

AgentSpeak Programming using Jason

Intelligent Agents and Multiagent Systems

Natalia Koliou



Academic Year: 2024-2025

Table of Contents

1 Preliminaries

- GitHub Repository
- Intelligent Agents
- Logic Programs
- Prolog Examples

2 Introduction to Jason

- Terms
- BDI Architecture
- Reasoning Cycle
- Plans
- Messages
- Communication Semantics
- Operators
- Negation

3 Understanding Plans

- Body
- Context
- Triggering Event

4 Jason in Action

- File Structure
- Code Examples
- Tips and Resources
- AirNet

Table of Contents

1 Preliminaries

- GitHub Repository
- Intelligent Agents
- Logic Programs
- Prolog Examples

2 Introduction to Jason

- Terms
- BDI Architecture
- Reasoning Cycle
- Plans
- Messages
- Communication Semantics
- Operators
- Negation

3 Understanding Plans

- Body
- Context
- Triggering Event

4 Jason in Action

- File Structure
- Code Examples
- Tips and Resources
- AirNet

[Here](#) you will find the course's GitHub repository.

- **README:** Follow the installation guide to set up Jason.
- **FAQs:** Find answers to common questions and solutions for trouble-shooting.
- **Examples:** Explore Jason applications like AirNet and MarsBots. Each example includes code files in AgentSpeak and Java.
- **Challenges:** Practice your skills with fun challenges. If you get stuck, you can refer to the provided solutions.
- **Slides:** Review the lecture slides to learn more about AgentSpeak programming and Jason concepts.

- Intelligent agents are autonomous entities that act on a certain environment. More specifically, they:
 - ① **observe** the environment.
 - ② have certain **goals** to achieve.
 - ③ **reason** over the knowledge they're provided.
 - ④ **act** on the environment based on the results from their reasoning process.

- We can represent such entities through **logic programs**.

- In general, a logic program consists of **rules** and **facts**.
- A logic program has the form:

$$\forall X_1, X_2, \dots, X_n (P_1 \wedge P_2 \wedge \dots \wedge P_m \rightarrow Q)$$

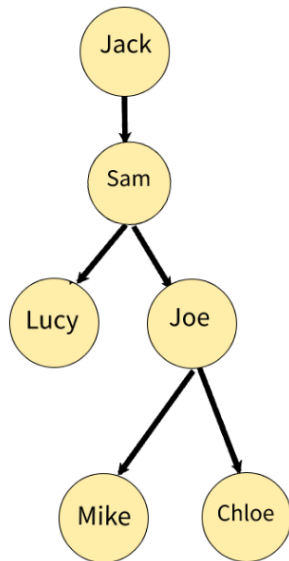
- Where P_1, P_2, P_m and Q are first-order predicates.
- The **rule** consists of a head, denoted as Q , and a body, represented by P_1, P_2, \dots, P_m :

$$Q \leftarrow P_1, P_2, \dots, P_m$$

- If $m = 0$ then Q is a **fact**.

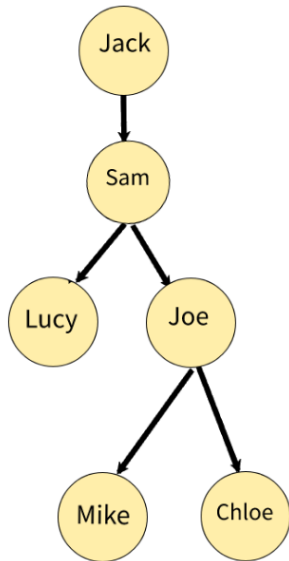
Prolog Examples | Family Tree

- 1 `parentof(sam, lucy).`
- 2 `parentof(sam, joe).`
- 3 `parentof(jack, sam).`
- 4 `parentof(joe, chloe).`
- 5 `parentof(joe, mike).`



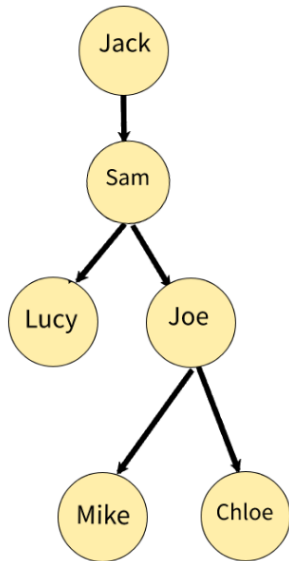
Prolog Examples | Family Tree

- 1 `parentof(sam, lacy).`
- 2 `parentof(sam, joe).`
- 3 `parentof(jack, sam).`
- 4 `parentof(joe, chloe).`
- 5 `parentof(joe, mike).`
- 6 `grandparent(X,Y) :-
parentof(X,Z), parentof(Z,Y).`



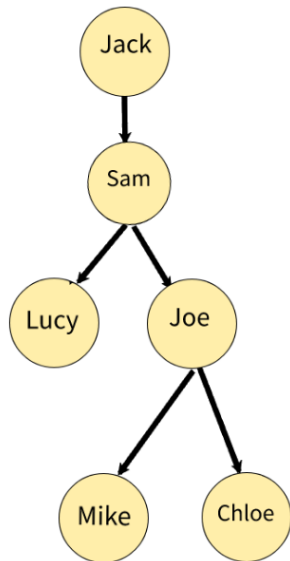
Prolog Examples | Family Tree

- 1 `parentof(sam, lacy).`
- 2 `parentof(sam, joe).`
- 3 `parentof(jack, sam).`
- 4 `parentof(joe, chloe).`
- 5 `parentof(joe, mike).`
- 6 `male(joe).`
- 7 `female(lacy).`
- 8 `male(mike).`
- 9 `female(chloe).`



Prolog Examples | Family Tree

- 1 `parentof(sam, lucy).`
- 2 `parentof(sam, joe).`
- 3 `parentof(jack, sam).`
- 4 `parentof(joe, chloe).`
- 5 `parentof(joe, mike).`
- 6 `male(joe).`
- 7 `female(lucy).`
- 8 `male(mike).`
- 9 `female(chloe).`
- 10 `sister(X,Y) :- parent(Z,X),
parent(Z,Y), female(X).`



- ① `person(sam).`
- ② `beverage(beer).`
- ③ `stock(beer,10).`
- ④ `drink(X,Y) :- person(X), beverage(Y),
stock(Y,N), N>0.`



- ① `person(sam).`
- ② `beverage(beer).`
- ③ `stock(beer,10).`
- ④ `count(sam,7).`
- ⑤ `limit(sam,8).`
- ⑥ `drink(X,Y) :- person(X), stock(Y,N),
count(X,Z), limit(X,K), N>0, Z<K.`



Table of Contents

1 Preliminaries

- GitHub Repository
- Intelligent Agents
- Logic Programs
- Prolog Examples

2 Introduction to Jason

- Terms
- BDI Architecture
- Reasoning Cycle
- Plans
- Messages
- Communication Semantics
- Operators
- Negation

3 Understanding Plans

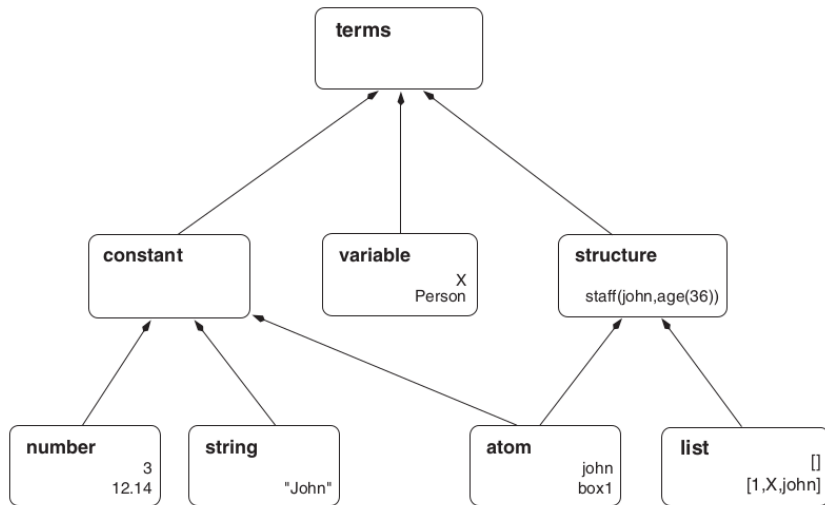
- Body
- Context
- Triggering Event

4 Jason in Action

- File Structure
- Code Examples
- Tips and Resources
- AirNet

- **Constants**: fixed values that do not change.
 - Numbers (e.g., 42)
 - Strings (e.g., "Hello, World!")
 - Atoms (e.g., beer)
- **Variables**: symbols that can take on different values.
 - e.g., Person
 - e.g., X and Y
- **Structures**: represent complex data using a functor and arguments.
 - e.g., staff("John Smith", lecturer)
 - **lists** are special type of structures (e.g., [1,2,3])

Terms



Beliefs

- Agent's belief base (Collection of literals)
- **Literals** are predicates or their negation.
- Represent **facts** about the world the agent perceives.
- Can change over time.
- May have **annotations** indicating their source:
 - perceptual information
 - communication
 - mental notes

Examples

- 1 likes(jack,music)
[source(*self*)].
- 2 not stock(beer,10)
[source(*percept*)].
- 3 limit(sam,8)
[source(*jack*)].

Desires

- Goals the agent would like to achieve.

Goals

- **Achievement Goals**: used to specify a desired state or condition that the agent would like to achieve (denoted by '!').
- **Test Goals**: used to retrieve information from BB (denoted by '?').

Examples

- 1 **!prepare(breakfast)**: achieve a state where prepare(breakfast) is believed to be true.
- 2 **?stock(beer,X)**: find out the stock of beer based on the BB.

Intentions

- Represent what the agent has chosen to do.
- They are associated with **plans** to be executed.
- If an agent intends to achieve a certain goal, they are **committed** to act upon it.

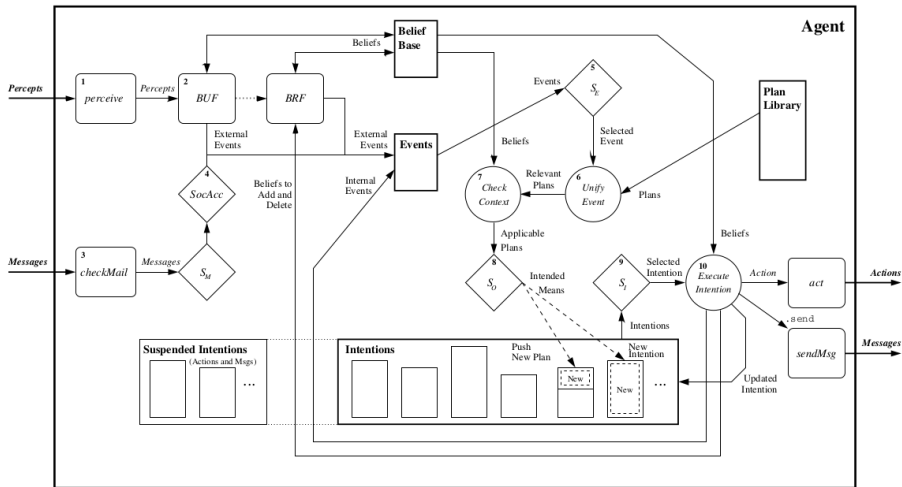
Structure of Plans

triggering event: context \leftarrow body

Example

grow(tree): plant(seed) \leftarrow water(soil)

Reasoning Cycle



Reasoning Cycle

- 1 Perceive the environment.
- 2 Update the belief base.
- 3 Communicate with other agents.
- 4 Select socially acceptable messages.
- 5 Choose an event.
- 6 Retrieve all relevant plans.
- 7 Determine the applicable plans.
- 8 Choose one applicable plan - might need hierarchy.
- 9 Select an intention for further execution.
- 10 Execute one step of an intention.

- Plans are Prolog-like rules with some differences.
- They consist of 3 parts:
 - ① **Triggering event**: the event for which the plan is to be used.
 - ② **Context**: the circumstances in which the plan can be used - it's basically a conjunction of literals and logical expressions.
 - ③ **Body**: the course of action to be used to handle the event.

triggering event: context ← body

Notation	Name
$+l$	Belief addition
$-l$	Belief deletion
$+!l$	Achievement-goal addition
$-!l$	Achievement-goal deletion
$+?l$	Test-goal addition
$-?l$	Test-goal deletion

- Belief addition and deletion upon new environmental percepts.
- Goal addition events from agent communication or plan execution.
- Goal deletion events for plan failure management.

Actions in the **body** of the rule are divided in two categories:

- 1 **External**: actions on the environment.
- 2 **Internal**: actions within the agent's mind that do not affect the environment.

Common Internal Actions

- `.print('The value of ',X)`: prints the concatenation of the string and the value of X.
- `.union(S1,S2,S3)`: computes the union of the sets S1 and S2 and stores the result in S3.
- `.intend(I)`: checks if there is a triggering event +I in any plan within an intention.
- `.drop desire(D)`: removes a desire D from the agent's set of desires.
- `.drop intention(I)`: abandons an intention I.
- `.send(B, ilf, m(X))`: sends a message m(X) to a belief base B using illocutionary force ilf.
- `.broadcast(ilf, m(X))`: broadcasts a message m(X) with illocutionary force ilf to all the other agents.

- Agents interact with each other through **messages**.
- An agent can send various messages to another agent.
- Each message received by **checkMail()** has the form:

<sender, performative, content>

sender: AgentSpeak term identifying the agent.

performative: Goal the sender aims for.

content: AgentSpeak formula defined by the performative.

Illocutionary force (ilf) or **performative** refers to one of the following:

- **tell**: A informs B that the sentence in the message content is true.
- **untell**: The message content is not in the knowledge base of B.
- **achive**: A requests from B to try and achieve a state where $m(X)$ is true.
- **unachieve**: A wants to revert the effect of an achieve previously sent.
- **ask-all**: A wants all of Bs answers to a question.
- The rest can be found [here](#).

- Each **formula** in the context and body of a plan must have a boolean value.
- Plans can include **relational expressions** like e.g. $X \geq Y \cdot 2$.
- Actions in the body are separated by **;**.
- Conjunction is denoted by **&**.
- Disjunction is denoted by **|**.
- **==** and **\==** indicate equality and inequality, respectively.
- **=** is used for unification of terms.

Operators

- `=..` deconstructs a literal into [functor, arguments, annotations], e.g., $p(b, c)[a_1, a_2] = ..[p, [b, c], [a_1, a_2]]$.
- Use `!!` when you want the agent to start a new goal without waiting for the current one to be achieved.
- `-+` removes former instances while adding new ones, helping maintain the latest information in the belief base.
- The anonymous variable `_` unifies with any value.
- Plans can be labeled for identification, and labels can include annotations for meta-level information, e.g. `@label te: context ← body`.

In Jason, there are two types of negation:

- **Weak Negation (not)**: "I don't believe it is true, because I don't have any proof about it."
- **Strong Negation (\sim)**: "I believe it's false, because I know it's not true."

Syntax	Meaning
l	The agent believes l is true
$\sim l$	The agent believes l is false
not l	The agent does not believe l is true
not $\sim l$	The agent does not believe l is false

Jason agents follow the following key principles:

- **Closed World Assumption:** Anything that is neither known to be true, nor derivable from the known facts using the rules in the program, is assumed to be false.
- **Negation as Failure:** If you cannot prove a statement to be true, then you can consider it false.

Example

cwa: $\sim\text{exist}(\text{aliens}) : \text{not observes}(\text{spaceship}).$

naf: $\text{-!meet}(\text{santa_claus}) \leftarrow \sim\text{exist}(\text{santa_claus}).$

Table of Contents

1 Preliminaries

- GitHub Repository
- Intelligent Agents
- Logic Programs
- Prolog Examples

2 Introduction to Jason

- Terms
- BDI Architecture
- Reasoning Cycle
- Plans
- Messages
- Communication Semantics
- Operators
- Negation

3 Understanding Plans

- Body
- Context
- Triggering Event

4 Jason in Action

- File Structure
- Code Examples
- Tips and Resources
- AirNet

- 1 **What are you doing right now?**

① What are you doing right now?

Answer: "I am attending your lab class."

① What are you doing right now?

Answer: "I am attending your lab class."

Goal: !attend(lab).

_____ : _____ ← !attend(lab).

② What are you going to do as soon as you get home?

② What are you going to do as soon as you get home?

Answer: "I am going to study."

② What are you going to do as soon as you get home?

Answer: "I am going to study."

Goal: !attend(lab); !study.

_____ : _____ ← !attend(lab); !study.

③ Under which conditions wouldn't you do that?

③ Under which conditions wouldn't you do that?

Answer: "I would do that, unless I was sick or bored."

③ Under which conditions wouldn't you do that?

Answer: "I would do that, unless I was sick or bored."

Belief: not sick & not bored

_____ : not sick & not bored \leftarrow !attend(lab); !study.

④ What makes you want to do that?

④ What makes you want to do that?

Answer: "I want to pass the class."

④ What makes you want to do that?

Answer: "I want to pass the class."

Goal Addition: +!pass

+!pass : not sick & not bored \leftarrow !attend(lab); !study.

Table of Contents

1 Preliminaries

- GitHub Repository
- Intelligent Agents
- Logic Programs
- Prolog Examples

2 Introduction to Jason

- Terms
- BDI Architecture
- Reasoning Cycle
- Plans
- Messages
- Communication Semantics
- Operators
- Negation

3 Understanding Plans

- Body
- Context
- Triggering Event

4 Jason in Action

- File Structure
- Code Examples
- Tips and Resources
- AirNet

System

- Defines the multi-agent system's structure and configuration.
- Text-based configuration file ending with the `.mas2j` extension.

Agents

- Models agent perception (beliefs), reasoning and behavior.
- AgentSpeak script file ending with the `.asl` extension.

Environment

- Creates the shared environment and simulates agent interactions.
- Java class file ending with the `.java` extension.

Fibonacci Project (no environment)

Agent Fibo is designed to calculate and print the Fibonacci number at a specified position.

It follows the logic of the Fibonacci sequence, which is a series of numbers where each number is the sum of the two preceding ones.

```
MAS fibonacci {  
  
    agents: fibo;  
  
    aslSourcePath: "src/agt";  
}
```

```
!print_fib(10) .
```

```
+!print_fib(N)  
  <- !fibonacci(N, F);  
      .print("Fibonacci number at position ", N,  
            " is ", F) .
```

```
+!fibonacci(N, 0) : N == 0 .
```

```
+!fibonacci(N, 1) : N == 1 .
```

```
+!fibonacci(N, F) : N > 1  
  <- !fibonacci(N-1, F1);  
      !fibonacci(N-2, F2);  
      F = F1 + F2 .
```

Elevator Project

In this system, the "machine" agent manages an elevator within a building. The elevator is used by multiple agents, including "bob" and others, to move between floors and reach their respective destinations.

```
MAS elevator {  
  
    environment: example.Env  
    agents: machine; bob; alice;  
  
    aslSourcePath: "src/agt";  
}
```

Bob wants to go from floor 0 to floor 1.

!served.

+!served : not served

```
<- !at(0);  
  .print("Take me to 1.");  
  .send(machine,achieve,at(1));  
  .print("Thanks, bye!").
```

+!at(0) : at(0).

```
+!at(0) <- .print("Pick me up from 0.");  
          .send(machine,achieve,at(0)).
```



```
package example;
import jason.asSyntax.*;
import jason.environment.*;

public class Env extends Environment {
    ...
    @Override
        public void init(String[] args) {
            addPercept(floor);
        }
    @Override
        public boolean executeAction(String ag,
                                    Structure act) {
            ...
        }
}
```

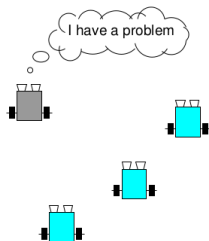
- When studying a built-in example, use the **Jason Interpreter API** as a helping tool.

- Search for the unfamiliar commands you come across.
- Identify their origins (methods or classes they belong to).
- Understand the context in which they can be applied.
- Return to the example and examine their current use.

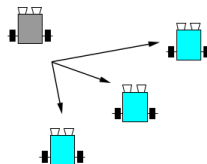
This example is a great starting point for your assignment!

The **AirNet** example demonstrates the **Contract Net Protocol** (CNP) in a multi-agent flight booking scenario.

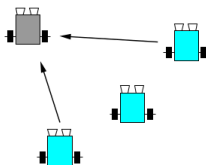
- CNP is a protocol for distributed problem-solving among agents in a multiagent system.
- Agents can request other agents to perform subtasks for them.
- An **initiator** issues a call for proposals (cfp) requesting bids from **participants** for a specific task.
- After some deadline, the initiator evaluates received bids and selects one participant to perform the task.



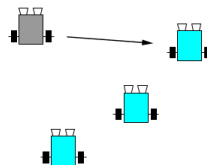
Recognition



Announcement



Bidding



Awarding

In the **AirNet** example, we simulate a flight booking system where:

- **Manager**: acts as the initiator who assigns flight requests and manages bids.
- **Aircrafts**: act as participants that respond to booking requests.
 - ① Aegean offers service and submits bids.
 - ② Lufthansa refuses service as it is under maintenance.
 - ③ Aeroflot refuses service as it is fully booked.
- The system has 1 Lufthansa, 1 Aeroflot and 3 Aegean aircrafts, making 6 agents in total, including the manager.

Walking through the flow of interactions:

- The manager sends a **Call for Proposals** (CFP) to the aircraft agents, requesting bids for a flight.
- Each agent decides whether to bid or refuse based on its status and availability.
- The manager collects all responses, selects the best offer, and announce the final winner.

- [1] BORDINI, R. H., HUBNER, J. F., & WOOLDRIDGE, M. J. (2007). *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons, Ltd. Hardcover.
- [2] BORDINI, R. H. & HUBNER, J. F. (2006). *BDI agent programming in AgentSpeak using Jason (tutorial paper)*. In TONI, F. & TORRONI, P. (Eds.), *Computational Logic in Multi-Agent Systems*, volume 3900 of *Lecture Notes in Computer Science*, pp. 143–164. Springer. 6th International Workshop, CLIMA VI, London, UK, June 27-29, 2005. Revised Selected and Invited Papers.
- [3] RAO, A. S. & GEORGEFF, M. P. (1995). *BDI Agents: From Theory to Practice*. In LESSER, V. R. & GASSER, L. (Eds.), *1st International Conference on Multi-Agent Systems (ICMAS 1995)*, pp. 312–319, San Francisco, CA, USA. The MIT Press.

Any Questions?

Should any further thoughts arise, feel free to contact:

Natalia Koliou: nataliakoliou@iit.demokritos.gr

Thank you!