

1 Dando nome aos bois

Neste capítulo serão explicadas algumas noções mais ou menos simples, assim como definidos termos a serem utilizados posteriormente. Vale notar que alguns dos termos usados aqui também são usados em outros contextos, mas com significado diferente. As definições aqui colocadas são referentes apenas à utilização no presente documento.

É uma quantidade relativamente grande de definições que a leitora poderia, possivelmente, reconhecer de capítulos posteriores e estão postas aqui para referência. Dependendo de seu estilo, talvez considere mais interessante começar do Capítulo 1: Continue por sua conta e risco.

Programação lógica surgiu da constatação de que programas são formados por uma parte lógica e uma parte procedural e da pergunta “Dá para separar a parte lógica da procedural, de modo que quem programa tenha que se preocupar somente com a lógica do programa?”, seguida por tentativas de respondê-la. Dessa tentativa nasceu o Prolog e a teoria de programação lógica. O Prolog não foi intencionado como uma resposta final, mas como uma tentativa de resposta, a partir da qual poderiam ser feitos melhoramentos. Apesar disso, por vários motivos (que serão notados ao longo do texto), vários mecanismos não puramente lógicos (isto é, não previstos pela *teoria de programação lógica*, que veremos mais adiante) foram notados como úteis e permaneceram na linguagem contribuindo para que Prolog também tenha um sabor procedural, apesar da aparência enganadora. Se você se pergunta qual a resposta àquela pergunta inicial, é “não completamente”.

Para desenvolver mais essa ideia, precisaremos de uma quantidade relativamente grande de novos conceitos e definições. Assim, existem duas rotas possíveis para a leitura: continuar por aqui e conhecer as definições antes de usá-las, ou ir para o próximo capítulo e voltar aqui na medida do necessário, usando o capítulo atual como uma referência.

Antes de mais nada, vale lembrar que (diferentemente do que algumas pessoas podem pensar), existe uma diferença entre programação lógica e Prolog: Prolog é uma implementação (imperfeita) de programação lógica. Uma entre várias outras. Aliás, esse é um dos motivos de ser importante estudar e perceber a programação lógica como algo separado: você pode querer aplicá-la em alguma outra linguagem. Outro motivo é a abstração a mais que permite desenvolver teorias com maior facilidade e liberdade. Apesar disso, as notações usadas para Prolog e para programação lógica são frequentemente muito parecidas e não parece muito interessante explicar o que é essencialmente a mesma coisa duas vezes (já que usaremos a linguagem Prolog em diversas situações). No lugar disso, adotaremos a seguinte convenção.

As definições apresentadas serão as usadas no Prolog; quando houver diferenças relevantes entre as definições usadas em Prolog, em Programação lógica, ou entre outra teoria/linguagem utilizada, essa diferença será explicitada (isto

é, quando não for dito o contrário, as definições e convenções de linguagem são assumidas como as mesmas).

Definição 1.1. *Átomo é algo que seja uma:*

- Sequência de caracteres entre aspas simples ('Qualquer coisa assim');
- Ou uma sequência de caracteres contendo apenas caracteres alfabéticos, numéricos e underscore ('_') começando com um caractere alfabético minúsculo;
- Ou uma sequência contínua de caracteres dos símbolos: +, −, *, /, \, ^, >, <, =, ', :, ., ?, @, #, \$, &. (e.g. ***+***+@);
- Ou algum dos seguintes átomos especiais: [], {}, ;, !.

Definição 1.2. *Funtor*¹ é um átomo seguido de n argumentos fechados por parênteses e separados por vírgula:

$$p(a_1, \dots, a_n)$$

em que p é um átomo e a_1 a a_n são quaisquer termos. Quando $n = 0$, não são colocados parênteses e o funtor é apenas um átomo. Quando $n > 0$, o funtor é chamado termo composto e p é chamado funtor principal (essa definição faz sentido porque cada um dos argumentos pode, por sua vez, ser um outro funtor). Não confunda funtor com função: sua única semelhança é na forma de se escrever.

Vale notar que os funtores são diferenciados por nome e aridade (quantidade de parâmetros). Assim, os funtores a seguir são distintos:

$$f(a_1, a_2)$$

$$f(a_1)$$

Um funtor de aridade n e nome p será denotado p/n .

Definição 1.3. *Termo é um funtor (composto ou não), uma variável (o que são variáveis e como funcionam será explicado adiante), ou um número (em especial, lidaremos com números inteiros ou reais).*

Quando um termo for um átomo ou um número, diremos que ele é **atômico**.

¹O termo *funtor* foi introduzido por Rudolph Carnap, filósofo alemão que participou do círculo de Viena, em seu *Logische Syntax der Sprache* e indica uma *palavra função*, que contribui primariamente com a sintaxe de uma sentença (em contraste com *palavras conteúdo*, que contribuem primariamente com o significado): resumidamente, em sua concepção original, funtores expressam a estrutura relacional que palavras tem umas com as outras. O termo atualmente também é usado em outras áreas (além de em linguística e programação lógica), como em *Teoria de Categoria*, com significado semelhante (é claro, levando em conta o contexto).

Quando um termo não contiver uma variável, diremos que ele é um termo **base**. Termos base serão um componente importante no desenvolvimento da teoria de programação lógica mais para frente.

Com base nessas definições, podemos definir os blocos de construção da programação lógica, o que chamaremos de **cláusulas**:

- Fatos e
- Regras

Intuitivamente, fatos e regras são como axiomas matemáticos: a partir deles é possível fazer deduções.

Dos dois, fatos são mais simples. Eles são escritos como $p(a_1, \dots, a_n)$.², ou, mais resumidamente, $p.$, isto é, um funtor e seus argumentos, seguido de um ponto final “.”. Fatos podem ser lidos e entendidos como “ $p(a_1, \dots, a_n)$ é verdade”.

Regras são escritas como $p : -b_1, \dots, b_n.$, onde p denota um funtor e b_1 a b_n são termos quaisquer. Regras podem ser lidas e entendidas como **p é verdade se b_1 a b_n forem** (se ajudar, você pode ler o “:-” como uma seta de implica, \leftarrow , estilizada). É conveniente dar nomes diferentes para a parte de uma regra que vem à esquerda do “:-” da que vem à direita dele: o que vem à esquerda de “:-” (o “ p ” do nosso exemplo) será chamado *cabeça* da cláusula e, o que vem à direita, *corpo* da cláusula. Assim, a *cabeça* é verificada como verdadeira se o *corpo* o for (mas não o contrário).

Um programa lógico consiste em um conjunto de cláusulas. Um exemplo de programa lógico é o seguinte:

Código 1: Café

```
pao( de ).
pao( de , queijo ).
com( cafe ) :- tem( cafe ).
tem( cafe ) :- cafe .
cafe .
```

Note que os nomes usados para os funtores são arbitrários. O seguinte programa tem essencialmente o mesmo significado³:

Código 2: Queijo

```
queijo( lua ).
```

²Trechos escritos com **esta fonte** no meio do texto devem ser entendidos como trechos de código.

³Aqui, “Essencialmente o mesmo” significa “o mesmo”, se não levarmos em conta os significados usuais dos nomes usados (em geral, levamos isso em conta)

```

queijo(lua, pao).
coma(terra) :- gem(terra).
gem(terra) :- terra.
terra.

```

A maneira primária de se utilizar um programa lógico é por meio de buscas, às vezes também chamadas objetivos ou de *goals* (daqui para frente referidas como goals, sem itálico, que é a nomenclatura mais utilizada). Um goal é escrito como um fato e é o que se busca “provar” a partir do programa. Intuitivamente, pode-se pensar no programa lógico como um conjunto de axiomas e no goal como uma hipótese que se quer provar a partir desses axiomas. Assim, um goal, relativo a algum programa, pode ter o status de sucesso, se ele pode ser provado a partir do programa, falha se não, ou, se foi um goal mal escrito, pode gerar um erro (isto é, o goal gera um erro quando não é “compreensível” a partir da gramática utilizada pelo interpretador). Mais especificamente, um goal é uma **conjunção**, escrita como p_1, p_2, \dots, p_n , onde cada vírgula pode ser lida como um e lógico (assim, lê-se um goal como p_1 e p_2 e ... e p_n , onde p_i é entendida como uma proposição que pode ser verdadeira ou falsa⁴).

Assim, goals, assim como regras são como cláusulas de Horn⁵ (isto é, cláusulas do tipo C se A_1 e ... e A_n). Como pode imaginar, o procedimento para a prova de uma cláusula de Horn é de alguma importância para programas lógicos.

Para diferenciarmos goals de fatos, goals serão, aqui, escritos como:

$$p?$$

onde $p?$ indica a conjunção dos p_i , como explicado anteriormente, e pode ser entendida como a proposição⁶ que se quer provar. Note que essa convenção será usada aqui mas não reflete como um goal pode parecer na natureza (em particular, no Prolog padrão, assim como nas outras implementações conhecidas pelo autor, o “?” é substituído pelo “.” e um goal, assim, seria diferenciável de um fato apenas pelo contexto).

Para melhor compreensão, considere o seguinte programa:

Código 3: Árvore binária

```
arvore_b(vazio).
```

⁴Na verdade, como veremos, uma interpretação mais próxima da realidade do programa lógico é uma baseada na lógica construtivista, mas, por enquanto, é suficiente pensar em um goal como uma conjunção no sentido clássico

⁵Também, apesar de mais raramente, chamadas *cláusulas de McKinsey*. Elas foram usadas primeiramente por McKinsey, como notado por Horn (*Journal of Symbolic Logic*, Vol. 16 de 1951, página 14).

⁶É algo como uma tradição usar “funtor” e “proposição” de forma quase que intercambiável, deixando a leitora potencialmente confusa quanto à diferença (se alguma) entre os dois. Este texto não tem a intenção de quebrar tradições (em particular, não essa), confiando que isso não lhe apresentará maior dificuldade no entendimento.

```

arvore_b(arvore(A, D, E)) :-
    arvore_b(D),
    arvore_b(E).

```

O goal *arvore_b(vazio)?* tem sucesso, já que é um dos fatos. O goal *arvore_b(arvore(raiz, vazio, vazio))?* também tem, já que *arvore_b(a, B, C)?* tem sucesso se *arvore_b(A)* e *arvore_b(B)* tiverem, o que ocorre. Perceba que, neste caso, o goal não tem sucesso segundo a primeira cláusula apenas e nem segundo a segunda. Ambas contribuem para a definição de *arvore_b*.

Utilizamos a convenção de que termos capitalizados (isto é, com inicial maiúscula) denotam variáveis, enquanto os demais denotam constantes (a leitora descobrirá que, por incrível coincidência, o Prolog faz uso da mesma convenção).

No geral, o que um programa lógico deveria fazer (isto é, o que a programadora tem em mente ao escrevê-lo) pode não ser a princípio claro. Numa tentativa de aliviar isso, usaremos aqui a convenção de usar nomes significativos, assumindo o significado usual (a não ser quando dito o contrário), para os elementos relevantes. Assim, por exemplo, o seguinte trecho:

Código 4: Pai e filho

```

filho(c, b).
pai(Pai, Filho) :- filho(Filho, Pai).

```

Indica que Pai tem a relação *pai/2* com Filho se Filho tem a relação *filho* com Pai (em outras palavras, um Pai é pai de um Filho de Filho é filho de Pai). Mas não é assumido que o interpretador do programa tenha conhecimento sobre o que é *pai* ou *filho*, ou seja, essa interpretação é relevante para o leitor do programa apenas (o que também significa que o leitor do programa ler *pai* e a *filho* de várias formas e um programa pode, assim, ser interpretado de mais de uma forma).

Com essa discussão em mente, será ocasionalmente útil ter o *significado* de um programa lógico definido de forma algo mais precisa:

Definição 1.4. *Significado de um programa lógico é o conjunto de goals deduzíveis a partir dele (isto é, os goals que obtêm sucesso se aplicados ao programa).*

Na verdade, essa é a definição procedural do significado de um programa lógico (mas veremos posteriormente que essa distinção é imaterial) e, vale notar, ela é sempre bem definida (isto é, todo programa lógico tem um significado bem definido). Nesse contexto, o significado intencionado pela programadora (isto é, “o que ela quer dizer com o programa”) é um conjunto de goals que pode ou não estar contido no significado do programa. Assim, podemos nos perguntar “Será que o programa diz tudo que se quer que ele diga?” (isto é, se o significado intencionado está contido no significado do programa), ou “Será que tudo o que ele diz é correto?” (isto é, se o significado do programa está contido no

intencionado). No primeiro caso, se o significado intencionado estiver contido no do programa, dizemos que o programa é *completo* e, no segundo, que ele é *correto*.

Por exemplo, digamos que o programa *x* seja um trecho de um programa no qual a programadora deseja modelar as relações entre programas e especificações de software atuais e antigos, de forma que *A* é pai de *B* se *A* veio antes de *B* e *B* herda características de *A*, como partes do código ou especificações (como é possível ver, ela apenas começou a escrever o programa). Assim, estão no significado intencionado goals como `pai(gnu, linux)?` e `pai(dos, windows)?`, enquanto que o significado do programa é `pai(d, c)?` (esse programa é correto? e completo?).

Leituras adicionais: