

Programación Lógica y Funcional

Paradigmas de Programación

Dr. Alejandro Guerra-Hernández

Instituto de Investigaciones en Inteligencia Artificial

Universidad Veracruzana

*Campus Sur, Calle Paseo Lote II, Sección Segunda No 112,
Nuevo Xalapa, Xalapa, Ver., México 91097*

<mailto:aguerra@uv.mx>

<https://www.uv.mx/personal/aguerra/plf>

Maestría en Inteligencia Artificial 2025



Universidad Veracruzana

¿Qué es un paradigma?

- ▶ Según Kuhn [12], un paradigma es un **modelo de trabajo compartido** por una comunidad científica, cuyos miembros están de acuerdo en qué es un **problema legítimo** y cuál es una **solución legítima** del problema.
- ▶ Los paradigmas permiten **compartir** conceptos básicos, procedimientos, etc.
- ▶ Las revoluciones científicas han emergido de **cambios** en los paradigmas dominantes.



Paradigmas de Programación

- ▶ Floyd [9] (*ACM Turing Award*, 1978), discute el **paradigma** aplicado a las Ciencias de la Computación:
 - ▶ Los libros de texto parecen implicar que el contenido de la ciencia está **exclusivamente** ejemplificado por las observaciones, leyes y teorías descritos en sus páginas.
 - ▶ El estudio de los paradigmas constituye la **fuente principal** en la formación del estudiante, para preparar su integración a una comunidad científica particular donde llevará a cabo su práctica.
 - ▶ Los lenguajes de programación **estimulan** el uso de algunos paradigmas, mientras que **inhiben** notoriamente otros.



Problemas en la programación y los paradigmas

- ▶ Los problemas al considerar aspectos como:
 - ▶ Confianza.
 - ▶ Puntualidad.
 - ▶ Flexibilidad.
 - ▶ Eficiencia.
 - ▶ Costo.
- ▶ Se deben a:
 - ▶ Un **repertorio inadecuado** de paradigmas de programación.
 - ▶ Un **conocimiento deficiente** de los paradigmas existentes.
 - ▶ La **manera en que enseñamos** esos paradigmas.
 - ▶ Y a la manera en que los lenguajes de programación **soportan**, o no, los paradigmas de sus comunidades de usuarios.



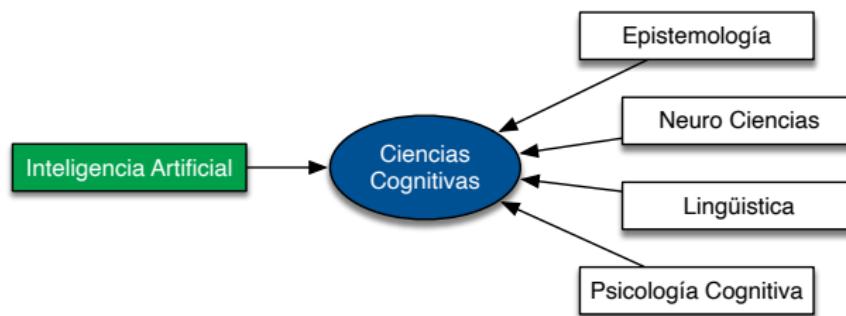
Solución

- ▶ Si el avance general del arte la programación requiere de la continua invención y **elaboración de paradigmas**; el avance individual requiere de la **expansión del repertorio** personal de paradigmas.
- ▶ Esto constituye la **justificación** del diseño curricular de nuestro curso, que propone expandir el repertorio de Paradigmas de Programación y posibilitar la elaboración de nuevos paradigmas, para alcanzar el objetivo del mismo.
- ▶ ¿Qué Paradigmas de Programación son **relevantes** en la Inteligencia Artificial (IA)?
- ▶ ¿IA?



¿Cuál es el sujeto de estudio de la IA?

- ▶ La IA tiene como objeto el estudio de las **entidades inteligentes**.



- ▶ *Modus operandi*: La construcción de **agentes racionales** [18].
- ▶ Es una **ciencia de lo artificial** [22].



Línea de tiempo

- 1936 • Máquina de Turing - Cálculo- λ
- 1943 • Cálculo Lógico del Sistema Nervioso
- 1950 • ¿Puede la Máquina de Turing pensar?
- 1960 • Funciones recursivas de expresiones simbólicas (Lisp)
- 1965 • Principio de resolución
- 1972 • Lógica para Funciones Computables (ML)
- 1973 • Programación Lógica (Prolog)
- 1976 • Hipótesis del Sistema Simbólico Físico



Church y Turing



Universidad Veracruzana

Máquinas inteligentes ¿Qué máquinas?

- ▶ Turing [24]: ¿Pueden las máquinas pensar?
- ▶ Objetiones, p. ej., Searle [19] y Dreyfus y Dreyfus [8].
- ▶ La adopción de la computadora digital como el tipo de máquina a considerar, provee uno de los fundamentos teóricos de la IA: **Pensar es una computación**.
- ▶ Turing [23]: Una computación es una manipulación formal de **símbolos**, no interpretados, mediante la aplicación de reglas formales.
- ▶ Church [4]: Un formalismo equivalente, el Cálculo- λ .



Pitts y McCulloch



Universidad Veracruzana

Paradigma conexionista

- ▶ McCulloch y Pitts [14] son los primeros en proponer un cómputo inspirado en nuestro **sistema nervioso**, una abstracción de las redes neuronales:
 - ▶ Ciertos tipos de red neuronal estrictamente definidos (acíclicos y de estructura fija) pueden computar, en principio, cierta clase de funciones lógicas.
 - ▶ Toda función del cálculo proposicional puede computarse con una red neuronal relativamente simple.
 - ▶ Toda red computa una función que es computable en una máquina de Turing, y vice-versa, toda función computable en una máquina de Turing puede computarse con una red neuronal.

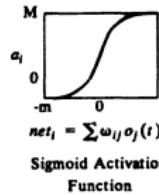
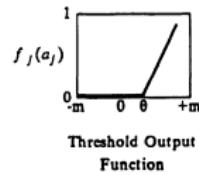
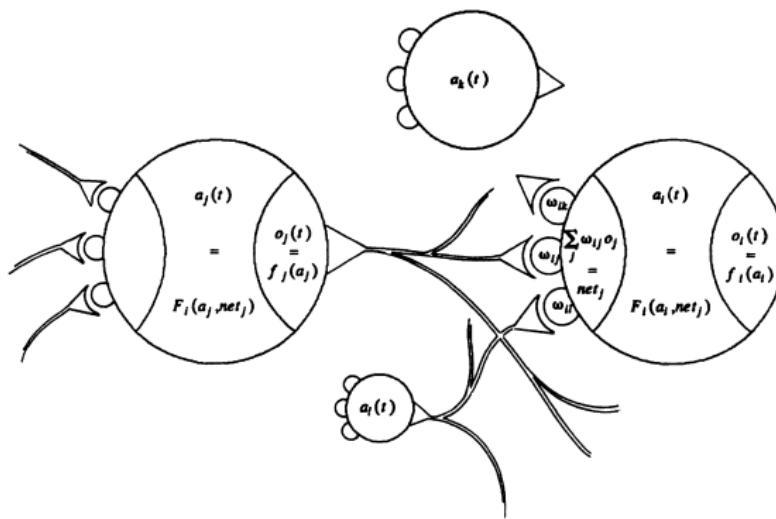


Procesamiento Distribuido Paralelo (PDP) I

- ▶ Se debe a Rumelhart, Hinton, McClelland et al. [17] y está caracterizado por:
 - ▶ Un conjunto de unidades de procesamiento.
 - ▶ Un nivel de activación para cada unidad.
 - ▶ Un patrón de conectividad.
 - ▶ Una regla de propagación de la activación.
 - ▶ Una regla de activación para combinar las entradas de cada unidad.
 - ▶ Una regla de aprendizaje para modificar los pesos de las conexiones con base en la experiencia.
 - ▶ Un ambiente que provee al sistema con experiencias.
- ▶ Siegelman y Sontag [21] demuestran que, en principio, una red neuronal puede simular una **máquina de Turing**.



El modelo PDP gráficamente



¿Paradigma de programación?

- ▶ No es evidente que el enfoque conexionista haya producido paradigmas de programación propios.
- ▶ Los paradigmas imperativo, orientado a objetos y principalmente el de **cómputo científico** (Octave, Matlab, R), parecieran cubrir las necesidades de esta comunidad.
- ▶ En todo caso, estas herramientas serán revisadas en la experiencia educativa optativa **Redes Neuronales** (tercer semestre).



Universidad Veracruzana

Paradigma simbólico

- ▶ Newell y Simon [15] Un **Sistema Simbólico Físico**:
 - ▶ Un conjunto de **símbolos**, patrones físicos, que pueden ser componentes de **expresiones** u otras estructuras simbólicas.
 - ▶ Toda expresión está compuesta por ocurrencias de símbolos relacionados de alguna forma física, p. ej., un símbolo es el siguiente con respecto a otro.
 - ▶ En todo momento, el sistema contiene una **colección de expresiones**.
 - ▶ Una colección de **procesos** que operan sobre expresiones para producir nuevas expresiones: creación, modificación, reproducción y destrucción.
 - ▶ Tal sistema existe en un mundo de objetos **más extenso** que sus expresiones.

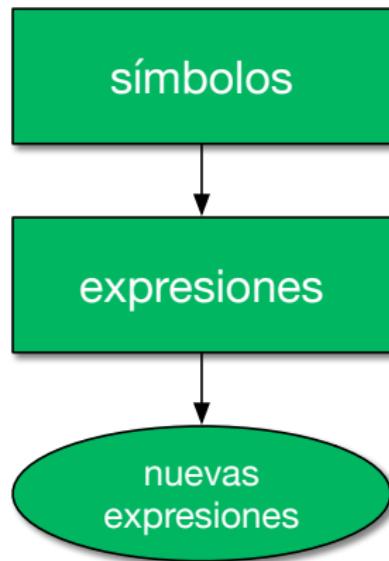


Simon y Newell



Universidad Veracruzana

Gráficamente



3, 4, +, (,)

(+ 3 4)

7



Universidad Veracruzana

Nociones auxiliares

Designación. Una expresión designa a un objeto, si dada la expresión, el sistema puede **afectar** al objeto o **comportarse** de maneras dependientes del objeto.

Interpretación. El sistema puede interpretar una expresión, si ésta designa un proceso y, dada la expresión, el sistema puede **ejecutar** ese proceso.



Universidad Veracruzana

Restricciones adicionales

- ▶ Un símbolo debe poder ser usado para designar **cualquier** expresión, sin prescripciones *a priori* sobre que expresiones puede designar.
- ▶ Existen expresiones para designar **todos** los procesos que el sistema puede ejecutar.
- ▶ Existen procesos para crear y modificar **cualquier** expresión de manera arbitraria.
- ▶ Las expresiones son **estables**, una vez creadas, seguirán existiendo hasta que sean explícitamente modificadas o borradas.
- ▶ El número de expresiones que un sistema puede contener, es **ilimitado**.



Hipótesis del Sistema Simbólico Físico

Un sistema simbólico físico tiene los medios necesarios y suficientes para generar comportamiento inteligente general.

► Precisiones:

Necesario. Cualquier sistema que exhiba inteligencia general, podrá verificarse mediante **análisis**, es un sistema simbólico físico.

Suficiente. Cualquier sistema simbólico físico del **tamaño adecuado**, puede ser organizado para exhibir inteligencia general.

► Puesto que todo sistema simbólico físico es una máquina universal de Turing, esto implica que la inteligencia es **implementable** en una computadora universal.



Universidad Veracruzana

Paradigmas de Programación Simbólicos I

Lógico. La **deducción lógica** es una forma de computación universal. Su lenguaje más representativo es **Prolog** [7], basado en el principio de resolución-SL de Kowalski y Kuehner [11], una **regla de inferencia** para demostrar que una cláusula es consecuencia de un conjunto de ellas y computar los valores de las variables en la cláusula, que hacen esto posible. Otros lenguajes: Datalog, Mercury, Logtalk.



Paradigmas de Programación Simbólicos II

Funcional. La computación en términos de **funciones matemáticas** y su transformación a través de **formas funcionales**. Su concepción original se debe a Backus [1], aunque tiene raíces en el Cálculo- λ de Church [5]. El primer lenguaje con características funcionales fue **Lisp** de McCarthy [13], que es de nuestro interés porque además es una implementación de sistema simbólico físico. Otros lenguajes más próximos a la idea original de Backus incluyen ML, Ocaml y Haskell.



Un trabajito propuesto en 1970 [6]

► ¿Cómo programar un sistema como el siguiente?

- 1 > Los gatos matan ratones.
- 2 > Tom es un gato al que no le gustan los
- 3 ratones que comen queso.
- 4 > Jerry es un ratón que come queso.
- 5 > Max no es un gato.
- 6 > Qué hace Tom?
- 7 A Tom no le gustan los ratones que comen queso.
- 8 Tom mata ratones.
- 9 > Quién es un gato?
- 10 Tom.
- 11 > Qué come Jerry?
- 12 Queso.



Elementos de una Lógica

- ▶ Observen que este lenguaje define **símbolos** para:
 - ▶ Designar elementos (Tom, Jerry, etc.).
 - ▶ Designar conjuntos (Gatos, Ratones, etc.).
 - ▶ Relaciones binarias (Comer, Matar, Gustar, etc.).
 - ▶ Funciones (The, Subset, True)
- ▶ Reglas para formar **expresiones** válidas (sintaxis).
- ▶ Un método de **razonamiento automático** (semántica).



À la française

- 1965 Alan Robinson [16] formula el **principio de resolución**. Reglas de inferencias menos humanas, pero más eficientes.
- 1970 Alain Colmerauer, Philippe Roussel y Robert Pasero [6] trabajan en traducción automática y **procesamiento de lenguaje natural**.
- 1971 Robert Kowalski [11] define la **resolución-SL**.
- 1974 El lenguaje **Prolog**.



Una versión más universal

Siglo XIX En su segunda mitad, Gottlob Frege introduce la **lógica de primer orden**. Modificada a su forma actual por Giuseppe Peano y Bertrand Russell.

1930s Kurt Göedel y Jacques Herbrand estudiaron la noción de computabilidad basada en **derivaciones**.

1965 La resolución y **unificación** por Alain Robinson.

1974 La resolución-SLD de Kowalski: fin del debate (en ese momento) sobre representaciones **declarativas** y **procedimentales**. Prolog.

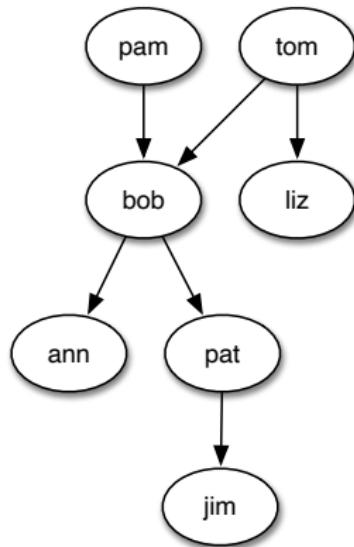
1983 La máquina abstracta de Warren (**WAM**) [25]. Compilación independiente al procesador usado.



Universidad Veracruzana

Objetos, Relaciones y Programas

- ▶ Asumamos que queremos razonar sobre una genealogía [2]:



```
1 progenitor(pam,bob).  
2 progenitor(tom,bob).  
3 progenitor(tom,liz).  
4 progenitor(bob,ann).  
5 progenitor(bob,pat).  
6 progenitor(pat,jim).
```



Cargando Prolog

- ▶ Prolog se ejecuta desde una terminal:

```
1  > swipl
2  Welcome to SWI-Prolog (threaded, 64 bits, v 8.1.8)
3  SWI-Prolog comes with ABSOLUTELY NO WARRANTY.
4  This is free software.
5  Please run ?- license. for legal details.
6
7  For online help, visit http://www.swi-prolog.org
8  For built-in help, use ?- help(Topic). or ?- apropos(Word).
9
10 ?-
```



Universidad Veracruzana

REPL

- ▶ El sistema está en un ciclo **Read-Eval-Print** (REPL).

```
1  ?- [clase01].  
2  % progenitor compiled  
3  true.  
4  ?- progenitor(bob,pat).  
5  true  
6  ?- progenitor(liz,pat).  
7  false.  
8  ?- progenitor(tom,ben).  
9  false.  
10 ?-
```

```
1  progenitor(pam,bob).  
2  progenitor(tom,bob).  
3  progenitor(tom,liz).  
4  progenitor(bob,ann).  
5  progenitor(bob,pat).  
6  progenitor(pat,jim).
```



Variabes

- ▶ Se pueden hacer preguntas más interesantes usando **variables**:

```
1  ?- progenitor(X,liz).  
2  X = tom.  
3  ?- progenitor(bob,X).  
4  X = ann ;  
5  X = pat.  
6  
7  ?-
```

```
1  progenitor(pam,bob).  
2  progenitor(tom,bob).  
3  progenitor(tom,liz).  
4  progenitor(bob,ann).  
5  progenitor(bob,pat).  
6  progenitor(pat,jim).
```

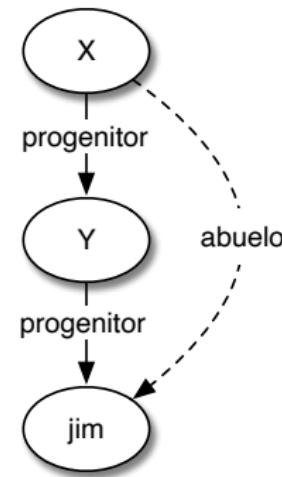


¿Quién es el abuelo de Jim?

```

1  ?- progenitor(Y,jim), progenitor(X,Y).
2  Y = pat
3  X = bob
4  ?- progenitor(X,Y), progenitor(Y,jim).
5  X = bob
6  Y = pat
7  ?- progenitor(tom,X), progenitor(X,Y).
8  X = bob
9  Y = ann ;
10 X = bob
11 Y = pat ;
12 false.

```



Resumiendo

- ▶ Es sencillo definir en Prolog una **relación** especificando las *n*-tuplas de objetos que la satisfacen. *n* es conocido como **aridad**.
- ▶ Un programa Prolog consiste de **cláusulas**.
- ▶ Los argumentos de una relación pueden ser: objetos concretos o **constantes** como tom y ann; objetos generales o **variables** como X e Y.
- ▶ Las preguntas planteadas a Prolog consisten en una o más metas. Una secuencia de metas significa **conjunción**.
- ▶ La respuesta a una pregunta puede ser positiva o negativa, dependiendo de si la meta se puede **satisfacer** o no.
- ▶ Si varias respuestas satisfacen una pregunta, Prolog encontrará **tantas** como el usuario quiera.



Reglas

- ▶ Las reglas tienen dos partes:
 - ▶ Una parte condicional (el lado derecho de la regla o **cuerpo** de la regla).
 - ▶ Una conclusión (el lado izquierdo de la regla o **cabeza** de la regla).
- ▶ Ejemplo:

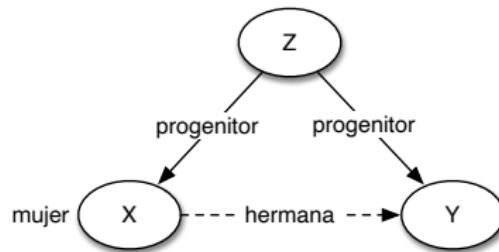
```
1  vastago(Y,X) :- progenitor(X,Y).
```



Universidad Veracruzana

Extendiendo nuestro conocimiento

```
1  mujer(pam).  
2  mujer(liz).  
3  mujer(pat).  
4  mujer(ann).  
5  hombre(tom).  
6  hombre(bob).  
7  hombre(jim).  
8  hermana(X,Y) :-  
9    progenitor(Z,X),  
10   progenitor(Z,Y),  
11   mujer(X).  
12  abuela(X,Y) :-  
13   progenitor(X,Z),  
14   progenitor(Z,Y),  
15   mujer(X).  
16  madre(X,Y) :-  
17   progenitor(X,Y),  
18   mujer(X).
```



Probando el nuevo conocimiento

- ▶ La relación *hermana/2* presenta una anomalía:

```
1  :- hermana(ann,pat).  
2  true.  
3  :- hermana(X,pat).  
4  X = ann ;  
5  X = pat ;  
6  false.
```

```
1  hermana(X,Y) :-  
2  progenitor(Z,X),  
3  progenitor(Z,Y),  
4  mujer(X),  
5  dif(X,Y).
```



Universidad Veracruzana

Resumiendo I

- ▶ Los programas Prolog pueden **extenderse** fácilmente agregando nuevas cláusulas.
- ▶ Las cláusulas en Prolog son de tres tipos: **hechos**, **reglas** y **metas**.
- ▶ Los hechos declaran cosas que son verdaderas siempre, **incondicionalmente**.
- ▶ Las reglas declaran cosas que son verdaderas dependiendo de ciertas **condiciones**.
- ▶ Por medio de las metas el usuario puede **computar** qué cosas son verdaderas.
- ▶ Las cláusulas de Prolog tienen **cabeza** y **cuerpo**. El cuerpo es una lista de metas separadas por comas. Las comas implican **conjunción**.

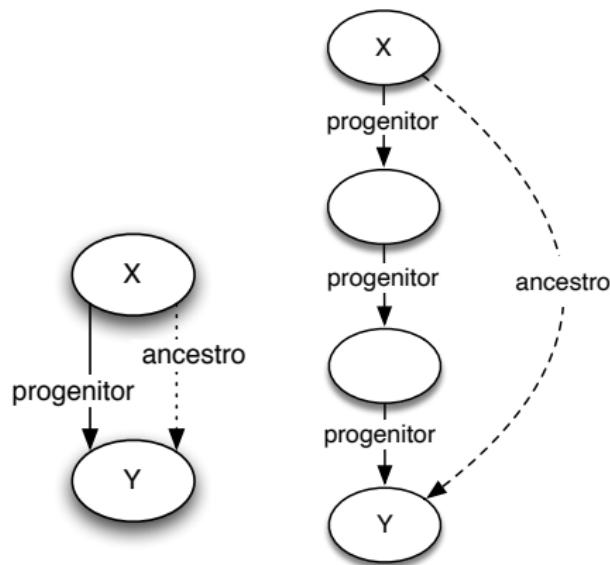


Resumiendo II

- ▶ Los hechos son cláusulas con el **cuerpo vacío**; las preguntas tienen la **cabeza vacía**; y las reglas tienen **cabeza y cuerpo**.
- ▶ En el curso de una computación, las variables pueden ser **substituidas** por otros objetos.
- ▶ Las variables se asumen **cuantificadas universalmente**. La cuantificación **existencial** sólo es posible en las variables que aparecen en el cuerpo de una cláusula.



Extensional vs Intensional



```

1  ancestro(X,Z) :- progenitor(X,Z).
2  ancestro(X,Z) :- progenitor(X,Y),
3  progenitor(Y,Z).
4  ancestro(X,Z) :- progenitor(X,Y0),
5  progenitor(Y0,Y1),
6  progenitor(Y1,Z).
7  ...

```



Definición recursiva

- Ancestro definido en términos de ancestro:

```
1  ?- ancestro(pam,X).  
2  X = bob ;  
3  X = ann ;  
4  X = pat ;  
5  X = jim ;  
6  false.  
7  ?- ancestro(X,jim).  
8  X = pat ;  
9  X = bob ;  
10 X = pam ;  
11 X = tom ;  
12 false.
```

```
1  ancestro(X,Z) :-  
2      progenitor(X,Z).  
3  
4  ancestro(X,Z) :-  
5      progenitor(X,Y),  
6      ancestro(Y,Z).
```



Resumiendo

- ▶ Las reglas recursivas definen conceptos en términos de **ellos mismos**.
- ▶ Están definidas por al menos dos casos: uno **terminal** (no recursivo) y la llamada **recursiva**.
- ▶ Una relación recursiva define **intensionalmente** un concepto.
- ▶ **intenSional** \neq **intenCional**.



Demostración como cómputo

- ▶ Satisfacer una meta implica **demostrar** que la meta es verdadera, asumiendo que las relaciones en el programa lógico son verdaderas.
- ▶ Satisfacer una meta significa entonces demostrar que la meta es una **consecuencia lógica** de los hechos y reglas definidas en un programa.
- ▶ Si la pregunta contiene variables, Prolog necesita también **computar** cuales son los objetos particulares (que remplazaran a las variables) para los cuales la meta se satisface.



Programas lógicos como matemáticas

- ▶ Prolog acepta hechos y reglas como un conjunto de **axiomas**.
- ▶ El usuario plantea preguntas **teoremas**.
- ▶ Prolog trata de probar este teorema, es decir, **demostrar** que el teorema se sigue lógicamente de los axiomas.



Ejemplos

Axioma 1 Sócrates es un hombre.

```
1  hombre(socrates).  
2  falible(X) :-  
3      hombre(X).
```

Axioma 2 Todos los hombres son falibles.

Conclusión Sócrates es falible.

```
1  ?- falible(socrates)  
2  true.
```

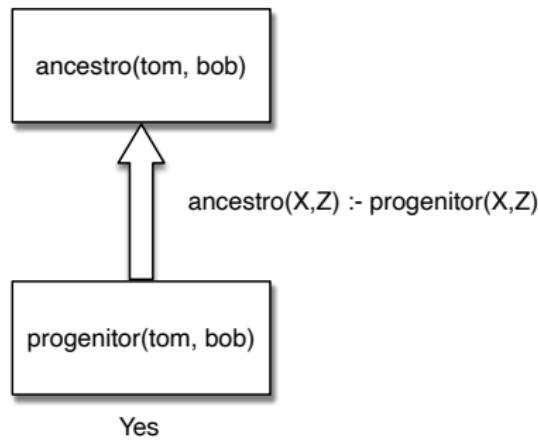


Universidad Veracruzana

?- ancestro(tom,bob).

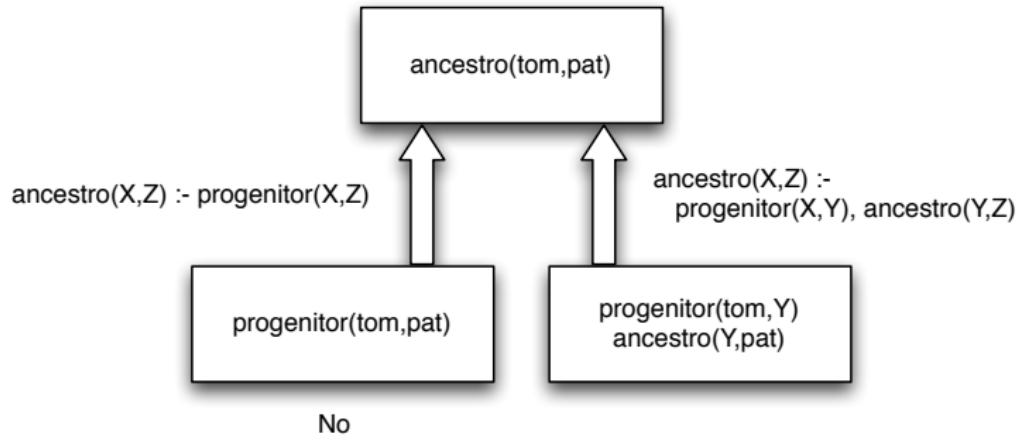
► El proceso en un paso:

```
1  ancestro(X,Z) :-  
2      progenitor(X,Z).  
3  
4  ancestro(X,Z) :-  
5      progenitor(X,Y),  
6      ancestro(Y,Z).
```

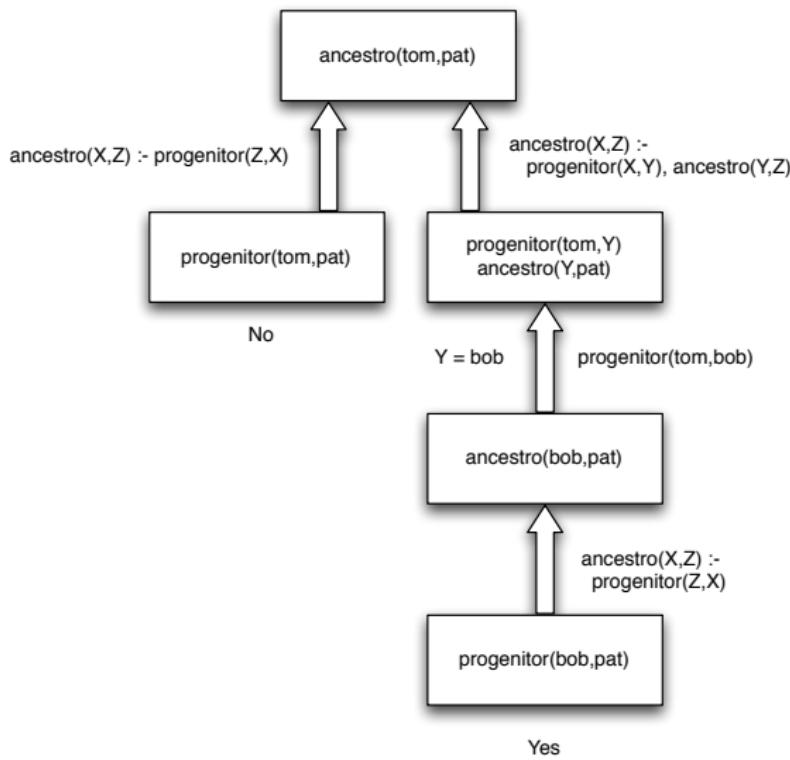


?- ancestro(tom,pat).

- ▶ El proceso en dos pasos:



?- ancestro(tom,pat). Recursión



Traza de un programa

```
1  ?- trace.
2  true.
3  [trace]  ?- ancestro(tom,pat).
4  Call: (6) ancestro(tom, pat) ?
5  Call: (7) progenitor(tom, pat) ?
6  Fail: (7) progenitor(tom, pat) ?
7  Redo: (6) ancestro(tom, pat) ?
8  Call: (7) progenitor(_G404, pat) ?
9  Exit: (7) progenitor(bob, pat) ?
10 Call: (7) ancestro(tom, bob) ?
11 Call: (8) progenitor(tom, bob) ?
12 Exit: (8) progenitor(tom, bob) ?
13 Exit: (7) ancestro(tom, bob) ?
14 Exit: (6) ancestro(tom, pat) ?
15 true
16 [trace]  ?- notrace, nodebug.
17 true.
18 ?-
```



Relevancia de la Programación Funcional

- ▶ ¿Cuales son las **ventajas** de la Programación Funcional?
- ▶ Se suelen expresar como las prácticas de otros paradigmas de programación que **no están presentes** en la programación funcional:
 - ▶ En programación funcional no hay **instrucciones de asignación**;
 - ▶ La evaluación de un programa funcional no tiene **efectos colaterales**; y
 - ▶ Las funciones pueden evaluarse en cualquier orden, por lo que en programación funcional no hay que preocuparse por el **flujo de control**.



Relevancia en términos positivos

- ▶ Siguiendo la propuesta de Hughes [10]:
 - ▶ Las **funciones de orden superior** y la **evaluación postergada**,
 - ▶ son herramientas conceptuales de la programación funcional que nos permiten descomponer problemas más allá del
 - ▶ **diseño modular** que inducen otros paradigmas de programación, como la estructurada.



Propiedades del Paradigma Funcional

1. Funciones puras y composición funcional.
2. Interfaz manifiesta.
3. Transparencia referencial.
4. Funciones de orden superior.
5. Recursividad.



Funciones

- ▶ Una función provee un mapeo entre objetos tomados de un conjunto de valores llamado **dominio** y objetos en otro conjunto llamado **codominio** o **rango**.
- ▶ **Ejemplo:** La función *signo* mapea el conjunto de los enteros a uno de los valores en $\{pos, neg, cero\}$, dependiendo si el entero es positivo, negativo o cero. El dominio de *signo* es entonces el conjunto de los enteros y su rango es el conjunto $\{pos, neg, cero\}$.

$$signo : \mathbb{Z} \mapsto \{pos, neg, cero\}$$



Caracterización por extensión

- Mostrando **explícitamente** los elementos en el dominio y el rango y el mapeo que establece la función.

⋮

signo(-2) = *neg*

signo(-1) = *neg*

signo(0) = *cero*

signo(1) = *pos*

signo(2) = *pos*

⋮



Caracterización intensional

- ▶ Podemos caracterizar una función a través de reglas que describan el mapeo que establece la función. A esta caracterización se le llama por intención o **intensional**.

$$\text{signo}(x) = \begin{cases} \text{neg} & \text{si } x < 0 \\ \text{cero} & \text{si } x = 0 \\ \text{pos} & \text{si } x > 0 \end{cases}$$

- ▶ En la definición de *signo/1*, la *x* se conoce como el **parámetro formal** de la función y representa cualquier elemento del dominio.
- ▶ 1 se conoce como la **aridad** de la función y denota el número de parámetros formales de ésta.



Funciones totales y parciales

- ▶ Si una función aplica a todos los elementos del dominio, se dice que se trata de una función **total**.
- ▶ Si la regla omitiera a uno o más elementos del dominio, diríamos que es una función **parcial**.

$$signo2(x) = \begin{cases} neg & \text{si } x < 0 \\ pos & \text{si } x > 0 \end{cases}$$

- ▶ La **signatura** es $signo2 : \mathbb{Z} \setminus 0 \mapsto \{neg, pos\}$



Lenguajes fuertemente tipificados

- ▶ En los lenguajes funcionales **fuertemente tipificados**, como Ocaml [3], el dominio y rango de una función deben especificarse, ya sea explícitamente o bien mediante un sistema de **inferencia de tipos**.
- ▶ **Ejemplo:** Definimos *add/2* en Ocaml, una función que suma sus dos argumentos:

```
1 # let add x y = x + y ;;
2 val add : int -> int -> int = <fun>
```



Universidad Veracruzana

Signo en Ocaml

- ▶ Si queremos definir *signo*/1 en Ocaml, antes debemos definir un tipo de datos que represente al rango de la función: $\{pos, cero, neg\}$:

```
1  # type signos = Neg | Cero | Pos;;
2  type signos = Neg | Cero | Pos
3  # let signo x =
4      if x<0 then Neg else
5          if x=0 then Cero else Pos;;
6  val signo : int -> signos = <fun>
7  # signo 5;;
8  - : signos = Pos
9  # signo 0;;
10 - : signos = Cero
11 # signo (-2);;
12 - : signos = Neg
```



Lenguajes tipificados dinámicamente

- ▶ En los lenguajes tipificados dinámicamente, como Lisp [20], esto no es necesario.
- ▶ **Ejemplo:** En Lisp, la definición de *add/2* sería como sigue:

```
1 CL-USER> (defun add (x y) (+ x y))
2 ADD
3 CL-USER> (add 3 4)
4 7
5 CL-USER> (type-of #'add)
6 FUNCTION
7 CL-USER> (type-of (add 3 4))
8 (INTEGER 0 4611686018427387903)
9 CL-USER> (type-of (add 3.0 4.0))
10 SINGLE-FLOAT
```



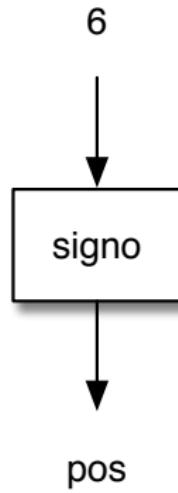
Signo en Lisp

- En Lisp, la definición de signo no requiere de una declaración de tipo:

```
1  CL-USER> (defun signo (x)
2          (cond ((< x 0) 'neg)
3                  ((zerop x) 'cero)
4                  (t 'pos)))
5
6  SIGNO
7  CL-USER> (signo 3)
8  POS
9  CL-USER> (signo -2)
10 NEG
11 CL-USER> (signo 0)
12 CERO
```



Funciones como cajas negras



- ▶ 6 aquí se conoce como **parámetro actual** de la función, es decir el valor que se le provee a la función.
- ▶ Al proceso de proveer un parámetro actual a una función se le conoce como **aplicación** de la función. En notación: $signo(6)$.
- ▶ Decimos que está aplicación **evaluó** a *pos*, lo que escribimos como: $signo(6) \rightarrow pos$.



Composición funcional

- ▶ La idea de una función como una transformadora de entradas en salidas es uno de los fundamentos de la programación funcional.
- ▶ Las cajas negras proveen bloques de construcción para un programa funcional, y uniendo varias cajas es posible especificar operaciones más sofisticadas.
- ▶ El proceso de “ensamblar cajas” se conoce como **composición** de funciones.



Ejemplo: *max/1*

- ▶ Para ilustrar este proceso de **composición**, definimos a continuación la función *max* que computa el máximo de un par de números *m* y *n*:

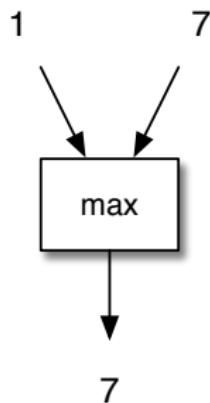
$$\max(m, n) = \begin{cases} m & \text{si } m > n \\ n & \text{en cualquier otro caso} \end{cases}$$

- ▶ El dominio de *max* es el conjunto de **pares** de números enteros y el rango es el conjunto de los enteros. Esto se puede escribir en notación matemática como:

$$\max : \text{int} \times \text{int} \rightarrow \text{int}$$



$max/1$ como caja negra



- ▶ Podemos ver a max como una caja negra para computar el máximo de dos números.
- ▶ Lo cual escribimos:

$$max(1, 7) \rightarrow 7$$



max/1 en Ocaml

- En Ocaml, nuestra función *max/1* quedaría definida como:

```
1  # let max (m,n) = if m > n then m else n ;;
2  val max : 'a * 'a -> 'a = <fun>
3  # max(1,7) ;;
4  - : int = 7
```

- La línea 2 nos dice que *max/1* es una **función polimórfica**, es decir, que acepta argumentos de varios tipos: Lo mismo puede computar el máximo de un par de enteros, que de un par de reales.



max/1 estrictamente sobre enteros

- ▶ Si queremos estrictamente una función de pares de enteros a enteros podemos usar:

```
1  # let max ((m:int),(n:int)) = if m > n then m else n ;;
2  val max : int * int -> int = <fun>
```

- ▶ Como saben, la principal utilidad de los tipos es **prevenir errores** en tiempo de ejecución y hacer **eficiente** el código compilado.
- ▶ Observen que en Ocaml los tipos son **inferidos**.



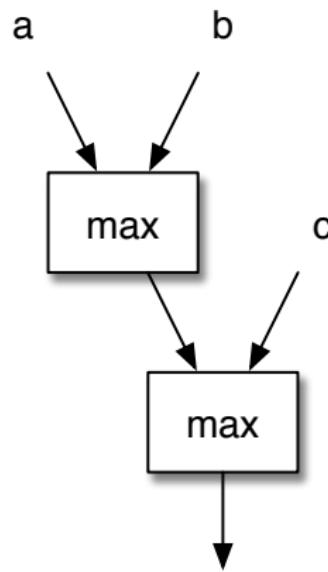
¿Y si ahora nos encargan *max3/1* para tripletas?

- ▶ Podemos definir esta función como *max3*:

$$\max3(a, b, c) = \begin{cases} a & \text{si } a \geq b \text{ y } a > c \text{ ó } a \geq c \text{ y } a > b \\ b & \text{si } b \geq a \text{ y } b > c \text{ ó } b \geq c \text{ y } b > a \\ c & \text{si } c \geq a \text{ y } c > b \text{ ó } c \geq b \text{ y } c > a \\ a & \text{en cualquier otro caso} \end{cases}$$



Pero es más divertido jugar con legos



Universidad Veracruzana

max3/1 en Ocaml

- ▶ En Ocaml, la función *max3/1* se escribiría como:

```
1 # let max3 (a,b,c) = max(max(a,b), c) ;;
2 val max3 : int * int * int -> int = <fun>
```

- ▶ Ahora podemos olvidarnos de los detalles internos de *max3* y usar esta función como una caja negra para construir nuevas funciones.

```
1 # let signomax4(a,b,c,d) = signo(max(max3(a,b,c),d)) ;;
2 val signomax4 : int * int * int * int -> signos = <fun>
```



Universidad Veracruzana

Transparencia referencial

- ▶ La propiedad fundamental de las funciones matemáticas que permite la analogía con los bloques de construcción se llama **transparencia referencial**: El valor de una expresión, depende exclusivamente del **valor de las sub-expresiones** que lo componen.
- ▶ Evitando así la presencia de **efectos colaterales** propios de lenguajes que presentan opacidad referencial.
- ▶ Una función con referencia transparente tiene como característica que dados los **mismos parámetros** para su aplicación, obtendremos siempre el **mismo resultado**.



Contra ejemplo

- ▶ Consideren la siguiente pseudo función en Pascal:

```
1  function F (x:integer) : integer;
2    begin
3      a := a+1;
4      F := x*x;
5    end
```

- ▶ No podríamos eliminar la sub expresión común en la expresión:

$$(a + 2 * F(b)) * (c + 2 * F(b))$$



Universidad Veracruzana

Funciones de orden superior

- ▶ Son funciones que toman como **argumentos otras funciones**, o bien, **regresan funciones** como su resultado.
- ▶ Las funciones tipo *mapcar*/2 en Lisp reciben funciones como argumentos funciones y datos:

```
1 CL-USER> (defun inc1 (x) (+ x 1))  
2 INC1  
3 CL-USER> (mapcar #'inc1 '(1 2 3 4 5))  
4 (2 3 4 5 6)
```



Universidad Veracruzana

Técnica de Curry

- ▶ En la **técnica de Curry** una función es aplicada a sus argumentos, uno a la vez.
- ▶ Cada aplicación regresa una función de orden superior que acepta el siguiente argumento.
- ▶ **Ejemplo:**

```
1 # let suma x y = x + y;;
2 val suma : int -> int -> int = <fun>
3 # let suma = function x -> function y -> x+y ;;
4 val suma : int -> int -> int = <fun>
5 # suma 3 4 ;;
6 - : int = 7
7 # suma 3 ;;
8 - : int -> int = <fun>
```



Ejemplo: listas, sumatoria, producto y patrones

- ▶ Recuerden que una lista es un **tipo de dato recursivo**, la lista es una lista vacía (*nil*) o algo pegado (*cons*) a una lista.
- ▶ La *sumatoria* de los elementos de una lista:

$$\text{sumatoria } \textit{nil} = 0$$

$$\text{sumatoria } (\textit{cons } \textit{num } \textit{lst}) = \textit{num} + \text{sumatoria } \textit{lst}$$

- ▶ Y el *producto* de los elementos de una lista:

$$\text{producto } \textit{nil} = 1$$

$$\text{producto } (\textit{cons } \textit{num } \textit{lst}) = \textit{num} * \text{producto } \textit{lst}$$



reduce/3 un patrón general

- ▶ un patrón recursivo recurrente general, conocido como *reduce*:

sumatoria = *reduce suma 0*

producto = *reduce mult 1*

donde por conveniencia usamos las funciones prefijas *suma/2* y *mult/2*:

$$\text{suma } x \text{ } y = x + y$$

- ▶ La definición de *reduce/3* es la siguiente:

$$\text{reduce } f \text{ } x \text{ } \text{nil} = x$$

$$\text{reduce } f \text{ } x \text{ } (\text{cons } a \text{ } l) = (f \text{ } a)(\text{reduce } f \text{ } x \text{ } l)$$



Sumatoria y Producto en OCaml

- ▶ Veamos la definición de *reduce/3* y *sumatoria/1*:

```
1  # let rec reduce f x l = match l with
2      [] -> x
3      | h::t -> f h (reduce f x t) ;;
4  val reduce: ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
5
6  # let sumatoria = reduce suma 0 ;;
7  val sumatoria : int list -> int = <fun>
8  # sumatoria [1;2;3;4] ;;
9  - : int = 10
```



¿Y el producto?

- ▶ Y ahora lo interesante:

```
1  # let mult x y = x * y;;
2  val mult : int -> int -> int = <fun>
3  # let producto = reduce mult 1;;
4  val producto : int list -> int = <fun>
5  # producto [1;2;3;4];;
6  - : int = 24
```



Más patrones: *Construyendo y pegando listas*

- ▶ Definimos el operador *cons*:

```
1 # let cons x y = x :: y;;
2 val cons : 'a -> 'a list -> 'a list = <fun>
3 # cons 1 []
4 - : int list = [1]
5 # cons 1 (cons 2 [])
6 - : int list = [1; 2]
```

- ▶ Ahora definimos *append/2* usando *reduce*:

```
1 # let append lst1 lst2 = reduce cons lst2 lst1;;
2 val append : 'a list -> 'a list -> 'a list = <fun>
3 # append [1;2] [3;4] []
4 - : int list = [1; 2; 3; 4]
```



Composición funcional y mapeos en Caml

- ▶ Las funciones de orden superior nos permiten definir fácilmente la **composición funcional** *o*; y el **mapeo** sobre listas *map*:

```
1  # let o f g h = f (g h);;
2  val o : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
3  # let map f = reduce (o cons f) [];;
4  val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- ▶ Ahora podemos incrementar en 1 una lista de números, como en Lisp:

```
1  # map (fun x -> x+1) [1;2;3;4] ;;
2  - : int list = [2; 3; 4; 5]
```



Sumatoria de matrices

- ▶ O bien, podemos extender nuestro concepto de sumatoria para que trabaje sobre matrices representadas como listas de listas:

```
1 # let sumatoria_matriz = o sumatoria (map sumatoria);;
2 val sumatoria_matriz : int list list -> int = <fun>
3 # sumatoria_matriz [[1;2];[3;4]];;
4 - : int = 10
```



Universidad Veracruzana

Recursividad y Programación Funcional

- ▶ Las funciones recursivas, como hemos visto, se definen en términos de **ellas mismas**, permitiendo de esta forma que una operación se repita una y otra vez.
- ▶ La recursión **a la cola** permite optimizar la implementación de estas funciones.
- ▶ La razón por la cual las funciones recursivas son naturales en los lenguajes funcionales, es porque normalmente en ellos operamos con **datos recursivos**.



Listas y Arboles

- ▶ Aunque las listas están definidas en estos lenguajes, observen las siguientes definiciones de tipo “lista” y “arbolbin” en Ocaml:

```
1  # type 'a lista =
2      Nil
3      | Cons of 'a * 'a lista ;;
4  type 'a lista = Nil | Cons of 'a * 'a lista
5
6  # type 'a arbolbin =
7      Hoja
8
9      | N 'a * 'a arbolbin * 'a arbolbin ;;
10 type 'a
11 arbolbin = Hoja | Nodo of 'a * 'a arbolbin * 'a arbolbin
```



Membresía I

- Definiremos *miembro/2* para *listas*:

```
1 # let rec miembro elt lst = match lst with
2   | Nil -> false
3   | Cons(e,l) -> if e=elt then true else miembro elt l;;
4 val miembro : 'a -> 'a lista -> bool = <fun>
5
6 # miembro 2 (Cons(1,Nil)) ;;
7 - : bool = false
8 # miembro 1 (Cons(1,Nil)) ;;
9 - : bool = true
```



Membresía II

► Y para árboles binarios:

```
1  # let rec miembro elt arbol = match arbol with
2      Hoja -> false
3      | Nodo(e,izq,der) -> if e=elt then true
4                      else miembro elt izq ||
5                          miembro elt der ;;
6  val miembro : 'a -> 'a arbolbin -> bool = <fun>
7
8  # miembro 2 (Nodo(1,Nodo(2, Hoja, Hoja),Nodo(3, Hoja, Hoja)));;
9  - : bool = true
```



Listas y Arboles en Lisp

- ▶ También tienen estructura recursiva, aunque no es necesario definir un tipo de datos para ello.

```
1 (defun enlistap (elm lst)
2   (cond ((null lst) nil)
3         ((eq elm (car lst)) t)
4         (t (miembrop elm (cdr lst))))))
5
6 (defun enarblop (elm arb)
7   (cond ((null arb) nil)
8         ((eq elm (car arb)) t)
9         (t (or (enarblop elm (cadr arb))
10            (enarblop elm (caddr arb))))))
```



Ejemplos de corridas

```
1 CL-USER> (enlistap 1 '())
2 NIL
3 CL-USER> (enlistap 1 '(1 2 3 4 5))
4 T
5 CL-USER> (enlistap 1 '(0 2 1 3 4))
6 T
7 CL-USER> (enlistap 1 '(5 4 3 2 1))
8 T
9 CL-USER> (enarbolp 1 '(1 nil nil))
10 T
11 CL-USER> (enarbolp 2 '(1 (2 nil nil) (3 nil nil)))
12 T
13 CL-USER> (enarbolp 3 '(1 (2 nil nil) (3 nil nil)))
14 T
15 CL-USER> (enarbolp 4 '(1 (2 nil nil) (3 nil nil)))
16 NIL
```



Consideraciones finales

- ▶ Hemos presentado las bondades de la programación funcional para responder a la pregunta de porqué es **relevante** estudiar este paradigma de programación.
- ▶ Una postura similar para responder a esta pregunta puede encontrarse en el artículo de Hugues [10] *Why Functional Programming matters*.



Referencias I

- [1] J Backus. "Can programming be liberated from the von Neumann style? a functional style and its algebra of programs a functional style and its algebra of programs". En: *ACM Turing award lectures* (1977), págs. 613-641. URL:
http://portal.acm.org/ft_gateway.cfm?id=1283933&type=pdf&coll=GUIDE&dl=GUIDE&CFID=72367179&CFTOKEN=75548516.
- [2] I Bratko. *Prolog programming for Artificial Intelligence*. 3rd. Addison-Wesley, 2001.
- [3] E Chailloux, P Manoury y B Pagano. *Developing Applications With Objective Caml*. Paris, France: O'Reilly, 2000.
- [4] A Church. "A Note on the Entscheidungsproblem". En: *Journal of Symbolic Logic* 1 (1936), págs. 40-41.
- [5] A Church. "An unsolvable problem of elementary number theory". En: *American journal of mathematics* 58.2 (1936), págs. 345-363.
- [6] A Colmerauer y P Roussel. "The birth of Prolog". En: *History of Programming Languages*. Ed. por TH Bergin y RG Gibson. ACM Press / Addison-Wesley, 1996. Cap. The birth of Prolog, págs. 331-367.
- [7] A Colmerauer et al. *Un système de communication homme-machine en français*. Rapport préliminaire de fin de contrat IRIA. Marseille, France: Groupe Intelligence Artificielle Faculté des Sciences de Luminy, Université d'Aix-Marseille II, 1972.



Referencias II

- [8] HL Dreyfus y SE Dreyfus. "Making a mind versus modelling the brain: artificial intelligence back at a branchpoint". En: *Daedalus* 117.1 (1988), págs. 185-197.
- [9] RW Floyd. "The paradigms of programming". En: *Communications of the ACM* 22.8 (1979), págs. 455-460.
- [10] J Hughes. "Why Functional Programming Matters". En: *Computer Journal* 32.2 (1989), págs. 98-107.
- [11] RA Kowalski y D Kuehner. "Linear Resolution with Selection Function". En: *Artificial Intelligence* 2.3/4 (1971), págs. 227-260.
- [12] TS Kuhn. *The structure of scientific revolutions*. USA: University of Chicago Press, 1962.
- [13] J McCarthy. "Recursive functions of symbolic expressions and their computation by machine, Part I". En: *Commun. ACM* 3.4 (1960), págs. 184-195. ISSN: 0001-0782.
- [14] WS McCulloch y W Pitts. "A logical calculus of the ideas immanent in nervous activity". En: *The bulletin of mathematical biophysics* 5.4 (1943), págs. 115-133.
- [15] A Newell y HA Simon. "Computer Science As Empirical Inquiry: Symbols and Search". En: *Commun. ACM* 19.3 (1976), págs. 113-126. ISSN: 0001-0782. URL: <http://doi.acm.org/10.1145/360018.360022>.



Referencias III

- [16] JA Robinson. "A Machine-Oriented Logic based on the Resolution Principle". En: *Journal of the ACM* 12.1 (1965), págs. 23-41.
- [17] DE Rumelhart, GE Hinton, JL McClelland et al. "Parallel distributed processing: Explorations in the microstructure of cognition". En: vol. 1. Cambridge, MA, USA: The MIT Press, 1986. Cap. A general framework for parallel distributed processing, págs. 45-76.
- [18] SJ Russell y P Norvig. *Artificial Intelligence: A Modern Approach*. Third. Prentice Hall Series in Artificial Intelligence. USA: Prentice Hall, 2009.
- [19] JR Searle. "Minds, brains, and programs". En: *Behavioral and brain sciences* 3.3 (1980), págs. 417-424.
- [20] P Seibel. *Practical Common Lisp*. New York, NY, USA: Apress, 2005.
- [21] HT Siegelman y ED Sontag. "Turing Computability with Neural Nets". En: *Applied Mathematical Letters* 4.6 (1991), págs. 77-80.
- [22] HA Simon. *The Sciences of the Artificial*. Cambridge, MA, USA: The MIT Press, 1969.
- [23] AM Turing. "On the Computable Numbers, with Applications to the Entscheidungsproblem". En: *Proceedings of the London Mathematical Society*. Vol. series 2. 1936, págs. 230-265.



Referencias IV

- [24] AM Turing. "Computing machinery and intelligence". En: *Mind* 59.236 (1950), págs. 433-460.
- [25] DHD Warren. *An abstract Prolog instruction set*. Inf. téc. 309. SRI, 1983.



Universidad Veracruzana