INTELIGENCIA ARTIFICIAL

E.T.S. de Ingenierías Informática y de Telecomunicación

Práctica 3

Agentes conversacionales

(Agenda de contactos y eventos)

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL

UNIVERSIDAD DE GRANADA

Curso 2016-2017

1. Objetivo

El objetivo de la tercera práctica de la asignatura es familiarizarse con el uso del lenguaje AIML. Ese lenguaje ha sido especialmente diseñado para el desarrollo de bases de conocimiento para la construcción de agentes conversacionales.

Un agente conversacional es un agente software, diseñado con el objetivo de mejorar la comunicación entre los seres humanos y las máquinas, haciendo que estas últimas sean capaces de manejar conceptos del lenguaje natural (palabras, frases) como símbolos y reglas que actúan sobre estos símbolos.

Los agentes conversacionales tienen multitud de aplicaciones, siendo su principal labor en estos últimos años la de asistente de los seres humanos. Es muy conocido **SIRI**, el agente conversacional incluido en los dispositivos desarrollados por la empresa ***Apple***. La relevancia de este tipo de agentes se pone de manifiesto en el creciente número de dispositivos y páginas de empresas que los incluyen, como por ejemplo, **GOOGLE NOW** una aplicación de características similares a SIRI desarrollada por ***Google*** para dispositivos Android, o **CORTANA** la aplicación desarrollada por ***Microsoft***, o los asistentes incluidos en las páginas web de empresas muy importantes como ***Ikea***. La propia ***Universidad de Granada*** tiene su asistente llamado **Elvira** que te ayuda a encontrar información relativa a la Universidad.

AIML no es el único lenguaje definido para el desarrollo de agentes conversacionales, ya que podemos encontrar otros como *API.AI* usado para el desarrollo de **Assistant** (una aplicación Android que incorpora una gran flexibilidad debida a su capacidad para aprender), pero sí que es el más estándar (aunque no lo es completamente) y del que es más fácil encontrar información.

Esta tercera práctica incluye un tutorial sobre el uso del lenguaje AIML, que incluye las sentencias más habituales e ilustra su uso con algunos ejemplos y ejercicios. El **objetivo de esta tercera práctica** consiste en definir bases de conocimiento en AIML para establecer algunas conversaciones sobre temas concretos, usando ***program-ab***, un intérprete para el lenguaje ***AIML 2.0*** (una de las últimas versiones del lenguaje[1]) de código abierto y que se puede descargar desde la página web de la asignatura (acceso identificado de decsai.ugr.es).

2. Detalles de la práctica

Como se ha indicado anteriormente, la práctica está orientada al estudio del lenguaje AIML y a mostrar sus posibilidades como medio para desarrollar agentes conversacionales. En concreto, se pide construir un agente conversacional que sea capaz de seguir una conversación sobre el uso y actualización de una agenda de contactos y eventos.

---

[1] La versión que se provee a los estudiantes es una versión modificada del AIML 2.0 realizada por un estudiante de la UGR para su Trabajo Fin de Grado.

2.1 Agenda de contactos

La agenda de contactos guardará el apellido, nombre, dirección y teléfono de cada contacto. Asumiremos que no hay dos contactos con el mismo apellido y que cada contacto solo tiene un nombre, una dirección y un teléfono.

En este nivel el agente conversacional debe permitir:

- Agregar un nuevo contacto. Por ejemplo:
  - Apellido: Perez Lopez
  - Nombre: Juan Jose
  - Direccion: Av de Madrid 6
  - Telefono: 12345678
- Indicar la cantidad de contactos almacenados.
- Listar los contactos (lista de apellidos únicamente) almacenados.
- Borrar un contacto identificado por su apellido. Debe preguntarse al usuario si está seguro de borrar antes de realizar la operación.
- Modificar un contacto identificado por el apellido. El usuario podrá modificar el domicilio o bien el teléfono.
- Buscar el nombre, dirección o teléfono de un contacto identificado por su apellido.

2.2 Agenda de eventos

La agenda de eventos debe permitir guardar un evento con un título dado (este título puede llegar a repetirse en la agenda) en una fecha y hora determinada, así como los contactos de la agenda que participan en dicho evento. La agenda corresponde a los años entre el 2017 y el 2027 únicamente. Todos los eventos tienen una duración de media hora que empieza en punto o media hora después. Por ejemplo, 09:00, 14:30, etc, pero nunca a las 22:45. Es decir, las horas posibles de eventos son 00:00, 00:30, 01:00, 01:30, …, 22:30, 23:00, 23:30. No pueden ocurrir dos eventos el mismo día a la misma hora.

En este nivel el agente conversacional debe permitir:

- Agregar un nuevo evento. Por ejemplo:
  - Título: Médico
  - Fecha: 20/08/2017
  - Hora: 22:30
  - Contactos: Perez Lopez, Sanchez Alvarez

- Modificar un evento identificado por su fecha y hora. Se puede modificar la lista de contactos agregando o quitando un contacto a la vez (identificado por su apellido). También es posible modificar la fecha, la hora y el título del evento.
- Borrar un evento identificado fecha y hora. Debe preguntarse al usuario si está seguro de borrar antes de realizar la operación.
- Contar los eventos de una fecha en particular.

2.3 Consultas

Una vez que se cuente con una agenda de contactos y eventos operativa, se le pueden realizar consultas algo más complejas que un simple listado. Los modelos de tipo de consultas que se quiere resolver son los siguientes:

- Tengo el dia libre?
- Cual es mi proximo evento de hoy/mañana/fecha?
- Cual es el ultimo evento de hoy/mañana/fecha?
- Con quienes me tengo que reunir hoy/mañana/fecha?
- Tengo algun hueco libre hoy/mañana/fecha? Alguna otra alternativa?
- Tengo algun hueco libre hoy/mañana/fecha por la mañana/tarde/noche? Alguna otra alternativa?
- Cuando tengo un evento con Perez_Lopez? Alguna otra alternativa?

3. Evaluación

3.1 Definición de los niveles de dificultad

Hemos diseñado un modelo de evaluación para que el estudiante decida con qué intensidad y a qué nivel desea implicarse en su elaboración. Obviamente, a mayor nivel de implicación, el alumno opta a una mayor calificación en la práctica. Por tanto hemos definido 3 niveles en la entrega de esta primera práctica que son, de menor a mayor implicación, las siguientes:

(a) **Nivel 1**: Entrega del conocimiento necesario para responder a las solicitudes sobre la agenda de contactos (detallado en el apartado 2.1). En este caso, el alumno puede optar hasta cuatro puntos sobre diez.

(b) **Nivel 2**: Entrega del conocimiento necesario para responder a las solicitudes del nivel 1 y de las cuestiones sobre la agenda de eventos (detallado en el apartado 2.2). En este caso, el alumno pude optar hasta una calificación de siete puntos sobre diez.

(c) **Nivel 3**: Entrega del conocimiento necesario para responder a las solicitudes de los niveles 1 y 2 y pueda afrontar las consultas algo más complejas sobre la agenda de contactos y de eventos (como las que se muestran en el apartado 2.3). Este es el nivel de mayor dificultad e implicación y el alumno opta a la valoración de diez sobre diez.

No se considerarán aquellas entregas que no puedan ubicarse en uno de estos niveles; es decir, estas son las únicas posibilidades de entrega.

Un ejemplo de entrega inválida, es aquella donde el alumno entrega el conocimiento para resolver la agenda de eventos pero no la de contactos.

3.2 ¿Cómo se evaluará cada uno de los niveles?

Adjunto a este guión, se establece una fecha límite para la entrega de la práctica. Antes de dicha fecha, el alumno debe subir el material que más adelante se indica conteniendo el conocimiento necesario para resolver los problemas hasta el nivel que ha decidido. En una segunda fecha posterior, que también se indica en este guión y que corresponderá con un día del horario habitual del grupo de prácticas se realizará la defensa de la práctica.

Hemos definido una forma de defensa para cada uno de los niveles definidos en la entrega. A continuación se indica cómo se evaluará cada nivel:

Nivel 1: Se pedirá al alumno que introduzca, modifique y elimine una serie de contactos siguiendo las indicaciones que le propone el profesor. Si el resultado es coherente con las preguntas formuladas, el alumno superará este nivel y tendrá una calificación de hasta 4. En otro caso, la defensa termina y la calificación del alumno será de 0.

En cualquier caso, el profesor siempre tiene la potestad de pedir al alumno que introduzca nuevo conocimiento o que le explique alguna parte de su código para asegurarse de que supera este nivel.

Nivel 2: Para llegar a este nivel en la defensa, debe haberse superado el nivel anterior. En este caso. Se pedirá al alumno que introduzca, modifique y elimine una serie de eventos siguiendo las indicaciones que le propone el profesor. Si el resultado es coherente con las preguntas formuladas, el alumno superará este nivel y añadirá 3 puntos más a su calificación. En otro caso, la defensa termina y la calificación del alumno será de 4 puntos.

Nivel 3: Al igual que antes, debe haberse superado el nivel anterior para pasar a evaluar este nivel. En este nivel se realizará una serie de consultas complejas sobre eventos y contactos. Si el agente mantiene la conversación y la secuencia de preguntas hasta llegar a la solución correcta, el alumno superará este nivel, y añadirá 3 puntos a su calificación. Si no es así, la defensa habrá terminado y la calificación del alumno será de 7 puntos.

3.3. ¿Qué hay que entregar?

Antes de que termine el plazo de entrega fijado en este guión, el alumno deberá subir a la plataforma de la asignatura en decsai.ugr.es, un archivo comprimido con zip llamado "practica3.zip". Este archivo debe tener la misma estructura en carpetas que cuelga de la carpeta "mybot", donde obviamente las carpetas "aiml", "aimlf", "sets" y "maps" contienen los ficheros necesarios para resolver la práctica al nivel que ha decidido el alumno.

3.4. Observaciones Finales

Esta práctica es INDIVIDUAL y trata de establecer la capacidad del alumno para desarrollar una base de conocimiento usando el lenguaje AIML. El profesorado para asegurar la originalidad de cada una de las entregas, someterá a estas a un procedimiento de detección de copias.

En el caso de detectar prácticas copiadas, los involucrados (tanto el que se copió como el que se ha dejado copiar) tendrán suspensa la asignatura. Por esta razón, recomendamos que en ningún caso se intercambie código entre los alumnos. No servirá como justificación del parecido entre dos prácticas el argumento "es que la hemos hecho juntos y por eso son tan parecidas", ya que como se ha dicho antes, las prácticas son INDIVIDUALES.

Como se ha comentado previamente, el objetivo de la defensa de prácticas es evaluar la capacidad del alumno para enfrentarse a este problema. Por consiguiente, se asume qué todo el código que aparece en su práctica ha sido introducido por él por alguna razón y que dicho alumno domina perfectamente el código que entrega. Así, si durante cualquier momento del proceso de defensa el alumno no justifica adecuadamente algo de lo que aparece en su código, la práctica se considerará copiada y tendrá suspensa la asignatura. Por esta razón, aconsejamos que el alumno no incluya nada en su código que no sea capaz de explicar qué misión cumple dentro de su práctica.

Por último, las prácticas presentadas en tiempo y forma, pero no defendidas por el alumno, se considerarán como no entregadas y el alumno obtendrá la calificación de 0. El supuesto anterior se aplica a aquellas prácticas no involucradas en un proceso de copia. En este último caso, el alumno tendrá suspensa la asignatura y deberá acudir a la convocatoria de septiembre.

4. Desarrollo temporal de la práctica

Esta tercera practica seguirá el siguiente desarrollo temporal:

a) **Semana del 24 al 28 de abril**: Presentación de la práctica.

b) **Semanas del 1 al 26 de mayo**: Desarrollo de la práctica en clase.

c) **Semana del 29 de mayo al 2 de junio**: Defensa de la práctica que se realizará en el día y hora de la sesión de prácticas que le corresponde habitualmente al alumno. Si por algún motivo, el alumno no pudiera asistir a su sesión esa semana, debe avisar al profesor de

prácticas para que le asigne a otro grupo de prácticas de la asignatura para esa misma semana.

d) **La fecha tope para la entrega** será **el 26 de mayo antes de las 23:55 horas.**

ANEXO

Descripción del fichero “utilidades.aiml”

Dentro del software entregado para la realización de la práctica 3, se incluye un fichero llamado “utilidades.aiml” que incorpora algunas reglas genéricas que permitirán al alumno realizar con mayor comodidad su práctica.

En este anexo se describen las reglas incluidas en este fichero con algunos ejemplos para ilustrar su uso.

Las utilidades son las siguientes:

1. Operaciones para manejar una lista de nombres
2. Generar un número aleatorio en un rango
3. Otra forma de hacer un ciclo y comparar dos cadenas

Ahora describiremos cada una de ellas.

A1. Operaciones para manejar lista de nombres

En AIML se trabaja exclusivamente con cadenas de caracteres y con procesamiento simbólico. Así, cada variable global o local que se usa es siempre de tipo cadena de caracteres. Por esa razón, es importante tener definidas funcionalidades dentro del lenguaje que faciliten el trabajo con este tipo de dato.

En el fichero “utilidades.aiml” hemos incluido algunas de las operaciones más frecuentes y que pueden tener uso para el desarrollo de los problemas que se piden en esta práctica. En concreto, se han incluido las siguientes operaciones:

- **TOP**: Dada una lista de palabras, devuelve la primera palabra de esa lista.
- **REMAIN**: Dada una lista de palabras, devuelve la misma lista de palabras quitando la primera palabra.
- **COUNT**: Dada una lista de palabras, devuelve el número de palabras que contiene.
- **FINDITEM [palabra] IN [listaPalabras]:** Determina si “palabra” es una palabra incluida en “listaPalabras”. Devuelve 0 si no la encuentro o su posición en la lista en caso de haberla encontrado.
- **SELECITEM [number] IN [listaPalabras]:** Devuelve la palabra que ocupa la posición “number” en “listaPalabras”.
- **REMOVEITEM [number] IN [listaPalabras]:** Elimina de “listaPalabras” la palabra de posición “number”.

- **ADDITEM [palabra] IN [listaPalabras]:** Añade "palabra" a principio "listaPalabras", sólo si "palabra" no estaba antes en "listaPalabras". En este segundo caso, la operación no hace nada.

Este repertorio de operaciones no tiene sentido si se utiliza como salida directa del agente, sino que tiene como función procesar de una manera más elaborada la salida que se ofrecerá, y está concebido para trabajar sobre variables, ya sean globales (predicados) o locales.

Una aclaración antes de pasar a ilustrar su uso con algunos ejemplos: En las definiciones anteriores se ha hecho uso de conceptos como "palabra" y "lista de palabras" para matizar la intención con la que se han construido estas herramientas pero, en realidad, en todos los casos, los parámetros son cadenas de caracteres. Cuando se hace referencia a "palabra" se quiere indicar que es una secuencia de símbolos que está entre 2 separadores (entendiendo aquí separador por uno o varios espacios en blanco). Cuando se hace referencia a "lista de palabras", se indica que contiene una secuencia de palabras separadas por espacios en blanco.

Para ilustrar como se pueden usar, supongamos que existe una variable global llamada list_fruit que almacena una secuencia de frutas.

Ejemplo 1: Darle algunos valores a esa variable global mediante la definición de una regla.

Para esto, aprovecharemos la pregunta "conoces algunas frutas?", para darle un valor inicial a esa variable.

```
<category>
<pattern> conoces algunas frutas</pattern>
<template>
    <think>
      <set name="list_fruit">fresa cereza naranja mandarina</set>
    </think>
    si, alguna conozco.
</template>
</category>
```

Ejemplo 2: Queremos ir actualizando esa variable global con nuevas frutas que nos pueda ir proporcionando el usuario.

Vamos a construir una regla qué aprenda nuevas frutas a través de afirmaciones del usuario del tipo "la manzana es una fruta" o "el membrillo es una fruta".

```
<category>
<pattern>la * es una fruta</pattern>
<template>
    <think>
```

```
      <set var="existe">
         <srai>FINDITEM <star/> IN <get name="list_fruit"/></srai>
      </set>
    </think>
    <condition var="existe">
      <li value="0">
        <think>
          <set name="list_fruit">
             <srai>
                ADDITEM <star/> IN <get name="list_fruit"/>
             </srai>
          </set>
       </think>
            Recordare que <star/> es una fruta.
        </li>
        <li>
            Ya sabia que <star/> es una fruta.
        </li>
    </condition>
</template>
</category>
```

El proceso de esta regla es bastante intuitivo,

1. Asigna en la variable local existe si lo que se pasa en <star/> es una de las frutas que ya conoce y se encuentran almacenadas en la variable global list_fruit. Para determinar la existencia de esa fruta hemos hecho uso de **FINDITEM**.
2. La variable existe almacena un 0, si no encuentra esa fruta en la lista y en ese caso lo que hace es añadirla al principio de la lista list_fruit, y ofrece el mensaje al interlocutor de que recordará esa fruta. Para añadir la nueva fruta hemos hecho uso de **ADDITEM**.
3. Si la variable existe almacena un valor distinto de 0, significa que en la posición almacenada por esa variable se encuentra la fruta. En este caso, no modifica la lista list_fruit, y lanza el mensaje de que ya sabía que eso era una fruta.

La regla anterior recoge el caso de un patrón del tipo *"la… es una fruta"*, pero nos gustaría recoger el caso también del patrón "el… es una fruta". La primera intención sería copiar esta regla, pegarla debajo en el editor y cambiar el "la" por un "el". Esto funcionaría, pero un experto programador en AIML se daría cuenta qué es más simple crear un nueva regla que invoque a la anterior y cambiando el nuevo patrón. El resultado sería el siguiente:

```
<category>
<pattern>el * es una fruta</pattern>
<template>
    <srai>la <star/> es una fruta</srai>
```

```
</template>
</category>
```

Como se puede observar <srai> hace un papel semejante al de la invocación de un método en un lenguaje de programación convencional, y aquí aprovechamos ese recurso.

Ejemplo 3: Ahora vamos a ampliar nuestro repertorio de reglas para permitir corregir algún error en nuestra lista de frutas. Supongamos que en un momento dado, por error, el interlocutor dijo: *"la mesa es una fruta"* y nuestro agente añadió *mesa* a la lista, pero el interlocutor poco después se da cuenta de su error y quiere corregirlo y nos dice *"fue un error, la mesa no es una fruta"*. Incluiremos una regla para permitir que esto se pueda hacer:

```
<category>
<pattern>no es una fruta la *</pattern>
<template>
   <think>
     <set var="pos">
        <srai>FINDITEM <star/> IN <get name="list_fruit"/></srai>
     </set>
   </think>
   <condition var="pos">
     <li value="0"> Como puedes pensar que considere eso una fruta
     </li>
     <li>
       <think>
         <set name="list_fruit">
          <srai>
           REMOVEITEM <get var="pos"/> IN <get name="list_fruit"/>
          </srai>
         </set>
       </think>
       Menos mal que me lo dices, yo creia que era una fruta.
     </li>
    </condition>
</template>
</category>
```

El proceso es semejante al que se ilustra en el *Ejemplo 2*, se busca si <star/> está en la lista de frutas mediante **FINDITEM** y almacenado su valor en pos. Si pos vale 0 entonces eso no se había considerado como fruta. En otro caso, el valor está dentro de la lista de frutas y hay que eliminarlo. Para eso usamos **REMOVEITEM** que elimina la palabra de posición pos de list_fruit y el resultado se reasigna a list_fruit.

De igual modo que en el ejemplo anterior, podemos incluir la regla que considera los patrones que son de la forma "… el… no es una fruta":

```
<category>
<pattern>no es una fruta el *</pattern>
<template>
    <srai> no es una fruta la <star/></srai>
</template>
</category>
```

Ejemplo 4: Ya podemos actualizar nuestra lista de frutas insertando y eliminando elementos de la misma, pero el interlocutor sólo nos puede dar la información para recordar la fruta de una en una. Sería interesante que pudiera darnos una lista de frutas que el agente fuera capaz de recordar del tipo "*la manzana, el platano, el melon y la ciruela son frutas*", donde aparecen delante de cada fruta un "el" o un "la" y antes de dar la última fruta aparece una "y". Obviamente, deberíamos usar las reglas definidas anteriormente que nos permiten modificar la lista.

```
<category>
<pattern> * son frutas</pattern>
<template>
    <think>
      <set var="lista"><star/></set>
      <set var="item">
        <srai>TOP <get var="lista"/></srai>
      </set>

    <condition var="item">
      <li value="y">
        <set var="item">
          <srai>SELECTITEM 3 IN <get var="lista"/></srai>
        </set>
        <srai> la <get var="item"/> es una fruta </srai>
      </li>

      <li value="la">
        <set var="lista">
          <srai>REMAIN <get var="lista"/></srai>
        </set>
        <set var="item">
          <srai>TOP <get var="lista"/></srai>
        </set>
        <loop/>
      </li>

      <li value="el">
        <set var="lista">
          <srai>REMAIN <get var="lista"/></srai>
```

```
        </set>
        <set var="item">
          <srai>TOP <get var="lista"/></srai>
        </set>
        <loop/>
      </li>

      <li>
        <srai> la <get var="item"/> es una fruta </srai>
        <set var="lista">
          <srai>REMAIN <get var="lista"/></srai>
        </set>
        <set var="item">
          <srai>TOP <get var="lista"/></srai>
        </set>
        <loop/>
      </li>
    </condition>
  </think>
  Recordare todas estas frutas
</template>
</category>
```

Vamos poco a poco:

1. La primero acción almacena el contenido de la parte variable del patrón en la variable local lista.
2. Se extrae la primera palabra de lista y se almacena en la variable ítem.
3. Ahora se plantea una estructura condicional en función del valor de ítem, que consideramos que puede tomar 4 valores diferentes: "y", "la", "el" u otro valor. En el caso de tomar el valor:
    a. "y", entonces estamos justo antes de terminar de la secuencia de palabras. La siguiente palabra a "y" es un artículo que podemos ignorar y la siguiente es el nombre de la fruta. Por esa razón, usamos **SELECTITEM** para tomar el segundo de la lista, que almacenamos en ítem, para después invocar a la regla que es de la forma *"la… es una fruta"*, dónde… es el valor de ítem. Esta es la última fruta de la lista, por tanto, se sale del ciclo. (por eso, a diferencia del resto de los casos, no termina con un <loop/>.
    b. "la" o "el", para las dos situaciones tengo que hacer lo mismo. En este caso, ignorar el artículo y pasar a evaluar la siguiente palabra que es la que contiene el nombre de la fruta. Esta operación se hace con la conjunción de **TOP** que extrae el primero de lo que queda en la lista y lo almacena en ítem, y **REMAIN** que devuelve la lista menos el primero, que se vuelve a almacenar en lista. Como en este caso la secuencia de palabras no ha terminado, se tiene que ciclar, por eso aparece <loop/>.

c. el caso por defecto. Si no es una "y", ni un "el", ni un "la", lo que tiene en ítem es el nombre de una fruta. En este caso, se invoca al igual que en el apartado (a) a la regla *"la… es una fruta"* que la añadirá si no existe en la lista, y pasa a la siguiente palabra de la misma forma que se indica en el apartado (b), es decir, con la conjunción de **TOP** y **REMAIN** y al igual que antes, quedan palabras por procesar, por tanto se cicla con <loop/>.

4. Terminado el ciclo el procesado de la cadena ha terminado y propone al interlocutor una frase en el sentido de que recordara las frutas.

En estos cuatro ejemplos hemos mostrado el uso de todas y cada una de las operaciones para el manejo de listas de palabras.

A2. RANDOM

La sintaxis de esta acción es **RANDOM [number]** y devuelve un número aleatorio entre 1 y number. Aquí mostramos un ejemplo de uso de RANDOM.

Ejemplo 5: Se plantea una regla simple que ante la entrada de "Dime una fruta", el agente conversacional devuelve aleatoriamente una de entre la lista de frutas que tiene almacenadas en la variable list_fruit.

La descripción de dicha regla sería la siguiente:

```
<category>
<pattern> Dime una fruta </pattern>
<template>
 <think>
  <set var="lista"> <get var="list_fruit"/> </set>
  <set var="cantidad"><srai>COUNT <get var="lista"/></srai></set>
  <set var="pos"><srai>RANDOM <get var="cantidad"/></srai></set>
  <set var="elegida">
     <srai>
       SELECTITEM <get var="pos"/> IN <get var="lista"/>
     </srai>
   </set>
  </think>
  <get var="elegida"/>
</template>
</category>
```

El proceso de respuesta a la pregunta sería el siguiente:

1. Asigna a la variable lista una secuencia de nombres de frutas.
2. Mediante **COUNT** determina el número de frutas introducidas en lista y se la asigna a la variable cantidad.

3. A partir de cantidad, qué indica el número de frutas elegibles, usa **RANDOM** para generar un número aleatorio entre 1 y cantidad que almacena en pos.
4. Selecciona la palabra que ocupa la posición pos de lista y la almacena en elegida usando **SELECTITEM.**
5. Devuelve como respuesta el valor de la variable elegida.

A3. Operaciones para manejar lista de nombres

En AIML se trabaja exclusivamente con cadenas de caracteres y con procesamiento simbólico. Así, cada variable global o local que se usa es siempre de tipo cadena de caracteres. Por esa razón, es importante tener definidas funcionalidades dentro del lenguaje que faciliten el trabajo con este tipo de dato.

A4. ITERATE / NEXT y COMPARE

Con el par ITERATE/NEXT tratamos de hacer una versión más convencional de un ciclo que itera sobre una lista de palabras. El proceso es simple: ITERATE se usa sólo una vez al principio del ciclo, y permite situarse sobre la primera palabra de la lista de palabras. El resto del proceso está guiado por NEXT, que devuelve el siguiente valor de la lista. Tanto ITERATE como NEXT devuelve la cadena "end" cuando se termina de recorren la lista de palabras.

Ejemplo 6: Construir una regla que devuelva todas las frutas que conoce el agente.

En principio, esta regla sería tan simple como devolver el valor de la variable list_fruit, pero nosotros vamos a complicarlo un poco para usar estas acciones. Una posible implementación es la siguiente:

```
<category>
<pattern> Dime todas las frutas que conoces </pattern>
<template>
   Estas son las frutas que conozco
   <think>
     <set var="item">
        <srai> ITERATE <get var="list_fruit"/> </srai>
     </set>
     <condition var="item">
        <li value="end"></li>
        <li> <get var="item"/>
             <think>
               <set var="item">
                <srai>NEXT</srai>
               </set>
             </think>
             <loop/>
```

```
        </li>
      </condition>
</template>
</category>
```

La sintaxis de esta acción es **COMPARE [palabra] WITH [palabra],** y devuelve "YES" si son iguales las palabras y "NO" si no lo son.

Ejemplo 7: Vamos a ver otra implementación para el problema del ejemplo 4, pero en este caso, se pasa una lista de palabras que no son frutas y queremos por un lado actualizar la variable list_fruit y por otro lado dar una respuesta diferente dependiendo de si había alguna "no fruta" incluida dentro de mi lista de frutas.

```
<category>
<pattern> no son frutas *</pattern>
<template>
    <think>
      <set var="cantidad">
        <srai>COUNT <get var="list_fruit"/></srai>
      </set>
      <set var="item">
        <srai>ITERATE <get var="lista"/></srai>
      </set>

    <condition var="item">
      <li value="y">
        <set var="item"><srai>NEXT</srai></set>
        <set var="item"><srai>NEXT</srai></set>
        <srai> no es una fruta la <get var="item"/> </srai>
      </li>

      <li value="la">
        <set var="item"><srai>NEXT</srai></set>
        <loop/>
      </li>

      <li value="el">
        <set var="item"><srai>NEXT</srai></set>
        <loop/>
      </li>

      <li>
        <srai> no es una fruta la <get var="item"/> </srai>
        <set var="item"><srai>NEXT</srai></set>
        <loop/>
```

```
      </li>
    </condition>

    <set var="cantidad2">
      <srai>COUNT <get name="list_fruit"/></srai>
    </set>

    <set var="iguales">
      <srai>
         COMPARE <get var="cantidad"/> WITH <get var="cantidad2"/>
      </srai>
    </set>
    </think>

 <condition var="iguales">
  <li value="YES"> Ya sabia que ninguna de esas era un fruta </li>
  <li> Pues vaya, he tenido que corregir algunos errores </li>
 </condition>
</template>
</category>
```

Creemos que este último ejemplo es fácil de seguir, si se ha entendido todo lo que se ha descrito anteriormente en este Anexo, así que dejaremos que lo intentéis por vosotros mismos.

REFLEXIÓN FINAL

¿Qué ocurrirá si las reglas del ejemplo 3 en vez de tener el patrón “no es una fruta la *”, tuviera un patrón semejante al del ejemplo 2, es decir, “la * no es una fruta”? ¿Qué se tendría que modificar para hacer compatibles las dos posibilidades anteriores? ¿Te atreves a corregirlo?