Inferência em Lógica de Primeira Ordem

`Expr` : tipo de dados para sentenças lógicas; mecanismo para representar expressões
`FolKB` : base de conhecimento para lógica de primeira ordem.
Forward chaining e backward chaining para `FolKB` e seu uso na base `crime_kb` .

In [18]:

```
from utils import *
from logic import *
```

Sentenças lógicas

The `Expr` class is designed to represent any kind of mathematical expression. The simplest type of `Expr` is a symbol, which can be defined with the function `Symbol` :

In [ ]:

```
Symbol('x')
```

In [2]:

```
(x, y, P, Q, f) = symbols('x, y, P, Q, f')
P & ~Q
```

Out[2]:

```
(P & ~Q)
```

In [3]:

```
Pxy = P(x, y)
Pxy.op
```

Out[3]:

```
'P'
```

In [4]:

```
Pxy.args
```

Out[4]:

```
(x, y)
```

Operadores para construção de sentenças lógicas

Operation	Book	Python Infix Input	Python Output	Python `Expr` Input
Negation	¬ P	`~P`	`~P`	`Expr('~', P)`
And	P ∧ Q	`P & Q`	`P & Q`	`Expr('&', P, Q)`
Or	P ∨ Q	`P | Q`	`P | Q`	`Expr('|', P, Q)`
Inequality (Xor)	P ≠ Q	`P ^ Q`	`P ^ Q`	`Expr('^', P, Q)`
Implication	P → Q	`P | '==>' | Q`	`P ==> Q`	`Expr('==>', P, Q)`
Reverse Implication	Q ← P	`Q | '<==' | P`	`Q <== P`	`Expr('<==', Q, P)`
Equivalence	P ↔ Q	`P | '<=>' | Q`	`P <=> Q`	`Expr('<=>', P, Q)`

In [5]:

```
~(P & Q)  |'==>'|  (~P | ~Q)
```

Out[5]:

```
(~(P & Q) ==> (~P | ~Q))
```

In [6]:

```
expr('~(P & Q)  ==>  (~P | ~Q)') #expr pega uma string e faz o parse parauma instância de Expr
```

Out[6]:

```
(~(P & Q) ==> (~P | ~Q))
```

In [7]:

```
expr('sqrt(b ** 2 - 4 * a * c)')
```

Out[7]:

```
sqrt(((b ** 2) - ((4 * a) * c)))
```

First-Order Logic Knowledge Bases: `FolKB`

Exemplo: Criminal KB

The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

Primeiro passo: extrair os fatos e converte-los isoladamente para cláusulas FOL.

"... it is a crime for an American to sell weapons to hostile nations"

- `Criminal(x)` : `x` is a criminal
- `American(x)` : `x` is an American
- `Sells(x ,y, z)` : `x` sells `y` to `z`
- `Weapon(x)` : `x` is a weapon
- `Hostile(x)` : `x` is a hostile nation

Segundo passo: unir os fatos em cláusulas compostas.

- $\texttt{American(x)} \wedge \texttt{Weapon(y)} \wedge \texttt{Sells(x, y, z)} \wedge \texttt{Hostile(z)} \implies \texttt{Criminal(x)}$

In [43]:

```
clauses = []
```

In [44]:

```
clauses.append(expr("(American(x) & Weapon(y) & Sells(x, y, z) & Hostile(z)) =
=> Criminal(x)"))
```

"*The country Nono, an enemy of America*"

In [45]:

```
clauses.append(expr("Enemy(Nono, America)"))
```

"*Nono ... has some missiles*"

In [46]:

```
clauses.append(expr("Owns(Nono, M1)"))
clauses.append(expr("Missile(M1)"))
```

"*All of its missiles were sold to it by Colonel West*"

In [47]:

```
clauses.append(expr("(Missile(x) & Owns(Nono, x)) ==> Sells(West, x, Nono)"))
```

"West, who is American"

In [48]:

```
clauses.append(expr("American(West)"))
```

É sabido ainda que mísseis são armas e inimigos são hostis.

In [49]:

```
clauses.append(expr("Missile(x) ==> Weapon(x)"))
clauses.append(expr("Enemy(x, America) ==> Hostile(x)"))
```

KB criada

In [50]:

```
crime_kb = FolKB(clauses)
```

Inferência em FOL

Os algoritmos **Forward chaining** e **Backward chaining** utilizam um componente/processo-chave chamado **Unificação**: a substituição de variáveis por constantes é um exemplo típico.

`unify` : algoritmo recursivo (dicionário).

In [27]:

```
unify(expr('x'), 3)
```

Out[27]:

```
{x: 3}
```

In [28]:

```
unify(expr('A(x)'), expr('A(B)'))
```

Out[28]:

```
{x: B}
```

In [29]:

```
unify(expr('Cat(x) & Dog(Dobby)'), expr('Cat(Bella) & Dog(y)'))
```

Out[29]:

```
{x: Bella, y: Dobby}
```

In [30]:

```
print(unify(expr('Cat(x)'), expr('Dog(Dobby)')))
print(unify(expr('Cat(x) & Dog(Dobby)'), expr('Cat(Bella) & Dog(x)')))
```

```
None
None
```

Forward Chaining

Tentativa de unificar cada uma das premissas com a cláusula na `KB`. Se possível, a conclusão é adicionada à `KB`. Processo de inferência é repetido até que a query possa ser respondida ou até que nenhuma sentença possa ser adicionada.

A função `fol_fc_ask` é um gerador que leva a todas as substituições que validam a query.

In [110]:

```
# todas as nações hostis da base
answer = fol_fc_ask(crime_kb, expr('Hostile(x)'))
print(list(answer))
```

```
[{x: Nono}]
```

Adicionando novos dados à base:

função `tell`

In [32]:

```
crime_kb.tell(expr('Enemy(JaJa, America)'))
answer = fol_fc_ask(crime_kb, expr('Hostile(x)'))
print(list(answer))
```

```
[{x: Nono}, {x: JaJa}]
```

***Note*:** `fol_fc_ask` faz mudanças na `KB`.

Backward Chaining

Parte do objetivo/query, utilizando as regras existentes na base para encontrar os fatos.

`fol_bc_or` e `fol_bc_and`

O método `ask` de `FolKB` usa `fol_bc_ask` e dispara a primeira substituição retornada pelo gerador para responder à query.

In [51]:

```
crime_kb = FolKB(clauses)
crime_kb.ask(expr('Hostile(x)'))
```

Out[51]:

```
{v_77: x, x: Nono}
```

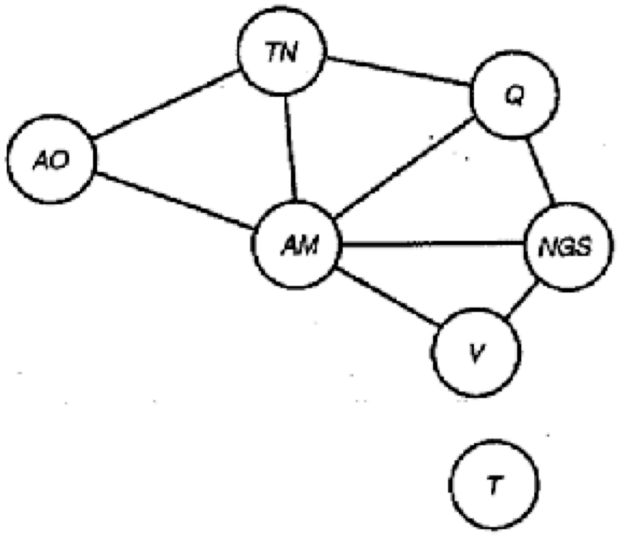
In [52]:

```
crime_kb.ask(expr('Hostile(Brasil)'))
```

Out[52]:

```
False
```

Exemplo: Coloração de mapas



In [34]:

```
import sys
#sys.setrecursionlimit(15000)
clauses = []
clauses.append(expr("Adj(R,G)"))
clauses.append(expr("Adj(G,R)"))
clauses.append(expr("Adj(R,B)"))
clauses.append(expr("Adj(B,R)"))
clauses.append(expr("Adj(B,G)"))
clauses.append(expr("Adj(G,B)"))
clauses.append(expr("Adj(R)"))
clauses.append(expr("Adj(B)"))
clauses.append(expr("Adj(G)"))
clauses.append(expr("(Adj(ao,tn) & Adj(ao,am) & Adj(tn,am) & Adj(tn,q) & Adj(a
m,q) & Adj(am,ngs) & Adj(q, ngs) & Adj(am,v) & Adj(ngs,v) & Adj(t)) ==> Colori
do(ao,tn,q,am,ngs,v,t)"))
```

In [35]:

```
mapa_kb = FolKB(clauses)
```

In [36]:

```
mapa_kb.ask(expr("Colorido(ao,tn,q,am,ngs,v,t)"))
```

Out[36]:

```
{v_9: ao,
 v_10: tn,
 v_12: q,
 v_11: am,
 v_13: ngs,
 v_14: v,
 v_15: t,
 ao: R,
 tn: G,
 am: B,
 q: R,
 ngs: G,
 v: R,
 t: R}
```

Preparação para o Exercicio 1:

In [182]:

```
clauses = []
clauses.append(expr("P(F(x)) ==> P(x)"))
clauses.append(expr("Q(x) ==> P(F(x))"))
clauses.append(expr("P(A)"))
clauses.append(expr("Q(B)"))
```

In [183]:

```
exercicio1_kb = FolKB(clauses)
```

In [ ]:

```
answer = fol_fc_ask(exercicio1_kb, expr("????"))
print(list(answer))
```

In [175]:

```
clauses = []
clauses.append(expr("P(F(x)) ==> P(x)"))
clauses.append(expr("Q(x) ==> P(F(x))"))
clauses.append(expr("P(A)"))
clauses.append(expr("Q(B)"))
```

In [176]:

```
exercicio1_kb = FolKB(clauses)
```

In [ ]:

```
exercicio1_kb.ask(expr("????"))
```

In [ ]:

```

```