



»Лекционен курс

»Интелигентни системи



# Изводи в предикатната логика



# Увод

- » В тази лекция ще разгледаме концепции, които лежат в основата на **съвременните системи за извод**.
- » Започваме с някои прости правила за извод, които могат да бъдат приложени към съждения **с квантори**, за да се получат съждения **без квантори**.
- » Тези правила естествено водят до идеята, че изводът може да бъде направен чрез **преобразуване** на предикатната база знания в съждителна логика и използване на съждителен извод, който вече знаем как да направим.
- » След това ще се запознаем с **пряк път**, водещ до методи за извод, които манипулират директно съждения в предикатната логика.

# Елиминирание на квантори

## » Идея:

- > Предикатната база знания (ПБЗ) се трансформира в съждителна база знания (СБЗ)
- > В трансформираната база знания (СБЗ) и се прилагат познатите методи за извод

## » Необходимо:

- > Универсално инстанциране
- > Екзистенциално инстанциране

# Пример

Един от начините да се направи предикатен извод е да се преобразува ПБЗ в СБЗ.

Първата стъпка е премахването на универсалните квантори.

Напр., да предположим, че нашата ПБЗ съдържа стандартната фолклорна аксиома, че “всички алчни крале са зли”:

$$\forall x \text{ King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x).$$

Тогава, можем да изведем следните съждения:

$$\begin{aligned} &\text{King}(\text{John}) \wedge \text{Greedy}(\text{John}) \Rightarrow \text{Evil}(\text{John}) \\ &\text{King}(\text{Richard}) \wedge \text{Greedy}(\text{Richard}) \Rightarrow \text{Evil}(\text{Richard}) \\ &\text{King}(\text{Father}(\text{John})) \wedge \text{Greedy}(\text{Father}(\text{John})) \Rightarrow \text{Evil}(\text{Father}(\text{John})). \\ &\vdots \end{aligned}$$



# Универсално инстанциране

- **Правило за универсално инстанциране (UI):** можем да изведем всяко съждение, което получаваме замествайки **променлива с базов терм**
- **Базов терм:** терм без променливи
- **За формално представяне на правилото:** използваме концепцията за субституция
- **Субституция:** със  $\text{SUBST}(\theta, \alpha)$  означаваме резултата от прилагане на субституцията  $\theta$  върху израз  $\alpha$ , където всяка променлива  $v$  се замества с базовия терм  $g$

|                                 |
|---------------------------------|
| $\forall v \alpha$              |
| $\text{SUBST}(\{v/g\}, \alpha)$ |

# Пример (коректен)

За примера трите съждения са получени при сибституции  
 $\{x/\text{John}\}$ ,  $\{x/\text{Richard}\}$ ,  $\{x/\text{Father}(\text{John})\}$

$$\forall x \text{ King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x).$$

$$\text{King}(\text{John}) \wedge \text{Greedy}(\text{John}) \Rightarrow \text{Evil}(\text{John})$$

$$\text{King}(\text{Richard}) \wedge \text{Greedy}(\text{Richard}) \Rightarrow \text{Evil}(\text{Richard})$$

$$\text{King}(\text{Father}(\text{John})) \wedge \text{Greedy}(\text{Father}(\text{John})) \Rightarrow \text{Evil}(\text{Father}(\text{John})).$$

$\vdots$

# Екзистенциално инстанциране

- » **Правило за екзистенциално инстанциране:** променливата се замества с **нов константен символ**
- » По принцип, екзистенциалното съждение казва, че има някакъв обект, който удовлетворява дадено условие
- » Прилагането на правилото за екзистенциално инстанциране просто дава име на този обект
- » Съществено: това име **не трябва вече да принадлежи на друг обект**
- » За разлика от универсалното инстанциране, което може да се прилага многократно за получаване на различни изводи, екзистенциалното инстанциране с прилага само веднъж

# Екзистенциално инстанциране

- » **Формално представяне на правилото:** за всеки израз  $\alpha$ , всяка променлива  $v$  и всеки константен символ  $k$ , който не се появява на друго място в базата знания.
- » В логиката този константен символ се нарича константа на Сколем.

|                                 |
|---------------------------------|
| $\exists v \alpha$              |
| $\text{SUBST}(\{v/k\}, \alpha)$ |



# За примера

$$\exists x \text{ Crown}(x) \wedge \text{OnHead}(x, \text{John})$$

Можем да изведем следното съждение, ако  $C_1$ , не се появява никъде другаде в БЗ:

$$\text{Crown}(C_1) \wedge \text{OnHead}(C_1, \text{John})$$

# Пример

- » Ще разгледаме един интересен пример от математиката
- » Да приемем, че откриваме едно число, което:
  - > Малко по-голямо от 2.71828
  - > Удовлетворява за  $x$  уравнението  $\frac{d(x^y)}{dy} = x^y$
  - > Можем да дадем на това число име, напр.  $e$
  - > Ще бъде грешка, ако дадем на числото име  $\pi$ , понеже това е име на вече съществуващ обект

# Неперово число

- » Ирационалното число  $e = 2,718281828459045...$
- » То е една от най-важните константи в математиката
- » Възниква естествено при описанието на различни процеси в природните и обществените науки
- » Числото е в основата на естествените логаритми
- » Наречено на името на шотландския математик Джон Непер

# Преобразуване ПБЗ в СБЗ

- » Преобразуване на предикатна БЗ в съждителна БЗ
  - > Едно екзистенциално съждение може да бъде заменено с една инстанция
  - > Едно универсално съждение може да бъде заменено с множество от всички възможни инстанции

# Пример

- » Да предположим, че БЗ съдържа следните съждения, където единствените обекти са John и Richard:

$\forall x \text{ King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x)$   
 $\text{King}(\text{John})$   
 $\text{Greedy}(\text{John})$   
 $\text{Brother}(\text{Richard}, \text{John}).$

- » Прилагаме универсално инстанциране към първото съждение, използвайки всички възможни субституции  $\{x/\text{John}\}$  и  $\{x/\text{Richard}\}$
- » Получаваме като резултат:

$\text{King}(\text{John}) \wedge \text{Greedy}(\text{John}) \Rightarrow \text{Evil}(\text{John})$   
 $\text{King}(\text{Richard}) \wedge \text{Greedy}(\text{Richard}) \Rightarrow \text{Evil}(\text{Richard}).$



# Пример

- » След това заменяме базовите атомарни съждения, като King(John) със съждителени символи като JohnIsKing
- » И накрая, прилагаме някой от пълните методи за извод от СЛ за да получим заключения като JohnIsEvil, което е еквивалентно на Evil(John)

# Обобщение

- » Тази техника на трансформация може да се обобщи
- » Въпреки това, има проблем, когато базата от знания включва функционални символи
- » Множеството от възможни субституции на базови терми става безкрайно
- » Например, ако в базата знания съдържа функционалния символ `Father`, тогава могат да се конструират безкрайно много вложени термини като напр.  
`Father(Father(Father(John)))`

# Теорема на Хербранд

- » За щастие има известна теорема, дължима на Жак Хербранд (1930), според която:
  - > Ако едно съждение е изведено от оригиналната ПБД, тогава има извод, включващ само ограничено подмножество от СБД.
- » Тъй като всяко такова подмножество има максимална дълбочина на влягане в базовите терми, можем да намерим подмножеството, като:
  - > Първо генерираме всички инстанции с постоянни символи (John и Richard)
  - > След това всички терми с дълбочина 1 (Father(Richard) и Father(John))
  - > След това всички терми с дълбочина 2
  - > И така нататък, докато не сме в състояние да изградим съждителен извод на изведеното съждение

# Обобщение

- » Разгледахме подход за предикатни изводи чрез съждителна трансформация, която е **пълна** – т.е. всяко изводимо съждение може да бъде изведено
- » Това е голямо постижение, като се има предвид, че пространството на възможните модели е безкрайно
- » От друга страна, ние не знаем (докато не се изведе), че съждението е изводимо!
- » Можем ли да кажем какво се случва, когато съждението е неизводимо?
  - > За предикатната логика не можем
  - > Процедурата за извод може да продължи безкрайно, генерирайки все по-дълбоко вложени терми без да знаем дали е затънала в безнадежден цикъл или доказателството е на път да се появи

# Обобщение

- » Това много прилича на **стоп проблема** за машината на Тюринг
  - > Алън Тюринг (1936) и Алонзо Чърч (1936) доказват по доста различни начини неизбежността на това състояние на нещата
- » Въпросът за изводимост в предикатната логика е полуразрешим
  - > Т.е. съществуват алгоритми, които казват „да“ на всяко изводимо съждение, но не съществува алгоритъм, който също да казва „не“ на всяко неизводимо съждение



# Обобщение

- » Можем да отбележим, че подходът на съждителна трансформация генерира много ненужни инстанции на универсално квантифицирани съждения
- » Предпочитаме да имаме подход, който използва само едно правило, извеждащо че  $\{x/\text{John}\}$  решава въпроса по следния начин:
  - > Като се има предвид правилото, че алчните крале са зли, намерете такъв  $x$ , който е крал и е алчен и
  - > След това заключи, че тая  $x$  е зъл
- » По-общо казано, ако има някаква субституция  $\theta$ , която прави идентична (унифицира) всяка от конюнктите (които вече са в БЗ) на предпоставката на импликацията, тогава можем да потвърдим заключението на импликацията, след като приложим  $\theta$
- » В този случай субституцията  $\theta = \{x/\text{John}\}$  постига тази цел

# Обобщение

- » Сега да предположим, че вместо да знаем  $\text{Greedy}(\text{John})$ , ние знаем, че всеки е алчен:

$\forall y \text{ Greedy}(y).$

- » Тогава все пак бихме искали да можем да заключим, че  $\text{Evil}(\text{John})$ , защото знаем, че Джон е крал (дадено) и Джон е алчен (защото всички са алчни).
- » Това, от което се нуждаем, за да работи това, е да намерим една субституция както на променливите в импликация, така и на променливите в съжденията, които са в БЗ
- » В този случай прилагането на субституцията  $\{x/\text{John}, y/\text{John}\}$  към импликационните предпоставки  $\text{King}(x)$  и  $\text{Greedy}(x)$  и съжденията в БЗ  $\text{King}(\text{John})$  и  $\text{Greedy}(y)$  ще ги направи идентични (унифицират)
- » По този начин можем да заключим следствието от импликацията

# Унификация и повдигане (lifting)

- » До сега разгледахме изводи в ПЛ, които съществуваха до 60-те години на миналия век
- » Съществен недостатък на трансформацията в СЛ:
  - > Неефективен метод
- » Могат да се въведат нови методи, използващи повдигане
  - > Или обобщен модус поненс

# Обобщен модус поненс

**Обобщен модус поненс:** ако за атомарните съждения  $p_i$ ,  $p_i'$  и  $q$  съществува субституция  $\theta$  със  $\text{SUBST}(\theta, p_i) = \text{SUBST}(\theta, p_i')$  за всички  $i$ , тогава важи:

$$\frac{p_1', p_2', \dots, p_n', (p_1 \wedge p_2 \wedge \dots \wedge p_n \Rightarrow q)}{\text{SUBST}(\theta, q)}.$$

Има  $n+1$  предпоставки за това правило: атомарните съждения и единствената импликация. Заключение е резултат от прилагането на субституцията  $\theta$  към следствието  $q$

# За примера: следната БЗ

$\forall x \text{ King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x)$   
 $\text{King}(\text{John})$

$\forall y \text{ Greedy}(y).$

$p_1'$  is  $\text{King}(\text{John})$

$p_1$  is  $\text{King}(x)$

$p_2'$  is  $\text{Greedy}(y)$

$p_2$  is  $\text{Greedy}(x)$

$\theta$  is  $\{x/\text{John}, y/\text{John}\}$

$q$  is  $\text{Evil}(x)$

$\text{SUBST}(\theta, q)$  is  $\text{Evil}(\text{John})$ .



# Обобщен модус поненс

- » Generalized Modus Ponens е повдигната версия на Modus Ponens — издига Modus Ponens от базова (без променливи) съждителна логика до предикатана логика
- » Основното предимство на повдигнатите правила за извод пред съждителната трансформация е, че те правят **само** онези замествания (сунцтитущии), които са необходими за да позволи да се правят конкретни изводи

# Унификация

- » Повдигнатите правила за извод изискват намиране на субституции, които правят различните логически изрази да изглеждат идентични
- » Този процес се нарича **унификация** и е ключов компонент на всички алгоритми за изводи в предикатната логика

# Унификация

**Унификация:** процесът за намиране на субституции за правилата за извод, които предизвикват **различни** логически изрази да изглеждат **идентични**

- Ключов компонент на всички алгоритми за извод, базирани на ПЛ
- Алгоритъмът Unify приема два изрази и връща като резултат един унификатор (ако съществува такъв)
- Могат да съществуват повече унификатора

$$\text{Unify}(p, q) = \theta, \text{ където } \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q)$$

# Пример

- Нека разгледаме някои примери за това как трябва да се реализира UNIFY
- Да предположим, че имаме въпрос  $\text{AskVars}(\text{Knows}(\text{John}, x))$ : кого познава Джон?
- Отговорите на този въпрос могат да бъдат намерени чрез търсене в БЗ на всички твърдения, които се унифицират с  $\text{Knows}(\text{John}, x)$
- Резултати:

$$\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(\text{John}, \text{Jane})) = \{x/\text{Jane}\}$$
$$\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(y, \text{Bill})) = \{x/\text{Bill}, y/\text{John}\}$$
$$\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(y, \text{Mother}(y))) = \{y/\text{John}, x/\text{Mother}(\text{John})\}$$
$$\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(x, \text{Elizabeth})) = \text{fail} .$$

Последната унификация не успява, защото  $x$  **не може да има в едно също време** стойностите John и Elizabeth

# Свързване напред

- » Идеята за свързване напред е проста:
  - > Започваме с атомарните съждения в БЗ и прилагаме Modus Ponens в посока напред, като добавяме нови атомарни съждения, докато не могат да се правят повече изводи
- » Тук ще представим как алгоритъмът се прилага към дефинитни клаузи от ПЛ



# Дефинитни клаузи

- » Дефинитните клаузи в ПЛ са дизюнкция от литерали, от които точно един е положителен
- » Една дефинитна клауза е или атомарна или е импликация, лявата част на която е конюнкция от положителни литерали, а дясната част е единичен положителен литерал
- » Пример:

# Дефинитни клаузи

- » За разлика от СЛ, литералите в ПЛ могат да включват променливи, като в този случай се приема, че те са универсално квалифицирани
- » Не всяка БЗ може да бъде преобразувана в набор от дефинитни клаузи, поради ограничението за единствен положителен литерал, но много могат

# Пример: търговия с оръжие

- Според закона, за американците е престъпление да продават оръжие на враждебни нации
- Страната Nono, която е враг на Америка, има няколко ракети и всичките те са продадени от Colonel West, който е американец
- Искаме да представим тези факти като дефинитни клаузи в ПЛ

# Пример: търговия с оръжие

“... престъпление е за един американец да продава оръжие на враждебни нации”:

$$\text{American}(x) \wedge \text{Weapon}(y) \wedge \text{Sells}(x, y, z) \wedge \text{Hostile}(z) \Rightarrow \text{Criminal}(x)$$

1

“Nono ... има ракети”: съждението  $\exists x \text{ Owns}(\text{Nono}, x) \wedge \text{Missile}(x)$  се трансформира в две дефинитни клаузи чрез екзистенциално инстанциране, въвеждайки нова константа  $M_1$

$$\begin{array}{l} \text{Owns}(\text{Nono}, M_1) \\ \text{Missile}(M_1) \end{array}$$

“... Всичките ракети бяха продадени от Colonel West”:

$$\text{Missile}(x) \wedge \text{Owns}(\text{Nono}, x) \Rightarrow \text{Sells}(\text{West}, x, \text{Nono})$$

2



# Пример: търговия с оръжие

Също така трябва да знаем, че ракетите са оръжия:

$$\text{Missile}(x) \Rightarrow \text{Weapon}(x)$$

3

И трябва да знаем, че врагът на Америка се счита за "враждебен":

$$\text{Enemy}(x, \text{America}) \Rightarrow \text{Hostile}(x)$$

4

"West, който е американец ...":

$$\text{American}(\text{West})$$

"Страната Nono, враг на Америка ...":

$$\text{Enemy}(\text{Nono}, \text{America})$$



# Свързване напред

» Алгоритъм за свързване напред:

- > Изхождайки от известните факти, активира всички правила, чиито предпоставки са изпълнени, като добавя своите заключения към известните факти
- > Процесът се повтаря, докато се отговори на заявката (ако се приеме че се изисква само един отговор) или не се добавят нови факти

# Алгоритъм

```
function FOL-FC-ASK( $KB, \alpha$ ) returns a substitution or false
  inputs:  $KB$ , the knowledge base, a set of first-order definite clauses
          $\alpha$ , the query, an atomic sentence
  local variables:  $new$ , the new sentences inferred on each iteration

  repeat until  $new$  is empty
     $new \leftarrow \{ \}$ 
    for each  $rule$  in  $KB$  do
       $(p_1 \wedge \dots \wedge p_n \Rightarrow q) \leftarrow \text{STANDARDIZE-VARIABLES}(rule)$ 
      for each  $\theta$  such that  $\text{SUBST}(\theta, p_1 \wedge \dots \wedge p_n) = \text{SUBST}(\theta, p'_1 \wedge \dots \wedge p'_n)$ 
        for some  $p'_1, \dots, p'_n$  in  $KB$ 
           $q' \leftarrow \text{SUBST}(\theta, q)$ 
          if  $q'$  does not unify with some sentence already in  $KB$  or  $new$  then
            add  $q'$  to  $new$ 
             $\phi \leftarrow \text{UNIFY}(q', \alpha)$ 
            if  $\phi$  is not fail then return  $\phi$ 
    add  $new$  to  $KB$ 
  return false
```

# Коментар

- » Изхождайки от известните факти, той задейства всички правила, чиито предпоставки са удовлетворени, добавяйки своите заключения към известните факти
- » Процесът се повтаря, докато се отговори на запитването (при условие, че е необходим само един отговор) или не се добавят нови факти
- » Забележете, че един факт не е „нов“, ако е просто преименуване на известен факт — едно съждение е преименуване на друго, ако са идентични, с изключение на имената на променливите
- » Например, Likes(x,IceCream) и Likes(y,IceCream) са взаимни преименувания – двете съждения означават едно и също нещо: „Всички обичат сладолед“
- » Функцията STANDARDIZE-VARIABLES заменя всички променливи в своите аргументи с нови, които не са били използвани преди

# Пример

## Първа итерация:

Правило **1** остава с неудовлетворени предусловия (премиси)

Правило **2** е удовлетворено с  $\{x/M_1\}$  *Sells*(*West*, *M<sub>1</sub>*, *Nono*)  
и се добавя към БЗ

Правило **3** е удовлетворено с  $\{x/M_1\}$  *Weapon*(*M<sub>1</sub>*)  
и се добавя към БЗ

Правило **4** е удовлетворено с  $\{x/Nono\}$  *Hostile*(*Nono*)  
и се добавя към БЗ



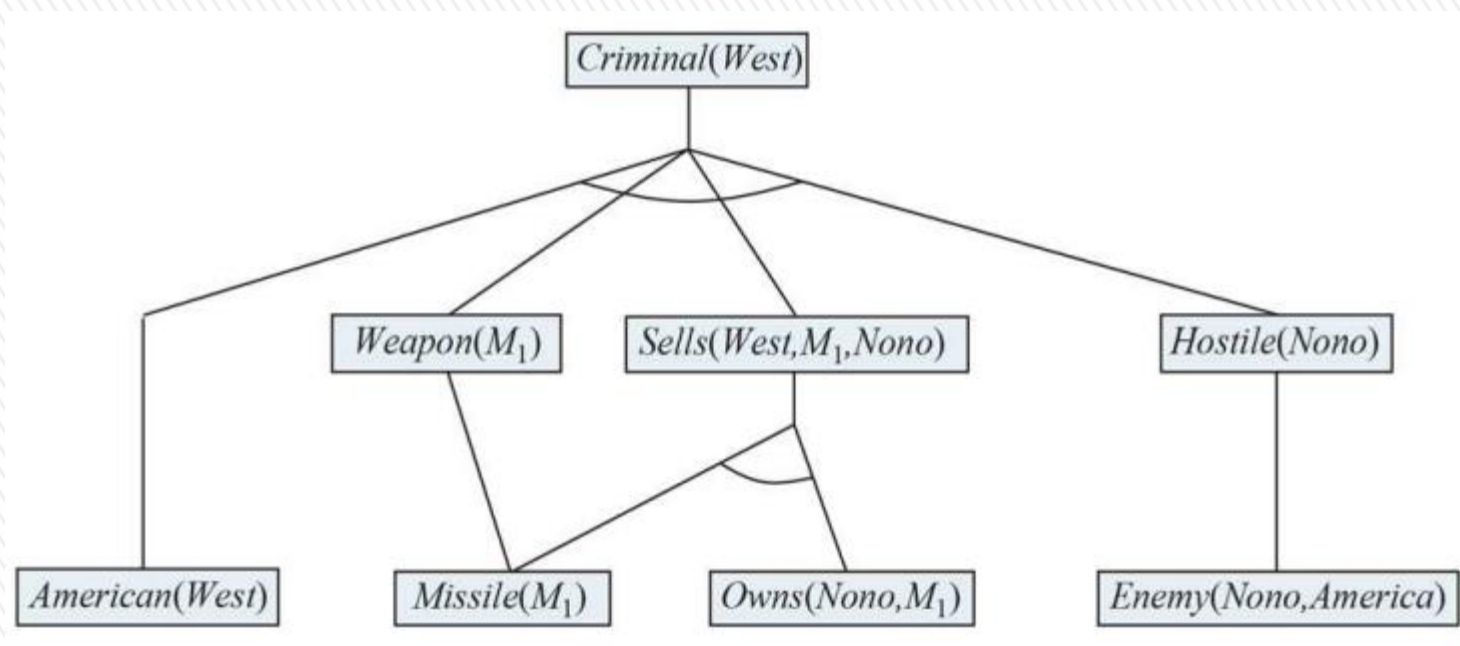
# Пример

Втора итерация:

Правило **1** е удовлетворено с  $\{x/ West, y/M_1, z/ Nono\}$ .  $Criminal( West)$   
и се добавя към БЗ



# Дърво на доказателство



# Свързване назад

- » Второто основно семейство алгоритми за логически изводи използва подхода на свързване назад
- » Тези алгоритми работят назад, започвайки от целта и свързвайки чрез правила, за да открият известни факти, които подкрепят доказателството

# Алгоритъм

```
function FOL-BC-ASK( $KB, query$ ) returns a generator of substitutions  
  return FOL-BC-OR( $KB, query, \{ \}$ )
```

---

```
generator FOL-BC-OR( $KB, goal, \theta$ ) yields a substitution  
  for each rule ( $lhs \Rightarrow rhs$ ) in FETCH-RULES-FOR-GOAL( $KB, goal$ ) do  
    ( $lhs, rhs$ )  $\leftarrow$  STANDARDIZE-VARIABLES( $(lhs, rhs)$ )  
    for each  $\theta'$  in FOL-BC-AND( $KB, lhs, UNIFY(rhs, goal, \theta)$ ) do  
      yield  $\theta'$ 
```

---

```
generator FOL-BC-AND( $KB, goals, \theta$ ) yields a substitution  
  if  $\theta = failure$  then return  
  else if LENGTH( $goals$ ) = 0 then yield  $\theta$   
  else do  
     $first, rest \leftarrow$  FIRST( $goals$ ), REST( $goals$ )  
    for each  $\theta'$  in FOL-BC-OR( $KB, SUBST(\theta, first), \theta$ ) do  
      for each  $\theta''$  in FOL-BC-AND( $KB, rest, \theta'$ ) do  
        yield  $\theta''$ 
```

# Коментар

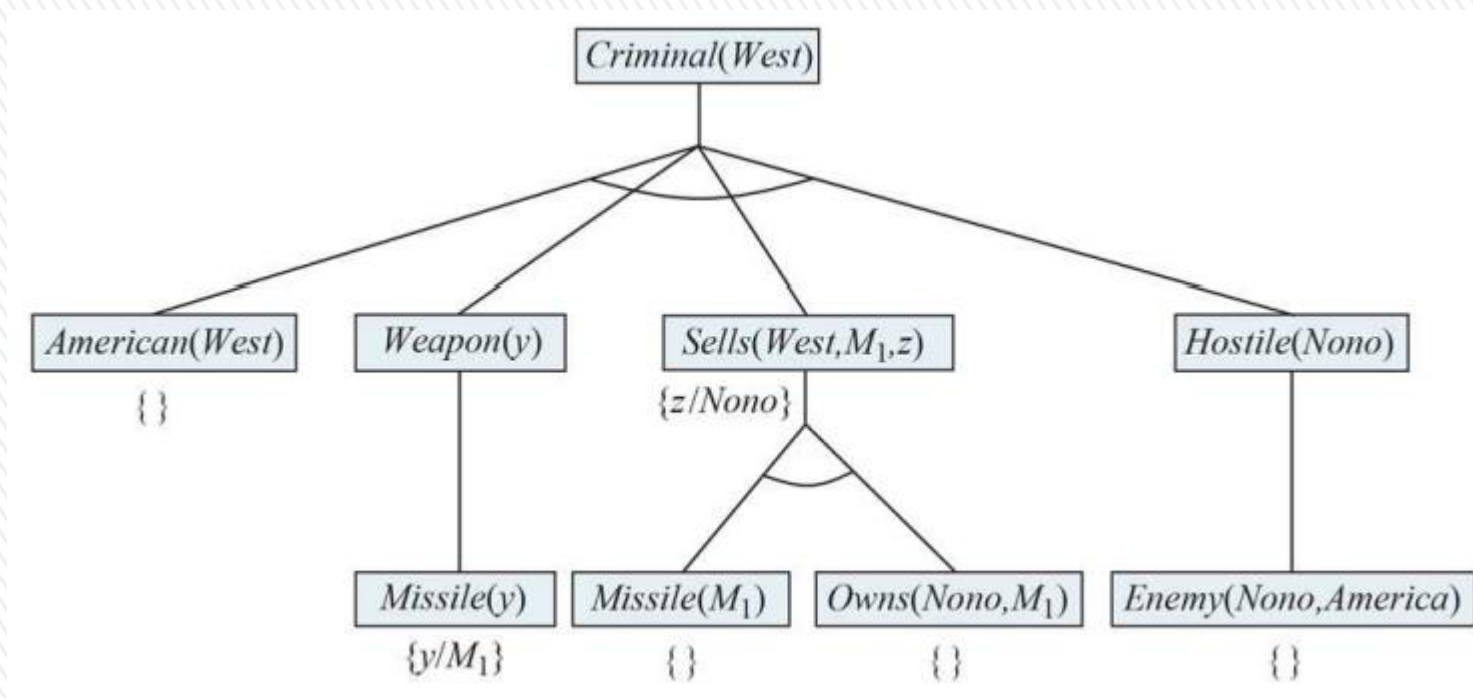
- » FOL-BC-ASK (KB, goal) ще бъде доказано, ако БЗ съдържа клауза във формата  $lhs \Rightarrow goal$ , където lhs (лявата страна) е списък от КОНЮНКТИ
- » Един атомарен факт (напр.  $American(West)$ ) се разглежда като клауза, чийто lhs е празен списък
- » Сега, една заявка, която съдържа променливи, може да се докаже по много начини
- » Напр., въпросът  $Person(x)$  може да бъде доказан при субституция  $\{x/John\}$ , както и при  $\{x/Richard\}$
- » Така че имплементираме FOL-BC-ASK като генератор - функция, която многократно връща резултат - всеки път един възможен резултат

# Коментар

- » Свързване назад е вид AND/OR търсене:
  - > OR частта, защото целевата заявка може да бъде доказана от някое правило в БЗ
  - > AND частта, защото всички конюнкти в lhs на клауза трябва да бъдат доказани
- » FOL-BC-OR работи чрез изпробване на всички клаузи, които унифицират целта, стандартизирайки променливите в клаузата да бъдат чисто нови променливи и след това, ако rhs на клаузата наистина унифицират целта, доказвайки всеки конюнкт в lhs, използвайки FOL-BC-AND
- » Тази функция на свой ред доказва последователно всеки конюнкт, отчитайки направените до момента субституции



# Дърво на доказателство



# Коментар

- » Свързването назад очевидно е алгоритъм за търсене първо в дълбочина
- » Това означава, че изискванията към пространството са линейни в размера на доказателството (пренебрегвайки, засега, пространството, необходимо за натрупване на решенията)
- » Това означава, че свързването назад (за разлика от свързването напред) страда от проблеми с повтарящи се състояния и непълнота

# Резолюция

- » Последното от трите групи логически системи и единствената, която работи за всяка БЗ (а не само за определени клаузи) е резолюцията
- » Съжителната резолюция е пълна процедура на извод – тук я разширяваме за предикатната логика
  - > Правилото за резолюция за клаузите в ПЛ е просто **обобщена версия** на правилото за резолюция в СЛ
- » Две клаузи могат да бъдат разюлирани, ако съдържат комплиментни литерали
  - > Литералите са комплиментни, ако единят е отрицание на другия
  - > Литералите в ПЛ се допълват, ако единият се **унифицира** с отрицанието на другия

# Конюнктивна нормална форма

- » Първата стъпка е конвертиране на съжденията в конюнктивна нормална форма (CNF)
  - > Както в СЛ, резолюцията в ПЛ изисква предикатите да са в конюнктивна нормална форма (CNF)
- » КНФ: конюнкция от клаузи, където всяка клауза е дизюнкция от литерали
- » Литералите могат да съдържат променливи, за които се приема, че са универсално квантифицирани
- » Всеки предикат от ПЛ може да бъде преобразуван по дедукция в еквивалентен предикат в CNF

# Пример

$$\forall x,y,z \text{ American}(x) \wedge \text{Weapon}(y) \wedge \text{Sells}(x,y,z) \wedge \text{Hostile}(z) \Rightarrow \text{Criminal}(x)$$

$$\neg \text{American}(x) \vee \neg \text{Weapon}(y) \vee \neg \text{Sells}(x,y,z) \vee \neg \text{Hostile}(z) \vee \text{Criminal}(x).$$



# Процедура за трансформация

- » По-специално, предикатите в CNF ще бъдат незадоволими само когато първоначалните предикати са незадоволими, така че имаме основание да правим доказателства чрез противоречие на предикати в CNF
- » Процедурата за трансформация в CNF е подобна на тази при СЛ
- » Основната разлика възниква от необходимостта да се елиминират екзистенциалните квантори

# Пример

Всеки, който обича всички животни, е обичан от  
някого

$$\forall x [\forall y \text{ Animal}(y) \Rightarrow \text{Loves}(x, y)] \Rightarrow [\exists y \text{ Loves}(y, x)] .$$

# Пример

1

Елиминирание на импликациите: заместваме  $P \Rightarrow Q$  с  $\neg P \vee Q$ . (трябва да го направим два пъти)

$$\forall x [\forall y \text{ Animal}(y) \Rightarrow \text{Loves}(x, y)] \Rightarrow [\exists y \text{ Loves}(y, x)] .$$



$$\begin{aligned} \forall x \quad & \neg[\forall y \text{ Animal}(y) \Rightarrow \text{Loves}(x, y)] \vee [\exists y \text{ Loves}(y, x)] \\ \forall x \quad & \neg[\forall y \neg \text{Animal}(y) \vee \text{Loves}(x, y)] \vee [\exists y \text{ Loves}(y, x)] . \end{aligned}$$



# Пример

2

Преместване  $\neg$  навътре (неявен  $\neg$ ): В допълнение към обичайните правила за отрицателни свързвания, имаме нужда от правила за отрицателни квантори. По този начин имаме:

|                   |         |                      |
|-------------------|---------|----------------------|
| $\neg\forall x p$ | becomes | $\exists x \neg p$   |
| $\neg\exists x p$ | becomes | $\forall x \neg p$ . |

Нашето съждение преминава през следните трансформации:

$$\begin{aligned} & \forall x [\exists y \neg(\neg Animal(y) \vee Loves(x,y))] \vee [\exists y Loves(y,x)]. \\ & \forall x [\exists y \neg\neg Animal(y) \wedge \neg Loves(x,y)] \vee [\exists y Loves(y,x)]. \\ & \forall x [\exists y Animal(y) \wedge \neg Loves(x,y)] \vee [\exists y Loves(y,x)]. \end{aligned}$$

Забележете как един универсален квантор (  $\forall$  ) в предпоставката на импликацията се е превърнал в екзистенциален квантор. Съждението сега гласи „Има някое животно, което  $x$  не обича или (ако това не е така) някой обича  $x$ “. Очевидно смисълът на оригиналното съждение е запазен.

# Пример

3

Стандартизиране на променливите: За съждения като  $(\exists x P(x)) \vee (\exists x Q(x))$  където едно и също име на променлива се използва два пъти, променяме името на една от променливите. Това избягва объркване по-късно, когато премахнем кванторите. По този начин получаваме:

$$\forall x [\exists y \text{ Animal}(y) \wedge \neg \text{Loves}(x,y)] \vee [\exists z \text{ Loves}(z,x)].$$



# Пример

4

Сколемизация: Сколемизацията е процес на премахване на екзистенциални квантори чрез елиминиране. В простия случай това е точно като правилото за екзистенциално инстанциране: тарнсформиране  $\exists x P(x)$  в  $P(A)$ , където  $A$  е нова константа. Въпреки това, не можем да приложим екзистенциално инстанциране към нашето съждение, защото не съвпада с шаблона  $\exists v \alpha$ ; - само части от съждението съответстват с него. Ако приложим сляпо правилото към двете съвпадащи части, получаваме

$$\forall x [Animal(A) \wedge \neg Loves(x,A)] \vee Loves(B,x) ,$$

# Пример

4

което има изцяло погрешно значение: казва, че всеки или не обича определено животно А или е обичан от някакво конкретно същество В. Всъщност нашето оригинално съждение позволява на всеки човек да не обича различно животно или да бъде обичан от различна личност. По този начин искаме Сколем идентичностите да зависят от  $x$ :

$$\forall x [Animal(F(x)) \wedge \neg Loves(x, F(x))] \vee Loves(G(x), x).$$

$F$  и  $G$  са функциите на Сколем. Общото правило е, че аргументите на една функция на Сколем са всички универсално квантифицирани променливи, в чийто обхват се появява екзистенциалният квантор. Както при екзистенциалното инстанциране, сколемизираното съждение е изпълнимо точно когато оригиналното съждение е изпълнимо.

# Пример

5

Премахване на универсалните квантори: В този момент всички останали променливи са универсално квалифицирани. Следователно не губим никаква информация, ако изпуснем квантора:

$$[Animal(F(x)) \wedge \neg Loves(x, F(x))] \vee Loves(G(x), x).$$

# Пример

6

Разпределяне  $\vee$  върху  $\wedge$ :


$$[Animal(F(x)) \vee Loves(G(x),x)] \wedge [\neg Loves(x,F(x)) \vee Loves(G(x),x)].$$

# Правило за резолюция

$$\frac{\ell_1 \vee \dots \vee \ell_k, \quad m_1 \vee \dots \vee m_n}{\text{SUBST}(\theta, \ell_1 \vee \dots \vee \ell_{i-1} \vee \ell_{i+1} \vee \dots \vee \ell_k \vee m_1 \vee \dots \vee m_{j-1} \vee m_{j+1} \vee \dots \vee m_n)}$$

Където:  $\text{UNIFY}(\ell_i, \neg m_j) = \theta.$

Двете клаузи резолират при  
дадената субституция:


$$\theta = \{u/G(x), v/x\}$$

$$[Animal(F(x)) \vee Loves(G(x), x)]$$

$$[\neg Loves(u, v) \vee \neg Kills(u, v)]$$



$$[Animal(F(x)) \vee \neg Kills(G(x), x)] .$$



# Пример

Резолюцията доказва, че  $KB \models \alpha$  като докаже, че  $KB \wedge \neg\alpha$  е неудовлетворимо — тоест, като изведе празната клауза.

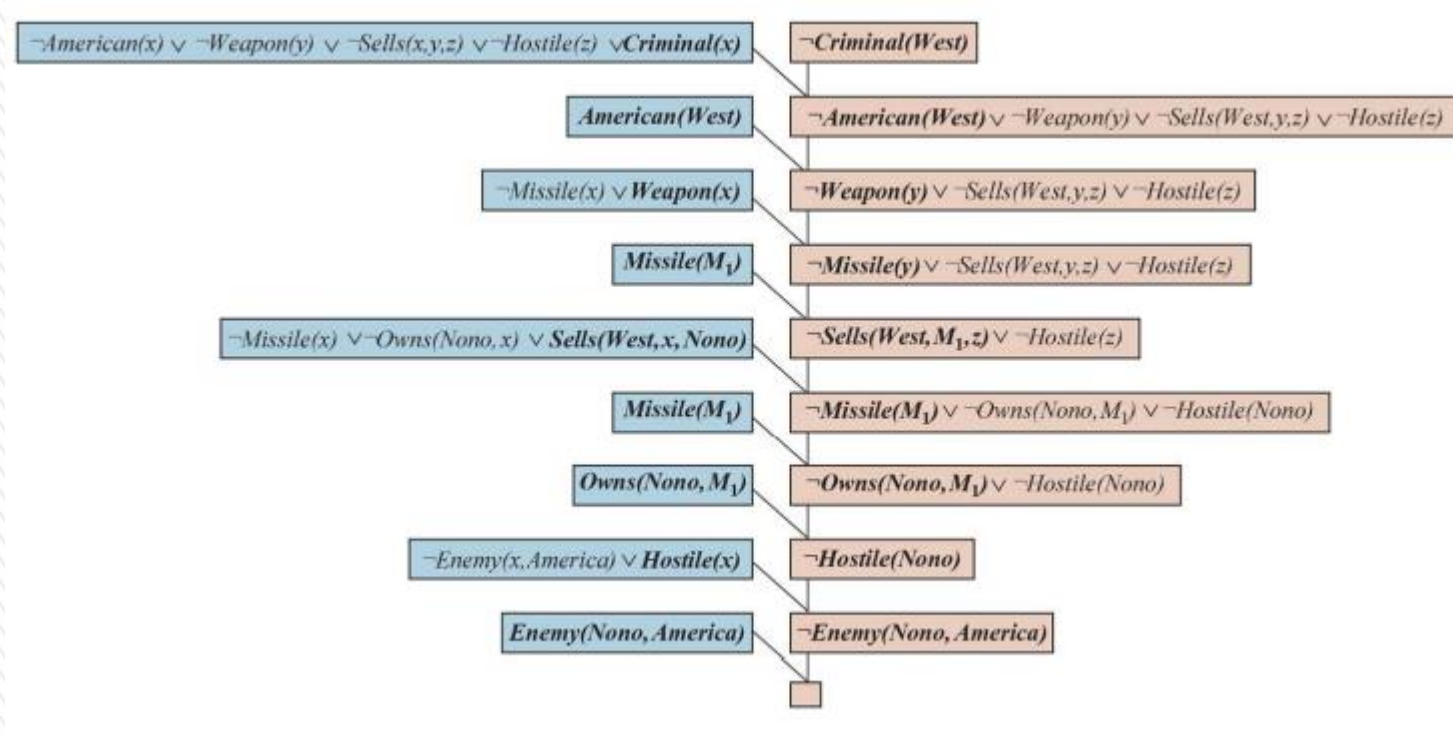
За примера за престъпление (съжденията са в CNF):

|   |                        |
|---|------------------------|
| $\neg American(x) \vee \neg Weapon(y) \vee \neg Sells(x,y,z) \vee \neg Hostile(z) \vee Criminal(x)$ |                        |
| $\neg Missile(x) \vee \neg Owns(Nono,x) \vee Sells(West,x,Nono)$                                    |                        |
| $\neg Enemy(x,America) \vee Hostile(x)$   |                        |
| $\neg Missile(x) \vee Weapon(x)$  |                        |
| $Owns(Nono,M_1)$  | $Missile(M_1)$         |
| $American(West)$  | $Enemy(Nono,America).$ |

# Пример

Искаме да докажем **Criminal(x)**

$\neg American(x) \vee \neg Weapon(y) \vee \neg Sells(x,y,z) \vee \neg Hostile(z) \vee Criminal(x)$   
 $\neg Missile(x) \vee \neg Owns(Nono,x) \vee Sells(West,x,Nono)$   
 $\neg Enemy(x,America) \vee Hostile(x)$   
 $\neg Missile(x) \vee Weapon(x)$   
 $Owns(Nono,M_1)$   
 $American(West)$   
 $Missile(M_1)$   
 $Enemy(Nono,America).$



# Логическо програмиране

- » Логическото програмиране е технология, която е доста близка до въплъщаването на декларативния идеал
  - > Системите трябва да бъдат конструирани чрез изразяване на знания на формален език
  - > Проблемите трябва да се решават чрез процеси на извод върху тези знания
- » Идеалът е обобщен в уравнението на Робърт Ковалски:

$$\textit{Algorithm} = \textit{Logic} + \textit{Control} .$$

# Пролог

- » Пролог е най-широко използваният език за логическото програмиране
- » Той се използва главно като език за бързо прототипиране и задачи за манипулиране на символи, като напр.:
  - > Създаване на компилатори
  - > Разбор на естествен език
  - > В Пролог са написани много експертни системи за правни, медицински, финансови и други области

# Пролог

- » Пролог програмите са групи от дефинитни клаузи, написани в нотация, малко по-различна от стандартната за ПЛ
- » Пролог използва главни букви за променливи и малки букви за константи
  - > Обратното на конвенцията в ПЛ
  - > “,” (запетаи) разделят отделни конюнкти в една клауза
  - > клаузите се пишат „обратно“ от това, на което сме свикнали в ПЛ
  - > вместо  $A \wedge B \Rightarrow C$  (както е в ПЛ) в Пролог  $C :- A, B$ .
- » Типичен пример:

```
criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z).
```



# Списъци

- » Нотацията  $[E|L]$  означава списък, чийто първи елемент е  $E$  и чийто остатък е  $L$
- » Следната Пролог програма успява, ако списък  $Z$  е резултат от свързването на списъци  $X$  и  $Y$ :

```
append([ ], Y, Y).  
append([A|X], Y, [A|Z]) :- append(X, Y, Z).
```

В повечето езици на високо ниво можем да напишем подобна рекурсивна функция, която описва как се свързват два списъка

Дефиницията на Prolog всъщност е много по-мощна, защото описва връзка, която се съдържа между три аргумента, вместо функция, изчислена от два аргумента

# Списъци

» Напр., можем да направим запитване за това кои два списъка могат да се свързат за да се получи списъка [1,2]:

```
append(X,Y,[1,2]);
```

```
X=[ ]      Y=[1,2];  
X=[1]      Y=[2];  
X=[1,2]    Y=[ ]
```

# Изпълнение на програми

- » Изпълнението на Пролог програми се осъществява чрез първо в дълбочина свързване назад, където клаузите се изпробват в реда, в който са зададени в БЗ
- » Някои аспекти на Пролог попадат извън стандартните логически изводи:
- » Пролог използва ДБ семантика, а не семантиката на ПЛ и това е очевидно в неговото третиране на еквивалентност и отрицание
- » Съществува набор от вградени функции за аритметика
  - > Литералите, които използват тези функционални символи, се „доказани“ чрез изпълнение на код, вместо извод
  - > Напр.  $X \text{ is } 4 + 3$ , когато  $X$  се свързва със 7,  $5 \text{ is } X + Y$  пропада, понеже вградените функции не решават произволни уравнения

# Изпълнение на програми

- » Има вградени предикати със странични ефекти при изпълнение
  - > Напр., предикати за вход-изход или предикати за промяна на БЗ (assert/retract)
  - > Такива предикати нямат аналог в логиката и могат да доведат до объркващи резултати – напр., ако фактите се твърдят (включат) в клон на дървото на доказателството, което евентуално се провали
- » Проверката за поява се подразбира от алгоритъма за унификация на Пролог
  - > Това означава, че могат да се направят някои несигурни изводи, които не са почти никога проблем в практиката
- » Пролог използва търсене на дълбочината свързване назад без никакви проверки за безкрайна рекурсия
  - > Това го прави много бърз, когато се дава правилен набор от аксиоми, но непълен, когато се дават грешни

# Изпълнение на програми

- » Създаването на Пролог представлява компромис между декларативност и ефективност на изпълнение
  - > Доколкото ефективността е била разбрана по време на проектирането на езика

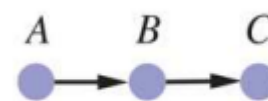


# Безкрайни цикли

- » Сега се обръщаме към ахилесовата пета на Пролог:
  - > Несъответствието между търсенето първо в дълбочина и дърветата за търсене, които включват повтарящи се състояния и безкрайни пътища
- » Нека разгледаме следните програми, които решават дали съществува път между две точки в насочен граф

# Пример

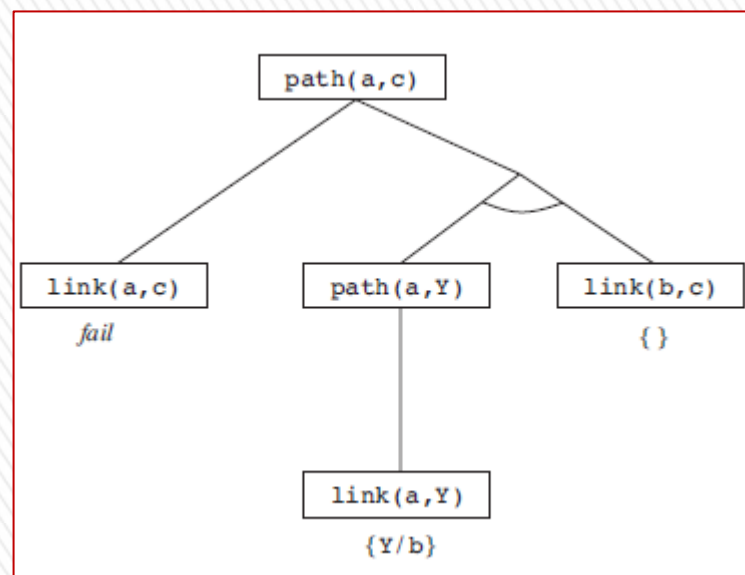
```
path(X,Z) :- link(X,Z).  
path(X,Z) :- path(X,Y), link(Y,Z).
```



```
link(a,b)  
link(b,c)
```

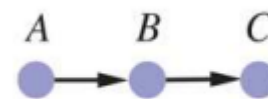
?

path(a,c)



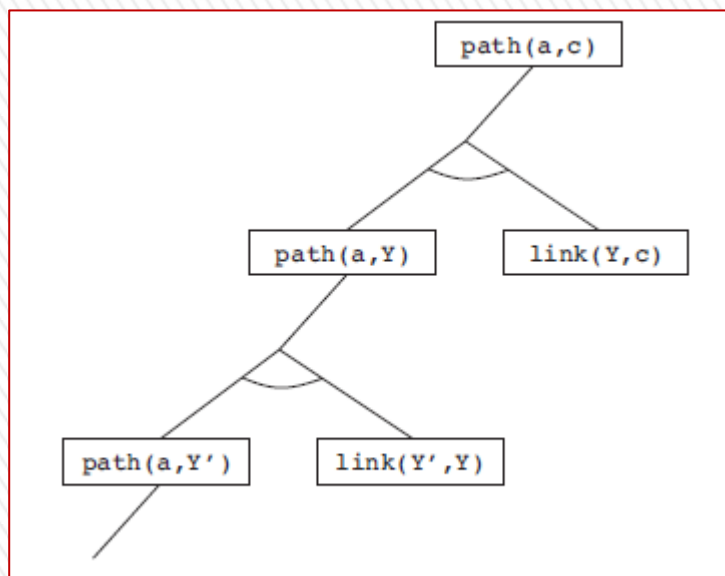
# Пример

```
path(X,Z) :- path(X,Y), link(Y,Z).  
path(X,Z) :- link(X,Z).
```



```
link(a,b)  
link(b,c)
```

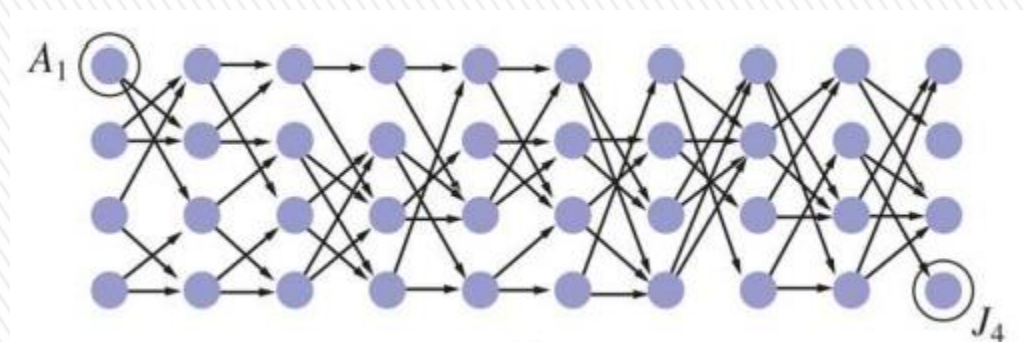
? path(a,c)



Машината на Пролог е непълна за определени клаузи (както показва този пример), защото за някои БЗ, тя не успява да докаже съждения, които са изводими

# Пример

Необходими са 877 извода, повечето от които включват намиране на всички възможни пътища до възли, от които целта е недостижима



# ДБ семантика

❓ Можем ли да представим това състояние на нещата, като съждението по-долу?

Да предположим, че Георги  
има двама братя – Иван и  
Йордан

Brother(Ivan, Georgi)  $\wedge$  Brother(Jordan, Georgi)



# ДБ семантика

? Можем ли да представим това състояние на нещата, като съждението по-долу?

Да предположим, че Георги  
има двама братя – Иван и  
Йордан

$\text{Brother}(\text{Ivan}, \text{Georgi}) \wedge \text{Brother}(\text{Jordan}, \text{Georgi})$

Не точно. Първо, това твърдение е вярно в модел, в който Иван има само един брат - трябва да добавим

$\text{Ivan} \neq \text{Jordan}$

# ДБ семантика

? Можем ли да представим това състояние на нещата, като съждението по-долу?

Да предположим, че Георги има двама братя – Иван и Йордан

$$\text{Brother}(\text{Ivan}, \text{Georgi}) \wedge \text{Brother}(\text{Jordan}, \text{Georgi})$$

Не точно. Първо, това съждението е вярно в модел, в който Георги има само един брат - трябва да добавим

$$\text{Ivan} \neq \text{Jordan}$$

Второ, съждението не изключва модели, в които Георги има повече братя освен Иван и Йордан. По този начин правилният запис на „Братя на Георги са Иван и Йордан“ е следният:

$$\text{Brother}(\text{Ivan}, \text{Georgi}) \wedge \text{Brother}(\text{Jordan}, \text{Georgi}) \wedge \text{Ivan} \neq \text{Jordan} \wedge \forall x \text{ Brother}(x, \text{Georgi}) \Rightarrow (x = \text{Ivan}) \vee (x = \text{Jordan})$$

# ДБ семантика

- » В много случаи това изглежда много по-тромаво от съответното изразяване на естествен език
- » В резултат на това хората могат да правят грешки в трансформацията на знанията в ПЛ
  - > Води до неинтуитивно поведение на логически системи, използващи знания
- » Можем ли да използваме семантика, която позволява по-ясен логически израз?

# ДБ семантика

- » Едно предложение, което е много популярно в системите за бази данни, работи по следния начин:
  - > **Допускане за уникални имена** – приемаме, че всеки константен символ да се отнася до отделен обект
  - > **Допускане за затворения свят** – приемаме, че атомарните съждения, за които не е известно, че са верни, се приемат за грешни
  - > **Затвореност на домейн** - всеки модел не съдържа повече елементи в домейна от тези, именувани с константни символи

# ДБ семантика в Пролог

- » Пролог използва ДБ семантика със следните означения:
  - > UNA: всяка константа на базов терм реферират отделен обект
  - > CWA: единствените верни съждения са тези, които следват от БЗ
- » Няма начин да се твърди, че едно съждение е грешно в Пролог
  - > Това прави Пролог по-слабо изразителен от ПЛ, но е част от това, което прави Пролог по-ефективен и по-кратък



# Пример

Нека разгледаме следните съждения на Пролог за предложения за курсове

$Course(CS, 101), Course(CS, 102), Course(CS, 106), Course(EE, 101).$

- Поради UNA: CS и EE са различни (както са 101, 102 и 106), което означава, че има **четири отделни курса**
- Поради CWA: няма други курсове, така че има **точно четири курса**
- Ако това са твърдения в ПЛ, а не в Пролог, тогава всичко, което можем да кажем е, че има някъде между един и безкрайни курсове
  - Това е така, защото съжденията (в ПЛ) не отричат възможността да се предлагат и други неназовани курсове, нито пък казват, че посочените курсове са различни един от друг
- Ако искаме да преведем горното съждение в ПЛ ще получим следното:

$$Course(d, n) \Leftrightarrow (d = CS \wedge n = 101) \vee (d = CS \wedge n = 102) \vee (d = CS \wedge n = 106) \vee (d = EE \wedge n = 101) .$$



*Благодаря за вниманието!*