

Representación del Conocimiento

Jason

Dr. Alejandro Guerra-Hernández

Universidad Veracruzana

Centro de Investigación en Inteligencia Artificial
Sebastián Camacho No. 5, Xalapa, Ver., México 91000

<mailto:aguerra@uv.mx>

<http://www.uv.mx/personal/aguerra/rc>

Maestría en Inteligencia Artificial 2020



Universidad Veracruzana

Jason

- ▶ Jason [3, 1, 4, 2] es un lenguaje de **programación orientado a agentes** (AOP) basado en una versión extendida de *AgentSpeak(L)*, implementado en **Java**.
- ▶ Sus **extensiones** incluyen:
 - ▶ Comunicación basada en **actos de habla** [12].
 - ▶ Herramientas para **simulación social** [5].
 - ▶ Un sistema de **módulos** [9]
 - ▶ Anotaciones, Negación fuerte, Ambientes, etc.
- ▶ Implementa la **semántica operacional** del lenguaje y provee varias **plataformas de desarrollo**, con muchas opciones configurables por el usuario.
- ▶ Se trata de un **código abierto**, distribuido bajo una licencia **GNU LGPL**.



Páginas web

- ▶ Jason puede descargarse desde [sourceforge](#):

<http://jason.sourceforge.net/wp/>

- ▶ El [libro](#) de Bordini, Hübner y Wooldridge [4], *Programando Sistemas Multi-Agentes en AgentSpeak(L) usando Jason*, tiene su página en:

<http://jason.sourceforge.net/jBook/jBook/Home.html>

- ▶ Los desarrolladores pueden clonar también su repositorio [github](#):

<https://github.com/jason-lang/jason>



Requisitos

- ▶ **Java** 1.7 mínimo (<https://www.oracle.com/java/index.html>)
- ▶ Opcionalmente necesitaremos:

Eclipse <https://eclipse.org>

Atom <https://atom.io>

Gradle <https://gradle.org>

Asciidoctor <https://asciidoctor.org>

Ant <https://ant.apache.org>



Directorio principal



build



build.gradle



demos



doc



examples



jedit



LICENSE



readme.adoc



release-
notes.adoc



scripts



src



Universidad Veracruzana

Observaciones

- ▶ El **código fuente** está disponible en `src`; los **ejemplos** y **demos** están incluidos en los folders `examples` y `demos`, respectivamente.
- ▶ La **documentación** en `doc` incluye algunos artículos relevantes y la descripción del API de Jason.
- ▶ La carpeta `scripts` tiene todos los scripts para **ejecutar** Jason desde consola. Es importante por lo tanto que esta carpeta esté en la ruta de acceso del sistema.
- ▶ Desde la versión 2.0 de Jason, no hay una aplicación en la interfaz gráfica de los diferentes sistemas operativos donde Jason puede ejecutarse. Deben usarse estos **scripts**.



Scripts de Jason

- ▶ Los siguientes scripts pueden **ejecutarse** en terminal, desde donde sea, una vez que Jason se ha configurado:

Script	Descripción
<code>jason</code>	Ejecuta un SMA. Recibe como argumento el nombre del sistema (archivo con extensión <code>mas2j</code>).
<code>jason-ide</code>	Lanza el IDE basado en JEdit.
<code>jason-setup</code>	Configura paths y la variable <code>JASON_HOME</code> .
<code>jason-new-project</code>	Crea un proyecto nuevo. Recibe como argumento el nombre del SMA a crear, sin extensión alguna. Crea el proyecto en el directorio actual.

- ▶ Para **configurar** ejecute `jason-setup` desde el directorio donde se encuentra.



Instalación

- ▶ Clonar el repositorio y compilar con gradle, en una terminal:

```
1 |> git clone https://github.com/jason-lang/jason.git
2 |> cd jason
3 |> gradle config
```

- ▶ La tarea de Gradle config **compila** las fuentes para generar los archivos jar correspondientes;
- ▶ **configura** el archivo de propiedades de jason y coloca todos los jar en la carpeta build/libs;
- ▶ y solicita al usuario su autorización para definir las **variables** JASON_HOME y PATH adecuadamente.



Otras tareas gradle

Acción	Descripción
jar	Genera un nuevo <code>Jason.jar</code>
doc	Genera javadoc y transforma asciidoc en html.
eclipse	Genera una configuración para proyectos eclipse.
config	Ejecuta las tres acciones anteriores.
clean	Borra los archivos generados.
release	Produce un zip en <code>build/distributions</code> .

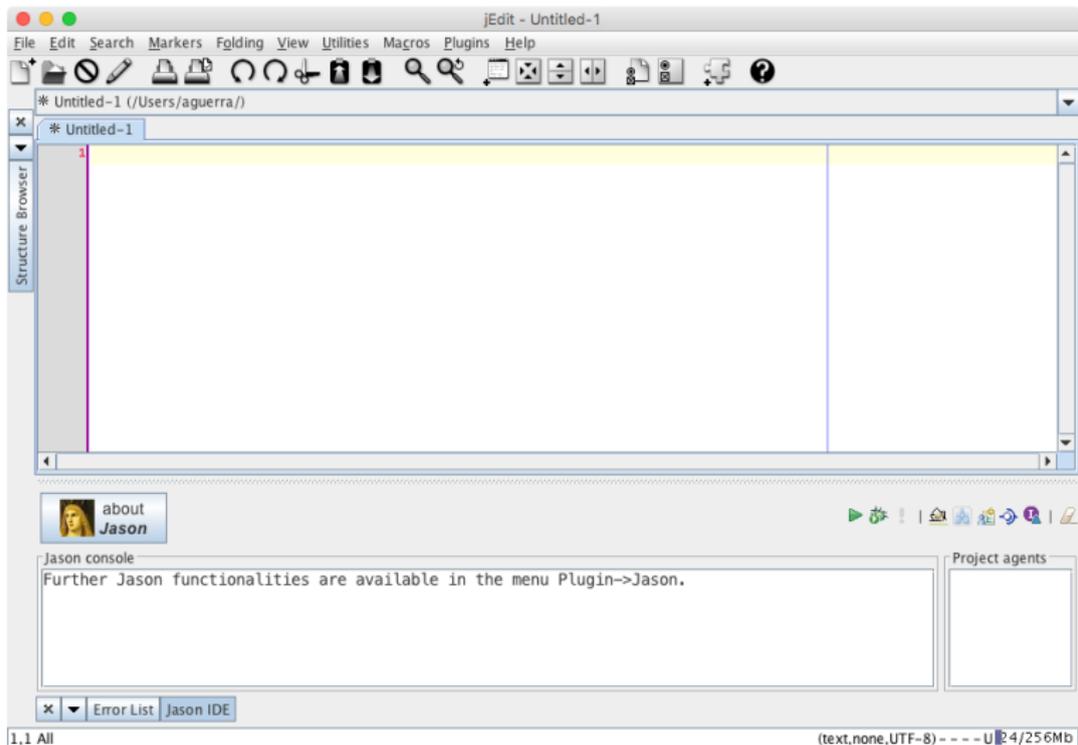


Opciones

- ▶ Jason cuenta con tres **ambientes de desarrollo** (IDEs):
 - ▶ El oficial basado en **jEdit** –Un editor de código multi lenguaje, implementado en Java;
 - ▶ Uno basado en **Eclipse**, con todas las ventajas para Java que esto conlleva.
 - ▶ Y uno más minimalista basado en **Atom**.



> jason-ide

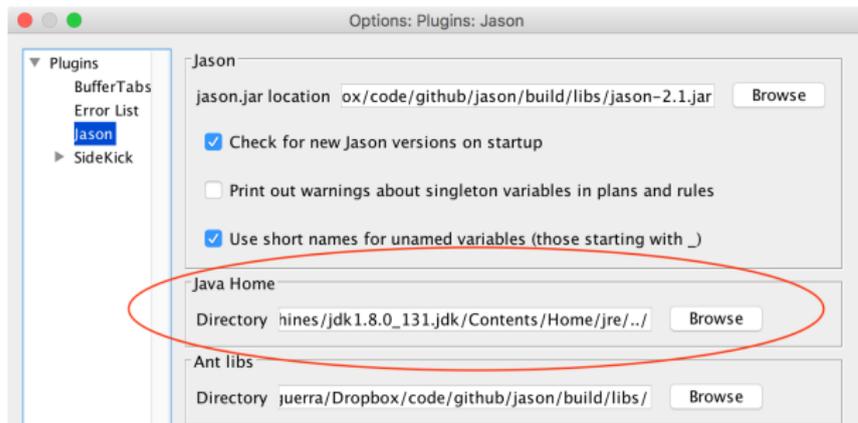


Path a Java

- ▶ Se puede configurar `JAVA_HOME` desde el shell, agregando la siguiente línea al archivo de configuración:

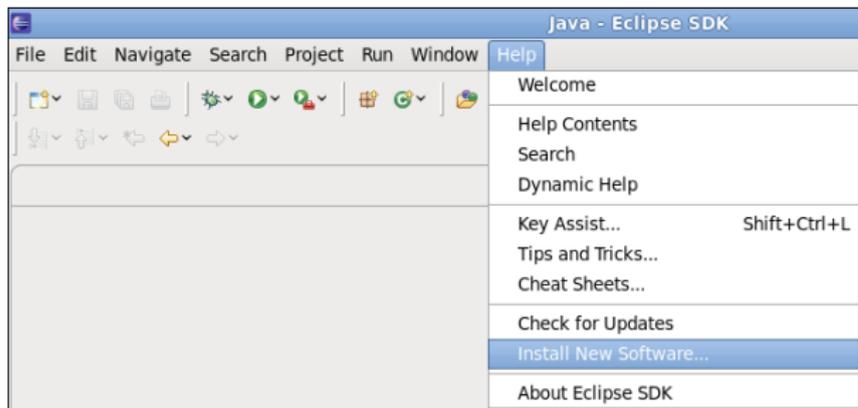
```
1 | export JAVA_HOME='/usr/libexec/java_home -v 1.8'
```

- ▶ O desde las opciones del plugin de Jason para JEdit:



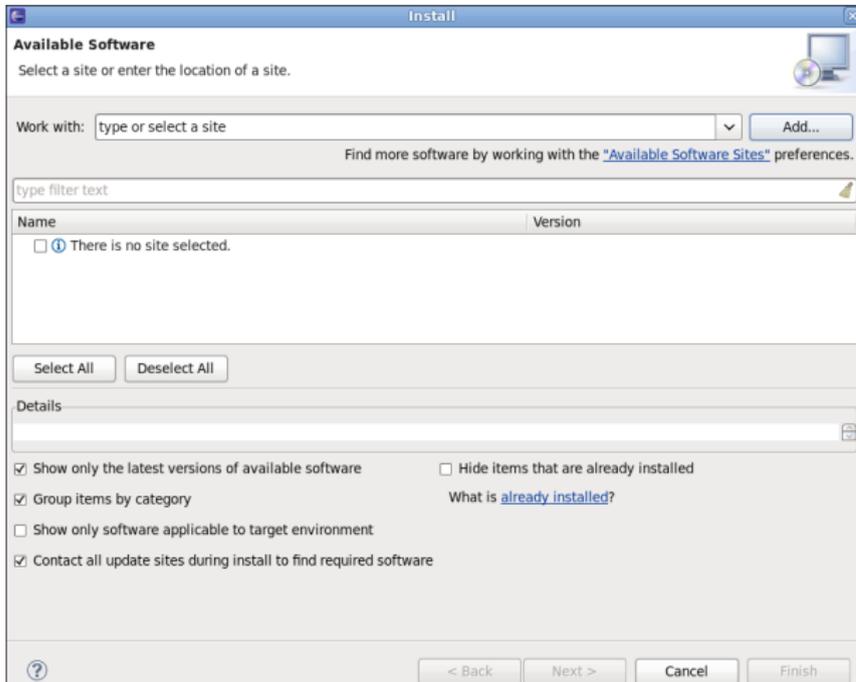
Instalación 1

- ▶ En Eclipse, vaya a la opción “Instalar Nuevo Software” del menú “Ayuda”:



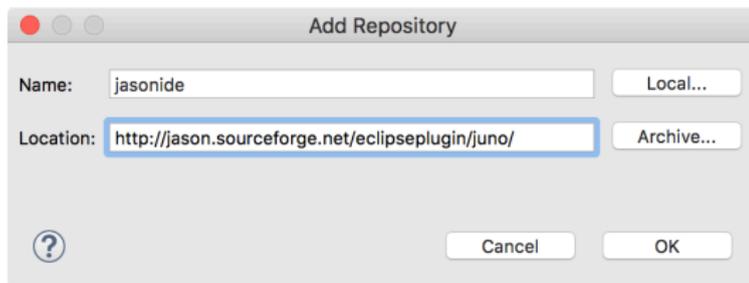
Instalación 2

- ▶ Lo cual abrirá la siguiente ventana:



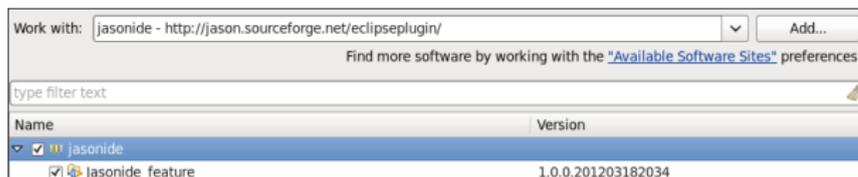
Instalación 3

- ▶ Dar clic en el botón de agregar repositorio (*Add...*) y:



Instalación 4

- ▶ Seleccionar la opción “jasonide” y dar clic en el botón de siguiente (“Next”).
- ▶ Espere un poco, eclipse buscará las **dependencias** de paquetes necesarias.



- ▶ Dar clic al botón siguiente una vez más, y aceptar la licencia del *plug-in*.
- ▶ Reiniciar eclipse.

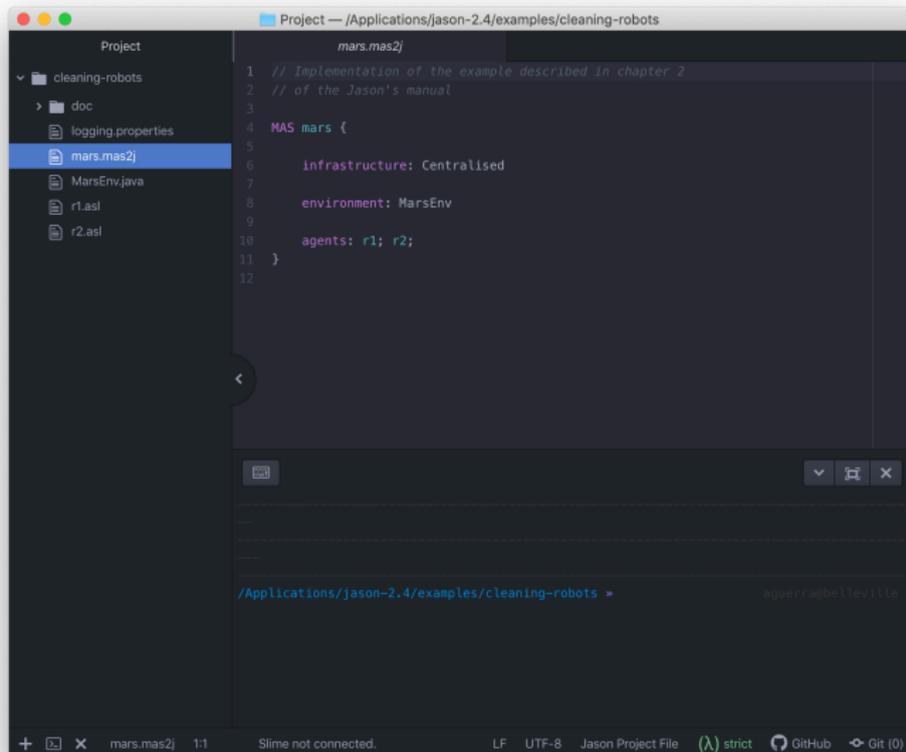


Paquetes

- ▶ Instalar los siguientes paquetes en Atom:
 - ▶ language-agentspeak
 - ▶ language-mas2j
- ▶ Y opcionalmente:
 - ▶ platformio-ide-terminal



Ambiente Atom



The screenshot shows the Atom IDE interface. The left sidebar displays a project structure for 'cleaning-robots' with files like 'mars.mas2j', 'MarsEnv.java', 'r1.asl', and 'r2.asl'. The main editor window shows the content of 'mars.mas2j' with the following code:

```
1 // Implementation of the example described in chapter 2
2 // of the Jason's manual
3
4 MAS mars {
5
6     infrastructure: Centralised
7
8     environment: MarsEnv
9
10    agents: r1; r2;
11 }
12
```

The status bar at the bottom indicates the current file is 'mars.mas2j', the editor is not connected to a server, and the encoding is UTF-8. It also shows the project file type as 'Jason Project File' and the current mode as 'strict'.



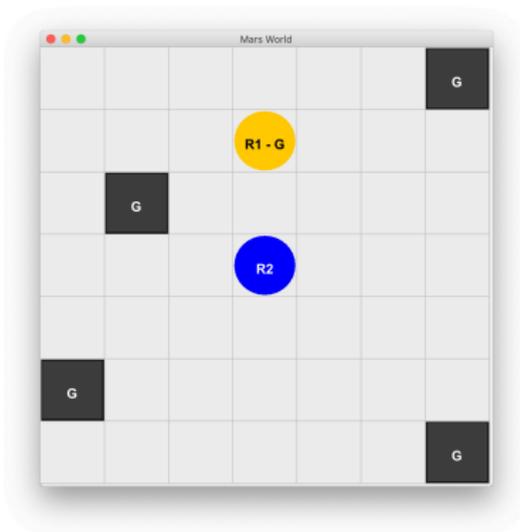
Diferencias

- ▶ Los **ejemplos** son SMAs completos, que ilustran problemas típicos de esta área de investigación.
- ▶ Los **demos** son proyectos más sencillos, que ilustran como usar algunas de las características técnicas de Jason.



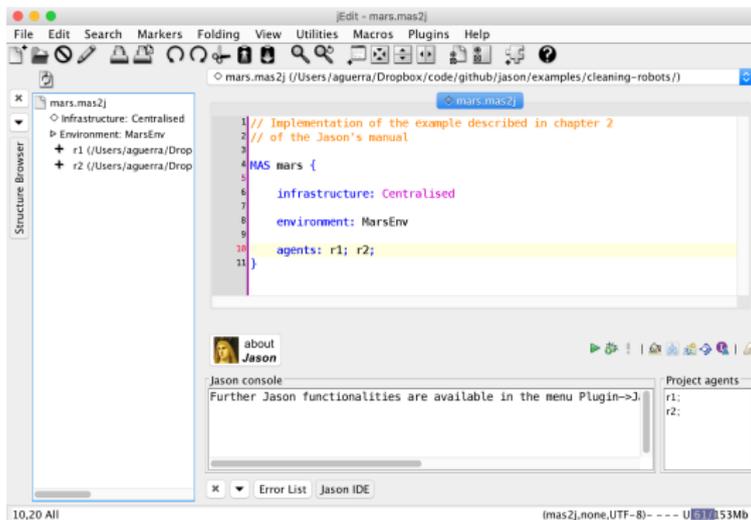
Descripción del ejemplo

- ▶ Vamos a trabajar con el ejemplo `cleaning-robots`.
- ▶ El robot `r1` explora el medio ambiente (rejilla 2D) buscando basura.
- ▶ Cuando la encuentra se la lleva a `r2` para incinerarla.
- ▶ `r1` regresa a la posición donde encontró la última basura y continua su exploración.
- ▶ A la derecha el GUI de este SMA.



Archivo de configuración

- ▶ Todos los proyectos Jason tienen un **archivo de configuración** con la extensión mas2j.
- ▶ Abrir `examples/cleaning-robots/mars.mas2j` en JEdit.

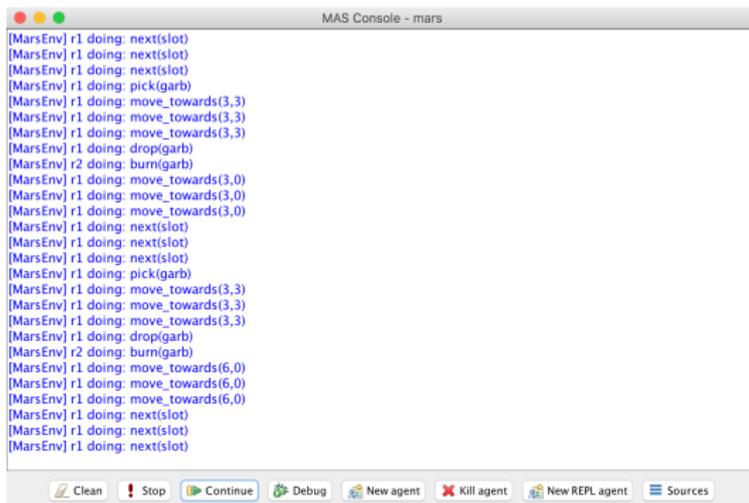


```
1 // Implementation of the example described in chapter 2
2 // of the Jason's manual
3
4 MAS mars {
5
6     infrastructure: Centralised
7
8     environment: MarsEnv
9
10    agents: r1; r2;
11 }
```



Ejecución

- ▶ En JEdit o Eclipse, de clic en el botón del *play*.
- ▶ Dos ventanas aparecen: una consola MAS y la GUI del proyecto.



```

MAS Console - mars
[MarsEnv] r1 doing: next(slot)
[MarsEnv] r1 doing: next(slot)
[MarsEnv] r1 doing: next(slot)
[MarsEnv] r1 doing: pick(garb)
[MarsEnv] r1 doing: move_towards(3,3)
[MarsEnv] r1 doing: move_towards(3,3)
[MarsEnv] r1 doing: move_towards(3,3)
[MarsEnv] r1 doing: drop(garb)
[MarsEnv] r2 doing: burn(garb)
[MarsEnv] r1 doing: move_towards(3,0)
[MarsEnv] r1 doing: move_towards(3,0)
[MarsEnv] r1 doing: move_towards(3,0)
[MarsEnv] r1 doing: next(slot)
[MarsEnv] r1 doing: next(slot)
[MarsEnv] r1 doing: next(slot)
[MarsEnv] r1 doing: pick(garb)
[MarsEnv] r1 doing: move_towards(3,3)
[MarsEnv] r1 doing: move_towards(3,3)
[MarsEnv] r1 doing: move_towards(3,3)
[MarsEnv] r1 doing: drop(garb)
[MarsEnv] r2 doing: burn(garb)
[MarsEnv] r1 doing: move_towards(6,0)
[MarsEnv] r1 doing: move_towards(6,0)
[MarsEnv] r1 doing: move_towards(6,0)
[MarsEnv] r1 doing: next(slot)
[MarsEnv] r1 doing: next(slot)
[MarsEnv] r1 doing: next(slot)

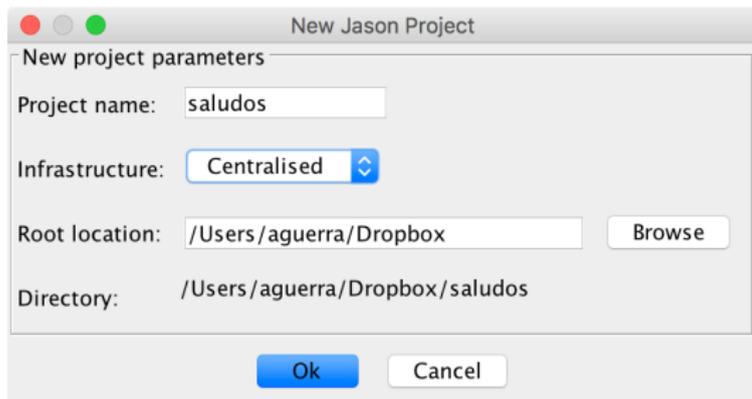
```

- ▶ En terminal (y Atom): `> jason mars.mas2j`



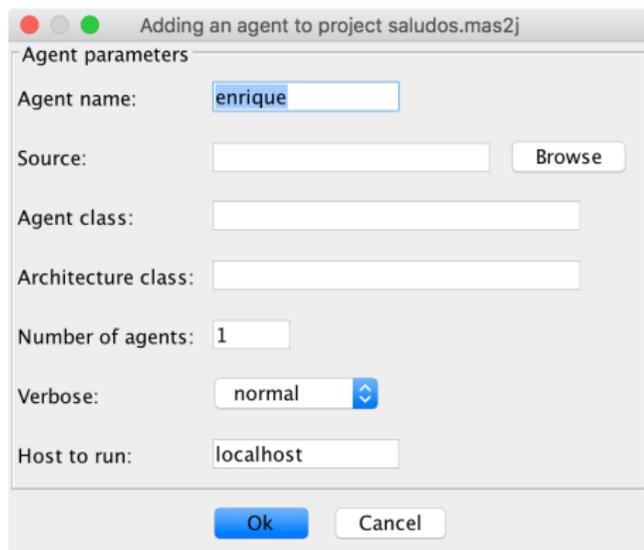
Proyecto nuevo

- ▶ Crearemos un proyecto con dos agentes **enrique** y **betto** que intercambian saludos.
- ▶ De click en el ícono de proyecto nuevo: 



Agregando agentes al proyecto

- ▶ Agregamos al agente enrique dando clic en 



Adding an agent to project saludos.mas2j

Agent parameters

Agent name:

Source:

Agent class:

Architecture class:

Number of agents:

Verbose:

Host to run:

- ▶ Hacemos lo mismo para beto



Código de los agentes

- ▶ Cuando un agente es creado, se le asigna un código por default.
- ▶ Modificaremos el código de `enrique` para que le envíe un saludo a `beto`.

```
1 // Agent enrique in project saludos.mas2j
2
3 /* Initial beliefs and rules */
4
5 /* Initial goals */
6
7 !start.
8
9 /* Plans */
10
11 +!start : true <- .send(beto,tell,hola).
```



Modificando a beto

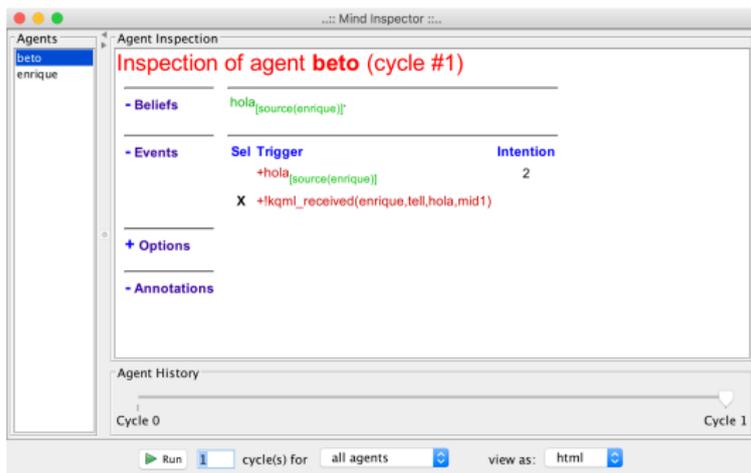
- ▶ Si queremos que beto responda al mensaje, agregamos un **plan** en su código:

```
1  // Agent beto in project saludos.mas2j
2
3  /* Initial beliefs and rules */
4
5  /* Initial goals */
6
7  /* Plans */
8
9  +hola[source(Ag)] <-
10  .print("Recibí un saludo de ", Ag);
11  .send(Ag,tell,hola).
```



Depuración del proyecto

- ▶ Dar clic al botón de debug 



The screenshot shows the Mind Inspector interface. The title bar reads "...: Mind Inspector :...". On the left, under "Agents", "beto" is selected. The main area is titled "Inspection of agent beto (cycle #1)". It displays the following information:

- Beliefs**: `hola[source(enrique)]`
- Events**:

Sel Trigger	Intention
<code>+hola[source(enrique)]</code>	2
<code>X +!kqml_received(enrique,tell,hola,mid1)</code>	
- + Options**
- Annotations**

Below the main area is the "Agent History" section, showing "Cycle 0" and "Cycle 1". At the bottom, there is a "Run" button, a field for "cycle(s) for" set to "1", a dropdown for "all agents", and a "view as:" dropdown set to "html".



Educando a enrique

- ▶ Hacemos lo mismo para enrique:

```
1  /* Initial goals */
2
3  !start.
4
5  /* Plans */
6
7  +!start : true <- .send(beto,tell,hola).
8
9  +hola[source(Ag)] <-
10   .print("Recibí un saludo de ", Ag);
11   .send(Ag,tell,hola).
```

- ▶ ¿Cómo es la corrida de estos agentes?



Sorpresas te da la vida

- ▶ No hay **ciclo** en la ejecución de nuestros agentes educados!



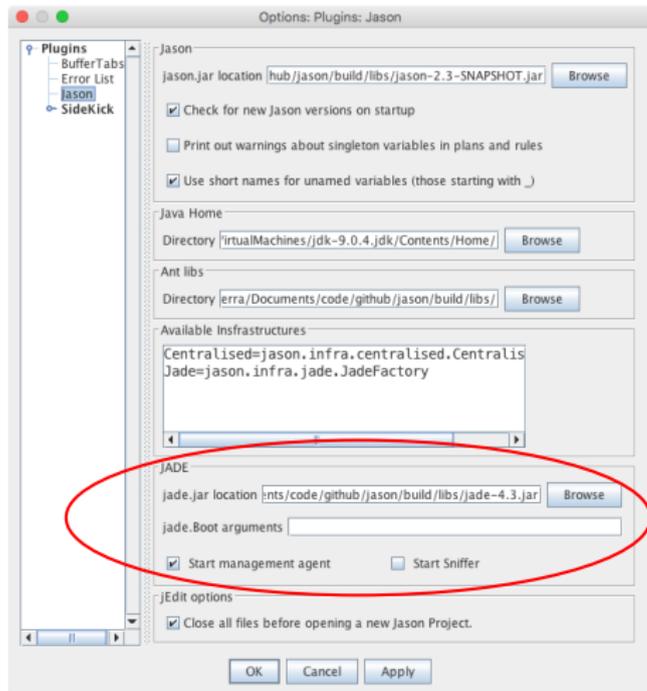
```
MAS Console - saludos
Jason Http Server running on http://192.168.1.82:3272
[beto] Recibí un saludo de enrique
[enrique] Recibí un saludo de beto
```

Buttons: Clean, Stop, Pause, Debug, New agent, Kill agent, New REPL agent

- ▶ La razón de ello es que cuando beto recibe el segundo saludo, la **creencia** asociada ya está en su base de creencias.
- ▶ Por lo tanto, no se genera ningún **evento** y por lo tanto
- ▶ Ningún **plan** es seleccionado para formar una **intención**.



Jade



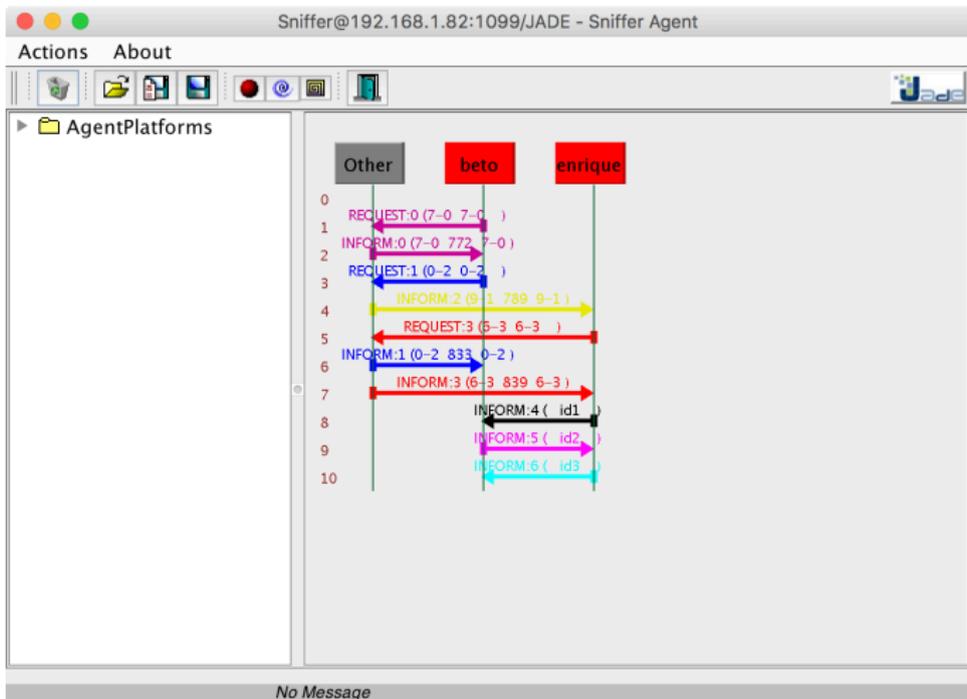
```

1  /* Jason Project */
2
3  MAS saludos {
4      infrastructure: Jade
5      agents:
6          enrique;
7          beto;
8  }

```



Jade: Sniffer



Creando un proyecto nuevo

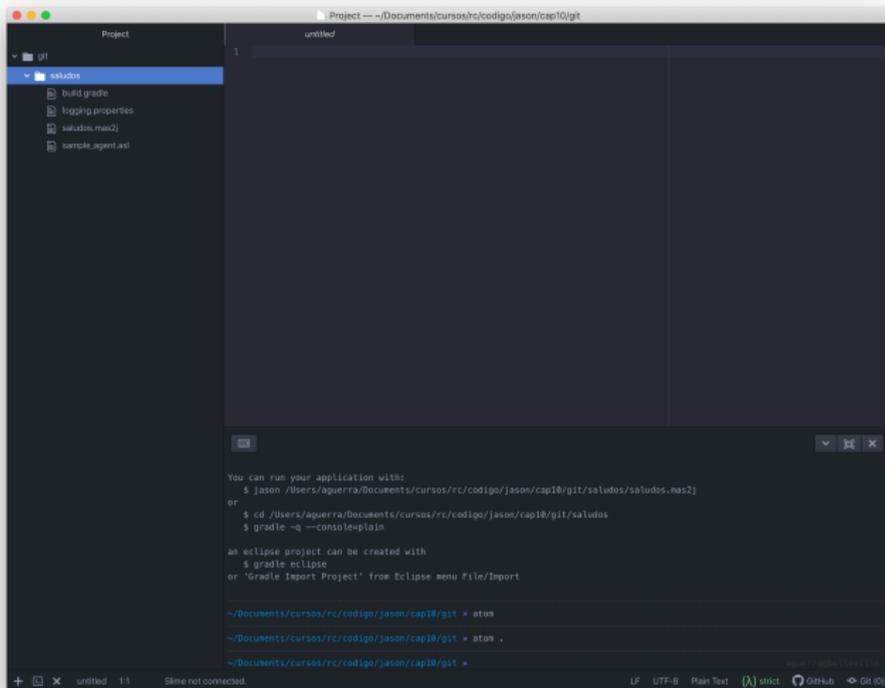
► Para crear el proyecto saludos:

```
1 > jason-new-project saludos
2 Creating new Jason project: saludos
3 Creating path /Documents/cursos/rc/codigo/jason/cap10/git/saludos
4
5 You can run your application with:
6 $ jason /Documents/cursos/rc/codigo/jason/cap10/git/saludos/saludos.mas2j
7 or
8 $ cd /Documents/cursos/rc/codigo/jason/cap10/git/saludos
9 $ gradle -q --console=plain
10
11 an eclipse project can be created with
12 $ gradle eclipse
13 or 'Gradle Import Project' from Eclipse menu File/Import
```



Explorando el proyecto con Atom

- Podemos abrir el proyecto nuevo con Atom:



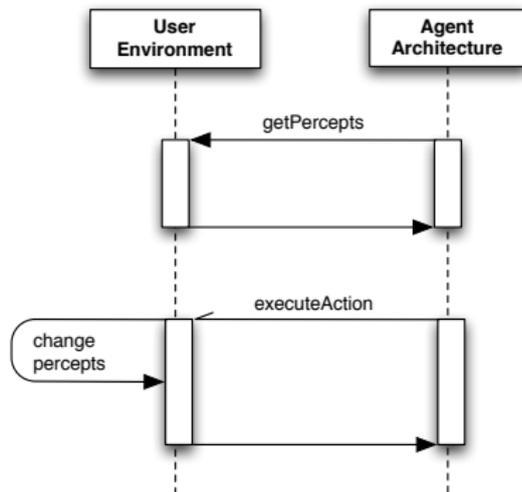
The screenshot shows the Atom text editor interface. On the left, a file explorer shows a project structure with a folder named 'git' containing files like 'build.gradle', 'logging.properties', 'saludos.mas2', and 'kernel_agent.txt'. The main editor area is currently blank. At the bottom, a terminal window displays the following text:

```
You can run your application with:  
$ jason /Users/aguerra/Documents/cursos/rc/codigo/jason/cap18/git/saludos/saludos.mas2  
or  
$ cd /Users/aguerra/Documents/cursos/rc/codigo/jason/cap18/git/saludos  
$ gradle -q --console=plain  
  
an eclipse project can be created with  
$ gradle eclipse  
or 'Gradle Import Project' from Eclipse menu File/Import  
  
~/Documents/cursos/rc/codigo/jason/cap18/git > atom  
~/Documents/cursos/rc/codigo/jason/cap18/git > atom .  
~/Documents/cursos/rc/codigo/jason/cap18/git >
```



Arquitectura y medio ambiente

- ▶ Los **agentes** y el **medio ambiente** son objetos independientes.
- ▶ La **arquitectura general** de un agente incluye los **métodos java** que definen la **interacción** con el ambiente, como se muestra en diagrama de secuencia UML:



La clase Environment

► Para cualquier ambiente:

```
1  import jason.asSyntax.*;
2  import jason.environment.*;
3
4  public class <EnvironmentName> extends Environment {
5      // Los miembros de la clase...
6
7      @Override
8      public void init(String[] args) {
9          // Qué hacer al iniciar la ejecución...
10     }
11
12     @Override
13     public boolean executeAction(String ag, Structure act) {
14         // Efectos de las acciones...
15     }
16
17     @Override
18     public void stop() {
19         // Qué hacer al detener el sistema...
20     }
21 }
```



¡Un piromaniaco en el ambiente!

- ▶ Crear un nuevo **proyecto** llamado **piros**
- ▶ Agregar un **agente** llamado **piro** con el siguiente código:

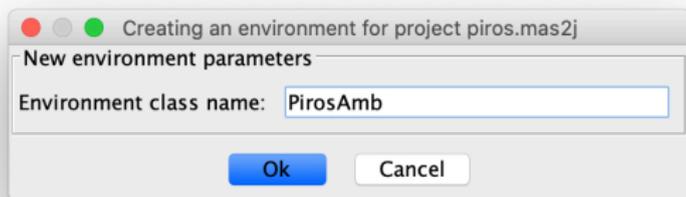
```
1 // Agent piro in project piros.mas2j
2
3 /* Initial beliefs and rules */
4
5 /* Initial goals */
6
7 !start.
8
9 /* Plans */
10
11 +!start : true <- incendiar.
12
13 +fuego <- .print("Fuego! Corran").
```

- ▶ **incendiar** aquí es una **acción externa**.



Creación del Medio Ambiente

- ▶ Dar clic en el botón de nuevo medio ambiente :



- ▶ Esto crea una plantilla de la clase `PirosAmb`, que se incluye como el ambiente de nuestro SMA:

```
1 | MAS piros {  
2 |  
3 |   infrastructure: Centralised  
4 |  
5 |   environment: PirosAmb
```



Métodos para implementar el medio ambiente

Método	Semántica
<code>addPercept(L)</code>	Agrega la literal L a la lista global de percepciones.
<code>addPercept(A,L)</code>	Agrega la literal L a las percepciones del agente A .
<code>removePercept(L)</code>	Remueve la literal L de la lista global de percepciones
<code>removePercept(A,L)</code>	Remueve la literal L de las percepciones del agente A .
<code>clearPercepts()</code>	Borra las percepciones de la lista global.
<code>clearPercepts(A)</code>	Borra las percepciones del agente A .



La clase PiroAmb completa I

```
1 // Environment code for project piro.mas2j
2
3 import jason.asSyntax.*;
4 import jason.environment.*;
5 import java.util.logging.*;
6
7 public class PiroAmb extends Environment {
8
9     private Logger logger = Logger.getLogger("piros.mas2j."+PiroAmb.class.getName());
10
11     @Override
12     public void init(String[] args) {
13         super.init(args);
14     }
15
16     @Override
17     public boolean executeAction(String agName, Structure action) {
18         if (action.getFunctor().equals("incendiar")) {
19             addPercept(Literal.parseLiteral("fuego"));
20             return true;
21         } else {
22             logger.info("executing: "+action+", but not implemented!");
23             return false;
24         }
25     }
26
27     @Override
28     public void stop() {
```



La clase PiroAmb completa II

```
29 |     super.stop();  
30 | }  
31 | }
```



A correr

- ▶ Ahora la agente piro puede responder a los cambios en su ambiente:

```
1 | // Agent piro in project piros.mas2j
2 |
3 | /* Initial beliefs and rules */
4 |
5 | /* Initial goals */
6 |
7 | !start.
8 |
9 | /* Plans */
10 |
11 | +!start : true <- incendiar.
12 |
13 | +fuego <- .print("Fuego! Corran").
```

- ▶ Corrida:

```
1 | Jason Http Server running on http://192.168.1.148:3273
2 | [piro] Fuego! Corran
```



Creencias

- ▶ De cierta forma, las creencias de Jason y las metas verificables ($?\alpha$) se comportan de manera muy similar a un sistema de Programación Lógica, p. ej., Prolog [11, 6, 7].
- ▶ Para ilustrar esto vamos a crear un nuevo proyecto en Eclipse, llamado creencias, con una infraestructura centralizada y sin un medio ambiente asociado.
- ▶ Si todo va bien, el navegador de Jason, debe mostrar el proyecto que incluye al agente `sample_agent`.



La familia I

- Modifiquemos este agente para incluir creencias sobre una familia:

```
1 // Agent agent1 in project creencias
2
3 /* Initial beliefs and rules */
4
5 progenitor(carmelo,alejandro).
6 progenitor(carmen,alejandro).
7 progenitor(carmelo,laura).
8 progenitor(carmen,laura).
9 progenitor(laura,rafael).
10 progenitor(isidro,rafael).
11
12 /* Initial goals */
13
14 !start.
15
16 /* Plans */
17
18 +!start <-
19     ?progenitor(laura,rafael);
20     .print("Laura es progenitor de Rafael");
21     ?progenitor(carmelo,Y);
22     .print("Carmelo es progenitor de ", Y, ".");
23     ?progenitor(X,rafael);
24     .print(X, " es un progenitor de Rafael").
```



Consultas

- ▶ A diferencia de Prolog, es el agente y no el usuario quien hace las **preguntas**. Para ello se define el **plan** que responde a la meta `start`.
- ▶ Su primera acción **verifica** si un hecho es verdadero (que Laura es progenitor de Rafael); y luego se hacen dos preguntas más para saber de quién es progenitor Carmelo y quién es progenitor de Rafael.
- ▶ La acción interna `.print`, imprime mensajes en consola.
- ▶ La salida del programa sería:

```
1 | [agent1] Laura es progenitor de Rafael
2 | [agent1] Caramelo es progenitor de alejandro.
3 | [agent1] laura es un progenitor de rafael
```



Fallo

- ▶ Si una pregunta **falla**, el plan falla y la intención asociada también.
- ▶ **Ej.** Si agregamos la meta verificable `?madre(laura,rafael)` al final del plan del agente, tendremos un fallo, ya que tal meta no puede ser resuelta:

```
1 [agent1] Laura es progenitor de Rafael
2 [agent1] Caramelo es progenitor de alejandro.
3 [agent1] laura es un progenitor de rafael
4 [agent1] No failure event was generated for +!start[code(madre(laura,
5 | rafaef)),code_line(25),code_src("../creencias/src/asl/sample_agent.asl"),
6 | error(test_goal_failed), error_msg("Failed to test '?madre(laura,rafael)')",
7 | source(self)]
```

- ▶ La línea 4 reporta el error. ¿Qué nos dice?



Fallos y etiquetas

- ▶ Observen el uso de las etiquetas para registrar el fallo.
- ▶ Ej. La meta `!start` falló, debido a que una meta verificable `?madre(laura,rafael)` ha fallado.
- ▶ Hay varios términos en las etiquetas del evento de fallo:

Término	Semántica
<code>code(C)</code>	<i>C</i> es el elemento del programa que causó el fallo.
<code>code_src(Asl)</code>	<i>Asl</i> es el programa de agente que falló.
<code>code_line(L)</code>	El error se produjo en la línea <i>L</i> .
<code>error(X)</code>	El error <i>X</i> se produjo.
<code>error_msg(Msg)</code>	<i>Msg</i> es el mensaje que será desplegado en consola para señalar el error.



Procesamiento de errores I

- ▶ Esta información puede ser usada al agregar planes relevantes ($\neg!\alpha$), para **contender con el error**:

```
1 | #!start[error(Error)] <-  
2 | .print("El plan +!start falló por el error ", Error).
```

con lo que el error es procesado adecuadamente:

```
1 | [agent1] Laura es progenitor de Rafael  
2 | [agent1] Carmelo es progenitor de alejandro.  
3 | [agent1] laura es un progenitor de Rafael  
4 | [agent1] El plan +!start falló por el error test_goal_failed
```



Agregando conocimiento I

- ▶ En realidad, querríamos agregar conocimiento al agente para contender con la meta problemática, en lugar de procesar el fallo.
- ▶ Agregar conocimiento, significa agregar creencias al agente, incluyendo reglas:

```
1  // Agent agent3 in project creencias
2
3  /* Initial beliefs and rules */
4
5  progenitor(carmelo,alejandro).
6  progenitor(carmen,alejandro).
7  progenitor(carmelo,laura).
8  progenitor(carmen,laura).
9  progenitor(laura,rafael).
10 progenitor(isidro,laura).
11
12 mujer(laura).
13 mujer(carmen).
14 hombre(carmelo).
15 hombre(alejandro).
16 hombre(isidro).
17
18 madre(X,Y) :- mujer(X) & progenitor(X,Y).
19 padre(X,Y) :- hombre(X) & progenitor(X,Y).
```



Agregando conocimiento II

```
20
21  /* Initial goals */
22
23  !start.
24
25  /* Plans */
26
27  +!start <-
28    ?progenitor(laura,rafael);
29    .print("Laura es progenitor de Rafael");
30    ?progenitor(carmelo,Y);
31    .print("Caramelo es progenitor de ", Y, ".");
32    ?progenitor(X,rafael);
33    .print(X," es un progenitor de Rafael");
34    ?madre(laura,rafael);
35    .print("Laura es madre de Rafael");
36    ?madre(Z,alejandro);
37    .print(Z, " es madre de Alejandro").
38
39  -!start[error(Error)] <-
40    .print("El plan +!start falló por el error ", Error).
```

► La salida es la siguiente:



Agregando conocimiento III

```
1 [agente3] Laura es progenitor de Rafael
2 [agente3] Caramelo es progenitor de alejandro.
3 [agente3] laura es un progenitor de Rafael
4 [agente3] Laura es madre de Rafael
5 [agente3] carmen es madre de Alejandro
```



Reglas recursivas I

- ▶ Por supuesto que las reglas pueden ser recursivas, por ejemplo:

```
1 // Agent agent4 in project creencias
2
3 /* Initial beliefs and rules */
4
5 progenitor(carmelo,alejandro).
6 progenitor(carmen,alejandro).
7 progenitor(carmelo,laura).
8 progenitor(carmen,laura).
9 progenitor(laura,rafael).
10 progenitor(isidro,rafael).
11
12 mujer(laura).
13 mujer(carmen).
14 hombre(carmelo).
15 hombre(alejandro).
16 hombre(isidro).
17
18 madre(X,Y) :- mujer(X) & progenitor(X,Y).
19 padre(X,Y) :- hombre(X) & progenitor(X,Y).
20
21 ancestro(X,Y) :- progenitor(X,Y).
22 ancestro(X,Y) :- progenitor(X,Z) & progenitor(Z,Y).
23
24 /* Initial goals */
25
```



Reglas recursivas II

```
26 | !start.  
27 |  
28 | /* Plans */  
29 |  
30 | +!start <-  
31 |   ?ancestro(carmelo,rafael);  
32 |   .print("Carmelo es un ancestro de Rafael");  
33 |   ?ancestro(X,rafael);  
34 |   .print(X, " es un ancestro de Rafael");  
35 |   .findall(Xs, ancestro(Xs,rafael),L);  
36 |   .print("Los ancestros de Rafael son ",L).
```

Con la siguiente salida:

```
1 | [agente4] Carmelo es un ancestro de Rafael  
2 | [agente4] laura es un ancestro de Rafael  
3 | [agente4] Los ancestros de Rafael son [laura, isidro, carmelo, carmen]
```



Metapredicados I

- ▶ La **acción interna** `.findall` se usa igual que en Prolog, para coleccionar todas las respuestas posibles a una meta dada.
- ▶ La acción interna `.setof` hace lo mismo, pero sin incluir soluciones repetidas, construyendo el conjunto solución de manera incremental.
- ▶ El primer argumento de estas acciones es un patrón que representa la forma en que los resultados serán recolectados.
- ▶ **Ej.** Si sustituimos `Xs` por `ancestro(Xs)` en la línea 35, obtendríamos una lista de estos.

```

1 | [agente4] Los ancestros de Rafael son [ancestro(laura), ancestro(isidro),
2 |   ancestro(carmelo), ancestro(carmen)]

```

- ▶ El segundo argumento de estas acciones es la meta a resolver.
- ▶ Su tercer argumento es una **lista**, donde los resultados son recolectados.



Listas I

- ▶ Las listas se representan igual que en Prolog.
- ▶ La lista **vacía** se denota como `[]`.
- ▶ La lista que tiene una **cabeza** `X` y una **cola** `[Xs]` se denota como `[X|Xs]`.
- ▶ **Ej.** Veamos un ejemplo de búsqueda en una lista.

```
1 // Agent agente5 in project creencias
2
3 /* Initial beliefs and rules */
4
5 busqueda(X,[X|_]).
6 busqueda(X,[Y|Ys]) :- busqueda(X,Ys).
7
8 /* Initial goals */
9
10 !start.
11
12 /* Plans */
13
14 +!start : true <-
15     Lista = [1,2,3,4,5];
16     ?busqueda(3,Lista);
```



Listas II

```
17 | .print("3 es miembro de la lista ",Lista);  
18 | .findall(X,busqueda(X,Lista),L);  
19 | .print("Los miembros de la Lista son ",L).
```

► Cuya salida en consola es:

```
1 | [agente5] 3 es miembro de la lista [1,2,3,4,5]  
2 | [agente5] Los miembros de la Lista son [1,2,3,4,5]
```



Ejemplo 1

- ▶ *elimina/3* el tercer argumento es la lista resultante de eliminar el primer argumento del segundo (una lista).

```
1 // Agent agente6 in project creencias
2
3 /* Initial beliefs and rules */
4
5 busqueda(X,[X|_]).
6 busqueda(X,[Y|Ys]) :- busqueda(X,Ys).
7
8 elimina(X,[X|Xs],Xs).
9 elimina(X,[Y|Ys],[Y|Zs]) :- elimina(X,Ys,Zs).
10
11 /* Initial goals */
12
13 !start.
14
15 /* Plans */
16
17 +!start : true <-
18   Lista = [1,2,3,4,5];
19   .print("La lista original es ",Lista);
20   ?elimina(3,Lista,Resultado);
21   .print("Eliminar 3 de la lista resulta en ",Resultado).
```



Ejemplo II

Cuya salida es:

```
1 | [agente6] La lista original es [1,2,3,4,5]
2 | [agente6] Eliminar 3 de la lista resulta en [1,2,4,5]
```



Acciones internas para listas I

Acción interna	Descripción
<code>.member(X, Xs)</code>	X es miembro de Xs .
<code>.length(X, L)</code>	La longitud de X es L .
<code>.empty(X)</code>	X es una lista vacía.
<code>.concat(L₁, ..., L_n)</code>	Concatena todas las listas en L_n .
<code>.delete(X, L, R)</code>	Elimina X de L resultando la lista R .
<code>.reverse(L, R)</code>	La lista R es el reverso de L .
<code>.shuffle(L, R)</code>	R es la lista L revuelta.
<code>.nth(N, L, R)</code>	R es en N -ésimo elemento de la lista L .
<code>.max(L, R)</code>	R es el máximo elemento de la lista L .
<code>.min(L, R)</code>	R es el mínimo elemento de la lista L .
<code>.sort(L, R)</code>	R es la lista resultante de ordenar L .
<code>.list(L)</code>	Verifica si L es una lista.



Acciones internas para listas II

Acción interna	Descripción
<i>.suffix</i> (R, L)	R es un sufijo de la lista L .
<i>.prefix</i> (R, L)	R es un prefijo de la lista L .
<i>.sublist</i> (R, L)	R es una sub-lista de la lista L .
<i>.difference</i> (L_1, L_2, R)	R es la diferencia entre L_1 y L_2 .
<i>.intersection</i> (L_1, L_2, R)	R es la intersección de L_1 y L_2 .
<i>.union</i> (L_1, L_2, R)	R es la unión de L_1 y L_2 .



Ejemplos I

- ▶ El siguiente agente prueba muchas de las acciones para listas:

```

1 // Agent agente7 in project creencias
2
3 /* Initial beliefs and rules */
4
5 /* Initial goals */
6
7 !start.
8
9 /* Plans */
10
11 +!start : true <-
12   Lista1 = [1,2,3,4,5];
13   Lista2 = [a,b,c,d,e];
14   .print("La lista 1 es ",Lista1);
15   .print("La lista 2 es ",Lista2);
16   .member(X,Lista1);
17   .print(X, " es miembro de la lista 1");
18   .length(Lista1,Long);
19   .print("La longitud de la lista 1 es ",Long);
20   .concat(Lista1,Lista2,L3);
21   .print("Pegar la lista 1 y 2 nos da ",L3);
22   .delete(X,Lista1,L4);
23   .print("Borrar ",X," de la lista 1, nos da ",L4," Oops!");
24   .delete(c,Lista2,L5);
25   .print("Borrar c de la lista 2 no es problema ",L5);

```



Ejemplos II

```

26 | .shuffle(Lista1,L6);
27 | .print("Revolver la lista 1 produce ",L6);
28 | .reverse(Lista2,L7);
29 | .print("Invertir la lista 2 ",L7);
30 | .nth(Long-1,Listal,Last);
31 | .print("El último elemento de la lista 1 es ",Last);
32 | .max(Lista1,MaxL1);
33 | .print("El máximo elemento en la lista 1 es ",MaxL1);
34 | .min(Lista2,MinL2);
35 | .print("El mínimo elemento de la lista 2 es ",MinL2);
36 | .sort(L6,L8);
37 | .print("Ordenar la lista 1 revuelta resulta en ",L8).

```

► Cuya salida se muestra a continuación:

```

1 | [agente7] La lista 1 es [1,2,3,4,5]
2 | [agente7] La lista 2 es [a,b,c,d,e]
3 | [agente7] 1 es miembro de la lista 1
4 | [agente7] La longitud de la lista 1 es 5
5 | [agente7] Pegar la lista 1 y 2 nos da [1,2,3,4,5,a,b,c,d,e]
6 | [agente7] Borrar 1 de la lista 1, nos da [1,3,4,5] Ooops!
7 | [agente7] Borrar c de la lista 2 no es problema [a,b,d,e]
8 | [agente7] Revolver la lista 1 produce [3,5,4,1,2]
9 | [agente7] Invertir la lista 2 [e,d,c,b,a]
10 | [agente7] El último elemento de la lista 1 es 5
11 | [agente7] El máximo elemento en la lista 1 es 5
12 | [agente7] El mínimo elemento de la lista 2 es a

```



Ejemplos III

```
13 | [agente7] Ordenar la lista 1 revuelta resulta en [1,2,3,4,5]
```



Observaciones

- ▶ Las acciones internas **no son** creencias del agente, como si lo son las reglas y los hechos, ejemplificados anteriormente.
- ▶ Las acciones internas son operaciones implementadas en **Java**, que no afectan el medio ambiente del agente.
- ▶ En principio, deberían ser más **eficientes** que sus contrapartes implementadas à la Prolog, aunque no son explotables al usar **actos de habla**.
- ▶ Al no ser cláusulas, la semántica de estas operaciones no se sigue de *AgentSpeak(L)*, sino de su implementación en Java: Todas son **booleanos**.



Ejemplo I

- ▶ Consideren *.delete*
- ▶ El primer argumento de esta operación puede ser un término, una cadena de texto, o un número; y su comportamiento **dependía** del tipo de argumento recibido de forma poco afortunada: Si queremos borrar las ocurrencias de 1 en una lista de números, esta acción no nos sirve, pues en realidad borrará el segundo elemento de la lista al ser su primer argumento un número.
- ▶ El siguiente agente define una cláusula *del* que borra **todas** las ocurrencias de un término, número o no, en una lista.



Ejemplo II

```

1 // Agent agente8 in project creencias
2
3 /* Initial beliefs and rules */
4
5 del(_, [], []).
6 del(X, [X|L1], L2) :- del(X, L1, L2).
7 del(X, [H|L1], [H|L2]) :- X\==H & del(X, L1, L2).
8
9 /* Initial goals */
10
11 !start.
12
13 /* Plans */
14
15 +!start : true <-
16     Lista = [1,2,3,2,4,2,5];
17     .delete(1,Lista,R1);
18     .print("Eliminar 1 de la lista ",Lista," resulta en ",R1);
19     ?del(2,Lista,R2);
20     .print("Eliminar 2 de la lista ",Lista," resulta en ",R2).

```

Su salida en consola es:

```

1 [agente8] Eliminar 1 de la lista [1,2,3,2,4,2,5] resulta en [1,3,2,4,2,5]
2 [agente8] Eliminar 2 de la lista [1,2,3,2,4,2,5] resulta en [1,3,4,5]

```



Acciones internas aritméticas

Acciones internas aritméticas

<i>math.abs(N)</i>	<i>math.acos(N)</i>	<i>math.asin(N)</i>	<i>math.atan(N)</i>
<i>math.average(L)</i>	<i>math.cell(N)</i>	<i>math.cos(N)</i>	<i>.count(B)</i>
<i>math.e</i>	<i>math.floor(N)</i>	<i>.length(L)</i>	<i>math.log(N)</i>
<i>math.max(N₁, N₂)</i>	<i>math.min(N₁, N₂)</i>	<i>math.pi</i>	<i>math.random(N)</i>
<i>math.round(N)</i>	<i>math.sin(N)</i>	<i>math.sqrt(N)</i>	<i>math.std_dev(L)</i>
<i>math.sum(L)</i>	<i>math.tan(N)</i>	<i>system.time</i>	



Ejemplo I

- ▶ El siguiente agente hace uso de algunas funciones aritméticas:

```
1 // Agent agente9 in project creencias
2
3 /* Initial beliefs and rules */
4
5 /* Initial goals */
6
7 !start.
8
9 /* Plans */
10
11 +!start : true <-
12   Lista1 = [1,2,3,4,5];
13   .print("La lista 1 es ",Lista1);
14   .print("La longitud de la lista 1 ", .length(Lista1));
15   .print("La sumatoria de la lista 1 es ", math.sum(Lista1));
16   .print("El promedio de la lista 1 es ", math.average(Lista1)).
```

- ▶ Su salida en consola es:

```
1 [agente9] La lista 1 es [1,2,3,4,5]
2 [agente9] La longitud de la lista 1 es 5
3 [agente9] La sumatoria de la lista 1 es 15
4 [agente9] El promedio de la lista 1 es 3
```



Ejemplo I

- ▶ El siguiente agente trabaja con **árboles binarios**.

```

1 // Agent agente10 in project creencias
2
3 /* Initial beliefs and rules */
4
5 insertaArbol(X,vacio,arbol(X,vacio,vacio)).
6
7 insertaArbol(X,arbol(X,A1,A2),arbol(X,A1,A2)).
8
9 insertaArbol(X,arbol(Y,A1,A2),arbol(Y,A1N,A2)) :-
10     X<Y & insertaArbol(X,A1,A1N).
11
12 insertaArbol(X,arbol(Y,A1,A2),arbol(Y,A1,A2N)) :-
13     X>Y & insertaArbol(X,A2,A2N).
14
15 creaArbol([],A,A).
16
17 creaArbol([X/Xs],AAux,A) :-
18     insertaArbol(X,AAux,A2) &
19     creaArbol(Xs,A2,A).
20
21 lista2arbol(Xs,A) :- creaArbol(Xs,vacio,A).
22
23 nodos(vacio, []).
24
25 nodos(arbol(X,A1,A2),Xs) :-

```



Ejemplo II

```

26  nodos(A1,Xs1) &
27  nodos(A2,Xs2) &
28  .concat(Xs1,[X|Xs2],Xs).
29
30  ordenaLista(L1,L2) :-
31  lista2arbol(L1,A) &
32  nodos(A,L2).
33
34  /* Initial goals */
35
36  !start.
37
38  /* Initial plans */
39
40  +!start <-
41  Lista1 = [5,3,4,1,2];
42  ?lista2arbol(Lista1,Arbol1);
43  .print("La lista 1 es ", Lista1);
44  .print("El árbol creado de la lista es ",Arbol1);
45  ?nodos(Arbol1,Nodos1);
46  .print("Cuyos nodos en orden son ",Nodos1).

```

► Su salida en consola es la siguiente:



Ejemplo III

```
1 [agente10] La lista 1 es [5,3,4,1,2]
2 [agente10] El árbol creado de la lista es arbol(5,arbol(3,arbol(1,
3 vacio,arbol(2,vacio,vacio)),arbol(4,vacio,vacio)),vacio)
4 [agente10] Cuyos nodos en orden son [1,2,3,4,5]
```



Anotaciones

- ▶ Todas las creencias de Jason tienen al menos una **anotación** asociada, su fuente.
- ▶ En el inspector de mentes podrán ver esto: `source(self)`.

```
Agent Inspection
Inspection of agent agente1
- Beliefs
progenitor(carmelo,alejandro){source(self)}
progenitor(carmen,alejandro){source(self)}
progenitor(carmelo,laura){source(self)}
progenitor(carmen,laura){source(self)}
progenitor(laura,rafael){source(self)}
progenitor(isidro,laura){source(self)}
```



Sintaxis y semántica

- ▶ Las anotaciones no cambian el poder expresivo del lenguaje de programación, pero mejoran su legibilidad.
- ▶ Su sintaxis es la de una lista de **términos**. Por ejemplo:

```
1 | p(t) [source(self), costo(10), prioritario]
```

- ▶ puede representar que la literal $p(t)$ ha sido agregada a las creencias por el agente mismo, tiene un costo de 10 unidades y se trata de algo prioritario. Observen que todo ello es meta-información.
- ▶ Aunque la sintaxis de las anotaciones se corresponde con la de una lista de términos, en realidad su semántica es la de un **conjunto** y así es como son consideradas por Jason.



Unificación y anotaciones

- ▶ El uso de las anotaciones introduce una restricción al computar el unificador más general entre dos literales. L_1 unifica con L_2 si y sólo si las anotaciones de L_1 son un subconjunto de las de L_2 .
- ▶ Ejemplo:

```
1 | p(t) = p(t)[a1];           // Unifica
2 | p(t)[a1] = p(t);         // No unifica
3 | p(t)[a2] = p(t)[a1,a2,a3] // Unifica
```



Las anotaciones son listas

- ▶ Como las anotaciones son listas que representan conjuntos, la notación de **acceso a listas** para cabeza y cola puede usarse:

```
1 | p(t)[a2/As] = p(t)[a1,a2,a3] // As unifica con [a1,a3]
2 | p(t)[a1,a2,a3] = p(t)[a1,a4/As] // As unifica con [a2,a3]
```



Variables

- ▶ La unificación entre **variables** debe considerar los diversos casos de unificación para $X[As] = Y[Bs]$; y si las variables en cuestión son de base o no.
- ▶ Cuando X e Y son de base:

```

1 | X      = p[Cs] // unifica X con p[Cs]
2 | Y      = p[Ds] // unifica Y con p[Ds]
3 | X[As] = Y[Bs] // unifica si (Cs ∪ As) ⊂ (Ds ∪ Bs)

```

- ▶ Ejemplo:

```

1 | X = p[a1, a2];
2 | Y = p[a1, a3];
3 | X[a4] = Y[a2, a4, a5]; // unifica

```



Casos de base

- ▶ Cuando solo X es de base, la unificación se resuelve de la siguiente forma:

$$\begin{array}{l|l}
 1 & X = p[Cs] \\
 2 & X[As] = Y[Bs] \quad // \text{unifica si } (Cs \cup As) \subset Bs \\
 3 & \quad \quad \quad // \text{e } Y \text{ unifica con } p
 \end{array}$$

- ▶ Cuando solo Y es de base, la unificación se resuelve de la siguiente forma:

$$\begin{array}{l|l}
 1 & Y = p[Ds] \\
 2 & X[As] = Y[Bs] \quad // \text{unifica si } As \subset (Ds \cup Bs) \\
 3 & \quad \quad \quad // \text{y } X \text{ unifica con } p
 \end{array}$$



Negaciones

- ▶ A diferencia de Prolog, donde el **principio del mundo cerrado** (*CWA*) se adopta automáticamente, Jason puede contender también con una representación **fuerte** de la negación.
- ▶ Recuerden que el *CWA* expresa que todo lo que no se sabe cierto, o no es derivable de lo que se sabe cierto siguiendo las reglas del programa, es falso. En este sentido, Jason provee el operador **débil** `not`, donde la negación de una fórmula es cierta, si el intérprete falla al derivar dicha fórmula.



Negación fuerte

- ▶ El operador de **negación fuerte** es utilizado para representar que el agente explícitamente cree que cierta fórmula no es el caso.
- ▶ La semántica de las negaciones, cuando se aplican a **literales**, es como sigue:

Sintaxis	Semántica
I	El agente cree que I es verdadera
$\sim I$	El agente cree que I es falsa
$not\ I$	El agente no cree que I es verdadera
$not\ \sim I$	El agente no cree que I es falsa.



Ejemplo I

- ▶ El agente11 cree que la *caja1* es *roja*, pero según *beto* la *caja1* es verde.
- ▶ Para complicar más la historia, según *enrique* la *caja1* no es verde.
- ▶ La meta principal del agente11 es reportar de que color es la caja.

```

1  // Agent agente11 in project creencias
2
3  /* Initial beliefs and rules */
4
5  color(caja1,verde)[source(beto)].
6  -color(caja1,verde)[source(enrique)]. // azul no causa contradicción
7  color(caja1,rojo). // verde hace que enrique sea el mentiroso
8
9  colorSegunYo(Caja,Color) :-
10     color(Caja,Color)[source(Src)] &
11     (Src == self | Src == percept).
12
13  descr(Ag,mentiroso) :-
14     mentiroso(Ag)[cert(C1)] &
15     daltonico(Ag)[cert(C2)] &
16     C1 > C2.
17  descr(Ag,daltonico) :- daltonico(Ag).
18  descr(Ag,confiable).
19

```



Ejemplo II

```

20  /* Initial goals */
21
22  !start.
23
24  /* Plans */
25
26  @contradiccion
27  +!start : color(caja1,Color) & ~color(caja1,Color)[source(S2)] <-
28  .print("Contradicción detectada");
29  ?color(caja1,Color1)[source(S1)];
30  .print("La caja1 es de color ",Color1,", según ",S1);
31  ?colorSegunYo(caja1,Color2);
32  .print("Aparentemente el color de la caja1 es ",Color2,", según yo");
33  if (Color1 \== Color2) {
34      +mentiroso(S1)[cert(0.7)]; // Invertir y beto será mentiroso
35      +daltonico(S1)[cert(0.3)];
36  } else {
37      +mentiroso(S2)[cert(0.3)];
38      +daltonico(S2)[cert(0.7)];
39  };
40  ?descr(S1,Des1);
41  .print(S1, " es un agente ", Des1);
42  ?descr(S2,Des2);
43  .print(S2, " es un agente ", Des2).
44
45  @sinContradiccion
46  +!start <-
47  .print("No hay contradicciones detectadas");

```



Ejemplo III

```
48 | ?colorSegunYo(caja1,Color);  
49 | .print("La caja1 es de color ",Color,",", según yo").
```

- ▶ Hay dos planes para contender con la meta principal del agente. El primero detecta contradicciones y el segundo no.
- ▶ En el segundo plan, el agente se pregunta por el color de la caja desde su propia perspectiva (la fuente es `self` o `percept`) y reporta el color encontrado.
- ▶ Cuando la contradicción es detectada el agente confronta la situación. Reporta el color según su perspectiva y ajusta cuentas con los otros agentes.
- ▶ Si hay otro agente reportando un color diferente, nuestro agente creerá que tal agente es mentiroso o daltónico, con cierto grado de certidumbre.



Ejemplo IV

- ▶ En caso contrario, hay un tercer agente causando la contradicción y éste es el mentiroso/daltónico.
- ▶ La salida en consola para este caso es:

```

1 | [agente11] Contradicción detectada [agente11] La caja1 es de color verde,
2 | segun beto [agente11] Apparently the color of the box1 is red, según yo
3 | [agente11] beto es un agente daltonico [agente11] Enrique es un agente
4 | confiable

```

- ▶ Si cambiamos la información sobre el color de la *caja1* provista por *enrique* a *azul* (línea 6), tendremos que ya no hay contradicción detectable y la salida del programa es la siguiente:

```

1 | [agente11] No hay contradicciones detectadas [agente11] La caja1 es de color
2 | rojo, según yo

```

- ▶ En cambio si nuestro agente creyera que la *caja1* es de color *verde* (línea 7), entonces el daltónico resultaría *enrique*:



Ejemplo V

```

1 | [agente11] Contradicción detectada [agente11] La caja1 es de color verde,
2 | segun beto [agente11] Aparentemente el color de la caja1 es verde, según yo
3 | [agente11] beto es un agente confiable [agente11] enrique es un agente
4 | daltonico

```

- ▶ Si se invierten los grados de certeza (líneas 34 y 35), resultará que *beto* es *mentiroso* en lugar de *daltonico*.

```

1 | [agente11] Contradicción detectada [agente11] La caja1 es de color verde,
2 | según beto [agente11] Aparentemente el color de la caja1 es roja, según yo
3 | [agente11] beto es un agente mentiroso [agente11] enrique es un agente
4 | confiable

```



Acciones internas personalizadas

- ▶ Es posible definir acciones internas personalizadas, similares a las que hemos introducido en la sección anterior, por ejemplo `math.abs`, etc.
- ▶ Se sugiere que las acciones estén organizadas en **librerías**, que son paquetes de Java; mientras que las acciones propiamente dichas, son clases de Java que implementan la interfaz `InternalAction`.
- ▶ Jason provee una implementación por defecto de esta interfaz, conocida como `DefaultInternalAction`.
- ▶ Las acciones internas se denotan como `librería.acción`.



Calculando distancias I

- ▶ Vamos a crear un SMA centralizado con un solo agente:

```
1 MAS distancia {  
2  
3   infrastructure: Centralised  
4  
5   agents:  
6     agent1 sample_agent;  
7  
8   aslSourcePath:  
9     "src/asl";  
10 }
```

- ▶ En donde el agente agent1 haga uso de una acción interna para calcular la distancia euclidibeto entre dos puntos:



Calculando distancias II

```
1 // Agent sample_agent in project distancia
2
3 /* Initial beliefs and rules */
4
5 /* Initial goals */
6
7 !start.
8
9 /* Plans */
10
11 +!start : true <-
12   ia.distancia(10,10,20,50,D);
13   .println("La distancia euclidiana entre (10,10) y (20,50) es ",D).
```

- ▶ Los ambientes de desarrollo de Jason, permiten agregar acciones internas al sistema. Detalles más, detalles menos, lo importante es decirle al ambiente de desarrollo en que paquete será incluida la acción interna.
- ▶ La implementación de la acción es como sigue:



Calculando distancias III

```
1 // Internal action code for project distancia
2
3 package ia;
4
5 import jason.*;
6 import jason.asSemantics.*;
7 import jason.asSyntax.*;
8
9 /**
10  * @author aguerra
11  * ia.distancia: Comuta la distancia euclidiana entre dos puntos.
12  */
13
14 public class distancia extends DefaultInternalAction {
15
16     private static final long serialVersionUID = 1L;
17
18     @Override
19     public Object execute(TransitionSystem ts, Unifier un, Term[] args)
20         throws Exception {
21         ts.getAg().getLogger().info("executing internal action 'distancia'");
22         try{
23             NumberTerm x1 = (NumberTerm) args[0];
24             NumberTerm y1 = (NumberTerm) args[1];
25             NumberTerm x2 = (NumberTerm) args[2];
26             NumberTerm y2 = (NumberTerm) args[3];
27
28             double distance = Math.abs(x1.solve()-x2.solve()) +
```



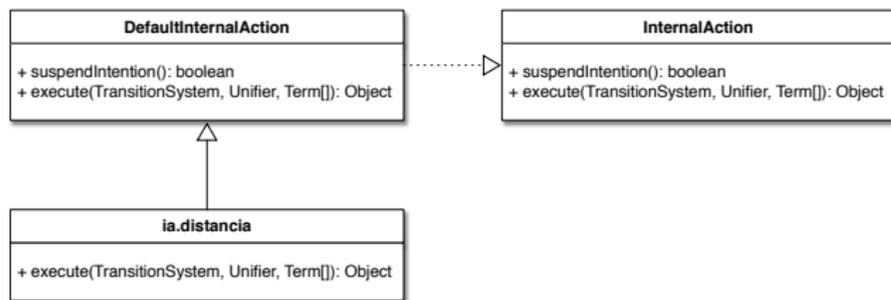
Calculando distancias IV

```
29         Math.abs(y1.solve()-y2.solve());
30
31     NumberTerm result = new NumberTermImpl(distance);
32     return un.unifies(result, args[4]);
33 } catch (ArrayIndexOutOfBoundsException e) {
34     throw new JasonException("La acción interna 'distancia'+
35         "no ha recibido cinco argumentos!");
36 } catch (ClassCastException e) {
37     throw new JasonException("La acción interna 'distancia'+
38         "ha recibido argumentos no numéricos!");
39 } catch (Exception e) {
40     throw new JasonException("Error en 'distancia'");
41 }
42 }
43 }
```

- El diagrama de clases de esta acción se muestra a continuación:



Calculando distancias V



► Si todo va bien, la salida en consola es la siguiente:

```

1 | [agent1] executing internal action 'ia.distancia'
2 | [agent1] La distancia euclidibeto entre (10,10) y (20,50) es 50
  
```



Idea

- ▶ Los módulos de Jason permiten implementar a los agentes a partir de **unidades de código** separables, independientes, reusables y más fáciles de mantener.
- ▶ El concepto de **espacio de nombres** es usado para organizar los componentes de los módulos, p. ej., creencias y eventos, y prevenir la colisión de nombres; proveyendo facilidades de interfaz y ocultamiento de información.



Módulo

- ▶ Un **módulo** [9] es un conjunto de creencias, metas y planes, tal y como se define un programa de agente *ag*.
- ▶ Todo agente tiene un **módulo inicial** (su programa inicial) en el cual pueden cargarse otros módulos.
- ▶ Las creencias, metas y planes se conocen con los **componentes** del módulo.

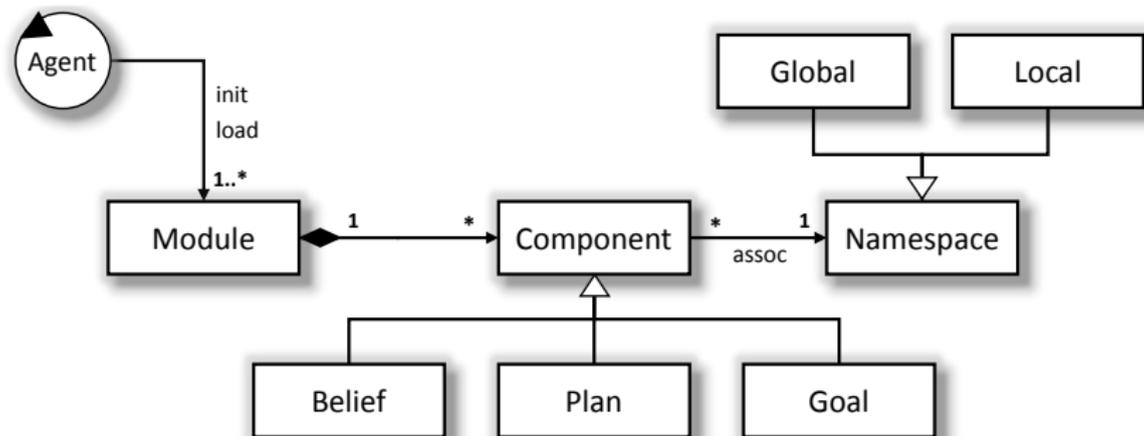


Espacios de nombres

- ▶ Un **espacio de nombres** es un contenedor lógico abstracto que agrupa componentes.
- ▶ **Ejemplo:** `ns1::color(caja,azul)` denota que la creencia `color(caja,azul)` está asociada con el espacio de nombres `ns1`. Por tanto, es diferente de `ns2::color(caja,azul`.
- ▶ Los espacios de nombres pueden ser **locenriques** y **globenriques**.
- ▶ Un espacio de nombres **abstracto** es aquel cuyo nombre es determinado en tiempo de ejecución.
- ▶ **Ejemplo:** `color(caja,azul)` está asociada a un espacio de nombres abstracto.

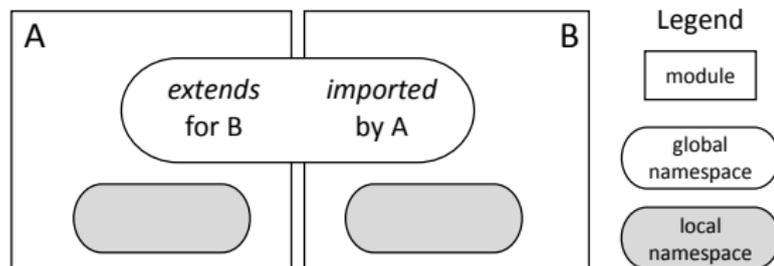


En notación UML



Carga de módulos

- ▶ Cuando un módulo carga a otro se da una interacción en dos direcciones:
 - ▶ El cargador **importa** los componentes del módulo cargado que están asociados con espacios de nombre globales;
 - ▶ El cargador **extiende** los módulos cargados al colocar componentes en esos espacios de nombre.



Identificadores

- ▶ Un **identificador** Jason es un predicado o functor de un término estructura, que aparece en un programa.
- ▶ **Ejemplo:** En el plan:

```
1 | +!ir(casa) : clima(soleado) <- !ir(parque); moverse(casa).
```

los identificadores son: ir, clima, moverse, casa, soleado y parque.



Prefijos

- ▶ Los identificadores pueden adornarse con un prefijo que denota el identificador de un **espacio de nombres**:

$$\langle id \rangle ::= [\langle nid \rangle ::] \langle jid \rangle$$

- ▶ **Ejemplo:** `caja(color, azul)`, `ns1::caja(color, azul)`.
- ▶ De esta forma creencias, metas y planes están siempre bajo el alcance de algún espacio de nombres particular.
- ▶ Los identificadores reservados, las cadenas de texto y los números no están sujetos a prefijos.



Espacio de nombres por default

- ▶ El módulo inicial de un agente es cargado en el espacio de nombres default.
- ▶ Se trata de un espacio de nombres global.
- ▶ Las percepciones del agente están asociadas al módulo default.



Ejemplo 1

- ▶ Consideremos un módulo `factorial.asl` para calcular e imprimir factoriales:

```
1  {begin namespace(fact_ns,local)}
2
3  factorial(0,1).
4
5  factorial(N,F) :-
6    factorial(N-1,F1) &
7    F = F1*N.
8
9  {end}
10
11  // El siguiente plan es exportado
12
13  @p1
14  +!print_factorial(N) <-
15    ?fact_ns::factorial(N,F);
16    .print("El factorial de N es ",F).
```

- ▶ La directiva `begin namespace` indica que las creencias acerca de `factorial/2` están declaradas en un espacio de nombres local, llamado `fact_ns`.



Ejemplo II

- ▶ El plan @p1 está definido en un espacio de nombre abstracto y por lo tanto es global y exportable.
- ▶ El plan @p1 puede hacer uso de factorial/2 por estar en el mismo módulo.
- ▶ Para ello tiene que usar el prefijo fact_ns.
- ▶ El módulo que carga factorial.asl es sample_agent.asl:

```
1 // Agent sample_agent in project modulos
2
3 /* Initial beliefs and rules */
4
5 /* Initial goals */
6
7 !start.
8
9 /* Plans */
10
11 +!start <-
12   .include("factorial.asl",fact);
13   !fact::print_factorial(5);
14   !fact::print_factorial(7).
```



Ejemplo III

- ▶ La acción interna `.include` se encarga de ello.
- ▶ La carga se realiza usando el espacio de nombres `fact`.
- ▶ Otra forma de cargar el módulo es con la siguiente directiva:

```
1 | {include("factorial.asl", local)}
```

- ▶ Si el espacio de nombres se omite, se usará `default`.
- ▶ Si intentamos usar `factorial/2` directamente, p. ej. agregando la meta `?fact::factorial(5,X)`; tendremos un error pues está creencia es local.
- ▶ La salida en consola es la esperada:

```
1 | [agent1] El factorial de N es 120  
2 | [agent1] El factorial de N es 5040
```



Estructura de los mensajes

- ▶ La comunicación en Jason está basada en los **Actos de Habla** de Searle [10], tal y como se definen en KQML [8].
- ▶ Todo mensaje tiene la siguiente estructura:

$$\langle ag_e, iloc, cont \rangle$$

donde ag_e es un átomo $AgentSpeak(L)$ que denota al agente que envía el mensaje, i.e., el **emisor**; $iloc$ es la **fuerza ilocutoria**, i.e., la intención del mensaje, a veces llamada también **performativa**; y $cont$ es el **contenido** del mensaje, que puede tomar varias formas dependiendo de la performativa.

- ▶ Los mensajes se interpretan conforme a la **semántica operacional** vista en el capítulo anterior.



Envío de mensajes

- ▶ Los mensajes se envían usando la siguiente **acción interna**:

$$.send(ag_r, iloc, cont)$$

Donde:

- ▶ ag_r es el agente **receptor**, o una lista de ellos, a quienes el mensaje será enviado.
 - ▶ $iloc$ es la **performativa** del mensaje.
 - ▶ $cont$ es el **contenido** del mismo.
- ▶ Ej.

```
1 | .send(beto,tell,curso(rc))
```

envía un informe (*tell*) a *beto* diciendole que el *curso* es *rc*.



Performativas I

Performativa	Descripción
tell	ag_e intenta que ag_r crea (que ag_e cree) que el contenido del mensaje es verdadero.
untell	ag_e intenta que ag_r no crea (que ag_e cree) que el contenido del mensaje es verdadero.
achieve	ag_e solicita a ag_r que logre un estado donde el contenido del mensaje es verdadero, i.e., una delegación de meta.
unachieve	ag_e solicita a ag_r que abandone la meta de lograr un estado donde el contenido del mensaje es verdadero.
askone	ag_e quiere saber si el contenido del mensaje es verdadero para ag_r , i.e., si existe una respuesta que haga que el contenido sea consecuencia lógica de las creencias de ag_r .
askall	Igual que la anterior, pero ag_e quiere todas las respuestas.



Performativas II

Performativa	Descripción
tellhow	ag_e le comparte a ag_r un plan, i.e, su <i>know-how</i>
untellhow	ag_e le pide a ag_r que olvide el plan comunicado.
askhow	ag_e quiere obtener todos los planes de ag_r que son relevantes para el evento disparador comunicado.



Semántica operacional

- ▶ ¿Qué pasa cuando un agente **recibe** un mensaje?
- ▶ Eso depende del **tipo** de mensaje definido, como lo definimos formalmente en el capítulo anterior.
- ▶ Jason implementa la semántica operacional mediante una **librería de planes** que todos los agentes cargan por default.
- ▶ La librería `kqmlPlans.asl` se encuentra en el directorio `/Applications/jason-2.4/src/main/resources/asl/`
- ▶ Revisaremos los **planes** en ella, ya que son un buen ejemplo de como programar en Jason.



Tell I

► Los planes para recibir un tell incluyen:

```

13  /* ---- tell performatives ---- */
14
15  @kqmlReceivedTellStructure
16  +!kqml_received(Sender, tell, NS::Content, _)
17    : .literal(Content) &
18      .ground(Content) &
19      not .list(Content) &
20      .add_nested_source(Content, Sender, CA)
21    <- ++NS::CA. // add with new focus (as external event)
22
23  @kqmlReceivedTellList
24  +!kqml_received(Sender, tell, Content, _)
25    : .list(Content)
26    <- !add_all_kqml_received(Sender, Content).
27
28  @kqmlReceivedTellList1
29  +!add_all_kqml_received(_, []).
30
31  @kqmlReceivedTellList2
32  +!add_all_kqml_received(Sender, [NS::H|T])
33    : .literal(H) &
34      .ground(H)
35    <- .add_nested_source(H, Sender, CA);
36      ++NS::CA;
37      !add_all_kqml_received(Sender, T).

```



Tell II

```
38 | @kqmlReceivedTellList3
39 | +!add_all_kqml_received(Sender, [_/T])
40 |   <- !add_all_kqml_received(Sender,T).
41 |
42 | @kqmlReceivedUnTell
43 | +!kqml_received(Sender, untell, NS::Content, _)
44 |   <- .add_nested_source(Content, Sender, CA);
45 |   --NS::CA.
```



Comentarios

- ▶ Los planes principales se **activan** (evento disparador) cuando se agrega una meta alcanzable `kqml_received/3`.
- ▶ El uso de los **módulos** puede apreciarse en `NS::Content`
- ▶ Recuerden que los nombres que inician con punto, indican que se trata de una **acción interna**, p. ej., `.literal` regresa verdadero si su argumento es una literal.
- ▶ Las acciones internas predefinidas están documentadas en la distribución de Jason:
`/doc/api/jason/stdlib/package-summary.html`
- ▶ El **primer plan** dice que si el contenido es una literal aterrizada y no es una lista, entonces agregar la creencia anotada a las creencias en el módulo del agente receptor (quien está ejecutando el plan).



Achieve

► Los planes para recibir una achieve incluyen:

```

48  /* ---- achieve performatives ---- */
49
50  @kqmlReceivedAchieve
51  +!kqml_received(Sender, achieve, NS::Content, _)
52    : not .list(Content) & .add_nested_source(Content, Sender, CA)
53    <- !!NS::CA.
54  @kqmlReceivedAchieveList
55  +!kqml_received(Sender, achieve, Content, _)
56    : .list(Content)
57    <- !add_all_kqml_achieve(Sender,Content).
58
59
60  @kqmlReceivedAchieveList1
61  +!add_all_kqml_achieve(_, []).
62
63  @kqmlReceivedAchieveList2
64  +!add_all_kqml_achieve(Sender, [NS::H/T])
65    <- .add_nested_source(H, Sender, CA);
66    !!NS::CA;
67    !add_all_kqml_achieve(Sender,T).
68
69
70  @kqmlReceivedUnAchieve[atomic]
71  +!kqml_received(_, unachieve, NS::Content, _)
72    <- .drop_desire(NS::Content).

```



Ask I

► Los planes para recibir un ask son:

```

87  /* ---- ask performatives ---- */
88
89  @kqmlReceivedAskOneia // (self belief, do not send back the source)
90  +!kqml_received(Sender, askOne, NS::Content, MsgId)
91      : .remove_source_annot(Content,C2) & NS::C2[source(self)]
92      <- .send(Sender, tell, NS::Content, MsgId).
93
94  @kqmlReceivedAskOneib // (belief from other, sends back the source)
95  +!kqml_received(Sender, askOne, NS::Content, MsgId)
96      : NS::Content[source(AGS)]
97      <- .send(Sender, tell, NS::Content[source(AGS)], MsgId).
98
99  @kqmlReceivedAskOneic // (no belief, try to trigger a plan with +?)
100 +!kqml_received(Sender, askOne, NS::Content, MsgId)
101     <- .add_nested_source(Content, Sender, CA);
102        ?NS::CA;
103        // remove source annot from CA
104        .remove_source_annot(CA,CA2);
105        //CA =.. [_ ,F,Ts, _];
106        //CA2 =.. [_ ,F,Ts, []];
107        .send(Sender, tell, NS::CA2, MsgId).
108
109  @kqmlReceivedAskOne2 // error in askOne, send untell
110  -!kqml_received(Sender, askOne, NS::Content, MsgId)
111     <- .send(Sender, untell, NS::Content, MsgId).

```



Ask II

```

112
113 @kqmlReceivedAskAll2
114 +!kqml_received(Sender, askAll, NS::Content, MsgId)
115     <- .findall(NS::Content[source(AGS)], NS::Content[source(AGS)], List);
116     !clear_source(self,List,L2);
117     .send(Sender, tell, L2, MsgId).
118
119 +!clear_source(_, [], []).
120 +!clear_source(S, [B[source(self)]|R], [B           |RC]) <- !clear_source(S,R,RC).

```



Know-How I

- ▶ Los planes para contender con el know-how incluyen:

```
92 |  
93 | /* ---- know-how performatives ---- */  
94 |  
95 | // In tellHow, content must be a string representation  
96 | // of the plan (or a list of such strings)  
97 |  
98 | @kqmlReceivedTellHow  
99 | +!kqml_received(Sender, tellHow, Content, _)   
100 |     <- .add_plan(Content, Sender).  
101 |  
102 | // In untellHow, content must be a plan's  
103 | // label (or a list of labels)  
104 | @kqmlReceivedUnTellHow  
105 | +!kqml_received(Sender, untellHow, Content, _)   
106 |     <- .remove_plan(Content, Sender).  
107 |  
108 | // In askHow, content must be a string representing  
109 | // the triggering event  
110 | @kqmlReceivedAskHow  
111 | +!kqml_received(Sender, askHow, Content, MsgId)   
112 |     <- .relevant_plans(Content, ListOfPlans);
```



Errores

- ▶ Los planes para contender con los errores incluyen:

```
112 |  
113 | /* ---- signal performatives ---- */  
114 |  
115 | @kqmlReceivedSignal  
116 | +!kqml_received(Sender, signal, NS::Content, _)
```



El SMA

- ▶ El SMA incluye dos agentes:

```
1  /*
2  Demo de comunicación
3
4  Un agente (enrique) se comunica con otro (beto) usando actos de
5  habla implementados en KQML y la acción interna .send
6  */
7
8  MAS comunicacion {
9
10     infrastructure: Centralised
11     agents:
12         enrique [beliefs="receptor(beto)"];
13         beto [verbose=1]; // verbose=2 para ver más detalles
14
15     aslSourcePath: "src/asl";
16 }
```

- ▶ Observen el uso de las anotaciones para **inicializar** los agentes.



El agente beto I

► El agente beto es como sigue:

```

1 // Agente beto en el proyecto comunicacion.mas2j
2
3 vl(1).
4 vl(2).
5
6 /* El siguiente plan se dispara cuando se recibe un mensaje
7    tell. El plan agrega una creencia cuya fuente es el agente
8    emisor del tell, enrique en este caso. */
9 +vl(X)[source(Ag)]
10    : Ag \== self
11    <- .print("Recibió un tell ",vl(X)," de ", Ag).
12
13 /* Igual que el caso anterior pero con una performativa achieve
14    en lugar de tell. */
15 +!ir(X,Y)[source(Ag)] : true
16    <- .println("Recibió un achieve ",ir(X,Y)," de ", Ag).
17
18 /* Cuando bob pregunta t2(X), la respuesta no está en mis
19    creencias. Por tanto el evento "+?t2(X)" se crea y es
20    manejado por el siguiente plan. */
21 +?t2(X) : vl(Y) <- X = 10 + Y.
22
23 /* El siguiente plan es usado para reconfigurar la respuesta a
24    un mensaje con performativa askOne. El plan solo es usado
25    si el contenido de askOne es "nombreComp". Se puede usar

```



El agente beto II

```
26 | un evento de tipo +? para esto, se trata solo de un ejemplo  
27 | de sobrecarga de directivas de comunicación KQML. */  
28 | #!kqml_received(Sender, askOne, nombreCompl, ReplyWith) : true  
29 | <- .send(Sender,tell,"Beto Guerra", ReplyWith). // respuesta
```



El agente enrique I

► enrique es más complicado:

```

1 // Agente enrique en el proyecto comunicacion.mas2j
2
3 !inicio.
4
5 +!inicio : receptor(A) // Esta creencia viene del mas2j
6   <- .println("Enviando tell vl(10)");
7     .send(A, tell, vl(10));
8
9     .println("Enviando achieve ir(10,2)");
10    .send(A, achieve, ir(10,2));
11
12    .println("Enviando solicitud síncrona ");
13    .send(A, askOne, vl(X), vl(X));
14    .println("La respuesta a la solicitud es: ", X, " (debe ser 10)");
15
16    .println("Enviando solicitud asíncrona ");
17    .send(A, askOne, vl(_)); // como es asíncrona no tiene 4o argumento
18    // la respuesta se recibe vía un evento +vl(X)
19
20    .println("Preguntando algo que Ana no cree, pero puede responder con ? ");
21    .send(A, askOne, t2(_), Ans2);
22    .println("La respuesta a la solicitud es: ", Ans2, " (debe ser t2(20)");
23
24    .println("Preguntando por algo que ",A," no sabe.");
25    .send(A, askOne, t1(_), Ans1);

```



El agente enrique II

```

26     .println("La respuesta es: ", Ans1, " (debe ser false)");
27
28     .println("Solicitando valores con askall");
29     .send(A, askAll, vl(Y), List1);
30     .println("La respuesta es: ", List1, " (debe ser [vl(10),vl(1),vl(2)])");
31
32     .println("Solicitando un askall de t1(X).");
33     .send(A, askAll, t1(Y), List2);
34     .println("La respuesta es: ", List2, " (debe ser []).");
35
36     .println("Preguntado el nombre completo de Beto.");
37     .send(A, askOne, nombreCompl, FN);
38     .println("El nombre completo de ",A," es ",FN);
39
40     // Preguntare a Ana el plan para ir a algún sitio
41     .send(A, askHow, {+!ir(_)[source(_)]});
42     .wait(500); // esperar la respuesta 500 ms
43     .print("Planes recibidos:");
44     .list_plans( {+!ir(_)[source(_)]} );
45     .print;
46
47     // Otra implementación (no agrega el plan automáticamente a la
48     // librería de planes)
49     .send(A, askHow, {+!ir(_)[source(_)]}, ListOfPlans);
50     .print("Planes recibidos: ", ListOfPlans);
51
52     // Enviándole a beto un plan para !hello
53     .plan_label(Plan,hp); // obtiene un Plan a partir de su etiqueta (hp)

```



El agente enrique III

```
54     .println("Enviando un tellhow de: ",Plan);
55     .send(A,tellHow,Plan);
56
57     .println("Pidiéndole a ",A," satisfacer !hola(ale).");
58     .send(A,achieve, hola(ale));
59     .wait(2000);
60
61     .println("Pidiéndole a ",A," satisfacer -!hola(ale).");
62     .send(A,unachieve, hola(ale));
63
64     // Enviar un untellHow a beto
65     .send(A,untellHow,hp).
66
67
68 +vl(X)[source(A)]
69     <- .print("Valor recibido ",X," de ",A).
70
71 @hp // El plan que será enviado a beto
72 +!hola(Quien)
73     <- .println("Hola ",Quien);
74     .wait(100);
75     !hola(Quien).
```



Salida

```

1 [enrique] Enviando solicitud síncrona
2 [beto] Recibió un tell vl(10) de enrique
3 [enrique] La respuesta a la solicitud es: 10 (debe ser 10)
4 [enrique] Enviando solicitud asíncrona
5 [beto] Recibió un achieve ir(10,2) de enrique
6 [enrique] Preguntando algo que Beto no cree, pero puede responder con +?
7 [enrique] Valor recibido 10 de beto
8 [enrique] La respuesta a la solicitud es: t2(20)[source(beto)] (debe ser t2(20))
9 [enrique] Preguntando por algo que beto no sabe.
10 [enrique] La respuesta es: false (debe ser false)
11 [enrique] Solicitando valores con askall
12 [enrique] La respuesta es: [vl(10)[source(beto)],vl(1)[source(beto)],vl(2)[source(beto)]]
13 (debe ser [vl(10),vl(1),vl(2)])
14 [enrique] Solicitando un askall de t1(X).
15 [enrique] La respuesta es: [] (debe ser []).
16 [enrique] Preguntado el nombre completo de Beto.
17 [enrique] El nombre completo de beto es Beto Guerra
18 [enrique] Planes recibidos:
19 [enrique] @l__4[source(beto)] +!ir(_41X,_42Y)[source(_40Ag)] <-
20 .println("Recibió un achieve ",ir(_41X,_42Y)," de ",_40Ag).
21 [enrique] Planes recibidos: [{ @l__4 +!ir(_44X,_45Y)[source(_43Ag)] <-
22 .println("Recibió un achieve ",ir(_44X,_45Y)," de ",_43Ag) }]
23 [enrique] Enviando un tellhow de: { @hp +!hola(_43Quien) <-
24 .println("Hola ",_43Quien); .wait(100); !hola(_43Quien) }
25 [enrique] Pidiéndole a beto satisfacer !hola(enrique).
26 [beto] Hola enrique
27 ...
28 [beto] Hola enrique
29 [enrique] Pidiéndole a beto satisfacer -!hola(enrique).

```



Comentarios

- ▶ Recuerden que los agentes son autónomos y asíncronos.
- ▶ Lean con cuidado los mensajes de comentario, que incluyen información detallada sobre el intercambio de mensajes que se está llevando a cabo.



Referencias I



RH Bordini y JF Hübner. "BDI agent programming in agentspeak using Jason". En: *Proceedings of the Sixth International Workshop on Computational Logic in Multi-Agent Systems (CLIMA VI), London, UK, 27-29 June, 2005, Revised Selected and Invited Papers*. Ed. por F Toni y P Torroni. Vol. 3900. Lecture Notes in Computer Science. Berlin: Springer-Verlag, 2006, págs. 143-164.



RH Bordini, JF Hübner y DM Tralamazza. "Using Jason to implement a team of gold miners". En: *CLIMA VII. Computational Logic in Multi-Agent Systems*. Ed. por K Inoue, K Satoh y F Toni. Vol. 4371. Lecture Notes in Artificial Intelligence. Berlin Heidelberg: Springer-Verlag, 2007, págs. 304-313.



RH Bordini, JF Hübner y R Vieira. "Multi-Agent Programming: Languages, Platforms and Applications". En: ed. por RH Bordini y col. Springer-Verlag, 2005. Cap. Jason and the Golden Fleece of Agent-Oriented Programming.



RH Bordini, JF Hübner y M Wooldridge. *Programming Multi-Agent Systems in Agent-Speak using Jason*. John Wiley & Sons Ltd, 2007.



RH Bordini y col. "The MAS-SOC Approach to Multi-agent Based Simulation". En: *RASTA 2002*. Ed. por G Lindermann y et al. Vol. 2934. Lecture Notes in Artificial Intelligence. Berlin Heidelberg: Springer-Verlag, 2004, págs. 70-91.



Referencias II



I Bratko. *Prolog programming for Artificial Intelligence*. Addison-Wesley, 2001.



WF Clocksin y CS Melish. *Programming in Prolog, using the ISO standard*. Berlin-Heidelberg, Germany: Springer-Verlag, 2003.



TW Finin y col. "KQML As An Agent Communication Language". En: *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94), Gaithersburg, Maryland, November 29 - December 2, 1994*. New York, NY, USA: ACM, 1994, págs. 456-463.



G Ortiz-Hernández y col. "A Namespace Approach for Modularity in BDI Programming Languages". En: *Engineering Multi-Agent Systems, 4th International Workshop, EMAS 2016. Singapore, Singapore, May 9–10. Revised, Selected, and Invited Papers*. Ed. por M Baldoni y col. Vol. 10093. Lecture Notes in Artificial Intelligence. Berlin Heidelberg: Springer Verlag, 2016, págs. 117-135.



JR Searle. *Speech Acts*. Cambridge University Press, 1969.



L Sterling y E Shapiro. *The Art of Prolog*. Cambridge, MA., USA: MIT Press, 1996.



Referencias III



R Vieira y col. "On the Formal Semantics of Speech-Act Based Communication in an Agent-Oriented Programming Language". En: *Journal of Artificial Intelligence Research* 29 (2007), págs. 221-267.

