0 Dando nome aos bois

Foi a influência da filosofia grega que fez da matemática uma ciência. De fato, os primeiros matemáticos gregos estão entre os primeiros filósofos, como é o caso de Tales e Pitágoras. A noção de que um fato matemático pode ser demonstrado é fruto da interação entre matemática e filosofia. Afinal, uma demonstração é essencialmente um argumento para esclarecer como um certo fato é consequência de algo que já conhecemos. E se há alguma coisa que os filósofos gregos gostavam de fazer era argumentar.

S. C. Coutinho - Números Inteiros e Criptografia RSA

Nosso objetivo final é aprendermos a resolver e desenvolver problemas de otimização por restrições lógicas. Para tanto, precisaremos de algumas ferramentas, talvez a mais importante das quais, que será a base de outras, é a programação lógica. Programação lógica é um paradigma de programação, como o de programação funcional, a de programação procedural, entre outros, mas, em princípio, lidaremos com ela como a linguagem Prolog (de *"Programmation Logique"*). Isso rende a vantagem de lidarmos com algo concreto, enquanto perdemos pouco do poder de abstração. Um detalhe que vale notar é que existem vários diferentes "sabores" de Prolog, cada um com suas peculiaridades, na maior parte de natureza técnica. Ignoraremos essas peculiaridades sempre que possível (este texto não tem a intenção de ser um texto técnico sobre Prolog), nos focando, sempre que possível, no "padrão de facto" Edinburgh Prolog.

Como dizia Sussman [3] , quando se quer aprender uma nova linguagem, as perguntas básicas que precisamos perguntar são

1. Quais são as "coisas" primitivas da linguagem?
2. Quais são seus meios de combinação, com os quais podemos usar as primitivas de maneira estruturada?
3. Quais são seus meios de abstração, com os quais podemos usar programas menores para criar programas maiores?

A resposta à primeira pergunta é que o Prolog tem uma primitiva: o **funtor**. **funtor**
O termo *funtor* foi introduzido por Rudolf Carnap, filósofo alemão do círculo

de Viena, em seu *Logische Syntax der Sprache*[1] e indica uma *palavra função* (não confundir com "uma 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, em essência, semelhante.

Para nossos propósitos, é uma simplificação razoável dizer que funtores expressam relações. Na linguagem natural, somos restritos a usar os funtores presentes na língua usada (mesmo uma "licença poética" não vai muito longe disso). Na programação lógica, os funtores só precisam ser sintaticamente corretos. Em Prolog, um funtor f de aridade n (aridade é a quantidade de parâmetros, ou "símbolos relacionados pelo funtor"), dito `f/n`, é escrito com uma sintaxe semelhante à de uma função:

`f(`a_1`, ..., `a_n`)`

em que os argumentos a_i são outros funtores. Em particular, símbolos e números são ditos funtores de aridade zero, também chamados **átomos***. Quando um funtor `f/n` não é um átomo, ele, na presença de seus argumentos, é dito um **termo composto** de **funtor principal** f. Mais em geral, um **termo** é um funtor ou variável (o que é uma variável e como se comporta veremos mais para frente) e, quando corresponder a um átomo ou uma variável, dizemos que ele é atômico.

átomos

termo composto

funtor principal

termo

Dados funtores `f/3` e `p/1`, exemplos de seus usos são:

- f(a,b,c)
- f(1,a,8)
- f(p(9),c,e)

Os exemplos acima são úteis como primeiro exemplo de "como escrever um funtor" mas, como colocados, não têm significado. Isto porque funtores expressam relações, mas relações arbitrárias entre símbolos e números arbitrários não têm necessidade de existir. A maneira de dizermos que uma dada relação existe em Prolog é por meio de um **fato**. Um fato em Prolog é expresso como um funtor, junto de seus argumentos, seguido de um ponto. Por exemplo

fato

```
mais(1,2,3).
```

diz que "é verdade que mais(1,2,3)" ou, mais naturalmente, "é verdade que existe a relação 'mais' entre 1, 2 e 3, nessa ordem" (vale notar que a ordem é importante: `mais(1,2,3)` não é o igual a `mais(3,2,1)`, em que por "igual

*Em Prolog, nem todo símbolo pode ser tratado como átomo (embora os usuais sejam). Quando na dúvida, recomendamos testar antes de confiar.

a" entende-se "idêntico a"). Ocasionalmente, é muito mais conveniente que um funtor receba argumentos pela direita e pela esquerda, quase como um operador, como `1 f 2`. É possível definir funtores dessa forma, mas nos restringiremos ao modo usual, pelo qual a última relação fica `f(1,2)`.

Eventualmente pode ser que, na ocasião de a relação `p(a,b)` ser verdadeira, outras relações também sejam. Em particular, pode ser que, em linguagem matemática, $f(1, l) \Rightarrow p(a, b)$. Esse tipo de relação é escrita em Prolog como:

```
p(a,b) :- f(1,l).
```

ou, se $(f(1, l)$ e $q(f)) \Rightarrow p(a, b)$, escrevemos

```
p(a,b) :- f(1,l), q(f).
```

na qual as vírgulas entre os termos fazem o papel do "e" lógico. Esse tipo de estrutura é chamada **regra** e é o mais importante meio de combinação em programação lógica. O que está para a esquerda de ":-" em uma regra é dita "cabeça da regra", ou só "cabeça", se a regra for clara do contexto, e o que está à direita de ":-" é dito o "corpo da regra". Quando não for necessário distinguir entre fatos e regras, usaremos o termo **cláusula*** para nos referir a qualquer um dos dois.

regra

cláusula

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

programa lógico

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

Código 1: Café

Note que os nomes usados para os funtores são arbitrários. O seguinte programa tem essencialmente o mesmo significado do ponto de vista computacional:

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

Código 2: queijo

Pelo código 1, por exemplo, podemos dizer que "é verdade que pao(de,

*Note que usamos "cláusula" em um sentido diferente do da lógica clássica (em que "cláusula" é definida como sendo uma disjunção de proposições).

queijo) e que com(cafe)". De fato, a pergunta mais geral que podemos fazer a um programa lógico é se existe alguma relação r entre os termos a_1, ..., a_n.

E essa constitui a maneira primária de se utilizar um programa lógico, que pode ocorrer, por exemplo, em um *prompt* no computador. Por meio de questões, ou buscas [*], às vezes também chamadas objetivos ou "**goals**" (daqui para frente preferiremos usar "goal", sem itálico[†], exceto quando for conveniente utilizar "busca" ou "objetivo"). 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.

goals

Assim como dizemos que uma busca foi um sucesso se foi encontrado o que se gostaria de encontrar e uma falha caso contrário, diremos que um goal relativo a algum programa pode ter o status de sucesso se ele pode ser provado a partir do programa, e de falha se não. Note que, em Prolog, "falhar" não é o mesmo que "quebrar". A diferença é essencialmente a mesma entre receber uma resposta de "não" a uma pergunta e receber uma resposta de "não entendo sua pergunta". Se foi um goal mal escrito (se o compilador "não entende a pergunta"), pode ocorrer um erro (isto é, o goal gera um erro quando não é "compreensível" a partir da gramática utilizada pelo compilador usado)[‡] e o programa pode não ser executado.

Mais especificamente, um goal é uma conjunção[§], escrita como $p\_1, p\_2, ..., p\_n$, em que p_i é entendida como uma proposição que pode ser verdadeira ou falsa [¶] (ou, interpretando como busca, que pode ter sucesso ou falha).

Um ponto interessante sobre goals e também sobre regras é que são como cláusulas de Horn[‖] (isto é, cláusulas do tipo C se A_1 e ... e A_n). Como pode imaginar, a teoria existente sobre cláusulas de Horn é de alguma importância para programas lógicos. Mais especificamente, dada uma disjunção[**] com ao menos um termo não negado, esta é dita uma *cláusula de Horn*. Se existe exatamente um termo não negado, a seguinte identidade é de simples constatação:

$$X_1 \vee \neg X_2 \ldots \neg X_n \Leftrightarrow [(X_2 \wedge \ldots X_n) \Rightarrow X_1]$$

em que o símbolo $\neg$ corresponde à negação lógica usual.

[*] Porque busca é um termo apropriado deve ficar cada vez mais claro no decorrer do texto.

[†] Lembre-se que, em geral, usaremos termos em língua estrangeira, em itálico.

[‡] Se um goal que gera erro é de fato um goal, é discutível, mas não entraremos nesse mérito.

[§] Isto é, uma sequência de proposições unidas por um *e* lógico. Algo como `p_1` e ... e `p_n`, em que os `p_i` são proposições.

[¶] Na 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.

[‖] Também, apesar de mais raramente, chamadas *cláusulas de McKinsey*. Elas foram usadas primeiramente por McKinsey, como notado por Horn [2]. É importante notar que a "cláusula" usada aqui não é a mesma "cláusula" definida anteriormente, mas a "cláusula" da lógica clássica

[**] Proposições unidas por "ou".

Para melhor diferenciarmos goals de fatos, goals serão, aqui, escritos como: `p?`

apesar de, na natureza*, não haver essa distinção (neles, goals são, como fatos, escritos com um ponto final no lugar de com um ponto de interrogação). Na verdade, a falta de distinção entre goals, fatos e termos de uma cláusula, como deve ficar mais claro nos próximos capítulos, faz sentido, já que no processo de resolução (que será explicado depois) eles cumprem papel semelhante.

Agora, considere o seguinte código:

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

Código 3: Árvore Binária

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 sempre a convenção de que termos capitalizados (isto é, com inicial maiúscula) denotam variáveis, enquanto os demais denotam constantes (a leitora descobrirá que, por notável coincidência, o Prolog faz uso da mesma convenção). Termos que não contém variáveis são ditos **termos base**. Variáveis serão explicadas mais a fundo no futuro.

termos base

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:

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

Código 4: Pai e Filho

Indica que Pai tem a relação *pai* com Filho se Filho tem a relação *filho* com Pai (em outras palavras, um Pai é pai de um Filho se 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 a leitora do programa apenas (o que também significa que a leitora do programa pode

*Isto é, no Prolog padrão.

ler *pai* e *filho* de várias formas e um programa pode, assim, ser interpretado de diversas maneiras).

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

Definição 0.1. ***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).*

Significado de um programa lógico

Na verdade, essa é a definição procedural do significado de um programa lógico (apesar de que, como veremos posteriormente, 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**.

completo

correto

Por exemplo, digamos que o programa 4 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)?`.

Fica a pergunta: esse programa é correto? E ele é completo?

Leituras adicionais

[1] Carnap, Rudolf, Logische Syntax der Sprache, Wien (Viena): Julius Springer (1968)

[2] Horn, Alfred, On Sentences Which are True of Direct Unions of Algebras, J. Symbolic Logic 16 (1951), no. 1, pp 14

[3] Abelsson, Harold and Sussmann, Gerald J. and Sussman Julie Structure and Interpretation of Computer Programs, second edition MIT Press