Corrado Santoro

http://www.dmi.unict.it/~santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica
Università di Catania, Italy
santoro@dmi.unict.it
PROFETA Basics
PROFETA (Python RObotic Framework for dEsining sTrAtegies) is a Python tool for programming autonomous systems (agents or robots) using a declarative approach.

The aim is to have a single environment to implement the software of an autonomous robot.

It provides an “all-in-one” environment supporting both imperative (algorithmical part) and declarative (behavioural part) programming models.

It can be downloaded from: http://github.com/corradosantoro/profeta
PROFETA Basics

- PROFETA is inspired by AgentSpeak(L), a kernel agent language based on the **belief-desire-intention (BDI)** paradigm.
- Its syntax is logic/declarative with some similarities with Prolog.
- Concepts:
  - **Beliefs**
  - **Goals**
  - **Actions**
  - **Plans**
**Beliefs** are used to represent *agent’s knowledge* (agent status, data, environment status, etc.)

Beliefs can be **added** to or **removed** from a **Knowledge Base (KB)** (provided by PROFETA)

The KB can be **queried** to check the presence of certain beliefs and behave accordingly

Beliefs are syntactically expressed by **atomic formulae with ground terms**:

- `position(150,60)`
- `block("red")`
- `stack("cube","pyramid")`
**PROFETA Goals**

- **Goals** are used to represent the *motivational state* of the agent, i.e. what the agent wants to do.
- Actions needed to achieve a certain goal are specified in a *goal plan* of the behaviour (see below).
- A goal can be achieved by “calling it”, in a way similar to a procedure call.
- Goals are syntactically expressed by *atomic forumlae with ground terms* or free variables:
  - `pick_block("red")`
  - `pick_block("X")`
**Actions** are used to represent the **execution units**, driving an agent to achieve their goals.

- Actions contain the specific code (written in Python) to let the robot concretely perform “that thing” on the environment.
- Actions may **fail** and may be **asynchronous**.
- Actions to execute are specified in the behaviour **plans** (see below).
- Actions are syntactically expressed by **atomic formulae with ground terms** or **bound variables**:
  - `go_to(1500, 120)`
  - `grip_open()`
A PROFETA behaviour is a set of **reactive plans** in the form **Event-Condition-Actions**:

- **The Event** is a predicate that triggers the plan:
  - belief assert: $+\text{bel}(\ldots)$
  - belief retract: $-\text{bel}(\ldots)$
  - goal achievement: $\text{goal}(\ldots)$
  - goal failure: $-\text{goal}(\ldots)$

- **Condition** is a *guard* which needs to be true in order to trigger the plan.
- It is a predicate on one or more beliefs present in the knowledge base.

- **Actions** is a list of “things to do” when the plan is triggered:
  - belief assert: $+\text{bel}(\ldots)$
  - belief retract: $-\text{bel}(\ldots)$
  - goal achievement: $\text{goal}(\ldots)$
  - goal failure: $-\text{goal}(\ldots)$
  - user-defined action: `go_to("X", "Y")`
  - python expression: `"X = X + 1"`
```
# import libraries
from profeta.lib import *
from profeta.main import *

class say_hello(Goal):
    pass

# instantiate the engine
PROFETA.start()

# define a goal plan
say_hello() >> [ show_line("hello world from PROFETA") ]

# achieve the "say_hello()" goal
PROFETA.achieve(say_hello())

# run the engine
PROFETA.run()
```
Introducing PROFETA

PROFETA Basics

Belief Plans in PROFETA

```python
# import libraries
from profeta.lib import *
from profeta.main import *

class number(Belief):
    pass

# instantiate the engine
PROFETA.start()

# define an event plan
+number("X") >> [ show_line("yeah! Now I know the number", "X") ]

# add the following beliefs in the knowledge base
PROFETA.assert_belief(number("1"))
PROFETA.assert_belief(number("2"))
PROFETA.assert_belief(number("5"))

# run the engine
PROFETA.run()
```
PROFETA Syntax
Basic Parts of a PROFETA Program

1. Python imports, PROFETA lib imports
2. **Declaration** of application-dependent PROFETA entities:
   - Our beliefs
   - Our goals
   - Our actions
3. PROFETA engine **instantiation**: “PROFETA.start()”
4. Specification of **robot behaviour** by means of the PROFETA declarative syntax
5. Assertion of **initial beliefs** (if needed): “PROFETA.assert_belief()”
6. Start everything with or without a shell:
   - PROFETA.run()
   - PROFETA.run_shell(globals())
The syntax of a profeta plan is based on the following expressions:

\[ \text{EVENT} \gg [ \text{ACT1}, \text{ACT2}, \ldots, \text{ACTn} ] \]
\[ \text{EVENT} / \text{COND} \gg [ \text{ACT1}, \text{ACT2}, \ldots, \text{ACTn} ] \]

- Goals, Beliefs and Actions are expressed as atomic formulae with no parameters or Python strings as parameters.
- Parameters/strings starting with a lowercase letter/number/symbol are treated as constants (atoms).
- Parameters/strings starting with a uppercase letter are treated as variables (Prolog-style).
- Underscore "_" is the special “dont-care” variable (Prolog-style).

```plaintext
pick("redbox") # redbox is a constant (atom)
before("redbox","X") # redbox is a constant (atom)
    # X is a variable
```
An **event** is able to trigger a plan. It can be:

- **Belief Assert**: `+bel(...)`

The plan is triggered when a belief with the given pattern is asserted:

```
+block("cube") >> [ ... ]
# triggered when block("cube") is asserted
```

```
+block("X") >> [ ... ]
# triggered when a block(...) belief is asserted
# variable X is bound to the parameter
```

- **Bound variables** hold their values in the context of the **same plan**
An **event** is able to trigger a plan. It can be:

- **Belief Retract**: `\texttt{-bel(...)}`

  The plan is triggered when a belief with the given pattern is retracted:

  ```
  \texttt{-block("cube") >> [ ... ]}
  \texttt{# triggered when block("cube") is retracted}
  ```

  ```
  \texttt{-block("X") >> [ ... ]}
  \texttt{# triggered when a block(...) belief is retracted}
  \texttt{# variable X is bound to the parameter}
  ```
An event is able to trigger a plan. It can be:

- **Goal Achievement**: `goal(...)`
- The plan is triggered when the achievement of a goal with the given pattern is requested:

  ```
  go(1500,1000) >> [ ... ]
  # triggered when go(1500,1000) is 'called'
  
  go("X", 1000) >> [ ... ]
  # triggered when go(.., 1000) is called
  # variable X is bound to the first parameter of the call
  ```
An **event** is able to trigger a plan. It can be:

- **Goal Failure**: \(-\text{goal}(...)\)

- The plan is triggered when that goal failed:
  
  \[-\text{go}(1500,1000) >> [\ldots]\]
  
  # triggered when the goal \(\text{go}(1500,1000)\) failed

\[-\text{go}("X",1000) >> [\ldots]\]

# triggered when the goal \(\text{go}(\ldots,1000)\) failed

# variable \(X\) is bound to the first parameter of the call
A **condition** is a guard that expresses a predicate on:

- The presence, in the KB, of certain beliefs (with given patterns)
- The values of the variables bound in the plan

```prolog
go("X","Y") / position("X", "Y") >> [ ... ]
```

```
# Triggered when go(.,.,.) goal is ‘‘called’’
#
# Variables X and Y are bound to the goal parameters
#
# The belief position(.,.,.) must be asserted, and its
# parameters must be equal to variables X and Y
```
Several belief patterns can be specified using the "&" operator

Variables, when present, are unified

```plaintext
go("X","Y") / (position("X", "Y") & cube("X", "Y1")) >> [ ... ]
```

# Triggered when go(.,...) goal is ‘‘called’’
# Variables X and Y are bound to the goal parameters
# The beliefs position(.,...) and cube(.,...) must be
# asserted with the given variable patterns
#
Boolean expressions on variables are possible using “lambdas”

```plaintext
go("X","Y") / (position("X", "Y") & (lambda : int(X) > 30) ) >>
[ ... ]
```

`# Triggered when go(.,.,.) goal is ‘called’`
`# Variables X and Y are bound to the goal parameters`
`# The belief position(.,.,.) must be asserted, and its parameters must be equal to variables X and Y;`
`# variable X must also be greater than 30`

(Since variables are not typed, it’s better to use typecasting in order to ensure correct evaluation of data)
Actions are the computations to be executed when a plan is triggered.

Actions are syntactically expressed, in a plan, as a Python list.

Elements of the list can be:

<table>
<thead>
<tr>
<th>Action Type</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>belief assert</td>
<td>+bel(...)</td>
</tr>
<tr>
<td>belief retract</td>
<td>-bel(...)</td>
</tr>
<tr>
<td>goal achievement</td>
<td>goal(...)</td>
</tr>
<tr>
<td>goal failure</td>
<td>-goal(...)</td>
</tr>
<tr>
<td>library action</td>
<td>show_line(&quot;variables are&quot;, &quot;X&quot;, &quot;Y&quot;)</td>
</tr>
<tr>
<td>user-defined action</td>
<td>go_to(&quot;X&quot;, &quot;Y&quot;)</td>
</tr>
<tr>
<td>python expression</td>
<td>&quot;X = X + 1&quot;</td>
</tr>
</tbody>
</table>

Certain actions (KB manipulation, goal achievement/failure) are able to trigger further plans (if the proper event is present).
Example: Summing “n” numbers

- Python imports, PROFETA lib imports

```python
# import libraries
from profeta.lib import *
from profeta.main import *
...
```
Example: Summing “n” numbers

- **Declaration** of application-dependent PROFETA entities:
  - A belief `number(x)` which is used to represent a number to sum
  - A goal `sum_all()` which contains the actions to perform our sum

- Beliefs and goals are defined by simply subclassing Belief and Goal

- Then the **PROFETA** engine can be instantiated

```python
... class number(Belief):
    pass

class sum_all(Goal):
    pass

PROFETA.start()
...```
Example: Summing “n” numbers

- Specification of the **behaviour** by means of a set of **plans**

```plaintext
sum_all() / number("X") >> [ -number("X"), sum_all("X") ]
sum_all("S") / number("X") >> [ -number("X"),
                      "S = S + X",
                      sum_all("S") ]
sum_all("S") >> [ show_line("the sum is", "S") ]
```
Example: Summing “n” numbers

The goal “sum_all()” is executed given that at least one belief “number(X)” is in the knowledge base; if this is the case, variable “X” is bound to the parameter; as a consequence, the belief is removed from the knowledge base, then the goal “sum_all()” is invoked with the number value as argument:

```
sum_all() / number("X") >> [ -number("X"), sum_all("X") ]
```
The goal \texttt{sum\_all(S)} is executed with the partial sum $S$ as parameter and given that at least one belief \texttt{number(Y)} is in the knowledge base; the belief is removed from the knowledge base, the partial sum is updated and the goal \texttt{sum\_all(S)} is recursively invoked:

\begin{verbatim}
sum\_all("S") / number("Y") >> [ -number("Y"),
   "S = S + Y",
   sum\_all("S") ]
\end{verbatim}
Example: Summing “n” numbers

The last plan does not require the presence of the belief; it is executed given that the previous plan is not triggered; it means that no more numbers need to be processed, so the final value can be printed on the screen:

```
sum_all("S") >> [ show_line("the sum is", "S") ]
```
After the plans, we assert our initial beliefs:

```python
PROFETA.assert_belief(number("4"))
PROFETA.assert_belief(number("8"))
PROFETA.assert_belief(number("10"))
PROFETA.assert_belief(number("20"))
PROFETA.assert_belief(number("25"))
PROFETA.assert_belief(number("1"))
PROFETA.assert_belief(number("21"))
PROFETA.assert_belief(number("12"))
```
Example: Summing “n” numbers

Finally, we specify the goal to be executed and we run the PROFETA engine:

```python
PROFETA.achieve(sum_all())
PROFETA.run()
```
Example: Summing “n” numbers

Here is the complete listing:

```python
from profeta.lib import *
from profeta.main import *

class number(Belief):
    pass

class sum_all(Goal):
    pass

PROFETA.start()

sum_all() / number("X") >> [ -number("X"); sum_all("X") ]
sum_all("S") / number("X") >> [ -number("X"); "S = int(S) + int(X)"; sum_all("S") ]
sum_all("S") >> [ show_line("the sum is"); "S" ]

PROFETA.assert_belief(number("4"));
PROFETA.assert_belief(number("8"));
...

PROFETA.achieve(sum_all());
PROFETA.run()
```
Example: the Sieve of Eratosthenes

1. Let us fill our KB with all numbers from 1 to 100, using a belief `number(X)`
2. Let us write a plan that searches for two beliefs representing numbers e.g. X and Y
3. If X can be divided by Y, let us remove the belief `number(X)`
4. Let us re-trigger the plan at step 2

```
-number("_") / (number("X") & number("Y") & (lambda : X != Y) & (lambda : (X % Y) == 0) ) >> [ -number("X") ]
```
from profeta.lib import *
from profeta.main import *

class number(Belief):
    pass

PROFETA.start()

-number("_") / (number("X") & \
   number("Y") & \
   (lambda : X != Y) & \
   (lambda : (X % Y) == 0) ) >> [ -number("X") ]

for i in range(1,100):
    PROFETA.assert_belief(number(i))

PROFETA.run_shell(globals())
PROFETA Semantics
The **same event** can be used in **different plans**, given that the **condition part is different** as well.

1. `EVENT / COND1 >> [ ... ]`
2. `EVENT / COND2 >> [ ... ]`
3. `EVENT / COND3 >> [ ... ]`

A set of plans with the same event is denoted as **plan group**.

**Only one plan** in a group can be selected for execution.
Execution in Plan Groups

When EVENT occurs, the PROFETA engine analyses the relevant plan group, **one plan at time, in strict declaration sequence**

- For each plan, the condition is evaluated, if it is **false**, the next plan is analysed
- If the condition is **true**, the plan is **executed**, all the other plans of the group are skipped, for this stage
Let's use plan groups and let's generate all numbers using plans

```python
class number(Belief):
    pass

+number("N") / (lambda : N < 100) >> [ "N = N + 1",
    +number("N") ]
+number("N") >> [ show_line("end of sequence generation"),
    -number(1) ]

... 
```

- Generation is triggered by asserting “number(1)”
- Plans in lines 4 and 6 could be triggered, but the condition in line 4 is **true** so that plan is executed
Let's use plan groups and let's generate all numbers using plans

```python
class number(Belief):
    pass

+number("N") / (lambda : N < 100) >> [ "N = N + 1",
    +number("N") ]
+number("N") >> [ show_line("end of sequence generation"),
    -number(1) ]
...
```

- In line 4, N is incremented and a new "number()" belief is asserted
- This cause plans in the group to be considered again
Let's use plan groups and let's generate all numbers using plans

```python
class number(Belief):
    pass

+number("N") / (lambda : N < 100) >> [ "N = N + 1",
    +number("N") ]
+number("N") >> [ show_line("end of sequence generation"),
    -number(1) ]
...
```

- When N reaches 100, condition in line 4 is false
- This cause execution of plan in line 6 which includes the removal of belief “number(1)” in order to start prime numbers detection
Let’s detect prime numbers

```
1   ...
2
3   -number("_") / (number("X") & \n     number("Y") & \n     (lambda : X != Y) & \n     (lambda : (X % Y) == 0) ) >> [ -number("X") ]
4
5   -number("_") >> [ show_line("prime numbers detected") ]
```

Here plan in line 3 is used to remove non-prime numbers

And plan in line 7 is executed when there are no more non-prime numbers to be removed
A plan is executed “atomically” until its completion (unless a failure is detected)

Therefore, the events generated from actions modifying the KB, i.e. +bel() or -bel(), are not immediately handled but only after plan execution completion

```
... >> [ +number(1), show_line("one"),
        -number(2), show_line("two") ]
+number("1") >> [ show_line("adding 1") ]
-number("2") >> [ show_line("removing 2") ]
```

The output will be:

```
one
two
adding 1
removing 2
```
Semantics of Goal Execution

- Goal achievement is interpreted as a classical “procedure call”
- The current plan is suspended and the goal plan is called

```plaintext
... >> [ show_line("one"), g1(), show_line("two") ]
g1() >> [ show_line("this is goal 1"), g2() ]
g2() >> [ show_line("this is goal 2") ]
```

- The output will be:

```
one
this is goal 1
this is goal 2
two
```
Goal Execution and KB modifications

- When goal achievement and KB modifications are mixed, KB events are processed when the **complete goal calling line is over**

```
... >> [ show_line("one"), +number(1),
    g1(), show_line("two") ]
g1() >> [ show_line("this is goal 1"), g2() ]
g2() >> [ show_line("this is goal 2") ]
+number(1) >> [ show_line("number 1") ]
```

- The output will be:

```
one
this is goal 1
this is goal 2
two
two
number 1
```
Semantics of Special Beliefs

Together with “classical” beliefs, PROFETA provides two kind of special beliefs:

- **Singleton Beliefs**: these are beliefs that can exist in single instance. Executing `+singleton_bel(...)` on an existing belief cause that belief to be modified.
- A Singleton Belief is declared by subclassing `SingletonBelief`.

- **Reactors**: these beliefs do not affect the KB but are only used to generate events.
- They are mainly used in “sensors” (see below).
- A Reactor is declared by subclassing `Reactor`.
Case Study: SHRDLU
The world of SHRDLU

- **SHRDLU** is one of first artificial intelligence programs (1968-1970) supporting **reasoning/planning** and natural **language processing**.

- It is based on a (virtual) environment composed of some “blocks” with different shapes and colors (cubes, pyramids, prisms, cylinders, etc.).

- The program can understand commands like “Pick up a big red block” or “Grasp the pyramid” and behave accordingly.

- The name **SHRDLU** comes from the sequence **ETAOIN SHRDLU** (why?) which is the same sequence of the keys of the Linotype (based on the frequency of letters in the English language).
The Linotype and the SHRDLU sequence
Let’s implement a small version of SHRDLU with PROFETA (see the code in file “SHRDLU.py”)

Our world is based on 3D objects: cube, cylinder, prism

These objects are placed on a table and they can be picked up by a robot

On the table, objects may be stacked, one on the top of another, in order to form a tower

An object can be picked only if it is free (it has no other objects on the top)
BELIEFS:

- \text{obj}(X)$, represents the presence of block \text{X} on the table
- \text{owned}(X)$, represents the fact that the block \text{X} is \text{owned} (picked) by the robot
- \text{upon}(X,Y)$, block \text{X} is placed upon block \text{Y}

GOALS:

- \text{pick}(X)$, request to pick block \text{X}, if possible
- \text{put}(X)$, request to put block \text{X} on the table
- \text{put}(X,Y)$, request to put block \text{X} upon block \text{Y}
SHRDLU plans: object picking

- Object X is on the table, but another object Y is upon it $\Rightarrow$ X cannot be picked

```
pick("X") / (obj("X") & upon("Y", "X")) >> \\n[ show_line("cannot pick", "X", "since it is under the","Y") ]
```

- If previous condition is false and object X is on the table and it is on the top of object Y $\Rightarrow$ X can be picked (update beliefs accordingly)

```
pick("X") / (obj("X") & upon("X", "Y")) >> \\n[ show_line("X", "picked"), \\
-obj("X"), -upon("X", "Y"), +owned("X") ]
```
SHRDLU plans: object picking

- If previous condition is **false** and object X is on the table (no other objects on the top of it) ⇒ **X can be picked** (update beliefs)

  ```
  pick("X") / obj("X") >> [ show_line("X", "picked"), -obj("X"), +owned("X") ]
  ```

- If previous condition is **false** and object X is owned by the robot ⇒ **X cannot be picked**

  ```
  pick("X") / owned("X") >> [ show("you’ve still got", "X") ]
  ```

- If all previous conditions are **false** the last plan is triggered ⇒ **it means that object X does not exist**

  ```
  pick("X") >> [ show_line("cannot pick", "X", "since it is not present") ]
  ```
SHRDLU plans: object picking

Summary of the plans for object picking:

- pick("X") / (obj("X") & upon("Y", "X")) >> \
  [ show_line("cannot pick", "X", "since it is under the","Y") ]

- pick("X") / (obj("X") & upon("X", "Y")) >> \
  [ show_line("X", "picked"),
    -obj("X"), -upon("X", "Y"), +owned("X") ]

- pick("X") / obj("X") >> [ show_line("X", "picked"),
    -obj("X"), +owned("X") ]

- pick("X") / owned("X") >> [ show("you’ve still got", "X") ]

- pick("X") >> [ show_line("cannot pick", "X",
    "since it is not present") ]
SHRDLU plans: object release

- Object X is owned $\Rightarrow$ put X on the table

  \[
  \text{put("X") / owned("X") >> } \\[
  \text{[ show_line("X", "is now on the table"),}
  \text{ -owned("X"), +obj("X") ]}
  \]

- Object X is already on the table $\Rightarrow$ nothing to do

  \[
  \text{put("X") / obj("X") >> } \\[
  \text{[ show_line("X", "is already on the table") ]}
  \]

- Object X is neither held nor on the table $\Rightarrow$ the object does not exist, nothing to do

  \[
  \text{put("X") >> [ show_line("X", "does not exist") ]}
  \]
SHRDLU plans: object release

We want to put object X on the top of object Y:

- Object X is owned, object Y is on the table but Y has another object Z on the top of it ⇒ we cannot do the job

\[
\text{put("X", "Y") / (owned("X") & obj("Y") & upon("Z", "Y") ) \ \\
>> [ show_line("Y", "has", "Z", "on its top") ]}
\]

- Object X is owned, object Y is on the table but, since here the previous condition is false, Y is free ⇒ we can put X upon Y

\[
\text{put("X", "Y") / (owned("X") & obj("Y") ) >> \ \\
[ -owned("X"), +obj("X"), +upon("X", "Y"), \\
  show_line("done") ]}
\]
Summary of the plans for object release:

- `pick("X") / (obj("X") & upon("Y", "X")) >> \
  [ show_line("cannot pick", "X", "since it is under the","Y") ]`

- `pick("X") / (obj("X") & upon("X", "Y")) >> \
  [ show_line("X", "picked"),
    -obj("X"), -upon("X", "Y"), +owned("X") ]`

- `pick("X") / obj("X") >> [ show_line("X", "picked"),
    -obj("X"), +owned("X") ]`

- `pick("X") / owned("X") >> [ show("you’ve still got", "X") ]`

- `pick("X") >> [ show_line("cannot pick", "X",
    "since it is not present") ]`
Let’s see a run of SHRDLU
Interaction with the Environment
We used the SHRDLU program from the PROFETA shell, but this is not an usual way to run a program in a production system.

We need a **standalone execution** with proper abstractions to perform interaction with the environment.

In order to **provide data** to a PROFETA program from the environment, PROFETA offers the **Sensor** class.

A **Sensor** includes the code to **get data from the environment** and convert it into a **belief** for a PROFETA program.
A sensor is defined by subclassing Sensor and overriding the method sense.

Method sense must include the code to retrieve data and return:

- None: no interesting data has been sampled
- +bel(...): a belief is asserted and the relevant “add” event is generated
- bel(...): a belief is asserted but no event is generated (only KB is updated)

Then an instance of that sensor must be added to the PROFETA engine through the PROFETA.add_sensor() API function.

```python
class my_s(Sensor):
    def sense(self):
        # do data poll
        return ...

... 
PROFETA.add_sensor(my_s())
```
Sensor Semantics

- All sensors are managed in a list, which is scanned when there are no pending events.
- Scanning the list means to call method `sense` of all defined sensors.
- This implies that the **dynamics** (i.e. time period) of environment polling cannot be determined.
- If a specific dynamics is needed, class `AsyncSensorProxy` provides a useful way to perform asynchronous handling.
Managing Sensors

- Sensors can be turned **on** and **off** according to system requirements.
- When a sensor is **off**, its method `sense` is not called at all.
- By default, a sensor is **on**.
- To manage sensors, the library actions `sensor_on(...)` and `sensor_off(...)` are provided, which can be used in plans.

```plaintext
>> [ ..., sensor_on(my_s), ... ]
>> [ ..., sensor_off(my_s), ... ]
```
User-defined Actions

- Acting on the environment implies to write proper code that drives actuators.
- To this aim, the class `Action` is provided to write a user-defined action (UDA).
User-Defined Action

- A UDA is implemented by subclassing Action and overriding the method execute.
- A UDA is called inside a plan by referring it as an atomic formula with one or more parameters.
- Parameters are referred, in method execute, as self[0], self[1], ...

```python
class move_to(Action):
    def execute(self):
        motion_system.perform_move_to(self[0], self[1])
```

```
pick_object_at("X", "Y") >> [ move_to("X", "Y") ]
```
A real implementation of SHRDLU should consider a “user-friendly” user-interface (not the PROFETA shell)

- A microphone and a speech-to-text API would be ideal, but (in the absence of it) ...
- ... let us implement a prompt in which a user can type a command in natural language
- And let us also to write a (very simple!) natural language parser in PROFETA
A “Prompt” Sensors

```python
class prompt(Sensor):
    def sense(self):
        e = raw_input("COMMAND: ")
        tokens = e.lower().split()
        # use '*' to expand tokens into a sequence of arguments
        return +say(*tokens)
```

Here the `prompt` sensor:

- Waits for a phrase typed by the user
- Splits the phrase into words
- Asserts a “`say(...)`” reactor using the list of words as arguments
- e.g., if the user types “`pick the cube`”, the generated reactor is “`say("pick","the","cube")`”
The **prompt** sensor generates the “*say(*...*)*” reactor.

The **NLP** interprets the “*say(*...*)*” reactor and calls one of the goals to pick and put objects.

The NLP is implemented by means a set of PROFETA plans.

The called goals are executed by the **SHRDLU** plans.
NLP in SHRDLU

+say("V", "X") / (lambda : V in ['pick', 'get', 'catch']) >> [ pick("X") ]
+say("V", "the", "X") / (lambda : V in ['pick', 'get', 'catch']) >> [ pick("X") ]
+say("V", "a", "X") / (lambda : V in ['pick', 'get', 'catch']) >> [ pick("X") ]

+say("V", "X") / (lambda : V in ['put', 'release']) >> [ put("X") ]
+say("V", "the", "X") / (lambda : V in ['put', 'release']) >> [ put("X") ]
+say("V", "the", "X", "upon", "the", "Y") / (lambda : V in ['put', 'release']) >> [ put("X", "Y") ]
+say("V", "X", "upon", "the", "Y") / (lambda : V in ['put', 'release']) >> [ put("X", "Y") ]
+say("V", "the", "X", "upon", "Y") / (lambda : V in ['put', 'release']) >> [ put("X", "Y") ]
+say("V", "X", "upon", "Y") / (lambda : V in ['put', 'release']) >> [ put("X", "Y") ]

# additional plans for other useful commands
+say("show", "me", "the", "table") >> [ show_table() ]
+say("show", "the", "table") >> [ show_table() ]
+say("show", "table") >> [ show_table() ]

+say("show", "me", "the", "tower", "of", "X") >> [ tower("X") ]
+say("show", "me", "tower", "of", "X") >> [ tower("X") ]
+say("show", "the", "tower", "of", "X") >> [ tower("X") ]
+say("show", "tower", "of", "X") >> [ tower("X") ]

+say("show", "my", "objects") >> [ show_my() ]

# last plan triggered when any set of parameters is given
+say("*") >> [ show_line("what?") ]
http://github.com/corradosantoro/profeta


References

References

Programming Autonomous Robots using PROFETA and the BDI model

Corrado Santoro
http://www.dmi.unict.it/~santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica
Università di Catania, Italy
santoro@dmi.unict.it