PROGRAMAÇÃO EM LÓGICA

CONCEITOS

Henrique Lopes Cardoso, DEI/FEUP, 2013

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

mother(sarah,isaac).
```

```
male(terach).
male(abraham).
male(nachor).
male(haran).
male(isaac).
male(lot).

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

1ª 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**

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

– Resposta negativa

- Significa que o facto não é uma consequência lógica do programa
- Mas não se sabe se é verdade ou não!

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`

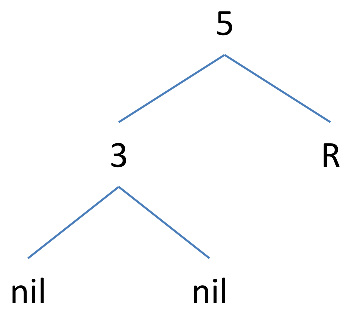
2ª Regra de Dedução – ***Generalização***: de *P*θ deduzir *P?*

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: $f(t_1, t_2, \ldots, t_n)$
    - Nome do functor: f
    - Aridade: n
    - Argumentos: $t_1, t_2, \ldots, t_n$
  - 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 = father(abraham,X), θ = {X=isaac}, $A\theta$ = father(abraham,isaac)

- *Instância*
  - A é uma instância de B se houver uma substituição θ tal que $A=B\theta$
    - father(abraham,isaac) é uma instância de father(abraham,X)

Soluções

- Uma *solução* de uma pergunta existencial é uma sua *instância*
- Uma pergunta existencial pode ter várias soluções

  `father(haran,X)?`

  – Soluções: {X=lot}, {X=milcah}, {X=yiscah}

- Outro exemplo:

```
plus(0,0,0).        plus(1,1,2).        plus(0,3,3).
plus(1,0,1).        plus(0,2,2).        plus(1,3,4).
plus(0,1,1).        plus(1,2,3).        ...
```

  `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: *$p(T_1, \ldots, T_n)$.*
    - Para todos os $X_1,\ldots,X_k$, que são variáveis que ocorrem no facto, $p(T_1,\ldots,T_n)$ é verdade

3ª 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 sintacticamente idêntico a $B\theta_2$
  - Responder a pergunta existencial com base em facto universal: Instanciação + Generalização

```
plus(0,3,Y)?
```

$$\boxed{\texttt{plus(0,X,X).}} \xrightarrow[\{X=3\}]{\text{Instanciação}} \underbrace{\boxed{\texttt{plus(0,3,3).}}}_{\text{instância comum}} \xrightarrow[\{Y=3\}]{\text{Generalização}} \boxed{\texttt{plus(0,3,Y)?}}$$

Perguntas Conjuntivas

- Conjunção de objectivos

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

  – a vírgula corresponde ao "e" lógico

- *Variáveis partilhadas*
  – Variáveis que ocorrem em dois objectivos diferentes na pergunta
    - **`father(haran,X), male(X)?`**
      – Existe um X tal que father(haran,X) e male(X) sejam ambos verdade?

  – O alcance de uma variável partilhada é toda a pergunta conjuntiva

- Uma pergunta conjuntiva $A_1, A_2, \ldots, A_n$ 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

  ```
  son(X,Y) ← father(Y,X), male(X).
  ```
  - no fundo, damos um nome a uma pergunta conjuntiva

- Forma genérica das cláusulas de Horn: $A \leftarrow B_1, B_2, \ldots, B_n.$
  - Cabeça da regra: A
  - Corpo da regra: $B_1, B_2, \ldots, B_n$
  - Casos especiais:
    - Facto (n=0): $A.$
    - Pergunta: $\leftarrow B_1, B_2, \ldots, B_n.$ (≡ $B_1, B_2, \ldots, B_n?$)

- Leitura procedimental
  - Uma regra permite expressar perguntas complexas a partir de perguntas mais simples
    - son(X,haran)? é traduzido para father(haran,X), 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

  `grandfather(X,Y) ← father(X,Z), 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:

$$\forall X \forall Y \forall Z \; grandfather(X,Y) \leftarrow father(X,Z),\, father(Z,Y). \quad \equiv$$
$$\forall X \forall Y \forall Z \; grandfather(X,Y) \vee \neg[\, father(X,Z),\, father(Z,Y)\,]. \quad \equiv$$
$$\forall X \forall Y \; grandfather(X,Y) \vee \forall Z \, \neg[\, father(X,Z),\, father(Z,Y)\,]. \quad \equiv$$
$$\forall X \forall Y \; grandfather(X,Y) \vee \neg\exists Z \, [\, father(X,Z),\, father(Z,Y)\,]. \quad \equiv$$
$$\forall X \forall Y \; grandfather(X,Y) \leftarrow \exists Z \, [\, father(X,Z),\, father(Z,Y)\,].$$

Modus Ponens

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

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

- Da regra $R = (A \leftarrow B_1, B_2,\ldots, B_n)$
- e dos factos $B_1'.$ $B_2'.$ … $B_n'.$
- pode-se deduzir A' se $A' \leftarrow B_1', B_2',\ldots, B_n'$ for uma instância de R

- Um objectivo G quantificado existencialmente (i.e., no corpo de uma cláusula) é uma <u>consequência lógica</u> de um programa P se houver uma cláusula em P que tenha uma instância sem variáveis da forma

$$A \leftarrow B_1, B_2, \ldots, B_n$$

tal que $B_1, B_2, \ldots, B_n$ são <u>consequências lógicas</u> 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,nachor).
father(terach,haran).
father(abraham,isaac).
father(haran,lot).
father(haran,milcah).
father(haran,yiscah).

mother(sarah,isaac).
```

```
male(terach).
male(abraham).
male(nachor).
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(S,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 $A_1 \leftarrow B_1, B_2, \ldots, B_n$ 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

```
abraham ≠ isaac.     abraham ≠ haran.    abraham ≠ lot.
abraham ≠ milcah.    abraham ≠ yiscah.   isaac ≠ haran.
isaac ≠ lot.         isaac ≠ milcah.     isaac ≠ yiscah.
haran ≠ lot.         haran ≠ milcah.     haran ≠ yiscah.
lot ≠ milcah.        lot ≠ yiscah.       milcah ≠ yiscah.
```

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

- Outra versão não recursiva linear, com o mesmo significado

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

Álgebra Relacional e PL

- União

$$\texttt{r\_union\_s}(X_1, \ldots, X_n) \leftarrow \texttt{r}(X_1, \ldots, X_n).$$
$$\texttt{r\_union\_s}(X_1, \ldots, X_n) \leftarrow \texttt{s}(X_1, \ldots, X_n).$$

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

$$\texttt{r\_diff\_s}(X_1, \ldots, X_n) \leftarrow \texttt{r}(X_1, \ldots, X_n),\ \texttt{not}\ \texttt{s}(X_1, \ldots, X_n).$$

- Produto cartesiano

$$\texttt{r\_x\_s}(X_1, \ldots, X_m, X_{m+1}, \ldots, X_{m+n}) \leftarrow \texttt{r}(X_1, \ldots, X_m),\ \texttt{s}(X_{m+1}, \ldots, X_{m+n}).$$

- Projecção

$$\texttt{r13}(X_1, X_3) \leftarrow \texttt{r}(X_1, X_2, X_3).$$

- Selecção

$$\texttt{r1}(X_1, X_2, X_3) \leftarrow \texttt{r}(X_1, X_2, X_3), X_3 > X_2.$$

- Intersecção

$$\texttt{r\_meet\_s}(X_1, \ldots, X_n) \leftarrow \texttt{r}(X_1, \ldots, X_n),\ \texttt{s}(X_1, \ldots, X_n).$$

- Junção

$$\texttt{r\_join\_s}(X_1, X_2, X_3) \leftarrow \texttt{r}(X_1, X_2),\ \texttt{s}(X_2, X_3).$$

Tipos

- Um tipo é um conjunto (possivelmente infinito) de termos

- Tipos definidos com base em relações unárias
  - `p/1   male/1    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))),`...
  - $s^n(0)$ 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))}

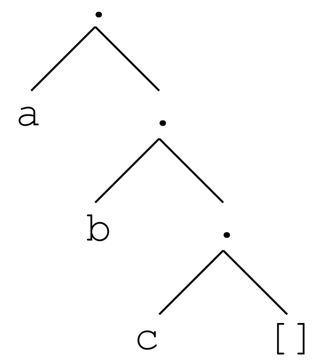
    - `plus(X2,Y,s(0))?` {X1=s(X2)}

      {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)`** ≡ **`[X|Y]`**

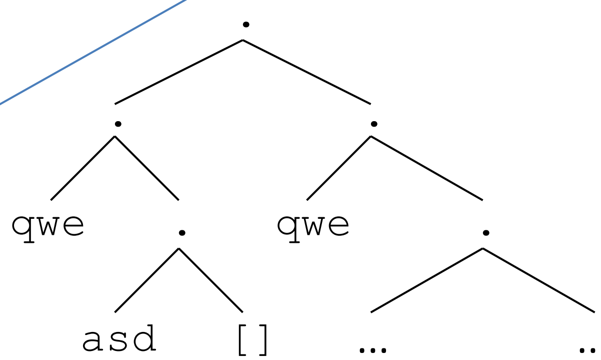
Formal object	Cons pair syntax	Element syntax
.(a,[ ])	[a|[ ]]	[a]
.(a,.(b,[ ]))	[a|[b|[ ]]]	[a,b]
.(a,.(b,.(c,[ ])))	[a|[b|[c |[ ]]]]	[a,b,c]
.(a,X)	[a|X]	[a|X]
.(a,.(b,X))	[a|[b|X]]	[a,b|X]



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]`
      - …



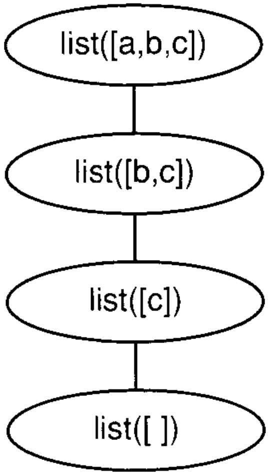
Definição de Lista

```
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).
```

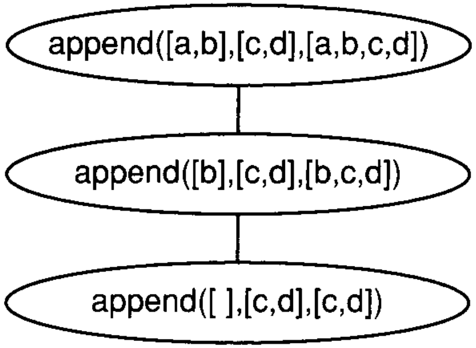
sublist([b,c],[a,b,c,d])?

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).
```

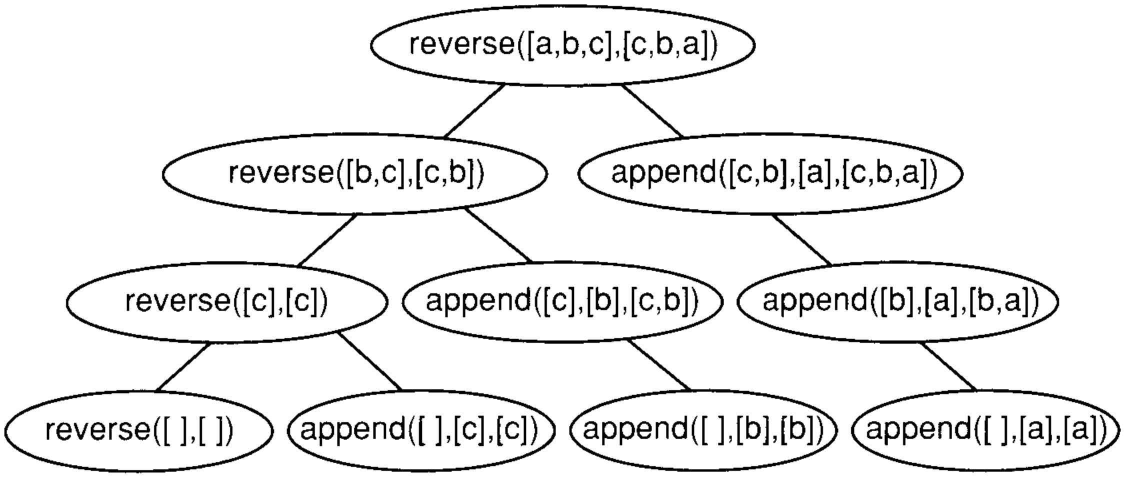
```
sublist(Xs,AsXsBs) ←
    append(As,XsBs,AsXsBs), append(Xs,Bs,XsBs).
```

```
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 <u>quadrático</u> no número de elementos da lista a inverter

Inversão de uma Lista (2)

- Evitar append:

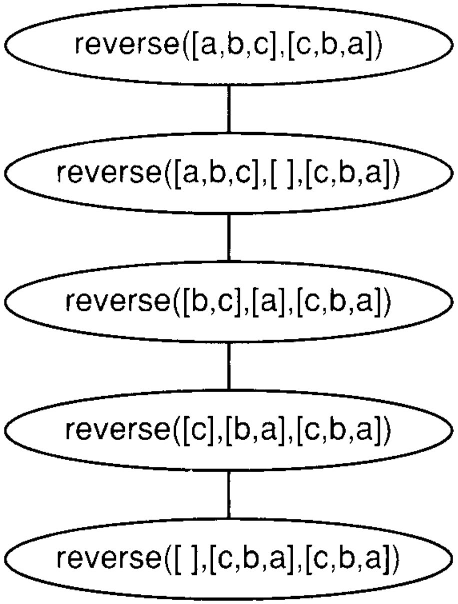
```
reverse(Xs,Ys) ← reverse(Xs,[ ],Ys).

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

  - tamanho da árvore de prova é <u>linear</u> 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≠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`

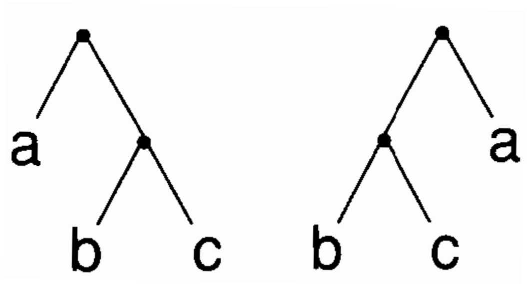
```
  a
 / \
b   c
```

```
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*DG+DF*G) :- derivative(F,DF), derivative(G,DG).
derivative(s(0)/F,-DF/(F*F)) :- derivative(F,DF).
derivative(F/G,(G*DF-F*DG)/(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}

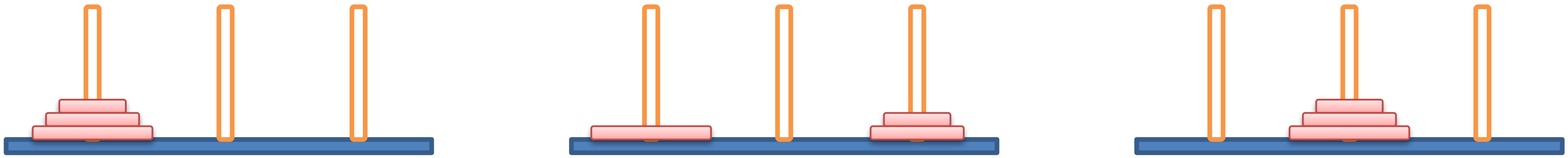
Manipulação de Expressões Simbólicas

- 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

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

teste de ocorrência
(e.g. *X* não unifica com *s(X)*)

```
initialize θ to empty
push T1=T2 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 θ
                        add X=Y to θ
                Y is a variable that does not occur in X:
                        …
                X and Y are identical constants or variables:
                        continue
                X is f(X1, …, Xn) and Y is f(Y1, …, Yn), for some functor f
                        push Xi=Yi, i=1…n, on the stack
                otherwise:
                        return failure
return θ
```

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 $A' \leftarrow B_1, \ldots, B_n$ from P such that A and A' unify with *mgu* θ
        (if no such clause exists, return *no*)
    replace A by $B_1, \ldots, B_n$ in the *resolvent*
    apply θ to the *resolvent* and to G
return G

Traçado

```
father(abraham,isaac).        male(isaac).        son(X,Y) ← father(Y,X), male(X).
father(haran,lot).            male(lot).          daughter(X,Y) ← father(Y,X), female(X).
father(haran,milcah).         female(milcah).
father(haran,yiscah).         female(yiscah).
```

- `son(S,haran)?`

Resolvente: son(S, haran)
  Escolhe son(S, haran)
  Escolhe **son(S, haran) ← father(haran, S), male(S).** {X=S, Y=haran}
  Resolvente: father(haran, S), male(S)
    Escolhe father(haran, S)
    Escolhe **father(haran, lot).** {S=lot}
    Resolvente: male(lot)
      Escolhe male(lot)
      Escolhe **male(lot).**
      Resolvente vazia

  Escolhe male(S)
  Escolhe **male(lot).** {S=lot}
  Resolvente: father(haran, lot)
    Escolhe father(haran, lot)
    Escolhe **father(haran, 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]
             ⋮                          ⋮
```

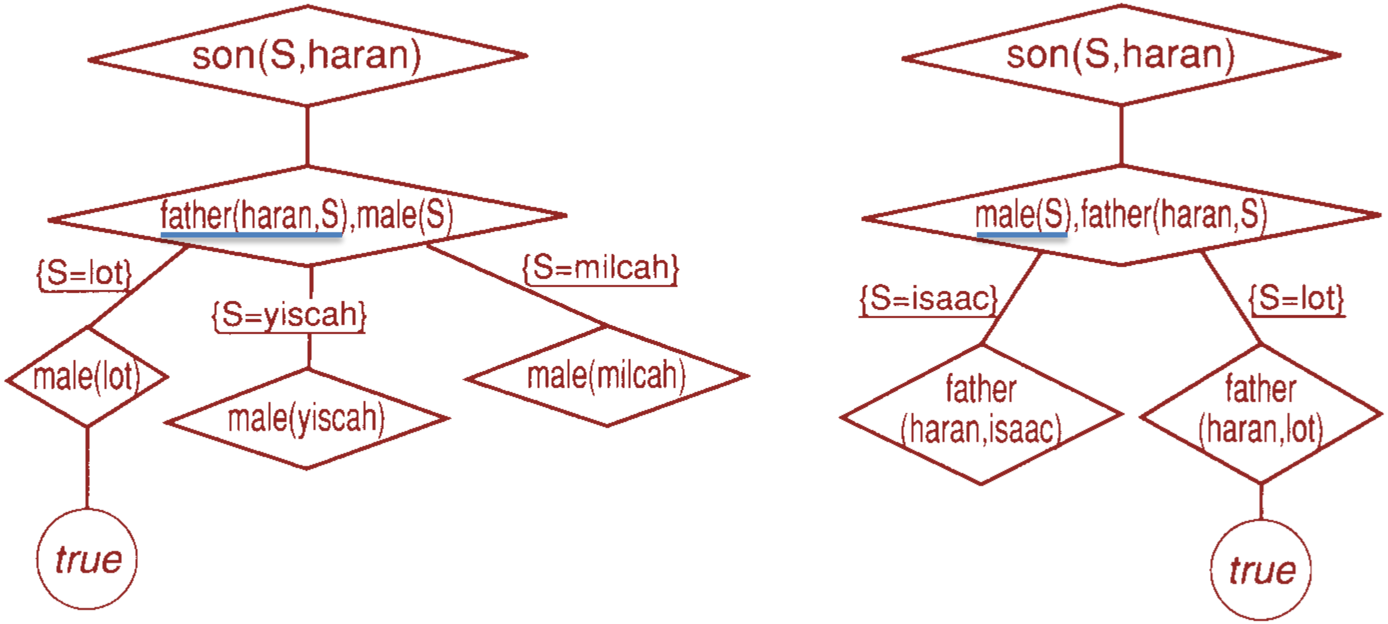
Árvores de Pesquisa

- 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

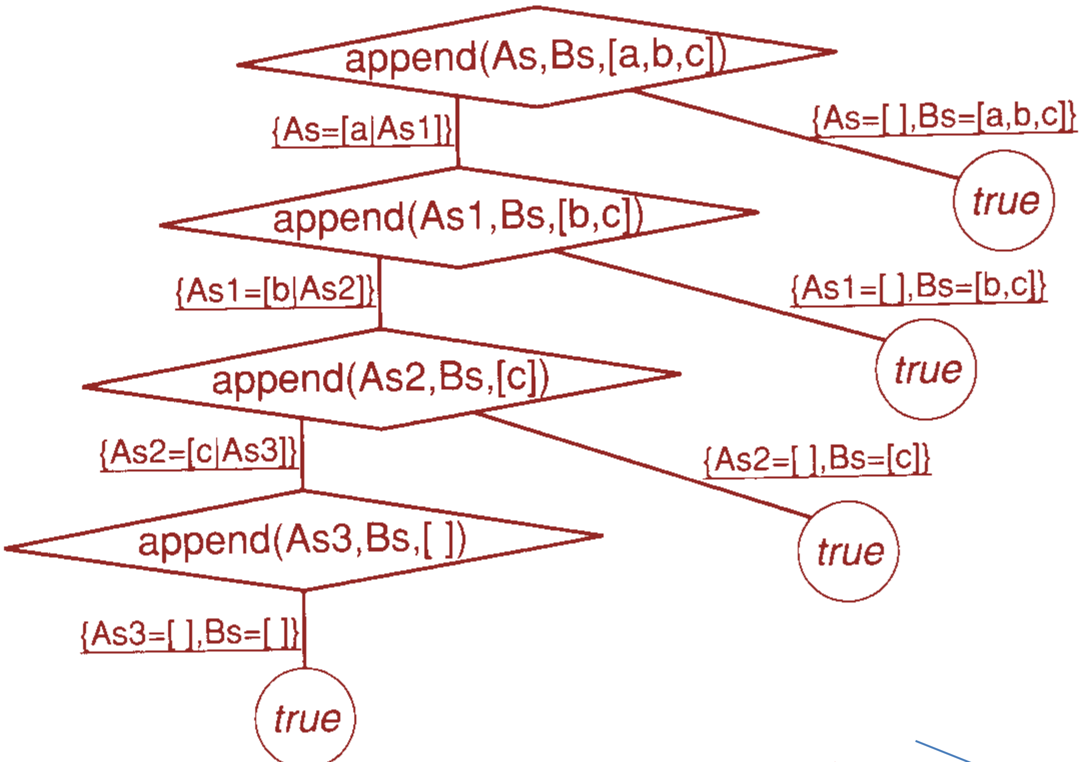
- **Árvore de pesquisa** porque um *interpretador concreto* terá uma estratégia para percorrer a árvore em busca de soluções
  - pesquisa em profundidade, em largura, em paralelo, …

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

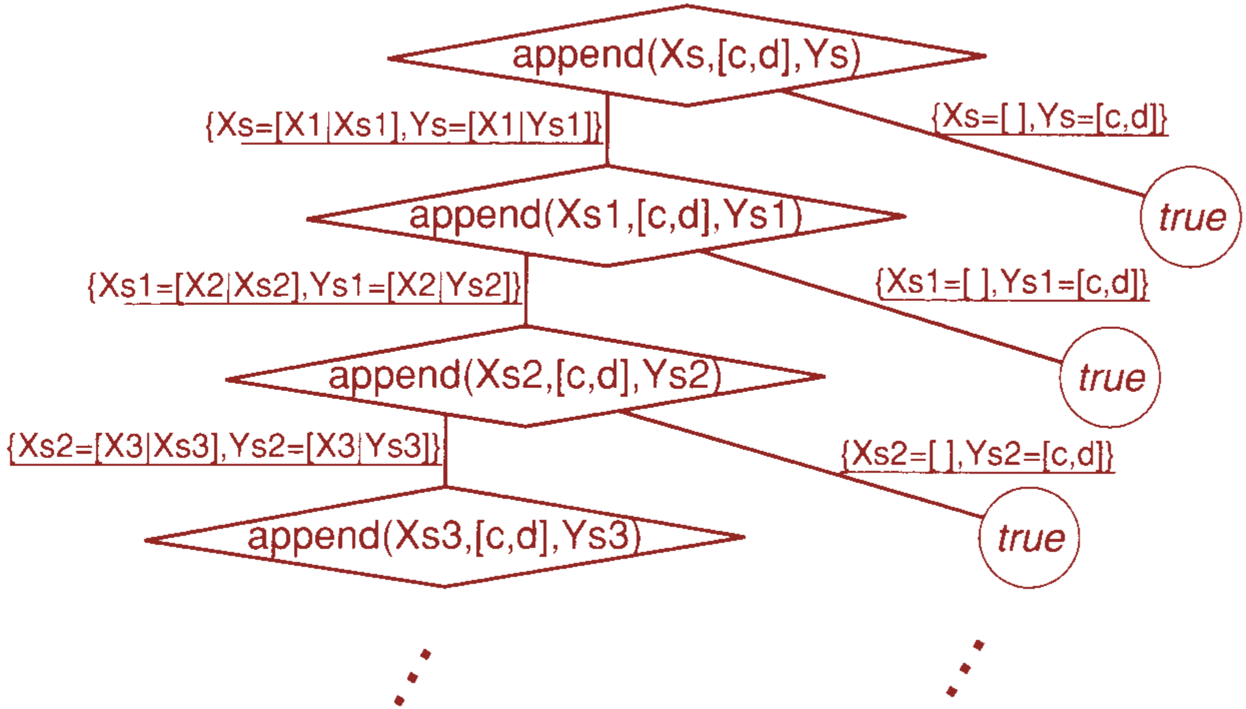


{As=[a|As1]}
{As1=[b|As2]}
{As2=[c|As3]}
{As3=[], Bs=[]}

{As=[a,b,c], Bs=[]}

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) ← male(X), **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) ← not (member(X,Xs), member(X,Ys)).
```