Capítulo 5 Introdução à Linguagem Prolog

5.1 Histórico de Prolog

A linguagem de programação **Prolog** foi desenvolvida em 1970, em Marselha, França, por Alain Colmerauer. O objetivo original do criador era integrar uma técnica de prova automática de teoremas — o princípio de resolução de Robinson — numa linguagem de programação a ser utilizada no processamento de linguagem natural. O princípio de resolução proposto por Robinson (v. **Apêndice A**) sugeria a utilização de apenas uma regra de inferência para prova automática de teoremas em vez das várias regras propostas pelos lógicos. Isso facilitou o projeto de uma linguagem de programação que permitiria que um programador fizesse o computador simular o processo de pensamento humano fazendo deduções a partir de informações dadas por fórmulas lógicas. Contribuiu ainda para o desenvolvimento teórico das ideias na qual Prolog é baseada o professor Robert Kowalski da Universidade de Edimburgo. A grande popularidade alcançada por Prolog, no entanto, deve-se à eficiente implementação realizada por David Warren, em meados de 1970, nessa mesma universidade.

A linguagem Prolog ganhou bastante popularidade na Europa desde o seu surgimento. No Japão, Prolog foi considerada vital para as pesquisas sobre o computador de quinta geração com o qual os japoneses sonhavam anos atrás. Nos Estados Unidos, após uma certa demora em sua aceitação, Prolog vem finalmente ganhando o espaço que lhe devido. Um fator que determinou a demora na aceitação de Prolog nos Estados Unidos foi o fato de que, por um longo tempo, não houve uma competidora séria para Lisp — largamente utilizada em pesquisas de inteligência artificial nesse país.

5.2 Programação Algorítmica e Programação Declarativa

Diferentemente de outras linguagens de programação, Prolog não requer que o programador forneça ao computador uma sequência de instruções a serem executadas (**programação algorítmica**). Em vez de informar ao computador

o que deve ser feito, o programador descreve o objeto que deve ser computado. Nesse sentido, a linguagem Prolog pode ser vista como um formalismo para definir conhecimento (**programação declarativa**), independentemente do método de computação.

Num sentido amplo, a evolução das linguagens de programação tem se processado no sentido de distanciar-se de linguagens de baixo nível — nas quais o programador especifica como alguma coisa deve ser feita — em direção de linguagens de alto nível — nas quais o programador especifica o quê deve ser feito. Por exemplo, com o surgimento de **Fortran** os programadores não foram mais forçados a falar com o computador na linguagem de baixo nível de registradores e endereços. Em vez disso, os programadores de Fortran puderam falar (aproximadamente) em sua própria linguagem.

Entretanto, Fortran e aproximadamente todas as outras linguagens ainda são linguagens que dizem ao computador como fazer alguma coisa. Tais linguagens são **algorítmicas** (ou **procedimentais**). Até a própria linguagem **Lisp** é uma linguagem fortemente algorítmica, segundo Winston — uma das maiores autoridades em inteligência artificial e autor de livros e artigos sobre Lisp.

Por outro lado, Prolog é uma **linguagem declarativa** que distancia-se das outras linguagens algorítmicas, encorajando os programadores a descreverem situações e problemas por meio de objetos e relações entre esses objetos e não especificando os meios pelos quais os problemas devem ser resolvidos. Assim um programador de Prolog deve estar mais interessado em conhecimento do que em algoritmos que manipulam o conhecimento.

5.3 Visão Geral de Prolog

Como foi dito, escrever um programa em Prolog não é como implementar um algoritmo numa linguagem de programação convencional. Em vez disso, o programador deve buscar objetos e relações entre esses objetos que ocorrem dentro da definição de seu problema. Por exemplo, quando se diz que Maria gosta de João, está-se declarando que existe uma relação *gosta de* entre os objetos Maria e João. Além disso, a relação possui uma ordem específica: o fato *Maria gosta de João* não implica necessariamente em *João gosta de Maria*.

Relações entre objetos podem ser estabelecidas por meio de **regras**. Por exemplo, a regra *duas pessoas são irmãs se elas são ambas do sexo feminino e têm os mesmos pais* diz algo sobre o que significa duas pessoas serem irmãs. Ela também informa como determinar se duas pessoas são irmãs: simplesmente verifique se elas são do sexo feminino e têm os mesmos pais.

Pode-se considerar Prolog como um depósito de fatos e regras, utilizados para responder questões (consultas). Programar em Prolog consiste em fornecer todos esses fatos e regras.

Em resumo, programação em Prolog consiste de:

- Declarar alguns fatos sobre objetos e suas relações
- Definir algumas regras sobre objetos e suas relações
- Fazer consultas sobre objetos e suas relações.

5.3.1 Fatos

Suponha que se deseje expressar o **fato** Maria gosta de João em Prolog. Esse fato consiste de dois objetos *Maria* e *João* e uma relação *gosta de*. Isso é escrito em Prolog como:

```
gosta(maria, joao).
```

Deve-se observar que:

- Os nomes de relações e objetos devem começar com letras minúsculas. Por exemplo, `gosta`, `maria`, `joao`.
- A relação é escrita em primeiro lugar, seguida pelos objetos entre parênteses e separados por vírgulas.
- Deve-se colocar um ponto no final do fato.

A ordem na qual os objetos são escritos é arbitrária, mas, uma vez que essa ordem é escolhida, ela deve ser respeitada para que haja consistência. No exemplo acima, escolheu-se `gosta(maria, joao)` com o sentido de que quem gosta aparece em primeiro lugar e de quem se gosta aparece em segundo lugar. Assim `gosta(maria, joao)` significa Maria gosta de João, enquanto `gosta(joao, maria)` significa João gosta de Maria, de acordo com a convenção estabelecida.

Exemplos de sentenças em português escritas como fatos em Prolog são apresentado na **Tabela 5–1**:

Português	Prolog
José é o pai de João	`pai(jose, joao)`
2 está entre 1 e 3	`entre(2, 1, 3)`
Ulysses é professor	`professor(ulysses)`
Paulo é irmão de Pedro	`irmao(paulo, pedro)`

Tabela 5–1: Sentenças em Português versus Fatos em Prolog

O nome de uma relação, i.e., o nome que aparece antes dos parênteses, é chamado **predicado** e os nomes dos objetos que aparecem dentro dos parênteses são chamados **argumentos**. A **aridade** de um predicado é o número de argumentos que ele possui. Assim, nos exemplos acima, `gosta` é um predicado de dois argumentos (ou de aridade 2), enquanto entre um predicado de três argumentos (ou de aridade 3).

Os fatos em Prolog, permitem expressar relações arbitrárias entre objetos. Uma coleção de fatos em Prolog é denominada **base de dados**.

5.3.2 Questões

Tendo-se alguns fatos, podem-se responder algumas questões sobre eles. Em Prolog, uma **questão** é semelhante a um fato, mas deve ser precedido por um símbolo especial (i.e., `?-`). Por exemplo:

```
?- pai(josé, joão).
```

pode ser interpretada como a questão *José é o pai de João?* ou *é um fato que José o pai de João?*

Quando faz-se uma pergunta a Prolog, o sistema faz uma busca na base de dados previamente introduzida. O sistema tenta casar o fato (nesse caso, denominado **objetivo**) contido na questão com um daqueles contidos na base de dados. Dois fatos **casam** se seus predicados são os mesmos e se cada um de seus argumentos correspondentes são os mesmos. Se Prolog encontra um fato que casa com a questão, o sistema responde `yes` (sim). Se tal fato não existe, Prolog responde `no` (não). A resposta é escrita na tela do computador logo abaixo da questão introduzida. Suponha a existência da seguinte base de dados previamente introduzida:

```
gosta(paulo, teresa).
gosta(pedro, ana).
gosta(joao, maria).
gosta(ana, jose).
```

As respostas às questões seguintes aparecem na linha que segue imediatamente cada questão:

```
?- gosta(joao, teresa).
no
?- gosta(maria, joao).
no
?- gosta(pedro, ana).
yes
?- pai(jose, paulo).
no
```

Observe que, em Prolog, a resposta negativa (`no`) é utilizada com o significado de nada casa com a questão e não com o significado de que a questão falsa. Por exemplo, suponha que se tenha a seguinte base de dados:

```
nasceu(jose, paraiba).
nasceu(joao, pernambuco).
brasileiro(paulo).
brasileiro(joao).
```

Poder-se-íam ter as seguintes questões e respostas:

```
?- nasceu(jose, paraiba).
yes
?- brasileiro(joao).
yes
?- brasileiro(jose).
no
```

Embora seja verdade que quem nasce na Paraíba é brasileiro, o sistema não pode concluir isso a partir dos fatos apresentados na base de dados. Assim, quando Prolog responde negativamente a uma questão, ele quer dizer que o fato contido na questão não pode ser provado.

Armazenamento e recuperação de informação não são as características mais interessantes de Prolog. Tudo que se fez até então foi obter de volta a informação introduzida na base de dados. Seria mais interessante se o sistema respondesse questões do tipo: quem é o pai de João? ou quem são os filhos de Pedro?

5.3.3 Variáveis

Em Prolog, uma questão do tipo Quem são os filhos de Pedro? pode ser reformulada como `X` é filho de Pedro? Nesse caso, não se sabe antecipadamente quais são os objetos que `X` pode significar e gostar-se-ía que Prolog apresentasse todas as possibilidades para o significado de `X`. Em Prolog, pode-se utilizar um nome como `X` para objetos a ser determinados por Prolog. Esses nomes são denominados **variáveis**. Quando Prolog utiliza uma variável, ela pode estar **instanciada** ou **não instanciada**. Uma variável está instanciada quando ela assume o valor de um objeto. Um variável está não instanciada quando ainda não assumiu o valor de nenhum objeto. Qualquer nome começando com letra maiúscula é interpretado por Prolog como uma variável.

Quando se formula para Prolog uma questão contendo uma variável, Prolog busca entre os fatos contidos na base de dados um objeto para o qual a variável poderia significar. Uma variável, como `X`, não denomina um objeto particular. Em vez disso, ela é utilizada para representar objetos cujos nomes não são conhecidos.

Considere a seguinte base de dados:

```
gosta(paulo, teresa).
gosta(joao, natureza).
gosta(maria, chocolate).
gosta(maria, natureza).
gosta(pedro, ana).
gosta(joao, maria).
```

A questão:

```
?- gosta(joao, X).
```

seria respondida por Prolog como:

```
X=natureza
```

O processamento dessa consulta é feito da forma explicada em seguida. Quando a pergunta é feita, a variável `X` está não instanciada. Prolog então procura um fato que case com o fato expresso na questão, levando em conta que uma variável não instanciada pode casar com qualquer objeto. Os fatos são pesquisados na ordem em que aparecem na base de dados. No caso do exemplo, Prolog procura por qualquer fato cujo predicado é `gosta` e cujo primeiro argumento é `joao`. Quando tal fato for encontrado (se for o caso), a variável será instanciada com o seu segundo argumento. Voltando ao exemplo, o primeiro fato que casa com a questão é `gosta(joao, natureza)`. Portanto a variável `X` é instanciada com `natureza`. Por razões que serão vistas em seguida, o sistema marca o local onde esse casamento ocorreu.

Uma vez que Prolog tenha encontrado o fato que casa com a questão, ele exibe o nome do objeto com o qual a variável foi instanciada. No exemplo, a variável `X` foi instanciada com `natureza`. Depois disso, o sistema fica esperando por mais instruções do programador. Se a tecla `[RETURN]` for pressionada, o sistema entenderá que o programador está satisfeito com apenas uma resposta e não tentar encontrar nenhuma outra. Por outro lado, se for digitado `;` seguido por `[RETURN]`, Prolog irá reiniciar a busca a partir do local onde ele colocou a marca mencionada acima. Diz-se nesse caso que Prolog tentará **ressatisfazer** a questão.

Suponha que se tenha digitado `;` após `X=natureza` no exemplo acima. Isso significa que se deseja que Prolog esqueça que `X` significa `natureza` e reinicia a busca com a variável `X` novamente não instanciada. A busca é reiniciada a partir do local marcado e o próximo casamento ocorre entre a questão e `gosta(joao, maria)`. A variável `X` é agora instanciada com `maria` e Prolog coloca uma marca no fato `gosta(joao, maria)`. O sistema então exibe `X=maria`. Se novamente for digitado `;`, Prolog reiniciará a busca. Nesse exemplo, nenhum

outro fato casa com a questão submetida e Prolog cessa a busca, exibindo `no`, significando que não há mais nenhuma outra resposta.

5.3.4 Conjunções

Considerando a base de dados do exemplo anterior, suponha que se deseje formular a pergunta: *Maria gosta de João e João gosta de Maria?* A letra *e* expressa o fato de se estar interessado na **conjunção** de duas questões, que devem ser ambas satisfeitas. Em Prolog, isso é representado por:

```
?- gosta(joao, maria), gosta(maria, joao).
```

A vírgula é lida como *e* e serve para separar qualquer número de objetivos diferentes que tenham que ser satisfeitos a fim de responder a uma consulta. Quando uma sequência de objetivos é fornecida, Prolog tenta satisfazer cada um dos objetivos procurando um casamento na base de dados. Todos os objetivos têm que ser satisfeitos para que a conjunção de objetivos seja satisfeita. Para a última questão, a resposta fornecida por Prolog é `no`, pois o primeiro objetivo `gosta(joao, maria)` é satisfeito, mas o segundo objetivo `gosta(maria, joao)` não é satisfeito. Assim a conjunção dos dois objetivos não é satisfeita.

Suponha, agora, que se queira verificar se existe um objeto tal que ambos João e Maria gostem. Essa questão também consiste de dois objetivos: primeiro determinar se existe algum `X` tal que Maria goste dele e, segundo, se João gosta desse mesmo `X`. Em Prolog, ter-se-ía:

```
?- gosta(maria, X), gosta(joao, X).
```

Prolog responde a consulta tentando satisfazer o primeiro objetivo. Se o primeiro objetivo casar na base de dados, então Prolog marcará o local na base de dados e tentará satisfazer o segundo objetivo. Se o segundo objetivo for satisfeito, Prolog marca o local de casamento na base de dados e apresenta o resultado.

É importante frisar que cada objetivo possui sua própria marca. Se o segundo objetivo não for satisfeito, então Prolog tentará ressatisfazer, automaticamente, o primeiro objetivo. Para cada objetivo, Prolog faz a busca em toda a base de dados. Se um fato na base de dados casa, satisfazendo um objetivo, então Prolog marca a posição desse fato para o caso em que o objetivo tenha que ser ressatisfeito posteriormente. Mas, quando um objetivo precisa ser ressatisfeito, Prolog começa a busca a partir da marca própria do objetivo e não a partir do início da base de dados. A questão do exemplo anterior demonstra esse comportamento de *retrocesso* do seguinte modo:

(1) A base de dados é pesquisada para o primeiro objetivo. O primeiro fato da base de dados que casa com esse objetivo é `gosta(maria, chocolate)`.

Agora, `X` é instanciada com `chocolate` em qualquer lugar onde ele aparecer na questão. O lugar onde se encontra o fato que casou com o primeiro objetivo é marcado.

(2) A base de dados é pesquisada agora para `gosta(joao, chocolate)`, pois o segundo objetivo era `gosta(joao, X)` e `X` agora está instanciada com `chocolate`. Agora não existe nenhum fato na base de dados que satisfaça `gosta(joao, chocolate)` e, assim, o objetivo falha. Quando um objetivo falha, Prolog tenta ressatisfazer (automaticamente) o objetivo anterior, de modo que, nesse caso, Prolog tenta ressatisfazer `gosta(maria, X)` partindo agora do local anteriormente marcado para esse objetivo. A variável `X` torna-se novamente não instanciada.

(3) O local marcado anteriormente na pesquisa de `gosta(maria, X)` foi aquele do fato `gosta(maria, chocolate)`. Assim Prolog recomeça a busca a partir desse fato. Como o final da base de dados ainda não foi atingido, busca-se um novo fato que possa satisfazer o primeiro objetivo. O próximo fato que casa é `gosta(maria, natureza)`. A variável `X` é instanciada com `natureza` e o Prolog marca o lugar desse fato.

(4) Prolog tenta agora satisfazer o segundo objetivo `gosta(joao, natureza)`. Nesse caso, Prolog não está tentando ressatisfazer o segundo objetivo. Ele está sendo avaliado novamente no sentido progressivo (como antes) e não no sentido regressivo como foi feito com o primeiro objetivo (essa diferença é crucial). Assim Prolog tenta satisfazer o segundo objetivo a partir do início da base de dados, encontra um fato nessa base que casa com o objetivo procurado e apresenta na tela o resultado da busca. Como o segundo objetivo foi satisfeito, Prolog marca o local onde ele satisfez esse objetivo na base de dados para o caso em que se queira ressatisfazê-lo. Note que, para cada objetivo que pode ser ressatisfeito, há uma marca (diferente) na base de dados.

(5) Neste ponto, ambos os objetivos foram satisfeitos e a conjunção como um todo foi satisfeita. A variável `X` está instanciada com `natureza`. O primeiro objetivo tem um marcador de lugar no fato `gosta(maria, natureza)` e o segundo objetivo tem uma marca no fato `gosta(joao, natureza)`.

Como em qualquer outra questão com variáveis, tão logo Prolog encontra uma resposta, ele para e espera por mais instruções do programador. Se for digitado `;`, Prolog ir tentar encontrar mais soluções para a questão. Agora, ele tentar ressatisfazer ambos os objetivos a partir das marcas deixadas na base de dados.

Resumindo, pode-se imaginar uma conjunção de objetivos como se eles fossem organizados da esquerda para a direita, separados por vírgulas. Cada objetivo possui um vizinho direito e outro esquerdo (exceto, claro, os objetivos mais extremos à esquerda e à direita que não possuem vizinhos esquerdo e direito, respectivamente). Quando Prolog manipula uma conjunção de objetivos, ele tenta satisfazer cada um deles, da esquerda para a direita. Se um objetivo for satisfeito, o sistema coloca uma marca no local da base de dados onde houve o casamento com esse objetivo. Nesse caso, pode-se imaginar uma seta dirigida do objetivo para o local da base de dados onde o casamento ocorreu. Além disso, qualquer variável não instanciada torna-se instanciada (em todas as suas ocorrências). Prolog então tenta o vizinho direito do objetivo, começando a partir do início da base de dados. Quando cada objetivo é satisfeito, ele deixa uma marca na base de dados no local do casamento para o caso em que o objetivo tenha que ser ressatisfeito posteriormente. Quando um objetivo falha (i.e., não é possível se encontrar um fato que case com ele), Prolog volta e tenta ressatisfazer seu vizinho da esquerda, a partir da marca deixada para esse vizinho na base de dados. Além disso, Prolog torna não instanciadas quaisquer variáveis que tenham se tornado instanciadas nesse objetivo no casamento anterior. Se um objetivo não puder ser ressatisfeito, Prolog, sucessivamente, volta ao vizinho da esquerda de cada objetivo que falha. Se o objetivo mais à esquerda falha, então não há nenhum vizinho anterior que possa ser ressatisfeito. Nesse caso, a conjunção inteira falha. Esse comportamento em que Prolog tenta repetidamente satisfazer e ressatisfazer objetivos numa conjunção é chamado de **retrocesso** (*backtracking*, em inglês).

5.3.5 Regras

Regras são utilizadas em Prolog quando se deseja afirmar que um fato depende de um grupo de outros fatos. Regras também são utilizadas para expressar definições. Por exemplo, em linguagem corrente, pode-se dizer que um indivíduo `X` é avô de outro indivíduo `Y` se existe um indivíduo `Z` que é filho de `X` e pai de `Y`.

Como outro exemplo, considere a definição (pouco rigorosa) de cachorro: *X é um cachorro se X é um animal e X late*. Observe que uma regra é uma afirmação geral sobre objetos e suas relações. Por exemplo, pode-se dizer que Rex um cachorro se Rex é um animal e Rex late e também que Lassie é um cachorro se Lassie um animal e Lassie late. Assim permite-se que uma variável signifique diferentes objetos em cada uso diferente da regra.

Em Prolog, uma regra consiste de uma **cabeça** e um **corpo**. A cabeça e o corpo são separados pelo símbolo `:-` (composto dos símbolos `:` e `-`). O símbolo `:-` é

pronunciado *se*. Os exemplos de regras dados em linguagem corrente podem ser escritos em Prolog como:

```
avo(X, Y) :- filho(Z, X), pai(Z, Y).
cachorro(X) :- animal(X), late(X).
```

Observe que as regras também são terminadas com ponto. A cabeça da primeira regra é `avo(X, Y)` e a cabeça da segunda `cachorro(X)`. A cabeça de uma a regra representa aquilo que a regra pretende definir. O corpo de uma regra representa um objetivo ou uma conjunção de objetivos que devem ser satisfeitos de modo que a própria regra seja satisfeita. Os corpos das regras acima são `filho(Z, X), pai(Z, Y)` e `animal(X), late(X)`.

O **escopo** de uma variável é toda a regra (da cabeça até o ponto) na qual a variável se encontra. Sempre que uma variável `X` está instanciada com algum objeto, todos os outras variáveis `X` dentro do escopo da variável devem também estar instanciados com esse mesmo objeto. Observe que a variável `X` que aparece na primeira regra do exemplo anterior não tem nenhuma relação com a variável da `X` da segunda regra, uma vez que cada uma delas está num escopo diferente.

Considere a seguinte base de dados sobre relações familiares:

```
homem(joao).
homem(jose).
homem(pedro).

mulher(maria).
mulher(ana).
mulher(paula).
mulher(joana).
mulher(alice).

pais(joao, maria, jose).
pais(paula, alice, pedro).
pais(ana, maria, jose).
pais(joana, alice, pedro).
```

Pode-se definir o fato de uma pessoa `X` ser irmã de outra `Y` se: (1) `X` é mulher, (2) `X` tem `M` como mãe e `P` como pai e (3) `Y` tem a mesma mãe `M` e o mesmo pai `P` que `X`. Essa regra pode ser escrita em Prolog como:

```
irma_de(X, Y) :- mulher(X), pais(X, M, P), pais(Y, M, P).
```

Suponha, agora, que se deseje responder à seguinte questão:

```
?- irma_de(joana, paula).
```

Prolog então procede da seguinte forma:

(1) A questão casa com a cabeça da regra `irma_de` definida acima, de forma que `X` e `Y` são instanciadas com `joana` e `paula`, respectivamente. Prolog coloca uma marca no local onde ele encontra essa regra. Em seguida, Prolog tenta satisfazer, um a um, os três objetivos que compõem o corpo da regra.

(2) O primeiro objetivo `mulher(joana)`, uma vez que `X` está instanciada com `joana`. Esse objetivo é satisfeito na lista de fatos da base de dados. Então Prolog marca o lugar na base de dados onde ocorreu o casamento.

(3) Agora Prolog tenta satisfazer `pais(joana, M, P)`. O casamento desse objetivo ocorre com o fato `pais(joana, alice, pedro)`, com as variáveis `M` e `P` sendo instanciadas com `alice` e `pedro`, respectivamente. Prolog marca o local da base de dados onde ocorreu esse casamento.

(4) Agora, Prolog tenta satisfazer o objetivo `pais(paula, alice, pedro)` (já que `Y` está instanciada com `paula` nessa questão e `M` e `P` estão instanciadas para `alice` e `pedro`, respectivamente). O objetivo é satisfeito, pois encontrado um fato na base de dados que casa com ele. Uma vez que esse o último objetivo na conjunção que forma o corpo da regra, o objetivo inteiro (i.e., a regra) é satisfeito e o fato `irma_de(joana, paula)` é considerado verdadeiro. Assim Prolog responde sim (`yes`).

Suponha agora que se apresente para Prolog a questão:

```
?- irma_de(joana, X).
```

que pode ser interpretada como: de quem Joana irmã? Nesse caso, Prolog procede da seguinte forma:

(1) A questão casa com a cabeça da regra `irma_de`. A variável `X` da regra é instanciada com `joana` e a variável `Y` da regra continua não instanciada, já que ela casa com a variável `X` da questão que está não instanciada. Entretanto, essas duas variáveis tornam-se compartilhadas. Isto é, logo que uma delas seja instanciada com algum objeto a outra também tornar-se instanciada com esse mesmo objeto.

(2) O primeiro objetivo `mulher(joana)` é satisfeito como antes na questão do exemplo anterior.

(3) O segundo objetivo `pais(joana, M, P)` também é satisfeito como antes. Agora as variáveis `M` e `P` tornam-se instanciadas com `alice` e `pedro`, respectivamente.

(4) Como `Y` ainda não está instanciada, o terceiro objetivo é:

```
pais(Y, alice, pedro)
```

Esse objetivo casa com `pais(paula, alice, pedro)` e a variável `Y` torna-se instanciada com `paula`.

(5) Como todos os objetivos foram satisfeitos, a regra toda é satisfeita, com `X` e `Y` (da regra) sendo instanciadas, respectivamente, com `joana` e `paula`. Uma vez que a variável `Y` na regra é compartilhada com `X` na questão, então `X` também torna-se instanciada com `paula`. E Prolog exibe como resposta `X=paula`.

Como já foi afirmado, Prolog espera que o programador informe se deseja mais respostas para a questão ou se está satisfeito. Observe que se for pedido a Prolog para ressatisfazer essa questão (digitando-se `;`), a próxima (e última) resposta será `X=joana`. Ou seja, Prolog responde que Joana é irmã dela própria (verifique isso). Como se poderia melhorar a definição da regra `irma_de` de forma a evitar a resposta de que uma pessoa irmã dela própria?

Como já foi visto, há duas maneiras de se fornecer informações a Prolog sobre um dado predicado: por meio de fatos e regras. Em geral, um predicado é definido usando uma mistura de fatos e regras. As regras e fatos de um predicado são denominadas **cláusulas** desse predicado.

5.4 Objetos de Dados

Os **objetos de dados** em Prolog (i. e., os elementos constituintes de um programa Prolog), são denominados **termos** e classificados de acordo com o diagrama mostrado na **Figura 5–1**.

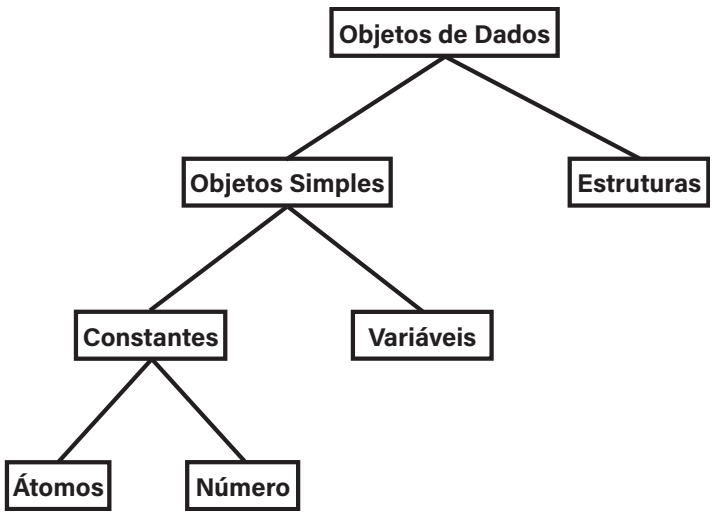

FIGURA 5–1: OBJETOS DE DADOS DE PROLOG

Prolog reconhece o tipo de um objeto (termo) por meio da sintaxe própria do objeto. Cada termo é escrito em Prolog como uma sequência de caracteres. Esses termos são classificados em quatro categorias:

1. Letras maiúsculas
2. Letras minúsculas
3. Os dígitos de 0 a 9
4. Caracteres especiais (que podem variar de uma implementação de Prolog para outra).

Cada tipo de termo possui diferentes regras de composição (i.e., como os caracteres são reunidos) para formar seu nome.

As **constantes** são utilizadas para dar nomes a objetos e a relações. Conforme visto acima, as constantes podem ser átomos ou números.

Há dois tipos de **átomos**: (1) aqueles construídos de letras e dígitos e (2) aqueles constituídos de símbolos especiais. O primeiro tipo deve normalmente começar com letra minúscula, conforme visto nas seções precedentes. Normalmente, átomos constituídos de símbolos especiais contêm apenas esses símbolos. Algumas vezes, pode-se desejar que um átomo inicie por uma letra maiúscula ou um dígito. Quando um átomo aparece entre apóstrofos, ele pode conter quaisquer caracteres em seu nome. O caractere _ (subtraço) pode ser inserido no meio de um átomo para melhorar sua legibilidade.

Exemplos de formação correta de átomos:

```
a1
maria
:-
?-
jose_maria
'jose maria'
```

Os seguintes nomes não são formações corretas de átomos:

```
1a
jose-maria
Jose
_xis
```

Deve-se tomar um certo cuidado na composição de nomes para átomos, pois algumas sequências de caracteres especiais possuem significado predefinido, como, por exemplo, o símbolo `:-`.

Os números utilizados em Prolog podem ser **inteiros** ou **reais**. A sintaxe utilizadas na escrita desses números em Prolog é semelhante àquela utilizada na

maioria das linguagens de programação de alto nível e os intervalos dentro dos quais os números são definidos é dependente da implementação de Prolog utilizada.

Exemplos de números inteiros são:

```
1
0
-123
1212
```

Exemplos de números reais que poderiam ser encontrados em determinada implementação:

```
3.1415
-0.1213
1000.1
```

Números reais não são muito utilizados em programação Prolog, já que Prolog é essencialmente uma linguagem para programação **simbólica** (não numérica). Em programação simbólica, números inteiros são frequentemente utilizados para contagem (por exemplo, para contar o número de elementos de uma lista), mas não existe muita necessidade de números reais.

5.4.1 Variáveis

As **variáveis** têm sintaxe semelhante àquela dos átomos. A diferença que elas começam sempre com letra maiúscula ou com subtraço _. Como exemplos de variáveis, têm-se:

```
Resposta
_a
Saida_de_valores_processados
```

Algumas vezes necessita-se utilizar uma variável de tal forma que não seja necessário saber com que valor essa variável será instanciada. Nesse caso, utiliza-se uma **variável anônima**, representada por subtraço _. Por exemplo, pode-se querer perguntar a Prolog se existe alguém que goste de Maria, mas não há interesse em saber quem é esse alguém. Nesse caso, a questão seria colocada para Prolog como:

```
?- gosta(_, maria).
```

Pode haver várias variáveis anônimas numa mesma cláusula. Nesse caso, elas não são instanciadas necessariamente com o mesmo objeto (como as variáveis comuns). Essa é uma característica peculiar das variáveis anônimas. As variáveis anônimas são utilizadas para não se ter que escrever variáveis de nomes diferentes quando elas não vão ser utilizadas em outro lugar numa mesma

cláusula. Serão vistas mais adiante vários usos práticos de variáveis anônimas que esclarecerão melhor sua utilidade.

5.4.2 Estruturas

Estruturas são termos constituídos de vários componentes que podem, eles próprios, ser estruturas. As estruturas auxiliam a organização dos dados em um programa Prolog porque permitem que um grupo de informações relacionadas seja tratado como um único objeto, em vez de entidades separadas.

Uma estrutura é escrita em Prolog especificando-se um **funtor** e seus **componentes** (ou **argumentos**). Os componentes de um funtor aparecem entre parênteses e separados por vírgulas, com o funtor vindo imediatamente antes de seus componentes. Suponha, por exemplo, que se deseje afirmar, em Prolog, o fato: *José é dono de um Onix*. Isso pode escrito como:

```
dono(jose, onix).
```

Nesse exemplo, `dono` é o funtor e `jose` e `onix` são seus argumentos. Ainda com relação a esse exemplo, suponha que o programador julgue que essa estrutura não seja muito esclarecedora para uma outra pessoa que eventualmente leia o programa (outra pessoa poderia interpretar *onix* como um mineral, por exemplo). O programador pode então querer especificar que o Onix de José é um carro azul. Assim a estrutura do exemplo pode ser rescrita como:

```
dono(jose, carro(onix, azul)).
```

Observe que, nesse último caso, o segundo argumento do funtor `dono` também é uma estrutura, cujo funtor é `carro` e cujos argumentos são `onix` e `azul`. Ainda nesse último exemplo, deve-se notar que tanto o funtor `dono` quanto o funtor `carro` possuem dois argumentos.

Considere a estrutura do último exemplo como fazendo parte de uma base de dados. A seguir são dados exemplos de prováveis consultas feitas e as correspondentes respostas de Prolog.

(1) José é dono de alguma coisa?

```
?- dono(jose, _).
yes
```

(2) José é dono de quê?

```
?- dono(jose, X).
X=carro(onix, azul)
```

(3) Quem é dono de um carro (de qualquer marca) azul?

```
?- dono(Quem, carro(_, azul)).
Quem=jose
```

(4) Existe alguém que seja dono de algum carro (de qualquer cor e marca)?

```
?- dono(_, carro(_, _)).
yes
```

As consultas:

```
?- X(jose, carro(onix, azul)).
```

e

```
?- dono(jose, Y(onix, azul)).
```

são desprovidas de qualquer significado para Prolog, pois um funtor não pode ser substituído por uma variável em nenhuma hipótese.

Um predicado, utilizado em fatos e regras, é justamente o funtor de uma estrutura. Os argumentos de um fato ou regra são justamente os componentes de uma estrutura. O número de argumentos de um funtor é chamado **aridade**. Pode-se ter, num mesmo programa Prolog, dois funtores de mesmo nome (construído de acordo com a regra de formação para átomos) e aridades diferentes. Nesse caso Prolog interpreta os dois funtores como sendo diferentes.

5.5 Operadores

Operadores são formas sintáticas convenientes para a escrita de certos funtores. Por exemplo, operações aritméticas são normalmente escritas com o uso de operadores. A expressão `x + y*z` escrita na forma normal de estruturas vista na **Seção 5.4** apareceria como `+(x, *(y, z))`, que, evidentemente, não é a forma comum com que pessoas normais acostumados. A forma `x + y*z` além de mais conveniente para leitura também evita a utilização de parênteses e vírgulas.

É importante notar que, em Prolog, os operadores não causam a execução de qualquer operação aritmética. Por exemplo, *2 + 3* não resulta em *5* em Prolog, mas é simplesmente outra forma de escrever-se o termo `+(2, 3)`. A forma como Prolog avalia expressões aritméticas será vista mais adiante.

Para se ler uma expressão aritmética contendo operadores, três informações básicas sobre cada operador devem ser conhecidas: sua posição, sua precedência e sua associatividade.

Pode-se ter uma grande variedade de operadores em Prolog, mas, por enquanto, serão utilizados como exemplos apenas os conhecidos operadores aritméticos:

+, -, * e /. A sintaxe de um termo contendo operadores depende inicialmente da posição do operador. Operadores como +, -, * e / são escritos entre seus argumentos e, por isso, são denominados **infixos**. É também possível escrever operadores antes de seus argumentos como em -a, em que – denota a inversão de sinal. Os operadores que aparecem antes de seus argumentos são denominados **prefixos**. Finalmente, um operador pode aparecer após seus argumentos, como, por exemplo, o operador ! utilizado em n! para indicar o fatorial de um número n. Operadores que aparecem após seus argumentos são denominados **sufixos**. Resumindo, a posição de um operador informa sua posição em relação aos seus argumentos.

Sabe-se que, para avaliar-se a expressão $x + y \times z$, deve-se primeiro multiplicar y por z e somar o resultado com x. Isso é sabido porque foi ensinado nos primeiros anos de escola que multiplicações e divisões devem ser efetuadas antes de soma e subtração, a não ser que parênteses sejam usados para modificar essa ordem. Assim, quando utilizam-se operadores, deve-se ter em mente uma convenção de ordem de execução das operações. Essa convenção é conhecida como **precedência dos operadores.**

A precedência de um operador é utilizada para indicar a ordem na qual ele deve ser aplicado. Cada operador utilizado em Prolog tem uma **classe de precedência** associada com ele. A classe de precedência é um número inteiro cujo valor depende da implementação de Prolog utilizada. (Por exemplo, na versão 4.0 de Arity Prolog, as classes de precedência variam de 1 a 1200). Entretanto, em qualquer implementação de Prolog, um operador com a mais baixa precedência tem sua classe de precedência próxima de 1. Em Prolog, os operadores de multiplicação e divisão têm precedências menores que os operadores de adição e subtração.

Associatividade de operadores é uma propriedade importante quando se têm vários operadores de mesma precedência numa mesma expressão. Por exemplo, a expressão *8/2/2* significa *8/(2/2)* ou *(8/2)/2*? A expressão seria resultaria em *8* no primeiro caso e em *2* no segundo caso. Para se distinguir entre esses dois casos, deve-se saber previamente se um operador é associativo à esquerda ou é associativo à direita. Um operador associativo à esquerda deve ter operações de precedência menor do que ou igual a ele à esquerda e operações de precedência menor do que ele à direita. Por exemplo, os operadores aritméticos de adição, subtração, multiplicação e divisão são associativos esquerda. Isso significa que a expressão *8/2/2* é interpretada como *(8/2)/2*.

Na prática, utilizam-se parênteses em expressões que são difíceis de se entender devido às regras de precedência e associatividade.

5.6 Igualdade e Casamento

Um predicado **embutido**[1] notável é a **igualdade**, que é um operador infixo escrito como `=`.

Quando Prolog tenta satisfazer o objetivo:

```
?- X = Y.
```

(pronunciado *X é igual a Y*), ele tenta casar `X` com `Y` e o objetivo é satisfeito quando o casamento ocorre. O predicado de igualdade é um predicado embutido; i.e., ele é predefinido em qualquer sistema Prolog. Esse predicado funciona como se fosse definido pelo seguinte fato:

```
X = X.
```

Dentro de uma mesma cláusula, `X` é sempre igual a `X` e essa propriedade é explorada na definição vista acima para o predicado de igualdade (o predicado `=` é utilizado como um operador na forma infixa).

Dado um objetivo da forma `X = Y`, com `X` e `Y` sendo dois termos quaisquer em que se permite a existência de variáveis não instanciadas dentro deles, as regras de Prolog para decidir se `X` e `Y` são iguais são as seguintes:

(1) Se `X` é uma variável não instanciada e `Y` está instanciada com qualquer termo, ou vice-versa, então `X` e `Y` são iguais. Também (importante!) `X` torna-se instanciada com o objeto com o qual `Y` está instanciada. Por exemplo, o objetivo:

```
?- gosta(maria, manga) = X.
```

é satisfeito e faz com que `X` seja instanciada com `gosta(maria, manga)`.

(2) Constantes são sempre iguais a si próprias. Por exemplo, os objetivos dos exemplos seguintes têm os seguintes resultados:

Objetivo	Resultado
`?- homem = homem.`	**satisfeito**
`?- poeira = po.`	**falha**
`?- 2001 = 2001.`	**satisfeito**
`?- 12 = 21.`	**falha**

(3) Duas estruturas são iguais se elas têm o mesmo funtor, a mesma aridade (número de argumentos) e os argumentos correspondentes são iguais. Por exemplo, o objetivo:

[1] Outros predicados embutidos serão discutidos no **Capítulo 6**.

```
?- anda(jose, bicicleta) = anda(jose, X).
```

é satisfeito e faz com que `X` seja instanciada com `bicicleta`.

Estruturas podem ser aninhadas em qualquer profundidade. Se estruturas aninhadas são testadas para igualdade, o teste pode demorar a ser efetuado porque há muitas estruturas para serem testadas. Por exemplo, o objetivo:

```
?- a(b, C, d(e, F, g(h, i, J))) = a(B, c, d(E, f, g(H, i, j))).
```

é satisfeito e faz com que `B` seja instanciada com `b`, `C` com `c`, `E` com `e`, `F` com `f`, `H` com `h` e `J` com `j`.

Se duas variáveis não instanciadas são testadas para igualdade, tem-se um caso especial da regra **(1)** acima. Nesse caso, o objetivo satisfeito e as variáveis tornam-se compartilhadas. Se duas variáveis são compartilhadas, quando uma torna-se instanciada com um objeto, a outra torna-se instanciada com esse mesmo objeto. Assim, na regra:

```
faz(X, Y) :- X = Y.
```

o segundo argumento será instanciado tão logo o primeiro seja instanciado. Um objetivo `X = Y` será sempre satisfeito se qualquer dos argumentos for não instanciado. Um meio mais fácil de se escrever essa última regra é:

```
faz(X, X).
```

Existe ainda em Prolog o predicado embutido `\=`, que se lê *diferente*. O objetivo `X \= Y` é satisfeito se `X = Y` falha e falha se o objetivo `X = Y` é satisfeito. Assim `X \= Y` significa `X` não pode ser igual a `Y`.

5.7 Aritmética

Operações aritméticas são úteis para comparações de números e para cálculo de resultados.

Prolog provê alguns predicados embutidos para comparar números. Os predicados `=` e `\=` discutidos na **Seção 5.6** podem ser utilizados para comparar números, conforme foi visto. Os argumentos seriam variáveis instanciadas com números inteiros ou números inteiros escritos como constantes. Os outros predicados para comparação existentes são mostrados na **Tabela 5–2** e descritos em seguida:

Predicado	Significado
`X = Y`	**X e Y representam o mesmo número**
`X \= Y`	**X e Y representam números diferentes**
`X < Y`	**X é menor que Y**
`X > Y`	**X é maior que Y**
`X =< Y`	**X é menor do que ou igual a Y**
`X >= Y`	**X é maior do que ou igual a Y**

Tabela 5-2: Predicados de Comparação de Prolog

Note que `X` é menor do que ou igual a `Y` é escrito como `X =< Y` e não `X <= Y` como na maioria das linguagens de programação. A razão para isso é que o programador pode utilizar o átomo `<=`, que assemelha-se a uma seta apontando para a esquerda, para outros propósitos de sua própria conveniência.

Embora esses operadores sejam predicados, eles não podem ser utilizados de forma indiscriminada como outros predicados não embutidos. Por exemplo, Prolog não permite que se introduza um fato como

```
2 > 3.
```

que afirma que 2 é maior que 3. Isso previne que se modifique o significado dos predicados embutidos arbitrariamente. Essa regra vale para os predicados embutidos de uma forma geral: não é permitido que se acrescentem novos fatos para esses predicados.

Exemplos de utilização dos predicados de comparação enumerados acima:

```
?- 5 =< 7.
yes
?- 4 > 4.
no
?- 4 >= 4.
yes
?- 3 <= 5.
```

Nesse último caso, Prolog apresenta uma mensagem de erro indicando que o predicado `<=` não é definido.

O predicado `is` é utilizado em Prolog para a avaliação de expressões aritméticas. Ele é um operador infixo e seu argumento direito é interpretado como uma expressão aritmética. Para satisfazer um predicado `is`, Prolog primeiro avalia a expressão aritmética à sua direita de acordo com as regras de aritmética (nesse

instante, todas as variáveis na expressão devem ser instanciadas). O resultado casa com o argumento da esquerda para determinar se o objetivo é satisfeito. Por exemplo, no objetivo:

```
?- R is 2 + 4*3.
```

a expressão à direita de `is` é avaliada, resultando em `14` que é o valor com o qual `R` é instanciada.

É necessário utilizar o predicado `is` sempre que se deseja avaliar uma expressão aritmética. É claro que, se o argumento do lado direito de `is` não puder ser avaliado por alguma razão, o objetivo contendo `is` irá falhar. Vários operadores aritméticos podem ser utilizados na expressão aritmética do lado direito de um predicado `is`.

Todas as implementações de Prolog devem conter os operadores exibidos na **Tabela 5–3**.

Predicado	Significado
`X + Y`	**Soma de X e Y**
`X - Y`	**Diferença entre X e Y**
`X * Y`	**Produto de X e Y**
`X / Y`	**Quociente de X por Y**
`X mod Y`	**Resto da divisão de X por Y**

Tabela 5–3: Operadores de Prolog

Essa lista, juntamente com aquela de operadores de comparação, deve ser suficiente para a solução de problemas simples de aritmética. Prolog é essencialmente uma linguagem para processamentos não numéricos, de modo que suas facilidades aritméticas não são tão importantes quanto em outras linguagens de programação.

5.8 Listas

Listas são estruturas de dados muito utilizadas em programação simbólica, como programação em Lisp e em Prolog. Uma lista é uma sequência ordenada de qualquer comprimento de elementos. Os elementos de uma lista podem ser quaisquer termos — constantes, variáveis ou estruturas que podem incluir outras listas. Essas propriedades são importantes quando não se pode predizer quão longa uma lista pode ser e que informações ela deve conter. Além disso, as listas podem representar qualquer tipo de estrutura que se deseje utilizar em

computação simbólica. Listas podem ser utilizadas para representar gramáticas, mapas de cidades, programas de computador, grafos, fórmulas, funções, etc. Em Prolog, listas são apenas um tipo particular de estrutura. Na linguagem de programação Lisp, as únicas estruturas de dados disponíveis são as constantes e as listas.

As listas podem ser representadas como um tipo especial de **árvore**[2]. Uma lista é uma lista vazia (i.e., sem nenhum elemento) ou é uma estrutura com dois componentes: **cabeça** e **cauda**. A **lista vazia** escrita como `[]`. A cabeça e a cauda de uma lista são componentes de um funtor escrito como `.` (ponto). A cabeça de uma lista o primeiro argumento desse funtor e a cauda é o segundo argumento dele. Assim a lista que consiste apenas do elemento `a` é representada por `.(a, [])`, que pode ser colocada em forma de representação gráfica de árvore conforme é mostrado na **Figura 5–2**.

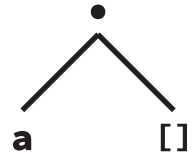

Figura 5–2: Lista com Um Elemento em Forma de Árvore

A lista consistindo dos átomos `a`, `b` e `c` seria escrita como `.(a, .(b, .(c, [])))` e sua representação em forma de árvore seria de acordo com o que mostra a **Figura 5–3**.

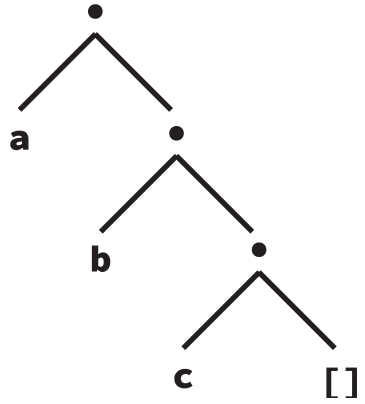

Figura 5–3: Lista com Três Elementos em Forma de Árvore

O funtor ponto é definido como operador infixo em Prolog. Assim é permitido escrever as listas dos exemplos acima como `a.[]` e `a.(b.(c.[]))`. O operador ponto é associativo à direita, de forma que a segunda lista poderia ser escrita

[2] Brevemente, árvore é uma estrutura de dados hierárquica.

simplesmente como: `a.b.c.[]`. Listas são sequências ordenadas, de modo que a lista `a.b.[]` é diferente da lista `b.a.[]`.

Como a notação de ponto é um pouco complicada para a escrita de listas mais complexas, há outra sintaxe que pode ser utilizada para escreverem-se listas em Prolog. Essa notação de listas consiste dos elementos da lista separados por vírgulas, com toda a lista entre colchetes. Por exemplo, as listas dos exemplos apresentados acima poderiam ser escritas nessa notação como: `[a]` e `[a, b, c]`.

Abaixo, encontram-se mais alguns exemplos de listas em Prolog:

```
[]
[o, homem, [gosta, de, pescar]]
[a, X0, b, Y0, [X1, Y1]]
```

Variáveis contidas dentro de listas são tratadas da mesma forma que variáveis em qualquer outra estrutura. Elas podem tornar-se instanciadas em qualquer instante. Assim o uso judicioso de variáveis pode prover um meio para colocarem-se espaços em listas que podem ser preenchidos com objetos num instante posterior.

As listas são manipuladas dividindo-as em cabeça e cauda. Quando uma lista aparece na notação de colchetes, a cabeça é o primeiro elemento da lista e a cauda a lista que consiste de todos os elementos da lista original exceto o primeiro. Os exemplos na **Tabela 5–4** esclarecem melhor os conceitos de cabeça e cauda de listas. Observe que a lista vazia não tem cabeça nem cauda.

Lista	Cabeça	Cauda
`[a, b, c, d]`	`a`	`[b, c, d]`
`[a]`	`a`	`[]`
`[]`	sem cabeça	sem cauda
`[[o, rato], roeu]`	`[o, rato]`	`[roeu]`
`[o, [rato, roeu]]`	`o`	`[[rato, roeu]]`
`[A+B, a+b]`	`A+B`	`[a+b]`

Tabela 5–4: Exemplos de Cabeça e Cauda de Lista 1

Uma operação comum em Prolog é dividir uma lista em cabeça e cauda, por isso há uma notação especial para representar uma lista com cabeça `X` e cauda `Y`, que é a notação `[X|Y]`. Um objetivo nesse formato irá instanciar `X` com a cabeça de uma lista e `Y` com a cauda dessa lista.

Os exemplos na **Tabela 5–5** mostram vários casamentos possíveis entre listas com as instanciações possíveis de variáveis contidas nelas:

Lista 1	Lista 2	Instanciações

`[X, Y, Z]`	`[joao, gosta, maria]`	`X = joao`
		`Y = gosta`
		`Z = maria`

`[gato]`	`[X|Y]`	`X = gato` `Y = []`

`[X, Y|Z]`	`[maria, gosta, joao]`	`X = maria`
		`Y = gosta`
		`Z = [joao]`

`[[o, Y]|Z]`	`[[X, roeu], [a, roupa]]`	`X = o`
		`Y = roupa`
		`Z = [[a, roupa]]`

`[X, Y|Z, W]`	**Sintaxe incorreta**	
`[ouro|X]`	`[ouro, brilha]`	`X = [brilha]`
`[cao, gato]`	`[gato, X]`	`(falha)`
`[cavalo|C]`	`[A|branco]`	`A = cavalo` `C = branco`

Tabela 5–5: Exemplos de Cabeça e Cauda de Lista 2

Conforme o último exemplo mostra, é possível usar a notação de lista para criar estruturas que se assemelham a listas, mas que não terminam com a lista vazia. A estrutura `[cavalo|branco]` tem `cavalo` como cabeça e `branco` como cauda. A constante `branco` nem é uma lista não vazia nem é a lista vazia.

Outro uso da sintaxe de listas é a representação de **cadeias de caracteres** (**strings**). Se uma cadeia de caracteres estiver entre aspas, ela é representada como uma lista de códigos inteiros que representam os caracteres no código ASCII. Por exemplo, a cadeia de caracteres `sistema` é transformada por Prolog na lista `[115, 105, 115, 116, 101, 109, 97].`

5.9 Operações Importantes Sobre Listas

Esta seção apresenta operações essenciais sobre listas em Prolog.

5.9.1 Membro de uma lista

A operação mais comum sobre uma lista é determinar se um dado objeto elemento é **membro** da lista. O objetivo aqui escrever um predicado `membro`, tal que `membro(X, Y)` seja satisfeito se o objeto representado por `X` for membro da lista representada por `Y`. Há duas situações a serem consideradas. Primeiro, é

sabido que um fato `X` é um membro da lista `Y` se `X` for a cabeça lista. Em Prolog esse fato pode ser escrito como:

```
membro(X, [X|_]).
```

que pode ser traduzido como `X` é membro de uma lista que tem `X` como cabeça. Note que foi utilizada a variável anônima _ para representar a cauda da lista. Isso ocorre porque a cauda não é utilizada para nada nesse fato particular. Esse fato também poderia ser escrito por meio da regra:

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

que, evidentemente, é mais complexa do que a primeira forma.

A segunda regra diz que `X` é membro de uma lista se `X` está na cauda da lista. Essa segunda regra pode ser escrita em Prolog como:

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

Essa regra pode ser lida como: `X` membro da lista se `X` é membro da cauda da lista[3]. Note que, novamente, a variável anônima _ foi utilizada aqui. Dessa vez, a razão é que não interessa saber qual é a cabeça da lista. As duas regras juntas definem o predicado **membro**.

Esse é um exemplo típico de utilização de definições recursivas em Prolog. O ponto mais importante a ser considerado, quando se tem um predicado definido recursivamente é observar as condições de fronteira (base da recursão) e o caso recursivo. Na realidade há duas condições de fronteira para o predicado membro: ou o objeto está na lista ou ele não está. A primeira condição de fronteira de membro é reconhecida pela primeira cláusula, que faz com que a busca na lista pelo objeto pare se o primeiro argumento de membro casa com a cabeça da lista representada pelo segundo argumento. A segunda condição de fronteira ocorre quando o segundo argumento de membro é a lista vazia.

Como se assegura que as condições de fronteira serão sempre satisfeitas? Deve-se olhar para o caso recursivo (a segunda cláusula de membro). Note que, cada vez que **membro** tenta satisfazer a si próprio, o objetivo é uma lista menor, pois a cauda de uma lista é sempre uma lista menor que a lista original. Eventualmente, uma das duas coisas irá ocorrer: ou a primeira regra de **membro** é satisfeita, ou a lista vazia é fornecida como seu segundo argumento. Quando qualquer uma dessas coisas acontece, a recursão de **membro** chega ao fim. A primeira condição de fronteira é reconhecida por um fato sem nenhum subobjetivo a ser considerado. A segunda condição de fronteira não é

[3] Aqueles acostumadas com recursividade perceberão que essa é uma definição recursiva, com o fato `membro(X, [X|_])` sendo a base da recursão.

reconhecida por nenhuma cláusula de **membro**, assim **membro** irá falhar nesse caso. Resumindo, se o predicado membro introduzido em Prolog:

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

as consultas seguintes resultariam em:

```
?- membro(c, [a, b, c, d, e]).
yes

?- membro(d, [a, b, c]).
no
```

Nessa última questão, Prolog procede da seguinte forma: a primeira cláusula não casa com o objetivo e Prolog tenta a segunda cláusula. A segunda cláusula casa com o objetivo, com **Y** sendo instanciada com **[b, c]** e **membro(d, [b, c])** torna-se o próximo objetivo. Novamente, a primeira cláusula não casa, mas a segunda cláusula casa, com **Y** agora sendo instanciada com **[c]**. O objetivo agora é **membro(d, [c])**, que falha na primeira cláusula e casa na segunda, com **Y** sendo instanciada com **[]**. O próximo objetivo então é **membro(d, [])** que falha em ambas as cláusulas de membro. Consequentemente, a resposta é negativa.

É importante lembrar que, cada vez que é utilizada a segunda cláusula de **membro** para tentar satisfazer esse predicado, Prolog trata cada chamada do objetivo **membro** como uma cópia diferente. Isso previne o fato de as variáveis num uso de uma cláusula serem confundidas com variáveis em outro uso dessa mesma cláusula.

Definições recursivas são frequentemente encontradas em Prolog e elas não são diferentes de qualquer outro tipo de definição. Entretanto, deve-se ter cuidado para não se escrever definições circulares. Considere o seguinte exemplo:

```
pai(X, Y) :- filho(Y, X).
filho(A, B) :- pai(B, A).
```

Nesse exemplo, para satisfazer **pai**, deve-se estabelecer **filho** como objetivo. Entretanto, a definição de **filho** utiliza apenas **pai** como objetivo. Assim a tentativa de responder uma questão sobre **pai** ou **filho** levaria a uma **repetição** (**infinita**) na qual Prolog nunca inferiria nada de novo e essa repetição nunca iria terminar.

Outro problema semelhante que pode surgir em definições recursivas é quando uma regra invoca um objetivo que é essencialmente equivalente ao objetivo original que causou a utilização dessa regra. Por exemplo, se fosse definido:

```
pessoa(X) :- pessoa(Y), mae(X, Y).
pessoa(jose).
```

e se fosse feita a consulta:

```
?- pessoa(X).
```

Prolog primeiro usaria a regra e geraria o subobjetivo `pessoa(Y)`. Ao tentar satisfazer esse subobjetivo, Prolog novamente consideraria a primeira regra e geraria outro objetivo equivalente e assim por diante. É claro que, se houvesse chance de retrocesso, ele encontraria o fato `pessoa(jose)` e produziria uma solução. O problema que, para retroceder, Prolog teria que falhar após tentar a primeira alternativa. Mas, nesse exemplo, a tarefa que Prolog encontra é infinitamente longa e, assim, ele nunca tem a chance de retroceder ou falhar. A lição, portanto, é:

Não suponha que, simplesmente porque você forneceu todos os fatos e regras relevantes, Prolog sempre os encontrará.

Você deve ter em mente, quando escreve programas em Prolog, como Prolog faz buscas na base de dados e quais variáveis serão instanciadas quando uma das regras for utilizada.

Nesse último exemplo, a solução mais simples seria colocar o fato antes da regra, em vez de depois dela. De fato, como heurística geral, uma boa ideia é colocar fatos antes de regras sempre que possível. Algumas vezes, a colocação das regras numa ordem particular funcionar se elas forem utilizadas para resolver objetivos de uma forma, mas não funcionará se objetivos de outra forma forem gerados.

Considere a seguinte definição de `eh_lista(X)`, no qual o predicado `eh_lista` é satisfeito se `X` é uma lista na qual a cauda de seu último elemento é a lista vazia:

```
eh_lista([A|B]) :- eh_lista(B).
eh_lista([]).
```

Se essas cláusulas forem utilizadas para responder questões do tipo:

```
?- eh_lista([a, b, c, d]).
?- eh_lista([]).
?- eh_lista(f(a, b)).
```

então a definição funcionará satisfatoriamente. Mas quando se introduz a questão:

```
?- eh_lista(X).
```

o programa entra em repetição infinita. Um predicado similar à **eh_lista** e que não é susceptível a repetição infinita é definido da seguinte forma:

```
eh_lista_novo([]).
eh_lista_novo([_|_]).
```

Essa versão testa a primeira parte da lista, em vez de verificar se a cauda da lista é `[]`. Esse não é um teste tão forte quanto aquele de **eh_lista** (por que?), mas não provoca repetição infinita se o argumento for uma variável.

5.9.2 Concatenação de Duas Listas

O predicado **anexa**, apresentado em seguida, é utilizado para unir duas listas para formar uma nova lista. Por exemplo, o objetivo seguinte é satisfeito:

```
?- anexa([a, b, c], [d, e, f], [a, b, c, d, e, f]).
```

O predicado **anexa** é utilizado com mais frequência para criar uma nova lista por meio da concatenação de duas outras listas como em:

```
?- anexa([esta, eh, uma], [nova, lista], Lista).
Lista = [esta, eh, uma, nova, lista]
```

Mas, ele pode ser utilizado de outras maneiras como em:

```
?- anexa(Lista, [c, d], [a, b, c, d]).
Lista = [a, b]
```

O predicado **anexa** pode ser definido como segue:

```
anexa([], L, L).
anexa([C|L1], L2, [C|L3]) :- anexa(L1, L2, L3).
```

A condição de parada ocorre quando a primeira lista é vazia. Nesse caso, qualquer lista anexada à lista vazia é a própria lista original. Caso contrário, os seguintes pontos mostram os princípios da segunda regra:

[1] O primeiro elemento da primeira lista (`C`) será sempre o primeiro elemento da terceira lista.

[2] A cauda da primeira lista (`L1`) terá sempre o segundo argumento (`L2`) anexado a ela para formar a cauda (`L3`) do terceiro argumento.

[3] O predicado **anexa** é utilizado para fazer a anexação mencionada no item [2].

[4] Ao separar-se repetidamente a cabeça do resto do primeiro argumento, ele vai sendo gradualmente reduzido à lista vazia, quando a condição de fronteira ocorre.

5.10 Geração de Soluções Múltiplas

Já foi visto nas seções precedentes o que pode acontecer quando Prolog tenta satisfazer ou ressatisfazer um objetivo:

1. Quando se tenta satisfazer um objetivo, faz-se uma busca a partir do início da base de dados. Duas coisas podem acontecer:
   - **1.1** Um fato ou a cabeça de uma regra que casa com o objetivo encontrado. Nesse caso, o objetivo casa. Marca-se o local na base de dados e instanciam-se quais quer variáveis não instanciadas que tenham casado. Se o casamento ocorre com a cabeça de uma regra, deve-se primeiro tentar satisfazer os (sub-)objetivos introduzidos pela regra. Se um objetivo for satisfeito, tenta-se satisfazer o próximo objetivo (se houver) e assim por diante.
   - **1.2** Nenhum fato ou cabeça de regra que case com o objetivo encontrado. Nesse caso, o objetivo **falha**. Se o objetivo aparece numa conjunção, tenta-se ressatisfazer o objetivo anterior (se houver), i.e., o objetivo à sua esquerda no programa.
2. Quando se tenta ressatisfazer um objetivo, procura-se ressatisfazer cada um dos seus possíveis subobjetivos. Se nenhum subobjetivo pode ser ressatisfeito de modo conveniente, tenta-se encontrar uma cláusula alternativa para o próprio objetivo. Nesse caso, deve-se retornar a não instanciadas quais quer variáveis que tenham se tornado instanciadas quando a cláusula prévia foi escolhida. Isso é o que se entende por *desfazer* todo o trabalho previamente feito para esse objetivo. Depois, reassume-se a busca na base de dados, mas começando a busca a partir do local onde foi previamente colocado o marcador de lugar. Como antes, esse novo objetivo retrocedido pode ser satisfeito ou falhar e ou o passo **1.1** ou o passo **1.2** acima deve ocorrer.

Nas próximas seções, o processo de retrocesso será examinado com maiores detalhes. Será também examinado um mecanismo especial de Prolog: o corte. O corte permite que se informe a Prolog que as escolhas anteriores não precisam ser consideradas novamente. Isto é, o corte permite um controle do processo (automático) de retrocesso de Prolog pelo programador. Antes disso, será visto como é possível a geração de soluções múltiplas em Prolog.

O modo mais simples pelo qual um conjunto de fatos pode permitir soluções múltiplas ocorre quando há vários fatos que casam com a questão. Por exemplo, se houver os seguintes fatos na base de dados, nos quais `pai(X, Y)` significa *o pai de X é Y*:

```
pai(maria, jorge).
pai(joao, jorge).
pai(paula, sergio).
pai(jorge, eduardo).
```

a questão:

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

terá várias respostas possíveis. Se for digitado ponto e vírgula após cada resposta, Prolog fornecer as seguintes soluções:

```
X = maria, Y = jorge ;
X = joao, Y = jorge ;
X = paula, Y = sergio ;
X = jorge, Y = eduardo ;
no
```

Prolog determina essas respostas fazendo uma busca na base de dados até encontrar fatos e regras sobre pai na ordem em que eles foram fornecidos. Prolog não particularmente *esperto* no sentido de que ele não se lembra de nada que tenha mostrado antes. Por exemplo, se for perguntado:

```
?- pai(_, X).
```

(que significa: para qual `X`, `X` é pai?) obtém-se:

```
X = jorge ;
X = jorge ;
X = sergio ;
X = eduardo
```

com **`jorge`** sendo repetido duas vezes porque Jorge é pai de Maria e João. Se Prolog encontra duas maneiras de mostrar a mesma coisa, ele as trata como duas soluções diferentes.

O retrocesso ocorre exatamente do mesmo modo se as alternativas são incorporadas mais profundamente no processamento. Por exemplo, uma regra de definição de um dos filhos de `X` é `Y` pode ser:

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

Então a questão:

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

resultaria em:

```
X = jorge, Y = maria ;
X = jorge, Y = joao ;
X = sergio, Y = paula ;
X = eduardo, Y = jorge
```

Como `pai(Y, X)` tem quatro soluções, `filho(X, Y)` também as tem. Além disso, as soluções são geradas na mesma ordem. O que é diferente é a ordem dos argumentos, conforme especificado na definição de `filho`. Similarmente, se for definido:

```
?- pai(X) :- pai(_, X).
```

(com `pai(X)` significando que `X` é um pai) então a questão:

```
?- pai(X).
```

resultará em:

```
X=jorge ;
X=jorge ;
X=sergio ;
X=eduardo
```

Se fatos e regras são misturados, as alternativas seguem-se novamente na ordem em que eles são apresentados. Assim, suponha que se tenha a seguinte base de dados:

```
pessoa(adao).
pessoa(X) :- mae(X, _).
pessoa(eva).

mae(cain, eva).
mae(abel, eva).
mae(jabal, adah).
mae(tubalcain, zillah).
```

(Interpretação: *adao é uma pessoa*, *X é uma pessoa se X tem mãe* e *eva uma pessoa*; também, várias pessoas têm várias mães.) Nesse caso, se for feita a consulta:

```
?- pessoa(X).
```

a resposta seria:

```
X=adao ;
X=caim ;
X=abel ;
X=jabal ;
X=tubalcain ;
X=eva
```

Observe um caso mais interessante em que há dois objetivos, cada um dos quais com várias soluções. Suponha que se esteja planejando uma festa e se deseja especular sobre quem pode dançar com quem. Pode-se começar escrevendo o seguinte programa:

```
possivel_par(X, Y) :- rapaz(X), moca(Y).
rapaz(joao).
rapaz(ricardo).
rapaz(roberto).
rapaz(carlos).
moca(gisele).
moca(susana).
moca(gertrudes).
```

Esse programa diz que `X` e `Y` formam um possível par se `X` é um rapaz e `Y` é uma moça. Os possíveis pares poderiam ser encontrados como:

```
?- possivel_par(X, Y).
X=joao, Y=gisele ;
X=joao, Y=susana ;
X=joao, Y=gertrudes ;
X=ricardo, Y=gisele ;
X=ricardo, Y=susana ;
X=ricardo, Y=gertrudes ;
X=roberto, Y=gisele ;
X=roberto, Y=susana ;
X=roberto, Y=gertrudes ;
X=carlos, Y=gisele ;
X=carlos, Y=susana ;
X=carlos, Y=gertrudes ;
no
```

Neste ponto, você deve estar seguro de que entende por que Prolog produz as soluções nessa ordem. Primeiro, Prolog satisfaz o objetivo `rapaz(X)`, encontrando `joao` como o primeiro rapaz. Então ele encontra `gisele`, a primeira moça. Neste ponto, outra solução é solicitada, causando uma falha. Prolog tenta ressatisfazer o objetivo que ele satisfez por último, que é o subobjetivo `moca` dentro do objetivo `possivel_par`. A alternativa susana é encontrada e, assim, a segunda solução é `joao` e `susana`. Do mesmo modo, Prolog gera `joao` e `gertrudes` como a terceira solução. Na próxima vez que tenta ressatisfazer `moca(Y)`, Prolog encontra o marcador de lugar no final da base de dados e, assim, o objetivo falha. Então Prolog tenta ressatisfazer `rapaz(X)`. O marcador de lugar para esse objetivo foi colocado no primeiro fato para `rapaz` e assim a próxima solução encontra o segundo `rapaz (ricardo)`. Depois de ressatisfazer esse objetivo, Prolog tenta satisfazer `moca(Y)` a partir novamente do início da base de dados. Assim ele encontra `gisele` — a primeira moça. As próximas

três soluções envolvem portanto `ricardo` e as três moças. Após essas soluções, quando uma alternativa for solicitada, o objetivo `moca`, novamente, não pode ser ressatisfeito. Assim outro rapaz encontrado e a pesquisa por meio das moças começa novamente a partir do início. E assim por diante. Eventualmente, o objetivo `moca` falha e também não há mais soluções para o objetivo `rapaz`. Assim o programa não pode encontrar mais nenhum par.

Esses exemplos são todos muitos simples. Eles envolvem apenas a especificação de muitos fatos ou o uso de regras para acessar esses fatos. Por causa disso, eles não podem gerar um número infinito de possíveis soluções. Algumas vezes, pode-se querer gerar um número infinito de possíveis soluções, não porque se queira considerá-las todas, mas porque não se sabe previamente quantas soluções serão necessárias. Nesse caso, precisa-se de uma definição recursiva.

Considere a seguinte definição do que é um número inteiro não negativo. O objetivo `eh_inteiro(N)` será satisfeito se `N` estiver instanciada com um número inteiro. Se `N` não estiver instanciada, então o objetivo `eh_inteiro(N)` fará com que um número inteiro seja escolhido e `N` seja instanciada com esse número inteiro:

```
eh_inteiro(0).
eh_inteiro(X) :- eh_inteiro(Y), X is Y + 1.
```

Se for feita a consulta:

```
?- eh_inteiro(X).
```

serão obtidos, como possíveis respostas, todos os números naturais em ordem ascendente (0, 1, 2, 3, ...), um de cada vez. Cada vez que um retrocesso for solicitado (digitando-se um ponto e vírgula), `eh_inteiro` será satisfeito, com seu argumento sendo instanciado com um novo inteiro. Assim, em princípio, essa pequena definição gera um número infinito de respostas, como é ilustrado a seguir.

```
?- eh_inteiro(X).
X = 1 ;
X = 2 ;
X = 3 ;
...
```

Como funciona esse programa? Inicialmente, tem-se uma escolha entre a primeira e a segunda cláusula para tentar satisfazer um objetivo `eh_inteiro(X)`. Se a primeira cláusula for escolhida, nenhuma outra escolha tem que ser feita e obtém-se `X=0`. Caso contrário, se a segunda cláusula for escolhida, tem-se uma escolha de como satisfazer o subobjetivo (novamente `eh_inteiro`) que ele introduz. Se for escolhido o fato (primeira cláusula), encerra-se a consulta

com a resposta `X=1`; caso contrário, utiliza-se a regra (segunda cláusula) e deve-se novamente escolher como satisfazer o subobjetivo produzido. E assim por diante. Em cada estágio, a primeira coisa que Prolog faz é considerar o fato. Apenas o retrocesso faz Prolog desfazer a última escolha. Cada vez que faz isso, Prolog volta para o fato (última escolha) e, em vez disso, ele escolhe a regra. Uma vez que Prolog tenha decidido utilizar a regra, um novo subobjetivo é introduzido e o fato é a primeira possibilidade de satisfazê-lo.

Muitas regras Prolog dão surgimento a soluções alternativas se elas são utilizadas em objetivos que contêm algumas variáveis não instanciadas. Por exemplo, a relação membro de uma lista:

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

pode gerar alternativas. Se for feita a consulta (note que a variável `X` na questão não está instanciada):

```
?- membro(a, X).
```

então os sucessivos valores de `X` serão listas parcialmente definidas, em que `a` é o primeiro, segundo, terceiro, etc membro. Veja se você consegue ver por que isso ocorre.

Outra consequência de se permitir que essa definição de `membro` retroceda é que a questão:

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

pode ser, na realidade, satisfeita cinco vezes.

Claramente, há algumas aplicações do predicado `membro` nas quais ele precisa ser satisfeito, se possível, apenas uma vez (quando não se deseja, por exemplo, saber o número de ocorrências de um elemento numa lista). Deve-se então descartar as outras escolhas possíveis. Pode-se dizer a Prolog para descartar escolhas indesejáveis utilizando-se o corte.

5.11 O Corte

O **corte** permite que se diga a Prolog que escolhas prévias não precisam ser reconsideradas quando ele retrocede na cadeia de objetivos satisfeitos. Há duas razões pelas quais importante fazer isso:

- O programa funcionará mais rápido porque ele não perderá tempo tentando satisfazer objetivos que se pode informar de antemão que nunca contribuirão para a solução.

- O programa poderá ocupar menos espaço de memória do computador porque alguns pontos de retrocesso não precisarão ser registrados para exame posterior.

Sintaticamente, o uso de um corte numa regra assemelha-se ao de um objetivo que tem o predicado `!` sem argumentos. Como um objetivo, ele satisfeito imediatamente e não pode ser ressatisfeito. Além disso, ele produz o efeito de alterar o modo como o retrocesso funciona. Esse efeito é tornar inacessível os marcadores de lugar para certos objetivos de modo que eles não podem ser ressatisfeitos. Será visto como isso funciona com um exemplo.

Imagine que um programa sobre biblioteca que tem uma base de dados Prolog contendo informações sobre os livros existentes, quem tomou emprestado o quê e quando os livros serão devolvidos. Uma coisa na qual deve-se estar interessado é em quais facilidades da biblioteca devem estar disponíveis para que pessoas. Algumas facilidades, que são denominadas aqui *facilidades básicas*, devem estar disponíveis para todos. Essas facilidades incluem o uso das seções de referências e de consultas. Por outro lado, a biblioteca pode querer ser seletiva sobre aqueles que podem utilizar outras facilidades adicionais, tais como tomar livros emprestados ou utilizar o serviço de intercâmbio entre bibliotecas. Uma regra que pode ser elaborada é que, se uma pessoa tiver um livro em atraso, então as facilidades adicionais não estarão disponíveis para essa pessoa até que o livro atrasado seja devolvido. Eis aqui uma parte desse programa:

```
facilidade(Pessoa, Facilidade) :-
   deve(Pessoa, Livro),
   !,
   facilidade_basica(Facilidade).
facilidade(Pessoa, Facilidade) :-
   facilidade_geral(Facilidade).
facilidade_geral(F) :-
   facilidade_basica(F).
facilidade_geral(F) :-
   facilidade_adicional(F).
facilidade_basica(referencia).
facilidade_basica(consulta).
facilidade_adicional(emprestimo).
facilidade_adicional(intercambio).

deve('J. Silva', livro12130).
deve('A. Melo', livro324419).
deve('A. Melo', livro432516).
 ...
cliente('A. Melo').
cliente('P. Pinto').
 ...
```

Por que há um corte no programa e qual o efeito que ele produz? Suponha que se deseje listar todos os clientes, verificando quais as facilidades que estão disponíveis para eles. Assim, se for apresentada a Prolog a questão:

```
?- cliente(X), facilidade(X, Y).
```

Prolog começará encontrando o primeiro cliente, `A. Melo`. Suponha que esse cliente tenha vários livros devidos. Para encontrar as facilidades disponíveis para ele, Prolog utilizará a primeira cláusula para `facilidade`. Essa cláusula introduz um novo objetivo que verifica se o cliente tem algum livro devido. Após fazer uma pequena consulta entre os fatos `deve`, o fato para o primeiro livro devido por `A. Melo` é encontrado. O próximo objetivo encontrado é o corte `!`. Esse objetivo é satisfeito e o efeito é confinar todas as decisões feitas desde que a primeira cláusula `facilidade` foi escolhida. Pode-se mostrar a situação justamente antes de o corte ser encontrado em um diagrama como o da **Figura 5–4** (a numeração entre parênteses refere-se ao número de ordem da cláusula utilizada):

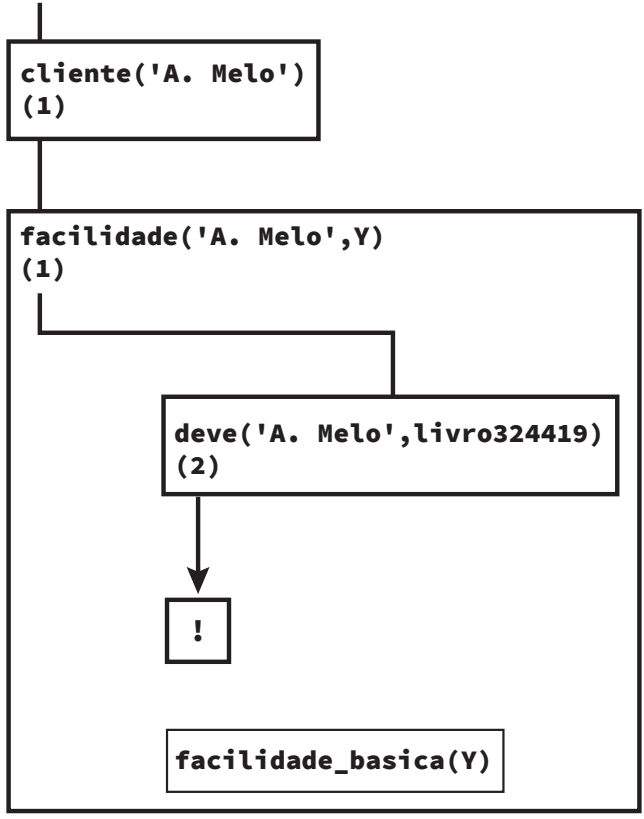

FIGURA 5–4: EXEMPLO DE CORTE EM PROLOG 1

Quando o corte é encontrado, ele corta o fluxo da linha de satisfação, de modo que se Prolog for forçado a retroceder além desse ponto, ele terá que tomar um pequeno atalho, como mostrado no diagrama da **Figura 5–5**.

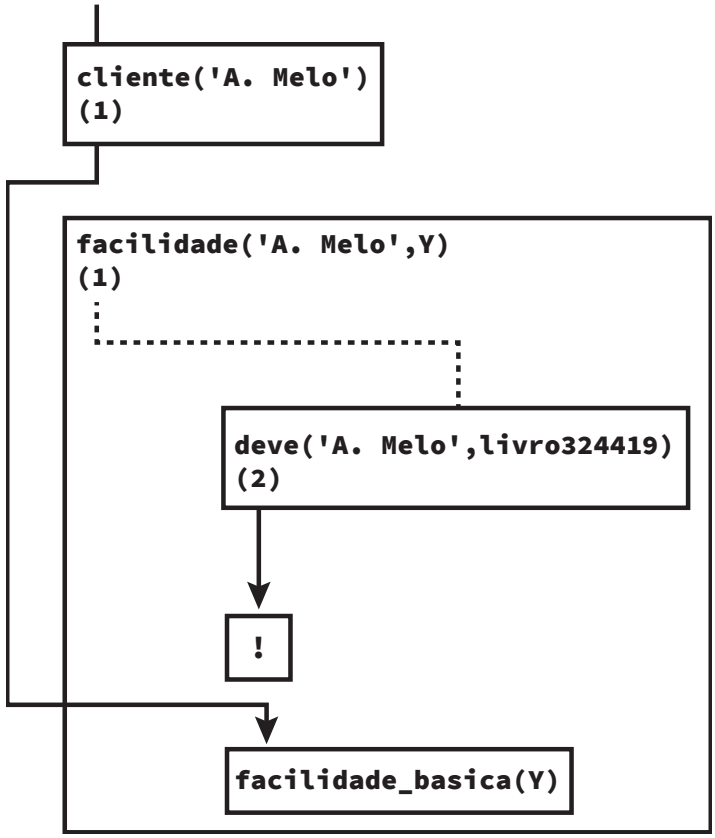

Figura 5–5: Exemplo de Corte em Prolog 2

O efeito do corte na regra `facilidade` (cláusula 1) é, portanto, congelar qualquer escolha que tenha sido feita a partir do instante em que essa regra foi considerada. O fluxo do caminho de satisfação foi modificado de modo a evitar todos os marcadores de lugar entre o objetivo `facilidade` e o objetivo `!`, inclusive. Assim, se um retrocesso posterior causar um retorno de volta a esse ponto, o objetivo `facilidade` irá falhar imediatamente. O programa não considera soluções alternativas para o objetivo `deve('A. Melo', Livro)` e isso é inteiramente razoável pois se está interessado apenas em saber se o cliente deve algum livro, e não em quais são todos os livros devidos por ele. Também, o sistema não considerar a cláusula 2 para `facilidade`, porque a escolha da regra na qual o corte aparece também é desviada no retrocesso. Isso novamente

é razoável aqui, porque não se deseja gerar soluções que digam que todas as facilidades estão disponíveis para `A. Melo`.

Em resumo, o efeito do corte nesse exemplo é informar a Prolog: se for encontrado um cliente que tem um livro devido, então permita ao cliente apenas as facilidades básicas da biblioteca. Não interessam todos os livros devidos pelo cliente e não considere qualquer outra regra sobre facilidades.

Nesse exemplo, o corte congelou todas as decisões feitas entre ele e o objetivo `facilidade`, que, nesse caso, é chamado **objetivo-pai** do objetivo corte, porque ele é o objetivo que causou o uso da regra contendo o corte. Nos diagramas vistos, o objetivo-pai é sempre o objetivo cujo retângulo é o menor envolvendo o retângulo que contém `!`. A definição formal do efeito do símbolo do corte será vista em seguida.

Quando um corte é encontrado como um objetivo, todas as escolhas feitas desde que o objetivo pai foi invocado imediatamente tornam-se congeladas. Todas as outras alternativas são descartadas. Consequentemente, uma tentativa de ressatisfazer qualquer objetivo entre o objetivo-pai (inclusive) e o objetivo corte irá falhar.

Há vários modos de se descrever o que acontece às escolhas que são afetadas por um corte. Pode-se dizer que as escolhas são **congeladas**, que o sistema **confinou** as escolhas feitas ou que as alternativas são **descartadas**. Pode-se também ver o símbolo do corte como semelhante a uma **cerca** que separa objetivos. Na conjunção de objetivos:

```
conj :- a, b, c, !, d, e, f.
```

Prolog pode retroceder normalmente entre os objetivos `a`, `b` e `c`, até que o sucesso de `c` cause o cruzamento da *cerca* para a direita até atingir o objetivo `d`. Então o retrocesso pode ocorrer entre `d`, `e` e `f`, talvez satisfazendo a conjunção inteira várias vezes. Entretanto, se o objetivo `d` falhar, causando o cruzamento da *cerca* para a esquerda, nenhuma tentativa de ressatisfazer o objetivo `c` será feita. Assim a conjunção inteira de objetivos irá falhar e o objetivo `conj` também irá falhar (independentemente de haver ou não outras cláusulas para esse objetivo).

Já foi dito que quando o corte aparece em alguma regra e o objetivo corte é satisfeito, Prolog congela todas as escolhas feitas desde que o objetivo-pai foi invocado. Isso significa que a escolha dessa regra e todas as outras feitas desde então tornaram-se fixas. Será visto adiante que é possível prover alternativas dentro de uma simples regra utilizando o predicado embutido `;` (significando *ou*). As escolhas introduzidas por essa facilidade são afetadas exatamente

do mesmo modo. Isto é, quando um objetivo corte é encontrado, todas as escolhas *ou* que tenham sido feitas desde que a regra foi escolhidas são fixadas.

Lembre-se sempre que:

> ***Em alguns casos, incluir um corte pode significar a diferença entre um programa que irá funcionar e outro que não irá funcionar.***

5.12 Usos Comuns do Corte

Pode-se dividir a utilização do corte em três áreas principais:

(1) Em situações nas quais se deseja informar a Prolog que a regra correta para um objetivo particular foi encontrada. Aqui, o corte significa:

> ***Se esse ponto foi atingido, a regra correta para esse objetivo foi encontrada.***

Situações como essas serão exploradas na **Seção 5.12.1**.

(2) Em situações nas quais se deseja instruir Prolog a forçar a falha de um objetivo particular sem tentar soluções alternativas. Aqui, utiliza-se o corte em conjunção com o predicado `fail` para dizer:

> ***Se esse ponto foi atingido, pare de tentar satisfazer esse objetivo.***

Essas situações serão vistas detalhadamente na **Seção 5.12.2**.

(3) Em situações nas quais se pretende terminar a geração de respostas alternativas por meio de retrocesso. Aqui, o corte diz:

> ***Se esse ponto foi atingido, a única solução para esse problema foi encontrada e não há nenhum sentido em procurar por alternativas.***

Situações como essas serão exploradas na **Seção 5.12.3**.

Serão vistos, em seguida, exemplos desses três usos do corte. Deve-se ter em mente, entretanto, que o corte tem um único significado em todas as três aplicações. A divisão em três áreas principais de utilização é feita por razões didáticas e para mostrar com que tipos de raciocínio você pode contar para colocar cortes em seus programas.

5.12.1 Confirmando a Escolha de uma Regra

Muito frequentemente, num programa Prolog, deseja-se associar várias cláusulas com um mesmo predicado. Uma cláusula é apropriada se os argumentos são de uma forma, outra apropriada se os argumentos são de outra forma e assim por diante. Usualmente, deseja-se especificar qual a regra que deve ser utilizada para um certo tipo de objetivo fornecendo padrões nas cabeças das regras que casarão apenas com objetivos dos tipos corretos. Isso significa fornecer regras para alguns tipos específicos de argumentos e depois fornecer uma regra de fechamento ao final que casa com qualquer outra coisa. Entretanto, esse modo nem sempre é possível. Se não se pode informar previamente quais são as formas que os argumentos podem ter, ou se não se pode especificar um conjunto exaustivo de padrões, o programa pode não funcionar conforme o desejado.

Como exemplo do que foi exposto, considere o programa seguinte. As regras definem o predicado `soma_ate`, tal que o objetivo `soma_ate(N, X)`, com `N` tendo um valor inteiro, faz com que `X` seja instanciada com a soma dos números inteiros de `1` até `N`. Assim, por exemplo, ter-se-ía:

```
?- soma_ate(5, X).
X=15 ;
no
```

pois *1 + 2 + 3 + 4 + 5* igual a *15* e sabe-se que não pode haver nenhuma outra solução alternativa. O programa em questão é:

```
soma_ate(1, 1) :- !.
soma_ate(N, Res) :- N1 is N - 1,
                    soma_ate(N1, Res1),
                    Res is Res1 + N.
```

Essa definição é recursiva. A ideia é que a condição de fronteira ocorra quando o primeiro número for igual a *1*. Nesse caso a resposta também é *1*. A segunda cláusula introduz um objetivo recursivo `soma_ate`. Entretanto, o primeiro número do novo objetivo é um a menos que o número original. O novo objetivo que esse objetivo irá invocar terá seu primeiro argumento um a menos novamente. E assim por diante até que a condição de fronteira seja atingida. Desde que os primeiros argumentos estão continuamente tornando-se menores, a condição de fronteira deve ser atingida eventualmente (assumindo que o objetivo original não tem o primeiro argumento menor que *1*) e o programa irá terminar.

O ponto interessante sobre esse programa é como são manipulados os dois casos: quando o número é *1* e quando ele é qualquer outro valor. Quando foram definidos predicados para manipulação de listas (v. **Seção 5.9**), foi fácil

especificar os dois casos que normalmente surgiriam — quando a lista estava vazia (`[]`) e quando a lista era da forma `[A|B]`. Com números, as coisas não são tão fáceis, porque não se pode especificar um padrão que case apenas com um inteiro diferente de *1*. A solução adotada nesse exemplo prover uma solução para o caso de o número ser igual a *1* e deixar uma variável casar com qualquer outro valor.

Sabe-se, pelo modo como Prolog pesquisa sua base de dados, que ele irá inicialmente tentar casar o número com *1* e tentar a segunda regra apenas se essa regra falhar. Assim a segunda regra deve ser utilizada apenas para números diferentes de *1*. Entretanto, se Prolog, sempre que retroceder, reconsiderar a escolha da regra aplicada ao número *1*, então ele achará que a segunda regra também é aplicável. Em sua interpretação, ambas as regras fornecem alternativas para o objetivo `soma_ate(1, X)`. Deve-se então instruir Prolog para não tentar a segunda regra sempre que o número for *1*. Um meio de fazer isso é colocando um corte na primeira regra (como foi feito acima). Isso informa a Prolog que, uma vez que ele tenha atingido a primeira regra ele não deve nunca refazer a decisão sobre qual regra usar para o objetivo `soma_ate`. Ele atinge essa regra apenas quando o número, de fato, for *1*.

Veja como isso funciona em termos de um diagrama de fluxo de satisfação, se `soma_ate(1, X)` invocado no contexto:

```
obj :- soma_ate(1, X), comida(banana).
?- obj.
```

e o objetivo `comida(banana)` falha. Então, no instante da falha, tem-se a situação ilustrada na **Figura 5–6**.

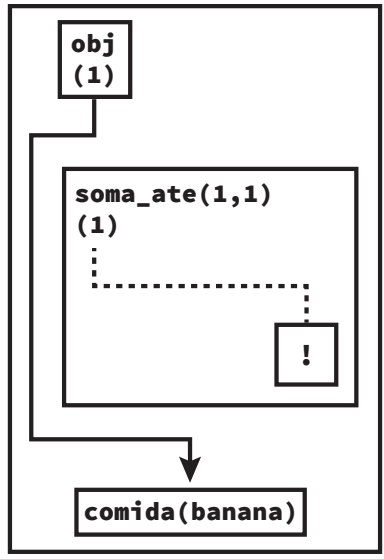

Figura 5–6: Diagrama de Fluxo de Satisfação em Prolog

Quando Prolog tenta ressatisfazer os objetivos na ordem inversa, ele conclui que dois dos objetivos não podem ser ressatisfeitos porque o caminho do fluxo de satisfação foi desviado. Consequentemente, ele evita corretamente tentar modos alternativos de satisfazer `soma_ate(1, X)`.

Exercício: O que aconteceria se o corte fosse deixado de fora aqui e o retrocesso ocorresse, de forma a considerar o objetivo `soma_ate`? Que resultados alternativos, se existe algum, seriam produzidos e por que?

O último exemplo mostrou como o corte pode ser utilizado para fazer Prolog comportar-se sensatamente quando não se pode distinguir entre todos os casos possíveis por meio da especificação de padrões nas cabeças das regras. Uma situação mais usual, na qual não se pode especificar padrões para decidir que regra utilizar, surge quando se deseja fornecer condições extras, na forma de objetivos, que decidirão sobre a regra apropriada. Considere a seguinte forma alternativa do exemplo acima:

```
soma_ate(N, 1) :- N =< 1, !.
soma_ate(N, R) :- N1 is N - 1,
                  soma_ate(N1, R1),
                  R is R1 + N.
```

Nesse caso, informa-se que a primeira regra é aquela a ser escolhida se o número fornecido for menor do que ou igual a *1*. Essa é uma formulação levemente melhor do que a formulação anterior porque o programa produz uma resposta (em vez de ser executado indefinidamente) se o número dado for menor do que

ou igual a zero. Se essa condição for verdadeira, o resultado *1* será produzido imediatamente e nenhum objetivo recursivo ser necessário. Apenas quando a condição `N =< 1` não for verdadeira, deseja-se tentar a segunda regra. Deve-se informar a Prolog que, uma vez que ele tenha provado `N =< 1`, ele não deve nunca reconsiderar a escolha sobre qual regra usar. É isso que o corte faz aqui.

Um princípio geral é que usos do corte para informar a Prolog quando ele considerou a única regra correta podem ser trocados por usos do predicado embutido `not` (os predicados embutidos em Prolog serão descritos completamente no **Capítulo 6**). O predicado `not` é definido de tal forma que o objetivo `not(X)` é satisfeito apenas quando `X`, visto como um objetivo, falha. Assim o fato de `not(X)` ser satisfeito significa que `X` não é um objetivo satisfazível em Prolog. Tomando o predicado `soma_ate` como exemplo de troca de corte por `not`, tem-se que as duas alternativas apresentadas para a definição de `soma_ate` podem ser reescritas respectivamente como:

```
soma_ate(1, 1).
soma_ate(N, R) :- not(N = 1), N1 is N - 1,
                  soma_ate(N1, R1),
                  R is N + R1.
```

e

```
soma_ate(N, 1) :- N =< 1.
soma_ate(N, R) :- not(N =< 1), N1 is N - 1,
                  soma_ate(N1, R1), R is N + R1.
```

Novamente, Prolog provê predicados convenientes para substituir esses dois usos de `not`. Por exemplo, pode-se trocar `not(N = 1)` por `N \= 1` e `not(N =< 1)` por `N > 1`. Mas, em geral, não se está apto a fazer isso em todas as situações imagináveis.

Considera-se um bom estilo de programação trocar usos de corte por usos de `not`, porque programas contendo cortes são, em geral, mais difíceis de se entender do que aqueles que não os contém. Entretanto, a definição de `not` envolve tentar satisfazer o objetivo que é seu argumento. Portanto, considerando um programa de forma geral, em que `a`, `b`, `c`, `d` estão representando objetivos quaisquer e não constantes como na sintaxe formal de Prolog:

```
a :- b, c.
a :- not(b), d.
```

Prolog poderá tentar satisfazer o objetivo `b` duas vezes. Ele deverá tentar satisfazer `b` quando considerar a primeira regra. Também, se ele retroceder e considerar a segunda regra, ele deverá tentar satisfazer `b` novamente para verificar se `not(b)` pode ser satisfeito. Essa repetição pode ser muito ineficiente

se o objetivo `b` for muito complicado. Isso não seria o caso se, ao invés disso, se tivesse usado um corte como:

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

Assim deve-se, algumas vezes, ponderar as vantagens de um programa mais legível com aquelas de outro que ser executado mais rapidamente. A discussão de eficiência conduz ao último exemplo do corte sendo utilizado para fixar a escolha de uma regra. Considere a definição de `anexa` (v. **Seção 5.9.2**):

```
anexa([], L, L).
anexa([C|L1], L2, [C|L3]) :- anexa(L1, L2, L3).
```

Se o predicado `anexa` está sempre sendo utilizado para o caso em que se têm duas listas conhecidas e se deseja saber qual lista consiste da primeira anexada à frente da segunda, pode-se achar que é ineficiente, quando ocorre um retrocesso sobre um objetivo tal como `anexa([], [a, b, c, d], X)`, tentar usar a segunda regra mesmo que essa tentativa esteja condenada a falhar. É claro que quando a primeira lista é `[]`, a primeira regra é a única correta e essa informação pode ser fornecida a Prolog por meio do corte. Em geral, as implementações de Prolog estarão aptas a fazer melhor uso da memória disponível se elas forem informadas sobre coisas desse tipo, em vez de terem que manter registros de escolhas aparentes que não existem realmente. Assim a definição de `anexa` deveria ser:

```
anexa([], X, X) :- !.
anexa([A|B], C, [A|D]) :- anexa(B, C, D).
```

Assumindo o uso restrito de `anexa`, isso não afeta absolutamente as soluções que o programa encontra. Ele apenas aumenta as eficiências de espaço e de tempo. Ao fazer essa modificação, o programador é responsável pelos outros usos de `anexa` não mais funcionarem como esperado, como será mostrado na **Seção 5.13**.

5.12.2 A Combinação corte-falha

Em sua segunda maior área de aplicação, o corte é usado em conjunção com o predicado embutido `fail` (falha). O predicado `fail` não possui nenhum argumento — o que significa que o sucesso do objetivo `fail` não depende do significado de qualquer variável. O predicado `fail` é definido de tal forma que, utilizado como um objetivo, ele sempre falha e provoca um retrocesso. Quando `fail` é encontrado depois de um corte, o comportamento normal do retrocesso é alterado pelo efeito do corte. De fato, a combinação particular **corte-falha** torna-se muito útil na prática.

Como primeiro exemplo de utilização de combinação corte-falha, considere o pequeno programa em Prolog no qual deseja-se informar a Prolog que Maria gosta de qualquer doce que não tenha sabor de anis. É fácil expressar parcialmente essa declaração (i.e., que Maria gosta de qualquer doce):

```
gosta(maria, X) :- doce(X).
```

No entanto, resta excluir o indesejável sabor de anis dentre os doces. Isso pode ser feito reformulando-se a declaração original como: se `X` tem sabor de anis então Maria não gosta de `X`, caso contrário Maria gosta de `X`. Pode-se traduzir facilmente isso em Prolog utilizando a combinação corte-falha da seguinte maneira:

```
gosta(maria, X) :- sabor(X, aniz), !, fail.
gosta(maria, X) :- doce(X).
```

Suponha que seja perguntado se Maria gosta de um certo doce `d` cujo sabor é de anis. Isto é, suponha que existam na base de dados os fatos `doce(d)` e `sabor(d, anis)`. Agora considere a seguinte consulta:

```
?- gosta(maria, d).
```

O que acontece quando Prolog tenta satisfazer esse objetivo? Primeiro, o objetivo casa com a primeira regra, com `X` sendo instanciada com `d`. Em seguida, Prolog tenta satisfazer `sabor(d, anis)`, o que ocorre de acordo com a suposição; o predicado `!` satisfeito normalmente e, finalmente, Prolog encontra o predicado `fail` que provoca uma falha, fazendo com que Prolog tente retroceder. A presença do corte, no entanto, impede esse retrocesso, fazendo com que toda regra falhe. Assim Prolog responde `no` à consulta, como seria esperado.

Outra interessante aplicação da combinação corte-falha é na definição do predicado `not`. Muitas implementações Prolog fornecem esse predicado já definido, mas é interessante observar como esse predicado pode ser definido. Requer-se que o objetivo `not(P)`, em que `P` significa outro objetivo, seja satisfeito se e somente se o objetivo `P` falha. Isso não está exatamente de acordo com a noção intuitiva de *não verdadeiro*, pois nem sempre seguro assumir que alguma coisa não verdadeira quando não se está apto a prová-la. Entretanto, usando essa definição, tem-se:

```
not(P) :- call(P), !, fail.
not(P).
```

A definição de `not` envolve invocar o argumento `P` como um objetivo, utilizando o predicado embutido `call`. O predicado `call` simplesmente trata seu argumento como um objetivo e tenta satisfazê-lo. Deseja-se que a primeira cláusula seja aplicável se `P` puder ser satisfeito e a segunda seja aplicável em

caso contrário. Assim, se Prolog pode satisfazer `call(P)`, ele deve abandonar a satisfação do objetivo `not`. A outra possibilidade é que Prolog não pode satisfazer `call(P)`. Nesse caso, ele nunca atinge o corte. Quando o objetivo `call(P)` falha, o retrocesso ocorre e Prolog tenta a segunda cláusula, que é imediatamente satisfeita. Consequentemente, o objetivo `not(P)` é satisfeito quando o `P` não satisfazível.

Como ocorre com o primeiro uso do corte, pode-se trocar usos de corte-falha por usos de `not`. Assim o programa `gosta`, visto anteriormente, poderia ser reescrito como:

```
gosta(maria, X) :- doce(X), not(sabor(X, aniz)).
```

5.12.3 Encerrando Geração-e-teste

Frequentemente, um programa tem partes que funcionam de acordo com o modelo geral descrito a seguir.

Há uma sequência de objetivos que podem ser satisfeitos de muitas maneiras e que gera muitas soluções possíveis por retrocesso. Depois dessa sequência, existem objetivos que verificam se uma solução gerada é aceitável para algum propósito. Se esses objetivos falham, o retrocesso faz com que outra solução seja proposta. Essa solução é testada para verificar se ela é apropriada e assim por diante. Esse processo irá parar quando uma solução aceitável for gerada (sucesso) ou quando nenhuma outra solução pode ser encontrada (falha). Podem-se chamar os objetivos que encontram todas as alternativas o **gerador** e aqueles que testam se uma solução aceitável o **testador**.

Será apresentado abaixo um exemplo de programa que funciona usando esse método de geração e teste. Muitos sistemas Prolog provêm a facilidade de divisão inteira, mas aqui ser apresentado um programa para divisão inteira (para inteiros não negativos apenas) que utiliza apenas adição e multiplicação:

```
divide(N1, N2, Resultado) :-
   eh_inteiro(Resultado),
   Produto1 is Resultado*N2,
   Produto2 is (Resultado + 1)*N2,
   Produto1 =< N1,
   Produto2 > N1, !.
```

Essa regra utiliza o predicado `eh_inteiro` (conforme foi definido na **Seção 5.10**) para gerar o número `Resultado` que é o resultado da *divisão* de `N1` por `N2`. Assim, por exemplo, o resultado de dividir *27* por *6* é *4* porque *4×6* menor do que ou igual a *27* e *5×6* maior do que *27*. A regra usa `eh_inteiro` como um gerador e o restante dos objetivos provê o testador apropriado. Agora, sabe-se de antemão que, dados valores para `N1` e `N2`, `divide(N1, N2, Resultado)`

pode apenas ser satisfeito para um valor possível de **Resultado**. Pois, embora **eh_inteiro** possa gerar uma quantidade infinita de candidatos, apenas um irá passar nos testes. Pode-se indicar esse conhecimento colocando um corte ao final da regra. Esse corte significa que, sempre que for gerado com sucesso um **Resultado** que passe nos testes para ser o resultado da divisão, nunca se precisar tentar mais nada. Em particular, não será preciso reconsiderar nenhuma das escolhas que foram envolvidas nas buscas das regras para **divide**, **eh_inteiro** e assim por diante. A única solução foi encontrada e não há nada que precise ser reconsiderado. Se o corte não houvesse sido colocado aqui, qualquer retrocesso eventualmente começaria a encontrar alternativas para **eh_inteiro** novamente. Assim a geração de valores possíveis para **Resultado** seria executada novamente. Nenhum desses valores seria o resultado correto da divisão e assim continuar-se-ía gerando soluções indefinidamente (mais um exemplo de repetição infinita).

5.13 Problemas com Corte

Foi visto que, algumas vezes, deve-se levar em consideração o modo como Prolog consulta a base de dados e quais estado de instanciação os objetivos terão ao decidir a ordem na qual escrever as cláusulas de um programa em Prolog. O problema com a introdução de cortes é que deve-se ter mais certeza de que se sabe como as regras do programa serão usadas, pois, embora um corte possa ser inócuo ou mesmo benéfico quando uma regra é usada de uma forma, o mesmo corte pode causar um comportamento estranho se a regra for usada repentinamente de outra forma.

Considere o predicado **anexa** da **Seção 5.12** modificado como:

```
anexa([], X, X) :- !.
anexa([A|B], C, [A|D]) :- anexa(B, C, D).
```

Quando são considerados objetivos tais como:

```
?- anexa([a, b, c], [d, e], X).
```

e

```
?- anexa([a, b, c], X, Y).
```

o corte é completamente apropriado. Se o primeiro argumento do objetivo já tem um valor, então tudo o que o corte faz é reafirmar que a primeira regra será relevante apenas se o valor do primeiro argumento é `[]`. Entretanto, considere o que acontece com o objetivo:

```
?- anexa(X, Y, [a, b, c]).
```

Esse objetivo irá casar com a cabeça da primeira cláusula, resultando em:

```
X=[], Y=[a, b, c]
```

mas, agora, o corte é encontrado. Isso congelará todas as escolhas feitas e, assim, se outra solução for requisitada, a resposta será **no**, mesmo havendo outras soluções para a questão (por exemplo, **X = [a], Y = [b, c]**).

Aqui está outro interessante exemplo do que pode acontecer se uma regra contendo um corte for utilizada de modo inesperado. Considere o predicado **numero_de_pais**, que pode expressar a informação sobre quantos pais alguém possui. Ele pode ser definido como segue:

```
numero_de_pais(adao, 0) :- !.
numero_de_pais(eva, 0) :- !.
numero_de_pais(X, 2).
```

Isto é, o número de pais é **0** para **adao** e **eva**, mas é **2** para qualquer outra pessoa. Agora, se essa definição de **numero_de_pais** sempre utilizada para encontrar o número de pais de uma dada pessoa, essa definição é boa. Obtém-se, por exemplo:

```
?- numero_de_pais(eva, X).
X=0 ;
no
?- numero_de_pais(joao, X).
X=2 ;
no
```

e assim por diante, como era esperado. Os cortes são necessários para prevenir que o retrocesso atinja a terceira regra sempre que a pessoa é **adao** ou **eva**. Entretanto, considere o que acontece se as mesmas regras são utilizadas para verificar se uma dada pessoa tem um dado número de pais:

```
?- numero_de_pais(eva, 2).
yes
```

Você deve convencer-se de que entende por que isso acontece. Isso é simplesmente uma consequência do modo pelo qual Prolog consulta sua base de dados. A implementação de *outros casos* simplesmente não funciona mais. Há duas modificações possíveis que podem ser feitas para prevenir o resultado indesejado:

```
numero_de_pais(adao, N) :- !, N = 0.
numero_de_pais(eva, N) :- !, N = 0.
numero_de_pais(X, 2).
```

ou

```
numero_de_pais(adao, 0) :- !.
numero_de_pais(eva, 0) :- !.
numero_de_pais(X, 2) :- X \= adao, X \= eva.
```

É claro que essas definições ainda não funcionariam apropriadamente para objetivos como:

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

se fosse esperado que todas as possibilidades fossem enumeradas por retrocesso.

Concluindo, a moral da estória é:

> ***Se você utiliza cortes para obter um comportamento correto quando os objetivos são de uma forma, não há nenhuma garantia de que algo sensível ocorrerá se aparecerem objetivos de outra forma.***

Segue-se que é possível usar o corte confiavelmente apenas se você tem uma política clara sobre como suas regras serão usadas. Se você modificar essa política, todos os usos do corte devem ser revistos.

5.14 Exercícios de Revisão

1. Defina o predicado **`ultimo(Lista, Item)`** que deve ser satisfeito quando a variável **`Item`** é instanciada com o último elemento da lista **`Lista`**. Por exemplo, os seguintes objetivos devem ser satisfeitos:

```
?- ultimo([a, b, c, d], d).
yes

?- ultimo([x, y, z], X).
X = z
```

2. Defina o predicado **`acrescenta(Item, L1, L2)`** que, quando satisfeito, acrescenta o argumento Item na lista L1 resultando na lista **`L2`**. Os seguintes resultados devem ser apresentados:

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

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

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

?- acrescenta(a, [b], [a, b, c]).
no
```

3. Defina o predicado `retira(Item, L1, L2)` que, quando satisfeito, retira o argumento `Item` da lista `L1` resultando na lista `L2`. Se `Item` não pertence à lista `L1`, o objetivo `retira(Item, L1, L2)` deve ser satisfeito. Exemplos de consultas a esse predicado:

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

?- retira(u, [a, b], L).
L = [a, b]

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

4. Defina o predicado `inverte(Lista1, Lista2)`, tal que quando o objetivo `inverte(Lista1, Lista2)` é satisfeito com um dos seus argumentos instanciados com uma lista, o outro argumento é instanciado com essa lista invertida. Exemplos de consultas a esse predicado:

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

?- inverte(L, [x, y]).
L = [y, x]

?- inverte([1, 2], [2, 1]).
yes

?- inverte([a, b, c], [c, a, b]).
no
```

5. Defina o predicado `maximo(X, Y, Max)`, tal que esse objetivo é satisfeito se `Max` é instanciada com o maior entre os números inteiros `X` e `Y`. As consultas seguintes devem fornecer os respectivos resultados:

```
?- maximo(2, 4, X).
X = 4

?- maximo(n, 2, X).
no

?- maximo(X, 2, Y).
no
```

6. Defina o predicado `entre(N1, N2, X)`, tal que ele é satisfeito se, dados dois inteiros `N1` e `N2`, gera, por retrocesso, todos os inteiros entre `N1` e `N2`. Exemplos de consultas feitas a esse predicado:

```
?- entre(5, 8, X).
Estao entre 5 e 8: 5, 6, 7, 8

?- entre(2, 5, 3).
3 esta entre 2 e 5

?- entre(3, 6, 9).
9 nao esta entre 3 e 6
```

7. Considere o programa Prolog:

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

Quais são todas as respostas possíveis de Prolog para cada uma das seguintes consultas (considere todas as possibilidades de ressatisfação dos objetivos por meio de `;`). Explique os resultados obtidos.

(a) `?- p(X).`

(b) `?- p(X), p(Y).`

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

8. O predicado seguinte classifica números inteiros em três classes de números: positivos, negativos e zero:

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

Redefina esse predicado, de modo mais eficiente, utilizando cortes.

9. Defina o predicado `rotacao(L1, L2)` que provoca uma rotação horária na lista `L1` resultando na lista `L2` (ou uma rotação anti-horária na lista `L2` resultando na lista `L1`). Exemplos de consultas:

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

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

10. Escreva um predicado `simplifica` para simplificar expressões aritméticas contendo somas de números e símbolos (letras minúsculas). Após a simplificação de uma expressão, as letras devem preceder os números. Exemplos de utilização desse predicado:

```
?- simplifica(1 + 1 + a, E).
E = a + 2

?- simplifica(1 + a + 4 + 2 + b + c, E).
E = a + b + c + 7

?- simplifica(3 + x + x, E).
E = 2*x + 3
```

11. Defina o predicado **acrescenta_fim(Item, L1, L2)** que acrescenta o membro **Item** ao final da lista **L1** resultando na lista **L2**. Exemplos de consultas utilizando esse predicado:

```
?- acrescenta_fim(a, [c, d, f], L).
L = [c, d, f, a]

?- acrescenta_fim(X, [c, d, f], [c, d, f, a]).
X = a

?- acrescenta_fim(a, L, [c, d, f, a]).
L = [c, d, f]
```

12. Considere a representação de conjuntos por meio listas. Defina os predicados:
    (a) **intersecao(C1, C2, C3)**, que é satisfeito se **C3** é a interseção dos conjuntos **C2** e **C3**.
    (b) **igual(C1, C2)**, que é satisfeito se os conjuntos **C1** e **C2** são iguais. (Lembre-se que é irrelevante a repetição ou a ordem dos elementos em um conjunto.)
    (c) **diferenca(C1, C2, C3)**, que é satisfeito se o conjunto **C3** é igual a **C1 - C2**.
13. Escreva um predicado **exclui** que exclui um elemento de uma lista. Esse predicado deve falhar se o elemento não for membro da lista.
14. Escreva um predicado para inserir um elemento numa lista usando o predicado **exclui** do exercício anterior.
15. Escreva um predicado que verifique se uma lista **L1** é sublista de outra lista **L2**.
16. Escreva um predicado que determine o comprimento de uma lista.
17. Defina o predicado **max_lista(Lista, Max)** que é satisfeito quando **Max** é o maior valor na lista de número inteiros **Lista**.
18. Escreva um programa Prolog que determine o maior entre dois números inteiros.