AgentSpeak Programming using Jason

Intelligent Agents and Multiagent Systems

Academic Year: 2025-2026

Table of Contents

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
- Fibonacci

GitHub Repository

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

- 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.

Logic Programs

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

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

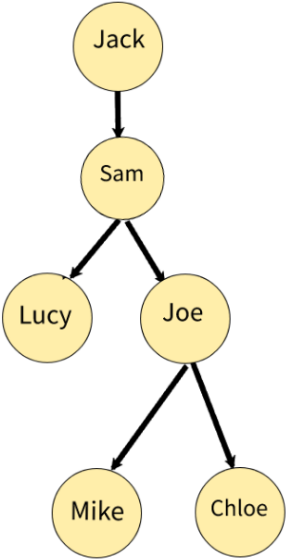
- Where P_1, P_2, P_m and Q are first-order predicates.
- The rule consists of a head, denoted as Q, and a body, represented by $P_1, P_2, ..., P_m$:

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

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

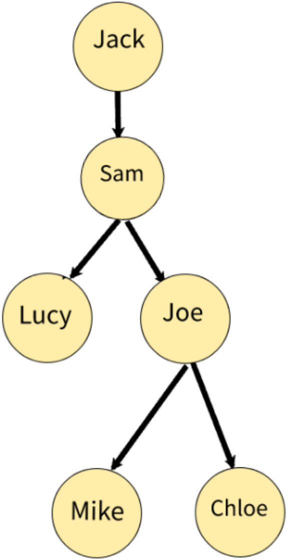
Prolog Examples | Family Tree

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



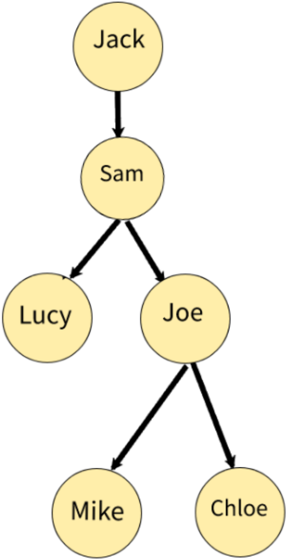
Prolog Examples | Family Tree

1. parentof(sam, lucy).
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).



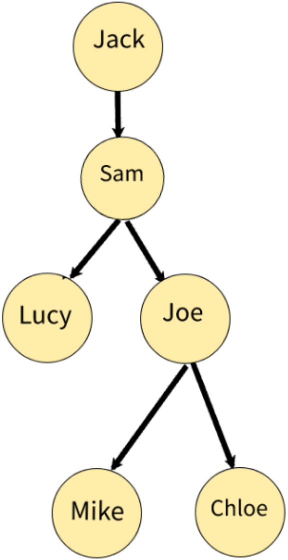
Prolog Examples | Family Tree

1. parentof(sam, 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).



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).



Prolog Examples | Drunk Agent

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

Prolog Examples | Drunk Agent

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

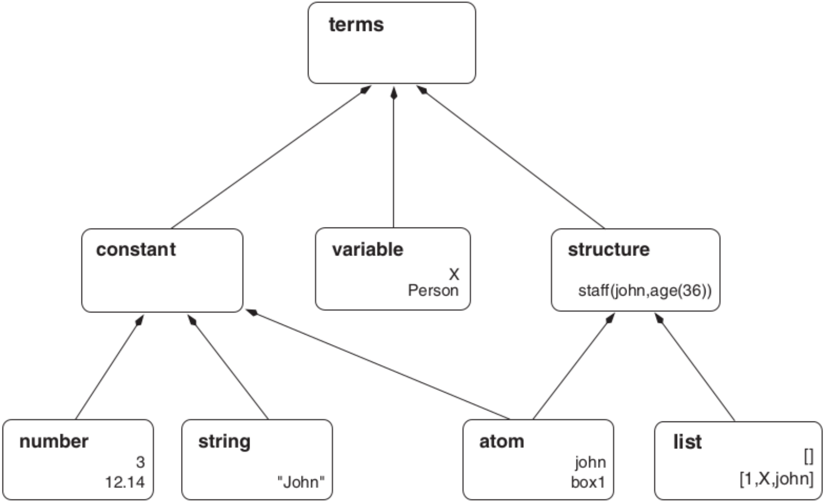
3 **Understanding Plans**
- Body
- Context
- Triggering Event

4 **Jason in Action**
- File Structure
- Code Examples
- Tips and Resources
- AirNet
- Fibonacci

Terms

- **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


terms
constant
variable
X
Person
structure
staff(john,age(36))
number
3
12.14
string
"John"
atom
john
box1
list
[]
[1,X,john]

BDI Architecture

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*)].

BDI Architecture

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.

BDI Architecture

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.

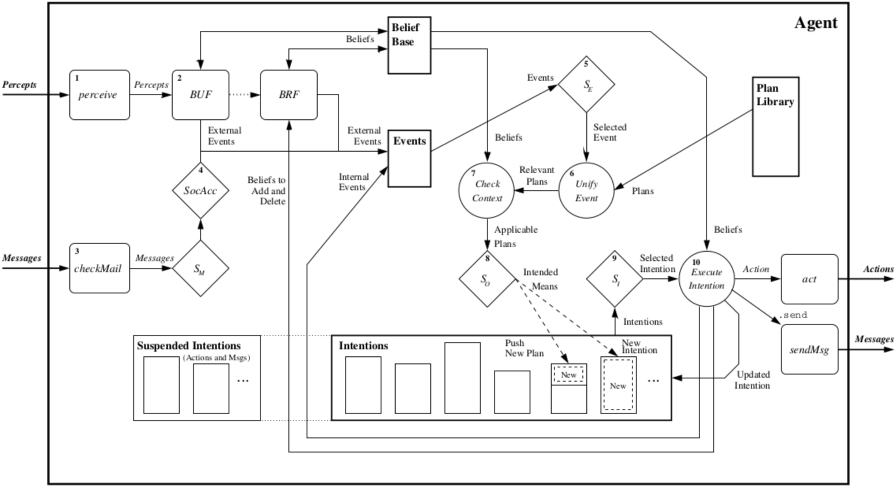
Structure of Plans

triggering event: context← body

Example

grow(tree): plant(seed)← water(soil)

Reasoning Cycle



Reasoning Cycle

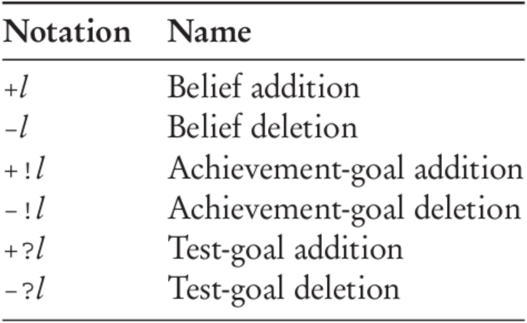
1. Perceive the environment.
2. Update the belief base.
3. Communicate with other agents.
4. Select socially acceptable messages.
5. Choose an event.
6. Retrieve all relevant plans.
7. Determine the applicable plans.
8. Choose one applicable plan - might need hierarchy.
9. Select an intention for further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

Plans

- 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

Plans



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.

Plans

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.

Messages

- 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: the agent sending the message.

performative: what the sender wants to achieve.

content: the actual information the sender is communicating.

Communication Semantics

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.

Operators

- 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 seperated by `;`.
- Conjunction is denoted by `&`.
- Disjunction is denoted by `|`.
- `==` and `\==` indicate equality and inequality, respectively.
- `=` is used for unification of terms.

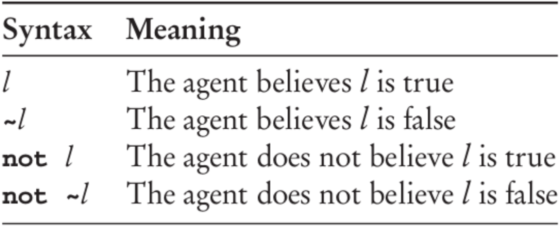
Operators

- `=..` 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.

Negation | Types

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 (∼)**: "I believe it’s false, because I know it’s not true."



Syntax	Meaning
l	The agent believes *l* is true
~*l*	The agent believes *l* is false
not *l*	The agent does not believe *l* is true
not ~*l*	The agent does not believe *l* is false

Negation | Key Principles

Jason agents follow the following key principles:

- **Closed World Assumption:** If a fact cannot be derived from the knowledge base, the agent assumes it is false.
- **Negation as Failure:** If the agent attempts to prove a fact and the proof fails, it infers its negation.

Example

cwa: KB = {`bird(tweety)`}. Since `bird(penguin)` $\notin$ KB and is not derivable, infer $\neg$`bird(penguin)`.

naf: To evaluate `not bird(penguin)`, attempt to prove `bird(penguin)`. Proof fails $\Rightarrow$ `not bird(penguin)` succeeds.

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
- Fibonacci

Body

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

Body

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

   Answer: "I am attending your lab class."

Body

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

   Answer: "I am attending your lab class."

   Goal: !attend(lab).

```
______ : ______ ← !attend(lab).
```

Body

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

Body

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

   Answer: "I am going to study."

Body

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.

Context

3. **Under which conditions wouldn't you do that?**

Context

3. **Under which conditions wouldn’t you do that?**

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

Context

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 ← !attend(lab); !study.

Triggering Event

4. **What makes you want to do that?**

Triggering Event

4. **What makes you want to do that?**

   Answer: "I want to pass the class."

Triggering Event

4. **What makes you want to do that?**

   Answer: "I want to pass the class."

   Goal Addition: +!pass

+!pass : not sick & not bored ← !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
- Fibonacci

File Structure

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.

Code Examples | System

```
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";
}
```

Code Examples | Agents

```
!print_fib(10).

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

+!fibo_num(N, 0) : N == 0.
+!fibo_num(N, 1) : N == 1.

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

Code Examples | System

```
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";
}
```

Code Examples | Agents

```
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)).
```

Code Examples | Environment

```
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) {
    ...
```

Tips and Resources

- 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.

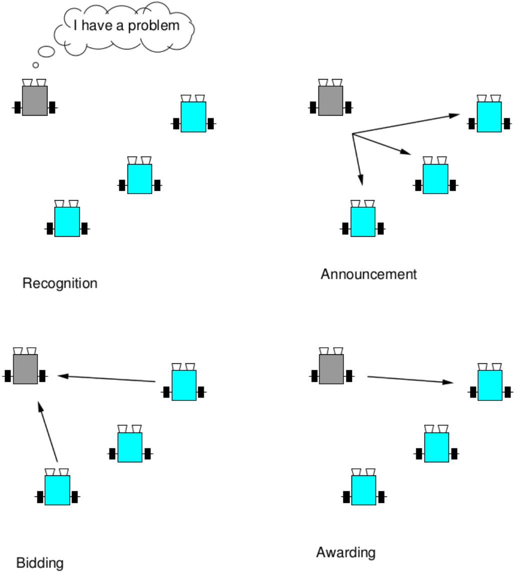
This example is a great starting point for your assignment!

AirNet

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.

AirNet


I have a problem
Recognition
Announcement
Bidding
Awarding

AirNet

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.

AirNet

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.

Fibonacci

The Fibonacci example demonstrates the recursive calculation of Fibonacci numbers.

- The goal is to compute the Fibonacci number at a specified position.
- A fibo agent performs the recursive calculations to determine the number at that position.
- The recursive logic is based on the following formula:

$$F(n) = F(n-1) + F(n-2)$$

Fibonacci

Walking through the flow of interactions:

- The agent is called with a specified position N to compute the Fibonacci number at that position.
- If $N = 0$, the Fibonacci number is 0.
- If $N = 1$, the Fibonacci number is 1.
- For $N > 1$, the agent recursively computes the Fibonacci numbers for $N - 1$ and $N - 2$, then returns their sum.

References

[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:

nathaliekoliou@gmail.com

Thank you!