APÊNDICE A
PROVA AUTOMÁTICA DE TEOREMAS

A.1 Uma Visão Pragmática de Lógica

A.1.1 Cálculo Proposicional

Sabe-se que um objeto é uma ave se ele tem penas e voa ou se tem penas e põe ovos. Para expressar essa ideia em lógica, deve-se ter um meio de capturar a ideia de que algo tem penas e que algo é uma ave. Isto é feito utilizando-se predicados, que podem ser considerados como funções que mapeiam argumentos (objetos) em verdadeiro (***V***) ou falso (***F***). Por exemplo, usando-se os objetos *pombo* e os predicados *penas* — representando *tem penas* — e *ave* — representando *é uma ave* — pode-se assegurar informalmente que as seguintes expressões são verdadeiras:

***penas*(*pombo*)**

***ave*(*pombo*)**

Agora considere o que significa dizer que a seguinte expressão é também verdadeira:

***ave*(*piupiu*)**

Nesse caso, *piupiu* é um símbolo que denota algo que tem penas pois ele satisfaz o predicado *ave*. Assim, tem-se uma restrição sobre o que *piupiu* pode representar. Pode-se expressar outras restrições com outros predicados, tais como voa e *poe_ovos* e limitar o que *piupiu* pode representar a objetos que satisfazem ambos os predicados, dizendo que ambas as expressões seguintes são verdadeiras:

***voa*(*piupiu*)**

***poe_ovos*(*piupiu*)**

Entretanto há uma forma mais tradicional de expressar essa ideia. Simplesmente, combina-se a primeira expressão com a segunda e diz-se que a combinação toda é verdadeira:

***voa*(*piupiu*) *e poe_ovos*(*piupiu*)**

Pode-se também querer interpretar *piupiu* como sendo o nome de algo que satisfaz um predicado ou o outro. Isso é feito utilizando-se a seguinte combinação:

***voa*(*piupiu*) *ou poe_ovos*(*piupiu*)**

Uma notação usual da lógica utiliza os símbolos ∧ e ∨ para representar *e* e *ou*, respectivamente. Assim, pode-se escrever as duas últimas expressões utilizando-se essa notação como:

***voa*(*piupiu*) ∧ *poe_ovos*(*piupiu*)**

***voa*(*piupiu*) ∨ *poe_ovos*(*piupiu*)**

Quando duas expressões são combinadas por ∧, elas formam uma **conjunção** e cada uma delas é denominada **conjunta**. Quando duas expressões são combinadas por ∨, elas formam uma **disjunção** e cada uma delas é denominada **disjunta**. Os símbolos ∧ e ∨ são denominados **conectivos lógicos**.

Além de ∧ e ∨, existem mais três importantes conectivos lógicos que são a negação ¬, a implicação → e a equivalência ↔. Considere o seguinte fato:

¬*penas*(*maria*)

Para que esse fato seja verdadeiro, *maria* deve denotar algum objeto tal que *penas*(*maria*) não seja verdadeiro. Isto é, *maria* deve ser algo para o qual o predicado *penas* não é satisfeito.

Por outro lado, dizer que a expressão:

***penas*(*maria*) → *ave*(*maria*)**

é verdadeira restringe aquilo que *maria* pode denotar. Uma possibilidade permitida é que *maria* seja um objeto tal que tanto *penas*(*maria*) quanto *ave*(*maria*) sejam verdadeiros. Naturalmente, a definição de → também permite que *penas*(*maria*) e *ave*(*maria*) sejam falsos. Curiosamente, outra possibilidade permitida pela definição de → é que *penas*(*maria*) seja falso mas *ave*(*maria*) seja verdadeiro. Se, entretanto, *penas*(*maria*) for verdadeiro e *ave*(*maria*) for falso, então a expressão *penas*(*maria*) → *ave*(*maria*) será falsa. Por outro lado, uma expressão como *penas*(*maria*) ↔ *ave*(*maria*) é verdadeira apenas quando os predicados *penas*(*maria*) e *ave*(*maria*) possuem os mesmos valores; i.e., quando ambos são verdadeiros ou ambos são falsos.

Se E_1 e E_2 denotam expressões atômicas, i.e., expressões sem nenhum conectivo lógico, podem-se enumerar os valores assumidos pelas expressões compostas por conectivos lógicos em função dos possíveis valores atribuídos às expressões

E_1 e E_2. Essa enumeração feita em forma de tabela para cada conectivo é denominada **tabela-verdade** do conectivo. As tabelas-verdade dos conectivos apresentados acima foram vistos na **Seção 3.2**.

De forma análoga ao que foi feito acima, podem ser utilizadas tabelas-verdade para obter os valores-verdade assumidos por expressões mais complexas em função dos possíveis valores atribuídos às expressões simples que as compõem, como foi visto na **Seção 3.3**.

Existem regras de precedência para utilização de conectivos em expressões mais complexas. A ordem de precedência dos conectivos lógicos é apresentada na **Tabela A–1**. Seguindo essa ordem de precedência, a expressão $\neg E_1 \vee E_2$ é interpretada como $(\neg E_1) \vee E_2$ e não deve ser confundida com $\neg(E_1 \vee E_2)$, que é uma expressão que não pode ser escrita sem parênteses. De qualquer modo, quando houver algum perigo de confusão, o uso de parênteses é sempre aconselhável.

Conectivo	Símbolo	Precedência
Negação	$\neg$	**Maior**
Conjunção	$\wedge$	
Disjunção	$\vee$	
Condicional	$\rightarrow$	
Bicondicional	$\leftrightarrow$	**Menor**

Tabela A–1: Precedências de Conectivos Lógicos

Por meio do uso de tabelas-verdade, podem ser determinadas propriedades importantes de expressões lógicas. Por exemplo, construindo-se a tabela-verdade da expressão $\neg E_1 \vee E_2$, vê-se que os valores assumidos por ela são os mesmos que os da expressão $E_1 \rightarrow E_2$, para cada combinação de valores de E_1 e E_2. Quando isso ocorre, como nesse caso particular, diz-se que as duas expressões são **logicamente equivalentes** e esse fato é denotado aqui por meio do símbolo $\Leftrightarrow$, que não ceve ser confundido com símbolo $\leftrightarrow$. Uma consequência importante do fato de duas expressões serem logicamente equivalentes é que uma das expressões pode ser substituída pela outra em qualquer instante. Por exemplo, $\neg E_1 \vee E_2$ pode ser sempre substituída por $E_1 \rightarrow E_2$ e vice-versa. Outros pares de expressões que são logicamente equivalentes, usualmente conhecidos como leis, foram apresentados na **Seção 3.6**.

A.1.2 Cálculo Quantificado

Para informar que uma expressão é **universalmente verdadeira** (i.e., verdadeira para todos os valores possíveis), é necessário utilizar um símbolo significando para todo — escrito como ∀ — bem como uma variável que representa qualquer objeto possível. O símbolo ∀ é chamado **quantificador universal**. A expressão do exemplo seguinte pode ser interpretada como: qualquer coisa que tem penas é uma ave.

$$\forall X\,[\,penas(X) \rightarrow ave(X)]$$

Como outras expressões, essa última expressão pode ser verdadeira ou falsa. Se for verdadeira, uma expressão com ∀ significa que se obtém uma expressão verdadeira quando se substitui X por qualquer objeto na expressão entre colchetes. Por exemplo, se $\forall X\,[\,penas(X) \rightarrow ave(X)]$ for verdadeira, então:

$$penas(maria) \rightarrow ave(maria)$$

e

$$penas(piupiu) \rightarrow ave(piupiu)$$

também são verdadeiras.

Quando uma expressão está entre os colchetes associados com um quantificador, diz-se que ela expressão está dentro do **escopo** deste quantificador. Assim, a expressão $penas(X) \rightarrow ave(X)$ do exemplo acima está dentro do escopo do quantificador $\forall X\,[...]$ e é **universalmente quantificada**.

Algumas expressões, embora não sejam sempre verdadeiras, são verdadeiras pelo menos para algum objeto. Elas são escritas utilizando-se um símbolo com o significado de existência, denotado aqui por ∃. Por exemplo, quando a expressão:

$$\exists X\,[ave(X)]$$

é verdadeira, deve-se entender que há pelo menos um possível objeto que, quando substitui X, torna a expressão entre colchetes verdadeira. Talvez, $ave(piupiu)$ seja verdadeira, por exemplo.

Expressões com ∃ são ditas serem **existencialmente quantificadas** e o símbolo ∃ é chamado **quantificador existencial.**

A lógica possui uma rica nomenclatura. Serão vistas em seguida algumas definições que fazem parte dessa nomenclatura.

- ❒ Objetos de um domínio são **termos**.

- Variáveis de um domínio de objetos também são termos.
- Funções são termos. Argumentos de funções e valores retornados por funções também são objetos e, consequentemente, termos.
- Apenas os termos podem ser argumentos de predicados.
- Uma **fórmula atômica** é um predicado com seus argumentos.
- **Literais** são fórmulas atômicas ou negações de fórmulas atômicas.
- Fórmulas bem formadas (fbfs) são definidas recursivamente:
  - Literais são fbfs
  - fbfs conectadas por ¬, ∧, ∨, → e ↔ são fbfs
  - fbfs no escopo de quantificadores também são fbfs.
  - Apenas são fbfs as expressões obtidas aplicando-se essa definição.

Existem alguns casos notáveis para fbfs:

- Uma fbf na qual todas as variáveis, se existir alguma, estão no escopo de quantificadores correspondentes é uma **sentença**.

  As fbfs seguintes, por exemplo, são sentenças:

  $$\forall X\,[penas(X) \rightarrow ave(X)]$$

  $$penas(urubu) \rightarrow ave(urubu)$$

- Variáveis, como X acima, que aparecem no escopo de quantificadores correspondentes são chamadas **variáveis ligadas**. Variáveis que não são ligadas são chamadas **variáveis livres**.

  A seguinte expressão não é uma sentença porque ela contém a variável Y livre:

  $$\forall X[\,penas(X) \vee \neg penas(Y)]$$

- Uma fbf consistindo de uma disjunção de literais é uma **cláusula**.

Permite-se apenas que as variáveis representem objetos (ou seja, variáveis não podem representar predicados). Por isso, esse ramo da lógica é conhecido como **cálculo de predicados de primeira ordem**. Ramos mais avançados da lógica permitem que variáveis representem predicados. O ramo mais simples da lógica é o **cálculo proposicional** (v. Capítulo 3), que não permite a existência de nenhuma variável.

O objetivo final da lógica é fazer afirmações sobre algum mundo imaginável. Consequentemente, deve-se associar símbolos de objetos, funções e predicados com coisas mais palpáveis. Como ilustrado na figura seguinte, as três

categorias de símbolos no mundo lógico correspondem às três categorias do mundo imaginável:

1. Objetos de algum domínio correspondem a símbolos de objetos na lógica. Na **Figura A–1**, os símbolos de objetos *A* e *B* na esquerda correspondem a dois objetos no mundo imaginável da direita.
2. Relações sobre algum domínio correspondem a predicados na lógica. Sempre que uma relação vale com respeito a alguns objetos, o predicado correspondente é verdadeiro quando aplicado aos símbolos de objetos correspondentes. No exemplo da **Figura A–1**, o predicado *sobre* do mundo lógico aplicado aos símbolos *A* e *B* é verdadeiro porque a relação *relação-sobre* do mundo imaginável é válida para os dois objetos do mundo imaginável.
3. Funções sobre algum domínio correspondem a funções na lógica. As funções no domínio retornam objetos do domínio quando lhes são atribuídos objetos do domínio. Funções correspondentes em lógica retornam símbolos lógicos correspondentes quando símbolos lógicos correspondentes lhes são atribuídos. Na **Figura A–1**, a função do mundo lógico *f* mapeia o símbolo de objeto *B* no símbolo de objeto *A*, porque a função do mundo imaginável *função-sobre* faz o mapeamento correspondente entre objetos do mundo imaginável.

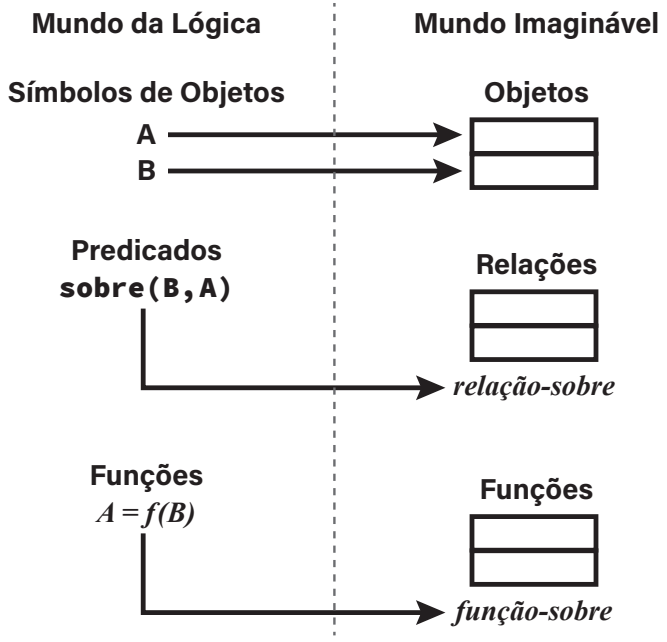

Figura A–1: Mapeamento entre Mundo da Lógica e Mundo Imaginável

Uma **interpretação** é um mapeamento entre objetos e símbolos de objetos, entre relações e predicados e entre funções de objetos e funções de símbolos de objetos.

Pode-se, agora, explorar a noção de prova. Suponha que as seguintes expressões sejam verdadeiras:

***penas*(*piupiu*)**

$\forall X\,[\,\textit{penas}(X) \rightarrow \textit{ave}(X)]$

Como já se conhece a noção de interpretação, entende-se o que significa dizer que tais expressões são verdadeiras: significa que as interpretações são restritas a símbolos de objetos, predicados e funções para os quais as relações do mundo imaginável valem. Tal interpretação é considerada um **modelo** para as expressões.

Quando se afirma que as expressões *penas*(*piupiu*) e $\forall X\,[\textit{penas}(X) \rightarrow \textit{ave}(X)]$ são verdadeiras, elas são chamadas **axiomas**. Suponha que se solicite para mostrar que todas as interpretações que tornam esses axiomas verdadeiros também tornam verdadeira a expressão:

***ave*(*piupiu*)**

Se a tarefa for bem sucedida, diz-se que foi provado que *ave*(*piupiu*) é um **teorema** com relação aos axiomas. Em outras palavras, prova-se que uma expressão é um teorema quando mostra-se que qualquer modelo para os axiomas também é um modelo para o teorema. Nesse caso, diz-se que o teorema resulta logicamente dos axiomas.

Em termos mais simples:

> ***Prova-se que uma expressão é um teorema quando mostra-se que o teorema é verdadeiro, quando os axiomas são verdadeiros.***

O modo usado para provar um teorema é por meio de um **procedimento de prova**. Procedimentos de prova utilizam transformações chamadas **regras de inferência** que produzem expressões novas a partir de expressões antigas, de forma que os modelos que tornam as expressões antigas verdadeiras também tornam as novas expressões verdadeiras. Assim, o procedimento de prova mais direto é a aplicação de uma regra de inferência aos axiomas e aos resultados dessas aplicações (teoremas), até que surja o teorema desejado.

Note que provar um teorema não é o mesmo que mostrar que uma dada expressão é **válida**, que significa que a expressão é verdadeira para todas as possíveis interpretações. Similarmente, provar um teorema não é o mesmo que

mostrar que uma expressão satisfazível, que significa que ela é verdadeira para alguma possível interpretação dos símbolos.

A regra de inferência mais utilizada em procedimentos de prova é conhecida como **modus ponens**. Modus ponens diz que se as expressões E_1 e $E_1 \rightarrow E_2$ são axiomas, E_2 resulta logicamente dessas expressões.

Se E_2 for o teorema a ser mostrado, tudo estará resolvido. Caso contrário, pode-se acrescentar E_2 ao conjunto de axiomas, pois ele será sempre verdadeiro quando todos os axiomas forem verdadeiros. Continuando a utilizar modus ponens com um conjunto de axiomas cada vez maior, pode-se mostrar que o teorema desejado é verdadeiro, o que conclui a prova desejada.

No último exemplo apresentado, os axiomas são perfeitos para a aplicação de modus ponens. Primeiro, entretanto, deve-se especificar (i.e., **instanciar**) a segunda expressão: $\forall[\,penas(X) \rightarrow ave(X)]$. Como se está lidando com interpretações para as quais $penas(X) \rightarrow ave(X)$ é verdadeira para todo X, ela deve ser verdadeira, em particular, quando X é *piupiu*. Consequentemente, $penas(piupiu) \rightarrow ave(piupiu)$ deve ser verdadeira. A primeira expressão, $penas(piupiu)$ e a **instanciação** da segunda, $penas(piupiu) \rightarrow ave(piupiu)$, satisfazem a regra de inferência modus ponens, com $penas(piupiu)$ representando E_1 e $(piupiu) \rightarrow ave(piupiu)$ representado $E_1 \rightarrow E_2$. Conclui-se, portanto, que a expressão $ave(piupiu)$ é verdadeira e o teorema está provado.

O uso de procedimentos de prova para derivar teoremas é uma atividade sintática (i.e., mera manipulação de símbolos), que independe dos significados das expressões envolvidas. Nesse caso, lida-se com mundos imagináveis e não com o mundo real, em que os axiomas podem colidir com o bom senso. No mundo real, *piupiu* pode não ter penas e existem coisas com penas que não são aves (p. ex., espanadores). Em resumo, o objetivo aqui é mostrar que teoremas são consequências de axiomas e não que axiomas correspondem ao bom senso.

Uma das regras de inferência mais importantes é a **resolução**, que diz: se há um axioma da forma $E_1 \vee E_2$ e outro axioma da forma $\neg E_2 \vee E_3$, $E_1 \vee E_3$ resulta logicamente desses axiomas. A expressão $E_1 \vee E_3$ é chamada o **resolvente** de $E_1 \vee E_2$ e $\neg E_2 \vee E_3$.

Será mostrado agora que, de fato, $E_1 \vee E_3$ é verdadeiro quando $E_1 \vee E_2$ e $\neg E_2 \vee E_3$ são verdadeiros ($\boldsymbol{V}$). Suponha que $E_1 \vee E_3$ seja $\boldsymbol{F}$. Então, tanto E_1 quanto E_3 são $\boldsymbol{F}$. Como $E_1 \vee E_2$ é $\boldsymbol{V}$, tem-se que E_2 deve ser $\boldsymbol{V}$. Mas, se E_2 é $\boldsymbol{V}$ e E_3 é $\boldsymbol{F}$, tem-se que $\neg E_2 \vee E_3$ é $\boldsymbol{F}$, o que contraria o fato de $\neg E_2 \vee E_3$ ser $\boldsymbol{V}$. Portanto, $E_1 \vee E_3$ não pode ser $\boldsymbol{F}$, conforme se queria mostrar.

Isso que se acabou de demonstrar é um metateorema de dedução por resolução.

É fácil generalizar a resolução de forma que possa haver qualquer número de disjuntas (inclusive apenas uma) em qualquer das duas expressões da resolução. A única exigência é que uma das expressões da resolução deve ser uma disjunta que seja a negação de uma disjunta na outra expressão resolvente.

Pode-se utilizar **resolução generalizada** para chegar à mesma conclusão sobre *piupiu* que se chegou antes com modus pones. Novamente, o primeiro passo é especializar a expressão quantificada para *piupiu*. O próximo passo é eliminar a implicação → usando sua equivalência lógica (v. **Seção 3.6.10**), o que resulta em:

***penas*(*piupiu*)**

¬*penas*(*piupiu*) ∨ *ave*(*piupiu*)

Com as expressões assim escritas, a resolução pode ser aplicada, resultando em *ave*(*piupiu*).

Na realidade, esse exemplo sugere um fato geral: a regra de inferência modus ponens pode ser vista como um caso particular de resolução, porque tudo que pode ser concluído com modus ponens pode também ser concluído com a resolução.

É fácil verificar essa última afirmação. Suponha que E_1 e $E_1 \rightarrow E_2$ são ***V***. De acordo com modus ponens, E_2 deve ser ***V***. Mas, $E_1 \rightarrow E_2$ é logicamente equivalente a $\neg E_1 \vee E_2$ e, de acordo com a regra de resolução, essa expressão pode ser resolvida com E_1 resultando em E_2 ser ***V***, que é o mesmo resultado obtido com modus ponens.

Similarmente, resolução também generaliza outra regra de inferência chamada **modus tolens**, que afirma que:

> ***Se houver um axioma da forma $E_1 \rightarrow E_2$ e outro da forma $\neg E_2$, então $\neg E_1$ resulta logicamente desses axiomas.***

Exercício: Mostre que a modus tolens é um caso particular da resolução.

A.2 Prova por Resolução

Para provar-se um teorema, a abordagem óbvia consiste em aplicar regras de inferência sobre os axiomas até que o teorema desejado seja eventualmente derivado. Outra abordagem é mostrar que a negação do teorema não pode ser verdadeira. Essa abordagem funciona do seguinte modo:

(1) Supõe-se que a negação do teorema é verdadeira.

(2) Mostra-se que os axiomas e a negação do teorema não podem ser verdadeiros ao mesmo tempo.

(3) Conclui-se que a suposta negação do teorema não pode ser verdadeira, uma vez que ela conduz a uma contradição.

(4) Conclui-se que o teorema deve ser verdadeiro, visto que a suposta negação do teorema não pode ser verdadeira.

Provar um teorema mostrando que sua negação não pode ser verdadeira é chamada **prova por refutação** (ou **prova por redução ao absurdo**).

Considere novamente o último exemplo apresentado na **Seção A.1.2**. Nesse exemplo, tem-se que:

$\neg$***penas*(*piupiu*)** $\vee$ ***ave*(*piupiu*)**

***penas*(*piupiu*)**

Adicionando-se a negação daquilo a ser provado, obtém-se:

$\neg$***penas*(*piupiu*)** $\vee$ ***ave*(*piupiu*)**

***penas*(*piupiu*)**

$\neg$***ave*(*piupiu*)**

A resolução dos dois primeiros axiomas permite o acréscimo de uma nova expressão verdadeira à lista:

$\neg$***penas*(*piupiu*)** $\vee$ ***ave*(*piupiu*)**

***penas*(*piupiu*)**

$\neg$***ave*(*piupiu*)**

***ave*(*piupiu*)**

Todas as expressões dessa lista são supostamente verdadeiras. Logo não se pode ter *ave*(*piupiu*) e $\neg$*ave*(*piupiu*) simultaneamente verdadeiras. Assim tem-se uma contradição. Consequentemente, a suposição que levou a essa contradição deve ser falsa. Isto é, $\neg$*ave*(*piupiu*) deve ser falsa e, assim, o teorema *ave*(*piupiu*) deve ser verdadeiro, como era esperado.

O meio tradicional para reconhecer que um teorema está provado consiste em esperar até que a resolução ocorra sobre um literal e sua negação contraditória. O resultado é uma expressão vazia, que ser denotada aqui por **nil**. Quando a resolução produz *nil*, tem-se garantido que ela já produziu uma expressão contraditória. Assim a produção de *nil* sinaliza de que a resolução provou o teorema.

Serão vistas algumas manipulações cujo objetivo é transformar expressões lógicas arbitrárias numa forma que permita resolução. Especificamente, são desejados meios para transformar os axiomas dados em axiomas novos equivalentes que sejam todos disjunções de literais. Esses novos axiomas estão na **forma clausular**.

Para ilustrar essas manipulações, será utilizado um axioma envolvendo blocos. Embora esse axioma seja um tanto artificial, ele serve para exercitar os passos de transformação. O axioma afirma três coisas:

1. O bloco está sobre algo que não seja uma pirâmide
2. Não existe algo sobre o qual o bloco esteja que também esteja sobre o bloco
3. Não existe algo que não seja um bloco e que seja a mesma coisa que um bloco. Esse axioma pode ser expresso simbolicamente como:

$$\forall X[\mathit{bloco}(X) \rightarrow (\exists Y[\mathit{sobre}(X, Y) \wedge \neg \mathit{piramide}(Y)] \wedge \neg \exists Y[\mathit{sobre}(X, Y) \wedge \mathit{sobre}(Y, X)] \wedge \forall Y[\neg \mathit{bloco}(Y) \rightarrow \neg \mathit{igual}(X, Y)])]$$

Do modo como apresentado, esse axioma não pode ser utilizado para produzir resolventes porque ele não está na forma clausular. Entretanto, esses axiomas podem ser colocados na forma clausular seguindo-se os passos apresentados a seguir.

[1] Elimine implicações e equivalências

Isso é feito por meio da substituição de expressões do tipo $E_1 \rightarrow E_2$ por expressões logicamente equivalentes do tipo $\neg E_1 \vee E_2$ e de expressões do tipo $E_1 \leftrightarrow E_2$ por expressões do tipo $(E_1 \wedge E_2) \vee (\neg E_1 \wedge \neg E_2)$.

A aplicação desse passo na expressão anterior do exemplo resulta em:

$$\forall X[\neg \mathit{bloco}(X) \vee (\exists Y[\mathit{sobre}(X, Y) \wedge \neg \mathit{piramide}(Y)] \wedge \neg \exists Y[\mathit{sobre}(X, Y) \wedge \mathit{sobre}(Y, X)] \wedge \forall Y[\neg \neg \mathit{bloco}(Y) \vee \neg \mathit{igual}(X, Y)])]$$

[2] Mova as negações para as fórmulas atômicas

Para fazer-se isso são necessárias as seguintes identidades lógicas:

$$\neg(E_1 \wedge E_2) \Leftrightarrow \neg E_1 \vee \neg E_2$$
$$\neg(E_1 \vee E_2) \Leftrightarrow \neg E_1 \wedge \neg E_2$$
$$\neg\neg E_1 \Leftrightarrow E_2$$
$$\neg\forall X[E_1(X)] \Leftrightarrow \exists X[\neg E_1(X)]$$
$$\neg\exists X[E_1(X)] \Leftrightarrow \forall X[\neg E_1(X)]$$

Utilizando-se essas identidades na última expressão do exemplo dado, obtém-se:

$$\forall X[\neg bloco(X) \vee (\exists Y[sobre(X, Y) \wedge \neg piramide(Y)] \wedge \forall Y[\neg sobre(X, Y) \vee \neg sobre(Y, X)] \wedge \forall Y[bloco(Y) \vee \neg igual(X, Y)])]$$

[3] Elimine os quantificadores existenciais

A remoção dos quantificadores existenciais é efetuada introduzindo-se novos símbolos de constantes, chamados **constantes de Skolem**, no lugar das variáveis introduzidas por quantificadores existenciais. Em vez de dizer-se que existe um objeto com uma certa propriedade, pode-se criar um nome para tal objeto e simplesmente dizer que ele tem aquela propriedade. A introdução de constantes de Skolem numa expressão é chamada **skolemização**. A skolemização introduz uma interpretação para a versão skolemizada da fórmula. Assim, por exemplo,

$$\exists X[mulher(X) \wedge mae(X, eva)]$$

é transformada, por skolemização, em:

$$mulher(c_1) \wedge mae(c_1, eva)$$

em que c_1 é uma constante nova que não é usada em nenhum outro lugar. A constante c_1 representa alguma mulher cuja mãe é Eva. É importante que se utilize um símbolo diferente de qualquer outro utilizado previamente, porque o quantificador ∃ não diz que alguma pessoa específica filha de Eva, mas apenas que existe uma tal pessoa. Pode acontecer que c_1 corresponda à mesma pessoa que algum outro símbolo de constante represente, mas essa é uma informação extra que não é fornecida por esse quantificador.

Quando há quantificadores universais numa expressão, skolemização não é tão simples. Por exemplo, se a expressão

$$\forall X[humano(X) \rightarrow \exists Y[mae(X, Y)]],$$

interpretada como todo ser humano tem uma mãe, fosse skolemizada como:

$$\forall X\,[humano(X) \rightarrow mae(X, c_2)],$$

estar-se-ía dizendo que todo ser humano tem a mesma mãe, que seria o objeto representado por c_2. Assim, quando há variáveis quantificadas universalmente, skolemização deve introduzir símbolos de função para expressar de qual modo aquilo que se afirma que existe depende dessas variáveis. Assim a fórmula do último exemplo deveria ser skolemizada como:

$$\forall X\,[humano(X) \rightarrow mae(X, g_2(X)]$$

Nesse caso, o símbolo de função g_2 corresponde à função do mundo imaginável que retorna como valor a mãe da pessoa que lhe fornecida como argumento.

Retornando ao exemplo sobre blocos, observe atentamente a parte do axioma envolvendo ∃:

$$\exists Y\,[sobre(X, Y) \wedge \neg piramide(Y)]$$

Evidentemente, se alguém fornecer um objeto particular X é possível encontrar um objeto Y que torne essa expressão verdadeira. Em outras palavras, há uma função que recebe X como argumento e retorna um Y apropriado. Não se sabe necessariamente como é essa função, mas tal função deve existir. Por enquanto, essa função será denotada por $f(X)$.

Agora, utilizando-se essa nova função, não é mais necessário dizer que existe Y, pois se tem um meio de produzir o valor de Y apropriado em qualquer circunstância. Consequentemente, pode-se reescrever a expressão anterior como:

$$sobre(X, f(X)) \wedge \neg piramide(f(X))$$

Funções que eliminam a necessidade de quantificadores existenciais são chamadas **funções de Skolem** e esse procedimento de eliminação é chamado **skolemização**. E, portanto, a regra geral de skolemização é:

> ***Quantificadores universais determinam os argumentos da função de Skolem. Deve haver um argumento para cada variável quantificada universalmente cujo escopo contém a variável quantificada existencialmente que a função de Skolem substitui.***

O axioma do último exemplo após a eliminação de ∃ e a introdução da função de Skolem, aqui denominada *suporta*, torna-se:

$$\forall X\,[\neg bloco(X) \vee ((sobre(X, suporta(X)) \wedge \neg piramide(suporta(X))) \wedge \forall Y\,[\neg sobre(X, Y) \vee \neg sobre(Y, X)] \wedge \forall Y\,[bloco(Y) \vee \neg igual(X, Y)])]$$

[4] Renomeie as variáveis, quando necessário, de forma que duas variáveis quantificadas não tenham o mesmo nome

Quantificadores não se interessam pelos nomes das variáveis em si. Renomeie qualquer variável duplicada dentro de cada expressão, de forma que cada quantificador tenha uma variável de nome único. Faz-se isso porque deseja-se mover todos os quantificadores universais no próximo passo, sem causar confusão. A execução desse passo no exemplo dado resulta em:

$$\forall X\,[\neg bloco(X) \vee ((sobre(X, suporta(X)) \wedge \neg piramide(suporta(X))) \wedge \forall Y\,[\neg sobre(X, Y) \vee \neg sobre(Y, X)] \wedge \forall Z[bloco(Z) \vee \neg igual(X, Z)])]$$

[5] Mova os quantificadores universais para a esquerda

Pode-se fazer isso porque, agora, cada quantificador universal tem um nome de variável único. No exemplo dado, isso resulta em:

$$\forall X \forall Y \forall Z\,[\neg bloco(X) \vee ((sobre(X, suporta(X)) \wedge \neg piramide(suporta(X))) \wedge (\neg sobre(X, Y) \vee \neg sobre(Y, X)) \wedge (bloco(Z) \vee \neg igual(X, Z)))]$$

[6] Forme uma conjunção de disjunções

Para fazer-se isso, precisa-se apenas utilizar uma das leis distributivas (v **Seção 3.6.6**):

$$E_1 \vee (E_2 \wedge E_3) \Leftrightarrow (E_1 \vee E_2) \wedge (E_1 \vee E_3)$$

No exemplo corrente, isso será feito em dois passos.

Passo 1:

$$\forall X \forall Y \forall Z\,[(\neg bloco(X) \vee (sobre(X, suporta(X))$$
$$\wedge \neg piramide(suporta(X))))$$
$$\wedge (\neg bloco(X) \vee \neg sobre(X, Y) \vee \neg sobre(Y, X))$$
$$\wedge (\neg bloco(X) \vee bloco(Z) \vee \neg igual(X, Z))]$$

Passo 2:

$$\forall X \forall Y \forall Z\,[(\neg bloco(X) \vee sobre(X, suporta(X))) \wedge$$
$$(\neg bloco(X) \vee \neg piramide(suporta(X))) \wedge$$
$$(\neg bloco(X) \vee \neg sobre(X, Y) \vee \neg sobre(Y, X)) \wedge$$
$$(\neg bloco(X) \vee bloco(Z) \vee \neg igual(X, Z))]$$

[7] Separe as conjuntas

Na realidade, as conjunções não são eliminadas. Em vez disso, cada conjunta é escrita separadamente como um axioma. Isso faz sentido, uma vez que cada parte de uma conjunção deve ser verdadeira para que toda a conjunção seja verdadeira.

Para o exemplo dado, o resultado da aplicação desse passo é:

$$\forall X\,[\neg bloco(X) \vee sobre(X, suporta(X))]$$
$$\forall X\,[\neg bloco(X) \vee \neg piramide(suporta(X))]$$
$$\forall X \forall Y\,[\neg bloco(X) \vee \neg sobre(X, Y) \vee \neg sobre(Y, X)]$$
$$\forall X \forall Z\,[\neg bloco(X) \vee bloco(Z) \vee \neg igual(X, Z)]$$

[8] Renomeie todas as variáveis, quando necessário, de forma que duas variáveis não tenham o mesmo nome

Nesse passo, renomear as variáveis não causa nenhum problema, pois elas são quantificadas universalmente em cada parte de uma conjunção. Como cada conjunta deve ser verdadeira para quaisquer valores de variáveis, não importa se as variáveis têm nomes diferentes em cada parte.

O resultado obtido para o exemplo em curso é:

$$\forall X\,[\neg bloco(X) \vee sobre(X, suporta(X))]$$
$$\forall W\,[\neg bloco(W) \vee \neg piramide(suporta(W))]$$
$$\forall U \forall Y\,[\neg bloco(U) \vee \neg sobre(U, Y) \vee \neg sobre(Y, U)]$$
$$\forall V \forall Z\,[\neg bloco(V) \vee bloco(Z) \vee \neg igual(V, Z)]$$

[9] Remova os quantificadores universais

Na realidade, os quantificadores universais não são eliminados. Apenas convenciona-se que todas as variáveis nesse ponto são, implicitamente, quantificadas universalmente.

A aplicação desse passo ao exemplo em custo resulta finalmente em:

$$\neg bloco(X) \vee sobre(X, suporta(X))$$

$$\neg bloco(W) \vee \neg piramide(suporta(W))$$

$$\neg bloco(U) \vee \neg sobre(U, Y) \vee \neg sobre(Y, U)$$

$$\neg bloco(V) \vee bloco(Z) \vee \neg igual(V, Z)$$

Nesse ponto, a expressão resultante está na forma clausal, como requerido para a utilização da resolução.

O procedimento a ser seguido para provar um teorema por resolução é:

1. Negue o teorema a ser provado e acrescente o resultado lista de axiomas.
2. Coloque a lista de axiomas (inclusive o teorema negado) na forma clausal.
3. Até que a cláusula vazia nil seja produzida ou não exista nenhum par de cláusulas que possa ser resolvido, encontre cláusulas que possam ser resolvidas, resolva-as e acrescente o resultado lista de cláusulas.
4. Se a cláusula vazia for produzida, o teorema está provado. Caso contrário, se não houver mais nenhum par de cláusulas resolvíveis, o teorema falso.

Esse procedimento não diz nada sobre que pares de cláusulas são resolvidos num determinado instante. Modos de especificar como a busca deve proceder serão discutidos mais adiante.

A seguir, será visto um exemplo completo de prova por resolução. Nesse exemplo, os seguintes axiomas são válidos para as relações de blocos:

$$sobre(b, a)$$

$$sobre(a, mesa)$$

Esses axiomas já estão na forma clausular e serão utilizados para mostrar que o objeto *b* está acima da mesa:

$$acima(b, mesa)$$

Para mostrar isso, são necessárias as formas clausulais de duas expressões quantificadas universalmente. A primeira diz que estar sobre um objeto implica em

estar acima desse objeto. A segunda cláusula informa que um objeto está acima de outro se há um objeto entre eles. Ou seja,

$$\forall X \forall Y\, [sobre(X, Y) \rightarrow acima(X, Y)]$$

$$\forall X \forall Y \forall Z\, [acima(X, Y) \wedge acima(Y, Z) \rightarrow acima(X, Z)]$$

Depois de utilizar-se o procedimento para transformar cada um desses axiomas para a forma clausular, obtém-se (verifique isso):

$$\neg sobre(U, V) \vee acima(U, V)$$

$$\neg acima(X, Y) \vee \neg acima(Y, Z) \vee acima(X, Z)$$

O que deve ser provado é *acima(b, mesa)* e não é necessária nenhuma conversão para a forma clausular após sua negação:

$$\neg acima(b, mesa)$$

A seguir, as cláusulas serão numeradas para facilitar referências:

$\neg sobre(U, V) \vee acima(U, V)$	**(1)**
$\neg acima(X, Y) \vee \neg acima(Y, Z) \vee acima(X, Z)$	**(2)**
$sobre(b, a)$	**(3)**
$sobre(a, mesa)$	**(4)**
$\neg acima(b, mesa)$	**(5)**

A abordagem inicial é trocar a relação *acima* por relações *sobre* que permitam resolver os axiomas que especificam o arranjo real de blocos. Primeiro, resolvem-se (2) e (5), instanciando-se X com b e Z com *mesa*, de forma que a parte final de (2) seja exatamente o que é negado em (5):

$\neg acima(b, Y) \vee \neg acima(Y, mesa) \vee acima(b, mesa)$	**(2)**
$\neg acima(b, mesa)$	**(5)**
$\neg acima(b, Y) \vee \neg acima(Y, mesa)$	**(6)**

Pode-se agora resolver (1) com (6), trocando-se U por Y e instanciando-se V com *mesa* em (1).

$\neg sobre(Y, mesa) \vee acima(Y, mesa)$	**(1)**
$\neg acima(b, Y) \vee \neg acima(Y, mesa)$	**(6)**
$\neg sobre(Y, mesa) \vee \neg acima(b, Y)$	**(7)**

Será utilizado novamente (1), com U instanciada com b e V trocada por Y, para ser resolvida com (7):

$\neg sobre(b, Y) \vee acima(b, Y)$	**(1)**
$\neg sobre(Y, mesa) \vee \neg acima(b, Y)$	**(7)**
$\neg sobre(b, Y) \vee \neg sobre(Y, mesa)$	**(8)**

Utilizam-se agora (3) e (8), instanciando-se *Y* com *a*:

$sobre(b, a)$	**(3)**
$\neg sobre(b, a) \vee \neg sobre(a, mesa)$	**(8)**
$\neg sobre(a, mesa)$	**(9)**

Agora, a resolução de (4) e (9) resulta na cláusula vazia *nil*:

$sobre(a, mesa)$	**(4)**
$\neg sobre(a, mesa)$	**(9)**
nil	**(10)**

Assim chegou-se a uma contradição e a negação do teorema ¬*acima*(*b, mesa*) deve ser falsa. Consequentemente, o teorema *acima*(*b, mesa*) deve ser verdadeiro e a prova está concluída.

Uma questão que surge é: como se é tão astuto para considerar as cláusulas corretas para resolver? A resposta é que uma pessoa fazendo uma prova por resolução leva as seguintes vantagens:

- ❒ Ela limita-se a resolver apenas o teorema negado ou cláusulas derivadas, direta ou indiretamente, do teorema negado.
- ❒ Ela sabe em que ponto se encontra, onde deseja chegar e utiliza muita sabedoria.

Infelizmente, existem limites para o que se pode expressar considerando-se apenas os conceitos da lógica pura. Por exemplo, a lógica pura não permite expressar coisas tais como diferenças, como requerido na análise intermediária, ou distâncias heurísticas, como requerido na busca heurística. Provadores de teoremas podem utilizar tais conceitos, mas uma grande parte do encargo de resolução de problemas fica fora da declaração, em notação lógica, do que conhecido e do que deve ser feito.

Mas, embora algumas abordagens de busca distanciem-se consideravelmente da lógica pura, com outras abordagens isso não ocorre. Uma dessas abordagens, é a abordagem de **preferência unitária**, dá preferência a resoluções envolvendo as cláusulas com o menor número de literais. A abordagem de **conjunto de suporte** permite apenas resoluções envolvendo o teorema negado ou cláusulas novas geradas, direta ou indiretamente, usando o teorema negado. A

abordagem de **primeiro melhor** resolve primeiro todos os pares possíveis de cláusulas iniciais e depois resolve todos os pares do conjunto resultante com o conjunto inicial. Todas essas abordagens são denominadas **completas** porque elas garantem encontrar uma prova se o teorema resultar logicamente dos axiomas. Infelizmente, existem alguns problemas:

- Todas as abordagens de busca para resolução são sujeitas ao problema de explosão combinatória, pois as árvores de busca podem crescer muito, impedindo o sucesso de provas que requerem longas cadeias de inferência.
- Todas as abordagens de busca para resolução são sujeitas a uma versão do problema de parada, pois não se garante que a busca termina, a não ser que haja realmente uma prova.

Na realidade, todos os procedimentos de prova para o cálculo de predicados são sujeitos ao problema de parada. Procedimentos completos de prova são **semidecidíveis** porque eles garantem concluir que uma expressão é um teorema apenas se a expressão for, de fato, um teorema.

Para duas cláusulas serem resolvidas, dois literais devem casar exatamente, com a exceção de que um deles é negado. Algumas vezes, os literais casam exatamente como são. Outras vezes, os literais podem casar por meio de uma substituição apropriada.

No último exemplo visto, a parte de casamento da resolução foi fácil: a mesma constante aparecia no mesmo lugar, casando obviamente; ou uma constante aparecia no lugar ocupado por uma variável quantificada universalmente, casando porque a variável podia ser a constante observada (bem como qualquer outra constante). Em outras palavras, foram casadas constantes idênticas ou uma constante com uma variável instanciada com essa mesma constante.

Em outros casos, precisa-se de um meio melhor para acompanhar as substituições e de regras pelas quais as substituições podem ser feitas. Frequentemente, utiliza-se a seguinte notação para denotar substituições:

$$\{v_1 \rightarrow C;\ v_2 \rightarrow v_3;\ v_4 \rightarrow f(...)\}$$

Com os significados de que a variável v_1 é trocada pela constante C, a variável v_2 é trocada pela variável v_3 e a variável v_4 é trocada pela função f com seus argumentos. As regras para tais substituições informam que se pode substituir uma variável por qualquer termo que não contenha a mesma variável:

- Uma variável pode ser substituída por uma constante. Isto é, pode-se ter a substituição $\{v_1 \rightarrow C\}$.

- Uma variável pode ser substituída por outra variável. Isso pode-se ter a substituição $\{v_2 \rightarrow v_3\}$.
- Uma variável pode ser substituída por uma função desde que a função não contenha a variável. Isto é, pode-se ter a substituição $\{v_4 \rightarrow f(...)\}$.

Uma substituição que torna duas cláusula resolvíveis é chamada **unificadora** e o processo de encontrar tal substituição denominado **unificação**. Existem muitos procedimentos de substituição.

Suponha que uma expressão seja um teorema com relação a um certo conjunto de axiomas. Será que essa expressão pode ainda ser um teorema depois de acrescentarem-se alguns novos axiomas? A resposta evidentemente é sim, pois a prova do teorema pode ser feita usando-se exclusivamente os axiomas antigos e ignorando-se os axiomas novos.

Como apenas axiomas novos são acrescentados à lista de axiomas e axiomas já existentes nunca são retirados, a lógica tradicional é denominada **monotônica**.

Entretanto, a propriedade de monoticidade é incompatível com algumas formas de pensamento. Suponha, por exemplo, que alguém lhe diz que todos os pássaros voam e, diante dessa informação, você conclui que alguns pássaros, em particular, voam, é uma conclusão perfeitamente razoável pelo que você sabe. Posteriormente, você aprende que nem pinguins nem pássaros mortos voam. Acrescentar esses fatos novos pode bloquear as conclusão já feitas, mas não pode parar um provador de teoremas: apenas retificando-se os axiomas iniciais pode-se para-lo.

Lidar com esse tipo de problema requer novas formas de lógica, o que conduz a lógicas chamadas **não monotônicas**. Discutir essas lógicas estão além do escopo deste livro.

A lógica é muito boa para certos problemas, mas não é tão boa para outros. As pessoas tentam usar lógica para problemas difíceis e não apenas para aqueles para os quais a lógica conveniente. Assim, quando a lógica e um provador de teoremas parecem ser adequados, deve-se observe o seguinte:

- **Provadores de teoremas podem ser muito demorados**. Prova dores de teoremas completos requerem busca e buscas são inerentemente exponenciais. Métodos para acelerar a busca, como resolução por conjunto de suporte, reduzem o tamanho do expoente associado com a busca, mas não modificam seu caráter exponencial.
- **Provadores de teoremas podem não ajudar muito, mesmo que sejam rápidos**. Alguns conhecimentos resistem a ser formula dos em axiomas. Formular alguns problemas em lógica pode requerer um

enorme esforço, enquanto resolver o problema formulado de uma outra forma pode ser mais simples.

- **A lógica é fraca como representação para certos tipos de conhecimentos**. A notação de lógica pura não permite expressar coisas tais como distâncias heurísticas, diferenças de estados, a ideia de que alguma abordagem particular é mais rápida, ou a ideia de que algumas manipulações funcionam bem apenas se forem feitas menos de três vezes. Provadores de teoremas podem utilizar tais conhecimentos, mas eles devem ser representados utilizando-se outros conceitos além da lógica pura.

A.3 Prolog x Programação em Lógica

A linguagem Prolog foi uma das primeiras tentativas de criação de uma linguagem que permitisse ao programador especificar seus problemas usando lógica. Essa motivação explica o nome da linguagem de programação, pois Prolog é derivado de **Pro**gramação em **Lóg**ica. Na realidade, porém, programação usando Prolog não é aquilo que se chama genuinamente **programação em lógica**. Nesta seção, será examinado brevemente como Prolog está relacionada com lógica e até que ponto programação em Prolog assemelha-se à programação em lógica.

No cálculo de predicados, os objetos são representados por termos. Um termo uma das seguintes coisas:

- **Um símbolo de constante**. Pode-se imaginar uma constante como um átomo de Prolog utilizando a sintaxe dessa linguagem. Assim `brasil`, `carro` e `maria` são símbolos de constantes.
- **Um símbolo de variável**. Pode-se equiparar essas variáveis com as variáveis de Prolog. Assim, utilizando a sintaxe de Prolog, `X`, `Homem` e `Algo` são variáveis.

Um termo é composto de um símbolo de função com seus argumentos (termos). Aqui, o termo composto representa algum objeto que depende dos objetos representados pelos argumentos. Por exemplo, pode-se ter um símbolo de função representando a noção de *distância* com dois argumentos. Nesse caso, o termo composto representa a distância entre os objetos representados pelos argumentos. Pode-se imaginar um termo composto como uma estrutura de Prolog, com o símbolo de função sendo o funtor da estrutura (v. **Seção 5.4.2**). Assim, por exemplo, `distancia(X,ponto2)` pode representar a distância entre um ponto a ser especificado (`X`) e um ponto específico (`ponto2`).

Conclui-se, assim, que, no cálculo de predicados, a forma de representação de objetos é exatamente como em Prolog.

Em lógica, as proposições sobre objetos são expressas utilizando-se símbolos de predicados. Uma proposição atômica consiste de um símbolo de predicado juntamente com um conjunto de termos como argumentos. É justamente assim que os objetivos aparecem em Prolog. Assim, por exemplo, são proposições atômicas: *pessoa(maria)*, *dono(jose, carro(azul))* e *gosta(maria, Doce)*. Em Prolog, uma estrutura pode aparecer como um objetivo, como um argumento para outra estrutura ou ambos. Isso não ocorre no cálculo de predicados, em que é feita uma rígida separação entre símbolos de função — que são funtores para construção de argumentos — e símbolos de predicados — que são funtores utilizados para construção de proposições.

Além de proposições atômicas, podem-se construir proposições compostas de várias maneiras. É aqui que começam a surgir características de lógica que não possuem semelhantes em Prolog. Em lógica, são utilizados conectivos lógicos para expressar as noções familiares de negação, conjunção, disjunção, implicação e equivalência. Assim, por exemplo,

$$\boldsymbol{homem(dinorah) \vee mulher(dinorah)}$$

poderia ser utilizado para representar a proposição: Dinorah é um homem ou uma mulher, enquanto a proposição:

$$\boldsymbol{homem(jose) \rightarrow pessoa(jose)}$$

poderia representar o fato de José ser um homem implicar em ele ser uma pessoa.

Em lógica, o significado de variáveis dentro de proposições é definido apenas quando elas são introduzidas por quantificadores. Esses quantificadores proveem um meio de fazerem-se afirmações sobre conjuntos de objetos. Os dois quantificadores do cálculo de predicados serão representados como se vê na **Tabela A–2**.

Expressão Quantificada	Representação
$\forall X\,[P(X)]$	$todo(X, P(X))$
$\exists X\,[P(X)]$	$existe(X, P(X))$

Tabela A–2: Representações de Quantificadores

Exemplos de usos dos quantificadores com essa notação:

$$\boldsymbol{todo(X, homem(X) \rightarrow mortal(X))}$$

significa que, para qualquer que seja o valor de X considerado, se X é um homem então X é mortal ou, simplesmente, todo homem mortal.

***existe*(*Z*, *pai*(*jose*, *Z*) ∧ *mulher*(*Z*))**

significa que existe algo representado por Z tal que José o é pai de Z e Z é mulher ou, simplesmente, José tem uma filha.

Será visto que uma proposição do cálculo de predicados na forma clausular é muito parecida com um conjunto de cláusulas de Prolog. Assim um exame mais profundo sobre a forma clausular é essencial para entender a relação entre Prolog e lógica.

A conversão de uma fórmula do cálculo de predicados na forma clausular tem nove passos, conforme foi visto na **Seção A.2**:

Passo 1: Elimine as implicações

Passo 2: Mova as negações para as fórmulas atômicas

Passo 3: Remova os quantificadores existenciais (skolemize)

Passo 4: Renomeie variáveis quando necessário

Passo 5: Mova os quantificadores universais para a esquerda

Passo 6: Forme uma conjunção de disjunções de literais

Passo 7: Separe as conjuntas

Passo 8: Renomeie as variáveis quando necessário

Passo 9: Remova os quantificadores universais

Qualquer fórmula do cálculo de predicados é equivalente a um conjunto de cláusulas. A forma clausular consiste de um conjunto de cláusulas, cada uma das quais é uma disjunção de literais. É importante relembrar as convenções que foram feitas quando o significado de alguma proposição na forma clausular for examinado.

Será apresentado a seguir mais um exemplo de transformação de uma fórmula para a forma clausular, utilizando agora a nova notação. A fórmula a ser utilizada é:

***todo*(*X*, *todo*(*Y*, *pessoa*(*Y*) → *respeita*(*Y*, *X*)) → *rei*(*X*))**

que diz que se todos respeitam alguém, então esse alguém é um rei. (Isto é, para cada X, se cada Y que uma pessoa respeita X, então X um rei). Removendo as implicações (**Passo 1**), resulta em:

***todo*(*X*, ¬(*todo*(*Y*, ¬*pessoa*(*Y*) ∨ *respeita*(*Y*, *X*))) ∨ *rei*(*X*))**

Movendo-se as negações para as proposições (**Passo 2**), tem-se:

$$todo(X, existe(Y, pessoa(Y) \wedge \neg respeita(Y, X)) \vee rei(X))$$

Após a skolemização (**Passo 3**), obtém-se:

$$todo(X, (pessoa(f_1(X)) \wedge \neg respeita(f_1(X), X)) \vee rei(X))$$

em que f_1 é uma função de Skolem. Os **Passos 4** e **5** são desnecessários aqui, pois existe uma única variável e o único quantificador universal já está à esquerda. A aplicação do **Passo 6** resulta em:

$$todo(X, (pessoa(f_1(X)) \vee rei(X)) \wedge (\neg respeita(f_1(X), X) \vee rei(X)))$$

Separando-se as conjuntas (**Passo 7**), obtém-se:

$$todo(X, pessoa(f_1(X)) \vee rei(X))$$
$$todo(X, \neg respeita(f_1(X), X) \vee rei(X))$$

Finalmente, isso resulta em duas cláusulas (**Passo 9**):

$$pessoa(f_1(X)) \vee rei(X)$$
$$\neg respeita(f_1(X), X) \vee rei(X)$$

Não necessária a renomeação das variáveis (**Passo 8**). A justificativa para tal simplificação que se está considerando a mesma regra de escopo de Prolog. Isto é, variáveis em cláusulas diferentes são diferentes. Assim a variável `X` da primeira cláusula diferente da variável *X* da segunda cláusula.

A seguir, será apresentada uma outra maneira de escrita de proposições na forma clausular. Uma boa convenção consiste em escrever uma cláusula após a outra, lembrando que a ordem em que elas aparecem é irrelevante. Dentro de uma cláusula há alguns literais negados e outros não negados. Adotar-se-á convenção de escrever os literais não negados primeiro e depois os literais negados. Os dois grupos serão separados pelo símbolo `:-`. Os literais não negados serão separados por `;` (ponto e vírgula) e os literais negados serão escritos sem o símbolo de negação ¬ e separados por `,` (vírgula). Finalmente, uma cláusula será terminada por um ponto. Nessa notação, uma cláusula com literais negados *K*, *L*, ... e literais não negados *A*, *B*, ... seria escrita como:

```
A; B; ... :- K, L, ... .
```

Embora essa convenção para escrita de cláusulas seja um tanto arbitrário, de fato, ela tem realmente algum significado mnemônico. Quando uma cláusula é escrita com os literais negados separados dos literais não negados, tem-se:

$$(A \vee B \vee ...) \vee (\neg K \vee \neg L \vee ...).$$

que é equivalente a:

$$(A \vee B \vee ...) \vee \neg(K \wedge L \wedge ...).$$

que é equivalente a:

$$(K \wedge L \wedge ...) \rightarrow (A \vee B \vee ...).$$

Substituindo-se $\wedge$ por **,** (vírgula), $\vee$ por **;** e $\rightarrow$ por **:-** seguindo a convenção de Prolog, essa cláusula torna-se:

```
A; B; ... :- K, L, ... .
```

Dadas essas convenções, a fórmula a seguir, com X e Y quantificadas universalmente:

$$pessoa(adao) \wedge pessoa(eva) \wedge (pessoa(X) \vee \neg mae(X, Y)) \vee \neg pessoa(Y))$$

transforma-se em:

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

Isso é semelhante a uma definição em Prolog para o significado de ser uma pessoa. Entretanto, algumas fórmulas dão origem a coisas intrigantes nessa notação. Por exemplo, uma cláusula contendo apenas os literais não negados A, B e C seria escrita nessa notação como:

```
A; B; C :- .
```

que não corresponde a nada em Prolog.

Agora que se tem um meio para escrita de fórmulas do cálculo de predicados numa forma elegante, deve-se considerar o que se pode fazer com elas. É interessante investigar se alguma coisa resulta das proposições, i.e., que consequências elas têm.

Na década de 60, muitos pesquisadores começaram a investigar a possibilidade de os computadores poderem ser programados para provar teoremas. Foi essa área de pesquisa científica que deu surgimento às ideias básicas de Prolog. Uma das principais façanha dessa época foi a descoberta do **princípio de resolução** por J. A. **Robinson** com sua aplicação à **prova automática de teoremas**. Usando-se o princípio de resolução de Robinson, pode-se provar teoremas de

um modo puramente mecânico a partir dos axiomas. O princípio de resolução foi elaborado para lidar com fórmulas na forma clausular.

Dadas duas cláusulas relacionadas de uma maneira apropriada, é gerada uma nova cláusula que é uma consequência das cláusulas dadas. A ideia básica é que se a mesma fórmula atômica aparece simultaneamente no lado esquerdo de uma cláusula e no lado direito de outra, então a cláusula obtida, juntando-se as duas cláusulas e abandonando-se as fórmulas duplicadas, resulta das cláusulas dadas. Por exemplo, das cláusulas

***triste*(*jose*); *zangado*(*jose*) :- *dia_util*(*hoje*), *chovendo*(*hoje*).**

e

***infeliz*(*jose*) :- *zangado*(*jose*), *cansado*(*jose*).**

obtém-se:

***triste*(*jose*); *infeliz*(*jose*) :-**
** *dia_util*(*hoje*), *chovendo*(*hoje*), *cansado*(*jose*).**

Em português: se hoje um dia útil e está chovendo, então José está triste ou zangado. Também, se José está zangado e cansado, ele está infeliz. Portanto, se hoje dia de trabalho e José está cansado, então José está triste ou infeliz.

Quando as cláusulas contêm variáveis, as duas fórmulas atômicas não têm que ser idênticas; i.e., elas têm apenas que casar. Além disso, a cláusula que resulta das duas primeiras (com as fórmulas duplicadas removidas) é obtida por meio de uma operação extra. Essa operação envolve instanciar as variáveis para que as fórmulas a ser casadas sejam idênticas. Em termos de Prolog, se houver duas cláusulas que casam, o resultado da junção delas é a representação da nova cláusula.

Em resolução, vários literais num lado direito podem casar com vários literais num lado esquerdo. Aqui, só serão considerados exemplos nos quais um literal de cada cláusula é escolhido.

Exemplo de resolução envolvendo variáveis:

***pessoa*(f_1(*X*)) ; *rei*(*X*).** (1)
***rei*(*Y*) :- *respeita*(f_1(*Y*), *Y*).** (2)
***respeita*(*Z*, *artur*) :- *pessoa*(*Z*).** (3)

As duas primeiras cláusulas são obtidas como forma clausular da fórmula:

***todo*(*X*, *todo*(*Y*, *pessoa*(*Y*) → *respeita*(*Y*, *X*)) → *rei*(*X*)).**

que significa: se toda pessoa respeita alguém, então esse alguém é um rei. As variáveis foram renomeadas para facilitar a explicação. A terceira cláusula expressa a proposição: toda pessoa respeita Artur. Resolvendo (2) com (3), obtém-se:

***rei*(*artur*) :- *pessoa*(f_1(*artur*)).** (4)

[Y em (2) casa com *artur* em (3) e Z em (3) casa com $f_1(Y)$ em (2)]. Pode-se agora resolver (1) com (4), resultando em:

***rei*(*artur*) ; *rei*(*artur*) :- .** (5)

Isso equivalente ao fato: Artur é um rei (por que?).

Na definição formal de resolução, o processo de casamento utilizado aqui é denominado **unificação**. Intuitivamente, fórmulas atômicas são unificáveis se, vistas como estruturas de Prolog, elas podem casar. Na realidade, será visto que o casamento em muitas implementações de Prolog não é exatamente o mesmo que unificação.

Uma maneira de se tentar provar algo com resolução é aplicando continuamente passos de resolução sobre as hipóteses (axiomas) e verificando se o que se deseja provar aparece. Infelizmente, não se pode garantir que isso irá acontecer, mesmo quando a proposição que se está interessado realmente resulta das hipóteses. Considerando o último exemplo, não há nenhum meio de derivar-se a cláusula simples *rei*(*artur*) a partir das cláusulas dadas, apesar de ela ser claramente uma consequência. Diante do exposto, pode-se concluir que a resolução não é suficientemente poderosa para o que se anseia? Felizmente, a resposta é *não*, pois pode-se redefinir os objetivos de tal maneira que se garanta que a resolução seja capaz de resolver o problema se isso for possível.

Uma propriedade formal importante da resolução que ela é **completa por refutação**. Isso significa que, se um conjunto de cláusulas é inconsistente, a resolução capaz de derivar delas a cláusula vazia:

:- .

Um conjunto de cláusulas é **inconsistente** quando não há nenhuma interpretação possível para os predicados, símbolos de constantes e símbolos de função que o faça, simultaneamente, expressar proposições verdadeiras. A cláusula vazia expressa contradição lógica. Isto é, ela representa uma proposição que não pode ser verdadeira. Assim a resolução garante informar quando as fórmulas são inconsistentes derivando essa expressão de contradição.

Sabe-se que se as fórmulas $\{A_1, A_2, ..., A_n\}$ forem consistentes, a fórmula B será uma consequência dessas fórmulas exatamente quando as fórmulas $\{A_1, A_2, ...,$

A_n, $\neg B\}$ forem inconsistentes. Assim, se as hipóteses forem consistentes, será preciso apenas acrescentar a elas as cláusulas para a negação da proposição que se deseja provar. A resolução irá derivar a cláusula vazia exatamente quando essa proposição resultar das hipóteses. As cláusulas que são acrescentadas às hipóteses são chamadas **declarações de objetivos**. Note que as declarações de objetivos não são diferentes das hipóteses, no sentido de que todas elas são cláusulas. Assim, quando um conjunto de cláusulas A_1, A_2, ..., A_n é fornecido e a tarefa é mostrar que elas são inconsistentes, não se pode realmente dizer se o correto é mostrar que $\neg A_1$ resulta de A_2, ..., A_n, ou que $\neg A_2$ resulta de A_1, A_3, ..., A_n ou que $\neg A_3$ resulta de A_1, A_2, ..., A_n etc. Portanto é importante enfatizar quais são as cláusulas consideradas declarações de objetivos, porque num sistema de resolução todas essas tarefas são equivalentes.

No exemplo anterior sobre o rei Artur, fácil verificar como se pode obter a cláusula vazia se for acrescentada a declaração de objetivo:

:- rei(artur). **(6)**

[Isto é, a cláusula para $\neg$*rei*(*artur*)]. Viu-se anteriormente como a cláusula:

rei(artur); rei(artur) :- . **(5)**

foi derivada das hipóteses. Resolvendo **(5)** com **(6)**, obtém-se:

rei(artur) :- . **(7)**

Finalmente, resolvendo **(6)** com **(7)**, obtém-se a cláusula vazia:

:- .

Assim a resolução mostrou que, como consequência do conjunto de cláusulas dadas, Artur é rei.

A completitude por refutação da resolução é uma propriedade lógica atraente. Ela significa que se algum fato resulta das hipóteses, prova-se sua veracidade (mostrando-se a inconsistência de sua negação com as hipóteses) usando-se resolução. Entretanto, quando se diz que a resolução capaz de derivar a cláusula vazia, se quer dizer que existe uma sequência de passos de resolução, cada um deles envolvendo axiomas ou cláusulas derivadas de passos anteriores, que termina na produção de uma cláusula sem nenhum literal. A única dificuldade é encontrar essa sequência de passos, pois, embora a resolução informe como derivar uma consequência de duas cláusulas, ela não informa como decidir que cláusulas considerar num dado instante ou que literais casar. Usualmente, tem-se um grande número de hipóteses e há muitas possibilidades de resolvê-las. Além disso, cada vez que se deriva uma nova cláusula, ela também torna-se

candidata a participar de resoluções posteriores. Muitas possibilidades são irrelevantes para a tarefa em questão e se não se tiver cuidado pode-se desperdiçar muito tempo com tarefas irrelevantes que nunca irão conduzir solução.

Muitos refinamentos do princípio de resolução original foram propostos para lidar com essas questões. Serão vistos agora alguns refinamentos para a resolução quando todas as cláusulas são de um tipo especial — as chamadas **cláusulas de Horn**.

Uma cláusula de Horn é uma cláusula com no máximo um literal não negado. Ocorre que, quando se está usando um provador de teoremas clausular para determinar os valores de funções computáveis, é estritamente necessário usar apenas cláusulas de Horn. Como resolução usando cláusulas de Horn é relativamente simples, ela é uma escolha natural como base de um provador de teoremas para um sistema de programação de natureza prático. Ver-se-á brevemente como a prova de teoremas por resolução restrita a cláusulas de Horn.

Existem obviamente dois tipos de cláusulas de Horn: aquelas com um literal não negado e aquelas sem nenhum literal não negado. Essas cláusulas serão chamadas, respectivamente, **cláusulas com cabeça** e **cláusulas sem cabeça**. Os dois tipos são exemplificados em seguida (lembre-se que os literais não negados são escritos do lado esquerdo de **:-**):

***solteiro*(*X*) :- *homem*(*X*), *descasado*(*X*).**

***:- solteiro*(*X*).**

Quando se consideram conjuntos de cláusulas de Horn (inclusive declarações de objetivos), é necessário apenas considerar os conjuntos nos quais todas as cláusulas, com exceção de uma delas, são cláusulas com cabeça. Isto é, qualquer problema solúvel que pode ser expresso em cláusulas de Horn é expresso numa forma tal que:

- ❒ Existe apenas uma cláusula sem cabeça
- ❒ Todas as outras cláusulas têm cabeça

Uma vez que é arbitrário decidir que cláusulas são realmente os objetivos, pode-se convencionar que a cláusula sem cabeça é o objetivo e as outras cláusulas são as hipóteses.

Mas, por que se deve considerar apenas os conjuntos de cláusulas de Horn que se ajustam a esse padrão? Primeiro, é fácil verificar que pelo menos uma cláusula sem cabeça deve estar presente para um problema ser solúvel. Isso deve-se ao fato de o resultado da resolução de duas cláusulas de Horn com cabeça ser também uma cláusula com cabeça. Assim, se todas as cláusulas tiverem

cabeça, será possível apenas derivar outras cláusulas com cabeça e, como a cláusula vazia não possui cabeça, não será possível derivá-la. A justificativa para o segundo requisito — que é necessário apenas uma cláusula sem cabeça — é um pouco mais sutil. Mas, se existirem várias cláusulas sem cabeça entre os axiomas, qualquer prova por resolução de uma cláusula pode ser convertida numa prova que utiliza no máximo uma das cláusulas sem cabeça. Portanto, se a cláusula vazia puder ser derivada dos axiomas, ela poder ser derivada das cláusulas com cabeça com, no máximo, uma das cláusulas sem cabeça.

Será resumido agora como Prolog se ajusta nesse esquema. Foi visto acima que algumas fórmulas foram transformadas em cláusulas muitos parecidas com cláusulas de Prolog, enquanto outras não pareciam tanto. As cláusulas que foram transformadas em cláusulas semelhantes às cláusulas de Prolog foram exatamente aquelas que foram transformadas em cláusulas de Horn. Quando se escreve uma cláusula de Horn, no máximo uma fórmula atômica aparece esquerda de `:-`. Em geral, as cláusulas podem ter várias dessas fórmulas, que correspondem aos literais que são fórmulas atômicas não negadas. Apenas as cláusulas de Horn podem ser expressas diretamente em Prolog.

As cláusulas de um programa em Prolog correspondem às cláusulas de Horn com cabeça. Mas, a que corresponde em Prolog uma declaração de objetivo? A resposta é simples, a questão em Prolog:

```
?- A1, A2, ..., An.
```

corresponde exatamente à cláusula de Horn sem cabeça:

:- A1, A2, ..., An.

Foi visto que, para qualquer problema que se queira resolver com cláusulas de Horn, é suficiente que se tenha exatamente uma cláusula sem cabeça. Isso corresponde à situação em Prolog na qual todas as cláusulas do programa têm cabeça e apenas um objetivo (sem cabeça) é considerado de cada vez.

Um sistema Prolog é baseado num provador de teoremas por resolução para cláusulas de Horn. A abordagem específica que ele utiliza é um tipo de **resolução linear de entrada**. Quando essa abordagem é utilizada, a escolha do que resolver e com que em qualquer instante é restringida como será visto em seguida. Começa-se com a declaração de objetivo e resolve-se ele com uma das hipóteses, resultando numa cláusula nova. Então resolve-se essa cláusula nova com uma das hipóteses e assim por diante. Em cada estágio, resolve-se a última cláusula obtida com uma das hipóteses originais. Em nenhum momento, utiliza-se uma cláusula que tenha sido derivada previamente ou resolvem-se duas das hipóteses. Em termos de Prolog, pode-se ver a última cláusula derivada

como a próxima conjunção de objetivos a ser satisfeita. O processo começa com a questão introduzida e, se for bem sucedido, termina com a cláusula vazia. Em cada estágio, encontra-se uma cláusula cuja cabeça casa com um dos objetivos, instanciam-se variáveis quando necessário, remove-se o objetivo que casou e então acrescenta-se o corpo da cláusula instanciada aos objetivos a serem satisfeitos. Assim, por exemplo, pode-se partir de:

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

e

```
mae(U, V) :- pais(U, V), mulher(V).
```

e chegar em

```
:- pais(jose, X), mulher(X), mae(X, Y).
```

Na realidade, a abordagem de prova de Prolog ainda é mais restrita do que a resolução linear de entrada. Nesse exemplo, decidiu-se casar o primeiro dos literais na cláusula objetivo, mas seria do mesmo modo possível ter casado o segundo literal. Em Prolog, o literal a ser casado sempre é o primeiro da cláusula objetivo como. Além disso, os novos objetivos derivados do uso de uma cláusula são colocados na frente da cláusula-objetivo, o que significa que Prolog termina de satisfazer um subobjetivo antes de continuar a tentar qualquer outra opção.

Como Prolog organiza a busca de cláusulas alternativas para satisfazer o mesmo objetivo? Bem, basicamente, Prolog adota uma abordagem de **busca em profundidade** que é sujeita a ciclos, mas muito é mais simples e consome menos espaço ao ser implementada em computador[1].

Deve-se notar que, algumas vezes, os casamentos efetuados por Prolog diferem da unificação utilizada em resolução. Muitas implementações de Prolog permitem que sejam satisfeitos objetivos tais como:

```
igual(X, X).
?- igual(f(Y), Y).
```

Isto é, pode-se casar um termo com um subtermo dele próprio. Nesse exemplo, `f(Y)` casado com `Y`, que aparece dentro dele. Como resultado, `Y` irá representar `f(Y)`, que é `f(f(Y))`, que é `f(f(f(Y)))` e assim por diante. Assim `Y` representa um tipo de estrutura infinita. Embora muitas implementações de Prolog permitir a construção desse tipo, elas não são capazes de escrevê-las no final. De acordo com a definição formal de unificação, esse tipo de termo

[1] O algoritmo de busca em profundidade não é difícil de entender, mas isso requer discutir conceitos que estão além do escopo deste livro.

infinito não deve nunca existir. Assim Prolog não age corretamente como um provador de teoremas por resolução nesse aspecto. Para superar isso, deve-se acrescentar um teste para garantir que uma variável não seja instanciada com alguma estrutura que a contenha. Tal teste de verificação seria fácil de implementar, mas retardaria consideravelmente a execução de programas em Prolog. Como, tipicamente, isso seria aplicado a muito poucos programas, a maioria das implementações não inclui esse teste de fora.

Viu-se que Prolog baseia-se na ideia de um provador de teoremas. Como resultado, pode-se ver que os programas escritos em Prolog são como as hipóteses que se têm sobre o mundo e as questões são teoremas que se deseja que sejam provados. Assim programar em Prolog não é como informar ao computador como e quando fazer algo. Em vez disso, informa-se o que verdadeiro e solicita-se que a linguagem tente tirar conclusões. A ideia de que programação deve ser assim tem levado muitas pessoas a investigar a noção de **programação em lógica** como uma possibilidade prática. A principal vantagem da programação em lógica é que os programas devem ser mais fáceis de ler, pois eles não são misturados com detalhes sobre como os problemas devem ser resolvidos. Assim, em programação em lógica, sabe-se *o que* um programa faz, em vez de *como* ele o faz.

Será examinado agora Prolog como uma linguagem candidata a linguagem de programação em lógica. Inicialmente, é claro que alguns programas em Prolog representam verdades lógicas sobre o mundo. Por exemplo, se for escrito:

```
mae(X, Y) :- pais(X, Y), mulher(Y).
```

pode-se ver isso como uma informação sobre o que deve ser uma mãe (i.e., uma mulher que é um dos pais de uma pessoa). Assim essa cláusula expressa uma proposição que se considera verdadeira, bem como diz como mostrar que alguém uma mãe.

De modo similar, as cláusulas:

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

informam o que significa uma lista ser anexada na frente de outra. Ou seja, se a lista vazia for colocada na frente de alguma lista `X`, o resultado será exatamente `X`. Por outro lado, se uma lista não vazia for anexada na frente de uma lista, então a cabeça da lista resultante será a mesma cabeça da lista que estiver sendo colocada na frente. Além disso, a cauda da lista resultante será a lista obtida anexando-se a cauda da primeira lista na frente da segunda. Assim

essas cláusulas podem ser vistas como uma expressão daquilo que é verdadeiro sobre a relação *anexa*.

Agora, que possíveis significados lógicos podem ser dados a cláusulas como as seguintes?

```
membro1(X, Lista) :- var(Lista), !, fail.
membro1(X, [X|_]).
membro1(X, [_|Lista]) :- membro1(X, Lista).

exibe(0) :- !.
exibe(N) :- write(N), N1 is N - 1, exibe(N1).

nome(N) :- name(N, Nome1), anexa(Nome2, [115], Nome1),
           name(R, Nome2),
nome(R).

implica(Sup, Conc) :-
   asserta(Sup),
   call(Conc),
   retract(Conc).
```

O problema surge com todos os predicados embutidos utilizados em Prolog. Por exemplo, `var(Lista)` não diz nada sobre listas ou pertinência, mas refere-se a um estado de coisas que pode valer durante a prova. Similarmente, o corte (v. **Seção 5.11**) refere-se a *como* se deve proceder na prova de uma proposição, em vez de referir-se à própria proposição. Esses dois predicados podem ser considerados como meios de expressar informações de controle sobre como a prova deve ser executada. Similarmente, por exemplo, `write(N)` não possui nenhuma propriedade lógica interessante, mas pressupõe que a prova atingirá um certo estado (com `N` instanciada) e efetuará uma comunicação com o programador em seu terminal. O objetivo `name(N, Nome1)` no último exemplo informa algo sobre a estrutura interna do que seria um átomo indivisível no cálculo de predicados. Em Prolog, pode-se converter símbolos em cadeias de caracteres, estruturas em listas e estruturas em cláusulas. Essas operações violam a natureza simples das proposições do cálculo de predicados. No último exemplo, o uso de `asserta` significa que a regra refere-se a acrescentar alguma coisa ao conjunto de axiomas. Em lógica, cada fato ou regra afirma uma verdade independente da existência de outras regras e fatos. Aqui, tem-se uma regra que viola esse princípio. Também, quando utiliza-se essa regra, tem-se um conjunto diferente de axiomas em diferentes instantes da prova. Finalmente, o fato de a regra considerar `Conc` como um objetivo significa que permite-se a uma variável lógica representar uma proposição, o que não pode ser expresso em cálculo de predicados de primeira ordem, mas sim em lógicas de ordens superiores.

Dados esses exemplos, pode-se ver que alguns programas em Prolog podem ser entendidos apenas em termos declarativos, mas, por outro lado, existem programas para os quais dificilmente podem ser feitas interpretações declarativas devido ao aspecto algorítmico incorporado no programa.

Assim faz sentido considerar Prolog como uma linguagem de programação lógica? Pode-se realmente esperar que alguma vantagem da programação em lógica aplique-se aos programas em Prolog? A resposta para ambas as questões é *sim* e a razão é que, adotando-se um estilo de programação apropriado, pode-se obter ainda algumas vantagens da relação de Prolog com a lógica. O segredo está em decompor os programas em partes, confinando o uso de operações não lógicas dentro de um pequeno conjunto de cláusulas. Como exemplo, viu-se no **Capítulo 5** como usos do corte podem ser substituídos por usos de `not`. Como resultado de tais substituições, um programa contendo vários cortes pode ser reduzido a um programa com o corte usado apenas uma vez (na definição de `not`). O uso do predicado `not`, mesmo que não capture exatamente o sentido da negação lógica, permite que se recupere parte do significado lógico básico de um programa. Analogamente, confinar os usos de `asserta` e `retract` dentro das definições de um pequeno número de predicados resulta num programa que é, em geral, mais claro do que aqueles nos quais esses predicados são utilizados arbitrariamente em todos os tipos de contexto.

Concluindo, o objetivo principal de uma linguagem de programação em lógica não foi atingido por Prolog. Contudo, Prolog apresenta-se como um sistema de programação prático que tem algumas das vantagens de claridade e declaratividade que uma linguagem de programação em lógica poderia oferecer. Por enquanto, continua o trabalho para desenvolverem-se versões melhoradas de Prolog que sejam mais próximas da lógica do que as que se tem correntemente disponível. Dentre as maiores prioridades dos pesquisadores dessa área está o desenvolvimento de um sistema prático que não necessite do corte e tenha uma versão do `not` que corresponda exatamente noção lógica de negação.