1 INTRODUCCIÓN

Este curso es acerca de los Paradigmas de Programación propuestos por la Inteligencia Artificial (IA). Aquí, el término **paradigma** hace referencia a la definición de Kuhn [50]: 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. La adopción de un paradigma hace posible compartir conceptos básicos, procedimientos, etc. Por otra parte, las revoluciones científicas han emergido de cambios en los paradigmas dominantes.

Paradigma

A continuación definiremos el concepto de Paradigma de Programación, influenciados por Kuhn, de forma que nos sea posible establecer los diferentes paradigmas propuestos al seno de la IA. También, justificaremos la decisión de abordar los paradigmas lógico y funcional, derivados de la llamada IA simbólica.

Con la idea de familiarizar al lector con los paradigmas propuestos, se presentan dos secciones como un primer acercamiento, informal, a la programación lógica y funcional. Posteriormente, se presenta la organización de estas notas de curso y el capítulo cierra con las lecturas y ejercicios sugeridos.

1.1 PARADIGMAS DE PROGRAMACIÓN

El término **Paradigma de Programación** fue adoptado por Floyd [28], al recibir el *ACM Turing Award* en 1978. Su idea es que varias de las observaciones de Kuhn sobre los paradigmas científicos, son pertinentes en el contexto particular de las Ciencias de la Computación, por ejemplo:

Paradigma de Programación

- 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. Esto suele ser el caso en los libros de computación que reducen esta ciencia a los algoritmos y lenguajes 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. En las Ciencias de la Computación es posible observar a diversas comunidades, hablando sus propios lenguajes y usando sus propios paradigmas. De hecho, los lenguajes de programación estimulan el uso de algunos paradigmas, mientras que inhiben notoriamente otros.

En opinión de Floyd, las dificultades típicas presentes en el desarrollo de programas de cómputo al considerar aspectos como confiabilidad, puntualidad, flexibilidad, eficiencia y costo; son reflejo de un **repertorio** inadecuado de paradigmas de programación, un conocimiento deficiente de los paradigmas existentes, la manera en que enseñamos esos paradigmas y en la manera en que los lenguajes de programación soportan, o no, los paradigmas de sus comunidades de usuarios. Y concluye –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.

Repertorio de paradigmas

Lo anterior constituye la **justificación** del diseño curricular de nuestro curso, cuya idea central es expandir nuestro repertorio de Paradigmas de Programación para alcanzar el objetivo del mismo: El estudiante será capaz de resolver problemas complejos que requieran la adopción de los paradigmas lógico y/o funcional.

Justificación

Ahora bien, para introducir los Paradigma de Programación de la IA será necesario revisar brevemente el concepto mismo de la IA y hacer una breve revisión histórica de sus orígenes y la adopción de la computadora como nuestro artefacto de interés. Primero, la IA tiene como objeto de estudio a las entidades inteligentes y su comportamiento; pero a diferencia de la filosofía, la psicología, las neurociencias, y demás disciplinas que comparten este objeto de estudio, su meta no tiene que ver únicamente con la comprensión de tales entidades, sino con su construcción. La **construcción de agentes racionales** constituye la idea central del curiosamente llamado enfoque moderno de la IA[1], propuesto por Russell y Norvig [78].

La IA como ciencia e ingeniería

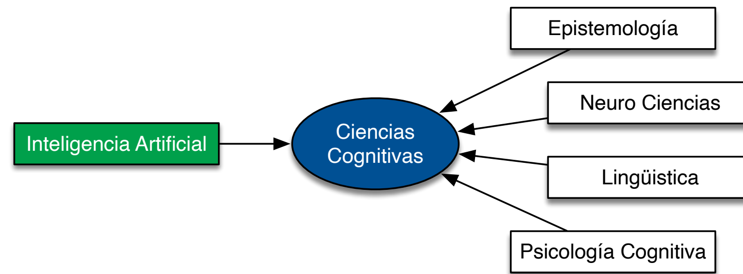


Figura 1.1: La IA como parte de las Ciencias Cognitivas. Resalta su interés en la síntesis de entidades inteligentes y no solo en su característico análisis presente en otras disciplinas. Adaptado de Varela [93].

De hecho, como se muestra en la Figura 1.1, Varela [93] ubica a la IA como parte de las Ciencias Cognitivas resaltando esta asimetría: su interés, único en el área, por la síntesis de entidades inteligentes. La IA es, como bien lo define Simon [84], una ciencia de lo artificial.

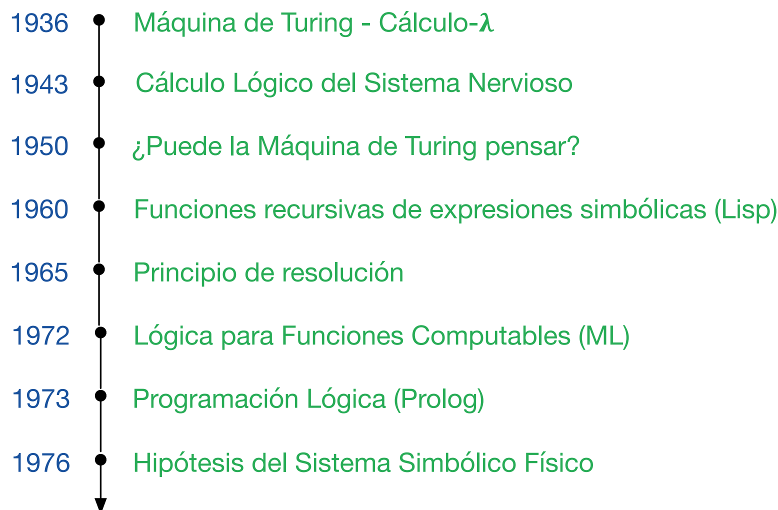


Figura 1.2: Una línea temporal sobre los eventos que han definido los Paradigmas de Programación propuestos por la IA.

Segundo, hagamos un poco de historia. La argumentación que sigue, se basa en una serie de eventos que se muestran en la Figura 1.2, iniciando en 1936. La feliz propuesta de considerar si las **máquinas** pueden pensar, se la debemos a Turing [92], en la formulación de la famosa prueba que lleva su nombre. Independientemente de las objeciones filosóficas a tomar en cuenta, por ejemplo, a Searle [80]

Máquinas inteligentes

1 AIMA (*Artificial Intelligence: A Modern Approach*) es el libro de texto más usado en el mundo sobre IA. Se tiene registro de su uso en más de 1500 universidades de 135 países. La página web del libro se encuentra en: `http://aima.cs.berkeley.edu/index.html`

y Dreyfus y Dreyfus [26], 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**, entendida ésta tal y como la concibe Turing [91] al definir su máquina: Como una manipulación formal de símbolos, no interpretados, mediante la aplicación de reglas formales. Ese mismo año, Church [15] introduce el Cálculo-λ con el objetivo de definir el concepto de funciones efectivamente calculables, donde efectivamente se refiere a que el cálculo puede llevarse a cabo por medios mecánicos (Ver Ejemplo 1.1). Pues bien, las tesis de Church y Turing son equivalentes, lo cual tendrá implicaciones en nuestros paradigmas de programación.

IA y computación

Ejemplo 1.1 (Cómputo efectivo)**.** *El uso de las tablas de verdad como una prueba de que una fórmula de la lógica proposicional es una tautología, es un método efectivamente computable en el sentido descrito anteriormente; Aunque irrealizable en la práctica, éste método es en principio aplicable a cualquier fórmula, independientemente del número de variables que involucre.*

Siguiendo el curso histórico de la IA, McCulloch y Pitts [57] proveen las bases del llamado **Paradigma Conexionista**, al proponer un cómputo inspirado en nuestro sistema nervioso, particularmente en una abstracción de las redes neuronales, demostrando que:

Paradigma Conexionista

- 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.

Cabe resaltar que, aunque el referente principal de este trabajo es neuro-fisiológico, no abandona el referente al cómputo simbólico originalmente propuesto por Turing. De hecho, establece una equivalencia con lo computable por la máquina de Turing y, por extensión, con el Cálculo-λ. Tal aproximación a la IA, resulta en un paradigma de computación conocido como **Procesamiento Paralelo Distribuido** (PDP) [77], que está caracterizado por los siguientes aspectos:

Procesamiento Paralelo Distribuido

- Un conjunto de unidades de procesamiento representadas como un conjunto de enteros.
- Una activación para cada unidad, representada como un vector de funciones dependientes del tiempo.
- Un patrón de conectividad entre las unidades, representada por una matriz de números reales que denotan la fuerza de las conexiones.
- Una regla de propagación para la activación de las unidades, representada como una función de salida.
- Una regla de activación para combinar las entradas de cada unidad y computar así su activación, representada por una función sobre el nivel actual de activación y propagación.
- Una regla de aprendizaje para modificar las conexiones con base en la experiencia, representada como una función de cambio en los pesos de las conexiones.
- Un ambiente que provee al sistema con experiencias, representado por un conjunto de vectores de activación para un subconjunto de unidades de procesamiento (capa de entrada).

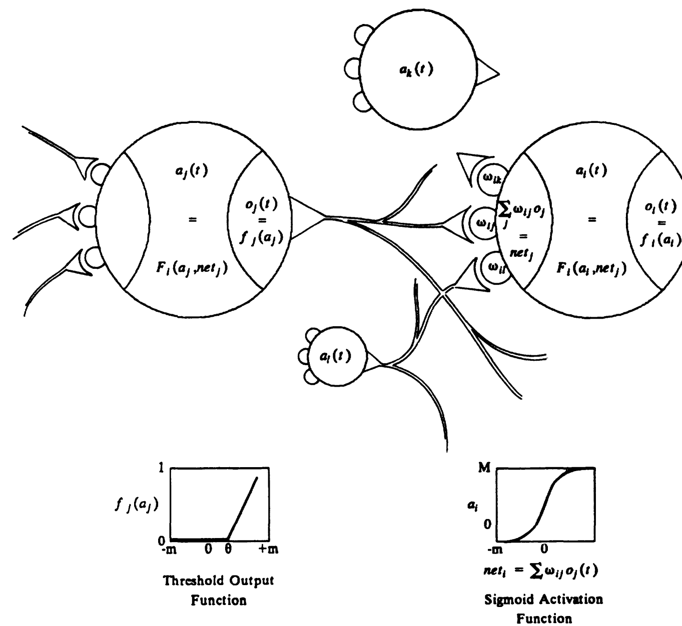


Figura 1.3: El modelo del Procesamiento Paralelo Distribuido. Tomada del libro de Rumelhart, Hinton, McClelland et al. [77]

La Figura 1.3 ilustra estos conceptos. Las unidades de procesamiento son los círculos en el diagrama. En todo momento t la unidad de procesamiento u_i tiene un valor de activación denotado por $a_i(t)$. Este valor de activación es pasado por una función f_i que produce un valor de salida $o_i(t)$. Este valor de salida es enviado a otras unidades de procesamiento, a través de un conjunto de conexiones unidireccionales. Cada conexión tiene asociado un valor real, conocido como el peso o la fuerza de la conexión, denotado como w_{ij}, que determina la intensidad del efecto de la primera unidad sobre la segunda. Todas las entradas se combinan mediante un operador (normalmente la suma) para computar un nuevo valor de activación mediante a función F_i.

Ahora bien, no es evidente que el enfoque PDP haya producido paradigmas de programación *ad hoc*. Los paradigmas imperativo, orientado a objetos y principalmente el de cómputo científico[2], parecieran cubrir las necesidades de esta comunidad, y aunque se han diseñado lenguajes de programación que sin duda facilitan la implementación de procesos paralelos distribuidos, por ejemplo, Erlang [1], difícilmente pueden considerarse un Paradigma de Programación orientado al PDP, en el sentido aquí definido.

Paradigma Simbólico

Sistema simbólico físico

El **Paradigma Simbólico** de la IA es descrito a la perfección por Newell y Simon [61], en la recepción de su *ACM Turing Award* en 1975, donde introducen el concepto de **sistema simbólico físico**, caracterizado como:

- Un conjunto de entidades, llamadas símbolos, que son patrones físicos que pueden ocurrir como componentes de otro tipo de entidades, llamadas expresiones (o estructuras simbólicas).
- Toda estructura simbólica está compuesta por ocurrencias de símbolos relacionados de alguna forma física, p. ej., un símbolo es el siguiente con respecto a otro.
- En cualquier momento, el sistema contiene una colección de expresiones.
- Una colección de procesos que operan sobre expresiones para producir nuevas expresiones: procesos de creación, modificación, reproducción y destrucción.

2 Es de esperar que estos paradigmas sean abordados en el curso optativo de Redes Neuronales.

- Tal sistema existe en un mundo de objetos más extenso que sus expresiones simbólicas.

Dos nociones son centrales en tales sistemas de expresiones, símbolos y objetos:

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 ejecutarlo.

Algunos requerimientos adicionales a los sistemas simbólicos físicos incluyen: Un símbolo debe poder ser usado para designar cualquier expresión, sin prescripciones *a priori* de qué 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 esencialmente ilimitado.

A partir de esta definición, Newell y Simon plantean la **hipótesis** central de la IA simbólica:

Hipótesis del Sistema Simbólico Físico

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

Por necesario, entendemos que cualquier sistema que exhiba inteligencia general, podrá verificarse mediante análisis, es un sistema simbólico físico. Por suficiente, entendemos que cualquier sistema simbólico físico del tamaño adecuado, puede ser organizado para exhibir inteligencia general. Puesto que todo sistema simbólico físico es una máquina universal, esto implica que la inteligencia es implementable en una computadora universal.

Dos **paradigmas** de programación se derivan de esta aproximación a la IA. Estos serán nuestro tema de estudio a lo largo del curso:

Paradigmas de la IA

PROGRAMACIÓN LÓGICA. Basado en la idea de que la deducción lógica puede constituirse en una forma de computación universal. Su lenguaje más representativo es Prolog [20], basado en el principio de resolución-SL de Kowalski y Kuehner [49], una regla de inferencia que permite hacer demostrar que una cláusula lógica es consecuencia de un conjunto de ellas (el programa) y computar los valores de las variables en la cláusula, que hacen esto posible. Otros lenguajes lógicos incluyen a Datalog [52], una eficiente y completa versión restringida de Prolog, sin funciones ni predicados extra lógicos; Mercury [86] con tipos de datos explícitos; y el orientado a objetos Logtalk [60].

PROGRAMACIÓN FUNCIONAL. Basado en la idea de la computación en términos de funciones matemáticas y su transformación a través de formas funcionales (funciones de orden superior). Su concepción original se debe a Backus [2], aunque tiene raíces en el Cálculo-λ de Church [15]. El primer lenguaje con características funcionales fue Lisp de McCarthy [54] que es de nuestro interés porque además es una implementación de los sistemas simbólicos físicos. Otros lenguajes más próximos a la idea original de Backus incluyen ML [33], Ocaml [74] y Haskell [89].

Dado que ambos paradigmas son bien distintos a los paradigmas más conocidos en la computación, p. ej., orientado a objetos o imperativo, comenzaremos por visión panorámica del tipo de cómputo que proponen.

1.2 PROGRAMACIÓN LÓGICA

La historia reciente, y *à la française*, de la programación lógica comienza en julio de 1970 en Montreal, Canadá, donde Colmerauer y Roussel [19] trabajaban en un proyecto sobre traducción automática y procesamiento del lenguaje natural. El sistema en cuestión incluía analizadores sintácticos y generadores de frases para el francés. Un estudiante de Colmerauer, decidió trabajar sobre la demostración automática de teoremas, con base en el trabajo sobre el principio de resolución de Robinson [75]. La conjunción de estos trabajos dio como resultado una interfaz entre el francés y las fórmulas lógicas del demostrador de teoremas que permitía interacciones como las que se muestran en el Cuadro 1.1.

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

Cuadro 1.1: Sistema de lenguaje natural de Colmerauer y Roussel [19].

Este sistema hacía uso de constantes para designar elementos del universo de discurso (Tom, Jerry, Max, Queso); para designar conjuntos de éstos (Gatos, Ratones, Ratones que comen queso, etc.); y relaciones binarias entre ellos (Matar, Comer, Gustar, etc.). Las constantes, junto con los símbolos funcionales *The*, *Subset*, y *True*, especificaban un lenguaje de fórmulas lógicas. Mientras se seguía trabajando en la demostración de teoremas en este lenguaje, apareció la referencia obligada al trabajo de Kowalski y Kuehner [49] sobre el método conocido como **resolución-SL**, que como veremos, es fundamental en el lenguaje Prolog.

Resolución-SL

Prolog es la realización más utilizada del paradigma de programación lógica. Escribir un programa en Prolog tiene menos que ver con la tarea de especificar un algoritmo, como es el caso de la programación imperativa; y más con la especificación de los objetos y las relaciones que ocurren entre ellos, en el contexto de un problema. En particular, tiene que ver con la especificación de las relaciones que conforman la solución deseada del problema. Veamos un ejemplo basado en la genealogía de una familia [8].

1.2.1 Hechos y relaciones

La Figura 1.4 muestra a los miembros de una familia y una relación entre ellos: las flechas $X \rightarrow Y$ indican que X es progenitor de Y. El **hecho** de que Tom[3] sea progenitor de Bob se escribe en Prolog: `progenitor(tom,bob).`

Hecho

Por razones que explicaremos más adelante, escribimos los nombres como `tom` con minúscula inicial. Se dice que `tom` y `bob` son los argumentos de la relación *progenitor*. La expresión *progenitor*/2 denota que esta relación tiene dos argumentos, decimos que *progenitor* es una relación de **aridad** 2. El árbol familiar completo puede definirse como un programa en Prolog:

Aridad

3 Decidí usar una familia gringa, porque nuestros bellos nombres como Asunción María del Pilar, no caben en un grafo fácil de leer. Si usted quiere llamar a Tom, Pancho; eso, como veremos, no cambia en nada la historia que voy a contar (a condición de que Pancho sea siempre Pancho).

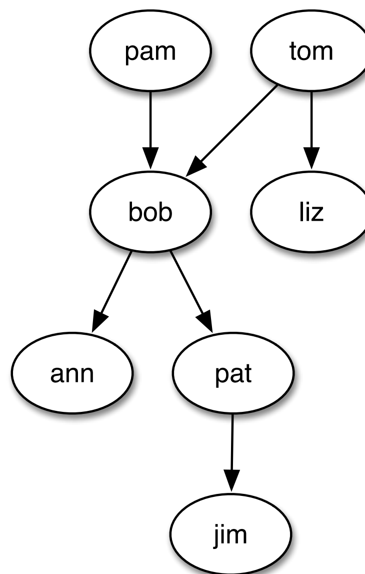


Figura 1.4: Los miembros de una familia y su relación progenitor.

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

Cláusula

Este programa consta de seis **cláusulas**. Cada cláusula declara un hecho sobre la relación *progenitor*. Por ejemplo, `progenitor(tom,bob)` es un caso particular de la relación *progenitor*. Una relación está definida por el conjunto de todos sus casos.

Podemos editar un archivo con este programa Prolog y llamarlo `familia.pl`. Para utilizar este programa es necesario invocar a Prolog, por ejemplo, si usamos SWI Prolog, en una terminal invocaríamos `swipl` (ó `pl` en algunos sistemas operativos):

```
1 > swipl
2 Welcome to SWI-Prolog (threaded, 64 bits, version 7.5.10)
3 SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
4 Please run ?- license. for legal details.
5
6 For online help and background, visit http://www.swi-prolog.org
7 For built-in help, use ?- help(Topic). or ?- apropos(Word).
8
9 ?-
```

El símbolo `?-` es el indicador de que Prolog espera una instrucción. Si tenemos un archivo llamado `clase01.pl` con el conjunto de casos que define la relación *progenitor*, podemos consultarla desde SWI Prolog:

```
1 ?- [familia].
2 true.
3 ?-
```

Prolog responde que el programa `progenitor` ha sido compilado (¿Sabían ustedes que el código de Prolog es compilado?) y espera una nueva instrucción. La instrucción puede ser la pregunta ¿Es progenitor Bob de Pat?

```
1 ?- progenitor(bob,pat).
2 true.
```

a lo que Prolog responderá `true`, al encontrar que ese hecho se encuentra en nuestro programa. Si preguntamos ¿Es Liz progenitora de Pat? obtendremos como respuesta `false` [4], porque nuestro programa no menciona nada[5] acerca de que Liz sea progenitora de Pat:

```
?- progenitor(liz,pat).
false.
```

Lo mismo sucede con la siguiente consulta, pues `ben` no es siquiera un objeto conocido por nuestro programa:

```
?- progenitor(tom,ben).
false.
```

Una pregunta más interesante sobre la relación progenitor es ¿Quién es el progenitor de Liz? Lo cual puede preguntarse como:

```
?- progenitor(X,liz).
X = tom
```

Prolog computa un valor para X tal que la relación progenitor se cumple. Si preguntamos por los hijos de Bob, tendremos varias respuestas posibles. Para obtenerlas todas, es necesario teclear punto y después de cada respuesta de Prolog:

```
?- progenitor(bob,X).
X = ann ;
X = pat.
```

Prolog nos da las respuestas `ann`, `pat`.

Es posible plantear preguntas más complicadas a nuestro programa, por ejemplo ¿Quién es abuelo/a de Jim? Como nuestro programa no conoce directamente la relación *abuelo/2*, esta pregunta debe descomponerse en dos preguntas como lo muestra la Figura 1.5:

1. ¿Quién es el progenitor de Jim? Asumamos que es alguién Y.
2. ¿Quién es el progenitor de Y? Asumamos que es alguién X.

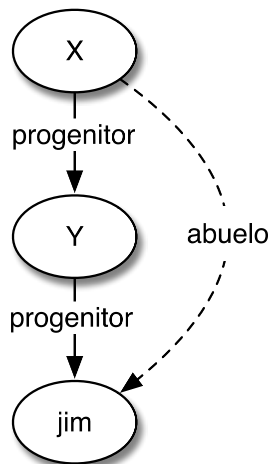


Figura 1.5: La relación abuelo expresada como una composición de dos relaciones progenitor.

La secuencia de preguntas en Prolog es como sigue:

4 Algunas implementaciones de Prolog, responden con `Yes` y `No` a estas preguntas
5 ¿Habían escuchado el termino "supuesto del mundo cerrado"?

```
?- progenitor(Y,jim), progenitor(X,Y).
Y = pat,
X = bob
```

Si invertimos el orden de las dos preguntas, el resultado sigue siendo el mismo:

```
?- progenitor(X,Y), progenitor(Y,jim).
X = bob,
Y = pat
```

Podemos preguntar también ¿Quien es nieto de Tom?:

```
?- progenitor(tom,X), progenitor(X,Y).
X = bob,
Y = ann ;
X = bob,
Y = pat ;
false.
```

Otra pregunta interesante sería ¿Tienen Ann y Pat progenitores en común? Esto puede descomponerse nuevamente en dos preguntas:

1. ¿Quién es el progenitor X de Ann?
2. ¿Es X (el mismo) progenitor de Pat?

```
?- progenitor(X,ann), progenitor(X,pat).
X = bob
```

Resumiendo:

- Es sencillo definir en Prolog una relación, como *progenitor/2*, especificando las n-tuplas de objetos que satisfacen la relación (n, conocido como aridad, es el número de argumentos de la relación, para progenitor $n = 2$). A este tipo de definiciones se les conoce como **extensionales**.
- El usuario puede plantear fácilmente preguntas a Prolog sobre las relaciones definidas en un programa.
- Un programa Prolog consiste de cláusulas. Cada cláusula termina con un punto.
- 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 como `progenitor(X,ann), progenitor(X,pat)` significa la conjunción de las metas: X es progenitor de *ann* y X es progenitor de *pat*.
- La respuesta a una pregunta puede ser positiva o negativa, dependiendo de si la meta se puede satisfacer o no. En el caso de una respuesta positiva, se dice que la meta fue satisfecha y tuvo éxito. En cualquier otro caso se dice que la meta no fue satisfecha y falló.
- Si varias respuestas satisfacen una pregunta, Prolog encontrará tantas como el usuario quiera.

1.2.2 Reglas

Nuestro ejemplo puede extenderse en muchas formas interesantes. Definamos las relaciones *mujer*/1 y *hombre*/1, para poder expresarnos sobre el género de los miembros de nuestra familia ejemplar:

```
1 mujer(pam).
2 mujer(liz).
3 mujer(pat).
4 mujer(ann).
5 hombre(tom).
6 hombre(bob).
7 hombre(jim).
```

Las relaciones unarias ($n = 1$) se usan normalmente para expresar propiedades de los objetos. Las relaciones binarias ($n = 2$) definen relaciones propiamente dichas entre pares de objetos. La cláusula `mujer(pam)` establece que Pam es una mujer. La misma información podría definirse como una relación *genero/2* como `genero(pam,mujer)`.

Nuestra siguiente extensión al programa será definir la relación *vastago*/2 como la inversa de la relación *progenitor*/2. Para ello podemos definir explícitamente las tuplas que satisfacen esta relación, por ejemplo: `vastago(liz,tom)`, etc. Sin embargo, se puede obtener una definición más elegante si tomamos en cuenta que la relación *vastago/2* es la inversa de *progenitor*/2 y que *progenitor*/2 ya fue definida. La alternativa se basa en el siguiente enunciado lógico: Para todo X y para todo Y, Y es un vástago de X si existe un X que es progenitor de un Y. Esta formulación es muy parecida al formalismo usado en Prolog. La cláusula correspondiente es la siguiente:

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

La cláusula puede leerse también como: Si X es un progenitor de Y entonces Y es un vástago de X. A este tipo de cláusulas se les conoce como **reglas**. Existe una diferencia fundamental entre los hechos y las reglas. Un hecho es algo que es siempre, incondicionalmente, verdadero. Las reglas especifican cosas que son ciertas si alguna condición se satisface. Por ello decimos que las reglas tienen:

Regla

- Una parte condicional (el lado derecho de la regla o **cuerpo** de la regla). *Cuerpo*
- Una conclusión (el lado izquierdo de la regla o **cabeza** de la regla). *Cabeza*

¿Qué hace Prolog cuando se le plantea una meta como la siguiente?

```
1 ?- vastago(liz,tom).
```

No existe ningún hecho sobre vástagos en nuestro programa, por lo tanto, la única alternativa es considerar la aplicación de la regla sobre los vástagos. La regla es general, en el sentido que es aplicable a cualquier objeto `X` e `Y`, por lo que puede ser aplicada a constantes como `liz` y `tom`. Para aplicar la regla a `liz` y a `tom` es necesario substituir `Y` por `liz` y `X` por `tom`. Con tal **substitución**, obtenemos un caso especial de nuestra regla:

Substitución

```
1 vastago(liz,tom) :- progenitor(tom,liz).
```

La parte condicional de la regla es ahora:

```
1 progenitor(tom,liz).
```

Ahora Prolog tratará de encontrar si esta condición es verdadera, de forma que la meta inicial:

```
vastago(liz,tom).
```

ha sido substituida por una sub-meta `progenitor(tom,liz)`. Esta nueva sub-meta puede satisfacerse fácilmente a partir de los hechos conocidos por el programa, lo cual significa que la conclusión de la regla también es verdadera, y Prolog responde con éxito:

```
?- vastago(liz,tom).
true.
```

Especifiquemos ahora la relación *madre*/2 a partir del siguiente enunciado lógico: Para toda X e Y, X es madre de Y si X es progenitor de Y y X es mujer. Esto se traduce a Prolog como:

```
madre(X,Y) :- progenitor(X,Y), mujer(X).
```

Conjunción

La coma en el cuerpo de la regla, indica una **conjunción**: ambas condiciones deben ser verdaderas para que la conclusión lo sea.

Las relaciones *abuela*/2 y *hermana*/2 pueden definirse como:

```
abuela(X,Y) :-
  progenitor(X,Z),
  progenitor(Z,Y),
  mujer(X).

hermana(X,Y) :-
  progenitor(Z,X),
  progenitor(Z,Y),
  mujer(X).
```

Observen, en el caso de *hermana*/2, la manera de especificar que `X` e `Y` tienen un mismo progenitor. La condición de esta regla se lee: existe un Z que es progenitor de X y el mismo Z es progenitor de Y y X es mujer. Gráficamente la relación *hermana*/2 se muestra en la figura 1.6. Ahora podemos preguntar:

```
?- hermana(ann,pat).
true
```

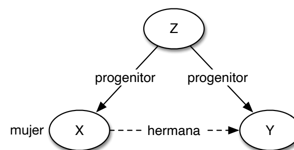


Figura 1.6: *La relación hermana.*

Tras nuestra primer pregunta sobre esta relación, podemos concluir que su definición es correcta, pero tiene un sutil error que se revela al preguntar:

```
?- hermana(X,pat).
X = ann ;
X = pat ;
false.
```

¿Es correcto que Pat sea su propia hermana? Ese es el comportamiento que esperábamos de la definición de *hermana*/2, y se debe a que no hay nada que diga que `X` e `Y` deben ser ¡diferentes! Esto se puede corregir definiendo *hermana* como:

```
hermana(X,Y) :-
  progenitor(Z,X),
  progenitor(Z,Y),
  mujer(X),
  dif(X,Y).
```

De forma que:

```
?- hermana(X,pat).
X = ann ;
false.
```

Resumiendo:

- 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 preguntas 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.
- 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. Por ejemplo la cláusula `tiene_hijo(X) :- progenitor(X,Y)` puede leerse como: Para todo X, X tiene un hijo si existe un Y y X es progenitor de Y.

1.2.3 Reglas recursivas

Agreguemos una relación nueva a nuestro programa: La relación *ancestro*/2. Esta relación será definida en términos de la relación *progenitor*/2. La definición completa puede expresarse por medio de dos reglas. La primera definiendo al ancestro inmediato (progenitor) y la segunda a los ancestros no inmediatos. Decimos que alguien X es ancestro indirecto de alguien Z, si hay una cadena de progenitores desde X hasta Z, como lo ilustra la figura 1.7. En nuestro ejemplo de la figura 1.4, Tom es ancestro directo de Liz e indirecto de Pat.

La primera regla es muy sencilla y se expresa en Prolog como:

```
ancestro(X,Z) :- progenitor(X,Z).
```

La segunda regla es más complicada porque las cadenas de progenitores presentan un problema: ¡No sabemos cuantas veces hay que aplicar la relación progenitor! Un primer intento podría ser algo como:

```
ancestro(X,Z) :-
   progenitor(X,Z).

ancestro(X,Z) :-
   progenitor(X,Y),
   progenitor(Y,Z).

ancestro(X,Z) :-
   progenitor(X,Y0),
   progenitor(Y0,Y1),
   progenitor(Y1,Z).
...
```

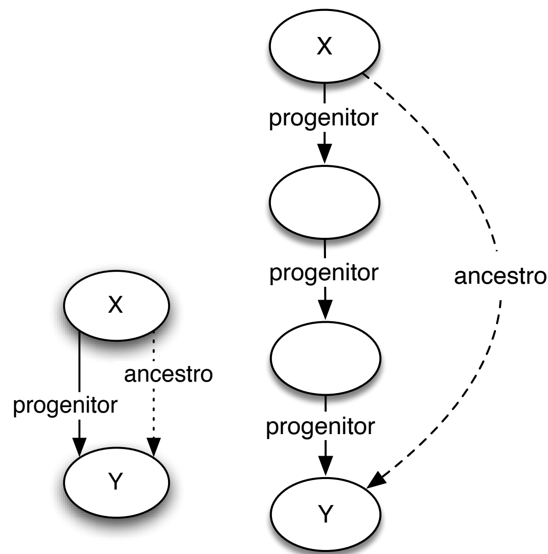


Figura 1.7: *La relación ancestro en términos de progenitor directo e indirecto.*

Lo cual resulta en un programa largo y, peor aún, que sólo funciona para un número limitado de ancestros, en el ejemplo: padres, abuelos y bisabuelos. Es decir, esta definición de *ancestro*/2 es correcta pero incompleta.

Existe una formulación elegante y completa de la relación *ancestro/2*, completa en el sentido que puede computar cualquier ancestro, independientemente de la longitud de la cadena de progenitores que deba aplicarse. La idea central es definir ancestro en términos de si misma:

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

Ahora podemos preguntar ¿De quien es ancestro Pam?

```
1  ?- ancestro(pam,X).
2  X = bob ;
3  X = ann ;
4  X = pat ;
5  X = jim ;
6  false.
```

O ¿Quienes son los ancestros de Jim?

```
1  ?- ancestro(X,jim).
2  X = pat ;
3  X = pam ;
4  X = tom ;
5  X = bob ;
6  No
```

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.

1.2.4 ¿Cómo computa Prolog una solución?

Una pregunta a Prolog es siempre una secuencia de una o más metas. Para responder, Prolog trata de satisfacer estas metas. ¿Qué significa **satisfacer** una meta? 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 encontrar cuales son los objetos particulares (que remplazaran a las variables) para los cuales la meta se satisface. La asignación de valores a variables es mostrada al usuario. Si Prolog no puede demostrar para alguna asignación de valores a variables, que las metas siguen lógicamente del programa, la respuesta a la pregunta será falso.

Satisfacción

En términos matemáticos, la interpretación de un programa en Prolog es como sigue: Prolog acepta hechos y reglas como un conjunto de axiomas, y el usuario plantea preguntas como un teorema; entonces Prolog trata de probar este teorema, es decir, demostrar que el teorema se sigue lógicamente de los axiomas.

Veamos un ejemplo clásico. Sean los axiomas:

- Todos los hombres son falibles.
- Socrates es un hombre.

Un teorema que lógicamente sigue de estos dos axiomas es:

- Socrates es falible.

El primer axioma puede reescribirse como: Para toda X, si X es un hombre, entonces X es falible. El ejemplo puede entonces traducirse a Prolog como sigue:

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

y

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

Un ejemplo más complicado, tomado de la familia de la Figura 1.4, es la meta: `?- ancestro(tom,pat).` Sabemos que `progenitor(bob,pat)` es un hecho. Podemos derivar entonces que `ancestro(bob,pat)`. Observen que este hecho derivado no puede ser encontrado explícitamente en nuestro programa sobre la familia, pero puede derivarse a partir de los hechos y reglas en el programa. Un paso en la inferencia de este tipo, puede ser escrito como: `progenitor(bob,pat)` $\Rightarrow$ `ancestro(bob,pat)`.

El proceso completo de inferencia en dos pasos puede escribirse como:

$$\begin{aligned} progenitor(bob, pat) &\Rightarrow ancestro(bob, pat) \\ progenitor(tom, bob) \wedge ancestro(bob, pat) &\Rightarrow ancestro(tom, pat) \end{aligned}$$

A este tipo de secuencias se les conoce como secuencias de prueba ¿Cómo encuentra Prolog una secuencia de prueba?

Prolog encuentra la secuencia de prueba en orden inverso al que acabamos de presentar. En lugar de comenzar con los hechos simples especificados en el programa, Prolog comienza con las metas y, usando reglas, substituye la meta actual por sub-metas, hasta que estas llegan a resolverse por hechos simples. Dada la pregunta:

```
1 ?- ancestro(tom,pat).
```

Prolog tratará de satisfacer esta meta. Para ello, tratará de encontrar una cláusula en el programa, a partir de la cual la meta dada pueda seguirse lógicamente. Obviamente, las únicas reglas acerca de la relación *ancestro/2* son:

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

Unificación

Decimos que la cabeza de estas reglas **unifica** con la meta planteada. Las reglas representan formas alternativas en las que Prolog puede resolver la meta. Al hecho de que dado un punto de ejecución de un programa en Prolog, puedan seguir todos los posibles cursos de ejecución, se le conoce como **no determinismo**.

No determinismo

Prolog intentará resolver la pregunta con la primer cláusula que aparece en el programa (líneas 1 y 2). Puesto que la meta es `ancestro(tom,pat)`, las variables de la regla pueden ser substituidas conforme a `X/tom` y `Z/pat`. La meta original `ancestro(tom,pat)`, es entonces remplazada por la sub-meta `progenitor(tom,pat)`. El paso consistente en usar una regla para transformar una meta en una sub-meta, se muestra gráficamente en la figura 1.8.

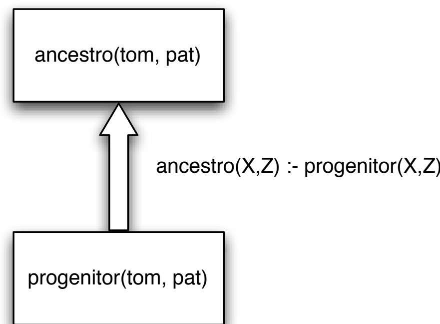


Figura 1.8: *El primer paso de la ejecución. La meta de arriba es verdadera si la meta de abajo es verdadera.*

Como no hay una cláusula en el programa que coincida con la nueva sub-meta `progenitor(tom,pat)`, la sub-meta falla. Ahora Prolog **vuelve atrás** (*backtrack*) para evaluar de forma alternativa su meta original. Ahora intentará la segunda cláusula del programa (líneas 4–6). Como antes, las variables de la meta toman los valores: `X/tom` y `Z/pat`. Pero `Y` no toma valor alguno aún. La meta es remplazada por las sub-metas: `progenitor(tom,Y), ancestro(Y,pat)`. La ejecución de este nuevo paso se muestra en la Figura 1.9.

Backtrack

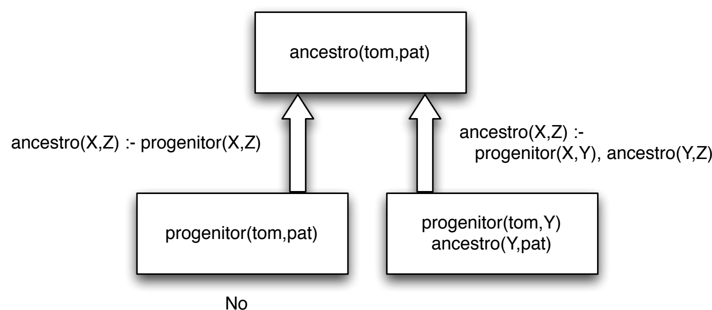


Figura 1.9: *El segundo paso de la ejecución. Dos sub-metas son generadas.*

Enfrentado ahora el problema de resolver dos sub-metas, Prolog intentará satisfacer la primer sub-meta definida en el programa (¿Porqué?). La primer sub-meta se resuelve fácilmente pues coincide con uno de los hechos del programa. Esto obliga a que `Y` tome el valor de `bob`, de forma que la segunda sub-meta se vuelve `ancestro(bob,pat)`.

Para satisfacer está sub-meta, Prolog usará nuevamente la primer cláusula del programa (líneas 1 y 2). Como en este paso se hace una nueva llamada a esta regla, en realidad Prolog utiliza variables diferentes a la llamada del paso anterior, renombrando las variables como sigue:

```
1 ancestro(X',Z') :- progenitor(X',Z').
```

Lo cual conduce a la substitución de variables: `X'/bob` y `Z'/pat`. La meta es remplazada por `progenitor(bob,pat)`. Esta meta es satisfecha porque coincide con uno de los hechos del programa. Gráficamente este proceso se muestra en la figura 1.10.

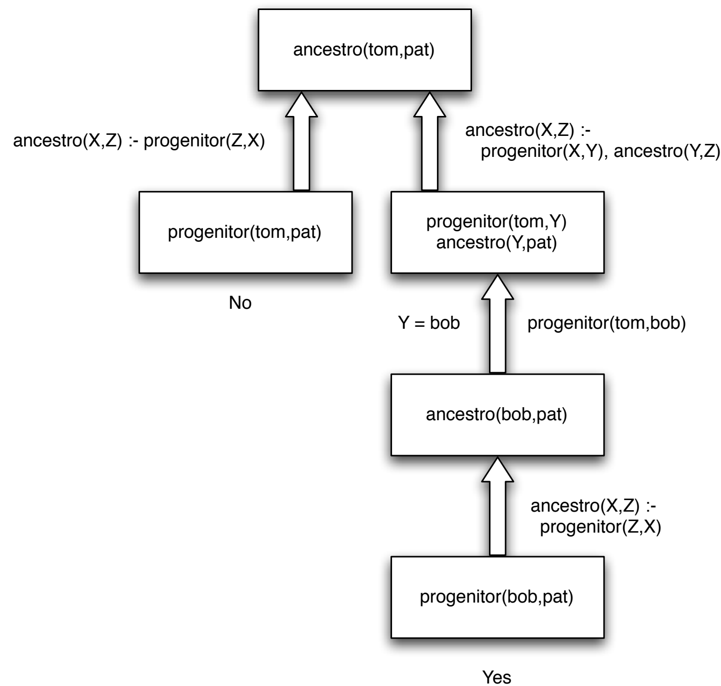


Figura 1.10: *El segundo paso de la ejecución. Dos sub-metas son generadas.*

Con esta explicación, estudien la siguiente sesión en Prolog:

```
1  ?- trace.
2  true.
3
4  [trace]  ?- ancestro(tom,pat).
5     Call: (6) ancestro(tom, pat) ?
6     Call: (7) progenitor(tom, pat) ?
7     Fail: (7) progenitor(tom, pat) ?
8     Redo: (6) ancestro(tom, pat) ?
9     Call: (7) progenitor(_G407, pat) ?
10    Exit: (7) progenitor(bob, pat) ?
11    Call: (7) ancestro(tom, bob) ?
12    Call: (8) progenitor(tom, bob) ?
13    Exit: (8) progenitor(tom, bob) ?
14    Exit: (7) ancestro(tom, bob) ?
15    Exit: (6) ancestro(tom, pat) ?
16 true
```

1.3 PROGRAMACIÓN FUNCIONAL

Resulta curioso que cuando explicamos las ventajas de la programación funcional sobre otros paradigmas de programación, lo hacemos en términos negativos, resaltando cuales son 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.

Siguiendo la propuesta de Hughes [39], esta sección se centra en presentar las ventajas de la programación funcional en términos positivos. La idea central es que 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. Omitir la operación de asignación, los efectos colaterales y el flujo de control son simples medios para este fin.

La sección se organiza de la siguiente manera: Primero presentaremos una serie de conceptos muy básicos sobre funciones puras y la composición de las mismas como parte del diseño modular de programas funcionales. A continuación profundizaremos en esta idea a través de los conceptos de interfaz manifiesta, transparencia referencial, función de orden superior y recursividad.

Utilizaré dos lenguajes de programación funcionales para ilustrar estos conceptos:

LISP Se trata de un lenguaje de programación desarrollado específicamente para satisfacer las necesidades de la comunidad de IA. Influenciado por ideas de la lógica matemática, particularmente por el Cálculo-λ, McCarthy [54] propone un lenguaje para la computación simbólica bajo la forma de procesamiento de listas. Lisp provee programas concisos y naturales, la mayoría de las veces muy cercanos a la definición de las funciones matemáticas que computan. Tres ideas centrales en este lenguaje son:

- Programación aplicable. [6] En lugar de escribir programas, definimos funciones. En lugar de ejecutar programas, evaluamos expresiones.
- Sintaxis simplificada. En particular, la precedencia de los operadores es eliminada; y datos y programas comparten la misma sintaxis.
- Recursividad como la principal estructura de control.

OCAML. Se trata de un lenguaje de programación funcional, propuesta por Rémy y Vouillon [74] como una extensión orientada a objetos al lenguaje funcional ML [33] (*Meta-Language*). Sus características principales son:

- Las funciones son valores de primera clase.
- Se trata de un lenguaje fuertemente tipificado, pero
- Usa inferencia de tipos.
- Es polimórfico, lo que permite escribir programas que funcionan para valores de cualquier tipo.
- Provee un potente mecanismo de emparejamiento de patrones.
- Comparte la semántica formal de todos los lenguajes en la familia ML.

6 Kamin [41] usa este término para diferenciar estas características, de otras presentes en lo que llamamos Programación Funcional. La distinción me parece pertinente, para efectos de esta introducción.

1.3.1 Funciones puras

En matemáticas una función provee un mapeo entre objetos tomados de un conjunto de valores llamado **dominio** y objetos en otro conjunto llamado **rango**.

Dominio y Rango

Ejemplo 1.2. *Un ejemplo simple de función es aquella que mapea el conjunto de los enteros a uno de los valores en* $\{pos, neg, cero\}$*, dependiendo si el entero es positivo, negativo o cero. Llamaremos a esta función signo. El dominio de signo es entonces el conjunto de los números enteros y su rango es el conjunto* $\{pos, neg, cero\}$.

Podemos caracterizar nuestra función mostrando explícitamente los elementos en el dominio y el rango y el mapeo que establece la función. A este tipo de caracterizaciones se les llama por **extensión**.

Extensión

Ejemplo 1.3. *Caracterización de la función signo por extensión:*

$$\begin{array}{rcl} & \vdots & \\ signo(-3) & = & neg \\ signo(-2) & = & neg \\ signo(-1) & = & neg \\ signo(0) & = & cero \\ signo(1) & = & pos \\ signo(2) & = & pos \\ signo(3) & = & pos \\ & \vdots & \end{array}$$

También podemos caracterizar una función a través de reglas que describan el mapeo que establece la función. A esta descripción se le llama por intensión o **intensional**.

Intensión

Ejemplo 1.4. *Caracterización intensional de la función signo:*

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

En la definición intensional de $signo/1$, x se conoce como el **parámetro formal** de la función y representa cualquier elemento dado del dominio. Como sabemos, 1 es la aridad de la función *signo*, es decir, el número de parámetros formales de la función en cuestión. El cuerpo de la regla simplemente especifica a que elemento del rango de la función, mapea cada elemento del dominio. La regla que define $signo/1$ representa un conjunto infinito de ecuaciones individuales, una para cada valor en el dominio. Debido a que esta 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**.

Parámetro formal

Función parcial y total

Ejemplo 1.5. *La función parcial signo2 indefinida cuando* $x = 0$*:*

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

En los lenguajes funcionales fuertemente tipificados, como Ocaml [53, 23, 12, 73], el dominio y rango de una función debe especificarse, ya sea explícitamente o bien mediante su sistema de **inferencia de tipos**. En los lenguajes tipificados dinámicamente, como Lisp [64, 34, 35, 81], esto no es necesario. Veamos esta diferencia en el siguiente ejemplo, donde definimos $suma/2$ en Ocaml, una función que suma sus dos argumentos:

Inferencia de tipos

```
# let suma x y = x + y ;;
val suma : int -> int -> int = <fun>
```

la línea 2 nos dice que *suma*/2 es una función que va de los enteros (*int*), a los enteros y a los enteros. Esto es, que dados dos enteros, computará un tercero [7]. Ocaml tiene predefinidos una serie de tipos de datos primitivos, que incluyen a los enteros. Observen que en este caso, Ocaml ha inferido el tipo de dato de los parámetros de la función y la función misma.

En Lisp, la definición de *add* sería como sigue:

```
CL-USER > (defun suma (x y) (+ x y))
SUMA
```

la línea 2 nos dice que *suma* ha sido definida como función, pero ha diferencia de la versión en Ocaml, los parámetros formales y la función misma no están restringidos a ser necesariamente enteros. Por ejemplo, la misma función puede sumar números reales. Esto no significa que estas expresiones no tengan un tipo asociado, sino que el tipo en cuestión será definido dinámicamente en tiempo de ejecución.

Revisemos otro ejemplo. Si queremos definir *signo*/1 en Ocaml, antes debemos definir un tipo de datos que represente al rango de la función, en este caso el conjunto $\{pos, cero, neg\}$:

```
# type signos = Neg | Cero | Pos ;;
type signos = Neg | Cero | Pos
# let signo x =
    if x<0 then Neg else
    if x=0 then Cero else Pos ;;
val signo : int -> signos = <fun>
# signo 5 ;;
- : signos = Pos
# signo 0 ;;
- : signos = Cero
# signo (-2) ;;
- : signos = Neg
```

Observen que la función *signo*/1 está definida como un mapeo de enteros a signos (línea 6). Los paréntesis en la línea 11 son necesarios, pues '$-/1$ es una función y queremos aplicar *signo*/1 a -2 y no a $-$.

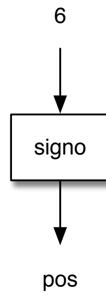
En Lisp, la definición de signo no requiere de una declaración de tipo:

```
CL-USER> (defun signo (x)
            (cond ((< x 0) 'neg)
                  ((zerop x) 'cero)
                  (t 'pos)))
SIGNO
CL-USER> (signo 3)
POS
CL-USER> (signo -2)
NEG
CL-USER> (signo 0)
CERO
```

Podemos ver a una función como una caja negra con entradas representadas por sus parámetros formales y una salida representando el resultado computado por la función. La salida obviamente es uno de los valores del rango de la función en cuestión. La elección de qué valor se coloca en la salida está determinada por la regla que define a la función.

Ejemplo 1.6. *La función signo como caja negra:*

7 ¿Saben ustedes qué hace esta función si sólo se le da un entero?



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 el ejemplo anterior diríamos que *signo* se aplica a 6, para expresar que la regla de signo es invocada usando 6 como parámetro actual. En muchas ocasiones nos referiremos al parámetro actual y formal de una función como los argumentos de la función. La aplicación de la función *signo* a 6 puede expresarse en notación matemática:

Parámetro actual

Aplicación

$$signo(6)$$

Decimos que está aplicación *evaluó* a *pos*, lo que escribimos como:

$$signo(6) \to pos$$

La expresión anterior también indica que *pos* es la salida de la caja negra *signo* cuando se le provee el parámetro actual 6.

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.

Composición

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

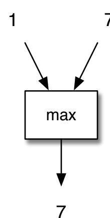
$$max(m,n) = \begin{cases} m & \text{si } m > n \\ n & \text{en cualquier otro caso} \end{cases}$$

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:

$$max : int \times int \to int$$

Podemos ver a *max* como una caja negra para computar el máximo de dos números:

Ejemplo 1.7. *La función max como caja negra:*



Lo cual escribimos:

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

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. Observan también que la aridad de la función es uno, pues acepta como argumento un par de valores numéricos de cualquier tipo. El resultado computado es del mismo tipo que los valores numéricos de entrada. Si queremos estrictamente una función de pares de enteros a enteros podemos usar:

Polimorfismo

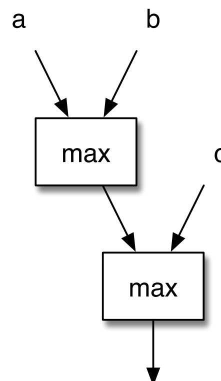
```
1 # let max ((m:int),(n:int)) = if m > n then m else n ;;
2 val max : int * int -> int = <fun>
3 # max(3.4,5.6) ;;
4 Characters 3-12:
5   max(3.4,5.6) ;;
6      ^^^^^^^^^
7 Error: This expression has type float * float but is here used
8        with type int * int
```

Ahora podemos usar max como bloque de construcción para obtener funciones más complejas. Supongamos que requerimos de una función que computa el máximo de tres números, en lugar de sólo dos. Podemos definir está función como $max3$:

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

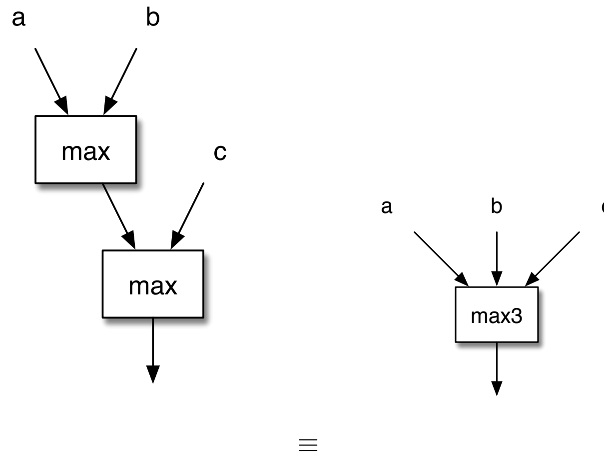
El último caso se requiere cuando $a = b = c$. Esta definición es bastante complicada. Una forma mucho más elegante de definir $max3$ consiste en usar la función max previamente definida.

Ejemplo 1.8. *La función max3 como composición usando max:*



Lo cual escribimos como $max3(a, b, c) = max(max(a, b), c)$. Podemos tratar a $max3$ como una caja negra con tres entradas.

Ejemplo 1.9. *La función max3 como caja negra usando la composición de max:*

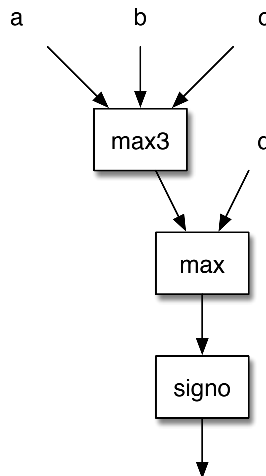


En Ocaml, la función $max3/1$ se escribiría como:

```
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 caha negra para construir nuevas funciones.

Ejemplo 1.10. *La función signomax4 computa el signo del máximo de 4 números:*



Que escribímos como $signomax4(a, b, c, d) = signo(max(max3(a, b, c), d))$. En Ocaml se definiría como:

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

Así que, dados un conjunto de funciones predefinidas, por ejemplo aritmética básica, podemos construir nuevas funciones en términos de esas primitivas. Luego, esas nuevas funciones pueden usarse para construir nuevas funciones más complejas.

1.3.2 Transparencia referencial

La propiedad fundamental de las funciones matemáticas que permite la analogía con los bloques de construcción se llama transparencia referencial. Intuitivamente esto quiere decir 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.

Efectos colaterales

Una función con referencia transparente tiene como característica que dados los mismos parámetros para su aplicación, obtendremos siempre el mismo resultado. Mientras que en matemáticas todas las funciones ofrecen transparencia referencial, ese no es el caso en los lenguajes de programación. Consideren la función `GetInput()`, su salida depende del lo que el usuario teclee! Múltiples llamadas a la función `GetInput` con el mismo parámetro (una cadena vacía), producen diferentes resultados.

Veamos otro ejemplo. Una persona evaluando la expresión $(2ax + b)(2ax + c)$ no se molestaría jamás por evaluar dos veces la sub expresión $2ax$. Una vez que determina que $2ax = 12$, la persona substituirá 12 por ambas ocurrencias de $2ax$. Esto se debe a que una expresión aritmética dada en un contexto fijo, producirá siempre el mismo valor como resultado. Dados los valores $a = 3$ y $x = 2$, $2ax$ será siempre igual a 12. La transparencia referencia resulta del hecho de que los operadores aritméticos no tienen memoria, por lo que toda llamada al operador con los mismos parámetros actuales, producirá la misma salida.

¿Porqué es importante una propiedad como la transparencia referencial? Por las matemáticas sabemos lo importante de poder substituir iguales por iguales. Esto nos permite derivar nuevas ecuaciones, a partir de las ecuaciones dadas, transformar expresiones en formas más útiles y probar propiedades acerca de tales expresiones. En el contexto de los lenguajes de programación, la transparencia referencia permite además optimizaciones tales como la eliminación de sub-expresiones comunes, como en el ejemplo anterior $2ax$.

Observemos ahora que pasa con lenguajes que no ofrecen transparencia referencial. Consideren la siguiente definición de una pseudo-función en Pascal:

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

Debido a que F guarda un registro en a del número de veces que la función ha sido aplicada, no podríamos eliminar la sub-expresión común en la expresión $(a + 2 * F(b)) * (c + 2 * F(b))$. Esto debido a que al cambiar el número de veces que F ha sido aplicada, cambia el resultado de la expresión.

1.3.3 Funciones de orden superior

Otra idea importante en la programación funcional es el concepto de función de orden superior, es decir, funciones que toman otras funciones como sus argumentos, o bien, regresan funciones como su resultado. La derivada en el cálculo, es ejemplo de una función que mapean a otra función.

Las funciones de orden superior permiten utilizar la **técnica de Curry** en la cual una función es aplicada a sus argumentos, uno a la vez. Cada aplicación regresa una función de orden superior que acepta el siguiente argumento. He aquí un ejemplo en Ocaml para la función $suma/2$ en dos versiones. La primera de ellas enfatiza el carácter curry del ejemplo:

Curry

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

```
4 val suma : int -> int -> int = <fun>
5 # suma 3 4 ;;
6 - : int = 7
7 # suma 3 ;;
8 - : int -> int = <fun>
```

El tipo de *suma*/2 nos indica que está función toma sus argumentos uno a uno. Puede ser aplicada a dos argumentos, como suele hacerse normalmente (líneas 5 y 6); pero puede ser llamada con un sólo argumento, !regresando una función en el domino de enteros a enteros! (líneas 7 y 8). Esta aplicación espera un segundo argumento para dar el resultado de la suma.

Las funciones de orden superior pueden verse como un nuevo pegamento conceptual que permite usar funciones simples para definir funciones más complejas. Esto se puede ilustrar con un problema de procesamiento de listas: la suma de los miembros de una lista. Recuerden que una lista es un **tipo de dato recursivo**, la lista es una lista vacía (*nil*) o algo pegado (*cons*) a una lista:

Tipo de datos recursivo

```
1 listade X ::= nil | cons X (listade X)
```

Por ejemplo: *nil* es la lista vacía; a veces la lista vacía también se representa como $[]$; la lista $[1]$, es una abreviatura de $cons\ 1\ nil$; y la lista $[1,2,3]$ es una abreviatura de $cons\ 1\ (cons\ 2\ (cons\ 3\ nil))$.

La *sumatoria* de los elementos de una lista se puede computar con una función recursiva:

$$\begin{aligned} sumatoria\ nil &= 0 \\ sumatoria\ (cons\ num\ lst) &= num + sumatoria\ lst \end{aligned}$$

Si pensamos en cómo programar el *producto* de los miembros de una lista, observaremos que esa operación y la *sumatoria* que acabamos de definir, pueden plantearse como un patrón recursivo recurrente general, conocido como *reduce*:

$$sumatoria = reduce\ suma\ 0$$

donde por conveniencia, en lugar de un operador infijo $+$ para la suma, usamos la función *suma*/2 definida previamente:

$$suma\ x\ y = x + y$$

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

$$\begin{aligned} reduce\ f\ x\ nil &= x \\ reduce\ f\ x\ (cons\ a\ l) &= (f\ a)(reduce\ f\ x\ l) \end{aligned}$$

Observen que al definir *sumatoria*/1 estamos aplicando *reduce*/3 a únicamente dos argumentos, por lo que el llamado resulta en una función de orden superior, que espera un parámetro más. En general, una función de aridad n, aplicada a sólo $m < n$ argumentos, genera una función de aridad $n - m$. En el resto de esta revisión, ejemplificaré estos conceptos con código en Ocaml. Observen la definición de *suma*/2 y las dos llamadas a esta función:

```
1 # let suma x y = x + y;;
2 val suma : int -> int -> int = <fun>
3 # suma 4 5;;
4 - : int = 9
5 # suma 4;;
6 - : int -> int = <fun>
```

la definición de $suma/2$ (línea 1) nos dice que *suma* es una función de los enteros, a los enteros, a los enteros (línea 2); esto es, el resultado de computar *suma* es también un entero, como puede observase en la primera aplicación de la función (línea 3) cuyo resultado es el entero nueve (línea 4). Sin embargo, la segunda llamada a la función (línea 5), da como resultado otra función de los enteros a los enteros (línea 6), que podríamos interpretar como "incrementa en 4 el entero que recibas". Veamos ahora la definición de $reduce/3$ y $sumatoria/1$ en términos de reduce:

```
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  # let sumatoria = reduce suma 0 ;;
6  val sumatoria : int list -> int = <fun>
```

la definición de *reduce* es recursiva, de ahí que se incluya la palabra reservada *rec* (línea 1) para que el nombre de la función pueda usarse en su misma definición al hacer el llamado recurrente. Esta función recibe tres argumentos una función f, un elemento x y una lista de elementos l. Si la lista de elementos está vacía, la función regresa x; si ese no es el caso, aplica f al primero elemento de la lista (línea 2), y esa función de orden superior es aplicada al *reduce* del resto de la lista (línea 3). La signatura de $reduce/3$ (línea 4) nos indica que hemos definido una función de orden superior, es decir, una función que recibe funciones como argumento. La lectura de los tipos aquí es como sigue: l es una lista de elementos de tipo $'a$ (cualquiera que sea el caso); x es de tipo $'b$; puesto que f se aplica a elementos de la lista y en última instancia a x, se trata de una función de $'a$ a $'b$ a $'b$; y por tanto el resultado de la función *reduce* es de tipo $'b$. La función $suma/2$ recibe una lista de enteros y produce como resultado un entero. La inferencia de tipos en este caso es porque hemos introducido 0 en la llamada y Ocaml puede inferir que $'b$ es este caso es el conjunto de los enteros.

Ahora viene lo interesante, podemos definir el producto de una lista reutilizando *reduce*:

```
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
7  # suma [1;2;3;4];;
8  - : int = 10
```

Una forma intuitiva de entender *reduce* es considerarla como un transformador de listas que substituye cada *cons* por el valor de su primer argumento f y cada *nil* por el valor del segundo x. Así, la llamada a $producto[1;2;3]$ es en realidad una abreviatura de

$$producto\ cons\ 1\ (cons\ 2\ (cons\ 3\ []))$$

que *reduce* convierte a

$$mult\ 1\ (mult\ 2\ (mult\ 3\ 1))$$

lo cual evalúa a 6.

Veamos otro ejemplo sobre este patrón recurrente aplicado a listas. Podemos hacer explicito el operador *cons* con la siguiente definición:

```
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]
```

cuya signatura nos dice que x debe ser un elemento de un cierto tipo, y que y es una lista de elementos de ese mismo tipo. La salida de la función es la lista construida al agregar x al frente de y, como lo muestran los ejemplos a partir de la línea 3. Ahora que contamos con $cons/2$ podemos definir $append/2$ usando esta definición:

```
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]
```

O, siguiendo la misma estrategia, podemos definir una función que reciba una lista enteros y regrese la lista formada por el doble de los enteros originales:

```
1  # let dobleycons num lst = cons (2*num) lst;;
2  val dobleycons : int -> int list -> int list = <fun>
3  # let dobles = reduce dobleycons [];;
4  val dobles : int list -> int list = <fun>
5  # dobles [1;2;3;4];;
6  - : int list = [2; 4; 6; 8]
```

Composición y mapeo

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>
```

De forma que ahora podemos definir $dobles2/1$ en términos de una **función anónima** y un mapeo:

```
1  # let dobles2 l = map (fun x -> 2*x) l;;
2  val dobles2 : int list -> int list = <fun>
3  # dobles2 [1;2;3;4] ;;
4  - : int list = [2; 4; 6; 8]
```

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
```

1.3.4 Recursividad

Las iteraciones de la programación tradicional, son normalmente implementados de manera recursiva en la programación funcional. Las funciones recursivas [54], 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 estructuras de datos (tipos de datos) recursivas. Aunque las listas están definidas en estos lenguajes, observen las siguientes definiciones de tipo “lista de” y “arbol de” en Ocaml:

```
# type 'a lista = Nil
                | Cons of 'a * 'a lista ;;
type 'a lista = Nil | Cons of 'a * 'a lista
# type 'a arbolbin = Hoja
                   | Nodo of 'a * 'a arbolbin * 'a arbolbin ;;
type 'a arbolbin = Hoja | Nodo of 'a * 'a arbolbin * 'a arbolbin
```

son definiciones de tipos de datos !recursivas! Una lista vacía es una lista y algo pegado a una lista vacía es una lista. Una hoja es un árbol, y un nodo pegado a dos arboles, es un árbol. A continuación definiremos *miembro*/2 para estos dos tipos de datos:

```
# let rec miembro elt lst = match lst with
    | Nil -> false
    | Cons(e,l) -> if e=elt then true else miembro elt l;;
val miembro : 'a -> 'a lista -> bool = <fun>
# let rec miembro elt arbol = match arbol with
      Hoja -> false
    | Nodo(e,izq,der) -> if e=elt then true
                         else miembro elt izq || miembro elt der ;;
val miembro : 'a -> 'a arbolbin -> bool = <fun>
# miembro 2 (Cons(1,Nil)) ;;
- : bool = false
# miembro 1 (Cons(1,Nil)) ;;
- : bool = true
# miembro 2 (Nodo(1,Nodo(2,Hoja,Hoja),Nodo(3,Hoja,Hoja)));;
- : bool = true
# miembro 4 (Nodo(1,Nodo(2,Hoja,Hoja),Nodo(3,Hoja,Hoja)));;
- : bool = false
```

Los patrones y la recursividad pueden factorizarse utilizando funciones de orden superior, los catamorfismos y los anamorfismos son los ejemplos más obvios. Estas funciones de orden superior juegan un papel análogo a las estructuras de control de la programación imperativa.

Observen que aunque en Prolog y Lisp no es necesario definir tipos de datos, las estructuras recursivas como las listas y los árboles pueden usarse de manera similar. Por ejemplo, un árbol en prolog puede definirse como un hecho:

```
arbol(1,arbol(2,hoja),arbol(3,hoja)).
```

por lo que un predicado miembro sigue un patrón similar al aquí revisado. En el caso de Lisp, el árbol puede representarse como una lista de listas y el patrón de miembro seguirá siendo similar al aquí visto:

```
(1 (2 hoja) (3 hoja))
```

1.4 ORGANIZACIÓN DEL CURSO

El curso está organizado en tres bloques. El primero de ellos aborda la Programación Lógica. El capítulo 2 introduce los fundamentos teóricos de este paradigma con base en los conceptos de lógica de primer orden, programas definitivos y resolución-SLD. El capítulo 3 introduce el lenguaje de programación Prolog, con el suficiente detalle, como para abordar la implementación de algoritmos de búsqueda (capítulo 4) e inducción de árboles de decisión (capítulo 5); ejemplos ambos de aplicaciones de la IA.

El segundo y tercer bloque abordan la Programación Funcional. El capítulo 6 introduce los fundamentos teóricos de este paradigma, con base en el Cálculo-λy la ayuda de un lenguaje aplicable simplificado, conocido como AL. El capítulo 7

introduce el lenguaje de programación Lisp, mientras que el 8 y el 9 presentan aplicaciones del mismo, el primero en el dominio de la bioinformática, y el segundo revisita los árboles de decisión, para comparar esta aproximación con la basada en programación lógica. El capítulo 10 introduce el lenguaje de programación Objective Caml, que es estrictamente más funcional que Lisp. El capítulo 11, ilustra su aplicación en la IA. Finalmente el capítulo 12 presenta otra implementación de la inducción de árboles de decisión.

1.5 LECTURAS Y EJERCICIOS SUGERIDOS

Con respecto a Prolog, existen numerosas referencias. Se recomienda la lectura de los capítulos 1 y 2 del libro de Sterling y Shapiro [87]. Una parte de este capítulo está basada en el capítulo 1 del libro de Bratko [10], una excelente aproximación a los problemas de la IA basada en Prolog. Otra lectura complementaria, aunque un poco más allá de lo hasta ahora expuesto, es el capítulo 1 del libro de Clocksin y Melish [18].

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 Hughes [39] *Why Functional Programming matters*. Hudak [38] nos ofrece otro artículo introductorio interesante, por la perspectiva histórica que asume, al presentar los lenguajes funcionales de programación. Dos textos donde pueden revisarse los conceptos generales de la programación funcional, son el libro de Field y Harrison [27] y el de MacLennan [51]. El texto definitivo sobre el uso de Lisp en la IA es de Norvig [64]. Un enfoque práctico fuera de la IA, actualizado y fresco, es presentado por Seibel [81]. Otra buena introducción a Lisp es la que ofrece Graham [35]. Las referencias a Ocaml incluyen los libros de Cousineau y Mauny [23], Chailloux, Manoury y Pagano [12] y Minsky, Madhavapeddy e Hickey [58].

Ejercicios

Ejercicio 1.1. *Verifique que las relaciones abuela/2 y madre/2 sean correctas. Señale las diferencias y similitudes con la verificación de la relación hermana/2*

Ejercicio 1.2. *Defina en Prolog las relaciones tio/2 y tia/2 y pruébelas usando la genealogía de la figura 1.4.*

Ejercicio 1.3. *¿Pueden verse estas relaciones como funciones puras? Justifique su respuesta.*

Ejercicio 1.4. *Defina en Ocaml una función suma para números reales.*

Ejercicio 1.5. *Observe el comportamiento de la función predefinida max:*

```
1 # max 1 2 ;;
2 - : int = 2
3 # max "alejandro" "alejandra" ;;
4 - : string = "alejandro"
```

¿Cómo se llama la propiedad de esta función que hace posible que reciba argumentos enteros y cadenas?

Ejercicio 1.6. *¿Qué significa el término función de orden superior? Ilustre su respuesta con un ejemplo.*

Ejercicio 1.7. *¿Cómo definiría una lista y un árbol en Prolog? Intenté definir una relación miembro sobre estas estructuras de datos. ¿Y en Lisp?*