

Sistemas Multi-Agentes

Jason

Dr. Alejandro Guerra-Hernández

Instituto de Investigaciones en Inteligencia Artificial
Universidad Veracruzana

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

mailto:aguerra@uv.mx
<https://www.uv.mx/personal/aguerra/sma>

Maestría en Inteligencia Artificial 2024



Jason

- ▶ Jason [2, 3, 1] es un lenguaje de **programación orientado a agentes** (AOP) basado en una versión **extendida** de *AgentSpeak(L)*, implementada en Java:
 - ▶ Comunicación basada en **actos de habla** [12].
 - ▶ Herramientas para **simulación social** [4].
 - ▶ Un sistema de **módulos** [8]
 - ▶ Anotaciones, Negación fuerte, Ambientes, etc.
- ▶ Implementa la **semántica operacional** del lenguaje, con muchas opciones configurables por el usuario.
- ▶ Se trata de un **código abierto**, distribuido bajo una licencia **GNU LGPL**.

Páginas web

- ▶ Jason tiene su **página principal** en:
`https://jason-lang.github.io`
- ▶ El **libro** de Bordini, Hübner y Wooldridge [3] tiene su página en:
`http://jason.sourceforge.net/jBook/jBook/Home.html`
- ▶ La **última versión** para usuarios se puede descargar de:
`https://github.com/jason-lang/jason/releases`
- ▶ Los desarrolladores pueden clonar también su **repositorio**:
`https://github.com/jason-lang/jason`

Requisitos

- ▶ **Java 17** (<https://openjdk.org>)
- ▶ Opcionalmente podríamos necesitar:

Gradle	https://gradle.org
Asciidoctor	https://asciidoctor.org
Visual Studio Code	https://code.visualstudio.com
Neovim	https://neovim.io

Clonando el repositorio Github

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

```
1 > git clone https://github.com/jason-lang/jason.git
2 > cd jason
3 > ./gradlew 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>jar</code> .
doc	Genera <code>javadoc</code> y transforma <code>asciidoc</code> en <code>html</code> .
clean	Borra los archivos generados.
release	Produce un <code>zip</code> en <code>build/distributions</code> .



Directorio principal



bin



build.gradle



demos



doc



examples



gradle



gradle.properties



gradlew



gradlew.bat



jason-cli



jason-interpretor



LICENSE



readme.adoc



readme.html



settings.gradle



Universidad Veracruzana

Observaciones

- ▶ La carpeta `bin` tiene el **ejecutable** de Jason-CLI, un intérprete de línea de comandos para Jason.
- ▶ El **código fuente** está disponible en `jason-interpreter/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.
- ▶ Desde la versión 2.0 de Jason, ya no se usa JEdit, pero pueden usarse otros **editores**, e.g., Visual Studio Code (o Vim).

Proyecto nuevo

- ▶ Para **crear** un nuevo proyecto usamos jason-CLI en la terminal:

```
1 > jason app create app1
2 Creating directory app1
3
4 You can run your application with:
5   > jason app1/app1.mas2j
```

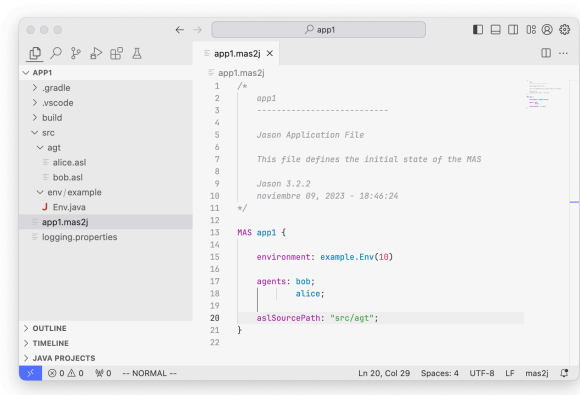
- ▶ Lo cual crea un **directorio** app1 con los archivos del proyecto:



- ▶ Lo más conveniente es instalar un **editor** que reconozca Jason para explorar el proyecto.

Visual Studio Code

- ▶ Después de instalar VSC, instalar la extensión JaCaMo4Code.
- ▶ Abrir la carpeta `app1` en el editor:

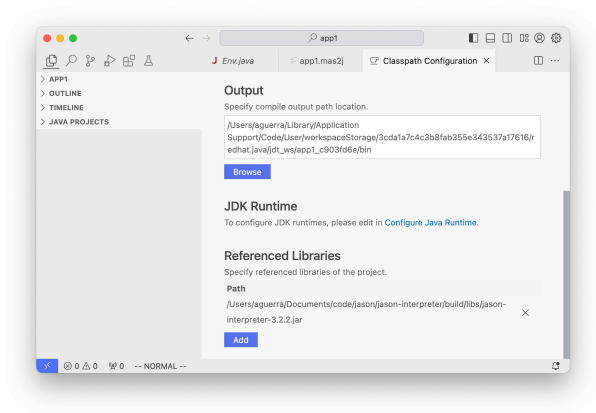


```
1 /*
2  app1
3  -----
4
5  Jason Application File
6
7  This file defines the initial state of the MAS
8
9  Jason 3.2.2
10 noviembre 09, 2023 - 18:46:24
11 */
12
13 MAS app1 {
14
15     environment: example.Env(10)
16
17     agents: bob;
18           | alice;
19
20     aslSourcePath: "src/agt";
21
22 }
```



CLASSPATH de Java

- ▶ Si VSC detecta errores en Java, lo más probable es que haya que actualizar el CLASSPATH de Java:



Archivo de configuración mas2j

```
1  /*
2      app1
3      -----
4
5      Jason Application File
6
7      This file defines the initial state of the MAS
8
9      Jason 3.2.2
10     noviembre 09, 2023 - 18:46:24
11  */
12
13  MAS app1 {
14
15     environment: example.Env(10)
16
17     agents: fran;
18           bob;
19           alice;
20
21     aslSourcePath: "src/agt";
22 }
```

Agente bob

```
1 // Agent bob in project app1
2
3 /* Initial beliefs and rules */
4
5 /* Initial goals */
6
7 !start.
8
9 /* Plans */
10
11 +!start : true <- .print("hello world.").
```



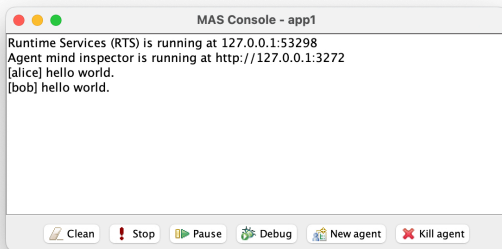
Agente alice

```
1 // Agent alice in project app1
2
3 /* Initial beliefs and rules */
4
5 /* Initial goals */
6
7 !start.
8
9 /* Plans */
10
11 +!start : true <- .print("hello world.").
```

Ejecución del SMA

- ▶ Para ejecutar el SMA hay que abrir una terminal y ejecutar:

```
1 > jason app1.mas2j
```



Agregando otro agente

- ▶ Para agregar a otro agente fran:

```
1 > jason app add-agent fran
2 agent fran (src/agt/fran.asl) was included in ./app1.mas2j
```

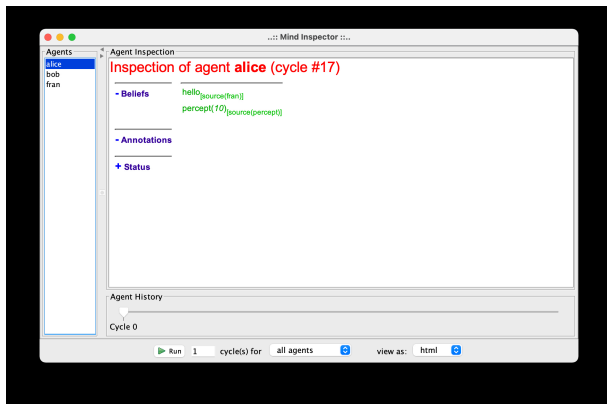
- ▶ Modifiquemos a fran para que le envíe saludos a alice:

```
1 // Agent fran in project
2
3 /* Initial beliefs and rules */
4
5 /* Initial goals */
6
7 !start.
8
9 /* Plans */
10
11 +!start <- .send(alice,tell,hello).
```



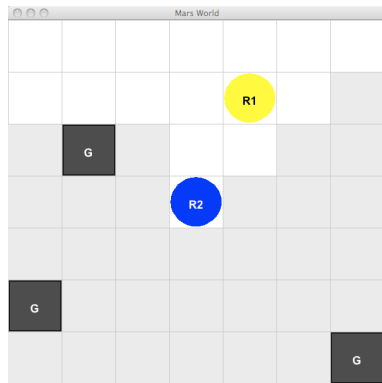
Viendo la mente de alice

- ▶ Exploren la mente de alice usando el mind-inspector (debug):



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.



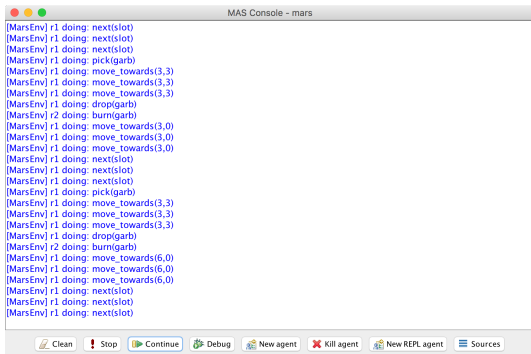
Archivo de configuración

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



Ejecución

- ▶ En terminal, desde el directorio del proyecto: `> jason mars.mas2j`
- ▶ Dos ventanas aparecen: una **consola MAS** y la GUI del proyecto.



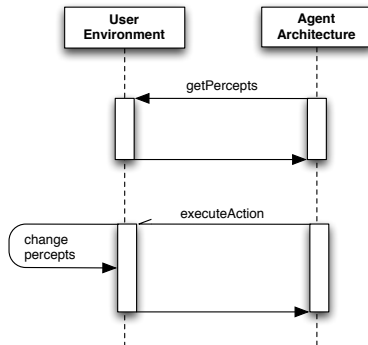
```
MAS Console - mars
[MarsEnv] r1 doing: next(slot)
[MarsEnv] r1 doing: next(slot)
[MarsEnv] r1 doing: next(slot)
[MarsEnv] r1 doing: pick(garb)
[MarsEnv] r1 doing: move_towards(3,3)
[MarsEnv] r1 doing: move_towards(3,3)
[MarsEnv] r1 doing: move_towards(3,3)
[MarsEnv] r1 doing: drop(garb)
[MarsEnv]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

Buttons: Clean, Stop, Continue, Debug, New agent, Kill agent, New REPL agent, Sources



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

```
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

- ▶ Jason-CLI no provee una forma de agregar ambientes a nuestra aplicación.
- ▶ Pero se puede crear/agregar el archivo java al proyecto a mano.
- ▶ Primero debemos declarar el ambiente `PirosAmb` en nuestro SMA:

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

- ▶ Luego definir la clase `PirosAmb` en el directorio `src/java` del proyecto.

La clase PirosAmb I

```
1 // Environment code for project piros.mas2j
2
3 import jason.asSyntax.*;
4 import jason.environment.*;
5 import java.util.logging.*;
6
7 public class PirosAmb extends Environment {
8
9     private Logger logger =
10         Logger.getLogger("piros.mas2j."+PirosAmb.class.getName());
11
12     /** Se ejecuta al iniciar el SMA con la información en .mas2j */
13     @Override
14     public void init(String[] args) {
15         super.init(args);
16     }
17
18     @Override
19     public boolean executeAction(String agName, Structure action) {
20         if (action.getFunctor().equals("incendiar")) {
21             addPercept(Literal.parseLiteral("fuego"));
22         }
23     }
24 }
```



La clase PiroAmb II

```
22     return true;
23 } else {
24     logger.info("executing: "+action+", but not implemented!");
25     return false;
26 }
27 }
28
29     /** Se ejecuta al cerrar el SMA */
30     @Override
31     public void stop() {
32         super.stop();
33     }
34 }
```

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 .



A correr

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

```
7  !start.  
8  
9  /* Plans */  
10  
11  +!start : true <- incendiar.  
12  
13  +fuego <- .print("Fuego! Corran").
```

- ▶ Corrida:

```
1  Jason Http Server running on http://XXX.XXX.X.X:XXXX  
2  [piro] Fuego! Corran
```

Creencias

- ▶ De cierta forma, las creencias de Jason y las metas verificables ($?α$) se comportan de manera muy similar a un sistema de Programación Lógica, p. ej., Prolog [11, 5, 6].
- ▶ Para ilustrar esto vamos a crear un nuevo proyecto llamado creencias, con una infraestructura centralizada y sin un medio ambiente asociado.

La familia I

- Definamos a agent1 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);
```



La familia II

```
20 .print("Laura es progenitor de Rafael");
21 ?progenitor(carmelo,Y);
22 .print("Caramelo 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,rafael)),
5     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 ($-!\alpha$), para **contender con el error**:

```
27 !start[error(Error)] <-  
28 .print("El plan !start falló por el error ", Error).
```

con lo que el error es procesado adecuadamente:

```
1 [agent2] Laura es progenitor de Rafael  
2 [agent2] Carmelo es progenitor de alejandro.  
3 [agent2] laura es un progenitor de Rafael  
4 [agent2] 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).
```



Agregando conocimiento II

```
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 /* 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");
```

Agregando conocimiento III

```
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:

```
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).
```

Reglas recursivas II

```
20
21 ancestro(X,Y) :- progenitor(X,Y).
22 ancestro(X,Y) :- progenitor(X,Z) & progenitor(Z,Y).
23
24 /* Initial goals */
25
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).
```



Reglas recursivas III

► 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
9 /* Initial goals */
10
11 !start.
12
```



Listas II

```
13  /* Plans */
14
15  +!start : true <-
16    Lista = [1,2,3,4,5];
17    ?busqueda(3,Lista);
18    .print("3 es miembro de la lista ",Lista);
19    .findall(X,busqueda(X,Lista),L);
20    .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 I

- ▶ *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 <-
```



Ejemplo II

```
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).
```

► 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);
```

Ejemplos II

```
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," Ooops!");
24 .delete(c,Lista2,L5);
25 .print("Borrar c de la lista 2 no es problema ",L5);
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,Lista1,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:



Ejemplos III

```
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
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     ?del(2,Lista,R);
18     .print("Eliminar 2 de la lista ",Lista," resulta en ",R).
```



Ejemplo III

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(N<sub>1</sub>, N<sub>2</sub>)</i> | <i>math.min(N<sub>1</sub>, N<sub>2</sub>)</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)).
```



# Ejemplo II

► 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).
```

# Ejemplo II

```
20
21 lista2arbol(Xs,A) :- creaArbol(Xs,vacio,A).
22
23 nodos(vacio, []).
24
25 nodos(arbol(X,A1,A2),Xs) :-
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 <-
```

# Ejemplo III

```
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).
```

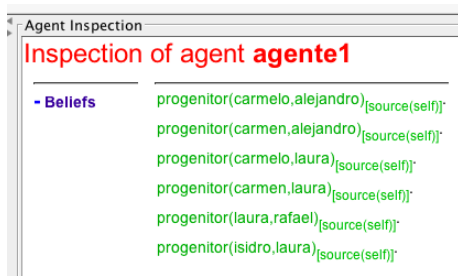
► Su salida en consola es la siguiente:

```
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:
  - `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 $(Cs \cup As) \subset (Ds \cup Bs)$

```

- ▶ 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:

```

1 X = p[Cs]
2 X[As] = Y[Bs] // unifica si $(Cs \cup As) \subset Bs$
3 // e Y unifica con p

```

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

```

1 Y = p[Ds]
2 X[As] = Y[Bs] // unifica si $As \subset (Ds \cup Bs)$
3 // y X unifica con p

```

# Negaciones

- ▶ El **principio del mundo cerrado** (*CWA*) expresa que todo lo que no es consecuencia lógica del programa es **falso**.
- ▶ Prolog adopta una noción **débil** de negación, usando el *CWA* bajo la forma de Negación por Fallo Finito (*NAF*).
- ▶ La meta  $\backslash + \text{ hoy}(\text{martes})$  tiene éxito si  $\text{ hoy}(\text{martes})$  falla finitamente.
- ▶ No hay una representación explícita de que hoy no es martes.
- ▶ Jason provee además una representación **fuerte** de la negación.

# Negación fuerte

- ▶ El **operador** de negación fuerte  $\sim$  denota 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

- ▶ Este ejemplo va de agentes mentirosos, daltónicos y confiables... o de **conflictos** entre creencias.
- ▶ El agente11 cree que la *caja1* es *roja*, pero según *beto* es verde. Para complicar más la historia, según *enrique* la *caja1* no es verde.

```

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

```

- ▶ Para saber el color de la *caja1* según el mismo, el agente11 cree:

```

9 colorSegunYo(Caja,Color) :-
10 color(Caja,Color)[source(Src)] &
11 (Src == self | Src == percept).

```

- ▶ La meta principal del agente11 es reportar de que color es la caja.

# Ejemplo II

- ▶ El agente11 puede describir a los otros agentes en el SMA como sigue:

```
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).
```



## Ejemplo III

- ▶ El primer plan del agente es **aplicable** si hay contradicciones:

```

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).

```



## Ejemplo IV

- ▶ En el segundo plan aplica si no hay contradicciones:

```
45 @sinContradiccion
46 +!start <-
47 .print("No hay contradicciones detectadas");
48 ?colorSegunYo(caja1,Color);
49 .print("La caja1 es de color ",Color,", según yo").
```

- ▶ 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.
- ▶ En caso contrario, hay un tercer agente causando la contradicción y éste es el mentiroso/daltónico.

# Ejemplo V

- ▶ La salida en consola para este caso es:

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

# Ejemplo VI

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

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

# Acciones internas personalizadas

- ▶ El usuario puede **implementar** sus propias acciones internas, al estilo de `math.abs`, etc.
- ▶ Las acciones se organizan 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 de esta interfaz, llamada `DefaultInternalAction`.
- ▶ Las acciones internas se denotan como `librería.acción`.

# Calculando distancias I

- ▶ Vamos a crear el SMA llamado `ia` y modificarlo para trabajar solo con `alice`:

```
13 MAS ia {
14
15 agents: alice;
16
17 aslSourcePath: "src/agt";
18 }
```

- ▶ Como no uso un medio ambiente, puedo borrar la carpeta correspondiente en el proyecto.



# Calculando distancias II

- ▶ Vamos a agregar una acción interna al SMA para calcular la distancia euclidiana entre dos puntos.
- ▶ Primero, creamos la acción desde la terminal:

```
1 $ jason app add-ia 'ia.distancia'
2 internal action ia.distancia (src/java/ia/distancia.java) created.
```

- ▶ Esto creará el archivo `distancia.java` en el directorio `ia` dentro de una carpeta `java` en `src`.
- ▶ Este archivo es una plantilla para implementar nuestra acción interna.

# Calculando distancias III

- ▶ El archivo `distancia.java` lo modificaremos para implementar nuestra acción:

```
1 // Internal action code for project
2
3 package ia;
4
5 import jason.*;
6 import jason.asSemantics.*;
7 import jason.asSyntax.*;
8
9 public class distancia extends jason.asSemantics.DefaultInternalAction {
10
11 @Override
12 public Object execute(TransitionSystem ts, Unifier un, Term[] args)
13 ↪ throws Exception {
14 ts.getAg().getLogger().info("Ejecutando acción interna
15 ↪ 'distancia'");
16 try{
17 NumberTerm x1 = (NumberTerm) args[0];
18 NumberTerm y1 = (NumberTerm) args[1];
```



# Calculando distancias IV

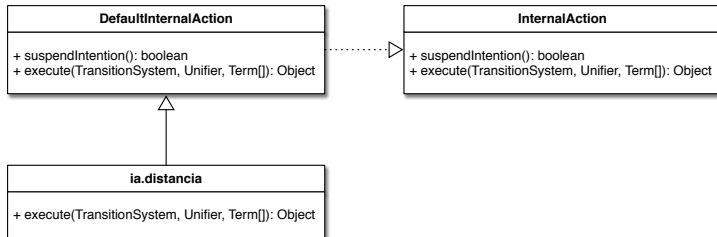
```
17 NumberTerm x2 = (NumberTerm) args[2];
18 NumberTerm y2 = (NumberTerm) args[3];
19
20 double distance = Math.abs(x1.solve()-x2.solve()) +
21 Math.abs(y1.solve()-y2.solve());
22
23 NumberTerm result = new NumberTermImpl(distance);
24 return un.unifies(result,args[4]);
25 } catch (ArrayIndexOutOfBoundsException e) {
26 throw new JasonException("La acción interna 'distancia'"+
27 "no ha recibido cinco argumentos!");
28 } catch (ClassCastException e) {
29 throw new JasonException("La acción interna 'distancia'"+
30 "ha recibido argumentos no numéricos!");
31 } catch (Exception e) {
32 throw new JasonException("Error en 'distancia'");
33 }
34 }
35 }
```





# Calculando distancias V

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



# Calculando distancias VI

- Modificamos a alice para que use su acción:

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



# Calculando distancias VII

► La salida en consola es la siguiente:

- 1 [alice] Ejecutando acción interna 'ia.distancia'
- 2 [alice] La distancia euclidiana 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

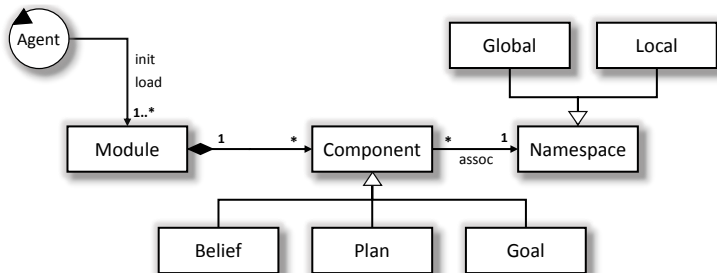
- ▶ Un **módulo** [8, 9] es un conjunto de creencias, metas y planes, tal y como se define un programa de agente *ag*.
- ▶ Todo agente tiene un **módulo inicial** (su programa inicial) en el cual pueden cargarse otros módulos.
- ▶ Las creencias, metas y planes constituyen los **componentes** del módulo.

# 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 **locales** y **globales**.
- ▶ 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.

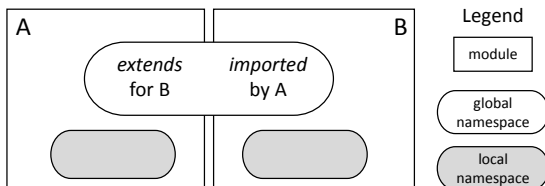


# En notación UML



# Carga de módulos

- ▶ Cuando un módulo carga a otro se da una **interacción bidireccional**:
  - ▶ 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 símbolo de predicado o el functor de un término, que aparece en un programa dado.
- ▶ Ej. 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 un espacio de nombres:

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

- ▶ Ej. `ns1::caja(color, azul)`.
- ▶ De forma que las 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 (su programa) 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 I

- ▶ El módulo `factorial.asl` calcula e imprime 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).
```



## Ejemplo II

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

## Ejemplo III

- ▶ El módulo que carga `factorial.asl` es `agent1.asl`:

```
1 // Agent agent1 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");
13 !print_factorial(5);
14 !print_factorial(7).
```

- ▶ La acción interna `.include` se encarga de ello.
- ▶ La carga se realiza usando el espacio de nombres `default`.



# Ejemplo IV

- ▶ 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_ns::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 [7].
- ▶ 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. Informar (*tell*) a *beto* que el *curso* es *sma*:

1 `.send(beto,tell,curso(sma))`



# 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 <b>delegación</b> 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, tal y como se define en la semántica operacional.
- ▶ Jason implementa la semántica operacional como una **librería de planes** que todos los agentes cargan por default.
- ▶ La librería `kqmlPlans.asl` se encuentra en el directorio `jason/jason-interpretter/src/main/resources/asl/`

# Ejemplo 1

- ▶ Los planes para recibir un tell incluyen:

```
13 @kqmlReceivedTellStructure
14 +!kqml_received(Sender, tell, NS::Content, _)
15 : .literal(Content) &
16 .ground(Content) &
17 not .list(Content) &
18 .add_nested_source(Content, Sender, CA)
19 <- ++NS::CA. // add with new focus (as external event)
20 @kqmlReceivedTellList
21 +!kqml_received(Sender, tell, Content, _)
22 : .list(Content)
23 <- !add_all_kqml_received(Sender, Content).
24
25 @kqmlReceivedTellList1
26 +!add_all_kqml_received(_, []).
```



# Ejemplo II

```
28 @kqmlReceivedTellList2
29 +!add_all_kqml_received(Sender, [NS::H|T])
30 : .literal(H) &
31 .ground(H)
32 <- .add_nested_source(H, Sender, CA);
33 ++NS::CA;
34 !add_all_kqml_received(Sender, T).
35
36 @kqmlReceivedTellList3
37 +!add_all_kqml_received(Sender, [_|T])
38 <- !add_all_kqml_received(Sender, T).
39
40 @kqmlReceivedUnTell
41 +!kqml_received(Sender, untell, NS::Content, _)
42 <- .add_nested_source(Content, Sender, CA);
43 --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:  

```
https://jason-lang.github.io/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).



# El SMA

- ▶ El SMA incluye dos agentes:

```
1
2 MAS comunicacion {
3
4 infrastructure: Centralised
5 agents:
6 enrique [beliefs="receptor(beto)"];
7 beto [verbose=1]; // verbose=2 para ver más detalles
8
9 aslSourcePath: "src/asl";
10 }
```

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





# El agente beto I

- ▶ El agente beto tiene dos **creencias** iniciales:

```
3 vl(1).
4 vl(2).
```

- ▶ Un **plan** para reportar cuando otro agente le informa un nuevo  $vl/1$ :

```
6 +vl(X)[source(Ag)] : Ag \== self
7 <- .print("Recibí un tell ",vl(X)," de ", Ag).
```

- ▶ También incluye planes para resolver dos **metas**:

```
9 +!ir(X,Y)[source(Ag)] : true
10 <- .println("Recibí un achieve ",ir(X,Y)," de ", Ag).
11
12 +?t2(X) : vl(Y) <- X = 10 + Y.
```

- ▶ Y **especializa** un acto de habla!

```
14 +!kqml_received(Sender, askOne, nombreCompl, ReplyWith)
15 <- .send(Sender,tell,"Beto Guerra", ReplyWith).
```

# El agente enrique l

- ▶ enrique envía una serie de mensajes a beto (recuerden que *receptor/2* se inicializa en el *mas2j*:

```

5 +!inicio : receptor(A) <-
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(_));
18
19 .println("Preguntando algo que Beto no cree, pero puede responder con t2");
20 ↪ ");
21 .send(A, askOne, t2(_), Ans2);

```



# El agente enriquece II

```
21 .println("La respuesta a la solicitud es: ", Ans2, " (debe ser t2(20))");
22
23 .println("Preguntando por algo que ",A," no sabe.");
24 .send(A, askOne, t1(_), Ans1);
25 .println("La respuesta es: ", Ans1, " (debe ser false)");
26
27 .println("Solicitando valores con askall");
28 .send(A, askAll, vl(Y), List1);
29 .println("La respuesta es: ", List1, " (debe ser [vl(10),vl(1),vl(2)])");
30
31 .println("Solicitando un askall de t1(X).");
32 .send(A, askAll, t1(Y), List2);
33 .println("La respuesta es: ", List2, " (debe ser []).");
34
35 .println("Preguntado el nombre completo de Beto.");
36 .send(A, askOne, nombreCompl, FN);
37 .println("El nombre completo de ",A," es ",FN);
38
39 .send(A, askHow, {+!ir(_,_) [source(_)]});
40 .wait(500);
41 .print("Planes recibidos:");
```

# El agente enrique III

```
42 .list_plans({+!ir(_,_) [source(_)]});
43 .print;
44
45 .send(A, askHow, {+!ir(_,_) [source(_)]}, ListOfPlans);
46 .print("Planes recibidos: ", ListOfPlans);
47
48 .plan_label(Plan, hp);
49 .println("Enviando un tellhow de: ", Plan);
50 .send(A, tellHow, Plan);
51
52 .println("Pidiéndole a ", A, " satisfacer !hola(ale).");
53 .send(A, achieve, hola(ale));
54 .wait(2000);
55
56 .println("Pidiéndole a ", A, " satisfacer -!hola(ale).");
57 .send(A, unachieve, hola(ale));
58
59 .send(A, untellHow, hp).
```



# El agente enriquece IV

- ▶ También incluye un plan para reportar haber recibido una creencia  $v/1$ :

```
61 +vl(X)[source(A)] <-
62 .print("Valor recibido ",X," de ",A).
```

- ▶ Y un plan para saludar que comunicará a beto:

```
64 @hp
65 +!hola(Quien) <-
66 .println("Hola ",Quien);
67 .wait(100);
68 !hola(Quien).
```

# Observaciones

- ▶ La acción interna `.send` suele usarse con tres argumentos. En ese caso el mensaje es **asíncrono** (enrique no espera la respuesta para continuar).
- ▶ Observen que puede usarse con un cuarto argumento opcional (un patrón de respuesta) y en ese caso el mensaje es **síncrono**.
- ▶ También puede usarse como un quinto argumento, el **tiempo de espera** para la respuesta e milisegundos.
- ▶ Observen que las **metas test** pueden resolverse también con planes.
- ▶ Vean el uso de `.plan_label` para recuperar el plan para saludar antes de enviarlo.
- ▶ Observen el uso del patrón de respuesta para evitar que los planes recibidos por un **askhow** se agreguen automáticamente.



# Salida I

```
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:
```



# Salida II

```
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).
```





# Referencias I

- [1] O Boissier et al. *Multi-Agent Oriented Programming: Programming Multi-Agent Systems using JaCaMo*. Intelligent Robotics and Autonomous Agents. Cambridge, MA, USA: MIT Press, 2020.
- [2] RH Bordini, JF Hübner y R Vieira. "Multi-Agent Programming: Languages, Platforms and Applications". En: ed. por RH Bordini et al. Springer-Verlag, 2005. Cap. Jason and the Golden Fleece of Agent-Oriented Programming.
- [3] RH Bordini, JF Hübner y M Wooldridge. *Programming Multi-Agent Systems in Agent-Speak using Jason*. John Wiley & Sons Ltd, 2007.
- [4] RH Bordini et al. "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, Germany: Springer-Verlag, 2004, págs. 70-91.
- [5] I Bratko. *Prolog programming for Artificial Intelligence*. Addison-Wesley, 2001.
- [6] WF Clocksin y CS Melish. *Programming in Prolog, using the ISO standard*. Berlin-Germany: Springer-Verlag, 2003.
- [7] TW Finin et al. "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.

# Referencias II

- [8] G Ortiz-Hernández et al. "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 et al. Vol. 10093. Lecture Notes in Artificial Intelligence. Berlin, Germany: Springer Verlag, 2016, págs. 117-135.
- [9] G Ortiz-Hernández et al. "Modularization in Belief-Desire-Intention agent programming and artifact-based environments". En: *PeerJ Comput. Sci.* 8 (2022), e1162.
- [10] JR Searle. *Speech Acts: An Essay in the Philosophy of Language*. New York, NY, USA: Cambridge University Press, 1969.
- [11] L Sterling y E Shapiro. *The Art of Prolog*. Cambridge, MA, USA: The MIT Press, 1999.
- [12] R Vieira et al. "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.