Revista de Sistemas de Informação da FSMA

n. 30 (2022) pp. 2-14

SISTEMAS DE INFORMAÇÃO

http://www.fsma.edu.br/si/sistemas.html

TUTORIAL

# Introdução às Linguagens Lógicas Usando Prolog

Ricardo Linden
Faculdade Católica Salesiana de Macaé

***Resumo* – Neste tutorial apresentamos uma visão inicial do paradigma lógico, seguido de uma exemplificação dos conceitos aplicados à linguagem Prolog (com sintaxe de acordo com o SWI-Prolog). Os principais conceitos teóricos de linguagens de programação são então discutidos no escopo da linguagem Prolog.**

***Palavras-chave* — Linguagens de Programação; Paradigma Lógico; Linguagens Lógicas; Prolog;**

***Abstract* – In this tutorial we present an initial view of the logical paradigm, followed by an instantiation of its concepts as implemented by the Prolog language (with the syntax as defined by SWI-Prolog). The main theoretical concepts associated with programming languages are then discussed in the scope of the Prolog programming language.**

***Index terms* — Programming Languages; Logical Paradigm; Logical Languages; Prolog;**

## I. INTRODUÇÃO

A maioria dos profissionais de computação acha que não existe nada na vida além de seus mundos imperativos e orientados a objetos, mas este artigo busca mostrar que a realidade é bem diferente destes antolhos que nós colocamos em nós mesmos.

O paradigma lógico é um que normalmente é malvisto pelos estudantes. A maioria dos professores o ensina em uma ou duas aulas, dá uma enorme ênfase à matemática do cálculo de predicados (que vamos ver aqui) e ensina os fundamentos de Prolog, mostrando dois ou três exemplos básicos (que vamos ver aqui também). Os alunos saem com uma noção de que aquilo é um exercício puramente acadêmico e totalmente inútil para a vida prática (algo que vamos desmentir aqui!).

Na verdade, o paradigma lógico é algo extremamente importante e disseminado. Primeiro, não existe apenas Prolog, mas várias outras linguagens, como Mercury[1], Curry[2] (que é um híbrido de linguagem lógica e funcional) e LogTalk[3], entre outras.

Segundo, existem diversas aplicações práticas do paradigma lógico. Por exemplo, a Alitalia usou um sistema feito em Prolog para roteamento de sua frota. Além disso, no final dos anos 2000, Prolog era usado para controlar uma um sistema ferroviário [1]. A conclusão óbvia é que aplicações críticas e em larga escala podem ser desenvolvidas usando o paradigma lógico e este não é apenas mais um brinquedinho acadêmico.

Entretanto, não é necessário ir muito longe para encontrar aplicações do paradigma lógico. O autor deste artigo desenvolveu um sistema especialista para diagnóstico de falhas em linhas de transmissão (Silva, 2009). Apesar de sistemas especialistas não serem considerados uma linguagem de programação pela maioria dos pesquisadores, seus conceitos são extremamente similares àqueles das linguagens de programação lógicas e o desenvolvimento de aplicações em ambos segue processos similares de engenharia de conhecimento.

Para tanto, este trabalho está estruturado em seis seções, além da introdução aqui apresentada. Na Seção 2, nós iniciamos com o mecanismo básico de execução de "programas" em linguagens lógicas. Na seção 3, nós seguimos para uma explicação dos conceitos mais básicos da lógica de predicados (o fundamento teórico das linguagens de programação lógicas).

## II. CONCEITOS BÁSICOS DE PROGRAMAS

O paradigma lógico é uma maneira de resolver problemas muito diferente daquela que estamos acostumados com os paradigmas imperativo e orientado a objetos. Nestes dois, nós temos que dizer tudo que o programa deve fazer e ele segue nossas ordens (daí a maldição da informática: os programas fazem o que a gente manda, não aquilo que queremos que eles façam).

O paradigma lógico é bem diferente. Ele faz parte de uma classe de paradigmas chamados de *declarativos*. Nos paradigmas declarativos, nós descrevemos o problema e adicionamos informações. A resposta é calculada pela linguagem de programação seguindo um processo próprio, supondo que temos informações suficientes para resolver o problema.

Para entender este contexto, vamos imaginar que temos um problema para determinar quais pessoas são corruptas, sabendo que temos a regra que diz que "todo político é corrupto".

Em uma linguagem imperativa, nós teríamos que ter uma estrutura de dados que armazena o nome das pessoas e suas profissões. Depois, teríamos que criar uma estrutura que iterasse por toda esta estrutura e retornasse aqueles cuja profissão é político. Em Python, nós teríamos o programa dado na listagem 1.

Em uma linguagem declarativa, nós simplesmente declaramos fatos e regras e depois fazemos uma pergunta para

1 http://www.mercurylang.org/
2 http://www-ps.informatik.uni-kiel.de/currywiki/
3 http://logtalk.org/

a linguagem. Como ele encontra a solução não é um problema nosso - nós somos usuários das respostas apenas.

Assim, tudo que precisamos fazer em Prolog é declarar a existência das pessoas, as chamadas proposições lógicas (ou, em português bem simples, os fatos). Depois, nós inserimos a regra que diz que todo político é ladrão e pedimos para o Prolog nos dizer quem são as pessoas que roubam. A listagem 2 mostra como implementar isso em Prolog[4].

Olhando este exemplo, percebe-se claramente que o trabalho "pesado" de iterar por todos os fatos e descobrir todos aqueles que satisfazem uma ou mais regras é feito pela linguagem de programação, sem que nós tenhamos que intervir ou mesmo conhecer o seu mecanismo.

É uma maneira bem diferente de resolver problemas e que pode estar causando alguma confusão ou perplexidade. Vamos continuar a ver este mecanismo, mas antes, precisamos ver um pouco da teoria relacionada a este paradigma.

```
profissoes = {"Ricardo":"Professor", "Fulano":"Politico", "Beltrano":"Politico", "Camoes": "Escritor"}

for nome in profissoes.keys():

    if profissoes[nome] == "Politico":

        print "A pessoa {0} e corrupta".format(nome)
```

Listagem 1 – Programa em Python para escrever que todos os políticos são corruptos.

```
assert(pessoa(ricardo,professor)).

assert(pessoa(fulano,politico)).

assert(pessoa(beltrano,politico)).

assert(pessoa(camoes,escritor)).
assert(ladrao(X):-pessoa(X, politico)).
forall(ladrao(X), writeln(X)).
```

Listagem 2 – "Programa" em Prolog para escrever todos os políticos corruptos. Não existe uma série de comandos como na listagem 1, mas sim uma sequência de fatos e regras, seguida de uma pergunta. Nós não controlamos a execução em si. A palavra `forall` faz parecer que se está buscando o resultado, mas ela só está lá para que o Prolog diga o nome de todos os ladrões existentes. Se a pergunta fosse `ladrao(X)`, ele responderia apenas o nome do primeiro ladrão.

## III. LÓGICA DE PREDICADOS

Os conceitos fundamentais da lógica matemática são amplamente ensinados em cursos da área de computação e, por conseguinte, não serão revisados aqui. Aqueles que necessitarem uma revisão podem buscar as referências [3] e [4].

A lógica proposicional trabalha com proposições (daí o seu nome, obviamente), que são sentenças lógicas valoradas. Por exemplo, "Sócrates é um homem", "O campeão brasileiro deste ano foi o Palmeiras" ou "O presidente do Brasil é o Juquinha".

Como sabemos de nossos estudos anteriores, cada uma destas afirmações pode ser avaliada de forma a se obter se um valor verdadeiro ou falso. Ao final de 2022, as duas primeiras poderiam ser avaliadas como verdadeiras e a terceira como falsa.

Sabemos também combinar sentenças usando os diversos operadores lógicos, como E, OU, NÃO, SE..ENTÃO e assim por diante. Assim, podemos gerar uma grande variedade de outras proposições e representar uma ampla gama de conhecimento.

O problema é que queremos também representar conhecimento genérico, que representa sentenças lógicas genéricas, para diversos valores possíveis. Por exemplo, "Todo homem é mortal". Usando a lógica proposicional tradicional, não posso usar este fato e o primeiro dos anteriores para concluir que "Sócrates é mortal", pois neste tipo de lógica, os operadores genéricos "Todo" e "Existe" não foram definidos.

Assim, precisamos de uma nova forma de representar conhecimento, que nos é fornecida através dos predicados. Um predicado nada mais é que uma sentença lógica que possui variáveis. Ao substituirmos estas variáveis por valores dos seus domínios obtemos proposições, às quais aplicamos as regras tradicionais do cálculo proposicional. Obviamente, fica mais fácil com um exemplo, que vemos na figura 1.

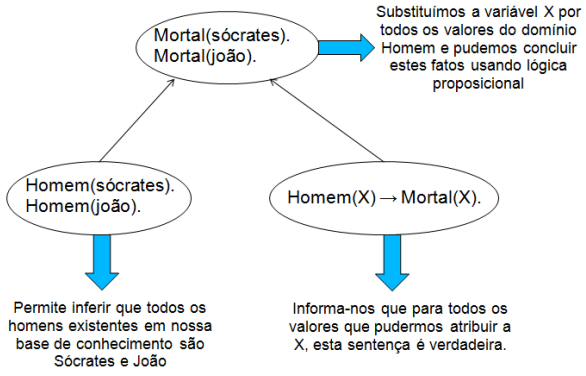


Figura 1. Exemplo de aplicação dos conceitos básicos de proposição e predicado. Existe uma razão para usar os nomes começando com letra minúscula, que será vista mais adiante neste trabalho.

Note que a figura 1 já colocou a sentença "Sócrates é homem" em um formato mais "computacional", "`Homem(socrates)`". Este é o formato padrão das linguagens lógicas, contendo o que se chama de um *functor*, o símbolo que representa o significado da relação (no caso, Homem) e uma lista ordenada de parâmetros

[4] Os códigos em Prolog foram testados no ambiente SWI Prolog (disponível no endereço http://www.swi-prolog.org/download/stable).

A lista de parâmetros tem que ser ordenada pois existe um significado implícito neste ordenamento. Por exemplo, estabelecemos que `Gosta(João, Maria)` significa que João gosta de Maria. Se quiséssemos dizer que Maria também gosta de João, então incluiríamos outros fato na nossa base, `Gosta(Maria, João)`. Note que estes dois fatos representam coisas diferentes (e mesmo que você interprete de maneira invertida, isto ainda é verdade). Logo, a ordem dos parâmetros importa.

Estas proposições que vimos até aqui são chamadas de proposições simples ou atômicas, porque elas contêm apenas um tipo de fato. Um segundo tipo de proposição é aquele chamado de proposição composta, que combina duas ou mais proposições simples usando os conectores lógicos que conhecemos da lógica proposicional, que são:

- Negação (~ ou ¬);
- Disjunção (∪ ou ∨), representando o operador OU;
- Conjunção (∩ ou ∧), representando o operador E;
- Equivalência lógica (≡ ou ⇔), implicando que as proposições A e B são equivalentes (operador SE SOMENTE SE);
- Implicação (⊃ ou →), indicando que a proposição da esquerda implica a da direita.

Até aí, parece tudo igual à lógica que estamos acostumados. Entretanto, a lógica de predicados permite acrescentar dois novos elementos que aumentam imensamente a capacidade do nosso sistema lógico: o elemento "para todo" (∀) e o elemento "existe" (∃).

O elemento "para todo" é a chamada proposição universal, isto é, algo que é sempre verdade. Por exemplo, "Todo político é ladrão" ou "Todo homem é mortal". Em "logiquês", estas sentenças correspondem a:

```
∀x(político(x) → ladrão(x))
```

```
∀y(homem(y) → mortal(y))
```

Traduzindo, estamos dizendo que existe um domínio de discurso (por exemplo, todos os políticos conhecidos) e para todos os valores pertencentes a este domínio (representados pela variável `x`), a implicação lógica é verdadeira.

Nós já vimos como as linguagens lógicas representam este tipo de coisa. Na prática, a notação delas é bem mais simples: elas representam estas universalidades da mesma forma que as proposições simples, somente trocando um valor fixo por uma variável. Assim, como vimos antes, estas duas proposições exemplo se tornam:

```
ladrao(X) :- politico(X).
```

```
mortal(Y) :- homem(Y).
```

Já estamos usando aqui a notação típica do Prolog que diz que quando um termo começa com letra maiúscula, então ele representa uma variável.

Quando uma linguagem lógica como o Prolog encontra este tipo de afirmação, ele procura substituir pelo quantificador mais apropriado e na hora que as consultas são feitas, faz as buscas pela base de conhecimento para dar as respostas viáveis.

O outro elemento da lógica de predicados é "existe", representado pelo símbolo ∃. Ele quer dizer que existe ao menos uma instância dentro da nossa base de conhecimentos que satisfaz algum tipo de condição. Por exemplo, "existe um autor deste livro que gosta de usar camisas polo". Se este livro tivesse um milhão de autores e apenas um usasse camisas polo, ainda assim esta expressão seria verdadeira. Formalmente, esta ideia é descrita pela seguinte expressão

∃X(autor(livro, X) ∧ usaPolo(X)).

Em Prolog, ela é dada pela seguinte expressão:

```
autor(livro1,X), usaPolo(X).
```

Note que neste caso nós só queremos um único caso (afinal, estamos falando da expressão "existe"). Assim, diferentemente do exemplo da seção 2, usamos apenas a condição com uma variável e o Prolog nos responderá a primeira instância da variável X que satisfaz esta condição.

Vamos discutir um pouco mais a sintaxe do Prolog mais à frente, mas já podemos ver diretamente por este exemplo que a conjunção (o E lógico) é dada através do uso da vírgula.

Usando nossa base de conhecimento e nossas regras quantificadas, podemos fazer inferências, como vimos na figura 1. Em linguagens lógicas como Prolog, isto é feito declarando-se um fato em que um ou mais parâmetros são uma variável, como por exemplo `mortal(X)`, para que atribuísse os valores de `X`.

O processo que permite esta atribuição e faz com que a linguagem lógica possa dar uma resposta a uma consulta é chamado de **unificação**. Para realizá-lo, a linguagem precisar **instanciar** cada uma das variáveis de acordo com os valores em sua base de conhecimento. Por exemplo, imagine que nossa base de conhecimento fosse dada por:

```
assert(autor(livro1,ricardo)).
```

```
assert(autor(livro1,flavio)).
```

```
assert(autor(livro1, vitor)).
```

```
assert(usaPolo(ricardo)).
```

No caso da pergunta acima sobre autores que usam camisas polo, o Prolog instanciaria a variável `X` para o valor `ricardo`, pois é o único valor possível que satisfaz a consulta

Cada linguagem lógica tem sua estratégia de unificação e é possível usarmos uma linguagem sem ter nenhuma noção sobre como este processo é realizado. Algumas linguagens fornecem opções para que o desenvolvedor controle o processo, mas isto é um conceito que foge do escopo deste artigo.

## IV. HISTÓRICO DAS LINGUAGENS DE PROGRAMAÇÃO LÓGICAS

Em 1943, o matemático J. C. C. McKinsey desenvolveu o conceito de cláusulas lógicas de primeira ordem, mas a álgebra de um subconjunto destas ideias só foi bem desenvolvida por Alfred Horn em 1951. Daí, o nome deste conceito passou a ser cláusulas de Horn.

As cláusulas de Horn eram um conceito poderoso em lógica. Elas são um conjunto de sentenças no formato dado por:

$$A_i \leftarrow A_0 \wedge A_1 \wedge ... \wedge A_{i-1}$$

Estas cláusulas podem ser usadas como elemento algébrico fundamental para se obter programas lógicos. Entretanto, isto era feito apenas por matemáticos.

A situação continuou igual até 1965, quando John Alan Robinson introduziu o conceito de resolução, uma técnica de prova automática de teoremas.

Com o advento de uma técnica automática de inferência, a ciência da lógica foi revolucionada. Muitos pesquisadores começaram a correr atrás de refinamentos desta técnica e outros começaram a tentar aplicá-la em sistemas de perguntas e respostas, como o QA3 de Cordell Green, desenvolvido em 1969 [5].

Nesta época, ainda havia grande dificuldade em se lidar com a incrível complexidade de sistemas dedutivos. Lembre-se de que em nossa cabeça nós fazemos uma filtragem e escolhemos as regras e variáveis mais apropriadas, mas o espaço de busca, havendo *n* regras e fatos, é proporcional a *n!*. E um computador tem problemas com bom senso para poder limitar as buscas.

Várias pessoas trabalharam na criação de técnicas para resolução e entre eles estava o professor Robert Kovalski, de Edimburgh. Em 1972 ele foi a Marselha, onde se reuniu com os professores Alain Colmerauer e Philippe Roussel e estes, usando os conhecimentos e arcabouço teórico do professor Kovalski, desenvolveram a primeira linguagem do paradigma lógico, Prolog.

O Prolog ficou um pouco na obscuridade, sendo usado mais em faculdades e áreas específicas de processamento (mais na prova automática de teoremas) até o começo da década de 80. Nesta época, o governo japonês decidiu adotá-la como a linguagem do seu projeto de 5ª geração computacional. Isto causou uma grande comoção e um interesse redobrado pelas áreas de lógica e inteligência artificial [6].

Pouco tempo depois, a Borland lança a sua implementação comercial, o Turbo Prolog. Obviamente, esta não foi a única implementação comercial, surgindo nesta época inclusive novas versões que trabalhavam com paralelização [7], uma demonstração de que se estava pensando em utilizações de grande porte da linguagem.

Ademais, a popularidade da linguagem e do paradigma lógico em geral era tão grande que foi fundada a Association for Logic Programming, cuja missão era contribuir para o desenvolvimento do paradigma e era capitaneada por alguns dos maiores pesquisadores da área naquela época.

Também nesta época, graças à sua disseminação, vários pesquisadores começam a estudar a possibilidade de criar linguagens híbridas, de Prolog e mais uma linguagem imperativa. Todos queriam atender necessidades específicas e foram sendo criadas várias versões e, por conseguinte, vários dialetos da linguagem.

Finalmente, em 1995, Prolog sofre uma padronização pela ISO, o que significa que todas as implementações passam a ter um núcleo comum, que vamos discutir neste capítulo. Entretanto, esta afirmação não deve ser pensada de forma peremptória pois alguns sites se dedicam a provar que não necessariamente as novas versões de Prolog seguem o padrão da ISO[5].

Conforme o paradigma lógico foi se tornando mais popular, surgiram outras linguagens que o implementavam, já citadas anteriormente neste capítulo. As mais populares buscam misturar algum outro paradigma para melhorar as capacidades da linguagem ganhando com isso vantagem em alguns pontos. Por exemplo, Mercury e Curry são híbridos de linguagem lógica e funcional, enquanto LogTalk é uma linguagem lógica orientada a objeto. O próprio Prolog recebeu várias extensões que são tão diferentes e mais capazes que são consideradas por muitos como novas linguagens (CLP(R) e CLP(Q) são exemplos).

Note que o fato destas extensões se considerarem novas linguagens pode ser a justificativa que seus implementadores dão para não respeitar o padrão ISO. Isto é só uma hipótese, mas ainda assim, use programas em Prolog ISO com cuidado nestas novas extensões.

Hoje em dia, várias implementações de Prolog oferecem interfaces para utilizar em conjunto com as linguagens C, Java e outras bem populares. Com isto, um desenvolvedor pode utilizar o melhor dos dois mundos e ter grande funcionalidade.

## V. INTRODUÇÃO À LINGUAGEM PROLOG

Nesta seção nós vamos ver alguns dos conceitos fundamentais do Prolog. Não temos espaço aqui para ver a linguagem completamente, mas existem livros que podem ajudar os leitores interessados a dominar esta linguagem, como por exemplo [8] e [9].

Ressaltamos que a escolha de Prolog foi devida ao fato de que esta é a linguagem mais famosa do paradigma. Existem outras, já mencionadas neste artigo, que podem ser úteis e que

[5] Por exemplo, ainda que um tanto antigo, temos o site http://www.cs.unipr.it/~bagnara/Papers/Abstracts/ALPN99a.

têm outras características. Entretanto, acreditamos que as principais ideias do paradigma e as noções de programação no mesmo são totalmente cobertas por este tutorial introdutório.

Vamos então começar a entender os fundamentos desta linguagem, começando pelo seu elemento fundamental, os fatos.

### V. I FATOS

Como dissemos antes, Prolog é uma linguagem declarativa. Isto quer dizer que nós apenas declaramos fatos e regras e a linguagem se encarrega de realizar inferências e responder nossas perguntas. Se quisermos, o procedimento que a linguagem usa para fazer isto pode permanecer um mistério[6].

Um fato é dado pela seguinte sintaxe:

```
functor(parametro1, parametro2, .....)
```

*Functor* é o nome do fato que estamos definindo. Ele usualmente é um verbo ou um adjetivo, representando o conceito que queremos definir. A lista de parâmetros tem o tamanho necessário e é ordenada, isto é, cada posição tem um significado específico (que nós definimos, de acordo com a semântica que desejarmos).

Como você pode perceber pelos exemplos nas seções acima, não podemos simplesmente digitar esta sintaxe em uma linha de comando Prolog. Na verdade, precisamos usar o comando `assert`. Assim, a sintaxe da declaração de um fato é dada por:

```
assert(functor(parametro1, parametro2, .....))
```

Alguns exemplos de declaração de fatos são os seguintes:

```
assert(ehVerao).
assert(mortal("Socrates")).
assert(gosta(maria, jose)).
assert(fezGols(messi, 34, espanhol)).
```

O primeiro exemplo nos mostra que a lista de parâmetros é opcional (ou, olhando de outra forma, podemos dizer que temos zero parâmetros neste caso). Quando não temos parâmetros, não precisamos usar os parênteses.

A questão da semântica implícita na ordenação dos parâmetros que colocamos fica mais fácil se olharmos os exemplos acima. Olhe para o quarto exemplo. Nós criamos um fato chamado `fezGols` cujo intuito é dizer quantos gols (segundo parâmetro) um atleta (primeiro parâmetro) fez em um campeonato (terceiro parâmetro). Logo, nós estamos dizendo que o Messi fez 34 gols no campeonato espanhol e não que o 34 fez espanhol gols no campeonato do Messi.

A mesma coisa vale para o terceiro fato. Nós estamos dizendo que Maria gosta de José e não vice-versa nem tampouco que os dois se gostam. A regra é dada por nós, e não pelo Prolog. Não tem nada ali dizendo que não pode ser o contrário (José gosta de Maria). Logo, cabe a nós manter a interpretação coerente em todos os fatos e regras que declaramos.

Aqui, reiteramos a ideia de que a legibilidade é um atributo fundamental do trabalho de um bom programador. Assim, o ideal é que seja mantida uma documentação completa sobre o significado de cada predicado para que amanhã um novo membro de uma equipe não o interprete "ao contrário". Uma das formas de fazê-lo é usando comentários que em Prolog, como em muitas linguagens, vêm em dois formatos: os comentários de linha (iniciados com o símbolo %) e os comentários de bloco (cercados pelos símbolos `/*` e `*/`). Posto que você provavelmente vai armazenar sua base de conhecimento (fatos e regras) dentro de um arquivo, a colocação de comentários no mesmo pode ajudar futuros desenvolvedores e usuários a descobrir qual é a interpretação correta de cada predicado.

Um ponto importante a ressaltar é que o Prolog usa palavras começadas em maiúsculas para definir variáveis, que são muito úteis nas regras que vamos ver na próxima seção. Assim, definimos `maria` e `jose` no terceiro fato usando minúsculas. Entretanto, se é imprescindível para seu sistema usar maiúsculas, pode cercar seus parâmetros por aspas. Neste caso, você está informando à linguagem que aquele parâmetro é uma literal do tipo String e não uma variável.

É claro que fatos podem mudar durante a execução de um programa. Neste caso, nós podemos querer retirar um fato da memória. Para tanto, basta usar o comando `retract(<fato>)`, onde `<fato>` é aquele que queremos "esquecer" (ou retratar, se vamos usar o jargão dos programadores em Prolog). Por exemplo, imagine que declaramos os seguintes fatos na nossa memória:

```
assert(pai(vader, luke)).
assert(pai(vader, lea)).
assert(pai(vader, ricardo)).
```

Se executarmos em um ambiente Prolog o comando `forall(pai(X,Y),writeln(pai(X,Y))).`, teremos como saída que Darth Vader é pai da Princesa Léa, do Luke e do autor deste tutorial. Entretanto, esta informação está ligeiramente equivocada, pois eu não tenho parentesco com a realeza de Alderaan. Assim, precisamos retirar o terceiro fato da memória, o que é conseguido com o comando:

```
retract(pai(vader, ricardo)).
```

Agora, a repetição do comando `forall` gerará apenas os dois resultados corretos esperados.

[6] Apesar de que conhecer o mecanismo pode lhe dar vários *insights* sobre a programação lógica. Ademais, a maioria das linguagens lógicas têm formas de alterar este procedimento. Não vamos explorar isto neste texto pois vai além dos objetivos propostos de apenas introduzir os conceitos fundamentais da linguagem.

### V.II REGRAS

Na seção anterior nós discutimos como incluir fatos na base de conhecimento do Prolog. Fatos são importantes, mas eles não permitem inferências e generalizações - afinal, eles consistem em descrições específicas. Para poder falar mais do mundo (e da solução dos nossos problemas, é claro), nós precisamos de regras.

As regras são afirmações do tipo SE..ENTÃO, que incluem duas partes: as cláusulas ou condições a serem satisfeitas (o lado do SE) e a conclusão possível (a parte do ENTÃO). Um exemplo simples é:

```
mortal(socrates):- humano(socrates).
```

Esta regra diz que se Sócrates é humano, então ele é mortal. Sim, o Prolog pede que coloquemos a conclusão primeiro, o que é terrivelmente confuso no começo. Entretanto, isto é assim pois o Prolog é uma implementação das cláusulas de Horn (vide seção IV). Entretanto, não se preocupe com isto: como todas as idiossincrasias de linguagem de programação, você acabará se acostumando com isso.

Neste momento, você está com a nítida impressão de que estamos trapaceando. Afinal, a regra acima nada mais é do que uma maneira alternativa de dizer que Sócrates é mortal e este não é nosso objetivo quando usamos uma regra - nós queremos, na realidade, oferecer formas de inferência generalizada, isto é, regras que se apliquem a todos e não apenas a uma pessoa específica.

Para poder fazer isto, nós usamos variáveis, como no exemplo a seguir:

```
mortal(X):- humano(X).
```

Agora, estamos dizendo que se X é humano, então `X` é mortal, onde X podem receber qualquer valoração unificável com um fato pertencente à base de conhecimento. Ou seja, no fundo, isto é equivalente à sentença lógica $\forall X, humano(X) \rightarrow mortal(X)$, que é nossa velha conhecida dos cursos de lógica e que entendemos bem.

O fato de usarmos a mesma variável no antecedente e no consequente não é coincidência. As variáveis são iguais porque elas representam a mesma valoração, isto é, no caso desta regra, representam a mesma pessoa.

Não há limitação do número de variáveis. Elas representam os valores possíveis que podem existir dentro de uma regra - se em dois pontos precisamos dos mesmos valores, usamos a mesma variável. Se são valores possivelmente distintos, então usamos variáveis diferentes.

Para entender este conceito, imagine que temos a afirmação de que homens casados são mais felizes. Ela pode ser colocada em Prolog da seguinte maneira:

```
mais_feliz(X):- homem(X), casado(X,Y).
```

O que esta regra quer dizer é o seguinte:

- Existe um homem qualquer (que é representado pela variável `X`, usado no fato `homem(X)`);
- Este homem é casado com uma outra pessoa (representada pela variável `Y`, usada no fato `casado(X,Y)`);
- Note que usamos novamente `X` no fato `casado(X,Y)` pois queremos saber se o mesmo indivíduo que sabemos ser homem (cláusula anterior), é casado. Ou seja, neste caso, temos que ter o mesmo valor.
- Note que não é obrigatório que `Y` seja diferente de `X` (se tivermos na base o fato `casado(joao, joao)`, então ele será aceito como cláusula antecedente desta regra). Ao usar variáveis diferentes, só determinamos que os valores não necessariamente são iguais, mas não que eles sejam necessariamente diferentes.

Uma conclusão pode ser obtida de várias formas diferentes. Por exemplo, podemos achar que a definição de uma pessoa X divertir-se consiste em jogar bola ou ir ao cinema com uma pessoa Y que X considera uma boa companhia. Isto seria expresso pelas seguintes regras:

```
diversao(X):- jogarBola(X).
diversao(X):-                cinema(X,Y),
boaCompanhia(X,Y).
```

Ao criarmos duas regras para a mesma conclusão, nós definimos um OU implícito (isto é, uma das duas basta para chegarmos à conclusão).

Da mesma forma, podemos ter regras que são encadeadas para chegarmos a uma conclusão. Por exemplo, imagine que não existam fatos específicos que indiquem se uma pessoa é uma boa companhia, mas sim regras. Poderíamos ter então:

```
boaCompanhia(X,Y) :- casados(X,Y).
boaCompanhia(X,Y) :- namorados(X,Y).
boaCompanhia(X,Y) :- amigos(X,Y).
```

Ou seja, basta ter um fato indicando que `X` é casado com `Y` e automaticamente inferimos que `X` considera `Y` uma boa companhia. Assim, os fatos `cinema(joao, maria)` e `casados(joao, maria)` poderiam nos permitir inferir que João se divertiu (depois de inferirmos que João considera Maria uma boa companhia).

Como colocado no início desta seção, não vamos oferecer aqui um curso completo de Prolog. Acreditamos que estes conceitos preliminares são suficientes para que você entenda como esta linguagem opera e fazer pequenos experimentos com linguagens lógicas. As referências que demos são bastante completas e vão ensiná-lo a programar em Prolog de forma completa.

O que nos falta neste momento é um entendimento mais completo do funcionamento por trás destas linguagens. Como elas realizam as inferências e respondem às nossas consultas? Para completar, então, este entendimento, vamos

ver na próxima seção algumas questões importantes da teoria de implementação de linguagens lógicas.

### V.III Conceitos de Linguagens de Programação Aplicados a PROLOG

Os livros de linguagens de programação [6][10] discutem vários aspectos definidores de linguagem, que as diferenciam e as caracterizam. Assim, para realizarmos um trabalho mais completo no sentido teórico, seria interessante ver como eles se aplicam ao Prolog.

Nesta seção, vamos ver uma aplicação prática dos conceitos que permeiam todas as referências da área de paradigmas de linguagens de programação. Obviamente, por limitação de espaço, não vamos discutir todas as características do Prolog, mas apenas entender de uma forma mais genérica como os conceitos teóricos mais relevantes na área se aplicam nesta linguagem de programação específica.

#### V.III.A Tipos de Dados

Prolog aceita constantes que são usadas para nomear objetos ou relacionamento. Existem dois tipos de constantes: átomos e números.

Os átomos são elementos que podem ser formados apenas por letras e números (começando com minúsculas, senão são variáveis), como podemos ver em cada definição de fato que fizemos neste capítulo. Por exemplo, ao definirmos o fato `pai(vader, ricardo)`, nós definimos três constantes: o nome do *functor* (`pai`) e os dois parâmetros do fato (`vader` e `ricardo` ).

Outro tipo de átomos são as strings, que são representadas por qualquer coisa entre aspas simples (')[7]. Quando precisar definir valores os fatos ou para as regras que comecem com maiúsculas, basta colocá-los entre aspas que eles passam a ser considerados átomos do tipo String. Por exemplo, poderíamos definir o fato `pai('Vader', 'Ricardo')`.

O outro tipo de constante são os números, que podem ser inteiros ou de ponto flutuante (aceitando, inclusive, a notação e, como em 2e-5, que representa o número $2x10^{-5}$).

Prolog aceita também a definição de estruturas de dados, que são definidas de forma idêntica aos fatos. Por exemplo, se quisermos definir uma empresa, poderíamos ter o seguinte fato definido:

```
(assert (empresa(nome('Companhia X'), diretor(nome('João da Silva'), idade(50)))).
```

Basicamente, o que este exemplo mostra é que a definição de fato é recursiva. Isto é, note que podemos ter dentro de um fato algo que se fosse colocado dentro de um `assert`, viraria um fato (`nome('Companhia X')`, por exemplo).

Note-se que apesar das estruturas serem devidamente nomeadas, os fatos em Prolog ainda são estritamente posicionais. Ou seja, uma consulta do tipo `(empresa(X, nome('Companhia X'))).` retorna `false`, pois não existe nenhum fato que de *functor* `empresa` tenha `nome("Companhia X")` na segunda posição (na nossa definição anterior, está na primeira).

Notem também que Prolog não tem identificadores reservados. Logo nós podemos inserir o fato `(assert(assert(2)).` em nossa base de conhecimento e depois fazer a consulta `(assert(X))`. Como em todas as outras linguagens em que isto é permitido, isto diminui a legibilidade dos programas e deve ser desencorajado.

#### V.III.B Variáveis

Como já vimos na seção anterior, em Prolog as variáveis são usadas dentro de regras e consultas para representar algum elemento genérico.

No caso das regras, as variáveis representam o equivalente dos símbolos ∀ (para todo) ou ∃ (existe) da lógica de predicados, dependendo da sintaxe da regra. Por exemplo, na regra `boaCompanhia(X,Y) :- amigos(X,Y)` definida na seção anterior, nós queremos dizer que para todo par de valores tais que estes dois elementos são amigos, é possível inferir que um gosta da companhia do outro.

No caso das consultas, ele representa aquilo que queremos saber e cujas possíveis instâncias serão retornadas pela linguagem. Por exemplo, se tivermos uma consulta do tipo `boaCompanhia(joao,Y)`, nós estamos querendo saber todos os valores possíveis que são considerados boas companhias pelo João.

Variáveis são definidas através de sua primeira letra. Se a palavra começa com maiúscula, então ela é uma variável. Senão, é tratada como um átomo. Ademais, variáveis não têm tipos, assumindo o tipo de cada instanciação realizada. Por exemplo, seja a base de fatos dada por:

```
assert(fato1(1234)).
assert(fato1('string qualquer')).
assert(fato1(meu_atomo)).
```

Se nós fizermos a consulta `fato1(X)` ao Prolog, ele retornará primeiro um número, depois uma string e depois um átomo[8], sem nenhum tipo de erro de sintaxe.

Prolog também permite que usemos as chamadas variáveis

[7] Note-se que Prolog é uma linguagem que tem um *standard*, mas isto não quer dizer que as implementações a obedeçam. Por exemplo, usamos o SWI-Prolog para testar os exemplos deste artigo e o implementador desta versão resolveu que *strings* são definidas por aspas, não *plics*. Mantivemos a definição do *standard* para sermos mais genéricos.

[8] A maioria dos Prologs retorna um valor de cada vez e espera por uma tecla pressionada pelo usuário. Se você teclar um ponto-e-vírgula, ele mostra o próximo. Em todos os outros casos ele interrompe a consulta. É possível (ainda que improvável) que seu Prolog se comporte diferente. Consulte o manual para ver como proceder.

não nomeadas, representadas pelo símbolo de sublinhado (_), que só estão lá para marcar posição (isto é, não queremos saber seu valor, mas sim se a regra é verdadeira para o caso em que haja um valor qualquer lá). Por exemplo, na nossa base de fatos acima, se fizemos a consulta `(fato1(_))`, receberemos como resposta o valor `true`, pois nós não estamos interessados no valor (dissemos isto ao usarmos uma variável não nomeada), mas sim se existe um `fato1` qualquer.

V.III.C Amarrações

Amarrações consistem em associação entre entidades de programação. Usualmente, estas associações consistem em atribuir tipos e valores a variáveis, entre outras características. Este conceito é muito importante, pois o tempo de amarração altera a flexibilidade oferecida ao usuário. Assim, se você quiser analisar o Prolog em relação à sua flexibilidade, devemos conhecer os seus tempos e regras de amarração.

Como vimos na seção anterior, variáveis em Prolog não têm tipos. Estes são atribuídos em cada instanciação de regra. Ou seja, uma variável pode mudar de tipo de acordo com os fatos que são usados para fazer as inferências ou responder às consultas.

Os valores de uma variável só valem também dentro da instanciação específica. Quando "usamos" a regra pela segunda vez, não existe nenhuma lembrança da designação de valor anterior. Ademais, se temos duas regras que usam a variável `X`, por exemplo, não existe nenhuma relação entre os valores assumidos por esta variável nestas regras.

Neste sentido, podemos dizer que cada regra é um ambiente de amarração estanque, uma espécie de escopo de cada variável.

A situação complica-se ainda mais se levarmos em consideração o *backtracking* (veja seção VI.2, adiante). Afinal, ele ocorre quando não é possível mais ter sucesso em instanciar uma regra. Logo, todas as atribuições de valor da variável feitas entre o momento do *backtracking* e o ponto para o qual voltamos são devidamente perdidas.

Aliás, é importante dizer que nem mesmo o nome da variável é aquele que imaginamos. Nós podemos usar as mesmas variáveis em regras diferentes, mas o Prolog representa internamente as variáveis com nomes únicos. Não nos interessa muito, como usuários e desenvolvedores da linguagem, quais são estes nomes (até porque eles podem ser escolhidos de forma distinta por diferentes desenvolvedores), mas o nome de uma variável é como a marca de fantasia de uma empresa. Todo mundo a conhece pela marca, mas por trás (na nota fiscal da empresa e na representação interna do Prolog), ela tem um nome complicado e, possivelmente, bem diferente.

V.III.D Estruturas de dados

Prolog admite o uso de listas para armazenamento de dados. Listas são uma estrutura muito comum na área de programação que consistem em uma sequência ordenada de dados.

Para montar uma lista, podemos simplesmente usar o operador ponto que monta uma lista a partir de dois elementos, como por exemplo:

```
.(1,[ ]).
.(a,.(b,.(c,[]))).
```

O primeiro predicado monta uma lista com um único elemento (`[1]`), enquanto o segundo monta uma lista com 3 elementos (`[a b c]`).

Notem que os exemplos deste artigo foram todos testados e funcionam na implementação gratuita SWI-Prolog (versão 7) e nesta implementação foi decidido substituir o operador ponto pela versão mais direta de representação, o símbolo de *pipe* (|). Assim, em SWI-Prolog, teríamos os seguintes comandos para os exemplos acima:

```
(1 | []).
(a | b | c | []).
```

Por que ela é mais direta? A resposta é simples: por que Prolog permite que tratemos listas da mesma maneira que as linguagens funcionais (usando, inclusive, modelos similares de recursão). Assim, podemos tratar uma lista como sendo, por exemplo `[H | T]` e `H` será a cabeça (*head*) da lista, isto é, seu primeiro elemento e `T` será a cauda (*tail*), isto é, todos os elementos restantes (no caso de uma lista de um único elemento, o *tail* será igual a uma lista vazia, `[]`).

Não cabe neste artigo uma descrição completa de toda a sintaxe. Apenas queremos mostrar as capacidades existentes no Prolog. Para mais detalhes sobre como todo o processo de uso de listas funciona, o leitor interessado deve buscar as referências [8] e [9].

V.III.E Estruturas de repetição

Prolog é uma linguagem do paradigma lógico, cuja ideia é implementar os conceitos de lógica de predicados descritos na seção IV. Desta forma, não faz sentido ter comandos do tipo `while` ou `for`.

Lembre-se de que nós colocamos os fatos e as regras dentro do sistema e o motor de inferência interno realiza os processos (como descrito na próxima seção deste capítulo).

Como colocamos na seção anterior, o Prolog permite que façamos alguns tipos de comparações recursivas, de forma semelhante às linguagens funcionais. Um exemplo tradicional, tirado de [9], é o seguinte:

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

Neste caso, estamos fazendo uma recursão e temos duas regras:

- O caso base (a primeira regra), que diz que um elemento pertence a uma lista se ele estiver na cabeça (primeira posição) da lista;
- O caso recursivo (a segunda regra), que diz que um

elemento pertence a uma lista se ele pertencer à cauda da lista (a lista sem a sua cabeça). Esta regra só é ativada se a primeira não o é e trivialmente o Prolog realiza o controle de não chamar novamente nenhuma das duas regras quando a lista está vazia (não tem cabeça, então não pode ativar a primeira regra, nem cauda, não podendo ativar a segunda).

É possível que isto seja complicado para quem não está acostumado com este paradigma. Logo, a melhor ideia seria aprofundar um pouco no estudo das linguagens do paradigma funcional, pois os conceitos colocados lá são exatamente iguais a estes que colocamos aqui.

Se fôssemos discutir a questão do ponto de vista purista, poderíamos dizer que esta questão não se aplica a linguagens lógicas, pois representa quase que uma hibridização de linguagens funcionais dentro do Prolog. Mas, em termos práticos, esta fronteira entre paradigmas não existe e aqueles que definem as linguagens e suas implementações estão mais interessados em acrescentar funcionalidades úteis, e não discutir se algo é puramente funcional ou puramente lógico.

### V.III.F Modularização

Existem funções em Prolog, mas elas não têm a aparência que estamos acostumados. No caso do Prolog, elas consistem em uma sequência de predicados que são executados em ordem e o retorno será simplesmente um valor lógico, que corresponderá ao AND dos resultados de todos os predicados dentro daquela regra.

Por exemplo, crie um arquivo chamado funcao.pl e coloque dentro dele a seguinte linha[9]:

```
aconteceu(Evento):- write('Entre o ano: '),read(Data),evento(Data,Evento).
```

Agora, na linha de comando do seu Prolog, coloque o comando `[funcao].`, assumindo que estamos no diretório onde gravamos o arquivo acima.

Esta função foi carregada e está pronta para uso. Podemos então incluir os predicados que correspondem à nossa base de conhecimento de fatos históricos:

```
evento(1822, 'Independencia').
evento(1889,'Republica').
```

Agora, estamos prontos a perguntar algo para esta função, executando a seguinte linha:

```
aconteceu('Republica').
```

O Prolog vai escrever na tela a mensagem e perguntar uma data. Se digitarmos 1889. (o ponto é necessário), ele responderá `True`. Qualquer outra data seguida de ponto retornará `False`.

Existe uma maneira de retornar algo diferente de um valor lógico. Para tanto, fazemos com que um dos parâmetros de nosso predicado seja algo a ser determinado. Por exemplo, imagine que chamamos o nosso arquivo `funcao.pl` após acrescentar a seguinte linha:

```
add(X, Y, Z) :- Z is X+Y.
```

Agora, podemos executar o predicado:

```
add(2,3,Z).
```

E receberemos o resultado `Z=5`. É quase como um retorno de função nas linguagens imperativas.

O uso de modularização mais comum em Prolog, entretanto, consiste na criação de módulos com predicados que podem ser reutilizados. Para tanto, a linguagem permite a criação de módulos que tenham uma interface muito bem definida, listando os predicados que ela exporta para que outros módulos possam usar.

Para definir um módulo, tudo que precisamos é colocar os predicados em um arquivo e iniciá-lo com a seguinte diretiva:

```
module(<nome>, <predicados_visíveis>).
```

No caso desta diretiva, `<nome>` consiste no espaço de nomes no qual as funções serão declaradas e `<predicados_visíveis>` consiste nas diretivas que serão vistas por todos aqueles que usarem aquele módulo. Todas as diretivas que não estiverem naquela lista serão privadas do módulo em si e não serão visíveis para qualquer outro espaço de nomes.

Para que outro espaço tenha acesso às diretivas daquele módulo, basta usar a seguinte diretiva:

```
use_module(<arquivo_do_módulo>).
```

A partir daquele momento, todas as diretivas do módulo estarão disponíveis para uso.

### V.III.G Tratamento de erros

Prolog permite que lidemos com erros através do lançamento de exceções (comando `throw`) e o seu devido tratamento (definido através de um predicado `catch`).

Os conceitos básicos relativos ao tratamento de exceções podem ser vistos no seguinte exemplo[10]:

```
quadrado :- catch(process, X, error_process(X)).
error_process(notNumber) :- write('Por favor digite um numero'), nl, quadrado.
error_process(X) :- write('Outro erro' : X), nl.
```

[9] Comando adaptado do site de Andrew Vartanian, encontrado em https://wiki.colby.edu/display/~amvartan/Functions+in+Prolog (último acesso em 25/11/2022)

[10] Exemplo adaptado do site de Andrew Vartanian, encontrado em https://wiki.colby.edu/display/~amvartan/Exception+and+Error+Handling+in+Prolog (último acesso em 25/11/2022)

```
  process:- write('Entre com um numero: '),read(N1),checkNumero(N1),
  V is N1*N1, write(V).
  checkNumero(N):- number(N).
  checkNumero(N):- throw(notNumber).
```

Para executar este exemplo, nós digitamos `quadrado` na linha de comando. Ao fazer isto, nós dizemos que vamos executar `process` (primeiro parâmetro do `catch`) e que se acontecer uma exceção `X` (segundo parâmetro do `catch`), nós executaremos o predicado `errorProcess`.

Note que nós estamos tratando toda e qualquer exceção. Se quiséssemos tratar apenas uma exceção específica, bastaria substituir a variável `X` por uma exceção específica. Prolog tem várias exceções pré-definidas e nós podemos ainda definir as nossas (como veremos daqui a poucas linhas).

O nosso processo em si é simples: lemos um número e checamos se ele é realmente um número (usando o predicado que definimos, `checkNumero`). Se ele for um número, então continuaremos o processamento, atribuindo o quadrado do valor entrado à variável `V`, que será depois impressa na tela.

O predicado `checkNumero` é interessante. Ele recebe um parâmetro e, se ele for numérico (primeiro caso), nada acontece. Se ele não for numérico (caso em que não ocorre o casamento com a primeira opção), ele lança uma exceção que nós definimos.

Para definir uma exceção não há qualquer necessidade de pré-processamento. Ao usar o predicado `throw` já está implícito que se está lançando uma exceção. Prolog não é estaticamente tipada[11] em nenhum contexto e o tratamento de erros não é exceção.

Note que o predicado `error_process(notNumber)` chama novamente a função quadrado. Não há nenhum tipo de recursão aqui - quando lança uma exceção, o fluxo original é interrompido. O que nós estamos fazendo aqui é simplesmente recomeçar, dando uma instrução ao nosso usuário.

Mais uma vez, não é nossa intenção ser completos em relação ao Prolog. Só queremos mostrar que se pode lidar com erros nesta linguagem usando o mecanismo de exceções explicado em detalhes na referência [10]. Quem quiser saber mais sobre as especificidades da linguagem, pode consultar uma das referências mencionadas ao fim da seção anterior.

[11] Aqui usei o termo estaticamente tipada onde muitos usarão fortemente tipada. Como haverá uma controvérsia, coloco aqui a diferença colocada neste texto: "estaticamente tipada" é aquela linguagem que define os tipos de antemão e não em tempo de execução, enquanto que "fortemente tipada" como uma linguagem que previne qualquer tipo de erro em seu sistema de tipos. Por exemplo, C é estaticamente tipada, porém não é fortemente tipada pois é possível causar um erro de tipo com union ou com void*. Mas é fato que esse conceito de "fortemente tipada" tem vários significados diferentes quando se procura na Internet, inclusive este, de ser sinônimo de "estaticamente tipada".

## VI. Como as Linguagens Lógicas Operam

Vamos agora analisar como as regras funcionam, isto é, o mecanismo usado para efetivamente obter conclusões.

Cada vez que fazemos uma *query* (pergunta) ao Prolog, ele faz uma busca dentro da base de fatos. Para realizar esta busca de forma bem-sucedida, ele precisa realizar duas tarefas específicas: a unificação, que consiste em ligar (*bind*) as variáveis a valores específicos e o *backtracking*, que consiste em desistir de uma busca quando é impossível obter uma resposta por aquele caminho.

Nesta seção nós vamos entender melhor estes dois conceitos. Para tanto, vamos começar com a teoria da unificação, que é o nome formal para o casamento de padrões efetuado pelo Prolog e por todas as linguagens que usam o paradigma lógico.

### VI.1 Unificação

Esta seção deveria estar no começo do artigo. Afinal, ela vai explicar para você como funcionam os algoritmos que fazem as linguagens de programação lógicas funcionarem. Entretanto, ela envolve uma matemática relativamente difícil (mesmo eu tendo pulado grande parte dos conceitos mais difíceis) e assim, considerei que ela poderia servir como uma espécie de apêndice ao conhecimento visto aqui.

Assim, se você tiver interesse em se aprofundar no paradigma lógico, leia esta seção como uma introdução necessária aos formalismos e à matemática envolvida.

Formalmente, dizemos que unificação é o processo algoritímico de resolver expressões lógicas. A frase complicada quer dizer apenas que temos um processo metódico (um algoritmo) de transformar expressões lógicas que contêm variáveis (como regras ou expressões genéricas) em termos que contêm apenas valores simbólicos (constantes). Por exemplo, na figura 1 nós fizemos uma unificação entre a variável X e os valores `joão` e `sócrates` para usar a regra colocada e determinar que estes dois eram mortais (gerando assim dois fatos diretos, sem variáveis, que são `mortal(joão)` e `mortal(sócrates)`).

O algoritmo de unificação segue a seguinte lógica: dois termos são unificados (isto é, considerados iguais) se eles são iguais ou se eles possuem variáveis que podem ser instanciadas de forma regular (isto é, única dentro daquela unificação) de tal maneira que os termos se tornem iguais.

O primeiro caso é relativamente óbvio. Afinal, se duas coisas são totalmente iguais, então elas unificam. Afinal, a unificação é o processo de encontrar coisas iguais.

Para o segundo caso, a definição é mais complexa. Logo, vamos fazer alguns exemplos para entender como ela procede. Para tanto, vamos imaginar que temos a seguinte base de conhecimento:

```
Homem(joão)
Idade(joão 25)
Mulher(maria)
Relacionamento(pedro maria)
Mortal (X) :- Homem(X;
HomemJovem(X) :- Homem(X), Idade (X 25)
HomemComprometido(X):-          Homem(X),
Relacionamento (X Y)
```

As regras que serão disparadas serão aquelas cujos antecedentes unificarem com fatos existentes na memória. Então vamos ver o que acontece com esta base.

A primeira regra[12] contém o antecedente Homem(X). Então vamos tentar unificá-lo com o fato `Mulher(maria)`. Para tanto, deveríamos unificar o *functor* `Homem` do antecedente com o *functor* `Mulher` do fato. Note que ambos são constantes e, pela definição anterior, constantes são unificadas se são iguais. Logo, este fato não pode disparar esta regra.

Em seguida, vamos tentar unificar o antecedente `Homem(X)` com o fato `Homem(joão)`. Os *functors* são unificados, pois são iguais. O segundo passo é encontrar uma instanciação para a variável `X` de forma que os fatos sejam iguais. No caso, se atribuirmos o valor `X=joão`, então teremos que o antecedente e o fato se tornam iguais. Logo, este fato unifica com este antecedente e a regra pode ser disparada.

A segunda regra contém dois antecedentes: `Homem(X)` e `Idade(X 25)`. Já vimos anteriormente que Homem(X) pode ser unificado com o fato `Homem(joão)` através da atribuição `X=joão`. Assim, de acordo com a definição, temos que manter esta atribuição até o fim da unificação (é isto que a expressão instanciação de forma única quer dizer) e encontrar algum outro fato. Assim, para efeitos de unificação, temos que encontrar agora o fato `Idade(joão 25)` dentro da nossa base. Como este existe, então podemos concluir que a unificação é bem-sucedida e a regra é disparada.

Podemos entender agora por que a terceira regra não será disparada. Ela depende de `Homem(X)` e `Relacionamento(X Y)`. Começando por `Homem(X)`, encontramos a atribuição `X=joão`, como vimos acima. Logo, precisaríamos ter na base um fato que unificasse com `Relacionamento(joão Y)`, pois como colocamos acima, cada uma das variáveis (X e Y) tem que ter um valor único a cada instanciação. Infelizmente, nosso amigo João não está em um relacionamento. Logo, não há nenhum fato que unifique com `Relacionamento(joão Y)`.

Se começássemos pela segunda "metade" do antecedente, então poderíamos unificar `Relacionamento(X Y)` com `Relacionamento(pedro maria)` através das atribuições `X=pedro` e `Y=maria`. Precisaríamos então encontrar na base um fato `Homem(pedro)`, pois `X` precisa ser instanciada com um único valor a cada instanciação. Não adianta ter homem(joão), pois já chegamos à conclusão que `X` tem que ser igual a `pedro`. Logo, como esquecemos de cadastrar o sexo de Pedro, fica impossível ativar esta regra.

E o que aconteceria com nossa base se acrescentássemos a ela o fato `homem(Pedro)`? O que mudaria nos resultados?

Neste caso, teríamos duas ativações para a primeira regra, chegando a duas conclusões diferentes (`Mortal(pedro)` e `Mortal(joão)`). Isto nos mostra que em duas unificações diferentes as variáveis recebem atribuições diferentes. Ou seja, duas unificações distintas são mutuamente independentes (pense em `X` como sendo uma variável local).

Segundo, agora temos uma maneira de unificar a regra com dois antecedentes. Para tanto, fazemos as atribuições `X=pedro` e `Y=maria`. Ou seja, fazendo uma instanciação regular, ou seja, atribuindo um único valor para cada variável, nós conseguimos encontrar fatos na memória que correspondem aos antecedentes com valores atribuídos.

Deve ser claro para você agora que o processo de unificação é bastante tedioso e consome bastante tempo. Isto é, ele é ideal para máquinas. Sua implementação é relativamente complexa, pois a entrada de novos fatos pode ativar regras em conjunto com fatos que já estavam na memória. Isto aconteceu quando inserimos `Homem(pedro)` na nossa base - o fato pré-armazenado `Relacionamento(pedro maria)` se juntou a este novo fato para ativar uma das regras. Logo, a cada nova inferência ou inserção de fato devemos fazer uma verificação completa de todas as regras com todos os fatos da memória. Obviamente, isto não é um problema para sistemas computacionais, cuja excelência está exatamente na execução de reclamam de tarefas tediosas e repetitivas.

### VI.2 *BACKTRACKING*

Quando o Prolog tenta resolver uma pergunta, ele tenta unificar (satisfazer) uma cláusula de cada vez. A unificação pode se dar com um fato ou com o consequente de uma regra.

Se a unificação se dá com um fato, então os valores das variáveis são atribuídos de acordo com os valores contidos nos fatos. Caso a unificação se dê com o consequente de uma regra (também chamado de sua cabeça, ou *head*), nós vamos reiniciar o processo (recursivamente!!!!) com os antecedentes das regras (também chamados de corpo, ou *body*, da regra).

Ao unificar com um fato, o Prolog, ao resolver a cláusula *n*, estabelece um valor para cada variável na cláusula e vai tentar usa estes valores na unificação da cláusula *n+1*. Se não der certo, ele tem que desistir e voltar para a cláusula anterior e recomeçar o processo de unificação com outras atribuições para as variáveis. A este retrocesso chamamos de *backtracking*.

[12] A ordem de execução das regras não necessariamente é igual à que mostramos aqui. A nossa sequência é determinada de forma que seja conveniente para a didática, mas não necessariamente reflete a estratégia do motor de inferência do seu Prolog.

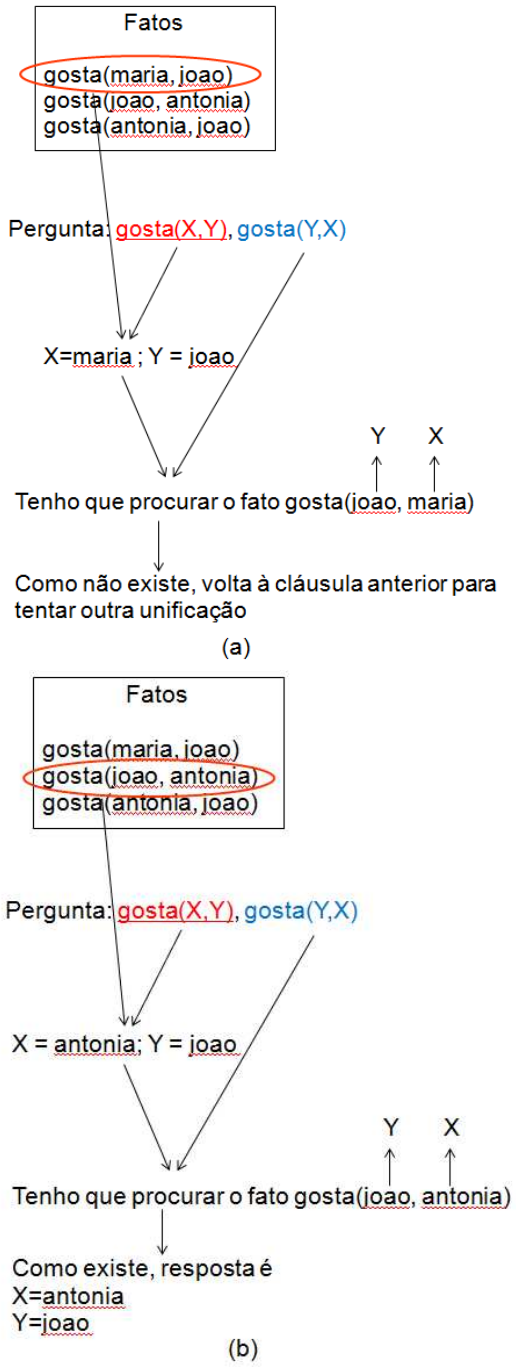


Figura 2. Como o Prolog tenta responder a pergunta que fazemos sobre a existência de um relacionamento viável em nossa base de fatos. Em (a), ele faz a unificação da primeira cláusula com o primeiro fato, obtendo valores para as variáveis X e Y e gerando a necessidade de encontrar uma segunda cláusula que não existe na memória. Logo, como ele falha na hora de unificar com a segunda *(n+1)* cláusula, ele faz o *backtracking*, retornando para tentar outra unificação com a primeira (*n)* cláusula. Em (b), vemos o processo bem-sucedido.

Vamos entender mais facilmente como este processo funciona através de exemplos. Imagine, então, a seguinte base de fatos:

```
gosta(maria, joao)
gosta(joao, antonia)
gosta(antonia, joao)
```

Agora nós perguntamos para o Prolog o seguinte:

```
gosta(X,Y), gosta(Y,X).
```

Isto é, nós estamos procurando um casal em que um goste do outro. Assim, temos chances de ter um relacionamento bem-sucedido! Como o Prolog vai responder a nossa pergunta?

O processo é simples, apesar daquela descrição ser em linguagem empolada: temos duas cláusulas `gosta`. Prolog vai unificar com a primeira (a chamada cláusula *n*), atribuindo valores a `X` e `Y` (o estabelecer valor para cada variável, conforme descrito no parágrafo). Depois, ao tentar fazer a unificação da segunda cláusula (a chamada cláusula *n+1*), ele vai usar os valores das variáveis que já foram definidos. Se ele encontrar uma cláusula que unifique, ele dá a resposta. Caso contrário, ele desiste e recomeça o processo da cláusula 1. A figura 2 explica este procedimento.

Agora vamos entender o segundo caso, a unificação com a cabeça de uma regra. Para tanto, vamos adicionar uma regra à base de fatos que usamos até agora:

```
casal(X,Y) :- gosta(X, Y), gosta(Y,X).
```

Basicamente, esta regra nos informa que se ambos se gostam, então eles podem formar um casal. Agora imagine que o Prolog recebe uma pergunta igual a:

```
casal(Z,joao).
```

Nós queremos saber com quem João pode formar um casal. A pergunta que nos resta neste momento é: como vai se dar a busca pela resposta?

Se olharmos em nossa base de fatos, não há nenhum fato que tenha o *functor* igual a `casal`. Logo, não há como unificar nada com esta consulta. Entretanto, nossa nova regra tem um consequente que pode unificar com a pergunta, desde que façamos a atribuição `Y=joao`.

Assim, vamos reiniciar o processo de busca com os antecedentes desta regra, já sabendo de antemão que `Y=joao`. Ou seja, temos que agora procurar unificar fatos ou regras de nossa base com os fatos `gosta(X, joao)` e `gosta(joao, X)`.

O processo é similar àquele visto na figura 3. Primeiro vamos unificar `gosta(X,joao)` com `gosta(maria, joao)` fazendo a atribuição `X=maria`. Entretanto, isto transforma o segundo fato buscado em `gosta(joao, maria)`, pois lembre-se que na unificação todas as variáveis têm que sofrer atribuição homogênea (a um valor único) e este fato não está presente na base. Assim, o processo falha e temos que realizar um *backtracking* para a última atribuição feita (`Y=maria`).

Logo, fazemos uma nova atribuiçao `Y=antonia`, de acordo com nossa base de fatos e podemos buscar `gosta(joao, antonia)`, sendo então bem sucedidos e

obtendo a resposta `Z=antonia`.

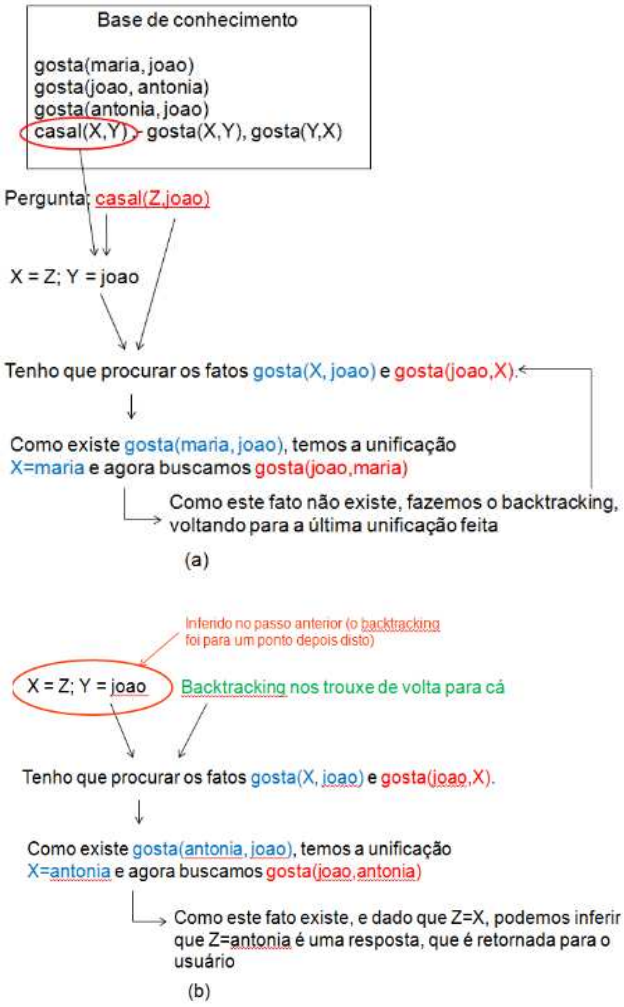


Figura 3. Processo de backtracking quando temos unificação com a cabeça de uma regra. No lado esquerdo, unificamos a pergunta casal(Z,joao) com a cabeça da nova regra e depois falhamos ao tentar a unificação com o fato gosta(maria,joao). Note que não voltamos ao primeiro passo da unificação (a unificação com a cabeça da regra), mas sim para o anterior (a unificação com o primeiro dos fatos do corpo da regra). Esta é a essência do processo de unificação: quando um passo falha irremediavelmente, voltamos para o anterior, do qual só desistimos quando não há mais tentativas de unificação possíveis. O processo termina quando a primeira unificação não pode ser tentada com nenhuma alternativa.

## VII. CONCLUSÕES

Este tutorial apresentou de forma superficial as principais características da linguagem Prolog, de forma a apresentar o paradigma lógico para os leitores que o desconhecem.

Não era o objetivo deste trabalho ser um tutorial exaustivo ou garantir que os leitores seriam capazes de programar em Prolog ao terminar a sua leitora. Ao contrário, queria apenas apresentar as principais ideias associadas ao paradigma em geral e à linguagem Prolog, motivando os leitores a buscarem posteriormente maior aprofundamento.

Assim, de forma não muito ortodoxa, recomenda-se que os leitores interessados realizem o trabalho futuro de baixar uma implementação de Prolog e estudá-la com os materiais apropriados para poder aprofundar o conhecimento que foi exposto aqui de forma introdutória.

## AGRADECIMENTOS

Este trabalho deveria ser um capítulo de um livro escrito a três mãos, mas que infelizmente acabou não sendo publicado. Entretanto, a amizade e a colaboração de Flávio M. Varejão e Vitor E. Silva Souza, ambos da UFES, foi incrivelmente valiosa e contribuiu de forma inestimável para a conclusão deste tutorial.

## REFERÊNCIAS

[1] Del Palù, A.; Torroni, P.; “25 Years of Applicatons of Logic Programming in Italy”, Proceedings of 25 Years GULP, Itália, 2010, disponível no site, https://www.cs.nmsu.edu/ALP/wp-content/uploads/2010/06/ALPNewsJune2010.pdf, ultimo acesso em 22/11/2022.

[2] Silva, V. N. A. L.; LINDEN, R; Ribeiro, G. F; Pereira, Maria de Fatima L; Santanna, L. P. (2009), "Aplicação de Processamento Inteligente de Alarmes em Tempo Real no COT Da Eletrosul". In: Décimo Tercer Encuentro Regional Iberoamericano De Cigré, Foz do Iguaçu. ANales Del Décimo Tercer Encuentro Regional Iberoamericano De Cigré, 2009. v. 1.

[3] Alencar Filho, E. (2002) - Iniciação à logica matemática, Ed. Nobel, Rio de Janeiro, 2002

[4] Coelho, R. M. (2014), "Introdução à lógica matemática", Edição do autor, Espírito Santo, 2014

[5] Kovalski, R., "Logic Programming", disponível no endereço da Internet https://www.doc.ic.ac.uk/~rak/papers/History.pdf. Último acesso em 24/11/2022.

[6] Sebesta, R. W. *Conceitos de linguagens de programação, 1ª* edição brasileira, Bookman, 2011.

[7] Rouchy, P., " Aspects of PROLOG History: Logic Programming and Professional Dynamics ", artigo disponível na Internet no endereço https://archive.cs.st-andrews.ac.uk/STSE-Handbook/Other/Team%20Ethno/Issue2/Rouchy.pdf. Último acesso em 24/11/2022

[8] Bratko, I., "Prolog Programming for Artificial Intelligence", 3ª Edição, Addison-Wesley, EUA, 2000

[9] Clocksin, W. F., Mellish, C. S., "Programming in Prolog using the ISO Standard", 4ª Edição, Springer-Verlag, EUA, 2013

[10] Varejão, F., “Linguagens de programação – Conceitos e Técnicas”, Ed Campus, Rio de Janeiro, 2004