

PROGRAMAÇÃO EM LÓGICA

FUNDAÇÕES

Plano

- Conhecimento
 - representação
 - inferência
- Lógica proposicional: revisões
- Lógica de primeira ordem
- Programação em Lógica

Representação de Conhecimento

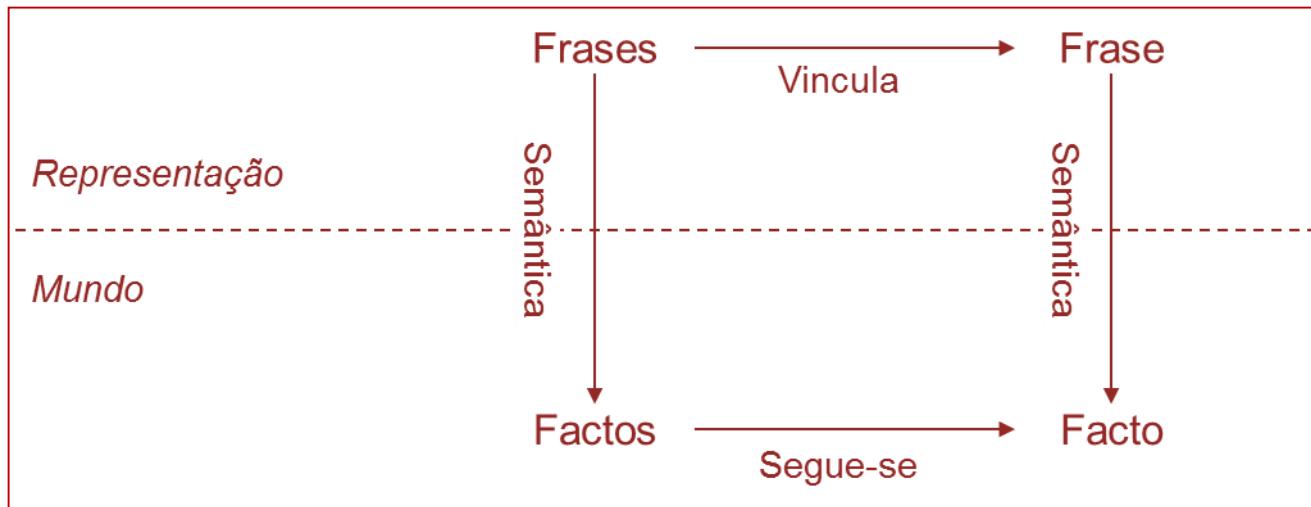
- Base de conhecimento (*KB*)
 - conjunto de **frases** expressas numa **linguagem de representação de conhecimento**
- **Sintaxe**: especifica as frases que estão bem formadas
 - e.g. “ $x+y=4$ ”, mas não “ $x4y+=$ ”
- **Semântica**: atribui significado às frases, determina veracidade em relação a cada “mundo possível” ou **modelo**
 - e.g. “ $x+y=4$ ” é verdade num mundo em que x e y são ambos 2
- Uma interpretação é um mapeamento entre: constantes da lógica e objetos de um determinado domínio; funções da lógica e funções do domínio (propriedades dos objetos); e predicados da lógica e relações nesse domínio
- Se a frase α é verdadeira numa interpretação m : diz-se que m é um **modelo de α**
- **$M(\alpha)$** : conjunto de todos os modelos de α

Raciocínio Lógico

- Implicação lógica (Logical Implication)
 - $\alpha \vDash \beta$
 - α implica logicamente β (ou de α segue-se logicamente β)
 - $\alpha \vDash \beta$ se e só se $M(\alpha) \subseteq M(\beta)$
 - α será portanto uma asserção mais forte do que β
- Inferência lógica ou dedução
 - aplica a relação de implicação lógica entre frases para derivar conclusões
- Procedimento de inferência
 - **correto** (*sound*) se deriva *apenas* frases vinculadas
 - **completo** se é capaz de derivar *todas* as frases vinculadas

Correspondência

- Se KB é verdade no mundo real, então qualquer frase α derivada de KB por um procedimento de inferência correto é também verdade no mundo real



- O procedimento de inferência
 - opera sobre as representações sintáticas (frases), mas corresponde à relação existente entre os factos no mundo real
 - consiste em construir novas frases a partir das existentes
 - para ser correto, deve derivar apenas novas frases que representem factos que se seguem dos factos representados pela KB

Plano

- Conhecimento
- Lógica proposicional: revisões
 - sintaxe
 - semântica
 - inferência
- Lógica de primeira ordem
- Programação em Lógica

Lógica Proposicional: Sintaxe

- Símbolos:
 - constantes lógicas *True* e *False*
 - símbolos proposicionais como P e Q
 - conectivas lógicas: $\wedge \quad \vee \quad \Rightarrow \quad \Leftrightarrow \quad \neg$
 - parêntesis (e)
- Frases são agrupamentos de símbolos, tais que:
 - *True*, *False*, P ou Q são frases por si só
 - frase dentro de parêntesis é uma frase: $(P \wedge Q)$
 - frase complexa combinando frases mais simples com conectivas lógicas:
 - \wedge (e) – frase cuja conectiva principal é \wedge chama-se uma **conjunção**: $P \wedge (Q \vee R)$
 - \vee (ou) – frase cuja conectiva principal é \vee chama-se uma **disjunção**: $A \vee (P \wedge Q)$
 - \Rightarrow (implica) – frase do tipo $(P \wedge Q \Rightarrow R)$ chama-se uma **implicação material** (ou simplesmente implicação)
 - \Leftrightarrow (equivalente) – a frase $(P \wedge Q) \Leftrightarrow (Q \wedge P)$ é uma **equivalência**
 - \neg (não) – uma frase como $\neg P$ chama-se a **negação** de P
 - ordem de precedência dos operadores: $\neg \quad \wedge \quad \vee \quad \Rightarrow \quad \Leftrightarrow$
 - a frase $\neg P \vee Q \wedge R \Rightarrow S$ é equivalente à frase $((\neg P) \vee (Q \wedge R)) \Rightarrow S$

Lógica Proposicional: Semântica

- Símbolo proposicional pode ter como interpretação qualquer facto
- *True* representa sempre um facto verdadeiro
- *False* representa sempre um facto falso
- Frase complexa tem significado derivado do significado das partes
 - valor pode ser obtido através de uma tabela de verdade
- Tabela de verdade para as conectivas lógicas:

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
<i>False</i>	<i>False</i>	<i>True</i>	<i>False</i>	<i>False</i>	<i>True</i>	<i>True</i>
<i>False</i>	<i>True</i>	<i>True</i>	<i>False</i>	<i>True</i>	<i>True</i>	<i>False</i>
<i>True</i>	<i>False</i>	<i>False</i>	<i>False</i>	<i>True</i>	<i>False</i>	<i>False</i>
<i>True</i>	<i>True</i>	<i>False</i>	<i>True</i>	<i>True</i>	<i>True</i>	<i>True</i>

- Frases complexas são definidas por um processo de decomposição
 - $(P \vee Q) \wedge \neg S$: primeiro determina-se o significado de $(P \vee Q)$ e de $\neg S$, depois combinam-se os dois usando a definição de \wedge

Equivalência

- Duas frases α e β são **logicamente equivalentes** se são verdade no mesmo conjunto de modelos: $M(\alpha) = M(\beta)$
 - $\alpha \equiv \beta$ se e só se $\alpha \models \beta$ e $\beta \models \alpha$

$$\begin{aligned}(\alpha \wedge \beta) &\equiv (\beta \wedge \alpha) \\(\alpha \vee \beta) &\equiv (\beta \vee \alpha) \\((\alpha \wedge \beta) \wedge \gamma) &\equiv (\alpha \wedge (\beta \wedge \gamma)) \\((\alpha \vee \beta) \vee \gamma) &\equiv (\alpha \vee (\beta \vee \gamma)) \\\neg(\neg \alpha) &\equiv \alpha \\(\alpha \Rightarrow \beta) &\equiv (\neg \beta \Rightarrow \neg \alpha) \\(\alpha \Rightarrow \beta) &\equiv (\neg \alpha \vee \beta) \\(\alpha \Leftrightarrow \beta) &\equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) \\\neg(\alpha \wedge \beta) &\equiv (\neg \alpha \vee \neg \beta) \\\neg(\alpha \vee \beta) &\equiv (\neg \alpha \wedge \neg \beta) \\(\alpha \wedge (\beta \vee \gamma)) &\equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) \\(\alpha \vee (\beta \wedge \gamma)) &\equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))\end{aligned}$$

Validade e Consistência

- **Validade**

- uma frase é **válida** se for verdadeira em todos os modelos
(tautologia: frase necessariamente verdadeira)
 - $P \vee \neg P$
- $\alpha \vDash \beta$ se e só se $(\alpha \Rightarrow \beta)$ é uma frase válida

- **Consistência (satisfiability)**

- uma frase é **consistente** (satisfazível) se for verdadeira nalgum modelo
 - α é válida se $\neg\alpha$ for **inconsistente**
 - α é consistente se $\neg\alpha$ for não válida
 - $\alpha \vDash \beta$ se e só se $(\alpha \wedge \neg\beta)$ for uma frase inconsistente
 - princípio da *prova por contradição*

Teste de Validade

- Tabelas de verdade podem ser usadas para testar frases válidas
 - se a frase for verdadeira em todas as linhas, então é válida
 - $((P \vee H) \wedge \neg H) \Rightarrow P$

P	H	$P \vee H$	$(P \vee H) \wedge \neg H$	$((P \vee H) \wedge \neg H) \Rightarrow P$
<i>False</i>	<i>False</i>	<i>False</i>	<i>False</i>	<i>True</i>
<i>False</i>	<i>True</i>	<i>True</i>	<i>False</i>	<i>True</i>
<i>True</i>	<i>False</i>	<i>True</i>	<i>True</i>	<i>True</i>
<i>True</i>	<i>True</i>	<i>True</i>	<i>False</i>	<i>True</i>

- **Regras de inferência** permitem-nos fazer inferências sem necessidade de construir tabelas de verdade
 - uma regra de inferência é lógica (*sound*) se a conclusão é verdadeira sempre que as premissas são verdadeiras

Regras de Inferência

$$\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta}$$

$$\frac{\alpha_1 \wedge \alpha_2 \wedge \dots \wedge \alpha_n}{\alpha_i}$$

$$\frac{\alpha_1, \alpha_2, \dots, \alpha_n}{\alpha_1 \wedge \alpha_2 \wedge \dots \wedge \alpha_n}$$

$$\frac{\alpha_i}{\alpha_1 \vee \alpha_2 \vee \dots \vee \alpha_n}$$

$$\frac{\neg \neg \alpha}{\alpha}$$

$$\frac{\alpha \vee \beta, \quad \neg \beta}{\alpha}$$

• **Modus Ponens:** de uma implicação e da sua premissa, podemos inferir a conclusão

• **And-Elimination:** de uma conjunção, podemos inferir qualquer dos conjuntores

• **And-Introduction:** de uma lista de frases, podemos inferir a sua conjunção

• **Or-Introduction:** de uma frase, podemos inferir a sua disjunção com qualquer coisa

• **Double-Negation Elimination:** de uma frase duplamente negada, podemos inferir uma frase afirmada

• **Unit Resolution:** de uma disjunção, se um dos disjuntores é falso, podemos inferir que o outro é verdadeiro

• **Resolution:** como β não pode ser verdadeiro e falso, um dos outros disjuntores tem que ser verdadeiro

$$\frac{\alpha \vee \beta, \quad \neg \beta \vee \gamma}{\alpha \vee \gamma}$$

ou

$$\frac{\neg \alpha \Rightarrow \beta, \quad \beta \Rightarrow \gamma}{\neg \alpha \Rightarrow \gamma}$$

Plano

- Conhecimento
- Lógica proposicional: revisões
- Lógica de 1^a ordem
 - sintaxe, semântica e inferência
 - unificação
 - cláusulas de Horn
 - resolução
 - Forma Normal Conjuntiva
- Programação em Lógica

Lógica de Primeira Ordem

- Mundo consiste em **objetos** com propriedades e **relações**
- Símbolos
 - **Constantes** (representam objetos): *A, B, C, João, PaiDoJoão, ...*
 - **Predicados** (representam relações): *Redondo, Irmão, MenorQue, ...*
 - **Funções** (relações com um só valor possível): *Coseno, Pai, PernaEsquerda, ...*
- Variáveis: *a, x, s, ...*
- **Termos**: constituídos por constantes, variáveis ou funções
 - *João, x, PernaEsquerda(João), ...*
- **Frases**
 - Atómicas: predicado e lista de termos
 - *Irmão(Ricardo, João)*
 - *Casados(Pai(Ricardo), Mãe(João))*
 - Complexas: usar conectivas lógicas
 - $\neg \wedge \vee \Rightarrow \Leftrightarrow$

Quantificadores (\forall e \exists)

- **Universal (\forall)**: expressar propriedades de colecções de objetos
 - “Todos os gatos são mamíferos.”
 - $\forall x \text{ Gato}(x) \Rightarrow \text{Mamífero}(x)$
- **Existencial (\exists)**: afirmar sobre um objeto sem dizer qual é
 - “O João tem uma irmã que é casada.”
 - $\exists x \text{ Irmã}(x, \text{João}) \wedge \text{Casada}(x)$
- Múltiplos quantificadores:
 - $\forall x, y \equiv \forall x \forall y \equiv \forall y \forall x$
 - $\forall x \exists y \neq \exists y \forall x$
 - $\forall x \exists y \text{ Gosta}(x, y)$ “Toda a gente gosta de alguém.”
 - $\exists y \forall x \text{ Gosta}(x, y)$ “Existe alguém de quem toda a gente gosta.”
 - $\forall y \exists x \text{ Gosta}(x, y)$ “Toda a gente tem alguém que gosta dela.”
 - $\exists x \forall y \text{ Gosta}(x, y)$ “Existe alguém que gosta de toda a gente.”
- Ligações entre \forall e \exists , através da negação (leis de De Morgan)
 - $\forall x \neg \text{Gosta}(x, \text{Exames}) \equiv \neg \exists x \text{ Gosta}(x, \text{Exames})$
 - $\forall x \text{ Gosta}(x, \text{Saúde}) \equiv \neg \exists x \neg \text{Gosta}(x, \text{Saúde})$
 - $\exists x \neg \text{Gosta}(x, \text{Sopa}) \equiv \neg \forall x \text{ Gosta}(x, \text{Sopa})$
 - $\exists x \text{ Gosta}(x, \text{Sopa}) \equiv \neg \forall x \neg \text{Gosta}(x, \text{Sopa})$

Regras de Inferência com Quantificadores

- São mais complexas: consistem em **substituir** variáveis por indivíduos particulares
- **SUBST(θ, α)** representa o resultado de aplicar a **substituição** θ à frase α
 - $SUBST(\{x/\text{João}, y/\text{Couve}\}, Gosta(x, y)) = Gosta(\text{João}, \text{Couve})$
- **Universal Elimination:**
 - para qualquer frase α , variável v e termo g :
$$\frac{\forall v \alpha}{SUBST(\{v/g\}, \alpha)}$$
 - p. ex., de $\forall x Gosta(x, \text{Gelado})$, podemos usar a substituição $\{x/Rui\}$ e inferir $Gosta(Rui, \text{Gelado})$
- **Existential Elimination:**
 - para qualquer frase α , variável v e constante k ainda não utilizada:
$$\frac{\exists v \alpha}{SUBST(\{v/k\}, \alpha)}$$
 - no fundo, damos um nome ao objeto que satisfaz a condição existencial
 - p. ex., de $\exists x Matou(x, \text{Vítima})$ podemos inferir $Matou(\text{Assassino}, \text{Vítima})$ desde que *Assassino* não seja o nome de outro objeto qualquer
- **Existential Introduction:**
 - para qualquer frase α , variável v que não ocorre em α e termo g :
$$\frac{\alpha}{\exists v SUBST(\{g/v\}, \alpha)}$$
 - p. ex., de $Gosta(\text{João}, \text{Gelado})$ podemos inferir $\exists x Gosta(x, \text{Gelado})$

Exemplo de Utilização

- É crime vender mísseis:
 $\forall x, y, z \text{ Missile}(y) \wedge \text{Sells}(x, z, y) \Rightarrow \text{Criminal}(x)$ (1)
- O país *Nono* tem mísseis:
 $\exists x \text{ Owns}(\text{Nono}, x) \wedge \text{Missile}(x)$ (2)
- Todos os seus mísseis foram vendidos pelo *West*:
 $\forall x \text{ Owns}(\text{Nono}, x) \wedge \text{Missile}(x) \Rightarrow \text{Sells}(\text{West}, \text{Nono}, x)$ (3)
- *West* é um criminoso?
 - (2) e Existential Elimination:
 $\text{Owns}(\text{Nono}, M1) \wedge \text{Missile}(M1)$ (4)
 - (4) e And-Elimination:
 $\text{Owns}(\text{Nono}, M1)$ (5)
 $\text{Missile}(M1)$ (6)
 - (3) e Universal Elimination:
 $\text{Owns}(\text{Nono}, M1) \wedge \text{Missile}(M1) \Rightarrow \text{Sells}(\text{West}, \text{Nono}, M1)$ (7)
 - (7), (4) e Modus Ponens:
 $\text{Sells}(\text{West}, \text{Nono}, M1)$ (8)
 - (1) e Universal Elimination:
 $\text{Missile}(M1) \wedge \text{Sells}(\text{West}, \text{Nono}, M1) \Rightarrow \text{Criminal}(\text{West})$ (9)
 - (6), (8) e And-Introduction:
 $\text{Missile}(M1) \wedge \text{Sells}(\text{West}, \text{Nono}, M1)$ (10)
 - (9), (10) e Modus Ponens:
 $\text{Criminal}(\text{West})$ (11)

Generalização do Modus Ponens

- Para frases atómicas p_i, p'_i e q , se houver uma substituição θ tal que $SUBST(\theta, p'_i) = SUBST(\theta, p_i)$ para todo o i :

$$\frac{p'_1, p'_2, \dots, p'_n, (p_1 \wedge p_2 \wedge \dots \wedge p_n \Rightarrow q)}{SUBST(\theta, q)}$$

- I.e., se houver uma substituição que torne as premissas da implicação idênticas a frases que estejam na KB, então podemos inferir a conclusão da implicação, depois de aplicar a substituição
 - (3), (5), (6) e Gen. Modus Ponens: $\theta = \{x/M1\}$ unifica $Owns(Nono, x) \wedge Missile(x)$ com $Owns(Nono, M1)$ e $Missile(M1)$, obtendo $Sells(West, Nono, M1)$
 - (1), (6), (8) e Gen. Modus Ponens: $\theta = \{x/West, y/M1, z/Nono\}$ unifica $Missile(y) \wedge Sells(x, z, y)$ com $Missile(M1)$ e $Sells(West, Nono, M1)$, obtendo $Criminal(West)$
- Mais eficiente pois combina várias inferências numa só
- Utiliza um algoritmo de **unificação**, que toma duas frases e retorna uma substituição que as torne idênticas (se houver)

Unificação

- Dadas duas frases atómicas p e q retorna uma substituição que as torne idênticas:
 $\text{UNIFY}(p, q) = \theta$ onde $\text{SUBST}(\theta, p) = \text{SUBST}(\theta, q)$
 - diz-se que θ é o unificador das duas frases
- Exemplo: de quem é que o João gosta?
 $\text{Conhece}(\text{João}, x) \Rightarrow \text{Gosta}(\text{João}, x)$
 $\text{Conhece}(\text{João}, \text{Joana}) \quad \text{Conhece}(y, \text{Leonel})$
 $\text{Conhece}(y, \text{Mãe}(y)) \quad \text{Conhece}(x, \text{Elizabete})$
 - Modus Ponens: encontrar frases que unificam com a premissa e aplicar o unificador à conclusão
 - $\text{UNIFY}(\text{Conhece}(\text{João}, x), \text{Conhece}(\text{João}, \text{Joana})) = \{x/\text{Joana}\}$
 - $\text{UNIFY}(\text{Conhece}(\text{João}, x), \text{Conhece}(y, \text{Leonel})) = \{x/\text{Leonel}, y/\text{João}\}$
 - $\text{UNIFY}(\text{Conhece}(\text{João}, x), \text{Conhece}(y, \text{Mãe}(y))) = \{y/\text{João}, x/\text{Mãe}(\text{João})\}$
 - $\text{UNIFY}(\text{Conhece}(\text{João}, x), \text{Conhece}(x_2, \text{Elizabete})) = \{x/\text{Elizabete}, x_2/\text{João}\}$
 - as frases $\forall x \text{ Conhece}(x, \text{Elizabete})$ e $\forall x_2 \text{ Conhece}(x_2, \text{Elizabete})$ têm o mesmo significado

- Unificador Mais Geral (**MGU**)
 - Substituição que compromete as variáveis o mínimo possível
 - $\text{UNIFY}(\text{Conhece}(\text{João}, x), \text{Conhece}(y, z)) = \{y/\text{João}, x/z\}$
 - em vez de, por exemplo, $\{y/\text{João}, x/\text{João}, z/\text{João}\}$

Cláusulas definidas

- “Limitação” do Modus Ponens: todas as frases da KB têm que estar numa das formas das premissas
 - frases ou **cláusulas definidas**:
 - frases atómicas
 - implicações com conjunção de frases atómicas na premissa e uma só frase atómica na conclusão
- Forma genérica: $P_1 \wedge P_2 \wedge \cdots \wedge P_n \Rightarrow Q$
- Casos especiais:
 - se Q for *False*, obtemos uma frase do tipo $\neg P_1 \vee \neg P_2 \vee \cdots \vee \neg P_n$
 - se $n = 1$ e $P_1 = \text{True}$, obtemos $\text{True} \Rightarrow Q$, o que é equivalente à frase atómica Q
- Cláusulas definidas têm exactamente um literal positivo
 - a forma genérica pode ser escrita como $\neg P_1 \vee \neg P_2 \vee \cdots \vee \neg P_n \vee Q$
- Exemplos de frases que ficam de fora (mais do que um literal positivo, cláusulas não definidas):
 - $\forall x \text{ Pessoa}(x) \Rightarrow \text{Homem}(x) \vee \text{Mulher}(x)$
 - $\forall x \neg \text{Português}(x) \Rightarrow \text{Estrangeiro}(x)$

Generalização da Resolução

- Cláusulas não de Horn:
 - forma genérica: $p_1 \vee p_2 \vee \dots \vee p_n$
 - ou: $p_1 \wedge p_2 \wedge \dots \wedge p_{n1} \Rightarrow q_1 \vee q_2 \vee \dots \vee q_{n2}$
- Generalizando a regra de inferência Resolution:
 - para duas disjunções de qualquer tamanho, se um dos disjuntores numa cláusula unificar com a *negação* de um disjuntor na outra cláusula, então podemos inferir a disjunção de todos os disjuntores excepto estes dois:

$$\begin{array}{c} a \vee h \vee c \\ d \vee \neg h \vee e \end{array} \quad \begin{array}{c} \diagup \\ \diagdown \end{array} \quad a \vee c \vee d \vee e$$

- para frases atómicas p_i e q_i , onde $\text{UNIFY}(p_j, \neg q_k) = \theta$:

$$p_1 \vee \dots \vee p_j \vee \dots \vee p_m$$

$$q_1 \vee \dots \vee q_k \vee \dots \vee q_n$$

$$\text{SUBST}(\theta, p_1 \vee \dots \vee p_{j-1} \vee p_{j+1} \vee \dots \vee p_m \vee q_1 \vee \dots \vee q_{k-1} \vee q_{k+1} \vee \dots \vee q_n)$$

- qualquer frase em lógica de primeira ordem pode ser convertida na forma das premissas da regra da Resolução: **forma normal conjuntiva (CNF)**

Forma Normal Conjuntiva

- Conversão de frases para a CNF:
 - Eliminar implicações
$$p \Rightarrow q \equiv \neg p \vee q$$
 - Mover as negações (\neg) para os átomos:
$$\neg(p \vee q) \equiv \neg p \wedge \neg q$$
$$\neg(\neg p) \equiv p$$
$$\neg\forall x p \equiv \exists x \neg p$$
$$\neg\exists x p \equiv \forall x \neg p$$
 - Estandardizar variáveis
 - $(\forall x P(x)) \vee (\exists x Q(x))$ fica $(\forall x P(x)) \vee (\exists y Q(y))$
 - Mover os quantificadores para a esquerda
 - $p \vee \forall x q$ fica $\forall x p \vee q$ – é possível pois sabemos que p não contém x
 - **Skolemização:** processo de remover quantificadores existenciais
 - Já só há quantificadores universais: ignorá-los
 - Distribuir \wedge por \vee :
$$(a \wedge b) \vee c \text{ fica } (a \vee c) \wedge (b \vee c)$$
- CNF: a frase é uma conjunção onde cada conjuntor é uma disjunção de literais

Skolemização

- Caso mais simples: Existential Elimination
 - $\exists x P(x)$ fica $P(A)$, onde A é uma constante ainda não utilizada
- Se existirem quantificadores universais antes do existencial: substitui-se x por função das variáveis quantificadas universalmente
 - $\forall x \exists y Pessoa(x) \Rightarrow Coração(y) \wedge Tem(x, y)$
 - todas as pessoas têm coração
 - substituindo y por constante:
 - $\forall x Pessoa(x) \Rightarrow Coração(C) \wedge Tem(x, C)$
 - Errado: toda a gente tem o mesmo coração C !
 - $\forall x Pessoa(x) \Rightarrow Coração(F(x)) \wedge Tem(x, F(x))$, onde F é o nome de uma função ainda não utilizada – **função de Skolem**

Provas com Resolução

- **Prova por contradição:** para provar P , assumir que P é falso (adicionar $\neg P$ à KB)
- Exemplo:

$$C1: \neg P(w) \vee Q(w) \equiv P(w) \Rightarrow Q(w)$$

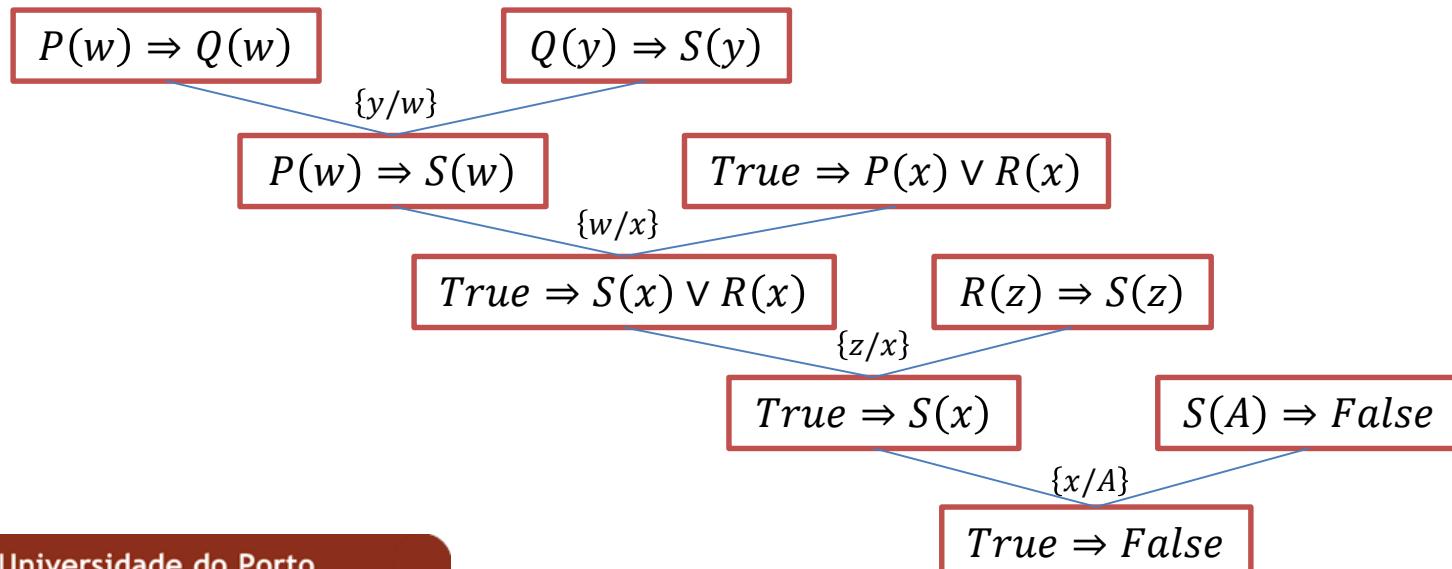
$$C2: P(x) \vee R(x) \equiv \text{True} \Rightarrow P(x) \vee R(x)$$

$$C3: \neg Q(y) \vee S(y) \equiv Q(y) \Rightarrow S(y)$$

$$C4: \neg R(z) \vee S(z) \equiv R(z) \Rightarrow S(z)$$

- Provar $S(A)$:

$$C5: \neg S(A) \equiv S(A) \Rightarrow \text{False}$$



Plano

- Conhecimento
- Lógica proposicional: revisões
- Lógica de 1^a ordem
- Programação em Lógica

Programação em Lógica

Programa = conjunto de axiomas

Computação = prova construtiva de um objectivo a partir do programa

- Interpretação procedural de uma cláusula de Horn (Kowalski)

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

- Leitura **declarativa**: A é verdade se os B_i são verdade
- Leitura **procedimental**: para resolver (executar) A , resolver (executar) $B_1, B_2 \text{ e } \dots \text{ e } B_n$
- Interpretador da linguagem: prova usando o princípio da **resolução**, com o algoritmo de **unificação**

Origens do Prolog

- 1970's: Kowalski
 - Leitura procedural de cláusulas de Horn
- 1970's: Colmerauer (Marselha)
 - Implementação de demonstrador de teoremas chamado Prolog (*Programmation en Logique*)
- 1978/79: Warren (Edimburgo)
 - Primeira implementação eficiente de Prolog
 - *Edinburgh Prolog*: standard *de facto*
- 1981: Projecto Japonês da Quinta Geração de Computadores
 - Construir máquinas capazes de executar directamente Prolog

PROGRAMAÇÃO EM LÓGICA

CONCEITOS

Factos

- Afirmar que uma relação entre objectos é verdadeira **father(abraham, isaac)**.
 - Relação ou **predicado**: **father**
 - Objectos ou indivíduos: **abraham** e **isaac**
 - Interpretação: Abraham é pai de Isaac
 - a interpretação é convencional, mas essencial para a compreensão da formulação do problema!
- Convenções sintácticas:
 - Nomes de predicados e objectos começam com minúscula (átomos)
 - Frases terminam com ponto final.

Programa Simples

- Um conjunto finito de factos é um **programa** em lógica
 - o conjunto de factos descreve uma situação

father(terach, abraham) .	male(terach) .
father(terach, nanchor) .	male(abraham) .
father(terach, haran) .	male(nanchor) .
father(abraham, isaac) .	male(haran) .
father(haran, lot) .	male(isaac) .
father(haran, milcah) .	male(lot) .
father(haran, yiscah) .	
mother(sarah, isaac) .	female(sarah) .
	female(milcah) .
	female(yiscah) .

Perguntas

- Obter informação de um programa em lógica
father(abraham,isaac) ?
 - Sintacticamente semelhante a um facto, mas inclui ‘?’
 - Responder à pergunta: determinar se é uma *consequência lógica* do programa

1^a Regra de Dedução – **Identidade**: de *P* deduzir *P*?

Uma pergunta é uma consequência lógica de um facto idêntico.

- Se há um facto idêntico à pergunta: **yes**
 - Se não há: **no**
- Resposta negativa
- Significa que o facto não é uma consequência lógica do programa
 - Mas não se sabe se é verdade ou não!

father(abraham,isaac) ?
yes
female(abraham) ?
no

Variável Lógica

- Representa um indivíduo não especificado
 - Não é uma posição de memória onde se coloca um valor!
 - Convenção: começa por maiúscula
- Utilização em perguntas:
father(abraham,X) ?
 - Há algum valor para a variável X que torne a pergunta uma consequência lógica do programa?
 - quantificação existencial implícita para as variáveis nas perguntas
 - Resposta: **X=isaac**

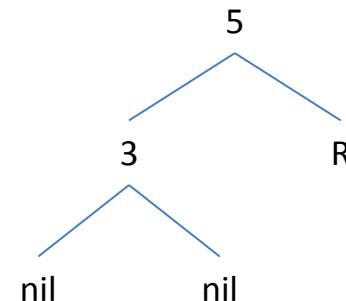
2^a Regra de Dedução – **Generalização**: de $P\theta$ deduzir $P?$

Uma pergunta existencial é uma consequência lógica de uma sua instância.

- O facto **father(abraham,isaac)** implica que existe um X tal que **father(abraham,X)** é verdade

Termos

- Constantes
- Variáveis
- Termos complexos: estruturas
 - Functor
 - Nome (um átomo)
 - Aridade (número de argumentos)
 - Sequência de um ou mais argumentos, que são termos
 - Forma genérica: $f(t_1, t_2, \dots, t_n)$
 - Nome do functor: f
 - Aridade: n
 - Argumentos: t_1, t_2, \dots, t_n
 - Exemplos:
 - $s(0)$
 - $hot(milk)$
 - $name(john, doe)$
 - $list(a, list(b, nil))$
 - $foo(X)$
 - $tree(tree(nil, 3, nil), 5, R)$



Substituição e Instância

- Termos
 - Sem variáveis: *ground* (totalmente instanciado)
 - Com variáveis: *nonground*
- *Substituição* é um conjunto de pares $X_i=t_i$
 - X_i é uma variável
 - t_i é um termo
 - $X_i \neq X_j$, para todo o $i \neq j$
 - X_i não ocorre em t_j , para quaisquer i e j
- Aplicação de substituição θ a termo A : $A\theta$
 - Substituir, em A , cada ocorrência de X por t , para todo o par $X=t$ em θ
 - $A = \text{father(abraham, }X)$, $\theta = \{X=\text{isaac}\}$, $A\theta = \text{father(abraham,isaac)}$
- *Instância*
 - A é uma instância de B se houver uma substituição θ tal que $A=B\theta$
 - $\text{father(abraham,isaac)}$ é uma instância de $\text{father(abraham, }X)$

Soluções

- Uma *solução* de uma pergunta existencial é uma sua *instância*
- Uma pergunta existencial pode ter várias soluções
 - Soluções: {X=lot}, {X=milcah}, {X=yiscah}
- Outro exemplo:

<code>plus(0, 0, 0) .</code>	<code>plus(1, 1, 2) .</code>	<code>plus(0, 3, 3) .</code>
<code>plus(1, 0, 1) .</code>	<code>plus(0, 2, 2) .</code>	<code>plus(1, 3, 4) .</code>
<code>plus(0, 1, 1) .</code>	<code>plus(1, 2, 3) .</code>	<code>...</code>

`plus(X, Y, 4) ?`

- Soluções: {X=0, Y=4}, {X=1, Y=3}, {X=2, Y=2}, {X=3, Y=1}, {X=4, Y=0}

`plus(X, X, 4) ?`

- Soluções: {X=2}

Factos Universais

likes(abraham, pomegranates) .

likes(sarah, pomegranates) .

.

.

.

likes(X, pomegranates) .

- Elemento neutro da adição: **plus(0, X, X)** .
- Toda a gente gosta de si próprio: **likes(X, X)** .
- Variáveis nos factos são quantificadas universalmente
 - Caso geral: **$p(T_1, \dots, T_n)$** .
 - Para todos os X_1, \dots, X_k , que são variáveis que ocorrem no facto, **$p(T_1, \dots, T_n)$** é verdade

3^a Regra de Dedução – *Instanciação*: de **P** deduzir **$P\theta$**

De um facto quantificado universalmente, deduz-se uma sua instância.

- De um facto quantificado universalmente podemos deduzir uma qualquer sua instância
 - de **likes(X, pomegranates)** deduz-se **likes(abraham, pomegranates)**

Perguntas e Factos com Variáveis

- Perguntas sem variáveis

- Procurar um facto do qual a pergunta seja uma instância

`plus(0,X,X).`

`plus(0,2,2)?`

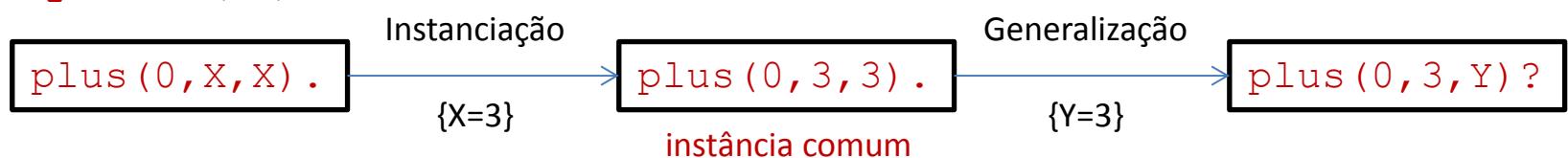
`yes`

- `plus(0,2,2)?` é uma instância de `plus(0,X,X)`.

- Perguntas com variáveis

- *Instância comum*: C é uma instância comum de A e B se houver substituições θ_1 e θ_2 tais que $C=A\theta_1$ é sintaticamente idêntico a $B\theta_2$
 - Responder a pergunta existencial com base em facto universal: Instanciação + Generalização

`plus(0,3,Y)?`



Perguntas Conjuntivas

- Conjunção de objectivos
 - $\text{father}(\text{terach}, \text{X}) , \text{father}(\text{X}, \text{Y}) ?$
 - a vírgula corresponde ao “e” lógico
- *Variáveis partilhadas*
 - Variáveis que ocorrem em dois objectivos diferentes na pergunta
 - $\text{father}(\text{haran}, \text{X}) , \text{male}(\text{X}) ?$
 - Existe um X tal que $\text{father}(\text{haran}, \text{X})$ e $\text{male}(\text{X})$ sejam ambos verdade?
 - O alcance de uma variável partilhada é toda a pergunta conjuntiva
- Uma pergunta conjuntiva A_1, A_2, \dots, A_n é uma consequência lógica de um programa P se cada objectivo A_i for consequência de P , em que as variáveis partilhadas são instanciadas com os mesmos valores em objectivos diferentes
 - i.e., se houver uma mesma substituição θ tal que $A_1\theta$ e $A_2\theta$ e ... e $A_n\theta$ são instâncias de factos em P

Perguntas Conjuntivas (2)

- Utilização: restrição aos valores de uma variável

father(haran, X) , male(X) ?

- soluções para o 1º objectivo são restritas a filhos que são homens
- ou: soluções para o 2º objectivo são restritas a indivíduos cujo pai é haran
- solução:
 - {X=lot}

father(terach, X) , father(X, Y) ?

- só interessam os filhos de terach que são eles próprios pais
- considera indivíduos Y cujos pais são filhos de terach
- soluções:
 - {X=abraham, Y=isaac}
 - {X=haran, Y=lot}
 - {X=haran, Y=milcah}
 - {X=haran, Y=yiscah}

Regras

- Definir novas relações a partir de relações existentes
 $\text{son}(X, Y) \leftarrow \text{father}(Y, X), \text{male}(X).$
 - no fundo, damos um nome a uma pergunta conjuntiva
- Forma genérica das **cláusulas** de Horn: $A \leftarrow B_1, B_2, \dots, B_n.$
 - Cabeça da regra: A
 - Corpo da regra: B_1, B_2, \dots, B_n
 - Casos especiais:
 - Facto ($n=0$): $A.$
 - Pergunta: $\leftarrow B_1, B_2, \dots, B_n.$ ($\equiv B_1, B_2, \dots, B_n?$)
- Leitura procedural
 - Uma regra permite expressar perguntas complexas a partir de perguntas mais simples
 - $\text{son}(X, \text{haran})?$ é traduzido para $\text{father}(\text{haran}, X), \text{male}(X)?$
- Leitura declarativa
 - Uma regra é um axioma lógico
 - “ X é um filho de Y se Y é o pai de X e X é homem”

Quantificação de Variáveis

- Formalmente, todas as variáveis numa cláusula são quantificadas universalmente
 - $\text{grandfather}(X, Y) \leftarrow \text{father}(X, Z), \text{father}(Z, Y).$
 - “Para todo o X, Y e Z , X é o avô de Y se X for o pai de Z e Z for o pai de Y . ”
- Leitura alternativa: variáveis que só ocorram no corpo da cláusula (e não na cabeça) podem ser quantificadas existencialmente
 - “Para todo o X e Y , X é o avô de Y se existir um Z tal que X é o pai de Z e Z é o pai de Y . ”
 - Porque:

$$\forall X \forall Y \forall Z \text{ grandfather}(X, Y) \leftarrow \text{father}(X, Z), \text{father}(Z, Y). \quad \equiv$$

$$\forall X \forall Y \forall Z \text{ grandfather}(X, Y) \vee \neg [\text{father}(X, Z), \text{father}(Z, Y)]. \quad \equiv$$

$$\forall X \forall Y \text{ grandfather}(X, Y) \vee \forall Z \neg [\text{father}(X, Z), \text{father}(Z, Y)]. \quad \equiv$$

$$\forall X \forall Y \text{ grandfather}(X, Y) \vee \exists Z [\text{father}(X, Z), \text{father}(Z, Y)]. \quad \equiv$$

$$\forall X \forall Y \text{ grandfather}(X, Y) \leftarrow \exists Z [\text{father}(X, Z), \text{father}(Z, Y)].$$

Modus Ponens

4^a Regra de Dedução – **Modus Ponens Universal**: de $A \leftarrow B_1, \dots, B_n$ e de $B_1\theta, \dots, B_n\theta$ deduzir $A\theta$

Da instanciação do corpo de uma regra podemos deduzir a instanciação da cabeça.

- Da regra $R = (A \leftarrow B_1, B_2, \dots, B_n)$
 - e dos factos B_1', B_2', \dots, B_n' .
 - pode-se deduzir A' se $A' \leftarrow B_1', B_2', \dots, B_n'$ for uma instância de R
-
- Um objectivo G quantificado existencialmente (i.e., no corpo de uma cláusula) é uma consequência lógica de um programa P se houver uma cláusula em P que tenha uma instância sem variáveis da forma
$$A \leftarrow B_1, B_2, \dots, B_n$$
tal que B_1, B_2, \dots, B_n são consequências lógicas de P , e A é instância de G
 - Um objectivo G é uma consequência lógica de um programa P se e só se G pode ser deduzido de P por um número finito de aplicações da regra Modus Ponens universal

Operacionalização

```
father(terach, abraham).  
father(terach, nanchor).  
father(terach, haran).  
father(abraham, isaac).  
father(haran, lot).  
father(haran, milcah).  
father(haran, yiscah).
```

```
mother(sarah, isaac).
```

son (S, haran) ?

```
male(terach).  
male(abraham).  
male(nanchor).  
male(haran).  
male(isaac).  
male(lot).  
  
female(sarah).  
female(milcah).  
female(yiscah).
```

```
son(X, Y) ← father(Y, X), male(X).
```

{X=lot, Y=haran}

```
son(lot, haran) ← father(haran, lot), male(lot).
```

S=lot

- Caso geral: para resolver um objectivo A com um programa P
 - escolher uma regra $A_1 \leftarrow B_1, B_2, \dots, B_n$ em P
 - obter uma substituição θ tal que $A\theta = A_1\theta$ e cada $B_i\theta$ não tem variáveis
 - resolver recursivamente cada $B_i\theta$

Procedimentos

```
son(X,Y) ← father(Y,X), male(X).  
son(X,Y) ← mother(Y,X), male(X).  
grandparent(X,Y) ← father(X,Z), father(Z,Y).  
grandparent(X,Y) ← father(X,Z), mother(Z,Y).  
grandparent(X,Y) ← mother(X,Z), father(Z,Y).  
grandparent(X,Y) ← mother(X,Z), mother(Z,Y).
```

```
parent(X,Y) ← father(X,Y).  
parent(X,Y) ← mother(X,Y).
```

```
son(X,Y) ← parent(Y,X), male(X).  
grandparent(X,Y) ← parent(X,Z), parent(Z,Y).
```

- *Procedimento*
 - colecção de regras com o mesmo predicado na cabeça
 - segundo uma interpretação operacional destas regras em Prolog, são análogos aos procedimentos ou sub-rotinas das linguagens de programação “convencionais”

Programa em Lógica

Um **programa em lógica** é um
conjunto finito de regras

- O **significado** de um programa em lógica P , $M(P)$, é o conjunto de objectivos totalmente instanciados dedutíveis de P
 - se o programa apenas tem factos (não universais), o seu significado é o próprio programa
- Relativamente a um *significado pretendido* M :
 - Programa **correcto**: $M(P) \subseteq M$
 - Um programa correcto não permite deduzir factos não pretendidos
 - Programa **completo**: $M \subseteq M(P)$
 - Um programa completo permite deduzir tudo o que se pretendia
 - Programa correcto e completo: $M = M(P)$

Bases de Dados

- Uma base de dados em lógica contém:
 - Factos: permitem definir **relações**, como em bases de dados relacionais
 - Regras: permitem definir perguntas relacionais complexas (**vistas**)
- **Esquema** da relação: especifica o papel de cada posição da relação
 - father(Father,Child)
 - mother(Mother,Child)
 - male(Person)
 - female(Person)
- Novas relações criadas a partir destas, usando regras
 - son(Son,Parent)
 - daughter(Daughter,Parent)
 - parent(Parent,Child)
 - grandparent(Grandparent,Grandchild)

Novas Relações

- Tornar explícitas relações implícitas

- `procreated(Man,Woman)`

```
procreated(Man,Woman) ←  
    father(Man,Child), mother(Woman,Child).
```

- `brother(Brother,Sib)`

```
brother(Brother,Sib) ←  
    parent(Parent,Brother), parent(Parent,Sib), male(Brother).
```

- **brother(X,X) ?**

- satisfeito com qualquer indivíduo do sexo masculino que tenha um progenitor definido na BD!

- Necessidade de diferenciar duas variáveis

- `≠(Term1,Term2)`

ou com notação infixa `Term1 ≠ Term2`

<code>abraham ≠ isaac.</code>	<code>abraham ≠ haran.</code>	<code>abraham ≠ lot.</code>
<code>abraham ≠ milcah.</code>	<code>abraham ≠ yiscah.</code>	<code>isaac ≠ haran.</code>
<code>isaac ≠ lot.</code>	<code>isaac ≠ milcah.</code>	<code>isaac ≠ yiscah.</code>
<code>haran ≠ lot.</code>	<code>haran ≠ milcah.</code>	<code>haran ≠ yiscah.</code>
<code>lot ≠ milcah.</code>	<code>lot ≠ yiscah.</code>	<code>milcah ≠ yiscah.</code>

Novas Relações (2)

```
brother(Brother,Sib) ←  
    parent(Parent,Brother),  
    parent(Parent,Sib),  
    male(Brother),  
    Brother ≠ Sib.
```

```
uncle(Uncle,Person) ←  
    brother(Uncle,Parent), parent(Parent,Person).  
sibling(Sib1,Sib2) ←  
    parent(Parent,Sib1), parent(Parent,Sib2), Sib1 ≠ Sib2.  
cousin(Cousin1,Cousin2) ←  
    parent(Parent1,Cousin1),  
    parent(Parent2,Cousin2),  
    sibling(Parent1,Parent2).
```

```
mother(Woman) ← mother(Woman,Child).
```

- mesmo nome, mas aridades diferentes: relações diferentes!

Dados Estruturados e Abstracção

```
course(complexity,monday,9,11,david,harel,feinberg,a).
```

```
course(complexity,time(monday,9,11),lecturer(david,harel),  
location(feinberg,a)).
```

```
lecturer(Lecturer,Course) ←  
    course(Course,Time,Lecturer,Location).  
  
duration(Course,Length) ←  
    course(Course,time(Day,Start,Finish),Lecturer,Location),  
    plus(Start,Length,Finish).  
  
teaches(Lecturer,Day) ←  
    course(Course,time(Day,Start,Finish),Lecturer,Location).  
  
occupied(Room,Day,Time) ←  
    course(Course,time(Day,Start,Finish),Lecturer,Room),  
    Start ≤ Time, Time ≤ Finish.
```

- Boa estruturação de dados
 - regras mais concisas, abstraindo detalhes
 - maior modularidade: alterações na representação dos dados não implicam alterar todo o programa

Representação Alternativa

- Com relações binárias mais simples:

```
day(complexity,monday).  
start_time(complexity,9).  
finish_time(complexity,11).  
lecturer(complexity,harel).  
building(complexity,feinberg).  
room(complexity,a).
```

- Exemplo de regra:

```
teaches(Lecturer,Day) ←  
    lecturer(Course,Lecturer), day(Course,Day).
```

Relações Recursivas

```
grandparent(Ancestor,Descendant) ←  
    parent(Ancestor,Person), parent(Person,Descendant).  
greatgrandparent(Ancestor,Descendant) ←  
    parent(Ancestor,Person), grandparent(Person,Descendant).  
greatgreatgrandparent(Ancestor,Descendant) ←  
    parent(Ancestor,Person), greatgrandparent(Person,  
                                                Descendant).
```

```
ancestor(Ancestor,Descendant) ←  
    parent(Ancestor,Person), ancestor(Person,Descendant).
```

- Necessidade de regra não recursiva:

```
ancestor(Ancestor,Descendant) ←  
    parent(Ancestor,Descendant).  
ancestor(Ancestor,Descendant) ←  
    parent(Ancestor,Person), ancestor(Person,Descendant).
```

Programa Recursivo Linear

- Programa *recursivo linear*

- o corpo da regra recursiva tem apenas um objectivo recursivo

```
ancestor(Anccestor,Descendant) ←  
    parent(Anccestor,Descendant).  
ancestor(Anccestor,Descendant) ←  
    parent(Anccestor,Person), ancestor(Person,Descendant).
```

- Outra versão não recursiva linear, com o mesmo significado

```
ancestor(Anccestor,Descendant) ←  
    parent(Anccestor,Descendant).  
ancestor(Anccestor,Descendant) ←  
    ancestor(Anccestor,Person), ancestor(Person,Descendant).
```

Álgebra Relacional e PL

- União

```
r_union_s(X1, ..., Xn) ← r(X1, ..., Xn).  
r_union_s(X1, ..., Xn) ← s(X1, ..., Xn).
```

- Diferença (requer predicado de negação: **not**)

```
r_diff_s(X1, ..., Xn) ← r(X1, ..., Xn), not s(X1, ..., Xn).
```

- Produto cartesiano

```
r_x_s(X1, ..., Xm, Xm+1, ..., Xm+n) ←  
r(X1, ..., Xm), s(Xm+1, ..., Xm+n).
```

- Projecção

```
r13(X1, X3) ← r(X1, X2, X3).
```

- Selecção

```
r1(X1, X2, X3) ← r(X1, X2, X3), X3 > X2.
```

- Intersecção

```
r_meet_s(X1, ..., Xn) ← r(X1, ..., Xn), s(X1, ..., Xn).
```

- Junção

```
r_join_s(X1, X2, X3) ← r(X1, X2), s(X2, X3).
```

Tipos

- Um tipo é um conjunto (possivelmente infinito) de termos
- Tipos definidos com base em **relações unárias**
 - $p/1$ $\text{male}/1$ $\text{female}/1$
- Tipos mais complexos definidos com predicados recursivos
 - **tipos recursivos**
 - inteiros, listas, árvores binárias, ...

Números Naturais

- 0 é natural
- $s/1$: $s(X)$ é o sucessor de X
 - $s(0), s(s(0)), s(s(s(0))), \dots$
 - $s^n(0)$ representa o número n (i.e., n aplicações de $s/1$)
- Predicado:

```
natural_number(0).  
natural_number(s(X)) ← natural_number(X).
```

- Relação de ordem:

```
0 ≤ X ← natural_number(X).  
s(X) ≤ s(Y) ← X ≤ Y.
```

Adição

- Relação ternária:
 - 2 argumentos da adição, relação de 3

```
plus(0,X,X) ← natural_number(X).  
plus(s(X),Y,s(Z)) ← plus(X,Y,Z).
```

- Significado: o conjunto dos factos **plus(X,Y,Z)** em que **X**, **Y** e **Z** são números naturais e **X+Y=Z**
- Diferentes usos (funcional e relacional):
 - **plus(s(0),s(0),X)?**
 - $1+1=X?$ $\{X=s(s(0))\}$
 - **plus(s(0),s(0),s(s(0)))?**
 - $1+1=2?$
 - **plus(s(0),X,s(s(s(0))))?**
 - $1+X=3?$ $\{X=s(s(0))\}$

Soluções Múltiplas

```
plus(0,X,X) ← natural_number(X).  
plus(s(X),Y,s(Z)) ← plus(X,Y,Z).
```

- **plus(X, Y, s(s(s(0)))) ?**

{X=0, Y=s(s(s(0)))}

– plus(X1, Y, s(s(0))) ? {X=s(X1)}

{X1=0, Y=s(s(0))}

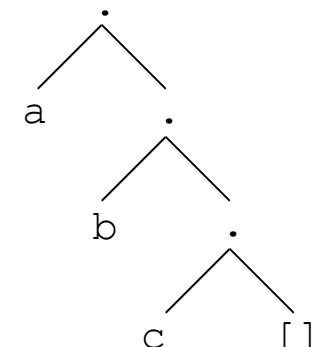
• plus(X2, Y, s(0)) ? {X1=s(X2)}

{X2=0, Y=s(0)}

Listas

- Uma **lista** é uma estrutura de dados binária: $.(X, Y)$
 - 1º argumento (**cabeça**): elemento; 2º argumento (**cauda**): resto da lista
 - Símbolo constante para fim da recursão: lista vazia (**nil** ou **[]**)
 - Sintaxe alternativa: $.(X, Y) \equiv [X|Y]$

Formal object	Cons pair syntax	Element syntax
$.(a,[\])$	$[a [\]]$	$[a]$
$.(a,.(b,[\]))$	$[a [b [\]]]$	$[a,b]$
$.(a,.(b,.(c,[\])))$	$[a [b [c [\]]]]$	$[a,b,c]$
$.(a,X)$	$[a X]$	$[a X]$
$.(a,.(b,X))$	$[a [b X]]$	$[a,b X]$



Elementos de uma Lista

- Podem ser quaisquer temos
 - incluindo listas!
- Lista com listas como elementos:

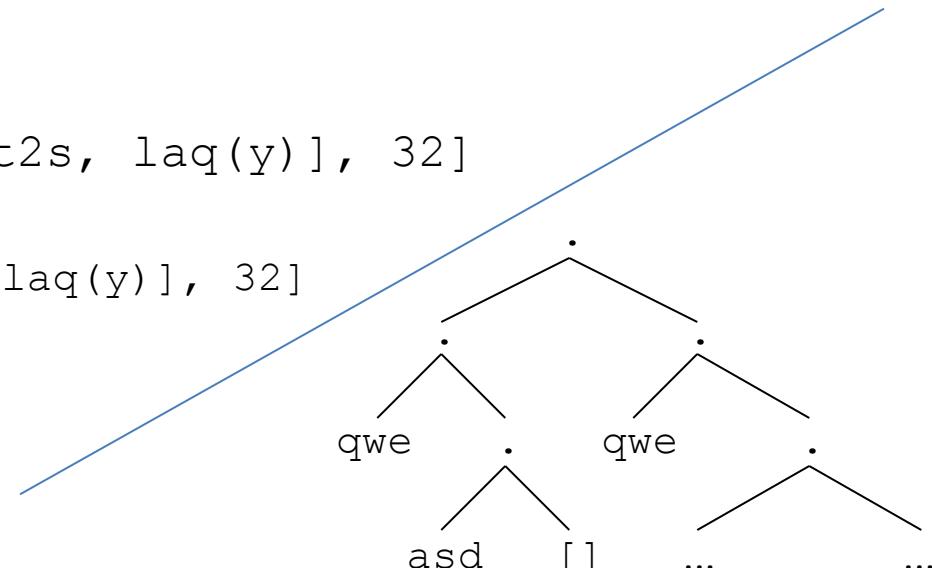
[[qwe, asd], qwe, [asd], [], [t2s, laq(y)], 32]

- cabeça: [qwe, asd]
 - cabeça: qwe
 - cauda: [asd]
 - cabeça: asd
 - cauda: []

– cauda: [qwe, [asd], [], [t2s, laq(y)], 32]

- cabeça: qwe
- cauda: [[asd], [], [t2s, laq(y)], 32]

– ...



Definição de Lista

```
list([ ]).  
list([X|Xs]) ← list(Xs) .
```

list([a,b,c])?

```
list([a|[b,c]]) ← list([b,c]) .  
list([b|[c]]) ← list([c]) .  
list([c|[]]) ← list([]) .  
list([]) .
```

yes

list([X|Xs])?

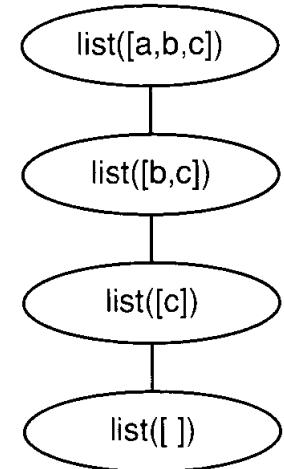
{Xs=[]}, {Xs=[X1]}, {Xs=[X1,X2]}, {Xs=[X1,X2,X3]}, ...

list([X,a])?

yes

list([X|a])?

no



Membro de uma Lista

```
member(X, [X|Xs]).  
member(X, [Y|Ys]) ← member(X, Ys).
```

- Leitura declarativa:
 - X é um elemento de uma lista se for a cabeça da lista ou se for um membro da cauda da lista
 - o significado do programa é o conjunto das instâncias sem variáveis do tipo **member(X,Xs)**, em que X é um elemento de Xs
- **member (b , [a , b , c]) ?**
 - verificar se um elemento é membro da lista
 - yes
- **member (X , [a , b , c]) ?**
 - obter um elemento da lista
 - { $X=a$ }, { $X=b$ }, { $X=c$ }
- **member (b , Xs) ?**
 - obter uma lista que contém um elemento
 - { $Xs=[b|Ys]$ }, { $Xs=[Y1,b|Ys]$ }, { $Xs=[Y1,Y2,b|Ys]$ }, ...

Prefixo, Sufixo, Sublista

```
prefix([], Ys).  
prefix([X|Xs], [X|Ys]) ← prefix(Xs, Ys).
```

prefix([a,b],[a,b,c])?

yes

suffix([b,c],[a,b,c])?

yes

```
suffix(Xs, Xs).  
suffix(Xs, [Y|Ys]) ← suffix(Xs, Ys).
```

```
sublist(Xs, Ys) ← prefix(Ps, Ys), suffix(Xs, Ps).
```

```
sublist(Xs, Ys) ← prefix(Xs, Ss), suffix(Ss, Ys).
```

sublist([b,c],[a,b,c,d])?

yes

```
sublist(Xs, Ys) ← prefix(Xs, Ys).  
sublist(Xs, [Y|Ys]) ← sublist(Xs, Ys).
```

Concatenação de Listas

```
append([ ], Ys, Ys).  
append([X|Xs], Ys, [X|Zs]) ← append(Xs, Ys, Zs).
```

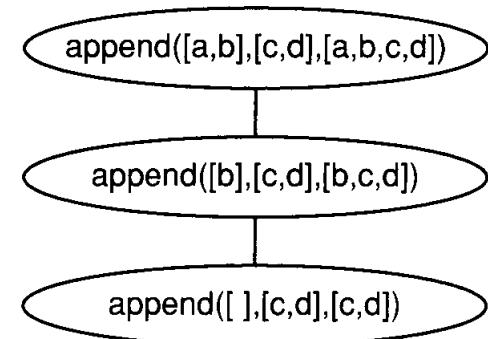
append([a,b],[c,d],[a,b,c,d])?
yes

append([a,b,c],[d,e],Xs)?
{Xs=[a,b,c,d,e]}

append(Xs,[c,d],[a,b,c,d])?
{Xs=[a,b]}

append([a,b],Xs,[a,b,c,d])?
{Xs=[c,d]}

append(As,Bs,[a,b,c,d])?
{As=[], Bs=[a,b,c,d]}, {As=[a], Bs=[b,c,d]}, {As=[a,b], Bs=[c,d]}, {As=[a,b,c], Bs=[d]},
{As=[a,b,c,d], Bs=[]}



Explorando o append

```
prefix(Xs, Ys) ← append(Xs, As, Ys).
```

```
suffix(Xs, Ys) ← append(As, Xs, Ys).
```

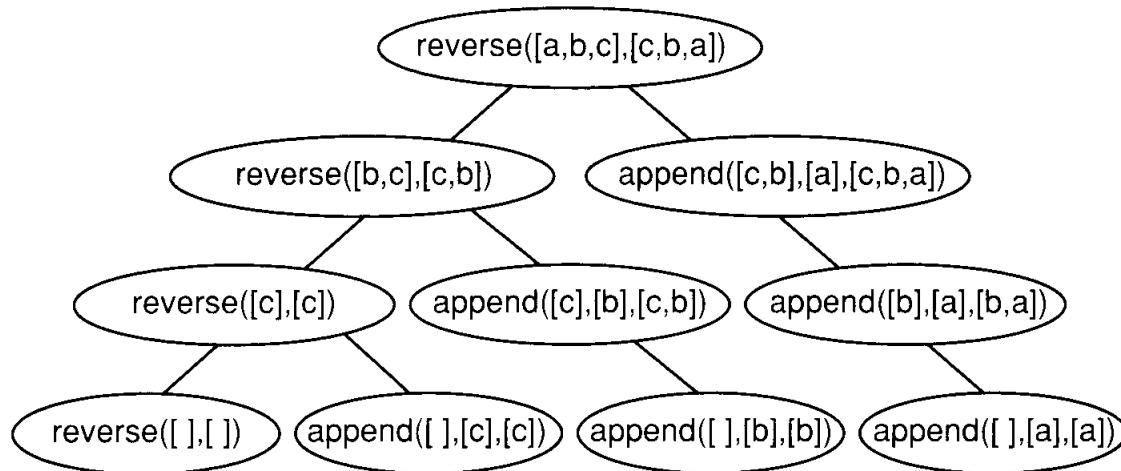
```
sublist(Xs, AsXsBs) ←  
    append(As, XsBs, AsXsBs), append(Xs, Bs, XsBs).
```

```
sublist(Xs, AsXsBs) ←  
    append(AsXs, Bs, AsXsBs), append(As, Xs, AsXs).
```

```
member(X, Ys) ← append(As, [X|Xs], Ys).
```

Inversão de uma Lista

```
reverse([ ], [ ]).  
reverse([X|Xs], Zs) ← reverse(Xs, Ys), append(Ys, [X], Zs).
```



- tamanho da árvore de prova é quadrático no número de elementos da lista a inverter

Inversão de uma Lista (2)

- Evitar append:

```
reverse(Xs, Ys) ← reverse(Xs, [ ], Ys).  
reverse([X|Xs], Acc, Ys) ← reverse(Xs, [X|Acc], Ys).  
reverse([ ], Ys, Ys).
```

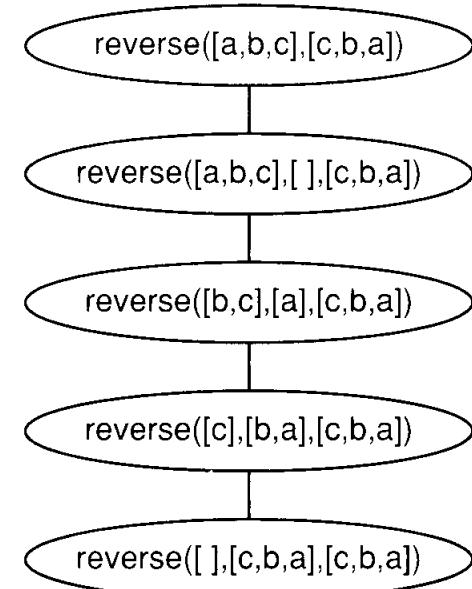
- tamanho da árvore de prova é linear no número de elementos da lista

reverse([a,b,c],Ys)?

```
reverse([a,b,c],[],Ys)  
reverse([b,c],[a],Ys)  
reverse([c],[b,a],Ys)  
reverse,[],[c,b,a],Ys)
```

{Ys=[c,b,a]}

- acumulador permite obter o resultado no último passo



Comprimento de uma Lista

```
length([ ],0).  
length([X|Xs],s(N)) ← length(Xs,N).
```

length([a,b],s(s(0)))?

yes

length([a,b],X)?

{X=s(s(0))}

length(Xs,s(s(0)))?

{Xs=[X1,X2]}

Eliminar Elementos

```
delete([X|Xs], X, Ys) ← delete(Xs, X, Ys).  
delete([X|Xs], Z, [X|Ys]) ← X ≠ Z, delete(Xs, Z, Ys).  
delete([], X, []).
```

delete([a,b,c,b], b, X) ?

{X=[a,c]}

- Omitindo $X \neq Z$:
 - podem ser eliminadas qualquer número de ocorrências
 - fazem parte do significado do programa (são dedutíveis a partir dele):

```
delete([a,b,c,b], b, [a,c])  
delete([a,b,c,b], b, [a,c,b])  
delete([a,b,c,b], b, [a,b,c])  
delete([a,b,c,b], b, [a,b,c,b])
```

Seleccionar Elemento

```
select(X, [X|Xs], Xs).  
select(X, [Y|Ys], [Y|Zs]) ← select(X, Ys, Zs).
```

select(a, [a,b,a,c], Ys) ?

{Ys=[b,a,c]}, {Ys=[a,b,c]}

select(X, [a,b,c], Ys) ?

{X=a, Ys=[b,c]}, {X=b, Ys=[a,c]}, {X=c, Ys=[a,b]}

select(d, [a,b,c], [a,b,c]) ?

no

delete([a,b,c], d, [a,b,c]) ?

yes

Permutações

```
permutation(Xs, [Z|Zs]) ← select(Z, Xs, Ys), permutation(Ys, Zs).  
permutation([], []).
```

permutation([a,b,c], Ys) ?

{Ys=[a,b,c]}, {Y=[a,c,b]}, {Y=[b,a,c]}, {Y=[b,c,a]}, {Y=[c,a,b]}, {Y=[c,b,a]}

- Ordenação do tipo “gerar e testar”:

```
sort(Xs, Ys) ← permutation(Xs, Ys), ordered(Ys).
```

```
ordered([]).
```

```
ordered([X]).
```

```
ordered([X, Y|Ys]) ← X ≤ Y, ordered([Y|Ys]).
```

- não é um bom método
- abordagens com uma estratégia do tipo “dividir e conquistar” são melhores

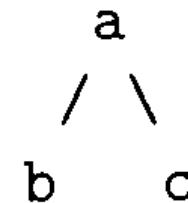
Quicksort

```
quicksort([X|Xs],Ys) ←  
    partition(Xs,X,Littles,Bigs),  
    quicksort(Littles,Ls),  
    quicksort(Bigs,Bs),  
    append(Ls,[X|Bs],Ys).  
quicksort([],[]).
```

```
partition([X|Xs],Y,[X|Ls],Bs) ← X ≤ Y, partition(Xs,Y,Ls,Bs).  
partition([X|Xs],Y,Ls,[X|Bs]) ← X > Y, partition(Xs,Y,Ls,Bs).  
partition([],Y,[],[]).
```

Árvores Binárias

- Tipo de dados recursivo
 - `tree(Element,Left,Right)`
 - árvore vazia: `void`

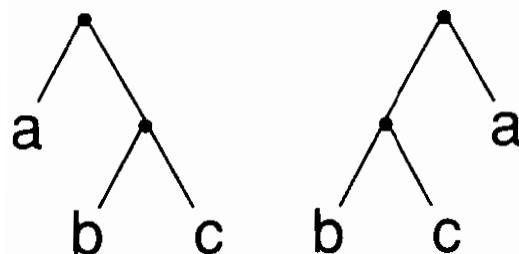


`tree(a,tree(b,void,void),tree(c,void,void)).`

```
binary_tree(void).  
binary_tree(tree(Element,Left,Right)) ←  
    binary_tree(Left), binary_tree(Right).
```

```
tree_member(X,tree(X,Left,Right)).  
tree_member(X,tree(Y,Left,Right)) ← tree_member(X,Left).  
tree_member(X,tree(Y,Left,Right)) ← tree_member(X,Right).
```

Isomorfismo



```
isotree(void,void).  
isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) ←  
    isotree(Left1,Left2), isotree(Right1,Right2).  
isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) ←  
    isotree(Left1,Right2), isotree(Right1,Left2).
```

Travessias

```
preorder(tree(X,L,R) ,Xs) ←  
    preorder(L,Ls) , preorder(R,Rs) , append([X|Ls] ,Rs ,Xs) .  
preorder(void,[ ]).
```

```
inorder(tree(X,L,R) ,Xs) ←  
    inorder(L,Ls) , inorder(R,Rs) , append(Ls ,[X|Rs] ,Xs) .  
inorder(void,[ ]).
```

```
postorder(tree(X,L,R) ,Xs) ←  
    postorder(L,Ls) ,  
    postorder(R,Rs) ,  
    append(Rs , [X] ,Rs1) ,  
    append(Ls ,Rs1 ,Xs) .  
postorder(void,[ ]).
```

Manipulação de Expressões Simbólicas

- Como as expressões são mantidas como tal, torna-se fácil construir programas de manipulação de expressões
- Cálculo de derivadas:

```
derivative(N,0) :- natural_number(N) .  
  
derivative(x,s(0)) .  
  
derivative(F+G,DF+DG) :- derivative(F,DF) , derivative(G,DG) .  
derivative(F-G,DF-DG) :- derivative(F,DF) , derivative(G,DG) .  
derivative(F*G,F*D G+D F*G) :- derivative(F,DF) , derivative(G,DG) .  
derivative(s(0)/F,-DF/(F*F)) :- derivative(F,DF) .  
derivative(F/G,(G*DF-F*D G)/(G*G)) :- derivative(F,DF) ,  
derivative(G,DG) .  
  
derivative(x^s(N),s(N)*x^N) .  
derivative(F^s(N),s(N)*(F^N)*DF) :- derivative(F,DF) .
```

derivative(x*x,D)?

{D = x*s(0)+s(0)*x}

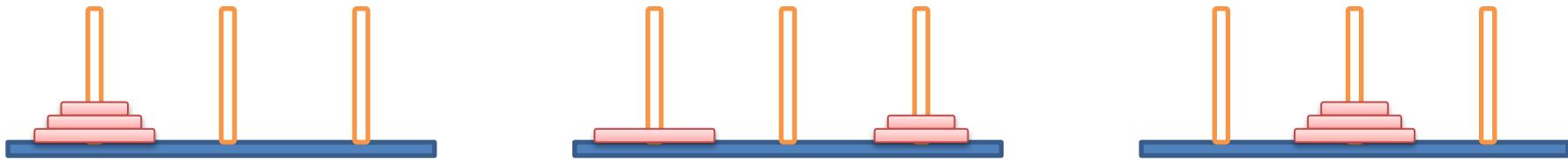
Manipulação de Expressões Simbólicas

- Consistência e invalidade de fórmulas Booleanas
 - Frase **consistente**: tem uma instância verdadeira
 - Frase **inválida**: tem uma instância falsa

```
satisfiable(true).  
satisfiable(X ∧ Y) ← satisfiable(X), satisfiable(Y).  
satisfiable(X ∨ Y) ← satisfiable(X).  
satisfiable(X ∨ Y) ← satisfiable(Y).  
satisfiable(~X) ← invalid(X).
```

```
invalid(false).  
invalid(X ∨ Y) ← invalid(X), invalid(Y).  
invalid(X ∧ Y) ← invalid(X).  
invalid(X ∧ Y) ← invalid(Y).  
invalid(~Y) ← satisfiable(Y).
```

Torres de Hanói



```
hanoi(s(0),A,B,C,[A to B]).  
hanoi(s(N),A,B,C,Moves) ←  
    hanoi(N,A,C,B,Ms1),  
    hanoi(N,C,B,A,Ms2),  
    append(Ms1, [A to B|Ms2], Moves).
```

Execução de um Programa em Lógica

- **Resolvente**
 - Uma conjunção de objectivos num passo de computação
- **Traçado**
 - Sequência de resolventes produzidos ao longo da computação
- **Redução**
 - Substituição, na resolvente, de um objectivo G pelo corpo de uma cláusula cuja cabeça **unifica** com G
 - Ao corpo da cláusula escolhida é aplicado o unificador
- **Computação**
 - Escolha sucessiva de um objectivo da resolvente e sua redução
 - Se a resolvente ficar vazia, a computação termina com sucesso

Unificação

- Um termo t é uma **instância comum** de dois termos t_1 e t_2 se existirem substituições θ_1 e θ_2 tais que $t=t_1\theta_1$ e $t=t_2\theta_2$
- Um **unificador** de dois termos é uma **substituição** que os torna idênticos
 - Portanto, um unificador encontra uma **instância comum**
- O **unificador mais geral (MGU)** de dois termos encontra a instância comum mais geral

Algoritmo da Unificação

- Input: Two terms T_1 and T_2
- Output: θ (the mgu of T_1 and T_2), or *failure*
- Algorithm:

```
initialize  $\theta$  to empty
push  $T_1=T_2$  into the stack
while stack is not empty do
    pop  $X=Y$  from the stack
    case
         $X$  is a variable that does not occur in  $Y$ :
            substitute  $Y$  for  $X$  in the stack and in  $\theta$ 
            add  $X=Y$  to  $\theta$ 
         $Y$  is a variable that does not occur in  $X$ :
            ...
         $X$  and  $Y$  are identical constants or variables:
            continue
         $X$  is  $f(X_1, \dots, X_n)$  and  $Y$  is  $f(Y_1, \dots, Y_n)$ , for some functor  $f$ 
            push  $X_i=Y_i, i=1\dots n$ , on the stack
        otherwise:
            return failure
return  $\theta$ 
```

The diagram shows a grey rectangular box containing the pseudocode. Above the box, a blue line connects the word "failure" in the "otherwise:" case to an oval labeled "teste de ocorrência (e.g. X não unifica com $s(X)$)". Another blue line connects the "failure" text back to the "stack" in the "case" section of the pseudocode.

Uma visão mais pragmática...

- Condições de unificação:
 - variável com variável: unificam sempre
 - quando uma delas for instanciada, passam a ter o mesmo valor (comportam-se como se fossem a mesma variável)
 - atómico com atómico: unificam se os valores forem iguais
 - atómico ou estrutura com variável: unificam sempre
 - a variável fica instanciada com o valor do atómico ou da estrutura
 - por razões pragmáticas, o teste de ocorrência é normalmente omitido
 - estrutura com estrutura: unificam se os functores são iguais, o número de argumentos é igual e os argumentos unificam dois a dois
 - o resultado é a unificação dos argumentos

Interpretador Abstracto

- Input: A goal G and a program P
- Output: An instance of G that is a logical consequence of P , or *no* otherwise
- Algorithm:

```
initialize the resolvent to  $G$ 
while the resolvent is not empty do
    choose a goal  $A$  from the resolvent
    choose a (renamed) clause  $A' \leftarrow B_1, \dots, B_n$  from  $P$  such that  $A$  and  $A'$  unify with mgu  $\theta$ 
        (if no such clause exists, return no)
    replace  $A$  by  $B_1, \dots, B_n$  in the resolvent
    apply  $\theta$  to the resolvent and to  $G$ 
return  $G$ 
```

Traçado

```
father(abraham,isaac).  
father(haran,lot).  
father(haran,milcah).  
father(haran,yiscah).
```

```
male(isaac).  
male(lot).  
female(milcah).  
female(yiscah).
```

```
son(X,Y) ← father(Y,X), male(X).  
daughter(X,Y) ← father(Y,X), female(X).
```

- $\text{son}(S, \text{haran})$?

Resolvente: $\text{son}(S, \text{haran})$

Escolhe $\text{son}(S, \text{haran})$

Escolhe $\text{son}(S, \text{haran}) \leftarrow \text{father}(\text{haran}, S), \text{male}(S)$.

{X=S, Y=haran}

Resolvente: $\text{father}(\text{haran}, S), \text{male}(S)$

Escolhe $\text{father}(\text{haran}, S)$

Escolhe $\text{father}(\text{haran}, \text{lot})$.

Resolvente: $\text{male}(\text{lot})$

Escolhe $\text{male}(\text{lot})$

Escolhe $\text{male}(\text{lot})$.

Resolvente vazia

{S=lot}

Escolhe $\text{male}(S)$

Escolhe $\text{male}(\text{lot})$.

{S=lot}

Resolvente: $\text{father}(\text{haran}, \text{lot})$

Escolhe $\text{father}(\text{haran}, \text{lot})$

Escolhe $\text{father}(\text{haran}, \text{lot})$.

Resolvente vazia

Escolhas

- **Escolha do objectivo**
 - é arbitrária
 - todos os objectivos da resolvente têm que ser reduzidos
 - a ordem de satisfação dos objectivos é indiferente
- **Escolha da cláusula**
 - é crítica
 - a escolha é feita de forma *não determinística*: nem todas as escolhas levam a uma computação bem sucedida
 - se em cada passo só houver uma cláusula para reduzir cada objectivo, então a computação é *determinística*

...

Resolvente: **father(haran, S), male(S)**

Escolhe **father(haran, S)**

Escolhe **father(haran, yiscah)**.

{S=yiscah}

Resolvente: **male(yiscah)**

Escolhe **male(yiscah)**

falha ==> não há cláusula cuja cabeça unifique com male(yiscah)

Computação Infinita

- Programas recursivos
 - podem dar origem a computações que não terminam

```
append( [ ] , Ys , Ys ).
```

```
append( [X|Xs] , Ys , [X|Zs] ) ← append(Xs , Ys , Zs) .
```

- se escolhermos sempre a segunda cláusula:

```
append(Xs,[c,d],Ys)
```

```
Xs=[X|Xs1], Ys=[X|Ys1]
```

```
append(Xs1,[c,d],Ys1)
```

```
Xs1=[X1|Xs2], Ys1=[X1|Ys2]
```

```
append(Xs2,[c,d],Ys2)
```

```
Xs2=[X2|Xs3], Ys2=[X2|Ys3]
```

```
append(Xs3,[c,d],Ys3)
```

```
Xs3=[X3|Xs4], Ys3=[X3|Ys4]
```

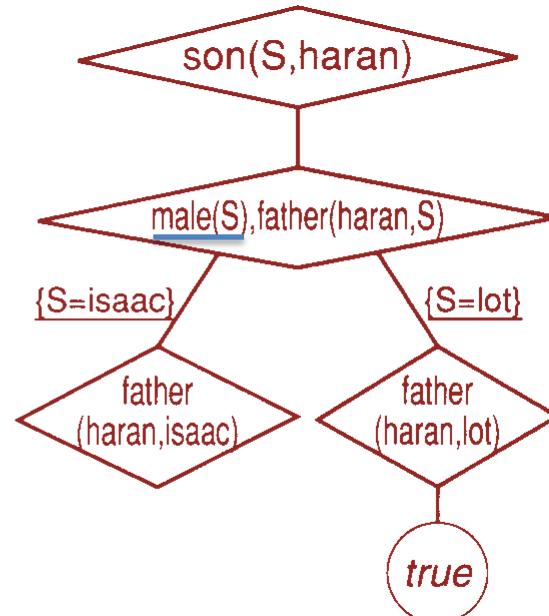
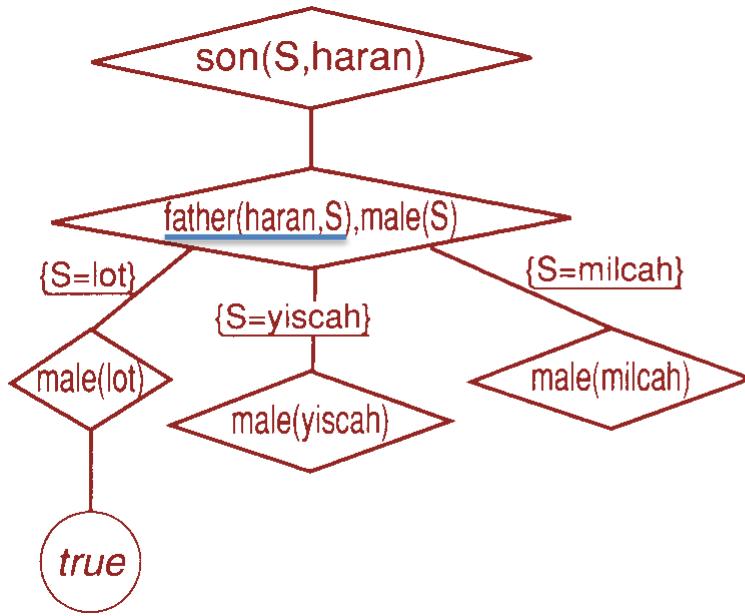
```
:
```

```
:
```

Árvores de Pesquisa

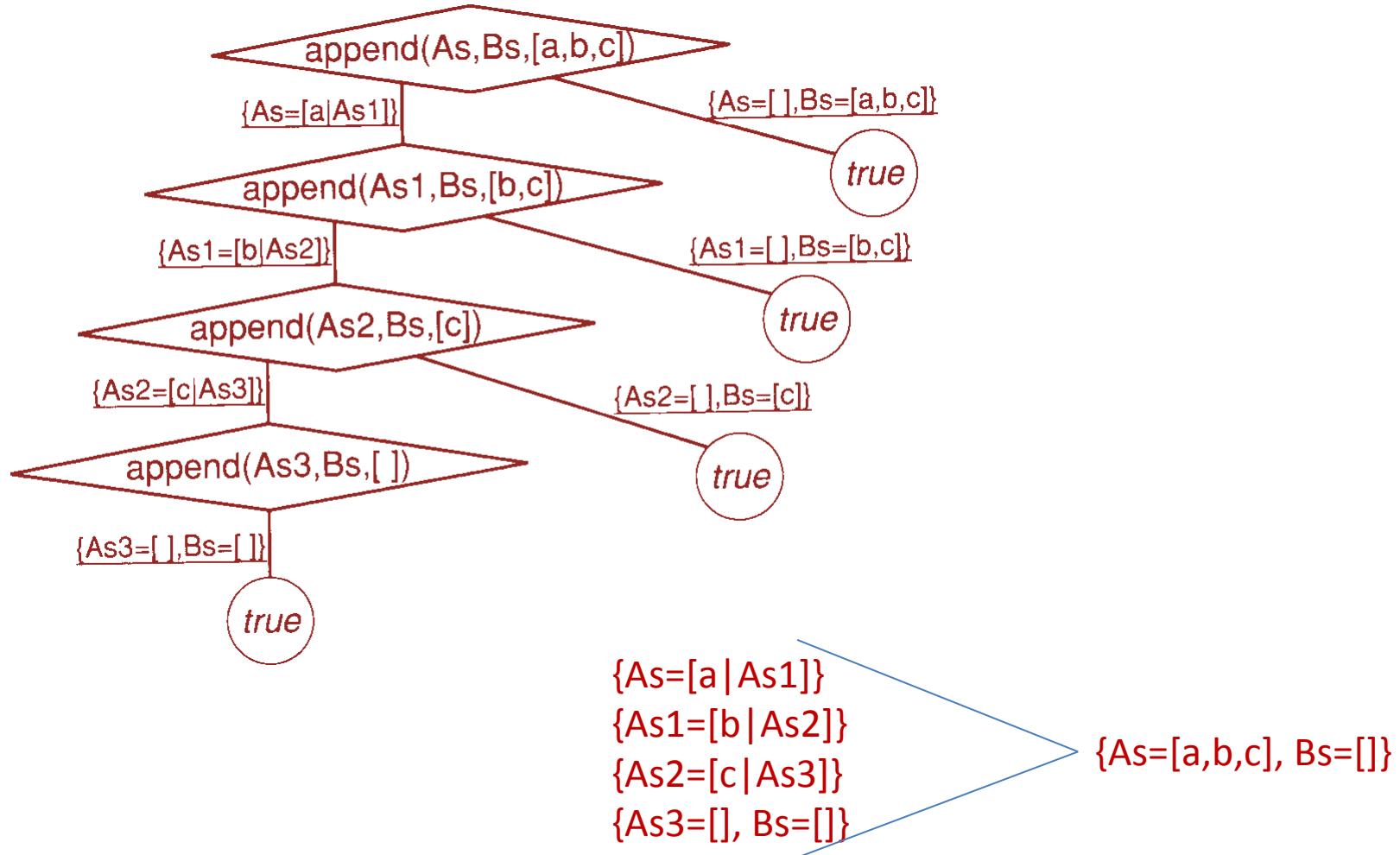
- Árvore de pesquisa de um objectivo G em relação a um programa P
 - Raíz: G
 - Nós: resolventes, com um objectivo seleccionado
 - Ligações a sair de um nó: uma para cada cláusula em P cuja cabeça unifica com o objectivo seleccionado
 - Folhas: nós indicando *sucesso* (resolvente vazia) ou *falha* (objectivo seleccionado não pode ser reduzido)
- Pode haver várias árvores de pesquisa para um objectivo em relação a um programa
 - dependendo do objectivo seleccionado em cada resolvente
 - mas o número de nós de sucesso é o mesmo em todas as árvores
- **Árvore de pesquisa** porque um *interpretador concreto* terá uma estratégia para percorrer a árvore em busca de soluções
 - pesquisa em profundidade, em largura, em paralelo, ...

Duas Árvores de Pesquisa



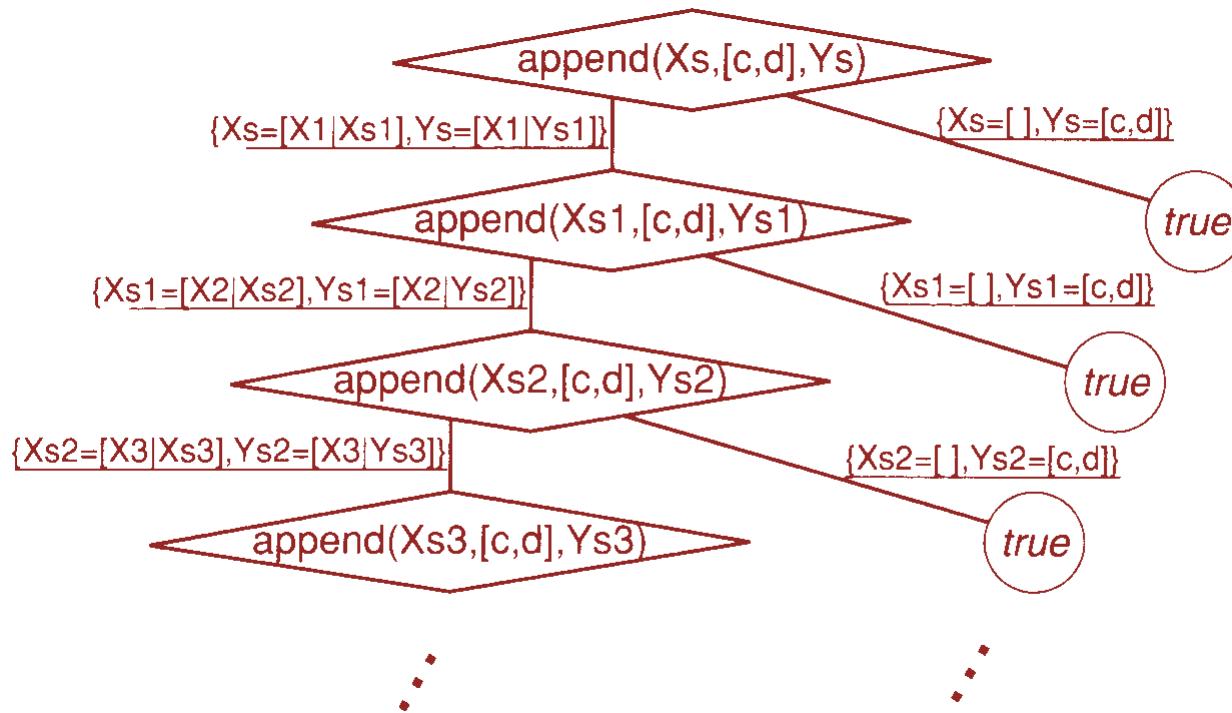
- Convenções:
 - o objectivo seleccionado da resolvente é o da esquerda
 - as ligações são etiquetadas com as substituições aplicadas às variáveis (resultado da unificação)

Várias Soluções



Árvores de Pesquisa Infinitas

- Correspondem a computações infinitas



Negação em PL

- Programas em lógica descrevem o que é verdade
 - factos falsos são omitidos
- Como incluir condições negativas?
 $bachelor(X) \leftarrow male(X), \text{not married}(X).$
- Semântica de **not G**: negação por falha (*negation as failure*)
 - **not G** é uma consequência lógica de um programa **P** se **G** não for uma consequência de **P**
 - assunção de mundo fechado (*closed world assumption*)
- Pensando em árvores de pesquisa:
 - árvore de pesquisa **finitamente falhada**: sem nós de sucesso nem ramos infinitos
 - **conjunto de falhas finitas**: conjunto de objectivos **G** tais que **G** tem uma árvore de pesquisa finitamente falhada
 - **negação por falha**: um objectivo **not G** é uma consequência de **P** se **G** estiver no conjunto de falhas finitas de **P**
- Definir relações com base na negação:
 $disjoint(Xs, Ys) \leftarrow \text{not } (\text{member}(X, Xs), \text{member}(X, Ys)).$

PROGRAMAÇÃO EM LÓGICA

PROLOG



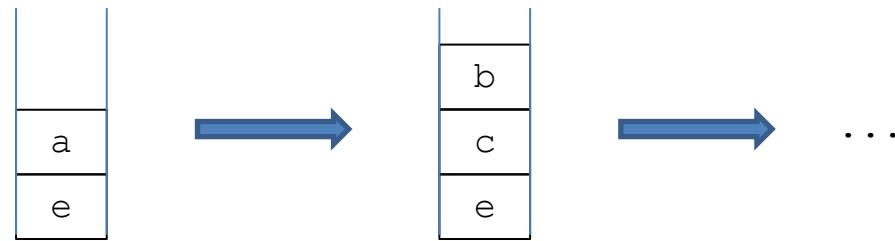
Modelo de Execução

- Implementação do interpretador abstracto numa linguagem de programação concreta
 - 2 decisões para concretizar
 - a *escolha arbitrária* do objectivo da resolvente a reduzir
 - a *escolha não determinística* da cláusula do programa para efectuar a redução
- Prolog
 - Execução sequencial, da esquerda para a direita, dos objectivos da resolvente
 - Pesquisa **sequencial** de uma cláusula unificável e **retrocesso (*backtracking*)**

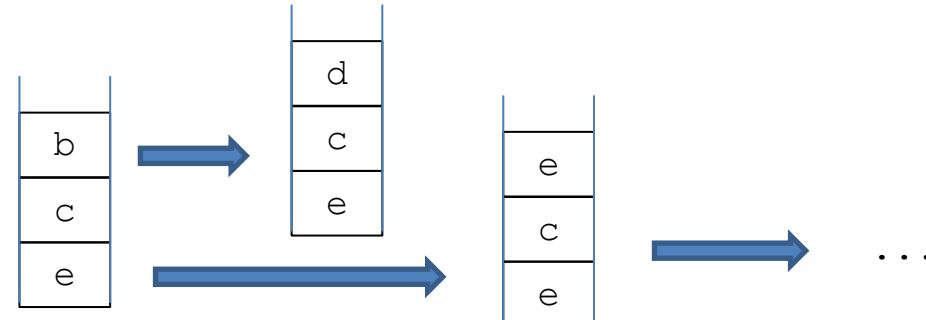
Modelo de Execução (2)

- Resolvente como uma pilha

```
a :- b, c.  
b :- d.  
b :- e.  
c.  
e.  
  
?- a, e.
```



- Pesquisa sequencial e retrocesso
 - Escolhe a primeira cláusula cuja cabeça unifica com o objectivo
 - Se não houver, a computação é desfeita até à última escolha (**ponto de escolha**), e é escolhida a cláusula unificável seguinte



Computação de um Objectivo

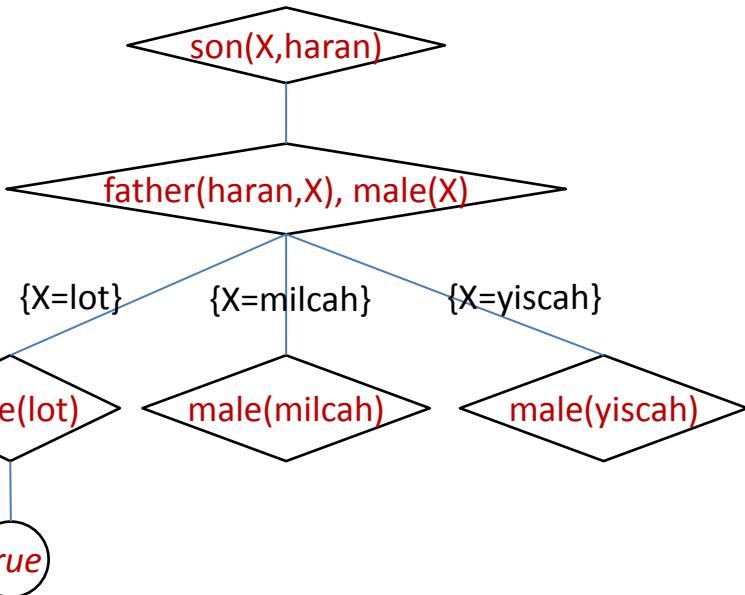
- A computação de um objectivo **G** em relação a um programa em Prolog **P** consiste em gerar todas as soluções de **G**
- Uma computação Prolog de um objectivo **G** é uma travessia completa em **profundidade primeiro** (*depth-first*) da árvore de pesquisa de **G** obtida escolhendo sempre o objectivo mais à esquerda
- A maior parte das implementações de Prolog:
 - pesquisa a árvore até encontrar a primeira solução
 - permite ao utilizador indicar que quer mais soluções através do símbolo **;** (ponto e vírgula)

Alternativa: Paralelismo

- Pesquisa em profundidade não é completa
 - pode não encontrar uma solução (ramo infinito na árvore de pesquisa)
- Pesquisa em largura
 - explorar todas as escolhas possíveis **em paralelo**
 - é completa: encontra sempre uma solução, se existir
 - PARLOG, Concurrent Prolog, GHC, ...
- Paralelismo “ou”
 - percorrer em paralelo todos os ramos da árvore de pesquisa
- Paralelismo “e”
 - executar em paralelo todos os objectivos da resolvente

Traçado

- Escolha determinística pode conduzir a falhanço e retrocesso
 - f**: falhanço (não há cláusulas cuja cabeça unifique com o objectivo)
 - objectivo a seguir a **f** é onde a computação prossegue ao retroceder
 - “;” indica continuação da computação para procurar mais soluções



```

father(abraham,isaac).   male(isaac).
father(haran,lot).        male(lot).
father(haran,milcah).     female(yiscah).
father(haran,yiscah).     female(milcah).

son(X,Y) ← father(Y,X), male(X).
daughter(X,Y) ← father(Y,X), female(X).
  
```

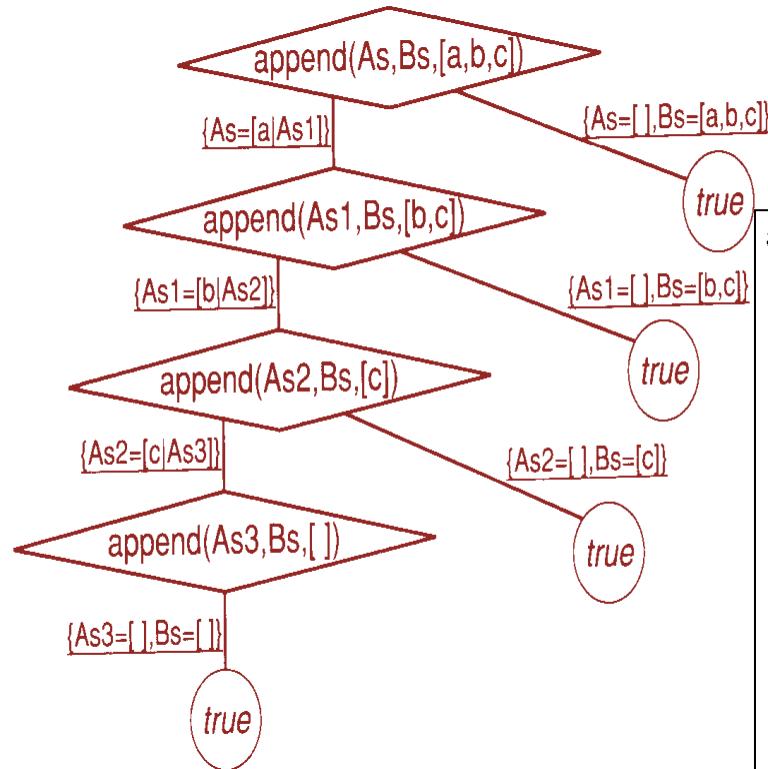
```

son(X,haran)?
  father(haran,X)
  male(lot)
    true
    Output: X=lot
    ;
    X=lot

    father(haran,X)
    male(milcah)      f
    father(haran,X)
    male(yiscah)      f
  X=milcah
  X=yiscah

no (more) solutions
  
```

Traçado (2)



```

append([X|Xs], Ys, [X|Zs]) ← append(Xs, Ys, Zs).
append([ ], Ys, Ys).
  
```

$\text{append}(Xs, Ys, [a, b, c]) \quad Xs = [a | Xs1]$
 $\text{append}(Xs1, Ys, [b, c]) \quad Xs1 = [b | Xs2]$
 $\text{append}(Xs2, Ys, [c]) \quad Xs2 = [c | Xs3]$
 $\text{append}(Xs3, Ys, []) \quad Xs3 = [] , Ys = []$
true
Output: ($Xs = [a, b, c]$, $Ys = []$)
;
 $\text{append}(Xs2, Ys, [c]) \quad Xs2 = [] , Ys = [c]$
true
Output: ($Xs = [a, b]$, $Ys = [c]$)
;
 $\text{append}(Xs1, Ys, [b, c]) \quad Xs1 = [] , Ys = [b, c]$
true
Output: ($Xs = [a]$, $Ys = [b, c]$)
;
 $\text{append}(Xs, Ys, [a, b, c]) \quad Xs = [] , Ys = [a, b, c]$
true
Output: ($Xs = []$, $Ys = [a, b, c]$)
;
no (more) solutions

Programação em Prolog

- A programação em lógica deveria permitir uma programação de alto nível
 - escrever axiomas definindo relações (**lógica**), ignorando a sua utilização pelo mecanismo de execução (**controlo**)
- Contudo: as escolhas do mecanismo de execução não podem ser ignoradas!
 - o modelo de execução do Prolog deve ser tido em conta
 - não basta uma axiomatização correcta e completa
 - eficiência e terminação (computação finita)

Ordem das Cláusulas

- Mudar a ordem das cláusulas de um procedimento corresponde a trocar os ramos da árvore de pesquisa construída para um objectivo
- Trocar os ramos provoca uma travessia da árvore por uma ordem diferente, e consequentemente uma ordem diferente nas soluções encontradas

parent(terach, abraham).	parent(abraham, isaac).
parent(isaac, jacob).	parent(jacob, benjamin).

```
ancestor(X,Y) ← parent(X,Y).  
ancestor(X,Z) ← parent(X,Y), ancestor(Y,Z).
```

```
?- ancestor(terach,X).  
X=abraham ;  
X=isaac ;  
X=jacob ;  
X=benjamin ;  
no
```

```
ancestor(X,Z) ← parent(X,Y), ancestor(Y,Z).  
ancestor(X,Y) ← parent(X,Y).
```

```
?- ancestor(terach,X).  
X=benjamin ;  
X=jacob ;  
X=isaac ;  
X=abraham ;  
no
```

Ordem das Cláusulas (2)

```
member(X, [X|Xs]).  
member(X, [Y|Ys]) ← member(X, Ys).
```

```
member(X, [Y|Ys]) ← member(X, Ys).  
member(X, [X|Xs]).
```

?- **member(X, [1,2,3]).**

```
member(X, [1,2,3])          X=1  
    X=1 ;  
member(X, [1,2,3])          X=X1  
    member(X1, [2,3])         X1=2  
        X=2 ;  
    member(X1, [2,3])         X1=X2  
        member(X2, [3])       X2=3  
            X=3 ;  
        member(X2, [3])       X2=X3  
            member(X3, [])  
                f
```

```
member(X, [1,2,3])          X=X1  
    member(X1, [2,3])         X1=X2  
        member(X2, [3])       X2=X3  
            member(X3, [])  
                f  
    member(X2, [3])          X2=3  
        X=3 ;  
    member(X1, [2,3])         X1=2  
        X=2 ;  
    member(X, [1,2,3])        X=1  
        X=1 ;  
            no
```

Terminação

- Se a árvore de pesquisa tiver um ramo infinito, a computação pode não terminar
 - o Prolog, fazendo pesquisa em profundidade, pode não conseguir encontrar a solução
- A não terminação tem origem nas regras recursivas

```
append([X|Xs],Ys,[X|Zs]) ← append(Xs,Ys,Zs).  
append([],Ys,Ys).
```

?- **append(Xs, [c,d], Ys) .**

```
append(Xs,[c,d],Ys)           Xs=[X|Xs1], Ys=[X|Ys1]  
append(Xs1,[c,d],Ys1)         Xs1=[X1|Xs2], Ys1=[X1|Ys2]  
append(Xs2,[c,d],Ys2)         Xs2=[X2|Xs3], Ys2=[X2|Ys3]  
append(Xs3,[c,d],Ys3)         Xs3=[X3|Xs4], Ys3=[X3|Ys4]
```

:

:

```
married(X,Y) ← married(Y,X).  
married(abraham,sarah).
```

?- **married(abraham, sarah) .**

```
married(abraham,sarah)  
married(sarah,abraham)  
married(abraham,sarah)  
married(sarah,abraham)
```

:

- Evitar *recursividade à esquerda*:

```
are_married(X,Y) ← married(X,Y).  
are_married(X,Y) ← married(Y,X).
```

Análise de Programas Recursivos

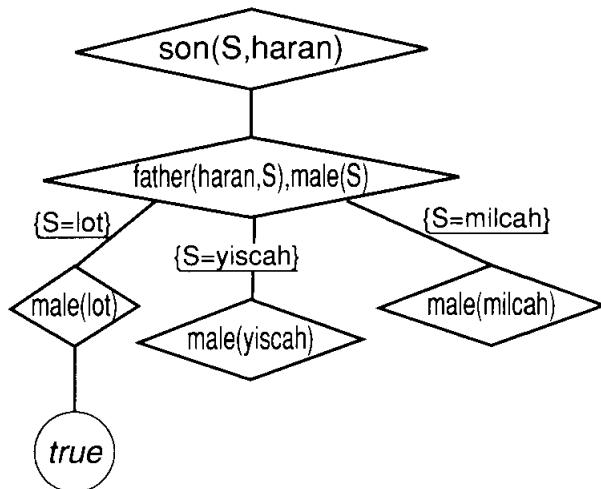
- Determinar que perguntas terminam em relação a um programa recursivo
- `append`
 - termina se 1º ou 3º argumento é uma lista completa
 - não termina quando o 1º e 3º argumentos são listas incompletas unificáveis
- `member`
 - termina se o 2º argumento é uma lista completa
 - não termina se o 2º argumento é uma lista incompleta
- Cuidado com definições circulares:

```
parent(X,Y) ← child(Y,X).  
child(X,Y) ← parent(Y,X).
```

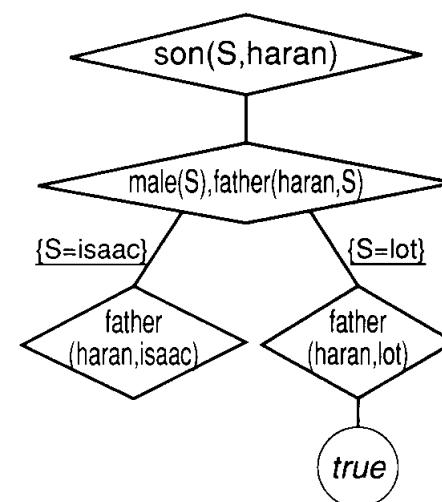
Ordem dos Objectivos

- Ordem das regras não determina a árvore de pesquisa
 - a ordem da travessia é que é diferente
- Ordem dos objectivos **determina a árvore de pesquisa**
 - ordem diferente obtém árvore de pesquisa diferente
 - logo, o esforço de pesquisa da solução será também diferente!

`son(X, Y) ← father(Y, X), male(X).`



`son(X, Y) ← male(X), father(Y, X).`



- Qual é melhor?
- Depende do uso: `son(sarah, X)` ?

Ordem dos Objectivos (2)

```
grandparent(X,Z) ← parent(X,Y), parent(Y,Z).
```

```
grandparent(X,Z) ← parent(Y,Z), parent(X,Y).
```

- Para perguntas do tipo `grandparent(abraham, GC)`? a 1^a regra é melhor
- Para perguntas do tipo `grandparent(GP, isaac)`? a 2^a regra é melhor
- Se a eficiência é um aspecto importante, definir relações distintas:

```
grandparent(X,Z) ← parent(Y,Z), parent(X,Y).
```

```
grandchild(Z,X) ← parent(X,Y), parent(Y,Z).
```

- Troca dos objectivos pode levar a *recursividade à esquerda*: ramo infinito?

```
ancestor(X,Y) ← parent(X,Z), ancestor(Z,Y).
```

- E não `ancestor(X,Y) ← ancestor(Z,Y), parent(X,Z)`.

– Mas:

```
reverse([ ], [ ]).
```

```
reverse([X|Xs], Zs) ← reverse(Xs, Ys), append(Ys, [X], Zs).
```

- termina se o 1º argumento é uma lista completa
- se os objectivos forem trocados, o critério de terminação passa para o 2º argumento

Heurísticas de Ordenação

- Colocar testes primeiro

```
partition([X|Xs],Y,[X|Ls],Bs) ← X ≤ Y, partition(Xs,Y,Ls,Bs).  
partition([X|Xs],Y,Ls,[X|Bs]) ← X > Y, partition(Xs,Y,Ls,Bs).  
partition([],Y,[],[]).
```

- Colocar primeiro os objectivos com menos soluções
 - depende da base de dados
- Colocar primeiro os objectivos mais instanciados
 - depende do uso
- Objectivo: **falhar o mais rápido possível!**
 - falhar significa podar a árvore de pesquisa, levando mais depressa à solução

Soluções Redundantes

- Havendo várias formas de obter a mesma solução, a árvore de pesquisa é desnecessariamente maior
 - logo, mais demora a computação
- Será melhor manter a árvore de pesquisa o mais pequena possível
- Possíveis origens da redundância:
 - cobrir o mesmo caso com regras diferentes

$\text{minimum}(X, Y, X) \leftarrow X \leq Y.$
 $\text{minimum}(X, Y, Y) \leftarrow Y \leq X.$

$\text{minimum}(X, Y, X) \leftarrow X \leq Y.$
 $\text{minimum}(X, Y, Y) \leftarrow Y < X.$

- casos especiais a mais (por vezes motivados por questões de eficiência)

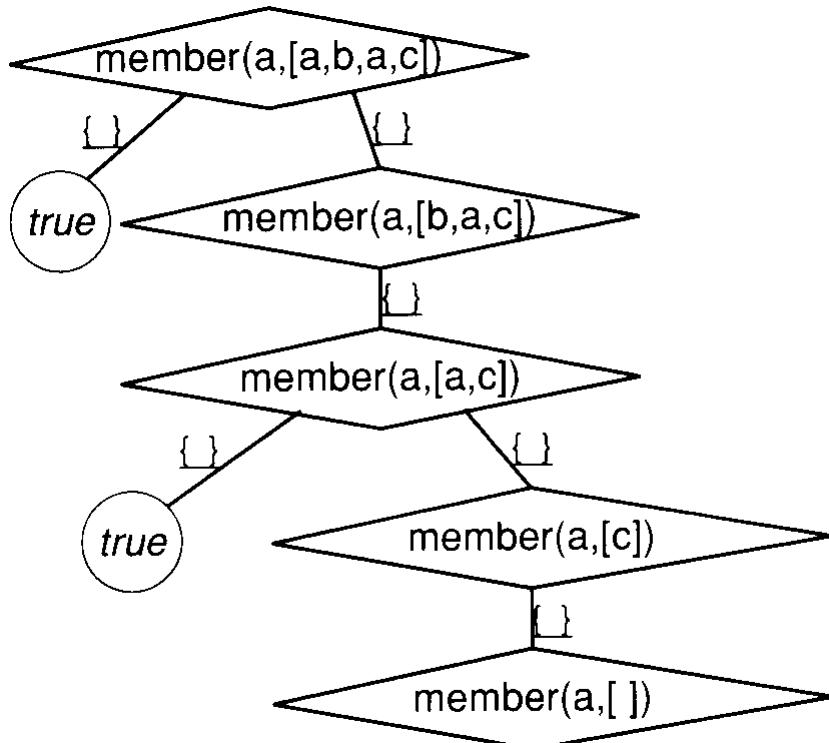
$\text{append}([X|Xs], Ys, [X|Zs]) \leftarrow \text{append}(Xs, Ys, Zs).$
 $\text{append}([], Ys, Ys).$
 $\text{append}(Xs, [], Xs).$

$\text{append}([X|Xs], [Y|Ys], [X|Zs]) \leftarrow \text{append}(Xs, [Y|Ys], Zs).$
 $\text{append}([], [Y|Ys], [Y|Ys]).$
 $\text{append}(Xs, [], Xs).$

Soluções Redundantes (2)

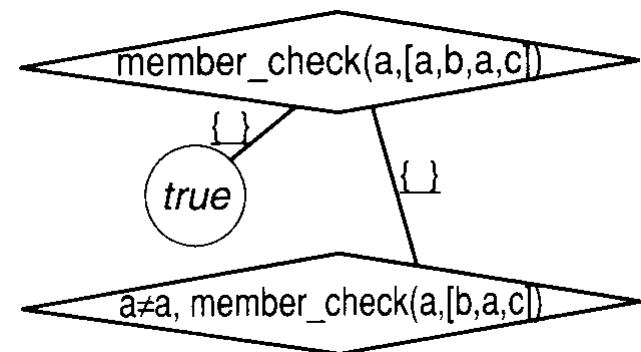
```
member(X, [X|Xs]).  
member(X, [Y|Ys]) ← member(X,Ys).
```

```
?- member(a, [a,b,a,c]).
```



```
member_check(X, [X|Xs]).  
member_check(X, [Y|Ys]) ← X ≠ Y, member_check(X,Ys).
```

```
?- member_check(a, [a,b,a,c]).
```



Aritmética

- Predicados de sistema (*built-in*) complementam o “Prolog puro”
 - perguntas usando estes predicados são tratadas de forma especial (*avaliação* em vez de redução)
- Aritmética
 - maior eficiência: usar capacidades aritméticas do computador
 - desvantagem: perda de generalidade
 - operações aritméticas menos genéricas do que versões baseadas em lógica
 - avaliador aritmético: **is (Value, Expression)**
 - Value **is** Expression
 - a expressão Expression é **avaliada** e o resultado é **unificado** com Value
 - a expressão não pode conter variáveis não instanciadas
 - tem sucesso se a unificação tiver sucesso
 - X **is** 3+5?
X=8
 - 8 **is** 3+5?
true
 - 3+5 **is** 3+5?
false
 - N **is** N+1? nunca poderá ter sucesso!
 - N instanciado: falha; N variável: erro (expressão não pode ser avaliada)

Operadores

- Aritméticos: $+$, $-$, $*$, $/$, mod
- Unificação: $=$, $\backslash=$
- Comparação: $<$, $=<$, $>$, $>=$, $=:=$, $=\backslash=$
 - ambas as expressões são avaliadas

$1 < 2?$

true

$3+5 = 4+4?$

false

$N = 3?$

$N = 3.$

$2+3 = N, N \text{ is } 5?$

false

$N = 2+3, N \backslash= 5?$

$N = 2+3.$

$3-2 < 2*3+1?$

true

$3+5 =:= 4+4?$

true

$N \text{ is } 3?$

$N = 3.$

$2+3 = N, N =:= 5?$

$N = 2+3.$

$N = 2+3, N = 5?$

false.

$2 < 1?$

false

$3+5 \text{ is } 4+4?$

false

$N =:= 3?$

ERROR



Programas Aritméticos

```
plus(0,X,X) ← natural_number(X).  
plus(s(X),Y,s(Z)) ← plus(X,Y,Z).
```



```
plus(X,Y,Z) ← Z is X+Y.
```

- Restrição nos usos múltiplos

plus(3,X,8) ?

ERROR

```
factorial(0,s(0)).  
factorial(s(N),F) ← factorial(N,F1), times(s(N),F1,F).
```



```
factorial(N,F) ←  
    N > 0, N1 is N-1, factorial(N1,F1), F is N*F1.  
factorial(0,1).
```

- Perda da estrutura recursiva dos números
 - necessário o cálculo explícito de $N-1$
 - condição $N>0$ para garantir terminação

Recursividade vs. Iteração

- Computações iterativas são mais eficientes do que as recursivas
 - n invocações recursivas => espaço linear em n
 - programa iterativo usa tipicamente um espaço constante (independente do número de iterações)
- No Prolog, a recursividade é usada para especificar algoritmos recursivos e iterativos
 - **cláusula iterativa**: chamada recursiva é o último objectivo do corpo
 - **procedimento iterativo**: se contiver apenas factos e cláusulas iterativas

Procedimentos Iterativos

```
int factorial(int n) {  
    int i=0, t=1;  
    while(i<n) {  
        i+=1;  
        t*=i;  
    }  
    return t;  
}  
  
factorial(N,F) ← factorial(0,N,1,F).  
factorial(I,N,T,F) ←  
    I < N, I1 is I+1, T1 is T*I1, factorial(I1,N,T1,F).  
factorial(N,N,F,F).
```

inicialização

- em Prolog não temos variáveis auxiliares para guardar resultados intermédios
- solução: aumentar o procedimento com argumentos extra – *acumuladores*
 - tipicamente um acumulador terá o resultado da computação aquando da terminação
- não esquecer: as variáveis lógicas são “write-once”!
 - daí que seja necessário passar uma nova variável lógica com o novo valor acumulado

```
factorial(N,F) ← factorial(N,1,F).  
  
factorial(N,T,F) ←  
    N > 0, T1 is T*N, N1 is N-1, factorial(N1,T1,F).  
factorial(0,F,F).
```

Mais Exemplos

```
between(I,J,I) ← I ≤ J.  
between(I,J,K) ← I < J, I1 is I+1, between(I1,J,K).
```

```
sumlist([I|Is],Sum) ← sumlist(Is,IsSum), Sum is I+IsSum.  
sumlist([],0).
```

```
sumlist(Is,Sum) ← sumlist(Is,0,Sum).  
sumlist([I|Is],Temp,Sum) ←  
    Temp1 is Temp+I, sumlist(Is,Temp1,Sum).  
sumlist([],Sum,Sum).
```

```
maxlist([X|Xs],M) ← maxlist(Xs,X,M).  
maxlist([X|Xs],Y,M) ← maximum(X,Y,Y1), maxlist(Xs,Y1,M).  
maxlist([],M,M).  
maximum(X,Y,Y) ← X ≤ Y.  
maximum(X,Y,X) ← X > Y.
```

```
length([X|Xs],N) ← N > 0, N1 is N-1, length(Xs,N1).  
length([],0).
```

```
length([X|Xs],N) ← length(Xs,N1), N is N1+1.  
length([],0).
```

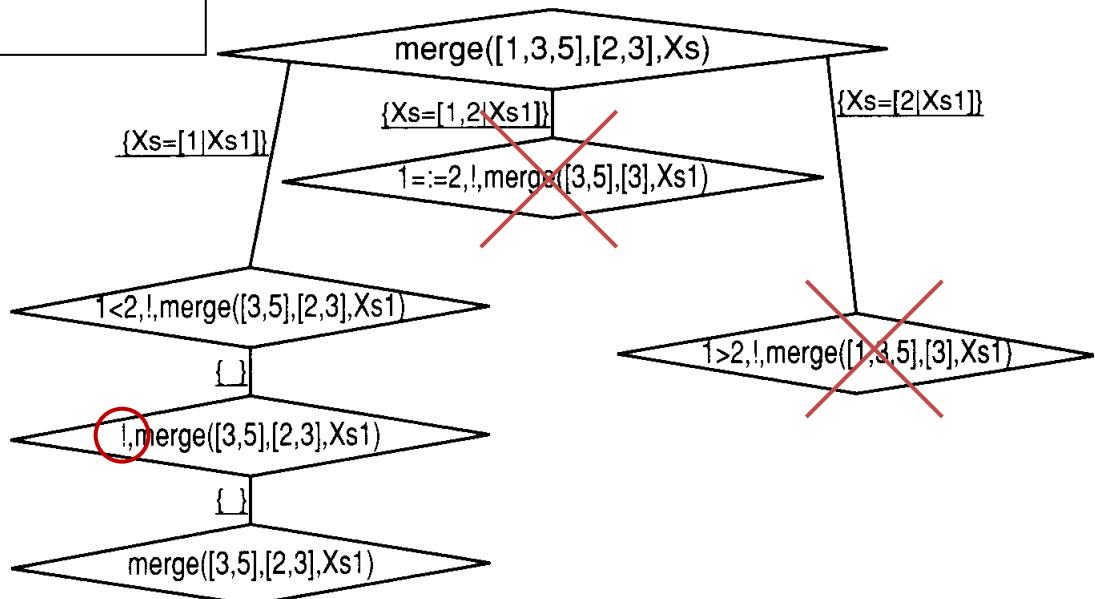
Cut – !

- O *cut* permite afectar o comportamento procedural dos programas
 - principal função: reduzir o espaço de procura podando dinamicamente a árvore de pesquisa
 - reduz tempo de computação
 - reduz espaço, pois alguns pontos de escolha deixam de ser necessários

```
merge([X|Xs],[Y|Ys],[X|Zs]) ← X < Y, merge(Xs,[Y|Ys],Zs).  
merge([X|Xs],[Y|Ys],[X,Y|Zs]) ← X==Y, merge(Xs,Ys,Zs).  
merge([X|Xs],[Y|Ys],[Y|Zs]) ← X > Y, merge([X|Xs],Ys,Zs).  
merge(Xs,[ ],Xs).  
merge([ ],Ys,Ys).
```

- Expressar exclusividade mútua:

```
merge([X|Xs],[Y|Ys],[X|Zs]) ←  
    X < Y, !, merge(Xs,[Y|Ys],Zs).  
merge([X|Xs],[Y|Ys],[X,Y|Zs]) ←  
    X==Y, !, merge(Xs,Ys,Zs).  
merge([X|Xs],[Y|Ys],[Y|Zs]) ←  
    X > Y, !, merge([X|Xs],Ys,Zs).  
merge(Xs,[ ],Xs) ← !.  
merge([ ],Ys,Ys) ← !.
```

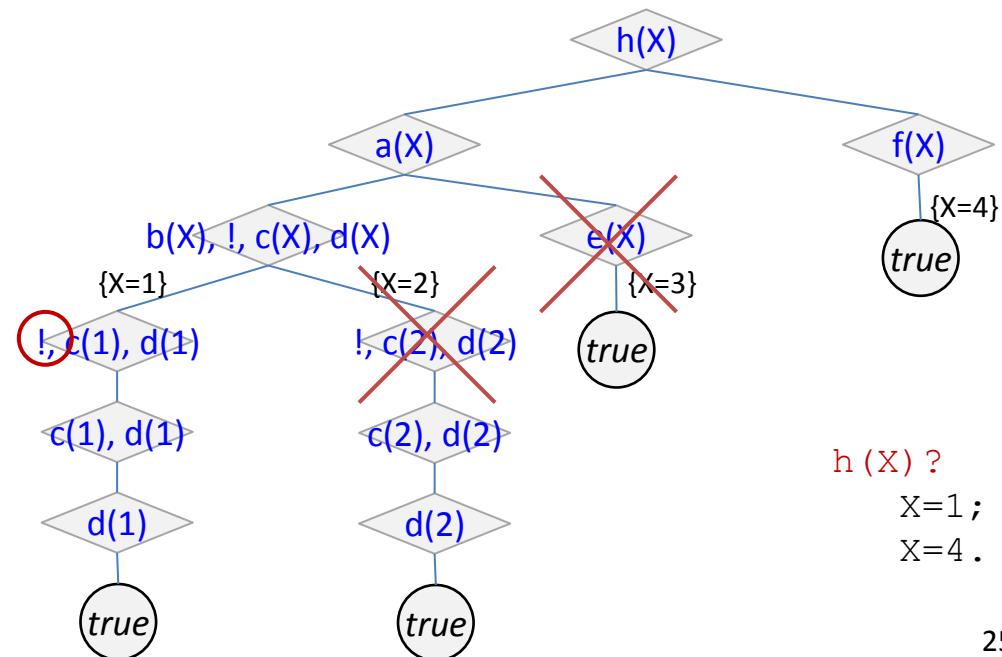


Cut – ! (2)

O *cut* sucede e compromete o Prolog com todas as escolhas feitas desde que o objectivo pai foi unificado com a cabeça da cláusula onde o *cut* ocorre.

- portanto:
 - o *cut* corta todas as cláusulas (do mesmo predicado) abaixo dele
 - o *cut* corta todas as soluções alternativas para os objectivos à sua esquerda na cláusula
 - o *cut* não afecta os objectivos à sua direita!
 - em caso de retrocesso no *cut*, a pesquisa continuará a partir da última escolha feita antes da escolha desta cláusula

```
h(X) :- a(X).  
h(X) :- f(X).  
a(X) :- b(X), !, c(X), d(X).  
a(X) :- e(X).  
b(1). b(2).  
c(1). c(2).  
d(1). d(2).  
e(3).  
f(4).
```



Remover Computações Redundantes

```
sort(Xs, Ys) ←  
    append(As, [X, Y|Bs], Xs),  
    X > Y,  
    append(As, [Y, X|Bs], Xs1),  
    sort(Xs1, Ys).  
  
sort(Xs, Xs) ← ordered(Xs).
```

não interessa quais dois elementos são trocados,
desde que $X > Y$
(no final a lista ordenada só tem uma solução)



```
sort(Xs, Ys) ←  
    append(As, [X, Y|Bs], Xs),  
    X > Y,  
    !,  
    append(As, [Y, X|Bs], Xs1),  
    sort(Xs1, Ys).  
  
sort(Xs, Xs) ←  
    ordered(Xs),  
    !.
```

Optimização com Recursividade em Cauda

- Possibilidade de executar um procedimento recursivo como iterativo (espaço constante)
- $A' \leftarrow B_1, B_2, \dots, B_n.$
 - possível aplicação da optimização ao último objectivo da cauda (B_n) desde que não haja pontos de escolha desde que esta cláusula foi seleccionada
 - não há cláusulas alternativas
 - não há pontos de escolha para B_1, \dots, B_{n-1} (i.e., este objectivo conjuntivo foi resolvido de forma determinística)

```
append([X|Xs], Ys, [X|Zs]) ← append(Xs, Ys, Zs).  
append([], Ys, Ys).
```

- com lista completa no 1º argumento, a invocação recursiva cumpre os requisitos!
 - requer análise da segunda cláusula
- Técnica:
 - cláusulas da forma $A' \leftarrow B_1, B_2, \dots, !, B_n.$
 - predicado determinístico

```
merge([X|Xs], [Y|Ys], [X|Zs]) ←  
    X < Y, !, merge(Xs, [Y|Ys], Zs).  
merge([X|Xs], [Y|Ys], [X,Y|Zs]) ←  
    X=:=Y, !, merge(Xs, Ys, Zs).  
merge([X|Xs], [Y|Ys], [Y|Zs]) ←  
    X > Y, !, merge([X|Xs], Ys, Zs).  
merge(Xs, [], Xs) ← !.  
merge([], Ys, Ys) ← !.
```

Implementação da Negação

```
not X ← X, !, fail.  
not X.
```

- `fail`: predicado que provoca falhanço
- `not G` falha se `G` sucede e sucede se `G` falha
- A ordem das regras é essencial
 - se as regras forem trocadas não é apenas a ordem das soluções que se altera, mas sim o significado do programa!
- Se `G` termina, `not G` também; se `G` não termina, `not G` pode terminar ou não
 - termina (sem sucesso) se for encontrada uma solução para `G` antes de um ramo infinito
- Esta é uma implementação incompleta da negação por falha

```
p(s(X)) ← p(X).  
q(a).  
  
not(p(X), q(X))?
```

```
unmarried_student(X) ← not married(X), student(X).  
student(bill).  
married(joe).  
  
unmarried_student(X)?
```

- Se usado com objectivos não totalmente instanciados, `not` pode não funcionar

Cuts Verdes e Cuts Vermelhos

- *Cut verde*
 - não altera o significado do programa: o mesmo conjunto de soluções é encontrado com ou sem o *cut*
 - corta apenas caminhos de computação que não levam a novas soluções

```
minimum(X,Y,X) ← X ≤ Y, !.  
minimum(X,Y,Y) ← X > Y, !.
```

- *Cut vermelho*
 - se retirado, altera o significado do programa: o conjunto de soluções será diferente
 - a ordem das cláusulas passa a ser fixa!

```
not X ← X, !, fail.  
not X.
```

Omissão de Condições

```
minimum(X,Y,X) ← X ≤ Y, !.  
minimum(X,Y,Y) ← X > Y, !.
```



```
minimum(X,Y,X) ← X ≤ Y, !.  
minimum(X,Y,Y).
```

mas: `minimum(2, 5, 5)` ? sucede!

```
minimum(X,Y,Z) ← X ≤ Y, !, Z = X.  
minimum(X,Y,Y).
```

- A omissão de condições transforma *cuts* verdes em vermelhos

```
delete([X|Xs], X, Ys) ← !, delete(Xs, X, Ys).  
delete([X|Xs], Z, [X|Ys]) ← X ≠ Z, !, delete(Xs, Z, Ys).  
delete([], X, []).
```

```
delete([X|Xs], X, Ys) ← !, delete(Xs, X, Ys).  
delete([X|Xs], Z, [X|Ys]) ← !, delete(Xs, Z, Ys).  
delete([], X, []).
```

```
if_then_else(P,Q,R) ← P, !, Q.  
if_then_else(P,Q,R) ← R.
```

- *cut* vermelho evita dupla computação de P (`not P` omitido na 2ª regra)

Predicados de Tipo

```
integer(X) ← X is an integer.  
atom(X) ← X is an atom.  
real(X) ← X is a floating-point number.  
compound(X) ← X is a compound term.  
number(X) ← X is integer or real.  
atomic(X) ← X is an atom or a number.
```

```
flatten([X|Xs], Ys) ←  
    flatten(X, Ys1), flatten(Xs, Ys2), append(Ys1, Ys2, Ys).  
flatten(X, [X]) ←  
    constant(X), X ≠ [ ].  
flatten([], []).  
  
constant(X) ← atomic(X).
```

```
flatten(Xs, Ys) ← flatten(Xs, [], Ys).  
flatten([X|Xs], S, Ys) ←  
    list(X), flatten(X, [Xs|S], Ys).  
flatten([X|Xs], S, [X|Ys]) ←  
    constant(X), X ≠ [ ], flatten(Xs, S, Ys).  
flatten([], [X|S], Ys) ←  
    flatten(X, S, Ys).  
flatten([], [], []).  
list([X|Xs]).
```

Inspecção de Estruturas

- **functor(Term,F,Arity) / arg(N,Term,Arg)**
 - decomposição de termos
 - `functor(father(haran,lot),X,Y)?`
`{X=father, Y=2}`
 - `arg(2,father(haran,lot),X)?`
`X=lot`
 - criação de termos
 - `functor(T,father,2)?`
`T=father(X,Y)`
 - `arg(1,father(X,lot),haran)?`
`X=haran`
- **Term =.. List**
 - construir um termo a partir de uma lista
 - `X =.. [father,haran,lot]?`
`X=father(haran,lot)`
 - construir uma lista a partir de um termo
 - `father(haran,lot) =.. Xs?`
`Xs=[father,haran,lot]`

Inspecção de Estruturas (2)

```
subterm(Term,Term).  
subterm(Sub,Term) ←  
    compound(Term), functor(Term,F,N), subterm(N,Sub,Term).  
subterm(N,Sub,Term) ←  
    N > 1, N1 is N-1, subterm(N1,Sub,Term).  
subterm(N,Sub,Term) ←  
    arg(N,Term,Arg), subterm(Sub,Arg).
```

```
subterm(t, a(b(c),d(t)))?  
true  
subterm(X, a(b(c),d(t)))?  
X = a(b(c),d(t)) ;  
X = b(c) ;  
X = c ;  
X = d(t) ;  
X = t ;  
false
```

```
subterm(Term,Term).  
subterm(Sub,Term) ←  
    compound(Term), Term =.. [F|Args], subterm_list(Sub,Args).  
subterm_list(Sub,[Arg|Args]) ←  
    subterm(Sub,Arg).  
subterm_list(Sub,[Arg|Args]) ←  
    subterm_list(Sub,Args).
```

Predicados Meta-Lógicos

- **var(Term) / nonvar(Term)**

var(X) ?

true

var(a) ?

false

var([X|Xs]) ?

false

```
plus(X,Y,Z) ← nonvar(X), nonvar(Y), Z is X+Y.  
plus(X,Y,Z) ← nonvar(X), nonvar(Z), Y is Z-X.  
plus(X,Y,Z) ← nonvar(Y), nonvar(Z), X is Z-Y.
```

plus(1,2,X) ?

X=3

plus(1,X,3) ?

X=2

plus(X,2,3) ?

X=1

plus(X,Y,3) ?

false

testes meta-lógicos

```
length(Xs,N) ← nonvar(Xs), length1(Xs,N).  
length(Xs,N) ← var(Xs), nonvar(N), length2(Xs,N).
```

```
ground(Term) ←  
    nonvar(Term), constant(Term).  
ground(Term) ←  
    nonvar(Term),  
    compound(Term),  
    functor(Term,F,N),  
    ground(N,Term).  
  
ground(N,Term) ←  
    N > 0,  
    arg(N,Term,Arg),  
    ground(Arg),  
    N1 is N-1,  
    ground(N1,Term).  
ground(0,Term).
```

Predicados Meta-Lógicos (2)

- \equiv / $\backslash\equiv$

- verificar se dois termos são ou não idênticos
- $X \equiv Y?$ tem sucesso se X e Y são:

- constantes idênticas
- variáveis idênticas
- estruturas com o mesmo nome e aridade, e recursivamente cada $X_i \equiv Y_i?$ tem sucesso para todos os argumentos de X e Y , respectivamente

$X \equiv 5?$

false

$X \backslash\equiv Y?$

true

$a(b) \equiv a(X) ?$

false

```
occurs_in(X,Term) ←  
    subterm(Sub,Term), X == Sub.
```

```
Y = b(X,c), subterm(a,Y) ?  
    Y=b(a,c), X=a  
Y = b(X,c), occurs_in(a,Y) ?  
    false  
Y = b(X,c), occurs_in(c,Y) ?  
    Y=b(X,c)
```

Meta-Variável

- Equivalência entre programas e dados
 - ambos podem ser representados como termos lógicos
- Converter um termo num objectivo
 - **call (X)** invoca o objectivo X
- **Meta-variável**: variável usada como um objectivo no corpo de uma cláusula
 - durante a computação, aquando da sua invocação a variável deverá estar instanciada com um termo (se não, erro)
 - **call (G) ≡ G**
- Principais utilizações:
 - meta-programação: meta-interpretadores, *shells*
 - definição da negação
 - definição de predicados de ordem mais elevada

```
read(G), call(G) ?  
| : write(qwe).  
qwe  
G = write(qwe)
```

X ; Y ← X.	or(X,Y) :- X.
X ; Y ← Y.	or(X,Y) :- Y.

Predicados Extra-Lógicos

- Produzem efeitos colaterais quando satisfeitos
- I/O
 - `read(X)`
 - lê termo do canal de entrada e unifica-o com `X`
 - `write(X)`
 - escreve `X` no canal de saída
 - `get_char(Char) / put_char(Char)`
 - `nl`
 - muda de linha (*new line*)

```
writeln([X|Xs]) ← write(X), writeln(Xs).  
writeln([]) ← nl.
```

```
read(X), writeln(['The value of X is ',X])?  
|: qwe  
The value of X is qwe
```

Strings e Códigos ASCII

- String (entre aspas) corresponde a uma lista de inteiros que são os códigos ASCII de cada carácter na string
 - "The ": [84, 104, 101, 32]
- name (X, Ys)
 - converte átomo X na lista Ys com os códigos ASCII dos caracteres de X
 - name ('The ', Ys).
Ys = [84, 104, 101, 32]
- put (N)
 - escreve o carácter cujo código ASCII é N
 - put (104)
h
- get0 (N)
 - lê carácter e unifica o seu código ASCII com N
- get (N)
 - lê carácter não branco

Ficheiros

- `see (F)`
 - abre um canal de leitura para o ficheiro `F`
 - as leituras passam a ser feitas a partir de `F`
- `tell (F)`
 - abre um canal de escrita para o ficheiro `F`
 - as escritas passam a ser feitas para `F`
- `seeing (F) / telling (F)`
 - `F` é unificado com o nome do ficheiro no canal corrente
- `seen / told`
 - fecha o canal corrente

Acesso e Manipulação do Programa

- `listing / listing(Pred)`
 - lista as cláusulas do programa/do predicado no canal de saída
- `clause(Head, Body)`
 - procura cláusula cuja cabeça unifica com `Head` (argumento instanciado)
 - em retrocesso, sucede uma vez por cada cláusula unificável
- `assertz(Clause) / asserta(Clause)`
 - adiciona `Clause` como última/primeira cláusula do procedimento
 - no caso de regras, é necessário aplicar parêntesis
 - `assertz((a :- b, c)).`
- `retract(C)`
 - remove a primeira cláusula que unifica com `C`
 - `retract((a :- Body)).`
 - em retrocesso, sucede uma vez por cada cláusula unificável
- `consult(File)`
 - lê e adiciona (`assert`) as cláusulas do ficheiro `File`
- `reconsult(File)`
 - substitui os predicados definidos no ficheiro (`retract` das cláusulas antes de `assert`)

Memorização

- Guardar resultados anteriores por questões de eficiência

```
lemma(P) ← P, asserta((P ← !)).
```

- da próxima vez que `P` for tentado, será utilizada a nova solução (`asserta`)
 - `!` impede a utilização de cláusulas posteriores

- Torres de Hanói

- N discos: solução tem $2^N - 1$ movimentos
 - solução consiste em retirar $N-1$ discos de cima do maior, mover o maior e voltar a deslocar $N-1$ discos para cima dele
 - memorizar solução para mover $N-1$ discos

```
hanoi(1,A,B,C,[A to B]).  
hanoi(N,A,B,C,Moves) ←  
    N > 1,  
    N1 is N-1,  
    lemma(hanoi(N1,A,C,B,Ms1)),  
    hanoi(N1,C,B,A,Ms2),  
    append(Ms1,[A to B|Ms2],Moves).
```

```
test_hanoi(N,Pegs,Moves) ←  
    hanoi(N,A,B,C,Moves), Pegs = [A,B,C].
```

Ciclos

- Baseiam-se em efeitos colaterais (e.g. leitura/escrita)
- Programa interactivo

```
echo ← read(X), echo(X).  
  
echo(X) ← last_input(X), !.  
echo(X) ← write(X), nl, read(Y), !, echo(Y).
```

- análogo a ciclo “while”
- é iterativo e determinístico (optimização com recursividade em cauda)
- Ciclo baseado em falha
 - análogo a ciclo “repeat”

```
echo ← repeat, read(X), echo(X), !.  
  
echo(X) ← last_input(X), !.  
echo(X) ← write(X), nl, fail.
```

```
repeat.  
repeat ← repeat.
```

```
tab(N) ← between(1,N,I), put_char(' '), fail.  
tab(N) ← !.
```

Interpretador de Comandos

```
shell ←  
    shell_prompt, read(Goal), shell(Goal).  
  
shell(exit) ← !.  
shell(Goal) ←  
    ground(Goal), !, shell_solve_ground(Goal), shell.  
shell(Goal) ←  
    shell_solve(Goal), shell.  
  
shell_solve(Goal) ←  
    Goal, write(Goal), nl, fail.  
shell_solve(Goal) ←  
    write('No (more) solutions'), nl.  
  
shell_solve_ground(Goal) ←  
    Goal, !, write('Yes'), nl.  
shell_solve_ground(Goal) ←  
    write('No'), nl.  
  
shell_prompt ← write('Next command? ').
```

ciclo interactivo

ciclo baseado em falha

PROGRAMAÇÃO EM LÓGICA

PROLOG AVANÇADO



Não Determinismo

- Como escolher criteriosamente o próximo passo?
- Geração e teste

```
find(X) ← generate(X), test(X).
```

- tira partido do mecanismo de retrocesso
 - o gerador tenta escolher correctamente a solução e o testador verifica se essa escolha está correcta
- Exemplos:

```
verb(Sentence,Word) ← member(Word,Sentence), verb(Word).  
noun(Sentence,Word) ← member(Word,Sentence), noun(Word).  
article(Sentence,Word) ← member(Word,Sentence), article(Word).  
  
noun(man).      noun(woman).    verb([a,man,loves,a,woman],V) ?  
article(a).      verb(loves).    V=loves
```

```
intersect(Xs,Ys) ← member(X,Xs), member(X,Ys).
```

```
sort(Xs,Ys) ← permutation(Xs,Ys), ordered(Ys).
```

Melhorar Geração de Soluções

- Geração e teste: programas são mais fáceis de construir do que obter directamente a solução, mas são também menos eficientes
- Como optimizar?
 - dirigir a geração de soluções possíveis incluindo o teste nessa geração

		Q	
Q			
			Q
	Q		

```
queens(N,Qs) ←
    range(1,N,Ns), permutation(Ns,Qs), safe(Qs).

safe([Q|Qs]) ← safe(Qs), not attack(Q,Qs).
safe([]).

attack(X,Xs) ← attack(X,1,Xs).

attack(X,N,[Y|Ys]) ← X is Y+N ; X is Y-N.
attack(X,N,[Y|Ys]) ← N1 is N+1, attack(X,N1,Ys).
```

```
queens(N,Qs) ← range(1,N,Ns), queens(Ns,[ ],Qs).

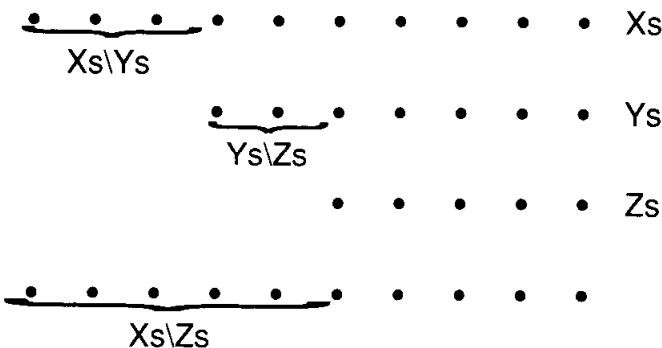
queens(UnplacedQs, SafeQs, Qs) ←
    select(Q, UnplacedQs, UnplacedQs1),
    not attack(Q, SafeQs),
    queens(UnplacedQs1, [Q|SafeQs], Qs).

queens([ ], Qs, Qs).
```

Listas de Diferença

- Representação alternativa para listas: diferença entre pares de listas
 - $[1, 2, 3]$
 - $[1, 2, 3, 4, 5] \setminus [4, 5]$ $[1, 2, 3, 8] \setminus [8]$ $[1, 2, 3] \setminus []$
 - usando listas incompletas: $[1, 2, 3 | Xs] \setminus Xs$
 - como as expressões lógicas são unificadas (e não avaliadas), o functor é arbitrário: \setminus , $-$, ...
 - qualquer lista pode ser representada como lista de diferença:
 - uma lista L fica $L \setminus []$ lista vazia: $As \setminus As$
- Vantagem: eficiência
 - duas listas de diferença incompletas podem ser concatenadas em tempo constante numa terceira lista de diferença
 - o append concatena listas em tempo linear no tamanho da 1^a lista

Concatenação com Listas de Diferença



```
append_dl(Xs\Ys, Ys\Zs, Xs\Zs).
```

- Condição: para concatenar $As \setminus Bs$ com $Xs \setminus Ys$, Bs têm que unificar com Xs – listas de diferença *compatíveis*
- Se a cauda de uma lista de diferença for não instanciada, essa lista de diferença é compatível com qualquer outra lista de diferença!

```
append_dl([a,b,c|Xs]\Xs, [1,2]\[], Ys)?  
Xs=[1,2], Ys=[a,b,c,1,2]\[]
```

Exemplos

```
flatten([X|Xs],Ys) ←  
    flatten(X,Ys1), flatten(Xs,Ys2), append(Ys1,Ys2,Ys).  
flatten(X,[X]) ←  
    constant(X), X ≠ [ ].  
flatten([],[]).
```

```
flatten(Xs,Ys) ← flatten_dl(Xs,Ys\[ ] ).  
flatten_dl([X|Xs],Ys\Zs) ←  
    flatten_dl(X,Ys\Ys1), flatten_dl(Xs,Ys1\Zs).  
flatten_dl(X,[X|Xs]\Xs) ←  
    constant(X), X ≠ [ ].  
flatten_dl([],Xs\Xs).
```

```
reverse([],[]).  
reverse([X|Xs],Zs) ← reverse(Xs,Ys), append(Ys,[X],Zs).
```

```
reverse(Xs,Ys) ← reverse_dl(Xs,Ys\[ ] ).  
reverse_dl([X|Xs],Ys\Zs) ←  
    reverse_dl(Xs,Ys\[X|Zs]).  
reverse_dl([],Xs\Xs).
```

```
quicksort([X|Xs],Ys) ←  
    partition(Xs,X,Littles,Bigs),  
    quicksort(Littles,Ls),  
    quicksort(Bigs,Bs),  
    append(Ls,[X|Bs],Ys).  
quicksort([],[]).
```

```
quicksort(Xs,Ys) ← quicksort_dl(Xs,Ys\[ ] ).  
quicksort_dl([X|Xs],Ys\Zs) ←  
    partition(Xs,X,Littles,Bigs),  
    quicksort_dl(Littles,Ys\[X|Ys1]),  
    quicksort_dl(Bigs,Ys1\Zs).  
quicksort_dl([],Xs\Xs).
```

Estruturas de Dados Incompletas

- Trabalhar com estruturas de dados incompletas facilita a adição de elementos

```
lookup(Key, [(Key, Value) | Dict], Value).  
lookup(Key, [(Key1, Value1) | Dict], Value) ←  
    Key ≠ Key1, lookup(Key, Dict, Value).
```

```
dict([(arnold, 8881), (barry, 4513), (cathy, 5950) | Xs]).  
dict(Dict), lookup(arnold, Dict, N)?  
    N=8881  
dict(Dict), lookup(david, Dict, 1199)?  
    Dict = [(arnold, 8881), (barry, 4513), (cathy, 5950), (david, 1199) | Xs1]
```

- Usando árvores binárias de pesquisa (ordenadas)

```
lookup(Key, dict(Key, X, Left, Right), Value) ←  
    !, X = Value.  
lookup(Key, dict(Key1, X, Left, Right), Value) ←  
    Key < Key1, lookup(Key, Left, Value).  
lookup(Key, dict(Key1, X, Left, Right), Value) ←  
    Key > Key1, lookup(Key, Right, Value).
```

Todas as Soluções

- Obter todos os filhos de um pai
 - usando um acumulador:

```
children(X,Kids) :- children(X,[],Kids).  
children(X,Cs,Kids) :-  
    father(X,Kid), not member(Kid,Cs), !,  
    children(X,[Kid|Cs],Kids).  
children(X,Cs,Cs).
```

- abordagem ineficiente: cada solução adicional para `father/2` recomeça no início da árvore de pesquisa
- alternativa: ciclo baseado em falha combinado com *assert/retract*

```
children(X,Kids) :- assert(kids(X,[])), fail.  
children(X,Kids) :-  
    father(X,Kid),  
    retract(kids(X,Cs)), assert(kids(X,[Kid|Cs])),  
    fail.  
children(X,Kids) :- retract(kids(X,Kids)).
```

- percorre a árvore de pesquisa apenas uma vez!
- mas a asserção/retracção de cláusulas são operações inherentemente pesadas

Findall

- `findall(Term, Goal, Bag)`
 - unifica `Bag` com a lista das instâncias de `Term` para as quais `Goal` é satisfeito
 - todos os `X` para os quais `call(Goal), X=Term?` é satisfeito
 - `Term` e `Goal` tipicamente partilham variáveis

```
children(X,Kids) ← findall(Kid,father(X,Kid),Kids).
```

```
father(terach,abraham).      father(haran,lot).          male(abraham).       male(haran).        female(yiscah).  
father(terach,nachor).       father(haran,milcah).       male(isaac).         male(nachor).       female(milcah).  
father(terach,haran).        father(haran,yiscah).       male(lot).  
father(abraham,isaac).
```

```
children(terach,Xs)?  
Xs = [abraham,nachor,haran].
```

```
findall(F,father(F,K),Fs)?  
Fs = [terach,haran,terach,haran,terach,haran,abraham].
```

```
findall(F-K,father(F,K),Fs)?  
Fs = [terach-abraham, haran-lot, terach-nachor, haran-milcah,  
      terach-haran, haran-yiscah, abraham-isaac].
```

Bagof/Setof

- **bagof (Term, Goal, Bag)**
 - idêntico ao `findall`, mas são encontradas soluções alternativas para as variáveis em `Goal`

```
bagof(F,father(F,K),Fs)?  
K = abraham,  
Fs = [terach] ;  
K = haran,  
Fs = [terach] ;  
K = isaac,  
Fs = [abraham] ;  
K = lot,  
Fs = [haran] ;  
K = milcah,  
Fs = [haran] ;  
K = nachor,  
Fs = [terach] ;  
K = yiscah,  
Fs = [haran].
```

```
bagof(K,father(F,K),Fs)?  
F = abraham,  
Fs = [isaac] ;  
F = haran,  
Fs = [lot, milcah, yiscah] ;  
F = terach,  
Fs = [abraham, nachor, haran].  
  
bagof(K,F^father(F,K),Fs)?  
Fs = [abraham, lot, nachor, milcah,  
      haran, yiscah, isaac].
```

- **setof (Term, Goal, Bag)**
 - soluções ordenadas, sem duplicados (conjunto)

Predicados de Segunda Ordem

- Lógica de primeira ordem permite quantificação sobre indivíduos
- Lógica de segunda ordem permite quantificação sobre predicados

```
has_property([X|Xs],P) ← P(X), has_property(Xs,P).  
has_property([],P).
```

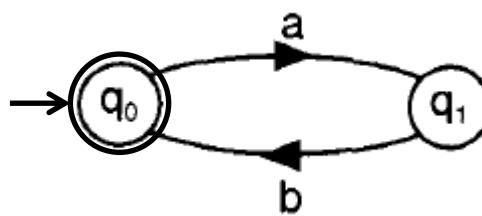
G =... [P,X], G

```
map_list([X|Xs],P,[Y|Ys]) ← P(X,Y), map_list(Xs,P,Ys).  
map_list([],P,[]).
```

G =... [P,X,Y], G

Interpretador para (N)DFA

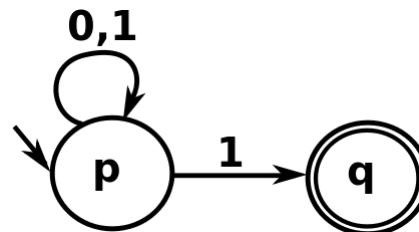
- $NDFA = \langle Q, \Sigma, \delta, I, F \rangle$



initial(q0).
final(q0).
delta(q0,a,q1).
delta(q1,b,q0).

- Interpretador:

```
accept(Xs) ← initial(Q), accept(Xs,Q).  
accept([X|Xs],Q) ← delta(Q,X,Q1), accept(Xs,Q1).  
accept([],Q) ← final(Q).
```



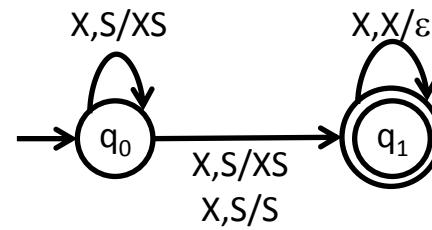
initial(p).
final(q).
delta(p,0,p).
delta(p,1,p).
delta(p,1,q).

Interpretador para NPDA

- $NPDA = \langle Q, \Sigma, G, \delta, I, Z, F \rangle$

- Palíndromos:

```
initial(q0).      final(q1).  
delta(q0,X,S,q0,[X|S]).  
delta(q0,X,S,q1,[X|S]).  
delta(q0,X,S,q1,S).  
delta(q1,X,[X|S],q1,S).
```



- Interpretador:

```
accept(Xs) ← initial(Q), accept(Xs,Q,[ ]).  
accept([X|Xs],Q,S) ← delta(Q,X,S,Q1,S1), accept(Xs,Q1,S1).  
accept([],Q,[ ]) ← final(Q).
```

Gramáticas sem Contexto

- $a^*b^*c^*$

$s \rightarrow a, b, c.$
 $a \rightarrow [a], a.$
 $a \rightarrow [] .$
 $b \rightarrow [b], b.$
 $b \rightarrow [] .$
 $c \rightarrow [c], c.$
 $c \rightarrow [] .$

```
s(Xs) :- append(As,BsCs,Xs), append(Bs,Cs,BsCs),
         a(As), b(Bs), c(Cs).

a(Xs) :- Xs=[a|As], a(As).
a([]).

b(Xs) :- Xs=[b|Bs], b(Bs).
b([]).

c(Xs) :- Xs=[c|Cs], c(Cs).
c([]).
```

- Com listas de diferença:

```
s(As\Xs) :- a(As\Bs), b(Bs\Cs), c(Cs\Xs).

a(Xs\Ys) :- Xs=[a|Xs1], a(Xs1\Ys).
a(Xs\Xs).

b(Xs\Ys) :- Xs=[b|Xs1], b(Xs1\Ys).
b(Xs\Xs).

c(Xs\Ys) :- Xs=[c|Xs1], c(Xs1\Ys).
c(Xs\Xs).
```

Meta-Interpretadores

- Um **meta-interpretador** de uma linguagem é um interpretador da linguagem escrito na própria linguagem
- Em Prolog, é particularmente fácil construir meta-interpretadores porque não há distinção entre programa e dados
- Interesse em desenvolver meta-interpretadores:
 - implementar diferentes estratégias de pesquisa da solução
 - incluir capacidades de explicação
 - incluir facilidades acrescidas de traçagem, teste e “debugging”
- Um meta-interpretador para Prolog puro (sem *cuts*):

```
solve(true).  
solve((A,B)) ← solve(A), solve(B).  
solve(A) ← clause(A,B), solve(B).  
  
solve(A) ← builtin(A), A.
```

Meta-Interpretadores (2)

- Com traçagem:

```
solve_trace(Goal) ← solve_trace(Goal,0).  
solve_trace(true,Depth) ← !.  
solve_trace((A,B),Depth) ←  
    !, solve_trace(A,Depth), solve_trace(B,Depth).  
solve_trace(A,Depth) ←  
    builtin(A), !, A, display(A,Depth), nl.  
solve_trace(A,Depth) ←  
    clause(A,B), display(A,Depth), nl, Depth1 is Depth + 1,  
    solve_trace(B,Depth1).  
display(A,Depth) ←  
    Spacing is 3*Depth, put_spaces(Spacing), write(A).  
put_spaces(N) ←  
    between(1,N,I), put_char(' '), fail.  
put_spaces(N).
```

- Geração da árvore de prova:

```
solve(true,true) ← !.  
solve((A,B),(ProofA,ProofB)) ←  
    !, solve(A,ProofA), solve(B,ProofB).  
solve(A,(A←builtin)) ← builtin(A), !, A.  
solve(A,(A←Proof)) ← clause(A,B), solve(B,Proof).
```

Definição de Operadores

- Nome (um átomo), tipo (classe e associatividade) e prioridade (inteiro entre 1 e 1200)
- Tipos de operadores:

Specifier	Class	Associativity
fx	prefix	non-associative
prefix	right-associative	
xfx	infix	non-associative
xfy	infix	right-associative
yfx	infix	left-associative
xf	postfix	non-associative
yf	postfix	left-associative

- Definir operadores:
:- op(Prioridade, Tipo, Nome) .

Definição de Operadores (Exemplo)

```
:- op(900, xfy, opa).  
:- op(800, yfx, opb).  
    - 1 opa 2 opb 3 ⇔ (1 opa (2 opb 3))  
        • opa tem prioridade mais elevada do que opb  
    - 1 opa 2 opa 3 ⇔ (1 opa (2 opa 3))  
        • opa é um operador associativo à direita
```

- Verificação:

```
X opa Y :- write(X:Y).  
X opb Y :- write(X:Y).
```

```
| ?- 1 opa 2 opb 3.  
    1 : (2 opb 3)  
    yes  
| ?- 1 opa 2 opa 3.  
    1 : (2 opa 3)  
    yes  
| ?- 1 opb 2 opb 3.  
    (1 opb 2) : 3  
    yes
```

Operadores Predefinidos

Priority	Specifier	Operator(s)
1200	xfx	(:- -->)
1200	fx	: -
1100	xfy	; ,
1000	xfy	,
700	xfx	= \=
700	xfx	== \==
700	xfx	= . .
700	xfx	is =:= =\= < <= > >=
500	yfx	+ -
400	yfx	* /
200	xfy	& ^
200	fy	-

Programa genérico para Jogos

```
play(Game) ←  
    initialize(Game,Position,Player),  
    display_game(Position,Player),  
    play(Position,Player,Result).  
  
play(Position,Player,Result) ←  
    game_over(Position,Player,Result), !, announce(Result).  
play(Position,Player,Result) ←  
    choose_move(Position,Player,Move),  
    move(Move,Position,Position1),  
    display_game(Position1,Player),  
    next_player(Player,Player1),  
    !, play(Position1,Player1,Result).
```

Escolher a melhor jogada

```
choose_move(Position,computer,Move) ←  
    findall(M,move(Position,M),Moves),  
    evaluate_and_choose(Moves,Position,(nil,-1000),Move).  
  
evaluate_and_choose([Move|Moves],Position,Record,BestMove) ←  
    move(Move,Position,Position1),  
    value(Position1,Value),  
    update(Move,Value,Record,Record1),  
    evaluate_and_choose(Moves,Position,Record1,BestMove).  
evaluate_and_choose([],Position,(Move,Value),Move).  
  
update(Move,Value,(Move1,Value1),(Move1,Value1)) ←  
    Value ≤ Value1.  
update(Move,Value,(Move1,Value1),(Move,Value)) ←  
    Value > Value1.
```