Representación del Conocimiento

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/rc

Maestría en Inteligencia Artificial 2024

Jason

- Jason [2, 1] es un lenguaje de programación orientado a agentes (AOP).
- Implementa en Java una versión extendida de *AgentSpeak*(*L*):
  - Comunicación basada en Actos de Habla [10].
  - Implementación de Medios ambientes en Java.
  - Herramientas para simulación social [3].
  - Un sistema de módulos [6, 7].
  - Otros: Anotaciones, negación fuerte, estructuras de control, etc.
- Código abierto bajo una licencia GNU LGPL.

Universidad Veracruzana

Páginas web

- Instalación versión 3.2:
  - Jason puede descargarse desde sourceforge:

    `http://jason.sourceforge.net/wp/`
  - Los desarrolladores pueden clonar también su repositorio github:

    `https://github.com/jason-lang/jason`
- Libros:
  - Programando SMAs en AgentSpeak [2]:

    `http://jason.sourceforge.net/jBook/jBook/Home.html`

Universidad Veracruzana

Requisitos

- Java 17.
- Opcionalmente necesitaremos:

VSC	`https://code.visualstudio.com`
Vim	`https://neovim.io`
Gradle	`https://gradle.org`
Asciidoctor	`https://asciidoctor.org`

Universidad Veracruzana

Directorio principal

Universidad Veracruzana

Path a Java

- En MacOS, se puede configurar `JAVA_HOME` agregando la siguiente línea al archivo de configuración:

```
export JAVA_HOME=`/usr/libexec/java_home -v 17`
```

- La versión de Java debería ser:

```
> java --version
openjdk 17.0.7 2023-04-18 OpenJDK Runtime Environment Homebrew (build
↪  17.0.7+0) OpenJDK 64-Bit Server VM Homebrew (build 17.0.7+0, mixed
↪  mode, sharing)
```

Universidad Veracruzana

Path a jason

- Para agregar la carpeta `bin` al *path* del sistema:

```
export PATH="/Applications/jason/bin:$PATH"
```

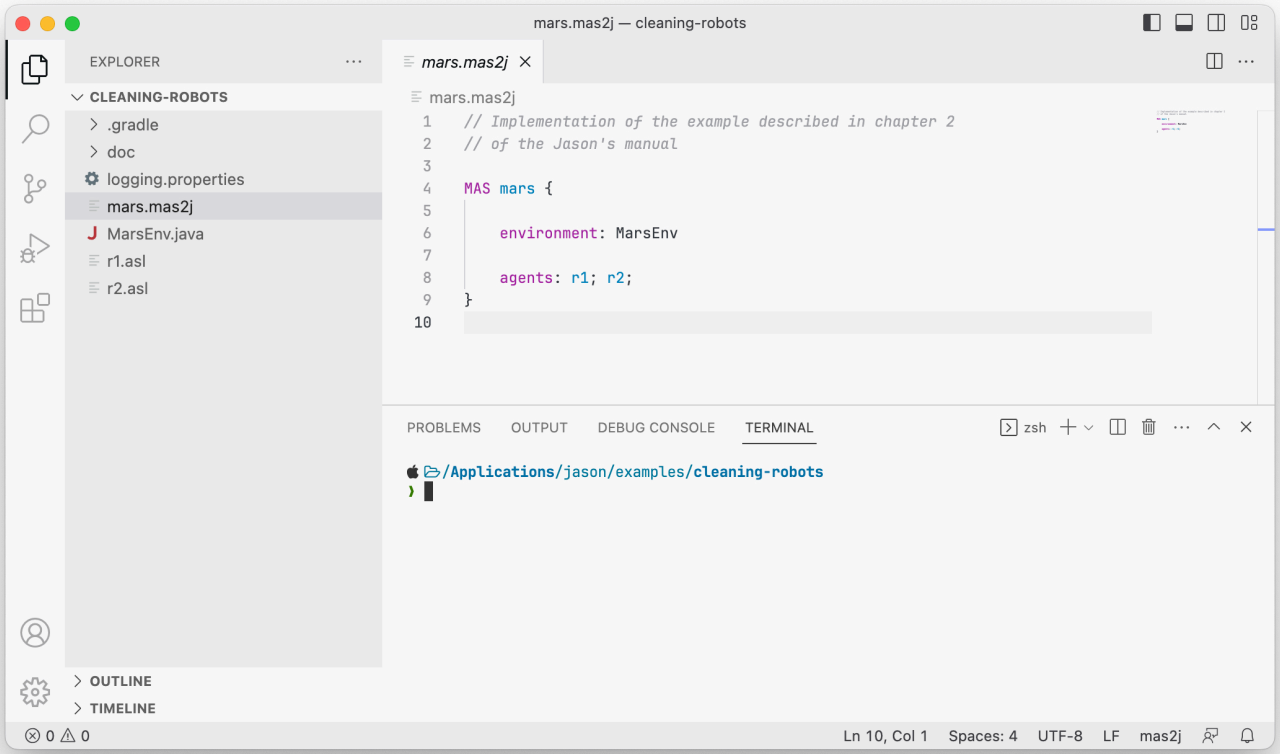
- De forma que sea posible ejecutar el intérprete de comandos en línea de jason desde cualquier directorio:

```
> jason
Jason interactive shell with completion and autosuggestions.
      Hit <TAB> to see available commands.
      Press Ctrl-D to exit.
jason>
```

Universidad Veracruzana

VSC como ambiente de desarrollo

- Se puede usar VSC como editor para Jason.
- Se debe instalar el *plug-in* JaCaMo4Code.



versidad Veracruzana

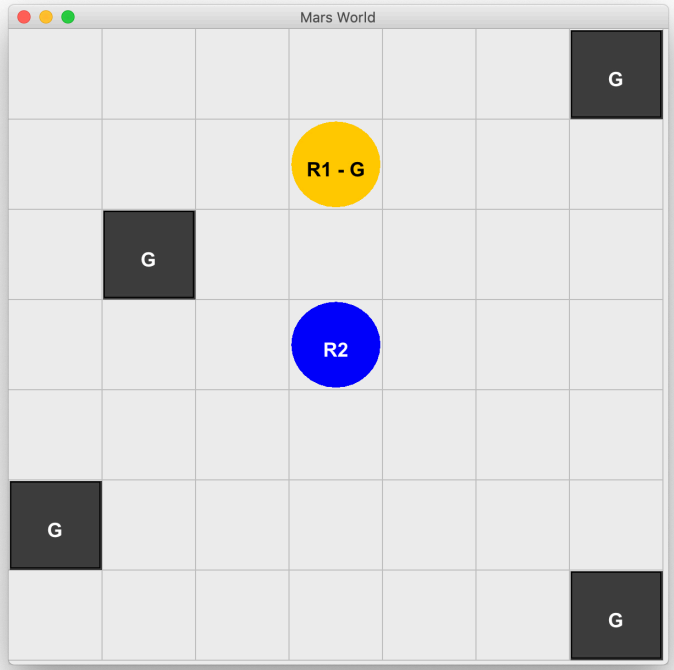
Diferencias

- 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.

Universidad Veracruzana

Descripción del ejemplo

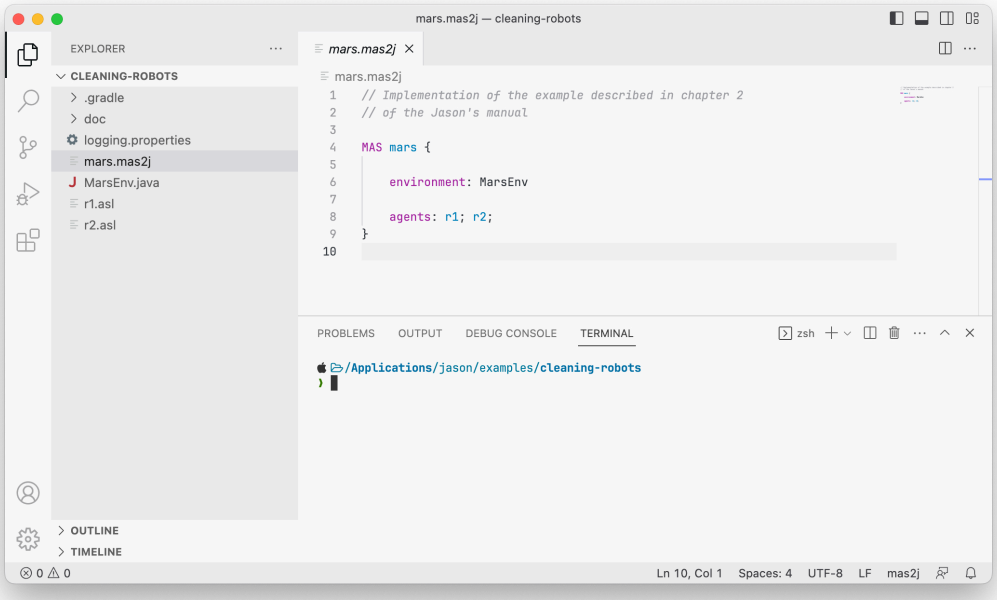
- Vamos a trabajar con el ejemplo `cleaning-robots`.
- El robot `r1` explora el medio ambiente (rejilla 2D) buscando basura.
- Cuando la encuentra se la lleva a `r2` para incinerarla.
- `r1` regresa a la posición donde encontró la última basura y continua su exploración.
- A la derecha el GUI de este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



Universidad Veracruzana

Archivo de configuración

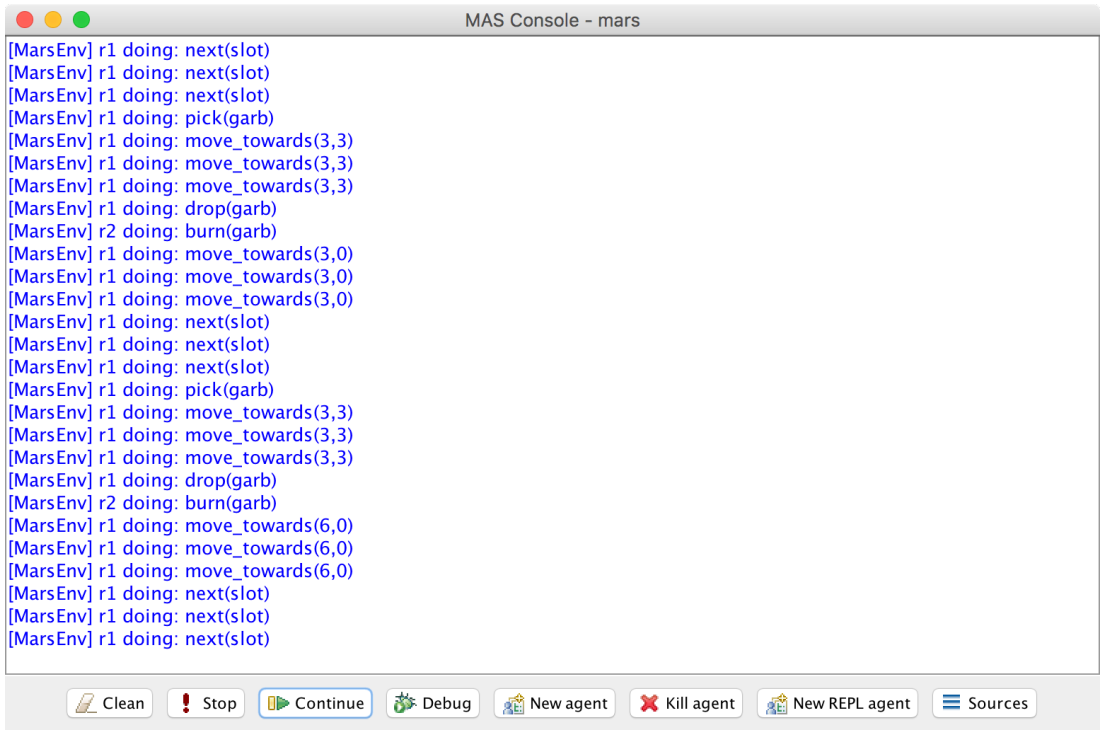
- Todos los proyectos Jason tienen un archivo de configuración con la extensión `mas2j`.
- Abrir la carpeta `examples/cleaning-robots/` en VSC.
- Seleccionar el archivo `mars.mas2j`



Universidad Veracruzana

Ejecución

- En terminal ejecute: `jason mars.mas2j`
- Dos ventanas aparecen: una consola MAS y la GUI del proyecto.



Universidad Veracruzana

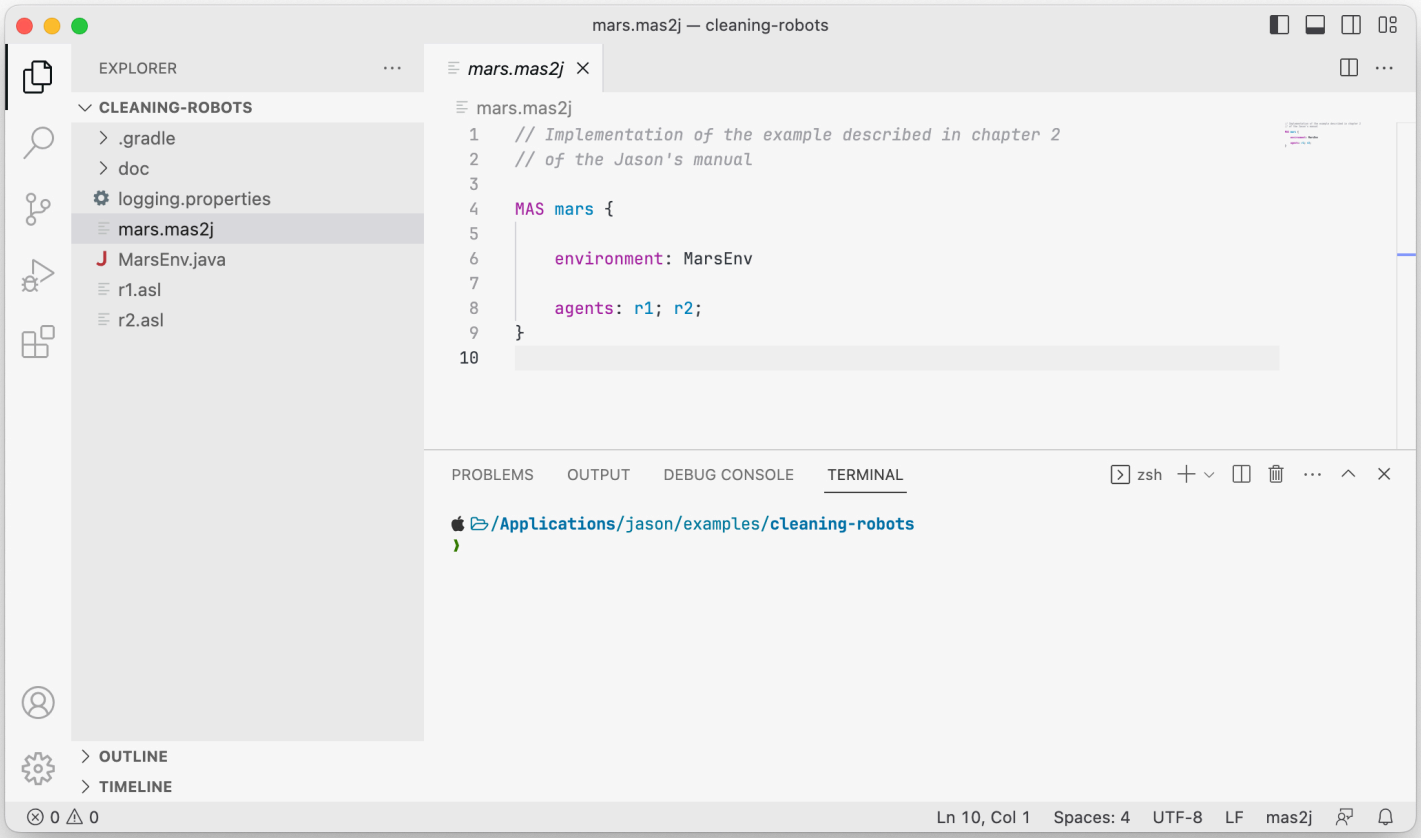
Proyecto nuevo

- Crearemos un proyecto nuevo con el comando:

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

Universidad Veracruzana

El proyecto en VSC



Universidad Veracruzana

Código de los agentes

- Cuando un agente es creado, se le asigna un código por default.
- Ej. El código de `alice` es:

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

- El código de bob es idéntico.

Universidad Veracruzana

Modificando a alice

- Modifiquemos el código de alice para que le envíe un saludo a bob:

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

Universidad Veracruzana

Modificando a bob

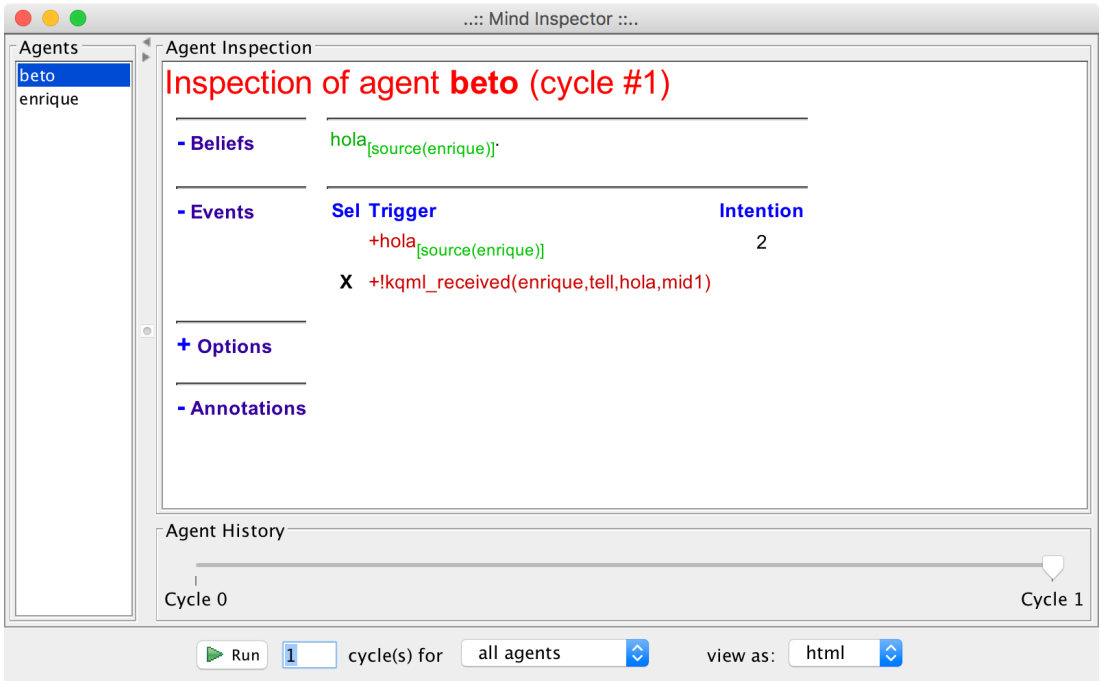
- Si queremos que bob responda al mensaje, agregamos un plan en su código:

```
1  // Agent bob in project saludos
2
3  /* Initial beliefs and rules */
4
5  /* Initial goals */
6
7  !start.
8
9  /* Plans */
10
11 +!start : true <- .print("hello world.").
12
13 +hola[source(X)] <-
14     .print("Recibí un saludo de ", X, ".");
15     .send(X,tell,hola).
```

Universidad Veracruzana

Depuración del proyecto

- Dar clic al botón de debug 



Universidad Veracruzana

Educando a `alice`

- ¿Qué sucede si hacemos que `alice` también responda a los saludos de otros agentes?
- ¿Cómo hacemos eso?
- ¿Cómo es la corrida de estos agentes?

Universidad Veracruzana

Sorpresas te da la vida

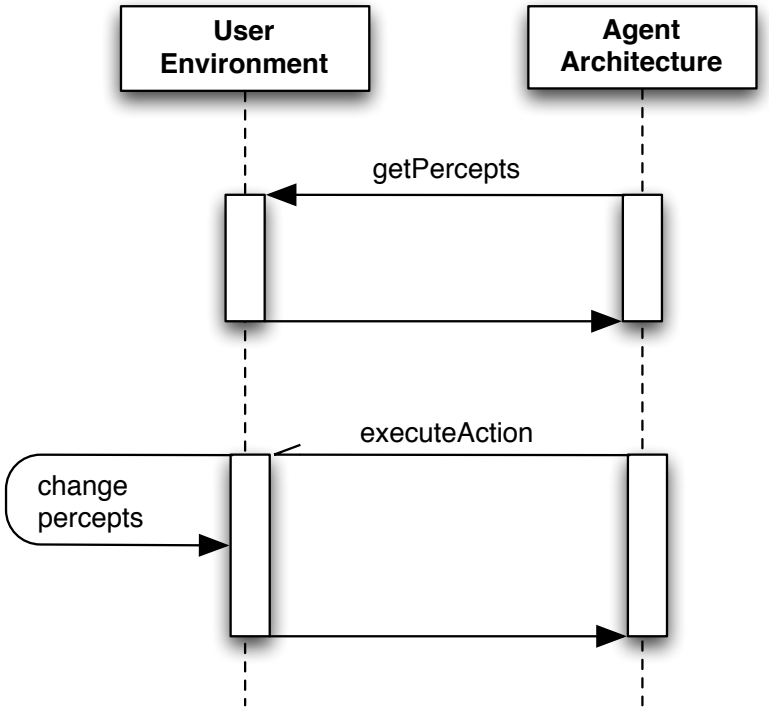
- No hay ciclo en la ejecución de nuestros agentes educados!

- Cuando bob 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.

Universidad Veracruzana

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:



Universidad Veracruzana

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 }
```

Universidad Veracruzana

!Un pirómano 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.

Universidad Veracruzana

Creación del Medio Ambiente

- Especificar en el ambiente de nuestro SMA que el ambiente será `PirosAmb`:

```
1  MAS piros {
2
3    infrastructure: Centralised
4
5    environment: PirosAmb
6
7    agents:
8      piro;
9
10   aslSourcePath:
11     "src/asl";
12 }
```

Universidad Veracruzana

Métodos para implementar el medio ambiente

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

Universidad Veracruzana

La clase PirosAmb completa 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 =
   ↪  Logger.getLogger("piros.mas2j."+PirosAmb.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;
```

Universidad Veracruzana

La clase PirosAmb completa II

```
21        } else {
22          logger.info("executing: "+action+", but not implemented!");
23          return false;
24        }
25      }
26
27      @Override
28      public void stop() {
29        super.stop();
30      }
31    }
```

Universidad Veracruzana

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
```

Universidad Veracruzana

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 [9, 4].
- Para ilustrar esto vamos a crear un nuevo proyecto llamado `creencias`.
- Eliminemos a `bob` y renombremos a `alice` como`agente1.asl`.

Universidad Veracruzana

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);
```

Universidad Veracruzana

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").
```

Universidad Veracruzana

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
```

Universidad Veracruzana

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  rafael)),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?

Universidad Veracruzana

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

Universidad Veracruzana

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  [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
```

Universidad Veracruzana

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).
```

Universidad Veracruzana

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");
```

Universidad Veracruzana

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
```

Universidad Veracruzana

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).
```

Universidad Veracruzana

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).
```

Universidad Veracruzana

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]
```

Universidad Veracruzana

Meta-predicados I

- La acción interna `.findall` se usa igual que en Prolog, para colectar 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 substituímos `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.

Universidad Veracruzana

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 */
```

Universidad Veracruzana

Listas II

```
13
14  +!start : true <-
15    Lista = [1,2,3,4,5];
16    ?busqueda(3,Lista);
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]
```

Universidad Veracruzana

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 <-
```

Universidad Veracruzana

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).
```

Universidad Veracruzana

Ejemplo III

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]
```

Universidad Veracruzana

Acciones internas para listas I

Acción interna	Descripción
$.member(X, Xs)$	X es miembro de Xs.
$.length(X, L)$	La longitud de X es L.
$.empty(X)$	X es una lista vacía.
$.concat(L_1, \ldots, L_n)$	Concatena todas las listas en L_n.
$.delete(X, L, R)$	Elimina X de L resultando la lista R.
$.reverse(L, R)$	La lista R es el reverso de L.
$.shuffle(L, R)$	R es la lista L revuelta.
$.nth(N, L, R)$	R es en N-ésimo elemento de la lista L.
$.max(L, R)$	R es el máximo elemento de la lista L.
$.min(L, R)$	R es el mínimo elemento de la lista L.
$.sort(L, R)$	R es la lista resultante de ordenar L.
$.list(L)$	Verifica si L es una lista.

Universidad Veracruzana

Acciones internas para listas II

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

Universidad Veracruzana

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);
```

Universidad Veracruzana

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:

Universidad Veracruzana

Ejemplos III

```
[agente7] La lista 1 es [1,2,3,4,5]
[agente7] La lista 2 es [a,b,c,d,e]
[agente7] 1 es miembro de la lista 1
[agente7] La longitud de la lista 1 es 5
[agente7] Pegar la lista 1 y 2 nos da [1,2,3,4,5,a,b,c,d,e]
[agente7] Borrar 1 de la lista 1, nos da [1,3,4,5] Ooops!
[agente7] Borrar c de la lista 2 no es problema [a,b,d,e]
[agente7] Revolver la lista 1 produce [3,5,4,1,2]
[agente7] Invertir la lista 2 [e,d,c,b,a]
[agente7] El último elemento de la lista 1 es 5
[agente7] El máximo elemento en la lista 1 es 5
[agente7] El mínimo elemento de la lista 2 es a
[agente7] Ordenar la lista 1 revuelta resulta en [1,2,3,4,5]
```

Universidad Veracruzana

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.

Universidad Veracruzana

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.

Universidad Veracruzana

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).
```

Universidad Veracruzana

Ejemplo III

Su salida en consola es:

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

Universidad Veracruzana

Acciones internas aritméticas

Acciones internas aritméticas			
$math.abs(N)$	$math.acos(N)$	$math.asin(N)$	$math.atan(N)$
$math.average(L)$	$math.cell(N)$	$math.cos(N)$	$.count(B)$
$math.e$	$math.floor(N)$	$.lenght(L)$	$math.log(N)$
$math.max(N_1, N_2)$	$math.min(N_1, N_2)$	$math.pi$	$math.random(N)$
$math.round(N)$	$math.sin(N)$	$math.sqrt(N)$	$math.std\_dev(L)$
$math.sum(L)$	$math.tan(N)$	$system.time$	

Universidad Veracruzana

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:

Universidad Veracruzana

Ejemplo II

```
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
```

Universidad Veracruzana

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) &
```

Universidad Veracruzana

Ejemplo II

```
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) :-
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
```

Universidad Veracruzana

Ejemplo III

```
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).
```

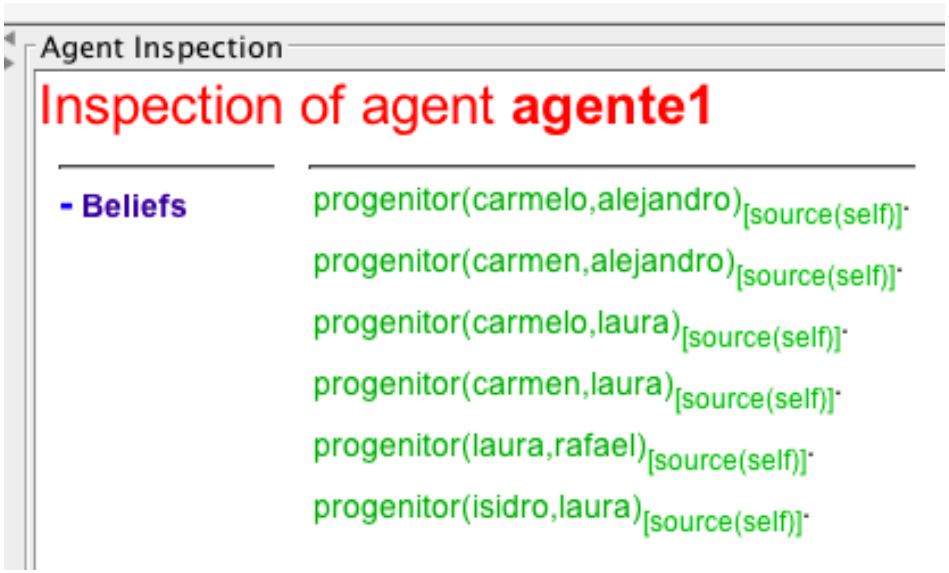
- Su salida en consola es la siguiente:

```
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]
```

Universidad Veracruzana

Anotaciones

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



Universidad Veracruzana

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.

Universidad Veracruzana

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
```

Universidad Veracruzana

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]
```

Universidad Veracruzana

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
```

Universidad Veracruzana

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 ∪ As) ⊂ 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 ⊂ (Ds ∪ Bs)
3                  // y X unifica con p
```

Universidad Veracruzana

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.
- 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.
- Jason provee el operador débil `not`, donde la negación de una fórmula es cierta, si el intérprete falla al derivar dicha fórmula.

Universidad Veracruzana

Negación fuerte

- El operador de negación fuerte $\sim$ 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
l	El agente cree que *l* es verdadera
$\sim$ *l*	El agente cree que *l* es falsa
not l	El agente no cree que *l* es verdadera
not $\sim$ *l*	El agente no cree que *l* es falsa.

Universidad Veracruzana

Ejemplo I

- El agente11 cree que la *caja*1 es *roja*, pero según *beto* la *caja*1 es verde.
- Para complicar más la historia, según *enrique* la *caja*1 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)] &
```

Universidad Veracruzana

Ejemplo II

```
15    daltonico(Ag)[cert(C2)] &
16    C1 > C2.
17  descr(Ag,daltonico) :- daltonico(Ag).
18  descr(Ag,confiable).
19
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)];
```

Universidad Veracruzana

Ejemplo III

```
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");
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.

Universidad Veracruzana

Ejemplo IV

- 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 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.
- La salida en consola para este caso es:

Universidad Veracruzana

Ejemplo V

```
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 *caja*1 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
```

- En cambio si nuestro agente creeyera que la *caja*1 es de color *verde* (línea 7), entonces el daltónico resultaría *enrique*:

Universidad Veracruzana

Ejemplo VI

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

Universidad Veracruzana

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`.

Universidad Veracruzana

Calculando distancias I

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

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

Universidad Veracruzana

Calculando distancias II

- En donde beto haga uso de una acción interna para calcular la distancia euclidiana entre dos puntos:

```
1  // 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).
```

Universidad Veracruzana

Calculando distancias III

- 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:

```
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: Computa la distancia euclidiana entre dos puntos.
12  */
13
```

Universidad Veracruzana

Calculando distancias IV

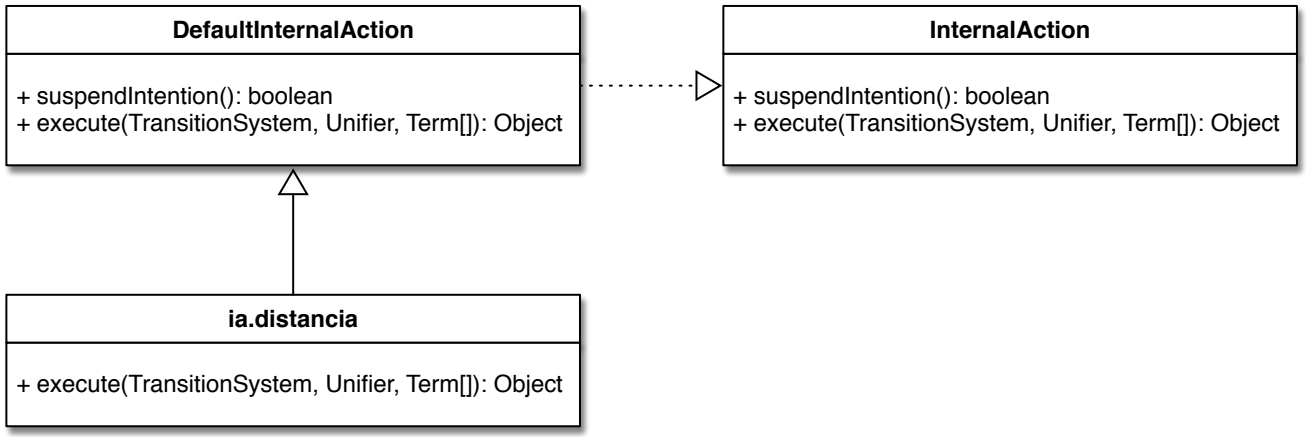
```
14  public class distancia extends DefaultInternalAction {
15
16    private static final long serialVersionUID = 1L;
17
18    @Override
19      public Object execute(TransitionSystem ts, Unifier un, Term[] args)
20      throws Exception {
21          ts.getAg().getLogger().info("Ejecutando la acción interna
            ↪  '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()) +
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) {
```

Universidad Veracruzana

Calculando distancias V

```
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:



Universidad Veracruzana

Calculando distancias VI

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

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

Universidad Veracruzana

Estructura de los mensajes

- La comunicación en Jason está basada en los Actos de Habla de Searle [8], tal y como se definen en KQML [5].
- 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 envia 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 mensaje se intepretan conforme a la semántica operacional vista en el capítulo anterior.

Universidad Veracruzana

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`, diciéndole que el *curso* es *rc*.

Universidad Veracruzana

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 vcerdadero, 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.

Universidad Veracruzana

Performativas II

Performativa	Descripción
tellhow	ag_e le comparte a ag_r un plan, i.e, su *know-how*
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.

Universidad Veracruzana

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 `jason/src/main/resources/asl/`

Universidad Veracruzana

Tell I

- Los planes para recibir un `tell` incluyen:

1

Universidad Veracruzana

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).

Universidad Veracruzana

El SMA

- El SMA incluye dos agentes:

```
/*
Demo de comunicación

Un agente (enrique) se comunica con otro (beto) usando actos de
habla implementados en KQML y la acción interna .send
*/

MAS comunicacion {

    infrastructure: Centralised
    agents:
        enrique [beliefs="receptor(beto)"];
        beto [verbose=1]; // verbose=2 para ver más detalles

  aslSourcePath: "src/asl";
}
```

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

Universidad Veracruzana

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
```

Universidad Veracruzana

El agente beto II

```
20      manejado por el siguiente plan. */
21  +?t2(X) : v1(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
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
```

Universidad Veracruzana

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
```

Universidad Veracruzana

El agente enrique II

```
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);
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);
```

Universidad Veracruzana

El agente enrique III

```
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          // Envíandole a beto un plan para !hello
53          .plan_label(Plan,hp); // obtiene un Plan a partir de su etiqueta (hp)
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));
```

Universidad Veracruzana

El agente enrique IV

```
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).
```

Universidad Veracruzana

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:
```

Universidad Veracruzana

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).
```

Universidad Veracruzana

Comentarios

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

Universidad Veracruzana

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

[3] 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.

[4] I Bratko. *Prolog programming for Artificial Intelligence*. Fourth. Essex, England: Pearson, 2012.

[5] 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.

Universidad Veracruzana

Referencias II

[6] 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.

[7] G Ortiz-Hernández et al. "Modularization in Belief-Desire-Intention agent programming and artifact-based envioronments". En: *PeerJ Comput. Sci.* 8 (2022), e1162.

[8] JR Searle. *Speech Acts*. Cambridge University Press, 1969.

[9] L Sterling y E Shapiro. *The Art of Prolog*. Cambridge, MA, USA: The MIT Press, 1999.

[10] 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.

Universidad Veracruzana