

# Inteligência Artificial

José Mendes 107188

2023/2024



universidade  
de aveiro

---

# 1 Noções de Programação Declarativa

## 1.1 Programação Declarativa

A Programação Declarativa abstrai-se da implementação, focando-se apenas na descrição do que se pretende fazer, enquanto faz uso de dois paradigmas:

- **Programação Funcional**, baseado em funções/cálculo-lambda, a entidade central é a função;
- **Programação em Lógica**, baseado em lógica de primeira ordem, a entidade central é o predicado;

## 1.2 Paradigma Imperativo

O fluxo de operações é especialmente sequenciado de operações com foco na forma como as tarefas são executadas (**instrução**). Podemos **alterar o conteúdo em memória** e ainda (instruções de afetação/atribuição) e ainda **realizar análise de casos** (if-then-else, switch/case, ...), **processamento iterativo** (while, repeat, for, ...) e ter associados **sub-programas** (procedimentos, funções).

**Exemplo:** SQL é uma linguagem declarativa, uma vez que nos seus comandos apenas descrevemos o que queremos obter. Assim, o programador é abstraído da forma como as operações são executadas na prática, ficando essa tarefa a cargo do compilador.

## 1.3 Paradigma Declarativo

	Funcional	Lógico
Fundamentos	Lambda calculus	Lógica de primeira ordem
Conceito central	Função	Predicado
Mecanismos	Aplicação de funções Unificação uni-direccional Estruturas decisórias	Inferência lógica (resolução SLD) Unificação bi-direccional
Programa	Um conjunto de declarações de funções e estruturas de dados	Um conjunto de fórmulas lógicas (factos e regras)

A sua origem data da segunda metade do século XX, mas ainda hoje é amplamente usada e até está em crescimento em áreas como a Inteligência Artificial.

---

## 1.4 Programação Funcional

Possibilidade de definir funções localmente e sem nome. Por exemplo as funções lambda presentes em Lisp e Python.

## 1.5 Programação em Lógica

Um programa é uma teoria sobre um domínio. Por exemplo, temos **socrates é um homem**,  $homem(socrates)$  e, um **homem é mortal**,  $homem(X) : \neg mortal(X)$ . Se perguntarmos se **socrates é mortal**,  $mortal(socrates)$ , a resposta é sim.

## 1.6 Atitude do programador

A programação declarativa, dada a sua elevada expressividade, é pouco compatível com aproximações empíricas (ou seja, tentativa e erro) à programação. Primeiro é preciso pensar bem na estrutura do problema antes de começar a teclar.

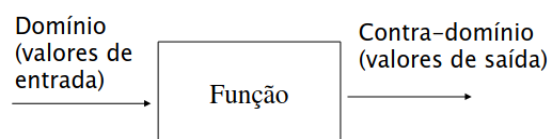
**Passos a seguir:** Perceber o problema  $\rightarrow$  Desenhar o programa  $\rightarrow$  Escrevê-lo  $\rightarrow$  Rever e testar

## 1.7 Características da Programação Funcional

- A entidade central é a função;
- A noção de função é diretamente herdada da matemática (ao contrário das linguagens imperativas, que pode ser muito diferente);
- A estrutura de controlo fundamental é a "aplicação de funções";
- A noção de "tipo da função" captura a noção matemática de domínio (de entrada e saída);
- Os elementos do domínio de entrada e saída podem ser funções;

## 1.8 Função

Tem valores de entrada (domínio) e valores de saída (contradomínio).



## 1.9 Programação em Lógica

Um programa numa linguagem baseada em lógica representa uma teoria sobre um problema. Um programa é uma sequência de frases ou fórmulas representando **factos** (informação sobre objetos do problema/domínio de aplicação) e **regras** (leis gerais sobre esse problema/domínio). Implicitamente as frases estão reunidas numa grande conjunção, e cada frase está universalmente quantificada. Portanto, **programação declarativa**.

---

## 2 Programação ao estilo funcional em Python

### 2.1 Python

Criada no final dos anos 90, é uma linguagem de programação interpretada, iterativa, portátil, funcional, orientada a objetos e de implementação aberta. Tem como principais objetivos: simplicidade sem prejuízo da utilidade, programação modular, legibilidade, desenvolvimento rápido, facilidade de integração, nomeadamente com outras linguagens. Apresenta-se como uma linguagem **multi-paradigma**:

- **Programação funcional:** Expressões lambda, funções de ordem superior, listas com sintaxe simples, iteradores, ...
- **Programação OO:** Classes, objetos, métodos, herança, ...
- **Programação imperativa/modular:** Atribuição, sequências, condicionais, ciclos, ...

#### 2.1.1 Python vs Java

Comparativamente ao JAVA revela-se menos consisa porque os espaços são sintaticamente relevantes e o código, não compilado para código nativo (interpretado), demora mais tempo a executar (diferença residual), no entanto apresenta uma maior facilidade de desenvolvimento e legibilidade.

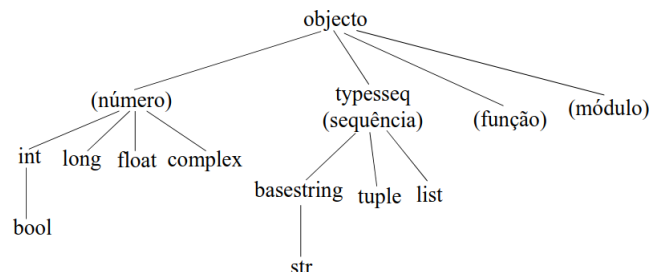
#### 2.1.2 Áreas de aplicação

As principais áreas de aplicação são: interligação de sistemas, aplicações gráficas, aplicações para bases de dados, multimédia, internet protocol/web e robótica e inteligência artificial.

### 2.2 Dados, ou "objetos"

**Objeto:** no contexto de Python, esta designação é aplicada a qualquer dado que possa ser armazenado numa variável, ou passado como parâmetro a uma função.

Cada objeto é caracterizado por uma identidade/referência (identifica a posição de memória onde está armazenado), um tipo e um valor. Alguns tipos de objetos podem ter atributos e métodos, outros (classes) podem ter sub-tipos (sub-classes).



Alguns destes são imutáveis, como as **str** e **tuple**. Para determinar o tipo de um objeto utiliza-se a função pré-definida **type()**.

---

## 2.3 Variáveis

As variáveis guardam objetos, mas não são declaradas (inicializadas apenas) nem têm tipos associados (o tipo é do objeto!). Praticamente tudo pode ser atribuído a uma variável, incluindo funções, módulos e classes. Similar as linguagens imperativas, e ao contrário das funcionais, o valor de uma variável pode ser alterado. Não se pode ler o valor de uma variável se este não tiver sido previamente inicializado.

## 2.4 Instrução de atribuição

Tal como é habitual na programação imperativa, e ao contrário do habitual na programação declarativa, Python possui instrução de atribuição.

**Exemplo:**

---

```
n = 10
a = b = c = 0
x = 7.25
cad = "cadeia"
t = (n, x)
l = [1, 2, 'quatro', 5.0]
```

---

Pode user a operação de atribuição para decompor estruturas:

---

```
triplo = (1, 2, 3)
(i, j, k) = triplo
```

---

## 2.5 Operadores

Operadores comuns matemáticos e lógicos. É importante saber que objetos de tipos diferentes nunca são iguais (exceto os diferentes tipos de números).

### 2.5.1 Operadores lógicos

Na conjunção e disjunção, o segundo elemento só é avaliado se for necessário para determinar o resultado.

---

```
if False and a: # a nao e avaliado
    ...

if True or a: # a nao e avaliado
    ...
```

---

---

## 2.6 Acesso a sequências

Os elementos das sequências são acedidos através de índices inteiros consecutivos (primeiro elemento tem índice 0).

É possível extrair “fatias” das sequências. A indexação é circular, pelo que podemos aceder a índices negativos.

---

```
# [inf:sup] retorna a sequencia entre os indices inf e sup-1 (uma opia da lista)

# Para fazer uma copia integral da lista usa-se [:]
```

---

**Detalhe importante:** instrução de atribuição, em vez de copiar valores, limita-se a associar um dado identificador a um dado objecto. Desta forma, a atribuição da variável x a uma variável y, apenas tem como resultado associar y ao mesmo objecto que x já estava associada. No caso dos objetos mütaveis, temos de ter cuidado:

---

```
a = [1, 2, 3]
b = a
b[1:2] = []
print(a) # [1, 3]
```

---

## 2.7 Definição de funções

Funções sem return, retornam None, são também conhecidas como procedimentos. As funções podem ser recursivas e são objetos do tipo **function**.

Os parâmetros são passados às funções segundo um mecanismo “passagem por valor” (“call by value”). Os parâmetros são passados por referência, não cópias!

Se atribuírmos um novo objeto a uma variável passada por parâmetro, essa atribuição ocorre apenas no espaço de nomes da função (imagem à esquerda). Se modificarmos um objeto passado por parâmetro (por exemplo, apagar um elemento de uma lista), isto não altera a referência do objeto, e portanto vai permanecer após o retorno da função (imagem à direita).

<pre>&gt;&gt;&gt; def incr(x): ...   x=x+1 ...   return x ... &gt;&gt;&gt; n=10 &gt;&gt;&gt; incr(n) 11 &gt;&gt;&gt; n 10</pre>	<pre>&gt;&gt;&gt; def acresc(l,x): ...   l[0:0]=[x] ...   return l ... &gt;&gt;&gt; lista=[5,12] &gt;&gt;&gt; acresc(lista,30) [30,5,12] &gt;&gt;&gt; lista [30,5,12]</pre>
---	---

O problema anterior resolve-se trabalhando sobre uma cópia local, no caso de uma lista l, podemos usar l[:], que dá uma cópia integral.

---

Podemos passar parâmetros com valores por defeito, podendo se, na chamada, omitir alguns parâmetros.

---

```
def abc(arg1, arg2=True, arg3=None):  
    ...  
# Quando so queremos passar um dos args por defeito devemos indicar o seu nome  
abc(1, arg3=3)
```

---

## 2.8 Funções Recursivas

---

```
# devolve factorial de um numero n  
def factorial(n):  
    if n==0:  
        return 1  
    if n>0:  
        return n*factorial(n-1)  
  
# devolve o comprimento de uma lista  
def comprimento(lista):  
    if lista==[]:  
        return 0  
    return 1+comprimento(lista[1:])
```

---

Esta última função, vai funcionar da seguinte forma:

---

```
comprimento([1, 2, 3])  
=  
1 + comprimento([2, 3])  
=  
1 + (1 + comprimento([3]))  
=  
1 + (1 + (1 + comprimento([])))  
=  
1 + (1 + (1 + 0))  
=  
3
```

---

---

```
# verifica se um elemento e membro de uma lista  
def membro(x,lista):  
    if l==[]:  
        return False  
    return (lista[0]==x) or membro(x,lista[1:])
```

---

---

```
# devolve uma lista com os elementos da lista de entrada por ordem inversa
def inverter(lista):
    if lista==[]:
        return []
    inv = inverter(lista[1:])
    inv[len(inv):] = [lista[0]]
    return inv
```

---

## 2.9 Expressões lambda

São **expressões cujo valor é uma função**. Funções que recebem expressões lambda como entrada e as retornam como saída chamam-se funções de ordem superior. Exemplo:

---

```
lambda x: x*x

# Pode ser atribuída a uma variável
m = lambda x,y: math.sqrt(x**2 + y**2)
m(1, 2) # 2.23606797749979
```

---

Como qualquer objeto, uma expressão lambda pode ser passada como:

---

```
# Tendo uma func h que produz f(x)*X
def h(f, x): return f(x)*x

# Para usar:
h(lambda x: x+1, 7) # 56
```

---

Podem ser produzidas por outras funções:

---

```
def faz_incrementador(n):
    return lambda x: x+n

# Para usar:
f = faz_incrementador(1)
f(10) # 11
```

---

As expressões lambda são também conhecidas como expressões funcionais. As funções que recebem estas expressões como entrada e/ou produzem expressões lambda como saída chamam-se funções de ordem superior.

**Nota importante:** As expressões lambda só são úteis enquanto são simples. Uma função complexa merece ser escrita de forma clara numa definição (def) à parte.



---

## 2.10 Aplicar uma função a uma lista

Para tal podemos usar a função **map()**, predefinida em Python, retorna um iterador que pode ser convertido numa lista. Esta função é equivalente a:

---

```
def aplicar(funcao, lista):
    if lista==[]:
        return []
    return [funcao(lista[0])] + aplicar(funcao, lista[1:])
```

---

## 2.11 Filtrar uma lista

Para tal podemos usar a função **filter()**, predefinida em Python, retorna um iterador que pode ser convertido numa lista. Esta função é equivalente a:

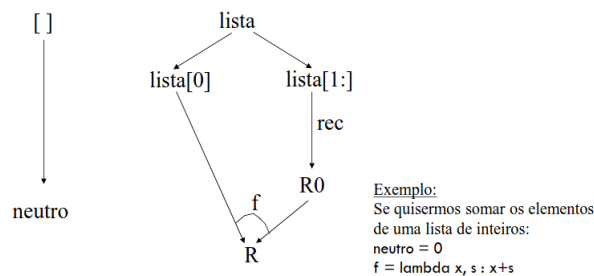
---

```
def filtrar(funcao, lista):
    if lista==[]:
        return []
    if funcao(lista[0]):
        return [lista[0]] + filtrar(funcao, lista[1:])
    return filtrar(funcao, lista[1:])
```

---

## 2.12 Reduzir uma lista a um valor

Muitos dos procedimentos aplicados a listas consistem em combinar a cabeça da lista com o resultado da chamada recursiva sobre os restantes elementos, retornando um valor “neutro” pré-definido para a lista vazia. Em Python, podemos usar a função **reduce()** da biblioteca **functools**.



---

```
# Dada uma lista, uma func de combinacao e um elemento neutro
def reduzir(lista, fcomb, neutro):
    if lista==[]:
        return neutro
    return fcomb(lista[0], reduzir(lista[1:], fcomb, neutro))
```

---

---

### 2.13 Listas por compreensão (list comprehension)

São mecanismos compactos para processar os elementos de uma lista (alguns ou todos), podendo ser aplicadas não só a estas como também a tuplos ou cadeias de caracteres (str), o resultado é uma lista. Podem funcionar com a função pré-definida **map()** ou **filter()**.

$$< \textit{expr} > \textbf{for} < \textit{var} > \textbf{in} < \textit{seq} > \textbf{if} < \textit{condition} >$$

---

```
[x**2 for x in [2, 3, 4]] # Output: [4, 9, 16]
map(lambda x: x**2, [2, 3, 4]) # Igual
```

```
[x for x in [2, 3, 4] if x%2==0] # Output: [2, 4]
filter(lambda x: x%2==0, [2, 3, 4]) # Igual
```

---

Dada uma lista de listas e uma função, podemos aplicar uma função a cada um dos elementos das listas, retornando a concatenação das listas resultantes.

---

```
[f(y) for x in lista for y in x]

# Exemplo:
lista = [[1, 2, 3], [4, 5, 6]]
[x+1 for x in lista for y in x] # [2, 3, 4, 5, 6, 7]
```

---

### 2.14 Classes

Tal como nas restantes linguagens Orientadas a Objetos (OO), em Python uma classe define um conjunto de objetos caracterizados por diversos atributos e métodos e organizam-se numa hierarquia que permite a herança.

---

```
# sintaxe
class <nome-classe>:
    <decl-1>
    ...
    <decl-n>
```

---

**Nota:** Tal como acontece com as variáveis normais, também os atributos das classes não são declarados. Os atributos são acessados através da notação "objeto.atributo". Podemos, em qualquer momento, criar um novo atributo para um objeto, bastando para isso atribuir-lhe um valor.

Existe ainda classes derivadas/herança, que herdam os métodos e atributos da classe mãe, sendo possível uma classe ter várias classes mães.

---

### 2.14.1 Métodos e atributos pré-definidos

#### Métodos:

- `__init__` - Construtor da classe;
- `__str__` - Conversão para cadeia de caracteres;
- `__repr__` - Representação em cadeia de caracteres que aparece na consola do interpretador;

#### Atributos:

- `__class__` - identifica a classe de um dado objecto

### 2.14.2 list de Python é uma classe

Tem os seguintes métodos:

- `append(x)` - Adiciona x ao fim da lista;
- `extend(L)` - Adiciona os elementos da lista L ao fim da lista;
- `insert(i, x)` - Adiciona x na posição i da lista;
- `remove(x)` - Remove a primeira ocorrência de x na lista;
- `index(x)` - Devolve o índice da primeira ocorrência de x na lista;
- `sort()` - Ordena a lista (modifica a lista);

## 3 Tópicos de Inteligência Artificial

### 3.1 Definição de "Inteligência"

Segundo a sua definição lexical, **inteligência** é a capacidade de pensar, raciocionar, adquirir e aplicar conhecimento. É o conjunto de capacidades superiores da mente.

O seu estudo envolve várias vertentes, desde a já referida aquisição, representação e armazenamento de conhecimento, a geração de comportamento inteligente, a origem das motivações, emoções e prioridades, como as percepções dão origem a entidades simbólicas, como raciocionar sobre o passado, planear o futuro, como surge a ilusão, a crença, esperança, amor ...

---

### 3.2 Definição de "Inteligência Artificial"

É a disciplina que estuda as teorias e técnicas necessárias ao desenvolvimento de "artefactos" inteligentes. Historicamente, estas vertentes resumem-se a quatro grandes abordagens, definidas por Russel & Norvig (1995).

Pensar como o ser humano	Pensar racionalmente
Agir como o ser humano	Agir racionalmente

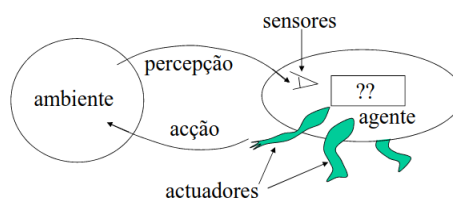
As **humanas** focam-se no ser humano, desenvolvendo uma ciência empírica, baseada na hipótese e na experiência. Por outro lado, a **abordagem racional** envolve uma combinação de matemática e engenharia.

**Nota:** No início estava muito focada na imitação das capacidades do ser humano, no entanto, mais recentemente tem-se voltado mais para a racionalidade como referência. Atualmente é maioritariamente utilizada para a resolução de problemas concretos e pouco abrangentes, focando-se na melhor utilização dos recursos de forma racional.

### 3.3 Definição de "Agente"

Segundo o dicionário, pode ser uma entidade com poder ou autoridade para agir, ou uma entidade que atua em representação de outrem.

**Agente** - Definido como uma entidade com capacidade de obter informação sobre o seu ambiente (sensores) e de executar (atuadores) ações em função dessa informação por Russel e Norvig.



---

### 3.3.1 Exemplo de agentes

Tipo de agente	Percepção	Ação	Objectivos	Ambiente
Sistema de diagnóstico médico	Sintomas, respostas do paciente	Perguntas, testes, tratamentos	Saúde do paciente, custo mínimo	Paciente, hospital
Sistema de análise de imagens de satélite	Imagem	Devolver uma categorização da cena	Categorização correcta	Imagens de um satélite em órbita
Braço robótico para embalagem	Imagem, sinal de força	Colocar peças em caixas	Colocar as peças na posição correcta	Alimentador de peças, caixas
Controlador de refinaria	Temperatura, pressão	Abrir e fechar válvulas; ajustar temperatura	Pureza, segurança	Refinaria
Tutor de inglês interactivo	Palavras introduzidas	Propor exercícios, corrigi-los, dar sugestões	Maximizar o resultado dos alunos num teste	Conjunto de alunos

### 3.4 Agir como o ser humano - Teste de Turing

O **teste de Turing** (1950) pretendeu estabelecer a definição operacional para o comportamento inteligente de nível humano, que consistia em submeter o artefacto a um interrogatório através de um terminal de texto. Se o humano não conseguir concluir que está a interrogar uma máquina, então conclui-se que o artefacto é inteligente.

### 3.5 A "sala chinesa" de Searle

Esta teoria foi refutada por **Searle**, um filósofo americano, com o argumento da **Sala Chinesa**. Neste, descreveu o cenário de colocar um humano que não percebe chinês fechado dentro de uma sala com um livro escrito na sua língua que ensine chinês a ser interrogado por outro humano através de papéis passados por baixo da porta.

**Searle** defendeu que apesar de não perceber os papéis entregues por baixo da porta, o humano iria conseguir traduzi-los através do livro e elaborar respostas. Fora da sala, o interrogador iria ter a ilusão de que o interrogado sabia chinês. No entanto, nem o humano, nem a sala, nem o livro percebem chinês. Logo, deduziu que não há qualquer compreensão de chinês naquela sala. No entanto, podemos contra-argumentar que, embora individualmente, os componentes do sistema não percebam chinês (a sala, o humano, o livro, as folhas de papel), não compreendam chinês, **o sistema no seu conjunto compreende chinês**.

Foi esta a premissa para a sua argumentação de que os computadores apenas fazem uso de regras sintáticas para manipular o texto, sem ter qualquer entendimento do seu significado ou semântica. Para ele a inteligência é um processo biológico, que apenas pode ser simulado pelas máquinas, mas nunca adquirido.

---

## 3.6 Tipos e arquiteturas de agentes

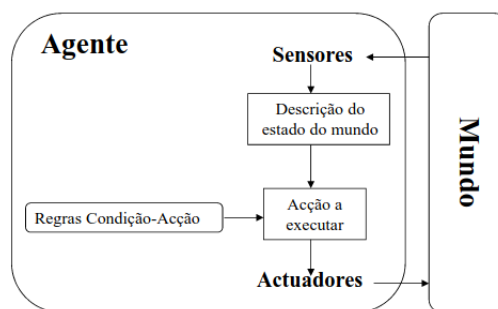
### Tipos de agentes:

- Reativo simples;
- Reativo com estado;
- Deliberativos orientados a objetivos;
- Deliberativos orientados por funções de utilidades;

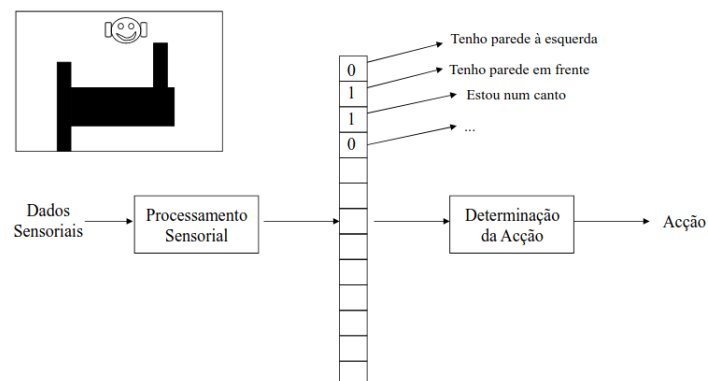
### Arquiteturas de agentes:

- Subsunção;
- Três torres;
- Três camadas;
- CARL;

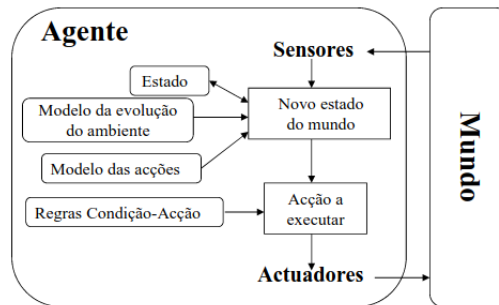
#### 3.6.1 Agentes reativos: simples



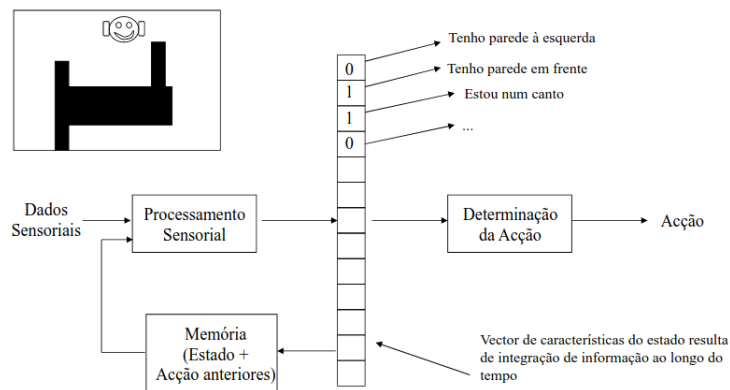
Também conhecidos por agentes de **estímulo-resposta** (ou "sistemas de produção"), este tipo de agente apresenta um conjunto de sensores, através dos quais recebe uma percepção do estado do mundo, sobre a qual aplica um conjunto de regras (regras de condição-ação, também conhecido como "regras de situação-ação" ou "regras de produção") e executa as ações correspondentes.



### 3.6.2 Agentes reativos: com estado interno

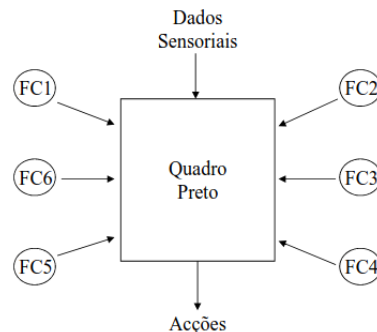


Funcionam de forma semelhante ao anterior, mas para além dos sensores, fazem uso de um **estado interno** e do histórico de ações anteriores para construir a percepção do estado do mundo.



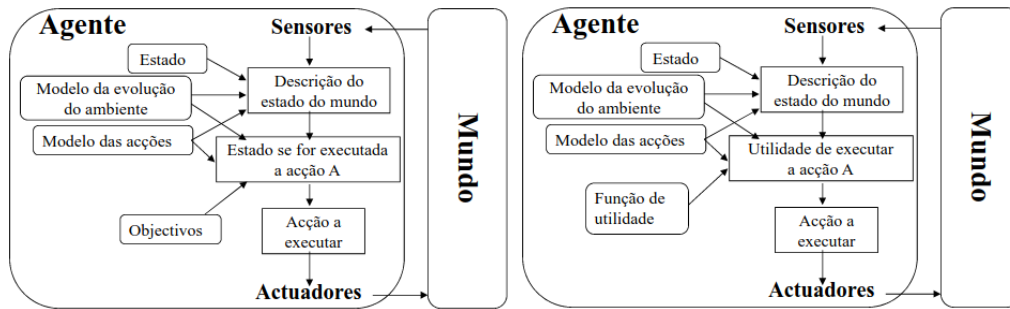
### Sistemas de Quadro Preto

Podem ser vistos como uma elaboração dos sistemas reactivos com estado interno. Uma **"fonte de conhecimento (FC)"** é um programa que vai fazendo alterações no Quadro Preto. Uma FC pode ser vista como um especialista num dado domínio. Tipicamente, cada FC rege-se por um conjunto de regras de situação-ação.



### 3.6.3 Agentes deliberativos

Contrariamente aos anteriores, executam as ações com base em **objetivos** ou **função de utilidade**.



**Primeira figura:** Agente deliberativo orientado a objetivos.

**Segunda figura:** Agente deliberativo orientado por funções de utilidade.

**Nota:** A função de atividade é uma função que avalia a relevância de cada estado para o agente.

## 3.7 Propriedades do mundo de um agente

Visto como se define um agente e os seus tipos, falta perceber como se caracteriza o seu ambiente. Este pode apresentar várias propriedades.

- **Acessibilidade** - O mundo é **acessível** se os sensores do agente tem acesso a toda a informação sobre o ambiente (descrição completa), ou pelo menos a toda a informação relevante. para o processo de escolha das ações (**efetivamente acessível**).
- **Determinismo** - Se o estado resultante da execução de uma ação é apenas e totalmente determinado pelo estado atual e pelos efeitos esperados da ação.
- **Mundo episódico** - Cada episódio de percepção-ação é totalmente independente dos outros.
- **Dinamismo** - O mundo é **dinâmico** se o seu estado pode mudar enquanto o agente delibera; caso contrário, o mundo diz-se **estático**.
- **Continuidade** - o mundo é **contínuo** quando a evolução do estado do mundo é um processo contínuo ou sem saltos; caso contrário o mundo diz-se **discreto**.

**Exemplo:** Um agente táxi opera num ambiente dinâmico, enquanto que um agente que jogue palavras-cruzadas opera num ambiente estático. Para um agente carro de condução autónoma, a estrada é um ambiente contínuo. Já para um agente que jogue xadrez, o tabuleiro é um ambiente discreto.

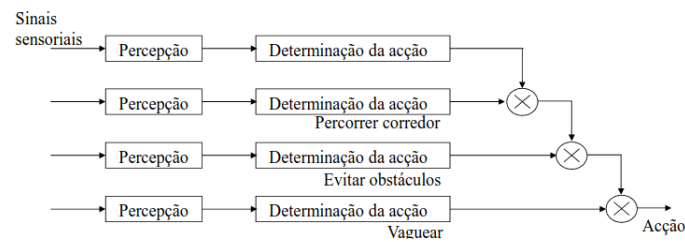


### 3.7.1 Mundo de um agente: Exemplos

Mundo	Acessível	Determinístico	Episódico	Dinâmico	Contínuo
Xadrês s/ relógio	Sim	Sim	Não	Não	Não
Xadrês c/ relógio	Sim	Sim	Não	Semi	Não
Poker	Não	Não	Não	Não	Não
Condução de carro	Não	Não	Não	Sim	Sim
Diagnóstico médico	Não	Não	Não	Sim	Sim
Sistema de análise de imagem	Sim	Sim	Sim	Semi	Sim
Manipulação robótica	Não	Não	Sim	Sim	Sim
Controlo de refinaria	Não	Não	Não	Sim	Sim
Tutor de Inglês interactivo	Não	Não	Não	Sim	Não

## 3.8 Arquiteturas de agentes

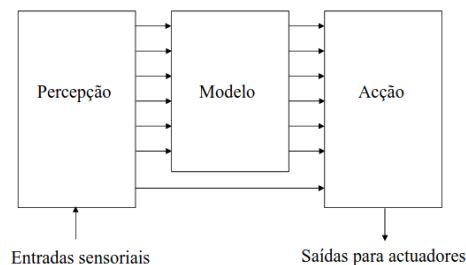
### 3.8.1 Subsunção



Esta arquitetura procura estabelecer a ligação entre a percepção e a acção a vários níveis, organizados em camadas, criando agentes com comportamento simultaneamente reativo e deliberativo.

A camada mais baixa é a mais reativa, diminuindo a reatividade e aumentando o peso da componente deliberativa à medida que se sobe nas camadas. A acção do agente resulta da fusão das decisões das várias camadas

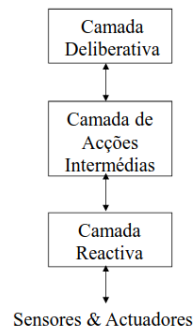
### 3.8.2 Três torres



O processo de **tomada de decisão é simbólico** e feito a partir de um modelo, mas tem **algumas acções reativas**.

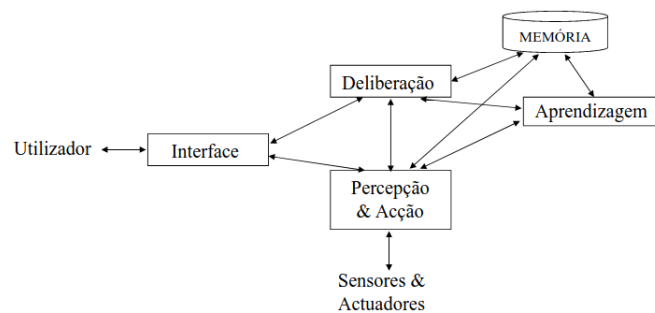
---

### 3.8.3 Três camadas



Para além das camadas deliberativas e reativas tem uma **camada de acções intermédias**, que é o intermediário entre as duas, produzindo uma alteração qualitativa no estado do mundo.

### 3.8.4 CARL



Esta arquitetura (anos 2000) veio introduzir dois novos conceitos:

- **Interfaces** - Facilitam a interação por pessoas não especializadas;
- **Aprendizagem** - Para funcionar num ambiente humano não estruturado vai ter de aprender para se ir adaptando às condições do mundo real. Está assente em **memória**.

## 4 Representação do conhecimento

Para poderem tomar decisões, os agentes (especialmente os deliberativos) precisam de ter algum conhecimento sobre o mundo em que se inserem.

### 4.1 Redes Semânticas

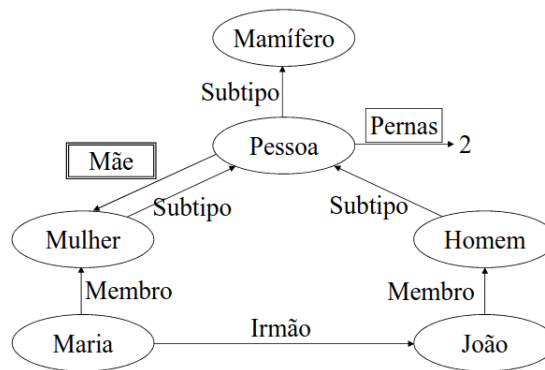
As **redes semânticas** são representações gráficas do conhecimento que facilitam a legibilidade. As redes semânticas podem ser tão expressivas quanto a lógica de primeira ordem.

---

#### 4.1.1 Tipos de Relções

- Sub-tipo (ou sub-conjunto ou ainda sub-classe):  $A \subset B$
- Membro (ou instância) de:  $A \in B$
- Relação objeto-objeto:  $R(A, B)$
- Relação conjunto-objeto:  $\forall x x \in A \rightarrow R(x, B)$
- Relação conjunto-conjunto:  $\forall x x \in A \rightarrow \exists (y \in B \wedge R(x, y))$

#### 4.1.2 Exemplo



#### 4.1.3 Herança

A herança pode ser representada em relações de **sub-tipo**, herdando todas as propriedades dos tipos mais abstratos dos quais descende, ou de **instância**, herdando as todas as propriedades do tipo a que pertence. Aplica-se um **raciocínio não monotónico** devido ao estabelecimento dos **valores por defeito**, propriedades comuns a todos os membros desse tipo e do **cancelamento de herança**, que ocorre quando um sub-tipo altera um valor por defeito herdado de um tipo do qual descende.

#### 4.1.4 Métodos e Demónios

Normalmente, por razões computacionais, usam-se redes semânticas bastante menos expressivas do que a lógica de primeira ordem, deixando-se de lado a **negação**, a **disjunção** e a **quantificação**.

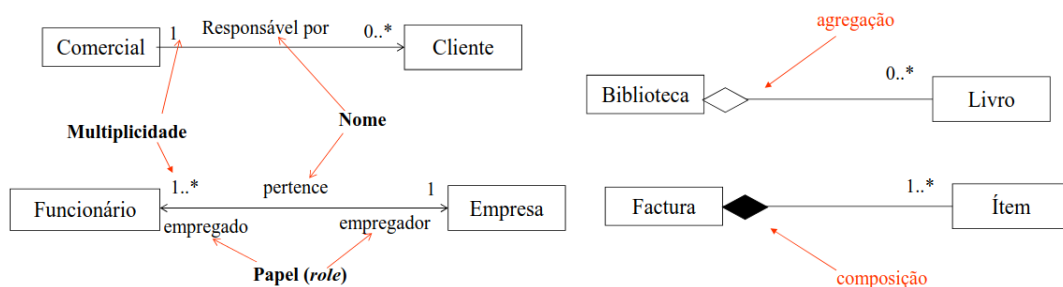
Em contra-partida, nomeadamente nos chamados sistemas de frames, usam-se métodos e demónios:

- **Métodos** - têm uma semântica similar à da programação orientada por objectos.
- **Demónios** - são procedimentos cuja execução é disparada automaticamente quando certas operações de leitura ou escrita são efetuadas.

## 4.2 UML / Diagramas de Classes

Na linguagem gráfica UML (Unified Modelling Language), os diagramas de classes definem as relações existentes entre as diferentes classes de objetos num dado domínio.

- **Classe** - Descrição de um conjunto de objetos que partilham os mesmos atributos, operações, relações e semântica; estes objetos podem ser físicos ou conceituais.
- **Atributo** - Uma propriedade de uma classe.
- **Operação (ou método)** - implementação de um serviço.
- **Instância** - Um objeto que pertence a uma classe.
  - Um aluno lê um livro
    - **Associação** : classe A “usa a”/ “interage com” classe B
  - Um recibo tem vários itens
    - **Composição**: relação todo-parte
  - Uma biblioteca tem vários livros
    - **Agregação**: representa relação estrutural entre instâncias de duas classes, em que a classe agregada existe independentemente da outra
  - Um aluno é uma pessoa
    - **Generalização**: classe A generaliza classe B e B especializa A (super-classe/sub-classe)



## 4.3 Redes semânticas vs UML

<i>Redes semânticas</i>	<i>UML</i>
subtipo(SubTipo,Tipo)	Generalização em diagramas de classes
membro(Obj,Tipo)	Diagramas de objectos
Relação Objecto/Objecto	Associação, agregação e composição em diagramas de objectos
Relação Objecto/Tipo	não tem
Relação Tipo/Tipo	Associação, agregação e composição em diagramas de classes

---

## 4.4 Indução versus Dedução

A **dedução** permite a partir de um conjunto de permissas consideradas verdadeiras, por mecanismos de inferências lógica, tirar conclusões verdadeiras.

No entanto, para adquirir conhecimento novo é necessário haver **indução**, ou seja, a capacidade de inferir regras gerais a partir de observações concretas.

A dedução permite inferir casos particulares a partir de regras gerais, enquanto que a indução permite inferir regras gerais a partir de casos particulares.

### 4.4.1 Exemplo de Indução

Casos conhecidos:

- O gato Tareco gosta de leite;
- O gato Ozzy gosta de leite;

Regra inferida: Os gatos (normalmente) gostam de leite.

Nas redes semânticas, a indução pode ser vista como uma "herança de baixo para cima".

## 4.5 Lógicas

Uma lógica tem:

- **Sintaxe** - descreve o conjunto de frases ou fórmulas que é possível escrever (WFF - Well Formed Formulas);
- **Semântica** - estabelece a relação entre as frases escritas nessa linguagem e os factos que representam;
- **Regras de Inferência** - permitem manipular as frases, gerando umas a partir das outras; as regras de inferência são a base do processo de raciocínio.

### 4.5.1 Lógica Proposicional

A **lógica proposicional** é baseada em proposições, frases declarativas que podem ser **verdadeiras ou falsas**, que podem ser representadas por variáveis proposicionais que tomam o seu valor de verdade. As fórmulas são compostas por uma ou mais variáveis proposicionais ligadas por conectivas lógicas.

**Exemplo:** As proposições não dependem da interpretação, uma vez que, contrariamente aos predicados, as proposições não têm argumentos. "A neve é branca." é uma proposição, que é verdadeira.

---

#### 4.5.2 Lógica de Primeira Ordem

Em contraste temos a **lógica de primeira ordem**, onde há uma distinção entre os objetos e as relações (predicados).

Além de variáveis, usadas para representar termos não especificados e constantes escalares, os objetos podem ainda tomar a forma de expressões funcionais, que se formam pela aplicação de uma função aos seus argumentos. Os objetos podem ser considerados como expressões funcionais cujo número de argumentos é zero.

O nome da relação aplica-se sempre ao seu primeiro elemento (argumento). Os seus argumentos só podem ser termos (nunca relação de relação).

**Exemplo:**  $\text{Potencia}(4, 3)$  é uma **expressão funcional**.  
 $\text{Pai}(\text{Rui}, \text{João})$  é um **predicado**.

#### 4.6 Conectivas Lógicas (LP, LPO)

Comuns às duas lógicas abordadas, servem para combinar frases lógicas elementares por forma a obter frases mais complexas.

- **Conjunção** -  $p \wedge q$
- **Disjunção** -  $p \vee q$
- **Implicação** -  $p \rightarrow q$
- **Negação** -  $\neg p$

#### 4.7 Variáveis, Quantificadores (LPO)

Para quantificar as variáveis da lógica de primeira ordem temos os quantificadores universais e os quantificadores existenciais.

- **Quantificador universal** -  $\forall x$ .
  - $\forall x A \equiv$  "Qualquer que seja  $x$ , a fórmula  $A$  é verdadeira"
- **Quantificador existencial** -  $\exists x$ 
  - $\exists x A \equiv$  "Existe um  $x$ , para o qual a fórmula  $A$  é verdadeira"

Se  $A$  é uma fórmula bem formada (FBF), então a sua quantificação também é uma FBF.

---

## 4.8 Gramáticas (LPO)

$Fórmula \rightarrow FórmulaAtômica$   
|  $Fórmula\ Conectiva\ Formula$   
|  $Quantificador\ Variável, \dots Fórmula$   
|  $'\neg' Fórmula$   
|  $'(' Fórmula \ )'$   
 $FórmulaAtômica \rightarrow Predicado\ '('\ Termo\ ',' \dots \ )' \mid Termo\ '='\ Termo$   
 $Termo \rightarrow Função\ '('\ Termo\ ',' \dots \ )' \mid Constante \mid Variável$   
 $Conectiva \rightarrow '\Rightarrow' \mid '\wedge' \mid '\vee' \mid '\Leftrightarrow'$   
 $Quantificador \rightarrow '\exists' \mid '\forall'$   
 $Constante \rightarrow A \mid X1 \mid Paula \mid \dots$   
 $Variável \rightarrow a \mid x \mid s \mid \dots$   
 $Predicado \rightarrow Portista \mid Cor \mid \dots$   
 $Função \rightarrow Registo \mid Mãe \mid \dots$

### 4.8.1 Exemplo

- “Todos em Oxford são espertos”:
  - $\forall x \text{ Estuda}(x, \text{Oxford}) \Rightarrow \text{Esperto}(x)$
  - Erro comum: Usar  $\wedge$  em vez de  $\Rightarrow$   
 $\forall x \text{ Estuda}(x, \text{Oxford}) \wedge \text{Esperto}(x)$   
Significa “Todos estão em Oxford e todos são espertos”
- “Alguém em Oxford é esperto”:
  - $\exists x \text{ Estuda}(x, \text{Oxford}) \wedge \text{Esperto}(x)$
  - Erro comum: Usar  $\Rightarrow$  em vez de  $\wedge$   
 $\exists x \text{ Estuda}(x, \text{Oxford}) \Rightarrow \text{Esperto}(x)$   
qualquer estudante de outra universidade forneceria uma interpretação verdadeira.
- “Existe uma pessoa que gosta de toda a gente”
  - $\exists x \forall y \text{ Gosta}(x, y)$

## 4.9 Interpretações em Lógica Proposicional

Na lógica proposicional, a interpretação de uma fórmula é a atribuição de valores de verdade ou falsidade às várias proposições que nela ocorrem.  $A \wedge B$  tem quatro interpretações possíveis que resultam da combinação dos valores de verdade e falsidade de A e B.

- **Satisfatibilidade** - Uma interpretação satisfaz uma fórmula sse a fórmula toma o valor “verdadeiro” para essa interpretação.
- **Modelo de uma fórmula** - uma interpretação que satisfaz essa fórmula.
- **Tautologia** - uma fórmula cujo valor é “verdadeiro” em qualquer interpretação.

---

#### 4.10 Interpretações em Lógica de Primeira Ordem

Na **lógica de primeira ordem**, a interpretação de uma fórmula é o estabelecimento de uma correspondência entre as constantes que ocorrem na fórmula e os objetos do mundo, funções e relações que essas constantes representam.

#### 4.11 Regras de Substituição

Estas regras permitem substituir uma fórmula por outra equivalente. São válidas tanto para a lógica proposicional como para a lógica de primeira ordem.

- |  |   |
|--|---|
| • Leis de DeMorgan                                 | • Comutação   |
| $\neg (A \wedge B) \equiv \neg A \vee \neg B$      | $A \wedge B \equiv B \wedge A$                              |
| $\neg (A \vee B) \equiv \neg A \wedge \neg B$      | $A \vee B \equiv B \vee A$                                  |
| • Dupla negação:                                   | • Associação:   |
| $\neg \neg A \equiv A$                             | $(A \wedge B) \wedge C \equiv A \wedge (B \wedge C)$        |
| • Definição da implicação:                         | $(A \vee B) \vee C \equiv A \vee (B \vee C)$                |
| $A \Rightarrow B \equiv \neg A \vee B$             | • Distribuição:   |
| • Transposição:                                    | $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$ |
| $A \Rightarrow B \equiv \neg B \Rightarrow \neg A$ | $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$   |

Leis de DeMorgan generalizadas (estas são específicas da lógica de primeira ordem):

$$\begin{aligned}\neg(\forall x P(x)) &\equiv \exists x \neg P(x) \\ \neg(\exists x P(x)) &\equiv \forall x \neg P(x)\end{aligned}$$

#### 4.12 CNF e Forma Clausal