UNIVERSIDADE DE SÃO PAULO

O cálculo proposicional: uma abordagem voltada à compreensão da linguagem Prolog

Versão 1.0

MARIA CAROLINA MONARD; MARIA DO CARMO NICOLETTI

RAUL HIDEO NOGUCHI

Nº 5

NOTAS DIDÁTICAS

Instituto de Ciências Matemáticas de São Carlos

ISNN - 0103-2585

O cálculo proposicional: uma abordagem voltada à compreensão da linguagem Prolog

Versão 1.0

MARIA CAROLINA MONARD; MARIA DO CARMO NICOLETTI
RAUL HIDEO NOGUCHI

Nº 5

NOTAS DIDATICAS DO ICMSC

São Carlos (SP)
ago. 1992

O Cálculo Proposicional: uma Abordagem Voltada à Compreensão da Linguagem Prolog

Maria Carolina Monard [1]
Universidade de São Paulo / ILTC
Instituto de Ciências Matemáticas de São Carlos
Departamento de Ciências de Computação e Estatística

Maria do Carmo Nicoletti
Universidade Federal de São Carlos / ILTC
Departamento de Computação

Raul Hideo Noguchi [1]
Universidade de São Paulo
Instituto de Ciências Matemáticas de São Carlos
Departamento de Ciências de Computação e Estatística

Versão 1.0

Agosto 1992

[1]Trabalho realizado com auxílio do RHAE/ILTC — Instituto de Lógica Filosofia e Teoria da Ciência

Conteúdo

1 Introdução

A Lógica pode ser considerada como a formalização de alguma linguagem. A formalização de uma linguagem consiste de três partes:

1. sintaxe
2. semântica
3. dedução

A sintaxe de uma linguagem é uma especificação precisa das expressões legais nesta linguagem. O componente semântico captura o significado das expressões na linguagem. O componente dedutivo da lógica provê regras — para manipular expressões — que preservam certos aspectos semânticos da linguagem.

Prolog é uma linguagem de programação lógica baseada em uma combinação de idéias poderosas que incluem:

- o uso de cláusula de Horn para representar conhecimento;
- estilo declarativo de programação;
- semântica declarativa bem como procedimental;
- habilidade de misturar código no metanível com código no nível objeto, etc.

Frequentemente Prolog é utilizada intuitivamente. Entretanto, é necessário um sistema formal para atribuir um significado preciso às cláusulas Prolog, bem como para definir deduções válidas. O sistema formal utilizado por Prolog é o Cálculo de Predicados de Primeira Ordem — ou Lógica de Predicados de Primeira Ordem.

Esta Nota Didática é dedicada a um subconjunto simples do Cálculo de Predicados chamado Cálculo Proposicional [2]. Deve ser ressaltado que exitem muitas publicações relacionadas a este assunto, algumas delas listadas nas Referências desta Nota. O objetivo principal desta Nota é o de introduzir conceitos fundamentais do Cálculo Proposicional para um melhor entendimento da linguagem de programação lógica Prolog.

São também apresentadas algumas implementações Prolog para a solução de alguns problemas típicos do Cálculo Proposicional.

2 Cálculo Proposicional

O Cálculo Proposicional -CP- (também conhecido como Lógica Proposicional) é um dos mais simples formalismos lógicos existentes. Este Cálculo lida apenas com enunciados declarativos, chamados de *proposições*. As sentenças exclamativas, imperativas e interrogativas são, pois, excluídas.

[2] O Cálculo de Predicados será abordado numa próxima Nota Didática.

2.1 Proposições

Não é o objetivo deste trabalho introduzir formalmente o que é uma proposição (ou sentença). Intuitivamente, sabe-se que na linguagem falada ou escrita, o elemento básico é a proposição, a qual se distingue do *termo*, este, por sua vez, usado para designar um objeto.

Exemplos de termos: Paula
Um filme de terror
Triângulo retângulo

Informalmente, uma proposição é uma sentença declarativa que pode assumir os valores *verdade* (**v**) ou *falso* (**f**), um excluindo o outro, e conhecidos como *valores-verdade* da proposição. Por exemplo:

a) Todo homem é mortal
b) Meu carro é um fusca
c) Está chovendo
d) Se Maria estuda então fará bons exames
e) Ele come e dorme.

As proposições dos exemplos a), b) e c) são *proposições atômicas* (ou *átomos*), uma vez que nelas não aparecem os conectivos **e**, **ou**, **se ... então**, **se e somente se**, explicados a seguir. As proposições dos exemplos d) e e) são *compostas*, isto é, são proposições onde aparecem um ou mais dos conectivos citados.

O Cálculo Proposicional lida apenas com proposições que têm valores-verdade **v** ou **f**. Por esta razão diz-se que vale o *princípio do terceiro excluído* ou que a lógica é *bivalente*. Qualquer proposição que não tenha o valor-verdade **v**, necessariamente terá que ter o valor-verdade **f**. Uma expressão do tipo:

A distância entre Paraty e Ubatuba

não é uma sentença e, portanto, não é tratada por este Cálculo.

As proposições atômicas, ou átomos, serão designadas por letras latinas minúsculas p, q, r,..., chamadas de *letras proposicionais*. Um *literal* é um átomo ou a negação de um átomo.

2.2 Proposições Compostas

É possível construir proposições compostas através do uso de conectivos (usados para construir proposições a partir de outras). Os conectivos são:

e ($\wedge$) com este conectivo, a partir de duas proposições, obtém-se uma terceira chamada *conjunção*. Assim, de:

Maria estuda o problema e José vai pescar

pode-se formar a conjunção:

Maria estuda o problema e José vai pescar

A conjunção de duas proposições tem o valor **v** somente se ambas tiverem o valor-verdade **v**.

ou ($\vee$) com este conectivo, a partir de duas proposições, obtém-se uma terceira, chamada *disjunção*. Assim, de:

Maria estuda o problema ou José vai pescar

pode-se formar a disjunção:

Maria estuda o problema ou José vai pescar

A disjunção de duas proposições tem o valor **f** somente se ambas tiverem o valor-verdade **f**.

não ($\neg$) nega o valor-verdade de uma proposição (operador).

condicional ($\rightarrow$) com o conectivo condicional lido como **se ... então ...**, de duas proposições obtém-se uma terceira, chamada *condicional* ou *implicação*. Por exemplo, a proposição condicional:

Se eu como muito então eu engordo

é obtida das sentenças:

eu como muito e eu engordo

A condicional $p \rightarrow q$ tem o valor-verdade **f** somente se os valores-verdade de p e q forem **v** e **f** respectivamente.

bicondicional ($\leftrightarrow$) utilizando o conectivo bicondicional lido como **...se e somente se ...**, de duas proposições obtém-se uma terceira chamada *bicondicional*. Por exemplo:

Um triângulo ABC é retângulo se e somente se tem um ângulo reto

A bicondicional $p \leftrightarrow q$ tem valor-verdade **v** se e somente se os valores-verdade de p e q são ambos **v** ou ambos **f**.

Deve ser observado que em expressões em linguagem natural frequentemente **p ou q** é usado com o significado de: **ou p é v, ou q é v, mas não ambos**. Por exemplo:

Ele está jogando futebol ou está nadando

Este ou é chamado de *ou exclusivo*. No Cálculo Proposicional usa-se o *ou inclusivo*, ou seja, usa-se *p ou q* com o significado de: *ou p é* **v**, *ou q é* **v**, *ou ambos são* **v**.

Por exemplo, a sentença **chove ou faz frio** é verdadeira nos casos em que: chove, faz frio, chove e faz frio.

2.3 Fórmulas Bem Formadas

Como visto na seção anterior, novas proposições podem ser construídas através da combinação de símbolos, que representam proposições, e de conectivos lógicos. Essas proposições — bem como as proposições atômicas — são chamadas de *fórmulas bem formadas* — *wff* (well-formed formula).

Para representar fórmulas bem formadas são usadas *metavariáveis proposicionais* representadas, por enquanto, pelas letras gregas α, β, γ, etc.

Uma wff é definida recursivamente como segue:

1. um átomo é uma wff.
2. se α e β são wff, então as seguintes também são wff:

wff	lida como
$\neg\alpha$	não α
$\alpha \wedge \beta$	α e β
$\alpha \vee \beta$	α ou β
$\alpha \rightarrow \beta$	se α então β α implica β
$\alpha \leftrightarrow \beta$	α se e somente se β α é equivalente a β

3. as únicas wff são aquelas definidas por 1. e 2.

Cada uma das expressões envolvendo α e β é chamada de *forma sentencial*. Uma forma sentencial é uma especificação abstrata da sintaxe de um número infinito de wff compostas de símbolos que representam proposições atômicas.

Uma wff que sintaticamente se ajusta a uma forma sentencial é chamada de uma *instância de substituição* da forma sentencial.

Por exemplo, a wff: $p \wedge (q \rightarrow r)$ é uma instância de substituição de qualquer uma das seguintes formas sentenciais:

1. α onde α é p $\wedge$ (q $\rightarrow$ r)

2. $\alpha \wedge \beta$ onde α é p e β é q $\rightarrow$ r

3. $\alpha \wedge (\beta \rightarrow \gamma)$ onde α é p, β é q e γ é r

2.4 Tabelas-Verdade das Fórmulas Bem Formadas

A forma de uma wff, isto é, a maneira como é construída a partir de proposições e conectivos lógicos, é denominada de *sintaxe da wff.*

A *semântica* — ou significado — de uma wff (tratada na seção 2.5) é o valor-verdade a ela associado.

Os valores-verdade de uma wff são definidos em termos dos valores-verdade de seus componentes. Geralmente uma *tabela-verdade* é usada para tabular os valores-verdade de uma fórmula em termos dos valores-verdade de seus componentes, como mostrado a seguir:

α	β	$\alpha \wedge \beta$	$\alpha \vee \beta$	$\neg\alpha$	$\alpha \rightarrow \beta$	$\alpha \leftrightarrow \beta$
v	v	v	v	f	v	v
v	f	f	v	f	f	f
f	v	f	v	v	v	f
f	f	f	f	v	v	v

Prioridade dos Conectivos

Dada a wff $\alpha \vee \beta \rightarrow \gamma$, existe a dúvida desta wff ser $(\alpha \vee \beta) \rightarrow \gamma$ ou $\alpha \vee (\beta \rightarrow \gamma)$. Este problema pode ser contornado através do estabelecimento de uma hierarquia total ou parcial entre os conectivos. A convenção de prioridade que a maioria dos autores estabelece para os conectivos tem a seguinte ordem decrescente de precedência:

$\neg$ maior
$\wedge$
$\vee$ $\downarrow$
$\rightarrow$
$\leftrightarrow$ menor

$\alpha \rightarrow \beta \vee \gamma$ significa $\alpha \rightarrow (\beta \vee \gamma)$
$\alpha \vee \beta \wedge \gamma$ significa $\alpha \vee (\beta \wedge \gamma)$
$\alpha \rightarrow \beta \wedge \neg\gamma \vee \delta$ significa $\alpha \rightarrow ((\beta \wedge (\neg\gamma)) \vee \delta)$

A precedência estabelecida pode ser alterada através da introdução de parênteses.

2.5 Semântica do Cálculo Proposicional

Como visto durante a construção de tabelas-verdade — seção 2.4 — o significado de uma fórmula do Cálculo Proposicional é dado pela interpretação dessa fórmula, ou seja, pela atribuição apropriada de valores-verdade — **v**, **f** — a cada um de seus componentes. Assim, a *semântica* do Cálculo Proposicional consiste na interpretação de suas fórmulas.

Se β for uma fórmula, uma interpretação de β consiste na atribuição de valores-verdade (**v** ou **f**) às fórmulas atômicas componentes de β, levando-se em consideração a interpretação dos conectivos $\wedge, \vee, \neg, \rightarrow, \leftrightarrow$

Seja a fórmula: $(p \vee q) \rightarrow (p \wedge q)$

Como esta fórmula possui dois componentes atômicos, ela admite 2^2 interpretações, onde o expoente é o número de componentes atômicos da fórmula. Para uma fórmula com n componentes atômicos, a tabela-verdade terá 2^n linhas, uma linha para cada interpretação.

	p	q	$p \vee q$	$p \wedge q$	$(p \vee q) \rightarrow (p \wedge q)$
interpretação 1	**v**	**v**	**v**	**v**	**v**
interpretação 2	**v**	**f**	**v**	**f**	**f**
interpretação 3	**f**	**v**	**v**	**f**	**f**
interpretação 4	**f**	**f**	**f**	**f**	**v**

2.6 Validade e Inconsistência

O valor-verdade — ou simplesmente valor — de uma fórmula diz sempre respeito a uma particular interpretação. Com isto pode-se ter as seguintes situações:

1. Se uma fórmula β tem valor **v** numa certa interpretação I, diz-se que β é *verdadeira na interpretação I.*
   No exemplo anterior, $(p \vee q) \rightarrow (p \wedge q)$ é verdadeira nas interpretações 1 e 4.

2. Se uma fórmula β é verdadeira segundo alguma interpretação, diz-se que β é *satisfatível (ou consistente).*
   Por exemplo, a fórmula do exemplo anterior é satisfatível.

3. Uma fórmula β é *válida* quando for verdadeira em todas as suas interpretações. São as chamadas *tautologias.*
   Por exemplo, a fórmula $p \vee \neg p$.

4. Se uma fórmula β tem valor **f** numa interpretação I, diz-se que β é *falsa na interpretação I.*
   A fórmula $(p \vee q) \rightarrow (p \wedge q)$ é falsa nas interpretações 2 e 3.

5. Uma fórmula β é *insatisfatível (ou inconsistente)* quando for falsa segundo qualquer interpretação. São também chamadas de *contradições.*
   Por exemplo, a fórmula $p \wedge \neg p$.

6. Uma fórmula β é *inválida* quando for falsa segundo alguma interpretação.
   Por exemplo, a fórmula p $\vee$ q.

7. No Cálculo Proposicional, as fórmulas que não são nem tautologias e nem contradições são comumente chamadas de *contingentes*.
   Por exemplo, a fórmula p.

As seguintes observações podem então ser constatadas:

- Uma fórmula é inconsistente se e somente se sua negação for válida;
- Uma fórmula é inválida se e somente se existe pelo menos uma interpretação na qual ela é falsa;
- Uma fórmula é consistente se e somente se existe pelo menos uma interpretação em que ela é verdadeira;
- Se uma fórmula é válida, então ela é consistente, mas não vice-versa;
- Se uma fórmula é inconsistente, então ela é inválida, mas não vice-versa.

Pode ser facilmente verificado através do uso de tabelas-verdade que:

$(p \wedge \neg p)$	é inconsistente e portanto inválida
$(p \vee \neg p)$	é válida e portanto consistente
$(p \rightarrow \neg p)$	é inválida, ainda que consistente

2.7 Consequência Lógica

Dadas as fórmulas $\beta_1, \beta_2, \ldots, \beta_n$ e uma fórmula α, diz-se que α é *consequência lógica* de $\beta_1, \beta_2, \ldots, \beta_n$ se e somente se para qualquer interpretação em que $\beta_1, \beta_2, \ldots, \beta_n$ forem simultaneamente verdadeiras, α é também verdadeira.

Se α é consequência lógica de $\beta_1, \beta_2, \ldots, \beta_n$, diz-se que α *segue-se logicamente de* $\beta_1, \beta_2, \ldots, \beta_n$. Para indicar este fato, usa-se a seguinte notação:

$$\beta_1, \beta_2, \ldots, \beta_n \models \alpha$$

Os dois teoremas seguintes permitem demonstrar quando uma fórmula α é consequência lógica de uma outra fórmula:

Teorema 2.7.1 *Dadas as fórmulas $\beta_1, \beta_2, \ldots, \beta_n$ e uma fórmula α, α é consequência lógica de $\beta_1, \beta_2, \ldots, \beta_n$ se e somente se a fórmula*

$$\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_n \rightarrow \alpha \quad \textit{é uma tautologia}$$

Prova:
Condição Necessária:
Sejam as fórmulas $\beta_1, \beta_2, \ldots, \beta_n$ e α, e seja α consequência lógica de $\beta_1, \beta_2, \ldots, \beta_n$. Seja I uma interpretação qualquer.

1. se $\beta_1, \beta_2, \ldots, \beta_n$ forem verdade em I, então α também será verdade em I, pois é consequência lógica dos β_i's.
   Portanto, $\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_n \rightarrow \alpha$ é verdade em I.

2. Se um dos β_i's for falso em I, $\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_n$ será também falso em I. Independente do valor de α, $\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_n \rightarrow \alpha$ é verdade em I.

De 1) e 2) tem-se que $\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_n \rightarrow \alpha$ é verdade em qualquer interpretação, ou seja, $\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_n \rightarrow \alpha$ é uma tautologia.

Condição Suficiente:
Do fato de $\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_n \rightarrow \alpha$ ser uma tautologia, tem-se que $\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_n \rightarrow \alpha$ é verdade em qualquer interpretação. Se $\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_n$ for verdade em I, α é também verdade em I, ou seja, α é consequência lógica de $\beta_1, \beta_2, \ldots, \beta_n$.

Teorema 2.7.2 *Dadas as fórmulas $\beta_1, \beta_2, \ldots, \beta_n$ e uma fórmula α, α é consequência lógica de $\beta_1, \beta_2, \ldots, \beta_n$ se e somente se a fórmula*

$$\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_n \wedge \neg\alpha \quad \textit{é uma contradição}$$

Prova:
Sabe-se pelo teorema anterior que:

Das as fórmulas $\beta_1, \beta_2, \ldots, \beta_n$ e α, α é consequência lógica de $\beta_1, \beta_2, \ldots, \beta_n$ se e somente se $\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_n \rightarrow \alpha$ for válida.

Equivalentemente, α é consequência lógica de $\beta_1, \beta_2, \ldots, \beta_n$ se e somente se a negação de $\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_n \rightarrow \alpha$ for uma contradição. Mas

$$\begin{aligned} &\neg(\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_n \rightarrow \alpha) \equiv \\ &\neg(\neg(\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_n) \vee \alpha) \equiv \\ &\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_n \wedge \neg\alpha \end{aligned}$$

ou seja, $\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_n \wedge \neg\alpha$ é uma contradição.

Estes metateoremas são muito importantes. Eles mostram que provar que uma fórmula é consequência lógica de um conjunto finito de fórmulas é equivalente a mostrar que uma fórmula relacionada a ela é uma tautologia ou uma contradição.

Para algumas estratégias de provas usa-se o Teorema 2.7.1, chamado de *Teorema da Dedução* ou de *Admissão de Premissas*. Outra estratégia de prova, conhecida como *Redução ao absurdo* é estabelecida pelo Teorema 2.7.2.

2.8 Equivalência Lógica

Diz-se que uma fórmula α é *logicamente equivalente* ($\equiv$) a uma fórmula β quando α for consequência lógica de β e β for consequência lógica de α.

$$\alpha \equiv \beta \text{ se e somente se } \alpha \leftrightarrow \beta \text{ é uma tautologia}$$

Em outras palavras, diz-se que duas fórmulas α e β são em equivalentes (ou α é equivalente a β) se e somente se os valores verdade de α e β são os mesmos para qualquer interpretação de α e β.

Por exemplo, pode-se verificar que $(p \rightarrow q)$ é equivalente a $(\neg p \vee q)$ examinando a tabela-verdade dessas fórmulas e constatando que $(p \rightarrow q) \leftrightarrow (\neg p \vee q)$ é uma tautologia.

p	q	$p \rightarrow q$	$\neg p \vee q$	$(p \rightarrow q) \leftrightarrow (\neg p \vee q)$
v	v	v	v	v
v	f	f	f	v
f	v	v	v	v
f	f	v	v	v

Argumentos

Um *argumento* é uma sequência $\alpha_1, \alpha_2, \ldots, \alpha_n$ $(n \geq 1)$ de proposições, onde as proposições α_i $(1 \leq i \leq n-1)$ chamam-se *premissas* e α_n, *conclusão*

Indica-se o argumento por:

$$\alpha_1, \alpha_2, \ldots, \alpha_{n-1} \vdash \alpha_n$$

Um argumento $\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_{n-1} \vdash \alpha_n$ é um *argumento válido* se e somente se a fórmula

$$\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_{n-1} \rightarrow \alpha_n \quad \text{for uma tautologia}$$

ou seja, $\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_{n-1} \models \alpha_n$. Esta afirmação é justificada pelo Teorema 2.7.1.

Um argumento válido pode ser lido como:

$\alpha_1, \alpha_2, \ldots, \alpha_{n-1}$ acarretam α_n ou
α_n decorre de $\alpha_1, \alpha_2, \ldots, \alpha_{n-1}$ ou
α_n é consequência lógica de $\alpha_1, \alpha_2, \ldots, \alpha_{n-1}$

Para $n = 1$, considera-se por extensão o argumento válido se e somente se α_1 for tautológica.

Eventualmente, a verificação da validade de um argumento através de tabelas-verdade pode ser um trabalho longo, uma vez que depende do número de átomos nele existentes.

Por exemplo, num argumento com 7 átomos, a tabela-verdade terá 128 linhas. Se um argumento tiver n átomos sua tabela-verdade terá 2^n linhas.

Uma outra maneira de mostrar a validade de argumentos é através do uso de argumentos válidos já conhecidos e também de equivalências. O princíipio conhecido como *princípio de substituição* viabiliza este procedimento.

Princípio de Substituição

Dada uma fórmula α, diz-se que β é uma subfórmula de α, se β é por sua vez uma fórmula que é parte de α.

Por exemplo, dada a fórmula

$$\alpha\text{: } (p \rightarrow q) \leftrightarrow r$$

$p \rightarrow q$, p, q, r, são algumas subfórmulas de α.

O *princípio de substituição* diz que: uma subfórmula de uma fórmula α, ou toda a fórmula α, pode ser substituída por uma fórmula equivalente, e a fórmula resultante γ é equivalente à α.

Por exemplo, pelo princípio de substituição, a fórmula

$$\alpha\text{: } (p \vee q) \wedge (p \vee r)$$

é equivalente à fórmula

$$\gamma\text{: } (\neg p \rightarrow q) \wedge (\neg p \rightarrow r)$$

isto porque as subfórmulas de α — $(p \vee q)$ e $(p \vee r)$ — são equivalentes, respectivamente, às fórmulas $(\neg p \rightarrow q)$ e $(\neg p \rightarrow r)$ de γ.

2.9 Propriedades da Negação, Conjunção e Disjunção

Estas propriedades podem ser verificadas como equivalências lógicas. Para demonstrar cada uma delas, basta utilizar as tabelas-verdade, constatando a tautologia. Sejam α, β e γ fórmulas.

a. Propriedades da Conjunção

comutativa	$\alpha \wedge \beta$	$\equiv$	$\beta \wedge \alpha$
associativa	$\alpha \wedge (\beta \wedge \gamma)$	$\equiv$	$(\alpha \wedge \beta) \wedge \gamma$
idempotente	$\alpha \wedge \alpha$	$\equiv$	α
propriedade de **v**(erdade)	$\alpha \wedge \mathbf{v}$	$\equiv$	α
propriedade de **f**(also)	$\alpha \wedge \mathbf{f}$	$\equiv$	**f**

b. Propriedades da Disjunção

comutativa	$\alpha \vee \beta$	$\equiv$	$\beta \vee \alpha$
associativa	$\alpha \vee (\beta \vee \gamma)$	$\equiv$	$(\alpha \vee \beta) \vee \gamma$
idempotente	$\alpha \vee \alpha$	$\equiv$	α
propriedade de **v**(erdade)	$\alpha \vee \mathbf{v}$	$\equiv$	$\mathbf{v}$
propriedade de **f**(also)	$\alpha \vee \mathbf{f}$	$\equiv$	α

c. Propriedades Distributivas

$$\alpha \wedge (\beta \vee \gamma) \equiv (\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$$
$$\alpha \vee (\beta \wedge \gamma) \equiv (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$$

d. Absorção

$$\alpha \vee (\alpha \wedge \beta) \equiv \alpha$$
$$\alpha \wedge (\alpha \vee \beta) \equiv \alpha$$

e. Negação

$$\neg(\neg\alpha) \equiv \alpha$$

f. De Morgan

$$\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$$
$$\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$$

g. Equivalência da Implicação

$$\alpha \rightarrow \beta \equiv \neg\alpha \vee \beta$$

2.10 Fórmulas Proposicionais Equivalentes

As principais fórmulas proposicionais equivalentes são dadas pelas propriedades de operações: comutativa, associativa, idempotente, De Morgan, distributiva, negação e absorção vistas na seção 2.9.

É apresentado, a seguir, um conjunto de esquemas de argumentos válidos, facilmente verificáveis, considerados clássicos e largamente conhecidos como *regras de inferência*. Neles α, β, e γ representam fórmulas.

Nome da Regra	Regra
modus ponens	$\alpha, \alpha \rightarrow \beta \models \beta$
modus tollens	$\alpha \rightarrow \beta, \neg\beta \models \neg\alpha$
silogismo hipotético ou regra da cadeia	$\alpha \rightarrow \beta, \beta \rightarrow \gamma \models \alpha \rightarrow \gamma$
silogismo disjuntivo	$\alpha \vee \beta, \neg\alpha \models \beta$
dilema construtivo	$\alpha \rightarrow \beta, \gamma \rightarrow \delta, \alpha \vee \gamma \models \beta \vee \delta$
dilema destrutivo	$\alpha \rightarrow \beta, \gamma \rightarrow \delta, \neg\beta \vee \neg\delta \models \neg\alpha \vee \neg\gamma$
simplificação	$\alpha \wedge \beta \models \alpha$
conjunção	$\alpha, \beta \models \alpha \wedge \beta$
adição	$\alpha \models \alpha \vee \beta$
contraposição	$\alpha \rightarrow \beta \models \neg\beta \rightarrow \neg\alpha$
exportação	$\alpha \rightarrow (\beta \rightarrow \gamma) \models (\alpha \wedge \beta) \rightarrow \gamma$
importação	$(\alpha \wedge \beta) \rightarrow \gamma \models \alpha \rightarrow (\beta \rightarrow \gamma)$

A forma de leitura da modus ponens, por exemplo é:

caso	α	seja verdade
e	$\alpha \rightarrow \beta$	seja verdade
obrigatoriamente	β	terá que ser verdade

Todas as outras regras são lidas de maneira análoga.

2.11 Verificação de Validade de Argumentos

Sejam $\alpha_1, \alpha_2, \ldots, \alpha_n$, β fórmulas do Cálculo Proposicional. Diz-se que uma sequência finita de fórmulas (ou proposições) $C_1, C_2, \ldots, C_k$, é uma *prova* ou *dedução* de β, a partir das premissas $\alpha_1, \alpha_2, \ldots, \alpha_n$ se e somente se:

1. cada C_i é uma premissa $\alpha_j (1 \geq j \geq n)$, ou
2. C_i provém das fórmulas precedentes, pelo uso de um argumento válido da tabela na seção 2.10, ou
3. C_i provém do uso do princípio de substituição numa fórmula anterior, ou
4. C_k é β.

Diz-se então que β é dedutível de $\alpha_1, \alpha_2, \ldots, \alpha_n$ ou que β é um *teorema*. Facilmente se verifica que se isto acontecer, o argumento é válido.

Seja provar a validade de:

$$(p \wedge q) \vee (p \rightarrow q), \neg(p \wedge q) \models p \rightarrow q$$

Constrói-se a sequência

C_1: $(p \wedge q) \vee (p \rightarrow q)$	premissa
C_2: $\neg(p \wedge q)$	premissa
C_3: $p \rightarrow q$	silogismo disjuntivo de C_1 e C_2

Tem-se então que C_1, C_2, C_3 é uma demonstração de $p \rightarrow q$ a partir de

$$(p \wedge q) \vee (p \rightarrow q), \neg(p \wedge q)$$

e, também, que $p \rightarrow q$ é dedutível de

$$(p \wedge q) \vee (p \rightarrow q), \neg(p \wedge q).$$

Pode-se então escrever que

$$(p \wedge q) \vee (p \rightarrow q), \neg(p \wedge q) \models p \rightarrow q$$

2.12 Exemplos

A seguir são mostrados alguns exemplos de verificação de validade de argumentos usando argumentos válidos já conhecidos e equivalências.

Exemplo 2.12.1 *Seja:*

Se as uvas caem, então a raposa as come.
Se a raposa as come, então estão maduras.
As uvas estão verdes ou caem.
logo,
A raposa come as uvas se e só se as uvas caem.

nomeando:	p: as uvas caem	
	q: a raposa come as uvas	
	r: as uvas estão maduras	
tem-se:	C_1: $p \rightarrow q$	
	C_2: $q \rightarrow r$	
	C_3: $\neg r \vee p$	
deduz-se:	C_4: $r \rightarrow p$	(C_3: equivalência)
	C_5: $q \rightarrow p$	(C_2 + C_4 + silogismo hipotético)
	C_6: $p \rightarrow q \wedge q \rightarrow p$	(C_1 + C_5: conjunção)
	C_7: $p \leftrightarrow q$	(C_6: equivalência)

Exemplo 2.12.2 *Seja:*

Carlos estuda ou não está cansado.
Se Carlos estuda, então dorme tarde.
Carlos não dorme tarde ou está cansado.
logo,
Carlos está cansado se e só se estuda.

nomeando:	p: Carlos estuda	
	q: Carlos está cansado	
	r: Carlos dorme tarde	
tem-se:	C_1: $p \vee \neg q$	
	C_2: $p \rightarrow r$	
	C_3: $\neg r \vee q$	
deduz-se:	C_4: $q \rightarrow p$	(C_1: equivalência)
	C_5: $r \rightarrow q$	(C_3: equivalência)
	C_6: $p \rightarrow q$	($C_2 + C_5$ + silogismo hipotético)
	C_7: $q \rightarrow p \wedge p \rightarrow q$	($C_4 + C_6$: conjunção)
	C_8: $p \leftrightarrow q$	(C_7:equivalência)

Exemplo 2.12.3 *Provar a validade de:* $p \wedge \neg p \vdash r$, *ou seja, provar que* $p \wedge \neg p \models r$

tem-se:	C_1: $p \wedge \neg p$	
deduz-se:	C_2: p	(C_1: simplificação)
	C_3: $p \vee r$	(C_2: adição)
	C_4: $\neg p$	(C_1: simplificação)
	C_5: r	($C_3 + C_4$ + silogismo disjuntivo)

Exemplo 2.12.4 *Mostrar a validade de* r *a partir de* $p \rightarrow q$, $p \vee r$, $\neg q$

tem-se:	C_1: $p \rightarrow q$	
	C_2: $p \vee r$	
	C_3: $\neg q$	
deduz-se:	C_4: $\neg p \vee q$	(C_1: equivalência)
	C_5: $\neg p$	($C_3 + C_4$ + silogismo disjuntivo)
	C_6: r	($C_2 + C_5$ + silogismo disjuntivo)

Exemplo 2.12.5 *Mostrar a validade de* $\neg p \rightarrow q$, $q \rightarrow r$,$\neg r \vee s$, $\neg s \vdash p$

tem-se:	C_1: $\neg p \rightarrow q$	
	C_2: $q \rightarrow r$	
	C_3: $\neg r \vee s$	
	C_4: $\neg s$	
deduz-se:	C_5: $\neg r$	($C_3 + C_4$ + silogismo disjuntivo)
	C_6: $\neg q \vee r$	(C_2: equivalência)
	C_7: $\neg q$	($C_5 + C_6$ + silogismo disjuntivo)
	C_8: $p \vee q$	(C_1: equivalência lógica)
	C_9: p	($C_7 + C_8$ + silogismo disjuntivo)

Exemplo 2.12.6 *Provar* $p \rightarrow q$, $r \rightarrow s$, $q \vee s \rightarrow t$, $\neg t \models \neg p \wedge \neg r$

tem-se:	C_1: p $\rightarrow$ q	
	C_2: r $\rightarrow$ s	
	C_3: q $\vee$ s $\rightarrow$ t	
	C_4: $\neg$t	
deduz-se:	C_5: $\neg$(q $\vee$ s) $\vee$ t	(C_3: equivalência)
	C_6: $\neg$(q $\vee$ s)	(C_4 + C_5 + silogismo disjuntivo)
	C_7: $\neg$q $\wedge$ $\neg$s	(C_6: De Morgan)
	C_8: $\neg$s	(C_7: simplificação)
	C_9: $\neg$q	(C_7: simplificação)
	C_{10}: $\neg$r $\vee$ s	(C_2: equivalência)
	C_{11}: $\neg$r	(C_8 + C_{10} + silogismo disjuntivo)
	C_{12}: $\neg$p $\vee$ q	(C_1: equivalência)
	C_{13}: $\neg$p	(C_9 + C_{12} + silogismo disjuntivo)
	C_{14}: $\neg$p $\wedge$ $\neg$r	(C_{11} + C_{13} + conjunção)
ou ainda:	C_5: (p $\vee$ r) $\rightarrow$ (q $\vee$ s)	(C_1 + C_2 + dilema construtivo)
	C_6: p $\vee$ r $\rightarrow$ t	(C_3 + C_5 + silogismo hipotético)
	C_7: $\neg$(p $\vee$ r) $\vee$ t	(C_6: equivalência)
	C_8: $\neg$(p $\vee$ r)	(C_4 + C_8 + silogismo disjuntivo)

3 Método Sintático de Prova de Teoremas

Lógica e Prova Automática de Teoremas são de significativa importância na área de Inteligência Artificial — a primeira por ser uma linguagem na qual se pode expressar problemas e, a segunda, por ser uma possível forma de obter suas soluções.

O Cálculo Proposicional embora aplicável a um número restrito de problemas, fornece uma ferramenta simples com a qual os conceitos básicos de prova automática de teoremas podem ser ilustrados.

Nesta seção é apresentado um método sintático para a prova automática de teoremas do CP, proposto originalmente por H.Wang [Wang 60],[Wang 64].

Duas implementações desenvolvidas na linguagem de programação lógica Prolog são apresentadas e discutidas, abordando as estruturas de dados utilizadas bem como a eficiência de execução, procurando evidenciar os aspectos positivos e negativos de cada uma delas.

3.1 Considerações Preliminares

Sejam p_1,p_2,...,p_n e c proposições para as quais:

$$p_1 \wedge p_2 \wedge ,\ldots, \wedge p_n \rightarrow c$$

é uma tautologia. Pode-se então dizer que a conjunção dos p_i logicamente implica c. Como visto na seção 2.7, as proposições $p_1,p_2,\ldots,p_n$ são chamadas de *premissas* e a proposição c de *conclusão* do *silogismo* **se** $p_1,p_2,\ldots,$ p_n **então c**. Neste caso, **se** p_1, $p_2,\ldots,p_n$ **então c** é um *teorema* e será escrito como [3]:

$$p_1,p_2,\ldots,p_n \Rightarrow c$$

No caso da conjunção dos p_i não implicar logicamente em c, não se tem um teorema e este fato é notado por:

$$p_1,p_2,\ldots,p_n \not\Rightarrow c$$

Como já foi visto, uma forma de determinar se um silogismo é ou não um teorema é através de sua tabela-verdade. O silogismo será um teorema se o valor-verdade da implicação for v. A verificação do valor-verdade de um silogismo através de sua tabela-verdade é conhecido como *método semântico de prova de teorema.*

Devido à maneira como a implicação $\rightarrow$ é definida, é necesssário apenas verificar as instâncias nas quais todas as premissas $p_1,p_2,\ldots,p_n$ são verdade. Se a conclusão c for verdade nessas instâncias, a implicação será uma tautologia. A técnica de verificar apenas as instâncias nas quais as premissas têm valor verdade é chamada de *método forward chaining.*

Pode-se também verificar o valor-verdade de um silogismo, se for possível garantir que para cada instância em que a conclusão for falsa, pelo menos uma das premissas é também falsa. Se isto acontecer, a implicação é uma tautologia. Este método de verificação de um silogismo é chamado de *método backward chaining.*

Para a verificação do valor-verdade de um silogismo utilizando-se o método *forward chaining*, parte-se das premissas e, através do exame de seus valores-verdade, tenta-se chegar à conclusão. Quando se utiliza o método *backward chaining* com o mesmo propósito, inicia-se com a conclusão e, a partir desta, investiga-se os valores-verdade das premissas.

3.2 Prova Sintática de Teoremas

Os métodos sintáticos de prova utilizam uma sequência lógica de tautologias, bem como teoremas provados previamente, para demonstrar que uma conclusão é uma consequência lógica de um dado conjunto de premissas. Uma demonstração deste tipo é chamada de *derivação.*

Na derivação, para provar sintaticamente o teorema:

$$p_1,p_2,\ldots,p_n \Rightarrow c$$

[3] O símbolo $\Rightarrow$ é equivalente ao símbolo $\models$ utilizado na seção 2.7

pode-se substituir qualquer das sentenças p_i, incluindo a conclusão c, por outra sentença que lhe seja logicamente equivalente; pode-se introduzir nas premissas qualquer sentença que seja logicamente implicada por qualquer coleção de sentenças participantes das premissas (até aquele presente estágio de derivação), e pode-se substituir a conclusão por qualquer sentença que logicamente a implique.

A derivação estará completa e o teorema estará provado quando (e se) ambos os lados de $\Rightarrow$ consistem de exatamente a mesma expressão. A existência de uma ocorrência deste tipo significa que a verdade de todas as premissas implica na verdade da conclusão, o que estabelece o teorema.

Na prova sintática de equivalência lógica, utiliza-se o fato de premissas e conclusão serem substituídas apenas por sentenças que lhes sejam logicamente equivalentes. Este fato garante a equivalência lógica entre o teorema original e a expressão a que se chega após substituições.

Deve ser notado que se em algum passo do processo de derivação for obtido um teorema conhecido, a derivação estará completa, uma vez que a presença de tal teorema garante a implicação lógica. Em tal passo as premissas implicam a conclusão e, portanto, podem ser substituídas por ela, o que resulta na mesma expressão aparecendo em ambos os lados do símbolo $\Rightarrow$.

Os métodos sintáticos de prova de teoremas são mais diretos que os semânticos e usam apenas a estrutura do silogismo. Há vários algoritmos para prova automática de teoremas no Cálculo Proposicional por métodos sintáticos. Um deles é o Algoritmo de Wang, apresentado a seguir.

3.3 Algoritmo de Wang

O Algoritmo de Wang [Wang 60],[Wang 64] se aplica a expressões da forma:

$$p_1,p_2,p_3,\ldots,p_n \Rightarrow c_1,c_2,c_3,\ldots,c_n$$

onde os p_i constituem as premissas e os c_i, constituem a conclusão:

$$c_1 \vee c_2 \vee c_3 \ldots \vee c_n$$

Em termos de valor-verdade — abordagem semântica — tem-se um teorema se o valor-verdade de:

$$p_1 \wedge p_2 \wedge \ldots \wedge p_n \rightarrow c_1 \vee c_2 \vee \ldots \vee c_n$$

for verdade. Embora no método automático de prova de teorema de Wang os valores-verdade não sejam utilizados, uma vez que se trata de uma abordagem sintática, uma compreensão semântica se faz necessária para um completo entendimento do algoritmo.

Para começar a provar um teorema usando o algoritmo de Wang, todas as premissas são escritas à esquerda do símbolo $\Rightarrow$ e a conclusão a que se deseja chegar é escrita à direita.

Por exemplo, sejam as seguintes premissas verdadeiras:

$$p \rightarrow q$$
$$q \rightarrow r$$
$$\neg r$$

e que se queira, a partir delas, saber se é verdade $\neg p$. Isto deve ser expresso como:

$$p \rightarrow q, q \rightarrow r, \neg r \Rightarrow \neg p$$

Transformações são agora aplicadas ao teorema, de forma a quebrá-lo em outros mais simples que, por sua vez, serão submetidos ao mesmo processo.

O algoritmo de Wang sempre termina em um número finito de passos, provando

$$p \Rightarrow q \text{ ou } p \not\Rightarrow q$$

O Algoritmo de Wang consiste de regras de transformação — procedimentos — que são aplicadas recursivamente e de condições de parada.

A seguir, são descritas as 6 regras de transformação, R_1 a R_6 e as regras de término R_7 e R_8, que compõem o algoritmo.

3.3.1 Regras de Transformação

O objetivo dos procedimentos recursivos — regras de transformação — é o de remover conectivos de maneira que as regras de término possam ser aplicadas.

R_1 se uma das fórmulas tem a forma $\neg X$, pode-se tirar a negação e move-la para o outro lado do símbolo $\Rightarrow$. X pode ser qualquer fórmula. Por exemplo:

$$p \vee q, \neg(r \wedge s), p \vee r \Rightarrow s, q$$

torna-se

$$p \vee q, p \vee r \Rightarrow s, q, r \wedge s$$

$$p, q, r \Rightarrow \neg s, t$$

torna-se

$$p, q, r, s \Rightarrow t$$

R_2 se uma fórmula à esquerda de $\Rightarrow$ tem a forma de X $\wedge$ Y, ou se uma à direita tem a forma de X $\vee$ Y, o conectivo pode ser substituído por uma vírgula. O $\wedge$ a ser removido deve ser o conectivo principal da sentença à esquerda de $\Rightarrow$. Observação análoga vale para o $\vee$ com relação à sentença à direita de $\Rightarrow$. Por exemplo:

$$p \wedge q, r \wedge(\neg p \vee s) \Rightarrow \neg p \wedge \neg r$$

torna-se

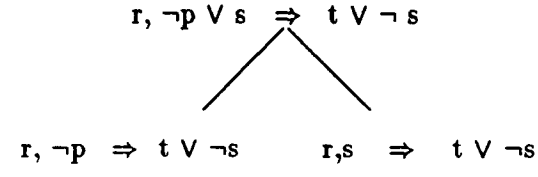
$$p,q,r,\neg p \vee s \Rightarrow \neg p \wedge \neg r$$

$$r \Rightarrow \neg s \vee \neg t$$

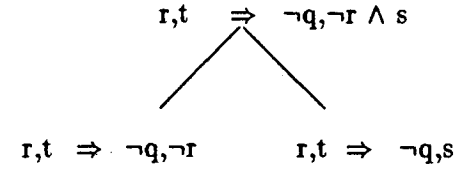
torna-se

$$r \Rightarrow \neg s,\neg t$$

R_3 se uma fórmula à esquerda de $\Rightarrow$ tem a forma X $\vee$ Y, pode-se remover o operador $\vee$ e separar os seus dois argumentos, de forma a dividir o teorema original em dois novos teoremas. Cada um dos dois teoremas obtidos pela separação deve ser provado separadamente. O $\vee$ deve ser o principal conectivo à esquerda. Por exemplo:



R_4 se uma fórmula com a forma X $\wedge$ Y ocorre à direita de $\Rightarrow$, pode-se remover o operador $\wedge$ e separar os seus dois argumentos, de forma a dividir o teorema original em dois novos teoremas. Cada um dos dois teoremas obtidos pela separação deve ser provado separadamente. O $\wedge$ deve ser o principal conectivo à direita. Por exemplo:



É importante notar que a aplicação das regras R_3 ou R_4 provoca um crescimento exponencial do número de teoremas a serem provados. Em geral, se existirem um total de k $\wedge$ e $\vee$ como principais conectivos à esquerda e direita de $\Rightarrow$ respectivamente, um total de 2^k novos teoremas resultam como consequência da aplicação daquelas regras.

R_5 uma fórmula, em qualquer nível, que tenha a forma $(X \rightarrow Y)$, pode ser substituída pela fórmula equivalente $(\neg X \vee Y)$, eliminando-se com isso a implicação.

R_6 uma fórmula, em qualquer nível, que tenha a forma $(X \leftrightarrow Y)$, pode ser substituída pela fórmula equivalente $(X \rightarrow Y) \wedge (Y \rightarrow X)$, eliminando-se com isso a dupla implicação.

3.3.2 Regras de Parada

As regras anteriores devem ser aplicadas tantas vezes quantas forem necessárias, para a remoção dos conectivos $\leftrightarrow$, $\rightarrow$, $\neg$, $\wedge$ e $\vee$, até que:

R_7 um teorema é considerado provado, se alguma fórmula X ocorre em ambos os lados de $\Rightarrow$. Tal teorema é chamado de *axioma*. Nenhuma transformação será mais necessária neste teorema, muito embora possam existir outros a serem provados. O teorema original não estará provado até que todos os teoremas obtidos a partir dele tenham sido provados. Portanto, esta regra deve ser verificada para todo novo teorema que eventualmente resulte da aplicação das regras de transformação.

R_8 um teorema é provado inválido se todas as fórmulas que nele comparecem são símbolos de proposições individuais — isto é, não existem mais conectivos — e um mesmo símbolo não ocorre em ambos os lados de $\Rightarrow$. Se um teorema como este é encontrado, o algorítmo termina; a conclusão inicial não é uma consequência lógica das premissas

A seguir são apresentados os padrões das transformações propostas por Wang:

R_1	$(\ldots,\neg X,\ldots \Rightarrow \ldots, Z)$	torna-se	$(\ldots \Rightarrow \ldots, Z,X)$
R_1	$(\ldots,Z \Rightarrow \ldots,\neg X,\ldots)$	torna-se	$(\ldots, Z, X \Rightarrow \ldots)$
R_2	$(\ldots, X \wedge Y,\ldots \Rightarrow \ldots)$	torna-se	$(\ldots, X, Y, \ldots \Rightarrow \ldots)$
R_2	$(\ldots \Rightarrow \ldots,X \vee Y,\ldots)$	torna-se	$(\ldots \Rightarrow \ldots,X,Y,\ldots)$
R_3	$(\ldots, X \vee Y, \ldots \Rightarrow \ldots)$	torna-se	$(\ldots,X,\ldots \Rightarrow \ldots)$
		e	$(\ldots,Y,\ldots \Rightarrow \ldots)$
R_4	$(\ldots \Rightarrow \ldots, X \wedge Y,\ldots)$	torna-se	$(\ldots \Rightarrow \ldots, X, \ldots)$
		e	$(\ldots \Rightarrow \ldots, Y, \ldots)$
R_5	$(\ldots \Rightarrow \ldots, X \rightarrow Y, \ldots)$	torna-se	$(\ldots, \ldots \Rightarrow \ldots, \neg X \vee Y, \ldots)$
R_5	$(\ldots, X \rightarrow Y, \ldots \Rightarrow \ldots)$	torna-se	$(\ldots,\neg X \vee Y,\ldots \Rightarrow \ldots, \ldots)$
R_6	$(\ldots, X \leftrightarrow Y, \ldots \Rightarrow \ldots)$	torna-se	$(\ldots,(X \rightarrow Y \wedge Y \rightarrow X), \ldots) \Rightarrow \ldots)$
R_6	$(\ldots \Rightarrow \ldots, X \leftrightarrow Y, \ldots)$	torna-se	$(\ldots \Rightarrow \ldots, (X \rightarrow Y \wedge Y \rightarrow X), \ldots)$
R_7	$(\ldots,X,\ldots \Rightarrow \ldots,X,\ldots)$	torna-se	true

O algoritmo de Wang sempre converge para a solução de um dado problema. Toda aplicação de uma transformação conduz a algum progresso no sentido de eliminar um

conectivo e assim diminuir sintaticamente o tamanho do teorema — mesmo que com isso sejam criados outros teoremas, como é o caso da aplicação das regras R_3 e R_4.

A seguir, o algoritmo de Wang é aplicado na busca da conclusão para o exemplo da seção 3.3. As linhas foram rotuladas de maneira a faciliar referências a elas.

Rótulo		Comentários
S1:	$p \rightarrow q, q \rightarrow r, \neg r \Rightarrow \neg p$	teorema
S2:	$\neg p \vee q, \neg q \vee r, \neg r \Rightarrow \neg p$	duas aplicacões de R_5
S3:	$\neg p \vee q, \neg q \vee r \Rightarrow \neg p, r$	R_1
S4:	$\neg p, \neg q \vee r \Rightarrow \neg p, r$	S4 e S5 obtidos de S3 aplicando R_3.S4 é axioma
S5:	$q, \neg q \vee r \Rightarrow \neg p, r$	gerado de S3 com R_3
S6:	$q, \neg q \Rightarrow \neg p, r$	S6 e S7 gerados de S5 via R_3
S7:	$q, r \Rightarrow \neg p, r$	axioma
S8:	$q \Rightarrow \neg p, r, q$	obtido de S6 usando R_1.Axioma

O teorema original está provado uma vez que foi transformado em um conjunto de 3 axiomas e todos os teoremas intermediários foram também provados.

3.4 Implementações do Algoritmo de Wang

A seguir são mostradas duas implementações diferentes do algoritmo de Wang, na linguagem de programação lógica Prolog.

A primeira implementação é de autoria de L. Pereira e encontra-se em [Coelho 88]; a segunda implementação, mais eficiente, é de nossa autoria e se encontra descrita em detalhes em [Monard 90].

Em ambas implementações, os conectivos lógicos são definidos em Prolog pelos seguintes operadores:

```
:- op(690,xfy,=>).          % teorema
:- op(670,xfy,<->).         % equivalencia
:- op(650,xfy,->).          % implicacao
:- op(600,xfy,v).           % disjuncao
:- op(550,xfy,&).           % conjuncao
:- op(500,fy,n).            % negacao
```

Os dois programas aceitam dois possíveis formatos de entrada:

1. uma fórmula bem formada do CP, como por exemplo

```
p <-> q -> (p & q) v (n p & n q).

p & ( p -> q) -> q.
```

2. duas fórmulas bem formadas f_1 e f_2 do CP, onde se queira provar que

$$f_1 \Rightarrow f_2$$

Por exemplo

```
p <-> q => (p & q) v (n p & n q).

p & ( p -> q) => q.
```

Deve ser observado que qualquer dos formatos pode ser utilizado para entrar o mesmo teorema.

Todos os programas são ativados pelo predicado `teorema/2`, onde o primeiro argumento é o teorema a ser provado e o segundo argumento é uma variável, cujo valor será `v` se o primeiro argumento for um teorema e `f` caso contrário.

Por exemplo:

```
?- teorema( p <-> q => (p & q) v (n p & n q), Val).
```

será bem sucedido com `Val` unificado com `v`, enquanto que

```
?- teorema( p & q -> p v r <-> p v p, Val).
```

será bem sucedido com `Val` unificado com `f`.

3.4.1 Implementação de L. Pereira

Nesta implementação é utilizado um operador adicional : usado para delimitar as listas que fazem parte da estrutura utilizada. Ele é definido por:

```
:- op(550,xfy,:).
```

A idéia geral usada na versão de Pereira é a de utilizar uma estrutura da forma:

$$\underset{1}{[\,]} : \underset{2}{[\,]} \quad => \quad \underset{3}{[\,]} : \underset{4}{[\,]}$$

Se a entrada T do usuário for fornecida sem o símbolo `=>` , a prova é iniciada na estrutura:

$$[\,] : [\,] \quad => \quad [\,] : [T]$$

enquanto que se a entrada for do tipo

$$L \quad => \quad R$$

a prova é iniciada na estrutura:

$$[\,] : [L] \Rightarrow [\,] : [R]$$

A partir daí, a implementação tenta aplicar as regras R_5, R_6 e R_1, nesta ordem, a fim de eliminar os conectivos -> , <-> , e n (que representa a negação). A seguir, tenta aplicar a regra R_2, que substitui nas listas L e R respectivamente, os conectivos $\wedge$ e $\vee$ por vírgulas, a fim de transformar uma fórmula do tipo:

$$p_1 \wedge p_2 \Rightarrow c_1 \vee c_2$$

para a notação:

$$p_1,p_2 \Rightarrow c_1,c_2$$

É importante notar que até este ponto as listas 1 e 3 não foram utilizadas.

Quando não for possível a aplicação das regras R_1,R_2, R_5 e/ou R_6, o programa tenta aplicar as regras R_3 ou R_4 que dividem o teorema que está sendo provado em dois outros teoremas, os quais por sua vez devem ser provados independentemente.

Quando não for possível aplicar nenhuma das regras anteriores, é porque a cabeça das listas 2 ou 4 é um átomo e, como tal, ele é transferido para a lista 1 ou 3 respectivamente. A função das listas 1 e 3 é a de acumular os átomos contidos no teorema original, de maneira a permitir a verificação de uma tautologia quando se obtém a estrutura

$$Lf : [\,] \Rightarrow Rf : [\,]$$

verificando simplesmente se um átomo da lista Lf está também presente na lista Rf, caso contrário o teorema não é válido.

Listagem do Programa

```
teorema( L => R,v) :-
    prove(□:[L] => □:[R]),
    !.

teorema( L => R,f) :-
    !.

teorema(T,v) :-
    prove(□:□=>□:[T]),
    !.

teorema(T,f).


prove(E1) :-
    regra(E1,E2,Reg),
```

```
    !,
    prove(E2).

% caso de v no lado esquerdo
prove(L : [H v I|T] => R) :-
    !,
    Esq = L : [H|T] => R,
    Dir = L : [I|T] => R,
    prove(Esq),
    prove(Dir).

% caso de & no lado direito
prove(L => R : [H & I|T]) :-
    !,
    Esq = L => R : [H|T],
    Dir = L => R : [I|T],
    prove(Esq),
    prove(Dir).

% caso de atomo
prove(L : [H|T] => R) :-
    !,
    prove([H|L] : T => R).

prove(L => R : [H|T]) :-
    !,
    prove(L => [H|R] : T).

% verifica se e tautologia
prove(T) :-
    tautologia(T).


% casos onde -> aparece em um dos lados
regra(L : [H -> I|T] => R, L : [n H v I|T] => R, regra_5).
regra(L => R : [H -> I|T], L => R : [n H v I|T], regra_5).

% caso onde <-> aparece em um dos lados
regra(L : [H <-> I|T] => R, L : [(H -> I) & (I -> H)|T] => R, regra_6).
regra(L => R : [H <-> I|T], L => R : [(H -> I) & (I -> H)|T], regra_6).

% casos onde n (negacao) aparece em um dos lados
regra(L : [n H|T] => R : R2, L : T => R : [H|R2],regra_1).
regra(L1 : L2 => R : [n H|T], L1 : [H|L2] => R : T ,regra_1).

% caso do conectivo & do lado esquerdo de =>
regra(L : [H & I|T] => R, L : [H,I|T] => R,regra_2).

% caso do conectivo v do lado direito de =>
regra(L => R : [H v I | T], L => R : [H,I|T],regra_2).
```

```
tautologia(L : [] => R : []) :-
    pertence(M,L),
    pertence(M,R),
    !.


pertence(H,[H|_]).
pertence(I,[_|T]) :-
    pertence(I,T).
```

A fim de exemplificar a execução deste programa, serão acrescentados a ele predicados de leitura e escrita. Os novos predicados, bem como aqueles que sofreram algumas modicações devido a esse acréscimo, são listados a seguir.

```
pereira :-
    info_intro('Pereira'),
    repeat,
    entrada(Formula),
    (Formula = fim ;
     teorema(Formula,Valor), info_valor(Valor), fail).


prove(E1) :-
    regra(E1,E2,Reg),
    !,
    info_regra(E2,Reg),
    prove(E2).

% caso de v no lado esquerdo
prove(L : [H v I|T] => R) :- !,
    Esq = L : [H|T] => R,
    Dir = L : [I|T] => R,
    info_regra(Esq,Dir,regra_3),
    info_provando(Esq),
    prove(Esq),
    info_provando(Dir),
    prove(Dir).

% caso de & no lado direito
prove(L => R : [H & I|T]) :- !,
    Esq = L => R : [H|T],
    Dir = L => R : [I|T],
    info_regra(Esq,Dir,regra_4),
    info_provando(Esq),
    prove(Esq),
    info_provando(Dir),
    prove(Dir).

% caso de atomo
prove(L : [H|T] => R) :- !,prove([H|L] : T => R).
prove(L => R : [H|T]) :- !,prove(L => [H|R] : T).
```

```
% verifica se e tautologia
prove(T) :-
    tautologia(T),
    !,
    info_tauto(T).

prove(_) :-
     info_falha,
     fail.


% Predicados auxiliares de leitura e escrita que serao usados
% para mostrar a execucao das tres implementacoes do algoritmo
info_intro(P):-
    nl,tab(4),
    write('Provador de Teoremas do Calculo Proposicional'),
    nl,tab(4),
    write('Algoritmo de Wang - Implementacao de '),
    write(P),
    nl,nl.


entrada(F) :-
    nl,tab(4),
    write('Entre com a formula (fim para terminar):'),
    nl,nl,read(F).


info_valor(v):-
    nl,tab(4),
    write('** A formula e um teorema **'),
    nl.

info_valor(f) :-
    nl,nl,tab(4),
    write('** A formula nao e um teorema **'),
    nl.


info_regra(E,Reg) :-
    nl,write(Reg),tab(3),
    write(E).

info_regra(E,D,Reg) :-
    nl,nl,
    write('Para provar formula anterior, pela regra '),
    write(Reg),
    write(' devemos provar:'),nl,nl,
    tab(15),write(E),nl,
    tab(30),write(e),nl,
    tab(15),write(D),nl,nl.
```

```
info_provando(E) :-
    write('Provando  '),
    write(E).


info_tauto(T) :-
    nl,tab(10),
    write(T),
    write('    Ramo provado'),nl,nl.


info_falha :-
    write('    Ramo nao pode ser provado').
```

Exemplos de Execução [4]

```
?-pereira.

    Provador de Teoremas do Calculo Proposicional
    Algoritmo de Wang - Implementacao de Pereira

    Entre com a formula (fim para terminar):

p & q -> r v q & p => p -> q -> r v q & p.

regra_5   [] : [n (p & q) v r v q & p] => [] : [p -> q -> r v q & p]
regra_5   [] : [n (p & q) v r v q & p] => [] : [n p v (q -> r v q & p)]
regra_2   [] : [n (p & q) v r v q & p] => [] : [n p,q -> r v q & p]
regra_1   [] : [p,n (p & q) v r v q & p] => [] : [q -> r v q & p]
regra_5   [] : [p,n (p & q) v r v q & p] => [] : [n q v r v q & p]
regra_2   [] : [p,n (p & q) v r v q & p] => [] : [n q,r v q & p]
regra_1   [] : [q,p,n (p & q) v r v q & p] => [] : [r v q & p]
regra_2   [] : [q,p,n (p & q) v r v q & p] => [] : [r,q & p]

Para provar formula anterior, pela regra regra_3 devemos provar:

              [p,q] : [n (p & q)] => [] : [r,q & p]
                                e
              [p,q] : [r v q & p] => [] : [r,q & p]

Provando  [p,q] : [n (p & q)] => [] : [r,q & p]
regra_1   [p,q] : [] => [] : [p & q,r,q & p]

Para provar formula anterior, pela regra regra_4 devemos provar:

              [p,q] : [] => [] : [p,r,q & p]
```

[4] As saídas das execuções de programas Prolog foram, quando necessário, editadas, objetivando uma melhor visualização.

e

[p,q] : [] => [] : [q,r,q & p]

Provando [p,q] : [] => [] : [p,r,q & p]

Para provar formula anterior, pela regra regra_4 devemos provar:

[p,q] : [] => [r,p] : [q]
e
[p,q] : [] => [r,p] : [p]

Provando [p,q] : [] => [r,p] : [q]
[p,q] : [] => [q,r,p] : [] Ramo provado

Provando [p,q] : [] => [r,p] : [p]
[p,q] : [] => [p,r,p] : [] Ramo provado

Provando [p,q] : [] => [] : [q,r,q & p]

Para provar formula anterior, pela regra regra_4 devemos provar:

[p,q] : [] => [r,q] : [q]
e
[p,q] : [] => [r,q] : [p]

Provando [p,q] : [] => [r,q] : [q]
[p,q] : [] => [q,r,q] : [] Ramo provado

Provando [p,q] : [] => [r,q] : [p]
[p,q] : [] => [p,r,q] : [] Ramo provado

Provando [p,q] : [r v q & p] => [] : [r,q & p]

Para provar formula anterior, pela regra regra_3 devemos provar:

[p,q] : [r] => [] : [r,q & p]
e
[p,q] : [q & p] => [] : [r,q & p]

Provando [p,q] : [r] => [] : [r,q & p]

Para provar formula anterior, pela regra regra_4 devemos provar:

[r,p,q] : [] => [r] : [q]
e
[r,p,q] : [] => [r] : [p]

Provando [r,p,q] : [] => [r] : [q]
[r,p,q] : [] => [q,r] : [] Ramo provado

Provando [r,p,q] : [] => [r] : [p]
[r,p,q] : [] => [p,r] : [] Ramo provado

```
Provando  [p,q] : [q & p] => [] : [r,q & p]
regra_2   [p,q] : [q,p] => [] : [r,q & p]

Para provar formula anterior, pela regra regra_4 devemos provar:

                [p,q,p,q] : [] => [r] : [q]
                                  e
                [p,q,p,q] : [] => [r] : [p]

Provando  [p,q,p,q] : [] => [r] : [q]
          [p,q,p,q] : [] => [q,r] : []    Ramo provado

Provando  [p,q,p,q] : [] => [r] : [p]
          [p,q,p,q] : [] => [p,r] : []    Ramo provado

    ** A formula e um teorema **


Entre com a formula (fim para terminar):

n p <-> n q <-> n r => p & r.
regra_6   [] : [(n p -> (n q <-> n r)) & ((n q <-> n r) -> n p)] =>
                                                        [] : [p & r]
regra_2   [] : [n p -> (n q <-> n r),(n q <-> n r) -> n p] =>
                                                        [] : [p & r]
regra_5   [] : [n n p v (n q <-> n r),(n q <-> n r) -> n p] =>
                                                        [] : [p & r]

Para provar formula anterior, pela regra regra_3 devemos provar:

                [] : [n n p,(n q <-> n r) -> n p] => [] : [p & r]
                                  e
                [] : [n q <-> n r,(n q <-> n r) -> n p] => [] : [p & r]

Provando  [] : [n n p,(n q <-> n r) -> n p] => [] : [p & r]
regra_1   [] : [(n q <-> n r) -> n p] => [] : [n p,p & r]
regra_5   [] : [n (n q <-> n r) v n p] => [] : [n p,p & r]
regra_1   [] : [p,n (n q <-> n r) v n p] => [] : [p & r]

Para provar formula anterior, pela regra regra_4 devemos provar:

                [] : [p,n (n q <-> n r) v n p] => [] : [p]
                                  e
                [] : [p,n (n q <-> n r) v n p] => [] : [r]

Provando  [] : [p,n (n q <-> n r) v n p] => [] : [p]

Para provar formula anterior, pela regra regra_3 devemos provar:

                [p] : [n (n q <-> n r)] => [] : [p]
                                  e
```

```
                [p] : [n p] => [] : [p]

Provando  [p] : [n (n q <-> n r)] => [] : [p]
regra_1   [p] : [] => [] : [n q <-> n r,p]
regra_6   [p] : [] => [] : [(n q -> n r) & (n r -> n q),p]
```

Para provar formula anterior, pela regra regra_4 devemos provar:

```
                [p] : [] => [] : [n q -> n r,p]
                                e
                [p] : [] => [] : [n r -> n q,p]

Provando  [p] : [] => [] : [n q -> n r,p]
regra_5   [p] : [] => [] : [n n q v n r,p]
regra_2   [p] : [] => [] : [n n q,n r,p]
regra_1   [p] : [n q] => [] : [n r,p]
regra_1   [p] : [] => [] : [q,n r,p]
regra_1   [p] : [r] => [q] : [p]
          [r,p] : [] => [p,q] : []     Ramo provado

Provando  [p] : [] => [] : [n r -> n q,p]
regra_5   [p] : [] => [] : [n n r v n q,p]
regra_2   [p] : [] => [] : [n n r,n q,p]
regra_1   [p] : [n r] => [] : [n q,p]
regra_1   [p] : [] => [] : [r,n q,p]
regra_1   [p] : [q] => [r] : [p]
          [q,p] : [] => [p,r] : []     Ramo provado

Provando  [p] : [n p] => [] : [p]
regra_1   [p] : [] => [] : [p,p]
          [p] : [] => [p,p] : []     Ramo provado

Provando  [] : [p,n (n q <-> n r) v n p] => [] : [r]
```

Para provar formula anterior, pela regra regra_3 devemos provar:

```
                [p] : [n (n q <-> n r)] => [] : [r]
                                e
                [p] : [n p] => [] : [r]

Provando  [p] : [n (n q <-> n r)] => [] : [r]
regra_1   [p] : [] => [] : [n q <-> n r,r]
regra_6   [p] : [] => [] : [(n q -> n r) & (n r -> n q),r]
```

Para provar formula anterior, pela regra regra_4 devemos provar:

```
                [p] : [] => [] : [n q -> n r,r]
                                e
                [p] : [] => [] : [n r -> n q,r]

Provando  [p] : [] => [] : [n q -> n r,r]
regra_5   [p] : [] => [] : [n n q v n r,r]
```

```
regra_2   [p] : [] => [] : [n n q,n r,r]
regra_1   [p] : [n q] => [] : [n r,r]
regra_1   [p] : [] => [] : [q,n r,r]
regra_1   [p] : [r] => [q] : [r]
          [r,p] : [] => [r,q] : []    Ramo provado

Provando  [p] : [] => [] : [n r -> n q,r]
regra_5   [p] : [] => [] : [n n r v n q,r]
regra_2   [p] : [] => [] : [n n r,n q,r]
regra_1   [p] : [n r] => [] : [n q,r]
regra_1   [p] : [] => [] : [r,n q,r]
regra_1   [p] : [q] => [r] : [r]    Ramo nao pode ser provado

    ** A formula nao e um teorema **

Entre com a formula (fim para terminar):

fim.
```

3.4.2 Implementação de MCM-MCN

Na implementação descrita a seguir foi usada a mesma estrutura de dados sugerida por Pereira. Entretanto, ela realiza a transformação da fórmula de entrada, logo no início, de maneira a simplificar negações bem como remover implicações `->` e equivalências `<->`.

Este passo inicial melhora o tempo de execução pois diminui o número de movimentos de uma mesma fórmula de um lado para outro do símbolo `=>`, a fim de simplificar as negações.

Uma outra melhora acrescentada refere-se à verificação de tautologia. Deve ser observado que Pereira somente verifica pela existência de uma tautologia quando a estrutura é da forma

$$\underset{1}{[L]} \ : \ \underset{2}{[\,]} \ \Rightarrow \ \underset{3}{[R]} \ : \ \underset{4}{[\,]}$$

onde L e R são listas contendo apenas átomos.

Contudo, não há necessidade das listas 2 e 4 estarem vazias para realizar esta verificação. Esta última abordagem foi aqui utilizada e, em parte, é devido a ela a melhora no tempo de execução da implementação proposta. Um estudo comparativo dos tempos de execução de ambas implementações pode ser encontrado em [Monard 90].

Listagem do Programa

```
teorema( L => R,v) :-
     transforme(L,L1),
     transforme(R,R1),
```

```
     prove([]:[L1] => []:[R1]),
     !.

teorema( L => R,f) :- !.

teorema(T,v) :-
    transforme(T,T1),
    prove([]:[]=>[]:[T1]),
    !.

teorema(T,f).


prove(E1) :-
    regra(E1,E2,Reg),
    !,
    prove(E2).

% caso de v no lado esquerdo
prove(L : [H v I|T] => R) :- !,
    Esq = L : [H|T] => R,
    Dir = L : [I|T] => R,
    prove(Esq),
    prove(Dir).

% caso de & no lado direito
prove(L => R : [H & I|T]) :- !,
    Esq = L => R : [H|T],
    Dir = L => R : [I|T],
    prove(Esq),
    prove(Dir).

% caso de atomo
prove(L : [H|T] => R1 : R2) :-
    pertence(H,R1),
    !.

prove(L1 : L2 => R : [H|T]) :-
    pertence(H,L1),
    !.

prove(L : [H|T] => R) :-
    !,
    prove([H|L] : T => R).

prove(L => R : [H|T]) :-
    !,
    prove(L => [H|R] : T).


transforme(n n P,P1) :-
    !,
```

```
    transforme(P,P1).

transforme((P <-> Q), ((P1 & Q1) v (n P1 & n Q1))) :-
    !,
    transforme(P,P1),
    transforme(Q,Q1).

transforme((P -> Q), (n P1 v Q1)) :-
    !,
    transforme(P,P1),
    transforme(Q,Q1).

transforme((P & Q), (P1 & Q1)) :-
    !,
    transforme(P,P1),
    transforme(Q,Q1).

transforme((P v Q), (P1 v Q1)) :-
    !,
    transforme(P,P1),
    transforme(Q,Q1).

transforme((n P), (n P1)):-
    !,
    transforme(P,P1).

transforme(P,P).


% casos onde n (negacao) aparece
regra(L : [n H|T] => R : R2,
     L : T => R : [H|R2],regra_1).

regra(L1 : L2 => R : [n H|T],
     L1 : [H|L2] => R : T ,regra_1).

% caso do & do lado esquerdo
regra(L : [H & I|T] => R,
     L : [H,I|T] => R,regra_2).

% caso do v do lado direito
regra(L => R : [H v I | T],
     L => R : [H,I|T],regra_2).


pertence(H,[H|_]) :- !.

pertence(I,[_|T]) :-
    pertence(I,T).
```

Analogamente à seção 3.4.1, serão acrescentados ao programa anterior os mesmos predicados de leitura e escrita, de maneira que a sua execução possa ser exibida. Os pre-

dicados que sofreram alguma modificação devido a tal acréscimo são listados a seguir.

```
mcm_mcn:-
    info_intro('MCM_MCN'),
    repeat,
    entrada(Formula),
    (Formula = fim ;
     teorema(Formula,Valor), info_valor(Valor), fail).


teorema( L => R,v) :-
     transforme(L,L1),
     transforme(R,R1),
     info_transf(L1 => R1),
     prove([]:[L1] => []:[R1]),
     !.

teorema( L => R,f) :- !.

teorema(T,v) :-
    transforme(T,T1),
    info_transf(T1),
    prove([]:[]=>[]:[T1]),
    !.

teorema(T,f).


prove(E1) :-
    regra(E1,E2,Reg),
    !,
    info_regra(E2,Reg),
    prove(E2).

% caso de v no lado esquerdo
prove(L : [H v I|T] => R) :- !,
    Esq = L : [H|T] => R,
    Dir = L : [I|T] => R,
    info_regra(Esq,Dir,regra_3),
    info_provando(Esq),
    prove(Esq),
    info_provando(Dir),
    prove(Dir).

% caso de & no lado direito
prove(L => R : [H & I|T]) :- !,
    Esq = L => R : [H|T],
    Dir = L => R : [I|T],
    info_regra(Esq,Dir,regra_4),
    info_provando(Esq),
    prove(Esq),
    info_provando(Dir),
```

```
    prove(Dir).

% caso de atomo
prove(L : [H|T] => R1 : R2) :-
    pertence(H,R1),
    !,
    info_tauto(L : [H|T] => R1 : R2).

prove(L1 : L2 => R : [H|T]) :-
    pertence(H,L1),
    !,
    info_tauto(L1 : L2 => R : [H|T]).

prove(L : [H|T] => R) :-
    !,
    prove([H|L] : T => R).

prove(L => R : [H|T]) :-
    !,
    prove(L => [H|R] : T).

prove(_) :-
    info_falha,
    fail.

% predicado adicional de escrita
info_transf(X) :-
    nl,
    write('provando a formula equivalente'),
    nl,tab(10),
    write(X),nl.
```

Exemplos de Execução A seguir, é mostrada a execução de `mcm_mcn/0` usando os mesmos dados de entrada anteriores.

```
?- mcm_mcn.

    Provador de Teoremas do Calculo Proposicional
    Algoritmo de Wang - Implementacao de MCM_MCN


    Entre com a formula (fim para terminar):

p & q -> r v q & p => p -> q -> r v q & p.

provando a formula equivalente
         n (p & q) v r v q & p => n p v n q v r v q & p

regra_2   [] : [n (p & q) v r v q & p] => [] : [n p,n q v r v q & p]
regra_1   [] : [p,n (p & q) v r v q & p] => [] : [n q v r v q & p]
regra_2   [] : [p,n (p & q) v r v q & p] => [] : [n q,r v q & p]
```

```
regra_1   [] : [q,p,n (p & q) v r v q & p] => [] : [r v q & p]
regra_2   [] : [q,p,n (p & q) v r v q & p] => [] : [r,q & p]

Para provar formula anterior, pela regra regra_3 devemos provar:

                [p,q] : [n (p & q)] => [] : [r,q & p]
                                    e
                [p,q] : [r v q & p] => [] : [r,q & p]

Provando  [p,q] : [n (p & q)] => [] : [r,q & p]
regra_1   [p,q] : [] => [] : [p & q,r,q & p]

Para provar formula anterior, pela regra regra_4 devemos provar:

                [p,q] : [] => [] : [p,r,q & p]
                                    e
                [p,q] : [] => [] : [q,r,q & p]

Provando  [p,q] : [] => [] : [p,r,q & p]
          [p,q] : [] => [] : [p,r,q & p]    Ramo provado

Provando  [p,q] : [] => [] : [q,r,q & p]
          [p,q] : [] => [] : [q,r,q & p]    Ramo provado

Provando  [p,q] : [r v q & p] => [] : [r,q & p]

Para provar formula anterior, pela regra regra_3 devemos provar:

                [p,q] : [r] => [] : [r,q & p]
                                    e
                [p,q] : [q & p] => [] : [r,q & p]

Provando  [p,q] : [r] => [] : [r,q & p]
          [r,p,q] : [] => [] : [r,q & p]    Ramo provado

Provando  [p,q] : [q & p] => [] : [r,q & p]
regra_2   [p,q] : [q,p] => [] : [r,q & p]

Para provar formula anterior, pela regra regra_4 devemos provar:

                [p,q,p,q] : [] => [r] : [q]
                                    e
                [p,q,p,q] : [] => [r] : [p]

Provando  [p,q,p,q] : [] => [r] : [q]
          [p,q,p,q] : [] => [r] : [q]    Ramo provado

Provando  [p,q,p,q] : [] => [r] : [p]
          [p,q,p,q] : [] => [r] : [p]    Ramo provado


    ** A formula e um teorema **
```

```
Entre com a formula (fim para terminar):

n p <-> n q <-> n r => p & r.

provando a formula equivalente
          n p & (n q & n r v n n q & n n r) v n n p &
                              n (n q & n r v n n q & n n r) => p & r


Para provar formula anterior, pela regra regra_3 devemos provar:

                [] : [n p & (n q & n r v n n q & n n r)] => [] : [p & r]
                                        e
                [] : [n n p & n (n q & n r v n n q & n n r)] => [] : [p & r]

Provando  [] : [n p & (n q & n r v n n q & n n r)] => [] : [p & r]
regra_2   [] : [n p,n q & n r v n n q & n n r] => [] : [p & r]
regra_1   [] : [n q & n r v n n q & n n r] => [] : [p,p & r]

Para provar formula anterior, pela regra regra_3 devemos provar:

                [] : [n q & n r] => [] : [p,p & r]
                                        e
                [] : [n n q & n n r] => [] : [p,p & r]

Provando  [] : [n q & n r] => [] : [p,p & r]
regra_2   [] : [n q,n r] => [] : [p,p & r]
regra_1   [] : [n r] => [] : [q,p,p & r]
regra_1   [] : [] => [] : [r,q,p,p & r]

Para provar formula anterior, pela regra regra_4 devemos provar:

                [] : [] => [p,q,r] : [p]
                                        e
                [] : [] => [p,q,r] : [r]

Provando  [] : [] => [p,q,r] : [p]    Ramo nao pode ser provado

    ** A formula nao e um teorema **


Entre com a formula (fim para terminar):

fim.
```

4 Forma Normal

Sejam as seguintes fórmulas equivalentes:

$$(p \rightarrow q) \quad \wedge \quad m$$
$$(\neg p \vee q) \quad \wedge \quad m$$

É possível observar que há várias maneiras de escrever uma mesma fórmula. Entretanto, é conveniente ter uma certa uniformidade na notação a fim de poder expressar as fórmulas de uma maneira única.

Duas formas, chamadas de Formas Normais -FN-, são particularmente utilizadas. Elas são conhecidas como Forma Normal Conjuntiva e Forma Normal Disjuntiva.

É importante observar que dado um enunciado do Cálculo Proposicional, é sempre possível determinar um enunciado equivalente àquele, que esteja em uma das duas formas normais.

Como ficará mais claro em seções posteriores, frequentemente é necessário transformar a expressão de uma fórmula para a forma normal.

4.1 Forma Normal Conjuntiva - FNC

Diz-se que uma fórmula proposicional α está na FNC quando α é uma conjunção $\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_n$, $(n \geq 1)$, em que cada β_i $(1 \leq i \leq n)$ é uma disjunção de literais, ou um literal.

Pode-se então dizer que uma fórmula α está na FNC se e somente se:

1. contém como conectivos apenas $\wedge$ e $\vee$ e o operador $\neg$;
2. $\neg$ só opera sobre proposições atômicas, isto é, não tem alcance sobre $\wedge$ e $\vee$;
3. não aparecem sinais de negação sucessivos como $\neg\neg$;
4. $\vee$ não tem alcance sobre $\wedge$, ou seja, não existem expressões do tipo $p \vee (q \wedge r)$.

Se β é uma fórmula na Forma Normal Conjuntiva equivalente a α, então β é referida como FNC(α).

Exemplo 4.1.1 *Seja a fórmula* α: $\neg p \vee q \rightarrow r$

$$\text{FNC}(\alpha): (\neg p \vee \neg q \vee r) \wedge (p \vee \neg q \vee r) \wedge (p \vee q \vee r)$$

É fácil mostrar que uma FNC é tautológica se e somente se cada elemento da conjunção é tautológico.

4.2 Obtenção da FNC de uma Fórmula Não Tautológica usando Tabela-Verdade

Para a obtenção da FNC de uma fórmula não-tautológica α, procura-se na tabela-verdade de α as interpretações $I_1, I_2, \ldots, I_k$ que tornam esta fórmula f. Para cada uma dessas interpretações I_i $(1 \leq i \leq k)$ constrói-se a disjunção da seguinte maneira:

se o valor do componente atômico p de α com respeito à interpretação I_i for **v**, toma-se $\neg$p, e se for **f**, toma-se p; em seguida determina-se a conjunção das disjunções obtidas em cada uma das interpretações I_i.

Exemplo 4.2.1 *Seja a fórmula:* $\neg p \vee q \rightarrow r$

	p	q	r	$\neg$p $\vee$ q $\rightarrow$ r	
I_1	v	v	v	v	
I_2	v	v	f	f	$\leftarrow$
I_3	v	f	v	v	
I_4	v	f	f	v	
I_5	f	v	v	v	
I_6	f	v	f	f	$\leftarrow$
I_7	f	f	v	v	
I_8	f	f	f	f	$\leftarrow$

então,

$$\text{FNC}(\neg p \vee q \rightarrow r)\text{: } (\neg\, p \vee \neg q \vee r) \wedge (p \vee \neg q \vee r) \wedge (p \vee q \vee r)$$

Se a fórmula α for uma tautologia, determina-se que

$$\text{FNC}(\alpha)\text{: } p \vee \neg p$$

onde p é uma fórmula atômica

4.3 Forma Normal Disjuntiva – FND

Diz-se que uma fórmula proposicional α está na FND quando α é uma disjunção $\beta_1 \vee \ldots \vee \beta_n$, $n \geq 1$, onde cada β_i $(1 \leq i \leq n)$ é uma conjunção de literais, ou um literal.

Pode-se dizer então que uma fórmula α está na FND se e somente se:

1. contém como conectivos apenas $\wedge$ e $\vee$ e o operador $\neg$;
2. $\neg$ só opera sobre proposições atômicas, isto é, não tem alcance sobre $\wedge$ e $\vee$;
3. não aparecem sinais de negação sucessivos, como $\neg\neg$;
4. $\wedge$ não tem alcance sobre $\vee$, ou seja, não existem expressões do tipo p $\wedge$ (q $\vee$ r).

Se β é uma fórmula na forma normal disjuntiva equivalente a α, então β é referida como FND(α).

Exemplo 4.3.1 *As fórmulas:*

$$(p \wedge q) \vee (r \wedge q) \quad \text{e} \quad (\neg p \wedge q) \vee r$$

estão na FND.

4.4 Obteção da FND de uma Fórmula Não Contraditória usando Tabela-Verdade

Para a obtenção da FND de uma fórmula não contraditória α, procura-se na tabela-verdade de α as interpretações $I_1, I_2, \ldots, I_k$ que tornam esta fórmula verdadeira. Para cada uma dessas interpretações I_i $(1 \leq i \leq k)$, constrói-se a conjunção da seguinte maneira:

se o valor do componente atômico p de α com respeito a I_i for **v**, toma-se p e se for **f**, toma-se $\neg$p; em seguida, determina-se a disjunção das conjunções obtidas em cada uma das interpretações de I_i.

Exemplo 4.4.1 *Seja α uma fórmula cuja tabela-verdade é a seguinte:*

	p	q	r	α	
I_1	v	v	v	v	$\leftarrow$
I_2	v	v	f	v	$\leftarrow$
I_3	v	f	v	f	
I_4	v	f	f	f	
I_5	f	v	v	f	
I_6	f	v	f	f	
I_7	f	f	v	f	
I_8	f	f	f	v	$\leftarrow$

então

$$\text{FND}(\alpha)\text{: } (p \wedge q \wedge r) \vee (p \wedge q \wedge \neg r) \vee (\neg p \wedge \neg q \wedge \neg r)$$

Se a fórmula α for uma contradição, determina-se que

$$\text{FND}(\alpha)\text{:} p \wedge \neg p$$

onde p é uma fórmula atômica.

4.5 Obtenção da FND e FNC sem o Uso de Tabela-Verdade

A obtenção da forma normal de uma dada fórmula pode ser conseguida através da substituição de fórmulas existentes na fórmula dada, por fórmulas equivalentes. Este processo é repetido até que a fórmula normal desejada seja obtida.

O procedimento de transformação deve seguir os seguintes passos:

1. repetidamente usar as equivalências

$$\begin{aligned} \alpha \leftrightarrow \beta &\equiv (\neg\alpha \vee \beta) \wedge (\neg\beta \vee \alpha) \\ \alpha \rightarrow \beta &\equiv \neg\alpha \vee \beta \end{aligned}$$

   para eliminar os conectivos lógicos $\leftrightarrow$ e $\rightarrow$.

2. repetidamente utilizar:

$$\neg(\neg\alpha) = \alpha$$

   para eliminação das negativas múltiplas, e

$$\begin{aligned} \neg(\alpha \vee \beta) &= (\neg\alpha \wedge \neg\beta) \\ \neg(\alpha \wedge \beta) &= (\neg\alpha \vee \neg\beta) \end{aligned}$$

   para eliminação do operador $\neg$ que precede o parêntese e posicionar o sinal de negação imediatamente antes dos átomos.

3. repetidamente, se necessário, aplicar leis distributivas:

$$\begin{aligned} \alpha \vee (\beta \wedge \gamma) &= (\alpha \vee \beta) \wedge (\alpha \vee \gamma) \\ (\alpha \wedge \beta) \vee \gamma &= (\alpha \vee \gamma) \wedge (\beta \vee \gamma) \end{aligned}$$

   para a obtenção da FNC, e

$$\begin{aligned} \alpha \wedge (\beta \vee \gamma) &= (\alpha \wedge \beta) \vee (\alpha \wedge \gamma) \\ (\alpha \vee \beta) \wedge \gamma &= (\alpha \wedge \gamma) \vee (\beta \wedge \gamma) \end{aligned}$$

   para obtenção da FND

É importante lembrar que uma Forma Normal é expressa em função de literais. Portanto, na FN as fórmulas α, β e γ são, necessariamente, literais.

Exemplo 4.5.1 *Obter a FNC da fórmula de* $((p \vee q) \wedge (\neg p \vee r)) \rightarrow s$

1. eliminar $\rightarrow$ e $\leftrightarrow$
   $\neg((p \vee q) \wedge (\neg p \vee r)) \vee s$

2. usar repetidamente $\neg(\neg\beta) = \beta$, leis De Morgan e distributivas

De Morgan	$(\neg(p \vee q) \vee \neg(\neg p \vee r)) \vee s$
De Morgan	$((\neg p \wedge \neg q) \vee (p \wedge \neg r)) \vee s$
Distributiva	$(((\neg p \wedge \neg q) \vee p) \wedge ((\neg p \wedge \neg q) \vee \neg r)) \vee s$
Distributiva	$(((\neg p \vee p) \wedge (\neg q \vee p)) \wedge ((\neg p \vee \neg r) \wedge (\neg q \vee \neg r))) \vee s$
Prop. do $\vee$ e $\wedge$	$((\neg q \vee p) \wedge ((\neg p \vee \neg r) \wedge (\neg q \vee \neg r))) \vee s$
Distributiva	$((s \vee \neg q \vee p) \wedge (s \vee \neg p \vee \neg r) \wedge (s \vee \neg q \vee \neg r))$

portanto:

$$\begin{gathered} \text{FNC}(((p \vee q) \wedge (\neg p \vee r)) \rightarrow s): \\ ((s \vee \neg q \vee p) \wedge (s \vee \neg p \vee \neg r) \wedge (s \vee \neg q \vee \neg r)) \end{gathered}$$

4.6 Notação Clausal

A Forma Normal Conjuntiva é de particular interesse no entendimento da linguagem Prolog. Uma *cláusula* é uma disjunção de literais, isto é:

$$F_i = L_1 \vee L_2 \vee \ldots \vee L_r$$

Na FNC, uma fórmula é escrita como uma *conjunção* de cláusulas

$$F_1 \wedge F_2 \wedge \ldots \wedge F_n$$

Uma das vantagens de se ter a FNC de uma dada fórmula é que se o valor da fórmula é **v**, então cada cláusula separadamente é **v**, uma vez que a FNC é uma conjunção de cláusulas. Este fato torna a fórmula mais facilmente manipulável.

Como a FNC de uma fórmula α do Cálculo Proposicional sempre é uma conjunção de cláusulas, a ordem em que estas cláusulas são escritas é irrelevante — pela propriedade associativa do $\wedge$. Assim sendo, pode-se dizer que a FNC é uma coleção de cláusulas. Escreve-se então a FNC de uma fórmula α como

$$\{F_1, F_2, \ldots, F_n\}$$

sendo que a conjunção $\wedge$ entre as cláusulas fica implícita. Nomeia-se de coleção apenas para indicar que a ordem não é importante. Neste sentido, pode-se dizer que qualquer fórmula do Cálculo Proposicional é uma coleção de cláusulas.

Analogamente, uma cláusula F_i terá a forma

$$F_i = \{L_1, L_2, \ldots, L_j\}$$

onde cada L_i $(1 \leq i \leq j)$ é um literal. Aplicando o mesmo raciocínio anterior pode-se dizer que uma cláusula é uma coleção de literais

$$\{L_1, L_2, \ldots, L_j\}$$

onde a disjunção está implícita.

Foi visto na seção anterior que

$$\begin{gathered}\text{FNC}(((\text{p} \vee \text{q}) \wedge (\neg\text{p} \vee \text{r})) \rightarrow \text{s}): \\ ((\text{s} \vee \neg\text{q} \vee \text{p}) \wedge (\text{s} \vee \neg\text{p} \vee \neg\text{r}) \wedge (\text{s} \vee \neg\text{q} \vee \neg\text{r}))\end{gathered}$$

Pode-se pois escrever que:

$$\text{FNC}(((\text{p} \vee \text{q}) \wedge (\neg\text{p} \vee \text{r})) \rightarrow \text{s}): F_1 \wedge F_2 \wedge F_3$$

onde F_1: s $\vee$ $\neg$q $\vee$ p, F_2: s $\vee$ $\neg$p $\vee$ $\neg$r, F_3: s $\vee$ $\neg$q $\vee$ $\neg$r

que pode ser representado pela coleção das três cláusulas, onde a conjunção está implícita, isto é:

$$F = \{F_1, F_2, F_3\}$$

Uma possível convenção é escrever a fórmula F como uma cláusula após a outra, lembrando que elas estão conectadas por $\wedge$.

$$\begin{aligned} F_1&: \quad s \vee \neg q \vee p \\ F_2&: \quad s \vee \neg p \vee \neg r \\ F_3&: \quad s \vee \neg q \vee \neg r \end{aligned}$$

Como cada cláusula é uma coleção de literais que estão conectados por $\vee$, para cada cláusula, pode-se escrever primeiro os literais positivos e logo após os negativos. Isto é:

$$\begin{aligned} F_1&: \quad s \vee p \vee \neg q \\ F_2&: \quad s \vee \neg p \vee \neg r \\ F_3&: \quad s \vee \neg q \vee \neg r \end{aligned}$$

Esta separação de literais positivos e negativos prepara a cláusula para a introdução da notação definida por Kowalsky [Kowalsky 74]. Nesta notação, as cláusulas anteriores são representadas por:

$$\begin{aligned} F_1&: \quad s, p \leftarrow q \\ F_2&: \quad s \leftarrow r, p \\ F_3&: \quad s \leftarrow q, r \end{aligned}$$

onde há uma disjunção implícita nos literais à esquerda de $\leftarrow$, chamados *conclusões*, e uma conjunção implícita nos literais à direita de $\leftarrow$, chamados *condições* ou *premissas*. Deve ser observado que esta notação é equivalente a:

$$\begin{aligned} F_1&: \quad q \rightarrow s \vee p \\ F_2&: \quad r \wedge p \rightarrow s \\ F_3&: \quad q \wedge r \rightarrow s \end{aligned}$$

pois $\quad p \rightarrow q \equiv \neg p \vee q \equiv q \leftarrow p$

Uma cláusula genérica na notação de Kowalsky

$$A_1, A_2, \ldots, A_m \leftarrow B_1, B_2, \ldots, B_n$$

é equivalente a

$$A_1 \vee A_2 \vee \ldots \vee A_m \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n$$

que é equivalente a

$$A_1 \vee A_2 \vee \ldots \vee A_m \vee \neg(B_1 \wedge B_2 \wedge \ldots \wedge B_n)$$

que é equivalente a

$$A_1 \vee A_2 \vee \ldots \vee A_m \vee \neg B_1 \vee \neg B_2 \vee \ldots \vee \neg B_n$$

Dependendo do número de literais envolvidos na cláusula, pode-se ter os seguintes casos especiais:

- $m > 1$ as conclusões são indefinidas, isto é, há várias conclusões
- $m \leq 1$ são as chamadas *cláusulas de Horn*, que têm como casos particulares:
  - $m = 1, n > 0$
    A $\leftarrow B_1, B_2, \ldots, B_n$
    chamada *cláusula definida*, isto é, existe somente uma conclusão
  - $m = 1, n = 0$
    A $\leftarrow$
    chamada *cláusula definida incondicional* ou *fato*. Neste caso o símbolo $\leftarrow$ é abandonado.
  - $m = 0, n > 0$
    $\leftarrow B_1, B_2, \ldots, B_n$
    negação pura de $B_1, B_2, \ldots, B_n$
  - $m = 0, n = 0$
    $\leftarrow$
    chamada *cláusula vazia* e denotada por $\square$

As únicas cláusulas que podem ser representadas usando a linguagem de programação Prolog são as cláusulas de Horn [5]

Numa situação em que um dado conhecimento pode ser representado utilizando cálculo proposicional, apenas as expressões que forem cláusulas de Horn serão passíveis de serem representadas em Prolog.

No que se segue, será adotada a convenção de escrever fórmulas com letras maiúsculas (A, B,...), possivelmente indexadas, pois esta notação é usada mais frequentemente na literatura referente à linguagem de programação lógica Prolog.

4.7 Procedimento de Prova por Resolução

O método de prova por resolução é bem geral. Ele é um método sintático de prova que se baseia no uso de uma simples regra de inferência; isto torna a sua aplicação fácil e vantajosa.

[5] O símbolo $\leftarrow$ é representado por :- na sintaxe do Prolog de Edinburgh.

Resolução pode ser aplicada apenas àquelas wff chamadas *cláusulas* — que são wff consistindo de uma disjunção de literais.

O processo de resolução, quando é aplicável, é aplicado a um par de cláusulas e produz, como resultado, uma cláusula derivada.

A regra de inferência de resolução é:

Resolução

de	$p \vee q$
e	$r \vee \neg q$
deduz-se	$p \vee r$

Esta regra permite combinar duas fórmulas através da eliminação de átomos complementares. Pode também ser vista como a Regra da Cadeia aplicada a uma fórmula na FNC, como visto a seguir:

Regra da Cadeia

de	$\neg p \rightarrow q$	$(\equiv p \vee q)$
e	$q \rightarrow r$	$(\equiv \neg q \vee r)$
deduz-se	$\neg p \rightarrow r$	$(\equiv p \vee r)$

Pode-se verificar que $(p \vee r)$ é uma consequência lógica de $(p \vee q)$ e $(r \vee \neg q)$, o que equivale a mostrar que $(p \vee q) \wedge (r \vee \neg q) \rightarrow (p \vee r)$ é uma tautologia.

4.8 Procedimentos para se usar Resolução

Para usar resolução, usa-se redução ao absurdo. Pode se fazer isso negando a conclusão. Os passos a seguir são:

1. achar para cada premissa e para a conclusão negada (adotada como premissa), a FNC, da seguinte maneira:

   remover $\leftrightarrow$ e $\rightarrow$ através de

   $$q \leftrightarrow p \equiv (q \rightarrow p) \wedge (p \rightarrow q)$$
   $$q \rightarrow p \equiv \neg q \vee p$$

   aplicar De Morgan

   usar distributiva: $q \vee (p \wedge r) \equiv (q \vee p) \wedge (q \vee r)$

2. cada premissa é agora uma conjunção de uma ou mais cláusulas. Escrever cada cláusula em uma linha diferente (cada uma delas é **v**, uma vez que a conjunção de todas é **v**).

3. cada cláusula contém uma disjunção de um ou mais literais; estão na forma correta para se aplicar resolução. Procurar então por duas cláusulas que contenham o mesmo átomo com sinais opostos, por exemplo:

$$q \vee r \vee t \vee \neg p$$
$$r \vee p \vee s$$

No exemplo, **p** é o átomo em questão. Usando resolução, ele é eliminado da cláusula, obtendo-se:

$$q \vee r \vee t \vee s$$

A regra é muito simples de ser aplicada, mesmo porque ela pode ser vista como um simples processo de cancelamento:

$q \vee q$, $\neg q \vee r$ deduz-se $q \vee r$
q, $\neg q \vee r$ deduz-se r

4. continuar neste processo até que se tenha derivado p e $\neg p$. Ao se aplicar resolução nestas duas cláusulas, obtém-se a cláusula vazia denotada por $\square$, o que expressa a contradição, completando então o método de redução ao absurdo.

p, $\neg p$ deduz-se falso

Pode-se também usar resolução através da negação do teorema. Neste caso aplicam-se os mesmos passos anteriores.

4.9 Exemplos do uso de Resolução

A seguir são mostrados dois exemplos de uso de resolução usando prova por redução ao absurdo: o primeiro exemplo através da negação da tese e o segundo através da negação do teorema.

4.9.1 Usando Prova por Redução ao Absurdo através da Negação da Tese

Suponha que se queira provar que

$r \vee s$ é consequência lógica de $p \vee q$, $p \rightarrow r$, $q \rightarrow s$

ou seja, pelo Teorema 2.7.1 deve-se mostrar que

$((p \vee q) \wedge (p \rightarrow r) \wedge (q \rightarrow s)) \rightarrow (r \vee s)$ é uma tautologia (teorema)

Os passos a serem seguidos são:

1. converter as premissas para a FNC e escrevê-las em linhas separadas:

(a) $p \vee q$
(b) $\neg p \vee r$
(c) $\neg q \vee s$

2. negar a conclusão e convertê-la para a FNC:

$$\neg(r \vee s) \equiv \neg r \wedge \neg s$$

$$\begin{array}{ll} (d) & \neg r \\ (e) & \neg s \end{array}$$

3. deduzir a cláusula vazia $\square$ por resolução

$$\begin{array}{llllll} (f) & \neg p & de & (d) & e & (b) \\ (g) & q & de & (f) & e & (a) \\ (h) & \neg q & de & (e) & e & (c) \\ (i) & \square & de & (g) & e & (h) \end{array}$$

A cláusula $\square$ sempre é gerada a partir de duas cláusulas na forma:

$$q \wedge \neg q$$

o que, obviamente, é uma contradição.

4.9.2 Usando Prova por Redução ao Absurdo através da Negação do Teorema

Suponha que se queira provar a regra da cadeia:

$$(\ p \rightarrow q) \wedge (q \rightarrow r) \rightarrow (p \rightarrow r)$$

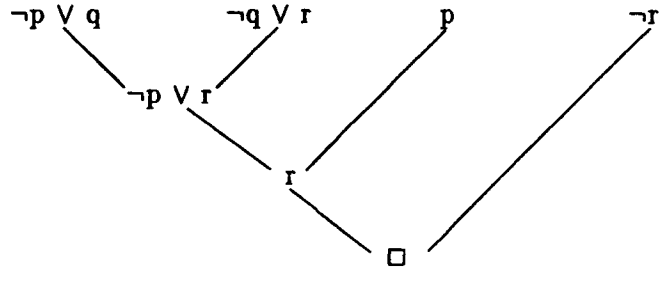
A negação do teorema é:

$$\neg((p \rightarrow q) \wedge (q \rightarrow r) \rightarrow (p \rightarrow r))$$

Da mesma forma que no exemplo anterior, deve-se obter a FNC do teorema negado, que no caso facilmente pode ser verificado que é:

$$(\neg p \vee q) \wedge (\neg q \vee r) \wedge p \wedge \neg r$$

Como já visto, o passo básico do método de resolução ocorre quando existem duas cláusulas tal que uma proposição p ocorre em uma delas e $\neg p$ ocorre na outra. Os passos de resolução são mostrados na figura a seguir:



4.10 Vantagens do Método de Resolução

Comparado com a lógica clássica, o método de resolução tem várias vantagens:

1. Não é necessário o uso de equivalências para rearranjar p ∨ q como q ∨ p, etc. Isso se deve ao fato de que tudo é colocado na FNC antes do método começar a ser aplicado e, particularmente, porque para o método é indiferente a posição, na cláusula, do átomo a ser eliminado;

2. Existe apenas uma regra de inferência para se lembrar;

3. É fácil de ser mecanizado.

Entretanto, em longas provas, é possível que se "ande em círculos"; a uniformidade de notação tende a tornar todas as cláusulas parecidas, o que torna difícil selecionar a cláusula certa e lembrar o que cada uma significa.

A linguagem Prolog está baseada no princípio de resolução aplicado a cláusulas de Horn, utilizando uma estratégia de busca bem definida [6]

O investimento inicial em programação lógica pode ser atribuído a Kowalski e Colmerauer. Foi Kowalski quem formulou a interpretação procedimental da cláusula de Horn. Ele mostrou que um axioma:

$$\text{A se } B_1 \text{ e } B_2 \text{ e } \ldots \text{ e } B_n$$

pode ser lido e executado como um procedimento em uma linguagem recursiva de programação, onde A é a cabeça do procedimento e os B_i's seu corpo.

Além da leitura declarativa da cláusula:

$$\text{A é verdade se os } B_i\text{'s são verdade}$$

ela pode ser lida como:

$$\text{Para resolver(executar) A, resolva (execute) } B_1 \text{ e } B_2 \text{ e } \ldots \text{ e } B_n$$

4.11 Propriedades do Cálculo Proposicional

O Cálculo Proposicional, embora insuficiente para o formalismo do raciocínio lógico, possui propriedades muito importantes que o distinguem. Essas propriedades são: consistência, completude e decidibilidade, definidas a seguir.

1. O sistema é *consistente*: não é possível derivar simultaneamente uma fórmula Q e sua negação ¬Q;

[6] A estratégia utilizada é busca em profundidade.

2. O sistema é *correto* ou *coerente*: todo teorema é uma tautologia;

3. *Completude*: toda tautologia é um teorema;

4. *Decidibilidade*: há um algoritmo que permite verificar se uma dada fórmula do sistema é ou não um teorema.

5 Programas Prolog Relacionados ao Cálculo Proposicional

A seguir são mostradas duas versões de programas Prolog para a determinação do valor-verdade de uma fórmula do CP, e um programa Prolog para a conversão de uma fórmula do CP para a forma clausal, bem como exemplos de suas execuções.

É importante lembrar que os programas mostrados a seguir não fazem a verificação da consistência dos dados fornecidos como entrada.

Todos os programas utilizam a seguinte definição de operadores:

```
:-op(200,fy,n).          /* negacao      */
:-op(400,xfy,&).         /* conjuncao    */
:-op(450,xfy,ou).        /* disjuncao    */
:-op(700,xfy,->).        /* implicacao   */
:-op(750,xfy,<->).       /* equivalencia */
```

5.1 Determinação do Valor-verdade de uma Fórmula do Cálculo Proposicional

O objetivo dos dois programas Prolog apresentados a seguir é encontrar o valor-verdade de uma fórmula do Cálculo de Proposicional. Ambos os programas fazem uso das seguintes equivalências

$$\begin{aligned} \alpha \leftrightarrow \beta &\equiv \alpha \rightarrow \beta \wedge \beta \rightarrow \alpha \\ \alpha \rightarrow \beta &\equiv \neg \alpha \vee \beta \\ \neg\neg \alpha &\equiv \alpha \\ \neg (\alpha \vee \beta) &\equiv \neg \alpha \wedge \neg \beta \\ \neg (\alpha \wedge \beta) &\equiv \neg \alpha \vee \neg \beta \end{aligned}$$

Os programas têm como entrada uma fórmula bem formada – wff – do Cálculo Proposicional, escrita com os operadores definidos anteriormente.

Primeiramente é mostrado um programa que encontra o valor-verdade de uma fórmula dado o valor-verdade de seus componentes. A seguir é apresentado um segundo programa, mais geral, onde os valores-verdade dos componentes da fórmula não são, necessariamente, conhecidos.

5.1.1 Versão I

A entrada deste programa é uma wff com os valores-verdade de cada um de seus componentes. Por exemplo

```
n (v <-> f) -> ((v ou f) & n n f)
```

a saída do programa é o valor-verdade indicado pelo átomo `v` ou `f`. O programa está definido pelo predicado `valor_verdade(Formula,Valor)` que consta de duas cláusulas:

```
valor_verdade(Formula,f):-
        falso(Formula),
        !.

valor_verdade(Formula,v).
```

onde o programa `falso(Formula)` deve ser bem sucedido se o valor-verdade de `Formula` for `f` e deve falhar caso contrário. Ele está definido pelas seguintes cláusulas, que simplesmente traduzem as equivalências acima mencionadas

```
falso(f):-!.

falso(n v):-!.

falso(P <-> Q):-
     falso((P -> Q) & (Q -> P)).

falso(P -> Q):-
     falso(n P ou Q).

falso(P ou Q):-
     falso(P),
     falso(Q).

falso(P & Q):-
     (falso(P) ; falso(Q)).

falso(n n P):-
     falso(P).

falso(n (P <-> Q)):-
     falso(n (P -> Q) ou n (Q ->P)).

falso(n (P -> Q)):-
     falso(n (n P ou Q)).

falso(n (P ou Q)):-
     falso(n P & n Q).

falso(n (P & Q)):-
     falso(n P ou n Q).
```

A fim de mostrar a execução do programa **valor_verdade/2**, será usado um outro programa — **encontre_valor/0** — acrescentado com predicados de leitura e escrita. Este programa implementa os seguintes passos:

repetir
leia fórmula
se fórmula = **fim** pare caso contrário
encontre o valor-verdade da fórmula
informe o valor-verdade da fórmula

O programa é:

```
encontre_valor :-
     repeat,
     leia(Formula),
     (Formula==fim ;
     valor_verdade(Formula,X),
     informe(Formula,X),
     fail).

leia(Formula) :-
     nl, write($Entre: $),
     read(Formula).

informe(Formula,X) :-
     write($Formula: $),
     write(Formula),
     escreva(X), nl.

escreva(v) :-
     write($  Verdadeira$),
     !.

escreva(f) :-
     write($  Falsa$).
```

A seguir é mostrado um exemplo de execução:

```
?- encontre_valor.

Entre: (v & n f) ou f.
Formula: v & n f ou f  Verdadeira

Entre: (v -> f) & (f -> v).
Formula: (v -> f) & (f -> v)  Falsa

Entre: (v ->f) <-> (f -> f).
Formula: v -> f <-> f -> f  Falsa

Entre: (f -> v) <-> (f -> v).
```

```
Formula: f -> v <-> f -> v  Verdadeira

Entre: fim.
yes
```

Deve ser observado que a primeira cláusula do programa `valor_verdade/2` está definida em função do programa `falso/2`. Este programa é tal que:

`falso(Formula,f)` é bem sucedido se o valor de `Formula` for `f`

Entretanto, é possível definir a primeira cláusula de `valor_verdade/2` em função de um programa análogo `verdade/2`, tal que:

`verdade(Formula,v)` é bem sucedido se o valor de `Formula` for `v`

Esta outra implementação do programa `valor_verdade/2` — que pode ser considerada a dual da versão anterior — é mostrada a seguir:

```
valor_verdade(Formula,v):-
     verdade(Formula),
     !.

valor_verdade(Formula,f).

verdade(v):-!.

verdade(n f):-!.

verdade(P <-> Q):-
     verdade((P -> Q) & (Q -> P)).

verdade(P -> Q):-
     verdade(n P ou Q).

verdade(P ou Q):-
     verdade(P) ; verdade(Q).

verdade(P & Q):-
     verdade(P), verdade(Q).

verdade(n n P):-
     verdade(P).

verdade(n (P <-> Q)):-
     verdade(n (P -> Q) ou n (Q ->P)).

verdade(n (P -> Q)):-
     verdade(n (n P ou Q)).

verdade(n (P ou Q)):-
```

```
     verdade(n P & n Q).

verdade(n (P & Q)):-
     verdade(n P ou n Q).
```

5.1.2 Versão II

É possível alterar o programa `valor_verdade/2` de maneira a que ele encontre o valor-verdade de uma fórmula onde os valores-verdade de seus componentes não são, necessariamente, conhecidos. Se componentes da fórmula forem variáveis livres, o programa deve instanciá-las com os átomos `f` e `v`, para calcular o correspondente valor-verdade da fórmula.

Isto é implementado pelo programa mais geral `valor_verdade/2`, mostrado a seguir:

```
valor_verdade(Formula,Val):-
     instancia_avalia(Formula,Val).


instancia_avalia(Formula,Formula):-
     var(Formula),
     !,
     pertence(Formula,[v,f]).

instancia_avalia(n Formula,Val):-
     var(Formula),
     !,
     pertence([Formula,Val],[[v,f],[f,v]]).

instancia_avalia(v,v).

instancia_avalia(f,f).

instancia_avalia(n v,f).

instancia_avalia(n f,v).

instancia_avalia(P <-> Q,Val):-
     instancia_avalia((P -> Q) & (Q -> P),Val).

instancia_avalia(P -> Q,Val):-
     instancia_avalia(n P ou Q,Val).

instancia_avalia(P ou Q,Val):-
     instancia_avalia(P,Val1),
     instancia_avalia(Q,Val2),
     (pertence1(v,[Val1,Val2]),
      Val = v;
      not pertence1(v,[Val1,Val2]),
      Val = f).
```

```
instancia_avalia(P & Q,Val):-
     instancia_avalia(P,Val1),
     instancia_avalia(Q,Val2),
     (pertence1(f,[Val1,Val2]),
      Val = f;
      not pertence1(f,[Val1,Val2]),
      Val = v).

instancia_avalia(n n P,Val):-
     instancia_avalia(P,Val).

instancia_avalia(n (P <-> Q),Val):-
     instancia_avalia(n (P -> Q) ou n (Q ->P),Val).

instancia_avalia(n (P -> Q),Val):-
     instancia_avalia(n (n P ou Q),Val).

instancia_avalia(n (P ou Q),Val):-
     instancia_avalia(n P & n Q,Val).

instancia_avalia(n (P & Q),Val):-
     instancia_avalia(n P ou n Q,Val).


pertence(X,[X|_]).
pertence(X,[_|L]) :-
     pertence(X,L).

pertence1(X,[X|_]):-
     !.
pertence1(X,[_|L]):-
     pertence1(X,L).
```

A seguir é mostrada a execução desta nova versão de **valor_verdade/2** através do programa **encontre_valor/0**, definido na seção 5.1.1

```
?- encontre_valor.

Entre: n (X ou Y) -> (X & Y <-> f).
Formula: n (v ou v) -> (v & v <-> f)  Verdadeira
Formula: n (v ou f) -> (v & f <-> f)  Verdadeira
Formula: n (f ou v) -> (f & v <-> f)  Verdadeira
Formula: n (f ou f) -> (f & f <-> f)  Verdadeira

Entre: (X -> Y) & (Y -> Z) -> X -> Z.
Formula: (v -> v) & (v -> v) -> v -> v  Verdadeira
Formula: (v -> v) & (v -> f) -> v -> f  Verdadeira
Formula: (v -> f) & (f -> v) -> v -> v  Verdadeira
Formula: (v -> f) & (f -> f) -> v -> f  Verdadeira
Formula: (f -> v) & (v -> v) -> f -> v  Verdadeira
Formula: (f -> v) & (v -> f) -> f -> f  Verdadeira
```

```
Formula: (f -> f) & (f -> v) -> f -> v  Verdadeira
Formula: (f -> f) & (f -> f) -> f -> f  Verdadeira

Entre: (v <-> X) <-> (n v ou X) & (n X ou v).
Formula: (v <-> v) <-> (n v ou v) & (n v ou v)  Verdadeira
Formula: (v <-> f) <-> (n v ou f) & (n f ou v)  Verdadeira

Entre: X -> n n n Y <-> n Z ou (n X -> n Z).
Formula: v -> n n n v <-> n v ou (n v -> n v)  Falsa
Formula: v -> n n n v <-> n f ou (n v -> n f)  Falsa
Formula: v -> n n n f <-> n v ou (n v -> n v)  Verdadeira
Formula: v -> n n n f <-> n f ou (n v -> n f)  Verdadeira
Formula: f -> n n n v <-> n v ou (n f -> n v)  Falsa
Formula: f -> n n n v <-> n f ou (n f -> n f)  Verdadeira
Formula: f -> n n n f <-> n v ou (n f -> n v)  Falsa
Formula: f -> n n n f <-> n f ou (n f -> n f)  Verdadeira

Entre: f ou X -> (v <-> Y).
Formula: f ou v -> (v <-> v)  Verdadeira
Formula: f ou v -> (v <-> f)  Falsa
Formula: f ou f -> (v <-> v)  Verdadeira
Formula: f ou f -> (v <-> f)  Verdadeira

Entre: f & v & (v ou v) <-> v -> f ou v.
Formula: f & v & (v ou v) <-> v -> f ou v  Falsa

Entre: fim.
yes
```

5.2 Conversão de uma Fórmula do Cálculo Proposicional para a Forma Clausal

Este programa implementa os seguintes passos:

1. remover equivalências e implicações
2. mover a negação para o interior da fórmula
3. colocar na forma normal conjuntiva
4. obter a representação clausal

```
traduza(Formula_logica,Clausulas) :-
% passo1: elimine os conetivos -> e <->
   tira_implicacao(Formula_logica,Formula_sem_implicacao),

% passo2: mova negacao para o interior da formula
   move_negacao(Formula_sem_implicacao,Formula_com_neg_movimentada),

% passo3: forma normal conjuntiva
```

```
   forma_normal_conj(Formula_com_neg_movimentada,Formula_normal_conjuntiva),

% passo4: representacao clausal
   clausifique(Formula_normal_conjuntiva,[],Clausulas).


/* passo1: elimine os conetivos -> e <->
Substitua:  q  -> p   por  n q v p
                  q <-> p   por (q & p) v (n q & n p)       */

tira_implicacao((P <-> Q),((P1 & Q1) v (n P1 & n Q1))) :-
   !,
   tira_implicacao(P,P1),
   tira_implicacao(Q,Q1).

tira_implicacao((P -> Q),(n P1 v Q1)) :-
   !,
   tira_implicacao(P,P1),
   tira_implicacao(Q,Q1).

tira_implicacao(para_todo(X,P),para_todo(X,P1)) :-
   !,
   tira_implicacao(P,P1).

tira_implicacao(existe(X,P),existe(X,P1))  :-
   !,
   tira_implicacao(P,P1).

tira_implicacao((P & Q),(P1 & Q1)) :-
   !,
   tira_implicacao(P,P1),
   tira_implicacao(Q,Q1).

tira_implicacao((P v Q),(P1 v Q1)) :-
   !,
   tira_implicacao(P,P1),
   tira_implicacao(Q,Q1).

tira_implicacao((n P), (n P1)) :-
   !,
   tira_implicacao(P,P1).

tira_implicacao(P,P).


/* passo2: move negacao para o interior da formula. Apos este passo as
   negacoes aparecerao apenas aplicadas  diretamente aos atomos.
   Substitua:  n(p & q) por   n p v n q
               n(p v q) por   n p & n q
               n n p    por   p                                  */

% move negacao para o interior da formula
```

```
move_negacao(( n P),P1) :-
   !,
   negacao(P,P1).


move_negacao(( P & Q ), (P1 & Q1)) :-
   !,
   move_negacao(P,P1),
   move_negacao(Q,Q1).

move_negacao(( P v Q), (P1 v Q1)) :-
   !,
   move_negacao(P,P1),
   move_negacao(Q,Q1).

move_negacao(P,P).

% aplica negacao a formula
negacao((n P),P1) :-
   !,
   move_negacao(P,P1).

negacao(( P & Q),(P1 v Q1)) :-
   !,
   negacao(P,P1),
   negacao(Q,Q1).

negacao(( P v Q),(P1 & Q1)) :-
   !,
   negacao(P,P1),
   negacao(Q,Q1).

negacao(P,(n P)).


/* passo3: forma normal conjuntiva
   transforma a formula de entrada em uma formula do tipo:
   (A v B v ...) & (C v D v ...) & ...
   Substitua: (A & B) v C    por    (A v C) & (B v C)
              A v (B v C)    por    (A v B) & (A v C)      */

forma_normal_conj(( P v Q ),R) :-
   !,
   forma_normal_conj(P,P1),
   forma_normal_conj(Q,Q1),
   conjtrans((P1 v Q1),R).

forma_normal_conj(( P & Q ),( P1 & Q1)) :-
   !,
   forma_normal_conj(P,P1),
   forma_normal_conj(Q,Q1).
```

```
forma_normal_conj(P,P).


conjtrans((( P & Q) v R),( P1 & Q1)) :-
   !,
   forma_normal_conj((P v R),P1),
   forma_normal_conj((Q v R),Q1).

conjtrans(( P v (Q & R)),( P1 & Q1)) :-
   !,
   forma_normal_conj(( P v Q),P1),
   forma_normal_conj(( P v R),Q1).

conjtrans(P,P).

/* passo4: representacao clausal
   Cada clausula e representada como uma estrutura clausula(A,B),
   onde A e uma lista de literais nao negados e B e uma lista  de
   literais negados.
   O terceiro parametro e a lista de clausulas e o segundo e  uma
   lista intermediaria.                                          */

clausifique((P & Q),C1,C2) :-
   !,
   clausifique(P,C1,C3),
   clausifique(Q,C3,C2).

clausifique(P,Cs,[clausula(A,B)|Cs]) :-
   na_clausula(P,A,[],B,[]),!.

clausifique(_,C,C).

/* Construcao das listas A e B de literais, que sao a cabeca e cauda,
   respectivamente, de uma clausula, verificando que uma mesma formu-
   la nao apareca nas duas listas ou duas vezes em uma lista.
   O primeiro parametro e uma disjuncao, A e B sao as listas de lite-
   rais e A1 e B1 sao listas intermediarias.                      */

na_clausula((P v Q),A,A1,B,B1) :-
   !,
   na_clausula(P,A2,A1,B2,B1),
   na_clausula(Q,A,A2,B,B2).

na_clausula((n P),A,A,B1,B) :-
   !,
   not(esta_dentro(P,A)),
   colocar_dentro(P,B,B1).

na_clausula(P,A1,A,B,B) :-
   not(esta_dentro(P,B)),
   colocar_dentro(P,A,A1).
```

```
esta_dentro(X,[X|_]):- !.

esta_dentro(X,[_|L]):-
   esta_dentro(X,L).


colocar_dentro(X,[],[X]):- !.

colocar_dentro(X,[X|L],[X|L]):- !.

colocar_dentro(X,[Y|L],[Y|L1]):-
   colocar_dentro(X,L,L1).
```

A seguir são mostrados alguns exemplos de execução.

```
?- traduza(p -> q v r,Y).
Y = [clausula([q,r],[p])] ->;
no

?-traduza( n p & n (n q) <-> m -> p v n q,Y).
Y = [clausula([p],[m,q]),clausula([q],[]),clausula([q],[p]),
     clausula([q,m],[]),clausula([q],[p]),clausula([],[p]),
     clausula([m],[p])] ->;
no

?-traduza( m v n (q & n r) -> n p & (r <-> n m),Y).
Y = [clausula([],[r,m]),clausula([],[r,p]),clausula([q],[m,r]),
      clausula([q,r,m],[]),clausula([q],[p]),clausula([],[m,r]),
      clausula([],[m,p])] ->;
no
```

A fim de mostrar a execução de cada um dos passos do programa, será usado o procedimento `traduza_escreva/2` descrito a seguir, que é simplesmente `traduza/2` incrementado com os seguintes procedimentos:

`escreva_saida_passo/3` que mostra a saída da execução de cada um dos passos do programa e
`escreva_notacao_clausal/1` que mostra a notação clausal na forma utilizada por Prolog.

```
/*  Programa para converter formulas do Calculo Preposicional
    para a Forma Clausal e mostrar saida de cada um dos Passos  */

traduza_escreva(Formula_logica,Clausulas) :-
   escreva_saida_passo($Formula do CP$,$a ser Transformada$,Formula_logica),

% passo1: elimine os conetivos -> e <->
   tira_implicacao(Formula_logica,Formula_sem_implicacao),
   escreva_saida_passo(1,$elimine os conetivos -> e <->$,
```

```
                                    Formula_sem_implicacao),

% passo2: mova negacao para o interior da formula
   move_negacao(Formula_sem_implicacao,Formula_com_neg_movimentada),
   escreva_saida_passo(2,$mova negacao para o interior da formula$,
                                    Formula_com_neg_movimentada),

% passo3: forma normal conjuntiva
   forma_normal_conj(Formula_com_neg_movimentada,Formula_normal_conjuntiva),
   escreva_saida_passo(3,$forma normal conjuntiva$,Formula_normal_conjuntiva),

% passo4: representacao clausal
   clausifique(Formula_normal_conjuntiva,[],Clausulas),
   escreva_saida_passo(4,$representacao clausal$,Clausulas),
   escreva_notacao_clausulas(Clausulas),!.


escreva_saida_passo(A,B,C) :-
   separe,
   write(A),tab(3),write(B),nl,nl,
   write(C),
   separe,
   continue.

separe:- nl,write($-----------------------------------------------$),nl.

continue :-
   tab(20),write($Aperte qquer tecla para continuar....$),get0(_),nl.

escreva_notacao_clausulas(Cls) :-
   separe,
   write($Notacao Clausal$),nl,nl,
   escreva_clausulas(Cls),
   separe.

/* escreva_clausulas transforma a representacao clausal da
   formula em uma notacao especial do tipo:
   A; B; ... :- K, L, ...
   onde A, B, ... sao literais nao negadas e
        K, L, ... sao literais negadas                  */

escreva_clausulas([]) :- !.

escreva_clausulas([clausula(A,B)|Cs]) :-
   escreva_clausula(A,B),nl,
   escreva_clausulas(Cs).

escreva_clausula(L,[]) :-
   !,
   escreva_disjuncao(L),write('.').

escreva_clausula([],L) :-
```

```
   !,
   write(' :- '),
   escreva_conjuncao(L),write('.').

escreva_clausula(L1,L2) :-
   escreva_disjuncao(L1),
   write(' :- '),
   escreva_conjuncao(L2),write('.').

escreva_disjuncao([L]) :-
   !,
   write(L).

escreva_disjuncao([L|Ls]) :-
   write(L),
   write(' ; '),
   escreva_disjuncao(Ls).

escreva_conjuncao([L]) :-
   !,
   write(L).

escreva_conjuncao([L|Ls]) :-
   !,
   write(L),
   write(', '),
   escreva_conjuncao(Ls).
```

A seguir é mostrada a execução deste programa, com os mesmos dados de entrada utilizados anteriormente.

```
?- traduza_escreva(p -> q v r,Y).

-----------------------------------------------
Formula do CP    a ser Transformada

p -> q v r
-----------------------------------------------
                    Aperte qquer tecla para continuar....

-----------------------------------------------
1   elimine os conetivos -> e <->

n p v q v r
-----------------------------------------------
                    Aperte qquer tecla para continuar....

-----------------------------------------------
2   mova negacao para o interior da formula

n p v q v r
```

```
-----------------------------------------------
                    Aperte qquer tecla para continuar....

-----------------------------------------------
3   forma normal conjuntiva

n p v q v r
-----------------------------------------------
                    Aperte qquer tecla para continuar....

-----------------------------------------------
4   representacao clausal

[clausula([q,r],[p])]
-----------------------------------------------
                    Aperte qquer tecla para continuar....

-----------------------------------------------
Notacao Clausal

q ; r :- p.

-----------------------------------------------

Y = [clausula([q,r],[p])] ->;
no


?-traduza( n p & n (n q) <-> m -> p v n q,Y).

-----------------------------------------------
Formula do CP   a ser Transformada

n p & n (n q) <-> m -> p v n q
-----------------------------------------------
                    Aperte qquer tecla para continuar....

-----------------------------------------------
1   elimine os conetivos -> e <->

(n p & n (n q)) & (n m v p v n q) v n (n p & n (n q)) &
n (n m v p v n q)
-----------------------------------------------
                    Aperte qquer tecla para continuar....

-----------------------------------------------
2   mova negacao para o interior da formula

(n p & q) & (n m v p v n q) v (p v n q) & m & n p & q
-----------------------------------------------
                    Aperte qquer tecla para continuar....
```

```
-----------------------------------------------
3   forma normal conjuntiva

(((n p v p v n q) & (n p v m) & (n p v n p) & (n p v q)) &
(q v p v n q) & (q v m) & (q v n p) & (q v q)) &
((n m v p v n q) v p v n q) & ((n m v p v n q) v m) &
((n m v p v n q) v n p) & ((n m v p v n q) v q)
-----------------------------------------------
                    Aperte qquer tecla para continuar....

-----------------------------------------------
4   representacao clausal

[clausula([p],[m,q]),clausula([q],[]),clausula([q],[p]),
 clausula([q,m],[]),clausula([q],[p]),clausula([],[p]),
 clausula([m],[p])]
-----------------------------------------------
                    Aperte qquer tecla para continuar....

-----------------------------------------------
Notacao Clausal

p :- m, q.
q.
q :- p.
q ; m.
q :- p.
 :- p.
m :- p.

-----------------------------------------------

Y = [clausula([p],[m,q]),clausula([q],[]),clausula([q],[p]),
     clausula([q,m],[]),clausula([q],[p]),clausula([],[p]),
     clausula([m],[p])] ->;
no


?-traduza( m v n (q & n r) -> n p & (r <-> n m),Y).

-----------------------------------------------
Formula do CP   a ser Transformada

m v n (q & n r) -> n p & (r <-> n m)
-----------------------------------------------
                    Aperte qquer tecla para continuar....

-----------------------------------------------
1   elimine os conetivos -> e <->

n (m v n (q & n r)) v n p & (r & n m v n r & n (n m))
-----------------------------------------------
```

```
                    Aperte qquer tecla para continuar....

-----------------------------------------------
2   mova negacao para o interior da formula

n m & q & n r v n p & (r & n m v n r & m)
-----------------------------------------------
                    Aperte qquer tecla para continuar....

-----------------------------------------------
3   forma normal conjuntiva

((n m v n p) & ((n m v r v n r) & (n m v r v m)) &
(n m v n m v n r) & (n m v n m v m)) & ((q v n p) &
((q v r v n r) & (q v r v m)) & (q v n m v n r) &
(q v n m v m)) & (n r v n p) & ((n r v r v n r) &
(n r v r v m)) & (n r v n m v n r) & (n r v n m v m)
-----------------------------------------------
                    Aperte qquer tecla para continuar....

-----------------------------------------------
4   representacao clausal

[clausula([],[r,m]),clausula([],[r,p]),clausula([q],[m,r]),
 clausula([q,r,m],[]),clausula([q],[p]),clausula([],[m,r]),
 clausula([],[m,p])]
-----------------------------------------------
                    Aperte qquer tecla para continuar....

-----------------------------------------------
Notacao Clausal

 :- r, m.
 :- r, p.
q :- m, r.
q ; r ; m.
q :- p.
 :- m, r.
 :- m, p.

-----------------------------------------------

 Y = [clausula([],[r,m]),clausula([],[r,p]),clausula([q],[m,r]),
      clausula([q,r,m],[]),clausula([q],[p]),clausula([],[m,r]),
      clausula([],[m,p])] ->;
no
```

6 Conclusões

Como comentado anteriormente, esta Nota Didática tem como objetivo familiarizar o leitor com os conceitos básicos do Cáculo Proposicional enfatizando aqueles que subsidiam a linguagem de programação lógica Prolog.

A publicação de programas Prolog que implementam soluções de problemas típicos do CP intenciona contribuir para uma maior divulgação de metaprogramação, que pode facilmente ser realizada nesta linguagem.

Esta Nota Didática terá continuidade em uma próxima nota intitulada *O Cálculo de Predicados: uma Abordagem Voltada à Compreensão da Linguagem Prolog*, a ser publicada futuramente.

Agradecimentos:

Gostaríamos de agradecer à Profa. Doris Ferraz de Aragão — ILTC/UFF — e à Profa. Lucia Helena Machado Rino — DC/UFSCar —- pela leitura, correção e comentários sobre o conteúdo de versões preliminares desta Nota.

Referências

[Aragon 90] Aragon, D.F.; Castro I.D.; Alcanforado, P. *Lógica para Informática: um Estudo Introdutório* ILTC, 1990.

[Araribóia 89] Araribóia, G. *Inteligência Artificial: Um Curso Prático.* Livros Técnicos e Científicos Editora Ltda, 1989.

[Arity 90] Arity Corporation. *The Arity/Prolog Programming Language.* 1990.

[Bratko 90] Bratko, I. *Prolog Programming for Artificial Intelligence.* Addison-Wesley, (2nd Ed.), 1990.

[Bundy 83] Bundy, A. *The Computer Modelling of Mathematical Reasoning.* Academic Press, London, 1983.

[Casanova 87] Casanova, M.A.; Giorno, F.A.C.; Furtado, A.L. *Programação em Lógica e a Linguagem Prolog.* Editora Edgard Blücher Ltda., São Paulo, 1987.

[Chang 73] Chang, C.; Lee, R.C. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, New York, 1973.

[Clocksin 87] Clocksin, W.F.; Mellish, C.S. *Programming in Prolog.* (3rd Ed.) Springer-Verlag, 1987.

[Coelho 88] Coelho, H.; Cotta, J.C. *Prolog by Examples.* Springer-Verlag, 1988.

[Copi 86] Copi, I.M. *Introduction to Logic.* Macmillan, New York, 1986.

[Cuena 85] Cuena, J. *Logica Informatica.* Alianza Editorial S.A., Madrid, 1985.

[Dougherty 88] Dougherty, E.R.; Giardina, C.R. *Mathematical Methods for Artificial Intelligence and Autonomous Systems.* Prentice Hall, 1988.

[Enderton 72] Enderton, H.B. *A Mathematical Introduction to Logic.* Academic Press, New York, 1972.

[Exner 59] Exner, R.M.; Rosskopf, M.F. *Logic in Elementary Mathematics.* MacGraw-Hill Book Company, New York, 1959.

[Gray 84] Gray, P.M. *Logic, Algebra and Database.* John Wiley & Sons, New York, 1984.

[Hamilton 78] Hamilton, A.G. *Logic for Mathematicians.* Cambridge University Press, Cambridge, England, 1978.

[Hegenberg 76] Hegenberg, L. *Simbolização no Cálculo de Predicados.* Ed. da USP, Vol III, São Paulo, 1976.

[Hegenberg 77] Hegenberg, L. *Dedução no Cálculo Sentencial.* Ed. da USP, Vol II, São Paulo, 1977.

[Hegenberg 78] Hegenberg, L. *Tabelas e Argumentos*. Ed. da USP, Vol I, São Paulo, 1978.

[Hegenberg 78] Hegenberg, L. *Dedução no Cálculo de Predicados*. Ed. da USP, Vol IV, São Paulo, 1978.

[Hogger 84] Hogger, C.J. *Introduction to Logic Programming*. Academic Press, London, 1984.

[Kluzniak 85] Kluzniak, F.; Szpakowisz, S. *Prolog for Programmers*. Academic Press, London, 1985.

[Kowalski 79] Kowalski, R.A. *Logic for Problem Solving*. Artificial Intelligence Series, Vol 7, Elsevier-North Holland, 1979.

[Lloyd 84] Lloyd, J,W. *Foundations of Logic Programs*. Springer-Verlag, New York, 1984.

[Loveland 78] Loveland, D.N. *Automated Theorem Proving: A Logical Basis*. North-Holland Publishing Company, Amsterdan, 1978.

[Maier 88] Maier, D.; Warren, D.S. *Computing with Logic: Logic Programming with Prolog*. The Benjamin/Cummings Publishing Company Inc., 1988.

[Malpas 87] Malpas, J. *Prolog: A Relational Language and its Applications*. Prentice-Hall, New Jersey, 1987.

[Manna 85] Manna, Z.; Waldinger, R. *The Logical Basis of Computer Programming*. Addison-Wesley, Reading, Ma, 1985.

[Marcus 86] Marcus, C. *Prolog Programming: Applications for DataBase Systems, Expert Systems and Natural Language Systems*. Addison-Wesley, 1986.

[Mates 72] Mates, B. *Elementary Logic*. Oxford University Press. Inc., New York, 1972.

[Medelson 64] Medelson, E. *Introduction to Mathematical Logic*. Princeton, 1964.

[Medelson 79] Medelson, E. *Introduction to Mathematical Logic*. D. Van Nostrand, New York, 1979.

[Monard 90] Monard, M.C.; Nicoletti, M.C. *Método Sintático de Prova de Teoremas Algoritmo de Wang*. Notas do ICMSC - USP, Nº62, 1990, 54 pg.

[Robison 65] Robison, J.A. *A Machine-Oriented Logic Based on the Resolution Principle*. JACM, Vol 12, No. 1, pp 23-41, January 1965.

[Shoenfield 67] Shoenfield, J.R. *Mathematical Logic*. Addison-Wesley Publishing Company, Massachusetts, 1967.

[Siekmann 83] Siekmann, J; Wrightson, G. (Eds.) *Automation of Reasoning 1 Classical Papers on Computational Logic 1957-1966*. Springer-Verlag, Berlin, 1983a.

[Siekmann 83] Siekmann, J; Wrightson, G. (Eds.) *Automation of Reasoning 2 Classical Papers on Computational Logic 1967-1970*. Springer-Verlag, Berlin, 1983b.

[Sterling 86] Sterling, L.; Shapiro, E. *The Art of Prolog*. The MIT Press, 1986.

[Walker 87] Walker, A. (Ed); McCord, M.; Sowa, J.E.; Wilson,W.G. *Knowledge Systems and Prolog. A Logical Approach for Expert Systems and Natural Language Processing*. Addison-Wesley, 1987.

[Wang 60] Wang, H. *Towards Mechanical Mathematics*. IBM Journal of Research and Development, Vol 4, pp 2-22, 1960.

[Wang 64] Wang,H. *A Survey of Mathematical Logic*. North-Holland Publishing Company, 1964.

[Warren 77] Warren, D.H.D.; Pereira, L.M.; Pereira, F. *Prolog-The Language and its Implementation Compared with Lisp*. SIGART Newsletter, No. 64, pp 109-115, August 1977.