Um exemplo usado comumente para ilustrar a diferença entre sistemas procedurais e não procedurais é a ordenação. Em uma linguagem como C++, a ordenação é feita explicando em um programa C++ todos os detalhes de algum algoritmo de ordenação para um computador que possui um compilador C++. O computador, após traduzir o programa C++ para código de máquina ou algum código intermediário interpretável, segue as instruções e produz a lista ordenada.

Em uma linguagem não procedural, é necessário apenas descrever as características da lista ordenada: é alguma permutação da lista tal que para cada par de elementos adjacentes, um relacionamento se mantém entre os dois elementos. Para descrever isso formalmente, suponha que a lista a ser ordenada é um vetor nomeado com uma faixa de índices de 1...*n*. O conceito de ordenar os elementos de uma lista, chamada `old_list`, e colocá-los em um vetor separado, chamado `new_list`, pode ser expressado como:

sort(old_list, new_list) ⊂ permute(old_list, new_list) ∩ sorted(new_list)
sorted(list) ⊂ ∀j such that 1 ≤ j < n, list(j) ≤ list(j+1)

onde permute é um predicado que retorna verdadeiro se seu segundo parâmetro, um vetor, é uma permutação do primeiro parâmetro, também um vetor.

A partir dessa descrição, o sistema de linguagem não procedural pode produzir a lista ordenada. Isso faz a programação não procedural soar como uma mera produção de especificações de requisitos de software concisas, o que é uma avaliação justa. Infelizmente, porém, não é tão simples. Os programas lógicos que usam apenas resolução encaram sérios problemas de eficiência de execução. Além disso, a melhor forma para uma linguagem lógica ainda não foi determinada, e bons métodos para criar programas em linguagens de programação lógica para grandes problemas ainda não foram desenvolvidos.

16.5 AS ORIGENS DO PROLOG

Conforme dito no Capítulo 2, Alain Colmerauer e Phillippe Roussel na Universidade de Aix-Marseille, com alguma assistência de Robert Kowalski na Universidade de Edimburgo, desenvolveram o projeto fundamental de Prolog. Colmerauer e Roussel estavam interessados em processamento de linguagem natural, e Kowalski, na prova automatizada de teoremas. A colaboração entre a Universidade de Aix-Marseille e a Universidade de Edimburgo continuou até meados dos anos 1970. Desde então, a pesquisa sobre o desenvolvimento e o uso da linguagem progrediu independentemente nesses dois locais, resultando em dois dialetos sintaticamente diferentes de Prolog, dentre outras coisas.

O desenvolvimento do Prolog e outros esforços de pesquisa em programação lógica receberam atenção limitada fora de Edimburgo e Marselha até o anúncio, em 1981, de que o governo japonês estava lançando um grande projeto de pesquisa chamado de Quinta Geração de Sistemas de Computação – Fifth Generation Computing Systems (FGCS; Fuchi, 1981; Moto-oka, 1981). Um dos objetivos primários do projeto era desenvolver máquinas inteligentes, e Prolog foi escolhido como base para esse esforço. O anúncio do FGCS fez surtir um forte interesse súbito em inteligência artificial e em programação lógica nos pesquisadores e governos dos EUA e de diversos países europeus.

Após uma década de esforço, o projeto FGCS foi silenciosamente abandonado. Apesar do imenso potencial que se supunha acerca da programação lógica e do Prolog, poucas coisas significativas foram descobertas. Isso levou ao declínio no interesse e no uso de Prolog, apesar de ele ainda ter suas aplicações e proponentes.

16.6 OS ELEMENTOS BÁSICOS DO PROLOG

Existem agora inúmeros dialetos de Prolog. Esses podem ser agrupados em diversas categorias: as que cresceram a partir do grupo de Marselha, as que vieram a partir do grupo de Edimburgo e alguns dialetos que têm sido desenvolvidos para microcomputadores, como micro-Prolog, descrito por Clark e McCabe (1984). As formas sintáticas desses são de certa forma diferentes. Em vez de tentar descrever a sintaxe de diversos dialetos de Prolog ou algum híbrido entre eles, escolhemos um dialeto específico, amplamente disponível: aquele desenvolvido em Edimburgo, às vezes chamado de **sintaxe de Edimburgo**. Sua primeira implementação foi em um DEC System-10 (Warren et al., 1979). Implementações de Prolog estão disponíveis para praticamente todas as plataformas de computadores populares, por exemplo, a partir da Organização de Software Livre – Free Software Organization (http://www.gnu.org).

16.6.1 Termos

Como programas em outras linguagens, os em Prolog consistem em coleções de sentenças. Existem apenas alguns tipos de sentenças em Prolog, mas elas podem ser complexas. Todas as sentenças em Prolog são construídas a partir de termos.

Um **termo** Prolog é uma constante, uma variável ou uma estrutura. Uma constante é um **átomo** ou um inteiro. Átomos são os valores simbólicos de Prolog e são similares aos seus correspondentes em LISP. Em particular, um átomo é uma cadeia de letras, dígitos e sublinhados que iniciam com uma letra minúscula ou uma cadeia de quaisquer caracteres ASCII delimitados por apóstrofes.

Uma variável é qualquer cadeia de letras, dígitos e sublinhados que iniciam com uma letra maiúscula. Variáveis não são vinculadas a tipos por declarações. A vinculação de um valor a uma variável, e dessa forma a um tipo, é chamada de uma **instanciação**, que ocorre apenas no processo de resolução. Uma variável que ainda não recebeu um valor é chamada de **não instanciada**. As instanciações duram apenas o tempo necessário para satisfazer um objetivo completo, o que envolve a prova ou a falsidade de uma proposição. Variáveis Prolog são apenas parentes distantes, tanto em termos de semântica quanto de uso, das variáveis das linguagens imperativas.

O último tipo de termo é chamado de uma estrutura. Estruturas representam as proposições atômicas do cálculo de predicados e sua forma geral é a mesma:

functor(lista de parâmetros)

O functor é qualquer átomo e é usado para identificar a estrutura. A lista de parâmetros pode ser qualquer lista de átomos, de variáveis ou de outras estruturas. Conforme discutido extensivamente na subseção a seguir, as estruturas são a maneira de especificar fatos em Prolog. Elas podem ser pensadas como objetos, permitindo que os fatos sejam definidos em termos de diversos átomos relacionados. Nesse sentido, as estruturas são relações, já que elas definem relacionamentos entre termos. Uma estrutura também é um predicado quando seu contexto a especifica como uma consulta (pergunta).

16.6.2 Sentenças de fatos

Nossa discussão de sentenças Prolog começa com aquelas sentenças usadas para construir as hipóteses ou base de dados de informações pré-definidas – as sentenças a partir das quais novas informações podem ser inferidas.

Prolog tem duas formas básicas de sentenças; aquelas que correspondem às cláusulas de Horn com cabeça e às sem cabeça do cálculo de predicados. A forma mais simples de cláusulas de Horn sem cabeça em Prolog é uma estrutura única, interpretada como uma asserção incondicional, ou fato. Logicamente, fatos são simplesmente proposições que se assume serem verdadeiras.

Os exemplos a seguir ilustram os tipos de fatos que alguém pode ter em um programa Prolog. Note que cada sentença Prolog é terminada por um ponto.

```
female(shelley).
male(bill).
female(mary).
male(jake).
father(bill, jake).
father(bill, shelley).
mother(mary, jake).
mother(mary, shelley).
```

Essas estruturas simples afirmam certos fatos acerca de `jake`, `shelley`, `bill` e `mary`. Por exemplo, a primeira diz que `shelley` é uma mulher (`female`). As últimas quatro conectam seus dois parâmetros com um relacionamento nomeado no átomo do functor; por exemplo, o significado da quinta proposição pode ser interpretado como `bill` é o pai (`father`) de `jake`. Note que essas proposições Prolog, como aquelas do cálculo de predicados, não têm uma semântica intrínseca. Elas significam qualquer coisa que os programadores queiram que elas signifiquem. Por exemplo, a proposição

```
father(bill, jake).
```

poderia significar que `bill` e `jake` têm o mesmo pai (`father`) ou que `jake` é o pai (`father`) de `bill`. O significado mais comum e mais direto, entretanto, é que `bill` é o pai (`father`) de `jake`.

16.6.3 Sentenças de regras

A outra forma básica de sentenças Prolog para construir a base de dados corresponde a uma cláusula de Horn com cabeça. Essa forma pode ser relacionada com um conhecido teorema na matemática a partir do qual uma conclusão pode ser tirada se o conjunto das condições dadas for satisfeito. O lado direito é o antecedente, ou parte *se*, e o lado esquerdo é o consequente, ou parte *então*. Se o antecedente de uma sentença Prolog é verdadeiro, então o consequente da sentença também deve ser. Como elas são cláusulas de Horn, o consequente de uma sentença Prolog é um termo simples, enquanto o antecedente pode ser um termo simples ou uma conjunção.

Conjunções contêm múltiplos termos separados por operações E lógicas. Em Prolog, a operação E é implicada. As estruturas que especificam proposições atômicas em uma conjunção são separadas por vírgulas, então alguém pode considerar as vírgulas como operadores E. Como um exemplo de uma conjunção, considere:

```
female(shelley), child(shelley).
```

A forma geral da sentença de cláusula de Horn com cabeça em Prolog é

consequente_1 :- expressão_antecedente.

Ela é lida da seguinte forma: "O consequente_1 pode ser concluído se a expressão antecedente for verdadeira ou puder ser tornada verdadeira por alguma instanciação de suas variáveis". Por exemplo,

```
ancestor(mary, shelley) :- mother(mary, shelley).
```

afirma que se `mary` é a mãe (`mother`) de `shelley`, então `mary` é uma ancestral (`ancestor`) de `shelley`. As cláusulas de Horn com cabeça são chamadas de **regras**, porque definem regras de implicação entre proposições.

Assim como as proposições de forma clausal no cálculo de predicados, sentenças Prolog podem usar variáveis para generalizar seu significado.

Lembre-se de que as variáveis em forma clausal fornecem um tipo de quantificador universal implícito. O seguinte demonstra o uso de variáveis em sentenças Prolog:

```
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).
grandparent(X, Z) :- parent(X, Y) , parent(Y, Z).
sibling(X, Y) :- mother(M, X) , mother(M, Y),
                 father(F, X) , father(F, Y).
```

Essas sentenças dão regras de implicação entre algumas variáveis ou objetos universais. Nesse caso, os objetos universais são `X`, `Y`, `Z`, `M` e `F`. A primeira regra diz que se existem instanciações de `X` e `Y` tal que `mother(X, Y)` é verdadeira, então para essas mesmas instanciações de `X` e `Y`, `parent(X, Y)` é verdadeira.

16.6.4 Sentenças de objetivos

Até agora, descrevemos as sentenças Prolog para proposições lógicas, usadas para descrever tanto fatos conhecidos quanto regras que descrevem relacionamentos lógicos entre fatos. Essas sentenças servem de base para o modelo de prova de teoremas. O teorema é na forma de uma proposição que queremos que o sistema prove sua veracidade ou falsidade. Em Prolog, essas proposições são chamadas de **objetivos** ou **consultas**. A forma sintática das sentenças de objetivo em Prolog é idêntica a das cláusulas de Horn sem cabeça. Por exemplo, poderíamos ter

```
man(fred).
```

para o qual o sistema responderia `yes` ou `no`. A resposta `yes` significa que o sistema provou o objetivo como verdadeiro de acordo com a base de dados de fatos e relacionamentos dados. A resposta `no` significa que se determinou o objetivo como falso ou o sistema não foi capaz de prová-lo.

Proposições conjuntivas e proposições com variáveis também são objetivos legais. Quando as variáveis estão presentes, o sistema não apenas avalia a validade do objetivo, mas também identifica as instanciações das variáveis que tornam o objetivo verdadeiro. Por exemplo,

```
father(X, mike).
```

pode ser perguntado. O sistema irá tentar, por unificação, encontrar uma instanciação de `X` que resulta em um valor verdadeiro para o objetivo.

Como sentenças de objetivo e algumas sentenças que não são de objetivo têm a mesma forma (cláusulas de Horn sem cabeça), uma implementação Prolog deve ter algum jeito de distinguir entre as duas. Implementações de Prolog interativas fazem isso por dois modos, indicados por diferentes interpretadores de comandos interativos: um para informar fatos e sentenças de regras e um para informar objetivos. O usuário pode modificar o modo a qualquer momento.

16.6.5 O processo de inferência do Prolog

Esta seção examina a resolução de Prolog. O uso eficiente de Prolog requer que o programador saiba precisamente o que o sistema Prolog faz com o seu programa.

As consultas são chamadas de **objetivos**. Quando um objetivo é uma proposição composta, cada um dos fatos (estruturas) é chamado de **subobjetivo**. Para provar que um objetivo é verdadeiro, o processo de inferência deve encontrar uma cadeia de regras de inferência e/ou fatos na base de dados que conecte o objetivo a um ou mais fatos nessa base. Por exemplo, se Q é o objetivo, então ou Q deve ser encontrado como um fato na base de dados ou o processo de inferência deve encontrar um fato P_1 e uma sequência de proposições P_2, P_3, ..., P_n tal que

P_2 :- P_1
P_3 :- P_2
...
Q :- P_n

É claro, o processo pode ser, e frequentemente é, complicado por regras com lados direitos compostos e regras com variáveis. O processo de encontrar os *Ps*, quando eles existem, é basicamente uma comparação, ou casamento, de termos uns com os outros.

Como o processo de fornecer um subobjetivo é feito por meio de um processo de casamento de proposições, ele é algumas vezes chamado de **casamento**. Em alguns casos, provar um subobjetivo é chamado de **satisfazer** o subobjetivo.

Considere a consulta:

```
man(bob).
```

Essa sentença de objetivo é do tipo mais simples. É relativamente fácil para a resolução determinar se ela é verdadeira ou falsa: o padrão desse objetivo é comparado com os fatos e com as regras na base de dados. Se a base incluir o fato

```
man(bob).
```

a prova é trivial. Se, entretanto, a base de dados contiver o seguinte fato e regra de inferência,

```
father(bob).
man(X) :- father(X).
```

o Prolog precisaria encontrar essas duas sentenças e usá-las para inferir a verdade do objetivo. Isso necessitaria de unificação para instanciar `X` temporariamente para `bob`.

Agora, considere o objetivo

```
man(X).
```

Nesse caso, o Prolog deveria casar o objetivo com as proposições na base de dados. A primeira proposição encontrada com o formato do objetivo, com qualquer objeto como seu parâmetro, fará `X` ser instanciado com o valor desse objeto. `X` então é mostrado como o resultado. Se não existe uma proposição com a forma do objetivo, o sistema indica, dizendo que não (`no`), o objetivo não pode ser satisfeito.

Existem duas abordagens opostas para tentar casar um objetivo com um fato na base de dados. O sistema pode começar com os fatos e regras da base de dados e tentar encontrar uma sequência de casamentos que levem ao objetivo. Essa abordagem é chamada de **resolução ascendente** (*bottom-up*) ou **encadeamento para frente** (*forward chaining*). A alternativa é começar com o objetivo e tentar encontrar uma sequência de proposições que casem com o objetivo que levem a algum conjunto de fatos originais na base de dados. Essa abordagem é chamada de **resolução descendente** (*top-down*) ou **encadeamento para trás** (*backward chaining*). Em geral, o encadeamento para trás funciona bem quando existe um conjunto razoavelmente pequeno de respostas candidatas. A abordagem de encadeamento para frente é melhor quando o número de possíveis respostas corretas é grande; nessa situação o encadeamento para trás iria requerer um número de casamentos muito grande para chegar a uma resposta. As implementações de Prolog usam encadeamento para trás para a resolução, presumivelmente porque os projetistas acreditavam que o encadeamento para trás seria mais adequado para uma classe maior de problemas do que o encadeamento para frente.

O seguinte exemplo ilustra a diferença entre o encadeamento para frente e o para trás. Considere a consulta:

```
man(bob).
```

Assuma que a base de dados contenha

```
father(bob).
man(X) :- father(X).
```

O encadeamento para frente procuraria e encontraria a primeira proposição. O objetivo é então inferido casando a primeira proposição com o lado direito da segunda regra (`father(X)`) pela instanciação de `X` como `bob` e casando o lado esquerdo da segunda proposição para o objetivo. O encadeamento para trás iria primeiro casar o objetivo com o lado esquerdo da segunda proposição (`man(X)`) pela instanciação de `X` para `bob`. Como seu último passo, ele casaria o lado direito da segunda proposição (agora `father(bob)`) com a primeira proposição.

A próxima questão de projeto surge sempre que o objetivo tiver mais de uma estrutura, como nosso exemplo. A questão então é se a busca da solução é feita primeiro em profundidade ou em largura. Uma busca **primeiro em profundidade** (*depth first*) encontra uma sequência completa de proposições – uma prova – para o primeiro subobjetivo antes de trabalhar com os outros. Uma busca **primeiro em largura** (*breadth first*) funciona em todos os subobjetivos de um objetivo em paralelo. Os projetistas de Prolog escolheram a abordagem de busca primeiro em profundidade porque ela pode ser feita com menos recursos computacionais. A abordagem primeiro em largura é uma busca paralela que pode requerer uma grande quantidade de memória.

O último recurso do mecanismo de resolução do Prolog que deve ser discutido é o rastreamento para trás (*backtraking*). Quando um objetivo com múltiplos subobjetivos está sendo processado e o sistema falha em mostrar a verdade de um de seus subobjetivos, os sistema abandona o subobjetivo que não pode provar. Ele então reconsidera o subobjetivo prévio, se existe um, e tenta encontrar uma solução alternativa para ele. Essa volta no objetivo para a reconsideração de um subobjetivo previamente provado é chamada de **rastreamento para trás** (*backtracking*). Uma nova solução é encontrada iniciando-se a busca onde a busca anterior para tal subobjetivo parou. Múltiplas soluções para um subobjetivo resultam de diferentes instanciações de suas variáveis. O rastreamento para trás pode requerer uma boa dose de tempo e espaço porque pode ter de encontrar todas as provas possíveis para cada um dos subobjetivos. Essas provas de cada subobjetivo podem não estar organizadas para minimizar o tempo necessário para encontrar aquele que resultará na prova final completa, o que piora o problema.

Para solidificar o seu entendimento sobre rastreamento para trás, considere o seguinte exemplo. Assuma que existe um conjunto de fatos e regras em uma base de dados e que foi apresentado ao Prolog o seguinte objetivo composto:

```
male(X), parent(X, shelley).
```

Esse objetivo pergunta se existe uma instanciação de `X` tal que `X` é um `homem (male)` e `X` é um dos pais (`parent`) de `shelley`. Como seu primeiro passo, o Prolog encontra o primeiro fato na base de dados com homem (`male`) como seu *functor*. Ele então instancia `X` para o parâmetro do fato encontrado, `mike` digamos. Ele então tenta provar que `parent(mike, shelley)` é verdadeiro. Se for falso, ele volta para o primeiro subobjetivo, `male(X)`, e tenta ressatisfazê-lo com alguma instanciação alternativa de `X`. O processo de resolução pode ter de encontrar cada homem na base de dados antes de encontrar aquele que é um dos pais de `shelley`. Ele certamente precisa encontrar todos os homens para provar que o objetivo não pode ser satisfeito. Note que o objetivo do nosso exemplo poderia ser processado mais eficientemente se a ordem dos dois subobjetivos fosse a inversa. Então, apenas após a resolução encontrar um dos pais de `shelley` ela tentará casar essa pessoa com o subobjetivo homem (`male`).

Seria mais eficiente se `shelley` tivesse menos pais do que homens existentes na base de dados, o que parece ser uma ideia razoável. A Seção 16.7.1 discute um método para limitar o rastreamento para trás feito por um sistema Prolog.

As buscas em bases de dados em Prolog sempre procedem na direção da primeira para a última.

As duas subseções a seguir descrevem exemplos em Prolog que ilustram mais sobre o processo de resolução.

16.6.6 Aritmética simples

O Prolog suporta variáveis inteiras e aritmética de inteiros. Originalmente, os operadores aritméticos eram *functores*, então a soma de `7` com a variável `X` era formada com

```
+(7, X)
```

O Prolog agora permite uma sintaxe mais abreviada para aritmética com o operador **`is`**. Esse operador recebe uma expressão aritmética como seu operando direito e uma variável como seu operando esquerdo. Todas as variáveis na expressão já devem estar instanciadas, mas a variável do lado esquerdo não pode ter sido instanciada anteriormente. Por exemplo, em

```
A is B / 17 + C.
```

se `B` e `C` são instanciadas, mas `A` não é, essa cláusula fará `A` ser instanciada com o valor da expressão. Quando isso acontece, a cláusula é satisfeita. Se ou `B` ou `C` não for instanciada ou `A` estiver instanciada, a cláusula não é satisfeita e nenhuma instanciação de `A` pode ocorrer. A semântica de uma proposição **`is`** é consideravelmente diferente de uma sentença de atribuição em uma linguagem imperativa. Essa diferença pode levar a um cenário interessante. Como o operador **`is`** torna a cláusula na qual ele aparece algo parecido com uma sentença de atribuição, um programador iniciante em Prolog pode ficar tentado escrever uma sentença como

```
Sum is Sum + Number.
```

que nunca é útil, nem mesmo permitida, em Prolog. Se `Sum` não está instanciada, a referência a ela no lado direito é indefinida e a cláusula falha. Se `Sum` já está instanciada, a cláusula falha, porque o operando esquerdo não pode ter uma instanciação atual quando `is` for avaliada. Em ambos os casos, a instanciação de `Sum` para o novo valor não ocorrerá. (Se o valor de `Sum + Number` for requerido, ele pode ser vinculado para algum novo nome).

Prolog não tem sentenças de atribuição no mesmo sentido das linguagens imperativas. Elas simplesmente não são necessárias para a maioria da programação para a qual o Prolog foi projetado. A utilidade das sentenças de atribuição em linguagens imperativas depende da capacidade do programador em controlar o fluxo de controle de execução do código no qual a sentença

de atribuição está inserida. Como esse tipo de controle nem sempre é possível em Prolog, tais sentenças são muito menos úteis.

Como um simples exemplo do uso de computação numérica em Prolog, considere o seguinte problema: suponha que soubéssemos as velocidades médias de diversos automóveis em uma pista de corrida e o tempo em que eles estão na pista. Essa informação básica pode ser codificada como fatos, e o relacionamento entre velocidade, tempo e distância pode ser escrito como uma regra:

```
speed(ford, 100).
speed(chevy, 105).
speed(dodge, 95).
speed(volvo, 80).
time(ford, 20).
time(chevy, 21).
time(dodge, 24).
time(volvo, 24).
distance(X, Y) :- speed(X, Speed),
                  time(X, Time),
                  Y is Speed * Time.
```

Agora, consultas podem solicitar a distância viajada por um carro em particular. Por exemplo, a consulta

```
distance(chevy, Chevy_Distance).
```

instancia `Chevy_Distance` com o valor `2205`. As primeiras duas cláusulas no lado direito da sentença de computação da distância instanciam as variáveis `Speed` e `Time` com os valores correspondentes do *functor* do automóvel dado. Após satisfazer o objetivo, o Prolog também mostra o nome `Chevy_Distance` e o seu valor.

Nesse ponto, é instrutivo ter uma visão operacional de como um sistema Prolog produz resultados. O Prolog tem uma estrutura pré-definida chamada `trace` que mostra as instanciações de valores a variáveis em cada um dos passos durante a tentativa de satisfazer um objetivo. Essa estrutura é usada para entender e depurar programas Prolog. Para entender `trace`, é melhor introduzir um modelo diferente de execução de programas Prolog, chamado de **modelo de rastreamento**.

O modelo de rastreamento descreve a execução de Prolog em termos de quatro eventos: (1) chamar, que ocorre no início de uma tentativa de satisfazer um objetivo; (2) sair, quando um objetivo foi satisfeito, (3) refazer, quando um retorno faz com que seja feita uma tentativa de ressatisfazer um objetivo, e (4) falhar, quando um objetivo falha. A chamada e a saída podem ser relacionadas diretamente ao modelo de execução de um subprograma em uma linguagem imperativa se processos como `distance` são pensados como subprogramas. Os outros dois eventos são únicos aos sistemas de programação lógica. No seguinte exemplo de rastreamento, um rastreamento da computação do valor para `Chevy_Distance`, o objetivo não requer nenhum evento refazer ou falhar:

```
trace.
distance(chevy, Chevy_Distance).

(1) 1 Call: distance(chevy, _0)?
(2) 2 Call: speed(chevy, _5)?
(2) 2 Exit: speed(chevy, 105)
(3) 2 Call: time(chevy, _6)?
(3) 2 Exit: time(chevy, 21)
(4) 2 Call: _0 is 105*21?
(4) 2 Exit: 2205 is 105*21
(1) 1 Exit: distance(chevy, 2205)

Chevy_Distance = 2205
```

Símbolos no rastreamento que iniciam com o caractere sublinhado (_) são variáveis internas usadas para valores instanciados. A primeira coluna do rastreamento indica o subobjetivo para o qual um casamento está sendo tentado. No rastreamento de exemplo, a primeira linha com a indicação `(3)` é uma tentativa de instanciar a variável temporária `_6` com um valor de tempo (`time`) para `chevy`, onde o tempo (`time`) é o segundo termo no lado direito da sentença que descreve a computação da distância (`distance`). A segunda coluna indica a profundidade da chamada do processo de casamento. A terceira coluna indica a ação atual.

Para ilustrar o rastreamento para trás, considere a seguinte base de dados de exemplo e o objetivo composto rastreado:

```
likes(jake, chocolate).
likes(jake, apricots).
likes(darcie, licorice).
likes(darcie, apricots).

trace.
likes(jake, X), likes(darcie, X).
(1) 1 Call: likes(jake, _0)?
(1) 1 Exit: likes(jake, chocolate)
(2) 1 Call: likes(darcie, chocolate)?
(2) 1 Fail: likes(darcie, chocolate)
(1) 1 Redo: likes(jake, _0)?
(1) 1 Exit: likes(jake, apricots)
(3) 1 Call: likes(darcie, apricots)?
(3) 1 Exit: likes(darcie, apricots)

X = apricots
```

Alguém pode pensar acerca de computações Prolog graficamente como segue: considere cada objetivo como uma caixa com quatro portas – chamar, falhar, sair, refazer. O controle entra em um objetivo na direção para frente por sua porta chamar. O controle também pode entrar em um objetivo a partir da direção inversa por sua porta refazer. O controle também pode deixar um objetivo de duas maneiras: se o objetivo for bem-sucedido, o controle deixa o objetivo pela porta sair; se o objetivo falhou,

o controle deixa o objetivo pela porta falhar. Um modelo do exemplo é mostrado na Figura 16.1. Nesse exemplo, o controle flui por cada subobjetivo duas vezes. O segundo subobjetivo falha a primeira vez, o que força um retorno pela porta refazer para o primeiro subobjetivo.

16.6.7 Estruturas de listas

Até agora, a única estrutura de dados Prolog que discutimos foi a proposição atômica, que se parece mais com uma chamada a função do que com uma estrutura de dados. Proposições atômicas, também chamadas de estruturas, são na verdade uma forma de registros. A outra estrutura de dados básica suportada é a lista, similar a estrutura de lista usada por LISP. Listas são sequências de qualquer número de elementos, onde os elementos podem ser átomos, proposições atômicas ou quaisquer outros termos, incluindo outras listas.

Prolog usa a sintaxe de ML e Haskell para especificar listas. Os elementos de lista são separados por vírgulas, e a lista inteira é delimitada por colchetes, como em

```
[apple, prune, grape, kumquat]
```

A notação `[]` é usada para denotar a lista vazia. Em vez de ter funções explícitas para construir e manipular listas, Prolog simplesmente usa uma notação especial. `[X | Y]` denota uma lista com cabeça `X` e cauda `Y`, onde a cabeça e a cauda correspondem a `CAR` e `CDR` em LISP. Isso é similar à notação usada em ML e Haskell.

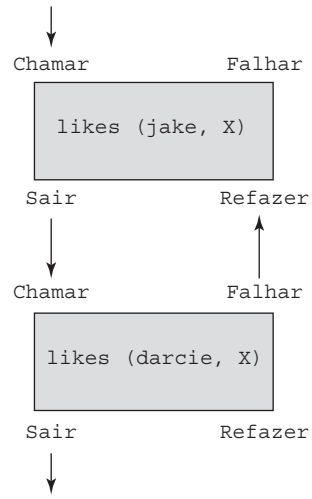

Figura 16.1 O Modelo de fluxo de controle para o objetivo `likes (jake, X), likes (darci, X).`

Uma lista pode ser criada com uma estrutura simples, como em

```
new_list([apple, prune, grape, kumquat]).
```

que diz que a lista constante `[apple, prune, grape, kumquat]` é um novo elemento da relação chamada `new_list` (um nome que inventamos). Essa sentença não vincula a lista a uma variável chamada `new_list`; em vez disso, ela faz o tipo de coisa que a proposição

```
male(jake)
```

faz. Isso é, ela diz que `[apple, prune, grape, kumquat]` é um novo elemento de `new_list`. Logo, poderíamos ter uma segunda proposição com um argumento de lista, como

```
new_list([apricot, peach, pear])
```

No modo de consulta, um dos elementos de `new_list` pode ser separado em cabeça e cauda com

```
new_list([New_List_Head | New_List_Tail]).
```

Se `new_list` foi configurada para ter os dois elementos mostrados, essa sentença instancia `New_List_Head` com a cabeça do primeiro elemento da lista (nesse caso `apple`) e `New_List_Tail` com a cauda da lista (ou seja `[prune, grape, kumquat]`). Se isso fizesse parte de um objetivo composto e um rastreamento para trás forçasse uma nova avaliação dele, `New_List_Head` e `New_List_Tail` seriam reinstanciadas para `apricot` e `[peach, pear]`, respectivamente, porque `[apricot,peach,pear]` é o próximo elemento de `new_list`.

O operador | usado para manipular listas também pode ser usado para criar listas a partir de componentes cabeça e cauda que foram instanciados e fornecidos, como em

```
[Element_1 | List_2]
```

Se `Element_1` tivesse sido instanciado com `pickle` e `List_2` tivesse sido instanciada com `[peanut, prune, popcorn]`, a mesma notação criaria, para essa referência, a lista `[pickle, peanut, prune, popcorn]`.

Conforme falamos anteriormente, a notação de lista que inclui o símbolo | é universal: ela pode especificar uma construção de lista ou uma manipulação de lista. Note que as seguintes sentenças são equivalentes:

```
[apricot, peach, pear | []]
[apricot, peach | [pear]]
[apricot | [peach, pear]]
```

Com listas, certas operações básicas são geralmente necessárias, como aquelas encontradas em LISP, ML e Haskell. Como um exemplo de tais ope-

rações em Prolog, examinamos uma definição de `append`, relacionada a tal função em LISP. Nesse exemplo, as diferenças e similaridades entre linguagens funcionais e declarativas podem ser vistas. Não precisamos especificar como o Prolog deve construir uma nova lista a partir de listas dadas; em vez disso, precisamos especificar apenas as características de uma nova lista em termos de listas dadas.

Na aparência, a definição de Prolog de `append` é bastante similar à versão de ML que aparece no Capítulo 15, e um tipo de recursão na resolução é usado de uma maneira similar para produzir a nova lista. No caso do Prolog, a recursão é causada e controlada pelo processo de resolução. Assim como com ML e Haskell, um processo de casamento de padrões é usado para escolher, baseado no parâmetro real, entre duas definições diferentes do processo de inserção.

Os primeiros dois parâmetros para a operação `append` no código a seguir são as duas listas a serem concatenadas, e o terceiro parâmetro é a lista resultante:

```
append([], List, List).
append([Head | List_1], List_2, [Head | List_3]) :-
            append(List_1, List_2, List_3).
```

A primeira proposição especifica que quando a lista vazia é inserida no final de qualquer outra lista, essa outra é o resultado. Essa sentença corresponde ao passo de término de recursão da função `append` de ML. Note que a proposição de término é colocada antes da proposição de recursão. Isso é feito porque sabemos que o Prolog casará as duas proposições em ordem, começando com a primeira (por causa de seu uso de ordem primeiro em profundidade).

A segunda proposição especifica diversas características da nova lista. Ela corresponde ao passo de recursão na função de ML. O predicado do lado esquerdo diz que o primeiro elemento da nova lista é o mesmo que o primeiro elemento da primeira lista, porque ambos são chamados `Head`. Sempre que `Head` for instanciado para um valor, todas as ocorrências de `Head` no objetivo são, na prática, simultaneamente instanciadas para esse valor. O lado direito da segunda sentença especifica que a cauda da primeira lista dada (`List_1`) possui a segunda lista (`List_2`), anexada a ela para formar a cauda (`List_3`) da lista resultante.

Uma maneira de ler a segunda sentença de `append` é: anexar a lista `[Head | List_1]` em qualquer lista `List_2` produz a lista `[Head | List_3]`, mas apenas se a lista `List_3` for formada pela junção de `List_1` com `List_2`. Em LISP, isso poderia ser

```
(CONS (CAR FIRST) (APPEND (CDR FIRST) SECOND))
```

Nas versões em Prolog e LISP, a lista resultante não é construída até a recursão produzir a condição de término; nesse caso, a primeira lista deve se tornar vazia. Então, a lista resultante é criada usando a função `append` propriamente

dita; os elementos retirados da primeira lista são adicionados, na ordem inversa, na segunda lista. A inversão é feita pela recursão.

Para ilustrar como o processo `append` progride, considere o seguinte exemplo rastreado:

```
trace.
append([bob, jo], [jake, darcie], Family).

(1) 1 Call: append([bob, jo], [jake, darcie], _10)?
(2) 2 Call: append([jo], [jake, darcie], _18)?
(3) 3 Call: append([], [jake, darcie], _25)?
(3) 3 Exit: append([], [jake, darcie], [jake, darcie])
(2) 2 Exit: append([jo], [jake, darcie], [jo, jake,
                    darcie])
(1) 1 Exit: append([bob, jo], [jake, darcie],
                   [bob, jo, jake, darcie])
Family = [bob, jo, jake, darcie]
yes
```

As primeiras duas chamadas, que representam subobjetivos, possuem `List_1` não vazia, então elas criam chamadas recursivas a partir do lado direito da segunda sentença. O lado esquerdo da segunda sentença efetivamente especifica os argumentos para as chamadas recursivas, ou objetivos, separando a primeira lista um elemento por passo. Quando a primeira lista fica vazia, em uma chamada, ou subobjetivo, a instância atual do lado direito da segunda sentença ocorre casando com a primeira. O efeito disso é retornar como o terceiro parâmetro o valor da lista vazia adicionado à lista original passada como segundo parâmetro. Em saídas sucessivas, as quais representam casamentos bem-sucedidos, os elementos que foram removidos da primeira lista são anexados à lista resultante, `Family`. Quando a saída do primeiro objetivo é realizada, o processo está completo e a lista resultante é mostrada.

As proposições `append` também podem ser usadas para criar outras operações de lista, como a seguinte, cujo efeito convidamos o leitor a determinar. Note que `list_op_2` deve ser usada fornecendo-se uma lista como seu primeiro parâmetro e uma variável como o segundo, e o resultado de `list_op_2` é o valor para o qual o segundo parâmetro é instanciado.

```
list_op_2([], []).
list_op_2([Head | Tail], List) :-
list_op_2(Tail, Result), append(Result, [Head], List).
```

Como você deve ter sido capaz de determinar, `list_op_2` faz o sistema Prolog instanciar seu segundo parâmetro com uma lista que tem os elementos da lista do primeiro parâmetro, mas na ordem inversa. Por exemplo, `([apple, orange, grape], Q)` instancia `Q` com a lista `[grape, orange, apple]`.

Mais uma vez, apesar de as linguagens LISP e Prolog serem fundamentalmente diferentes, operações similares podem usar abordagens similares. No caso da operação de inversão, tanto `list_op_2` de Prolog quanto a função `reverse` de LISP incluem a condição de término da recursão, com o processo básico de inserir o inverso do `CDR` ou cauda da lista ao `CAR` ou cabeça da lista para criar a lista resultante.

A seguir, temos um rastreamento desse processo, agora chamado `reverse`:

```
trace.
reverse([a, b, c], Q).

(1) 1 Call: reverse([a, b, c], _6)?
(2) 2 Call: reverse([b, c], _65636)?
(3) 3 Call: reverse([c], _65646)?
(4) 4 Call: reverse([], _65656)?
(4) 4 Exit: reverse([], [])
(5) 4 Call: append([], [c], _65646)?
(5) 4 Exit: append([], [c], [c])
(3) 3 Exit: reverse([c], [c])
(6) 3 Call: append([c], [b], _65636)?
(7) 4 Call: append([], [b], _25)?
(7) 4 Exit: append([], [b], [b])
(6) 3 Exit: append([c], [b], [c, b])
(2) 2 Exit: reverse([b, c], [c, b])
(8) 2 Call: append([c, b], [a], _6)?
(9) 3 Call: append([b], [a], _32)?
(10) 4 Call: append([], [a], _39)?
(10) 4 Exit: append([], [a], [a])
(9) 3 Exit: append([b], [a], [b, a])
(8) 2 Exit: append([c, b], [a], [c, b, a])
(1) 1 Exit: reverse([a, b, c], [c, b, a])

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

Suponha que precisássemos ser capazes de determinar se um símbolo está em uma lista. Uma descrição direta em Prolog disso seria

```
member(Element, [Element | _]).
member(Element, [_ | List]) :- member(Element, List).
```

O sublinhado indica uma variável "anônima", usada para significar que não nos importamos que instanciação ela poderia obter com a unificação. A primeira sentença no exemplo anterior era bem-sucedida se `Element` fosse a cabeça da lista, seja inicialmente ou após diversas recursões por meio do segundo elemento. A segunda sentença era bem-sucedida se `Element` estivesse na cauda da lista. Considere os exemplos rastreados:

```
trace.
member(a, [b, c, d]).
(1) 1 Call: member(a, [b, c, d])?
```

```
(2) 2 Call: member(a, [c, d])?
(3) 3 Call: member(a, [d])?
(4) 4 Call: member(a, [])?
(4) 4 Fail: member(a, [])
(3) 3 Fail: member(a, [d])
(2) 2 Fail: member(a, [c, d])
(1) 1 Fail: member(a, [b, c, d])
no

member(a, [b, a, c]).
(1) 1 Call: member(a, [b, a, c])?
(2) 2 Call: member(a, [a, c])?
(2) 2 Exit: member(a, [a, c])
(1) 1 Exit: member(a, [b, a, c])
yes
```

16.7 DEFICIÊNCIAS DO PROLOG

Apesar de Prolog ser uma ferramenta útil, ele não é nem uma linguagem de programação lógica pura, nem perfeita. Esta seção descreve alguns dos problemas com o Prolog.

16.7.1 Controle da ordem de resolução

O Prolog, por razões de eficiência, permite que o usuário controle a ordem do casamento de padrões durante a resolução. Em um ambiente de programação lógica puro, a ordem dos casamentos tentados que ocorrem durante a resolução é não determinística, e todos os casamentos podem ser tentados concorrentemente. Entretanto, como Prolog sempre casa na mesma ordem, começando do início da base de dados e no final esquerdo de um objetivo, o usuário pode afetar profundamente a eficiência ordenando as sentenças da base de dados para otimizar uma aplicação em particular. Por exemplo, se o usuário sabe que certas regras são muito mais propensas a serem bem-sucedidas do que outras durante uma "execução" em particular, o programa pode ser mais eficiente colocando essas regras primeiro na base de dados.

A execução lenta de programas não é o único resultado negativo da ordenação definida pelo usuário em programas Prolog. É muito fácil escrever sentenças em formas que causam laços infinitos e logo uma falha total do programa. Por exemplo, considere a forma sentencial recursiva:

```
f(X, Y) :- f(Z, Y), g(X, Z).
```

Devido à ordem de avaliação da esquerda para a direita, primeiro em profundidade, do Prolog, independentemente do propósito da sentença, ela causará um laço infinito. Como um exemplo desse tipo de sentença, considere

```
ancestor(X, X).
ancestor(X, Y) :- ancestor(Z, Y), parent(X, Z).
```

Ao tentar satisfazer o primeiro subobjetivo do lado direito da segunda proposição, o Prolog instancia Z para tornar `ancestor` verdadeira. Ele então tenta satisfazer esse novo subobjetivo, voltando à definição de `ancestor` e repetindo o mesmo processo, levando à recursão infinita.

Esse problema em particular é idêntico ao que um analisador sintático descendente recursivo tem com a recursão à esquerda em uma regra gramatical, conforme discutido no Capítulo 3. Como era o caso com regras gramaticais na análise sintática, simplesmente inverter a ordem dos termos no lado direito da proposição de exemplo elimina o problema. O problema com isso é que uma simples mudança no ordenamento dos termos não deveria ser crucial para a corretude do programa. Até porque, a falta da necessidade de o programador se preocupar com a ordem de controle é supostamente uma das vantagens da programação lógica.

Além de permitir que o usuário controle a ordem da base de dados e dos subobjetivos, o Prolog, em outra concessão à eficiência, permite algum controle explícito de rastreamento para trás. Isso é feito com o operador de corte, especificado por um ponto de exclamação (!). O operador de corte é na verdade um objetivo, não um operador. Como um objetivo, ele sempre é bem-sucedido de imediato, mas não pode ser ressatisfeito por rastreamento para trás. Logo, um efeito colateral do corte é que o subobjetivo a sua esquerda em um objetivo composto também não pode ser ressatisfeito por rastreamento para trás. Por exemplo, no objetivo

```
a, b, !, c, d.
```

se tanto `a` quanto `b` forem bem-sucedidos, mas `c` falhar, o objetivo completo falha. Esse objetivo seria usado caso se soubesse que sempre que `c` falha, o que, neste caso, seria uma perda de tempo tentar ressatisfazer `b` ou `a`.

O propósito do corte então é permitir que o usuário torne os programas mais eficientes dizendo ao sistema quando ele não deve tentar ressatisfazer subobjetivos que presumivelmente não podem resultar em uma prova completa.

Como um exemplo do uso do operador de corte, considere as regras `member` da Seção 16.6.7:

```
member(Element, [Element | _]).
member(Element, [_ | List]) :- member(Element, List).
```

Se a lista passada como parâmetro para `member` representa um conjunto, ela só pode ser satisfeita uma vez (conjuntos não contêm elementos duplicados). Logo, se `member` é usada como um subobjetivo em uma sentença de objetivo com múltiplos subobjetivos, pode existir um problema. O problema é que se `member` for bem-sucedida, mas o próximo subobjetivo falhar, o rastreamento para trás tentará ressatisfazer `member` ao continuar um casamento anterior. Mas devido ao fato de a lista passada como parâmetro para `member` ter apenas uma cópia do elemento para começar, `member` não pode ser bem-sucedida novamente, o que eventualmente faz todo o objetivo

falhar, apesar de quaisquer tentativas adicionais de ressatisfazer `member`. Por exemplo, considere:

```
dem_candidate(X) :- member(X, democrats), tests(X).
```

Esse objetivo determina se uma pessoa é um democrata e se é uma boa candidata para concorrer a uma determinada posição. O subobjetivo `tests` verifica uma variedade de características do democrata informado para determinar a adequação da pessoa para o cargo. Se o conjunto de democratas não tem duplicatas, não precisamos voltar ao subobjetivo `member` se o subobjetivo `tests` falhar – porque `member` procurará todos os outros democratas e falhará, porque não existem duplicatas. A segunda tentativa ao subobjetivo `member` será uma perda de tempo computacional. A solução para essa ineficiência é adicionar um lado direito à primeira sentença da definição `member`, com o operador de corte como o único elemento, como em

```
member(Element, [Element | _]) :- !.
```

O rastreamento para trás não tentará ressatisfazer `member`, mas fará o subobjetivo inteiro falhar.

O corte é particularmente útil em uma estratégia de programação em Prolog chamada **gerar e testar**. Em subprogramas que usam a estratégia gerar e testar, o objetivo consiste em subobjetivos que geram soluções em potencial, as quais são então verificadas por subobjetivos posteriores do tipo "teste". Soluções rejeitadas requerem rastreamento para trás para subobjetivos de "geração", os quais geram novas soluções em potencial. Como um exemplo de um programa gerar e testar, considere o seguinte, que aparece em Clocksin e Mellish (2003):

```
divide(N1, N2, Result) :- is_integer(Result),
                          Product1 is Result * N2,
                          Product2 is (Result + 1) * N2,
                          Product1 =< N1, Product2 > N1, !.
```

Esse programa realiza divisão inteira, usando adição e multiplicação. Como a maioria dos sistemas Prolog fornece divisão como um operador, esse programa na verdade não é útil, exceto para ilustrar um programa simples de geração e teste.

O predicado `is_integer` é bem-sucedido desde que seu parâmetro possa ser instanciado para algum valor não negativo. Se seu argumento não estiver instanciado, `is_integer` instancia seu valor como `0`. Se o argumento estiver instanciado para um inteiro, `is_integer` o instancia para o próximo valor superior inteiro.

Então, em `divide`, `is_integer` é o subobjetivo gerador. Ele gera elementos da sequência 0, 1, 2, ..., um a cada vez que for satisfeito. Todos os outros subobjetivos são de testes – eles verificam na tentativa de determinar se os valores produzidos por `is_integer` são, na verdade, a divisão dos dois primeiros parâmetros, `N1` e `N2`. O propósito do corte como o último

subobjetivo é simples: ele previne que `divide` tente encontrar uma solução alternativa uma vez que tenha encontrado *a* solução. Apesar de `is_integer` poder gerar um grande número de candidatos, apenas um é a solução, então o corte aqui previne tentativas desnecessárias de produzir soluções secundárias.

O uso do operador de corte tem sido comparado ao de desvios incondicionais (*gotos*) em linguagens imperativas (van Emden, 1980). Apesar de ele ser algumas vezes necessário, é possível abusar. De fato, ele é algumas vezes usado para fazer programas lógicos terem um fluxo de controle inspirado em estilos de programação imperativa.

A habilidade modificar o fluxo de controle em um programa Prolog é uma deficiência, porque é diretamente prejudicial a uma das vantagens mais importantes da programação lógica – que os programas não especificam como as soluções devem ser encontradas. Em vez disso, eles simplesmente dizem como a solução deve parecer. Esse projeto torna os programas mais fáceis de escrever e de ler. Eles não são entremeados com os detalhes de como as soluções devem ser determinadas e, em particular, qual a ordem precisa das computações que devem ser feitas para produzir a solução. Então, embora a programação lógica não requeira modificações no fluxo de controle, os programas Prolog geralmente as usam, em sua maioria por questões de eficiência.

16.7.2 A premissa do mundo fechado

A natureza da resolução em Prolog algumas vezes cria resultados enganosos. As únicas verdades, na preocupação de Prolog, são aquelas que podem ser provadas usando a sua base de dados. Qualquer consulta em que existe informação insuficiente na base de dados para prová-la em absoluto é tratada como falsa. O Prolog pode provar que um dado objetivo é verdadeiro, mas não pode prová-lo como falso. Ele simplesmente assume que, como não pode provar um objetivo como verdadeiro, esse objetivo deve ser falso. Em sua essência, o Prolog é um sistema verdadeiro/falha, em vez de um sistema verdadeiro/falso.

Na verdade, a premissa do mundo fechado não deve ser de todo estranha a você – nosso sistema judicial opera da mesma maneira. Os suspeitos são inocentes até que se prove ao contrário. Eles não precisam ser provados inocentes. Se um julgamento não pode provar que uma pessoa é culpada, ela é considerada inocente.

O problema com a premissa do mundo fechado está relacionada ao problema da negação, discutido na subseção a seguir.

16.7.3 O problema da negação

Outro problema com Prolog é sua dificuldade com negação. Considere a seguinte base de dados de dois fatos e um relacionamento:

```
parent(bill, jake).
parent(bill, shelley).
sibling(X, Y) :- (parent(M, X), parent(M, Y).
```

Agora, suponha que digitássemos a consulta

```
sibling(X, Y).
```

O Prolog responderia com

```
X = jake
Y = jake
```

Então, o Prolog "pensa" que `jake` é irmão (`sibling`) dele próprio. Isso acontece porque o sistema primeiro instancia `M` com `bill` e `X` com `jake` para tornar o primeiro subobjetivo, `parent(M, X)`, verdadeiro. Ele então começa no princípio da base de dados novamente para casar o segundo subobjetivo, `parent(M, Y)`, e chega às instanciações de `M` com `bill` e `Y` com `jake`. Como os dois subobjetivos são satisfeitos independentemente, com ambos os casamentos começando no princípio da base de dados, as respostas mostradas aparecem. Para evitar esse resultado, `X` deve ser especificado como um irmão (`sibling`) de `Y` apenas se eles não têm os mesmos pais (`parents`) e não são os mesmos. Infelizmente, dizer que eles não são iguais não é direto em Prolog, como iremos discutir. O método mais exato requereria adicionar um fato para cada par de átomos, dizendo que eles não são a mesma coisa. Isso, é claro, torna a base de dados muito grande, o que geralmente é mais informação negativa do que positiva. Por exemplo, as pessoas têm 364 mais dias de "não aniversário" do que de aniversário.

Uma solução alternativa simples é dizer no objetivo que `X` não deve ser o mesmo que `Y`, como em

```
sibling(X, Y) :- parent(M, X), parent(M, Y), not(X = Y).
```

Em outras situações, a solução não é tão simples.

O operador `not` de Prolog é satisfeito nesse caso se a resolução não puder satisfazer o subobjetivo `X = Y`. Logo; se `not` for bem-sucedido, não necessariamente significa que `X` não é igual a `Y`; em vez disso, significa que a resolução não pode provar a partir da base de dados que `X` é o mesmo que `Y`. Logo, o operador `not` em Prolog não é equivalente ao operador lógico NÃO, em que NÃO significa que o operando pode ser provado como verdadeiro. Essa não equivalência pode levar a um problema se ocorrer de termos um objetivo da forma

```
not(not(some_goal)).
```

que seria equivalente a

```
some_goal.
```

se o operador `not` de Prolog fosse um operador lógico NÃO. Em alguns casos, entretanto, eles não são a mesma coisa. Por exemplo, considere novamente as regras `member`:

```
member(Element, [Element | _]) :- !.
member(Element, [_ | List]) :- member(Element, List).
```

Para descobrir um dos elementos de uma lista, poderíamos usar o objetivo

```
member(X, [mary, fred, barb]).
```

que faria X ser instanciado com mary, que por sua vez seria impresso. Mas se usássemos

```
not(not(member(X, [mary, fred, barb]))).
```

a seguinte sequência de eventos ocorreria: primeiro o objetivo interno seria bem-sucedido, instanciando X para mary. Então, o Prolog tentaria satisfazer o próximo objetivo:

```
not(member(X, [mary, fred, barb])).
```

Essa sentença falharia porque member seria bem-sucedido. Quando esse objetivo falhasse, X teria sua instância removida, porque o Prolog sempre remove as instâncias de todas as variáveis em todos os objetivos que falham. A seguir, o Prolog tentaria satisfazer o objetivo externo not, que seria bem-sucedido, porque seu argumento falhou. Por fim, o resultado, que é X, seria impresso. Mas X não estaria atualmente instanciado, então o sistema indicaria isso. Geralmente, variáveis não instanciadas são impressas na forma de uma cadeia de dígitos precedida por um sublinhado. Então, o fato do not de Prolog não ser equivalente ao NÃO lógico pode ser, no mínimo, enganoso.

A razão fundamental pela qual o NÃO lógico não pode ser uma parte integral de Prolog é a forma da cláusula de Horn:

$$A \text{ :- } B_1 \cap B_2 \cap \ldots \cap B_n$$

Se todas as proposições B forem verdadeiras, pode-se concluir que A é verdadeira. Mas independentemente da verdade ou da falsidade de qualquer um dos Bs, não se pode provar que A é falso. A partir de lógica positiva, alguém pode concluir apenas lógica negativa. Então, o uso da forma de cláusula de Horn previne quaisquer conclusões negativas.

16.7.4 Limitações intrínsecas

Um objetivo fundamental da programação lógica, conforme descrito na Seção 16.4, é fornecer programação não procedural; ou seja, um sistema no qual os programadores especificam o que um programa deve fazer, mas não precisam especificar como isso deve ser realizado. O exemplo dado lá é reescrito aqui:

$$\text{sort(old\_list, new\_list)} \subset \text{permute(old\_list, new\_list)} \cap \text{sorted(new\_list)}$$
$$\text{sorted(list)} \subset \forall j \text{ such that } 1 \leq j < n, \text{list}(j) \leq \text{list}(j+1)$$

É direto escrever isso em Prolog. Por exemplo, o subobjetivo ordenado pode ser expresso como

```
sorted ([]).
sorted ([x]).
sorted ([x, y | list]) :- x <= y, sorted ([y | list]).
```

O problema com esse processo de ordenação é que ele não tem ideia de como ordenar, além de simplesmente enumerando todas as permutações de uma lista até que aconteça de ser criada uma que tem a lista ordenada – um processo muito lento, de fato.

Até agora, ninguém descobriu um processo pelo qual a descrição de uma lista ordenada possa ser transformada em algum algoritmo eficiente para a ordenação. A resolução é capaz de muitas coisas interessantes, mas certamente não isso. Logo, um programa em Prolog que ordena uma lista deve especificar os detalhes de como essa ordenação pode ser feita, como é o caso nas linguagens imperativas e funcionais.

Todos esses problemas significam que a programação lógica deve ser abandonada? De forma alguma! Como é atualmente, ela é capaz de lidar com muitas aplicações úteis. Além disso, é baseada em um conceito intrigante e, dessa forma, é interessante por si só ou para uso externo. Por fim, existe a possibilidade de desenvolvimentos de novas técnicas de inferência que permitirão um sistema de linguagem de programação lógica tratar de classes de problemas progressivamente maiores.

16.8 APLICAÇÕES DE PROGRAMAÇÃO LÓGICA

Nesta seção, descrevemos brevemente algumas das maiores classes de aplicações em potencial da programação lógica em geral e de Prolog em particular.

16.8.1 Sistemas de gerenciamento de bases de dados relacionais

Sistemas de gerenciamento de bases de dados relacionais (SGBDRs) armazenam dados na forma de tabelas. Consultas em tais tabelas são geralmente descritas em uma linguagem de consulta chamada Linguagem de Consulta Estruturada – Structured Query Language (SQL). SQL é não procedural no mesmo sentido que a programação lógica é não procedural. O usuário não descreve como obter a resposta; em vez disso, ele descreve apenas as características da resposta. A conexão entre programação lógica e SGBDRs deve ser óbvia. Tabelas simples de informação podem ser descritas por estruturas Prolog, e relacionamentos entre tabelas podem ser facilmente e convenien-