



Universidad de Granada

decsai.ugr.es

Inteligencia Artificial

Seminario 1

Aplicaciones: Agentes Conversacionales



DECSAI

**Departamento de Ciencias de la
Computación e Inteligencia Artificial**

- 1. Método evaluación de las prácticas**
- 2. Presentación de la práctica**
- 3. Introducción a los Agentes Conversacionales**
- 4. AIML**

Evaluación de la asignatura

En JUNIO

Teoría	Examen final & Pruebas parciales	50%
Prácticas	Prácticas de laboratorio & Resolución de problemas	50%

Ficha de la asignatura: “... se establece como requisito adicional para superar la asignatura que tanto la calificación correspondiente a la parte teórica como la correspondiente a la parte práctica sean mayores o iguales a 3 (sobre 10).”

En SEPTIEMBRE

Hay un único examen escrito que evalúa teoría y prácticas
NO HAY ENTREGA DE PRÁCTICAS

Evaluación de la parte de prácticas

En JUNIO

Práctica 1	Agente conversacional	25%
Práctica 2	Resolución de problemas con agentes reactivos	25%
Práctica 3	Resolución de un juego basado en técnicas de búsqueda	25%
Examen de problemas		25%

En SEPTIEMBRE

Hay un único examen escrito que evalúa teoría y prácticas
NO HAY ENTREGA DE PRÁCTICAS

Práctica 1

Agentes Conversacionales



- Familiarizarse con una aplicación concreta de la Inteligencia Artificial (IA): la construcción de agentes conversacionales.
- Aprender un lenguaje de representación del conocimiento diseñado específicamente para este problema: el lenguaje AIML 2.0.
- Aprender a utilizar un intérprete de este lenguaje: **program-ab**, la implementación de referencia de AIML.

Se pide construir agentes conversacionales en AIML 2.0 para:

1. Mantener una conversación sobre la asignatura Inteligencia Artificial, de manera que el agente conversacional sea capaz de responder a las preguntas más habituales sobre el funcionamiento y la evaluación de esta asignatura.
2. Jugar al juego “¿Quién es quién?”, un juego bipersonal infantil que consiste en que dos humanos piensan en un personaje (sobre un conjunto concreto y limitado de ellos) y, realizando preguntas sobre sus rasgos, intentan acertar el personaje pensado por el adversario. Gana aquél que acierta antes el personaje.

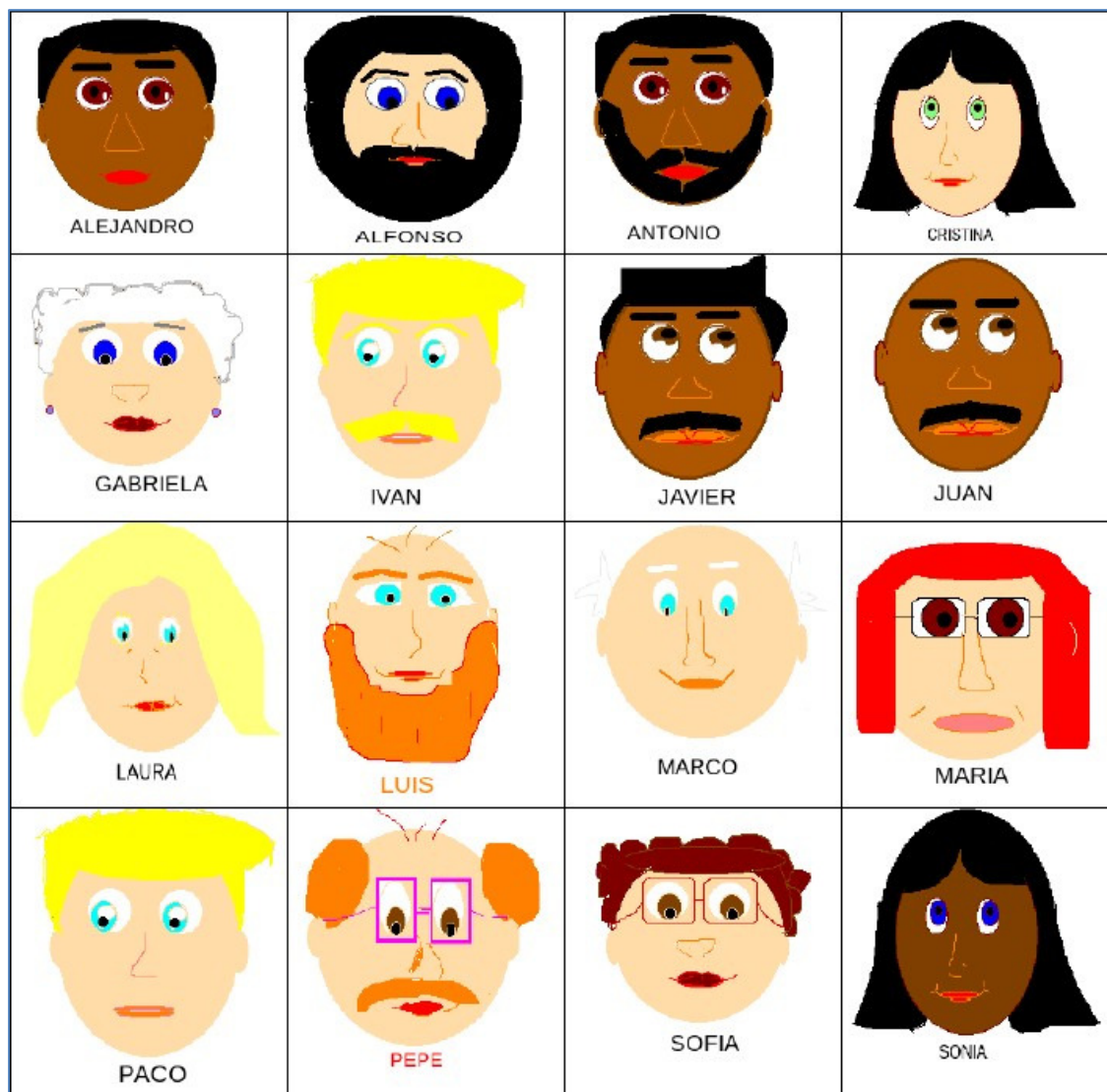


Sobre el primero de ellos, se pide que el agente sea capaz de responder correctamente a las 50 preguntas que aparecen en el anexo I del guion de prácticas. Como resultado, se obtiene un asistente capaz de resolver las dudas más frecuentes que tienen los alumnos sobre la asignatura.

Algunas de las preguntas del anexo 1:

1. COMO APRUEBO IA?
2. COMO HAGO PARA APROBAR LA TEORIA?
3. COMO CONSIGO SUPERAR LA PARTE PRACTICA?
4. COMO SE EVALUA EN LA CONVOCATORIA DE JUNIO?
5. QUE PASA SI TENGO QUE PRESENTARME EN LA CONVOCATORIA DE SEPTIEMBRE?
6. HAY EXAMEN DE PRACTICAS EN LA CONVOCATORIA DE SEPTIEMBRE?
7. QUE PASA SI NO PUEDO ASISTIR A CLASE REGULARMENTE?
8. HAY NOTA MINIMA PARA APROBAR?

En el juego “¿Quién es quién?”, el conjunto de personajes que intervienen son los 16 que se muestran a continuación:



Las preguntas están limitadas a los siguientes rasgos:

- Color de pelo
- Color de ojos
- Color de piel
- Tiene gafas
- Tiene bigote
- Tiene barba
- Sexo

Dibujos de Rosa Rodríguez

Esta es la descripción completa de los rasgos de cada personaje:

	Pelo	Ojos	Piel	Gafas	Bigote	Barba	Sexo
Alejandro	Moreno	Oscuros	Oscura	No	No	No	Hombre
Alfonso	Moreno	Claros	Clara	No	Si	Si	Hombre
Antonio	Moreno	Oscuros	Oscura	No	Si	Si	Hombre
Cristina	Moreno	Claros	Clara	No	No	No	Mujer
Gabriela	Blanco	Claros	Clara	No	No	No	Mujer
Ivan	Rubio	Claros	Clara	No	Si	No	Hombre
Javier	Moreno	Oscuros	Oscura	No	Si	No	Hombre
Juan	Calvo	Oscuros	Oscura	No	No	No	Hombre
Laura	Rubio	Claros	Clara	No	No	No	Mujer
Luis	Pelirrojo	Claros	Clara	No	No	Si	Hombre
Marco	Calvo	Claros	Clara	No	No	No	Hombre
Maria	Pelirrojo	Oscuros	Clara	Si	No	No	Mujer
Paco	Rubio	Claros	Clara	No	No	No	Hombre
Pepe	Pelirrojo	Oscuros	Clara	Si	Si	No	Hombre
Sofia	Marron	Oscuros	Clara	Si	No	No	Mujer
Sonia	Moreno	Claros	Oscura	No	No	No	Mujer

El juego “¿Quién es quién?” lo descomponemos en 3 modalidades, en las que se incrementa gradualmente la dificultad del problema.

En las 3 modalidades, tanto el humano como el agente toman sólo un papel; es decir, sólo uno de ellos hace las preguntas y el otro sólo las responde.

MODALIDAD 1: El humano intenta acertar el personaje

El agente debe responder a preguntas del tipo:

- Tiene el pelo moreno?
- Tiene los ojos claros?
- Es mujer?

El juego termina cuando el humano introduce una afirmación del tipo:

El personaje es Antonio

Debe terminar indicando si ha acertado o no.

MODALIDAD 2: El humano piensa el personaje

El agente es el que debe formular preguntas del tipo:

- Tiene el pelo moreno?
- Tiene los ojos claros?
- Es mujer?

El agente, en función de las respuestas del jugador humano (“sí” o “no”), debe ir descartando rasgos para acertar el personaje.

El juego termina cuando el agente introduce una afirmación del tipo:

El personaje es Laura

El humano le indicará si ha acertado o no.

MODALIDAD 3: El humano piensa el personaje y el agente tiene un comportamiento más inteligente

Similar a la modalidad 2, pero en esta variante se introducen nuevas características que dotan de mayor inteligencia al agente:

1. Cada pregunta que realiza el agente permite reducir el número de personajes posibles sea cual sea la respuesta del humano.

Ejemplo de conversación que ilustra un comportamiento no permitido:

robot: Tiene el pelo rubio?

human: Si

robot: Tiene el pelo moreno?

2. El agente es capaz de detectar que el humano hace trampa.

Ejemplo de conversación que ilustra una trampa del humano detectada por el agente:

robot: Tiene el pelo pelirrojo?

human: Si

robot: Tiene gafas?

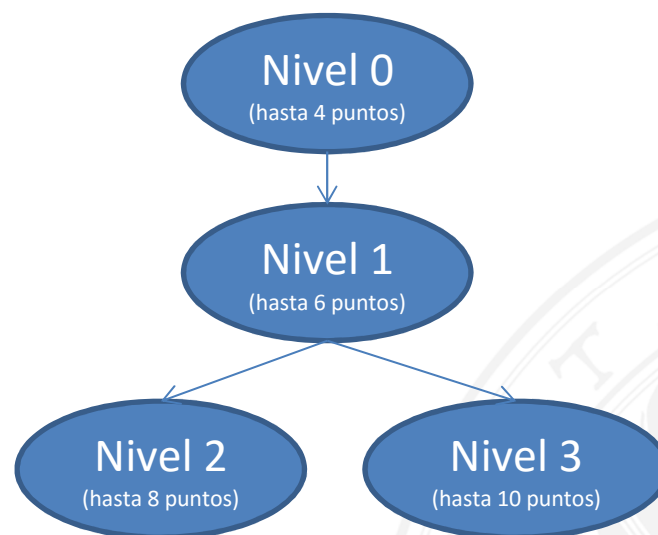
human: No

robot: El personaje es Luis

human: No, es Maria

robot: Eres un tramposo, Maria tiene gafas y dijiste que no tenia gafas

EVALUACIÓN



NIVEL 0: Mínimo que hay que entregar para que la práctica sea considerada para su evaluación.

Consiste en resolver el agente conversacional para responder a preguntas sobre la asignatura I.A.

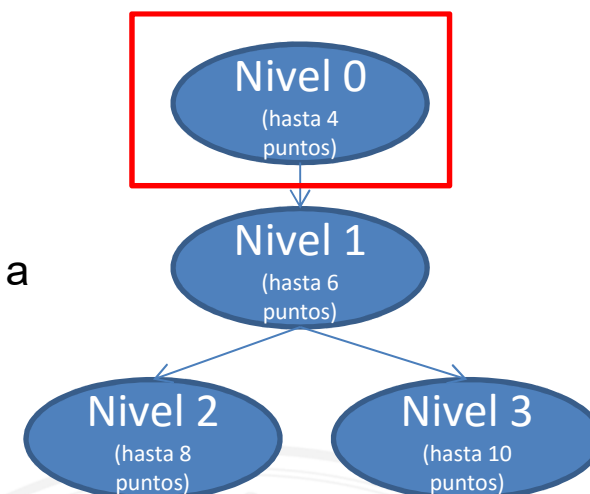
Método de evaluación

Se pedirá al alumno que introduzca 2 de las 50 preguntas que aparecen en el anexo 1 del guion de prácticas.

- Si no responde correctamente a las 2 => 0 puntos y termina la evaluación de la práctica
- Si responde correctamente a las 2 => 2 puntos.

Luego, se le pedirá que introduzca otras 2 preguntas que no son de las establecidas en el anexo 1, porque introducen alguna variación sobre la forma de preguntar.

- Si responde correctamente al menos a uno de las dos => 4 puntos y supera el nivel 0.
- Si no responde a ninguna => el profesor le pedirá que modifique el conocimiento para incluir la nueva pregunta. Si el proceso de inclusión es correcto => 4 puntos y supera el nivel 0.
- En otro caso, se queda con 2 puntos y termina la evaluación.

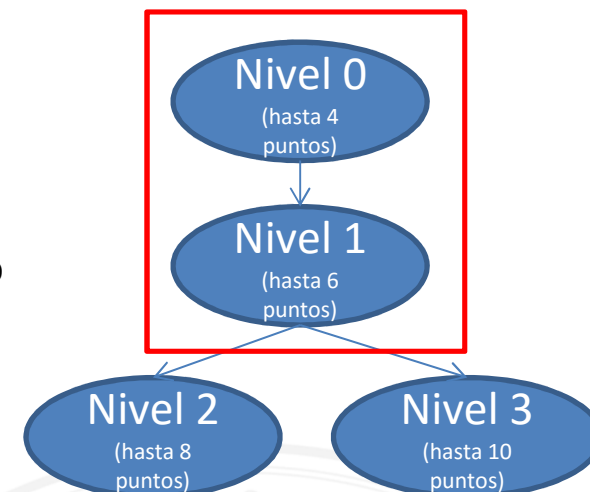


NIVEL 1: Completar el Nivel 0 y la Modalidad 1 del juego “¿Quién es quién?”.

Para evaluar este nivel, se pedirá al alumno que inicie el juego usando su agente y que piense un personaje. El alumno irá introduciendo las preguntas que le indique el profesor.

Si el personaje pensado es coherente con la secuencia de preguntas y respuestas, se dará por superado este nivel y el alumno obtendrá la calificación de 6 puntos.

En otro caso, la calificación del alumno será de 4 puntos y el proceso de evaluación se dará por terminado.



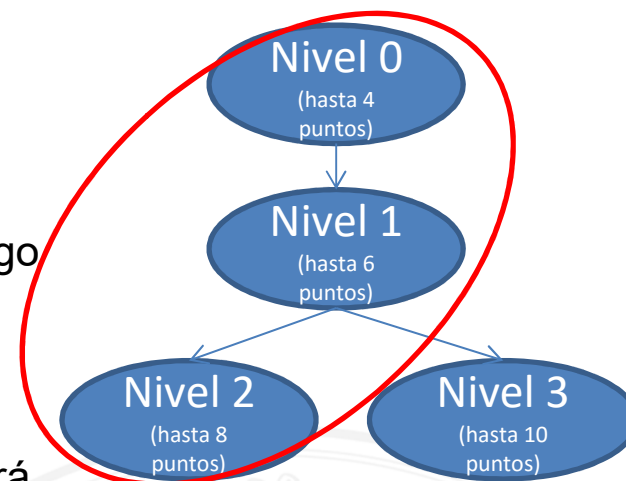
NIVEL 2: Completar el Nivel 1 y la Modalidad 2 del juego “¿Quién es quién?”.

Para evaluar este nivel, se pedirá al alumno que inicie el juego usando su agente y le pedirá empezar el juego donde el humano piensa el personaje.

El agente irá formulando las preguntas y el alumno introducirá las respuestas que le indique el profesor.

Si acierta el personaje, el alumno obtiene un 8 en la calificación y la evaluación termina.

En otro caso, la calificación del alumno será de 6 puntos y el proceso de evaluación se dará por terminado.



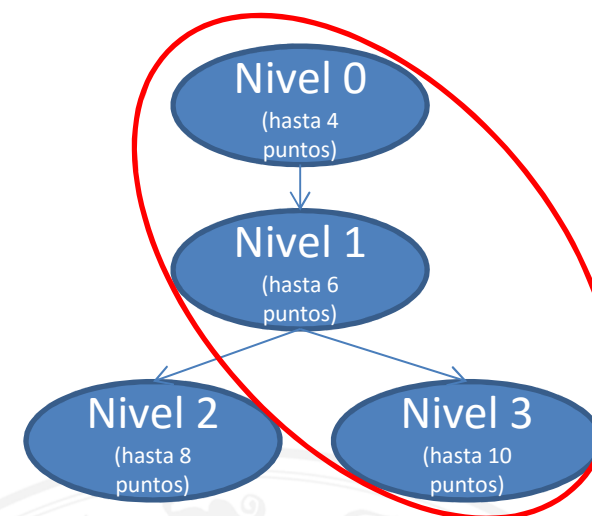
NIVEL 3: Completar el Nivel 1 y la Modalidad 3 del juego “¿Quién es quién?”.

Para evaluar este nivel, se realizará el mismo proceso que se pide en la evaluación del nivel 2.

Además, se repetirá ese proceso para probar que el agente es capaz de detectar si el humano hace trampas.

Si el funcionamiento es correcto, el alumno obtendrá una calificación de entre 9 y 10 dependiendo de los mecanismos incorporados para implementar las características inteligentes del agente.

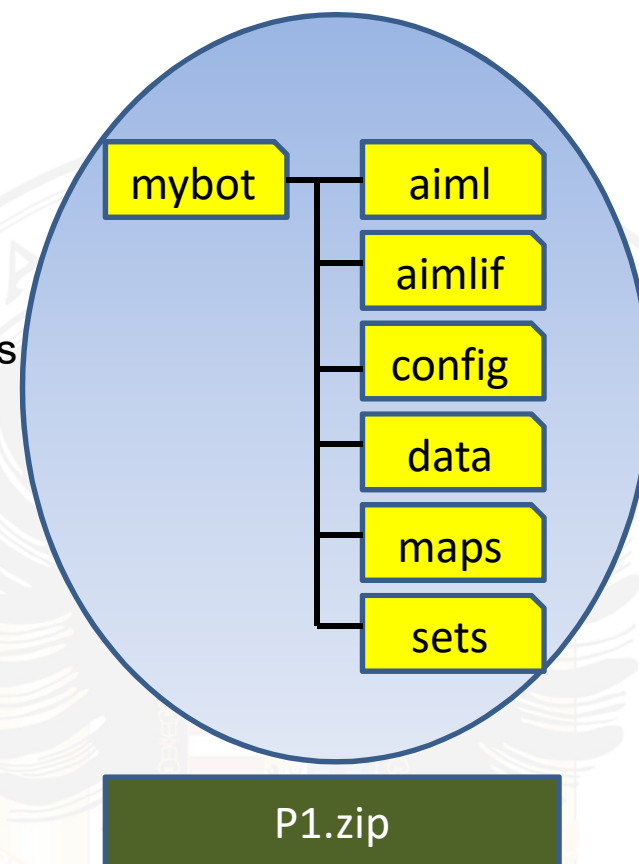
Si el sistema realiza preguntas no inteligentes o no es capaz de detectar la trampa del humano, el alumno obtendrá una calificación de 6.



¿Qué hay que entregar?

Se ha de entregar un archivo comprimido zip llamado “**P1.zip**” que contenga la estructura en directorios de un agente conversacional en **program-ab**, tomando la carpeta “**mybot**” como raíz e incluyendo las carpetas:

1. “aiml”, donde se encuentran los archivos **aiml** que describen el conocimiento del agente conversacional hasta el nivel que ha deseado realizar,
2. “sets”, con los **sets** que necesita para que funcione correctamente su agente,
3. “maps”, con los **maps** que necesita para que funciones correctamente su agente,
4. “aimlif”, con los archivos **csv** que se hayan generado.



Consideraciones Finales

1. Las prácticas son **INDIVIDUALES**.
2. Entre la entrega y la defensa, se procederá a pasar un detector de copias a los ficheros entregados por los alumnos de todos los grupos de prácticas independientemente del grupo de teoría al que pertenezca.
3. Los alumnos asociados con las prácticas que se detecten copiadas, ya sea durante el proceso de detección de copia o durante la defensa de la práctica tendrán automáticamente suspensa la asignatura y deberán presentarse a la convocatoria de septiembre.
4. **Tiene la misma penalización el que copia como el que se deja copiar.** Por eso razón, para prevenir que sucedan estas situaciones, os aconsejamos que en ningún caso paséis vuestro código a nadie y bajo ninguna circunstancia.
5. El objetivo de la defensa es evaluar lo que vosotros habéis hecho, por consiguiente, quién asiste tiene que saber justificar cada cosa de la que aparece en su código. La justificación no apropiada de algún aspecto del código implica considerar la práctica copiada. **CONSEJO: no metáis nada en vuestro código que no sepáis explicar.**
6. El alumno que entrega una práctica y no se presenta al proceso de defensa tiene una calificación de cero en la práctica

Desarrollo Temporal

- 22 de febrero: Presentación de la práctica
- 28 de marzo: Entrega de la práctica.
- 4 de abril: Defensa de la práctica.

La fecha tope para la entrega será el lunes 28 de Marzo antes de las 23:00 horas.

Agentes Conversacionales

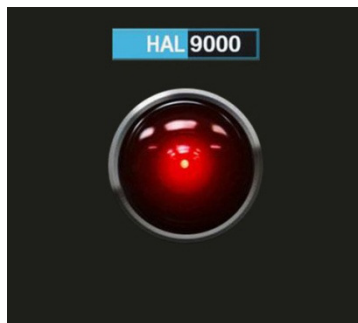
Introducción



Alan Turing establece un test para determinar la inteligencia de un sistema artificial basado en la capacidad de mantener la coherencia durante una conversación con un ser humano.



El cine ha ayudado a establecer esta vinculación entre I.A. y sistemas capaces de conversar con seres humanos:



2001 Una odisea
en el espacio
(1968)



C3PO

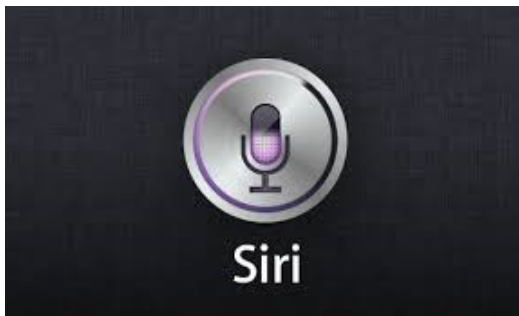
Star Wars
(1977)



Her
(2013)

En estos últimos años ha surgido una gran cantidad de sistemas basados en agentes conversacionales cuyo objetivo es facilitar o ayudar a los seres humanos a realizar algunas tareas.

Son conocidos como asistentes...



JIBO

El lenguaje AIML

1. Estructura básica de AIML
2. El intérprete “program-ab”
3. Wildcards o “comodines”
4. Variables
5. Reducción Simbólica (<srai>)
6. Sets y Maps
7. Contexto
8. Random, estructuras condicionales y ciclos
9. Aprender

El lenguaje AIML

Estructura Básica de AIML



AIML (Artificial Intelligence Markup Language) es un lenguaje basado en XML para crear aplicaciones de Inteligencia Artificial: Orientado al desarrollo de interfaces que simulan el comportamiento humano,

¿Por qué usar AIML?

- Es un lenguaje simple, fácil de entender y mantener.
- Existen distintos intérpretes para este lenguaje.

Estructura básica:

```
<?xml version="1.0" encoding="UTF-8"?>  
<aiml version="2.0">
```

```
<category>
```

```
<pattern>Esta es la pregunta</pattern>
```

```
<template>Esta es la respuesta</template>
```

```
</category>
```

```
</aiml>
```

El lenguaje AIML

El intérprete
“program-ab”



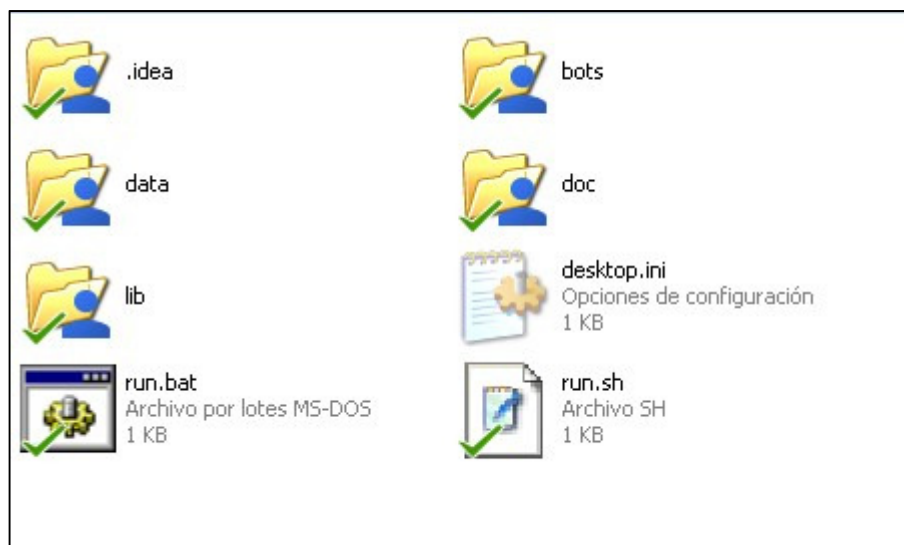
program-ab es un intérprete “open source” de AIML 2.0.

Será el que usaremos en la práctica...

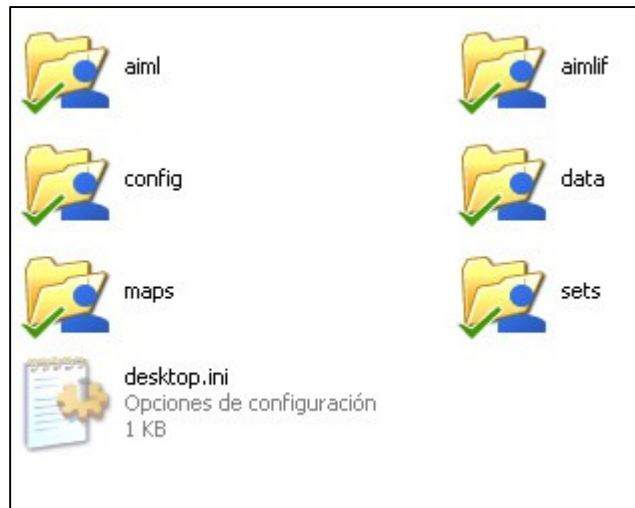
Para comenzar a utilizarlo:

1. Descargamos el archivo “program-ab.zip”
2. Se descomprime el ZIP en una carpeta.

3. Accedemos a la carpeta “bots”.
4. Accedemos a la carpeta “mybot”.



La estructura de un “bot”



- **aiml** (carpeta donde se incluyen los archivos con extensión .aiml).
- **aimlif** (carpeta donde se almacena lo aprendido por el “bot”). En esta carpeta, la extensión de los ficheros es .csv
- **config** (carpeta que contiene la información de configuración del “bot”)
- **data** (carpeta donde se almacena información temporal del intérprete). En esta práctica, podemos ignorarla.
- **sets** (carpeta donde se almacenan los “sets” que utilizará el intérprete).
- **maps** (carpeta donde se almacenan los “maps” que usará el intérprete).

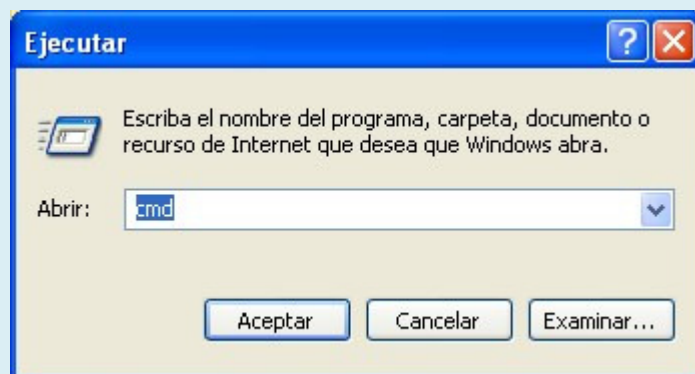
5. Accedemos a la carpeta **aiml**
6. Creamos un fichero que llamaremos "**primero.aiml**", (en Windows, pulsamos botón derecho, elegimos "**nuevo**"-> "**documento de texto**" y, in a vez creado, le cambiamos la extensión **.txt** a **.aiml**)
7. En las aulas de prácticas, podemos usar *CodeBlocks*. Abrimos el fichero "**primero.aiml**"
8. Copiamos en el editor el siguiente texto:

```
<?xml version="1.0" encoding="UTF-8"?>
<aiml version="2.0">

<!-- Primera regla -->
<category>
<pattern>Hola!!</pattern>
<template>Hola, que tal?</template>
</category>

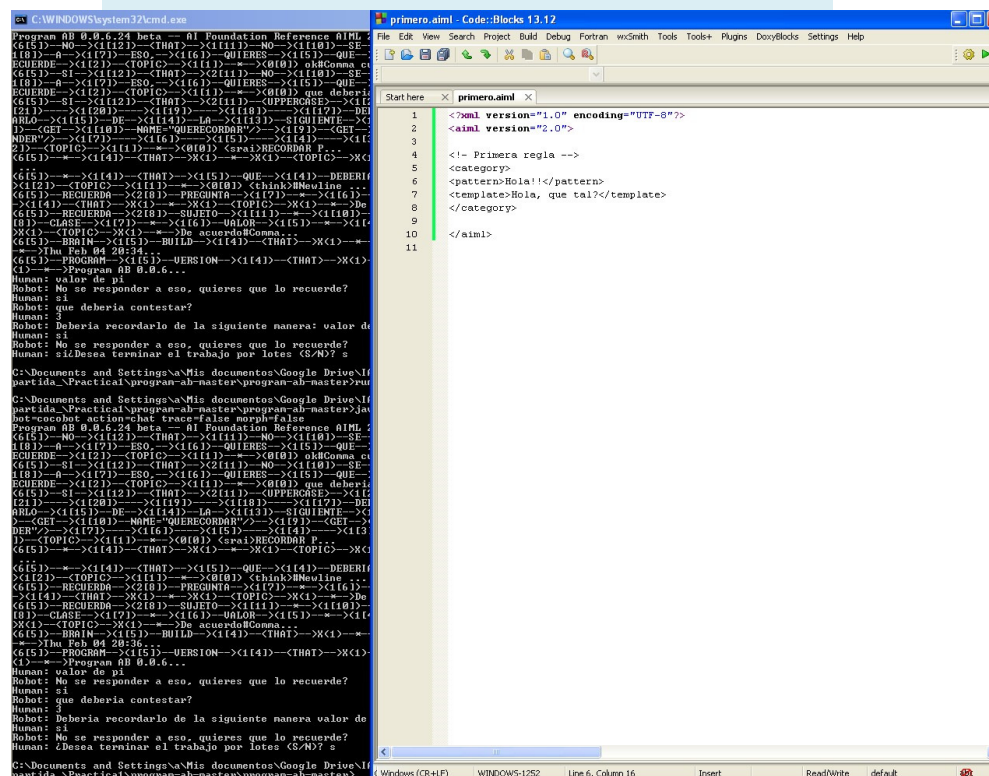
</aiml>
```

9. Ya que el editor y el intérprete no están integrados, haremos lo siguiente: "inicio -> ejecutar", pondremos "cmd" y le daremos a "Aceptar"



10. En el terminal que nos aparece nos movemos hasta el directorio raíz de **program-ab**, donde está el archivo **"run.bat"**

11. Y organizamos la pantalla de la siguiente forma:



12. En el terminal ejecutamos **"run"**...

13. **program-ab** cargará todos los archivos con extensión **aiml** que encuentre en la carpeta **aiml**; en nuestro caso, solamente el archivo "**primero.aiml**" que contiene una única regla. Cuando termine aparecerá "**Human:**" que es donde nosotros introducimos nuestra parte del diálogo con el bot. En este caso, tecleamos "**Hola!!**", y en "**Robot:**" nos contestará "**Hola, que tal?**".

```
C:\Documents and Settings\A\Mis documentos\Google Drive\IA 2015-2016 Carpeta com
partida_\Practical\program-ab-master\program-ab>run

C:\Documents and Settings\A\Mis documentos\Google Drive\IA 2015-2016 Carpeta com
partida_\Practical\program-ab-master\program-ab>java -cp lib/Ab.jar Main bot=myb
ot action=chat trace=false morph=false
Program AB 0.0.6.24 beta -- AI Foundation Reference AIML 2.1 implementation
(3[5])--HOLA-->(1[4])--<THAT>-->X(1)--*-->X(1)--<TOPIC>-->X(1)--*-->Hola#Comma q
ue t...
(3[5])--BRAIN-->(1[5])--BUILD-->(1[4])--<THAT>-->X(1)--*-->X(1)--<TOPIC>-->X(1)-
*-->Mon Feb 08 11:43...
(3[5])--PROGRAM-->(1[5])--VERSION-->(1[4])--<THAT>-->X(1)--*-->X(1)--<TOPIC>-->X
(1)--*-->Program AB 0.0.6...
Human: Hola!!
Robot: Hola, que tal?
Human:
```

Antes de que **program-ab** consulte el conocimiento contenido en los archivos **aiml**, realiza un preprocesamiento consistente en lo siguiente:

- Eliminación de los signos de puntuación, interrogación, admiración, ...
- Transformación a mayúsculas de todo el contenido.
- Extiende las contracciones (esto heredado del proceso del inglés)

Así, para nuestra regla, las siguientes entradas son equivalentes:

- Hola!!
- Hola
- hola!
- !!HOla!!
- Hola

IMPORTANTE: Esta versión del intérprete AIML no reconoce las tildes ni la “ñ”, así que no usaremos estos símbolos para definir las reglas!!

¿Qué ocurre si respondemos al “Hola, que tal?” del robot con “Estoy bien”.

```
C:\Documents and Settings\A\Mis documentos\Google Drive\IA 2015-2016 Carpeta com
partida_\Practical\program-ab-master\program-ab>run

C:\Documents and Settings\A\Mis documentos\Google Drive\IA 2015-2016 Carpeta com
partida_\Practical\program-ab-master\program-ab>java -cp lib/Ab.jar Main bot=myb
ot action=chat trace=false morph=false
Program AB 0.0.6.24 beta -- AI Foundation Reference AIML 2.1 implementation
<3[51]--HOLA--><1[41]--<THAT-->X(1)--*-->X(1)--<TOPIC-->X(1)--*-->Hola#Comma q
ue t...
<3[51]--BRAIN--><1[51]--BUILD--><1[41]--<THAT-->X(1)--*-->X(1)--<TOPIC-->X(1)--
*-->Mon Feb 08 11:43...
<3[51]--PROGRAM--><1[51]--VERSION--><1[41]--<THAT-->X(1)--*-->X(1)--<TOPIC-->X
(1)--*-->Program AB 0.0.6...
Human: Hola!!
Robot: Hola, que tal?
Human: Estoy bien
Robot: I have no answer for that.
Human: 
```

Como vemos, responde

“I have no answer for that” (“No tengo respuesta para eso”).

El lenguaje tiene definida una regla por defecto (UDC, Ultimate Default Category) a la que recurre si la entrada no se adapta con ningún **<pattern>**.

¿Cómo funciona el proceso de encontrar la regla (<category> en AIML) que se dispara ante una entrada?

El proceso consiste tomar la entrada proporcionada por el usuario y buscar las reglas cuyo patrón se ajuste a esa entrada.

Si hay una única regla que encaje, esa es la que se dispara.

Si hay más de una regla, se dispara la de mayor prioridad (más adelante hablaremos sobre esto).

Si no hay ninguna regla adecuada para la entrada, se disparará la regla por defecto: “I have no answer for that”.

Ejercicio 1:

Añada en primero.aiml las reglas necesarias para mantener la siguiente conversación con el bot:

Human: Hola!

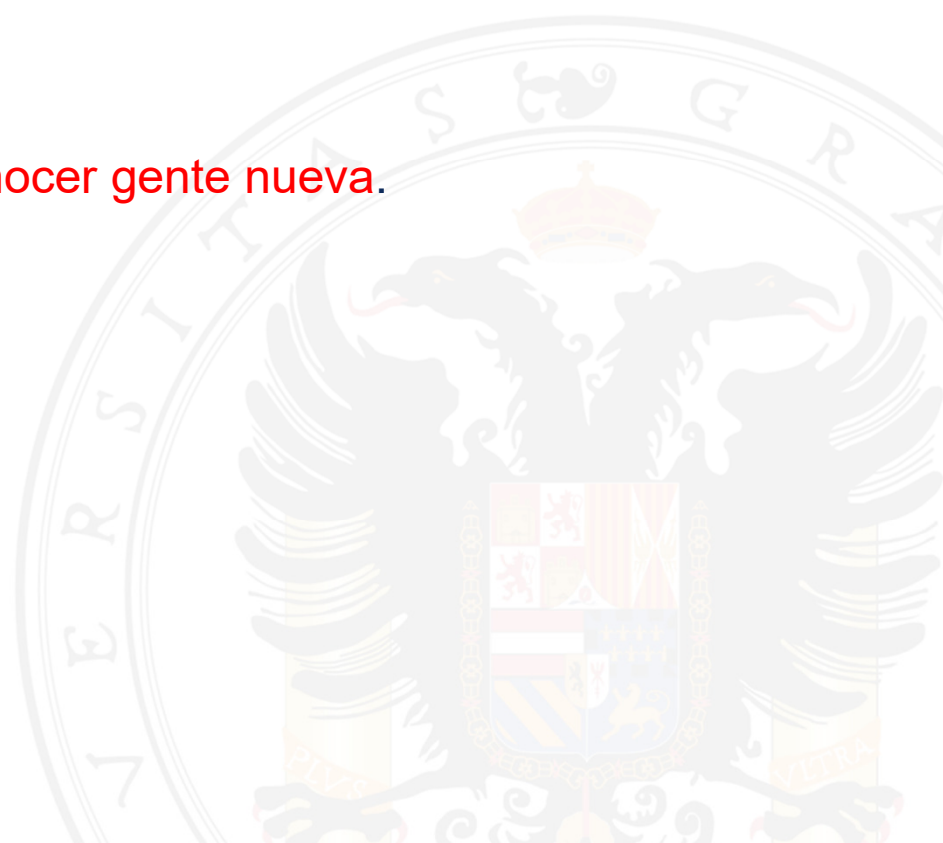
Robot: Hola, que tal?

Human: Yo bien, que tal tu?

Robot: Estoy genial!!! Me encanta conocer gente nueva.

Human: Genial!! Como te llamas?

Robot: Mi nombre es HALfonso



Resolución Ejercicio 1:

Human: Hola

Robot: Hola, que tal?

Human: Yo bien, que tal tu?

Robot: Estoy genial!!!

Human: Como te llamas?

Robot: Mi nombre es HALfonso



```
<!-- Primera regla -->
<category>
<pattern>Hola</pattern>
<template>Hola, que tal?</template>
</category>
```

Resolución Ejercicio 1:

Human: Hola

Robot: Hola, que tal?

Human: Yo bien, que tal tu?

Robot: Estoy genial!!!

Human: Como te llamas?

Robot: Mi nombre es HALfonso



```

<!-- Primera regla -->
<category>
<pattern>Hola</pattern>
<template>Hola, que tal?</template>
</category>

<!-- Segunda regla -->
<category>
<pattern>yo bien, que tal tu</pattern>
<template>Estoy genial!!!</template>
</category>
  
```

Resolución Ejercicio 1:

Human: Hola

Robot: Hola, que tal?

Human: Yo bien, que tal tu?

Robot: Estoy genial!!!

Human: Como te llamas?

Robot: Mi nombre es HALfonso

<!-- Primera regla -->

<category>

<pattern>Hola</pattern>

<template>Hola, que tal?</template>

</category>

<!-- Segunda regla -->

<category>

<pattern>yo bien, que tal tu</pattern>

<template>Estoy genial!!!</template>

</category>

<!-- Tercera regla -->

<category>

<pattern>como te llamas</pattern>

<template>

Mi nombre es HALfonso

</template>

</category>

```

primero.aiml - Code::Blocks 13.12
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Pl
...
Start here X primero.aiml X primero.aiml X
1  <?xml version="1.0" encoding="UTF-8"?>
2  <aiml version="2.0">
3
4
5  <!-- regla 1 -->
6  <category>
7  <pattern>Hola</pattern>
8  <template>Hola, que tal?</template>
9  </category>
10
11 <!-- regla 2 -->
12 <category>
13 <pattern>yo bien, que tal tu</pattern>
14 <template>Estoy genial!!!</template>
15 </category>
16
17 <!-- regla 3 -->
18 <category>
19 <pattern> Como te llamas</pattern>
20 <template> Mi nombre es HALfonso</template>
21 </category>
22
23
24 </aiml>
25

```

El lenguaje AIML

Wildcards
“comodines”



Los wildcards o “comodines” permiten capturar varias entradas para una misma regla (o categoría).

Hay varios comodines:

- El comodín “*”: captura **una o más palabras** de la entrada

<pattern>Hola *</pattern>

Captura entradas como:

Hola amigo

Hola, estoy aquí de nuevo

Hola Arturo

- El comodín “^”: captura **cero o más palabras** de la entrada

<pattern>Hola ^</pattern>

Captura entradas como:

Hola

Hola, estoy aquí de nuevo

Hola Arturo

¿Qué ocurre si existen los dos siguientes patrones?

<pattern>Hola *</pattern>

<pattern>Hola ^</pattern>

El patrón que contiene “^” tiene mayor prioridad y, por consiguiente, será esta la regla con “^” la que se dispare.

En este caso concreto, el patrón con el “*” no se disparará nunca si existe el otro patrón.

¿y qué ocurre si aparecen los siguientes 3 patrones

<pattern>Hola *</pattern>

<pattern>Hola ^</pattern>

<pattern>Hola amigo</pattern>

ante la entrada “Hola amigo”?

En este caso, el emparejamiento exacto tiene mayor prioridad que “^” (y, por tanto, que “*”).

Hay otros dos “comodines”

- El comodín “_”: captura **una o más palabras** de la entrada (como el *)

```
<pattern>Hola _</pattern>
```

- El comodín “#”: captura **cero o más palabras** de la entrada (como el ^)

```
<pattern>Hola #</pattern>
```

La única diferencia con los anteriores es la prioridad.

El orden de prioridad de mayor a menor es el siguiente:

Hola # > Hola _ > Hola amigo > Hola ^ > Hola *

A veces, se desea definir un patrón que tenga mayor prioridad que “#” o “_”:
Para esos casos, se recurre al símbolo “\$”, que indica que ese patrón tiene la mayor prioridad si el texto contiene la palabra exacta a la que acompaña “\$”.

```
<pattern>$Quien * Luis</pattern>
```

```
<pattern>_ Luis </pattern>
```

En este ejemplo, si en la entrada aparece “Quien”,
el primer patrón tiene prioridad sobre el segundo.

\$ no es un comodín, es sólo un marcador de prioridad.

Los comodines pueden ser capturados dentro del **<template>** usando **<star/>**.

```
<category>
  <pattern>Mi nombre es *</pattern>
  <template>Hola <star/></template>
</category>
```

Human: Mi nombre es Rocio

Robot: **Hola Rocio**

Cuando hay más de un comodín se hace uso de **<star index="x"/>**, dónde x indica la posición que ocupa el comodín desde el principio del patrón.

```
<category>
  <pattern>Estudio * en *</pattern>
  <template>En <star index="2"/>, yo también estudio <star/></template>
</category>
```

Human: Estudio Informatica en Granada

Robot: **En Granada, yo también estudio Informatica**

Ejercicio 2:

Modifique o añada reglas con comodines a las reglas del ejercicio 1 para que pueda seguir la siguiente conversación, donde “...” representa que al menos aparece una palabra más en la entrada y “..*..” representa que los comodines están vinculados.

Human: Hola ...

Robot: Hola, que tal?

Human: Hola

Robot: Hola de nuevo, que tal?

Human: ... que tal tu?

Robot: Estoy genial!!!

Human: Fenomeno, me llamo ..*..

Robot: Que casualidad ..*.. yo tambien tengo nombre, me llamo HALberto

Resolución Ejercicio 2:

```

primero.aiml - Code::Blocks 13.12
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help

Start here x primero.aiml x
1  <?xml version="1.0" encoding="UTF-8"?>
2  <aiml version="2.0">
3
4
5  <!-- regla 1 -->
6  <category>
7    <pattern>Hola */</pattern>
8    <template>Hola, que tal?</template>
9  </category>
10
11 <!-- regla 2 -->
12 <category>
13   <pattern>^ que tal tu</pattern>
14   <template>Estoy genial!!!</template>
15 </category>
16
17 <!-- regla 3 -->
18 <category>
19   <pattern> ^ me llamo */</pattern>
20   <template> Que casualidad <star index="2"/>, yo tambien tengo nombre, me llamo HALfonso</
21 </category>
22
23 <!-- regla 4 -->
24 <category>
25   <pattern>Hola</pattern>
26   <template>Hola de nuevo, que tal?</template>
27 </category>
28
29
30 </aiml>

```

El lenguaje AIML

Variables



En AIML, existen 3 tipos de variables:

1. Propiedades del bot

Definen los datos que el bot puede proporcionar sobre sí mismo y sólo puede ser asignadas por el botmaster (el creador del bot).

2. Predicados o variables globales

En el lenguaje se denominan predicados, pero no tienen ningún tipo de asociación con el concepto de predicado en lógica. En realidad, son variables globales. En AIML, una variable es global cuando su valor puede ser consultado o modificado fuera de una regla (categoría).

3. Variables locales

Como su nombre indica, son variables cuyo ámbito es local a una regla (categoría) y su valor no puede ser consultado fuera de dicha regla.

Propiedades del bot

Las propiedades del bot vienen definidas en el fichero “**properties.txt**” de la carpeta de configuración del bot: “**program-ab/bots/.../config/**”

Por defecto, vienen definidas las variables:

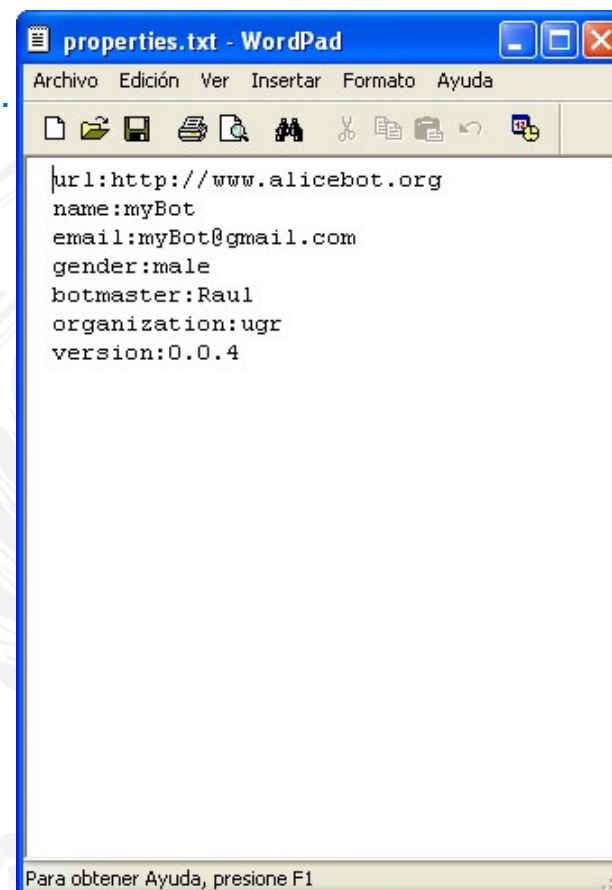
url, name, email, gender, botmaster, organization, version.

Se pueden añadir nuevas variables siguiendo el siguiente formato:

<nombre variable>:<valor>

Por ejemplo, añadamos los siguiente:

age:20
job:estudiante



```
properties.txt - WordPad
Archivo  Edición  Ver  Insertar  Formato  Ayuda
url:http://www.alicebot.org
name:myBot
email:myBot@gmail.com
gender:male
botmaster:Raul
organization:ugr
version:0.0.4
```

Propiedades del bot

Las propiedades del bot vienen definidas en el fichero “**properties.txt**” de la carpeta de configuración del bot: “**program-ab/bots/.../config/**”

Por defecto, vienen definidas las variables:

url, name, email, gender, botmaster, organization, version.

Se pueden añadir nuevas variables siguiendo el siguiente formato:

<nombre variable>:<valor>

Por ejemplo, añadamos los siguiente:

age:20
job:estudiante



```
url:http://www.alicebot.org
name:myBot
email:myBot@gmail.com
gender:male
botmaster:Raul
organization:ugr
version:0.0.4
age:20
job:estudiante
```

Uso de las propiedades del bot

La sintaxis es que se desea.

```
<bot name="x"/>
```

donde x representa la propiedad

```
<category>
```

```
<pattern>Cual es tu edad</pattern>
```

```
<template>Tengo <bot name="age"/> años</template>
```

```
</category>
```

Human: Cual es tu edad?

Robot: Tengo 20 años

primero.aiml

```
29 <!-- regla 5 -->
30 <category>
31 <pattern>Cual es tu edad</pattern>
32 <template>Tengo <bot name="age"/> años</template>
33 </category>
34
```

Predicados o variables globales

La sintaxis para asignarle un valor determinado es donde x representa el nombre de la variable.

```
<set name="x">value</set>
```

Hay que tener en cuenta que AIML no tiene declaración de variables, así que hay que tener cuidado con las referencias a los nombres de variables.

La sintaxis para acceder al valor de una variable global es donde x representa el nombre de la variable.

```
<get name="x"/>
```

Las variables globales tienen sentido cuando el valor va a ser usado en varias reglas. Si no es así, las variables que deben utilizarse son las locales.

Predicados o variables globales

Modifiquemos la regla 3 del fichero *primero.aiml*

La versión original es:

```

17 <!-- regla 3 -->
18 <category>
19 <pattern> ^ me llamo */pattern>
20 <template> Que casualidad <star index="2"/>, yo tambien tengo nombre, me llamo HALfonso</
21 </category>

```

Y la cambiamos por la siguiente, para almacenar el nombre del usuario:

```

17 <!-- regla 3 -->
18 <category>
19 <pattern> ^ me llamo */pattern>
20 <template>
21 <set name="nombre"><star index="2"/></set>
22 <get name="nombre"/> es un bonito nombre.
23 </template>
24 </category>
25

```

Predicados o variables globales

Incluyamos una nueva regla en *primero.aiml* para devolver el nombre del usuario:

```

38 <!-- regla 6 -->
39 <category>
40 <pattern>Cual es mi nombre</pattern>
41 <template>Tu nombre es <get name="nombre"/></template>
42 </category>
43

```

Y ahora probemos la siguiente secuencia.

Human: Cual es mi nombre?
 Robot: **Tu nombre es unknown**
 Human: Me llamo Raul
 Robot: **Raul es un bonito nombre.**
 Human: Cual es mi nombre?
 Robot: **Tu nombre es Raul**

Una variable a la que se accede sin haberle asignado un valor previamente devuelve siempre como valor "unknown"

Variables locales

La sintaxis para establecer el valor de una variable local es

```
<set var="x">value</set>
```

donde x representa el nombre de la variable.

La sintaxis para acceder al valor de una variable local es

```
<get var="x"/>
```

donde x representa el nombre de la variable.

Las variables locales tienen como ámbito el **template** de la regla, a diferencia de las variables globales.

```
44 <!-- regla 7 -->
45 <category>
46 <pattern>mi color favorito es el *</pattern>
47 <template>
48   <set var="color"><star/></set>
49   El <get var="color"/> es un color que no me gusta.
50 </template>
51 </category>
```

El tag <think>

Tanto el acceso como la asignación de una variable provoca “eco” por pantalla. Así, si en el intérprete ponemos:

Human: Mi color favorito es el amarillo

Robot: amarillo

El amarillo es un color que no me gusta.

Para evitar ese “eco”, las asignaciones y acceso se encierran en un par <think></think>

```
44 <!-- regla 7 -->
45 <category>
46 <pattern>mi color favorito es el *</pattern>
47 <template>
48     <set var="color"><star/></set>
49     El <get var="color"/> es un color que no me gusta.
50 </template>
51 </category>
```

```
44 <!-- regla 7 -->
45 <category>
46 <pattern>mi color favorito es el *</pattern>
47 <template>
48     <think><set var="color"><star/></set></think>
49     El <get var="color"/> es un color que no me gusta.
50 </template>
51 </category>
```

El lenguaje AIML

Reducción
Simbólica
<srai>



Una herramienta muy importante en AIML es la reducción simbólica, ya que permite:

- Simplificar las entradas usando pocas palabras
- Enlazar distintas entradas sinónimas con un mismo *template*
- Corregir errores ortográficos por parte del usuario
- Reemplazar expresiones coloquiales por expresiones formales
- Eliminar palabras innecesarias en las entradas

En realidad, la reducción simbólica es una invocación recursiva a la propia base de conocimiento (conjunto de categorías) y nos permite reducir el tamaño de la base conocimiento.

El tag asociado a la reducción simbólica es donde ... representa un patrón de búsqueda.

```
<srai>...</srai>
```

En nuestro fichero “primero.aiml”, tenemos la regla 6 que responde a la pregunta “Cual es mi nombre?”, pero hay muchas formas de expresarlo, como “Dime mi nombre?”, “Como me llamo?”, “Te acuerdas de cómo me llamo?” o “Sabrías decir mi nombre?”, para las cuales la respuesta es la misma que ofrece la regla 6.

Estas expresiones las podemos agrupar en aquéllas que terminan por “mi nombre” y las que terminan con “como me llamo”. De esta forma, podemos construir dos reglas nuevas con estos patrones cuyo *template* invoque a la regla 6:

```
<category>
<pattern>* mi nombre</pattern>
<template><srai>CUAL ES MI NOMBRE</srai></template>
</category>

<category>
<pattern>^ como me llamo</pattern>
<template><srai>CUAL ES MI NOMBRE</srai></template>
</category>
```

Añadimos estas dos nuevas reglas a nuestro fichero.

```

38 <!-- regla 6 -->
39 <category>
40 <pattern>Cual es mi nombre</pattern>
41 <template>Tu nombre es <get name="nombre"/></template>
42 </category>
43
44 <!-- regla 7 -->
45 <category>
46 <pattern>mi color favorito es el *</pattern>
47 <template>
48   <think><set var="color"><star/></set></think>
49   El <get var="color"/> es un color que no me gusta.
50 </template>
51 </category>
52
53 <!-- regla 8 -->
54 <category>
55 <pattern>* mi nombre</pattern>
56 <template><srai>CUAL ES MI NOMBRE</srai></template>
57 </category>
58
59 <!-- regla 9 -->
60 <category>
61 <pattern>^ como me llamo</pattern>
62 <template><srai>CUAL ES MI NOMBRE</srai></template>
63 </category>

```

Human: me llamo Raul
 Robot: **Raul es un bonito nombre ...**
 Human: como me llamo?
 Robot: **Tu nombre es Raul**
 Human: sabes mi nombre?
 Robot: **Tu nombre es Raul**
 Human: mi nombre?
 Robot: **I have no answer for that**

Ejercicio 3:

Aplique la reducción simbólica sobre la base de reglas actual de primero.aiml para contemplar los saludos más habituales.



El lenguaje AIML

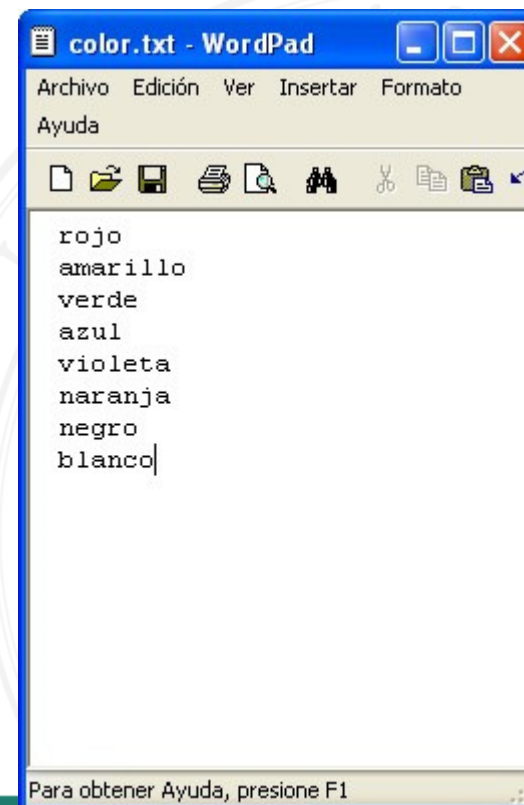
Sets y Maps



Una de las aportaciones relevantes que se incluyen en la versión 2.0 de AIML es el uso de Sets y Maps.

Un set es una lista de cadenas que se almacena sobre un fichero. El nombre del fichero da la denominación del set y la extensión debe ser “txt”. Este fichero debe ubicarse en la carpeta “sets”.

En la siguiente figura se muestra un ejemplo de set “color” donde se ilustra la sintaxis que debe tener el fichero, con un valor en cada línea del archivo txt.



Los sets permiten reducir el número de reglas necesario.
Por ejemplo, si defino un set con la lista de colores, con dos únicas reglas puedo determinar si algo es un color o no.

```
<category>
<pattern>Es <set>color</set> un color</pattern>
<template>Si, <star/> es un color </template>
</category>

<category>
<pattern> Es * un color</pattern>
<template>No, <star/> no es un color</template>
</category>
```

La secuencia <set>color</set> en el patrón verifica si la entrada coincide con alguna de las palabras que aparecen en el fichero “color.txt”. Si es así, la regla se dispara. En otro caso, será la segunda regla la que se dispare.

<set> tiene mayor prioridad que “*” y “^”, pero menos que “_” y “#”.

Si añadimos estas reglas a nuestro fichero **primero.aiml**, podemos realizarle las siguientes preguntas a nuestro bot:

```

65 <!-- regla 10 -->
66 <category>
67 <pattern>es <set>color</set> un color</pattern>
68 <template>Si, <star/> es un color.</template>
69 </category>
70
71
72 <!-- regla 11 -->
73 <category>
74 <pattern>es * un color</pattern>
75 <template>No, <star/> no es un color.</template>
76 </category>

```

Human: Es amarillo un color?

Robot: Si, amarillo es un color.

Human: Es rojo un color?

Robot: Si, rojo es un color.

Human: Es verde un color?

Robot: Si, verde es un color.

Human: Es lapiz un color?

Robot: No, lapiz no es un color.

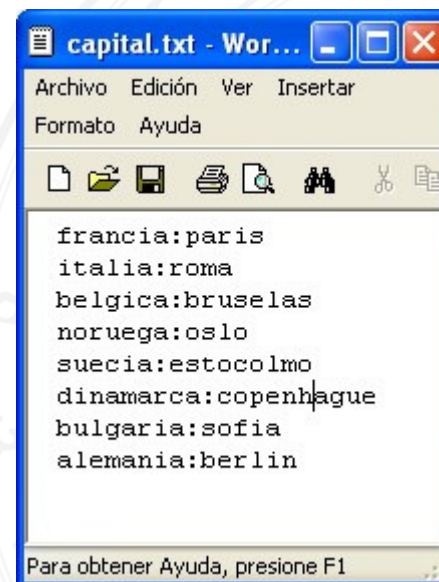
Los **maps** corresponden al tipo de dato diccionario y, al igual que los **sets**, se especifican en un fichero de texto independiente, cuyo nombre identifica al **map** y extensión ha de ser “txt”.

Este fichero debe estar alojado en la carpeta **maps**.

En cada línea del fichero se codifica una entrada del diccionario utilizando la siguiente sintaxis:

```
cadena1:cadena2
```

Como ejemplo, definamos un **map** de nombre “**capital**” para asociar a cada país el nombre de su capital.



Con la siguiente regla podemos hacer que el bot sepa responder a preguntas por la capital de un país:

```
<category>
<pattern>Cual es la capital de *</pattern>
<template>
  La capital de <star/> es <map name="capital"><star/></map>.
</template>
</category>
```

<map name="capital">KEY</map>
devuelve el valor asociado a la clave **KEY**.

Añadamos esta regla al fichero primero.aiml:

```

79 <!-- regla 12 -->
80 <category>
81 <pattern>Cual es la capital de */</pattern>
82 <template>La capital de <star/> es <map name="capital"><star/></map>.</template>
83 </category>
84

```

Human: Cual es la capital de francia?

Robot: La capital de francia es paris.

Human: Cual es la capital de italia?

Robot: La capital de italia es roma.

Human: Cual es la capital de Cuba?

Robot: I have no answer for that



Se suele definir un set con las claves de cada *map*.
En nuestro ejemplo, definiendo un set de países podemos contemplar, a través de 2 reglas, si conocemos o no la capital de un determinado país:

```

79 <!-- regla 12 -->
80 <category>
81 <pattern>Cual es la capital de <set>pais</set></pattern>
82 <template>La capital de <star/> es <map name="capital"><star/></map>.</template>
83 </category>
84
85 <!-- regla 13 -->
86 <category>
87 <pattern>Cual es la capital de *</pattern>
88 <template>No se cual es la capital de <star/>.</template>
89 </category>

```

Human: Cual es la capital de francia?

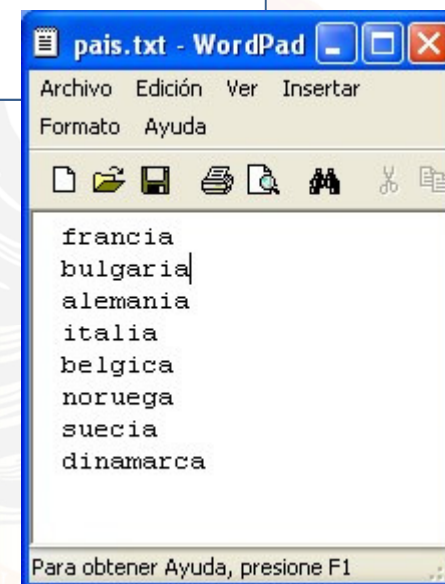
Robot: La capital de francia es paris.

Human: Cual es la capital de italia?

Robot: La capital de italia es roma.

Human: Cual es la capital de Cuba?

Robot: No se cual es la capital de Cuba



AIML tiene implícitamente definidos los siguientes sets y maps:

- `<set>number</set>`
Números naturales
- `<map name="successor">`
Dado un número natural "n" devuelve "n+1"
- `<map name="predecessor">`
Dado un número natural "n" devuelve "n-1"
- `<map name="plural">`
Devuelve el plural de un palabra en singular (sólo inglés)
- `<map name="singular">`
Devuelve el singular de un palabra en plural (sólo inglés)

Ejercicio 4:

Construya un fichero set, llamado “compi.txt” que contenga al menos el nombre de 5 compañeros de clase, y defina 2 ficheros map, uno que asocie a cada uno de ellos su color de pelo (“pelo.txt”) y otro que le asocie su color de ojos (“ojos.txt”).

Una vez hecho esto, elabore un fichero llamado “ejer4.aiml” y defina el conjunto de reglas necesario para responder a preguntas sobre el color de ojos y de pelo de sus compañeros.

El lenguaje AIML

Contexto



El contexto

es fundamental para que una conversación mantenga su coherencia y nos permite recordar cosas que se han dicho previamente.

En AIML, hay 3 elementos para recordar el contexto:

- Los predicados o variables globales (ya vistas previamente)
- El tag <that>
- Un set predefinido en el lenguaje llamado “topic”

El tag <that>

El bot recuerda la última respuesta que dio el usuario de forma que, a partir de esa respuesta, puede alterar la respuesta a su siguiente pregunta.

El tag <that> se sitúa entre <pattern> y <template> siendo su sintaxis la siguiente:

Regla 1

```
<category>
<pattern>Si</pattern>
<that> TE GUSTA EL CAFE </that>
<template>
  Lo prefieres solo o con leche
</template>
</category>
```

Regla 2

```
<category>
<pattern>^ cafe ^</pattern>
<template>
  Te gusta el cafe
</template>
</category>
```

Obviamente, para que en una conversación se dispare la Regla 1, es necesario que justo antes se haya disparado una regla como la Regla 2.

El tag <that>

Añadamos estas dos reglas al fichero primero.aiml.

```

92 <!-- regla 14 -->
93 <category>
94 <pattern>^ cafe ^</pattern>
95 <template>Te gusta el cafe.</template>
96 </category>
97
98
99 <!-- regla 15 -->
00 <category>
01 <pattern>Si</pattern>
02 <that>TE GUSTA EL CAFE</that>
03 <template>Lo prefieres solo o con leche.</template>
04 </category>

```

Human: esta mañana me tome un cafe

Robot: Te gusta el cafe.

Human: Si

Robot: Lo prefieres solo o con leche.

<set name="topic"> & <topic name="x"></topic>

Esta variable global predefinida en el lenguaje permite agrupar las reglas, que manera que estas sólo se activen cuando la conversación se centra en un tema concreto.

Por defecto, el valor de "topic" es "unknown" (como cualquier otra variable).

Esta variable tiene dos usos:

- Como una variable más ,indicando su valor dentro del template de una regla.
- Encerrando entre <topic name="x"> ... </topic> las reglas asociadas a ese tema.

<template> te gusta el <set name="topic"> cafe </set></template>

```
<topic name="cafe">
<category> ..... </category>
.....
<category> ..... </category>
</topic>
```

El lenguaje AIML

Random,
estructuras
condicionales y
bucles



No responder exactamente de la misma forma ante la misma pregunta o ante preguntas similares ayuda al bot a dar la impresión de presentar un comportamiento más semejante al de un humano.

El lenguaje AIML tiene el tag `<random>` para conseguir este comportamiento. Su sintaxis es la siguiente:

```
<random>
  <li> .... </li>
  <li> ... </li>
  .....
  <li> ... </li>
</random>
```

Ejemplo

```
<category>
  <pattern>hola *</pattern>
  <template>
    <random>
      <li> Hola! </li>
      <li> Buenas! Qué tal? </li>
    </random>
  </template>
</category>
```

Las etiquetas `...` delimitan las distintas salidas posibles. Aleatoriamente elige entre uno de los elementos `` como respuesta.

En AIML también existe una estructura condicional.

Esta estructura condicional funciona como el “switch” de C.
Su sintaxis es la siguiente:

Para variables locales

```
<condition var =“x”>
<li value=“x1”> .... </li>
<li value=“x2”> ... </li>
.....
<li> ... </li>
</condition>
```

Para variables globales

```
<condition name =“x”>
<li value=“x1”> .... </li>
<li value=“x2”> ... </li>
.....
<li> ... </li>
</condition>
```

Las etiquetas <li value> separan los distintos casos y el último se aplica cuando ninguno de los casos anteriores se cumple.

Veamos un ejemplo de su uso...

En nuestro fichero primero.aiml, nos dimos cuenta que si le pedíamos nuestro nombre en la regla 6 y aún no lo habíamos almacenado, ya que no se había invocado a la regla 4, el bot nos decía “**Tu nombre es unknown**”.

Corrijamos la regla 6 de la siguiente manera: si ya se ha asignado valor a la variable “name”, entonces que funcione como está ahora mismo, pero si no, que diga que aún no le hemos dicho nuestro nombre.

Regla Original

```

38 <!-- regla 6 -->
39 <category>
40 <pattern>Cual es mi nombre</pattern>
41 <template>Tu nombre es <get name="nombre"/></template>
42 </category>
43

```

Regla Modificada

```

38 <!-- regla 6 -->
39 <category>
40 <pattern>Cual es mi nombre</pattern>
41 <template>
42 <condition name="nombre">
43 <li value="unknown"> Aun no me has dicho tu nombre</li>
44 <li>Tu nombre es <get name="nombre"/></li>
45 </condition>
46 </template>
47 </category>

```

Human: como me llamo?

Robot: Aun no me has dicho tu nombre

Human: me llamo Raul

Robot: Raul es un bonito nombre...

Human: como me llamo?

Robot: Tu nombre es Raul

Otro elemento básico en un lenguaje de programación son los bucles.

AIML tiene una forma muy peculiar para la construcción de bucles.

Son bucles del tipo “*mientras <condición> hacer <bloque de operaciones>*” y eso implica el uso de una condición.

Veámoslo con un ejemplo:

Supongamos que queremos construir una regla que cuente hasta un determinado número.

```
<category>
<pattern>Cuenta hasta <set>number</set></pattern>
<template>

</template>
</category>
```

```
<category>  
<pattern>Cuenta hasta <set>number</set></pattern>  
<template>  
</template>  
</category>
```

Planteo la regla y defino como patrón responder a las consultas que son de la forma “Cuenta hasta n” siendo n un número.


```
<category>
<pattern>Cuenta hasta <set>number</set></pattern>
<template>
  <think>
    <set var="contador">1</set>
    <set var="salida">1</set>
  </think>
</template>
</category>
```

Es un bucle con contador lo que tengo que hacer, así que declaro una variable local “contador” para ir almacenando los distintos valores hasta llegar a number.

Además, declaro otra variable local “salida”, que va a ir almacenando la secuencia de números por los que va pasando contador. La idea es que salida se comporte como una cadena de caracteres.

Ambas variables se inicializan con el valor 1 y están incluidas en un bloque <think> para que no produzcan “eco” por pantalla.

```
<category>
<pattern>Cuenta hasta <set>number</set></pattern>
<template>
  <think>
    <set var="contador">1</set>
    <set var="salida">1</set>
  </think>
  <condition var="contador">
    <li><value><star/></value><get var="salida"/></li>
    <li></li>
  </condition>
</template>
</category>
```

Ahora planteo un comportamiento diferente en función de una condición.

La condición revisa el valor de la variable "contador".

El primer caso nos dice que, si el valor de comodín (en este caso, es el número introducido en el patrón) coincide con el valor de la variable "contador", entonces devuelva por el terminal el valor de la variable salida.

SINTAXIS: Hasta ahora habíamos visto <li value="x"> pero, cuando hay un valor que no podemos encerrar entre comillas, se puede descomponer de la forma que aquí aparece:
<value>valor</value>.....

```

<category>
<pattern>Cuenta hasta <set>number</set></pattern>
<template>
  <think>
    <set var="contador">1</set>
    <set var="salida">1</set>
  </think>
  <condition var="contador">
    <li><value><star/></value><get var="salida"/></li>
    <li>
      <think>
        <set var="contador">
          <map name="successor"><get var="contador"/></map>
        </set>
      </think>
    </li>
  </condition>
</template>
</category>

```

El caso anterior corresponde a la condición de salida del bucle. Así que, este segundo caso correspondería al cuerpo del bucle en un lenguaje de programación convencional. En este caso, lo que tenemos que hacer es incrementar el contador. AIML nos permite hacer esto mediante el map "successor".

Este grupo de acciones lo que hace es calcular el sucesor del valor actual de "contador" y el resultado de ese cálculo lo vuelve a almacenar en la variable "contador".

```

<category>
<pattern>Cuenta hasta <set>number</set></pattern>
<template>
  <think>
    <set var="contador">1</set>
    <set var="salida">1</set>
  </think>
  <condition var="contador">
    <li><value><star/></value><get var="salida"/></li>
    <li>
      <think>
        <set var="contador">
          <map name="successor"><get var="contador"/></map>
        </set>
        <set var="salida"><get var="salida"/> <get var="contador"/></set>
      </think>
    </li>
  </condition>
</template>
</category>

```

Además, de incrementar “contador”, actualizamos la variable “salida”, en este caso, mediante una concatenación de cadenas.

Se puede observar que el nuevo valor de “salida” es su anterior valor seguido del valor de la variable “contador”. Es importante que haya un espacio en blanco entre el get de “salida” y el get de “contador” para separar los números.

```

<category>
<pattern>Cuenta hasta <set>number</set></pattern>
<template>
  <think>
    <set var="contador">1</set>
    <set var="salida">1</set>
  </think>
  <condition var="contador">
    <li><value><star/></value><get var="salida"/></li>
    <li>
      <think>
        <set var="contador">
          <map name="successor"><get var="contador"/></map>
        </set>
        <set var="salida"><get var="salida"/> <get var="contador"/></set>
      </think>
      <loop/>
    </li>
  </condition>
</template>
</category>

```

Terminamos el cuerpo del bucle con el comando `<loop/>`, que indica que se revise el valor de la última condición examinada (en nuestro caso, la única condición que hay).

Si hubiera varias condiciones anidadas, `<loop/>` implica siempre a la más interna.

Así quedaría la regla al final:

```

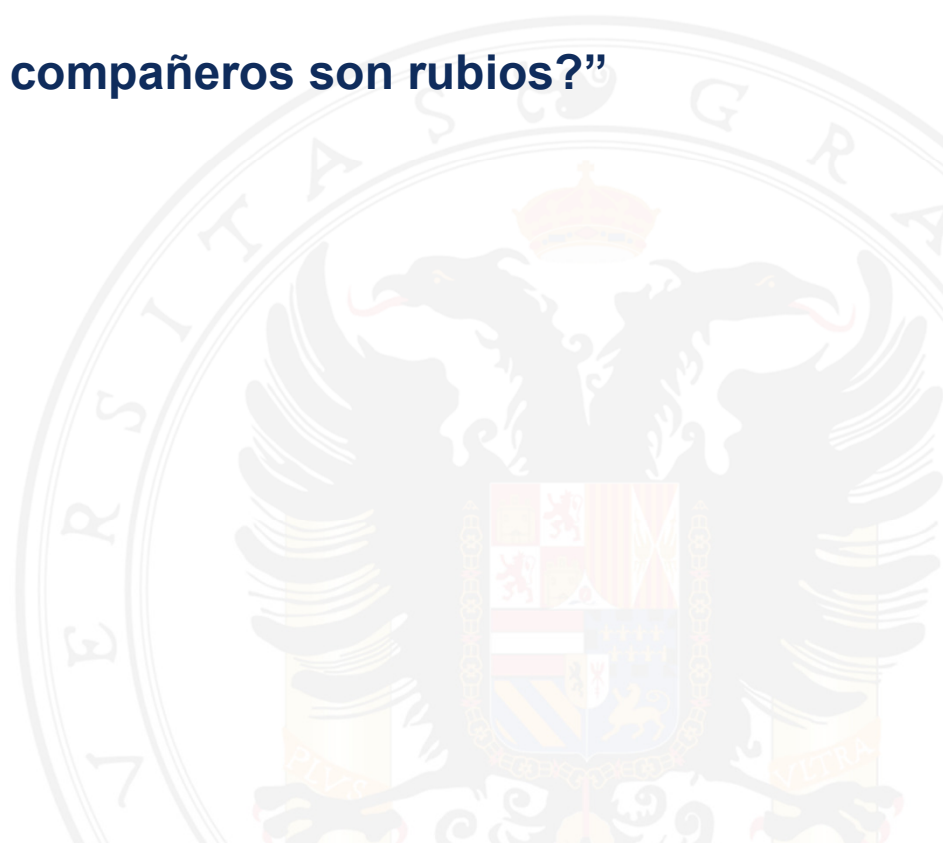
08 <!-- regla 16 -->
09 <category>
10 <pattern>Cuenta hasta <set>number</set></pattern>
11 <template>
12 <think>
13 <set var="contador">1</set>
14 <set var="salida">1</set>
15 </think>
16 <condition var="contador">
17 <li><value><star/></value><get var="salida"/></li>
18 <li>
19 <think>
20 <set var="contador">
21 <map name="successor"><get var="contador"/></map>
22 </set>
23 <set var="salida"><get var="salida"/> <get var="contador"/></set>
24 </think>
25 <loop/>
26 </li>
27 </condition>
28 </template>
29 </category>

```

Ejercicio 5:

Utilice el conocimiento desarrollado en el ejercicio 4 acerca del color de ojos y pelo de algunos compañeros y añada las reglas necesarias para que, dado un color concreto de pelo o de ojos, el bot devuelva los nombres de los compañeros tienen esa característica física.

La pregunta puede ser del tipo “Que compañeros son rubios?”



El lenguaje AIML

Aprender



Una de las características más importante de AIML es que permite al bot de aprender de los usuarios.

Para eso se usan dos tags: **<learn>** y **<learnf>**.

Ambos se usan de la misma manera. La única diferencia entre ambos es si lo que aprende sólo se usa en la conversación actual (en ese caso, se usa **<learn>**) o si lo aprendido se desea que se mantenga como parte de la base de conocimiento del bot (en cuyo caso hay que usar **<learnf>**).

Aquí explicaremos el uso de **<learn>**, asumiendo que se hace exactamente igual con **<learnf>**.

Una aclaración: cuando se usa **<learnf>** el botmaster pierde el control del bot y éste puede que este aprenda “cosas malas” ;-)

Aprenderemos el uso de **<learn>** con un ejemplo:

Las reglas 12 y 13 del fichero primero.aiml permiten al bot responder cuál es la capital de un determinado país. La 12 en el caso afirmativo de que el país ya esté en el **set pais** y la 13 indicando que no lo sabemos.

Complementemos estas dos reglas con una regla adicional que nos permita, si no conocemos el país o su capital, aprenderlo.

En concreto, dada la entrada “La capital de * es *”, haremos lo siguiente:

1. Verificar si el país está en set “pais”
(en cuyo caso invocamos a la regla 12).
2. Verificar si es una capital que ya habíamos aprendido antes.
3. En otro caso, aprendemos algo nuevo...

Las reglas 12 y 13 son las siguientes:

```

79 <!-- regla 12 -->
80 <category>
81 <pattern>Cual es la capital de <set>pais</set></pattern>
82 <template>La capital de <star/> es <map name="capital"><star/></map>.</template>
83 </category>
84
85 <!-- regla 13 -->
86 <category>
87 <pattern>Cual es la capital de *</pattern>
88 <template>No se cual es la capital de <star/>.</template>
89 </category>

```

0. Transformamos el **template** de la regla 13 para que devuelva “No lo se” (simplemente, por comodidad ahora mismo):

```

84 <!-- regla 12 -->
85 <category>
86 <pattern>Cual es la capital de <set>pais</set></pattern>
87 <template>La capital de <star/> es <map name="capital"><star/></map>.</template>
88 </category>
89
90 <!-- regla 13 -->
91 <category>
92 <pattern>Cual es la capital de *</pattern>
93 <template>No lo se</template>
94 </category>

```

<category>

<pattern>la capital de * es *</pattern>

<template>

<template>

</category>

Proponemos la estructura básica de la regla y fijamos el patrón con dos comodines, el primero recoge para el país y el segundo para su capital.

```
<category>
<pattern>la capital de * es *</pattern>
<template>
  <think>
    <set var="cap"><srai>CUAL ES LA CAPITAL DE <star/></srai></set>
  </think>
</template>
</category>
```

Añadimos en el **template** una invocación a la regla 12 o 13. Esto devolverá "NO LO SE" si se dispara la regla 13.

En otro caso, devuelve "LA CAPITAL DE ...".

El valor devuelto se almacena en la variable local "cap" y está incrustado en un bloque <think> para que no muestre "eco" por pantalla.

```

<category>
<pattern>la capital de * es *</pattern>
<template>
  <think>
    <set var="cap"><srai>CUAL ES LA CAPITAL DE <star/></srai></set>
  </think>
  <condition var="cap">
    <li value="NO LO SE">
    </li>
  </condition>
</template>
</category>

```

Ahora especificamos una condición para determinar si el bot ya sabe la respuesta.

Si no lo sabe, la variable “cap” contendrá “NO LO SE”. En tal caso, queremos aprender del usuario.


```

<category>
<pattern>la capital de * es *</pattern>
<template>
  <think>
    <set var="cap"><srai>CUAL ES LA CAPITAL DE <star/></srai></set>
  </think>
  <condition var="cap">
    <li value="NO LO SE">
      <learn>
        <category>
          <pattern>CUAL ES LA CAPITAL DE <eval><star/></eval></pattern>
          <template>
            La capital de <eval><star/></eval> es <eval><star index="2"/></eval>
          </template>
        </category>
      </learn>
    </li>
  </condition>
</template>
</category>

```

Aquí aparece la sintaxis de `<learn>`:

Como se puede observar, dentro del bloque *learn* aparece incrustada la estructura de una regla con su *category*, su *pattern* y su *template*.

La nueva regla que se propone es que, ante la entrada de "CUAL ES LA CAPITAL DE ...", siendo ... un país concreto, la respuesta será "La capital de .1. es .2., donde .1. es un país concreto y .2. es una ciudad concreta.

El *tag* `<eval>` transforma el comodín por el valor concreto con el que se ha instanciado esa regla. Si no estuviera `<eval>`, no se instanciaría con el valor concreto el comodín, sino que dejaría directamente el comodín.

```

<category>
<pattern>la capital de * es *</pattern>
<template>
  <think>
    <set var="cap"><srai>CUAL ES LA CAPITAL DE <star/></srai></set>
  </think>
  <condition var="cap">
    <li value="NO LO SE">
      <learn>
        <category>
          <pattern>CUAL ES LA CAPITAL DE <eval><star/></eval></pattern>
          <template>
            La capital de <eval><star/></eval> es <eval><star index="2"/></eval>
          </template>
        </category>
      </learn>
      Recordare que la capital de <star/> es <star index="2"/>.
    </li>
  </condition>
</template>
</category>

```

Antes de terminar el caso,
mostramos un mensaje al usuario
indicando que recordaremos lo aprendido.

```

<category>
<pattern>la capital de * es *</pattern>
<template>
  <think>
    <set var="cap"><srai>CUAL ES LA CAPITAL DE <star/></srai></set>
  </think>
  <condition var="cap">
    <li value="NO LO SE">
      <learn>
        <category>
          <pattern>CUAL ES LA CAPITAL DE <eval><star/></eval></pattern>
          <template>
            La capital de <eval><star/></eval> es <eval><star index="2"/></eval>
          </template>
        </category>
      </learn>
      Recordare que la capital de <star/> es <star index="2"/>.
    </li>
    <li>
      Ya lo sabia.
    </li>
  </condition>
</template>
</category>

```

Por último, cubrimos el caso en que la variable "cap" no tenga el valor "NO LO SE". Llegados a esta situación, le decimos al usuario que ya sabíamos lo que nos dice.

Así quedaría la nueva regla.

```

35 <!-- regla 17 -->
36 <category>
37 <pattern>la capital de * es *</pattern>
38 <template>
39   <think>
40     <set var="cap"><srai>CUAL ES LA CAPITAL DE <star/></srai></set>
41   </think>
42   <condition var="cap">
43     <li value="NO LO SE">
44       <learn>
45         <category>
46         <pattern>
47           CUAL ES LA CAPITAL DE <eval><star/></eval>
48         </pattern>
49         <template>
50           La capital de <eval><star/></eval> es <eval><star index="2"/></eval>
51         </template>
52       </category>
53     </learn>
54     Recordare que la capital de <star/> es <star index="2"/>.
55   </li>
56   <li>
57     Ya lo sabia.
58   </li>
59 </condition>
60 </template>
61 </category>

```

Ejercicio 6:

- a) **Corrija la regla 17 para que detecte la no coincidencia entre el nombre de la capital que introduce el usuario con la que el bot tenía almacenada.**
- b) **Tome el conocimiento del ejercicio 5 sobre el color de pelo y de ojos de compañeros y añada las reglas necesarias para que, ante una afirmación del tipo “Laura tiene los ojos azules y el pelo moreno”, aprenda esos datos para responder a las preguntas sobre el color de pelo o de ojos de Laura en el futuro.**

El lenguaje AIML

Algunos enlaces

- [AIML 2.0 Working Draft](#)
- [A.L.I.C.E. The Artificial Intelligence Foundation](#)
- [Pandorabots](#)
- [AIML Quick Guide](#)
- [Build a Simple Virtual Assistant with AIML 2.0](#)