Lab: Agentes (I)

Sistemas Inteligentes Distribuidos

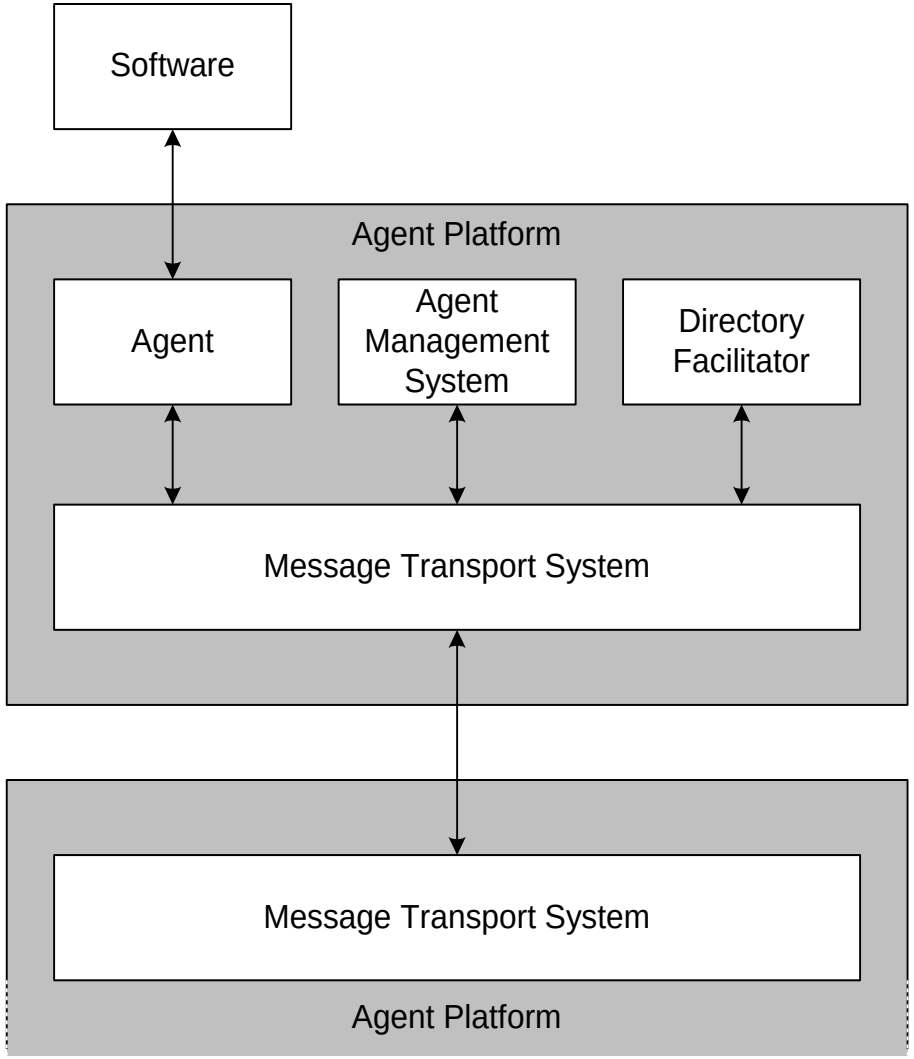
Sergio Alvarez
Javier Vázquez

Objetivos de la sesión

- Entender el concepto de plataforma de agentes
- Conocer la plataforma de agentes SPADE
- Configurar el entorno (Python 3.10.11) para utilizar la plataforma
- Conocer el lenguaje AgentSpeak para desarrollar agentes
- Aprender a hacer razonamiento basado en objetivos con AgentSpeak
- Configurar el entorno pyGOMAS con el que se desarrollará la práctica

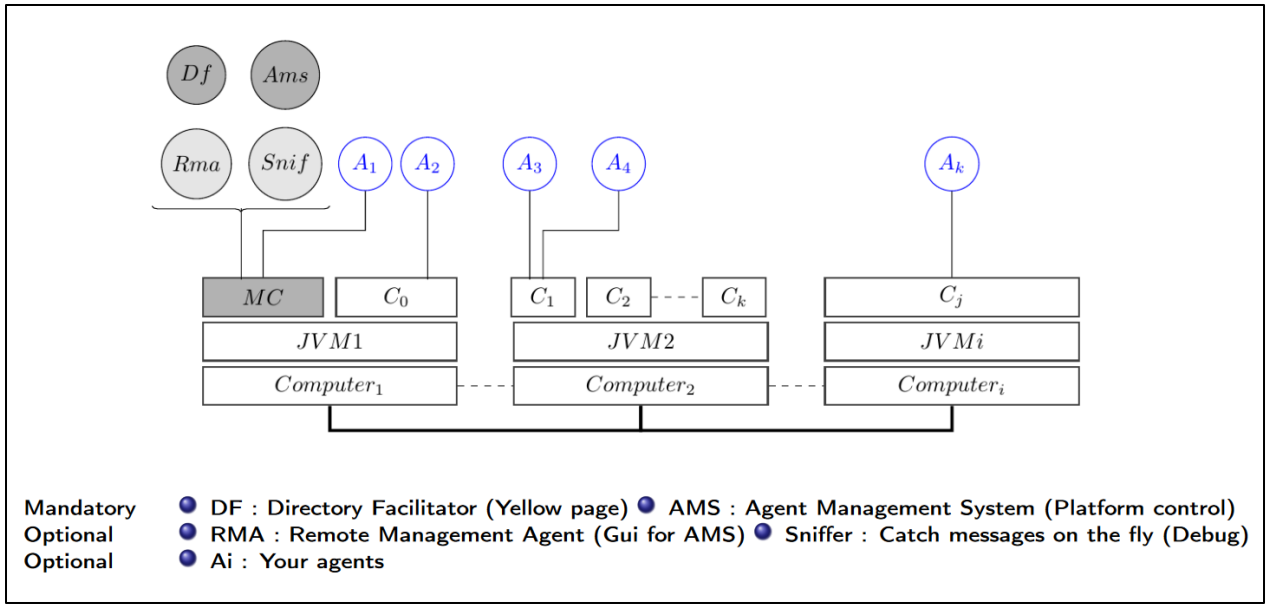
Plataformas de agentes

- Conjunto de herramientas, librerías o *middlewares* que sirven para:
  - Desplegar agentes
  - Conectarlos con un entorno externo
  - Proveer de mecanismos de comunicación
- La *Foundation for Intelligent Physical Agents* (FIPA) define los componentes de una plataforma estándar
- La implementación de referencia es JADE (Java): http://jade.tilab.com



Plataformas de agentes (modelo JADE)

- Los agentes tienen acceso a servicios básicos (también agentes):
  - Sistema de gestión (crear, eliminar, portar agentes)
  - Directorio
  - Monitorización
  - Nivel de transporte (envío y recepción de mensajes)
- Generalmente, cada agente es un *thread* y el comportamiento de cada agente es concurrente (que no paralelo)



SPADE

- SPADE es una plataforma de agentes en Python
  - Sigue las mismas recomendaciones de FIPA
  - El modelo de concurrencia es similar
  - Desarrollado por profesores de la UPV
  - Permite desarrollar agentes dirigidos por objetivos (BDI, teoría sesión 3) con AgentSpeak
- Código fuente: https://github.com/javipalanca/spade
- Documentación: http://spade-mas.readthedocs.io/

- SPADE depende de un gestor de mensajes externo
  - Protocolo XMPP para sistemas de mensajería instantánea: https://xmpp.org
- Usaremos un servidor *ad hoc*
  - Host: *sidfib.mooo.com*, puerto: *9091*
  - En principio no necesita autenticación, pero por favor **usad nombres únicos para cada grupo**
  - Podéis instalar un servidor XMPP local, pero no os daremos soporte:
    - https://igniterealtime.org/projects/openfire/
    - https://prosody.im/download/start

Configuración del entorno (aulas FIB)

- Probado en Windows, pero *debería* funcionar en Linux
- Cread una carpeta para las sesiones de laboratorio y entrad en ella por consola
- Cread un entorno virtual para la asignatura (sólo es necesario una vez):
  - `python3 -m venv venv`
- Activad el entorno virtual (**siempre que abráis consola para SID**)
  - `(Linux)`
    - `$ bash`
    - `$ source venv/bin/activate`
  - `(Windows) venv\Scripts\activate`

Configuración del entorno (NO aulas)

- Si usáis un IDE, adelante, pero no daremos soporte
- Cread una carpeta para las sesiones de laboratorio y entrad en ella por consola
- Mirad qué versión de Python tenéis en el sistema (3.10 o 3.11, no compatible con 3.12):
  - `python --version` o `python3 --version`
- Si no:
  - Instalad *miniconda*: https://docs.anaconda.com/free/miniconda/miniconda-install/
  - Cread y activad un entorno virtual:
    - `conda create –y -n sid python=3.10.11`
    - `conda activate sid` **(este paso lo tendréis que hacer siempre que abráis una consola)**
- Si ya tenéis Python 3.10 o 3.11 instalado en vuestro sistema:
  - Cread y activad un entorno virtual:
    - `pip install virtualenv`
    - `python -m venv venv`
    - `source venv/bin/activate` (Mac/Linux) o `venv\Scripts\activate` (Windows) **(este paso lo tendréis que hacer siempre que abráis una consola)**

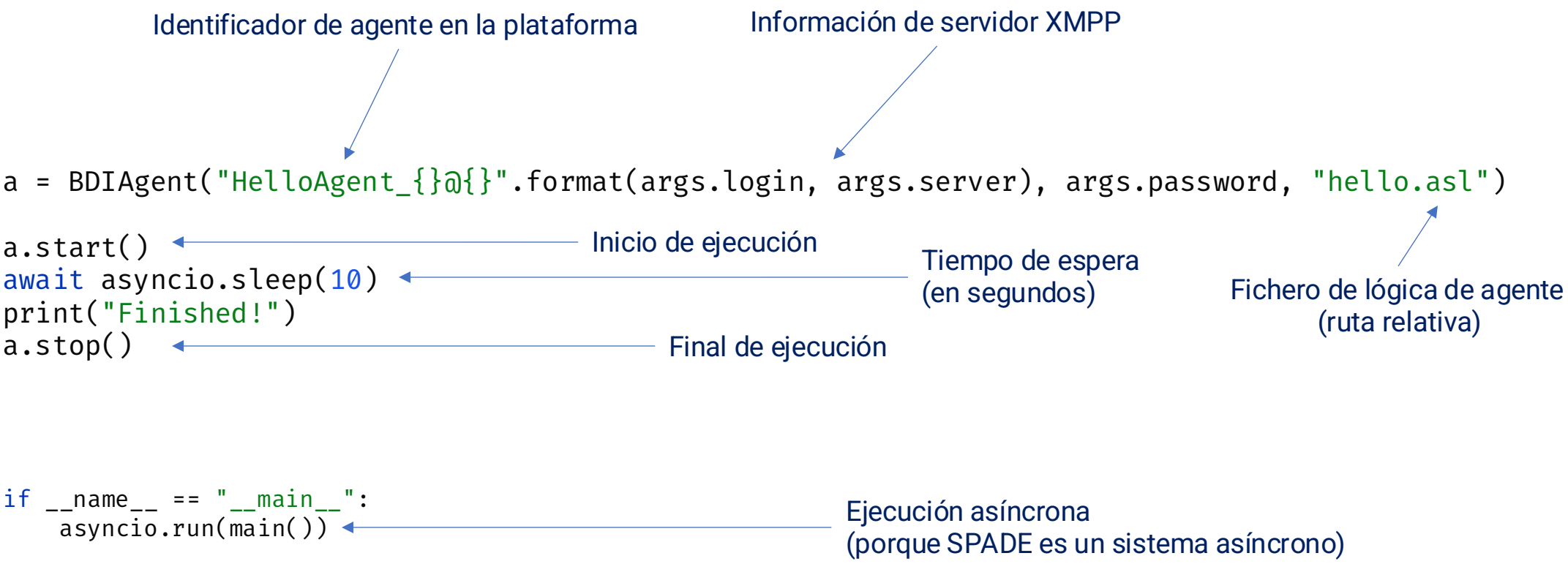
Configuración del entorno

- Instalad `pygomas` en el entorno virtual:
  - **(Windows)** `pip install pygomas==0.5.1 windows-curses`
  - **(Mac/Linux)** `pip install pygomas==0.5.1`
- Comprobad que el binario ejecute bien:
  - `pygomas --help`

Primera prueba de ejecución de agente

- Descargad el fichero `sesion1_1.zip` de la web de la asignatura y descomprimidlo en vuestra carpeta:
  - https://sites.google.com/upc.edu/grau-sid
- Ejecutad el siguiente comando:
  - `python hello.py --login <nombre_equipo> --server sidfib.mooo.com`
    - `nombre_equipo` puede ser lo que queráis, pero ha de ser único entre alumnos/grupos
    - En vez de `sidfib.mooo.com` puede ser `localhost`, si tenéis un servidor local de XMPP
  - Deberíais ver cómo la plataforma registra el mensaje que se ha enviado
  - Analizad `hello.py` y `hello.asl`, ¿qué interpretáis que hace el código?

Código de creación de un agente



Primera prueba de ejecución de agente

- Probad a reducir el tiempo del `sleep` a 1 segundo, ¿qué ocurre?
- Probad a comentar la llamada a `sleep`, ¿qué ocurre?

Código de lógica de agente

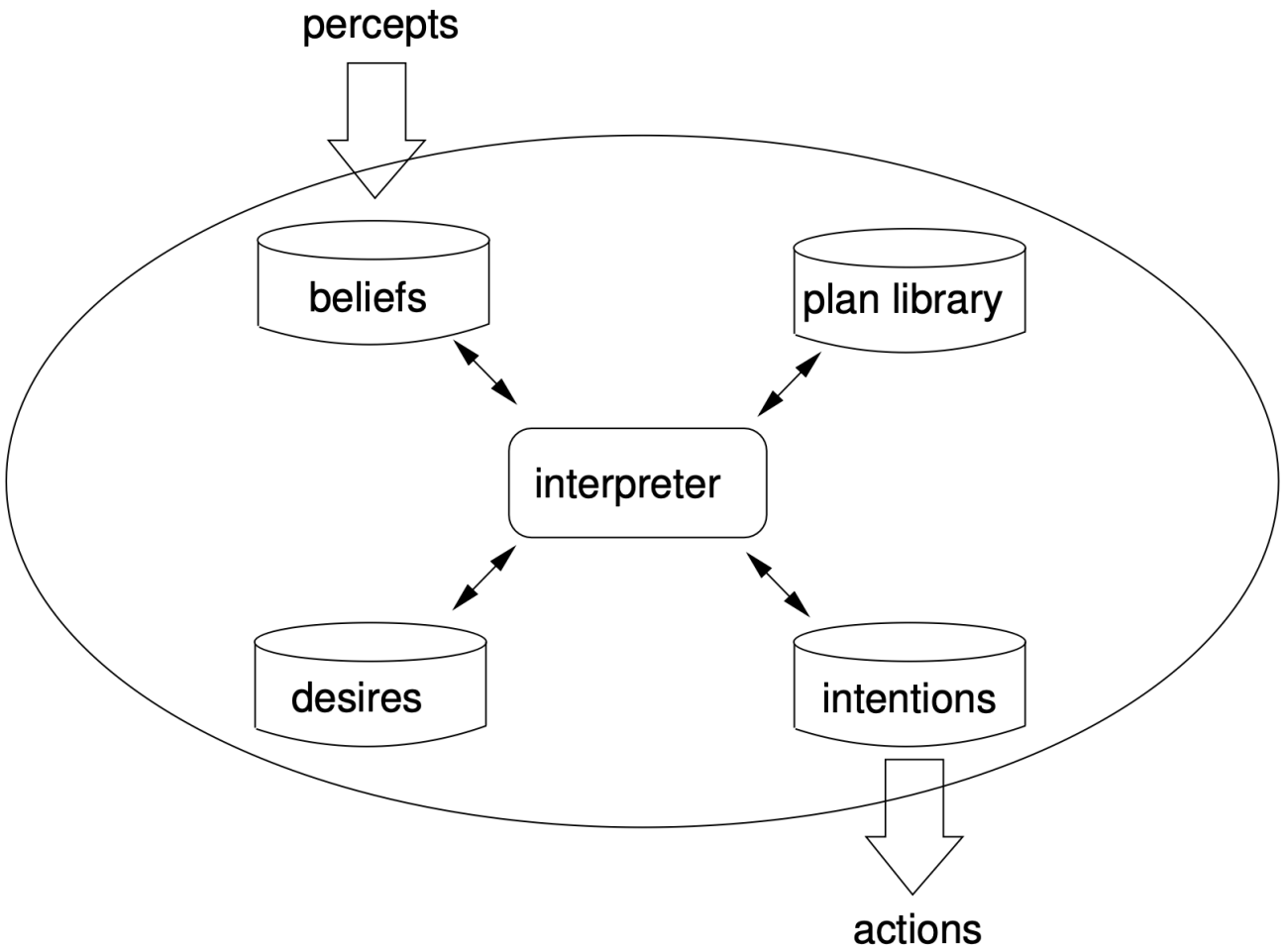
```
!start.

+!start <-
    .print("Hello World!").
```

AgentSpeak

- Lenguaje de programación para agentes lógico-simbólicos
- Creado en 1996 por A. Rao & M. Georgeff
- Inspiraciones
  - Lógica BDI (Belief-Desire-Intention o Creencia-Deseo-Intención)
    - Sesión 3 de teoría
  - Sistemas de razonamiento procedural (PRS)
  - Prolog

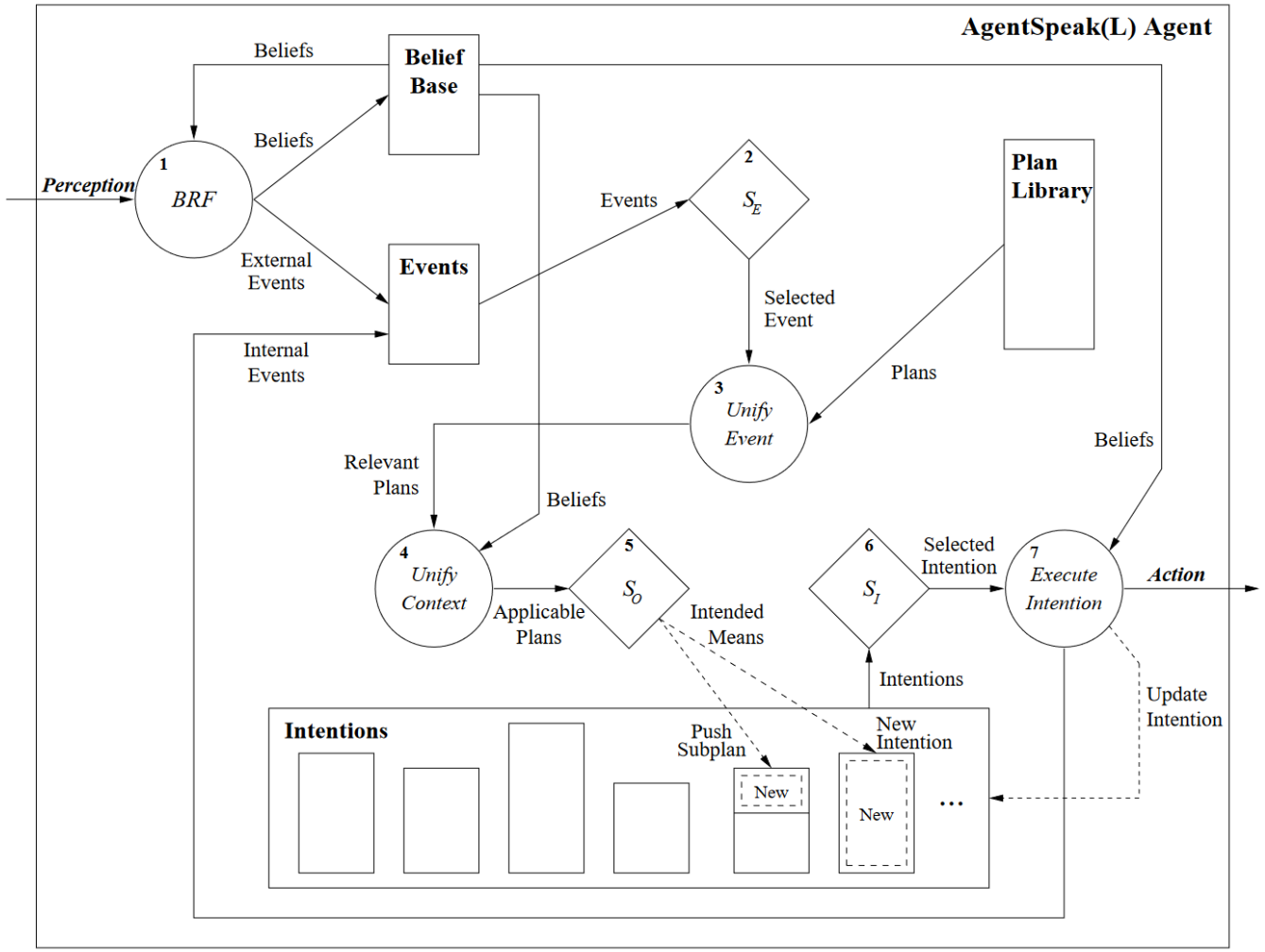
PRS


percepts
beliefs
plan library
interpreter
desires
intentions
actions

AgentSpeak

- Un sistema de ejecución de AgentSpeak tiene los siguientes componentes:
  - Base de creencias
  - Librería de planes
  - Conjunto de eventos
  - Conjunto de intenciones
- El lenguaje permite definir y trabajar con los siguientes elementos de primer orden:
  - Creencias (*beliefs*)
  - Objetivos (*desires*)
  - Planes (*plans*)

Ciclo de razonamiento AgentSpeak



Creencias

- Una creencia en AgentSpeak es un programa de Prolog: un hecho o una regla
- Los hechos son fórmulas que se entienden como ciertas
  - Los literales siempre empiezan por minúscula
  - Ejemplo: `parent(river, parker)`
- Las reglas son fórmulas que enuncian relaciones entre hechos
  - Permiten la inferencia de nuevos hechos (bajo demanda, no es automática)
  - Tenemos operadores `not`, `&` y `|`
  - Las variables libres siempre empiezan por mayúscula
  - Ejemplo: `child(Y, X) :- parent(X, Y)`
    - Permite inferir `child(parker, river)`

Objetivos

Los objetivos se representan como estados que el agente debe conseguir que se cumplan

Objetivos de logro (!): cuando haya un plan satisfactorio para el objetivo, éste desaparece (a menos que se vuelva a declarar)

Ejemplo: `!abierta(puerta)`

"Si esa puerta no está abierta, quiero que lo esté"

Objetivos de conocimiento (?): creencias que el agente quiere saber si se cumplen (son true en la base de creencias) o se infieren (a partir de reglas declaradas)

Ejemplo: `?abierta(puerta)`

"¿Está esa puerta abierta?"

Ejemplo: `?abierta(Puerta)`

"¿Qué puerta está abierta?"

Eventos

- Un agente reacciona a **eventos** ejecutando **planes**
- Los eventos son los **cambios** en
  - Creencias, o
  - Planes
- Eventos posibles:
  - belief addition: `+b`
  - belief deletion: `-b`
  - achievement-goal addition: `+!g`
  - achievement-goal deletion: `-!g`
  - test-goal addition: `+?g`
  - test-goal deletion: `-?g`

Planes

- Los planes tienen la siguiente estructura:

```
evento: contexto <- cuerpo_del_plan
```

  - el evento indica el cambio en creencias o planes que el plan en cuestión puede atender
  - el contexto representa las condiciones que se tienen que dar para que el plan sea aplicable
  - el cuerpo del plan representa la secuencia de acciones a ejecutar si ha habido el evento correspondiente y, en el momento que se dio el evento, el contexto se cumplía
  - si el plan es incondicional, se puede abreviar: `evento <- cuerpo_del_plan`
- El orden de los planes **importa**

Planes

- El cuerpo del plan es una secuencia de acciones separadas por punto y coma: **;** y acabada en un punto: **.**
- En esta secuencia puede haber una combinación de:
  - declaraciones de objetivos (de logro o de creencia)
  - modificaciones de la base de creencias
  - llamadas a funciones de python

```
has_to_write(X) :- message(X). /* Regla */

not_started. /* Hecho */

!start. /* Objetivo de logro */

+!start <- /* Plan incondicional */
    +message("Hello World!"); /* Nuevo hecho */
    -not_started; /* Borrado de hecho */
    !finish. /* Nuevo objetivo de logro */

/* Regla condicional: hecho que no existe
                      y hecho que si existe */
+!finish : not not_started & message(X) <-
    ?has_to_write(X); /* Inferencia */
    .print(X). /* Llamada a python */
```

Segundo agente: API de AgentSpeak

- Probad ahora el agente declarado en `basic.py`
- La invocación desde consola es igual que con `hello.py`:
  - `python basic.py --login <nombre_equipo> --server sidfib.mooo.com`
- En esta ocasión, basic.py utiliza funciones de la librería `python-agentspeak` para acceder y modificar las creencias del agente, desde fuera del código AgentSpeak
- Intentad entender qué está ocurriendo tras cada instrucción del `main()`
  - ¿Por qué no aparece el mensaje de que el coche es azul?
  - ¿Es posible mostrar ese mensaje sin cambiar las creencias que se añaden?

Tercer agente: factorial

- El tercer agente es `asl_launcher.py`, que nos permite aplicar la lógica de cualquier fichero `.asl` al que tengamos acceso
- Probadlo con la implementación en AgentSpeak del factorial:
  - `python .\asl_launcher.py --login <nombre_equipo> --server sidfib.mooo.com --time 10 --asl .\factorial.asl`
- Observad cómo se ha implementado la iteración del factorial
- Modificad la lógica para que la N del factorial dependa de una creencia `limit` (por ejemplo `limit(5)`) en vez de estar *hardcoded*
- **¡Vigilad!** El orden de los eventos importa. Si declaramos `limit(5)` después de `fact(0, 1)` el plan `+fact` no tendrá acceso a `limit(5)`

Cuarto agente: robot limpiador

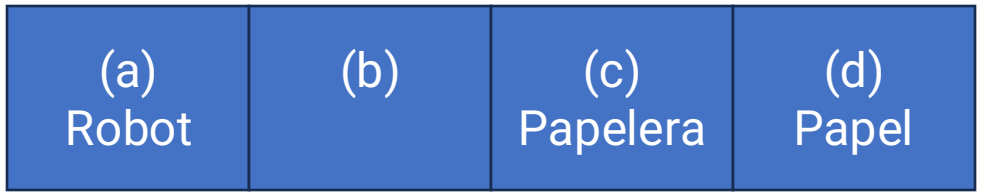
- Utilizad el launcher para ejecutar la lógica de `robot.asl`
- Esta lógica está resolviendo el siguiente escenario:

de manera que el robot acabe tirando el papel en la papelera.

- Observad el código con detenimiento para entender cómo se está resolviendo el problema
  - Recordad: esto no es PDDL y **no planifica**, sino que escoge entre planes ya implementados.

Cuarto agente: robot limpiador

- Modificad la lógica para que:
  - Se pueda ir también de *d* hacia *a*
  - El objetivo sea `!tirado(papel_usado, papelera)` en lugar de `!localizado`
  - El predicado `localizado`, así como sus creencias y eventos asociados, no cambien
  - El escenario inicial sea el siguiente:



pyGOMAS

- Cread una nueva subcarpeta y volcar ahí los contenidos del fichero `sesion1_2.zip`
- Editad `<nombre_equipo>` en `ejemplo.json` para que los nombres de `manager` y `service` contengan vuestro *nombre único*
- Abrid tres terminales, recordando activar el entorno virtual en cada uno, y ejecutad, en cada uno, uno de los siguientes comandos:
  - `pygomas manager -j cmanager-<nombre_equipo>@sidfib.mooo.com -sj cservice-<nombre_equipo>@sidfib.mooo.com -np 6`
  - `pygomas render` (quizá tengáis que esperar a que el manager arranque)
  - `pygomas run -g ejemplo.json`

pyGOMAS

- Juego basado en “Captura La Bandera” usando agentes en SPADE
  - https://github.com/javipalanca/pygomas
- La partida se puede observar a través de visores gráficos (HTTP, pygame, consola)
- 2 equipos (Aliados y Eje) que se enfrentan en un terreno limitado durante un tiempo limitado
- Cada equipo tiene su base, donde se sitúan inicialmente
- Objetivo del juego:
  - Aliados: capturar la bandera y llevarla a su base
  - Eje: impedir la captura (eliminando todos los aliados o si se agota el tiempo)

pyGOMAS

- En la sesión 2 veremos el entorno con más detalle, pero podéis ir probando y mirando el fichero `bdisoldier.asl` para analizar la estrategia base del agente que habéis visto en la interfaz
- Tenéis el manual oficial en la página web de SID, ahí tenéis detalles sobre:
  - Creencias que el agente recibe (percepciones)
  - Acciones sobre el entorno
  - Comunicación entre agentes
- La semana que viene saldrá el enunciado de la práctica 1 de laboratorio
  - Fecha estimada de entrega: 30 de marzo
  - Tres agentes soldado, médico, reconocimiento con estrategias propias basadas en un diseño de objetivos y planes razonados

Ejecutar AgentSpeak sin SPADE

- Los ficheros .asl se pueden ejecutar sin SPADE
  - `python -m agentspeak <fichero>.asl`
- Este método no carga el agente en la plataforma, por lo que:
  - No tenéis acceso a las acciones ni percepciones del entorno pyGOMAS
  - No existe el concepto de agente por lo que no hay comunicación ni acceso a los beliefs del agente desde Python