos pares de argumentos correspondentes ocorre. Assim o processo completo pode ser visto como a seguinte sequência de operações de unificação simples:

```
triângulo = triângulo
ponto(1, 1) = X
A = ponto(4, Y)
ponto(2, 3) = ponto(2, Z)
```

O processo completo de unificação é bem sucedido porque todas as unificações na sequência acima também o são. A instanciação resultante é:

```
X = ponto(1, 1)
A = ponto(4, Y)
Z = 3
```

3.3 SEMÂNTICA DECLARATIVA E SEMÂNTICA PROCEDIMENTAL

Conforme se estudou no capítulo anterior, os programas Prolog podem ser interpretados de três maneiras distintas: declarativamente, procedimentalmente e operacionalmente. Iremos agora aprofundar

um pouco tais idéias. Seja por exemplo a cláusula:

`P :- Q, R`

onde P, Q e R possuem a sintaxe de termos Prolog. Duas alternativas para a leitura declarativa dessa cláusula são:

`P é verdadeira se Q e R são verdadeiras`

e

`De Q e R segue P`

Por outro lado, duas leituras procedimentais alternativas são:

```
Para solucionar o problema P
  primeiro solucione o subproblema Q
  e depois solucione o subproblema R

Para satisfazer P, primeiro satisfaça Q e depois R
```

Assim a diferença entre as leituras declarativa e procedimental reside principalmente no fato que essa última não apenas define o relacionamento lógico existente entre a cabeça e o corpo da cláusula, como também exige a existência de uma *ordem* na qual os objetivos serão processados.

A semântica declarativa dos programas determina se um dado objetivo é verdadeiro e, se for, para que valores de variáveis isto se verifica. Para definir precisamente o significado declarativo precisamos introduzir o conceito de *instância de uma cláusula*. Uma instância de uma cláusula C é essa mesma cláusula C com cada uma de suas variáveis substituída por algum termo. Uma *variante* de uma cláusula C é uma instância dessa mesma cláusula C com cada uma de suas variáveis substituída por outra variável. Considere, por exemplo, a cláusula:

`temFilho(X) :- progenitor(X, Y).`

Duas variantes dela são:

```
temFilho(A) :- progenitor(A, B).
temFilho(João) :- progenitor(João, Alguém).
```

Duas instâncias dela são:

```
temFilho(joão) :- progenitor(joão, Alguém).
temFilho(sr(J)) :- progenitor(sr(J), jr(J)).
```

Assim, dado um programa e um objetivo G, o significado declarativo nos diz que:

"Um objetivo G é verdadeiro (isto é, é satisfatível ou segue logicamente do programa) se e somente se há uma cláusula C no programa e uma instância I de C tal que: (1) A cabeça de I é idêntica a G, e (2) todos os objetivos no corpo de I são verdadeiros."

Essa definição pode ser estendida para as consultas como se segue: Em geral uma consulta ao sistema Prolog é uma lista de objetivos separados por vírgulas. Uma lista de objetivos é verdadeira se *todos* os objetivos nela contidos são verdadeiros para alguma instanciação de suas variáveis. Os valores atribuídos às variáveis que tornam os objetivos da lista simultaneamente verdadeiros correspondem à sua *instanciação mais geral*.

Uma vírgula entre os objetivos significa a conjunção destes objetivos, isto é, *todos* devem ser satisfeitos. A linguagem Prolog também aceita a *disjunção* de objetivos: basta que um só dentre os objetivos da disjunção seja satisfeito para que todo o conjunto seja considerado satisfeito. A operação de disjunção é representada pelo ponto-e-vírgula (;). Por exemplo, a cláusula abaixo:

`P :- Q; R.`

é lida: P é verdadeiro se Q é verdadeiro ou R é verdadeiro. O significado da cláusula é portanto o mesmo que:

```
P :- Q.
P :- R.
```

A operação de conjunção é mais *forte* do que a disjunção, assim a cláusula:

$$P :- Q, R; S, T, U.$$

deve ser entendida como:

$$P :- (Q, R); (S, T, U).$$

e significa o mesmo que as cláusulas:

$$\begin{aligned} P &:- Q, R. \\ P &:- S, T, U. \end{aligned}$$

3.4 SEMÂNTICA OPERACIONAL

O significado operacional especifica como o Prolog responde as consultas que lhe são formuladas. Responder a uma consulta significa satisfazer uma lista de objetivos. Estes podem ser satisfeitos se as variáveis que neles ocorrem podem ser instanciadas de forma que eles possam ser consequência lógica do programa. Assim, o significado operacional do Prolog é o de um procedimento computacional para *executar* uma lista de objetivos com respeito a um dado programa. Com *executar* objetivos se quer significar *tentar satisfazê-los*. Considere o diagrama mostrado na Figura 3.5, representando tal procedimento, que denominaremos *executor*. Suas entradas e saídas são: (1) entrada: um programa e uma lista de objetivos; (2) saída: um indicador de sucesso/falha e instâncias de variáveis.

O significado dos resultados de saída do *executor* é o seguinte:

- O indicador de *sucesso/falha* tem o valor "sim" se os objetivos forem todos satisfeitos e "não" em caso contrário;
- As *instâncias* são produzidas somente no caso de conclusão bem-sucedida e correspondem aos valores das variáveis que satisfazem os objetivos.

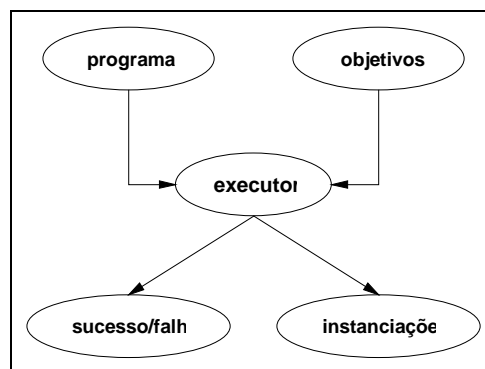


Figura 3.5 Procedimento de execução do sistema Prolog

RESUMO

Até aqui estudou-se um tipo de Prolog básico, denominado também de Prolog "puro". Esta denominação é devida ao fato de corresponder muito de perto à lógica de predicados de primeira ordem. Extensões cujo objetivo é adequar a linguagem a necessidades práticas serão estudadas mais adiante. Os pontos mais importantes do presente capítulo são:

- Objetos simples, em Prolog, são átomos, variáveis e números. Objetos estruturados, ou estruturas são empregados para representar entidades que possuem diversos componentes;
- As estruturas são construídas por meio de *functores*. Cada functor é definido por meio de seu *nome* e sua *aridade* ou número de argumentos;
- O *tipo* de um objeto é reconhecido exclusivamente através de sua forma sintática;

- O *escopo léxico* das variáveis em um programa é *uma* cláusula. O mesmo nome de variável em duas cláusulas distintas representa duas variáveis diferentes;
- As estruturas Prolog podem ser sempre representadas por meio de árvores. Prolog pode ser vista como uma linguagem orientada ao processamento de árvores;
- A operação de *unificação* toma dois termos e tenta torná-los idênticos por meio da *instanciação* das variáveis em ambos;
- Quando a unificação é bem sucedida, resulta na instanciação *mais geral* das variáveis envolvidas;
- A semântica *declarativa* do Prolog define se um objetivo é verdadeiro com relação a um dado programa e, se for, para que particulares instanciações de variáveis isto ocorre;
- Uma vírgula entre os objetivos significa a sua *conjunção*, enquanto que um ponto-e-vírgula significa a sua *disjunção*;
- A semântica *operacional* representa um procedimento para satisfazer a lista de objetivos no contexto de um dado programa. A saída desse procedimento é o valor-verdade da lista de objetivos com a respectiva instanciação de sua variáveis. O procedimento permite o retorno automático (backtracking) para o exame de novas alternativas;
- A interpretação declarativa de programas escritos em Prolog *puro* não depende da ordem das cláusulas nem da ordem dos objetivos dentro das cláusulas;
- A interpretação procedimental depende da ordem dos objetivos e cláusulas. Assim a ordem pode afetar a eficiência de um programa. Uma ordenação inadequada pode mesmo conduzir a chamadas recursivas infinitas;

EXERCÍCIOS

3.1 Quais dos seguintes objetos estão sintaticamente corretos e a que tipo de objeto pertencem?

- Daniela
- daniela
- 'Daniela'
- _daniela
- 'Daniela vai a Paris'
- vai(daniela, paris)
- 8118
- 2(X, Y)
- +(sul, oeste)
- três(Cavalos(Baios))

3.2 Sugira uma representação para retângulos, quadrados, círculos e elipses, usando uma abordagem similar à apresentada na Figura 3.4. Procure obter a representação mais geral possível, por exemplo, um quadrado é um caso especial de retângulo e um círculo pode ser considerado um caso especial de elipse.

3.3 Quais das próximas operações de unificação serão bem sucedidas e quais irão falhar? Para as que forem bem sucedidas, quais são as instanciações de variáveis resultantes?

- ponto(A, B) = ponto(1, 2)
- ponto(A, B) = ponto(X, Y, Z)
- mais(2, 2) = 4
- +(2, D) = +(E, 2)
- t(p(-1,0), P2, P3) = t(P1, p(1, 0), p(0, Y))

3.4 Defina uma representação Prolog para segmentos de reta no plano expressos em função dos

pontos limites. Que termo irá representar qualquer segmento de reta vertical em $X=5$?

3.5 Supondo que um retângulo seja representado pelo termo:

`retângulo(SupEsq, InfDir)`

onde SupEsq representa o ponto superior esquerdo e InfDir o ponto inferior direito de um retângulo em uma tela de vídeo (1280 x 1024), defina a relação

`quadrado(R, ...)`

que é verdadeira se R é um quadrado.

3.6 Considere o seguinte programa:

```
f(1, um).  
f(s(1), dois).  
f(s(s(1))), três).  
f(s(s(s(X))), N) :- f(X, N).
```

Como iria o sistema Prolog responder as seguintes questões? Quando várias respostas são possíveis, dê pelo menos duas:

- a. `?-f(s(1), A).`
- b. `?-f(s(s(1)), dois).`
- c. `?-f(s(s(s(s(s(s(1))))))), C).`
- d. `?-f(D, três).`

4. OPERADORES E ARITMÉTICA

4.1 OPERADORES

Na matemática costuma-se escrever expressões como

$$2*a + b*c$$

onde $+$ e $*$ são operadores e 2, a, b e c são argumentos. Em particular, $+$ e $*$ são denominados operadores *infixos* porque se localizam entre os dois argumentos que operam. Tais expressões são representadas por árvores como na Figura 4.1 e podem ser escritas, se for desejado, sob a forma de termos Prolog, com os símbolos $+$ e $*$ como funtores:

$$+(* (2, a), *(b, c))$$

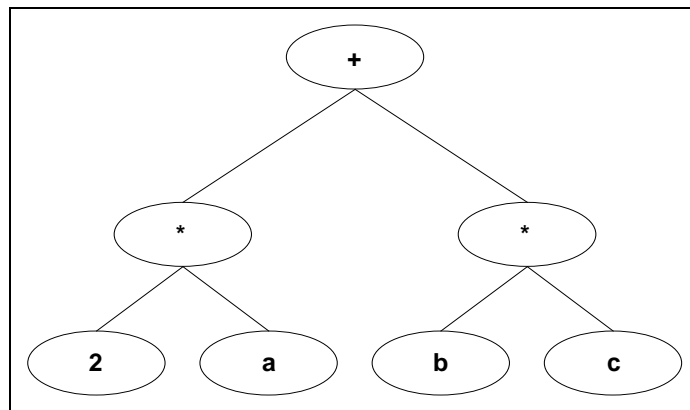


Figura 4.1 Representação em árvore da expressão $+(*(2, a), *(b, c))$

Normalmente, entretanto, é preferível escrever as expressões matemáticas na forma usual, com os operadores infixos, como em:

$$2*a + b*c$$

Tal notação é também aceita pelo Prolog, entretanto, trata-se apenas da representação externa deste objeto, que será automaticamente convertida para a forma convencional dos termos Prolog. Na saída, entretanto, o termo será novamente convertido para a forma externa, com os operadores infixos.

Assim, as expressões matemáticas são manipuladas pelo Prolog como meras extensões notacionais e nenhum novo princípio para a estruturação de objetos está sendo proposto. Se for escrito $a+b$, o sistema irá reconhecer e manipular tal expressão exatamente como se houvesse sido escrito $+(a, b)$. Para que o sistema entenda apropriadamente expressões tais como $a+b*c$, é necessário existir uma prioridade de execução entre os operadores. Assim o operador $+$ é executado prioritariamente ao operador $*$. É essa prioridade de execução que decide qual a interpretação correta da expressão. Por exemplo, a expressão $a+b*c$ poderia em princípio ser entendida como:

$$+(a, *(b, c)) \text{ ou } *+(a, b), c)$$

A regra geral é que o operador de maior prioridade seja o functor principal do termo. Se expressões contendo $+$ e $*$ devem ser entendidas segundo as convenções usuais, então $+$ deve ter maior precedência que $*$. Assim a expressão $a+b*c$ deve ser entendida como $a+(b*c)$. Se outra interpretação é pretendida, deve ser indicada explicitamente com o uso de parênteses, como em $(a+b)*c$.

O programador Prolog pode também definir os seus próprios operadores, isto é, definir átomos tais como *tem* e *suporta* como se fossem operadores infixos e então escrever no programa fatos como:

```
pedro tem informações
assoalho suporta mesa
```

que são exatamente equivalentes a

```
tem(pedro, informações)
suporta(assoalho, mesa)
```

A definição de novos operadores é realizada pela inserção no programa de um certo tipo especial de cláusulas, denominadas *diretivas*, que atuam como definidoras de operadores. Uma expressão definidora de um operador deve aparecer no programa antes de qualquer expressão que contenha esse operador. Por exemplo, o operador *tem* pode ser definido pela diretiva:

```
:- op(600, xfx, tem).
```

Isso informa ao sistema que se deseja usar *tem* como um operador de prioridade 600 e cujo tipo é "xfx", que designa uma classe de operadores infixos. A forma de especificação, "xfx", sugere que o operador, denotado por "f", deva ser colocado entre dois argumentos, denotados por "x".

Deve-se notar que as definições de operadores não especificam qualquer operação ou ação. Em princípio, nenhuma operação sobre objetos é associada à definição de operadores. Os operadores são normalmente empregados como funtores, isto é, somente para combinar objetos em estruturas e não para executar alterações sobre tais objetos, apesar do termo "operador" sugerir essa execução. Os nomes dos operadores são átomos e sua prioridade encontra-se delimitada por valores inteiros cujo intervalo depende da implementação. Assumiremos aqui que esse intervalo varie entre 1 e 1200. Há três tipos básicos de operadores, conforme a tabela abaixo:

Tabela 4.1 Tipos de Operadores Prolog

OPERADORES	TIPO		
Infixos	xfx	xfy	yfx
Prefixos	fx	fy	-
Posfixos	xf	yf	-

A notação dos especificadores de tipo foi projetada para refletir a estrutura da expressão, onde f representa o operador e x e y representam os argumentos. Um f aparecendo entre os argumentos indica que o operador é infixo. As formas prefixo e posfixo possuem apenas um argumento que segue ou precede o operador respectivamente.

Há uma diferença entre x e y. Para explicá-la é necessário introduzir a noção de *prioridade de argumento*. Se um argumento estiver entre parênteses, ou for um objeto simples, então sua prioridade é zero. Se um argumento é uma estrutura, então sua prioridade é igual à prioridade de seu functor principal. O x representa um argumento cuja prioridade é obrigatoriamente menor do que a do operador, enquanto que y representa um argumento cuja prioridade é menor ou igual à prioridade do operador. Essas regras auxiliam a evitar ambigüidades em expressões com muitos operadores de mesma prioridade. Por exemplo, a expressão:

```
a - b - c
```

é normalmente entendida como (a-b)-c e não como a-(b-c). Para atingir a interpretação usual, o operador "-" tem que ser definido como yfx. A Figura 4.2 mostra como isso ocorre.

Na figura 4.2, assumindo que "-" tem a prioridade 500, se "-" for do tipo yfx, então a interpretação (b) é inválida, porque a precedência de (b-c) tem de ser obrigatoriamente menor do que a precedência de "-". Como outro exemplo, considere o operador prefixo *not*. Se *not* for definido como fy, então a expressão

```
not not p
```

é válida. Por outro lado, se *not* for definido como fx a expressão é ilegal, porque o argumento do primeiro *not* é *not p*, que tem a mesma prioridade que o *not*. Neste último caso a expressão precisa ser escrita entre parênteses:

```
not(not p)
```

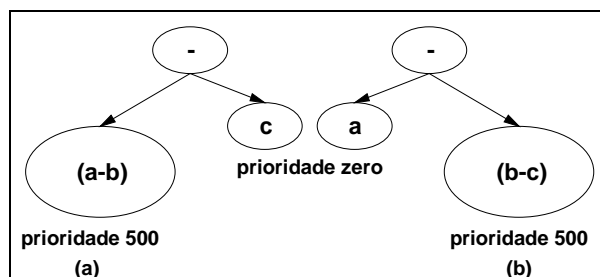


Figura 4.2 Duas interpretações para a expressão a - b - c

Para a conveniência do programador, alguns operadores são pré-definidos no sistema Prolog, de forma que estão sempre disponíveis para utilização, sem que seja necessário defini-los. O que esses operadores fazem e quais são as suas prioridades irá depender de cada particular implementação. Adotaremos aqui um conjunto padrão de operadores, conforme as definições apresentadas na Figura 4.3. Como também é mostrado ali, diversos operadores podem ser definidos em uma única diretiva, se eles tem todos a mesma prioridade e são todos do mesmo tipo. Neste caso os nomes dos operadores são escritos como uma lista delimitada por colchetes.

```

:- op(1200, xfx, ':-').
:- op(1200, fx, [':-', '?-']).
:- op(1100, xfy, ';').
:- op(1000, xfy, ',').
:- op( 700, xfx, [is, =, <, >, =<, >=, ==, =\=, \==, ==]).
:- op( 500, yfx, [+ , -]).
:- op( 500, fx,  [+ , - , not]).
:- op( 400, yfx, [* , / , div]).
:- op( 300, xfx, mod).
:- op( 200, xfy, ^).

```

Figura 4.3 Um conjunto padrão de operadores pré-definidos

O uso de operadores pode melhorar muito a legibilidade de alguns programas. Como um exemplo, vamos assumir que estejamos escrevendo um programa para a manipulação de expressões booleanas e que em tal programa desejamos estabelecer uma das leis de equivalência de De Morgan:

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

que pode ser estabelecida em Prolog pela cláusula:

```

equivale(não(e(A, B)), ou(não(A), não(B))).

```

Entretanto é uma boa prática em programação Prolog tentar reter a maior semelhança possível entre a notação original do problema e a notação usada no programa. Em nosso exemplo isso pode ser obtido por meio do uso de operadores. Um conjunto adequado de operadores para o nosso propósito pode ser definido como:

```

:- op( 800, xfx, <==> ).
:- op( 700, xfy, ^ ).
:- op( 600, xfy, v ).
:- op( 500, fxy, ¬ ).

```

com os quais a lei de De Morgan pode ser escrita como o fato:

$$\neg(A \wedge B) \iff \neg A \vee \neg B.$$

que, conforme estabelecido anteriormente, pode ser entendido como mostrado na figura abaixo:

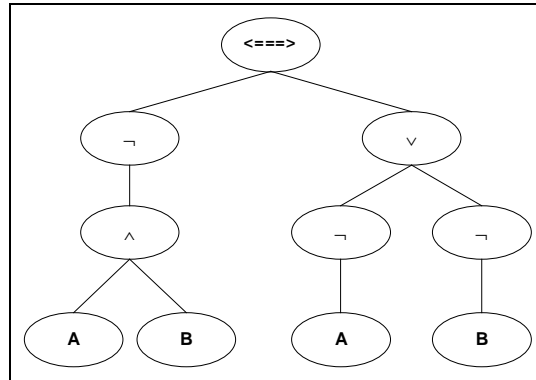


Figura 4.4 Interpretação do termo $\neg(A \wedge B) \iff \neg A \vee \neg B$

4.2 ARITMÉTICA

A linguagem Prolog é principalmente utilizada - como já se viu - para a computação simbólica, onde as necessidades de cálculo são comparativamente modestas. Assim, o instrumental da linguagem Prolog destinado a computações numéricas é algo simples em comparação com outras linguagens destinadas especificamente para esse fim, como por exemplo o Pascal-SC. Alguns dos operadores pré-definidos, anteriormente vistos podem ser usados para computação numérica. Tais operadores são mostrados na Tabela 4.2.

Tais operadores, excepcionalmente, *executam* uma certa operação. Mesmo em tais casos, entretanto, é necessário introduzir uma indicação adicional para executar a ação necessária. O sistema de tipos

dependendo da implementação. Aqui, "/" denotará a divisão em ponto flutuante, enquanto que o operador "div" denotará a divisão inteira. Exemplificando, na consulta abaixo:

```
?-X is 3/2, Y is 3 div 2.  
X=1.5    Y=1
```

O argumento à esquerda do operador "is" deve ser um objeto simples. O argumento à direita deve ser uma expressão aritmética, composta de operadores aritméticos, variáveis e números. Uma vez que o operador "is" irá forçar a execução da operação indicada, todas as variáveis contidas na expressão devem estar instanciadas com números no momento da execução de tal objetivo. A prioridade dos operadores aritméticos pré-definidos (ver Figura 4.4) é tal que a associatividade dos argumentos com os operadores é a mesma normalmente usada em matemática. Os parênteses podem ser usados para indicar associações diferentes. Note que +, -, *, / e div são definidos como yfx, o que significa que a execução se dará da esquerda para a direita. Por exemplo, X is 5-2-1 é interpretado como X is (5-2)-1.

A aritmética também é envolvida na comparação de valores numéricos. Por exemplo, a verificação se o produto de 277 por 37 é maior que 10000 pode ser especificada pelo objetivo:

```
?-277 * 37 > 10000.  
sim
```

Note que de forma semelhante ao operador "is", o operador ">" também força a avaliação de expressões. Suponha-se, por exemplo, uma relação denominada "nasceu", que relaciona nomes de pessoas com seus respectivos anos de nascimento. Então é possível recuperar os nomes das pessoas nascidas entre 1970 e 1980 inclusive, com a seguinte questão:

```
?-nasceu(Nome, Ano),  
Ano >= 1970,  
Ano <= 1980.
```

Na tabela abaixo apresenta-se um conjunto padrão de operadores de comparação utilizados em Prolog:

Tabela 4.3 Operadores de Comparação

OPERADOR	PRIORIDADE	TIPO	SIGNIFICADO
>	700	xfx	maior que
<	700	xfx	menor que
>=	700	xfx	maior ou igual a
<=	700	xfx	menor ou igual a
==	700	xfx	valores iguais
!=	700	xfx	valores diferentes

Note que a diferença existente entre o operador de unificação e o operador ==, por exemplo, nos objetivos X = Y e X == Y. O primeiro objetivo irá ocasionar a unificação dos objetos X e Y, instanciando, se for o caso, alguma variável em X e Y. Por outro lado, X == Y ocasiona a avaliação aritmética sem causar a instanciação de nenhuma variável. As diferenças se tornam claras nos exemplos a seguir:

```
?-1+2 == 2+1.  
sim  
  
?-1+2 = 2+1.  
não  
  
?-1+A = B+2.  
A=2 B=1
```

Mesmo não sendo direcionadas para a computação aritmética, as diferentes implementações do Prolog normalmente possuem um conjunto de funções pré-definidas para a execução de cálculos científicos. Tais funções podem ser empregadas em expressões matemáticas de modo similar às linguagens convencionais. Um conjunto padrão de tais funções é apresentado na tabela abaixo:

Tabela 4.4 Funções Pré-Definidas em Prolog

FUNÇÃO	SIGNIFICADO
abs(X)	Valor absoluto de X
acos(X)	Arco-cosseno de X
asin(X)	Arco-seno de X
atan(X)	Arco-tangente de X
cos(X)	Cosseno de X
exp(X)	Valor de "e" elevado a X
ln(X)	Logaritmo natural de X
log(X)	Logaritmo decimal de X
sin(X)	Seno de X
sqrt(X)	Raiz quadrada de X
tan(X)	Tangente de X
round(X,N)	Arredonda X para N casas decimais
Pi	Valor de p com 15 casas decimais
Random	Um número aleatório entre 0 e 1

Por exemplo, são válidas as seguintes expressões Prolog:

```
X is 3 * (cos(random))^2.
Y is sin(pi/6)*sqrt(tan(pi/12)).
```

Como um exemplo mais complexo, suponha o problema de computar o máximo divisor comum de dois números. Dados dois inteiros positivos, X e Y, seu máximo divisor comum D pode ser encontrado segundo três casos distintos:

- (1) Se X e Y são iguais, então D é igual a X;
- (2) Se $X < Y$, então D é igual ao mdc entre X e a diferença $X - Y$;
- (3) Se $X > Y$, então cai-se no mesmo caso (2), com X substituído por Y e vice-versa.

As três cláusulas Prolog que expressam os três casos acima são:

```
mdc(X, X, X).
mdc(X, Y, D) :-
    X < Y,
    Y1 is Y-X,
    mdc(X, Y1, D).
mdc(X, Y, D) :-
    X > Y,
    mdc(Y, X, D).
```

Naturalmente, o último objetivo na terceira cláusula poderia ser de modo equivalente substituído por:

```
X1 is X-Y,
mdc(X1, Y, D).
```

Um último exemplo será dado para recursivamente calcular o fatorial de um número inteiro dado. O programa é:

```
fatorial(0, 1).
fatorial(X, Y) :-
    X1 is X-1,
    fatorial(X1, Y1),
    Y is X*Y1.
```

É interessante notar aqui que o processo recursivo mantém latentes todas as operações aritméticas até que o fato "fatorial(0, 1)" seja alcançado, quando então, todas as operações pendentes são executadas para fornecer em Y o fatorial desejado.

RESUMO

- A notação para definição de operadores permite ao programador adequar a sintaxe de seus programas para suas necessidades particulares, melhorando consideravelmente sua legibilidade;

- Novos operadores são definidos por meio da diretiva "op", que estabelece o nome do operador, seu tipo e prioridade;
- Em princípio não há nenhuma execução associada a um operador, que são meramente dispositivos sintáticos que oferecem a possibilidade de se escrever termos Prolog em uma sintaxe alternativa;
 - A aritmética é executada por meio de procedimentos embutidos. A avaliação de uma expressão aritmética é forçada pelo uso do operador "is" e dos operadores de comparação. No momento da avaliação, todas as variáveis devem estar instanciadas.

EXERCÍCIOS

4.1 Assumindo as seguintes definições de operadores:

```
:- op(300, xfx, joga).
:- op(200, xfy, e).
```

então os dois termos seguintes possuem sintaxe válida:

```
T1 = marcelo joga futebol e squash.
T2 = renata joga tenis e basquete e volei.
```

Como estes termos são interpretados pelo Prolog? Qual é o functor principal de cada termo e qual a sua estrutura?

4.2 Sugira uma apropriada definição dos operadores "era" e "do" para que seja possível a escrita de cláusulas como:

```
vera era secretária do departamento.
```

e

```
paulo era professor do curso.
```

4.3 Considere o seguinte programa Prolog:

```
t(0+1, 1+0).
t(X+0+1, X+1+0).
t(X+1+1, Z) :-
    t(X+1, X1),
    t(X1+1, Z).
```

Como irá este programa responder as seguintes questões, considerando ser + um operador infix do tipo yfx (como usual).

- ?-t(0+1, A).
- ?-t(0+1+1, B).
- ?-t(1+0+1+1+1, C).
- ?-t(D, 1+1+1+0).

4.4 Defina os operadores "se", "então", "senão" e ":-=" de modo que seja válido o termo:

```
se X>Y então Z := X senão Z := Y
```

Escolha a precedência dos operadores de modo que "se" venha a ser o functor principal. Depois defina a relação "se" como um mini-interpretador para um tipo de comando se-então da forma:

```
se V1>V2 então Var:=V3 senão Var:=V4
```

onde V1, V2, V3 e V4 são números (ou variáveis instanciadas com números) e Var é uma variável. O significado da relação "se" deve ser: "se o valor de V1 é maior que o valor de V2, então Var é instanciada com V3, senão Var é instanciada com V4. Um exemplo do uso do mini-interpretador seria:

```
?-X=2, Y=3, V2 is 2*X, V4 is 4*X,
  se Y > V2 então Z:=Y senão Z:=V4,
  se Z > 5 então W=1 senão W=0.
X=2 Y=3 Z=8 W=1
```

4.5 Defina o procedimento

`entre(N1, N2, X)`

que, para dois inteiros dados, N1 e N2, produz através de backtracking todos os inteiros X que satisfazem a restrição

`N1 ≤ X ≤ N2`

- 4.6 Estude a definição de um "mundo de polígonos" onde os objetos são definidos em função das coordenadas de seus vértices no plano. Indivíduos desse universo seriam triângulos, retângulos, quadrados, etc. Por exemplo o termo:

`triângulo((1,1), (1,2), (2,2))`

definiria um triângulo cujos vértices seriam os pontos (1,1), (1,2) e (2, 2) em um sistema de coordenadas cartesianas.

Formule as propriedades básicas de cada objeto através de relações unárias, tais como:

`isósceles(X)`

Formule relações entre diferentes indivíduos, representando assertivas tais como:

`"Uma casa é um quadrado com um triângulo em cima".`

ou

`"D é distância entre os centros geométricos de A e B".`

Pense numa versão deste programa para gerar trajetórias de figuras planas ao longo de curvas de equações dadas.

5. PROCESSAMENTO DE LISTAS

Uma importante classe de estruturas de dados em Prolog é composta de expressões simbólicas, também denominadas "S-Expressões", que permitem a representação de listas de tamanho indefinido como tipos de árvores onde os ramos, também denominados sub-árvores, são reunidos entre parênteses e outros delimitadores para formar sequências de objetos. A analogia entre listas aninhadas e árvores é fundamental para o perfeito entendimento de algumas operações realizadas em listas. A sintaxe das listas em Prolog é uma variante da sintaxe empregada em LISP, que é uma linguagem tradicionalmente empregada em inteligência artificial e computação simbólica. No presente capítulo abordase a representação em listas, a codificação em Prolog de diversas operações e a construção de algumas aplicações empregando estruturas em listas.

5.1 REPRESENTAÇÃO DE LISTAS

Listas são estruturas simples de dados, largamente empregadas em computação não-numérica. Uma lista é uma sequência de qualquer número de itens, como: brasil, uruguai, argentina, paraguai. Uma lista deste tipo pode ser escrita em Prolog como:

```
[brasil, uruguai, argentina, paraguai]
```

Essa, entretanto, é apenas a aparência externa das listas. Como já foi visto, todos os objetos estruturados em Prolog são na realidade árvores e as listas seguem a regra. Como representar listas como objetos Prolog? Dois casos devem ser considerados: a lista vazia e a lista não-vazia. No primeiro caso, a lista é representada simplesmente como um átomo, []. No segundo, a lista deve ser pensada como constituída de dois componentes: uma "cabeça" e um "corpo". Por exemplo, na lista dada, a cabeça é "brasil" e o corpo é a lista [uruguai, argentina, paraguai].

Em geral, a cabeça pode ser qualquer objeto Prolog - como uma árvore ou uma variável. O corpo, entretanto, deve ser obrigatoriamente uma lista. A cabeça e o corpo são combinados em uma estrutura por meio de um functor especial. A escolha desse functor depende da implementação considerada da linguagem Prolog. Aqui será assumido o ponto "•" que é o símbolo funcional adotado com maior freqüência na representação de listas nas diversas implementações Prolog:

```
•(Cabeça, Corpo)
```

Uma vez que a variável Corpo representa, por sua vez, uma lista, esta pode ser vazia ou possuir a sua própria cabeça e corpo, portanto, para a representação de listas de qualquer tamanho, nenhum princípio adicional é necessário. O exemplo de lista dado é então representado pelo termo Prolog:

```
•(brasil, •(uruguai, •(argentina, •(paraguai, [])))).
```

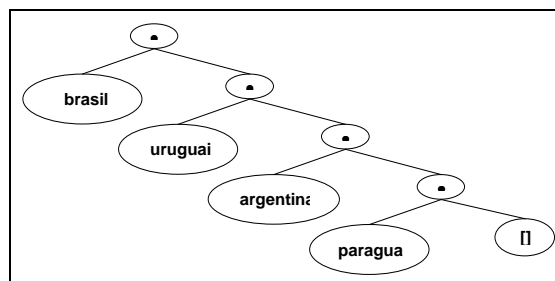


Figura 5.1 Uma lista representada como árvore.

Na Figura 5.1 apresenta-se a correspondente estrutura em árvore. Note que a lista vazia aparece no termo acima. Isso ocorre porque o último "corpo" é uma lista de um único item [paraguai], que possui uma lista vazia como seu corpo:

```
[paraguai] = •(paraguai, [])
```

Esse exemplo mostra como o princípio geral para a estruturação de objetos Prolog também se aplica a listas de qualquer tamanho. Como o exemplo também mostra, a notação direta com o uso do functor "•" pode produzir expressões bastante confusas. Por essa razão o sistema Prolog oferece uma notação simplificada para as listas, permitindo que as mesmas sejam escritas como seqüências de itens separados por vírgulas e incluídos entre colchetes. O programador pode empregar qualquer notação, entretanto, a que utiliza colchetes é normalmente preferida. Segundo tal notação, um termo da forma [H|T] é tratado como uma lista de cabeça H e corpo T. Listas do tipo [H|T] são estruturas muito comuns em programação não-numérica. Deve-se recordar que o corpo de uma lista é sempre outra lista, mesmo que seja vazia. Os seguintes exemplos devem servir para demonstrar tais idéias:

```
[x | y] ou [x | [y | z]] unificam com [a, b, c, d]
```

```
[x, y, z] não unifica com [a, b, c, d]
```

```
[a, b, c] == [a | [b | [c]]] == [a | [b, c]] == [a, b | [c]] == [a, b, c | []]
```

As consultas abaixo também são elucidativas:

```
?-[x | y] = [a, b, c].  
x=a y=[b, c]
```

```
?-[x, y, z] = [a, b, c, d].  
não
```

```
?-[x | [y | z]] = [a, b, c, d].  
x=a y=b z=[c, d]
```

5.2 OPERAÇÕES SOBRE LISTAS

Estruturas em lista podem ser definidas e transformadas em Prolog de diversas maneiras diferentes. Na presente seção procura-se, através de uma variedade de exemplos, mostrar a flexibilidade das listas na representação de situações complexas. Emprega-se, para maior clareza, de agora em diante a notação:

```
simbolo_predicativo/aridade
```

para a identificação de predicados. Por exemplo gráfico/3 denota uma relação denominada gráfico com três argumentos. Esse detalhamento é às vezes importante. Nome e aridade são os elementos necessários e suficientes para a perfeita identificação de um predicado.

5.2.1 CONSTRUÇÃO DE LISTAS

A primeira necessidade para a manipulação de listas é ser capaz de construí-las a partir de seus elementos básicos: uma cabeça e um corpo. Tal relação pode ser escrita em um único fato:

```
cons(x, y, [x | y]).
```

Por exemplo:

```
?-cons(a, b, z).  
z=[a | b]
```

Durante a unificação a variável X é instanciada com a, Y com b e Z com [X|Y], que por sua vez é instanciada com [a|b], devido aos valores de X e Y. Se X for um elemento e Y uma lista, então [X|Y] é uma nova lista com X como primeiro elemento. Por exemplo:

```
?-cons(a, [b, c], z).  
z=[a, b, c]  
  
?-cons(a, [], z).  
z=[a]
```

A generalidade da unificação permite a definição de um resultado implícito:

```
?-cons(a, X, [a, b, c]).
X=[b, c]
```

Neste último exemplo as propriedades de simetria dos argumentos, lembram um solucionador de equações: um X é encontrado tal que $[a|X] = [a, b, c]$. Entretanto, se o primeiro argumento for uma lista com, digamos, três elementos e o segundo uma lista com dois, o resultado não será uma lista com cinco elementos:

```
?-cons([a, b, c], [d, e], Z).
Z=[[a, b, c], d, e]
```

de modo que o predicado `cons/3` não resolve o problema de concatenar duas listas em uma terceira. Mais adiante será estudado o predicado `conc/3` que realiza tal função.

5.2.2 OCORRÊNCIA DE ELEMENTOS EM UMA LISTA

Vamos implementar um tipo de relação de ocorrência que estabelece se determinado objeto é membro de uma lista, como em:

```
membro(X, L)
```

onde X é um objeto e L uma lista. O objetivo `membro(X, L)` é verdadeiro se X ocorre em L. Por exemplo, são verdadeiros:

```
membro(b, [a, b, c])
membro([b,c], [a, [b, c], d])
```

mas a declaração

```
membro(b, [a, [b, c]])
```

é falsa. O programa que define a relação `membro/2` baseia-se na seguinte afirmação:

```
X é membro de L se
(1) X é a cabeça de L, ou
(2) X é membro do corpo de L.
```

que pode ser representada em Prolog por meio de duas cláusulas. A primeira, um fato, estabelece a primeira condição: X é membro de L, se X é a cabeça de L. A segunda, uma regra que será empregada quando X não é cabeça de L, é uma chamada recursiva que diz que X ainda pode ser membro de L, desde que seja membro do corpo de L. Em Prolog:

```
membro(X, [X | C]).
membro(X, [_ | C]) :-
    membro(X, C).
```

Note-se que o corpo da lista na primeira cláusula é sempre um resultado sem qualquer interesse, o mesmo ocorrendo com a cabeça da lista na segunda. É possível então empregar variáveis anônimas e escrever o predicado de forma mais elegante:

```
membro(X, [X | _]).
membro(X, [_ | C]) :-
    membro(X, C).
```

5.2.3 CONCATENAÇÃO DE LISTAS

Para a concatenação de duas listas quaisquer, resultando em uma terceira, se definirá a relação:

```
conc(L1, L2, L3)
```

onde L1 e L2 são duas listas e L3 é a concatenação resultante. Por exemplo:

```
conc([a, b], [c, d], [a, b, c, d])
```

Novamente, dois casos devem ser considerados para a definição de `conc/3`, dependendo do primeiro argumento L1:

- (1) Se o primeiro argumento é uma lista vazia, então o segundo e o terceiro argumentos devem ser

a mesma lista. Chamando tal lista de L, essa situação pode ser representada pelo seguinte fato Prolog:

```
conc([], L, L).
```

- (2) Se o primeiro argumento de conc/3 for uma lista não-vazia, então é porque ela possui uma cabeça e um corpo e pode ser denotada por [X|L1]. A concatenação de [X|L1] com uma segunda lista L2, produzirá uma terceira lista com a mesma cabeça X da primeira e um corpo L3 que é a concatenação do corpo da primeira lista, L1, com toda a segunda, L2. Isso pode ser visto na figura 5.2, e se representa em Prolog por meio da regra:

```
conc([X | L1], L2, [X | L3]) :-  
    conc(L1, L2, L3).
```

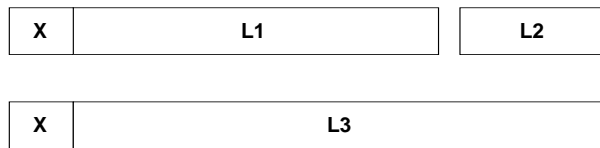


Figura 5.2 Concatenação de duas listas

O programa completo para a concatenação de listas, descrevendo o predicado conc/3 é apresentado a seguir:

```
conc([], L, L).  
conc([X | L1], L2, [X | L3]) :-  
    conc(L1, L2, L3).
```

Exemplos simples de utilização de tal programa são:

```
?-conc([a, b, c], [1, 2, 3], L).  
L=[a, b, c, 1, 2, 3]  
  
?-conc([a, [b, c], d], [a, [], b], L).  
L=[a, [b, c], d, a, [], b]  
  
?-conc([a, b], [c | R], L).  
L=[a, b, c | R]
```

O programa conc/3, apesar de muito simples, é também muito flexível e pode ser usado em inúmeras aplicações. Por exemplo, ele pode ser usado no sentido inverso ao que foi originalmente projetado para decompor uma lista em duas partes:

```
?- conc(L1, L2, [a, b, c]).  
L1=[] L2=[a, b, c];  
L1=[a] L2=[b, c];  
L1=[a, b] L2=[c];  
L1=[a, b, c] L2=[];  
não
```

Esse resultado mostra que é sempre possível decompor uma lista de n elementos em n+1 modos, todos eles obtidos pelo programa através de backtracking. Podemos também usar o programa para procurar por um determinado padrão em uma lista. Por exemplo, podemos encontrar os meses antes e depois de um determinado mês:

```
?-M=[jan,fev,mar,abr,mai,jun,jul,ago,set,out,nov,dez], conc(Antes, [mai | Depois], M).  
Antes=[jan,fev,mar,abr] Depois=[jun,jul,ago,set,out,nov, dez]
```

e também achar o sucessor e o predecessor imediatos (os vizinhos) de um determinado item da lista:

```
?-conc(_, [X, g, Y | _], [a, b, c, d, e, f, g, h]).  
X=f Y=h
```

É possível ainda apagar de uma lista todos os elementos que se seguem a um determinado padrão. No exemplo abaixo, retira-se da lista dos dias da semana a sexta-feira e todos os dias que a seguem.

```
?-conc(Trab, [sex | _], [seg,ter,qua,qui,sex,sab,dom]).  
Trab=[seg,ter,qua,qui]
```


A própria relação de ocorrência, `membro/2`, vista na seção anterior pode ser reprogramada em função de `conc/3`:

```
membro1(X, L) :-
    conc(_, [X | _], L).
```

Essa cláusula nos diz que `X` é membro de uma lista `L` se `L` pode ser decomposta em duas outras listas onde a cabeça da segunda é `X`. Na verdade, `membro1/2` define a mesma relação que `membro/2`, apenas adotou-se um nome diferente para estabelecer uma distinção entre ambas.

5.2.4 REMOÇÃO DE ELEMENTOS DE UMA LISTA

A remoção de um elemento `X` de uma lista `L` pode ser programada através da relação:

```
remover(X, L, L1)
```

onde `L1` é a mesma lista `L` com o elemento `X` removido. A relação `remover/3` pode ser definida de maneira similar à relação de ocorrência. Novamente são dois casos a estudar:

- (1) Se `X` é a cabeça da lista `L`, então `L1` será o seu corpo;
- (2) Se `X` está no corpo de `L`, então `L1` é obtida removendo `X` desse corpo.

Em Prolog, isso é escrito da seguinte maneira:

```
remover(X, [X | C], C).
remover(X, [Y | C], [Y | D]) :-
    remover(X, C, D).
```

Assim como a relação `membro/2`, `remover/3` é também não-determinística por natureza. Se há diversas ocorrências de `X` em `L`, a relação `remove/3` é capaz de retirar cada uma delas através do mecanismo de backtracking do Prolog. Evidentemente, em cada execução do programa `remove/3` retiramos somente uma das ocorrências de `X`, deixando as demais intocáveis. Por exemplo:

```
?-remover(a, [a, b, a, a], L).
L=[b, a, a];
L=[a, b, a];
L=[a, b, a];
não
```

`remover/3` irá falhar se a lista `L` não contiver nenhuma ocorrência do elemento `X`. Essa relação pode ser ainda usada no sentido inverso para inserir um novo item em qualquer lugar da lista. Por exemplo, pode-se formular a questão: "Qual é a lista `L`, da qual retirando-se '`a`', obtem-se a lista `[b, c, d]`?"

```
?-remover(a, L, [b, c, d]).
L=[a, b, c, d];
L=[b, a, c, d];
L=[b, c, a, d];
L=[b, c, d, a];
não
```

De modo geral, pode-se inserir um elemento `X` em algum lugar de uma lista `L`, resultando em uma nova lista `L1`, com o elemento `X` inserido na posição desejada, por meio da cláusula:

```
inserir(X, L, L1) :-
    remover(X, L1, L).
```

Em `membro1/2` foi obtida uma forma alternativa para a relação de ocorrência, utilizando o predicado `conc/3`. Pode-se obter a mesma relação por meio de `remover/3`:

```
membro2(X, L) :-
    remover(X, L, _).
```

5.2.5 INVERSÃO DE LISTAS

A relação que inverte uma lista, isto é, que organiza seus elementos na ordem inversa é útil para os mais diversos propósitos. Abaixo temos alguns exemplos de inversão:

```
inverter([a, b, c], [c, b, a]).
inverter([], []).
inverter([a, [b, c], d], [d, [b, c], a])
```

Dentre os diversos mecanismos lógicos capazes de inverter uma lista, o denominado "inversão ingênua" baseia-se numa abordagem muito direta, embora seu tempo de execução seja proporcional ao quadrado do tamanho da lista:

- (1) Tomar o primeiro elemento da lista;
- (2) Inverter o restante;
- (3) Concatenar a lista formada pelo primeiro elemento ao inverso do restante.

Em Prolog, escreve-se:

```
inverter([], []).
inverter([X | Y], Z) :-
    inverter(Y, Y1),
    conc(Y1, [X], Z).
```

Esse programa, juntamente com o predicado `conc/3`, costuma ser empregado como um teste benchmark para sistemas Prolog. Quando o número de inferências lógicas, ou chamadas de objetivos Prolog é dividido pelo número de segundos gastos, o número obtido mede a velocidade do sistema Prolog em LIPS (logic inferences per second). A inversão de listas pode, entretanto ser obtida de modo mais eficiente por meio de um predicado auxiliar, iterativo, `aux/3`, tornando o tempo de execução apenas linearmente proporcional ao tamanho da lista a inverter:

```
inverter(X, Y) :-
    aux([], X, Y).

aux(L, [], L).
aux(L, [X | Y], Z) :-
    aux([X | L], Y, Z).
```

5.2.6 SUBLISTAS

Iremos considerar agora a relação `sublista/2` que possui como argumentos uma lista `S` e uma lista `L` tais que `S` ocorre em `L` como sublista. Assim, é verdadeira a afirmação:

```
sublista([c, d, e], [a, b, c, d, e, f])
```

mas é falso declarar que:

```
sublista([c, e], [a,b,c,d,e,f])
```

O programa Prolog para a relação `sublista/2` pode se basear na mesma idéia explorada na definição do predicado `membro1/2`, com a diferença que, desta vez, a relação é mais genérica, podendo ser formulada por:

```
S é sublista de L se:
(1) L pode ser decomposta em duas listas, L1 e L2, e
(2) L2 pode ser decomposta em S e L3.
```

Como foi visto anteriormente, a relação `conc/3` pode ser usada para a decomposição de listas. Assim a formulação acima pode ser expressa em Prolog por:

```
sublista(S, L) :-
    conc(L1, L2, L),
    conc(S, L3, L2).
```

O programa `sublista/2` pode ser usado de modo bastante flexível em diversas aplicações. Apesar de ter sido projetado para verificar se alguma lista ocorre como sublista de outra, ele pode, por exemplo, ser

usado para obter todas as sublistas de uma lista:

```
?-sublista(S, [a, b, c]).
S=[];
S=[a];
S=[a, b];
S=[a, b, c];
S=[b];
S=[b, c];
S=[c];
não
```

5.2.7 PERMUTAÇÃO DE LISTAS

Algumas vezes pode ser interessante gerar permutações de uma dada lista. Com essa finalidade define-se a relação `permutar/2` cujos argumentos são duas listas tais que cada uma é permutação da outra. A intenção é permitir a geração de todas as permutações possíveis de uma dada lista empregando o mecanismo de backtracking que pode ser disparado a partir da relação `permutar/2`, como por exemplo em:

```
?-permutar([a, b, c], P).
P=[a, b, c];
P=[a, c, b];
P=[b, a, c];
P=[b, c, a];
P=[c, a, b];
P=[c, b, a];
não
```

O programa `permutar/2` deve novamente basear-se na consideração de dois casos, dependendo da lista a ser permutada:

- (1) Se a primeira lista é vazia, então a segunda também é;
- (2) Se a primeira lista é não-vazia, então possui a forma `[X|L]` e uma permutação de tal lista pode ser construída primeiro permutando `L` para obter `L1` e depois inserindo `X` em qualquer posição de `L1`, conforme a Figura 5.3.

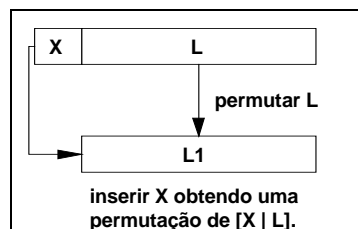


Figura 5.3 Permutação de Listas

A relação Prolog correspondente é:

```
permutar([], []).
permutar([X | L], P) :-
    permutar(L, L1),
    inserir(X, L1, P).
```

O uso normal da relação `permutar/2` seria como no exemplo dado anteriormente, `permutar([a, b, c], P)`. Uma tentativa diferente seria propor ao sistema:

```
?-permutar(L, [a, b, c]).
```

Aqui o programa dado irá, de início, obter em `L` as seis permutações existentes para `[a, b, c]`, mas depois, se o usuário pedir mais soluções, o programa nunca irá responder "não", entrando em um laço infinito na tentativa de encontrar outra permutação onde já não há mais nenhuma. Assim, algum cuidado é necessário no uso desta relação.

5.3 OUTROS EXEMPLOS

Dada a importância do uso de listas em Prolog, apresenta-se informalmente na presente seção algumas aplicações adicionais definidas sobre listas que podem vir a ser de grande utilidade em programas futuros, deixando-se ao leitor a tarefa de verificar o seu funcionamento segundo as diferentes interpretações estudadas.

5.3.1 TAMANHO DE UMA LISTA

A relação tamanho/2, representada por tamanho(L, T) será verdadeira quando T for o número de elementos existentes em L:

```
tamanho([], 0).
tamanho([_ | R], N) :-
    tamanho(R, N1),
    N is N1+1.
```

Por exemplo:

```
?-tamanho([a, b, c, d, e], X).
X=5
```

5.3.2 SELEÇÃO DE ELEMENTOS PARTICULARES

Muitas vezes é necessário identificar em uma lista um determinado elemento que possua uma certa propriedade. Isso pode ser realizado através da relação prop/2, abaixo, onde p/1 representa a propriedade procurada, devendo estar definida no programa. Note a semelhança dessa relação com o predicado membro/2, anteriormente discutido.

```
prop(X, [X | _]) :-
    p(X).
prop(X, [_ | Y]) :-
    prop(X, Y).
```

Outras vezes é necessário selecionar exatamente o enésimo elemento de uma lista. O predicado enésimo/3, a seguir, realiza esta função:

```
enésimo(1, X, [X | _]).
enésimo(N, X, [_ | Y]) :-
    enésimo(M, X, Y),
    N is M+1.
```

Exemplos de utilização desse predicado são:

```
?-enésimo(3, X, [a, b, c, d]).
X=c

?-enésimo(N, b, [a, b, c, d]).
N=2
```

Outra necessidade freqüente é reunir em uma lista separada determinados elementos de uma lista, identificados pela sua posição. Isso é obtido pelo predicado seleciona/3, abaixo, que por sua vez emprega a relação enésimo/3:

```
seleciona([], _, []).
seleciona([M | N], L, [X | Y]) :-
    enésimo(M, X, L),
    seleciona(N, L, Y).
```

Por exemplo:

```
?-seleciona([2, 4], [a, b, c, d, e], L).
L=[b, d]
```

5.3.3 SOMA E PRODUTO

O somatório e o produtório de uma lista são dados respectivamente pelas relações soma/2 e produto/2 abaixo. Observe o artifício empregado na definição de produto/2, para garantir que o produtório de uma lista vazia seja zero.

```
soma([], 0).
soma([X | Y], S) :-
    S is R+X,
    soma(Y, R).

produto([], 0).
produto([X], X).
produto(L, P) :-
    prod(L, P).

prod([], 1).
prod([X | Y], P) :-
    P is Q*X,
    prod(Y, Q).
```

Exemplos dos predicados soma/2 e produto/2 são dados abaixo:

```
?-soma([1, 2, 3, 4], X).
X=10

?-soma([1,2, X, 4], 10).
X=3

?-produto([], X).
X=0

?-produto([1, X, Y, 4], 24).
X=1 Y=6;
X=2 Y=3;
X=3 Y=2;
X=6 Y=1;
não
```

Este último exemplo, apesar da interpretação declarativa correta, no domínio dos inteiros positivos, poderá não funcionar corretamente em todas as implementações Prolog devido a características operacionais particulares de irreversibilidade dos operadores aritméticos.

5.3.4 INTERSECÇÃO DE LISTAS

O predicado intersec/3, a seguir, computa a intersecção de duas listas em uma terceira:

```
intersec([X | Y], L, [X | Z]) :-
    membro(X, L),
    intersec(Y, L, Z).
intersec([_ | X], L, Y) :-
    intersec(X, L, Y).
intersec(_, _, []).
```

Por exemplo:

```
?-intersec([a, b, c, d], [aa, b, d], L).
L=[b, d]
```

RESUMO

- Listas são estruturas freqüentemente usadas em Prolog. Estas podem ser vazias (representadas pelo átomo []), ou constituídas por uma cabeça (seu primeiro elemento) e um corpo (os demais);
- A notação usual para listas emprega o functor "•" (ponto) reunindo dois argumentos, a cabeça e o corpo, em uma única lista. Por exemplo, •(a, •(b, •(c, []))) representa a lista [a, b, c];
- Há uma notação simplificada em Prolog que permite a representação de listas na forma [H|T], onde H é a cabeça e T o corpo da lista.
- A cabeça de uma lista pode ser qualquer termo Prolog, entretanto o corpo de uma lista sempre será uma lista;

- Há uma correspondência entre listas e estruturas em árvore, permitindo que listas sejam elementos de outras listas;
- Operações comuns sobre listas apresentadas no presente capítulo foram: construção, ocorrência, concatenação, inserção, remoção, inversão, sublistas e permutações de listas.

EXERCÍCIOS

- 5.1 Escreva um programa denominado `acomoda/2` cujo primeiro argumento é uma lista permitindo listas como elementos, tal como `[a, [a, [b, c]], b, [c, d]]`, e cujo segundo argumento é outra lista com todos os elementos da primeira acomodados em uma única lista, como `[a, a, b, c, b, c, d]`. Por exemplo:

```
?-acomoda([a, [b], [c, d]], L).
L=[a, b, c, d]
```

Examine a reversibilidade do predicado obtido. O que é possível obter por meio de backtracking?

- 5.2 Qual o número de inferências necessário para computar o inverso de uma lista pelo método da inversão ingênua? Use-o para medir a velocidade em LIPS do seu sistema Prolog.
- 5.3 Escreva um programa que inverta uma lista de elementos e que também, recursivamente, inverta esses próprios elementos quando eles forem listas.
- 5.4 Escreva um programa denominado

```
escore(X, Y, A, B)
```

onde `X` e `Y` são listas de inteiros do mesmo tamanho, `A` é o número de posições que possuem números idênticos e `B` é o número de elementos que ocorrem simultaneamente em ambas as listas, mas em posições diferentes. Por exemplo:

```
?-escore([7, 2, 3, 4], [2, 3, 4, 4], A, B).
A=1 B=2
```

- 5.5 Escreva um programa denominado

```
limpa(X, L1, L2)
```

que produz `L2` como sendo `L1` sem nenhuma ocorrência do termo `X`.

- 5.6 Escreva um predicado denominado

```
palindromo(X)
```

que é verdadeiro se `X` é uma lista cujos elementos invertidos produzem a mesma ordem original. Por exemplo:

```
?-palindromo([a, X, a, r, Y]).
X=r Y=a
```

- 5.7 Escreva um predicado denominado

```
estat(L, Max, Min, Med, DP)
```

onde `L` é uma lista de números, `Max` o maior destes números, `Min` o menor, `Med` sua média aritmética e `DP` o seu desvio padrão.

- 5.8 Escreva um programa denominado

```
ordena(X, Y)
```

onde `Y` é uma versão ordenada da lista `X`. Por exemplo:

```
?-ordena([9, 6, 5, 1, 6], L).
L=[1, 5, 6, 6, 9]
```

6. CONTROLE

Como já foi visto, o programador pode controlar a execução de seu programa através da reordenação das cláusulas ou de objetivos no interior destas. Neste capítulo se estudará um outro instrumento para o controle de programas - denominado "cut" - que se destina a prevenir a execução do backtracking quando este não for desejado. Também se introduzirá a "negação por falha", uma forma operacional da negação lógica. Exemplos serão apresentados com a finalidade de ilustrar os conceitos desenvolvidos.

6.1 BACKTRACKING

Na execução dos programas Prolog, a evolução da busca por soluções assume a forma de uma árvore - denominada "árvore de pesquisa" ou "search tree" - que é percorrida sistematicamente de cima para baixo (top-down) e da esquerda para direita, segundo o método denominado "depth-first search" ou "pesquisa primeiro em profundidade". A Figura 6.1 ilustra esta idéia. Ali é representada a árvore correspondente à execução do seguinte programa abstrato, onde a, b, c, etc. possuem a sintaxe de termos Prolog:

```
a :- b.  
a :- c.  
a :- d.  
  
b :- e.  
b :- f.  
  
f :- g.  
f :- h.  
f :- i.  
  
d.
```

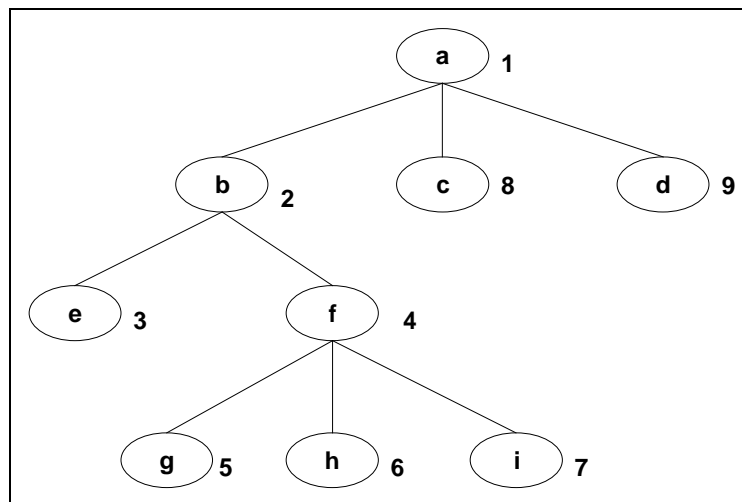


Figura 6.1 Ordem de visita aos nodos da árvore de pesquisa

O programa representado na figura acima será bem sucedido somente quando o nodo d for atingido, uma vez que este é o único fato declarado no programa. De acordo com a ordenação das cláusulas, d será também o último nodo a ser visitado no processo de execução. O caminho percorrido é dado abaixo

a, b, e, (b), f, g, (f), h, (f), i, (f), (b), (a), c, (a), d

onde o caminho em *backtracking* é representado entre parênteses.

Como foi visto, os objetivos em um programa Prolog podem ser bem-sucedidos ou falhar. Para um objetivo ser bem-sucedido ele deve ser unificado com a cabeça de uma cláusula do programa e todos os objetivos no corpo desta cláusula devem também ser bem-sucedidos. Se tais condições não ocorrerem, então o objetivo falha.

Quando um objetivo falha, em um nodo terminal da árvore de pesquisa, o sistema Prolog aciona o mecanismo de backtracking, retornando pelo mesmo caminho percorrido, na tentativa de encontrar soluções alternativas. Ao voltar pelo caminho já percorrido, todo o trabalho executado é desfeito. O seguinte exemplo, sobre o predicado `gosta/2` pode ajudar a esclarecer tais idéias.

```
gosta(joão, jazz).
gosta(joão, renata).
gosta(joão, lasanha).
gosta(renata, joão).
gosta(renata, lasanha).
```

O significado intuitivo do predicado `gosta(X, Y)` é "X gosta de Y". Supondo o conhecimento acima, queremos saber do que ambos, João e Renata, gostam. Isto pode ser formulado pelos objetivos:

```
gosta(joão, X), gosta(renata, X).
```

O sistema Prolog tenta satisfazer o primeiro objetivo, desencadeando a seguinte execução top-down:

1. Encontra que João gosta de jazz
2. Instancia X com "jazz"
3. Tenta satisfazer o segundo objetivo, determinando se "Renata gosta de jazz"
4. Falha, porque não consegue determinar se Renata gosta de jazz
5. Realiza um backtracking na repetição da tentativa de satisfazer `gosta(joão, X)`, esquecendo o valor "jazz"
6. Encontra que João gosta de Renata
7. Instancia X com "Renata"
8. Tenta satisfazer o segundo objetivo determinando se "Renata gosta de Renata"
9. Falha porque não consegue demonstrar que Renata gosta de Renata
10. Realiza um backtracking, mais uma vez tentando satisfazer `gosta(joão, X)`, esquecendo o valor "Renata"
11. Encontra que João gosta de lasanha
12. Instancia X com "lasanha"
13. Encontra que "Renata gosta de lasanha"
14. É bem-sucedido, com X instanciado com "lasanha"

O backtracking automático é uma ferramenta muito poderosa e a sua exploração é de grande utilidade para o programador. Às vezes, entretanto, ele pode se transformar em fonte de ineficiência. A seguir se introduzirá um mecanismo para "podar" a árvore de pesquisa, evitando o backtracking quando este for indesejável.

6.2 O OPERADOR "CUT"

O papel desempenhado pelo operador "cut", é de extrema importância para semântica operacional dos programas Prolog, permitindo declarar ramificações da árvore de pesquisa que não devem ser retomadas no backtracking. Seu uso deve ser considerado pelas seguintes razões:

- (i) O programa irá executar mais rapidamente, porque não irá desperdiçar tempo tentando satisfa-

zer objetivos que não irão contribuir para a solução desejada.

- (ii) Também a memória será economizada, uma vez que determinados pontos de backtracking não necessitam ser armazenados para exame posterior.

Algumas das principais aplicações do cut são as seguintes:

- Unificação de padrões, de forma que quando um padrão é encontrado os outros padrões possíveis são descartados
- Na implementação da negação como regra de falha
- Para eliminar da árvore de pesquisa soluções alternativas quando uma só é suficiente
- Para encerrar a pesquisa quando a continuação iria conduzir a uma pesquisa infinita, etc.

Sintaticamente o uso do cut em uma cláusula tem a aparência de um objetivo sem nenhum argumento, representado por um ponto de exclamação "!".

Vamos estudar agora o comportamento de um pequeno programa que realiza algum backtracking desnecessário. Identificaremos onde isso ocorre e mostraremos como a eficiência do programa pode ser melhorada. Considere a função cujo gráfico é apresentado na Figura 6.2.

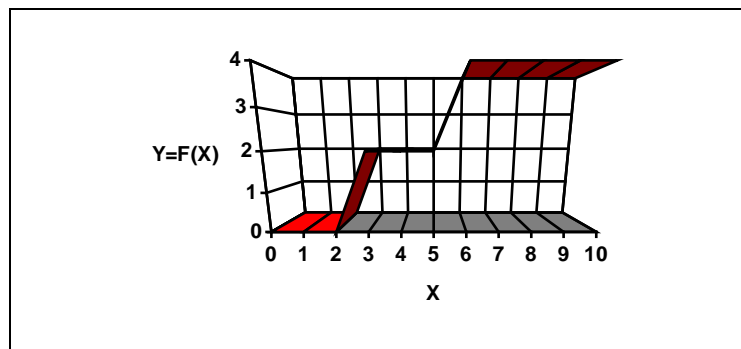


Figura 6.2 Uma função em degraus

A relação entre X e Y para a função apresentada na figura acima pode ser especificada, para o domínio dos inteiros não negativos, por meio de três regras:

- (1) Se $X < 3$, então $Y = 0$
- (2) Se $3 \leq X$ e $X < 6$, então $Y = 2$
- (3) Se $6 \leq X$, então $Y = 4$

que podem ser escritas em Prolog como uma relação binária $f(X, Y)$, como se segue:

```
f(x, 0) :- x < 3.  
f(x, 2) :- 3 =< x, x < 6.  
f(x, 4) :- 6 =< x.
```

Este programa assume que antes de $f(X, Y)$ ser avaliada, X deve obrigatoriamente estar instanciada para algum número, como é requerido pelos operadores de comparação. Faremos duas experiências com esse programa. Em cada uma delas será identificada uma fonte de ineficiência no programa, que será removida com o uso do cut.

6.2.1 EXCLUSÃO MÚTUA

Vamos analisar o que ocorre quando a seguinte questão é formulada:

```
?-f(1, Y), 2 < Y
```

Na execução do primeiro objetivo, $f(1, Y)$, Y é instanciada com 0, de forma que o segundo objetivo passa a ser $2 < 0$, que obviamente falha e, por meio de backtracking, conduz à avaliação das outras duas regras que, por sua vez, também irão falhar. Esse raciocínio é direto, entretanto, antes de tentar as duas últimas regras, já sabíamos (nós humanos) que elas não funcionariam. A execução completa é mostrada na Figura 6.3.

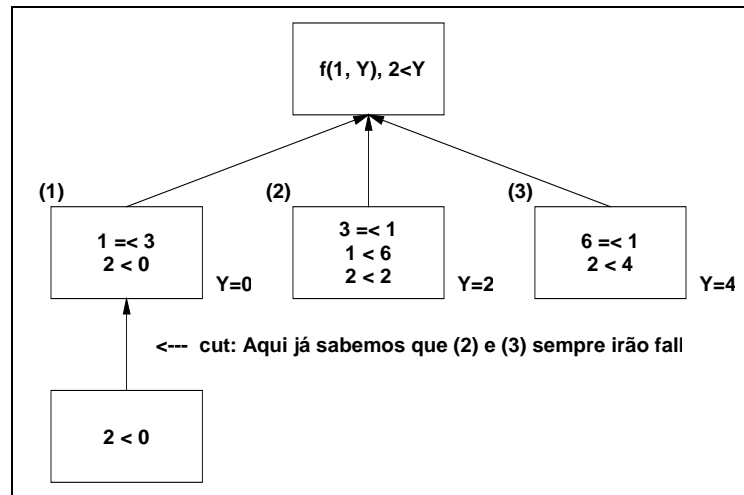


Figura 6.3 Execução da consulta $?-f(1, Y), 2 < Y$.

No exemplo apresentado na figura acima, no ponto indicado por "cut" no desdobramento da regra 1, já conhecemos o seu intervalo de aplicação e sabemos que, se este estiver correto e o restante da regra falhar, não há sentido em explorar outra alternativa. Para prevenir o sistema de apresentar um backtracking desnecessário, devemos indicar isto especificamente, o que é feito através do mecanismo de corte. Este é representado explicitamente por um "!" e é inserido entre os objetivos como uma espécie de pseudo-objetivo que sempre é bem sucedido quando ativado na direção top-down, mas que sempre falha quando é atingido através de backtracking, ocasionando ainda, como efeito colateral, a falha de todas as demais cláusulas do predicado onde o cut é declarado. O programa do exemplo, reescrito com cuts assume o seguinte aspecto:

```
f(x, 0) :- x < 3, !.
f(x, 2) :- 3 =< x, x < 6, !.
f(x, 4) :- 6 =< x.
```

Aqui o símbolo "!" evita o backtracking nos pontos em que aparece no programa. Se agora novamente fosse formulada a consulta $?-f(1, Y), 2 < Y$, o sistema Prolog iria inicialmente produzir o mesmo desvio mais à esquerda apresentado na Figura 6.3. O caminho produzido na árvore de pesquisa irá falhar no objetivo $2 < 0$. O Prolog irá então tentar o backtracking, mas não além do ponto marcado com um "!" no programa. Os desvios correspondentes às regras (2) e (3) não são gerados. O novo programa, equipado com cuts, é, em geral, de execução mais eficiente do que a versão original, que não possui cuts. Esta irá certamente produzir os mesmos resultados, apesar de ser menos eficiente. Pode-se dizer, neste caso, que a introdução de cuts afetou somente a interpretação operacional do programa, sem interferir na sua interpretação declarativa. Veremos a seguir que o uso do cut pode afetar também o significado declarativo do programa.

6.2.2 INTERFERINDO COM A INTERPRETAÇÃO DECLARATIVA

Efetuiremos uma segunda experiência, agora sobre a segunda versão do nosso programa. Seja a seguinte consulta, já acompanhada da solução:

```
?-f(7, Y).
Y=4
```

Vamos analisar o que aconteceu. Todas as três regras foram tentadas antes da resposta ter sido obtida, produzindo a seguinte sequência de objetivos:

- (1) Tenta a regra (1): $7 < 3$ falha. Aciona o backtracking e tenta a regra (2). O cut não foi atingido.
- (2) Tenta a regra (2): $3 = 7$ é bem-sucedido, mas $7 < 6$ falha. Aciona o backtracking e tenta a regra (3). O cut não foi atingido.
- (3) Tenta a regra (3): $6 = 7$ é bem-sucedido.

A sequência permite identificar uma segunda fonte de ineficiência no programa. Primeiro é estabelecido que $X < 3$ não é verdadeiro, pois $7 < 3$ falha. O objetivo seguinte é $3 = X$ que, se o primeiro objetivo falhou, só pode ser verdadeiro, pois é a negação dele. Portanto, o segundo teste é redundante e o objetivo correspondente pode ser omitido. O mesmo pode ser dito do objetivo $6 = X$ na regra (3). Isso conduz a uma formulação ainda mais econômica do programa:

```
f(x, 0) :- x < 3, !.
f(x, 2) :- x < 6, !.
f(x, 4).
```

Este programa produz os mesmos resultados que a versão original, mas da forma mais eficiente vista até agora. O que aconteceria entretanto se os cuts fossem removidos? O programa fica:

```
f(x, 0) :- x < 3.
f(x, 2) :- x < 6.
f(x, 4).
```

que pode produzir múltiplas soluções, as quais nem sempre estarão corretas. Por exemplo:

```
?-f(1, Y).
Y=0; Y=2; Y=4;
não
```

É importante notar aqui que, diferentemente da segunda versão, na terceira os cuts não afetam somente o comportamento operacional do programa, mas também o seu significado declarativo. Uma idéia mais precisa do funcionamento do mecanismo de corte do Prolog é o seguinte:

Vamos denominar "objetivo pai" o objetivo que unifica com a cabeça da cláusula que contém o cut. Quando o cut é encontrado, como um objetivo, ele é sempre bem-sucedido, mas elimina do sistema a pesquisa via backtracking de todas as cláusulas entre o objetivo pai e o cut.

Por exemplo, considere-se a cláusula:

```
H :- B1, B2, ..., Bm, !, ..., Bn.
```

Vamos assumir que ela tenha sido ativada por um objetivo G, que unifica com H. Então G é um objetivo pai. No momento em que o cut é encontrado o sistema já possui alguma solução para os objetivos B1, ..., Bm. Quando o cut é executado, a solução para B1, ..., Bm fica "congelada" e todas as demais soluções possíveis são descartadas. Além disso, o objetivo G agora passa a se limitar a *essa* cláusula. Qualquer tentativa de unificar G com com a cabeça de alguma outra cláusula fica impedida de se realizar. Vamos aplicar tais regras ao seguinte exemplo:

```
C :- P, Q, R, !, S, T, U.
C :- V.
A :- B, C, D.

?-A.
```

onde A, B, C, etc. possuem a sintaxe de termos Prolog. O cut irá afetar a execução do objetivo C da seguinte maneira: O backtracking é possível na sequência de objetivos P, Q, R, entretanto, tão logo o cut é alcançado todas as soluções alternativas para os objetivos P, Q, R são descartadas. A cláusula alternativa para C, $C \neg V$, também é descartada, entretanto, o backtracking ainda é possível na lista de objetivos S, T, U.

O objetivo pai da cláusula contendo o cut é C em $A \neg B, C, D$. Portanto o cut irá afetar somente a

execução de C, sendo completamente invisível do ponto de vista de A. Assim o backtracking automático continua ativo independentemente do cut na cláusula usada para satisfazer o objetivo C.

6.3 APLICAÇÕES DO CUT

Apresenta-se nesta seção alguns exemplos de pequenas aplicações empregando o operador cut, visando ilustrar o seu uso em programas reais.

6.3.1 MÁXIMO DE DOIS NÚMEROS

O procedimento para encontrar o maior de dois números pode ser programado como uma relação $\text{max}(X, Y, \text{Max})$, onde $\text{Max}=X$ se X for maior ou igual a Y e $\text{Max}=Y$ se este for maior que X . Isto pode ser escrito em Prolog por meio das seguintes cláusulas:

```
max(X, Y, X) :- X >= Y.  
max(X, Y, Y) :- X < Y.
```

Essas duas regras são mutuamente exclusivas. Se a primeira for bem sucedida, então a segunda certamente irá falhar e vice-versa. Portanto uma forma mais econômica de representar o mesmo programa com o uso do cut seria:

```
max(X, Y, X) :- X >= Y, !.  
max(X, Y, Y).
```

6.3.2 SOLUÇÃO ÚNICA PARA A OCORRÊNCIA

No capítulo anterior definiu-se a relação $\text{membro}(X, L)$ para estabelecer se X está presente na lista L . O programa era:

```
membro(X, [_|_]).  
membro(X, [_|Y]) :-  
    membro(X, Y).
```

Essa solução é não-determinística. Se X ocorre várias vezes, então qualquer ocorrência pode ser encontrada. Vamos agora mudar o predicado $\text{membro}/2$, tornando-o um programa determinístico que irá sempre encontrar a *primeira* ocorrência de X . A modificação a fazer é simples: apenas evitamos o backtracking tão logo X tenha sido encontrado, o que acontece quando a primeira cláusula é bem-sucedida. O programa modificado fica:

```
membro(X, [_|_]) :- !.  
membro(X, [_|Y]) :-  
    membro(X, Y).
```

e agora somente a primeira solução será encontrada. Por exemplo:

```
?-membro(X, [a, b, c, d]).  
X=a;  
não
```

6.3.3 ADIÇÃO DE ELEMENTOS SEM DUPLICAÇÃO

Freqüentemente deseja-se adicionar um item X a uma lista L de modo que X seja adicionado a L somente se X ainda não estiver em L . Se X já estiver em L , então a lista permanecerá a mesma, porque não desejamos a duplicação dos elementos em L . Uma adição $\text{adicionar}(X, L, L1)$, com essas características, pode ser formulada da seguinte maneira:

*Se X é membro da lista L , então $L1 = L$
senão, $L1$ é igual a L com a inserção de X na cabeça.*

É mais fácil inserir X como primeiro elemento, de modo que X se torna a cabeça de L1 quando se verifica a sua ausência na lista. Em Prolog escreve-se:

```
adicionar(X, L, L) :-
    membro(X, L), !.
adicionar(X, L, L1).
```

O comportamento desse programa pode ser ilustrado pelos exemplos abaixo:

```
?-adicionar(a, [b,c], L).
L=[a, b, c]

?-adicionar(X, [b,c], L).
X=b L=[b, c]

?-adicionar(a, [b, c, X], L).
X=a L=[b, c, a]
```

Esse exemplo é instrutivo, porque não é possível programar facilmente a "adição sem duplicatas" sem o uso do cut ou de alguma outra construção dele derivada. Portanto o cut é necessário aqui para especificar a relação correta e não somente para incrementar a eficiência do programa.

6.3.4 IF-THEN-ELSE

A programação procedimental estruturada pode ser simulada em Prolog. Neste exemplo a estrutura if-then-else é descrita através da relação:

```
ifThenElse(X, Y, Z)
```

que deve ser interpretada da seguinte maneira: "Se X for verdadeiro, então execute Y, senão execute Z". O programa Prolog é:

```
ifThenElse(X, Y, _) :- X, !, Y.
ifThenElse(_, _, Z) :- Z.
```

Deve-se notar que este programa emprega meta-variáveis (variáveis que podem ser instanciadas com chamadas a predicados), o que não é diretamente representado em todas as implementações Prolog. Em sendo possível, o exemplo abaixo ilustra a utilização de tal programa:

```
?-ifThenElse(X, Y is Z+1, Y is 0).
```

Aqui, se o predicado representado por X for verdadeiro, a variável Y será instanciada com 32 0 Td (t)Tj 32 (n)

especial, "fail", que sempre falha, forçando o objetivo pai a falhar. A formulação acima pode ser dada em Prolog com o uso do fail da seguinte maneira:

```
gosta(maria, X) :-  
    cobra(X), !, fail.  
gosta(maria, X) :-  
    animal(X).
```

Aqui a primeira regra se encarrega das cobras. Se X é uma cobra, então o cut evita o backtracking (assim excluindo a segunda regra) e o fail irá ocasionar a falha da cláusula. As duas regras podem ser escritas de modo mais compacto como uma única cláusula, por meio do uso do conetivo ";":

```
gosta(maria, X) :-  
    cobra(X), !, fail;  
    animal(X).
```

Pode-se usar essa mesma idéia para definir a relação diferente(X, Y) que, se for verdadeira, significa que X e Y não unificam. Isto pode ser formulado por:

```
Se X e Y unificam  
então diferente(X, Y) falha  
senão diferente(X, Y) é bem-sucedido.
```

Em Prolog:

```
diferente(X, X) :- !, fail.  
diferente(X, Y).
```

que também pode ser escrito sob a forma de uma só cláusula:

```
diferente(X, Y) :- X=Y, !, fail; true.
```

onde "true" é um objetivo pré-definido que sempre é bem-sucedido. Esses exemplos indicam que seria útil dispor de um objetivo unário "not" tal que not(Objetivo) seja verdadeiro se Objetivo não for verdadeiro. A relação not/1 pode ser definida da seguinte maneira:

```
Se Objetivo é bem-sucedido  
então not(Objetivo) falha  
senão not(Objetivo) é bem-sucedido.
```

que pode ser escrita em Prolog como:

```
not(P) :- P, !, fail; true.
```

A relação not/1 é pré definida na maioria das implementações Prolog e se comporta como o procedimento apresentado acima. Vamos assumir ainda, como ocorre na maioria das vezes, que o not seja definido como um operador prefixo, de modo que podemos escrever not(cobra(X)) como not cobra(X).

Deve ser notado que a relação not/1, definida como negação por falha, como foi feito, não corresponde exatamente à negação da lógica matemática. Essa diferença pode ocasionar um comportamento inesperado do programa, se o not for usado sem cuidado. Apesar disso, o not é um instrumento muito útil e pode ser utilizado com vantagem no lugar do cut. Os dois exemplos dados anteriormente poderiam ser escritos com o uso do not da seguinte maneira:

```
gosta(maria, X) :-  
    animal(X), not cobra(X).  
  
diferente(X, Y) :-  
    not (X = Y).
```

que certamente são formulações melhores que as anteriores. São mais naturais e mais fáceis de ler.

6.5 CUIDADOS COM O CUT E A NEGAÇÃO

As vantagens e desvantagens do uso do cut foram ilustradas por meio de exemplos nas seções anteriores. Vamos resumir primeiro as vantagens:

- Por meio do cut podemos freqüentemente aumentar a eficiência dos programas Prolog. A idéia é dizer explicitamente ao sistema: "Não tente outras alternativas pois elas estão destinadas a falhar".
- Usando o cut podemos especificar regras mutuamente exclusivas, expressas na forma:

`Se P então Q senão R`

realçando desta maneira a expressividade da linguagem Prolog.

As reservas ao uso do cut vem do fato que podemos perder a valiosa correspondência entre o significado declarativo e a interpretação operacional do programa. Se não houver cuts no programa, podemos trocar a ordem das cláusulas e objetivos de modo que isso irá afetar apenas a eficiência do programa e não o seu significado declarativo. Por outro lado, em programas com cuts, uma modificação na ordem das cláusulas pode afetar o significado declarativo, conduzindo a resultados inesperados. O ponto importante aqui é que, quando se emprega o recurso do cut, deve-se atentar para os aspectos operacionais envolvidos. Infelizmente essa dificuldade adicional aumenta a possibilidade de erro no programa.

Nos exemplos dados nas seções anteriores viu-se que em alguns casos a remoção dos cuts podia alterar o significado declarativo do programa. Em outros casos, entretanto, isso não ocorria, ou seja, o emprego de cuts não ocasionava nenhum efeito sobre o significado declarativo. O uso de cuts desse último tipo é menos delicado e por vezes estes são denominados "cuts verdes". Do ponto de vista da legibilidade dos programas os cuts verdes são "inocentes" e o seu uso é bastante aceitável. Na leitura dos programas os cuts verdes podem ser simplesmente ignorados.

Ao contrário, os cuts que afetam o significado declarativo são denominados "cuts vermelhos" e são os que tornam os programas difíceis de serem lidos, devendo ser empregados com especial cuidado.

Os cuts são freqüentemente utilizados em combinação com o predicado pré-definido fail/0. Em particular, definimos a negação de um objetivo (not) como sendo a falha deste objetivo. A negação assim definida corresponde a uma forma mais restrita do uso do cut. Por razões de clareza deve-se preferir o uso do operador not ao invés da combinação cut-fail sempre que possível, porque a negação é um conceito de nível mais elevado e é entendida de forma intuitiva mais claramente do que a combinação cut-fail.

Deve-se notar ainda que o uso do not pode também ocasionar alguns problemas e, portanto, esse operador deve também ser usado com cuidado. O problema é que o not é definido em Prolog como "negação por falha", que não corresponde exatamente à negação da lógica matemática. Se perguntarmos ao sistema:

`?-not humano(joão).`

a resposta será possivelmente "sim", entretanto isso não deve ser entendido como se o Prolog estivesse dizendo que "joão não é humano", mas na verdade que "não há informação suficiente no programa que permita provar que joão é humano". Isso acontece porque no processamento do objetivo not/1 o Prolog não tenta prová-lo diretamente. Ao invés disso ele tenta provar o oposto e, se o oposto não pode ser provado, então ele assume que o objetivo not /1 é bem-sucedido.

Tal raciocínio é baseado na denominada "Hipótese do Mundo Fechado". Segundo tal hipótese, o mundo é fechado no sentido que "tudo o que existe está no programa ou pode ser dele derivado". Assim, se alguma coisa não está no programa (ou não pode ser dele derivada), então não é verdadeira e consequentemente a sua negação é verdadeira. Isso demanda cuidados especiais por parte do programador, uma vez que normalmente não se assume que "o mundo é fechado", isto é, não colocando explicitamente a cláusula humano(joão), não se estava querendo dizer que "joão não é humano".

Finalmente, considere o seguinte programa:

```

r(a).
q(b).
p(X) :- not r(X).

```

Se consultarmos tal programa com:

```
?-q(X), p(X).
```

o sistema Prolog responderá $X=b$, entretanto se a mesma consulta fosse formulada de modo "aparentemente" equivalente:

```
?-p(X), q(X).
```

a resposta seria "não". Convidamos o leitor a estabelecer o "trace" do programa de modo a entender porque obtivemos respostas diferentes. A diferença chave entre as duas consultas reside no fato de que, no primeiro caso., a variável X já está instanciada quando $p(X)$ é executado, o que não ocorre no segundo caso.

RESUMO

- O uso do cut evita o backtracking. Esse recurso é empregado tanto para aumentar a eficiência dos programas quanto para realçar a sua expressividade;
- A eficiência é aumentada dizendo explicitamente ao Prolog, por meio do cut, para não explorar alternativas adicionais porque estas estão fadadas ao fracasso;
- Por meio do cut é possível formular conclusões mutuamente exclusivas por meio de regras da forma:

```
Se Condição então Conclusão1 senão Conclusão2;
```

- O cut torna possível introduzir a "negação por falha": $\text{not } X$ é definido em função da falha de X ;
- Dois predicados especiais pré-definidos são de grande utilidade em certos casos: o `true/0` que sempre é bem sucedido e o `fail/0` que sempre falha;
- Há alguma reserva quanto ao uso do cut. Sua inserção em um programa pode destruir a correspondência entre os significados declarativo e operacional. Um bom estilo de programação deve dar preferência ao uso de "cuts verdes", que não afetam o significado declarativo do programa, evitando os "cuts vermelhos" que o fazem;
- O operador unário `not` define uma forma particular de negação denominada "negação por falha", que não corresponde exatamente à negação da lógica matemática, de modo que o seu uso também requer cuidados especiais.

EXERCÍCIOS

6.1 Seja o seguinte programa Prolog:

```

p(1).
p(2) :- !.
p(3).

```

Escreva todas as respostas do sistema Prolog para as seguintes consultas:

- (a) `?-p(X).`
- (b) `?-p(X), p(Y).`
- (c) `?-p(X), !, p(Y).`

6.2 A seguinte relação classifica números em três classes: positivo, nulo ou negativo:

```

classe(N, positivo) :- N > 0.
classe(0, nulo).
classe(N, negativo) :- N < 0.

```


Defina este procedimento de uma forma mais eficiente usando cuts.

6.3 Escreva um programa denominado

```
reparte(Números, Positivos, Negativos).
```

que reparte uma lista de números em duas listas: uma de números positivos (incluindo o zero) e outra de números negativos. Por exemplo:

```
reparte([3,-1,0,5,-2], [3,0,5], [-1,-2]).
```

Proponha duas versões: uma com um único cut e outra sem nenhum.

6.4 Defina o predicado:

```
unificável(Lista1, Termo, Lista2)
```

onde Lista2 é a lista de todos os elementos de Lista1 que unificam com Termo, deixando-os não instanciados na resposta. Por exemplo:

```
?-unificável([X, b, t(Y)], t(a), Lista).  
Lista=[X, t(Y)]
```

Note que X e Y devem permanecer não-instanciados, apesar de a unificação com t(a) causar sua instanciação. Dica: use not (Termo1=Termo2). Se Termo1=Termo2 for bem-sucedido, então not (Termo1=Termo2) falha e a instanciação realizada é desfeita.

7. ESTRUTURAS DE DADOS

A possibilidade de empregar em Prolog estruturas de dados com unificação, backtracking e aritmética tornam essa linguagem de programação extremamente poderosa. No presente capítulo estudaremos estruturas de dados complexas por meio de exemplos de programas: recuperação de informação estruturada em uma base de dados, a simulação de um autômato não-determinístico e um planejamento de roteiros de viagens. Também se introduzirá o conceito de abstração de dados em Prolog.

7.1 RECUPERAÇÃO DE INFORMAÇÕES

O exercício apresentado a seguir desenvolve a habilidade de representar e estruturar objetos de dados e também ilustra a visão do Prolog como uma linguagem natural de consulta a bases de dados. Considere a figura 7.1.

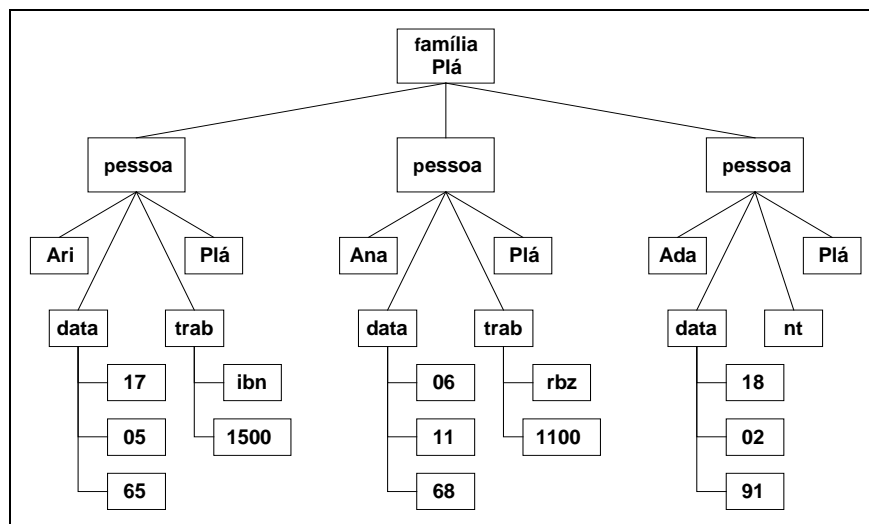


Figura 7.1 Informação estruturada sobre uma família

Uma base de dados pode ser naturalmente representada em Prolog como um conjunto de fatos. Por exemplo, uma base de dados sobre famílias pode ser representada de modo que cada família seja descrita como um termo. A Figura 7.1 mostra como a informação sobre cada família pode ser estruturada em um termo família/3, com a seguinte forma:

`família(Pai, Mãe, Filhos)`

onde Pai e Mãe são pessoas e Filhos é uma lista de pessoas. Cada pessoa é, por sua vez, representada por uma estrutura com quatro componentes: nome, sobrenome, data de nascimento e trabalho. A data de nascimento é fornecida como um termo estruturado data(Dia, Mes, Ano). O trabalho, ou é fornecido por um termo trab(Empresa, Salário), ou pela constante nt, indicando que a pessoa em questão não trabalha. A família exemplificada pode então ser armazenada na base de dados como uma cláusula do tipo:

```
família(pessoa(ari, plá, data(17,05,65), trab(ibn,1500)),
        pessoa(ana, plá, data(06,11,68), trab(rbs,1100)),
        [pessoa(ada, plá, data(18,02,91), nt)] )
```

A base de dados poderia ser vista então como uma sequência de fatos, descrevendo todas as famílias que interessam ao programa. A linguagem Prolog é, na verdade, muito adequada para a recuperação da informação desejada a partir de uma base de dados. Um detalhe muito interessante é que os objetos desejados não precisam ser completamente especificados. Pode-se simplesmente indicar a estrutura

dos objetos que interessam e deixar os componentes particulares apenas indicados. Por exemplo, se queremos recuperar todas as famílias "Oliveira", basta especificar:

```
?-família(pessoa(_, oliveira, _, _), _, _).
```

ou as famílias cujas mães não trabalham:

```
?-família(_, pessoa(_, _, _, nt), _).
```

as famílias que não possuem filhos:

```
?-família(_, _, []).
```

ou ainda famílias que possuem três ou mais filhos:

```
?-família(_, _, [_|_|_|_]).
```

As possibilidades de consulta são as mais diversas. Com esses exemplos queremos demonstrar que é possível especificar os objetos de interesse, não pelo seu conteúdo, mas sim pela sua estrutura, sobre a qual restringimos os componentes conforme nossas necessidades e/ou disponibilidades, deixando os demais indefinidos. Na Figura 7.2 é apresentado um programa demonstrando algumas das relações que podem ser estabelecidas em função de uma base de dados estruturada na forma definida por família/3:

```
pai(X) :-
    família(X, _, _).

mãe(X) :-
    família(_, X, _).

filho(X) :-
    família(_, _, Filhos),
    membro(X, Filhos).

membro(X, [X|_]).
membro(X, [_|Y]) :-
    membro(X, Y).

existe(Pessoa) :-
    pai(Pessoa);
    mãe(Pessoa);
    filho(Pessoa).

nasceu(pessoa(_, _, Data, _), Data).

salário(pessoa(_, _, _, trab(_,S)), S).
salário(pessoa(_, _, _, nt), 0).
```

Figura 7.2 Um programa baseado na relação família/3

Algumas aplicações para os procedimentos mostrados na figura acima podem ser encontrados nas seguintes consultas à base de dados:

- Achar o nome e sobrenome de todas as pessoas existentes na base de dados:

```
?-existe(pessoa(Nome, Sobrenome, _, _)).
```

- Achar todas as crianças nascidas em 1993:

```
?-filho(X), nasceu(X, data(_,_,93)).
```

- Achar todas as pessoas desempregadas que nasceram antes de 1976:

```
?-existe(pessoa(_, _, data(_,_,A), nt), A < 76).
```

- Achar as pessoas nascidas após 1965 cujo salário é maior do que 5000:

```
?- existe(Pessoa),
    nasceu(Pessoa, data(_,_,A)),
    A > 65,
    salário(Pessoa, Salário),
    Salário > 5000.
```

Para calcular o total da renda familiar, pode ser útil definir a soma dos salários de uma lista de pessoas como uma relação de dois argumentos:

```
total(L, T)
```

que pode ser declarada em Prolog como mostrado abaixo:

```
total([], 0).
total([Pessoa | Lista], Total) :-
    salário(Pessoa, Salário)
    total(Lista, Soma),
    Total is Soma + Salário.
```

Esta relação nos permite interrogar a base de dados para saber a renda familiar de cada família:

```
?-família(Pai, Mãe, Filhos), total([Pai, Mãe | Filhos], RFam).
```

7.2 ABSTRAÇÃO DE DADOS

O conceito de "abstração de dados" pode ser entendido como um processo de organização de diversas peças de conhecimento ou informação em uma forma conceitualmente significativa. Cada uma dessas unidades de informação deveria ser facilmente acessada no programa. Idealmente, todos os detalhes de implementação dessa estrutura deveriam ser invisíveis ao usuário. O programador pode então concentrar-se nos objetos e nas relações existentes entre eles. A idéia principal é permitir ao usuário o uso de informação complexa sem que seja necessário envolvê-lo com detalhes de representação. Discutiremos aqui uma forma de utilizar esse princípio.

Considere novamente o exemplo dado para a caracterização de uma família na seção anterior. Cada família é uma coleção de peças de informação. Tais peças ficam armazenadas em unidades naturais, como pessoa/4 ou família/3, de modo que podem ser tratadas como objetos simples. Assuma novamente que a informação sobre uma determinada família se estruture na forma apresentada na Figura 7.1. Vamos agora definir algumas relações através das quais o usuário pode acessar componentes particulares da estrutura família/3, sem conhecer os detalhes de particulares empregados na sua representação. Tais relações são denominadas "seletoras", uma vez que elas selecionam componentes particulares da estrutura sobre a qual se aplicam. Normalmente o nome de cada relação seletora será o próprio nome do objeto que ele seleciona e os seus argumentos serão dois: primeiro, o objeto que representa a estrutura da qual desejamos selecionar um determinado componente. Depois, o próprio componente a ser selecionado. Alguns exemplos de relações seletoras para a estrutura família/3 são mostrados a seguir:

```
pai(família(Pai, _, _), Pai).
mãe(família(_, Mãe, _), Mãe).
primogênito(família(_, _, [Prim | _]), Prim).
```

Outro objeto do qual podemos selecionar componentes é pessoa/4. Alguns exemplos são:

```
empresa(pessoa(_, _, _, trab(Empr, _)), Empr).
sobrenome(pessoa(_, Sobrenome, _, _), Sobrenome).
```

Uma vez que as relações seletoras estejam definidas, o usuário pode esquecer a forma particular usada na representação de sua estrutura original. Para criar e manipular tal informação é necessário somente conhecer os nomes das relações seletoras e empregar tais nomes ao longo do programa. No caso de representações complicadas, isso é muito mais simples do que usar a representação original de modo implícito. No exemplo da relação família/3, o usuário não precisa saber que os filhos são representados por uma lista.

O uso de relações seletoras também torna os programas mais fáceis de modificar. Suponha que fosse desejado aumentar a eficiência de um programa, mudando a forma de representar sua informação. Tudo que é necessário fazer é mudar as definições das relações seletoras e o restante do programa funcionará sem qualquer alteração com a nova representação.

7.3 UM AUTÔMATO FINITO NÃO-DETERMINÍSTICO

O exemplo apresentado na presente seção mostra como uma construção matemática abstrata pode ser descrita em Prolog. Além disso, o programa final resultante mostrará ser muito mais poderoso e flexível do que originalmente planejado. Um autômato finito não determinístico é uma máquina abstrata que lê, como entrada, um string de símbolos e decide se deve aceitar ou rejeitar o string lido. O autômato possui um certo número de estados e está sempre em um desses estados. O estado pode ser mudado pela troca de um estado para outro, em decorrência da situação em que o autômato se encontra. A estrutura interna de um autômato pode ser representada por um grafo de transição, como é mostrado na Figura 7.3.

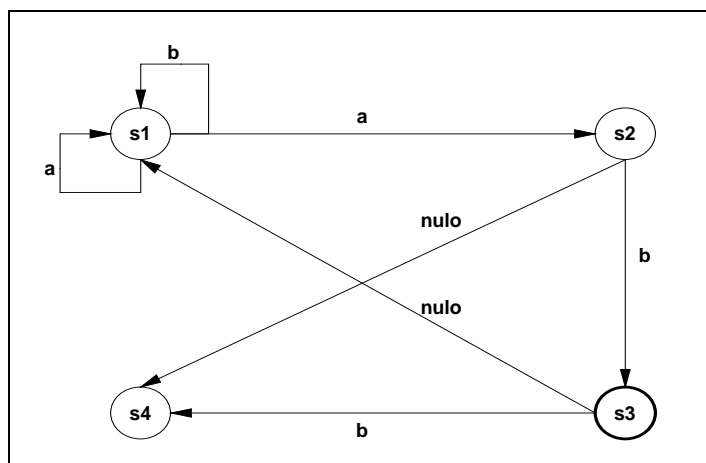


Figura 7.3 Um autômato finito não determinístico

No exemplo ali apresentado, s1, s2, s3 e s4 são os "estados" do autômato. A partir do estado inicial, s1 no exemplo dado, o autômato muda de estado para estado à medida em que vai lendo o string de entrada. As transições de estado do autômato dependem do símbolo de entrada correntemente lido, conforme indicado pelas legendas dos arcos no grafo de transição.

Uma transição ocorre toda vez que um símbolo do string de entrada é lido. Note que a transição, como representada na Figura 7.3 é não-determinística. Se o autômato estiver em s1, e o símbolo de entrada é "a", então a transição pode ser realizada tanto para s1 quanto para s2. Alguns arcos são rotulados como "nulo" para denotar o "símbolo nulo". Tais arcos correspondem ao que se denomina "movimentos silenciosos" do autômato. Esses são denominados "silenciosos" porque eles ocorrem sem que haja qualquer leitura de símbolos a partir do string de entrada e o observador, visualizando o autômato como uma "caixa-preta" não é capaz de notar que uma transição de estado ocorreu. O estado s3 é representado em negrito para denotar que este é um "estado terminal", onde é possível encerrar a ação do autômato. Dizemos que o autômato "aceitou" o string de entrada se há um caminho de transições no grafo tal que:

- (1) Começa no estado inicial,
- (2) Termina no estado final, e
- (3) As legendas dos arcos ao longo do caminho de transições correspondem ao string de entrada.

Fica inteiramente a critério do autômato decidir quais das possíveis transições serão executadas num dado instante. Em particular, o autômato pode escolher entre realizar ou não um movimento silencioso, se este for possível a partir do estado corrente. Os autômatos abstratos não-determinísticos desse tipo possuem ainda uma propriedade "mágica": se há possibilidade de uma escolha ocorrer, esta é feita do modo "correto", isto é, de um modo que conduza à aceitação do string de entrada, se tal modo existir. O autômato da Figura 7.3 irá, por exemplo, aceitar os strings "ab" e "aabaab", mas irá rejeitar

os strings "abb" e "abba". É fácil demonstrar que o autômato aceita qualquer string que termina em "ab" e rejeita todos os demais. Autômatos como esse podem ser descritos por meio de três relações:

- (1) Uma relação unária, final/1, que define os estados finais do autômato;
- (2) Uma relação de três argumentos, trans/3, que define as transições de estado de forma que

`trans(s1, x, s2)`

significa que uma transição do estado S1 para o estado S2 é possível quando o símbolo de entrada X for lido;

- (3) Uma relação binária, silêncio(S1, S2), significando que um "movimento silencioso" é possível de S1 para S2.

Para o autômato apresentado na Figura 7.3 essas três relações podem ser formuladas em Prolog da seguinte maneira:

```
final(s3).
trans(s1, a, s1).
trans(s1, a, s2).
trans(s1, b, s1).
trans(s2, b, s3).
trans(s3, b, s4).

silêncio(s2, s4).
silêncio(s3, s1).
```

Representaremos os strings de entrada como listas, de modo que o string "aab" será representado por [a, a, b]. Dada a descrição do autômato, o simulador processará um determinado string de entrada e decidirá se este deve ser aceito ou rejeitado. Por definição, os autômatos não-determinísticos aceitam um dado string se, partindo de um estado inicial, após ter lido o string completo o autômato pode estar em seu estado final. O simulador é programado por meio de uma relação binária, aceita/2, que define a aceitação de um determinado string a partir de um estado inicial. Assim a relação

`aceita(Estado, String).`

é verdadeira se o autômato, a partir do de um estado inicial "Estado", aceita o string "String". A relação aceita/2 pode ser definida por meio de três cláusulas, que correspondem aos três casos seguintes:

- (1) O string vazio, [], é aceito a partir de um determinado estado S se S é um estado final;
- (2) Um string não-vazio é aceito a partir de um estado S, se a leitura do primeiro símbolo no string pode conduzir o autômato a algum estado S1 e o resto do string é aceito a partir de S1;
- (3) Um string é aceito a partir de um estado S, se o autômato pode realizar um movimento silencioso de S para S1, e então aceitar o string completo a partir de S1.

Esses três casos originam as seguintes cláusulas:

```
aceita(S, []) :-
    final(S).
aceita(S, [X | R]) :-
    trans(S, X, S1), aceita(S1, R).
aceita(S, L) :-
    silêncio(S, S1), aceita(S1, L).
```

Por meio dessa relação é possível perguntar se um determinado string é aceito pelo autômato. Por exemplo:

```
?-aceita(s1, [a, a, a, b]).
sim
```

Como foi visto anteriormente, os programas Prolog são frequentemente capazes de solucionar problemas mais gerais do que aqueles para os quais foram originalmente concebidos. No presente caso, podemos por exemplo perguntar ao simulador a partir de quais estados ele aceitaria um determinado string:

```
?-aceita(S, [a, b]).
S=s1; S=s3;
não
```

Outra possibilidade seria perguntar quais são os strings de três símbolos que são aceitos pelo autômato a partir de um determinado estado:

```
?-aceita(s1, [X, Y, Z]).
X=a Y=a Z=b;
X=b Y=a Z=b;
não
```

É possível ainda realizar diversos outros experimentos envolvendo questões ainda mais gerais, como por exemplo: "a partir de que estados o autômato aceitará strings de tamanho sete?", etc. Experimentos ainda mais complexos podem inclusive requerer modificações na estrutura do autômato, mudando as relações final/1, trans/3 e silêncio/2.

7.4 PLANEJAMENTO DE ROTEIROS AÉREOS

Na presente seção iremos construir um programa para auxiliar o planejamento de roteiros aéreos. Apesar de bastante simples, o programa será capaz de responder questões tais como:

- Em que dias da semana há vôos entre o Rio e Munique?
- Como se pode chegar a Tóquio partindo de Porto Alegre numa terça-feira?
- Tenho que visitar Montevideú, Buenos Aires e Assunção, partindo de Brasília numa terça-feira à noite e chegando ao Rio na sexta-feira para o fim-de-semana. Em que seqüência deve ser realizada a viagem de forma que eu não tenha de fazer mais de um vôo por dia?

O programa será desenvolvido em função de uma base de dados possuindo informações sobre os vôos, representada por meio de uma relação com três argumentos:

```
horário(Cidade1, Cidade2, ListaDeVôos).
```

onde ListaDeVôos é uma lista de termos estruturados na forma:

```
Partida/Chegada/CódVôo/DiasDaSemana
```

Partida e Chegada representam termos contendo os horários de partida, em Cidade1, e chegada em Cidade2. CódVôo é uma constante utilizada na identificação do vôo. DiasDaSemana é uma lista contendo os dias da semana em que o vôo é realizado, ou a constante "todos", significando que o vôo é realizado todos os dias. Uma cláusula da relação horário/3 poderia ser, por exemplo:

```
horário('porto alegre', miami, [12:30/21:00/vrg127/todos, 15:30/24:00/vrg911/[seg,qua,sex]]).
```

Os horários são representados como objetos estruturados com dois componentes, horas e minutos, separados por ":". O problema principal será encontrar uma rota exata entre duas cidades, partindo em um determinado dia da semana. Isso será programado como uma relação de quatro argumentos:

```
rota(Cidade1, Cidade2, Dia, Rota)
```

onde Rota é uma seqüência de vôos que satisfaz aos seguintes critérios:

- (1) O ponto de partida da Rota é Cidade1;
- (2) O ponto de chegada da Rota é Cidade2;
- (3) Todos os vôos são no mesmo dia Dia;
- (4) Todos os vôos em Rota estão na relação horário/3;
- (5) Há tempo suficiente para as transferências de vôo.

A rota é representada por uma lista de termos estruturados na forma:

```
De-Para : CódVôo : Partida
```

e serão empregados os seguintes predicados auxiliares:

- (1) vôo(Cidade1, Cidade2, Dia, CódVôo, Partida, Chegada): dizendo que há um vôo (CódVôo) entre Cidade1 e Cidade2, no dia da semana Dia, que parte no horário de Partida e chega no horário de Chegada;
- (2) partida(Rota, Hora): A partida da rota Rota ocorre na hora Hora;
- (3) transferência(Hora1, Hora2): Há pelo menos 40 minutos entre Hora1 e Hora2, que devem ser suficientes para a transferência entre dois vôos.

O problema de encontrar uma rota, dadas as condições apresentadas, é em muitos pontos semelhante à simulação de um autômato finito não-determinístico apresentada na seção anterior. Os pontos comuns aos dois problemas são:

- Os estados do autômato correspondem às cidades;
- Uma transição entre dois estados corresponde a um vôo entre duas cidades;
- A relação trans/3 do autômato corresponde à relação horário/3 do planejador de vôo;
- O simulador do autômato encontra um caminho no grafo de transição entre um estado inicial e um estado final. O planejador de vôo encontra uma rota entre a cidade de partida e a cidade destino da viagem.

Não é portanto surpresa que a relação rota/4 possa ser definida de maneira semelhante à relação aceita/2. Uma vez que agora não há "movimentos silenciosos", devemos nos concentrar em dois casos:

- (1) Vôo Direto: Se há um vôo direto entre as cidades C1 e C2, então a rota consiste em um único vôo:

```
rota(C1, C2, Dia, [C1-C2:CodVôo:Partida]) :-  
    vôo(C1,C2,Dia,CodVôo,Partida,Chegada).
```

- (2) Vôo com Conexões: A rota entre C1 e C2 consiste em: primeiro um vôo de C1 para alguma cidade intermediária, C3, seguida por uma rota entre C3 e C2. Além disso, deve haver tempo suficiente entre a chegada de um vôo e a partida do seguinte para a transferência de avião:

```
rota(C1,C2,Dia,[C1-C3:CodVôo1:Partida1 | Rota]) :-  
    rota(C2, C3, Dia, Rota),  
    vôo(C1, C3, Dia, CodVôo1, Partida1, Chegada1),  
    partida(Rota, Partida2),  
    transferência(Chegada1, Partida2).
```

As relações auxiliares vôo/6, partida/2 e transferência/2 são facilmente programadas e estão definidas juntamente com o programa completo de planejamento de roteiros aéreos, apresentado na Figura 7.4, onde também se encontra incluído um pequeno exemplo da base de dados construída com a relação horário/3.

O planejador de roteiros aéreos ali apresentado, apesar de extremamente simples pode resolver com eficiência o planejamento de rotas aéreas desde que a base de dados não seja demasiadamente grande. Para esses casos seria necessário um planejador mais eficiente que permitisse lidar com um número muito grande de rotas alternativas.


```

:- op(50, xfy, ':').
vão(C1, C2, Dia, NVôo, Part, Cheg) :-
    horário(C1, C2, LVôos),
    membro(Part/Cheg/NVôo/Dias, LVôos),
    diaV(Dia, Dias).
membro(X, [X | _]).
membro(X, [_ | Y]) :-
    membro(X, Y).
diaV(Dia, todos).
diaV(Dia, Dias) :-
    membro(Dia, Dias).
rota(C1, C2, Dia, [C1-C2:NVôo:Part]) :-
    vão(C1, C2, Dia, NVôo, Part, _).
rota(C1, C2, Dia, [C1-C3:NVôo1:Part1 | Rota]) :-
    rota(C3, C2, Dia, Rota),
    vão(C1, C3, Dia, NVôo1, Part1, Cheg1),
    partida(Rota, Part2),
    transferência(Cheg1, Part2).
partida([C1-C2:NVôo:Part | _], Part).
transferência(H1:M1; H2:M2) :-
    60 * (H2 - H1) + (M2 - M1) >= 40.
horário(poa, rio, [12:30/14:10/vrg501/todos]).
horário(rio, poa, [12:30/14:10/vrg502/todos]).
horário(rio, mtv, [14:00/16:45/vrg660/[seg,qua,sex]]).
horário(rio, bue, [15:00/18:00/aar601/todos]).
horário(rio, ass, [08:00/09:50/vrg915/todos]).
horário(rio, par, [10:30/20:45/afr333/todos]).
horário(rio, tok, [08:00/22:00/jar712/[ter,qui,sab]]).
horário(bue, rio, [10:00/13:30/aar180/todos]).
horário(mtv, rio, [17:00/19:30/vrg661/todos]).
horário(ass, rio, [17:00/19:00/vrg916/todos]).
horário(par, nyc, [07:00/15:00/pan379/todos]).

```

Figura 7.4: Um planejador de roteiros aéreos e um exemplo de base de dados

RESUMO

- Uma base de dados pode ser naturalmente representada em Prolog como um conjunto de fatos;
- Os mecanismos de consulta e unificação do Prolog podem ser usados com grande flexibilidade na recuperação de informação estruturada em uma base de dados. Adicionalmente, procedimentos utilitários podem ser facilmente desenvolvidos para melhorar a comunicação com a base de dados;
- O conceito de abstração de dados pode ser visto como uma técnica de programação que facilita o uso de estruturas de dados muito complexas e contribui para a legibilidade dos programas. É muito natural para a linguagem Prolog lidar com os princípios básicos da abstração de dados;
- Construções matemáticas abstratas, como os autômatos, podem frequentemente ser traduzidas diretamente para especificações executáveis em Prolog;
- O mesmo problema pode muitas vezes ser abordado de diversas maneiras distintas, pela variação de sua representação. A introdução de redundâncias nessa representação pode muitas vezes ocasionar economia de computação;
- Muitas vezes o passo chave para a solução de um problema é a generalização desse problema. Paradoxalmente, considerando-se o problema de forma mais abrangente, pode-se muitas vezes formular a solução de maneira mais fácil.

EXERCÍCIOS

7.1 Escreva as consultas necessárias para extrair as seguintes informações da base de dados "família":

- (a) As famílias que não tem filhos;

- (b) Todos os filhos que trabalham;
 - (c) As famílias em que o pai está desempregado;
 - (d) As crianças cujos pais possuem uma diferença de idade superior a 15 anos;
 - (e) As famílias cuja renda per capita é inferior a 1000.
- 7.2 Defina as seguintes relações sobre a base de dados "família":
- (a) `gêmeos(Filho1, Filho2)`, onde Filho1 e Filho2 são irmãos gêmeos;
 - (b) `enésimoFilho(N, Filho)`, onde Filho é o N^o filho de uma família.
- 7.3 Defina uma relação `aceita(Estado, String, Max)`, onde Max é o tamanho máximo do string String que pode ser aceito a partir do estado Estado do autômato apresentado na Figura 7.3.
- 7.4 Considere um tabuleiro de xadrez onde as casas são representadas por pares de coordenadas na forma X/Y, assumindo X e Y valores entre 1 e 8.
- (a) Defina a relação `salta(Casa1, Casa2)` de acordo com o movimento do cavalo no tabuleiro. Assuma que Casa1 está sempre instanciada para a posição corrente do cavalo e que Casa2 pode ou não estar instanciada;
 - (b) Defina a relação `caminho(Lista)`, onde Lista é uma lista de casas que representam um caminho válido para um cavalo em um tabuleiro vazio;
 - (c) Formular a consulta necessária para, empregando a relação `caminho/1` definida em (b), encontrar o caminho que o cavalo deve percorrer para, iniciando em uma casa qualquer, percorrer todas as casas do tabuleiro, encerrando o trajeto no mesmo ponto de partida.
- 7.5 Escreva a consulta necessária ao planejador de roteiros aéreos para definir como é possível, partindo de Porto Alegre numa segunda-feira, visitar Assunção, Buenos Aires e Montevideu, retornando a Porto Alegre na quinta-feira efetuando não mais que um vôo por dia.

8. ENTRADA E SAÍDA

Neste capítulo estudaremos alguns recursos embutidos, presentes na maioria das implementações Prolog, destinados à leitura e gravação de dados em arquivos. Tais recursos podem também ser empregados pelo usuário para a formatação de objetos no programa, de modo a atingir alguma representação externa desejada para tais objetos. Também introduziremos os recursos para a leitura de programas e para a construção e decomposição de átomos e termos.

8.1 ARQUIVOS DE DADOS

O método de comunicação entre o usuário e o programa que estivemos usando até agora consiste em consultas realizadas pelo usuário que são respondidas pelo programa por meio de instâncias de variáveis. Esse método é simples e prático e, apesar de sua simplicidade, é suficiente para obter a entrada e saída de informações. Muitas vezes, entretanto, tal método não é suficientemente adequado tendo em vista a sua rigidez. Extensões a esse método básico tornam-se necessárias nos seguintes casos:

- Entrada de dados sob forma diferente das consultas, por exemplo, sob a forma de sentenças em linguagem natural,
- Saída de informações em qualquer formato desejado, e
- Entrada e saída para qualquer arquivo periférico do computador e não somente para o terminal do usuário.

Predicados pré-definidos, construídos com o objetivo de apoiar tais intenções são dependentes de cada particular implementação da linguagem Prolog. Aqui se introduz um repertório simples, que se encontra presente na maioria dessas implementações, apesar disso, o manual específico do Prolog utilizado deve ser consultado para detalhes.

Inicialmente se estudará o problema de direcionar a entrada e saída de dados para arquivos e, depois, como os dados podem entrar e sair em diferentes formatos. A Figura 8.1 mostra uma situação geral onde um programa Prolog se comunica com diversos arquivos:

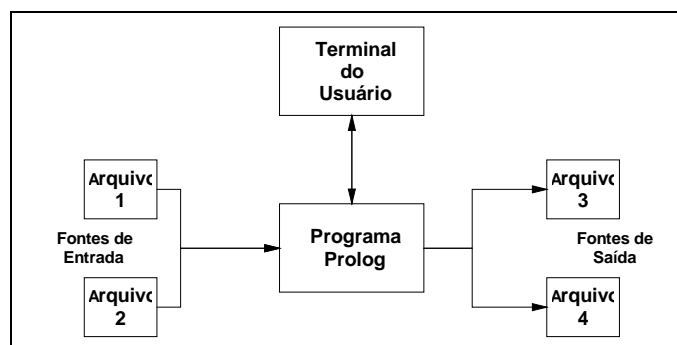


Figura 8.1: Comunicação entre um programa Prolog e diversos arquivos

Como pode ser visto na figura acima, o programa pode se comunicar com diversos arquivos, recebendo informações das denominadas "fontes de entrada" e transmitindo informações às denominadas "fontes de saída". Os dados que vem do terminal do usuário são tratados como uma outra fonte de entrada qualquer. Da mesma forma, os dados transmitidos ao terminal do usuário são tratados como uma fonte de saída. Esses dois pseudo-arquivos são nomeados pela constante "user". Os nomes dos outros arquivos podem ser escolhidos pelo programador de acordo com as regras adotadas em cada particular implementação.

A qualquer momento da execução de um programa Prolog, somente dois arquivos estão ativos: um para entrada e outro para saída. Esses dois arquivos se denominam respectivamente "fonte de entrada corrente" e "fonte de saída corrente.. No início da execução essas duas fontes correspondem ao terminal do usuário. A fonte de entrada corrente pode ser mudada a qualquer momento para um outro arquivo qualquer, digamos "novoArqEnt", por meio do objetivo:

```
see(novoArqEnt).
```

Esse objetivo é sempre bem sucedido (a menos que haja alguma coisa errada com NovoArqEnt. Um exemplo típico de utilização do predicado see/1 é a seguinte sequência de objetivos, que lê alguma coisa de um certo arquivo, "arq1", e então retorna ao terminal do usuário:

```
...
see(arq1).
lê_do_arquivo(Informação).
see(user).
...
```

A fonte de saída corrente pode também ser mudada por um objetivo da forma:

```
tell(novoArqSai).
```

Uma sequência de objetivos para enviar alguma informação para "arq3" e depois redirecionar a saída para o terminal do usuário poderia ser:

```
...
tell(arq3).
grava_no_arquivo(Informação).
tell(user).
...
```

Dois outros predicados pré-definidos que devem ser mencionados aqui são seen/0 e told/0, cujo efeito é fechar os arquivos correntes de entrada e saída respectivamente.

Os arquivos podem ser processados somente na forma sequencial. nesse sentido, todos os arquivos se comportam da mesma maneira que o terminal do usuário. Cada requisição para a leitura de alguma coisa a partir de alguma fonte de entrada irá ocasionar a leitura a partir da posição corrente dessa fonte de entrada. Após a leitura, a posição corrente dessa fonte de entrada será, naturalmente, o próximo item que ainda não foi lido, de forma que uma nova requisição de leitura irá iniciar a ser executada iniciando nessa nova posição corrente. Se uma requisição de leitura é feita para o fim do arquivo, então a informação devolvida será a constante "end_of_file", indicando que o fim do arquivo foi atingido. Uma vez que alguma informação foi lida, não é possível lê-la novamente a menos que se retome a leitura do arquivo a partir do início.

A saída de informações ocorre de maneira similar. Cada requisição de saída irá adicionar a informação requisitada no final da fonte de saída corrente. Da mesma forma que na leitura, não é possível retornar e reescrever sobre a porção do arquivo que já foi escrita.

Todos os arquivos são do tipo "texto", isto é, arquivos de caracteres. Os caracteres podem ser letras, dígitos, ou de algum tipo especial. Alguns desses últimos são ditos ser "não-imprimíveis" porque quando são direcionados para o terminal do usuário eles não aparecem no vídeo. Podem, no entanto, possuir algum outro efeito como o espaçamento entre colunas e linhas, reposicionamento do cursor, etc.

Há duas maneiras diferentes de se utilizar os arquivos em Prolog, dependendo da forma que se deseja empregar para os dados. A primeira delas considera o caracter como o elemento básico do arquivo. Assim uma requisição de entrada ou saída ocasionará a leitura ou escrita de um único caracter. Os predicados pré-definidos para tratar essa modalidade de arquivo são get/1, get0/1 e put/1.

A outra forma de utilizar arquivos em Prolog é considerar unidades maiores de informação como elementos básicos de entrada e saída. Tais unidades são os termos Prolog. Assim, cada requisição de

entrada ou saída desse tipo irá ocasionar a transferência de um termo inteiro. Os predicados que executam a transferência de termos são `read/1` e `write/1`. Naturalmente, nesse caso, a informação deverá se encontrar numa forma que seja consistente com a sintaxe dos termos Prolog.

O tipo de organização a ser escolhido para um determinado arquivo depende naturalmente do problema que se está tentando resolver, entretanto, sempre que a especificação do problema permitir, iremos preferir trabalhar com arquivos de termos, que permitem a transferência de uma unidade significativa completa através de uma única requisição. Por outro lado, há problemas cuja natureza determina o emprego de alguma outra organização. Um exemplo é o processamento de sentenças em linguagem natural para, digamos, estabelecer um diálogo com o usuário. Em tais casos os arquivos deverão ser vistos como seqüências de caracteres, uma vez que a linguagem natural não pode, normalmente, ser reduzida para a forma de termos.

8.2 PROCESSAMENTO DE ARQUIVOS DE TERMOS

8.2.1 READ & WRITE

O predicado pré-definido `read/1` é usado para a leitura de termos a partir da fonte de entrada corrente. O objetivo

```
read(X)
```

irá ocasionar a leitura do próximo termo `T` que será unificado com `X`. Se `X` é uma variável, então, como resultado da leitura, `X` será instanciada com `T`. Se a unificação não for possível, então o objetivo `read(X)` irá falhar. O predicado `read/1` é determinístico, significando que, em caso de falha, não haverá backtracking para a leitura de outro termo. cada termo, no arquivo de entrada, deve ser seguido por um ponto e um espaço ou "carriage-return". Se `read(X)` é executado sobre o final do arquivo de entrada, então a variável `X` será instanciada com o termo "end_of_file".

O predicado pré-definido `write/1` fornece a saída de um termo. Assim o objetivo `write(X)` irá ocasionar a escrita do termo `X` sobre a fonte de entrada corrente. `X` será escrito com a mesma forma sintática padrão utilizada pelo Prolog na apresentação de termos. Um recurso muito útil do Prolog é que o predicado `write/1` "sabe" apresentar qualquer termo, independente de sua complexidade.

Há ainda dois predicados adicionais para a formatação da saída. Eles são usados para inserir espaços e linhas na fonte de saída. O objetivo `tab(N)` irá ocasionar a saída de "`N`" espaços. O predicado `nl/0` (sem argumentos) irá ocasionar o início de uma nova linha. os seguintes exemplos ilustram o uso dos procedimentos estudados. Vamos assumir que temos um procedimento que computa o cubo de um número dado:

```
cubo(N, C) :- C is N*N*N.
```

Suponha que desejamos empregá-lo para calcular os cubos de uma seqüência de números. Isso pode ser obtido por meio de uma seqüência de questões:

```
?-cubo(2, X).  
X=8  
  
?-cubo(5, Y).  
Y=125  
  
?-cubo(12, Z).  
Z=1728
```

Aqui, para cada número é necessário formular um objetivo completo. Vamos agora modificar o programa de forma a "interiorizar" a ação, tornando mais suave o interface com o usuário. O programa agora irá manter-se lendo um número e apresentando o seu cubo até que a constante "fim" seja lida da fonte de entrada.

```
cubo :-  
    read(X), processa(X).
```

```

processa(fim) :- !.
processa(N) :-
    C is N*N*N,
    write(C),
    cubo.

```

Esse é um programa cujo significado declarativo é difícil de formular, entretanto, a sua interpretação operacional é direta: "Para executar cubo/0, primeiro leia X e depois processe-o. Se X=fim, então, tudo já foi feito. Senão, calcule o cubo de X, escreva-o e chame recursivamente o procedimento cubo/0 para o processamento de mais valores. Por exemplo:

```

?-cubo.
2.
8
5.
25
12.
1728
fim.
sim

```

Os números 2, 5 e 12 (seguidos de "." e "enter") são digitados pelo usuário no teclado do terminal. Os outros números correspondem a saída do programa. Note que após cada número digitado pelo usuário deve haver um ponto, que seguido de um carriage-return (cr, enter, return ou ζ , na maioria dos terminais), sinaliza ao sistema o final de um termo.

O procedimento cubo/0 conduz então a interação entre o usuário e o programa. Em tais casos, é normalmente desejável que o programa, antes de ler um novo valor, sinalize ao usuário que está pronto a receber uma nova informação, e que talvez ainda torne explícito o tipo de informação que é esperado. Isso normalmente é realizado pelo envio de um sinal "prompt" - de "prontidão" - ao usuário, antes de efetuar a leitura. O procedimento cubo/0 seria modificado para algo como:

```

cubo :-
    write('Próximo valor: '),
    read(X),
    processa(X).

processa(fim) :- !.
processa(N) :-
    C is N*N*N,
    write('O cubo de '), write(N), write('é '),
    write(C), nl, cubo.

```

Um diálogo com essa nova versão do programa seria:

```

?-cubo.
Próximo valor: 5.
O cubo de 5 é 125
Próximo valor: 8.
O cubo de 8 é 512
Próximo valor: 12.
O cubo de 12 é 1728
Próximo valor: fim.
sim

```

Dependendo da implementação, uma requisição adicional (como "flush/0" para o descarregamento dos buffers de i/o) pode ser necessária após o comando de escrita do prompt para forçá-lo a aparecer na tela antes da leitura

8.2.2 ESCRIVENDO LISTAS

Paralelamente ao formato padrão que o Prolog possui para listas, há ainda diversas outras formas para a apresentação de listas que podem ser vantajosas em certos casos. vamos definir o procedimento escreveLista(L), que escreve a lista L na fonte de saída corrente, de modo que cada elemento de L seja escrito em uma nova linha:

```

escreveLista([]).
escreveLista([X | L]) :-
    write(X), nl, escreveLista(L).

```

Se tivermos uma lista de listas, uma forma natural de saída é escrever os elementos de cada lista em uma linha. Um exemplo é:

```

?-escreveLista2([[a, b, c], [d, e, f], [g, h, i]]).
a b c
d e f
g h i
sim

```

O procedimento `escreveLista2/1` que permite obter essa saída é:

```

escreveLista2([]).
escreveLista2([L | LL]) :-
    imprime(L), nl, escreveLista2(LL).

imprime([]).
imprime([X | L]) :-
    write(X), tab(1), imprime(L).

```

Uma lista de números inteiros pode algumas vezes ser convenientemente apresentada sob a forma de um gráfico de barras. O procedimento `barras(L)` irá escrever uma lista nessa forma. Um exemplo do seu uso seria:

```

?-barras([6, 7, 9, 12]).
□□□□□□
□□□□□□□
□□□□□□□□
□□□□□□□□□□
sim

```

```

escreveFam(família(Pai, Mãe, Filhos)) :-
    nl, nl, write('Pais:'), nl,
    escrevePes(Pai), nl, escrevePes(Mãe), nl,
    write('Filhos:'), nl,
    escrevePesList(Filhos).

escrevePes(pessoa(Nome, SNome, dat(D,M,A), Trab)) :-
    tab(10), write(Nome), tab(1), write(SNome),
    write(', nasc: '),
    write(D), write('/'), write(M), write('/'), write(A),
    write(','), escreveTrab(Trab).

escrevePesList([]).
escrevePesList([P | L]) :-
    escrevePes(P), nl, escrevePesList(L).

escreveTrab(NT) :-
    write('não trabalha').
escreveTrab(trab(Emp, Sal)) :-
    write('trab: '), write(Emp), write(', '),
    write('sal: '), write(Sal).

```

Figura 8.2 Um programa para a formatação do termo "família"

O procedimento `barras/1` pode ser definido da seguinte maneira, assumindo que a representação '□' seja válida no Prolog utilizado:

```

barras([]).
barras([N | L]) :-
    quadrinho(N), nl, barras(L).

quadrinho(N) :-
    N>0,
    write('□'), N1 is N-1, quadrinho(N1).
quadrinho(N) :-
    N=<0, !.

```

8.2.3 FORMATAÇÃO DE TERMOS

Vamos considerar novamente a representação sob a forma de termos usada para definir famílias, discutida na seção 7.1. Se uma variável F for instanciada com o termo cuja estrutura é mostrada na figura 7.1, o objetivo write(F) irá ocasionar a saída do termo correspondente no formato padrão do Prolog. Alguma coisa como:

```
familia(pessoa(ari, plá, data(17,05,65), trab(ibn,1500)),
pessoa(ana, plá, data(06,11,58), trab(rbz,1100)),
[pessoa(ada, plá, data(18,02,91), nt)])
```

O termo acima contém, sem dúvida, toda a informação, entretanto sob uma forma bastante confusa, tornando difícil seguir as partes da informação que formam as unidades semânticas. Iríamos, certamente, preferir que a informação fosse apresentada de outra maneira, por exemplo, na forma abaixo:

```
Pais:
    ari plá, nasc: 16/05/65, trab: ibn, sal: 1500
    ana plá, nasc: 06/11/68, trab: rbz, sal: 1100
Filhos:
    ada plá, nasc: 18/02/91, não trabalha.
```

Tal formato pode ser obtido por meio do procedimento escreveFam/1 mostrado na Figura 8.2.

8.2.4 PROCESSAMENTO DE ARQUIVOS DE TERMOS

Uma típica seqüência de objetivos para processar completamente um arquivo "A" se pareceria com o seguinte:

```
... see(A), processaArq, see(user), ...
```

Aqui processaArq/0 é um procedimento para ler e processar cada termo em A, um após o outro, até que o fim do arquivo seja encontrado. Um esquema típico para processaArq é o seguinte:

```
processaArq :-
    read(Termo), processa(Termo).
processa(end_of_file) :- !.
processa(Termo) :-
    trata(Termo), processaArq.
```

Aqui o procedimento trata/1 representa qualquer coisa que se deseje fazer com cada um dos termos presentes no arquivo. Um exemplo poderia ser um procedimento para apresentar no terminal cada um dos termos do arquivo, juntamente com o seu respectivo número de ordem. Vamos chamar tal procedimento mostraArq(N), onde N é um argumento adicional para contar os termos lidos.

```
mostraArq(N) :-
    read(Termo), mostra(1, Termo).

mostra(_, end_of_file) :- !.
mostra(N, Termo) :-
    write(N), tab(2), write(Termo),
    N1 is N+1,
    mostraArq(N1).
```

outro exemplo de utilização do esquema dado para o processamento de arquivos de termos é o seguinte: Temos um arquivo denominado "arq1" que contém termos na forma:

```
item(Nro, Descrição, Preço, Fornecedor)
```

Cada termo descreve uma entrada num catálogo de itens. Desejamos produzir um outro arquivo que contenha somente os itens fornecidos por um determinado fornecedor. Como o fornecedor nesse novo arquivo será sempre o mesmo, o seu nome somente precisa ser escrito no início do arquivo, sendo omitido nos demais termos. Denominaremos tal procedimento de

```
fazArq(Fornecedor)
```

Por exemplo, se o catálogo original é armazenado em arq1 e desejamos produzir um arquivo arq2 com todos os artigos fornecidos por "Palmeira & Cia", então usaremos o procedimento fazArq/1 da se-

guinte maneira:

```
..., see(arq1),tell(arq2),fazArq('Palmeira & Cia'),see(user),tell(user), ...
```

O procedimento fazArq/1 é apresentado na Figura 8.3

```
fazArq(F) :-  
    write(F), write('.'), nl, fazResto(F).  
  
fazResto(F) :-  
    read(Item), processa(Item, F).  
  
processa(end_of_file, _) :- !.  
processa(item(N, D, P, F), F) :-  
    !, write(item(N, D, P)), write('.'), nl, fazResto(F).  
processa(_, F) :-  
    fazResto(F).
```

Figura 8.3 Processando um arquivo de itens

Note que no programa acima, o predicado processa/2 grava um ponto após cada termo escrito em arq2, de modo a possibilitar leituras posteriores desse arquivo por meio do comando read/1.

8.3 PROCESSAMENTO DE CARACTERES

Um caracter é escrito na fonte de saída corrente por meio do objetivo:

```
put(C)
```

onde C é o código ASCII (um número entre 0 e 255) do caracter a ser escrito. Por exemplo, a consulta:

```
?-put(65), put(66), put(67).
```

produz a saída:

```
ABC
```

uma vez que 65 é o código ASCII de 'A', 66 de 'B' e 67 de 'C'. Por sua vez um caracter pode ser lido a partir da fonte de entrada corrente por meio do objetivo:

```
get0(C)
```

que ocasiona a leitura do caracter corrente e torna a variável C instanciada para com o código ASCII deste caracter. Uma variação do predicado get0/1 é o get/1, que é utilizado para a leitura apenas de caracteres imprimíveis, saltando sobre todos os caracteres não-imprimíveis, particularmente espaços em branco. Como um exemplo do uso de predicados que transferem caracteres, vamos definir um procedimento comprime/0 para ler uma sentença da fonte de entrada corrente e apresentar essa sentença reformatada, de forma que múltiplos espaços em branco entre as palavras sejam substituídos por um único espaço em branco (código ASCII = 32). Para simplificar, vamos assumir que toda sentença de entrada processada pelo procedimento comprime/0 termina com um ponto final (código ASCII = 46) e que as palavras estejam separadas por um ou mais espaços em branco e nenhum outro caracter. Uma entrada aceitável seria:

```
Genialidade é 1% de inspiração e 99% de transpiração.
```

para a qual o procedimento comprime/0 devolveria:

```
Genialidade é 1% de inspiração e 99% de transpiração.
```

O procedimento comprime/0 terá uma estrutura similar aos procedimentos para processamento de arquivos estudados nas seções anteriores. Inicialmente ele vai ler o primeiro caracter e enviá-lo à saída e então completar o processo, dependendo do caracter que for lido. A exclusão mútua entre as três alternativas é obtida por meio de cuts:

```
comprime :-  
    get0(C), put(C), continua(C).
```

```

continua(46) :- !.
continua(32) :-
    !, get(C), put(C), continua(C).
continua(_) :-
    comprime.

```

8.4 CONVERSÃO DE TERMOS

Frequentemente deseja-se trabalhar com informações que foram lidas sob a forma de caracteres, convertidas em termos como representação interna para processamento de entrada e saída. Há um predicado pré-definido, `name/2`, que pode ser usado com essa finalidade, relacionando os átomos com o seu código ASCII. Assim, `name(X, L)` é verdadeiro, se `L` é a lista dos códigos dos caracteres em `A`. Por exemplo, a assertiva abaixo é verdadeira:

```
name(zx232, [122, 120, 50, 51, 50])
```

Há dois usos típicos para o predicado `name/2`:

- Decompor um termo dado em seus caracteres, e
- Dada uma lista de caracteres, converte-la em um termo.

Um exemplo do primeiro tipo de aplicação seria a decomposição de átomos em átomos menores, com tamanho pré-definido. Suponhamos que recebemos, de alguma fonte de entrada, átomos de tamanho fixo de 13 caracteres, dos quais os oito primeiros correspondem ao CEP, os dois seguintes à unidade da federação (UF) e os três últimos à sigla internacional de cidade. Por exemplo:

```
90120040rspoa e 70605220dfbsb
```

e desejamos, para fins de processamento, separá-los nos sub-átomos:

```
90120040 rs poa e 70605220 df bsb
```

O predicado `separa/4`, abaixo, obtém o resultado desejado:

```

separa(A, S1, S2, S3) :-
    name(A, L),
    conc([S1, S2, S3], [], L),
    tam(S1, 8), !,
    tam(S2, 2), !,
    tam(S3, 3).

conc([], L, L).
conc([X | L1], L2, [X | L3]) :-
    conc(L1, L2, L3).

tam([], 0).
tam([_|R], N) :-
    tam(R, N1), N1 is N+1.

```

O próximo exemplo ilustra o uso da combinação de caracteres em átomos. Definiremos um predicado, `fazFrase(Lista)` que lê uma sentença em linguagem natural e instancia `Lista` com cada palavra da sentença representada por um átomo. Por exemplo, se a entrada fosse a seguinte frase, atribuída a Albert Einstein dirigindo-se a Sigmund Freud:

```
"No matter what mind is and never mind what matter is."
```

o objetivo `fazFrase(Lista)` ocasionaria a seguinte instanciação:

```
Lista = ['No',matter,what,mind,is,and,never,mind,what,matter,is]
```

para simplificar, assume-se que cada sentença termina com um ponto final e que não há símbolos de pontuação na sentença. O programa completo é mostrado na Figura 8.4. O procedimento `fazFrase/1` lê o caracter corrente, `C`, e então transmite esse caracter ao procedimento `fazResto` para completar o serviço.

```

fazFrase(Lista) :-
    get0(C), fazResto(C, Lista).

```

```

fazResto(46, []) :- !.
fazResto(32, Lista) :-
    !, fazFrase(Lista).
fazResto(Let, [Pal | Lista]) :-
    fazLetras(Let, Lets, Prox),
    name(Pal, Lets),
    fazResto(Prox, Lista).

fazLetras(46, [], 46) :- !.
fazLetras(32, [], 32) :- !.
fazLetras(Let, [Let | Lets], Prox) :-
    get0(C), fazLetras(C, Lets, Prox).

```

Figura 8.4: Transformando uma sentença em uma lista de palavras

O procedimento `fazResto/2`, na Figura 8.4, precisa considerar três casos:

- C é um ponto (ASCII=46). Então tudo já foi lido;
- C é um branco (ASCII=32). Então deve ser ignorado;
- C é uma letra. Primeiro ler a palavra `Pal`, que começa com C, e depois, ppor meio de `fazFrase/1`, ler o resto da sentença, produzindo `Lista`. O resultado cumulativo é `[Pal | Lista]`.

O procedimento que lê os caracteres de uma palavra é:

```
fazLetra(Let, Lets, Prox)
```

onde:

- (1) `Let` é a letra corrente (já lida) da palavra que está sendo processada,
- (2) `Lets` é a lista de letras (começando com `Let`), até o final da palavra, e
- (3) `Prox` é o caracter de entrada que imediatamente segue a palavra lida, podendo ser um branco ou um ponto.

O programa `fazFrase/1` pode ser usado para o processamento de textos em linguagem natural. As sentenças representadas como listas de palavras encontram-se em uma forma adequada para processamento adicional em Prolog. Um exemplo simples seria o tratamento de certas palavras do texto. Uma tarefa muito mais difícil seria "entender" a sentença, isto é, extrair dela o seu significado, representado por algum formalismo. Esta é uma importante área de pesquisa em inteligência artificial.

8.5 LEITURA DE PROGRAMAS

É possível carregar programas no sistema Prolog por meio de dois predicados pré-definidos: `consult/1` e `reconsult/1`. Diz-se ao Prolog para ler um programa que esteja contido em um arquivo "programa.log" da seguinte maneira:

```
?-consult('programa.log').
```

cujo efeito é a leitura de todas as cláusulas em `programa.log` de modo que estas possam ser usadas pelo sistema para responder as consultas que se seguirem. Se um outro arquivo for "consultado" durante a mesma seção, as cláusulas presentes nesse novo arquivo serão simplesmente adicionadas ao final do conjunto de cláusulas corrente. Não é necessário, entretanto, gravar nosso programa em um arquivo para depois carregá-lo no sistema. Ao invés de ler um arquivo o Prolog pode também aceitar o nosso programa diretamente do terminal, que corresponde ao pseudo-arquivo "user". Obtemos isso por meio de:

```
?-consult(user).
```

que leva o Prolog a aceitar cláusulas digitadas diretamente no teclado do terminal.

Uma notação mais curta para a carga de programas consiste em colocar os arquivos que devem ser

lidos em uma lista e declará-la como objetivo. Por exemplo:

```
?-[prog1, prog2, prog3].
```

que corresponde exatamente ao obtido por:

```
?-consult(prog1), consult(prog2), consult(prog3).
```

O predicado pré-definido `reconsult/1` opera de maneira semelhante ao `consult/1`. Um objetivo

```
?-reconsult(programa).
```

terá o mesmo efeito de `consult` com uma exceção: se houver cláusulas em "programa" sobre alguma relação já definida no sistema, a definição anterior será substituída pelas novas cláusulas presentes em "programa". A diferença entre `consult/1` e `reconsult/1` é que o primeiro sempre adiciona as novas cláusulas, ao passo que o segundo redefine as relações previamente definidas, sem afetar, entretanto, as relações para as quais não existem cláusulas em "programa".

RESUMO

- Entradas e saídas (além das efetuadas em consultas ao programa) são executadas por meio de predicados pré-definidos;
- Os arquivos são sequenciais. Há uma fonte de entrada corrente e uma fonte de saída corrente. O terminal do usuário é tratado como um arquivo denominado "user".
- A mudança entre fontes de entrada e de saída correntes /e efetuada pelos predicados:

```
see(A):  A se torna a fonte de entrada corrente
tell(A): A se torna a fonte de saída corrente
seen:    Fecha a fonte de entrada corrente
told:    Fecha a fonte de saída corrente
```

- Os arquivos são lidos ou gravados de dois modos diferentes: como uma sequência de caracteres ou como uma sequência de termos;
- Predicados pré-definidos para a leitura e escrita de termos e caracteres são:

```
read(Termo)
write(Termo)
put(Código)
get0(Código)
get(Código)
```

- Dois predicados utilizados para formatação são:

```
nl
tab(N)
```
- O procedimento `name(Átomo, Lista)` decompõe e constrói átomos. Lista é a lista dos códigos ASCII dos caracteres em Átomo.

EXERCÍCIOS

- 8.1 Seja `arq` um arquivo de termos. Defina um procedimento `achaTermo(Termo)` que apresenta no terminal do usuário o primeiro termo em `arq` que unifica com `Termo`.
- 8.2 Seja `arq` um arquivo de termos. Escreva um procedimento `achaTodos(Termo)` que apresenta no terminal todos os termos em `arq` que unificam com `Termo`.
- 8.3 Defina a relação `começaCom(Átomo, Caracter)`, para verificar se `Átomo` inicia com o caracter `Caracter`.
- 8.4 Escreva um procedimento `acha(PalavraChave, Sentença)` que irá, a cada vez que for chamado, localizar uma sentença na fonte de entrada corrente que contenha a palavra chave dada. A sen-

tença deve ser fornecida em sua forma original, representada como uma seqüência de caracteres ou como um átomo. O programa fazFrase/2 apresentado neste capítulo pode ser adequadamente modificado para atender as necessidades deste exercício.

- 8.5 Escreva um programa relatório/0 para ler um arquivo de termos na forma cliente(Nome, Endereço, Telefone) e produzir um relatório formatado da seguinte maneira:

NRO	CLIENTE	ENDEREÇO	TELEFONE
001	XXX...	XXX...	XXX....
002	XXX...	XXX...	XXX...
....

- 8.6 Escreva um programa, plural(Palavra, Plural), para a formação do plural de palavras em português. Crie para isso uma base de regras de formação do plural de palavras. O resultado esperado é, por exemplo:

```
?-plural(pássaro, X).  
X=pássaros
```

9. PREDICADOS EXTRALÓGICOS

Todas as implementações Prolog oferecem, em maior ou menor quantidade, um certo número de predicados pré-definidos orientados a execução de rotinas que, ou são necessárias com muita frequência, ou são de difícil programação, ou se destinam a um domínio particular realçado pela implementação, ou por todas essas razões em diferentes proporções. No presente capítulo se introduz alguns desses predicados, que facilitam muito a construção de programas interativos e orientados a aplicações concretas.

9.1 TIPOS DE TERMOS

Os termos Prolog podem assumir os mais diversos aspectos, desde simples constantes até estruturas complexas altamente elaboradas. Se um termo é uma variável, então esta pode ou não estar instanciada em algum momento da execução do programa. Além disso, se estiver instanciada, seu valor pode ser uma constante, uma estrutura, etc. Algumas vezes pode ser de utilidade para o programador identificar de que tipo é esse valor. Por exemplo, podemos querer adicionar os valores de duas variáveis, X e Y, por meio de:

`Z is X + Y`

Antes desse objetivo ser executado, X e Y devem ser instanciados com valores inteiros. Se não há certeza de que tal instanciação ocorreu, então deve-se fazer tal verificação antes de executar a operação aritmética envolvida.

Com essa finalidade podemos utilizar o predicado pré-definido `integer(X)`, que é verdadeiro se X estiver instanciada com um valor inteiro. O objetivo de adicionar X e Y então pode ser protegido da seguinte maneira, garantindo a validade dos operandos:

`..., integer(X), integer(Y), Z is X + Y, ...`

Se X e Y não estiverem ambas instanciadas com valores inteiros, então a operação aritmética que se segue ao teste não será realizada. Os predicados pré-definidos para a classificação de dados comuns a maioria das implementações são os seguintes:

Predicado	Descrição
<code>atom(X)</code>	É bem sucedido se X é uma constante textual (átomo).
<code>integer(X)</code>	É bem sucedido se X é um número inteiro.
<code>float(X)</code>	É bem sucedido se X é um número em ponto flutuante.
<code>number(X)</code>	É bem sucedido se X é um número.
<code>string(X)</code>	É bem sucedido se X é um string.
<code>atomic(X)</code>	É bem sucedido se X é do tipo atômico.
<code>var(X)</code>	É bem sucedido se X é uma variável não-instanciada.
<code>nonvar(X)</code>	É bem-sucedido se X não é uma variável ou se X é uma variável instanciada.

O programa `classifica/1`, apresentado na figura abaixo, ilustra o emprego de tais predicados.

O programa `classifica/1` (Figura 9.1) irá reconhecer o tipo do seu argumento, informando-o ao usuário. Em particular, se o dado é do tipo atômico, o subtipo também é informado, como é ilustrado abaixo:

```

?-X=1, classifica(X).
Tipo Atômico
----> Numero Inteiro

?-X=[], classifica(X).
Tipo Atômico
----> Lista Vazia

?-X=tio(josé), classifica(X).
Termo Estruturado

```

```

classifica(X) :-
    var(X), !, nl, write('Variável Não-instanciada').
classifica(X) :-
    atomic(X), !, nl, write('Tipo Atômico:'),
    tipoAtomico(X).
classifica([_|_]) :-
    !, nl, write('Lista').
classifica(X) :-
    nl, write('Termo Estruturado').

tipoAtomico([]) :-
    !, nl, tab(5), write('----> Lista Vazia').
tipoAtomico(X) :-
    atom(X), !, nl, tab(5), write('----> Átomo').
tipoAtomico(X) :-
    integer(X), !, nl, tab(5),
    write('----> Número Inteiro').
tipoAtomico(X) :-
    float(X), !, nl, tab(5),
    write('----> Número em Ponto Flutuante').
tipoAtomico(X) :-
    string(X), !, nl, tab(5), write('----> String').

```

Figura 9.1 Programa para classificar tipos de dados.

Vamos supor agora que se deseje contar quantas vezes um determinado átomo ocorre em uma lista de objetos dada. Com esse propósito se definirá o procedimento

`conta(A, L, N)`

onde A é o átomo, L é a lista e N é o número de vezes que A ocorre em L. Uma primeira tentativa de definir conta/3 seria:

```

conta(_, [], 0).
conta(A, [A | L], N) :-
    !, conta(A, L, N1), N is N1+1.
conta(A, [_ | L], N) :-
    conta(A, L, N).

```

Algumas tentativas de utilização de tal programa são:

```

?-conta(a, [a, b, a, a], N).
N=3

?-conta(a, [a, b, X, Y], Na).
X=a Y=a Na=3

?-conta(b, [a, b, X, Y], Nb).
X=b Y=b Nb=3

?-L=[a, b, X, Y], conta(a, L, Na), conta(b, L, Nb).
X=a Y=a Na=3 Nb=1

```

Neste último exemplo, X e Y foram ambas instanciadas com "a", e portanto obtivemos Nb=1 somente. Não era isso, entretanto que se tinha em mente na construção do procedimento conta/3. Na verdade o que se queria era o número real de ocorrências de um dado átomo e não o número de termos capazes de unificar com esse átomo. De acordo com essa definição mais precisa da relação conta/3, devemos verificar se a cabeça da lista é um átomo. A nova versão da relação conta é a seguinte:

```

conta(_, [], 0).
conta(A, [B | L], N) :-
    atom(B), A=B, !, conta(A, L, N1), N is N1+1.
conta(A, [_ | L], N) :-

```

```
conta(A, L, N).
```

9.2 CONSTRUÇÃO E DECOMPOSIÇÃO DE TERMOS

Há três predicados pré-definidos para a decomposição de termos e construção de novos termos: `functor/3`, `arg/3` e `=../2`. Estudaremos primeiro o `=../2`, também referido como "univ", que é definido como um operador infix. O objetivo

```
Termo =.. L
```

é bem-sucedido se `L` é uma lista contendo como primeiro elemento o functor principal de `Termo`, seguido pelos seus argumentos. Os seguintes exemplos dão uma idéia do seu funcionamento:

```
?-f(a, b) =.. L.
L=[f, a, b]

?-T =.. [retângulo, 3, 5].
T=retângulo(3, 5)

?-Z =.. [p, X, f(X, Y)].
Z=p(X, f(X, Y))
```

Para melhor ilustrar a utilidade do operador `=../2`, vamos considerar um programa que manipula figuras geométricas como quadrados, retângulos, triângulos, círculos, etc. Estas entidades podem ser representadas por meio de termos tais que o functor principal indica o tipo de figura e os argumentos especificam o tamanho da figura, como em:

```
quadrado(Lado)
triângulo(Lado1, Lado2, Lado3)
círculo(Raio)
```

Uma operação sobre tais figuras poderia ser a ampliação das mesmas. Pode-se implementá-la como uma relação de três argumentos

```
amplia(Fig, Fator, Fig1)
```

onde `Fig` e `Fig1` são figuras geométricas do mesmo tipo (mesmo functor) e os parâmetros de `Fig1` são os mesmos de `Fig`, multiplicados por `Fator`. Para maior simplicidade assumiremos que os parâmetros de `Fig` são previamente conhecidos, isto é, instanciados com números, o mesmo ocorrendo com `Fator`. Uma maneira de programar a relação `amplia/3` é a seguinte:

```
amplia(quadrado(A), F, quadrado(A1)) :-
    A1 is F * A.
amplia(círculo(R), F, círculo(R1)) :-
    R1 is F * R.
amplia(retângulo(A, B), F, retângulo(A1, B1)) :-
    A1 is F * A, B1 is F * B.
...
```

Esse procedimento funciona, mas é um tanto grosseiro no caso em que há muitos tipos diferentes de figuras. É necessário prever todos os tipos de figuras que podem acontecer, empregando uma cláusula para cada tipo, apesar de todos dizerem essencialmente a mesma coisa: tome os parâmetros da figura original e multiplique-os pelo fator de ampliação formando uma figura do mesmo tipo com os novos parâmetros. Uma tentativa (mal-sucedida) de manipular pelo menos todas as figuras de um único argumento seria:

```
amplia(Tipo(Arg), F, Tipo(Arg1)) :-
    Arg1 is Arg * F.
```

Entretanto, não é permitido representar um functor em Prolog diretamente por meio de uma variável, ou seja, funtores devem ser sempre átomos, portanto a variável `Tipo` não seria aceita pela sintaxe da linguagem. O método correto é utilizar o predicado `=../2`. Assim a relação `amplia/3`, genérica, pode ser escrita como se segue:

```
amplia(Fig, F, Fig1) :-
    Fig =.. [Tipo | Parâmetros],
    multLista(Parâmetros, F, NovosParâmetros),
    Fig1 =.. [Tipo | NovosParâmetros].

multLista([], _, []).
```



```
multLista([X | L], F, [X1 | L1]) :-
    X1 is F*X, multLista(L, F, L1).
```

Os termos construídos com o predicado `=../2` podem também ser executados como objetivos. A vantagem disto é que o próprio programa pode, durante a execução gerar e executar objetivos. Uma sequência de objetivos ilustrando esse efeito poderia ser a seguinte:

```
...
obtenha(Funcor),
compute(ListaDeArgumentos),
Obj =.. [Funcor | ListaDeArgumentos],
Obj, ...
```

Aqui, `obtenha/1` e `compute/1` correspondem a procedimentos definidos pelo usuário para obter os componentes do objetivo a ser construído. O objetivo é formado por meio do predicado `=../2` e disparado para execução por meio da variável que o nomeia, `Obj`.

Algumas implementações da linguagem Prolog podem requerer que todos os objetivos que aparecem no programa sejam átomos ou uma estrutura com um átomo como functor principal, de forma que uma variável, independentemente de sua eventual instanciação, pode não ser sintaticamente aceita como um objetivo. Esse problema é contornado por meio de outro predicado pré-definido, `call/1`, cujo argumento é um objetivo a ser executado. Assim o exemplo dado acima poderia ser reescrito como:

```
...
Obj =.. [Funcor | ListaDeArgumentos]
call(Obj).
```

Às vezes pode-se desejar extrair de um termo apenas o seu functor principal, ou um de seus argumentos. Em tais casos pode-se, naturalmente, empregar o predicado `=../2`, entretanto, pode ser mais prático e eficiente usar um dos outros dois predicados pré-definidos para a manipulação de termos: `functor/3` e `arg/3`, cujo significado é o seguinte:

```
functor(Termo, Functor, Aridade)
```

é verdadeiro se `Functor` é o functor principal de `Termo` e `Aridade` é o seu número de argumentos, ao passo que

```
arg(N, Termo, Argumento)
```

é verdadeiro se `Argumento` é o `N`-ésimo argumento em `Termo`, assumindo que os argumentos são numerados da esquerda para direita iniciando em 1. Os seguintes exemplos servem como ilustração:

```
?-functor(teste(f(X), X, t), Functor, Aridade).
Functor=teste Aridade=3

?-arg(2, teste(X, t(a), t(b)), Argumento).
Argumento=t(a)

?-functor(D, data, 3), arg(1, D, 5), arg(2, D, abril), arg(3, D, 1994).
D=data(5, abril, 1994)
```

Esse último exemplo mostra uma aplicação especial do predicado `functor/3`. O objetivo `functor(D, data, 3)` produz em `D` um termo "geral" cujo functor principal é "data", com 3 argumentos. O termo é geral no sentido em que os três argumentos são variáveis não-instanciadas geradas pelo sistema Prolog. Por exemplo:

```
D=data(_02e, _02f, _030)
```

Essas três variáveis são então instanciadas como no exemplo acima, por meio dos três objetivos `arg/3`.

Relacionado a esse conjunto de predicados está o predicado `name/2`, para a construção e decomposição de átomos, introduzido no capítulo anterior. Seu significado é repetido aqui para manter completa a seção:

```
name(Átomo, Lista)
```

é verdadeiro se `Lista` é a lista dos códigos ASCII correspondentes aos caracteres do átomo `A`.

9.3 EQUIVALÊNCIAS E DESIGUALDADES

Até o momento, três "tipos de igualdade" foram estudados, iniciando pela baseada na unificação, representada por:

$X = Y$

que é verdadeira se X é Y unificam. Um outro tipo de igualdade é

$X \text{ is Expressão}$

que é verdadeira se X unifica com o valor da expressão aritmética Expressão. Tem-se também:

$\text{Expressão1} ::= \text{Expressão2}$

que é verdadeira se os valores das expressões aritméticas Expressão1 e Expressão2 são iguais. Se, ao contrário as expressões possuem valor diferente, escreve-se:

$\text{Expressão1} \neq \text{Expressão2}$

Algumas vezes poderá ser necessário um tipo mais estrito de igualdade: a igualdade literal entre dois termos. Esse tipo de igualdade é implementado por meio de um predicado pré-definido escrito como o operador infixo "==" , de modo que

$\text{Termo1} == \text{Termo2}$

é verdadeira se os termos Termo1 e Termo2 são idênticos, isto é, possuem exatamente a mesma estrutura e todos os componentes correspondentes são os mesmos. Em particular, os nomes das variáveis devem também ser os mesmos. A relação complementar é a não-identidade, escrita como:

$\text{Termo1} \neq \text{Termo2}$

Os exemplos abaixo abordam o uso de tais operadores:

```
?-f(a, b) == f(a, b).
sim

?-f(a, b) == f(a, X).
não

?-f(a, X) == f(a, Y).
não

?-X \== Y.
sim

?-t(X, f(a, Y)) \== t(X, f(a, Y)).
não
```

9.4 PROGRAMAS OU BASES DE DADOS?

De acordo com o modelo relacional, uma base de dados é a especificação de um conjunto de relações. Sob tal prisma, um programa Prolog pode ser visto como uma base de dados: a especificação das relações é parcialmente implícita (regras) e parcialmente explícita (fatos). Além disso existem predicados pré-definidos que tornam possível a atualização da base de dados durante a execução do programa. Isso é feito em tempo de execução, pela adição ou remoção de cláusulas do programa. Os predicados que servem a tais propósitos são assert/1, asserta/1, assertz/1 e retract/1. Um objetivo como:

`assert(C)`

é sempre bem sucedido e, como efeito colateral, ocasiona a adição da cláusula C na base de dados. Por outro lado um objetivo

`retract(C)`

faz o oposto, isto é, apaga uma cláusula que unifica com C da base de dados. O diálogo abaixo exemplifica esses dois predicados:

```
?-crise.
não

?-assert(crise).
sim
```

```

?-crise.
sim

?-retract(crise).
sim

?-crise.
não

```

As cláusulas inseridas por meio do predicado `assert/1`, atuam exatamente como se fossem parte do programa original. O seguinte exemplo ilustra o uso de `assert/1` e `retract/1` como um método para controlar situações que se modificam ao longo do tempo. Vamos assumir o programa abaixo, sobre as condições do tempo:

```

bom :-
    sol, not chuva.

instável :-
    sol, chuva.

deprimente :-
    chuva, neblina.

chuva.

neblina.

```

O diálogo a seguir mostra como a base de dados pode ir sendo gradualmente atualizada:

```

?-bom.
não

?-deprimente.
sim.

?-retract(neblina).
sim

?-deprimente.
não

?-assert(sol)
sim

?-instável.
sim

?-retract(chuva).
sim

?-bom
sim

```

Qualquer tipo de cláusula pode ser objeto dos predicados `assert/1` ou `retract/1`. No próximo exemplo mostraremos que `retract/1` é também não-determinístico: um conjunto completo de cláusulas pode ser removido, por meio do mecanismo de backtracking, através de um único objetivo `retract/1`. Vamos assumir um programa com os seguintes fatos:

```

veloz(senna).
veloz(prost).

meiaBoca(alesi).
meiaBoca(barrichello).

lento(katayama).
lento(moreno).

```

Podemos adicionar uma regra ao programa da seguinte maneira:

```

?-assert( (vence(X, Y) :- veloz(X), not veloz(Y)) ).
sim

?-vence(A, B).
A=senna B=alesi;
A=senna B=barrichello;
A=senna B=katayama;
A=senna B=moreno;
A=prost B=alesi;
A=prost B=barrichello;
A=prost B=katayama;
A=prost B=moreno;

```

não

Note que quando uma regra é inserida na base de dados, por meio do predicado `assert`, as regras sintáticas do Prolog exigem que esta seja fornecida entre parênteses.

Na introdução de uma cláusula, podemos desejar especificar a posição na qual a cláusula deve ser inserida na base de dados. Os predicados `asserta/1` e `assertz/1` permitem controlar a posição de inserção. O objetivo

`asserta(C)`

introduz a cláusula `C` no início da base de dados, enquanto que o objetivo

`assertz(C)`

adiciona a cláusula `C` no final da base de dados. O seguinte exemplo ilustra esses efeitos:

```
?-assert(p(a)), assertz(p(b)), asserta(p(c)).
sim
?-p(X).
X=c;
X=a;
X=b;
não
```

Há uma relação entre `consult/1` e `assertz/1`. "Consultar" um arquivo pode ser definido em termos de `assertz/1` da seguinte maneira: para "consultar" um arquivo, ler cada um dos seus termos (cláusulas) e inserí-los no final da base de dados:

```
consult(X) :-
    see(X), transfere(C), see(user).

transfere(end_of_file) :- !.
transfere(C) :-
    read(C), assertz(C), transfere(C1).
```

Já uma aplicação útil do predicado `asserta/1` é armazenar respostas já computadas para consultas formuladas ao programa. Por exemplo, vamos considerar que o predicado

`resolve(Problema, Solução)`

esteja definido. Podemos agora formular alguma consulta e requerer que a resposta seja lembrada para consultas futuras:

```
?-resolve(probl, Sol), asserta(resolve(probl, Sol)).
```

Se o primeiro objetivo acima é bem-sucedido, então a resposta(`Solução`) é armazenada e utilizada, como qualquer outra cláusula, na resposta a questões futuras. A vantagem de memorizar as respostas é que uma consulta posterior que unifique com "`probl`" será respondida muito mais rapidamente. O resultado é obtido agora pela recuperação de um fato, não sendo necessárias computações adicionais que possivelmente consumiriam muito mais tempo.

Uma extensão dessa idéia é a utilização do `assert` para gerar todas as soluções possíveis na forma de uma tabela de fatos. Por exemplo, podemos gerar uma tabela com os produtos de todos os pares de inteiros de 0 a 9 da seguinte maneira: geramos um par de inteiros, `X` e `Y`, computamos `Z` is `X*Y`, inserimos os três números como uma linha da tabela de produtos e então forçamos a falha do procedimento que, por meio de `backtracking`, irá gerar a tabela completa. O procedimento `tabMult/0`, abaixo, implementa essa idéia:

```
tabMult :-
    L = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
    membro(X, L), membro(Y, L),
    Z is X*Y, assert(produto(X, Y, Z)), fail.
tabMult.
```

O efeito colateral da execução de `tabMult/0` é adicionar a correspondente tabela de multiplicação à base de dados. Depois disso, podemos perguntar, por exemplo, que pares da tabela resultam em 8:

```

?-produto(A, B, 8).
A=1 B=8;
A=2 B=4;
A=4 B=2;
A=8 B=1;
não

```

Uma advertência sobre o uso indiscriminado de `assert` e `retract` deve ser feita aqui. Os exemplos dados ilustram algumas aplicações obviamente úteis desses predicados, entretanto o seu uso requer um cuidado especial. O uso excessivo e descuidado de tais recursos não é recomendado como um bom estilo de programação, uma vez que se está na realidade modificando o programa original em tempo de execução. Assim, relações válidas em um determinado momento, podem não mais ser válidas em um momento subsequente, isto é, em momentos diferentes, a mesma consulta pode ter respostas diferentes. O uso abusivo de `assert-retract` pode obscurecer o significado do programa e dificultar a compreensão do que é verdadeiro e o que não é num dado instante. O comportamento resultante do programa pode se tornar difícil de entender, de explicar e de confiar.

9.5 RECURSOS PARA O CONTROLE DE PROGRAMAS

A maioria dos recursos de controle de programas Prolog já foi apresentada anteriormente. Com vistas a permitir uma visão conjunta de tais predicados, apresenta-se a seguir um resumo de todos eles:

- `cut`: representado nos programas por `!`, previne a execução indesejada do mecanismo de backtracking;
- `fail`: é um objetivo que sempre falha;
- `true`: é um objetivo que sempre é bem sucedido;
- `not(P)`: é um tipo de negação que se comporta exatamente como se houvesse sido definido por: $\text{not}(P) \rightarrow P, !, \text{fail}; \text{true}$.
- `call(P)`: dispara um objetivo `P`. Será bem-sucedido se e somente se `P` também o for;
- `repeat`: é um objetivo que sempre é bem-sucedido. Sua principal propriedade é ser não-determinístico, isto é, toda vez que é alcançado por backtracking ele gera um caminho alternativo para a execução. Seu comportamento ocorre como se ele houvesse sido definido por:

```

repeat.
repeat :- repeat.

```

Uma forma típica de uso desse último predicado é ilustrada pelo procedimento `quadrado/0`, que lê uma seqüência de números e fornece o seu quadrado. A seqüência é dada por concluída quando for lido o átomo `"fim"`, que sinaliza o encerramento da execução:

```

quadrado :-
    repeat, read(X),
    (X=fim, !; Y is X*X, write(X), fail).

```

A construção acima é também muito empregada em programas interativos, que possuem diversas alternativas de execução mutuamente exclusivas, como em um menu de opções:

```

executa :-
    repeat, menu(X),
    (X=fim, !; exec(X), fail).

```

Aqui um menu é apresentado, uma ação selecionada, executada e o menu é novamente apresentado, repetindo-se esse ciclo até que a opção `"fim"` seja escolhida.

9.6 BAGOF, SETOF E FINDALL

Podemos gerar, através de backtracking, todos os objetos, um a um, que satisfazem algum objetivo. Cada vez que uma nova solução é gerada, a anterior desaparece e não é mais acessível. Algumas ve-

zes, entretanto, deseja-se dispor de todos os objetos gerados, por exemplo, coletados em uma lista. . Os predicados `bagof/3` e `setof/3` servem exatamente para tal propósito. O predicado `findall/3` é, em algumas implementações, oferecido como alternativa. O objetivo:

```
bagof(X, P, L)
```

irá produzir uma lista `L` de todos os objetos `X` que satisfazem ao objetivo `P`. Isto, naturalmente, só faz sentido se `X` e `P` possuem alguma variável em comum. Por exemplo, assumindo que temos em um programa Prolog uma especificação que classifica letras em vogais e consoante:

```
classe(a, vog).
classe(b, con).
classe(c, con).
classe(d, con).
classe(e, vog).
. . .
```

Então podemos obter a lista de todas as consoantes nessa especificação através do objetivo:

```
?-bagof(Letra, classe(Letra, con), Consoantes).
Consoantes=[b, c, d, ..., z]
```

Se, neste último objetivo, a classe das letras não estivesse especificada, obter-se-ia, por meio de backtracking, duas listas, uma correspondendo às vogais e outra às consoantes:

```
?-bagof(Letra, classe(Letra, Classe), Letras).
Classe=vog Letras=[a, e, i, o, u];
Classe=con Letras=[b, c, d, f, ..., z].
```

Se não houver solução para `P` no objetivo `bagof(X, P, L)`, então este simplesmente falha. Se algum objeto `X` é encontrado repetidamente, então todas as suas ocorrências irão aparecer em `L`, o que conduz à possibilidade de existência de elementos duplicados em `L`. O predicado `setof/3` é similar ao `bagof`. O objetivo:

```
setof(X, P, L)
```

irá novamente produzir uma lista `L` dos objetos `X` que satisfazem a `P`, só que desta vez a lista `L` estará ordenada e itens duplicados, se houver, serão eliminados. A ordem dos objetos é estabelecida em função de sua ordem alfabética ou de acordo com a relação "<" se os objetos na lista form números. Se os objetos forem estruturas, então seus funtores principais são comparados para fins de ordenação. Se estes são iguais, então a decisão fica por conta dos primeiros argumentos diferentes a contar da esquerda.

Não há restrição quanto ao tipo de objeto a ser coletado. Assim podemos, por exemplo construir uma lista de pares da forma `Classe/Letra` de forma que as constantes apareçam em primeiro lugar na lista ("con" antecede alfabeticamente "vog"):

```
?-setof(Classe/Letra, classe(Letra, Classe), Letras).
Letras=[con/b, con/c, ..., con/z, vog/a, ..., vog/u]
```

Um outro predicado dessa mesma família é `findall(X, P, L)`, que novamente produz a lista `L` de todos os objetos `X` que satisfazem `P`. A diferença entre esse predicado e o `bagof` é que todos os objetos `X` são coletados sem considerar eventuais soluções diferentes para as variáveis em `P` que não são compartilhadas com `X`. Essa diferente é ilustrada no seguinte exemplo:

```
?-findall(Letra, classe(Letra, Classe), Letras).
Letras=[a, b, c, ..., z]
```

Além disso, se não há nenhum objeto `X` que satisfaça `P`, então o predicado `findall(X, P, L)` resulta bem-sucedido com `L=[]`. Caso o predicado `findall/3` não se encontre entre os predicados pré-definidos em uma determinada implementação Prolog, podemos programá-lo facilmente da seguinte maneira:

```
findall(X, Objetivo, Lista) :-
    call(Objetivo), assertz(solução(X)), fail;
    assertz(solução(fim)), coleta(Lista).
```

```

coleta(Lista) :-
    retract(solução(X)), !,
    (X==fim, !, Lista=[];
     Lista=[X | Resto], coleta(Resto)).

```

No programa acima, todas as soluções para o objetivo "Objetivo" são geradas por meio de backtracking. Toda solução gerada é imediatamente incluída na base de dados, de forma que não é perdida quando a próxima solução é encontrada. Depois de encontrar todas as soluções, estas devem ser coletadas em uma lista e retiradas da base de dados.

RESUMO

- Uma implementação Prolog normalmente fornece um conjunto de predicados pré-definidos para diversas operações de uso frequente que nem sempre são de fácil codificação em Prolog "puro";
- O tipo de um termo Prolog pode ser testado por meio dos seguintes predicados pré-definidos:

```

var(X)      X é uma variável não-instanciada,
nonvar(X)   X não é uma variável não-instanciada,
atom(X)     X é um átomo,
integer(X)  X é um valor inteiro,
float(X)    X é um valor em ponto flutuante,
atomic(X)   X é um átomo ou um valor inteiro, e
string(X)   X é um string;

```

- Termos Prolog podem ser construídos os decompostos através dos seguintes predicados pré-definidos:

```

Termo =.. [Functor | Argumentos]
functor(Termo, Functor, Aridade)
arg(Ord, Termo, Argumento)
name(Atomo, Códigos)

```

- Os seguintes operadores pré-definidos são empregados na verificação de equivalências e desigualdades:

```

X = Y      X e Y unificam,
X is E     X é o valor da expressão aritmética E,
E1 == E2   E1 e E2 tem o mesmo valor,
E1 \= E2   E1 e E2 tem valores diferentes,
T1 == T2   T1 e T2 são idênticos,
T1 \= T2   T1 e T2 não são idênticos;

```

- Um programa Prolog pode ser visto como uma base de dados relacional, que pode ser atualizada por meio dos seguintes predicados:

```

assert(Cláusula)
asserta(Cláusula)
assertz(Cláusula)
retract(Cláusula)

```

- Um predicado pré-definido não-determinístico para o controle de programas é o repeat/0, destinado à geração de um número ilimitado de alternativas para o backtracking, que é definido como:

```

repeat.
repeat :- repeat.

```

- Todos os objetos que satisfazem uma dada condição podem ser coletados em uma lista por meio dos seguintes predicados:

```

bagof(Objeto, Condição, Lista)
setof(Objeto, Condição, Lista)
findall(Objeto, Condição, Lista)

```

EXERCÍCIOS

- 9.1 Escreva um procedimento denominado `simplifica/2` para simplificar simbolicamente expressões de soma envolvendo números e átomos representando variáveis. O procedimento deve rearranjar a expressão resultante de modo que os átomos precedam os números. Alguns exemplos do seu uso seriam:

```
?-simplifica(1+1+a, E).
E=a+2

?-simplifica(1+b+4+2+c+a, E).
E=a+b+c+7

?-simplifica(3+x+x, E).
E=2*x+3
```

- 9.2 Defina o predicado `básico(Termo)`, que é verdadeiro se `Termo` não possui nenhuma variável não-instanciada.
- 9.3 Defina a relação `subentende(Termo1, Termo2)`, que é verdadeira se `Termo1` é "mais geral" que `Termo2`. Por exemplo:

```
?-subentende(X, c).
sim

?-subentende(g(X), g(t(Y))).
sim

?-subentende(f(X, Y), f(a, a)).
sim

?-subentende(f(X, X), f(a, b)).
não
```

- 9.4 Defina a relação `copia(Termo, Cópia)`, que produz em `Cópia` uma cópia de `Termo` com todas as suas variáveis renomeadas. Isso pode ser facilmente programado empregando os predicados `assert/1` e `retract/1`.
- 9.5 Use o predicado `bagof/3` para definir a relação `potência(Conjunto, Subconjuntos)`, que computa o conjunto de todos os subconjuntos de um dado conjunto, sendo todos os conjuntos representados como listas. Por exemplo:

```
?-potência([a, b, c], P).
P=[[], [a], [b], [c], [a, b], [a, c], [b, c], [a, b, c]]
```


10. LÓGICA E BASES DE DADOS

10.1 BASES DE DADOS RELACIONAIS

Uma "base de dados" pode ser entendida como uma coleção de dados interrelacionados, armazenada de modo independente do programa que a utiliza, permitindo a recuperação, inserção, remoção e modificação de forma controlada. A quantidade de dados é tipicamente grande e o conteúdo muda ao longo do tempo. Em Prolog, uma base de dados é definida como um conjunto de fatos, não havendo, entretanto, nada que impeça a linguagem de trabalhar diretamente com bases de dados convencionais. Além disso a linguagem Prolog possui características que a tornam um excelente interface para lidar com bases de dados relacionais.

Um dos marcos mais importantes no desenvolvimento da pesquisa acerca de bases de dados foi a introdução do modelo relacional, por Codd em 1970. Em tal modelo, os dados são definidos por meio de relações sobre domínios e os fatos individuais são representados como tuplas de valores sobre tais domínios. Uma relação com um conjunto de tuplas é também denominada uma "tabela". O modelo relacional é conceitualmente muito "limpo" e elegante, apoiado por sólida fundamentação matemática.

Diferentemente de outros modelos de bases de dados, o modelo relacional não possui o conceito de "pointer", de modo que a associação entre diferentes tabelas é feita através da identidade explícita de valores de atributos. Este princípio concentra o esforço de implementação em obter maior velocidade de acesso, ao passo que a vantagem natural é a grande flexibilidade e fácil entendimento do processo de modelagem de dados.

O modelo relacional tem produzido um grande esforço de pesquisa. O propósito de sua introdução aqui tem sua origem no fato de que tabelas correspondem a uma forma muito natural de armazenar fatos interrelacionados em Prolog.

10.1.1 EXEMPLO DE UMA BASE DE DADOS RELACIONAL

Considere as seguintes relações:

- pessoa/4, contendo nome, sexo, pai e mãe;
- carro/4, contendo a placa, o fabricante, o proprietário e a cor.

Tais relações podem originar tabelas como as apresentadas abaixo:

Tabela 10.1(a): Relação pessoa/4

Nome	Sexo	Pai	Mãe
Marcelo	m	Luiz	Gilda
Luiz	m	Alfredo	Lina
Gilda	f	Miguel	Ana
Lúcia	f	Luiz	Gilda
Paulo	m	Miguel	Ana
Lina	f	Francisco	Júlia

Tabela 10.1(b): Relação carro/4

Placa	Fabricante	Proprietário	Cor
ABC-4590	Volkswagen	Alfredo	azul
XYZ-1211	Ford	Lina	branco
RTC-9004	Fiat	Luiz	vermelho
LLZ-7533	GM	Gilda	prata

Uma base de dados Prolog, formada a partir das tabelas 10.1(a) e (b), seria representada através dos seguintes fatos:

```

pessoa(marcelo, m, luiz, gilda).
pessoa(luiz, m, alfredo, lina).
pessoa(gilda, f, miguel, ana).
pessoa(lúcia, f, luiz, gilda).
pessoa(paulo, m, miguel, ana).
pessoa(lina, f, francisco, júlia).

carro(abc-4590, vw, alfredo, azul).
carro(xyz-1211, ford, lina, branco).
carro(rtc-9004, fiat, luiz, vermelho).
carro(llz-7533, gm, gilda, prata).

```

Um ou mais atributos em cada relação possui a propriedade especial de serem únicos na tabela. Tais atributos são denominados "chaves" e identificam os objetos acerca dos quais armazenamos informações. Usualmente se costuma sublinhar os atributos que são chaves, por exemplo:

pessoa:
nome sexo pai mãe

10.1.2 RELAÇÕES BINÁRIAS

As relações mais simples que existem são as relações binárias, que associam um único atributo a cada chave. A relação pessoa/4, que possui a chave "Nome", seria assim dividida em 3 relações:

Nome-Sexo	Nome-Pai	Nome-Mãe
sexo(marcelo, m)	pai(marcelo, luiz)	mãe(marcelo, gilda)
etc...

o mesmo se aplica à relação carro/4, cuja chave é "Placa":

Placa-Fabricante	Placa-Proprietário	Placa-Cor
Fabr(abc-4590, vw)	pr(abc-4590, alfredo)	cor(abc-4590, azul)
etc...

entretanto, por questões de conveniência e economia, toda a informação relacionada é reunida em uma única relação.

Uma situação de exceção ocorre quando é necessário manipular informação incompleta no modelo relacional. Em uma relação binária, a tupla correspondente é desprezada, por exemplo, um carro sem um proprietário. Entretanto, no caso em que é formada uma tupla com diversos atributos, um símbolo especial "nil" é empregado para representar tal informação. Por exemplo:

```
carro(ajk-6712, honda, nil, verde)
```

10.1.3 CHAVES COMPOSTAS

Em uma estratégia de implementação simples, assume-se que há uma única chave em cada tupla, normalmente ocupando a posição do primeiro argumento. Para chaves compostas assumiremos aqui uma convenção ad-hoc, representando-as como uma lista de argumentos:

[ch1, ch2, ch3]

que possui o seu próprio nome, mantendo entretanto em separado os atributos individuais ch1, ch2 e ch3.

10.2 RECUPERAÇÃO DE INFORMAÇÕES

Recuperar informações significa combinar e apresentar o conteúdo da base de dados em uma forma que satisfaça nossas necessidades. Em bases de dados convencionais isto é executado por um programa que atua sobre a base de dados. Em Prolog isto é feito através da definição das condições de solução em lógica. Por exemplo:

- Quem possui um fiat?

```
?-carro(_, fiat, Prop, _).  
Prop = luiz
```

- Quem fabrica os carros preferidos pelas mulheres?

```
?-pessoa(N, f, _, _), carro(_, Fabr, N, _).  
N = lina Fabr = ford;  
N = gilda Fabr = gm;  
não
```

10.2.1 RECUPERAÇÃO EFICIENTE

Os sistemas Prolog permitem a representação de informação relacional e a sua recuperação é facilmente formulada. Grandes bases de dados, entretanto, devem ser tratadas com cuidado, principalmente quando da combinação de tuplas distribuídas em duas ou mais tabelas. Assim, sistemas Prolog destinados a tais atividades normalmente devem possuir um "otimizador de consultas", que é um programa escrito em Prolog que manipula consultas como se fossem dados de entrada expressos sob a forma de termos, isto é, tal programa desempenha o papel de um "meta-interpretador".

Uma estratégia possível a empregar seria selecionar primeiro a condição que apresentasse o menor número de soluções possíveis, supondo que todas as variáveis estivessem instanciadas.

Suponha, por exemplo, que um crime tenha sido cometido e está sendo procurado um homem em um ford azul. A base de dados da polícia possui duas tabelas: uma com 3000 carros e outra com 10000 pessoas suspeitas. Lembre-se que uma pessoa pode possuir mais de um carro. Vamos imaginar que haja dez fords azuis e que metade das pessoas na base de dados sejam homens. Há duas formas de formular a questão:

```
?-carro(Placa, ford, X, azul), pessoa(X, m, _, _).
```

e

```
?-pessoa(X, m, _, _), carro(Placa, ford, X, azul).
```

Supondo que haja um acesso direto quando se dispõe da chave da tabela, é fácil verificar que, no primeiro caso, serão realizadas 3000 tentativas de unificação na tabela de carros, das quais apenas 10 serão bem sucedidas (só há 10 fords azuis), produzindo 10 acessos diretos à tabela de pessoas para verificar o sexo, num total de 3010 unificações. No segundo caso, entretanto, serão realizadas primeiro 10000 tentativas de unificação na tabela de pessoas, das quais 5000 serão bem sucedidas. Para cada uma dessas unificações bem sucedidas, 3000 acessos deverão ser feitos à tabela de carros, uma vez

que não se dispõe da chave, que é "Placa". O número de tentativas de unificação realizadas aqui será portanto $5000 \times 3000 + 10 = 15\,000\,010$. Isso mostra porque as condições com o menor número de soluções possíveis devem ser colocadas em primeiro lugar na formulação de consultas.

10.2.2 TABELAS VIRTUAIS

Uma das facilidades proporcionadas pelo Prolog no tratamento do modelo relacional é a possibilidade de definir novas tabelas sem ter de criá-las, empregando a implicação lógica. Tais tabelas são denominadas "tabelas virtuais". Por exemplo, uma tabela `corDoCarro/2` que contém como argumentos somente a placa e a cor de um carro pode ser definida da seguinte maneira:

```
corDoCarro(X, Y) :- carro(X, _, _, Y).
```

O conceito de tabelas virtuais é uma adaptação das "relações extratoras" introduzidas no capítulo anterior. Um subconjunto do Prolog convencional, sem os símbolos funcionais e o tratamento de listas, denominado Datalog, foi proposto com essa finalidade. Na verdade o uso de Prolog para representar bases de dados relacionais, introduz novos conceitos e regras, ampliando o nível da informação.. Considere por exemplo a questão:

```
Quem tem uma avó que possui um ford branco?
```

Em Prolog as regras para definir as relações `avó/2`, `corDoCarro/2`, etc. são facilmente construídas e incorporadas à base de dados, transcendendo o modelo relacional. A questão apropriada poderia ser construída assim:

```
?-avó(X, P), carro(_, ford, X, branco).
```

10.2.3 NOMES SIMBÓLICOS

Quando as tabelas Prolog são acessadas, o programa usa a posição do argumento na relação para acessar a coluna correspondente. Isso se torna difícil, quando o número de argumentos é muito grande. Além disso, constrange o programa a realizar concretamente as relações. O que se necessita, portanto é uma representação mais abstrata que permita ao programa lidar com modificações na modelagem dos dados. Uma solução é empregar tabelas virtuais binárias, contendo o nome do atributo como argumento explícito. No caso de tabelas com muitos argumentos, esta técnica pode se tornar uma necessidade. Um predicado geral, `atributo/4` pode ser definido para todos os nomes de atributos:

```
atributo(carro, P, placa, P) :- carro(P, _, _, _).
atributo(carro, P, fabricante, F) :- carro(P, F, _, _).
atributo(carro, P, proprietário, X) :- carro(P, _, X, _).
atributo(carro, P, cor, C) :- carro(P, _, _, C).
```

10.3 ATUALIZAÇÃO DA BASE DE DADOS

O modelo relacional impõe a restrição de que certos campos devem ser campos chaves, cujo valor deve ser único em uma tabela. Assim, "Nome" é a chave na relação `relação pessoa/4`, enquanto que na relação `carro/4` a chave é "Placa". Em Prolog, um sistema para o gerenciamento de bases de dados relacionais pode ser implementado de forma muito natural. As operações básicas são:

```
esquece(T)      % Remove a tupla T
memoriza(T)     % Insere a tupla T, se já não estiver lá
atualiza(V, N)  % Remove a velha e insere a nova tupla
```

Por exemplo:

```
?-esquece(carro(_, _, gilda, _)).
```

irá remover da base de dados todos os carros que pertencem a Gilda. Da mesma forma

```
memoriza(carro(flt-5455, honda, gilda, cor-de-rosa)).
```

irá introduzir o novo - e único - carro de Gilda na base de dados.

Na construção dos predicados `esquece/1` e `memoriza/1`, emprega-se a chave originalmente definida como elemento de referência, devendo-se preservá-la única em qualquer circunstância. Assim tais predicados devem ser construídos na forma abaixo:

```
esquece(X) :-
    esquecel(X), fail.
esquece(X).

esquecel(X) :-
    retract(X).
esquecel(X).

memoriza(X) :-
    esquece(X), assert(X).
memoriza(X) :-
    assert(X).
```

O predicado `esquece(X)` irá excluir da base de dados todas as sentenças que unificam com `X`. Se for desejada a exclusão somente da primeira ocorrência, deve ser usado o predicado `esquecel(X)`. Ambos, `esquece/1` e `esquecel/1` são sempre bem sucedidos, garantindo o primeiro, com sua execução, que não há mais na base de dados nenhuma sentença que unifique com `X` e o segundo que a primeira sentença encontrada unificando com `X` foi removida. Por outro lado o predicado `memoriza(X)` inicia com uma chamada a `esquece/1`, preservando assim a unicidade da chave estipulada em `X`. Deve ser também notado que esses predicados são extremamente poderosos e devem ser usados com absoluto cuidado para evitar "acidentes". Um cuidado interessante seria restringir a execução de `esquece/1`, `esquecel/1` e `memoriza/1` a argumentos que possuíssem uma instância explícita para a chave da tupla a esquecer ou memorizar.

10.4 MODELAGEM DE DADOS

Uma base de dados não é somente uma coleção de dados ou entidades, mas também as associações ou relacionamentos entre eles. Tais associações constituem o denominado "modelo de dados". A tecnologia de bases de dados vem oferecendo métodos e ferramentas para a solução de problemas em ambientes complexos e de grande porte. O projeto de modelos lógicos de dados é um importante objetivo nas áreas de representação e aquisição de conhecimento. O que se verifica é que a pura lógica de predicados é um formalismo extremamente poderoso, de expressividade ou capacidade de representação virtualmente ilimitada, de modo que freqüentemente temos que impor restrições à linguagem empregada na modelagem, retornando porém à lógica de predicados para explicar a semântica ou projetar extensões não convencionais.

10.4.1 FORMAS NORMAIS

Como em toda modelagem, as únicas coisas importantes a serem modeladas são os invariantes fundamentais do domínio do problema. A mais importante propriedade dos invariantes é que os objetos pertencem a classes que podem ser armazenadas uniformemente como relações.

Um outro princípio básico aqui é a evidência de que um determinado dado em uma certa relação é funcionalmente dependente de outro. Um conjunto de dados `B` é dito "funcionalmente dependente" de um outro conjunto de dados `A` se para todo elemento `a` em `A` há um único elemento `b` em `B` tal que `b` está relacionado com `a`. As notações mais empregadas são as seguintes:

```
A ----> B
A, B ----> C
```

significando respectivamente: "`B` é funcionalmente dependente de `A`" e "`C` é funcionalmente dependente da combinação de `A` e `B`". Por exemplo:

```
trabalhador ----> empregador
```

Devido ao fato de que as chaves são únicas, segue automaticamente que todos os atributos de uma

entidade são funcionalmente dependentes de sua chave.

10.4.2 FORMAS NORMAIS RELACIONAIS

Outro importante princípio da boa modelagem de dados é evitar redundâncias. A mesma peça de informação deve ser armazenada uma única vez. Assim, para qualquer modificação em seus valores, a base de dados necessitará ser atualizada em um único ponto. Em bases de dados relacionais, tais princípios são definidos por meio de um processo denominado "normalização". As diferentes formas normais são denominadas: "primeira forma normal", "segunda forma normal", etc., e abreviadas respectivamente por 1FN, 2FN, etc. Aqui introduzimos as três primeiras delas.

PRIMEIRA FORMA NORMAL (1FN)

Evita repetir grupos, como no exemplo:

```
empregador empregado1, empregado2, ..., empregadon
```

Não usar a representação:

```
empregados(joão, [josé, júlia, jorge, josefina, jane]).
```

mas sim a representação

```
empr(josé, joão).  
empr(júlia, joão).  
empr(jorge, joão).  
empr(josefina, joão).  
empr(jane, joão).
```

onde os empregados (por exemplo, josé) não são funcionalmente dependentes do empregador (joão). Ao contrário, o empregador é funcionalmente dependente dos empregados. Na prática, o benefício acontece quando um novo empregado (por exemplo, jonas) é contratado, porque tal fato pode ser incluído na base de dados com:

```
?-memoriza(empr(jonas, joão)).
```

não necessitando o programador:

- (1) Selecionar a lista de empregados de joão,
- (2) Adicionar Jonas,
- (3) Produzir uma nova lista,
- (4) Apagar a tupla corrente, com a velha lista, e
- (5) Produzir uma nova tupla, com a nova lista.

Um modelo na primeira forma normal deveria portanto ser:

```
empregador empregado.
```

SEGUNDA FORMA NORMAL (2FN)

Esta forma é relevante para tuplas com chaves compostas:

```
empregado nomeEmpregado  
empregado projeto nomeProjeto horas
```

Neste caso, cada empregado possui um número (a chave "empregado") e um nome (nomeEmpregado). O empregado trabalha em um conjunto de projetos com números (a chave "projeto") e nomes (nomeProjeto), dedicando a cada um certo número de "horas".

A anomalia nesta representação é que nomeProjeto não é funcionalmente dependente da chave (empregado, projeto) como um todo, mas apenas de uma parte dela (projeto). Assim a informação nomeProjeto é armazenada muitas vezes mais do que o necessário. Se o nome do projeto muda, todas as ocorrências de nomeProjeto devem ser alteradas, uma vez para cada empregado que nele trabalha. Um modelo na segunda forma normal seria:

empregado nomeEmpregado
empregado projeto horas
projeto nomeProjeto

Aqui nomeProjeto é armazenado uma única vez para cada projeto e modificado através de uma única atualização.

TERCEIRA FORMA NORMAL (3FN)

Um bom exemplo da 3FN ocorre quando a informação sobre uma pessoa, seu empregador e o endereço de seu empregador são armazenados. Se a relação

empregado empregador endereçoEmpregador

existe, então a entidade endereçoEmpregador não é funcionalmente dependente da chave "empregado" sozinha, mas na verdade de "empregador", que por sua vez é dependente de "empregado". Como nos casos anteriores, problemas de redundância e de múltiplas atualizações surgem, de modo que a normalização recomenda que a relação acima seja dividida em duas relações independentes:

empregado empregador
empregador endereçoEmpregador

Os princípios da normalização podem ser aplicados manualmente para modelos pequenos, entretanto, para grandes modelos a normalização deve preferencialmente ser apoiada por ferramentas de engenharia de software.

10.5 ALÉM DO MODELO RELACIONAL

O modelo relacional puro nem sempre é poderoso o bastante para modelagens avançadas, devido à falta de expressividade semântica. Por exemplo, o modelo relacional não requer que, para cada empregado, o atributo empregador corresponda a uma tupla existente na base de dados. Em modelos reais há dois tipos de regras que relacionam as tabelas uma à outra:

- regras genéricas, que definem novas tabelas virtuais que não são explicitamente armazenadas, e
- regras restritoras, que estabelecem restrições sobre o que é permitido na base de dados.

Um exemplo de regras restritoras é dada pelas dependências funcionais, que especificam que atributos-chave são e devem ser únicos. Um outro exemplo seria uma regra como:

"Todos os elefantes são cor-de-cinza."

que deduz a cor de um elefante na base de dados, produzindo ainda uma restrição que garante que, nas atualizações subsequentes, nenhum elefante de outra cor será armazenado na base de dados. Tais bases de dados são denominadas "dedutivas".

10.6 REDES SEMÂNTICAS

Questões de semântica são mais importantes para o projeto de uma base de conhecimento do que do que métodos para a codificação de dados. Quando os projetistas de base de dados adicionam mais informação semântica às bases de dados, os modelos resultantes começam a assemelhar-se aos sistemas de representação de conhecimento desenvolvidos pelos pesquisadores de inteligência artificial. Um desses esquemas de representação de conhecimento é conhecido como "rede semântica". Uma

rede semântica é um formalismo para representar fatos e relacionamentos entre fatos por meio de relações binárias. Por exemplo, na Figura 10.1, José, João e 555-2455 representam objetos. "telefone" representa uma relação entre os objetos José e 555-2455, enquanto que "empregador" representa uma relação entre José e João.

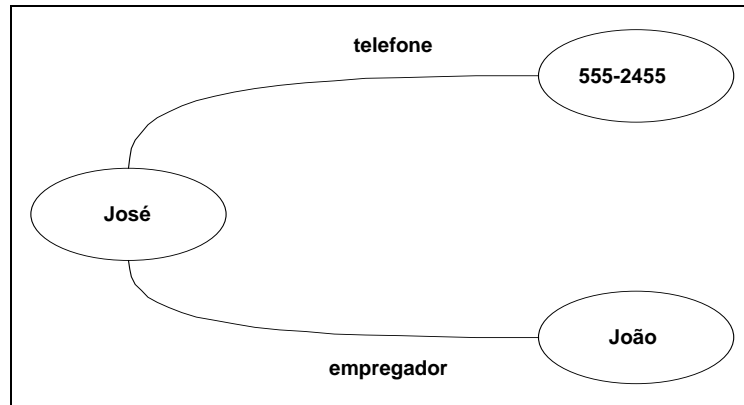


Figura 10.1: Uma rede semântica simples

Os relacionamentos individuais são conectados em uma rede, onde os objetos, por exemplo, "José", são representados uma única vez. Para relações binárias, as redes semânticas são um excelente formalismo com uma notação gráfica simples. Quando se tenta, entretanto, representar relações n-árias em redes semânticas é-se forçado a empregar construções artificiais, perdendo o formalismo das redes semânticas grande parte dos seus atrativos.

Acredita-se que grande parte do raciocínio humano seja baseado em associações lineares, de modo que o modelo das redes semânticas é também um interessante modelo do pensamento humano. Em Prolog as relações binárias são implementadas individualmente, repetindo os nomes dos objetos como em:

```
telefone(josé, 555-2455).
empregador(josé, joão).
```

Armazenar uma rede semântica como uma rede com ponteiros é um método de implementação que oferece rápido acesso no processo de associação. Em Prolog, na falta do conceito de ponteiro, as redes são armazenadas como relações binárias. Isto é um pouco mais lento, mas muito flexível, tanto para recuperar informações quanto para sua atualização.

10.6.1 O CONCEITO DE CLASSE

Tão logo um objeto é classificado, grande quantidade de conhecimento se torna disponível a seu respeito. Uma "classe" é a descrição de atributos e propriedades que são comuns a determinados indivíduos, denominados os "membros" da classe. José, por exemplo, é um objeto pertencente à classe dos empregados. Um "atributo" é alguma coisa que pode assumir um valor. Telefone, por exemplo, é um atributo dos membros da classe dos empregados. Uma "propriedade" é um atributo juntamente com um valor. Por exemplo, uma rosa tem a propriedade cor = vermelha. José tem a propriedade telefone = 555-2455.

Uma classe pode ser vazia, por exemplo, a classe dos unicórnios, e duas classes com os mesmos elementos podem ser bastante diferentes, por exemplo, a classe dos diretores de pesquisa e a classe dos possuidores de aquários. São exemplos de classes:

```
animal
mamífero
baleia
elefante
```


tubarão

São exemplos de atributos:

cor
alimento
habitat
tamanho
temperamento

Uma classe pode ser subclasse de outra classe. Se S é uma subclasse de C e x é membro de S, então x é também membro de C. Por exemplo, "mamífero" é uma subclasse de "animal" e "elefante" é uma subclasse de "mamífero". Se Clyde é um elefante (isto é, um membro da classe elefante), então Clyde é ao mesmo tempo membro da classe mamífero e portanto também é membro da classe animal.

Se a classe possui um atributo, este é compartilhado por todas as suas subclasses. Note que uma classe pode ter um atributo, mesmo se não possui membros no momento. Valores de atributos inexistentes, tais como o telefone de um elefante, são rejeitados como não significativos, não devendo ser empregado o átomo "nil".

De modo similar, se uma entidade possui um atributo que é funcionalmente dependente dela, por exemplo, "toda pessoa tem um nome", e o valor do atributo estiver faltando, o átomo apropriado para representar isso é "desconhecido" e não "nil" ou algo parecido. Se, por outro lado, um atributo não é funcionalmente dependente, tal como os filhos de uma pessoa, então a sua ausência deve ser pelo átomo "nil" ou "nenhum" e não por "desconhecido".

Por exemplo: todos os animais tem uma cor, que varia. Portanto, todos os mamíferos tem uma cor. Os elefantes, portanto, tem uma cor, de modo que Clyde, que é um elefante, tem também uma cor. Se a classe tem uma propriedade, esta é automaticamente herdada por todos os seus membros. Por exemplo:

"Todos os elefantes tem uma cor = cinza"

implica em:

"Se Clyde é um elefante, então Clyde tem uma cor = cinza"

O armazenamento de informação sobre classes em conjunto com informação sobre objetos, requer alguns relacionamentos de uso geral, como os apresentados na figura 10.2

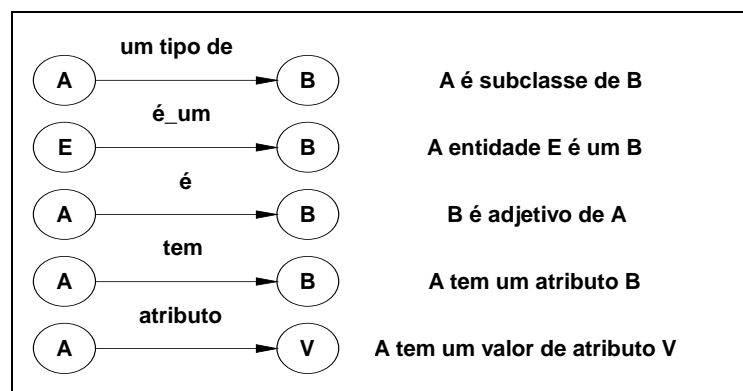


Figura 10.2 Relacionamentos em Redes Semânticas

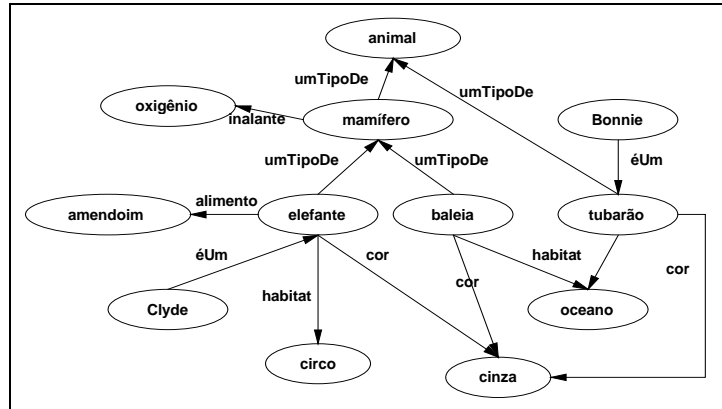


Figura 10.3 Uma rede semântica

Seja então a rede semântica mostrada na Figura 10.3. A informação ali representada pode ser adequadamente descrita através de um conjunto de cláusulas Prolog. A declaração de operadores infixos contribui para tornar o programa mais legível. Define-se assim a sintaxe dos relacionamentos descritos na Figura 10.2 por meio da assertiva:

```
:- op(900, xfx, [éUmTipoDe, éUm, é, tem, atributo]).
```

O seguinte programa Prolog descreve a rede semântica acima:

```
:- op(900, xfx, [éUmTipoDe, éUm, é, tem, temUm]).

animal temUm inalante.
animal temUm alimento.
animal temUm habitat.
animal temUm cor.

mamífero éUmTipoDe animal.
mamífero tem inalante=oxigênio.

elefante éUmTipoDe mamífero.
elefante tem alimento=amendoim.
elefante tem habitat=circo.
elefante tem cor=cinza.

baleia éUmTipoDe mamífero.
baleia tem habitat=oceano.
baleia tem cor=cinza.

tubarão éUmTipoDe animal.
tubarão tem habitat=oceano
tubarão tem cor=cinza

bonnie éUm tubarão.
clyde éUm elefante.
```

A estrutura de classes em redes semânticas é definida pelos seguintes axiomas:

```
X éUm Z2 :-
    Z1 éUmTipoDe Z2, X éUm Z1.

X tem Atributo=Valor :-
    X éUm C, C tem Atributo=Valor.
```

O primeiro destes axiomas é o fecho transitivo de éUm/2 e o segundo o fecho transitivo de tem/2. Com o emprego deles é possível consultar a base de conhecimento em busca de questões de caráter geral tais como:

```
"Que propriedades possui Clyde?"

?-clyde tem Atr=Val.
Atr=alimento Val=amendoim;
Atr=habitat Val=circo;
Atr=cor Val=cinza;
Atr=inalante Val=oxigênio;
não
```

RESUMO

- Em bases de dados relacionais os dados são definidos por meio de relações sobre domínios, e os fatos individuais são representados como tuplas de valores extraídos de tais domínios, cada um deles representando um "atributo";
- Pelo menos um dentre os atributos possui a característica especial de ser "único" em toda a tabela de tuplas. Tal atributo é denominado uma "chave" e identifica os objetos acerca dos quais é armazenada informação;
- Duas facilidades importantes oferecidas pelo modelo relacional são as tabelas virtuais, que definem relacionamentos implícitos, e o uso de nomes simbólicos;
- A atualização de base de dados deve ser projetada de modo a preservar a unicidade dos atributos-chave. Os predicados esquece/1, esquece1/1 e memoriza/1 foram desenvolvidos com essa idéia em mente;
- Na modelagem de dados é importante a adoção de formas normalizadas para garantir certos princípios organizacionais, evitando redundâncias e a necessidade de realizar múltiplas atualizações;
- Nem sempre o modelo relacional irá apresentar a expressividade necessária para a modelagem avançada. Um modelo mais expressivo, empregado em inteligência artificial é o das redes semânticas;
- A modelagem através de redes semânticas introduz os conceitos de classe e herança de atributos, os quais são de fácil construção em Prolog;
- Alguns dos relacionamentos empregados em redes semânticas são: `éUmTipoDe`, `éUm`, `é`, `tem` e `temUm`. Todos eles são binários, e podem ser representados em Prolog por meio de operadores infixos.

EXERCÍCIOS

10.1 Defina duas relações:

```
empr(Nome, Depto, Salário)
depto(Departamento, Gerente)
```

Escreva uma consulta ao sistema Prolog respondendo "Que empregados possuem salário superior ao de seu gerente?"

10.2 Defina as seguintes relações:

```
país(X)           % X é um país
mar(X)            % X é um mar
população(X,Y)    % X tem a população Y
fronteira(X, Y)   % X faz fronteira com Y
```

Escreva uma consulta ao sistema Prolog para responder a questão: "Que país, banhado pelo mediterrâneo, faz fronteira com um país que faz fronteira com um país cuja população excede a população da Índia?"

10.3 Modifique os predicados para a manipulação de bases de dados relacionais apresentados no presente capítulo de forma que múltiplas chaves sejam armazenadas sem redundância. (Dica: Use tabelas virtuais).

10.4 Amplie a base de conhecimento sobre animais. Como representar um avestruz como membro da classe dos pássaros se se definiu "voar" como uma propriedade dessa classe? Em outras palavras, como introduzir o conceito de exceção nas propriedades herdadas por um objeto a partir de sua classe?

10.5 Modele uma base de conhecimento, empregando redes semânticas para descrever automóveis, introduzindo os relacionamentos $\acute{e}ParteDe(X, Y)$, que é verdadeiro se X é parte de Y (por exemplo: $\acute{e}ParteDe(motor, carro)$) e $subconjDe(X, Y)$, que é verdadeiro se X é subconjunto de Y .

11. PROGRAMAÇÃO SIMBÓLICA

11.1 DIFERENCIAÇÃO SIMBÓLICA

Um exemplo conhecido de manipulação de fórmulas sem o emprego de computação numérica é a diferenciação de funções matemáticas. As regras são simples e diretamente implementadas em Prolog de uma forma muito elegante, empregando tão somente o mecanismo de unificação. No presente exemplo, todas as diferenciações irão se referir a uma variável matemática fixa, x , que será tratada como uma constante pelo sistema Prolog. As regras de diferenciação são definidas pelo predicado $\text{deriv}(U, V)$, que é verdadeiro quando $V = dU / dx$:

```
deriv(x, 1).
deriv(N, 0) :-
    number(N).           % number/1: embutido
deriv(U+V, U1+V1) :-
    deriv(U, U1),
    deriv(V, V1).
deriv(U-V, U1-V1) :-
    deriv(U, U1),
    deriv(V, V1).
deriv(U*V, U1*V+U*V1) :-
    deriv(U, U1),
    deriv(V, V1).
deriv(U/V, (V*U1-V1*U)/(V*V)) :-
    deriv(U, U1),
    deriv(V, V1).
deriv(U^N, N*U^(N1*U1)) :-
    number(N),
    N1 is N-1,
    deriv(U, U1).
deriv(exp(U), exp(U)*U1) :-
    deriv(U, U1).
...
```

Por exemplo:

```
?-deriv(x*x, Y).
Y=1*x+x*1
```

Entretanto, certamente seria mais apreciada uma saída melhor, tal como $2 \cdot X$ ou simplesmente $2X$. A razão da apresentação inadequada do resultado é que o Prolog não possui simplificação algébrica inerente, entretanto esta pode ser facilmente implementada, como será visto mais adiante neste mesmo capítulo.

11.2 MANIPULAÇÃO DE FÓRMULAS

Em uma linguagem de programação simbólica, como Prolog, os programadores precisam considerar as fórmulas e não apenas os seus valores. Em geral as fórmulas não envolvem apenas aritmética, mas podem ser combinadas arbitrariamente através dos mais variados operadores e operandos, de acordo com o princípio recursivo da decomposição: "o valor de uma expressão é o resultado da aplicação de um operador ao resultado dos restantes".

Em linguagens como Pascal e Lisp este princípio recursivo é parte da semântica da linguagem. Em Prolog isto deve ser feito explicitamente, mas pode ser feito sem dificuldades por um predicado recursivamente definido. Este esquema é geral e é uma réplica do princípio recursivo da decomposição:

"Para resolver uma expressão, primeiro (i) resolva seus operandos, e depois (ii) aplique o operador sobre os resultados obtidos".

11.3 OS OPERADORES REVISITADOS

Para lidar com uma expressão, é necessário ser capaz de manipular os seus subcomponentes. Na Tabela 11.1 relaciona-se um conjunto de operadores embutidos disponíveis na maioria das implementações Prolog. Há-se que lembrar entretanto que internamente tais operadores são representados sob a forma de termos funcionais, onde os operadores são functores. Por exemplo:

$X+Y$ é armazenado como `'+'(X, Y)`

O operador embutido `=./2` (univ) é capaz de atuar sobre uma expressão vista como uma lista de componentes:

$X+Y = .. ['+', X, Y]$
 $-X = .. ['-', X]$

Por exemplo:

`?- 3+2*7 =.. [X, Y, Z].`
`X='+' Y=3 Z=2*7`

`?-X =.. ['-', 3+5, 5*9].`
`X=3+5-5*9`

Também são importantes neste contexto os predicados embutidos `functor/3` e `arg/3` (ver seção 9.2) que atuam normalmente sobre operadores, empregando a notação funcional.

Tabela 11.1 Operações Comuns em Prolog

(a) Operações Binárias
$X+Y$ Adição
$X-Y$ Subtração
$X*Y$ Multiplicação
X/Y Divisão
$X=Y$ Igual
$X \neq Y$ Não igual
$X \geq Y$ Maior ou Igual
$X \leq Y$ Menor ou Igual
$X < Y$ Menor que
$X > Y$ Maior que
X and Y Conjunção
X or Y Disjunção
X impl Y Implicação
(b) Operações Unárias
$-X$ Negação Aritmética
<code>not X</code> Negação Lógica

11.4 AVALIAÇÃO DE FÓRMULAS

O efeito do operador `"is"` é conhecido:

`?-X is 3*7*37.`
`X=777`

`?-X is 7*11*13.`
`X=1001`

A avaliação das fórmulas numéricas é escondida do usuário, apesar de poder ser definida em Prolog. Sua implementação em Prolog é útil por duas razões: Primeiro para ensinar os princípios da avaliação de fórmulas em Prolog. Depois, pode vir a ser necessário incluir regras de operação que não se comportam estritamente com a semântica do operador `"is"`. Vamos agora implementar o operador `"$"` com a finalidade de estender os efeitos de `"is"`, de modo que a expressão seja esperada do lado esquerdo e o valor à direita, assim:

`10 + 10 $ 20`

que pode ser lido: "o valor de 10+10 é 20". O operador "\$" estende o "is" também na avaliação de variáveis globais, armazenadas como valor(A, B). Por exemplo:

```
valor(a, 3).
valor(b, 7).

?-a*b*37 $ X.
X=777
```

A avaliação estendida, \$, é definida da seguinte maneira:

```
:- op(900, xfx, '$').

(X $ X) :-
    number(X), !.
(X $ Y) :-
    valor(X, Y), !.
V $ U :-
    V =.. [Op, X, Y], !, X $ X1, Y $ Y1,
    W=.. [Op, X1, Y1], U is W.
V $ U :-
    V =.. [Op, X], !, X $ X1, W=..[Op, X1], U is W.
```

O operador \$ pode ser usado para implementar a atribuição ordinária de variáveis globais como no programa abaixo, onde o predicado esquece/1 é o mesmo introduzido no capítulo anterior e repetido aqui como recordação:

```
:- op(901, xfx, ':=').

(V:=E) :-
    E $ T, esquece(valor(V,X)), assert(valor(V, T)).

esquece(X) :-
    esquecel(X), fail.
esquece(X).

esquecel(X) :-
    retract(X).
esquecel(X).
```

A partir da definição acima podemos escrever:

```
?-a:=4, b:=13, c:=b*a, valor(c, X).
X=52
```

11.5 SIMPLIFICAÇÃO ALGÉBRICA

Outras aplicações importantes da programação simbólica são a manipulação de fórmulas, prova de teoremas no domínio da matemática e análise de programas. A prova de teoremas é também parte integrante da disciplina de verificação de programas, programando a correção de programas. Um teorema pode ser provado se pode ser reduzido à constante "true". Descreve-se inicialmente aqui a simplificação algébrica.

Há diversas regras para as várias fórmulas, por exemplo, as leis comutativa, associativa e distributiva. Na área da simplificação algébrica, as regras que reduzem a complexidade das fórmulas são especialmente interessantes. Algumas dessas regras são fornecidas abaixo através do predicado reduz/2.

A partir dos axiomas básicos de redução apresentados na figura acima, um pequeno programa simplificador pode ser construído baseado no seguinte princípio recursivo:

- Simplifique os operandos primeiro, depois a operação, e
- Repita até que nenhum dos operandos seja modificado.

reduz(X+0, X).	reduz(X=X, true).
reduz(0+X, X).	reduz(X or true, true).
reduz(X-X, 0).	reduz(true or X, true).
reduz(X-0, X).	reduz(X and false, false).
reduz(0-X, -X).	reduz(false and X, false).
reduz(X*0, 0).	reduz(X and true, X).

<code>reduz(0*X, 0).</code>	<code>reduz(true and X, X).</code>
<code>reduz(X*1, X).</code>	<code>reduz(X or false, X).</code>
<code>reduz(1*X, X).</code>	<code>reduz(false or X, X).</code>
<code>reduz(0/X, 0).</code>	<code>reduz(X impl true, true).</code>
	<code>reduz(true impl X, X).</code>
	<code>reduz(false impl X, true).</code>
	<code>reduz(X impl X, true).</code>
	<code>reduz(X and X, X).</code>
	<code>reduz(X or X, X).</code>
<code>reduz(U, V) :-</code>	
<code>U =..[Op, X, Y], number(X), number(Y), !, V is U.</code>	

Figura 11.1 O predicado reduz/2

O algoritmo está correto, porém não é completo. Também não possui eficiência ótima porque irá tentar ressimplicar uma expressão que um algoritmo mais refinado reconheceria como já simplificada. Tal refinamento será deixado ao leitor a título de exercício.

```

simplifica(U, V) :-
    simp(U, V, Teste). % Teste é verdadeiro se V<>U.

simp(F, H, true) :-
    reduz(F, G), !, simplifica(G, H).
simp(F, Z, true) :-
    F=..[Op, X, Y],
    simp(X, X1, mudaX),
    simp(Y, Y1, mudaY),
    membro(true, [mudaX, mudaY]), !,
    G=..[Op, X1, Y1],
    simplifica(G, Z).
simp(F, F, false).

```

O efeito do programa acima pode ser visualizado por meio dos seguintes exemplos:

```

?-simplifica(1*x-x*1, S).
S=0

?-simplifica(1*x+x*1, S).
S=x+x

```

A diferenciação e a simplificação algébrica podem agora ser integradas em um só predicado:

```

deriva(U, V) :-
    deriv(U, U1), simplifica(U1, V).

?-deriva(x*x, S).
S=x+x

```

11.5.1 SUBEXPRESSÕES COMUNS

A simplificação é possível quando operações adjacentes podem ser encontradas por meio do reconhecimento de padrões fixos. Por exemplo:

$$(a+b*j-f) - (a+b*j-f)$$

é reconhecido pela unificação com o padrão $X-X$. Entretanto, uma classe de problemas resta ainda por ser solucionada, que é quando há subexpressões que poderiam ser movidas de acordo com as regras comutativas e associativas, e então reduzidas quando um padrão unificável for reconhecido. A expressão

$$(a+b+c)-b$$

poderia ser transformada em

$$((a+c)+b)-b$$

que segue o padrão:

$$(X+Y)-Y$$

sendo redutível a

$$X=a+c$$

Na verdade a formação de subexpressões comuns é um dos importantes princípios heurísticos que de que se valem as pessoas na realização de simplificações algébricas. A tarefa básica, no caso, é descobrir subexpressões comuns.

Uma subexpressão ocorrendo em uma expressão é facilmente formulada como:

```
ocorre(X, X).
ocorre(S, Z) :-
    Z=..[Op, X, Y], (ocorre(S, X); ocorre(S, Y)).
```

de modo que o problema de descobrir se uma determinada expressão emprega a mesma subexpressão diversas vezes é solucionado por:

```
comum(Z, U) :-
    Z=..[Op, X, Y], ocorre(U, X), ocorre(U, Y).
```

Por exemplo:

```
?-comum((w+1+2*(w+1)), Z), fail.
w+1
w
1
não
```

11.6 INTEGRAÇÃO

Tem sido dito que a diferenciação é uma técnica, ao passo que a integração é uma arte. A tarefa de integração simbólica é objeto da engenharia de conhecimento, onde as especializações humanas são transferidas para sistemas computacionais. Uma primeira tentativa de obter integração poderia ser por meio da exploração da reversibilidade dos predicados Prolog :

```
integr(Y, Z) :-
    deriv(Z, Y).

?-integr(1*x+x*1, Int).
Int=x*x
```

Infelizmente a capacidade Prolog de inverter predicados é limitado. Se for solicitado:

```
?-integr(0, Int).
```

em um determinado momento será ativado o objetivo number(Int). Entretanto, tal predicado pré-definido não é inversível. Se o fosse, deveria gerar números instanciados (0, 1, 2, ...) , que são todas integrações corretas de 0. Mas ao invés disso produz a penas a resposta "não".

Um outro problema diz respeito a reversão da simplificação. Se for tentado

```
?-integr(x+x, Int).
```

com vistas a obter $x*x$, nenhuma resposta é obtida, porque $x+x$ somente é atingido após uma simplificação. Se simplifica/2 estiver sendo executado de modo reverso, irá cair num laço recursivo infinito. Entretanto, é possível modificar o predicado simplifica/2 para controlar a profundidade máxima da recursão. Essa aplicação pode resultar em um sistema de integração simbólica bastante lento, mas teoricamente completo, baseado no princípio da geração e teste exaustivos. A construção de tal sistema é deixada como um exercício ao leitor.

RESUMO

- A capacidade de programação simbólica é uma das principais características da linguagem Prolog. A diferenciação é facilmente implementada através de suas regras;
- O princípio recursivo de decomposição que é parte da semântica de linguagens tais como Pascal e Lisp, deve ser explicitado em Prolog, o que pode feito com grande facilidade;

- Os predicados embutidos `=./2`, `functor/3` e `arg/3` são de grande valia na programação simbólica para a separação dos subcomponentes das expressões;
- O operador `"$"` estende a semântica do operador `"is"` permitindo a avaliação de variáveis globais e a implementação do mecanismo de atribuição de valores;
- A simplificação algébrica é implementada simbolicamente por meio do predicado `reduz/2`, que associa os fatos e regras relevantes para a simplificação desejada;
- A identificação de subexpressões comuns para fins de simplificação necessita de heurísticas especiais para ser eficiente. O predicado `comum/2`, baseado em `ocorre/2`, representa uma implementação simples com esse objetivo;
- A integração pode ser implementada em parte como o inverso da diferenciação, entretanto as limitações de reversibilidade do Prolog irão exigir o uso de estratégias e heurísticas especiais para a execução desta tarefa.

EXERCÍCIOS

11.1 Escreva um programa de simplificação que nunca re-simplifique uma expressão já simplificada.

11.2 Estenda o predicado `deriva/2`, incluindo simplificação algébrica para lidar com as funções:

`ln`, `exp`, `sin`, `cos`, `arctan` e U^V

onde U e V são expressões genéricas.

11.3 Estenda o exemplo das subexpressões comuns para levar em conta a equivalência comutativa.

11.4 Escreva um programa Prolog para mover subexpressões comuns para próximas umas das outras e então executar reduções com base no reconhecimento de padrões, tal como antes. Por exemplo:

$(a+b+c+d - (a+c))$

Aqui, c é uma subexpressão comum que é removida dos dois operandos principais:

$(a+b+c+d)$	\Rightarrow	$((a+b+d) + c)$
$-(a+c)$	\Rightarrow	$-(a+c)$

		$(a+b+d) - a$

e então reduzida de acordo com o padrão:

$(x+c)-(y+c) \Rightarrow x - y$

que aplicado recursivamente produz o resultado $(b + d)$

11.5 Modifique o predicado `deriva/2` para obter a integração por inversão da derivação e a simplificação de acordo com o esquema:

```
integralN(U, V, N) :-
    nível(N), simplificaN(Fórmula, U, N), deriv(U, Fórmula).

nível(0).
nível(N1) :- nível(N), N1 is N+1.
```

onde `simplificaN/3` simplifica uma fórmula em exatamente N passos recursivos ($N = 0, 1, \dots$).

12. METODOLOGIA DA PROGRAMAÇÃO EM LÓGICA

A engenharia de software estabeleceu, ao longo do tempo, diversos critérios para a caracterização de programas de boa qualidade, assim como técnicas e práticas que, se empregadas, conduzem naturalmente à construção de bons programas. Ainda que tais técnicas tenham sido desenvolvidas geralmente do ponto de vista da programação procedimental convencional, é importante lembrar que programas em Prolog são também software e como tal devem estar sujeitos à mesma disciplina e método preconizados para o desenvolvimento de programas convencionais.

O estilo declarativo inerente à linguagem Prolog permite solucionar automaticamente diversos problemas relacionados com a recuperação de informações e representação de estruturas complexas de dados, entretanto, uma boa parte dos problemas com que se deparam os programadores são algorítmicos por natureza, devendo portanto ser interpretados e solucionados de forma algorítmica.

No presente capítulo são revisados alguns princípios gerais da engenharia de software, abordando os elementos necessários ao desenvolvimento de um bom estilo de programação em Prolog. Critérios de correção e eficiência são também introduzidos, visando oferecer ao leitor alguma instrumentação metodológica para a construção de programas de boa qualidade.

12.1 PRINCÍPIOS GERAIS DA BOA PROGRAMAÇÃO

Uma questão fundamental a esse respeito é: "O que é um bom programa?". A resposta a esta questão não é tarefa trivial, uma vez que há diversos critérios para julgar quão bom um programa é. Critérios geralmente aceitos incluem, entre outros, os seguintes:

- **CORREÇÃO:** Acima de tudo, um programa deve ser "correto", isto é, deve fazer exatamente o que se espera dele. Um erro comum, cometido por alguns programadores é negligenciar esse critério óbvio em favor de outros, como por exemplo a eficiência;
- **EFICIÊNCIA:** Um bom programa não deve consumir sem necessidade grandes quantidades de recursos, tais como memória e tempo de execução;
- **TRANSPARÊNCIA E LEGIBILIDADE:** Um bom programa deve ser fácil de ler e entender. Não deve ser mais complicado do que o necessário. Truques de programação que obscurecem o significado do programa devem ser evitados;
- **MODIFICABILIDADE:** Um bom programa deve ser fácil de ser modificado ou estendido. A transparência e a adoção de uma organização modular auxiliam a atingir tal objetivo;
- **ROBUSTEZ:** Um bom programa deve ser "robusto". Isso significa que ele não deve ser abortado facilmente quando o usuário entrar com dados incorretos ou inesperados. O programa deve, no caso de tais erros, manter-se em execução e comportar-se "racionalmente" (por exemplo: relatando o erro ao usuário e solicitando nova entrada de dados).
- **DOCUMENTAÇÃO:** Um bom programa deve ser adequadamente documentado. A documentação mínima aceitável para um programa é a sua listagem enriquecida com comentários suficientes para o seu entendimento.

A importância de cada critério vai depender do problema, das circunstâncias em que o programa é desenvolvido e do ambiente em que será utilizado. Não há dúvida, entretanto, de que a correção deve ser o critério de mais alta prioridade. Aos critérios de transparência, modificabilidade, robustez e documentação é normalmente atribuída uma prioridade no mínimo igual ao requisito de eficiência.

Há algumas regras gerais para atingir na prática os critérios apresentados acima. Uma delas, muito importante, é primeiro "pensar" sobre o problema a ser resolvido e somente iniciar a codificação na linguagem de programação escolhida depois de se ter formulado uma idéia clara sobre o que deve ser feito. Uma vez que um bom entendimento do problema foi desenvolvido e definida a sua solução, a codificação do programa torna-se fácil e rápida, havendo uma boa chance de se obter sem demora um programa correto.

reem qp

A formulação inicial obtida para a solução do problema deverá então ser convertida para a linguagem de programação escolhida. Tal processo, entretanto, pode não ser uma tarefa fácil. Uma abordagem consagrada é a de utilizar o "princípio dos refinamentos sucessivos", que considera a solução inicial uma formulação em "alto nível" e o programa finalmente obtido como uma solução em "baixo nível".

De acordo com o princípio dos refinamentos sucessivos, o programa final é obtido por meio de uma sequência de transformações ou refinamentos da solução inicial. Inicia-se com a formulação em alto nível da solução do problema e então passa-se a transformá-la de maneira que cada nova formulação obtida é equivalente à anterior, porém expressa de forma mais detalhada. Em cada passo de refinamento os conceitos usados na formulação anterior são elaborados em maior detalhe e a sua representação vai se aproximando da linguagem de programação. Deve-se ter em mente que os refinamentos se aplicam tanto às definições de procedimentos quanto às estruturas de dados. Nos estágios iniciais normalmente se trabalha com unidades de informação mais abstratas, cuja estrutura é refinada na medida em que avançamos com o processo. A estratégia dos refinamentos sucessivos possui as seguintes vantagens:

- Permite a formulação de uma solução inicial nos termos mais relevantes ao problema,
- Essa solução inicial é, por conseguinte, mais simples e sucinta, sendo a sua correção facilmente verificável, e
- Cada passo de refinamento deve ser suficientemente pequeno para ser manejado intelectualmente. Assim a transformação da solução em uma representação mais detalhada preserva com mais facilidade a sua correção.

No caso da linguagem Prolog, pode-se pensar em tal processo como sendo o de refinamento de refinamento de relações. Se, entretanto, a natureza do problema sugerir uma abordagem em termos algorítmicos, também é possível pensar em refinamento de algoritmos, adotando então a visão procedimental do Prolog.

Para refinar apropriadamente uma solução em algum nível de detalhamento e introduzir conceitos adequados ao próximo, é necessário "ter idéias". Portanto a programação é uma atividade criativa, especialmente para programadores iniciantes. À medida em que a experiência em programação aumenta, esta se torna menos uma arte e mais uma técnica. Assim, a questão principal é: "Como ter idéias?" A maioria das idéias surge da experiência com problemas similares, cuja solução é conhecida.

buscar as idéias adequadas para decompor um problema em subproblemas de solução mais fácil. Uma questão importante aqui é: "Como encontrar os subproblemas apropriados?". Os princípios fundamentais para responder tal questão serão discutidos agora.

12.2.1 USO DE RECURSÃO

Na solução de problemas envolvendo o processamento sequencial por meio de recursão, é uma boa heurística aplicar pensamento indutivo e resolver os seguintes dois casos separadamente:

- (1) Os casos triviais, ou básicos, em que o argumento é uma lista vazia ou unitária, e
- (2) Os casos gerais, em que o argumento é uma lista [Cabeça|Corpo] e o problema é assumido resolvido para "Corpo".

Em Prolog, essa técnica é utilizada frequentemente. Seja por exemplo o problema de processar uma lista de itens de tal maneira que cada item seja operado por uma mesma regra de transformação:

```
transforma(Lista, F, NovaLista)
```

onde Lista é a lista original, F é uma regra de transformação e NovaLista é a lista de todos os itens transformados. O problema de transformar Lista em NovaLista pode ser subdividido em dois casos:

- (1) Caso Básico: Lista = []

Se Lista = [], então NovaLista = [], independentemente de F.

- (2) Caso Geral: Lista = [X | Resto]

Para transformar uma lista do tipo [X | Resto] em uma lista do tipo [NovoX | NovoResto], transforme Resto, obtendo NovoResto e transforme X, obtendo NovoX.

Em Prolog:

```
transforma([], _, []).
transforma([X | Resto], F, [NovoX | NovoResto]) :-
    G =.. [F, X, NovoX], call(G),
    transforma(Resto, F, NovoResto).
```

A razão pela qual a recursão se aplica tão naturalmente em Prolog reside no fato de que os objetos estruturados, como árvores e listas, possuem uma organização recursiva intrínseca. Uma lista, por exemplo, ou é vazia (caso básico), ou possui uma cabeça e um corpo (caso geral).

12.2.2 GENERALIZAÇÃO

Muitas vezes é uma boa idéia generalizar o problema original, de forma a permitir que a solução do problema generalizado seja formulada recursivamente. O problema original é então solucionado como um caso especial da versão mais geral. A generalização de uma relação envolve tipicamente a introdução de um ou mais argumentos extras. O maior problema, que pode requerer uma profunda intuição, é: "Como encontrar a generalização correta?". Como ilustração examinaremos um clássico da pesquisa em inteligência artificial que é o "problema das oito damas". O enunciado original desse problema propõe dispor oito damas em um tabuleiro de xadrez de maneira que nenhuma delas ataque as demais. A relação correspondente poderia ser representada por:

```
oitoDamas(Posição)
```

que será verdadeira se Posição representar uma posição do tabuleiro tal que nenhuma dama ataque as restantes. Uma idéia interessante, nesse caso é generalizar o número de damas de oito para N, de forma que o número de damas se torna o argumento adicional.

```
nDamas(Posição, N)
```

A vantagem dessa generalização é que há uma formulação recursiva imediata para a relação $n\text{Damas}/2$:

(1) Caso Básico: $N = 0$

Colocar "zero" damas em segurança é trivial.

(2) Caso Geral: $N > 0$

Para colocar N damas em segurança no tabuleiro é necessário satisfazer as seguintes condições:

- Obter uma configuração segura para $N - 1$ damas, e
- Adicionar as damas restantes de forma que nenhuma delas ataque as demais.

Uma vez que o problema generalizado está solucionado, a solução do problema original é imediata:

```
oitoDamas(Posição) :-  
    nDamas(Posição, 8).
```

12.2.3 REPRESENTAÇÃO GRÁFICA DE PROBLEMAS

Na busca por idéias para solucionar um dado problema, frequentemente é de grande utilidade introduzir alguma representação gráfica do mesmo. Um desenho pode ajudar na percepção de algumas relações essenciais do problema. Só então passa-se a descrever o que se vê no desenho na linguagem de programação escolhida. No caso do Prolog, essa técnica parece ser especialmente produtiva, eis que:

- Prolog é particularmente adequado para problemas envolvendo objetos e relações entre objetos. De modo geral tais problemas podem ser naturalmente ilustrados por meio de grafos, onde os nodos correspondem a objetos e os arcos a relações;
- Os objetos estruturados em Prolog são naturalmente representados por meio de árvores;
- O significado declarativo dos programas Prolog facilita a tradução de representações gráficas porque, em princípio, a ordem na qual o desenho é feito não constitui um fator importante.

12.3 ESTILO DE PROGRAMAÇÃO

O propósito de adotar algumas convenções relacionadas ao método ou estilo de programação adotado é fundamentalmente:

- Reduzir o risco de erros de programação, e
- Produzir programas de boa legibilidade, fáceis de entender, corrigir e modificar.

Algumas normas cuja observância produz um bom estilo de programação em Prolog serão introduzidas a seguir: regras gerais, organização tabular de procedimentos longos e o uso apropriado de comentários.

12.3.1 REGRAS GERAIS PARA UM BOM ESTILO

- As cláusulas do programa devem ser curtas. Seu corpo não deve conter mais que uns poucos objetivos. Empregar sempre

```
proclA :- a, b, c.  
proclB :- d, e, f.  
proclC :- g, h, i.
```

ao invés de

```
procl :- a, b, c, d, e, f, g, h, i.
```

- Os procedimentos do programa devem também ser curtos (conter poucas cláusulas), porque procedimentos longos demais são difíceis de entender. (Apesar disso, procedimentos longos podem ser aceitáveis, desde que possuam uma estrutura uniforme, conforme será discutido mais adiante);
- Adotar nomes mnemônicos para procedimentos e variáveis, indicando o significado das relações e o papel desempenhado pelos objetos que nelas se fazem presentes;
- O lay-out dos programas é importante, incluindo um bom espaçamento, uso de linhas em branco, e indentação. Cláusulas sobre o mesmo procedimento devem ser agrupadas conjuntamente. Deve haver linhas em branco entre os procedimentos. Cada objetivo deve ser escrito em uma nova linha. Segundo Bratko [Bra 86]:

"Programas Prolog muitas vezes lembram poemas, devido ao apelo estético produzido pelas idéias e formas que contém."

- Convenções de estilo desse tipo podem variar de programa para programa, uma vez que dependem do problema e do gosto de cada programador. É importante que as mesmas convenções sejam usadas de forma consistente em todo o programa;
- O operador cut deve ser usado com cuidado. Seu uso deve ser evitado quando não for absolutamente necessário. Se não for possível evitar o uso de cuts, é melhor usar apenas os cuts "verdes" e jamais os "vermelhos". Como foi discutido no capítulo 6, um cut é "verde" quando pode ser removido sem alterar o significado declarativo da cláusula em que se encontra. Caso contrário o cut é "vermelho";
- O operador not, devido a sua relação com o cut também pode apresentar comportamento inesperado. É necessário ter completo conhecimento sobre a forma em que o not é definido em Prolog:

```
not(P) :- P, !, fail; true.
```

bem como as consequências da adoção da hipótese do mundo fechado na execução da negação em Prolog (ver capítulo 6). Se entretanto estivermos em dúvida entre usar o not ou o cut, o primeiro é preferível a alguma construção obscura com o uso do cut;

- A modificação do programa por meio dos predicados assert/1 e retract/1 pode degradar em grande escala a transparência do seu comportamento. Em particular, um mesmo programa pode responder a mesma consulta de maneira diferente em momentos diferentes. Em tais casos, se quisermos reproduzir o mesmo comportamento, temos que nos certificar que todos os estados anteriores do programa, desde o início de sua execução, foram perfeitamente reproduzidos;
- O uso do ponto-e-vírgula (correspondendo ao conetivo "ou") pode obscurecer o significado de uma cláusula. A legibilidade pode ser algumas vezes incrementada pela divisão da cláusula que contém o ";" em duas.

Para ilustrar os pontos discutidos até aqui, vamos considerar a seguinte relação:

```
merge(L1, L2, L3)
```

onde L1 e L2 são listas ordenadas que são reunidas ordenadamente em L3. Por exemplo:

```
merge([2, 4, 7], [1, 3, 4, 8], [1, 2, 3, 4, 4, 7, 8])
```

A implementação abaixo é um contra-exemplo de definição da relação merge/3, empregando um estilo que deixa muito a desejar:

```
merge(L1, L2, L3) :-
    L1 = [], !, L3 = L2;
    L2 = [], !, L3 = L1;
    L1 = [X | Resto1],
    L2 = [Y | Resto2],
    (X < Y, !, Z = X, merge(Resto1, L2, Resto3);
     (Z = Y, merge(L1, Resto2, Resto3))),
    L3 = [Z | Resto3].
```

Já a versão a seguir possui um estilo muito mais transparente e legível, além de ser com certeza mais eficiente, uma vez que tira partido da unificação dos argumentos correspondentes ao caso básico na própria cabeça da cláusula:

```
merge([], L, L).
merge(L, [], L).
merge([X | R1], [Y | R2], [X | R3]) :-
    X < Y, !, merge(R1, [Y | R2], R3).
merge(L1, [Y | R2], [Y | R3]) :-
    merge(L1, R2, R3).
```

12.3.2 ORGANIZAÇÃO TABULAR DE PROCEDIMENTOS LONGOS

Procedimentos longos podem vir a ser aceitáveis, desde que apresentem uma estrutura uniforme. Uma estrutura uniforme típica é a formada por um conjunto de fatos que efetivamente definem uma relação em forma tabular. As vantagens de tal organização são as seguintes:

- A estrutura é facilmente entendida,
- A estrutura é incremental: pode ser facilmente refinada pela adição de novos fatos, e
- É de fácil verificação, correção e modificação (pela simples substituição de algum fato, independentemente dos demais).

12.3.3 O USO DE COMENTÁRIOS

Os comentários no programa devem, antes de mais nada, explicar a sua finalidade e como deve ser utilizado. Somente depois disso é que devem aparecer os detalhes do método empregado e outras características do programa. O propósito inicial dos comentários é facilitar ao usuário o uso do programa e, se for o caso, a sua modificação. Os comentários devem descrever, da forma mais sucinta possível, porém sem perda de informação, tudo o que for essencial para tais finalidades. Um erro muito comum é a produção de programas sub-comentados, entretanto, programas supercomentados também não são desejáveis. A explicação de detalhes óbvios da codificação de um programa é uma carga desnecessária. Longos trechos de comentários devem preceder o código ao qual se referem, enquanto que pequenas notas devem ser intercaladas na própria codificação. A informação que normalmente deve ser incluída como comentário compreende o seguinte:

- O que o programa faz, como deve ser utilizado, (por exemplo, que tipo de consulta deve ser formulada e quais são os resultados esperados) e exemplos de utilização;
- Descrição dos predicados de nível mais alto;
- Descrição dos principais conceitos representados;
- Tempos de execução e requisitos de memória;
- Limitações do programa;
- Utilização de recursos especiais dependentes do hardware;
- Idem com relação ao software básico;
- Descrição dos predicados do programa;
- Detalhes algorítmicos e de implementação.

12.4 DEPURAÇÃO DE PROGRAMAS

Quando um programa apresenta um comportamento diferente do esperado, o principal problema passa a ser a localização do(s) erro(s). É mais fácil localizar um erro em uma parte do programa ou módulo

do que no programa inteiro, portanto, um bom princípio de correção de programas é começar pelo teste de pequenas unidades do programa e, quando estas forem consideradas confiáveis, passar a testar módulos maiores até que o programa inteiro possa ser testado.

A correção de programas em Prolog é facilitada por duas circunstâncias: primeiro, Prolog é uma linguagem interativa, de forma que qualquer parte do programa pode ser ativada diretamente por meio de uma consulta apropriada; segundo, as implementações Prolog normalmente oferecem ferramentas especiais para "debugging". Como resultado desses dois recursos, a correção de programas em Prolog pode, em geral, ser executada de forma bem mais eficiente do que a maioria das linguagens de programação.

A ferramenta básica para a depuração de programas é o processo de "tracing". Sua aplicação a um objetivo significa que as informações associadas à satisfação desse objetivo irão sendo apresentadas durante a execução. Tais informações incluem:

- **Informação de Entrada:** O nome do predicado e os valores dos argumentos quando o objetivo é disparado;
- **Informação de Saída:** No caso do objetivo ser bem sucedido, são apresentados os valores dos argumentos que o satisfazem. Em caso contrário, a indicação de falha no ponto em que esta ocorreu;
- **Informação de Reentrada:** Na chamada do mesmo objetivo através de backtracking.

Entre a entrada e a saída, pode-se obter a mesma informação de todos os sub-objetivos envolvidos, de forma que podemos dispor do tracing da execução de qualquer consulta ao programa, desde os níveis mais elevados até que os fatos correspondentes sejam encontrados. Isso pode, em determinadas circunstâncias, ocasionar um excesso de informação, assim, é permitido ao usuário especificar um tracing seletivo. Há dois mecanismos dedicados a essa seleção: primeiro, suprimir a informação de tracing além de determinado nível; segundo, executar o tracing apenas sobre algum subconjunto específico de predicados e não sobre o programa inteiro. Tais ferramentas para a depuração de programas são ativadas por meio de predicados pré-definidos que variam de uma implementação para outra. Um conjunto típico desses predicados é o seguinte:

- **trace:** Dispara um processo exaustivo de tracing para todos os objetivos que se seguirem;
- **notrace:** Interrompe o processo de tracing;
- **spy(P):** Especifica o nome de uma relação P para o processo de tracing. O predicado spy/1 é empregado quando se está particularmente interessado no comportamento da relação nomeada e se deseja evitar o tracing de outros objetivos (tanto acima quanto abaixo de P);
- **nospy(P):** Interrompe o tracing da relação P.

O processo de tracing pode ser interrompido além de uma certa profundidade por meio de comandos especiais acionados durante a execução. Dependendo da implementação pode haver ainda diversos comandos de depuração disponíveis, tais como retornar a um determinado ponto anterior da execução. Após tal retorno podemos, por exemplo, repetir a execução de forma mais detalhada.

12.5 EFICIÊNCIA

Há diversos aspectos de eficiência, incluindo os mais comuns: tempo de execução e consumo de memória de um programa. Um outro aspecto, pouco considerado mas indubitavelmente de grande importância é o tempo consumido no desenvolvimento de um programa. A arquitetura dos computadores convencionais não é especialmente adequada para o estilo de execução de programas adotado pelo Prolog - ou seja, a satisfação de uma lista de objetivos. Portanto, as limitações de espaço e tempo a

que todas as linguagens de programação estão sujeitas, podem vir a ser sentidas antes pelos programas Prolog.

Por outro lado, em muitas áreas de aplicação, o uso do Prolog vai reduzir consideravelmente o tempo de desenvolvimento, pois os programas Prolog são em geral mais fáceis de escrever, entender e depurar do que os escritos em linguagens convencionais. Problemas que gravitam em torno do "domínio Prolog" envolvem processamento simbólico, não-numérico, sobre objetos estruturados e as relações entre eles. Em particular, o uso de Prolog tem sido especialmente bem sucedido em áreas envolvendo a solução simbólica de equações, planejamento, bases de dados, solucionadores genéricos, prototipação, implementação de linguagens de programação, simulação discreta e qualitativa, projeto arquitetônico, aprendizado de máquina, interpretação da linguagem natural, sistemas especialistas e diversas outras áreas da inteligência artificial. Sob outro ângulo, matemática numérica é uma área na qual os programas Prolog não conseguem competir.

Com respeito a eficiência na execução, um programa compilado é sempre mais eficiente do que um programa interpretado, portanto, se o sistema Prolog adotado possui um interpretador e um compilador, este último deve ser usado preferencialmente ao primeiro quando a eficiência se tornar um ponto crítico. Se um programa se apresenta ineficiente, muitas vezes isso pode ser radicalmente modificado por meio de alterações no seu próprio algoritmo, entretanto, para fazer isso, os aspectos procedimentais do programa devem ser considerados. Uma maneira simples de aumentar a eficiência de um programa é encontrar uma ordenação mais adequada para as cláusulas no interior dos procedimentos e para os objetivos no interior das cláusulas. Um outro método, relativamente simples, é a introdução de cuts em posições apropriadas. Idéias para aumentar a eficiência de um programa normalmente surgem de um entendimento mais profundo do problema. Um algoritmo mais eficiente resulta de melhorias de dois tipos:

- Aumento na eficiência de busca, evitando backtracking desnecessário e interrompendo a execução de alternativas inúteis o mais cedo possível, e
- Uso de estruturas de dados mais adequadas para a representação de objetos no programa, de forma que as operações sobre esses objetos possam ser implementadas de maneira mais eficiente.

Esses dois tipos de melhorias serão abordados em maior detalhe nos exemplos apresentados nas próximas seções.

12.5.1 O PROBLEMA DE COLORIR UM MAPA

O problema de colorir um mapa corresponde a atribuir, a cada país em um determinado mapa, uma certa cor, escolhida de um conjunto de quatro cores diferentes, de maneira que dois países vizinhos nunca sejam coloridos com a mesma cor. Há um teorema que garante que isso sempre é possível de ser feito. Vamos assumir que o mapa seja especificado pela relação:

`viz(País, Vizinhos)`

onde Vizinhos é a lista de todos os países que possuem alguma fronteira em comum com País. Assim, o mapa da América do Sul, com 13 países, seria especificado em ordem alfabética por:

```
viz(argentina, [bolívia, brasil, chile, paraguai, uruguai]).
viz(bolívia,   [argentina, brasil, chile, paraguai, peru]).
viz(brasil,    [argentina, bolívia, colômbia, guiana,
guiana_francesa, paraguai, suriname,
uruguai, venezuela]).
...
```

Uma possível solução para o problema das cores de cada país seria representar a correspondência entre estes e suas cores por uma lista de pares do tipo:

País/Cor

que especifica uma cor para cada país em um determinado mapa. Para o mapa proposto, os nomes dos países são dados antecipadamente e o problema será encontrar a cor adequada para colorir cada um deles. Assim, no caso da América do Sul, o problema corresponde a encontrar uma instanciade adequada para as variáveis C1, C2, C3, etc. na lista:

```
[argentina/C1, bolívia/C2, brasil/C3, ...]
```

Para isso define-se a relação `cores/1`, cujo único argumento é a lista acima e que será verdadeira se a lista satisfizer a restrição do colorido do mapa, com respeito à relação `viz/2` definida anteriormente. Sejam as cores escolhidas azul, amarelo, vermelho e verde. A condição de que dois países vizinhos não podem ter a mesma cor pode ser formulada em Prolog por:

```
cores([]).
cores([País/Cor | Resto]) :-
    cores(Resto),
    membro(Cor, [azul, amarelo, vermelho, verde]),
    not(membro(País1/Cor, Resto), vizinho(País, País1)).

vizinho(País, País1) :-
    viz(País, Vizinhos), membro(País1, Vizinhos).
```

onde `membro/2` é a relação usual de ocorrência em listas. O procedimento `cores/1` funciona relativamente bem para mapas simples, com poucos países, entretanto, para mapas complexos como o da América do Sul, sua eficiência deixará a desejar. Assumindo que o predicado pré-definido `setof/3` esteja disponível, uma tentativa de colorir a América do Sul poderia ser a seguinte: Primeiro define-se uma relação auxiliar:

```
país(P) :- viz(P, _).
```

Então uma consulta adequada para colorir a América do Sul poderia ser formulada por:

```
?-setof(P/Cor, país(P), Lista), cores(Lista).
```

O objetivo `setof/3` irá primeiro construir uma lista de itens `P/Cor`, na qual as cores serão representadas por variáveis não-instanciadas. Depois o objetivo `cores/1` irá produzir a instanciade adequada. É provável, entretanto, que essa tentativa falhe devido à sua ineficiência. Um estudo detalhado de como o Prolog tenta satisfazer o objetivo `cores/1` revela a fonte de tal ineficiência. Os países em `Lista` são organizados em ordem alfabética, que não tem nada a ver com a sua disposição geográfica. A ordem em que as cores são atribuídas aos países corresponde à ordem da `Lista` (começando pelo final), o que é, no caso em questão, independente da relação `viz/2`. Assim, o processo de colorir os países começa em algum ponto do mapa, continua em um outro extremo, etc, movendo-se de forma mais ou menos aleatória. Isso pode conduzir facilmente a uma situação na qual um país que deva ser colorido encontra-se rodeado por outros países já coloridos com todas as quatro cores disponíveis, sendo então necessário acionar o mecanismo de backtracking, com elevado ônus para a eficiência do programa.

Fica claro então que a eficiência depende da ordem na qual os países serão coloridos. A intuição sugere uma estratégia simples de ordenação que apresenta um desempenho muito superior ao método aleatório. Começa-se com algum país que tenha muitos vizinhos. Depois são coloridos os seus vizinhos. Depois os vizinhos dos vizinhos e assim por diante. Para a América do Sul, então, o Brasil (que faz fronteira com nove países, parece ser um bom candidato para iniciar o processo. Assim, quando a lista de `País/Cor` for construída, o Brasil deve ser colocado no fim, com todos os demais países o antecedendo. Dessa forma o algoritmo, que começa a processar a partir do último elemento da lista iniciará com o Brasil e continuará dali a processar os países vizinhos como foi explicado anteriormente.

Essa nova ordenação aumenta muito a eficiência do programa em comparação com a ordenação alfabética original, produzindo sem dificuldade os possíveis coloridos do mapa da América do Sul.

Pode-se construir manualmente uma lista apropriada dos países da América do Sul, mas não é necessário fazer isso. O procedimento `fazLista/1`, definido abaixo executará essa tarefa para nós. Ele inicia a construção com algum país especificado (Brasil, no nosso caso) e coleta os países em uma lista denominada "Fechada". Cada país é, inicialmente colocado em outra lista, denominada "Aberta", antes

de ser transferido para Fechada. Toda vez que um país for transferido de Aberta para Fechada, os seus vizinhos serão colocados em Aberta.

```
fazLista(Lista) :-
    coleta([brasil], [], Lista).

coleta([], Fechada, Fechada).
coleta([X | Aberta], Fechada, Lista) :-
    membro(X, Fechada), !,
    coleta(Aberta, Fechada, Lista).
coleta([X | Aberta], Fechada, Lista) :-
    viz(X, Vizinhos),
    conc(Vizinhos, Aberta, Aberta1),
    coleta(Aberta1, [X | Fechada], Lista).
```

onde a relação conc/3 é a relação já estudada anteriormente para a concatenação de listas.

12.5.2 APERFEIÇOANDO AS ESTRUTURAS DE DADOS

Nos programas apresentados até aqui, a concatenação de listas tem sido programada da seguinte maneira:

```
conc([], L, L).
conc([X | L1], L2, [X | L3]) :-
    conc(L1, L2, L3).
```

Essa forma de programar a concatenação de listas pode tornar-se bastante ineficiente quando a primeira lista é muito longa, uma vez que esta deve ser inteiramente percorrida até que a lista vazia seja encontrada. Para tornar a relação conc/3 verdadeiramente eficiente, deve-se pular diretamente para o fim da primeira lista em um único passo de computação. Isso somente é possível se soubermos localizar o fim de uma lista, o que não pode ser feito a partir da representação adotada até o momento. É necessário portanto uma nova representação para listas. Uma solução possível é representar cada lista por meio de um par de listas. Por exemplo, a lista [a, b, c] pode ser representada por meio de duas listas:

$L1 = [a, b, c, d, e] \quad \text{e} \quad L2 = [d, e]$

Esse par de listas, que denotaremos por L1-L2, representa a diferença entre L1 e L2. Isso, naturalmente, só vai funcionar se a lista L2 for um sufixo de L1. Note que a mesma lista pode ser representada por diversos pares-diferença. Por exemplo, a lista [a, b, c] pode ser representada por:

```
[a, b, c] - []
[a, b, c, d, e] - [d, e]
[a, b, c, d | T] - [d | T]
[a, b, c, | T] - T
...
```

A lista vazia é representada por qualquer par L-L. Como o segundo membro do par indica o final da lista, este passa a poder ser acessado diretamente. Isso pode ser usado para uma implementação muito mais eficiente da concatenação de listas. O método proposto é ilustrado na figura 12.1 e a correspondente relação em Prolog que denominaremos concat/3 pode ser representada por um único fato:

concat(A1-Z1, Z1-Z2, A1-Z2)

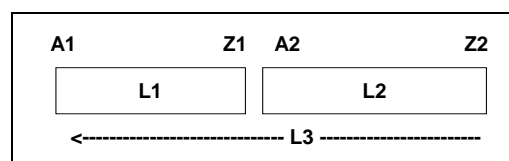


Figura 12.1 Concatenação de listas representadas por pares-diferença

Na figura acima, L1 é representada por A1-Z1, L2 por A2-Z2 e o resultado, L3, por A1-Z2, o que é verdadeiro quando Z1=A2. Vamos usar a relação concat/3 para concatenar as listas [a, b, c] (repre-

sentada pelo par [a, b, c, | T1] - T1 e [d, e] (representada pelo par [d, e | T2] - T2). A concatenação é obtida pela simples unificação do objetivo proposto na consulta com a cláusula que define concat/3.

```
?-concat([a, b, c | T1] - T1, [d, e | T2] - T2, Lista.  
T1 = [d, e | T2]  
Lista = [a, b, c, d, e | T2] - T2
```

12.5.3 DECLARAÇÃO DE FATOS INFERIDOS

Algumas vezes, durante a computação, o mesmo objetivo tem que ser satisfeito várias vezes. Como o Prolog não possui nenhum mecanismo adequado para identificar essa situação, toda a sequência de computações será repetida cada vez que o objetivo tiver de ser satisfeito. Como um exemplo, vamos considerar um programa para computar o enésimo número da sequência de Fibonacci. A sequência de Fibonacci é:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

onde cada número, com exceção dos dois primeiros, é a soma dos dois números anteriores. Definiremos um predicado fib(N, F) para computar para um dado número N, o enésimo número F da sequência de Fibonacci. Contaremos os números da sequência iniciando com N=1. O programa a seguir trata inicialmente os dois primeiros números de Fibonacci como casos especiais e depois especifica a regra geral para a geração da sequência.

```
fib(1, 1).  
fib(2, 1).  
fib(N, F) :-  
    N > 2,  
    N1 is N-1, fib(N1, F1),  
    N2 is N-2, fib(N2, F2),  
    F is F1+F2.
```

Esse programa tende a refazer partes da computação. Isso pode ser facilmente constatado se gerarmos o tracing da execução de uma consulta, por exemplo, ?-fib(6, F). A repetição desnecessária de computações intermediárias pode ser facilmente evitada se o programa "lembrar" cada um dos números de Fibonacci gerados como resultados parciais. A idéia é utilizar o predicado pré-definido assert/1 e adicionar esses resultados parciais à base de dados na forma de fatos. Esses fatos devem preceder todas as outras cláusulas sobre fib para prevenir o uso da regra geral nos casos em que o resultado já é conhecido. O procedimento modificado fibo/2 difere de fib/2 apenas pela inclusão de um objetivo adicional:

```
fibo(1, 1).  
fibo(2, 1).  
fibo(N, F) :-  
    N > 2,  
    N1 is N-1, fibo(N1, F1), N2 is N-2, fibo(N2, F2),  
    F is F1+F2,  
    asserta(fibo(N, F)).
```

Guardar os resultados intermediários é uma técnica convencional para evitar computações repetidas. No caso dos números de Fibonacci podemos evitar essa repetição por meio do uso de outro algoritmo, diferente do proposto acima. Esse novo algoritmo produzirá um programa mais difícil de entender, porém de execução mais eficiente. A idéia básica é não definir o enésimo número de Fibonacci como a simples soma de seus dois antecessores imediatos, deixando que chamadas recursivas completem o processamento recuando até os dois primeiros números de Fibonacci. Ao invés disso, podemos trabalhar "para frente" começando com os dois números iniciais e computando os números na sequência natural, parando quando o enésimo número for encontrado. A maior parte do trabalho é executada pelo procedimento

```
geraFib(M, N, F1, F2, F)
```

onde F1, F2 e F são respectivamente o (M-1)-ésimo, o M-ésimo e o N-ésimo número da seqüência. O procedimento geraFib/5 encontra uma seqüência de transformações até atingir uma configuração final (quando M=N), a partir de uma configuração inicial.

Quando geraFib/5 é ativado, todos os argumentos, com exceção de F, devem estar instanciados e M deve ser menor ou igual a N. O programa fica então:

```
fibonacci(N, F) :-
    geraFib(2, N, 1, 1, F).

geraFib(M, N, F1, F2, F) :-
    M >= N.
geraFib(M, N, F1, F2, F) :-
    M < N, ProxM is M+1, ProxF2 is F1+F2,
    geraFib(ProxM, N, F2, ProxF2, F).
```

12.6 PROGRAMAÇÃO ITERATIVA

Como foi visto, Prolog é uma linguagem recursiva, requerendo portanto uma certa maturidade em termos de pensamento recursivo por parte de seus programadores. Também o backtracking é uma poderosa técnica para os mais diversos propósitos. Por outro lado, recursão e backtracking são difíceis de combinar, porque a recursão constrói estruturas durante chamadas sucessivas que são esquecidas durante o backtracking, a menos que o predicado assert/1 (ou suas opções) seja empregado para lembrá-las. Nesse caso porém a execução do programa produz uma estrutura muito complexa e intrincada.

Esta situação pode ser melhorada com o emprego de algumas técnicas de programação estruturada. Nem todos os problemas recursivos o são no sentido profundo da palavra. Alguns são apenas "iterativos" e devem ser reconhecidos como tal. Um predicado pré-definido desenvolvido para executar iteração é o repeat/0, que se comporta exatamente como se houvesse sido definido por:

```
repeat.
repeat :- repeat.
```

Por exemplo, um laço para executar processamento de entrada e saída poderia assumir a forma seguinte:

```
loop :-
    repeat,
    read(X),
    (X = fim, !; processa(X,Y), write(Y), nl, fail).
```

Tal procedimento irá ler um termo, processá-lo, imprimir alguma saída e falhar, ocasionando por backtracking a repetição destas operações até que o termo lido seja "fim".

Considere agora um programa para imprimir todos os elementos de uma lista L. Uma solução recursiva seria:

```
imprimeLista([]).
imprimeLista([X | Y]) :-
    write(X), nl, imprimeLista(Y).
```

Esse padrão de recursão é comum em Prolog. Pode-se entretanto generalizá-lo definindo um predicado:

```
for(X, Y) :-
    X, Y, fail.
for(X, Y).
```

de modo que para todas as soluções de X, Y será ativado. O laço produzido pelo predicado for/2 termina quando não houver mais soluções para X. A impressão de todos os elementos de uma lista assumiria então a forma abaixo:

```
imprimeLista(L) :-
    for(membro(X, L), (write(X), nl)).
```

Um outro exemplo seria o problema de listar todos os números de um a dez. Uma solução recursiva seria:

```
listaNúmeros(N) :- N > 10, !.
listaNúmeros(N) :-
    write(N), nl, N1 is N+1, listaNúmeros(N1).

?-listaNúmeros(1).
1
2
...
10
```

Ao invés disso definiremos um predicado in/3 que, por backtracking, retorna com os valores de 1 a N:

```
in(I, I, H) :-
    H >= I.
in(I, L, H) :-
    N is L+1, in(I, N, H).

?-for(in(I, 1, 10), (write(I), nl)).
```

O predicado for/3 pode ser combinado de diversas formas diferentes. Por exemplo, para imprimir tabelas de multiplicação:

```
tabMult :-
    for(in(I, 1, 10),
        (for(in(J, 1, 10),
            (K is I*J, out(K))), nl)).

out(X) :-
    write(X), write(' ').
```

Como uma aplicação do conceito acima, e ao mesmo tempo como um exercício de utilização de operadores, eis como é possível fazer programas iterativos em Prolog se parecerem com Pascal:

```
:-op(1110, xfy, do).
:-op(1109, fx, for).
:-op(1108, xfx, to).
:-op(1107, fx, begin).
:-op(700, xfx, ':=').

dol(begin end) :- !.
dol(begin X ; Y) :- !, dol(X), dol(begin Y).
dol(begin X) :- !, dol(X).
dol(X) :- X.

X := Y :- X is Y.

(for X := Y to Z do U) :- for(in(X, Y, Z), dol(U)).

writeln(X) :- write(X), nl.
```

A partir da definição acima, podemos escrever o seguinte programa para calcular e imprimir os quadrados dos números de 1 a 10:

```
quadrados :-
    for I := 1 to 10 do
        begin
            K := I*I;
            writeln(K);
        end.
```

Essa construção, apesar da aparência, continua sendo um programa em Prolog. Deve-se entretanto advertir o leitor para não tomar esse exemplo como uma tentativa séria de implementar um interpretador Pascal em Prolog. Por exemplo, é difícil modelar estruturas aninhadas tais como begin-end, porque tais constantes são na verdade delimitadores, como os parênteses, e não operadores.

RESUMO

- Há diversos critérios para a avaliação de programas, entre outros:

Correção;

Eficiência;
Transparência e Legibilidade;
Modificabilidade;
Robustez;
Documentação.

- O princípio dos refinamentos sucessivos é uma boa maneira de organizar o processo de desenvolvimento de programas. Em Prolog essa técnica pode ser tanto aplicada às relações e algoritmos quanto às estruturas de dados;
- As seguintes técnicas frequentemente auxiliam o programador Prolog a encontrar idéias para os refinamentos:

Recursão;
Generalização;
Uso de Gráficos.

- É de grande utilidade o emprego de convenções de estilo para reduzir o perigo de erros de programação e tornar os programas mais fáceis de ler, depurar e modificar;
- As diferentes implementações da linguagem Prolog normalmente oferecem ferramentas auxiliares para a depuração de programas, dentre as quais o mecanismo de tracing é uma das mais úteis;
- Maneiras de aumentar a eficiência de programas Prolog são:
 - Reordenação de objetivos e cláusulas;
 - Controle do backtracking por meio de cuts;
 - Uso de assert/1 para evitar recomputação;
 - Uso de algoritmos e estruturas mais eficientes;
 - Uso de iteração preferivelmente à recursão.

EXERCÍCIOS

12.1 Defina um predicado para inverter o predicado que encontra a soma dos elementos em uma lista de valores inteiros.

12.2 Defina um programa que registre os elementos de uma lista nomeada como fatos individualmente numerados. Por exemplo:

```
enumera(tab, [a, b, p(c), a])
```

produziria os seguintes fatos:

```
elem(tab, 1, a).  
elem(tab, 2, b).  
elem(tab, 3, p(c)).  
elem(tab, 4, a).
```

12.3 Defina a relação adiciona(Lista, Item, NovaLista) para adicionar Item ao final de Lista produzindo NovaLista, sendo que tanto Lista quanto NovaLista devem ser representadas por pares-diferença.

12.4 Defina a relação inverte(Lista, ListaInvertida) onde ambas as listas são representadas por pares-diferença.

12.5 Defina os operadores e as relações necessárias para representar as construções if-then-else e while-do no estilo Pascal.

13. OPERAÇÕES SOBRE ESTRUTURAS DE DADOS

Uma questão fundamental da programação é como representar objetos complexos, tais como conjuntos e implementar operações eficientes sobre tais objetos. Como foi visto no capítulo anterior, a seleção da estrutura de dados apropriada é essencial para garantir a eficiência de tais operações. No presente capítulo serão examinadas as estruturas de dados mais frequentemente utilizadas, que pertencem a três grandes famílias: listas, árvores e grafos, e diversos exemplos serão desenvolvidos visando ilustrar o seu uso e adequação.

13.1 CLASSIFICAÇÃO EM LISTAS

Classificação é uma operação frequentemente necessária em diversos contextos. Uma lista pode ser classificada desde que haja uma relação de ordem entre os elementos que a compõem. Para os propósitos assume-se a relação de ordem representada por

`mq(X, Y)`

significando que "X é maior que Y", independentemente do que significa "maior que". Se os itens da lista são números, então a relação `mq/2` será talvez definida por:

`mq(X, Y) :- X > Y.`

Se os itens da lista são átomos, então a relação `mq` pode corresponder, por exemplo, à ordem do código ASCII correspondente aos caracteres. Vamos considerar que

`classifica(Lista, Saída)`

denote uma relação onde *Lista* é uma lista de itens e *Saída* é uma lista dos mesmos itens classificados em ordem crescente, de acordo com a relação `mq/2`. Desenvolveremos três definições de tal relação em Prolog, baseadas em diferentes idéias sobre a classificação de listas. A primeira delas é bastante simples, chegando a ser ingênua: Para classificar uma lista *Lista*:

- (1) Encontre dois elementos adjacentes, *X* e *Y*, nesta ordem em *Lista*, tais que `mq(X, Y)`. Troque as posições de *X* e *Y*, obtendo *Lista1* e, depois, classifique *Lista1*;
- (2) Se não houver nenhum par de elementos adjacentes, *X* e *Y*, nesta ordem em *Lista*, então esta já está classificada.

O propósito da troca das posições dos itens *X* e *Y*, que aparecem fora de ordem em *Lista* é que, após a troca, a nova lista obtida está mais próxima de ser uma lista classificada.. Após um determinado número de trocas de posição, a lista estará completamente ordenada. Esse princípio de classificação é denominado "bubble sort" (ou classificação "bôlha"). A relação correspondente, `bubblesort/2`, é apresentada abaixo:

```
bubblesort(Lista, Saída) :-
    troca(Lista, Lista1), !, bubblesort(Lista1, Saída).
bubblesort(Saída, Saída).

troca([X, Y | Resto], [Y, X | Resto]) :-
    mq(X, Y).
troca([Z | Resto], [Z | Resto1]) :-
    troca(Resto, Resto1).
```

Um outro algoritmo simples de classificação é o sort por inserção (insert sort), que se baseia na seguinte idéia: Para classificar uma lista não vazia, $L = [X | R]$:

- (1) Classifique o corpo *R* da lista *L*;
- (2) Insira a cabeça, *X*, de *L* no corpo classificado em uma posição tal que a lista resultante esteja classificada.

O resultado é a lista completamente classificada. Esse algoritmo é representado em Prolog pela relação `insertsort/2`:

```

insertsort([], []).
insertsort([X | Resto], Saída) :-
    insertsort(Resto, Resto1), insere(X, Resto1, Saída).

insere(X, [Y | Saída], [Y | Saída1]) :-
    mq(X, Y), !, insere(X, Saída, Saída1).
insere(X, Saída, [X | Saída]).

```

Os procedimentos de classificação bubblesort/2 e insertsort/2 são simples, porém ineficientes. Dos dois, o último é o mais eficiente, entretanto, o tempo médio que o insertsort/2 requer para classificar uma lista de tamanho n cresce proporcionalmente a n^2 . Para listas muito longas, portanto, um algoritmo melhor é o quicksort/2, baseado na idéia abaixo e ilustrado na Figura 13.1. Para classificar uma lista não vazia, L:

- (1) Separe algum elemento X de L e divida o restante em duas listas, denominadas Menor e Maior, da seguinte maneira: Todos os elementos de L que são maiores do que X pertencem a Maior e os restantes pertencem a Menor;
- (2) Classifique Menor, obtendo Menor1;
- (3) Classifique Maior, obtendo Maior1;
- (4) A lista completa é a concatenação de Menor1 com [X | Maior1].

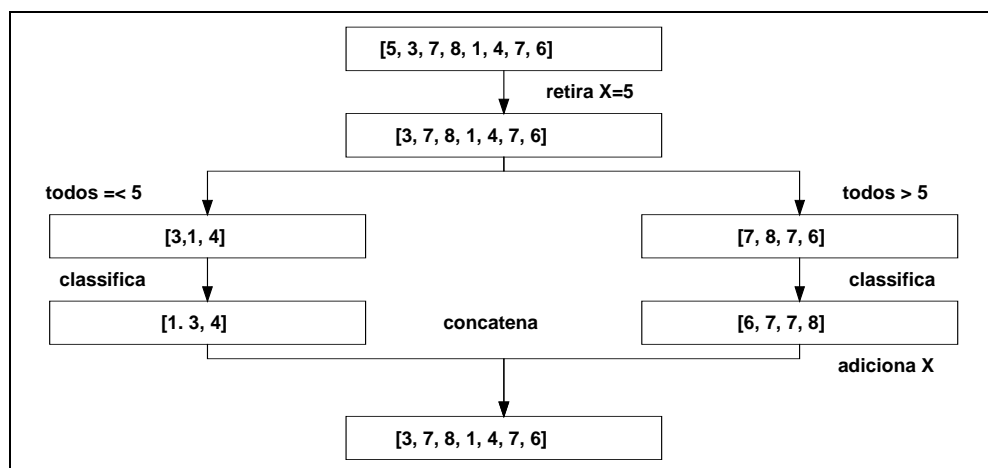


Figura 13.1: Classificando uma lista com o algoritmo quicksort/2

Se a lista a ser classificada estiver vazia, então o resultado da classificação é também uma lista vazia. Uma implementação do quicksort/2 em Prolog é apresentada na Figura 13.2. Um detalhe particular dessa implementação é que o elemento X que é retirado de L é sempre a cabeça da lista. O procedimento que divide L em Maior e Menor é uma relação de quatro argumentos:

```
divide(X, L, Maior, Menor)
```

A complexidade temporal deste algoritmo depende de nossa sorte ao dividirmos a lista a ser classificada. Se a lista for dividida em duas outras com aproximadamente o mesmo tamanho, então a complexidade temporal do procedimento será proporcional a $n \cdot \log(n)$, onde n é o tamanho da lista a classificar. Se, ao contrário, a divisão resultar em duas listas de tamanho muito desigual, então a complexidade será da ordem n^2 . Análises mais acuradas mostram que, felizmente, o desempenho médio do algoritmo quicksort/2 se aproxima bem mais da primeira situação do que da segunda.

```

quicksort([X | R], Saída) :-
    divide(X, R, Maior, Menor),
    quicksort(Menor, Menor1),
    quicksort(Maior, Maior1),
    conc(Menor1, [X | Maior1], Saída).

divide(X, [], [], []).
divide(X, [Y | R], [Y | Menor], Maior) :-
    mq(X, Y), !, divide(X, R, Menor, Maior).
divide(X, [Y | R], Menor, [Y | Maior]) :-
    divide(X, R, Menor, Maior).

conc([], L, L).
conc([X | L1], L2, [X | L3]) :-
    conc(L1, L2, L3).

```

Figura 13.2: Uma implementação do algoritmo quicksort/2 em Prolog

13.2 REPRESENTAÇÃO DE CONJUNTOS

Uma aplicação usual para listas é a representação de conjuntos de objetos, entretanto, tal representação não é adequada, uma vez que o teste de ocorrência de um item em uma lista se mostra relativamente ineficiente como teste de pertinência de um elemento a um conjunto. O predicado `membro(X, L)`, que verifica se `X` é membro da lista `L`, é usualmente programado como:

```

membro(X, [X | _]).
membro(X, [_ | Y]) :-
    membro(X, Y).

```

Para encontrar `X` em uma lista `L`, esse procedimento percorre a lista elemento por elemento até que `X` seja encontrado ou o fim da lista seja atingido. Isso se torna especialmente ineficiente no caso de listas muito longas. Para a representação de conjuntos há diversas estruturas em árvore que possibilitam uma implementação muito mais eficiente da relação de pertinência. Vamos considerar neste ponto as denominadas "árvores binárias". Uma árvore binária, ou é vazia, ou é constituída por três argumentos:

- (1) uma raiz,
- (2) uma sub-árvore esquerda, e
- (3) uma sub-árvore direita.

A raiz pode ser qualquer coisa, mas as sub-árvores devem necessariamente ser árvores. Na Figura 13.3, é apresentado um exemplo. A árvore ali mostrada representa o conjunto $\{a, b, c, d\}$. Os elementos do conjunto são armazenados nos nodos da árvore e, normalmente, sub-árvores vazias não são representadas. Por exemplo, o nodo "b" possui duas sub-árvores que são ambas vazias.

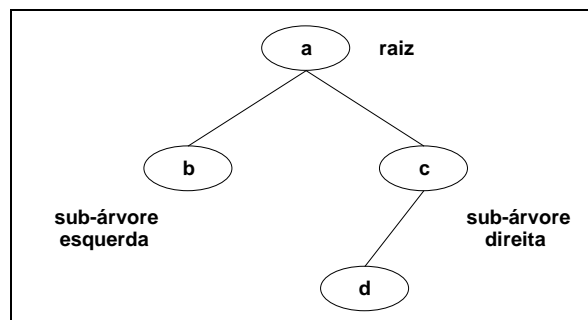


Figura 13.3 Uma árvore binária

Há diversas maneiras de se representar uma árvore binária através de um termo Prolog. Uma possibilidade simples é tornar a raiz da árvore o functor principal do termo e as sub-árvores os seus argumentos. Assim a árvore da Figura 13.3 seria representada pelo termo:

```
a(b, c(d)).
```

Entre outras desvantagens, essa representação requer um novo functor para cada nodo da árvore. Isso pode ocasionar problemas, se os nodos, por sua vez, forem objetos estruturados. Uma maneira melhor e mais usual de representar árvores binárias é o seguinte: Emprega-se um símbolo especial para representar a árvore vazia e um functor para representar árvores não-vazias a partir de seus três componentes (a raiz e as duas sub-árvores). Assumiremos o seguinte:

- A árvore vazia será representada pelo átomo "nil", e
- Será empregado um functor t, de forma que a árvore que possui uma raiz R, uma sub-árvore esquerda E e uma sub-árvore direita D seja representada pelo termo:

$t(E, R, D).$

Nessa representação a árvore da Figura 13.3 corresponderia ao termo:

$t(t(\text{nil}, b, \text{nil}), a, t(t(\text{nil}, d, \text{nil}), c, \text{nil}))$

Vamos agora considerar a relação de pertinência para conjuntos, que denominaremos *pertence/2*. O objetivo *pertence(X,T)* é verdadeiro se X é um nodo da árvore T. A relação *pertence/2* pode ser definida da seguinte maneira:

X *pertence* a uma árvore T se:

- A raiz de T é X, ou
- X está na sub-árvore esquerda de T, ou
- X está na sub-árvore direita de T.

Tais regras podem ser traduzidas diretamente para Prolog, da seguinte maneira:

```
pertence(X, t(_, X, _)) :- !.
pertence(X, t(E, _, _)) :-
    pertence(X, E).
pertence(X, t(_, _, D)) :-
    pertence(X, D).
```

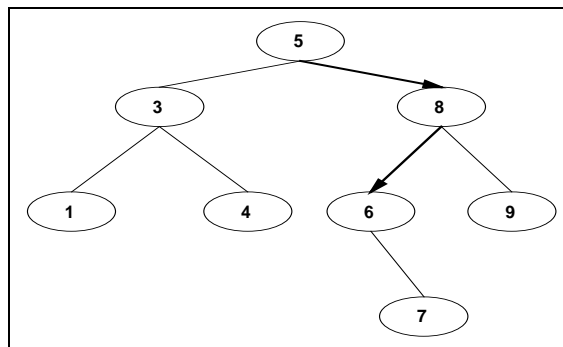


Figura 13.4: Um dicionário binário

Obviamente o objetivo *pertence(X, nil)* irá falhar para qualquer valor de X. Vamos investigar agora o comportamento do predicado *pertence/2*. Considerando a árvore apresentada na Figura 13.3, temos:

```
?-pertence(X, T).
X=a; X=b; X=c; X=d;
não
```

onde os valores de X são obtidos por backtracking. Sob o ponto de vista da eficiência, entretanto, o procedimento *pertence/2* é tão ineficiente quanto o emprego do predicado *membro/2*. Um aumento considerável de eficiência poderá entretanto ser obtido se houver uma relação de ordem entre os elementos do conjunto. Então os dados na árvore podem ser ordenados da esquerda para a direita de acordo com essa relação de ordem. Dizemos que uma árvore não-vazia *t(E, R, D)* está ordenada da esquerda para a direita se:

- (1) Todos os nodos na sub-árvore E são menores do que X,
- (2) Todos os nodos na sub-árvore D são maiores do que X, e

(3) Ambas as sub-árvores estão também ordenadas.

Tal árvore binária é denominada um "dicionário binário". Um exemplo é apresentado na Figura 13.4.

13.3 DICIONÁRIOS BINÁRIOS

A vantagem da ordenação é que, para procurar um objeto em um dicionário binário é suficiente pesquisar no máximo uma sub-árvore. A chave dessa economia, na busca por um elemento X é que podemos comparar X e a raiz, imediatamente descartando pelo menos uma sub-árvore. Por exemplo, a pesquisa pelo elemento 6 na Figura 13.4 está indicada em **negrito** e corresponde ao seguinte:

Começa-se na raiz, 5;
Compara-se 6 com 5, estabelecendo que $6 > 5$;
A pesquisa continua na sub-árvore direita;
Compara-se 6 com 8, estabelecendo que $6 < 8$;
A pesquisa continua na sub-árvore esquerda;
Compara-se 6 com 6, estabelecendo que $6 = 6$;
A pesquisa é encerrada com sucesso.

O método de pesquisa em um dicionário binário é, portanto:

Para encontrar um item X em um dicionário binário D:

- Se X é a raiz de D, então X já foi encontrado, senão
- Se X é menor do que a raiz de D, então X deve ser procurado na sub-árvore esquerda de D, senão
- Procurar X na sub-árvore direita de D, e
- Se D estiver vazio a pesquisa falha.

Essas regras são programadas em Prolog como o procedimento `pertence/2`, mostrado abaixo na Figura 13.5, onde a relação `mq(X, Y)` significa que X é maior do que Y. Se os itens armazenados na árvore são numéricos, então a relação é simplesmente $X > Y$.

```
pertence(X, t(_,X,_)).
pertence(X, t(E, R, _)) :-
    mq(R, X), pertence(X, E).
pertence(X, t(_, R, D)) :-
    mq(X, R), pertence(X, D).
```

Figura 13.5: Encontrando um item X em um Dicionário Binário

O procedimento `pertence/2` pode também ser empregado para construir um dicionário binário. Por exemplo, a consulta abaixo irá construir um dicionário binário D que contém os elementos 5, 3 e 8:

```
?-pertence(5, D), pertence(3, D), pertence(8, D).
D=t(t(D1, 3, D2), 5, t(D3, 8, D4))
```

As variáveis D1, D2, D3 e D4 são sub-árvores não especificadas, que podem conter qualquer coisa. O dicionário que será construído irá depender da ordem dos objetivos na consulta.

Um comentário sobre a eficiência da pesquisa em dicionários binários é interessante neste ponto. Em geral a busca por um item em um dicionário binário é bem mais eficiente do que em uma lista. Tal eficiência é devida ao seguinte: Vamos supor que n seja o número de itens em nosso conjunto de dados. Se o conjunto é representado por uma lista, então o tempo esperado de pesquisa é proporcional ao tamanho n da lista. Em média iremos pesquisar a lista até a metade para encontrar um determinado item, se os valores tiverem uma distribuição normal. Agora, se o conjunto for representado por um dicionário binário, o tempo de procura será proporcional à "altura" da árvore, representada pelo maior caminho entre a raiz e uma folha da árvore, dependendo portanto de sua conformação.

Diz-se que uma árvore binária é (aproximadamente) balanceada se, para cada nodo da árvore, as duas

sub-árvores são (aproximadamente) do mesmo tamanho, isto é, acomodam o mesmo número de itens. Se um dicionário de n nodos é balanceado de maneira ótima, então sua altura é proporcional a $\log(n)$. Pode-se dizer então que uma árvore balanceada possui complexidade logarítmica. A diferença entre n e $\log(n)$ é o ganho de eficiência que um dicionário binário possui sobre uma lista. Isso vale entretanto somente quando a árvore for aproximadamente balanceada. Se a árvore se afasta de uma conformação balanceada, então o seu desempenho irá degradar. Em casos extremos, de árvores completamente desbalanceadas, uma árvore fica reduzida a uma lista, tanto em conformação quanto em desempenho.

13.4 INSERÇÃO E REMOÇÃO DE ITENS EM DICIONÁRIOS BINÁRIOS

Na manutenção de um conjunto dinâmico de dados, pode-se desejar inserir novos dados ou remover dados desatualizados do conjunto. Assim, um repertório comum de operações sobre um conjunto S de dados é dado na tabela abaixo:

RELAÇÃO	SIGNIFICADO
pertence(X, S)	X pertence a S
inserir($S, X, S1$)	Inserir X em S produzindo $S1$
remover($S, X, S1$)	Remover X de S produzindo $S1$

A relação `pertence/2` foi definida na seção anterior. Definiremos agora a relação `insere/3`. É mais fácil inserir novos dados no nível mais "alto" de uma árvore, de modo que um novo item se torna uma "folha" da árvore em uma posição tal que a ordenação da árvore seja preservada. Representaremos esse tipo de inserção por:

`insFolha(D, X, D1)`

cujas regras são as seguintes:

- O resultado da inserção de X a uma árvore vazia é a árvore `t(nil,X,nil)`;
- Se X é a raiz de D , $D1=D$ (itens duplicados não são inseridos);
- Se a raiz de D é maior do que X , então X deve ser inserido na sub-árvore esquerda de D . Caso contrário X deve ser inserido na sub-árvore direita de D .

Na Figura 13.6, as árvores correspondem a seguinte sequência de inserções:

`insFolha(D1, 6, D2),`
`insFolha(D2, 7, D3),`
`insFolha(D3, 4, D4).`

para a relação `insFolha/3`, definida pelo procedimento abaixo:

```
insFolha(nil, X, t(nil, X, nil)).
insFolha(t(E, X, D), X, t(E, X, D)).
insFolha(t(E, R, D), X, t(E1, R, D)) :-
    mq(R, X), insFolha(E, X, E1).

insFolha(t(E, R, D), X, t(E, R, D1)) :-
    mq(X, R), insFolha(D, X, D1).
```

Vamos agora considerar a operação `remover/3`. É fácil remover uma folha, mas a remoção de um nodo é mais complicada. A remoção de uma folha pode, na verdade, ser definida como o inverso da inserção, isto é:

`remFolha(D1, X, D2) :-`
`insFolha(D2, X, D1).`

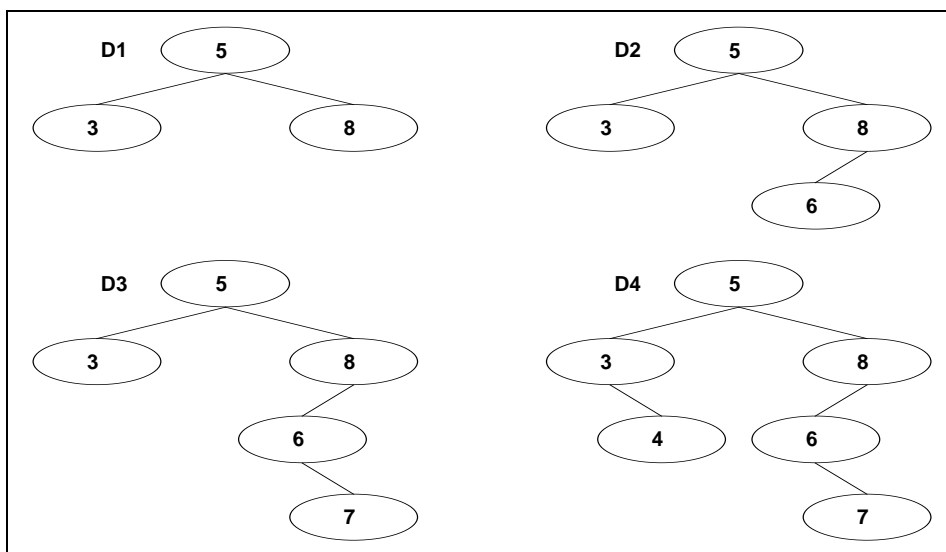


Figura 13.6: Inserção ao nível de folha em um dicionário binário

Entretanto, se X é um nodo interno, isso não vai funcionar, devido ao problema ilustrado na Figura 13.7: X tem duas sub-árvores, E e D. Após a remoção de X ficamos com uma lacuna na árvore e E e D ficam desconectadas do restante dela, sem possibilidade de se conectarem ao nodo pai de X, A, uma vez que este pode somente acomodar uma delas.

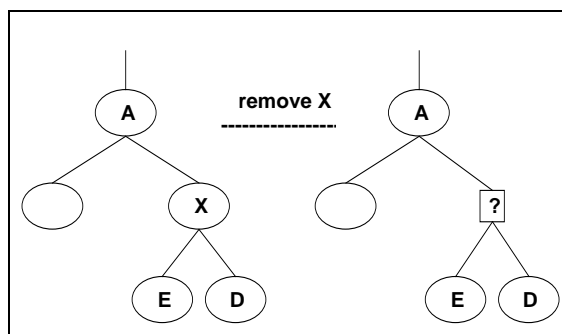


Figura 13.7: O problema de remover um nodo interior em um dicionário binário

Se uma das duas sub-árvores, E ou D, estiver vazia, então a solução é simples: A sub-árvore não-vazia é conectada a A. Se ambas forem não-vazias, então uma idéia é a seguinte: O nodo mais à esquerda de D, digamos Y, é removido de sua posição e conduzido a ocupar a lacuna deixada por X. Após esta transferência, a árvore resultante continuará ordenada. Naturalmente a mesma idéia funciona simetricamente, com a transferência do nodo mais à direita de E. De acordo com essas considerações, a operação de remover um item de um dicionário binário pode ser programada conforme é mostrado na Figura 13.8. A transferência do nodo mais à esquerda da sub-árvore direita é realizada pela relação:

$\text{trans}(T, Y, T1)$

onde Y é o nodo mais à esquerda de T e T1 é T após remover Y.

Há ainda uma outra solução, mais elegante, para as relações de inserção e remoção de nodos. Uma relação $\text{insere}/3$ pode ser definida, de forma não-determinística, de maneira que um novo item seja inserido em qualquer nível da árvore e não apenas como um nodo folha. As regras correspondentes são:

Para inserir um nodo X em um dicionário binário D:

- Inserir X como raiz de D, ou

```

remove(t(nil, X, D), X, D).
remove(t(E, X, nil), X, E).
remove(t(E, X, D), X, t(E, Y, D1)) :-
    trans(D, Y, D1).
remove(t(E, R, D), X, t(E1, R, D)) :-
    mq(R, X), remove(E, X, E1).
remove(t(E, R, D), X, t(E, R, D1)) :-
    mq(X, R), remove(D, X, D1).

trans(t(nil, Y, D), Y, D).
trans(t(E, R, D), Y, t(E1, R, D)) :-
    trans(E, Y, E1).

```

Figura 13.8: Removendo um nodo interior em um dicionário binário

- Se a raiz de D é maior do que X, então inserir X na sub-árvore esquerda de D. Caso contrário inserir X na sub-árvore direita de D.

A dificuldade aqui é a inserção de X como raiz de D. Vamos formular essa operação como a relação:

`insRaiz(D, X, D1)`

onde X é o item a ser inserido como raiz de D e D1 é o dicionário resultante, com X como raiz. A figura 13.9 ilustra as relações entre X, D e D1.

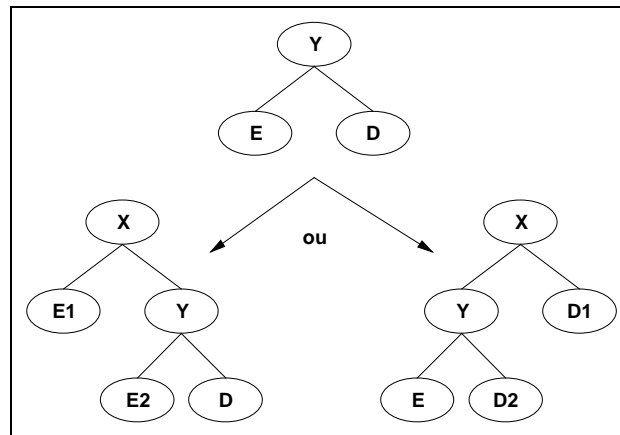


Figura 13.9: Inserção de um item X como raiz de um dicionário binário

A questão agora é: O que são as sub-árvores E1 e E2 na figura 13.9? (ou D1 e D2, alternativamente). A resposta deriva das seguintes restrições:

- E1 e E2 devem ser, necessariamente, dicionários binários;
- O conjunto de nodos em E1 e E2 é igual ao conjunto de nodos em E;
- Todos os nodos em E1 são menores do que X e todos os nodos em E2 são maiores do que X.

A relação que impõe todas essas restrições é exatamente a relação procurada, `insRaiz/3`. Assim, se X foi inserido em E como raiz, então as sub-árvores resultantes são E1 e E2 que, em Prolog, devem satisfazer a:

`insRaiz(E, X, t(E1, X, E2))`

assim como se X for inserido em D, D1 e D2 devem respeitar:

`insRaiz(D, X, t(D1, X, D2))`

A Figura 13.10 apresenta o programa completo para a inserção não-determinística em um dicionário binário. A característica principal de tal programa é que não há restrição quanto ao nível de inserção. Assim, `insere/3` pode ser empregada na direção inversa para a remoção de um item do dicionário.


```

insere(DB, X, DB1) :-
    insRaiz(DB, X, DB1).

insRaiz(nil, X, t(nil, X, nil)).
insRaiz(t(E, X, D), X, t(E, X, D)).
insRaiz(t(E, Y, D), X, t(E1, X, t(E2, Y, D))) :-
    mq(Y, X), insRaiz(E, X, t(E1, X, E2)).
insRaiz(t(E, Y, D), X, t(t(E, Y, D1), X, D2)) :-
    mq(X, Y), insRaiz(D, X, t(D1, X, D2)).

```

Figura 13.10: Inserção não-determinística em um dicionário binário

13.5 APRESENTAÇÃO DE ÁRVORES

Como todos os objetos em Prolog, uma árvore binária pode ser apresentada por meio do predicado embutido `write/1`. Entretanto o objetivo `write(T)` irá apresentar toda a informação contida em `T`, sem indicar graficamente a real estrutura de uma árvore. Pode ser bastante cansativo imaginar a estrutura de uma árvore a partir do termo Prolog que a representa. Assim muitas vezes é desejável se dispor de um procedimento que permita a representação gráfica de sua estrutura.

Há um método relativamente simples de apresentar graficamente a estrutura de árvores binárias. O truque é apresentar a árvore da esquerda para a direita, e não da raiz para as folhas, como são usualmente representadas. Vamos definir um procedimento, `apresenta(T)` para apresentar desse modo a estrutura de uma árvore `T`. O princípio é o seguinte:

Para apresentar uma árvore não-vazia `T`:

- Apresentar a sub-árvore esquerda de `T`, indentada por alguma distância, `H`, para a direita,
- Apresentar a raiz de `T`, e
- Apresentar a sub-árvore direita de `T`, indentada por alguma distância, `H`, para a direita.

A distância de indentação, `H`, que pode ser adequadamente escolhida, é um parâmetro adicional para a indentação de árvores. Pela introdução de `H`, precisamos de um procedimento, `ap(T, H)`, para apresentar `T` indentada `H` espaços a partir da margem esquerda. A relação entre os procedimentos `apresenta/1` e `ap/2` é a seguinte:

```
apresenta(T) :- ap(T, H).
```

A Figura 13.11 mostra o procedimento `apresenta/1` codificado em Prolog. O princípio adotado para obter esse formato de saída pode ser facilmente adaptado para a apresentação dos mais diversos tipos de árvores.

```

apresenta(T) :- ap(T, 0).

ap(nil, _).
ap(t(E, R, D), H) :-
    H2 is H+2,
    ap(D, H2),
    tab(H), write(R), nl,
    ap(E, H2).

```

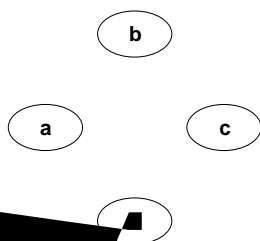
Figura 13.11: Apresentação de uma Árvore Binária

13.6 GRAFOS

13.6.1 REPRESENTAÇÃO

As estruturas em forma de grafos são empregadas em diversas aplicações, tais como a representação

de relações, situações e problemas. Um grafo é definido por um conjunto de nodos e um conjunto de arestas, onde cada aresta interliga um par de nodos. Quando as arestas são direcionadas, são também denominadas arcos. Os arcos são representados por meio de pares ordenados. Os grafos assim constituídos são denominados grafos direcionados. Aos arcos podem ser associados custos, nomes ou qualquer tipo de rótulo, dependendo da aplicação. Na figura 13.12 são apresentados exemplos de grafos.



13.6.2 CAMINHAMENTO EM GRAFOS

Seja G um grafo e A e Z dois nodos de G . Vamos definir uma relação

$\text{caminho}(A, Z, G, C)$

onde C é um caminho acíclico entre A e Z em G . O caminho C é representado por uma lista de nodos. Se G é o grafo representado na Figura 13.12(a), então podemos escrever, por exemplo:

$\text{caminho}(a, d, G, [a, b, d])$

$\text{caminho}(a, d, G, [a, b, c, d])$

Uma vez que o caminho não deve conter nenhum ciclo, cada nodo pode aparecer na lista no máximo uma vez. Um método para se encontrar um caminho entre dois nodos em um grafo é o seguinte:

Para encontrar um caminho acíclico C entre os nodos A e Z de um grafo G :

- Se $A = Z$, então $C = [A]$, senão
- Encontrar um caminho acíclico C_1 , de algum nodo Y até o nodo Z e encontrar um caminho de A até Y , evitando os nodos em C_1 .

Essa formulação implica em outra relação: Encontre um caminho sob a restrição de evitar um determinado conjunto de nodos. Assim, definiremos um segundo procedimento:

$\text{caminho1}(A, C_1, G, C)$

cuja relação com o procedimento $\text{caminho}/4$ é mostrada na Figura 13.13 a seguir:

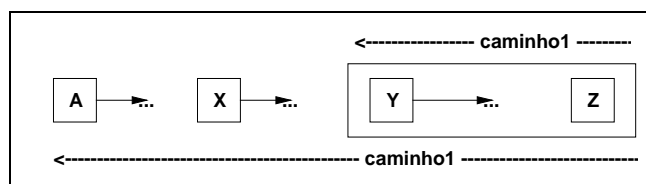


Figura 13.13: Relação entre os procedimentos $\text{caminho}/4$ e $\text{caminho1}/4$

Como ilustrado na Figura 13.13, os argumentos do procedimento $\text{caminho1}/4$ são:

- A , que é um nodo,
- G , que é um grafo,
- C_1 , que é um caminho acíclico em G , e
- C , que é um caminho acíclico em G , de A até o início de C_1 e continuando ao longo de C_1 até o seu final.

A relação entre $\text{caminho}/4$ e $\text{caminho1}/4$ é dada por:

$\text{caminho}(A, Z, G, C) \equiv \text{caminho}(A, [Z], G, C).$

A Figura 13.13 sugere uma definição recursiva de $\text{caminho1}/4$. O caso básico surge quando o nodo inicial de C_1 (Y , na figura) coincide com o nodo inicial de C , que é A . Se isso não ocorrer, então deve haver um nodo X tal que:

- (1) Y é adjacente a X ,
- (2) X não está em C_1 , e
- (3) C satisfaz a $\text{caminho1}(A, [X | C_1], G, C)$

O programa completo é apresentado na Figura 13.14, abaixo, onde $\text{membro}/2$ é a relação de ocorrência para listas e a relação $\text{adjacente}(X, Y, G)$ significa que há um arco conectando os nodos X e Y no grafo G .

$\text{caminho}(A, Z, G, C) :-$

```

caminhol(A, [Z], G, C).

caminhol(A, [A | C1], _, [A | C1]).
caminhol(A, [Y | C1], G, C) :-
    adjacente(X, Y, G),
    not membro(X, C1),
    caminhol(A, [X, Y | C1], G, C).

adjacente(X, Y, grafo(Nodos, Arestas)) :-
    membro(ar(X, Y), Arestas);
    membro(ar(Y, X), Arestas).

```

Figura 13.14: Encontrando caminhos acíclicos entre os nodos A e Z no grafo G

Um problema clássico sobre estruturas em grafo é encontrar um caminho "hamiltoniano", isto é, um caminho acíclico que percorra todos os nodos do grafo. Usando o procedimento caminho/4, anteriormente definido, isso pode ser realizado da maneira apresentada abaixo, onde nodo(N, Grafo) significa que N é um nodo do grafo Grafo.

```

hamiltoniano(Grafo, Caminho) :-
    caminho( _, _, Grafo, Caminho),
    cobre(Caminho, Grafo).

cobre(Caminho, Grafo) :-
    not(nodo(N, Grafo), not membro(N, Caminho)).

```

Pode-se associar custos aos caminhos em um grafo. O custo total de um caminho é a soma dos custos associados aos arcos que formam o caminho. Se não há custos associados aos arcos, então pode-se falar sobre a "extensão" do caminho, contando uma unidade para cada um dos arcos que o constituem. As relações caminho/4 e caminhol/4 podem ser modificadas de modo a manipular os custos, por meio da introdução de um argumento adicional para cada caminho:

```
caminho(A, Z, G, C, Custo)
```

e

```
caminhol(A, C1, Custo1, G, C, Custo)
```

onde Custo é o custo do caminho C e Custo1 é o custo do caminho C1. A relação adjacente/5 é também resultado da adição de um argumento extra - o custo de um arco - à relação original adjacente/4.

A Figura 13.15 mostra um programa que computa caminhos e os seus custos, podendo ser utilizado para encontrar um "caminho de custo mínimo" entre dois nodos de um grafo. Isso é obtido por meio dos objetivos:

```

?-caminho(n1, n2, G, CaminhoMin, CustoMin),
not(caminho(n1, n2, G, _, Custo), Custo < CustoMin).

```

```

caminho(A, Z, G, C, Custo) :-
    caminhol(A, [Z], 0, G, C, Custo).

caminhol(A, [A | C1], Custo1, G, [A | C1], Custo1).
caminhol(A, [Y | C1], Custo1, G, C, Custo) :-
    adjacente(X, Y, CustoXY, G),
    not membro(X, C1),
    Custo2 is Custo1+CustoXY,
    caminhol(A, [X, Y | C1], Custo2, G, C, Custo).

```

Figura 13.15: C é um caminho acíclico de A a Z em G cujo custo é Custo

De modo semelhante também é possível encontrar um "caminho de custo máximo" entre qualquer par de nodos em um grafo G através da conjunção de objetivos abaixo:

```

?-caminho( _, _, G, CaminhoMax, CustoMax),
not(caminho( _, _, G, _, Custo), Custo > CustoMax).

```

Deve-se ressaltar que esse método de encontrar caminhos de custo mínimo ou máximo é extremamente ineficiente, uma vez que investiga todos os caminhos possíveis de forma completamente não seletiva, sendo totalmente inadequado para grandes grafos, devido à sua elevada complexidade temporal.

13.6.3 ÁRVORES GERADORAS

Como já foi comentado, um grafo é dito ser conexo se há um caminho entre quaisquer dois nodos que dele fazem parte. Seja $G = (N, A)$ um grafo conexo com um conjunto de nodos N e um conjunto de arestas A . Uma "árvore geradora" de G é um grafo conexo $T = (N, A')$, onde A' é um subconjunto de A tal que:

- (1) T é conexo, e
- (2) Não há ciclos em T .

Essas duas condições garantem que T é uma árvore. Para o grafo apresentado na Figura 13.12(a), por exemplo, há três árvores geradoras que correspondem às seguintes três listas de arestas:

$T_1 = [a-b, b-c, c-d]$
 $T_2 = [a-b, b-d, d-c]$
 $T_3 = [a-b, b-d, b-c]$

onde cada termo da forma $X-Y$ denota uma aresta entre os nodos X e Y . Pode-se escolher qualquer nodo na lista para raiz da árvore. As árvores geradoras são de interesse, por exemplo, em problemas de comunicação, porque fornecem, com o menor número de linhas de comunicação possível, um caminho entre qualquer par de nodos. Definiremos um procedimento:

$\text{arvG}(T, G)$

onde T é uma árvore geradora de G . Assumiremos para isso que G é um grafo conexo. Podemos imaginar a construção algorítmica de uma árvore geradora da seguinte maneira: Iniciamos com um conjunto vazio de arestas, ao qual gradualmente vamos adicionando arestas de G , tomando cuidado para que nunca seja gerado um ciclo, até que mais nenhuma aresta de G possa ser adicionada ao conjunto, porque isso determinaria a geração de um ciclo. O conjunto de arestas resultante define uma das árvores geradoras de G . A condição de não-ciclo pode ser mantida por meio de uma regra simples: Uma aresta pode ser adicionada ao conjunto somente se um de seus nodos já pertence à árvore geradora em formação e o outro ainda não pertence. Um programa que implementa essa idéia é mostrado na Figura 13.11. A relação fundamental ali é $\text{desenvolve}(T_1, T, G)$, onde todos os três argumentos são conjuntos de arestas. G é um grafo conexo. T_1 e T são subconjuntos de G tais que ambos representam árvores. T é uma árvore geradora de G , obtida pela adição de zero ou mais arestas de G a T_1 . Pode-se dizer que T_1 origina o desenvolvimento de T .

É interessante desenvolver também um programa para a construção de árvores geradoras de forma completamente declarativa, pelo simples estabelecimento de relações matemáticas. Assumiremos que tanto grafos conexos como árvores sejam representados por meio de listas de arestas como no programa da Figura 13.16. As definições necessárias são:

- (1) T é uma árvore geradora de G se:
 - T é um subconjunto de G ,
 - T é uma árvore, e
 - T "cobre" G , isto é, todo nodo de G está também em T .
- (2) Um conjunto de arestas T é uma árvore se:
 - T é conexo, e
 - T não possui ciclos.

Usando a relação caminho/4, definida na seção anterior, tais definições podem ser estabelecidas em Prolog conforme é mostrado na figura 13.17. Deve-se notar, entretanto, que o programa ali definido é de pequeno interesse prático devido a sua ineficiência.

```

arvG(G, T) :-
    membro(Aresta, G),
    desenvolve([Aresta], T, G).

desenvolve(T1, T, G) :-
    novaAresta(T1, T2, G),
    desenvolve(T2, T, G).
desenvolve(T, T, G) :-
    not novaAresta(T, _, G).

novaAresta(T, [A-B | T], G) :-
    adjacente(A, B, G),
    nodo(A, T),
    not nodo(B, T).

adjacente(A, _, G) :-
    membro(A-B, G); membro(B-A, G).

nodo(A, G) :-
    adjacente(A, _, G).

```

Figura 13.16: Um procedimento algorítmico para obter a árvore geradora T de um grafo G, assumido conexo

```

arvG(G, T) :-
    subconj(G, T),
    árvore(T),
    cobre(T, G).

árvore(T) :-
    conexo(T),
    not temCiclos(T).

conexo(T) :-
    not(nodo(A,T), nodo(B,T), not caminho(A,B,T, _)).

temCiclos(T) :-
    adjacente(A,B,T),
    caminho(A,B,T, [A, X, Y | _]).

cobre(T, G) :-
    not(nodo(A, G), not nodo(A, T)).

subconj([], []).
subconj([X | L], S) :-
    subconj(L, L1),
    (S = L1; S = [X | L1]).

```

Figura 13.17: Um procedimento declarativo para obter as árvores geradoras de G

RESUMO

- No presente capítulo estudou-se diferentes métodos de classificação de listas, tecendo considerações acerca da sua eficiência:
 - (1) bubblesort/2;
 - (2) insertsort/2;
 - (3) quicksort/2.
- Representação de conjuntos como árvores binárias e dicionários binários:
 - (1) Procura por um item em uma árvore;
 - (2) Inserção de itens;
 - (3) Remoção de itens;
 - (4) Balanceamento de árvores e sua relação com a eficiência;
 - (5) Apresentação de árvores.
- Grafos:

- (1) Representação de grafos;
- (2) Caminhamento em grafos;
- (3) Obtenção das árvores geradoras de um grafo.

EXERCÍCIOS

- 13.1 Escreva um programa para intercalar duas listas classificadas, produzindo uma terceira lista, também classificada. Por exemplo:

```
?-intercala([3,4,5], [1,2,2,5,7], L).
L = [1,2,2,3,4,5,5,7]
```

- 13.2 Escreva um programa para descrever a relação quicksort/2, empregando pares-diferença na representação de listas.
- 13.3 O programa quicksort/2, apresentado neste capítulo, possui um desempenho sofrível quanta a lista a ser classificada já está classificada ou quase classificada. Analise porque isso ocorre e proponha modificações no algoritmo capazes de solucionar tal problema.
- 13.4 Um outro algoritmo de classificação de listas baseia-se na seguinte proposta: Para classificar uma lista L:

- (1) Divida L em duas listas, L1 e L2, com aproximadamente o mesmo tamanho,
- (2) Classifique L1 e L2, obtendo S1 e S2, e
- (3) Intercale S1 e S2, obtendo a lista L classificada.

Implemente este princípio de classificação e compare sua eficiência com a obtida no programa quicksort/2.

- 13.5 Defina os predicados:

`arvBin(Objeto)` e `dicBin(Objeto)`

para reconhecer, respectivamente se Objeto é uma árvore binária ou um dicionário binário.

- 13.6 Defina o procedimento

`altura(ÁrvoreBinária, Altura)`

para computar a altura de uma árvore binária. Assuma que a altura de uma árvore vazia é zero e que a de uma árvore com um único elemento é 1.

- 13.7 Defina a relação

`lineariza(Árvore, Lista)`

para representar uma árvore linearizada sob a forma de lista.

- 13.8 Considere as árvores geradoras de um grafo que possui custos associados às arestas. Seja o custo de uma árvore geradora definido como a soma dos custos de todas as arestas nela presentes. Escreva um programa para encontrar a árvore geradora de custo mínimo em um grafo.

14. ESTRATÉGIAS PARA A SOLUÇÃO DE PROBLEMAS

O presente capítulo introduz um esquema genérico para a solução de problemas denominado "espaço de estados". Um espaço de estados é um grafo cujos nodos correspondem a possíveis situações de um problema, de modo que sua solução é reduzida, em tal representação, à procura de um caminho sobre tal grafo. Estudaremos exemplos de formulação de problemas usando a abordagem do espaço de estados e discutiremos métodos gerais para a solução de problemas representados por meio desse formalismo. A solução de problemas envolve, portanto, a pesquisa em grafos e, tipicamente, a lidar com alternativas. As estratégias básicas apresentadas neste capítulo para a exploração de alternativas são a pesquisa em profundidade (depth-first search) e a pesquisa em amplitude (breadth-first search).

14.1 CONCEITOS BÁSICOS

Vamos considerar o exemplo apresentado na Figura 14.1. O problema é formular um planejamento para reorganizar uma pilha de blocos sobre uma mesa da maneira mostrada na figura, obedecendo as seguintes três regras:

- (1) Pode-se mover somente um bloco de cada vez;
- (2) Um bloco pode ser movido somente se não houver nada sobre ele;
- (3) Os blocos somente podem ser colocados diretamente na mesa ou sobre algum outro bloco.

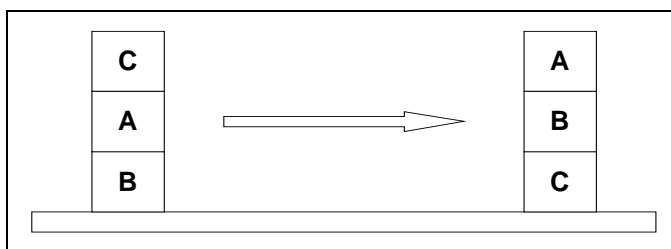


Figura 14.1 O problema da reorganização dos blocos A, B, C

Deve-se observar que o objetivo não é apenas obter a situação final desejada. O que se quer realmente é o "plano" que nos permite alcançá-la. Para isso é necessário descobrir uma seqüência de operações que permita realizar a transformação proposta.

Podemos pensar neste problema como um caso de exploração entre alternativas. Na situação inicial, existe apenas uma possibilidade: colocar o bloco C na mesa. Após fazer isto, surgem três alternativas possíveis:

- Colocar o bloco A na mesa, ou
- Colocar o bloco A sobre o bloco C, ou
- Colocar o bloco C sobre o bloco A.

Como o exemplo ilustra, dois conceitos devem ser considerados nesse tipo de problema:

- (1) As situações do problema, e
- (2) Os movimentos ou ações válidas que transformam uma situação em outra.

As situações e os movimentos possíveis formam um grafo direcionado, denominado "espaço de estados". Um espaço de estados para o problema exemplificado é apresentado na Figura 14.2. Os nodos do grafo correspondem a situações do problema e os arcos correspondem a transições legais entre os

estados. Encontrar um plano cuja execução solucione o problema original é equivalente a encontrar um caminho entre a situação inicial dada (o nodo inicial) e alguma situação final especificada, também denominada "o nodo objetivo".

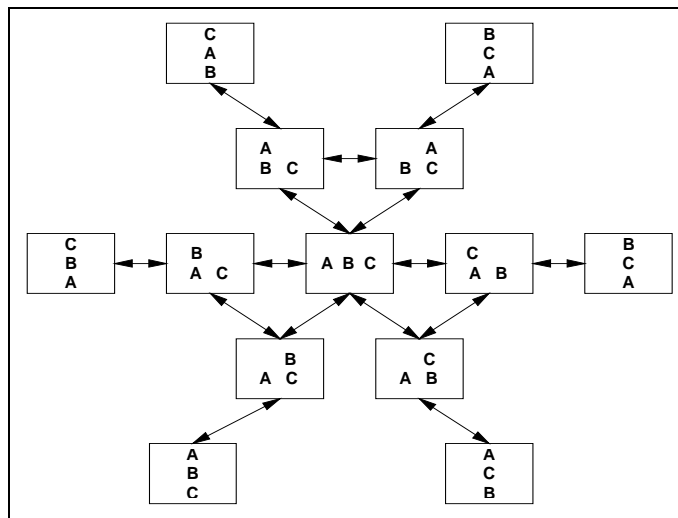


Figura 14.2: O espaço de estados do problema da reorganização de A, B e C

A Figura 14.3 apresenta um outro exemplo do mesmo tipo de problema: o "jogo do oito" e a sua representação reduzida ao problema de caminhamento em um grafo. O jogo do oito é um clássico da pesquisa em inteligência artificial e consiste em oito peças deslizantes, numeradas de 1 a 8 e dispostas em uma matriz 3x3 de nove casas. Uma das casas está sempre vazia e qualquer peça a ela adjacente pode ser movida para essa casa. Podemos imaginar que a casa vazia pode "mover-se", trocando de lugar com qualquer uma das peças adjacentes. A situação final é algum arranjo especial das peças, como pode ser visto na Figura 14.3.

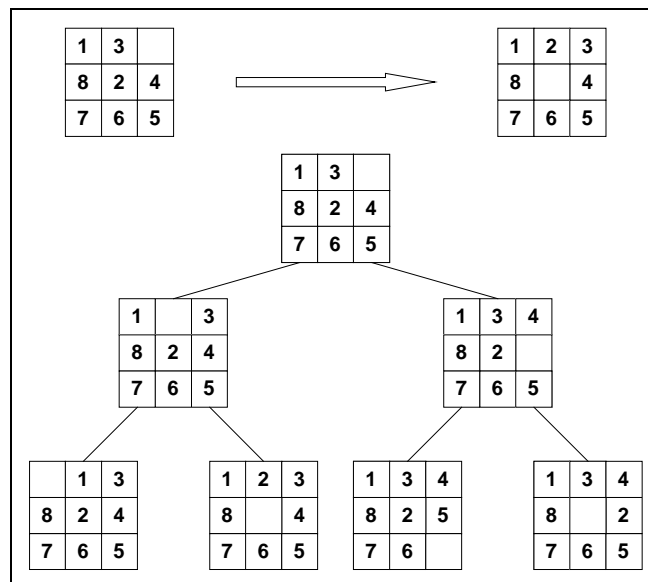


Figura 14.3: O "jogo do oito" em uma particular configuração

É fácil construir aplicações gráficas similares para outros quebra-cabeças populares que se enquadram no mesmo tipo de problema como, por exemplo, o problema das torres de Hanói, ou de como conduzir a raposa, o ganso e o milho através do rio. Neste último problema há um bote que somente pode conduzir o fazendeiro e algum outro objeto. O fazendeiro tem que proteger o ganso da raposa e o milho do ganso. Muitos problemas práticos também se encaixam nesse mesmo paradigma, dentre os quais

talvez o mais importante seja o "problema do caixeiro-viajante", que é modelo formal de diversas aplicações práticas. Este problema é definido por um mapa com n cidades interligadas por diversas rodovias. A idéia é encontrar a rota mais curta, partindo de alguma cidade inicial, visitando todas as demais cidades e retornando ao ponto de partida. Nenhuma cidade, com exceção da inicial, pode aparecer duas vezes no trajeto. O problema pode ser facilmente solucionado através de uma adaptação dos procedimentos de caminhamento em grafos estudados no capítulo 13.

Vamos resumir os conceitos introduzidos nestes exemplos. O espaço de estados de um dado problema especifica "as regras do jogo". Os nodos no espaço de estados correspondem a situações possíveis e os arcos correspondem a movimentos válidos ou "passos de solução". Um problema particular pode ser definido por:

- Um espaço de estados,
- Um nodo inicial, e
- Uma condição-objetivo, isto é, a situação a ser atingida. Denomina-se nodos-objetivos aos nodos que satisfazem essa condição.

Podemos associar custos às ações válidas de um espaço de estados. Por exemplos, custos associados à movimentação, no problema de organização de blocos, indicariam que alguns blocos são mais difíceis de mover do que outros. No problema do caixeiro-viajante, os movimentos correspondem a viagens diretas entre duas cidades. Ali os custos dos movimentos podem corresponder a distâncias entre as cidades envolvidas.

Nos casos em que temos custos associados aos movimentos, estaremos normalmente interessados em obter as soluções de menor custo possível. O custo total de uma solução é a soma de todos os custos associados aos que compõem o caminho entre o nodo inicial e o nodo objetivo. Mesmo que não haja custos, iremos sempre nos deparar com um problema de otimização: qual o caminho mais curto entre esses dois pontos?

Antes de apresentar alguns programas que implementam algoritmos clássicos para a pesquisa em espaços de estados, vamos estudar como um espaço de estados pode ser representado em Prolog. Representaremos um espaço de estados pela relação

$$s(X, Y)$$

que é verdadeira se há um movimento válido no espaço de estados de um nodo X a um nodo Y . Dizemos que o nodo Y é um *sucessor* de X . Se há custos associados aos movimentos, então adicionaremos um terceiro argumento, representando o *custo do movimento*:

$$s(X, Y, \text{Custo})$$

Essa relação pode ser representada explicitamente no programa por meio de um conjunto de fatos, entretanto, para espaços de estado de maior complexidade, isso se torna impraticável. Assim a relação $s/3$ é usualmente definida de maneira implícita, pelo estabelecimento de regras para computar os nodos sucessores de um determinado nodo.

Outra questão de importância geral é como representar as situações do problema, isto é, os nodos do espaço de estados. A representação deve ser compacta, mas por outro lado deve permitir uma execução eficiente das operações requeridas, particularmente a relação de sucessão, $s/3$.

Vamos considerar, por exemplo, o problema de organização de blocos apresentado na Figura 14.1. Estudaremos um caso mais geral, em que existe um número qualquer de blocos organizados em uma ou mais pilhas. O número de pilhas será limitado a um determinado máximo para tornar o problema mais interessante. Isso pode corresponder também a uma restrição realística, uma vez que a um robô que manipule blocos somente pode ser oferecido um espaço de trabalho limitado, sobre uma mesa.

Uma situação do problema pode ser representada por uma lista de pilhas. Cada pilha, por sua vez, será representada por uma lista de blocos, ordenada de forma que o bloco no topo da pilha é a cabeça da lista. pilhas vazias serão representadas por listas vazias. A situação inicial do problema apresentado na Figura 13.1 pode ser representada por:

```
[[c, a, b], [], []]
```

Uma situação objetivo é qualquer arranjo com uma pilha ordenada de todos os blocos. Há três situações objetivo possíveis:

```
[[a, b, c], [], []]
[[], [a, b, c], []]
[[], [], [a, b, c]]
```

A relação $s/3$ pode ser programada de acordo com a seguinte regra: Uma situação $s2$ é sucessora de alguma situação $s1$, se há duas pilhas, $P1$ e $P2$ em $s1$ e bloco no topo de $P1$ pode ser movido para $P2$, configurando a nova situação $s2$. Como todas as situações são representadas por listas de pilhas, isso pode ser escrito em Prolog da seguinte maneira:

```
s(Pilhas, [P1, [T1 | P2] | OutrasPilhas]) :-
    remove([T1 | P1], Pilhas, Pilhas1),
    remove(P2, Pilhas1, OutrasPilhas).

remove(X, [X | L], L).
remove(X, [Y | L], [Y | L1]) :-
    remove(X, L, L1).
```

A condição objetivo para o problema dado é:

```
objetivo(Situação) :-
    membro([a, b, c], Situação).
```

Os algoritmos de pesquisa em espaços de estados serão solucionados por meio da relação:

```
resolve(Início, Solução)
```

onde Início é o nodo inicial do espaço de estados e Solução é um caminho entre Início e qualquer nodo objetivo. Para o problema da organização de blocos proposto, a consulta correspondente seria:

```
?-resolve([[c, a, b], [], []], Solução).
```

Como resultado de uma pesquisa bem sucedida, a variável Solução será instanciada para uma lista de arranjos de blocos representando um "plano" para transformar o estado inicial em um estado onde os três blocos estejam em uma pilha organizada segundo a lista [a, b, c].

14.2 PESQUISA EM PROFUNDIDADE

Dada uma formulação do espaço de estados de um problema, há diversas abordagens para encontrar o caminho da solução. Duas estratégias básicas são: a pesquisa em profundidade (depth-first search) e a pesquisa em amplitude (breadth-first search). na presente seção será estudada a pesquisa em profundidade, que pode ser formulada a partir de uma idéia bastante simples:

Para encontrar uma linha de solução, Sol, a partir de um determinado nodo, N, até algum nodo objetivo:

- Se N é um nodo objetivo, então Sol = [N], senão
- Se há um nodo sucessor de N, N1, tal que existe um caminho, Sol1, de N1 até algum nodo objetivo, então Sol = [N | Sol1].

Pode-se representar essa formulação em Prolog por meio da seguinte relação $resolve/2$:

```
resolve(N, [N]) :-
    objetivo(N).
resolve(N, [N | Sol1]) :-
    s(N, N1), resolve(N1, Sol1).
```

Esse programa é, na verdade, uma implementação da estratégia de pesquisa em profundidade. O método é denominado "em profundidade" devido à ordem em que as alternativas são exploradas no espaço de estados. Sempre que o algoritmo de pesquisa em profundidade tem oportunidade de continuar sua pesquisa escolhendo entre diversos nodos alternativos, a decisão conduz ao nó que se encontra em maior profundidade, isto é, ao mais distante possível do nodo inicial. A Figura 14.4 ilustra a ordem na qual os nodos são visitados, que corresponde à ordem seguida pelo Prolog na solução da consulta:

`?-resolve(a, Sol).`

A figura 14.4 representa um espaço de estados onde "a" é o nodo inicial e "f" e "g" são nodos objetivos. A ordem na qual a pesquisa é realizada é dada pelo número entre parênteses à esquerda de cada nodo. A pesquisa em profundidade é a mais adequada ao estilo recursivo da linguagem Prolog, uma vez que esta, na execução de seus objetivos, explora as alternativas segundo esse mesmo princípio. Essa técnica é simples, fácil de programar, e funciona bem na maioria dos casos. Entretanto, há várias situações em que o procedimento `resolve/2`, que adota o método de pesquisa em profundidade, pode se mostrar ineficiente. Se isso vai acontecer ou não, depende do espaço de estados. Para complicar o procedimento `resolve/2`, é suficiente uma leve modificação no problema apresentado na Figura 14.4: adicionar um arco do nodo h ao nodo d, originando um ciclo, como é mostrado na Figura 14.5. Nesse caso a pesquisa em profundidade irá ocorrer da seguinte maneira: inicia em a e desce até h, seguindo o ramo mais à esquerda no grafo. Neste ponto, ao contrário do que ocorre na Figura 14.4, h tem um sucessor, que é d. Por sua vez, d tem h como sucessor, resultando em um ciclo infinito:

a, b, d, h, d, h, d, h, ...

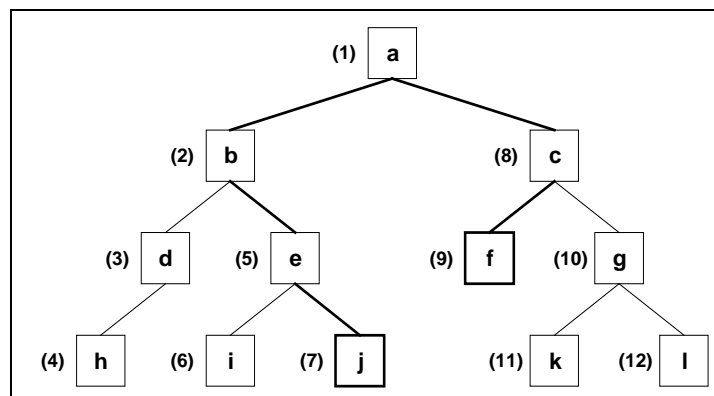


Figura 14.4: Pesquisa em Profundidade

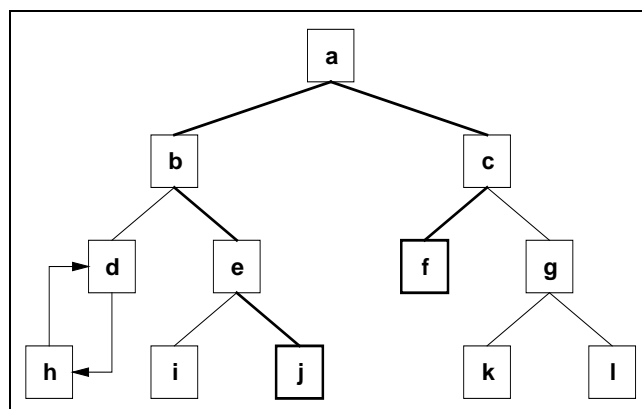


Figura 14.5: Um espaço de estados originando um caminho cíclico

Um aperfeiçoamento óbvio, portanto, em nosso programa de pesquisa em profundidade é acrescentar um mecanismo detector de ciclos. Assim, qualquer nodo que já houver sido visitado não deve ser considerado novamente. Tal idéia pode ser formulada por meio da relação:

`profundidade(Caminho, Nodo, Solução)`

Na relação `profundidade/3`, `Nodo` é o estado a partir do qual o nodo objetivo deve ser encontrado. `Caminho` é um caminho (uma lista de nodos) entre o nodo inicial e `Nodo`, enquanto que `Solução` é uma extensão de `Caminho`, passando por `Nodo`, até atingir um nodo objetivo. Essas idéias são apresentadas na Figura 14.6.

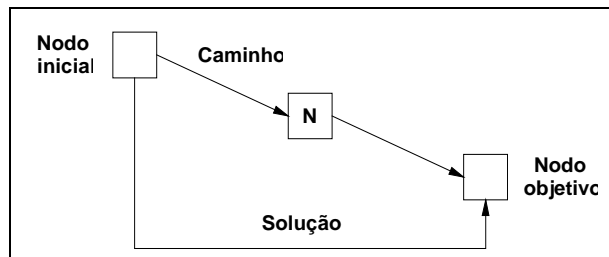


Figura 14.6: A relação `profundidade(Caminho, N, Solução)`

Para garantir uma programação simplificada, os caminhos serão representados em nossos programas como listas em ordem inversa, isto é, iniciando em um nodo objetivo (ou corrente, durante a execução) e terminando no nodo inicial. O argumento "Caminho" pode ser utilizado para dois propósitos:

- (1) Garantir que o algoritmo não irá considerar os sucessores de `Nodo` que já foram visitados (detecção de ciclos), e
- (2) Construir um caminho, `Solução`, que soluciona o problema.

Na Figura 14.7 apresentamos um programa que executa a pesquisa em profundidade em grafos com a detecção de ciclos, conforme foi anteriormente comentado. Vamos considerar agora uma variação desse programa. Dois argumentos que ali aparecem, `N` e `Caminho`, podem ser combinados em uma lista `[N | Caminho]`, assim, ao invés de se ter um "nodo candidato", `N`, para ser adicionado a um caminho que conduza ao objetivo desejado, temos um "caminho candidato", `C = [N | Caminho]`, para ser ampliado até alcançar o objetivo. A programação do predicado correspondente, `profundidade1(C, Solução)` é deixada como um exercício.

```

resolve(N, Solução) :-
    profundidade([], N, Solução).

profundidade(Caminho, N, [N | Caminho]) :-
    objetivo(N).
profundidade(Caminho, N, Solução) :-
    s(N, N1),
    not membro(N1, Caminho),
    profundidade([N | Caminho], N, Solução).

```

Figura 14.7: Pesquisa em profundidade com detecção de ciclos

Com o mecanismo de detecção de ciclos, o procedimento de pesquisa em profundidade vai encontrar o caminho apropriado para atingir uma solução em espaços de estados tais como o apresentado na Figura 14.5. Há entretanto espaços de estados para os quais esse procedimento não funcionará adequadamente. Muitos espaços de estado são infinitos. Em tais espaços o algoritmo de pesquisa em profundidade pode se desviar do caminho correto para atingir um determinado objetivo, explorando indefinidamente uma ramificação infinita que jamais se aproximará do objetivo formulado. Para evitar a pesquisa em profundidade em ramificações infinitas (não- cíclicas, neste caso) do espaço de estados, adicionaremos mais um refinamento em nosso procedimento básico de pesquisa em profundidade: limitamos a profundidade máxima de pesquisa, obtendo uma nova relação, `profundidade2/3`, representada por:

`profundidade2(Nodo, Solução, ProfMáxima)`

onde a pesquisa não é permitida além de ProfMáxima. Essa restrição pode ser programada decrementando o valor estabelecido para ProfMáxima a cada chamada recursiva, não permitindo que esse limite se torne negativo. O programa resultante é mostrado na Figura 14.8.

```
profundidade2(Nodo, [Nodo], _) :-
    objetivo(Nodo).
profundidade2(Nodo, [Nodo | Sol], ProfMáxima) :-
    ProfMáxima > 0,
    s(Nodo, Nodol),
    Max1 is ProfMáxima -1,
    profundidade2(Nodol, Sol, Max1).
```

Figura 14.8: Um programa para pesquisa em profundidade limitada

14.3 PESQUISA EM AMPLITUDE

Em contraste com a pesquisa em profundidade, a pesquisa em amplitude escolhe visitar primeiro os nodos que estão mais próximos do nodo inicial. Isso resulta em um processo de busca que tende a se desenvolver mais em amplitude do que em profundidade, como pode ser visto na Figura 14.9. O espaço de estados ali apresentado é basicamente o mesmo da Figura 14.4, entretanto, a ordem em que os nodos serão visitados, dada pelo número entre parênteses à esquerda de cada nodo, é agora diferente.

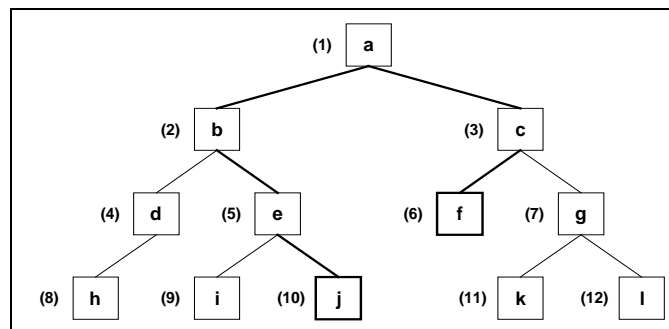


Figura 14.9: Pesquisa em amplitude

A estratégia de pesquisa em amplitude não é tão fácil de programar quanto a de pesquisa em profundidade. A razão dessa dificuldade é que temos de manter um conjunto de nodos candidatos alternativos, e não apenas um nodo como na pesquisa em profundidade. Entretanto, mesmo esse conjunto de nodos não é suficiente se desejarmos extrair um caminho-solução por meio desse processo. Assim, ao invés de manter um conjunto de nodos candidatos, iremos manter um conjunto de caminhos candidatos. Isso é representado pela relação:

`amplitude(Caminhos, Solução)`

que é verdadeira quando algum caminho pertencente ao conjunto de candidatos Caminhos pode ser estendido até algum nodo objetivo. O argumento solução representa tal caminho estendido.

14.3.1 REPRESENTAÇÃO DO CONJUNTO DE CAMINHOS CANDIDATOS

Vamos adotar inicialmente a seguinte representação para o conjunto de caminhos candidatos: O conjunto será representado como uma lista de caminhos, onde cada caminho é uma lista de nodos em ordem inversa, isto é, a cabeça da lista será o nodo mais recentemente visitado e o último elemento da lista será o nodo inicial da pesquisa. O conjunto inicia como um conjunto unitário de caminhos candidatos:

`[[NodoInicial]]`

Um esquema para definir o processo de pesquisa em amplitude pode ser formulado da seguinte maneira:

ra:

Para executar a pesquisa em amplitude, dado um conjunto de caminhos candidatos:

- Se o primeiro caminho contém um nodo objetivo como cabeça da lista que o representa, então ele é uma solução para o problema, senão
- Remover o primeiro caminho da lista de caminhos candidatos e gerar o conjunto de todas as possíveis extensões de um só nodo a esse caminho, adicionando o conjunto das extensões geradas ao final da lista de caminhos candidatos. Após, voltar a executar a pesquisa em amplitude sobre esse conjunto atualizado.

Por exemplo, para o espaço de estados apresentado na Figura 14.9, onde f e j são nodos objetivo, o processo se desenvolve da seguinte maneira:

- (1) O conjunto de caminhos candidatos inicialmente contém apenas o nodo raiz:

[[a]]

- (2) Determinar o conjunto de extensões de um só nodo de [a]:

[[b,a], [c,a]]

- (3) Remover o primeiro caminho candidato, [b, a] do conjunto e determinar suas extensões de um só nodo:

[[d,b,a], [e,b,a]]

Acrescentar essa lista de extensões ao final do conjunto de caminhos candidatos:

[[c,a], [d,b,a], [e,b,a]]

- (4) Remover [c, a] e acrescentar suas extensões ao final do conjunto de caminhos candidatos produzindo:

[[d,b,a], [e,b,a], [f,c,a], [g,c,a]]

- (5) Remover [d, b, a] e acrescentar sua única extensão ao final do conjunto de caminhos candidatos:

[[e,b,a], [f,c,a], [g,c,a], [h,d,b,a]]

- (6) Executar a mesma operação para [e, b, a], obtendo:

[[f,c,a], [g,c,a], [h,d,b,a], [i,e,b,a], [j,e,b,a]]

Aqui o processo de busca encontra [f, c, a], que contém como cabeça o nodo objetivo f. Então esse caminho é apresentado como solução.

Um programa que executa esse processo é apresentado na Figura 14.10. Ali, todas as extensões aos conjuntos candidatos são geradas através do predicado pré-definido bagof/3. Um teste para prevenir a geração de ciclos é também incluído. Note que no caso em que nenhuma extensão é possível, o predicado bagof/3 falha, portanto é fornecida uma chamada alternativa ao procedimento amplitude/2. Os predicados membro/2 e conc/3 são respectivamente as relações de ocorrência de um item em uma lista e a concatenação de listas, ambas já estudadas.

O problema desse programa é a ineficiência da operação conc/3. Isso entretanto pode ser reparado se representarmos listas por meio de pares-diferença conforme foi apresentado no capítulo 12. O conjunto de caminhos candidatos seria então representado como um par de listas: Caminhos e Z, e escrito como

Caminhos-Z

Introduzindo essa representação no programa da Figura 14.10, este pode ser sistematicamente transformado no programa apresentado na Figura 14.11. A transformação (simples) é deixada ao leitor a título de exercício.

resolve(Início, Solução) :-

```

amplitude([ [Início] ], Solução).

amplitude([ [N | Caminho] | _ ], [N | Caminho]) :-
    objetivo(N).
amplitude([ [N | Caminho] | Caminhos ], Solução) :-
    bagof([M, N | Caminho],
        (s(M, N), not membro(M, [N | Caminho])),
        NovosCaminhos),
    conc(Caminhos, NovosCaminhos, Caminhos1), !,
    amplitude(Caminhos1, Solução);
    amplitude(Caminhos, Solução).

```

Figura 14.10: Uma implementação da pesquisa em amplitude

```

resolve(Início, Solução) :-
    amplitude1([ [Início] | Z ] - Z, Solução).

amplitude1([ [N | Caminho] | _ ] - _, [N | Caminho]) :-
    objetivo(N).
amplitude1([ [N | Caminho] | Caminhos ] - Z, Solução) :-
    bagof([M, N | Caminho],
        (s(M, N), not membro(M, [N | Caminho])),
        NovosCaminhos),
    conc(NovosCaminhos, ZZ, Z), !,
    amplitude1(Caminhos1 - ZZ, Solução);
    Caminhos \== Z,
    amplitude1(Caminhos - Z, Solução).

```

Figura 14.11: Uma implementação mais eficiente do programa da Figura 14.10

14.3.2 REPRESENTAÇÃO EM ÁRVORE DO CONJUNTO DE CAMINHOS CANDIDATOS

Vamos considerar agora outra modificação no programa de pesquisa em amplitude. Até então o conjunto de caminhos candidatos vinha sendo representado como uma lista de caminhos. Isso gera um consumo exagerado de memória, uma vez que a parte inicial é a mesma para diversos caminhos, sendo armazenada de forma redundante. A maneira mais eficiente de representar os caminhos candidatos é em forma de árvore, onde a parte comum a diversos caminhos é armazenada sem redundância, nos ramos superiores da árvore. Adotaremos a seguinte representação de árvore. É necessário considerar dois casos:

- (1) A árvore consiste em um único nodo N. Então ela será representada pelo termo $f(N)$, onde o functor f indica que N é uma folha da árvore;
- (2) A árvore consiste em um nodo raiz, N , e um conjunto de sub-árvores. Tal árvore é dada pelo termo $t(N, Subs)$, onde $Subs = [S1, S2, \dots]$ é uma lista de sub-árvores.

Por exemplo, vamos considerar uma situação onde os três primeiros níveis da árvore apresentada na Figura 14.9 foram gerados. O conjunto de caminhos candidatos nesse momento é o seguinte:

```
[ [d, b, a], [e, b, a], [f, c, a], [g, c, a] ]
```

Na representação em árvore, esse mesmo conjunto de caminhos candidatos é representado pelo termo:

```
t(a, [t(b, [f(d), f(e)]), t(c, [f(f), f(g)])])
```

Essa representação pode parecer complexa e ainda mais consumidora de memória do que a representação em listas, entretanto isso é apenas a aparência superficial, devido à representação compacta que o Prolog utiliza para listas. Na representação do conjunto candidato por meio de listas, o efeito da pesquisa em profundidade era atingido pela movimentação dos caminhos expandidos para o fim do conjunto candidato. Não é possível usar o mesmo truque na representação em árvore, portanto nosso novo programa será algo mais complicado. A relação chave aqui será:

```
expande(Caminho, Arv, Arv1, Sol, Solução)
```

A Figura 14.12 ilustra a relação entre os argumentos da relação `expande/5`. Sempre que esta for ativa-

da, as variáveis Caminho e Arv já devem estar instanciadas. Arv é uma sub-árvore do espaço de estados e representa o conjunto de caminhos candidatos a um objetivo nessa sub-árvore. Caminho é o caminho entre o nodo inicial e a raiz de Arv. A idéia geral da relação `expande/5` é produzir Arv1 como uma extensão de um nível de Arv. Se, entretanto, durante a expansão de Arv, um nodo objetivo for encontrado, `expande/5` produzirá o correspondente caminho solução. Assim a relação `expande/5` irá produzir dois tipos de resultados. O tipo de resultado produzido será indicado pelo valor da variável Sol, como se segue:

(1) Sol = sim,

Solução = um caminho para solucionar o problema, e
Arv1 = não instanciada.

Resultados desse tipo somente serão produzidos quando houver um nodo objetivo em Arv (uma "folha-objetivo").

(2) Sol = não,

Solução = não instanciada,
Arv1 = Arv expandida de um nível.

Aqui Arv1 não contém nenhum desvio bloqueado, (desvios que não podem ser expandidos porque não possuem sucessores)

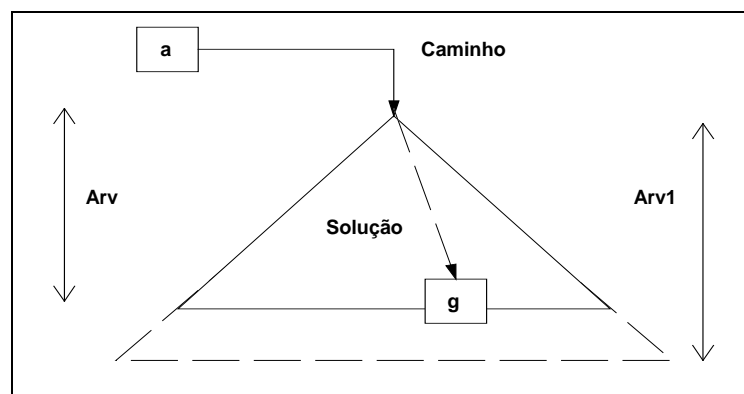


Figura 14.12: A relação `expande(Caminho,Arv,Arv1,Sol,Solução)`

A Figura 14.13 apresenta um programa completo, baseado nas idéias discutidas acima, empregando representação em árvore para o conjunto de caminhos candidatos. Um procedimento auxiliar é `expTodos/6`, similar ao `expande/5`, que realiza a expansão de um nível sobre um conjunto de árvores e armazena todas as árvores expandidas resultantes, removendo todas as árvores bloqueadas. Além disso esse procedimento produz, através de backtracking, todas as soluções encontradas nessa lista de árvores.

```

resolve(Início, Solução) :-
    ampl(f(Início), Solução).

ampl(Arv, Solução) :-
    expande([], Arv, Arv1, Sol, Solução),
    (Sol=sim; Sol=não, ampl(Arv1, Solução)).

expande(P, f(N), _, sim, [N | P]) :-
    objetivo(N).
expande(P, f(N), t(N, Subs), não, _) :-
    bagof(f(N), (s(N, M), not membro(M, P)), Subs).
expande(P, t(N, Subs), t(N, Subs1), Sol, Solução) :-
    expTodos([N | P], Subs, [], Subs1, Sol, Solução).

expTodos(_, [], [T | Ts], [T | Ts], não, _).
expTodos(P, [T | Ts], Ts1, Subs1, Sol, Solução) :-
    expande(P, T, T1, Sol1, Solução),
    ( Sol1=sim, Sol=Sim;
      Sol1=não, !, expTodos(P,Ts,[T1 | Ts1], Subs, Sol, Solução));
    expTodos(P, Ts, Ts1, Subs1, Sol, Solução).

```

Figura 14.13: Uma implementação do método de pesquisa em profundidade usando representação em árvore para o conjunto de caminhos candidatos

14.4 PESQUISA EM GRAFOS, OTIMIZAÇÃO E COMPLEXIDADE

Neste ponto é conveniente tecer alguns comentários sobre as técnicas estudadas até agora para a pesquisa em espaços de estados: pesquisa em grafos, otimização das soluções produzidas e complexidade de pesquisa.

Os exemplos apresentados neste capítulo podem produzir a falsa impressão de que os programas de pesquisa em amplitude somente funcionam para espaços de estado que podem ser representados por meio de árvores e que não são adequados para grafos em geral. O fato de se haver adotado uma representação em árvore não significa que o espaço de estados tenha obrigatoriamente de ser uma árvore. Na verdade, quando um espaço de estados na forma de um grafo é pesquisado, ele se desdobra em uma árvore, de forma que os mesmos caminhos percorridos podem ser representados em ambas as estruturas. Isso é ilustrado pela figura 14.14.

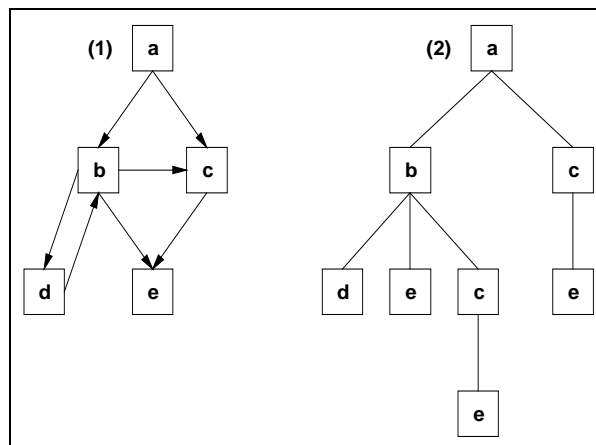


Figura 14.14: Desdobrando um grafo em uma árvore.

Em (1) representa-se um espaço de estados na forma de grafo. Se "a" é arbitrado o nodo inicial, então o grafo pode ser desdobrado na forma da árvore mostrada em (2), que contém todos os caminhos não-cíclicos possíveis desenvolvidos a partir de "a". A técnica de pesquisa em amplitude gera caminhos de solução, um após outro, ordenados de acordo com o seu tamanho: os caminhos mais curtos aparecem primeiro. Isso é importante se a otimização (no que toca ao comprimento do caminho deva ser consi-

derada. A técnica de pesquisa em amplitude garantidamente produz o caminho mais curto primeiro, o que não ocorre com a técnica de pesquisa em profundidade. O programa dado na Figura 14.13, entretanto, não leva em conta os custos associados aos arcos do espaço de estados. Se o custo mínimo de um caminho de solução é o critério para otimização (e não o seu tamanho), então a técnica de pesquisa em amplitude não é suficiente.

Outro problema típico associado com a pesquisa de espaços de estado é o da complexidade combinatória. Para os domínios de problemas não-triviais, o número de alternativas a ser explorado é tão grande que o problema da complexidade frequentemente se torna crítico. É fácil entender porque isso acontece: se cada nodo no espaço de estados tem n sucessores, então o número de caminhos de comprimento c a partir do nodo inicial é n^c (assumindo a inexistência de ciclos). Assim, o conjunto de caminhos candidatos cresce exponencialmente com o seu tamanho, o que conduz ao que se denomina explosão combinatória. As técnicas de pesquisa em profundidade e em amplitude não possuem nenhum recurso contra essa complexidade, uma vez que todos os caminhos candidatos são tratados de forma não-seletiva.

Um procedimento mais sofisticado para a pesquisa em espaços de estados complexos deveria empregar informações especificamente relacionadas ao problema de decidir a maneira mais promissora de agir em cada ponto da pesquisa. Isso teria o efeito de projetar o processo de pesquisa diretamente para o objetivo procurado, evitando os caminhos improdutivos. Informação associada ao problema específico que pode então ser empregada para dirigir a pesquisa é denominada heurística. Os algoritmos que utilizam heurísticas são denominados heurísticamente guiados e executam um tipo de pesquisa chamada pesquisa heurística, que será introduzida no próximo capítulo.

RESUMO

- Um espaço de estados é um formalismo para a representação de problemas de planejamento.
- Um espaço de estados é representado por meio de um grafo direcionado cujos nodos correspondem a situações do problema e os arcos a movimentos válidos que transformam uma situação em outra. Um problema particular é definido por um nodo inicial e um nodo objetivo. Uma solução do problema corresponde então a um caminho no grafo. Assim, a solução do problema é reduzida à procura por um caminho em um grafo.
- Problemas de otimização podem ser modelados pela associação de custos aos arcos de um espaço de estados.
 - Duas estratégias básicas que sistematicamente exploram um espaço de estados são: a pesquisa em profundidade (depth-first search) e a pesquisa em amplitude (breadth-first search).
- A pesquisa em profundidade é mais fácil de programar, mas é suscetível à presença de ciclos entre os nodos, conduzindo a ramificações infinitas da árvore de pesquisa.
- A implementação da estratégia de pesquisa em amplitude é mais complexa, uma vez que requer a manutenção de um conjunto de caminhos candidatos. Isso pode ser mais facilmente representado por meio de uma lista de listas, entretanto, o método mais eficiente emprega representação em árvore.
- No caso de grandes espaços de estados há o perigo da explosão combinatória. Tanto a pesquisa em profundidade quanto a pesquisa em amplitude são ferramentas pobres no combate a tal dificuldade, onde a aplicação de técnicas de pesquisa heurística se faz necessária.

EXERCÍCIOS

14.1 Escreva um procedimento denominado

`profundidade1(CaminhoCandidato, Solução)`

com detecção de ciclos, para encontrar um caminho, Solução, como uma extensão de Caminho-Candidato. Represente ambos os caminhos como listas de nodos em ordem inversa, de forma que o nodo objetivo é a cabeça da lista Solução.

14.2 Escreva um procedimento para pesquisa em profundidade combinando os mecanismos de detecção de ciclos e o de limitação da profundidade pesquisada.

14.3 Escreva um procedimento denominado

`apresenta(Situação)`

para representar um estado do mundo dos blocos. Situação deve ser representada por uma lista de pilhas e cada pilha como uma lista de blocos. Por exemplo, o objetivo

`mostra([[a], [e, d], [c, b]]).`

irá ocasionar a apresentação de

```
      e      c
      d      b
a
=====
```

14.4 Como se pode usar os procedimentos de pesquisa em amplitude estudados para permitir a pesquisa a partir de um conjunto de nodos iniciais, ao invés de um único?

14.5 Como se pode usar os procedimentos de pesquisa em profundidade e amplitude estudados para executar a pesquisa em direção inversa, isto é, a partir dos nodos objetivos retroagir até atingir um nodo inicial. (Dica: redefina a relação $s/2$). Em que situações a pesquisa retroativa seria vantajosa em relação à pesquisa progressiva?

14.6 Considere que há custos associados aos arcos de um espaço de estados. Escreva um programa (com detecção de ciclos que efetue a progressão em profundidade ou em amplitude, buscando minimizar o custo total da pesquisa.

15. PESQUISA HEURÍSTICA

A pesquisa em grafos para a solução de problemas pode conduzir ao problema da explosão combinatória, devido à proliferação de alternativas. A pesquisa heurística representa uma maneira de combater tal situação. Uma forma de utilizar informação heurística sobre um problema é computar estimativas heurísticas numéricas para os nodos no espaço de estados. Tal estimativa em um nodo indica o quanto promissor ele se mostra para atingir um nodo objetivo. A idéia é continuar a pesquisa sempre a partir do nodo mais promissor dentre os que compõem o conjunto de candidatos. O programa de pesquisa heurística (best-first search), apresentado no presente capítulo, baseia-se nesse princípio.

15.1 BEST-FIRST SEARCH

Um programa de pesquisa heurística pode ser derivado como um refinamento do programa de pesquisa em amplitude apresentado no capítulo anterior. A pesquisa heurística também inicia no primeiro nodo e mantém um conjunto de caminhos candidatos. A pesquisa em amplitude sempre escolhe para expansão os caminhos mais curtos. A pesquisa heurística refina este princípio pela computação de uma estimativa heurística para cada candidato e escolhe para expansão o melhor candidato de acordo com essa estimativa. A partir de agora vamos assumir que há uma função de custo definida sobre os arcos do espaço de estados. Assim, $c(n, n')$ é o custo de movimentação de um nodo n para um nodo sucessor n' no espaço de estados.

Seja a estimativa heurística traduzida por uma função f tal que para cada nodo n do espaço de estados, $f(n)$ estima a "dificuldade" de n . Então, o nodo candidato mais promissor será aquele que minimizar o valor de f . A função $f(n)$ é projetada de forma que, para estimar o custo de um caminho que conduza até a solução, percorrendo o caminho entre um nodo s , inicial, e um nodo objetivo, final, deva necessariamente passar por um determinado nodo n . Vamos supor que existe tal caminho e que um nodo objetivo que minimize o seu custo seja t . Então a função $f(n)$ pode ser construída como a soma de dois termos:

$$f(n) = g(n) + h(n)$$

conforme é ilustrado na figura 15.1.

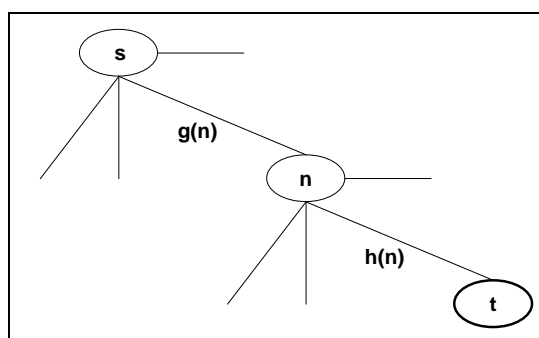


Figura 15.1: Construção de uma estimativa heurística $f(n)$ do custo do caminho mais barato de s a t via n : $f(n) = g(n) + h(n)$.

A função $g(n)$ representa a estimativa do custo de um caminho ótimo de s a n . $h(n)$ representa a estimativa do custo de um caminho ótimo de n a t .

Quando um nodo n é encontrado pelo processo de pesquisa, temos a seguinte situação: Um caminho de s a n já foi encontrado e o seu custo pode ser computado como a soma dos custos dos arcos nesse caminho. Esse caminho não é necessariamente um caminho ótimo de s a n (pode haver um caminho

melhor de s a n ainda não encontrado pela pesquisa) mas o seu custo pode servir *como um valor estimativo $g(n)$ do custo mínimo de s a n . O outro termo, $h(n)$ é mais problemático porque o espaço entre n e t ainda não foi explorado nesse ponto. Assim o valor de $h(n)$ é tipicamente uma perspectiva heurística real, baseada no conhecimento geral do algoritmo acerca do domínio do problema particular que esta sendo solucionado. Como $h(n)$ depende muito fortemente do domínio do problema, não há um método universal para a sua construção. Exemplos concretos de como essa previsão heurística pode ser estabelecida serão apresentados mais adiante. De momento vamos assumir que a função h é dada e nos concentrar nos outros detalhes do programa de pesquisa heurística.

Pode-se imaginar que a pesquisa heurística funcione da seguinte maneira: o processo de pesquisa consiste em diversos subprocessos competindo entre si, cada um dos quais explorando suas próprias alternativas, isto é, executando a pesquisa sobre os ramos de suas próprias sub-árvores. As sub-árvores, por sua vez, são constituídas de outras sub-árvores que são exploradas por subprocessos de subprocessos e assim por diante.

Entre todos esses processos competitivos, somente um está ativo em cada instante: o que lida com a alternativa mais promissora naquele momento, isto é a alternativa cujo valor para a função f é o mais baixo. Os processos restantes permanecem congelados até que o valor de f do processo em curso seja modificado de maneira que uma outra alternativa se revele mais promissora. Então a atividade é dada a essa alternativa. Pode-se imaginar esse mecanismo de ativação e desativação da seguinte maneira: ao processo trabalhando sobre a alternativa mais promissora é dado um crédito e o processo permanece ativo até que esse crédito tenha se esgotado. Durante o período em que está ativo, o processo continua a expansão da sua sub-árvore, informando uma solução se algum nodo objetivo for encontrado. O crédito para cada passo de execução é definido pela estimativa heurística da alternativa competidora mais próxima. Esse comportamento é exemplificado na Figura 15.2, que se divide em duas partes principais. Em (a) é representado uma mapa rodoviário (sem qualquer pretensão de representação em escala) onde as cidades são os nodos e os arcos representam estradas, rotuladas com as respectivas distâncias em alguma unidade qualquer. O objetivo é atingir a cidade t partindo de s , no menor trajeto rodoviário possível. os valores entre colchetes representam a distância em linha reta entre cada cidade e a cidade objetivo t . Esses valores serão utilizados para computar a função heurística que prevê a distância que resta a ser percorrida a partir de cada nodo, $h(n)$. Em (b) é representada a ordem na qual o mapa é explorado por meio da pesquisa heurística. A função estimativa heurística considera a distância até então percorrida e a que resta percorrer é estimada em função da distância em linha reta até t , dada em (a) pelos valores entre colchetes. os valores entre parênteses em (b) indicam a troca de atividade entre os caminhos alternativos, representando a ordem em que os nodos são expandidos e não a ordem em que são gerados. Na estimativa do custo da distância que resta a percorrer a partir de uma cidade X até o objetivo t , usamos a distância em linha reta denotada por $\text{dist}(X, t)$, de modo que:

$$f(x) = g(x) + h(x) = g(x) + \text{dist}(x, t)$$

No exemplo dado podemos imaginar a pesquisa heurística como constituída por dois processos, cada um deles explorando uma das duas rotas alternativas: o processo 1 a rota a partir de a e o processo 2 a rota a partir de e . Nos estágios iniciais, o processo 1 é mais ativo porque os valores de f ao longo desse caminho são os mais baixos. No momento em que o processo 1 está em c e o processo 2 ainda não saiu de e , a situação muda:

$$\begin{aligned} f(c) &= g(c) + h(c) = 6 + 4 = 10 \\ f(e) &= g(e) + h(e) = 2 + 7 = 9 \end{aligned}$$

então, como $f(e) < f(c)$, o processo 2 é ativado, deslocando a rota para f enquanto o processo 1 espera. Aqui, entretanto, a situação mais uma vez se inverte, pois:

$$\begin{aligned} f(f) &= 7 + 4 = 11 \\ f(c) &= 6 + 4 = 10 \\ f(c) &< f(f) \end{aligned}$$

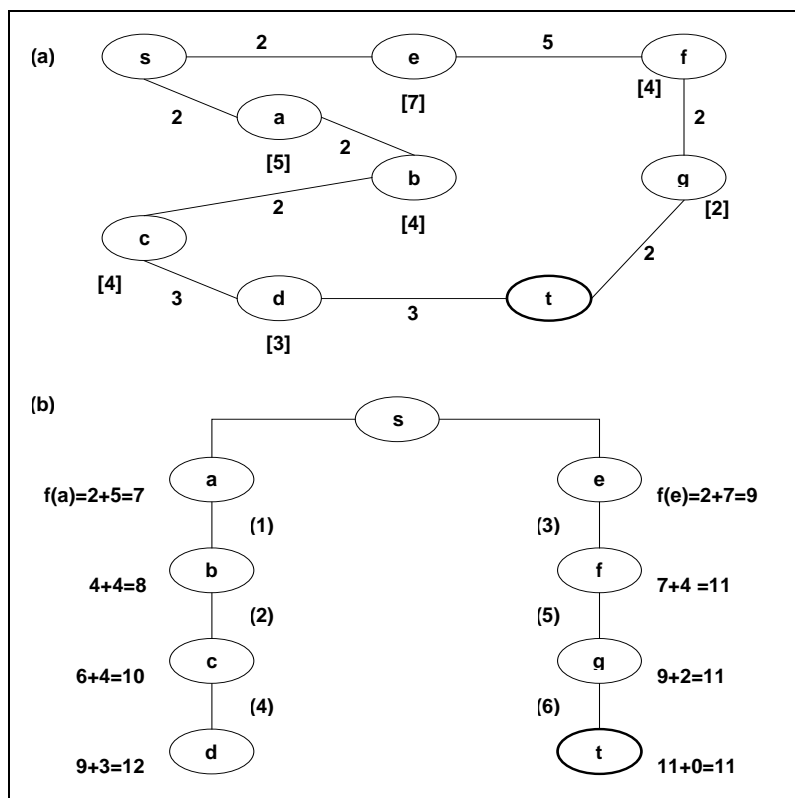


Figura 15.2: Encontrando a rota mais curta entre s e t em um mapa

Portanto o processo 2 pára e o processo 1 é novamente ativado. mas em seguida, no nodo d, temos $f(d) = 12 > 11$. A ativação passa mais uma vez ao processo 2 que, a partir daí, acaba por atingir o objetivo t. Vamos programar a pesquisa heurística como um refinamento do programa de pesquisa em amplitude estudado no capítulo anterior. O conjunto de caminhos candidatos será mais uma vez representado por uma árvore dada por meio de dois tipos de termos:

- (1) $n(N, F/G)$ representa um único nodo (uma folha). N é um nodo no espaço de estados. G é $g(N)$, o custo do caminho percorrido desde o nodo inicial até N, e F é $f(N)=G+h(N)$, e
- (2) $t(N, F/G, \text{subs})$ representa uma árvore com sub-árvores não-vazias. N é a raiz da árvore, Subs é uma lista de sub-árvores, G é $g(N)$ e F é o valor atualizado da função f, isto é, o valor de f para o sucessor mais promissor de N. A lista Subs é ordenada de acordo com valores crescentes de f para as sub-árvores que a compõem.

A atualização dos valores de f é necessária para permitir ao programa reconhecer a sub-árvore mais promissora a cada nível da árvore de pesquisa, isto é, a sub-árvore que contém o nodo mais promissor. Essa modificação dos valores de f conduz, na verdade, a uma generalização da definição da função f. Tal generalização amplia o domínio de definição de f, de nodos para árvores. Para um único nodo da árvore (uma folha), n, temos a definição original:

$$f(n) = g(n) + h(n)$$

Para uma árvore T cuja raiz é n e cujos sucessores de n são m_1, m_2, m_3 , etc, temos:

$$f(T) = \min f(m_i)$$

Um programa para executar pesquisa heurística segundo as linhas apresentadas é dado na Figura 15.3

```

heurist(Início, Solução) :-
    maior(M),          % M > qualquer valor de f
    expande([], n(Início, O/O), M, _, sim, Solução).

expande(P, n(N, _), _, _, sim, [N | P]) :-
    objetivo(N).
expande(P, n(N, F/G), Limite, , Arv1, Sol, Solução) :-
    F <= Limite,
    (bagof(M/C, (s(N,M,C), not membro(M,P)), Suc), !,
     sucLista(G, Suc, Ts), bestf(Ts, F1),
     expande(P,t(N,F1/G,Ts),Limite,Arv1,Sol,Solução);
     Sol = nunca).
expande(P,t(N,F/G,[T|Ts]),Limite, ,Arv1,Sol,Solução) :-
    F <= Limite,
    bestf(Ts, BF),
    min(Limite, BF, Limite1),
    expande([N|P], T, Limite1, T1, Sol, Solução),
    continua(P, t(N, F/G, [T1|Ts]),
             Limite, Arv1, Sol1, Sol, Solução).
expande(_, t(_, _, []), _, _, nunca, _).
expande(_, Arv, Limite, Arv, não, _) :-
    f(Arv, F), F > Limite.

continua(_,_,_,_,sim,sim,Solução).
continua(P,t(N,F/G,[T1|Ts]),Lim,Arv1,Sol1,Sol,Solução):-
    (Sol1=não, insere(T1,Ts,NTs); Sol1=nunca, Ts=Ts),
    bestf(NTs, F1),
    expande(P,t(N,F1/G,NTs),Lim,Arv1,Sol,Solução).

sucLista(_, [], []).
sucLista(G0, [N/C | NCs], Ts) :-
    G is G0+C, h(N, H), F is G+H,
    sucLista(G0, NCs, Ts1),
    insere((n(N, F/G), Ts1, Ts).

insere(T, Ts, [T|Ts]) :-
    f(T, F),
    bestf(Ts, F1),
    F <= F1, !.
insere(T, [T1|Ts], [T1|Ts1]) :-
    insere(T, Ts, Ts1).

f(n(_, F/_), F).
f(t(_, F/_), F).

bestf([T|_], F) :-
    f(T, F).
bestf([], M) :-
    maior(M).

min(X, Y, X) :- X <= Y, !.
min(X, Y, Y).

```

Figura 15.3: Um programa de pesquisa heurística.

Como no programa de pesquisa em amplitude apresentado no capítulo anterior, o procedimento chave aqui é `expande/6`, agora com um argumento adicional:

```
expande(P, Arv, Limite, Arv1, Sol, Solução)
```

e que corresponde à expansão da sub-árvore corrente enquanto o seu valor de `f` for menor ou igual a `Limite`. Os argumentos de `expande/6` são:

- **P:** Caminho entre o nodo inicial e `Arv`;
- **Arv:** Sub-árvore de pesquisa corrente;
- **Limite:** Limite do valor de `f` para a expansão de `Arv`;
- **Arv1:** `Arv` expandida dentro do valor `Limite`. Consequentemente o valor de `f` para `Arv1` é maior do que `Limite`, a menos que um nodo objetivo tenha sido atingido;
- **sol:** Indicador que pode assumir os valores `sim`, `não` OU `nunca`;

- **solução:** Um caminho-solução a partir do nodo inicial, passando por Arv1 até um nodo objetivo dentro do valor Limite.

Os parâmetros P, Arv e Limite são os argumentos de entrada do procedimento `expande/6`, isto é, eles devem estar instanciados sempre que esse procedimento for chamado. O procedimento `expande/6` produz três tipos de resultados, o que é indicado pelo valor do argumento Sol, da seguinte maneira:

- (1) `Sol = sim;`
`solução =` Um caminho-solução, encontrado pela expansão de Arv dentro do valor Limite;
`Arv1 =` Não instanciada.
- (2) `Sol = não;`
`solução =` Não instanciada;
`Arv1 =` Arv expandida de forma que seu valor para f excede o valor de Limite.
- (3) `Sol = nunca;`
`solução =` Não instanciada;
`Arv1 =` Não instanciada..

O último caso indica que Arv é uma alternativa "morta" à qual não deve ser dada nenhuma chance de expansão posterior. Esse caso surge quando o valor de f para Arv é menor ou igual ao valor de Limite mas a árvore não pode ser expandida porque nenhum nodo nela possui sucessor, ou então tal sucessor iria originar um ciclo.

Algumas das cláusulas sobre `expande/6` merecem uma explicação mais detalhada: As cláusulas que lidam com o caso mais complexo, quando Arv possui sub-árvores, isto é:

$$\text{Arv} = t(N, F/G, [T \mid Ts])$$

Em tais casos, primeiro a sub-árvore mais promissora, T é expandida. À essa expansão não é dado o limite Limite, mas possivelmente algum valor mais baixo, dependendo dos valores de f para as outras sub-árvores competidoras, Ts. Isso assegura que a sub-árvore em expansão em um determinado momento é sempre a mais promissora. Após o melhor caminho candidato ter sido expandido, um procedimento auxiliar, `continua/7` decide o que fazer a seguir. Isso depende do tipo de resultado produzido pela última expansão. Se uma solução foi encontrada, então ela é relatada ao usuário, senão o processo continua.

A cláusula que lida com o caso

$$\text{Arv} = n(N, F/G)$$

gera os nodos sucessores de N, juntamente com os custos dos arcos entre N e esses sucessores. O procedimento `sucLista/3` organiza uma lista de sub-árvores a partir desses nodos sucessores, também computando seus valores para as funções f e g. A árvore resultante é então expandida enquanto o valor de Limite permitir. Se, por outro lado, não há sucessores, então o nodo é abandonado para sempre, pela instancição de `Sol = nunca`. Outras relações a considerar são:

- **s(N, M, C):** M é um nodo sucessor de N no espaço de estados. C é o custo do arco que liga N a M.
- **h(N, H):** H é uma estimativa heurística do custo do melhor caminho do nodo N a algum nodo objetivo.
- **maior(M):** M é algum valor especificado pelo usuário, reconhecidamente maior do que qualquer valor possível para f

O programa apresentado na Figura 15.3 é uma variação de um algoritmo heurístico, conhecido na literatura como A*. Esse algoritmo sempre atraiu a atenção dos pesquisadores devido a suas características particulares. Um importante resultado extraído da análise matemática de A* é o seguinte:

Um algoritmo de pesquisa é dito ser admissível se sempre produz uma solução ótima (isto é,

um caminho de custo mínimo), se tal solução existe. O programa da Figura 15.3, que produz por backtracking todas as soluções possíveis, pode ser considerado admissível se a primeira solução encontrada for uma solução ótima. Considerando que para cada nodo n no espaço de estados, $h^*(n)$ denota o custo do caminho ótimo de n até um nodo objetivo. Um teorema sobre a admissibilidade de A^* diz que: Um algoritmo A^* que utiliza uma função heurística h tal que para todos os nodos n de um espaço de estados, $h(n) \leq h^*(n)$, é admissível.

Esse resultado possui um grande valor prático. Mesmo sem conhecer o valor exato de h^* podemos encontrar um limite inferior de h^* e empregá-lo como se fosse h em A^* . Isso é garantia suficiente de que A^* irá produzir uma solução ótima. Há um limite inferior trivial: $h(n) = 0$, para todo n no espaço de estados. Isso, na verdade, garante a admissibilidade. A desvantagem de se ter $h = 0$ é que isso não possui qualquer potencial heurístico e não oferece nenhuma orientação para a pesquisa. A^* usando $h = 0$ se comporta de maneira similar à pesquisa em amplitude. Na verdade se reduz à pesquisa em amplitude no caso em que a função de custo dos arcos $c(n, n')$ possui o valor 1 para todos os arcos (n, n') do espaço de estados. A falta de potencial heurístico resulta em elevada complexidade, assim desejaríamos ter valores para h que fossem limites inferiores de h^* (para assegurar a admissibilidade), mas que fossem tão próximos quanto possível de h^* (para assegurar a eficiência). No caso ideal, se conhecermos o valor de h^* , usamos esse próprio valor. Um algoritmo A^* usando h^* encontra a solução ótima diretamente, sem a necessidade de backtracking.

15.2 UMA APLICAÇÃO DA PESQUISA HEURÍSTICA

Para aplicar o programa da Figura 15.3 a algum problema particular, temos que adicionar as relações específicas do problema em questão. Tais relações, além de definir o problema, também transmitem, na forma de funções heurísticas, a informação heurística necessária à sua resolução. Os predicados específicos do problema são (1) $s(\text{Nodo}, \text{Nodo1}, \text{Custo})$, que é verdadeiro se existe um arco de custo Custo entre Nodo e Nodo1 no espaço de estados, (2) $\text{objetivo}(\text{Nodo})$, que é verdadeiro se Nodo é um nodo objetivo no espaço de estados, e (3) $h(\text{Nodo}, H)$, onde H é uma estimativa heurística do custo do caminho mais barato de Nodo até um nodo objetivo.

Como um exemplo iremos retomar o processo do jogo do oito, apresentado no capítulo anterior. As relações específicas do jogo do oito são apresentadas na figura 15.5. Um nodo no espaço de estados corresponde, nesse caso, a alguma configuração das peças do jogo. No programa isto é representado por meio de uma lista contendo as posições correntes das peças. Cada posição é representada por um par de coordenadas X/Y . A ordem das posições na lista é a seguinte:

- (1) Posição da casa vazia,
- (2) Posição da peça 1,
- (3) Posição da peça 2, etc...

A situação objetivo (ver Figura 15.4) é definida pela cláusula

```
objetivo( [ 2/2, 1/3, 2/3, 3/3, 3/2, 3/1, 2/1, 1/1, 1/2 ] ).
```

Uma relação auxiliar usada no programa é $d(S1, S2, D)$, onde D é a distância horizontal entre $S1$ e $S2$, somada com a distância vertical entre essas mesmas posições. Queremos minimizar z .

Uic

<table><tr><td>1</td><td>3</td><td>4</td></tr><tr><td>8</td><td></td><td>2</td></tr><tr><td>7</td><td>6</td><td>5</td></tr></table> <p>(a)</p>	1	3	4	8		2	7	6	5	<table><tr><td>2</td><td>8</td><td>3</td></tr><tr><td>1</td><td>6</td><td>4</td></tr><tr><td>7</td><td></td><td>5</td></tr></table> <p>(b)</p>	2	8	3	1	6	4	7		5	<table><tr><td>2</td><td>1</td><td>6</td></tr><tr><td>4</td><td></td><td>8</td></tr><tr><td>7</td><td>5</td><td>3</td></tr></table> <p>(c)</p>	2	1	6	4		8	7	5	3	<table><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>8</td><td></td><td>4</td></tr><tr><td>7</td><td>6</td><td>5</td></tr></table> <p>objetivo</p>	1	2	3	8		4	7	6	5
1	3	4																																					
8		2																																					
7	6	5																																					
2	8	3																																					
1	6	4																																					
7		5																																					
2	1	6																																					
4		8																																					
7	5	3																																					
1	2	3																																					
8		4																																					
7	6	5																																					

Figura 15.4: Três posições iniciais para atingir um objetivo:
(a) requer 4 movimentos, (b) requer 5 movimentos e (c) requer 18 movimentos.

No programa apresentado na Figura 15.5, a função heurística, h , é definida por meio da relação $h(\text{Pos}, H)$, onde Pos é uma posição do jogo e H é a combinação de dois fatores:

- (1) **disTot**: É a distância total das oito peças em Pos às suas casas correspondentes na situação objetivo. Por exemplo, na posição inicial apresentada na Figura 15.4, $\text{disTot} = 4$.
- (2) **seq**: É o "escore de sequência", que mede o grau de ordenação das peças na posição corrente em relação à ordem estabelecida na configuração objetivo. seq é computado pela soma dos escores de todas as peças de acordo com as seguintes regras:
 - Uma peça no centro do tabuleiro tem escore 1;
 - Uma peça em uma posição não-central tem escore 0 se é seguida pelo seu sucessor apropriado, em sentido horário;
 - Em qualquer outra situação, uma peça tem escore 2.

Por exemplo, para a posição inicial apresentada na Figura 15.4, $\text{seq} = 6$.

A estimativa heurística, H , é computada por:

$$H = \text{disTot} + 3 * \text{seq}$$

Essa função heurística funciona bem, no sentido em que dirige, de maneira muito eficiente, a pesquisa para o objetivo estabelecido. Por exemplo, na solução do jogo proposto pelas configurações iniciais apresentadas na Figura 15.4(a) e (b), nenhum nodo é expandido além dos que compõem o caminho mais curto para a solução. Isso significa que as soluções ótimas, nesses dois casos são encontradas diretamente, sem necessidade de backtracking. Mesmo o problema mais difícil, proposto na Figura 15.4(c) é solucionado quase que diretamente. Um problema com essa heurística, entretanto, é que ela não garante que a solução ótima será encontrada sempre antes de qualquer outra solução mais longa. A função h não satisfaz a condição de admissibilidade: $h \nleq h^*$ para todos os nodos do espaço de estados. Por exemplo, na posição inicial da Figura 15.4(a) temos:

$$h = 4 + 3 * 6 = 22$$

e

$$h^* = 4$$

Por outro lado, a medida de distância total é admissível, pois para todas as posições vale:

$$\text{disTot} \leq h^*$$

Essa relação pode ser facilmente demonstrada por meio do seguinte argumento: Se simplificássemos o problema, permitindo às peças passar umas por cima das outras, então cada peça poderia alcançar sua casa-objetivo seguindo uma trajetória cuja distância é exatamente a soma da sua distância horizontal com a distância vertical até esse objetivo. Então a solução ótima seria exatamente do tamanho computado por disTot . No problema original, entretanto, há interação entre as peças, que se encontram umas nos caminhos das outras. Isso evita que as peças possam ser movidas ao longo de suas trajetórias mais curtas, o que assegura que o tamanho da solução ótima encontrada será sempre maior ou igual a disTot .

```

s([Vazio | L], [T | L1], 1) :-
    move(Vazio, T, L, L1).

move(E, T, [T | L], [E | L]) :-
    d(E, T, 1).
move(E, T, [T1 | L], [T1 | L1]) :-
    move(E, T, L, L1).

d(X/Y, X1/Y1, D) :-
    dif(X, X1, Dx),
    dif(Y, Y1, Dy),
    D is Dx + Dy.

dif(A, B, D) :-
    D is A - B, D >= 0, !;
    D is B - A.

h([Vazio | L], H) :-
    objetivo([Vazio | G]),
    distTot(L, G, D),
    seq(L, S), H is D + 3*S.

distTot([], [], 0).
distTot([T | L], [T1 | L1], D) :-
    d(T, T1, D), distTot(L, L1, D2), D is D1+D2.

seq([Prim | L], S) :-
    seq([Prim | L], Prim, S).
seq([T1, T2 | L], Prim, S) :-
    escore(T1,T2,S1),
    seq([T2 | L], Prim,S2), S is S1+S2.
seq([Ult], Prim, S) :-
    escore(Ult, Prim, S).

escore(2/2, _, 1) :- !. escore(1/3, 2/3, 0) :- !.
escore(2/3, 3/3, 0) :- !. escore(3/3, 3/2, 0) :- !.
escore(3/2, 3/1, 0) :- !. escore(3/1, 2/1, 0) :- !.
escore(2/1, 1/1, 0) :- !. escore(1/1, 1/2, 0) :- !.
escore(1/2, 1/3, 0) :- !. escore(_, _, 2).

objetivo([2/2, 1/3, 2/3, 3/3, 3/2, 3/1, 2/1, 1/1, 1/2]).
início1([2/2, 1/3, 3/2, 2/3, 3/3, 3/1, 2/1, 1/1, 1/2]).
início2([2/1, 1/2, 1/3, 3/3, 3/2, 3/1, 2/2, 1/1, 2/3]).
início3([2/2, 2/3, 1/3, 3/1, 1/2, 2/1, 3/3, 1/1, 3/2]).

mostraSol([]).
mostraSol([P | L]) :-
    mostraSol(L),
    nl, write('---'),
    mostraPos(P).

mostraPos( [ S0, S1, S2, S3, S4, S5, S6, S7, S8 ] ) :-
    membro(Y, [3, 2, 1]), membro(X, [1, 2, 3]),
    membro(P-X/Y, [ '-S0, 1-S1, 2-S2, 3-S3, 4-S4
                    5-S5, 6-S6, 7-S7, 8-S8' ] ),
    nl, write(P), fail.
mostraPos(_).

```

Figura 15.5: Procedimentos específicos para o jogo do oito.

RESUMO

- Informações heurísticas podem ser usadas para estimar a distância entre um nodo e um objetivo em um espaço de estados. Neste capítulo considerou-se estimativas heurísticas numéricas;
- O princípio da pesquisa heurística orienta o processo de pesquisa de forma que o nodo expandido é sempre o mais promissor, de acordo com a estimativa heurística;
- O algoritmo de pesquisa heurística A*, que adota esse princípio, foi implementado na Figura 15.3;
- Para usar o algoritmo A* na solução de problemas concretos, um espaço de estados e uma função heurística devem ser definidos. Para problemas de grande complexidade a dificuldade reside em encontrar a função heurística apropriada;

- O teorema da admissibilidade ajuda a estabelecer se A^* , usando uma particular função heurística, irá sempre encontrar uma solução ótima.

EXERCÍCIOS

- 15.1 Proponha outras aplicações para o programa de pesquisa heurística apresentado no presente capítulo e formalize sua representação em Prolog.
- 15.2 Para os problemas propostos no exercício 15.1, determine a admissibilidade da função heurística escolhida.

16. REDUÇÃO DE PROBLEMAS E GRAFOS E/OU

Os grafos E/OU são uma representação adequada para problemas que podem ser decompostos em subproblemas mutuamente independentes. Exemplos de tais problemas incluem a seleção de roteiros, integração simbólica, jogos, prova automática de teoremas, etc. No presente capítulo serão desenvolvidos programas para a pesquisa heurística em grafos E/OU.

16.1 REPRESENTAÇÃO DE PROBLEMAS

Nos capítulos anteriores, a solução de problemas estava centrada na representação de seu espaço de estados. Assim, um problema podia ser reduzido a encontrar um caminho adequado em um espaço de estados. Uma outra representação, a dos grafos E/OU, parece adequar-se mais naturalmente a certos tipos de problemas, tirando partido da possibilidade de decomposição do problema original em subproblemas mutuamente exclusivos, que podem ser solucionados de forma independente.

Isso será ilustrado por meio de um exemplo. Seja o problema de encontrar uma rota em um mapa rodoviário entre duas cidades dadas, como é ilustrado na Figura 16.1. As distâncias entre as cidades são inicialmente desconsideradas. O problema poderia, naturalmente, ser reduzido a encontrar um caminho em um espaço de estados, que teria a mesma aparência do mapa, com os nodos correspondendo a cidades, os arcos a conexões diretas entre cidades e os custos dos arcos correspondendo às distâncias entre elas. Entretanto, vamos construir outra representação, baseada em uma decomposição natural do problema.

No mapa da Figura 16.1 há também um rio. Vamos assumir que há também duas pontes através das quais o rio pode ser cruzado: Uma ponte na cidade f e outra na cidade g. Obviamente a rota deverá incluir uma dessas pontes, de modo que forçosamente deve-se passar por f ou por g. Surgem então duas alternativas:

Para encontrar um caminho entre a e z:

- (1) Encontrar um caminho de a até z via f, ou
- (2) Encontrar um caminho de a até z via g.

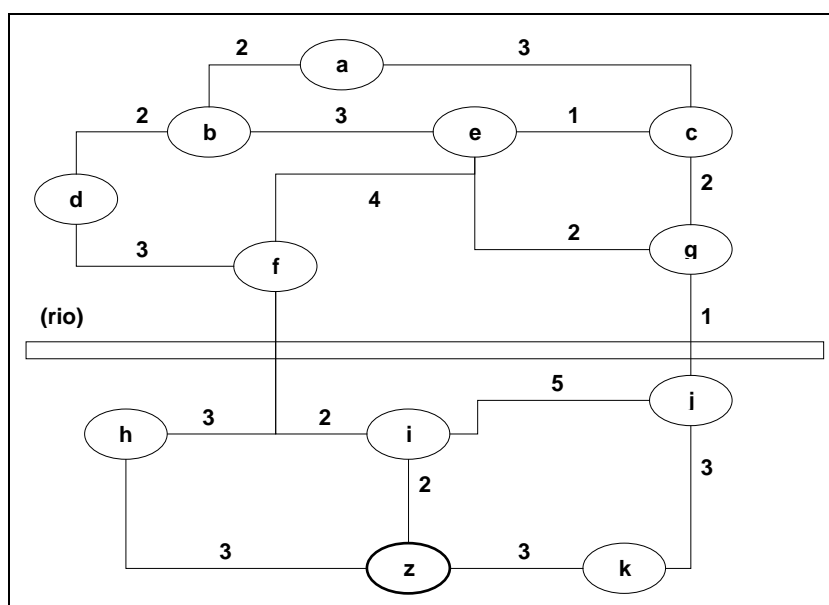


Figura 16.1: Encontrar um roteiro de a a z em um mapa rodoviário

As duas alternativas dadas podem agora ser decompostas da seguinte maneira:

(1) Para encontrar um caminho de a a z via f:

- 1.1 Encontrar um caminho de a a f;
- 1.2 Encontrar um caminho de f a z.

(2) Para encontrar um caminho de a a z via g:

- 2.1 Encontrar um caminho de a a g;
- 2.2 Encontrar um caminho de g a z.

Em resumo, tem-se duas alternativas para a solução do problema principal: (1) via f ou (2) via g.

Além disso, cada uma dessas duas alternativas pode ser decomposta em dois subproblemas (1.1 e 1.2) ou (2.1 e 2.2).

OU auxiliares, quando necessário. Assim um nodo a partir do qual são emitidos somente arcos E são denominados nodos E e os que emitem apenas arcos OU são chamados nodos OU.

Na representação através de espaços de estado, uma solução para um problema era dada por um caminho nesse espaço de estados. Na representação E/OU, uma solução tem necessariamente que incluir todos os subproblemas decorrentes de um nodo E, de maneira que esta não é representada mais por um caminho e sim por uma árvore. Essa árvore-solução, que denominaremos T é definida da seguinte maneira:

- O problema original P é a raiz da árvore T;
- Se P é um nodo OU, então somente um único dentre os seus sucessores, juntamente com a sua particular sub-árvore solução está em T;
- Se P é um nodo E, então todos os seus sucessores, juntamente com suas sub-árvores solução estão em T.

A Figura 16.4 ilustra essa definição. Ali temos custos associados aos arcos, que nos permitem formular um critério de otimização. Podemos, por exemplo, definir o custo de uma árvore solução como sendo a soma dos custos de todos os seus arcos. Como normalmente estamos interessados em minimizar os custos, a árvore solução apresentada em (c) deverá ser a preferida.

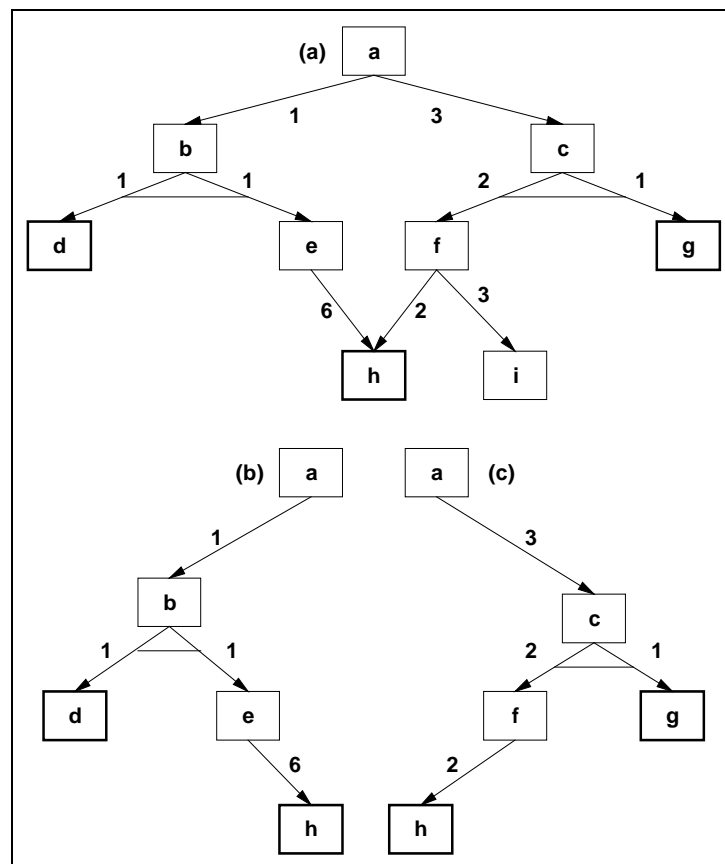


Figura 16.4: (a) Um grafo E/OU: d, g e h são nodos objetivos. (b) e (c) são árvores solução com custos 9 e 8 respectivamente

Não temos, entretanto, que basear nossa medida de otimização exclusivamente no custo dos arcos. Algumas vezes pode ser mais natural associar os custos com os nodos ao invés de com os arcos, ou mesmo com arcos e nodos simultaneamente. Resumindo os conceitos relacionados com a representação de grafos E/OU:

- A representação em grafos E/OU baseia-se no princípio da redução de problemas em subproblemas;
- Os nodos em um grafo E/OU correspondem aos problemas. As ligações entre os nodos correspondem às relações entre problemas;
- Um nodo do qual partem ligações OU é um nodo OU. Para solucionar um nodo OU basta solucionar um de seus sucessores;
- Um nodo do qual partem ligações E é um nodo E. Para solucionar um nodo E deve-se solucionar todos os seus sucessores;
- Para um determinado grafo E/OU, um particular problema é especificado através de duas coisas:
 - (1) Um nodo inicial, e
 - (2) Uma condição de reconhecimento de nodos objetivos;
- Nodos objetivos ou terminais correspondem a problemas triviais ou primitivos;
- Uma solução é representada por um grafo-solução, um subgrafo do grafo E/OU original;
- A representação dos espaços de estado pode ser vista como um caso especial da representação E/OU, na qual todos os nodos são nodos OU;
- Para nos beneficiarmos da representação E/OU, os nodos relacionados a uma condição E devem representar subproblemas que possam ser solucionados de forma mutuamente independente. O critério de independência pode ser relaxado se há uma ordenação entre os subproblemas E tal que as soluções dos subproblemas anteriores não sejam destruídas na solução dos que se sucedem;
- Custos podem ser associados aos arcos ou aos nodos ou a ambos, para a formalização de um critério de otimização.

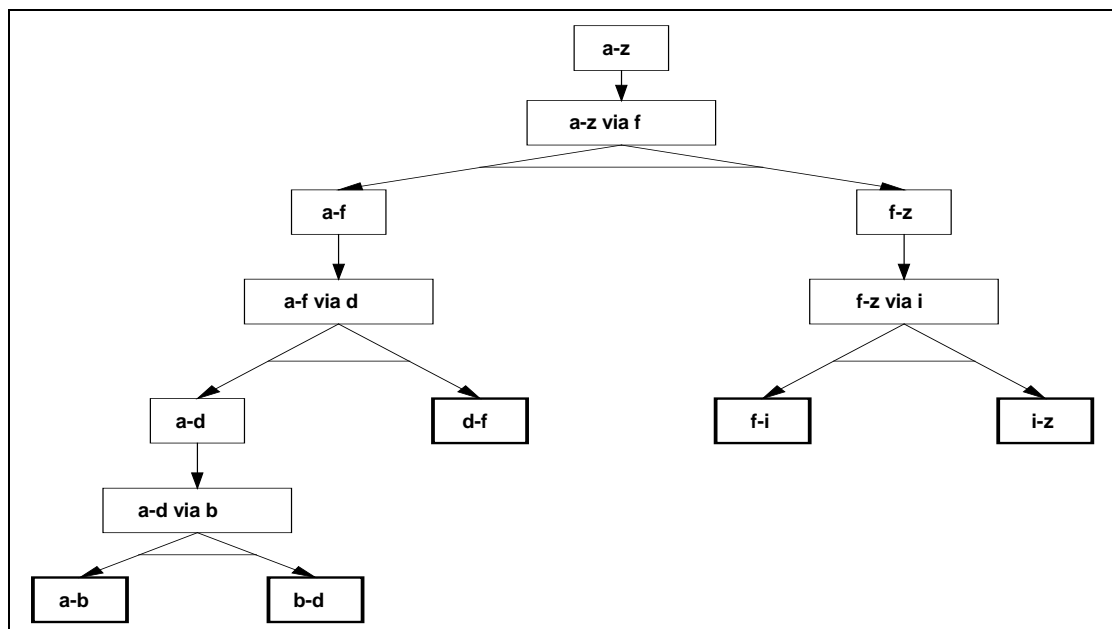


Figura 16.5: Uma solução para o problema de roteiros da Figura 16.1

16.2 EXEMPLOS DE REPRESENTAÇÃO DE PROBLEMAS EM GRAFOS E/OU

16.2.1 O PROBLEMA DE SELEÇÃO DE ROTEIROS

Para encontrar o caminho mais curto no problema proposto na Figura 16.1, um grafo E/OU, incluindo uma função de custo, pode ser definido da seguinte maneira:

- Os nodos OU são da forma X-Z, significando: encontre o caminho de X até Z;
- Os nodos E são da forma X-Z via Y, significando: encontre o caminho mais curto de X até Z, sob a restrição de que o caminho passe por Y;
- Um nodo X-Z é um nodo objetivo (um problema primitivo), se X e Z estão diretamente conectados no mapa;
- O custo de cada nodo objetivo X-Z é dado pela distância "rodoviária" entre as cidades X e Z;
- O custo de todos os demais nodos (não terminais) é zero.

16.2.2 O PROBLEMA DAS TORRES DE HANÓI

O problema das torres de Hanói, mostrado na Figura 16.6 é um outro exemplo clássico de uma aplicação efetiva do princípio de decomposição de problemas representado através dos grafos E/OU. Para fins de simplicidade consideraremos uma versão reduzida do problema contendo apenas três anéis.

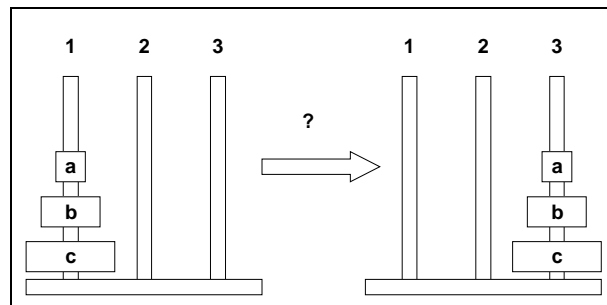


Figura 16.6: O problema das Torres de Hanói

Podemos enunciar o problema das Torres de Hanói da seguinte maneira: Há três "estacas", 1, 2 e 3, e três "anéis", a, b e c (sendo a o menor e c o maior). Inicialmente, todos os anéis estão empilhados ordenadamente na estaca 1. O problema é transferir-los para a estaca 3, na mesma ordem original, movendo apenas um anel de cada vez e respeitando a restrição de que nenhum anel pode ser colocado sobre outro menor do que ele. Este problema pode ser decomposto em três subproblemas:

- (1) Colocar o anel a na estaca 3;
- (2) Colocar o anel b na estaca 3;
- (3) Colocar o anel c na estaca 3.

Tais objetivos, entretanto, não são mutuamente independentes. Por exemplo, o anel a pode ser colocado imediatamente na estaca 3, entretanto isso impedirá a solução dos outros dois subproblemas (a menos que se desmanche a solução do primeiro), porque o enunciado original do problema proíbe a colocação de qualquer anel sobre outro menor do que ele. Por outro lado há uma ordenação conveniente dos objetivos que permite a derivação de uma solução. Essa ordenação deriva do seguinte raciocínio: O terceiro objetivo (anel c na estaca 3) é o mais difícil, porque a movimentação do anel c está sujeita a mais restrições. Uma boa idéia em casos como esse, que na maioria das vezes funciona, é tentar atingir primeiro o objetivo mais difícil. A lógica por trás deste princípio é a seguinte: como os outros objetivos são mais fáceis (não sujeitos a tantas restrições, possivelmente serão atingidos sem a necessidade de desmanchar a solução do mais difícil. A estratégia de solução que resulta desse princípio para o problema em questão é:

- (1) Primeiro satisfazer o objetivo: anel c na estaca 3;

(2) Depois, satisfazer os demais objetivos.

Mas esse primeiro objetivo não pode ser imediatamente atingido, porque na solução inicial o anel c não pode ser movido. Portanto é necessário primeiro possibilitar esse movimento, refinando a estratégia para:

- (1) Possibilitar a movimentação do anel c da estaca 1 para a estaca 3;
- (2) Mover o anel c da estaca 1 para a estaca 3;
- (3) Satisfazer os demais objetivos.

O anel c somente pode ser movido de 1 para 3 se tanto a como b estiverem empilhados na estaca 2. Assim o problema inicial, de mover a, b e c para a estaca 3 fica reduzido à seguinte formulação, composta de três subproblemas:

- (1) Mover a e b de 1 para 2;
- (2) Mover c de 1 para 3;
- (3) Mover a e b de 2 para 3.

O subproblema (2) é trivial (tem solução imediata). Os outros dois subproblemas podem ser resolvidos independentemente do problema (2) porque os anéis a e b podem ser movidos sem considerar a posição de c. Para resolver os problemas (1) e (3), o mesmo princípio de decomposição pode ser empregado (agora a movimentação do anel b é o problema mais difícil). O problema (1) pode então ser reduzido a três subproblemas triviais:

Para mover a e b de 1 para 2:

- (1) Mover a de 1 para 3;
- (2) Mover b de 1 para 2;
- (3) Mover a de 3 para 2.

A decomposição do problema restante (mover a e b de 2 para 3) fica como um exercício para o leitor.

16.3 PROCEDIMENTOS BÁSICOS DE PESQUISA EM GRAFOS E/OU

O modo mais simples de pesquisar grafos E/OU em Prolog é empregar o próprio mecanismo de pesquisa do Prolog. Isso é trivial, uma vez que o significado operacional dos programas Prolog nada mais é do que um procedimento para pesquisa em grafos E/OU. Por exemplo, o grafo E/OU apresentado na Figura 16.4 (ignorando os custos associados aos arcos), pode ser especificado por meio das seguintes cláusulas:

```
a :- b.
a :- c.
b :- d, e.
c :- f, g.
e :- h.
f :- h, i.
d. g. h.
```

Para perguntar se o problema a pode ser resolvido, simplesmente formula-se a consulta:

```
?- a.
```

e o sistema Prolog irá efetivamente pesquisar a árvore apresentada na Figura 16.4 em profundidade e responder "sim" após haver visitado a parte do grafo de pesquisa correspondente a árvore solução. A grande vantagem desta técnica de pesquisa E/OU é a sua simplicidade, entretanto ela possui algumas desvantagens:

- Somente se consegue obter respostas do tipo sim/não, e não a árvore solução como seria dese-

jável. Poder-se-ia inclusive reconstruir a árvore-solução a partir do mecanismo de tracing, mas essa seria uma solução grosseira e insuficiente no caso de se desejar a árvore solução explicitamente acessível como um objeto do programa;

- Programas desse tipo são difíceis de estender de modo a permitir a manipulação de custos;
- Se o grafo E/OU for um grafo genérico, contendo ciclos, então a estratégia de pesquisa em profundidade do Prolog poderia entrar em um laço recursivo infinito.

Tais deficiências serão removidas gradualmente. Inicialmente definiremos uma estratégia mais adequada para a pesquisa em profundidade em grafos E/OU. Para isso será introduzida uma relação binária que será representada pelo operador infixo "---->". Por exemplo, o nodo a, ligados ao seus dois sucessores "OU" será representado pela cláusula:

```
a ----> ou:[b, c]
```

Os símbolos "---->" e ":" são ambos operadores infixos que podem ser definidos da seguinte maneira:

```
:- op(600, xfx, '---->').
```

e

```
:- op(500, xfx, ':').
```

de modo que o grafo E/OU da Figura 16.4 pode ser completamente especificado por meio das cláusulas:

```
a ----> ou:[b, c].
b ----> e:[d, e].
c ----> e:[f, g].
e ----> ou:[h].
f ----> ou:[h, i].

objetivo(d).
objetivo(g).
objetivo(h).
```

A correspondente pesquisa em profundidade para grafos E/OU pode ser definida a partir dos seguintes princípios:

Para resolver um nodo N:

- (1) Se N é um nodo objetivo, então já está solucionado de forma trivial;
- (2) Se N possui sucessores OU, então solucione um deles. (Tente um de cada vez até que uma solução seja encontrada);
- (3) Se N possui sucessores E, então solucione todos eles. (Tente um de cada vez até que todos estejam solucionados);
- (4) Se as regras acima não produzirem uma solução, então assuma que o problema não pode ser resolvido.

Um programa para executar tais regras pode ser o seguinte:

```
resolve(Nodo) :-
    objetivo(Nodo).
resolve(Nodo) :-
    Nodo ----> ou:Nodos,
    membro(Nodo1, Nodos), resolve(Nodo1).
resolve(Nodo) :-
    Nodo ----> e:Nodos, resolveTodos(Nodos).

resolveTodos([]).
resolveTodos([Nodo|Nodos]) :-
    resolve(Nodo), resolveTodos(Nodos).
```

onde membro/2 é a relação usual de ocorrência em listas. O programa acima, no entanto, tem ainda as seguintes desvantagens:

- Não produz uma árvore solução, e
- É suscetível a laços infinitos, dependendo da presença de ciclos no grafo E/OU.

Pode-se entretanto modificá-lo facilmente para produzir uma árvore solução. Para isso modifica-se a relação `resolve/1` de modo que ela passe a ter dois argumentos:

```
resolve(Nodo, ArvSol).
```

Para a representação da árvore solução há três casos a considerar:

- (1) Se `Nodo` é um nodo objetivo, então a árvore solução correspondente é o próprio `Nodo`;
- (2) Se `Nodo` é um nodo OU, então sua árvore solução é da forma:

```
Nodo ---> SubArv
```

onde `SubArv` é uma árvore solução para um dos sucessores de `Nodo`;

- (3) Se `Nodo` é um nodo E, então sua árvore solução é da forma:

```
Nodo ---> e:SubArvs
```

onde `SubArvs` é a lista das árvores solução de todos os sucessores de `Nodo`.

Por exemplo, para o grafo E/OU da Figura 16.4, a primeira solução obtida a partir do nodo `a` é representada por:

```
a ---> b ---> e:[d, e ---> h]
```

As três formas de uma árvore solução correspondem às três cláusulas da relação `resolve/1` original. Assim, para modificar o programa é suficiente adicionar uma árvore solução como segundo argumento de `resolve/1`. Na Figura 16.7 é apresentado o programa resultante acrescido de um procedimento adicional, `mostra/2` para a apresentação de árvores solução. Tal programa, entretanto, ainda está sujeito a laços infinitos. Uma maneira simples de evitá-los é manter o acompanhamento da profundidade da pesquisa, impedindo o programa de ultrapassar um certo limite. Isso é obtido por meio da introdução de um terceiro argumento na relação `resolve/2`:

```
resolve(Nodo, ArvSol, ProfMax)
```

Como anteriormente, `Nodo` representa um problema a ser solucionado e `ArvSol` é uma solução cuja profundidade não ultrapassa `ProfMax`, que é a profundidade máxima permitida de pesquisa no grafo. No caso em que `ProfMax=0`, nenhuma expansão adicional é permitida. Por outro lado, se `ProfMax>0`, então `Nodo` pode ser expandido e seus sucessores serão examinados até uma profundidade limitada em `ProfMax-1`. Isso pode ser facilmente incorporado ao programa da Figura 16.7. Por exemplo, a segunda cláusula de `resolve/2`, acrescida do novo argumento fica:

```
resolve(Nodo, Arv, ProfMax) :-
    ProfMax > 0, Nodo ---> or:Nodos,
    membro(Nodo1, Nodos),
    P1 is ProfMax-1, resolve(Nodo1, Arv, P1).
```

Esse procedimento de pesquisa em profundidade limitada pode também ser utilizado para simular a pesquisa em amplitude. A idéia aqui é executar a pesquisa em profundidade de forma repetitiva, cada vez com um limite maior de profundidade, até que uma solução seja encontrada. Isto é, tentar o problema com `ProfMax=0`, depois 1, depois 2, etc. Um programa que implementa essa idéia é:

```
simulaAmpl(Nodo, ArvSol) :-
    tentaProf(Nodo, ArvSol, 0).

tentaProf(Nodo, ArvSol, Prof) :-
    resolve(Nodo, ArvSol, Prof),
    Prof1 is Prof+1,
    tentaProf(Nodo, ArvSol, Prof1).
```

```
resolve(Nodo, ArvSol) :-
    objetivo(Nodo).
resolve(Nodo, ArvSol) :-
    Nodo ---> ou:Nodos,
```

```

membro(Nodo1, Nodos), resolve(Nodo1, Arv).
resolve(Nodo, Nodos ---> e:Arvs) :-
    Nodos ---> e:Nodos, resolveTodos(Nodos, Arvs).
resolveTodos([], []).
resolveTodos([Nodo|Nodos], [Arv|Arvs] :-
    resolve(Nodo, Arv),
    resolveTodos(Nodos, Arvs).
mostra(Arv) :-
    mostra(Arv, 0), !.
mostra(Nodo ---> Arv, H) :-
    write(Nodo), write(--->),
    H1 is H+7, mostra(Arv, H1), !.
mostra(e:[T], H) :-
    mostra(T, H).
mostra(e:[T|Ts], H) :-
    mostra(T, H), tab(H), mostra(e:Ts, H), !.
mostra(Nodo, H) :-
    write(Nodo), nl.

```

Figura 16.7: Um programa para a pesquisa em profundidade em grafos E/OU

A desvantagem desse programa é que ele repete a pesquisa nos níveis superiores do grafo de pesquisa cada vez que a profundidade limite é incrementada.

16.4 PESQUISA HEURÍSTICA EM GRAFOS E/OU

Os procedimentos de pesquisa em grafos E/Ou apresentados na seção anterior executam sua tarefa de forma sistemática e exaustiva sem empregar qualquer perspectiva heurística. Para problemas complexos, tais procedimentos se apresentam ineficientes, devido à complexidade combinatória do espaço de pesquisa. É portanto necessário empregar funções heurísticas com o propósito de evitar as alternativas que acabarão por se tornar improdutivas. As perspectivas heurísticas que serão introduzidas na presente seção irão se basear em estimativas numéricas relacionadas com a dificuldade dos problemas em grafos E/OU. O programa que será desenvolvido pode ser visto como uma generalização do programa de pesquisa heurística em espaços de estados apresentado no capítulo anterior.

Inicialmente deve-se introduzir um critério de otimização baseado nos custos dos arcos em um grafo E/OU. Primeiro a representação de tais grafos será estendida para incluir custos. Por exemplo, o grafo E/OU da Figura 16.4 pode ser representado através das seguintes cláusulas:

```

a ---> ou:[b/1, c/3]
b ---> e:[d/1, e/1].
c ---> e:[f/2, g/1].
e ---> ou:[h/6].
f ---> ou:[h/2, i/3].

objetivo(d).
objetivo(g).
objetivo(h).

```

O custo da árvore solução será definido como sendo a soma dos custos de todos os arcos na árvore. O objetivo será uma árvore solução de custo mínimo. Para ilustração empregaremos mais uma vez a Figura 16.4.

É interessante definir o custo de um nodo em grafos E/OU como sendo o custo da árvore solução ótima para esse nodo. Assim definido, o custo do nodo passa a representar a dificuldade desse nodo. Assumiremos agora que podemos estimar os custos dos nodos no grafo E/OU, através de alguma função heurística h , mesmo sem conhecer suas árvores solução. Tais estimativas serão utilizadas para orientar a pesquisa. O programa começará a pesquisa no nodo inicial e , através de expansões realizadas sobre os nodos já visitados, construirá gradualmente uma árvore de pesquisa. Esse processo irá construir uma árvore mesmo nos casos em que o grafo E/OU não seja uma árvore. Em tais casos o grafo irá se desdobrar em uma árvore pela duplicação de algumas de suas partes.

O processo de pesquisa irá, a cada momento, selecionar para expansão a árvore candidata mais promissora ao desenvolvimento da árvore solução. A questão agora é: Como a função h é usada para estimar o quanto é promissora uma certa árvore candidata? Ou o quanto é promissor um determinado nodo candidato a raiz de uma árvore solução?

Para um determinado nodo N na árvore de pesquisa, $H(N)$ irá denotar a sua dificuldade estimada. para um nodo folha da árvore de pesquisa corrente, $H(N) = h(N)$. Por outro lado, para um nodo interior dessa árvore, não é necessário empregar a função h diretamente porque já se possui alguma informação adicional sobre ele, isto é, já conhecemos os seus sucessores. Portanto, como é ilustrado pela Figura 16.8, a dificuldade de um nodo OU interior pode ser dada aproximadamente por:

$$H(N) = \min(\text{custo}(N, N_i) + H(N_i))$$

onde $\text{custo}(N, N_i)$ é o custo do arco entre N e N_i . A regra de minimização se justifica pelo fato de que, para solucionar N , deve-se solucionar um dos seus sucessores. Já a dificuldade de um nodo E, N , é aproximada por:

$$H(N) = S(\text{custo}(N, N_i) + H(N_i))$$

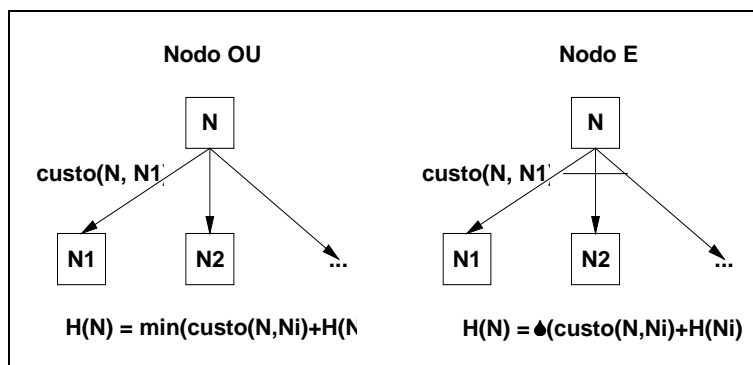


Figura 16.8: Estimativa da dificuldade, H , de problemas em grafos E/OU

Em nosso programa de pesquisa será mais prático, ao invés de valores de H utilizar uma outra medida, F , definida em termos de H da seguinte maneira: Seja M um nodo antecessor de N na árvore de pesquisa e $\text{custo}(M, N)$ o custo do arco que interliga M a N . Podemos definir:

$$F(N) = \text{custo}(M, N) + H(N)$$

De acordo com isso, se M é um nodo antecessor de N e N_1, N_2, \dots são nodos sucessores de N , então:

$$F(N) = \text{custo}(M, N) + \min F(N_i)$$

se N é um nodo OU, e

$$F(N) = \text{custo}(M, N) + S F(N_i)$$

se N é um nodo E.

O nodo inicial (representado por S) não possui antecessor, de modo que tem o seu custo (virtual) de chegada definido como zero. Entretanto, se h for igual a zero para todos os nodos objetivos do grafo E/OU e uma árvore solução ótima houver sido encontrada, então $F(S)$ tem o custo desta árvore solução, isto é, a soma dos custos de todos os seus arcos. Em qualquer estágio da pesquisa, cada sucessor de um nodo OU representa uma sub-árvore solução alternativa. O processo de pesquisa sempre irá escolher continuar a exploração através do sucessor cujo valor de F é mínimo. Esse processo pode ser acompanhado, mais uma vez, a partir da Figura 16.4. Inicialmente a árvore de pesquisa é o próprio nodo a . Depois essa árvore se expande até que uma solução seja encontrada. A Figura 16.9 mostra alguns momentos dessa expansão. para simplificar assumiremos que $h = 0$ para todos os nodos. Os números associados aos nodos na figura são os valores de F para esses nodos (que naturalmente serão alterados durante a pesquisa, à medida em que novas informações forem se acumulando).

A expansão da árvore inicial de pesquisa, (A), produz a árvore (B). O nodo a é um nodo OU, de modo que temos duas árvores solução candidatas: b e c. Como $F(b) = 1 < (F(c) = 3)$, a alternativa b será escolhida para expansão. Agora, até onde a alternativa b pode ser expandida? A expansão da árvore escolhida pode prosseguir até que:

- (1) O valor de F para o nodo b se torne maior do que o nodo c, que disputa com b a possibilidade de ser expandido, ou
- (2) Se torne claro que uma árvore solução foi encontrada.

Na Figura 16.9, o candidato b é o primeiro a ser expandido, uma vez que $F(b) \leq 3 = F(c)$. Inicialmente os sucessores de b, d e e são gerados (situação C) e o valor de F para o nodo b é aumentado para 3. Uma vez que isso não excede o valor limite, a árvore com raiz em b continua a ser expandida. O nodo d é reconhecido como um nodo solução e então o nodo e é expandido, resultando na situação D. Neste ponto, $F(b) = 9 > 3$, o que interrompe a expansão da alternativa b. Isso impede que o processo perceba que h é também um nodo objetivo e que uma árvore solução já foi gerada. Ao invés disso a atividade passa agora ao nodo c. O "crédito" para a expansão de F(c) agora é 9, uma vez que nesse ponto $F(b) = 9$. Dentro desse limite a árvore candidata de raiz c é expandida até que a situação E seja atingida. Agora o processo identifica que uma árvore solução (que inclui os objetivos g e h) foi encontrada, e o processo é encerrado. Deve-se notar que a solução final é a mais barata das duas possíveis árvores-solução, correspondendo à apresentada na Figura 16.4 (c).

16.4.1 UM PROGRAMA DE PESQUISA HEURÍSTICA EM GRAFOS E/OU

Um programa que implementa as idéias apresentadas na seção anterior é dado na Figura 16.11⁸. Antes de comentar os detalhes do programa iremos considerar as convenções empregadas na representação escolhida para a árvore de pesquisa.

⁸ Este programa gera uma única solução, que é garantidamente a mais barata se a função heurística empregada gerar valores não maiores do que os custos reais das árvores-solução.

A árvore de pesquisa pode ser:

- **arv(Nodo, F, C, SubArvs)**, correspondendo a uma árvore de soluções candidatas;
- **folha(Nodo, F, C)**, correspondendo a uma folha de uma árvore de pesquisa;
- **arvSol(Nodo, F, SubArvs)**, correspondendo a uma árvore solução;
- **folSol(Nodo, F)**, correspondendo a uma folha da árvore solução.

C é o custo do arco que chega a um nodo.

$F = C + H$, onde H é a estimativa heurística de uma árvore solução ótima cuja raiz é Nodo.

As sub-árvores são sempre ordenadas de modo que:

- (1) Todas as sub-árvores solucionadas se encontram no fim da lista, e
- (2) As demais sub-árvores são ordenadas em ordem crescente do seu valor de F

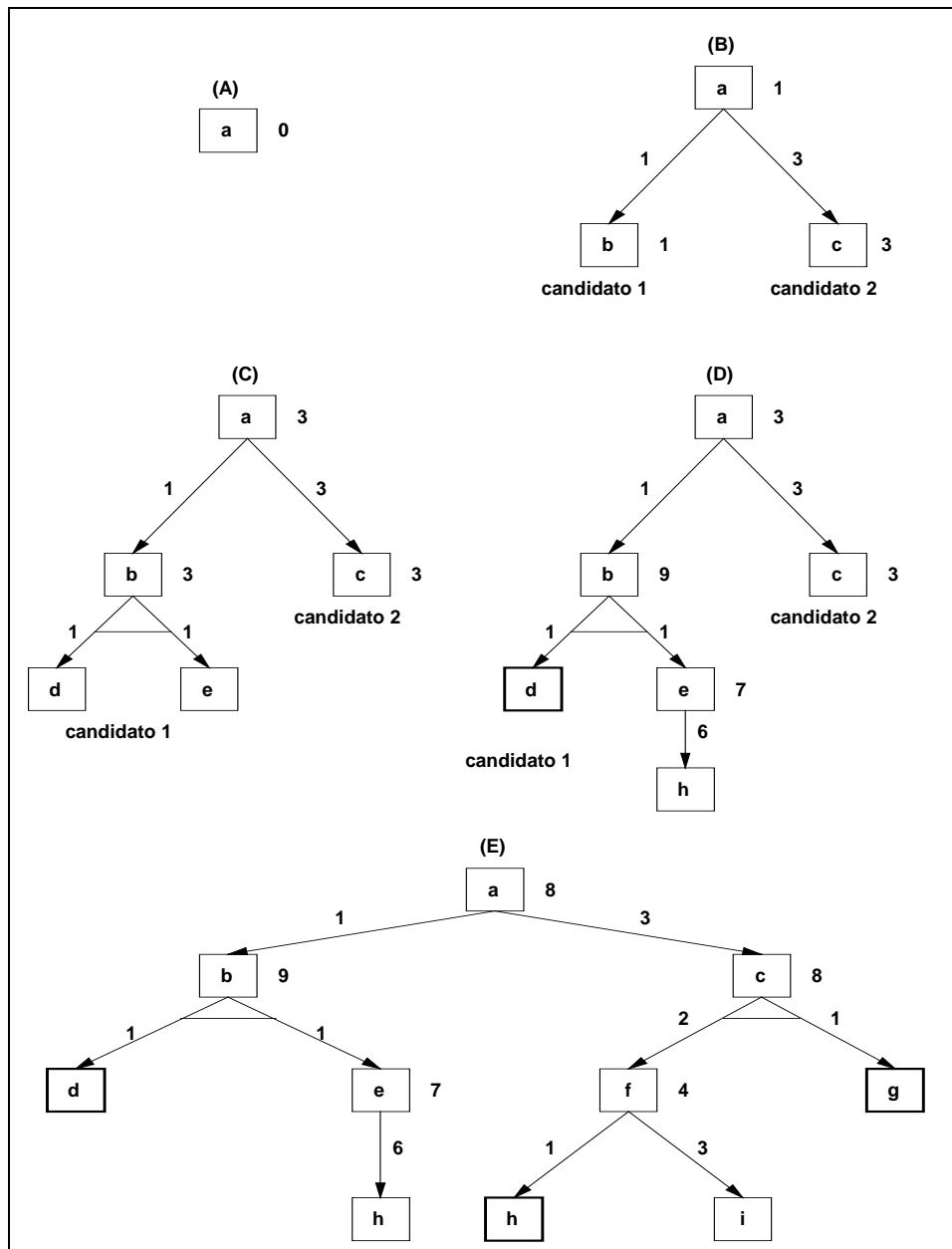


Figura 16.9: Aspectos de uma pesquisa heurística em um grafo E/OU

Há diversos casos a analisar, como pode ser visto na Figura 16.10. As diferentes formas que a árvore de pesquisa assume surgem em decorrência das seguintes possibilidades de combinação entre o tamanho da árvore e o seu estado de solução.

TAMANHO:

- (1) A árvore de pesquisa é formada por um único nodo (uma folha), ou
- (2) A árvore possui uma raiz que tem sub-árvores não-vazias.

ESTADO DE SOLUÇÃO:

- (1) A árvore já foi considerada como solucionada (é uma árvore-solução), ou
- (2) Ela ainda é uma árvore candidata a ser uma árvore-solução.

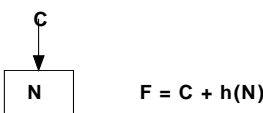
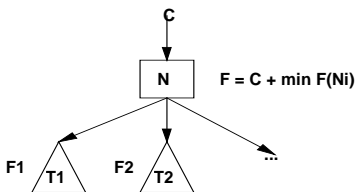
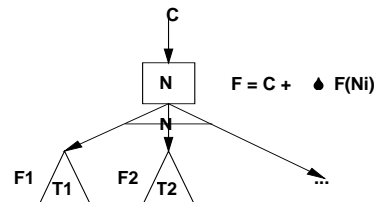
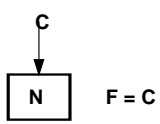
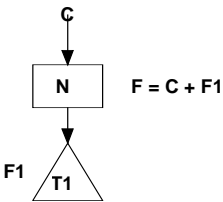
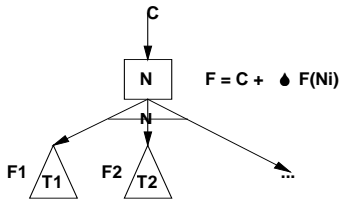
Caso 1: Folha de Pesquisa <code>folha(N, F, C)</code>	
Caso 2: Árvore de Pesquisa com sub-árvores OU <code>arv(N, F, C, ou:[T1, T2, ...])</code>	
Caso 3: Árvore de Pesquisa com sub-árvores E <code>arv(N, F, C, e:[T1, T2, ...])</code>	
Caso 4: Folha Solução <code>folSol(N, F)</code>	
Caso 5: Árvore Solução com raiz em um nodo OU <code>arvSol(N, F, T)</code>	
Caso 6: Árvore Solução com raiz em um nodo E <code>arvSol(N, F, e:[T1, T2, ...])</code>	

Figura 16.10: Representação da Árvore de Pesquisa

O functor principal usado para representar a árvore de pesquisa indica uma combinação dessas possibilidades, podendo ser um dos seguintes:

`folha/3 arv/4 folSol/2 arvSol/3`

Além disso, a representação abrange pelo menos algumas das informações seguintes:

- O nodo raiz da árvore;
- O valor da função F para a árvore;
- O custo C do arco no grafo E/OU que chega até a raiz da árvore;
- A lista das sub-árvores;
- A relação entre as sub-árvores (E ou OU).

A lista das sub-árvores estará sempre ordenada segundo valores crescentes para a função F. Uma sub-árvore pode inclusive já estar solucionada, sendo, nesse caso, acomodada no final da lista.

```

:- op(500, xfx, ':').      :- op(600, xfx, '--->').
eou(Nodo, ArvSol) :- expande(folha(Nodo,0,0), 9999, ArvSol, sim).

% Procedimento expande(Arv, Limite, NovaArv, Sol)
% Caso 1: Limite Ultrapassado.
expande(Arv, Limite, Arv, não) :- f(Arv, F), F > Limite, !.
% Caso 2: Objetivo Encontrado
expande(folha(Nodo,F,C), _, folSol(Nodo,F), sim) :- objetivo(Nodo).
% Caso 3: Expandindo uma Folha
expande(folha(Nodo,F,C), Limite, NovaArv, Sol) :-
    expNodo(Nodo, C, Arv1), !, expande(Arv1,Limite,NovaArv,Sol); Sol=nunca.
% Caso 4: Expandindo uma Árvore
expande(arv(N,F,C,SubArvs), Limite, NovaArv, Sol) :-
    Limite1 is Limite - C, expLista( SubArvs, Limite1, NovaSubs, Sol1),
    continua(Sol1, N, C, NovaSubs, Limite, NovaArv, Sol).

% Procedimento expLista(Arvs, Limite, NovaArvs, Sol) - Expande uma lista de árv.
% Arvs produzindo NovaArvs
expLista(Arvs, Limite, NovaArvs, Sol) :-
    selArv(Arvs,Arv,OutrasArvs,Limite,Limite1),
    expande(Arv, Limite1, NovaArv, Sol1), combina(OutrasArvs, NovaArv, Sol1, NovaArvs,Sol).

% Procedimento continua(Sol1,N,C,SubArvs,Limite,NovaArv,Sol) - Decide como continuar após
% expandir uma lista de árvores
continua(sim, N, C, SubArvs, _, arvSol(N,F,SubArvs), sim) :- backup(SubArvs, H), F is C+H, !.
continua(nunca, _, _, _, _, nunca) :- !.
continua(não, N, C, SubArvs, Limite, NovaArv, Sol) :-
    backup(SubArvs, H), F is C+H, !, expande(arv(N,F,C,SubArvs), Limite, NovaArv, Sol).

% Procedimento combina(Arvs,Arv,Sol1,NovaArvs,Sol) - Combina as sub-árvores expandidas em uma
% lista
combina(ou:_, Arv, sim, Arv, sim) :- !.
combina(ou:Arvs, Arv, não, ou:NovaArvs, não) :- insere(Arv, Arvs, NovaArvs), !.
combina(ou:[], _, nunca, _, nunca) :- !.
combina(ou:Arvs, _, nunca, ou:Arvs, não) :- !.
combina(e:Arvs, Arv, sim, e:[Arv|Arvs], sim) :- solTodas(Arvs), !.
combina(e:_, _, nunca, _, nunca) :- !.
combina(e:Arvs, Arv, Sol1, e:NovaArvs, não) :- insere(Arv, Arvs, NovaArvs), !.

% Procedimento expNodo(Nodo, C, Arv) - Constrói uma árvore com um nodo e seus sucessores
expNodo(Nodo, C, arv(Nodo, F, C, Op:SubArvs)) :-
    Nodo ---> Op:Sucessores, avalia(Sucessores, SubArvs), backup(Op:SubArvs, H), F is C+H.
avalia([], []).
avalia([Nodo/C | CustosDosNodos], Arvs) :-
    h(Nodo,H), F is C+H, avalia(CustosDosNodos,Arvs1), insere(folha(Nodo,F,C),Arvs1,Arvs).

% Procedimento solTodas(Arvs) - Verifica se todas as árvores da lista estão resolvidas
solTodas([]).
solTodas([Arv | Arvs]) :- sol(Arv), solTodas(Arvs).
sol(arvSol(_,_,_)).
sol(folSol(_,_)).

% Procedimento Insere(Arv, Arvs, NovaArvs) - Insere Arv na lista Arvs, produzindo Nova Arvs
insere(T, [], [T]) :- !.
insere(T, [T1 | Ts], [T, T1 | Ts]) :- sol(T1), !.
insere(T, [T1 | Ts], [T1 | Ts1]) :- sol(T), insere(T, Ts, Ts1), !.
insere(T, [T1 | Ts], [T, T1 | Ts]) :- f(T, F), f(T1, F1), F =< F1, !.
insere(T, [T1 | Ts], [T1 | Ts1]) :- insere(T, Ts, Ts1).
f(Arv, F) :- arg(2, Arv, F).

% Procedimento backup(Arvs, F)
% Recupera o valor de F armazenado em Arvs
backup(ou:[Arv | _], F) :- f(Arv, F), !.
backup(e:[], 0) :- !.
backup(e:[Arv1 | Arvs], F) :- f(Arv1, F1), backup(e:Arvs, F2), F is F1+F2, !.
backup(Arv, F) :- f(Arv, F).

% Relação selArv(Arvs, MelhorArv, Outras, Lim, Lim1) - Seleciona a melhor árvore, MelhorArv,
% de uma lista Arvs, deixando Outras. Lim é o limite de expansão para Arvs e Lim1 é o limite
% de expansão para MelhorArv.
selArv(Op:[Arv], Arv, Op:[], Lim, Lim) :- !.
selArv(Op:[Arv | Arvs], Arv, Op:Arvs, Lim, Lim1) :-
    backup(Op:Arvs, F), (Op=ou, !, min(Lim, F, Lim1); Op=e, Lim1 is Lim-F).

min(A, B, A) :- A < B, !.
min(A, B, B).

```

Figura 16.11: Programa de Pesquisa Heurística em Grafos E/OU.

No programa da Figura 16.11, a relação principal é:

`eou(Nodo, ArvSol)`

onde `Nodo` é o nodo inicial da pesquisa. O programa produz uma árvore solução `ArvSol` (se esta existir) que deve corresponder a uma solução ótima para o problema. Se esta será realmente a solução mais barata, isso vai depender da função heurística h adotada pelo algoritmo. Há um teorema, semelhante ao teorema da admissibilidade estudado no capítulo anterior, que se refere a essa dependência. Seja $CUSTO(N)$ uma função que denota o custo de uma árvore solução mais barata para um nodo N . Se, para cada nodo N no grafo E/OU , a estimativa heurística $h(N) \leq CUSTO(N)$, então a relação `eou/2` garantidamente irá encontrar uma solução ótima. Se a função $h(N)$ não satisfaz a essa condição, então a solução encontrada pode não ser uma solução ótima. Uma função heurística trivial que satisfaz a condição de admissibilidade é $h = 0$ para todos os nodos. A desvantagem dessa função é, naturalmente, a ausência de potencial heurístico. A relação chave acionada por `eou/2` é

`expande(Arv, Limite, Arv1, Sol)`

onde `Arv` e `Limite` são argumentos de entrada e `Arv1` e `Sol` são argumentos de saída. Seu significado é o seguinte:

- `Arv` é uma árvore de pesquisa que deve ser expandida;
- `Limite` é o limite para o valor de F que deve ser respeitado na expansão de `Arv`;
- `Sol` é um indicador cujo valor indica um dos seguintes três casos:
 - (1) `Sol=sim`: `Arv` pode ser expandida dentro de `Limite` de forma a abranger uma árvore solução `Arv1`,
 - (2) `Sol=não`: `Arv` pode ser expandida até `Arv1`, de forma que o valor de F para `Arv1` exceda `Limite` e não seja encontrada nenhuma sub-árvore solução, ou
 - (3) `Sol=nunca`: `Arv` é insolúvel;
- `Arv1` é, dependendo dos casos acima, uma árvore solução, uma extensão de `Arv` cujo valor de F ultrapassou o valor `Limite` ou permanecer não instanciada no caso em que `Sol=nunca`.

O procedimento `expLista/4`, definido por

`expLista(Arvs, Limite, Arvs1, Sol)`

é similar a `expande/4`. Assim como em `expande/4`, `Limite` é o limite de expansão de uma árvore e `Sol` é um indicador do que ocorreu durante a expansão (`sim`, `não` ou `nunca`). O primeiro argumento, entretanto é uma lista de árvores- E ou de árvores- OU :

`Arvs = e:[T1, T2, ...]` ou `Arvs = ou:[T1, T2, ...]`

O procedimento `expLista/4` seleciona a árvore mais promissora, T (conforme os valores de F) dentre os membros de `Arvs`. Devido à ordenação das sub-árvores em `Arv`, a mais promissora será sempre a primeira da lista, e será expandida com um novo limite, `Limite1`, que depende de `Limite` e também das outras sub-árvores em `Arvs`.

Se `Arvs` é uma lista OU , então `Limite1` corresponde ao valor de F para a próxima árvore mais promissora em `Arvs`. Se `Arvs` for uma lista E , então `Limite1` é `Limite` menos a soma dos valores de F das árvores restantes em `Arvs`. O conteúdo de `Arvs1` depende da situação indicada por `Sol`. No caso em que `Sol=não`, `Arvs1` é `Arvs` com a sua árvore mais promissora expandida até `Limite1`. Quando `Sol=sim`, `Arvs1` é uma solução da lista `Arvs` encontrada antes de `Limite` haver sido alcançado. Se `Sol=nunca`, `Arvs1` não possui instanciação.

O procedimento `continua/7`, chamado após a expansão de uma lista de árvores, decide o que deve ser feito a seguir, dependendo do resultado de `expLista/4`: Se constrói uma árvore solução, se atualiza a árvore de pesquisa e continua a sua expansão ou se informa "nunca" no caso em que a lista foi considerada insolúvel. Já o procedimento

```
combina(OutrasArvs, NovaArv, Sol1, NovaArvs, Sol)
```

relaciona diversos objetos manipulados por `expLista/4`. `NovaArvs` é a árvore expandida obtida por `expLista/4`, `OutrasArvs` são as árvores restantes e `Sol1` é o estado de solução de `NovaArv`. Esse procedimento manipula diversos casos, dependendo de `Sol1` e de se a lista de árvores é do tipo OU ou E. Por exemplo, a cláusula:

```
combina(ou:_, Arv, sim, Arv, sim)
```

significa: No caso em que a lista é do tipo OU e a árvore expandida foi solucionada e sua árvore solução é `Arv`, então toda a lista foi solucionada e a sua solução é a própria `Arv`. Para a apresentação de árvores solução pode-se definir um procedimento semelhante a `mostra/2`, apresentado na Figura 16.7. A construção de tal procedimento é deixada como um exercício para o leitor.

16.4.3 UM EXEMPLO DE DEFINIÇÃO DE PROBLEMA

Vamos agora formular o problema de seleção de roteiros sob a forma de um grafo E/OU de modo que a formulação obtida possa ser usada diretamente pelo procedimento `eu/2`, definido na Figura 16.11. Assumiremos que o mapa rodoviário será representado pela relação:

```
s(Cidade1, Cidade2, D)
```

significando que há uma ligação direta entre `Cidade1` e `Cidade2` a uma distância `D`. Assumiremos também a relação:

```
chave(Cidade1-Cidade2, Cidade3)
```

significando que, para encontrar um roteiro entre `Cidade1` e `Cidade2`, devemos considerar somente os caminhos que passam por `Cidade3` (`Cidade3` é ponto de passagem obrigatório entre `Cidade1` e `Cidade2`). Por exemplo, no mapa da Figura 16.1, `f` e `g` são pontos de passagem obrigatória entre `a` e `z`:

```
chave(a-z, f) e chave(a-z, g)
```

Implementaremos então os seguintes princípios relacionados com a seleção de roteiros:

Para encontrar um roteiro entre duas cidades, `a` e `z`

- (1) Se há pontos-chaves, `Y1`, `Y2`, ..., entre `a` e `z`, encontrar:
 - Um roteiro de `a` até `z` passando por `Y1`", ou
 - Um roteiro de `a` até `z` passando por `Y2`", ou
 - ...
- (2) Se não há nenhum ponto-chave entre `a` e `z`, então simplesmente encontre alguma cidade `b`, vizinha de `a`, tal que exista um roteiro entre `b` e `z`.

Temos então dois tipos de problemas que serão representados por:

- (1) `a-z`: Encontre um roteiro entre `a` e `z`;
- (2) `a-z via y`: Encontre um roteiro entre `a` e `z` passando em `y`.

Aqui "via" é um operador infixo com prioridade superior a "-" e inferior a "---->". O grafo E/OU correspondente pode agora ser implicitamente definido por:

```
:- op(550, xfx, via).
A-Z ----> ou:Lista :-
    bagof((A-Z via Y)/0, chave(A-Z, Y), Lista), !.
A-Z ----> ou:Lista :-
    bagof((Y-Z)/D, s(A, Y, D), Lista).
A-Z ----> e: [(A-Y)/0, (Y-Z)/0].
objetivo(A-A).
```

RESUMO

- A representação através de grafos E/OU é um formalismo adequado para a representação de problemas que podem ser decompostos em subproblemas independentes;
- Os nodos em um grafo E/OU podem ser nodos E ou nodos OU;
- Um problema concreto é definido por um nodo inicial e uma condição objetivo. A solução de um problema é apresentada através de um grafo solução;
- Custos de arcos e nodos podem ser introduzidos em um grafo E/OU na modelagem de problemas que exijam otimização;
- A solução de problemas representados por meio de grafos E/OU envolvem pesquisa nesse grafo. A pesquisa em profundidade é executada de maneira sistemática e é fácil de programar, entretanto, pode ser ineficiente em problemas complexos devido à explosão combinatória;
- Funções heurísticas podem ser introduzidas para estimar a dificuldade dos problemas. O princípio da pesquisa heurística pode ser usado como orientação, entretanto, a implementação dessa estratégia não é tão simples.

EXERCÍCIOS

- 16.1 Defina em Prolog um espaço E/OU para o Problema das Torres de Hanói. Use a definição encontrada com os procedimentos de pesquisa estudados no presente capítulo.
- 16.2 Considere algum jogo simples para duas pessoas e escreva a sua representação E/OU. Use um programa de pesquisa em profundidade em grafos E/OU para encontrar estratégias vitoriosas sob a forma de árvores E/OU.

APÊNDICE A

FUNDAMENTOS TEÓRICOS DA PROGRAMAÇÃO EM LÓGICA

O presente texto apresenta a evolução ordenada dos conceitos associados à Programação em Lógica, inclusive desenvolvendo as semânticas Modelo e Prova-Teoréticas. Seu objetivo é oferecer ao leitor uma visão abrangente das idéias fundamentais que sustentam a Programação em Lógica, ensejando a indagação científica e o desenvolvimento de novos estudos nessa área.

A.1 PROGRAMAÇÃO EM LÓGICA DE PRIMEIRA ORDEM

A.1.1 PROGRAMAS EM LÓGICA

Um programa em lógica é constituído por sentenças que expressam o conhecimento relevante para o problema que se pretende solucionar. A formulação de tal conhecimento emprega dois conceitos básicos: a existência de objetos discretos, que denominaremos *indivíduos*, e a existência de *relações* entre eles. Os indivíduos, considerados no contexto de um problema particular, constituem o *domínio* do problema. Por exemplo, se o problema é solucionar uma equação algébrica, então o domínio deve incluir pelo menos os números reais.

Para que possam ser representados por meio de um sistema simbólico tal como a lógica, tanto os indivíduos quanto as relações devem receber *nomes*. A atribuição de nomes é, entretanto, apenas uma tarefa preliminar na criação de modelos simbólicos para a representação de conhecimento. A tarefa principal é a construção de sentenças expressando as diversas propriedades lógicas das relações nomeadas. O raciocínio sobre algum problema baseado no domínio representado é obtido através da manipulação de tais sentenças por meio de inferência lógica. Em um ambiente típico de programação em lógica, o programador estabelece sentenças lógicas que, reunidas, formam um programa. O computador então executa as inferências necessárias para a solução dos problemas propostos.

A lógica de primeira ordem possui dois aspectos: sintático e semântico. O aspecto sintático diz respeito às *fórmulas bem-formadas* (fbfs) admitidas pela gramática de uma linguagem formal. O aspecto semântico está relacionado com o significado atribuído aos símbolos presentes nas fbfs da teoria. Apresenta-se a seguir os principais conceitos necessários para a definição de linguagens lógicas de primeira ordem.

DEFINIÇÃO A1: Teoria de Primeira Ordem

Uma *Teoria de Primeira Ordem* (TPO) consiste em uma *linguagem de primeira ordem* definida sobre um *alfabeto de primeira ordem*, um conjunto de axiomas e um conjunto de regras de inferência. A linguagem de primeira ordem consiste nas fbfs da teoria. Os axiomas e regras de inferência são utilizados para a derivação dos teoremas da teoria. ■

DEFINIÇÃO A2: Alfabeto de Primeira Ordem

Um *Alfabeto de Primeira Ordem* é constituído por sete classes de símbolos:

- (i) Variáveis Individuais,
- (ii) Constantes Individuais,
- (iii) Constantes Funcionais,
- (iv) Constantes Predicativas,
- (v) Conetivos,
- (vi) Quantificadores, e
- (vii) Símbolos de Pontuação.

As classes (v) a (vii) são as mesmas para todos os alfabetos, sendo denominadas *símbolos lógicos*. As classes (i) a (iv) podem variar de alfabeto para alfabeto e são denominadas *símbolos não-lógicos*. Para qualquer alfabeto de primeira ordem, somente as classes (ii) e (iii) podem ser vazias. Adotaremos aqui as seguintes convenções para a notação dos símbolos do alfabeto: As variáveis individuais serão denotadas por cadeias de símbolos iniciando com letras minúsculas (a, b, ..., z). Os conectivos são: \neg , \wedge , \vee , \leftarrow , e \leftrightarrow . Os quantificadores são \forall e \exists . Os símbolos de pontuação são '(', ')', e ', '. Adotaremos a seguinte hierarquia para a precedência entre conectivos e quantificadores. Em ordem decrescente:

$$\neg, \forall, \exists$$

$$\vee$$

$$\wedge$$

$$\leftarrow, \leftrightarrow$$

DEFINIÇÃO A3: Termo

Um *termo* é definido recursivamente da seguinte maneira:

- (i) Uma variável individual é um termo;
- (ii) Uma constante individual é um termo;
- (iii) Se f é uma função n -ária e t_1, t_2, \dots, t_n são termos, então $f(t_1, t_2, \dots, t_n)$ é um termo. ■

DEFINIÇÃO A4: Fórmula Bem-Formada (fbf)

Uma *fórmula bem-formada* (fbf) é definida indutivamente da seguinte maneira:

- (i) Se p é uma constante predicativa e t_1, t_2, \dots, t_n são termos, então $p(t_1, t_2, \dots, t_n)$ é uma fórmula bem formada (denominada fórmula atômica ou simplesmente átomo);
- (ii) Se f e g são fórmulas bem formadas, então $(\neg f)$, $(f \wedge g)$, $(f \vee g)$, $(f \leftarrow g)$ e $(f \leftrightarrow g)$ são fórmulas bem formadas;
- (iii) Se f é uma fórmula bem formada e X é uma variável, então $(\forall Xf)$ e $(\exists Xf)$ são fórmulas bem formadas. ■

Adotou-se a convenção de escrever a implicação de modo reverso, isto é $(f \leftarrow g)$, devido a sua conveniência na representação da forma clausal, que será descrita mais adiante. Também, por abuso da linguagem, de agora em diante se empregará indistintamente a palavra *fórmula* para fazer referência a *fórmulas bem formadas*.

DEFINIÇÃO A5: Linguagem de Primeira Ordem

Uma *linguagem de primeira ordem* sobre um alfabeto de primeira ordem é o conjunto de todas as fórmulas bem formadas construídas a partir dos símbolos deste alfabeto. ■

A semântica informal dos conectivos e quantificadores é a seguinte: \neg representa a negação, \wedge a conjunção (e), \vee a disjunção (ou), \leftarrow a implicação e \leftrightarrow a equivalência. \exists é o quantificador existencial, tal que ' $\exists X$ ' significa 'existe um X ', enquanto que \forall é o quantificador universal e ' $\forall X$ ' significa 'para todo X ' ou 'qualquer que seja X '. Assim a semântica informal de $\forall X(p(X, g(X)) \leftarrow q(X) \wedge \neg r(X))$ é: 'para todo X , se $q(X)$ é verdadeiro e $r(X)$ é falso, então $p(X, g(X))$ é verdadeiro'.

DEFINIÇÃO A6: Escopo de um Quantificador e Ocorrência Ligada de uma Variável em uma Fórmula

O escopo de $\forall X$ em $\forall Xf$ e de $\exists X$ em $\exists Xf$ é f . Uma *ocorrência ligada* de uma variável em uma fórmula é uma ocorrência que imediatamente segue o quantificador e qualquer ocorrência dessa mesma variável no escopo desse quantificador. Qualquer outra ocorrência de variável é dita ser *livre*. ■

DEFINIÇÃO A7: Fórmula Fechada

Uma fórmula é dita ser *fechada* quando não contém nenhuma ocorrência de variáveis livres. ■

DEFINIÇÃO A8: Fecho Universal e Fecho Existencial

Se f é uma fórmula, então $\forall(f)$ denota o *fecho universal* de f , que é a fórmula fechada obtida pela imposição de um quantificador universal a todas as variáveis que ocorrem livremente em f . Da mesma forma, $\exists(f)$ denota o *fecho existencial* de f , obtido pela imposição de um quantificador existencial a todas as variáveis que ocorrem livremente em f . ■

DEFINIÇÃO A9: Literal

Um literal é um átomo ou a negação de um átomo. Um literal positivo é um átomo, enquanto que um literal negativo é a negação de um átomo. ■

DEFINIÇÃO A10: Cláusula

Uma cláusula é uma fórmula do tipo: $\forall X_1 \dots \forall X_s (l_1 \vee \dots \vee l_m)$, onde cada l_i é um literal e X_1, \dots, X_s são todas as variáveis que ocorrem em l_1, \dots, l_m . ■

Por exemplo, são cláusulas:

$$\forall X \forall Y \forall Z (p(X,Z) \vee \neg q(X,Y) \vee \neg r(Y,Z))$$

e

$$\forall X \forall Y (\neg p(X,Y) \vee r(f(X,Y),a))$$

Uma vez que as cláusulas são tão comuns na programação em lógica, é conveniente adotar-se uma notação clausal particular. Assim a cláusula:

$$\forall X_1 \dots \forall X_s (a_1 \vee \dots \vee a_k \vee \neg b_1 \vee \dots \vee \neg b_n),$$

onde $a_1, \dots, a_k, b_1, \dots, b_n$ são átomos e X_1, \dots, X_s são todas as variáveis que ocorrem nestes átomos, será representada por:

$$a_1, \dots, a_k \leftarrow b_1, \dots, b_n$$

Na notação clausal, todas as variáveis são assumidas universalmente quantificadas. As vírgulas no antecedente, b_1, \dots, b_n , denotam conjunção, enquanto que as vírgulas no conseqüente, a_1, \dots, a_k , denotam disjunção. Tais convenções se justificam uma vez que:

$$\forall X_1 \dots \forall X_s (a_1 \vee \dots \vee a_k \vee \neg b_1 \vee \dots \vee \neg b_n)$$

é equivalente a

$$\forall X_1 \dots \forall X_s (a_1 \vee \dots \vee a_k \leftarrow b_1 \wedge \dots \wedge b_n)$$

DEFINIÇÃO A11: Cláusula de Programa

Uma *cláusula de programa* é uma cláusula do tipo $a \leftarrow b_1, \dots, b_n$, que contém exatamente um literal positivo. O literal positivo, a , é denominado a *cabeça* da cláusula, enquanto que a conjunção de literais b_1, \dots, b_n é o *corpo* da mesma. ■

DEFINIÇÃO A12: Cláusula Unitária

Uma *cláusula unitária* é uma cláusula do tipo $a \leftarrow$. Isto é, uma cláusula de programa com o corpo vazio. ■

A semântica informal de $a \leftarrow b_1, \dots, b_n$ é: "*para todas as possíveis atribuições de cada uma das variáveis presentes na cláusula, se b_1, \dots, b_n são todos verdadeiros, então a é verdadeiro*". Assim, se $n > 0$, uma cláusula de programa é condicional. Por outro lado, a cláusula unitária é incondicional. Sua semântica informal é "*para todas as possíveis atribuições de cada uma das variáveis presentes em a , a é verdadeiro*".

DEFINIÇÃO A13: Programa em Lógica

Um *programa em lógica* é um conjunto finito de cláusulas de programa. ■

DEFINIÇÃO A14: Definição de um Predicado

Em um programa em lógica, o conjunto de todas as cláusulas de programa que possuem o mesmo predicado p na cabeça é denominado a *definição* do predicado p . ■

DEFINIÇÃO A15: Cláusula Objetivo

Uma *cláusula objetivo* é uma cláusula do tipo $\leftarrow b_1, \dots, b_n$, isto é, uma cláusula que possui o conseqüente vazio. Cada b_i ($i = 1, \dots, n$) é denominado um *sub-objetivo* da cláusula. ■

DEFINIÇÃO A16: Cláusula vazia

A *cláusula vazia*, denotada por \square , é a cláusula que possui tanto o antecedente quanto o conseqüente vazios. Tal cláusula deve ser interpretada como uma contradição. ■

DEFINIÇÃO A17: Cláusula de Horn

Uma *cláusula de Horn* é uma cláusula de programa ou uma cláusula objetivo. ■

As cláusulas de Horn são assim denominadas em homenagem ao matemático Alfred Horn, que primeiro lhes estudou as propriedades, em 1951. Uma de suas mais importantes características é que qualquer problema solúvel capaz de ser representado por meio delas, pode ser representado de tal forma que apenas uma das cláusulas seja uma cláusula objetivo, enquanto que todas as restantes serão cláusulas de programa. Para um grande número de aplicações da lógica, é suficiente empregar o contexto restrito das cláusulas de Horn. Na Figura A1 posicionamos as cláusulas de Horn em sua relação com a lógica matemática, o cálculo de predicados de primeira ordem e a forma clausal.

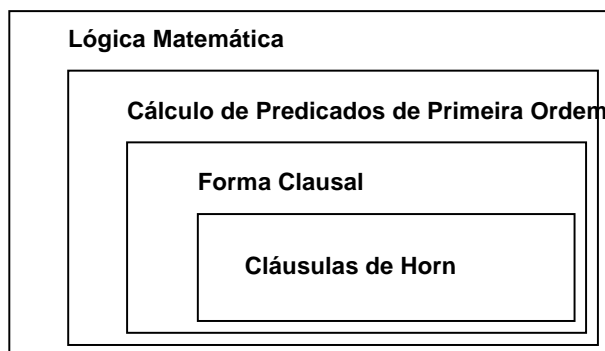


Figura A.1: Supercontextos das Cláusulas de Horn

A.2 SEMÂNTICA MODELO-TEORÉTICA**A.2.1 MODELOS DE PROGRAMAS EM LÓGICA**

Para que sejamos capazes de discutir sobre a verdade ou falsidade representadas através de fórmulas da lógica de primeira ordem, é necessário atribuir inicialmente algum significado a cada um dos símbolos nelas presentes. Os diversos conectivos e quantificadores possuem um significado fixo, entretanto, o significado atribuído às constantes individuais, constantes funcionais e constantes predicativas pode variar. Uma *interpretação* consiste simplesmente em algum universo de discurso, ou domínio, sobre o qual as variáveis podem assumir valores, na atribuição de um elemento desse universo a cada constante individual, na atribuição de um mapeamento sobre o domínio a cada constante funcional, e de uma relação sobre o domínio a cada constante predicativa. Cada interpretação especifica assim um significado para cada símbolo na fórmula. Estamos particularmente interessados em interpretações para as quais as fórmulas expressam uma declaração verdadeira. Tais interpretações são denominadas *modelos* para as fórmulas. Normalmente haverá alguma interpretação especial, denominada *interpretação pretendida*, que irá especificar o significado principal dos símbolos. Natu-

ralmente a interpretação pretendida sempre será um modelo. A partir de agora emprega-se os termos *constante*, *função* e *predicado* para designar respectivamente constantes individuais, constantes funcionais e constantes predicativas. A lógica de primeira ordem oferece métodos para a dedução dos teoremas presentes em alguma teoria. Estes podem ser caracterizados como sendo as fórmulas que são consequência lógica dos axiomas da teoria, isto é, que são verdadeiras em todas as interpretações que são modelos para cada um dos axiomas da teoria. Em particular, cada teorema deve ser verdadeiro na interpretação pretendida da teoria. Os sistemas de programação em lógica que são objeto do presente estudo adotam o Princípio da Resolução como única regra de inferência. Suponha-se que se deseja provar que a fórmula

$$\exists Y_1 \dots \exists Y_r (b_1 \wedge \dots \wedge b_n)$$

é uma consequência lógica de um programa P. Com esse objetivo emprega-se o Princípio da Resolução por meio de um sistema de refutação, isto é, a negação da fórmula a ser provada é adicionada aos axiomas e uma contradição deve ser derivada. Negando-se a fórmula que se deseja provar obtém-se a cláusula objetivo:

$$\leftarrow b_1, \dots, b_n$$

A partir dessa fórmula objetivo e operando de forma top-down sobre os axiomas de P, o sistema deriva sucessivas cláusulas objetivo. Se, em um determinado momento, for derivada a cláusula vazia, então uma contradição foi obtida (a cláusula vazia é contraditória) e esse resultado assegura que $\exists Y_1 \dots \exists Y_r (b_1 \wedge \dots \wedge b_n)$ é uma consequência lógica de P. De agora em diante se usará simplesmente *objetivo* para designar cláusulas objetivo.

Do ponto de vista da prova de teoremas, o único interesse é demonstrar a existência da relação de consequência lógica. Por outro lado, do ponto de vista da Programação em Lógica, o interesse se concentra muito mais sobre as *ligações* que foram realizadas sobre as variáveis Y_1, \dots, Y_r , uma vez que estas fornecem o *resultado* da execução do programa. Segundo [Llo 84], a visão ideal de um sistema de Programação em Lógica é a de uma caixa preta para a computação de ligações e o único interesse reside no seu comportamento de entrada e saída, isto é, as operações executadas internamente pelo sistema deveriam ser transparentes para o programador. Infelizmente tal situação não ocorre, em maior ou menor grau nos sistemas Prolog atualmente disponíveis, de forma que muitos programas Prolog somente podem ser entendidos a partir de sua interpretação operacional, devido ao emprego de *cuts* e outros mecanismos extra-lógicos.

DEFINIÇÃO A18: Interpretação

Uma *interpretação* de uma linguagem L de primeira ordem é constituída por:

- (i) Um conjunto não-vazio D, denominado o *Domínio* da interpretação;
- (ii) Para cada constante em L a atribuição de um elemento em D;
- (iii) Para cada função *n*-ária em L, a atribuição de um mapeamento de D^n em D;
- (iv) Para cada predicado *n*-ário em L a atribuição de um mapeamento de D^n em $\{V, F\}$, isto é, de uma *relação* sobre D^n . ■

DEFINIÇÃO A19: Atribuição de Variáveis

Seja I uma interpretação de uma linguagem L de primeira ordem. Uma *atribuição de variáveis* (com respeito a I) é uma atribuição de um elemento do domínio de I a cada uma das variáveis em L. ■

DEFINIÇÃO A20: Atribuição de Termos

Seja I uma interpretação de uma linguagem L de primeira ordem, com domínio D, e seja A uma atribuição de variáveis. Uma *atribuição de termos* (com respeito a I e A) para os termos em L é definida da seguinte maneira:

- (i) A cada variável em L é dada uma atribuição de acordo com A;
- (ii) A cada constante em L é dada uma atribuição de acordo com I;
- (iii) Se t_1', \dots, t_n' são as atribuições dos termos t_1, \dots, t_n e f' é a atribuição de f , então $f'(t_1', \dots,$

t_n') é a atribuição de termos $f(t_1, \dots, t_n)$. ■

DEFINIÇÃO A21: Valor Verdade de uma Fórmula

Seja I uma interpretação de domínio D de uma linguagem L de primeira ordem, e seja A uma atribuição de variáveis. Então a uma fórmula em L pode ser atribuído um *valor-verdade* (verdadeiro ou falso, que denotaremos por F e V respectivamente) com respeito a I e a A , da seguinte maneira:

- (i) Se a fórmula é um átomo, $p(t_1, \dots, t_n)$, então o valor verdade é obtido pelo cálculo do valor verdade de $p'(t_1', \dots, t_n')$, onde p' é o mapeamento atribuído a p por I e t_1', \dots, t_n' é a atribuição de termos para t_1, \dots, t_n com respeito a I e a A ;
- (ii) Se a fórmula tem a forma $\neg f$, $f \wedge g$, $f \vee g$, $f \leftarrow g$ ou $f \leftrightarrow g$, então o valor verdade da fórmula é dado pela tabela verdade:
- (iii) Se a fórmula tem a forma $\exists Xf$, então o valor verdade da fórmula é V se existe $d \in D$ tal que f tem valor verdade V com respeito a I e a $A(X/d)$, onde $A(X/d)$ é A , exceto que a X é atribuído o valor d . Caso contrário o seu valor verdade é F ;
- (iv) Se a fórmula tem a forma $\forall Xf$, então o valor verdade da fórmula é V se para todo $d \in D$, f tem valor verdade V com respeito a I e a $A(X/d)$. Caso contrário o seu valor verdade é F . ■

f	g	$\neg f$	$f \vee g$	$f \wedge g$	$f \leftarrow g$	$f \leftrightarrow g$
V	V	F	V	V	V	V
V	F	F	F	V	V	F
F	V	V	F	V	F	F
F	F	V	F	F	V	V

DEFINIÇÃO A22: Modelo de uma Fórmula

Seja I uma interpretação de uma linguagem L de primeira ordem e seja f uma fórmula fechada de L . Então I é um modelo para f se o valor verdade de f com respeito a I é V . ■

DEFINIÇÃO A23: Modelo de um Conjunto de Fórmulas Fechadas

Seja S um conjunto de fórmulas fechadas de uma linguagem L de primeira ordem e seja I uma interpretação de L . Dizemos que I é um modelo para S se I for modelo para cada uma das fórmulas em S . ■

DEFINIÇÃO A24: Conjunto de Fórmulas Satisfatível

Seja S um conjunto de fórmulas fechadas de uma linguagem L de primeira ordem. Dizemos que S é *satisfatível*, se L possui uma interpretação que é um modelo para S . ■

DEFINIÇÃO A25: Conjunto de Fórmulas Válido

Seja S um conjunto de fórmulas fechadas de uma linguagem L de primeira ordem. Dizemos que S é *válido* se toda interpretação de L é um modelo para S . ■

DEFINIÇÃO A26: Conjunto de Fórmulas Insatisfatível

Seja S um conjunto de fórmulas fechadas de uma linguagem L de primeira ordem. Dizemos que S é *insatisfatível*, se S não possui modelos em L . Note que $\{f, \neg f\}$ é insatisfatível, assim como a cláusula vazia, denotada por \square . ■

DEFINIÇÃO A27: Consequência Lógica de um Conjunto de Fórmulas Fechadas

Seja S um conjunto de fórmulas fechadas e seja f uma fórmula fechada de uma linguagem L de primeira ordem. Dizemos que f é *consequência lógica* de S , isto é, $S \models f$, se para toda interpretação I de L , se I é um modelo para S , então I é também um modelo para f . Note que se $S = \{f_1, \dots, f_n\}$ é um conjunto finito de fórmulas fechadas, então f é consequência lógica de S se e somente se $f \leftarrow f_1 \wedge \dots \wedge f_n$ é válida. ■

PROPOSIÇÃO A.1

Seja S um conjunto de fórmulas fechadas e f uma fórmula fechada de uma linguagem L de primeira ordem. Então f é consequência lógica de S se e somente se $S \cup \{\neg f\}$ é insatisfatível.

Prova:

(\rightarrow) Vamos supor que f seja consequência lógica de S . Se $S \cup \{\neg f\}$ é satisfatível, então existe uma interpretação I da linguagem L tal que I é modelo de $S \cup \{\neg f\}$. Por outro lado, se f é consequência lógica de S , então I é também modelo de f , ou seja de $\{f, \neg f\}$, o que não é possível. Logo $S \cup \{\neg f\}$ é insatisfatível.

(\leftarrow) Inversamente, vamos supor que $S \cup \{\neg f\}$ seja insatisfatível e seja I uma interpretação da linguagem L . Suponhamos que I seja um modelo para S . Uma vez que $S \cup \{\neg f\}$ é insatisfatível, I não pode ser um modelo para $\neg f$. Assim, I é um modelo para f e portanto f é consequência lógica de S . ■

Aplicando essas últimas definições a programas em lógica, constata-se que quando se fornece um objetivo G ao sistema com o programa P carregado, está-se pedindo ao sistema para provar que $P \cup \{G\}$ é insatisfatível. Se G é o objetivo $\leftarrow b_1, \dots, b_n$ com as variáveis Y_1, \dots, Y_r , então a Proposição A.1 estabelece que provar que $P \cup \{G\}$ é insatisfatível equivale a provar que $\exists Y_1 \dots \exists Y_r (b_1 \wedge \dots \wedge b_n)$ é consequência lógica de P . Assim o problema básico é a determinação da insatisfatibilidade de $P \cup \{G\}$, onde P é um programa e G é um objetivo. De acordo com a definição de insatisfatibilidade, isso implica em mostrar que *nenhuma* interpretação de $P \cup \{G\}$ é um modelo.

DEFINIÇÃO A28: Termo Básico e Átomo Básico

Um termo básico é um termo que não contém variáveis. Da mesma forma um átomo básico é um átomo que não contém variáveis. ■

DEFINIÇÃO A29: Universo de Herbrand

Seja L uma linguagem de primeira ordem. O *Universo de Herbrand*, UL para L é o conjunto de todos os termos básicos que podem ser obtidos a partir das constantes e funções presentes em L . No caso em que L não possui constantes, introduz-se uma constante (por exemplo, "a") para a formação de termos básicos. ■

DEFINIÇÃO A30: Base de Herbrand

Seja L uma linguagem de primeira ordem. A *Base de Herbrand*, BL para L é o conjunto de todos os átomos básicos que podem ser formados usando os predicados de L com os termos básicos do correspondente Universo de Herbrand como argumentos. ■

DEFINIÇÃO A31: Interpretação de Herbrand

Seja L uma linguagem de primeira ordem. Uma interpretação sobre L é uma *Interpretação de Herbrand*, se as seguintes condições forem satisfeitas:

- (i) O domínio da interpretação é o Universo de Herbrand, UL ;
- (ii) As constantes em L são atribuídas a si próprias em UL ;
- (iii) Se f é uma função n -ária em L , então a f é atribuído o mapeamento de $(UL)^n$ em UL definido por $(t_1, \dots, t_n) \rightarrow f(t_1, \dots, t_n)$. ■

Nenhuma restrição é feita sobre a atribuição de predicados em L de forma que diferentes interpretações de Herbrand surgem quando se emprega diferentes atribuições sobre eles. Uma vez que, para as interpretações de Herbrand, as atribuições de constantes e funções é fixa, é possível identificar uma interpretação de Herbrand como um subconjunto da Base de Herbrand. Para toda interpretação de Herbrand, o correspondente subconjunto da Base de Herbrand é o conjunto de todos os átomos básicos que são verdadeiros com respeito a essa interpretação. Inversamente, dado um subconjunto arbitrário da Base de Herbrand, há uma interpretação de Herbrand que a ele corresponde.

DEFINIÇÃO A32: Modelo de Herbrand

Seja L uma linguagem de primeira ordem e S um conjunto de fórmulas fechadas de L . Um *Modelo de Herbrand* para S é uma interpretação de Herbrand que é um modelo para S . ■

PROPOSIÇÃO A.2

Seja S um conjunto de cláusulas e suponha que S tem um modelo. Então S tem um modelo de Herbrand.

Prova:

Seja I uma interpretação de S . Uma interpretação de Herbrand de S , I_H , é definida por:

$$I_H = \{p(t_1, \dots, t_n) \in S \mid p(t_1, \dots, t_n) \text{ é V c.r.a } I\}$$

Segue diretamente que se I é um modelo para S , então I_H também é. ■

PROPOSIÇÃO A.3

Seja S um conjunto de cláusulas. Então S é insatisfatível se e somente se S não possui um modelo de Herbrand.

Prova:

Se S é satisfatível, então a Proposição A.2 demonstra que S tem um modelo de Herbrand. ■

A.2.2 SUBSTITUIÇÕES RESPOSTA

Conforme foi anteriormente estabelecido, o propósito principal de um sistema de programação em lógica é a computação de ligações. Na presente seção será introduzido o conceito de *substituição resposta correta*, que permite um entendimento declarativo da saída desejada de um programa e um objetivo.

DEFINIÇÃO A33: Substituição

Uma substituição θ é um conjunto finito da forma $\{v_1/t_1, \dots, v_n/t_n\}$, onde cada v_i é uma variável e cada t_i é um termo distinto de v_i . Além disso, as variáveis v_1, \dots, v_n devem ser distintas. Cada elemento v_i/t_i é denominado uma *ligação* para v_i . Se os t_i são todos básicos, então θ é denominada uma *substituição básica*. Se os t_i são todos variáveis, então θ é denominada uma *substituição variável pura*. ■

DEFINIÇÃO A34: Expressão

Uma expressão é um termo, um literal ou uma conjunção ou disjunção de literais. Uma *expressão simples* é um termo ou um átomo. ■

DEFINIÇÃO A35: Instância de uma Expressão

Seja $\theta = \{v_1/t_1, \dots, v_n/t_n\}$ uma substituição e E uma expressão. Então $E\theta$, a *instância de E pela substituição θ* , é a expressão obtida a partir de E através da substituição simultânea de todas as ocorrências da variável v_i em E , pelo termo t_i , para $i = 1, \dots, n$. Se $E\theta$ é básica, então $E\theta$ é denominada uma *instância básica* de E . Se $S = \{E_1, \dots, E_n\}$ é um conjunto finito de expressões e θ é uma substituição, então $S\theta$ denota o conjunto $\{E_1\theta, \dots, E_n\theta\}$. ■

DEFINIÇÃO A36: Composição de Substituições

Sejam $\theta = \{u_1/s_1, \dots, u_m/s_m\}$ e $\sigma = \{v_1/t_1, \dots, v_n/t_n\}$ duas substituições. Então a composição $\theta\sigma$ é a substituição obtida do conjunto $\{u_1/s_1\sigma, \dots, u_m/s_m\sigma, v_1/t_1, \dots, v_n/t_n\}$, retirando-se dele todas as ligações $u_i/s_i\sigma$ para as quais $u_i = s_i\sigma$ e todas as ligações v_j/t_j para as quais $v_j \in \{u_1, \dots, u_m\}$. ■

DEFINIÇÃO A37: Substituição Identidade

Substituição Identidade é a substituição dada pelo conjunto vazio. Denota-se a substituição identidade

por ε . Note que $E\varepsilon = E$ para todas as expressões E . ■

PROPOSIÇÃO A.4

Sejam θ , σ e γ substituições e ε a substituição identidade. Então:

- (i) $\theta\varepsilon = \varepsilon\theta = \theta$
- (ii) $\forall E (E\theta)\sigma = E(\theta\sigma)$
- (iii) $(\theta\sigma)\gamma = \theta(\sigma\gamma)$

Prova:

- (i) Segue diretamente da definição de ε .
- (ii) É suficiente provar o resultado quando E é uma variável, digamos X .

Seja $\theta = \{u_1/s_1, \dots, u_m/s_m\}$ e $\sigma = \{v_1/t_1, \dots, v_n/t_n\}$.

Se $X \notin \{u_1, \dots, u_m\} \cup \{v_1, \dots, v_n\}$,
então $(X\theta)\sigma = X(\theta\sigma) = X$.

Se $X \in \{u_1, \dots, u_m\}$, digamos $X = u_i$,
então $(X\theta)\sigma = s_i\sigma = X(\theta\sigma)$.

Se $X \in \{v_1, \dots, v_n\} \setminus \{u_1, \dots, u_m\}$, digamos $X = v_j$,
então $(X\theta)\sigma = t_j = X(\theta\sigma)$.

- (iii) É suficiente mostrar que, se X é uma variável, então $X((\theta\sigma)\gamma) = X(\theta(\sigma\gamma))$.

De fato, $X((\theta\sigma)\gamma) = ((X\theta)\sigma)\gamma = (X\theta)(\sigma\gamma) = X(\theta(\sigma\gamma))$, em função de (ii). ■

DEFINIÇÃO A38: Variantes

Sejam E e F expressões. Diz-se que E e F são variantes se existem as substituições θ e σ tais que $E = F\theta$ e $F = E\sigma$. Diz-se também que E é variante de F ou que F é variante de E . ■

DEFINIÇÃO A39: Renomeação

Seja E uma expressão e V o conjunto das variáveis que ocorrem em E . Uma renomeação para E é uma substituição variável pura $\{X_1/Y_1, \dots, X_n/Y_n\}$ tal que $\{X_1, \dots, X_n\} \subseteq V$, os Y_i são distintos e $(V \setminus \{X_1, \dots, X_n\}) \cap \{Y_1, \dots, Y_n\} = \emptyset$. ■

PROPOSIÇÃO A.5

Sejam E e F expressões variantes. Então existem as substituições θ e s tais que $E = F\theta$ e $F = E\sigma$, onde θ é uma renomeação para F e σ é uma renomeação para E .

Prova:

Uma vez que E e F são variantes, existem as substituições θ_1 e σ_1 tais que $E = F\theta_1$ e $F = E\sigma_1$. Seja V o conjunto das variáveis que ocorrem em E e seja σ a substituição obtida de σ_1 através da remoção de todas as ligações da forma X/t , onde $X \notin V$. Claramente então $F = E\sigma$. Além disso, $E = F\theta_1 = E\sigma\theta_1$, de onde segue que s deve ser uma renomeação para E . ■

Estaremos interessados principalmente nas substituições que *unificam* um conjunto de expressões, isto é, que tornam as expressões contidas em um conjunto sintaticamente idênticas. O conceito de unificação remonta aos estudos de Herbrand em 1930, tendo sido empregado por Robinson [Rob 65] no estabelecimento do princípio da Resolução. O foco do presente texto se restringirá a conjuntos finitos (não-vazios) de expressões simples (termos ou átomos).

DEFINIÇÃO A40: Unificador

Seja S um conjunto finito de expressões simples. Uma substituição θ é dita ser um *unificador* para S se $S\theta$ é única. Um unificador θ é dito ser um *unificador mais geral* (umg) para S se, para todo unificador s de S há uma substituição γ tal que $\sigma = \theta\gamma$. ■

Segue da definição de umg que se σ e θ são ambos umg's de $\{E_1, \dots, E_n\}$, então $E_i\theta$ é variante de $E_i\sigma$. A Proposição A.5 garante então que $E_i\theta$ pode ser obtida de $E_i\sigma$ por simples renomeação de variáveis.

DEFINIÇÃO A41: Conjunto de Desacordo

Seja S um conjunto finito de expressões simples. O *conjunto de desacordo* de S é definido da seguinte maneira: Localizamos a posição do símbolo mais à esquerda que não é o mesmo para todas as expressões de S e extraímos de cada uma delas a sub-expressão que inicia com tal símbolo. O conjunto de todas as sub-expressões assim retiradas é o conjunto de desacordo de S . ■

ALGORITMO DA UNIFICAÇÃO

- (i) Faça $k = 0$ e $\sigma_k = \varepsilon$,
- (ii) Se $S\sigma_k$ é único, então pare: σ_k é um umg de S ,
senão encontre o conjunto de desacordo D_k de $S\sigma_k$;
- (iii) Se existem V e t em D_k tais que V é uma variável que não ocorre em t ,
então faça $\sigma_{k+1} = \sigma_k\{V/t\}$, incremente o valor de k e volte ao passo (ii),
senão pare: S não é unificável. ■

Na forma apresentada acima, o algoritmo da unificação é não-determinístico, uma vez que podem ser consideradas diversas escolhas para V no passo (iii), entretanto a aplicação de quaisquer dois umg's produzidos pelo algoritmo irá conduzir a expressões que diferem entre si somente pelo nome das variáveis envolvidas. Deve ficar claro também que o algoritmo sempre termina, uma vez que S contém um conjunto finito de variáveis e cada aplicação do passo (iii) elimina uma delas. Ainda devemos considerar que no passo (iii) uma verificação é feita para garantir que V não ocorre em t . Tal verificação é denominada *verificação de ocorrência* (occurs check).

TEOREMA A.1 (TEOREMA DA UNIFICAÇÃO)

- (a) S é um conjunto unificável de expressões simples se e somente se o Algoritmo da Unificação termina, retornando um umg para S .
- (b) S não é um conjunto unificável de expressões simples se e somente se o Algoritmo da Unificação termina, retornando a resposta "não". ■

DEFINIÇÃO A42: Substituições Resposta

Seja P um programa e G um objetivo. Uma *substituição resposta* para $P \cup \{G\}$ é uma substituição para as variáveis de G . ■

Entende-se que tal substituição não precisa necessariamente conter uma ligação para cada uma das variáveis em G . Em particular, se G não contém variáveis, a única substituição possível é a substituição identidade.

DEFINIÇÃO A43: Substituição Resposta Correta

Seja P um programa, G um objetivo $\leftarrow A_1, \dots, A_k$ e θ uma substituição resposta para $P \cup \{G\}$. Dizemos que θ é uma *substituição resposta correta* para $P \cup \{G\}$ se $\forall((A_1 \wedge \dots \wedge A_k)\theta)$ é consequência lógica de P . ■

A partir da Proposição A.1 pode-se afirmar que θ é uma substituição resposta correta se e somente se $P \cup \{\neg \forall((A_1 \wedge \dots \wedge A_k)\theta)\}$ for insatisfável. Esta definição de substituição resposta correta captura o sentido intuitivo de "resposta correta". Da mesma forma que fornece substituições respostas, um sistema de programação em lógica pode também retornar com a resposta "não". Dizemos que a resposta "não" é correta, se $P \cup \{G\}$ for satisfável.

A.3 SEMÂNTICA PROVA-TEORÉTICA

A lógica clássica de primeira ordem é definida pela especificação de um esquema de axiomas e regras de inferência. (Para as definições básicas ver a Seção A.1).

AXIOMAS

Para todas as fórmulas bem-formadas A, B e C de uma certa linguagem L da lógica de predicados de primeira ordem:

- (i) $A \rightarrow (B \rightarrow A)$
- (ii) $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$.
- (iii) $(\neg B \rightarrow \neg A) \rightarrow ((\neg B \rightarrow A) \rightarrow B)$.
- (iv) $\forall X A(X) \rightarrow A(t)$, onde t é um termo livre de X em A(X), isto é, nenhuma ocorrência livre de X em A surge no escopo de qualquer quantificador ($\forall X'$), onde X' é uma variável em t.
- (v) $(\forall X) (A \rightarrow B) \rightarrow (A \rightarrow \forall X B)$, onde A não contém nenhuma ocorrência livre de X. ■

REGRAS DE INFERÊNCIA

- (i)
$$\frac{A, A \rightarrow B}{B} \quad [\text{MP - modus ponens}]$$
- (ii)
$$\frac{A}{\forall X(A)} \quad [\text{GEN - generalização}]$$

DEFINIÇÃO A44: Prova

Uma *prova* é qualquer seqüência da forma A_1, \dots, A_n onde cada A_i ou é uma instância de um esquema de axiomas ou deriva dos membros anteriores da seqüência por meio da aplicação de MP ou GEN. ■

DEFINIÇÃO A45: Teorema

Um *teorema* é qualquer fbf que resulte de uma prova, isto é, o último membro de uma seqüência de prova. ■

DEFINIÇÃO A46: Frame de Primeira Ordem

Um *frame de primeira ordem* M para uma linguagem L da lógica de primeira ordem consiste em um domínio não vazio D, juntamente com uma função que atribui a cada símbolo funcional n-ário f uma função f' de $D^n \rightarrow D$ e a cada constante relacional C, um elemento C' de 2^{D^n} . ■

Para estabelecer a semântica da linguagem L com respeito a esse frame, utilizaremos uma função de atribuição g que atribui a cada variável individual um elemento de D. A notação $M(g) \models A$ indica que a função de atribuição g satisfaz a fbf A no frame M.

- (i) $M(g) \models C(t_0, \dots, t_{n-1}) \leftrightarrow (V(t_0, g), \dots, V(t_{n-1}, g)) \in C'$
onde $V(t, g) = g(t)$ se t é uma variável individual e em $f'(V(t_0', \dots, V(t_{m-1}', g)))$ os t_i são da forma $f(t_0', \dots, t_{m-1}')$.
- (ii) $M(g) \models \neg A \leftrightarrow M(g) \not\models A$.
- (iii) $M(g) \models A \wedge B \leftrightarrow M(g) \models A \text{ e } M(g) \models B$.
- (iv) $M(g) \models \forall X A \leftrightarrow M(g\{d/X\}) \models A$,
onde $g\{d/X\}$ é uma função de atribuição idêntica a g, exceto para a variável X, à qual é atribuído o valor d.

As condições de verdade para os demais conectivos podem ser estabelecidas a partir das seguintes equivalências:

$$(v) \quad A \vee B \leftrightarrow \neg(\neg A \wedge \neg B)$$

$$(vi) \quad A \rightarrow B \leftrightarrow \neg A \vee B$$

$$(vii) \quad A \leftrightarrow B \leftrightarrow (A \rightarrow B) \wedge (B \rightarrow A)$$

$$(viii) \quad \exists X A \leftrightarrow \neg \forall X \neg A$$

DEFINIÇÃO A47: Fórmula Universalmente Válida

Uma fbf A é dita ser *universalmente válida* se e somente se, para todo frame M e para toda função de atribuição g , $M(g) \models A$. ■

TEOREMA A.2: Completeza do Cálculo de Predicados

Uma fbf do cálculo de predicados de primeira ordem é um teorema se e somente se é universalmente válida. ■

BIBLIOGRAFIA

- [AMB 87] AMBLE, T.: **Logic Programming and Knowledge Engineering**. Reading: Addison-Wesley, 1987, 348p.
- [AND 93] ANDREWS, J.: **Prolog Frequently Asked Questions**. E-Text (Internet) by jamie@cs.sfu.ca. Stanford University, 1993.
- [ARI 86] ARITY Corporation, **The Arity Prolog Programming Manual**, Arity Corporation, 1986.
- [BOW 82] BOWEN, K.A.; KOWALSKI, R.A.: **Amalgamating Language and Metalanguage in Logic Programming**. In: LOGIC PROGRAMMING. London: Academic Press, 1982. 366p. p.153-172.
- [BOW 85] BOWEN, K.A.: **Meta Level Programming and Knowledge Representation**. New Generation Computing, Tokyo, v.3 n.12, p.359-383, Oct. 1985.
- [BOW 86] BOWEN, K.A.: **Meta Level Techniques in Logic Programming**. In: INTERNATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE AND ITS APPLICATIONS, 1986, Singapore. Proceedings ... Amsterdam: North-Holland, 1986. p.262-271.
- [BRA 86] BRATKO, I.: **Prolog Programming for Artificial Intelligence**. Englewood Cliffs: Addison-Wesley, 1986. 423p.
- [BRO 86b] BRODIE, M.L.; JARKE, M.: **On Integrating Logic Programming and Databases**. In: EXPERT DATABASE SYSTEMS. Menlo Park: Benjamin/Cummings, 1986. 701p. p.191-208.
- [CAR 88] CARNOTA, R.J.; TESZKIEWICZ, A.D.: **Sistemas Expertos y Representación del Conocimiento**. Buenos Aires: EBAI, 1988.
- [CAS 87] CASANOVA, M.A.; GIORNO, F.A.; FURTADO, A.L.: **Programação em Lógica e a Linguagem Prolog**. São Paulo: Edgard Blücher, 1987. 461p.
- [CER 86] CERRO, L.F.D.: **MOLOG: A System that Extends PROLOG with Modal Logic**. New Generation Computing, Tokyo, v.4, n.1, p.35-50, 1986.
- [CHA 82] CHANDRA, A.K.; HAREL, D.: **Horn Clauses and Fixpoint Query Hierarchy**. In: ACM SYMPOSIUM ON PRINCIPLES OF DATABASE SYSTEMS, March 1982, Los Angeles. Proceedings ... New York: ACM, 1982. 304p. p.158-163.
- [CLA 82] CLARK, K.; TÄRNLUND, S-A.: **Logic Programming**. London: Academic Press, 1982.
- [CLO 84] CLOCKSIN, W.; MELLISH, C.: **Programming in Prolog**, Springer-Verlag, 1984.
- [COE 80] COELHO, H. et al.: **How to Solve it in Prolog**. Lisboa: LNEC, Universidade Nova de Lisboa, 1980.
- [DAH 83] DAHL, V.: **Logic Programming as a Representation of Knowledge**. Computer, Los Alamitos, v.16, n.10, p.106-111, Oct. 1983.
- [DAT 83] DATE, C.J.: **An Introduction to Database Systems**. 3rd. Edition. Reading: Addison-Wesley, 1983. 513p.
- [DOD 90] DODD, T.: **Prolog: A Logical Approach**. New York: Oxford University Press, 1990. 556p.
- [FIS 87] FISCHLER, M.; FIRSCHEIN, O.: **The Eye, The Brain and The Computer**. Reading: Addison-Wesley, 1987. 331p.

- [FUR 84] FURUKAWA, K. et al.: **Mandala: A Logic Based Programming System**. In: INTERNATIONAL CONFERENCE ON FIFTH GENERATION COMPUTER SYSTEMS, 1984, Tokyo. Proceedings ... Amsterdam: North-Holland, 1984. 703p. p.613-622.
- [GAL 78] GALLAIRE, H.; MINKER, J.: **Logic and Databases**. New York: Plenum Press, 1978.
- [GAL 83] GALLAIRE, H.: **Logic Databases vs, Deductive Databases**. In LOGIC PROGRAMMING WORKSHOP '83, 1983, Albufeira, Portugal. Proceedings ... Amsterdam: North-Holland, 1983.
- [GAL 84] GALLAIRE, H.; MINKER, J.; NICOLAS, J.-M.: **Logic and Databases: A Deductive Approach**. Computing Surveys, New York, v.16, n.2, p.153-185, Jun. 1984.
- [GÖD 31] GÖDEL, K.: **Über Formal Unentscheidbare Satze der Principia Mathematica und Verwandter System 1**. Tradução em Ingles em: *From Frege to Gödel: A Sourcebook in Mathematical Logic*. Harvard University Press, Cambridge, Mass.
- [GRE 69] GREEN C.: **Theorem Proving by Resolution as a Basis for Question-Answering Systems**. In: MACHINE INTELLIGENCE, 4. Edimburgh: Edimburgh University Press, 1969. p.183-205.
- [HOF 79] HOFSTADTER, D.: **Gödel, Escher and Bach**. New York: Basic Books, 1979.
- [HOG 84] HOGGER, C.J.: **Introduction to Logic Programming**. London: Academic Press, 1984. 278p.
- [ISR 83] ISRAEL, D.; BERANEK, B.: **The Role of Logic in Knowledge Representation**. Computer, Los Alamitos, v.16, n.10, p.37-41, Oct. 1983.
- [IWA 88] IWANUMA, K.; HARAO, M.: **Knowledge Representation and Inference Based on First-Order Modal Logic**. In: LOGIC PROGRAMMING '88. Proceedings ... Berlin: Springer-Verlag, 1988. p.237-251.
- [JAC 86] JACKSON, P.: **Introduction to Expert Systems**. Reading: Addison-Wesley, 1986. 292p.
- [KAN 93] KANTROWITZ, M.: **Prolog Resource Guide**. E-Text (Internet) by mkant+prolog-guide@cs.cmu.edu. Carnegie-Mellon University, 1993.
- [KIT 84] KITAKAMI, H.S.; MIYACHI, T.; FURUKAWA, K.: **A Methodology for Implementation of a Knowledge Acquisition System**. In: INTERNATIONAL SYMPOSIUM ON LOGIC PROGRAMMING, Feb. 1984, Atlantic City. Proceedings ... New York: ACM, 1984.
- [KOW 74] KOWALSKI, R.A.: **Predicate Logic as a Programming Language**. In: IFIP '74. Proceedings ... Amsterdam: North-Holland, 1974. p.569-574.
- [KOW 75] KOWALSKI, R.A.: **A Proof Procedure Using Conection Graphs**. Journal of ACM, New York, v.22, n.4, p.572-595, Apr. 1975.
- [KOW 78] KOWALSKI, R.A.: **Logic for Data Description**. In: LOGIC AND DATABASES. New York: Plenum Press, 1978.
- [KOW 79a] KOWALSKI, R.A.: **Algorithm = Logic + Control**. Communications of ACM, New York, v.22, n.7, p.424-436, Jul. 1979.
- [KOW 79b] KOWALSKI, R.A.: **Logic for Problem Solving**. New York: Elsevier, 1979. 287p.
- [LID 84] LI, D.: **A Prolog Database System**. Hertfordshire: Research Studies Press, 1984. 207p.
- [LLO 84] LLOYD, J.W.: **Foundations of Logic Programming**. Berlin: Springer-Verlag, 1984. 124p.

- [MAE 88] MAES, P.: **Issues in Computational Reflection**. In: META LEVEL ARCHITECTURES AND REFLECTION. Amsterdam: North-Holland, 1988. 355p. p.21-36.
- [MAT 89] MATTOS, N.M.: **An Approach to Knowledge Basis Management**. Kaiserslautern: University of Kaiserslautern, 1989. PhD Thesis, Department of Computer Science. 255p.
- [MCC 69] McCARTHY, J.; HAYES, P.J.: **Some Philosophical Problems from the Standpoint of Artificial Intelligence**. In: MACHINE INTELLIGENCE, 4. Edimburgh: Edimburgh University Press, 1969. p.463-502.
- [MCC 77] McCARTHY, J.: **Epistemological Problems of Artificial Intelligence**. In: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, 5., Aug. 1977, Cambridge, Massachusetts. Proceedings ... New York: ACM, 1977
- [MCC 80] McCARTHY, J.: **Circumscription: A Form of Non-Monotonic Reasoning**. Artificial Intelligence, v.13, n.1, p.27-39, 1980.
- [MIN 75] MINSKI, M.: **A Framework for Representing Knowledge**. In: THE PSICOLOGY OF COMPUTER VISION. New York: McGraw-Hill, 1975. p.211-280.
- [MIN 82] MINSKI, M.: **Why People Think Computers Can't?** AI Magazine, v.3, n.1, p.2-8, 1982.
- [MON 88] MONTEIRO, L.; PORTO A.: **Contextual Logic Programming**. Lisboa: Departamento de Informática, Universidade Nova de Lisboa, 1988.
- [MOO 84] MOORE, R.C. **A Formal Theory of Knowledge and Action**. In: FORMAL THEORIES OF THE COMMON SENSE WORLD. Norwood: Ablex, 1984. p.319-358.
- [NEW 82] NEWELL, A.: **The Knowledge Level**. Artificial Intelligence v.18, n.1, p.87-127, 1982.
- [NIL 80] NILSSON, N.J.: **Principles of Artificial Intelligence**. Palo Alto: Tioga, 1980.
- [PAL 89] PALAZZO, L.A.M.: **Rhesus: Um Modelo Experimental para Representação de Conhecimento**. Porto Alegre: CPGCC da UFRGS, 1989. 115p.
- [PAL 91] PALAZZO, L.A.M.: **Representação de Conhecimento: Programação em Lógica e o Modelo das Hiperredes**. Porto Alegre: CPGCC da UFRGS, 1991. Dissertação de Mestrado. 291p.
- [PAR 86] PARKER Jr, D.S. et al.: **Logic Programming and Databases**. In: EXPERT DATABASE SYSTEMS. Menlo Park: Benjamin Cummings, 1986. 701p. p.35-48.
- [PEN 83] PENTLAND, A.P.; FISCHLER, M.A.: **A More Rational View of Logic**. AI Magazine, v.4, n.4, Winter, 1983.
- [PER 82] PEREIRA, L.M.: **Logic Control with Logic**. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 1., Sept. 1982, Marseille, France. Proceedings ... Berlin: Springer-Verlag, 1982.
- [PER 88] PERLIS, D.: **Meta in Logic**. In: META LEVEL ARCHITECTURES AND REFLECTION. Amsterdam: North-Holland, 1988. 355p. p.37-50.
- [ROB 65] ROBINSON, J.A.: **A Machine-Oriented Logic Based On The Resolution Principle**. Journal of ACM, New York, v.12, n.1, p.23-41, Jan. 1965.
- [STE 86] STERLING, L.; SHAPIRO, E.: **The Art of Prolog**. Cambridge: MIT Press, 1986. 427p.
- [SAK 86] SAKAKIBARA, I.: **Programming in Modal Logic: An Extension of Prolog Based on Modal Logic**. In: LOGIC PROGRAMMING CONFERENCE, 5., 1986, Tokyo. Proceedings ... Berlin: Springer-Verlag, 1987.
- [SHA 83] SHAPIRO, E.Y.: **Logic Programming with Uncertainties: A Tool for Implementing Rule-Based Systems**. In: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL

INTELLIGENCE, 8., 1983, Karlsruhe. Proceedings ... Los Altos, Calif.: Distributed by W. Kaufmann, 1983. p.529-532.

- [SHA 93] SHAPIRO, E.; WARREN, D.: **The Fifth Generation Project: Personal Perspectives.** Communications of ACM, v.36, n.3, March 1993. p.48-101.
- [STI 85] STICKEL, M.; TYSON, W.: **An Analysis of Consecutively Bounded Depth-First Search With Applications in Automated Deduction.** In: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, 9., 1985, Los Angeles. Proceedings ... Los Altos, Calif.: Distributed by M. Kaufmann, 1985. p.465-471.
- [STE 86] STERLING, L.; SHAPIRO, E.: **The Art of Prolog.** Cambridge: MIT Press, 1986.
- [TAR 75] TÄRNLUND, S.-A.: **An Interpreter for the Programming Language Predicate Logic.** In: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, 4., 1975, Tbilisi. Proceedings ... New York: ACM, 1975. p.601-608.
- [TUR 84] TURNER, R.: **Logics for Artificial Intelligence.** West Sussex: Ellis Horwood, 1984, 121p.